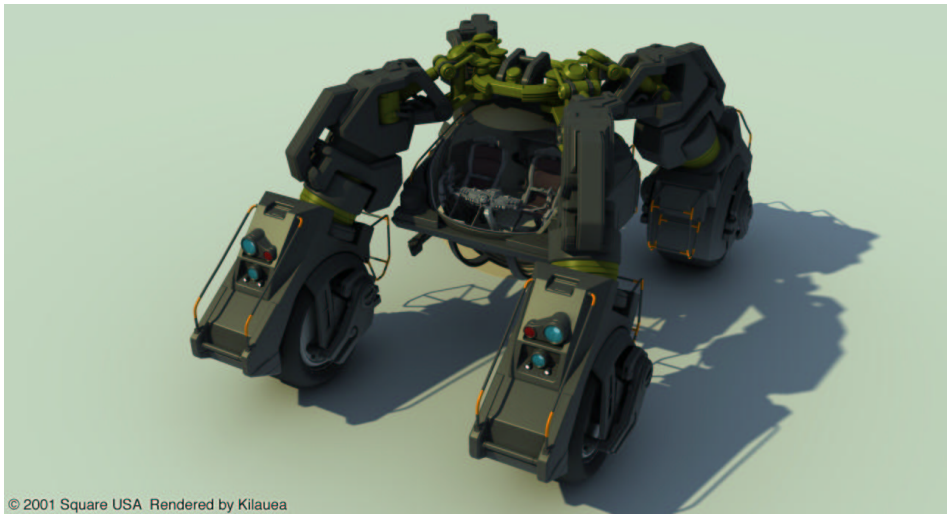
Practical Parallel Rendering

Alan Chalmers Timothy A. Davis Toshi Kato Erik Reinhard



Course 22 **SIGGRAPH 2002**

San Antonio
July 21-26, 2002

SAN ANTONIO SIGGRAPH 2002

About the authors

Alan Chalmers is a senior lecturer in the Department of Computer Science at the University of Bristol, UK. He has published over 70 papers in journals and international conferences on parallel photo-realistic graphics. He was the co-chairman of the recent IEEE Parallel Visualization and Graphics Symposium and is chairman of the Eurographics workshop series on Parallel Graphics and Visualisation.Recently he and Professor F. W. Jansen were the guest editors of the Journal of Parallel Computing for its special edition on Parallel Graphics and Visualisation. He is also currently vice-chair of ACM SIGGRAPH. His research interests include the application of parallel photo-realistic graphics to archaeological site visualisation in order to provide a flexible tool for investigating site reconstruction and utilization.

Timothy Davis is an assistant professor in the Computer Science Department at Clemson University where he works within the newly established MFAC (Master of Fine Arts in Computing) group, which trains students to produce special effects for entertainment and commercial projects. Most recently, he has published papers at the 1999 IEEE Parallel Visualization and Graphics Symposium and the 1998 Eurographics Workshop on Parallel Graphics and Visualisation. His research involves exploiting spatio-temporal coherence in a parallel environment to reduce rendering times for ray-traced animations. He received his Ph.D. from North Carolina State University and has worked in technical positions for the Environmental Protection Agency and NASA Goddard Space Flight Center.

For eighteen years Toshi Kato has devoted his time to the development of high-end production-level renderers. He was the lead of the project "Kilauea," a parallel ray tracer developed at Square USA's R&D division. The development of "Kilauea" involved various fields including system architecture design, implementation of a message passing layer, and a parallel shading and ray tracing engine. From 1993 to 1998 he developed an in-house scanline-based renderer at Rhythm & Hues in LA and participated in the production of movies such as Mouse Hunt (1998), Kazaam (1996) and Babe (1995). He also developed a special stereoscopic renderer for IMAX's Omnimax Solido dome.

Erik Reinhard is a researcher at the University of Utah in the fields of parallel ray tracing and visual perception. He received a 'TWAIO' diploma in parallel computer graphics from Delft University of Technology in 1996 and a PhD degree from the University of Bristol in 2000. His current research interests include algorithms and data structures for real-time ray tracing. This includes mechanisms to trade-off visual quality for interactivity and to add animation capabilities to real-time ray tracing. He has published more than 20 papers on parallel rendering.

Structure of the Course

Parallel processing offers the potential of rendering high-quality images and animations in reasonable times. This course begins by reviewing the basic issues involved in rendering within a parallel or distributed computing environment. Specifically, various methods are presented for dividing the original rendering problem into subtasks and distributing them efficiently to independent processors. Careful consideration must be taken to balance the processing load across the processors, as well as reduce communication between these subtasks for faster processing.

The course continues by examining the strengths and weaknesses of multiprocessor machines and networked render farms for graphics rendering. Case studies of working applications including “real time raytracing," and streamlining the creation of full CG movies (“Final Fantasy”) will be presented to demonstrate in detail practical ways of dealing with the issues involved.

Introduction (10 minutes) (Davis)

- The need for speed in satisfying the demand for high-quality graphics
- Rendering and parallel processing: a holy union
- The exciting possibilities of parallel processing in a world of advanced 3D graphics cards

Parallel/Distributed Rendering Issues (45 minutes) (Chalmers)

- Task subdivision
- Load balancing
- Communication
- Task migration
- Data Management

Classification of Parallel Rendering Systems (25 minutes) (Davis)

By rendering technique

- Polygon rendering
  - Sort first, sort middle , sort last
- Photo-realistic methods
  - Image space subdivision, object space subdivision, object subdivision

By hardware:

- Multiprocessor machines
  - Pros and cons
- Distributed Computing
  - Introduction to render farms
  - Pros and cons

Practical Applications (25 minutes) (Davis)

- Distributed computing and spatial/temporal coherence

- Animations

Practical Applications continued

Getting the most from your machine (40 minutes) (Reinhard)

- Real time raytracing
  - basic operations
  - animation and interactivity
  - adding complexity

Parallel rendering and the quest for realism (45 minutes) (Kato)

- The “Kilauea” massively parallel ray tracer
  - System design
  - Memory management for thread environments
  - Parallel shading calculations, space traversal, photon maps
  - Debugging and stability

Summary (10 minutes) (Chalmers)

Discussion and questions (10 minutes) (All)

Structure of the notes

The notes contain important background information as well as detailed descriptions of the Parallel Processing techniques described. The notes are arranged as follows:

Section I introduces the concepts which are necessary to understand the difficulties confronting any parallel implementation. The section goes on in chapter 2 to describe the issues associated with scheduling tasks on a parallel system. A number of ways of decomposing a problem on a parallel machine are considered and the advantages and disadvantages of each approach are highlighted. Chapter 3 of Section I concentrates on managing large data requirements which are distributed across the parallel environment. Issues of consistency and latency are considered.

Section II considers the classification of parallel rendering systems according to the method of task subdivision and/or by the hardware used.

Section III considers Interactive Ray Tracing in depth including hardware considerations, animation and reuse techniques. Appendices are provided on the SGI Origin 2000.

Finally, Section IV provides details on the “Kilauea” massively parallel raytracer.

Section I
Parallel/Distributed Rendering Issues

Alan Chalmers

Contents

Section I: Parallel/Distributed Rendering Issues

Chapter 1

Introduction

Parallel processing is like a dog's walking on its hind legs. It is not done well, but you are surprised to find it done at all.

[Steve Fiddes (University of Bristol) with apologies to Samuel Johnson]

Realistic computer graphics is an area of research which develops algorithms and methods to render images of artificial models or worlds as realistically as possible. Such algorithms are known for their unpredictable data accesses and their high computational complexity. Rendering a single high quality image may take several hours, or even days. Parallel processing offers the potential for solving such complex problems in reasonable times.

However, there are a number of fundamental issues: task scheduling, data management and caching techniques, which must be addressed if parallel processing is to achieve the desired performance when computing realistic images. These are applicable for all three rendering techniques presented in this tutorial: ray tracing, radiosity and particle tracing.

This chapter introduces the concepts of parallel processing, describes its development and considers the difficulties associated with solving problems in parallel.

Parallel processing is an integral part of everyday life. The concept is so ingrained in our existence that we benefit from it without realising. When faced with a taxing problem, we involve others to solve it more easily. This co-operation of more than one worker to facilitate the solution of a particular problem may be termed parallel processing. The goal of parallel processing is thus to solve a given problem more rapidly, or to enable the solution of a problem that would otherwise be impracticable by a single worker.

The principles of parallel processing are, however, not new, as evidence suggests that the computational devices used over 2000 years ago by the Greeks recognised and exploited such concepts. In the Nineteenth Century, Babbage used parallel processing in order to improve the performance of his Analytical Engine [48]. Indeed, the first general purpose electronic digital computer, the ENIAC, was conceived as a highly parallel and decentralised machine with twenty-five independent computing units, co-operating towards the solution of a single problem [27].

However, the early computer developers rapidly identified two obstacles restricting the widespread acceptance of parallel machines: the complexity of construction; and, the seemingly high programming effort required [10]. As a result of these early set-backs, the developmental thrust shifted to computers with a single computing unit, to the detriment of parallel designs. Additionally, the availability of sequential machines resulted in the development of algorithms and techniques optimised for these particular architectures.

The evolution of serial computers may be finally reaching its zenith due to the limitations imposed on the design by its physical implementation and inherent bottlenecks [5]. As users continue to demand improved performance, computer designers have been looking increasingly at parallel approaches to overcome these limitations. All modern computer architectures incorporate a degree of parallelism. Improved hardware design and manufacture coupled with a growing understanding of how to tackle the difficulties of parallel programming has re-established parallel processing at the forefront of computer technology.

1.1 Concepts

Parallel processing is the solution of a single problem by dividing it into a number of sub-problems, each of which may be solved by a separate worker. Co-operation will always be necessary between workers during problem solution, even if this is a simple agreement on the division of labour. These ideas can be illustrated by a simple analogy of tackling the problem of emptying a swimming pool using buckets. This job may be sub-divided into the repeated *task* of removing one bucket of water.

A single person will complete all the tasks, and complete the job, in a certain time. This process may be speeded-up by utilising additional workers. Ideally, two people should be able to empty the pool in half the time. Extending this argument, a large number of workers should be able to complete the job in a small fraction of the original time. However, practically there are physical limitations preventing this hypothetical situation.

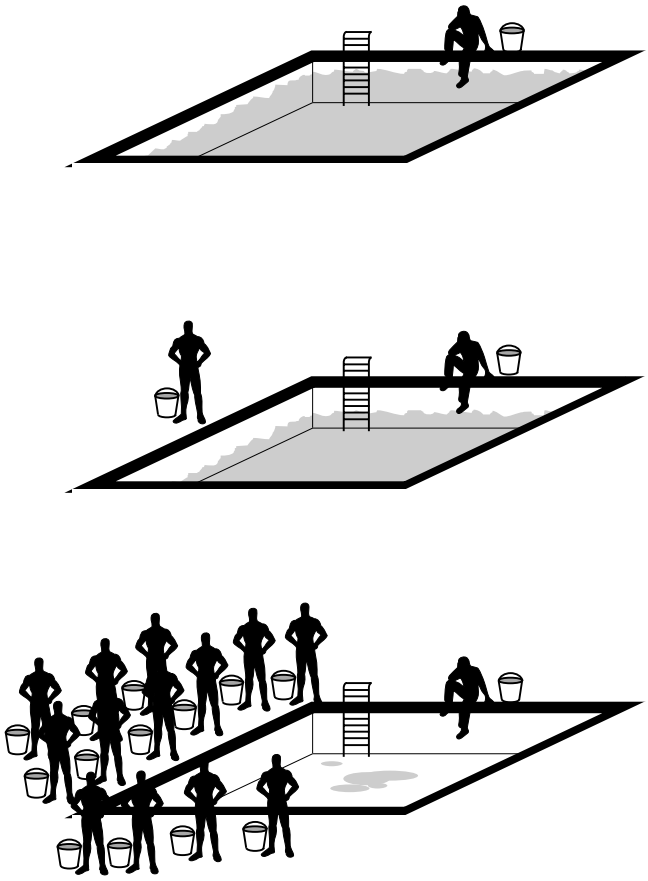


Figure 1.1: Emptying a pool by means of a bucket

The physical realisation of this solution necessitates a basic level of co-operation between workers. This manifests itself due to the contention for access to the pool, and the need to avoid collision. The time required to achieve this co-operation involves inter-worker communication which detracts from the overall solution time, and as such may be termed an *overhead*.

1.1.1 Dependencies

Another factor preventing an ideal parallel solution are termed: dependencies. Consider the problem of constructing a house. In simple terms, building the roof can only commence after the walls have been completed. Similarly, the walls can only be erected once the foundations are laid. The roof is thus dependent upon the walls, which are in turn dependent on the foundations. These dependencies divide the whole

problem into a number of distinct stages. The parallel solution of each stage must be completed before the subsequent stage can start.

The dependencies within a problem may be so severe that it is not amenable to parallel processing. A strictly sequential problem consists of a number of stages, each comprising a single task, and each dependent upon the previous stage. For example, in figure 1.2, building a tower of toy blocks requires a strictly sequential order of task completion. The situation is the antithesis of dependency-free problems, such as placing blocks in a row on the floor. In this case, the order of task completion is unimportant, but the need for co-operation will still exist.

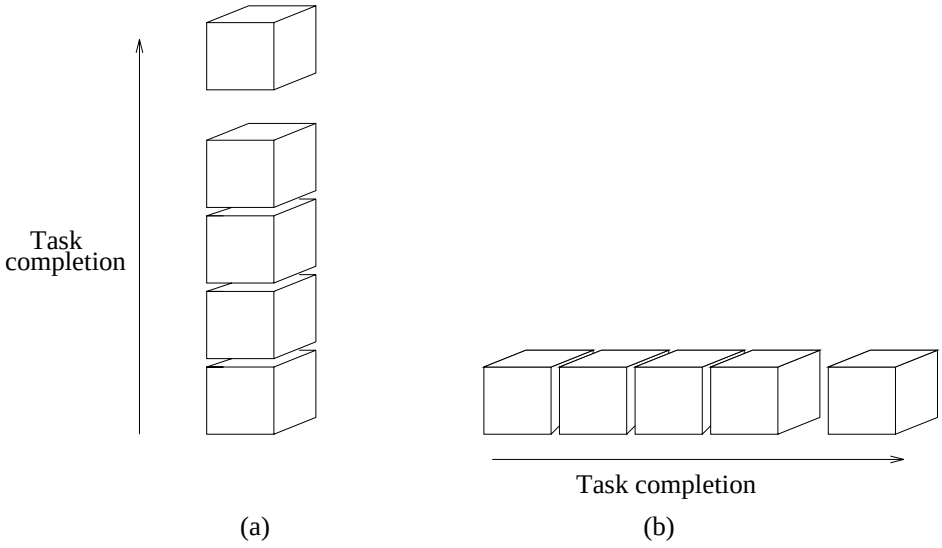


Figure 1.2: Building with blocks: (a) Strictly sequential (b) dependency-free

Pipelining is the classic methodology for minimising the effects of dependencies. This technique can only be exploited when a process, consisting of a number of distinct stages, needs to be repeated several times. An automotive assembly line is an example of an efficient pipeline. In a simplistic form, the construction of a car may consist of four linearly dependent stages as shown in figure 1.3: chassis fabrication; body assembly; wheel fitting; and, windscreen installation. An initial lump of metal is introduced into the pipeline then, as the partially completed car passes each stage, a new section is added until finally the finished car is available outside the factory.

Consider an implementation of this process consisting of four workers, each performing their task in one time unit. Having completed the task, the worker passes the partially completed car on to the next stage. This worker is now free to repeat its task on a new component fed from the previous stage. The completion of the first car occurs after four time units, but each subsequent car is completed every time unit.

The completion of a car is, of course, sensitive to the time taken by each worker. If one worker were to take longer than one time unit to complete its task then the worker after this difficult task would stand idle awaiting the next component, whilst those before the worker with the difficult task would be unable to move their component on to the next stage of the pipeline. The other workers would thus also be unable to do any further work until the difficult task was completed. Should there be any interruption in the input to the pipeline then the pipeline would once more have to be "refilled" before it could operate at maximum efficiency.

1.1.2 Scalability

Every problem contains a upper bound on the number of workers which can be meaningfully employed in its solution. Additional workers beyond this number will not improve solution time, and can indeed be detrimental. This upper bound provides an idea as to how suitable a problem is to parallel implementation: a measure of its *scalability*.

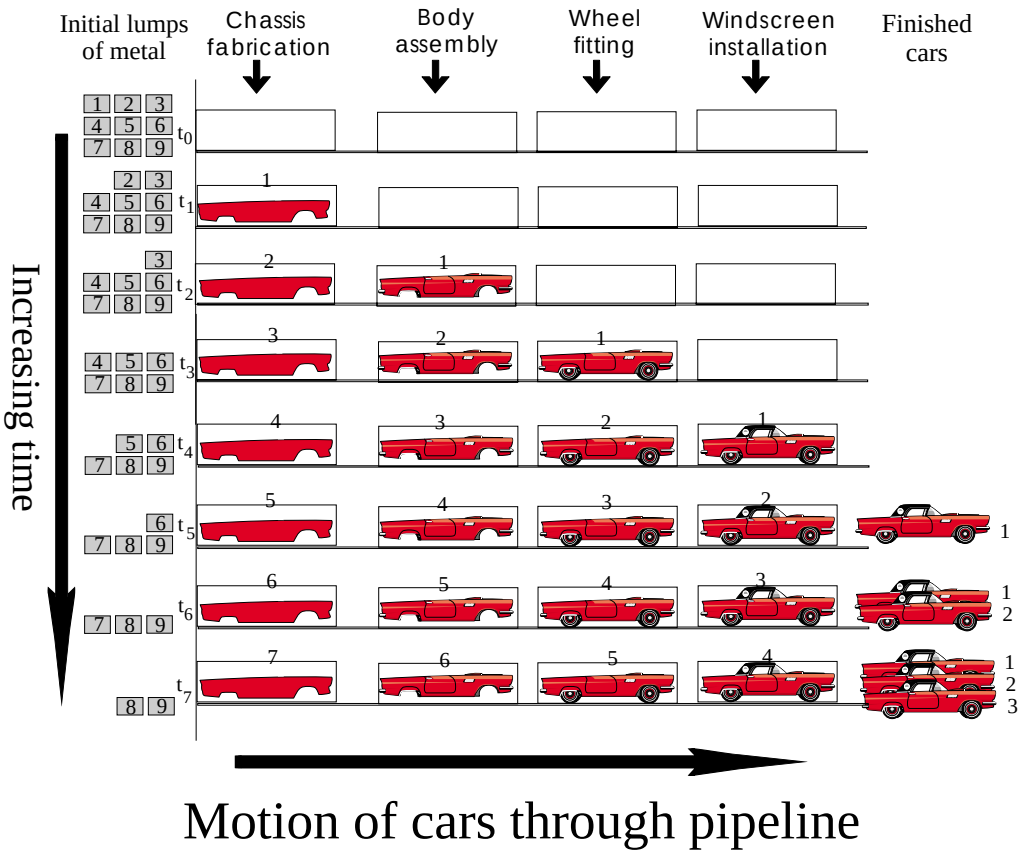


Figure 1.3: Pipeline assembly of a car

A given problem may only be divided into a finite number of sub-problems, corresponding to the smallest tasks. The availability of more workers than there are tasks, will not improve solution time. The problem of clearing a room of 100 chairs may be divided into 100 tasks consisting of removing a single chair. A maximum of 100 workers can be allocated one of these tasks and hence perform useful work.

The optimum solution time for clearing the room may not in fact occur when employing 100 workers due to certain aspects of the problem limiting effective worker utilisation. This phenomenon can be illustrated by adding a constraint to the problem, in the form of a single doorway providing egress from the room. A *bottleneck* will occur as large numbers of workers attempt to move their chairs through the door simultaneously, as shown in figure 1.4.

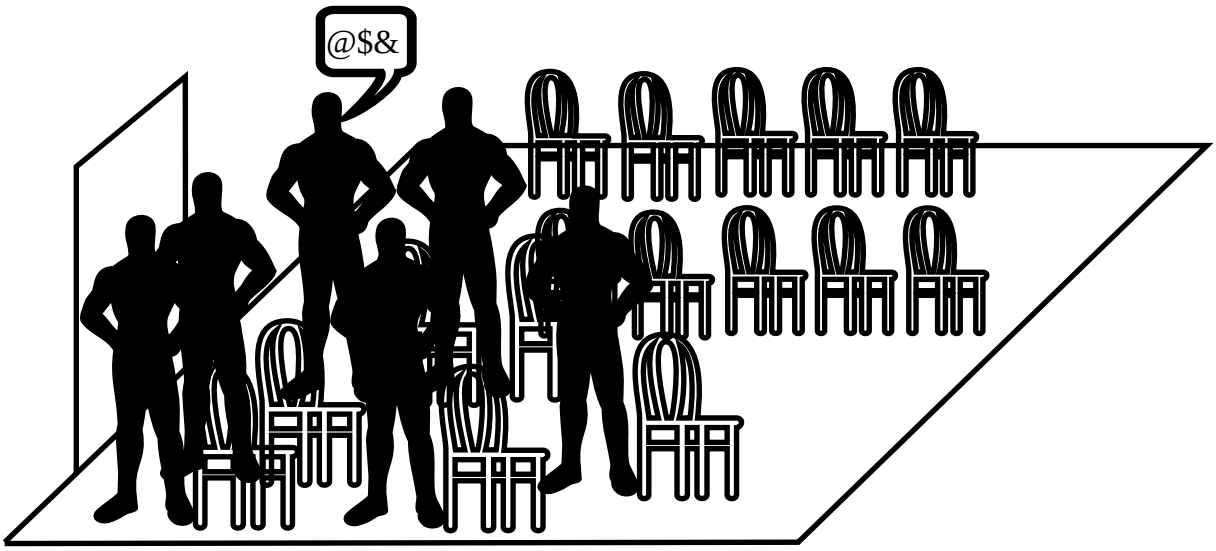


Figure 1.4: Bottleneck caused by doorway

The delays caused by this bottleneck may be so great that the time taken to empty the room of chairs by this large number of workers may in fact be *longer* than the original time taken by the single worker. In this case, reducing the number of workers can alleviate the bottleneck and thus reduce solution time.

1.1.3 Control

All parallel solutions of a problem require some form of control. This may be as simple as the control needed to determine what will constitute a task and to ascertain when the problem has been solved satisfactorily. More complex problems may require control at several stages of their solution. For example, solution time could be improved when clearing the room if a controller was placed at the door to schedule its usage. This control would ensure that no time was wasted by two (or more) workers attempting to exit simultaneously and then having to "reverse" to allow a single worker through. An alternative to this explicit *centralised* control would be some form of *distributed* control. Here the workers themselves could have a way of preventing simultaneous access, for example, if two (or more) workers reach the door at the same time then the biggest worker will always go first while the others wait.

Figure 1.5(a) shows the sequential approach to solving a problem. Computation is applied to the problem domain to produce the desired results. The controlled parallel approach shown in figure 1.5(b) achieves a parallel implementation of the same problem via three steps. In step 1, the problem domain is divided into a number of sub-problems, in this case four. Parallel processing is introduced in step 2 to enable each of the sub-problems to be computed in parallel to produce sub-results. In step 3, these results must now be collated to achieve the desired final results. Control is necessary in steps 1 and 3 to divide the problem amongst the workers and then to collect and collate the results that the workers have independently produced.

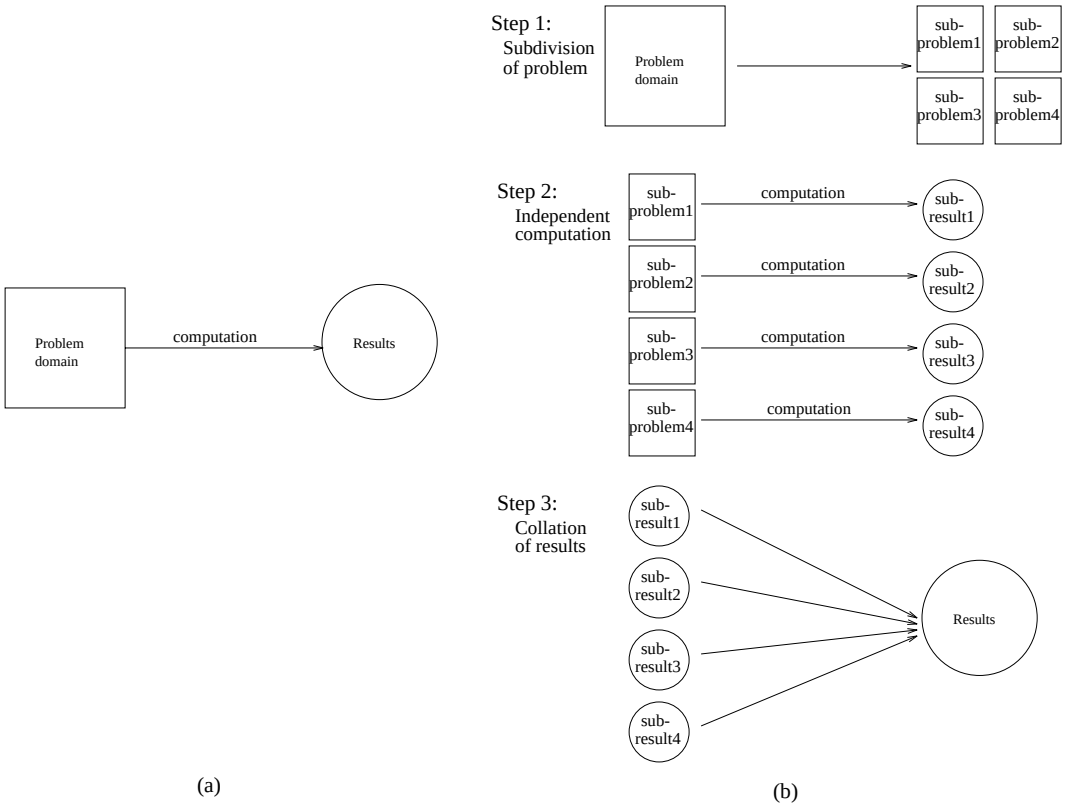


Figure 1.5: Control required in (a) a Sequential versus (b) a parallel implementation

1.2 Classification of Parallel Systems

A traditional sequential computer conforms to the *von Neumann* model. Shown in figure 1.6, this model comprises a processor, an associated memory, an input/output interface and various busses connecting these devices. The processor in the von Neumann model is the single computational unit responsible for the functions of fetching, decoding and executing a program's instructions. Parallel processing may be added to this architecture through pipelining using multiple functional units within a single computational unit or by replicating entire computational units (which may contain pipelining). With pipelining, each functional unit repeatedly performs the same operation on data which is received from the preceding functional unit. So in the simplest case, a pipeline for a computational unit could consist of three functional units, one to fetch the instructions from memory, one to decode these instructions and one to execute the decoded instructions. As we saw with the automobile assemblage example, a pipeline is only as effective as its slowest component. Any delay in the pipeline has repercussions for the whole system.

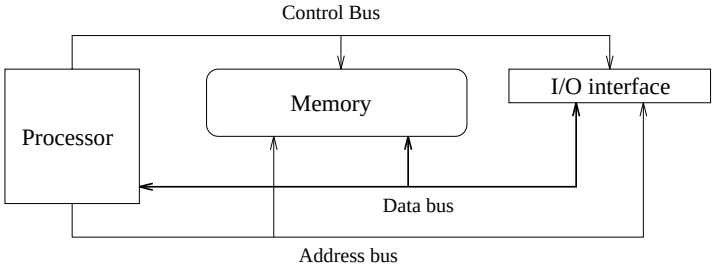


Figure 1.6: Von Neumann model architecture

Vector processing was introduced to provide efficient execution of program loops on large array data structures. By providing multiple registers as special vector registers to be used alongside the central processing unit, a vector processor is able to perform the *same operation* on *all elements* of a vector simultaneously. This simultaneous execution on every element of large arrays can produce significant performance improvements over conventional *scalar processing*. However, often problems need to be reformulate to benefit from this form of parallelism. A large number of scientific problems, such as weather forecasting, nuclear research and seismic data analysis, are well suited to vector processing.

Replication of the entire computational unit, the *processor*, allows individual tasks to be executed on different processors. Tasks are thus sometimes referred to as *virtual processors* which are allocated a physical processor on which to run. The completion of each task contributes to the solution of the problem.

Tasks which are executing on distinct processors at any point in time are said to be running in *parallel*. It may also be possible to execute several tasks on a single processor. Over a period of time the impression is given that they are running in parallel, when in fact at any point in time only *one* task has control of the processor. In this case we say that the tasks are being performed *concurrently*, that is their execution is being shared by the same processor. The difference between parallel tasks and concurrent tasks is shown in figure 1.7.

The workers which perform the computational work and co-operate to facilitate the solution of a problem on a parallel computer are known as *processing elements* and are often abbreviated as *PEs*. A processing element consists of a processor, one or more tasks, and the software to enable the co-operation with other processing elements. A parallel system comprises of *more than one* processing element.

1.2.1 Flynn's taxonomy

The wide diversity of computer architectures that have been proposed, and in a large number of cases realised, since the 1940's has led to the desire to classify the designs to facilitate evaluation and comparison. Classification requires a means of identifying distinctive architectural or behavioural features of a machine.

In 1972 Flynn proposed a classification of processors according to a macroscopic view of their principal interaction patterns relating to instruction and data *streams* [21]. The term stream was used by Flynn to

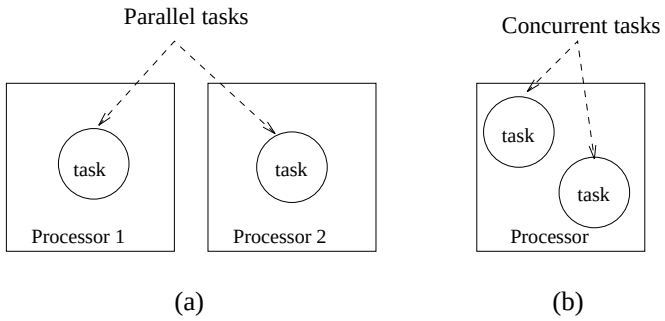


Figure 1.7: (a) Parallel tasks (b) Concurrent tasks

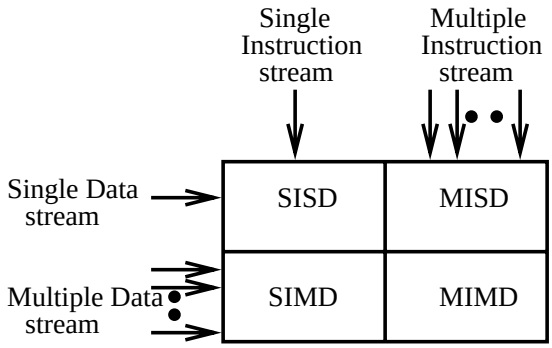


Figure 1.8: Flynn's taxonomy for processors

refer to the sequence of instructions to be executed, or data to be operated on, by the processor. What has become known as *Flynn's taxonomy* thus categorises architectures into the four areas shown in figure 1.8.

Since its inception, Flynn's taxonomy has been criticised as being too broad and has thus been enlarged by several other authors, for example, Shore in 1973 [55], Treleaven, Brownbridge and Hopkins in 1982 [58], Basu in 1984 [7], and perhaps one of the most detailed classifications was given by Hockney and Jesshope in 1988 [31].

Real architectures are, of course, much more complex than Flynn suggested. For example, an architecture may exhibit properties from more than one of his classes. However, if we are not too worried about the minute details of any individual machine then Flynn's taxonomy serves to separate fundamentally different architectures into four broad categories. The classification scheme is simple (which is one of the main reasons for its popularity) and thus useful to show an overview of the concepts of multiprocessor computers.

SISD: Single Instruction Single Data embraces the conventional sequential, or von Neumann, processor. The single processing element executes instructions sequentially on a single data stream. The operations are thus ordered in time and may be easily traced from start to finish. Modern adaptations of this uniprocessor use some form of pipelining technique to improve performance and, as demonstrated by the Cray supercomputers, minimise the length of the component interconnections to reduce signal propagation times [54].

SIMD: Single Instruction Multiple Data machines apply a single instruction to a group of data items simultaneously. A master instruction is thus acting over a vector of related operands. A number of processors, therefore, obey the same instruction in the same cycle and may be said to be executing in strict *lock-step*. Facilities exist to exclude particular processors from participating in a given instruction cycle. Vector processors, for example the Cyber 205, Fujitsu FACOM VP-200 and NEC SX1, and array processors, such as the DAP [53], Goodyear MPP (Massively Parallel Processor) [8], or the Connection Machine CM-1 [30], may be grouped in this category.

MISD: Multiple Instruction Single Data Although part of Flynn's taxonomy, no architecture falls obviously into the MISD category. One the closest architecture to this concept is a pipelined computer. Another is systolic array architectures which derives their from the medical term "systole" used to describe the rhythmic contraction of chambers of the heart. Data arrives from different directions at regular intervals to be combined at the "cells" of the array. The Intel iWarp system was designed to support systolic computation [4]. Systolic arrays are well suited to specially designed algorithms rather than general purpose computing [40, 41].

MIMD: Multiple Instruction Multiple Data The processors within the MIMD classification autonomously obey their own instruction sequence and apply these instructions to their own data. The processors are, therefore, no longer bound to the synchronous method of the SIMD processors and may choose to operate *asynchronously*. By providing these processors with the ability to communicate with each other, they may interact and therefore, co-operate in the solution of a single problem. This interaction has led to MIMD systems sometimes being classified as *tightly coupled* if the degree of interaction is high, or *loosely coupled* if the degree of interaction is low.

Two methods are available to facilitate this interprocessor communication. *Shared memory* systems allow the processors to communicate by reading and writing to a common address space. Controls are necessary to prevent processors updating the same portion of the shared memory simultaneously. Examples of such shared memory systems are the Sequent Balance [57] and the Alliant FX/8 [17].

In *distributed memory* systems, on the other hand, processors address only their private memory and communicate by passing messages along some form of communication path. Examples of MIMD processors from which such distributed memory systems can be built are the Intel i860 [50],the Inmos transputer [33] and Analog Devices SHARC processor.

The conceptual difference between shared memory and distributed memory systems of MIMD processors is shown in figure 1.9. The interconnection method for the shared memory system, figure 1.9(a), allows all the processors to be connected to the shared memory. If two, or more, processors wish to access the same portion of this shared memory at the same time then some arbitration mechanism must be used to ensure only one processor accesses that memory portion at a time. This problem of memory contention may restrict the number of processors that can be interconnected

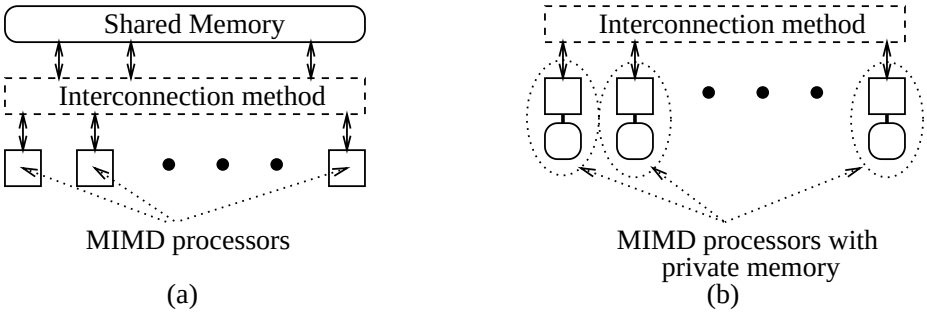


Figure 1.9: Systems of MIMD processors (a) shared memory (b) distributed memory

using the shared memory model. The interconnection method of the distributed memory system, figure 1.9(b), connects the processors in some fashion and if one, or more, processors wish to access another processor's private memory, it, or they, can only do so by sending a message to the appropriate processor along this interconnection network. There is thus no memory contention as such. However, the density of the messages that result in distributed memory systems may still limit the number of processors that may be interconnected, although this number is generally larger than that of the shared memory systems.

Busses have been used successfully as an interconnection structure to connect low numbers of processors together. However, if more than one processor wishes to send a message on the bus at the same time, an arbiter must decide which message gets access to the bus first. As the number of processors increases, so the contention for use of the bus grows. Thus a bus is inappropriate for large multiprocessor systems. An alternative to the bus is to connect processors via dedicated links to form large networks. This removes the bus-contention problem by spreading the communication load across many independent links.

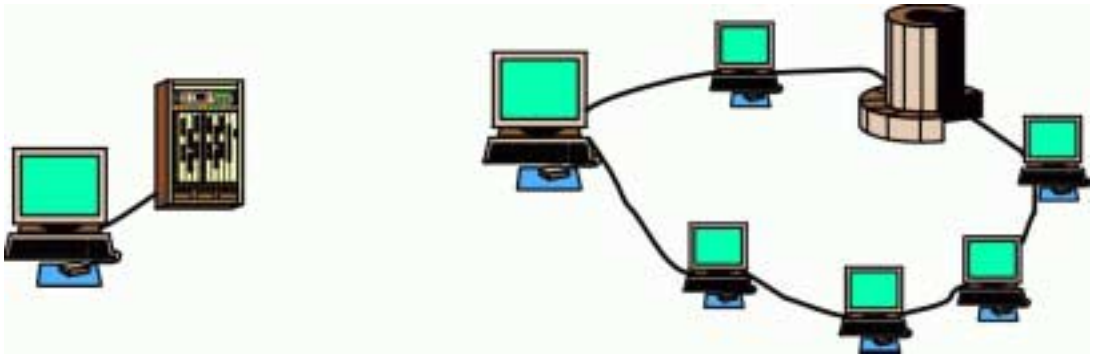
1.2.2 Parallel versus Distributed systems

Distributed memory MIMD systems consist of autonomous processors together with their own memory which co-operate in the solution of a single complex problem. Such systems may consist of a number of interconnected, dedicated processor and memory nodes, or interconnected "stand-alone" workstations. To distinguish between these two, the former configuration is refered to as a (dedicated) *parallel* system, while the latter is known as a *distibuted* system, as shown in figure 1.10.

The main distinguishing features of these two systems are typically the computation-to-communication ratio and the cost. Parallel systems make use of fast, "purpose-built" (and thus expensive) communication infrastructures, while distributed systems rely on exisiting network facilities such as ethernet, which are significantly slower and suceptible to other non-related traffic.

The advantage of distributed systems is that they may consist of a cluster of existing workstations which can be used by many (sequential) users when not employed in a parallel capacity. A number of valuable tools have been developed to enable these workstations to act in parallel, such as Parallel Virtual Machine (PVM), and Message Passing Interface (MPI). These provide an easy framework for coupling heterogeneous computers including, workstations, mainframes and even parallel systems.

However, while some of the properties of a distributed computing system may be different from those of a parallel system, many of the underlying concepts are equivalent. For example, both systems achieve co-operation between computational units by passing messages, and each computational unit has its own distinct memory. Thus, the ideas presented in this tutorial should prove equally useful to the reader faced with implementing his or her realistic rendering problem on either system.



(a) (b)

Figure 1.10: (a) Parallel system (b) Distributed system

1.3 The Relationship of Tasks and Data

The implementation of any problem on a computer comprises two components:

- the algorithm chosen to solve the problem; and,
- the domain of the problem which encompasses all the data requirements for that problem.

The algorithm interacts with the domain to produce the result for the problem, as shown diagrammatically in figure 1.11.

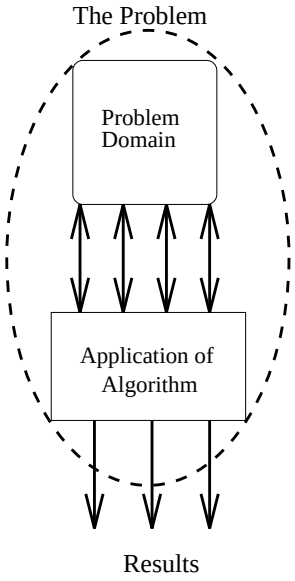


Figure 1.11: The components of a problem

A sequential implementation of the problem means that the entire algorithm and domain reside on a single processor. To achieve a parallel implementation it is necessary to divide the problem's components in some manner amongst the parallel processors. Now no longer resident on a single processor, the components will have to interact within the multiprocessor system in order to obtain the same result. This co-operation requirement introduces a number of novel difficulties into any parallel implementation which are not present in the sequential version of the same problem.

1.3.1 Inherent difficulties

User confidence in any computer implementation of a problem is bolstered by the successful termination of the computation and the fact that the results meet design specifications. The reliability of modern computer architectures and languages is such that any failure of a sequential implementation to complete successfully will point automatically to deficiencies in either the algorithm used or data supplied to the program. In addition to these possibilities of failure, a parallel implementation may also be affected by a number of other factors which arise from the manner of the implementation:

Deadlock: An active parallel processor is said to be deadlocked if it is waiting indefinitely for an event which will never occur. A simple example of deadlock is when two processors, using synchronised communication, attempt to send a message to each other at the same time. Each process will then wait for the other process to perform the corresponding input operation which will never occur.

Data consistency: In a parallel implementation, the problem's data may be distributed across several processors. Care has to be taken to ensure:

- if multiple copies of the same data item exists then the value of this item is kept consistent;
- mutual exclusion is maintained to avoid several processors accessing a shared resource simultaneously; and,
- the data items are fetched from remote locations efficiently in order to avoid processor idle time.

While there is meaningful computation to be performed, a sequential computer is able to devote 100% of its time for this purpose. In a parallel system it may happen that some of the processors become idle, not because there is no more work to be done, but because current circumstances prevent those processors being able to perform any computation.

Parallel processing introduces communication overheads. The effect of these overheads is to introduce latency into the multiprocessor system. Unless some way is found to minimise communication delays, the percentage of time that a processor can spend on useful computation may be significantly affected. So, as well as the factors affecting the successful termination of the parallel implementation, one of the fundamental considerations also facing parallel programmers is the *computation to communication* ratio.

1.3.2 Tasks

Subdividing a single problem amongst many processors introduces the notion of a task. In its most general sense, a task is a unit of computation which is assigned to a processor within the parallel system. In any parallel implementation a decision has to be taken as to what exactly constitutes a task. The *task granularity* of a problem is a measure of the amount of computational effort associated with any task. The choice of granularity has a direct bearing on the computation to communication ratio. Selection of too large a granularity may prevent the solution of the problem on a large parallel system, while too fine a granularity may result in significant processor idle time while the system attempts to keep processors supplied with fresh tasks.

On completion of a sequential implementation of a problem, any statistics that may have been gathered during the course of the computation, may now be displayed in a straightforward manner. Furthermore, the computer is in a state ready to commence the next sequential program. In a multiprocessor system, the statistics would have been gathered at each processor, so after the solution of the problem the programmer is still faced with the task of collecting and collating these statistics. To ensure that the multiprocessor system is in the correct state for the next parallel program, the programmer must also ensure that all the processors have *terminated gracefully*.

1.3.3 Data

The problem domains of many rendering applications are very large. The size of these domains are typically far more than can be accommodated within the local memory of any processing element (or indeed in the memory of many sequential computers). Yet it is precisely these complex problems that we wish to solve using parallel processing.

Consider a multiprocessor system consisting of sixty-four processing elements each with 4 MBytes of local memory. If we were to insist that the entire problem domain were to reside at each processing element then we would be restricted to solving problems with a maximum domain of 4 MBytes. The total memory within the system is $64 \times 4 = 256$ MBytes. So, if we were to consider the memory of the multiprocessor system as a whole, then we could contemplate solving problems with domains of up to 256 MBytes in size; a far more attractive proposition. (If the problem domain was even larger than this, then we could also consider the secondary storage devices as part of the combined memory and that should be sufficient for most problems.)

There is a price to pay in treating the combined memory as a single unit. Data management strategies will be necessary to translate between the conceptual single memory unit and the physical distributed implementation. The aims of these strategies will be to keep track of the data items so that an item will always be available at a processing element when required by the task being performed. The distributed nature of the data items will thus be invisible to the application processes performing the computation. However, any delay between the application process requesting an item and this request being satisfied will result in idle time. As we will see, it is the responsibility of data management to avoid this idle time.

1.4 Evaluating Parallel Implementations

The chief reason for opting for a parallel implementation should be: *to obtain answers faster*. The time that the parallel implementation takes to compute results is perhaps the most natural way of determining the benefits of the approach that has been taken. If the parallel solution takes *longer* than any sequential implementation then the decision to use parallel processing needs to be re-examined. Other measurements, such as speed-up and efficiency, may also provide useful insight on the maximum scalability of the implementation.

Of course, there are many issues that need to be considered when comparing parallel and sequential implementations of the same problem, for example:

- Was the same processor used in each case?
- If not, what is the price of the sequential machine compared with that of the multiprocessor system?
- Was the algorithm chosen already optimised for sequential use, that is, did the data dependencies present preclude an efficient parallel implementation?

1.4.1 Realisation Penalties

If we assume that the same processor was used in both the sequential and parallel implementation, then we should expect,, that the time to solve the problem decreases as more processing elements are added. The best we can reasonably hope for is that two processing elements will solve the problem twice as quickly, three processing elements three times faster, and n processing elements, n times faster. If n is sufficiently large then by this process, we should expect our large scale parallel implementation to produce the answer in a tiny fraction of the sequential computation, as shown by the "optimum time" curve in the graph in figure 1.12.

However, in reality we are unlikely to achieve these optimised times as the number of processors is increased. A more realistic scenario is that shown by the curve "actual times" in figure 1.12. This curve shows an initial decrease in time taken to solve the example problem on the parallel system up to a certain number of processing elements. Beyond this point, adding more processors actually leads to an *increase* in computation time.

Failure to achieve the optimum solution time means that the parallel solution has suffered some form of *realisation* penalty. A realisation penalty can arise from two sources:

- an *algorithmic* penalty; and,
- an *implementation* penalty.

The algorithmic penalty stems from the very nature of the algorithm selected for parallel processing. The more inherently sequential the algorithm, the less likely the algorithm will be a good candidate for parallel processing.

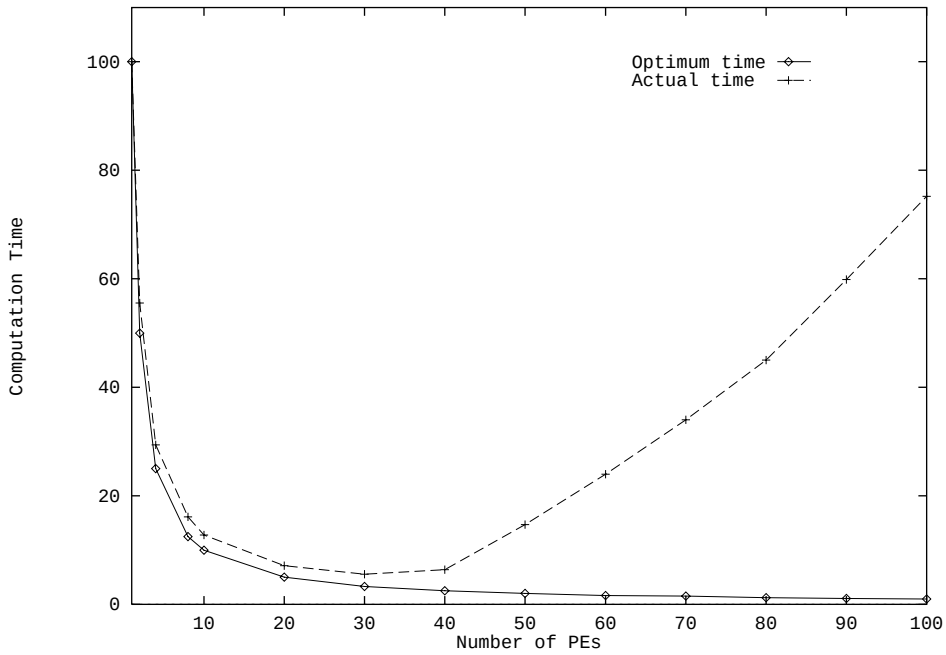


Figure 1.12: Optimum and actual parallel implementation times

Aside: It has also been shown, albeit not conclusively, that the more experience the writer of the parallel algorithm has in sequential algorithms, the less parallelism that algorithm is likely to exhibit [13].

This sequential nature of an algorithm and its implicit data dependencies will translate, in the domain decomposition approach, to a requirement to *synchronise* the processing elements at certain points in the algorithm. This can result in processing elements standing idle awaiting messages from other processing elements. A further algorithmic penalty may also come about from the need to reconstruct sequentially the results generated by the individual processors into an overall result for the computation.

Solving the same problem twice as fast on two processing elements implies that those two processing elements must spend 100% of their time on computation. We know that a parallel implementation requires some form of communication. The time a processing element is forced to spend on communication will naturally impinge on the time a processor has for computation. Any time that a processor cannot spend doing useful computation is an implementation penalty. Implementation penalties are thus caused by:

- **the need to communicate**

  As mentioned above, in a multiprocessor system, processing elements need to communicate. This communication may not only be that which is necessary for a processing element's own actions, but in some architectures, a processing element may also have to act as a intermediate for other processing elements' communication.

- **idle time**

  Idle time is any period of time when an application process is available to perform some useful computation, but is unable to do so because either there is no work locally available, or its current task is suspended awaiting a synchronisation signal, or a data item which has yet to arrive.

  It is the job of the local task manager to ensure that an application process is kept supplied with work. The computation to communication ratio within the system will determine how much time a task manager has to fetch a task before the current one is completed. A *load imbalance* is said to exist if some processing elements still have tasks to complete, while the others do not.

While synchronisation points are introduced by the algorithm, the management of data items for a processing element is the job for the local data manager. The domain decomposition approach means that the problem domain is divided amongst the processing elements in some fashion. If an application process requires a data item that is not available locally, then this must be fetched from some other processing element within the system. If the processing element is unable to perform other useful computation while this fetch is being performed, for example by means of multi-threading as discussed in section 3.5.2, then the processing element is said to be idle

- **concurrent communication, data management and task management activity**

  Implementing each of a processing element's activities as a separate concurrent process on the same processor, means that the physical processor has to be shared. When another process other than the application process is scheduled then the processing element is not performing useful computation even though its current activity is necessary for the parallel implementation.

The fundamental goal of the system software is to minimise the implementation penalty. While this penalty can never be removed, intelligent communication, data management and task scheduling strategies can avoid idle time and significantly reduce the impact of the need to communicate.

1.4.2 Performance Metrics

Solution time provides a simple way of evaluating a parallel implementation. However, if we wish to investigate the relative merits of our implementation then further insight can be gained by additional metrics. A range of metrics will allow us to compare aspects of different implementations and perhaps provide clues as to how overall system performance may be improved.

Speed-up

A useful measure of any multiprocessor implementation of a problem is *speed-up*. This relates the time taken to solve the problem on a single processor machine to the time taken to solve the same problem using the parallel implementation. We will define the speed-up of a multiprocessor system in terms of the elapsed time that is taken to complete a given problem, as follows:

$$\text{Speed-up} = \frac{\text{elapsed time of a uniprocessor}}{\text{elapsed time of the multiprocessors}} \tag{1.1}$$

The term *linear speed-up* is used when the solution time on an n processor system is n times faster than the solution time on the uniprocessor. This linear speed-up is thus equivalent to the optimum time shown in section 1.4.1. The optimum and actual computation times in figure 1.12 are represented as a graph of linear and actual speed-ups in figure 1.13. Note that the actual speed-up curve increases until a certain point and then subsequently decreases. Beyond this point we say that the parallel implementation has suffered a *speed-down*.

The third curve in figure 1.13 represents so-called *super-linear speed-up*. In this example, the implementation on 20 processors has achieved a computation time which is approximately 32 times faster than the uniprocessor solution. It has been argued, see [19], that it is not possible to achieve a speed-up greater than the number of processors used. While in practice it certainly is possible to achieve super-linear speed-up, such implementation may have exploited "unfair" circumstances to obtain such timings. For example, most modern processors have a limited amount of cache memory with an access time significantly faster compared with a standard memory access. Two processors would have double the amount of this cache memory. Given we are investigating a fixed size problem, this means that a larger proportion of the problem domain is in the cache in the parallel implementation than in the sequential implementation. It is not unreasonable, therefore, to imagine a situation where the two processor solution time is more than twice as fast than the uniprocessor time.

Although super-linear speed-up is desirable, in this tutorial we will assume a "fair" comparison between uniprocessor and multiprocessor implementations. The results that are presented in the case studies thus make no attempt to exploit any hardware advantages offered by the increasing number of processors. This will enable the performance improvements offered by the proposed system software extensions to be highlighted without being masked by any variations in underlying hardware. In practice, of course, it would be

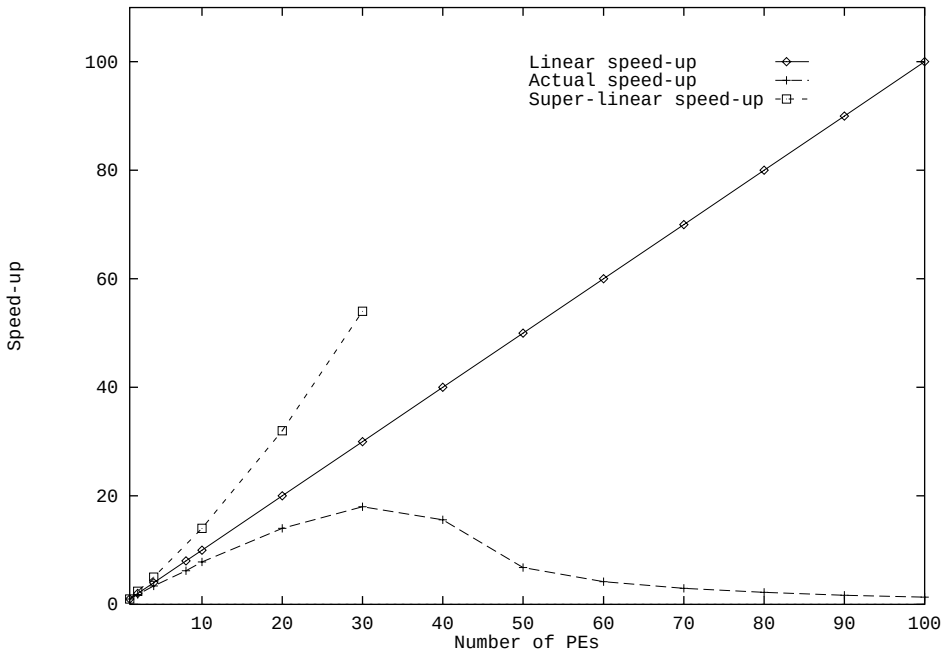


Figure 1.13: Linear and actual speed-ups

foolish to ignore these benefits and readers are encouraged to "squeeze every last ounce of performance" out of their parallel implementation.

Two possibilities exist for determining the "elapsed time of a uniprocessor". This could be the time obtained when executing:

1. an optimised sequential algorithm on a single processor, T_s; or,

2. the parallel implementation on *one* processing element, T_1.

The time taken to solve the problem on n processing elements we will term T_n. The difference between how the two sequential times are obtained is shown in figure 1.14. There are advantages in acquiring both these sequential times. Comparing the parallel to the optimised sequential implementation highlights any algorithmic efficiencies that had to be sacrificed to achieve the parallel version. In addition, none of the parallel implementation penalties are hidden by this comparison and thus the speed-up is not exaggerated. One of these penalties is the time taken simply to supply the data to the processing element and collect the results.

The comparison of the single processing element with the multiple processing element implementation shows how well the problem is "coping" with an increasing number of processing elements. Speed-up calculated as $\frac{T_1}{T_n}$, therefore, provides the indication as to the *scalability* of the parallel implementation. Unless otherwise stated, we will use this alternative for speed-up in the case studies in this book as it better emphasizes the performance improvements brought about by the system software we shall be introducing.

As we can see from the curve for "actual speed-up" in figure 1.13, the speed-up obtained for that problem increased to a maximum value and then subsequently decreased as more processing elements were added. In 1967 Amdahl presented what has become known as "Amdahl's law" [3]. This "law" attempts to give a maximum bound for speed-up from the nature of the algorithm chosen for the parallel implementation. We are given an algorithm in which the proportion of time that needs to be spent on the purely sequential parts is s, and the proportion of time that might be done in parallel is p, by definition. The total time for the algorithm on a single processor is $s + p = 1$ (where the 1 is for algebraic simplicity), and the maximum speed-up that can be achieved on n processors is:

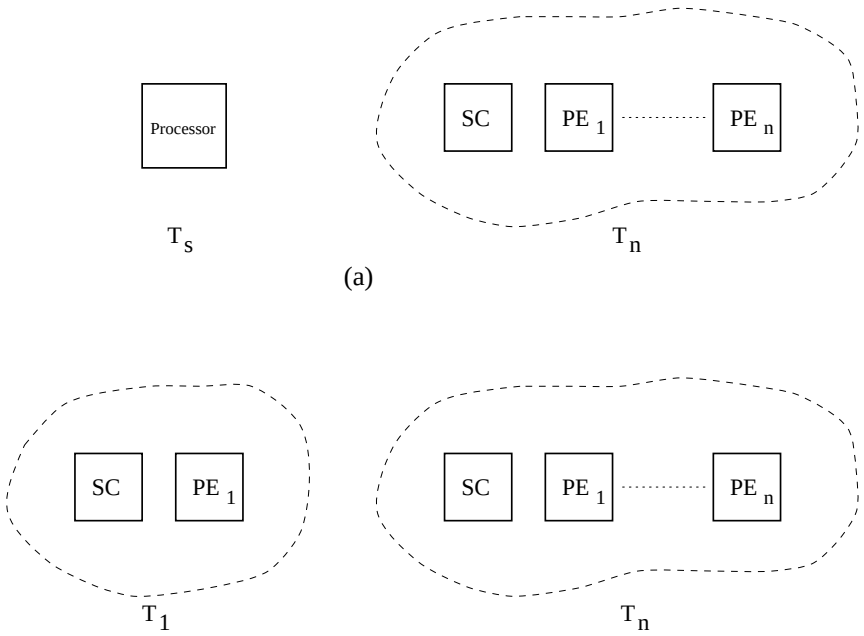


Figure 1.14: Systems used to obtain T_n and (a) T_s (b) T_1

$$\begin{aligned} \text{maximum speed-up} &= \frac{(s+p)}{s+\frac{p}{n}} \\ &= \frac{1}{s+\frac{p}{n}} \end{aligned} \tag{1.2}$$

Figure 1.15 shows the maximum speed-up predicted by Amdahl's law for a sequential portion of an algorithm requiring 0.1%, 0.5%, 1% and 10% of the total algorithm time, that is $s = 0.001,\ 0.005,\ 0.01$ and 0.1 respectively. For 1000 processors the maximum speed-up that can be achieved for a sequential portion of only 1% is less than 91. This rather depressing forecast put a serious damper on the possibilities of massive parallel implementations of algorithms and led Gustafson in 1988 to issue a counter claim [26]. Gustafson stated that a problem size is virtually never independent of the number of processors, as it appears in equation (1.2), but rather:

. . . in practice, the problem size scales with the number of processors.

Gustafson thus derives a maximum speed-up of:

$$\begin{aligned} \text{maximum speed-up} &= \frac{(s+(p\times n))}{s+p} \\ &= n+(1-n)\times s \end{aligned} \tag{1.3}$$

This maximum speed-up according to Gustafson is also shown in figure 1.15. As the curve shows, the maximum achievable speed-up is nearly linear when the problem size is increased as more processing elements are added. Despite this optimistic forecast, Gustafson's premise is not applicable in a large number of cases. Most scientists and engineers have a particular problem they want to solve in as short a time as possible. Typically, the application already has a specified size for the problem domain. For example, in parallel radiosity we will be considering the diffuse lighting within a particular environment subdivided into

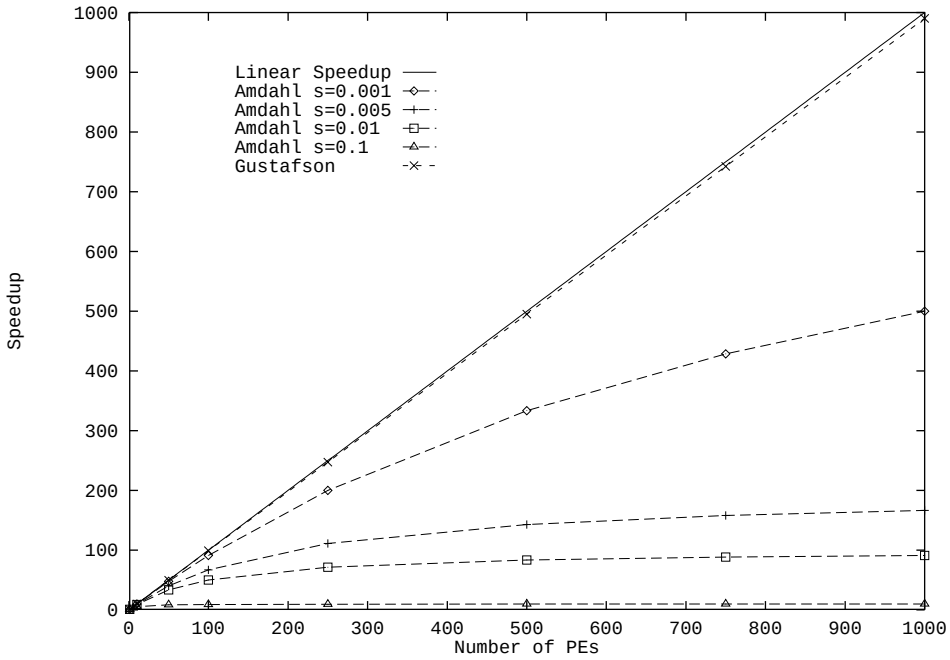


Figure 1.15: Example maximum speed-up from Amdahl and Gustafson's laws

a necessary number of patches. In this example it would be inappropriate for us to follow Gustafson's advice and increase the problem size as more processing elements were added to their parallel implementation, because to do so would mean either:

- the physical size of the three dimensional objects within the environmentwould have to be increased, which is of course not possible; or,
- the size of the patches used to approximate the surface would have to be reduced, thereby increasing the number of patches and thus the size of the problem domain.

This latter case is also not an option, because the computational method is sensitive to the size of the patches relative to their distances apart. Artificially significantly decreasing the size of the patches may introduce numerical instabilities into the method. Furthermore, artificially increasing the size of the problem domain may improve speed-up, but it *will not* improve the time taken to solve the problem.

For fixed sized problems it appears that we are left with Amdahl's gloomy prediction of the maximum speed-up that is possible for our parallel implementation. However, all is not lost, as Amdahl's assumption that an algorithm can be separated into a component which has to be executed sequentially and part which can be performed in parallel, may not be totally appropriate for the domain decomposition approach. Remember, in this model we are retaining the complete sequential algorithm and exploiting the parallelism that exists in the problem domain. So, in this case, an equivalent to Amdahl's law would imply that the data can be divided into two parts, that which must be dealt with in a strictly sequential manner and that which can executed in parallel. Any data dependencies will certainly imply some form of sequential ordering when dealing with the data, however, for a large number of problems such data dependencies may not exist. It may also be possible to reduce the effect of dependencies by clever scheduling.

The achievable speed-up for a problem using the domain decomposition approach is, however, bounded by the number of tasks that make up the problem. Solving a problem comprising a maximum of twenty tasks on more than twenty processors makes no sense. In practice, of course, any parallel implementation suffers from realisation penalties which increase as more processing elements are added. The actual speed-up obtained will thus be less than the maximum possible speed-up.

1.4.3 Efficiency

A relative efficiency based on the performance of the problem on one processor, can be a useful measure as to what percentage of a processor's time is being spent in useful computation. This, therefore, determines what the system overheads are. The relative efficiency we will measure as:

$$\text{Efficiency} = \frac{\text{speed-up} \times 100}{\text{number of processors}} \tag{1.4}$$

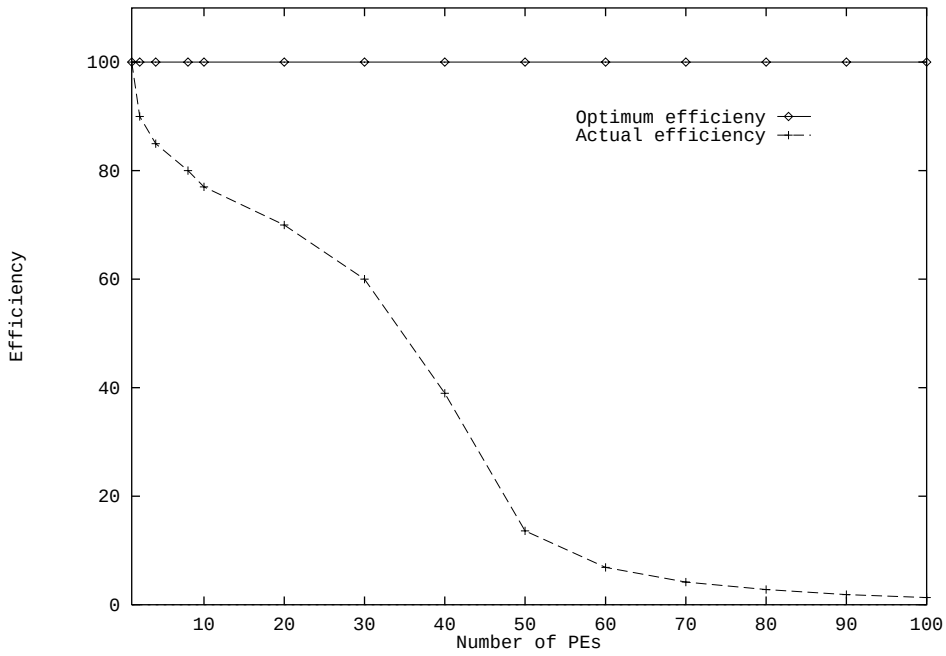


Figure 1.16: Optimum and actual processing element efficiency

Figure 1.16 shows the optimum and actual computation times given in figure 1.12 represented as processing element efficiency. The graph shows that optimum computation time, and therefore linear speed-up, equates to an efficiency of 100% for each processing element. This again shows that to achieve this level of efficiency every processing element must spend 100% of its time performing useful computation. Any implementation penalty would be immediately reflected by a decrease in efficiency. This is clearly shown in the curve for the actual computation times. Here the efficiency of each processing element decreases steadily as more are added until by the time 100 processing elements are incorporated, the realisation penalties are so high that each processing element is only able to devote just over 1% of its time to useful computation.

Optimum number of processing elements

Faced with implementing a fixed size problem on a parallel system, it may be useful to know the optimum number of processing elements on which this particular problem should be implemented in order to achieve the best possible performance. We term this optimum number n_{opt}. We shall judge the *maximum performance* for a particular problem with a fixed problem domain size, as the shortest possible time required to produce the desired results for a certain parallel implementation. This optimum number of processing elements may be derived directly from the "computation time" graph. In figure 1.12 the minimum actual computation time occurred when the problem was implemented on 30 processing elements. As figure 1.17 shows, this optimum number of processing elements is also the point on the horizontal axis in figure 1.13 at which the maximum speed-up was obtained.

The optimum number of processing elements is also the upper bound for the scalability of the problem for that parallel implementation. To improve the scalability of the problem it is necessary to re-examine the

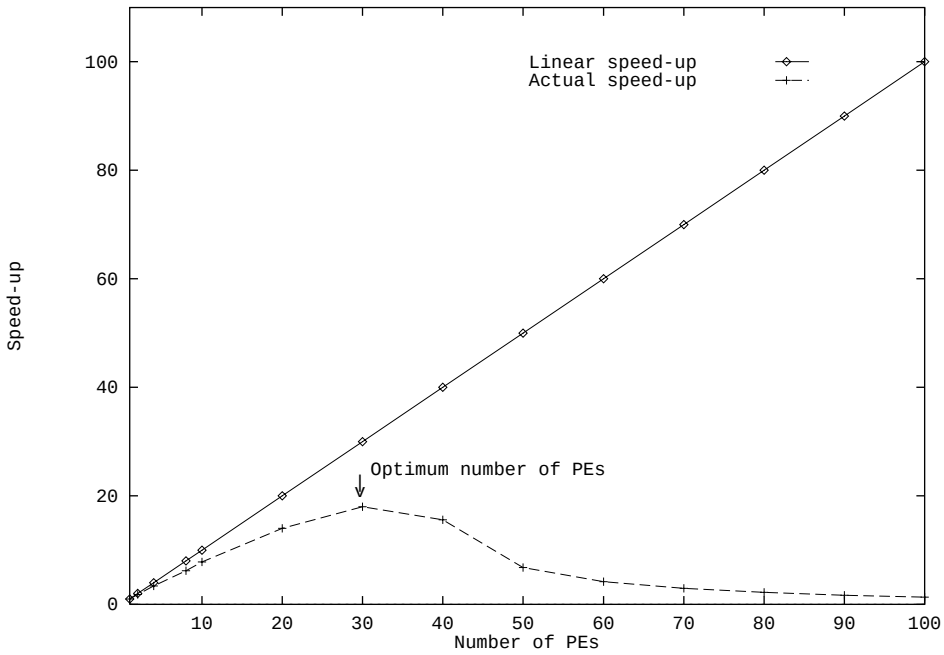


Figure 1.17: Optimum number of processing elements related to speed-up

decisions concerning the algorithm chosen and the make-up of the system software that has been adopted for supporting the parallel implementation. As we will see in the subsequent chapters, the correct choice of system software can have a significant effect on the performance of a parallel implementation.

Figure 1.18 shows the speed-up graphs for different system software decisions for the *same* problem. The goal of a parallel implementation may be restated as:

> *"to ensure that the optimum number of processing elements for your problem is greater than the number of processing elements physically available to solve the problem!"*

Other metrics

Computation time, speed-up and efficiency provide insight into how successful a parallel implementation of a problem has been. As figure 1.18 shows, different implementations of the same algorithm on the same multiprocessor system may produce very different performances. A multitude of other metrics have been proposed over the years as a means of comparing the relative merits of different architectures and to provide a way of assessing their suitability as the chosen multiprocessor machine.

The performance of a computer is frequently measured as the rate of some number of events per second. Within a multi-user environment the elapsed time to solve a problem will comprise the user's CPU time plus the system's CPU time. Assuming that the computer's clock is running at a constant rate, the user's CPU performance may be measured as:

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{clock rate (eg. 100MHz)}}$$

The average clock cycles per instruction (CPI) may be calculated as:

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

We can also compute the CPU time from the time a program took to run:

$$\text{CPU time} \quad = \quad \frac{\text{seconds}}{\text{program}}$$

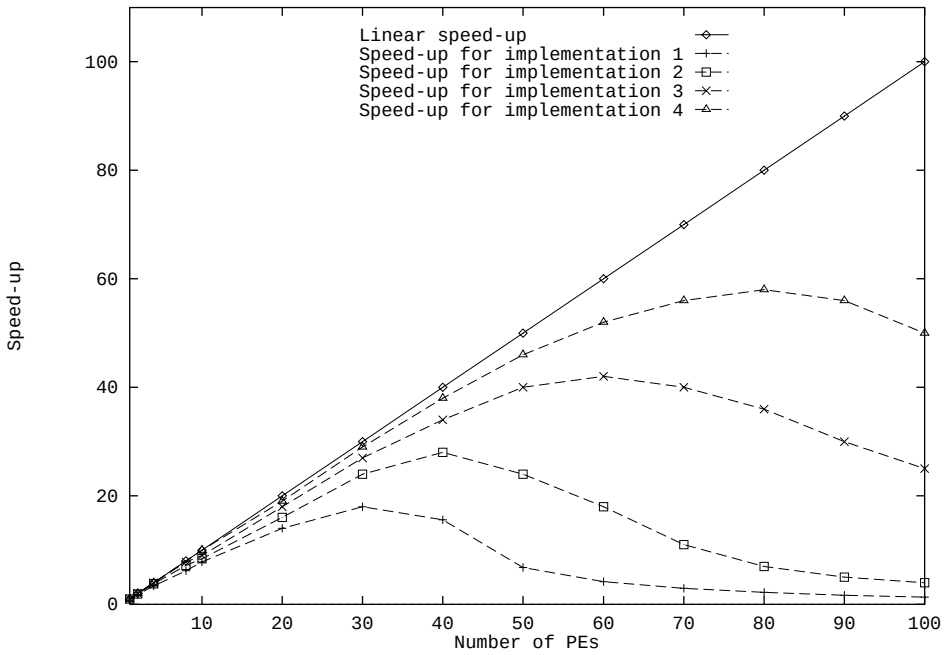


Figure 1.18: Speed-up graphs for different system software for the same problem

$$= \frac{\text{seconds}}{\text{clock cycle}} \times \frac{\text{clock cycles}}{\text{instructions}} \times \frac{\text{instructions}}{\text{program}}$$

Such a performance metric is dependent on:

Clock rate: this is determined by the hardware technology and the organisation of the architecture;

CPI: a function of the system organisation and the instruction set architecture; and,

Instruction count: this is affected by the instruction set architecture and the compiler technology utilised.

One of the most frequently used performance metrics is the *MIPS* rating of a computer, that is how many Million Instructions Per Second the computer is capable of performing:

$$\text{MIPS} = \frac{\text{instruction count}}{\text{execution time} \times 10^6} = \frac{\text{clock rate}}{\text{CPI} \times 10^6}$$

However, the MIPS value is dependent on the instruction set used and thus any comparison between computers with different instruction sets is not valid. The MIPS value may even vary between programs running on the same computer. Furthermore, a program which makes use of hardware floating point routines may take *less time* to complete than a similar program which uses a software floating point implementation, but the first program will have a *lower* MIPS rating than the second [28]. These anomalies have led to MIPS sometimes being referred to as *"Meaningless Indication of Processor Speed"*.

Similar to MIPS is the "Mega-FLOPS" (MFLOPS) rating for computers, where MFLOPS represents Million FLoating point OPerations per Second:

$$\text{MFLOPS} = \frac{\text{no. of floating point operations in a program}}{\text{execution time} \times 10^6}$$

MFLOPS is not universally applicable, for example a word processor utilising no floating point operations would register no MFLOPS rating. However, the same program executing on different machines should be comparable, because, although the computers may execute a different number of instructions,

they should perform the same number of operations, provided the set of floating point operations is consistent across both architectures. The MFLOPS value will vary for programs running on the same computer which have different mixtures of integer and floating point instructions as well as a different blend of "fast" and "slow" floating point instructions. For example, the *add* instruction often executes in less time than a *divide* instruction.

A MFLOPS rating for a single program can not, therefore, be generalised to provide a single performance metric for a computer. A suite of benchmark programs, such as the LINPACK or Livermore Loops routines, have been developed to allow a more meaningful method of comparison between machines. When examining the relative performance of computers using such benchmarks it is important to discover the *sustained* MFLOPS performance as a more accurate indication of the machines' potential rather than merely the *peak* MFLOPS rating, a figure that *"can be guaranteed never to be exceeded"*.

Other metrics for comparing computers include:

Dhrystone: A CPU intensive benchmark used to measure the integer performance especially as it pertains to system programming.

Whetstone: A synthetic benchmark without any vectorisable code for evaluating floating point performance.

TPS: Transactions Per Second measure for applications, such as airline reservation systems, which require on-line database transactions.

KLIPS: Kilo Logic Inferences Per Second is used to measure the relative inference performance of artificial intelligence machines

Tables showing the comparison of the results of these metrics for a number of architectures can be found in several books, for example [32, 37].

Cost is seldom an issue that can be ignored when purchasing a high performance computer. The desirability of a particular computer or even the number of processors within a system may be offset by the extraordinarily high costs associated with many high performance architectures. This prompted an early "law" by Grosch that the speed of a computer is proportional to its cost [25, 24]. Fortunately, although this is no longer completely true, multiprocessor machines are nevertheless typically more expensive than their general purpose counterparts. The parallel computer eventually purchased should provide acceptable computation times for an affordable price, that is maximise: *"the bangs per buck"* (performance per unit price).

Chapter 2

Task Scheduling

The efficient solution of a problem on a parallel system requires the computational performance of the processing elements to be fully utilised. Any processing element that is not busy performing useful computations is degrading overall system performance. Task scheduling strategies may be used to minimise these potential performance limitations.

2.1 Problem Decomposition

A problem may be solved on a parallel system by either exploiting the parallelism inherent in the algorithm, known as *algorithmic decomposition*, or by making use of the fact that the algorithm can be applied to different parts of the problem domain in parallel, which is termed *domain decomposition*. These two decomposition methods can be further categorised as shown in figure 2.1.

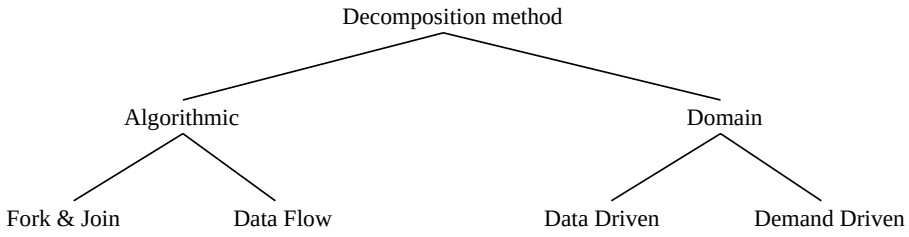


Figure 2.1: Methods of decomposing a problem to exploit parallelism

Over the years, an abundance of algorithms have been developed to solve a multitude of problems on sequential machines. A great deal of time and effort has been invested in the production of these sequential algorithms. Users are thus loathed to undertake the development of novel parallel algorithms, and yet still demand the performance that multiprocessor machines have to offer.

Algorithmic decomposition approaches to this dilemma have led to the development of compilers, such as those for High Performance Fortran, which attempt to parallelise automatically these existing algorithms. Not only do these compilers have to identify the parallelism hidden in the algorithm, but they also need to decide upon an effective strategy to place the identified segments of code within the multiprocessor system so that they can interact efficiently. This has proved to be an extremely hard goal to accomplish.

The domain decomposition approach, on the other hand, requires little or no modification to the existing sequential algorithm. There is thus no need for sophisticated compiler technology to analyse the algorithm. However, there will be a need for a parallel framework in the form of system software to support the division of the problem domain amongst the parallel processors.

2.1.1 Algorithmic decomposition

In algorithmic decomposition the algorithm itself is analysed to identify which of its features are capable of being executed in parallel. The finest granularity of parallelism is achievable at the operation level. Known as *dataflow*, at this level of parallelism the data "flows" between individual operands which are being executed in parallel [1]. An advantage of this type of decomposition is that little data space is required per processor [29], however, the communication overheads may be very large due to the very poor computation to communication ratio.

Fork & join parallelism, on the other hand, allocates portions of the algorithm to separate processors as the computation proceeds. These portions are typically several statements or complete procedures. The difference between the two algorithmic forms of decomposition is shown for a simple case in figure 2.2.

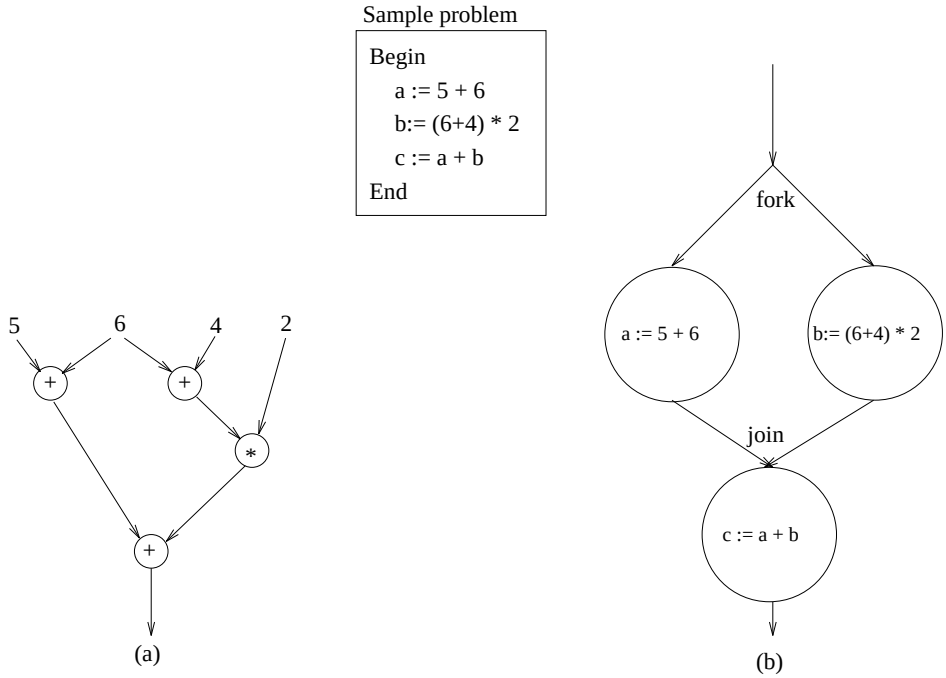


Figure 2.2: Algorithmic decomposition: (a) dataflow (b) fork & join

2.1.2 Domain decomposition

Instead of determining the parallelism inherent in the algorithm, domain decomposition examines the problem domain to ascertain the parallelism that may be exploited by solving the algorithm on distinct data items in parallel. Each parallel processor in this approach will, therefore, have a complete copy of the algorithm and it is the problem domain that is divided amongst the processors. Domain decomposition can be accomplished using either a data driven or demand driven approach.

As we shall see, given this framework, the domain decomposition approach is applicable to a wide range of problems. Adoption of this approach to solve a particular problem in parallel, consists of two steps:

1. **Choosing the appropriate sequential algorithm.**

   Many algorithms have been honed over a number of years to a high level of perfection for implementation on sequential machines. The data dependencies that these highly sequential algorithms exhibit may substantially inhibit their use in a parallel system. In this case alternative sequential algorithms which are more suitable to the domain decomposition approach will need to be considered.

2. **Analysis of the problem in order to extract the criteria necessary to determine the optimum system software.**

The system software provides the framework in which the sequential algorithm can execute. This system software takes care of ensuring each processor is kept busy, the data is correctly managed, and any communication within the parallel system is performed rapidly. To provide maximum efficiency, the system software needs to be tailored to the requirements of the problem. There is thus no *general purpose* parallel solution using the domain decomposition approach, but, as we shall see, a straightforward analysis of any problem's parallel requirements, will determine the correct construction of the system software and lead to an efficient parallel implementation.

Before commencing the detailed description of how we intend to tackle the solution of realistic rendering problems in parallel, it might be useful to clarify some of the terminology we shall be using.

2.1.3 Abstract definition of a task

The domain decomposition model solves a single problem in parallel by having multiple processors apply the same sequential algorithm to different data items from the problem domain in parallel. The lowest unit of computation within the parallel system is thus the application of the algorithm to one data item within the problem domain.

The data required to solve this unit of computation consists of two parts:

1. the *principal data items* (or PDIs) on which the algorithm is to be applied; and
2. *additional data items* (or ADIs) that may be needed to complete this computation on the PDIs.

For example, in ray tracing, we are computing the value at each pixel of our image plane. Thus these pixels would form our PDIs, while all the data describing the scene would constitute the ADIs. The problem domain is thus the pixels *plus* the scene description.

The application of the algorithm to a specified principal data item may be regarded as performing a single *task*. The task forms the elemental unit of computation within the parallel implementation. This is shown diagrammatically in figure 2.3.

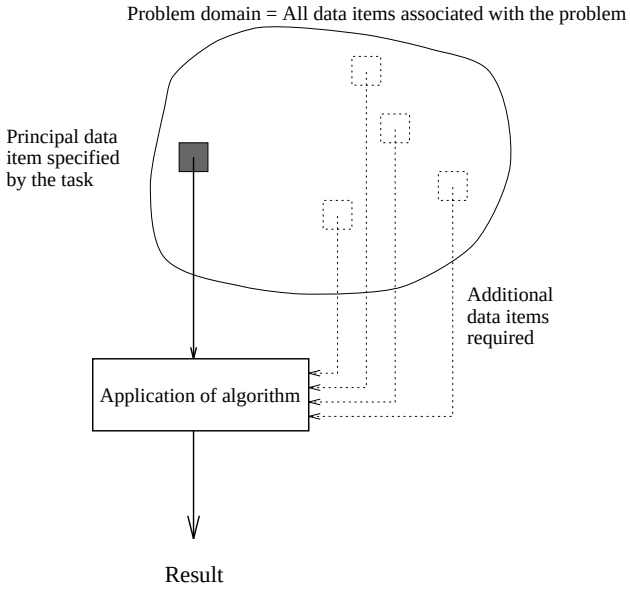


Figure 2.3: A task: the processing of a *principal* data item

2.1.4 System architecture

This tutorial is concentrating on implementing realistic rendering techniques on distributed memory systems (either a dedicated parallel machine or a distributed system of workstations). These processors may be

connected together in some manner to form a *configuration*. A *process* is a segment of code that runs concurrently with other processes on a single processor. Several processes will be needed at each processor to implement the desired application and provide the necessary system software support. A *processing element* consists of a single processor together with these application and system processes and is thus the building block of the *multiprocessor system*. (We shall sometimes use the abbreviation *PE* for processing element in the figures and code segments.) When discussing configurations of processing elements, we shall use the term *links* to mean the communication paths between processes.

Structure of the system controller

To provide a useful parallel processing platform, a multiprocessor system must have access to input/output facilities. Most systems achieve this by designating at least one processing element as the *system controller* (SC) with the responsibilities of providing this input/output interface, as shown in figure 2.4. If the need for input/output facilities becomes a serious bottleneck then more than one system controller may be required. Other processing elements perform the actual computation associated with the problem.

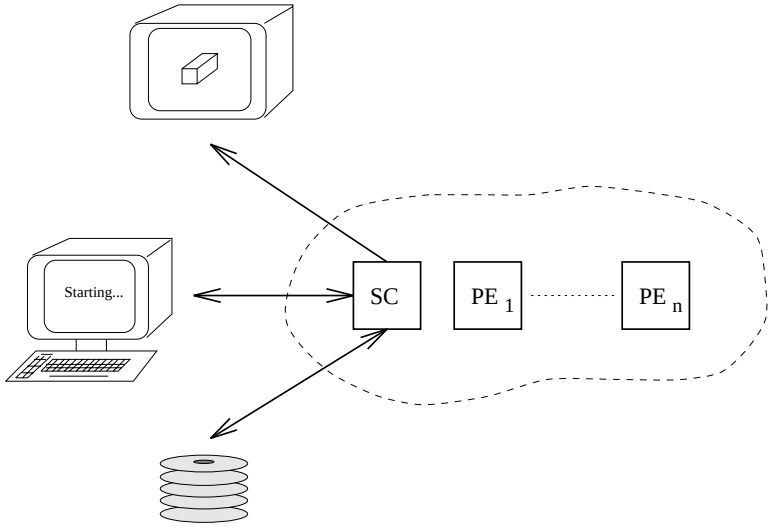


Figure 2.4: The system controller as part of a parallel system

In addition to providing the input/output facilities, the system controller may also be used to collect and collate results computed by the processing elements. In this case the system controller is in the useful position of being able to determine when the computation is complete and gracefully terminate the concurrent processes at every processing element.

2.2 Computational Models

The computational model chosen to solve a particular problem determines the manner in which work is distributed across the processors of the multiprocessor system. In our quest for an efficient parallel implementation we must maximise the proportion of time the processors spend performing necessary computation. Any imbalance may result in processors standing idle while others struggle to complete their allocated work, thus limiting potential performance. Load balancing techniques aim to provide an even division of computational effort to all processors.

The solution of a problem using the domain decomposition model involves each processing element applying the specified algorithm to a set of principal data items. The computational model ensures that every principal data item is acted upon and determines how the tasks are allocated amongst the processing elements. A choice of computation model exists for each problem. To achieve maximum system performance, the model chosen must see that the total work load is distributed evenly amongst the processing

elements. This balances the overheads associated with communicating principal data items to processing elements with the need to avoid processing element idle time. A simplified ray tracing example illustrate the differences between the computational models.

A sequential solution to this problem may be achieved by dividing the image plane into twenty-four distinct regions, with each region constituting a single principal data item, as shown in figure 2.5, and then applying the ray tracing algorithm at each of these regions in turn. There are thus twenty-four tasks to be performed for this problem where each task is to compute the pixel value at one area of the image plane. To understand the computational models, it is not necessary to know the details of the algorithm suffice to say that each principal data item represents an area of the image plane on which the algorithm can be applied to determine the value forthat position. We will assume that no additional data items are required to complete any task.

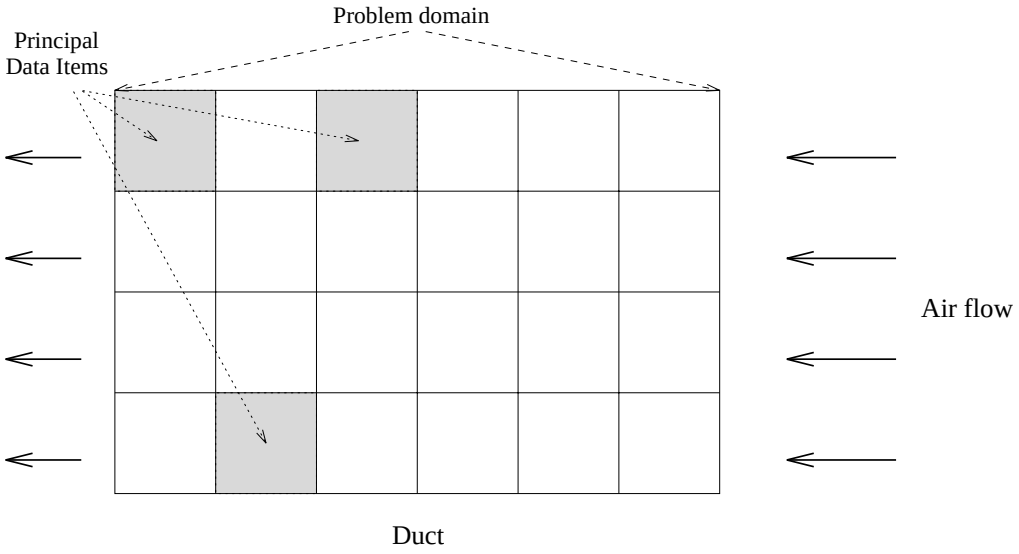


Figure 2.5: Principal data items for calculating the pixels in the image plane

2.2.1 Data driven model

The data driven model allocates all the principal data items to specific processing elements before computation commences. Each processing element thus knows *a priori* the principal data items to which they are required to apply the algorithm. Providing there is sufficient memory to hold the allocated set at each processing element, then, apart from the initial distribution, there is no further communication of principal data items. If there is insufficient local memory, then the extra items must be *fetched* as soon as memory space allows. This fetching of remote data items will be discussed further when data management is examined in Chapter 3.

Balanced data driven

In balanced data driven systems (also known as geometric decompositions), an equal number of principal data items is allocated to each processing element. This portion is determined simply by dividing the total number of principal data items by the number of processing elements:

$$\text{portion at each PE} = \frac{\text{number of principal data items}}{\text{number of PEs}}$$

If the number of principal data items is not an exact multiple of the number of processing elements, then

$$(number\ of\ principal\ data\ items)\ MOD\ (number\ of\ PEs)$$

will each have one extra principal data item, and thus perform one extra task. The required start task and the number of tasks is communicated by the system controller to each processing element and these can then apply the required algorithm to their allotted principal data items. This is similar to the way in which problems are solved on arrays of SIMD processors.

In this example, consider the simple ray tracing calculation for an empty scene. The principal data items (the pixels) may be allocated equally to three processing elements, labelled PE_1, PE_2 and PE_3, as shown in figure 2.6. In this case, each processing element is allotted eight principal data items.

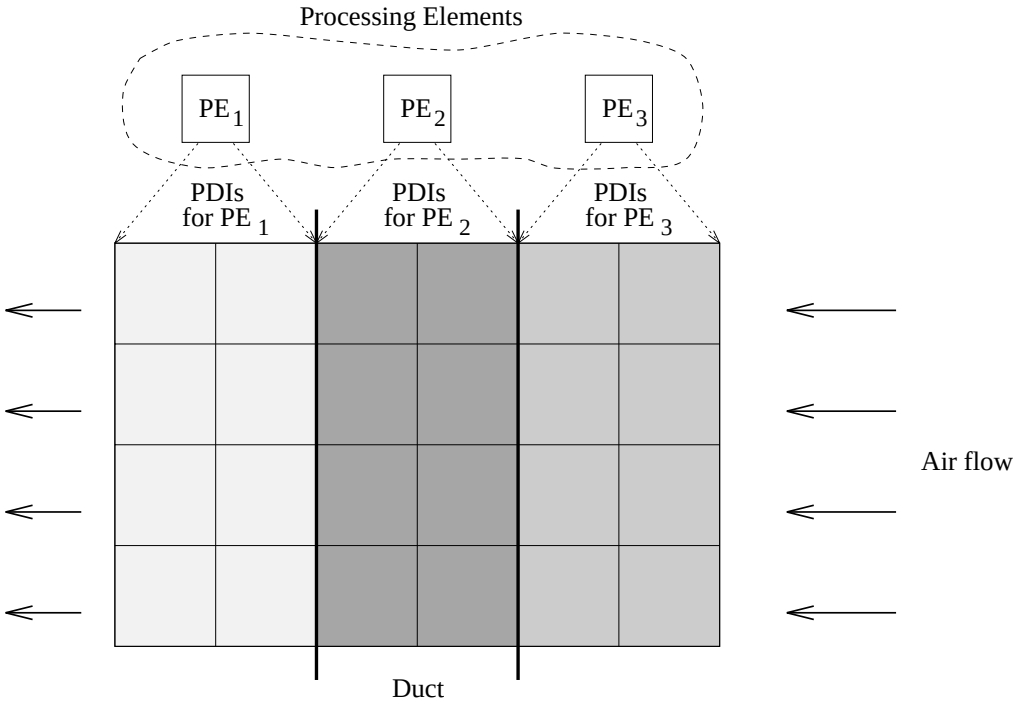


Figure 2.6: Equal allocation of data items to processing elements

As no further principal data item allocation takes place after the initial distribution, a balanced work load is only achieved for the balanced data driven computational model if the computational effort associated with each portion of principal data items is *identical*. If not, some processing elements will have finished their portions while others still have work to do. With the balanced data driven model the division of principal data items amongst processing elements is geometric in nature, that is each processing element simply may be allocated an equal number of principal data items irrespective of their position within the problem domain. Thus, to ensure a balanced work load, this model should only be used if the computational effort associated with each principal data item is the same, and preferably where the number of principal data items is an exact multiple of the number of processing elements. This implies *a priori* knowledge, but given this, the balanced data driven approach is the simplest of the computational models to implement.

Using figure 2.6, if the computation of each pixel 1 *time unit* to complete, then the sequential solution of this problem would take 24 *time units*. The parallel implementation of this problem using the three processing elements each allocated eight tasks should take approximately 8 *time units*, a third of the time required by the sequential implementation. Note, however, that the parallel solution will not be exactly one third of the sequential time as this would ignore the time required to communicate the portions from the system controller to the processing elements. This also ignores time required to receive the results back from the processing elements and for the system controller to collate the solution. A balanced data driven version of this problem on the three processing elements would more accurately take:

$$\text{Solution time} = \text{initial distribution} + \lceil \frac{24}{3} \rceil + \text{result collation}$$

Assuming low communication times, this model gives the solution in approximately one third of the time

of the sequential solution, close to the maximum possible linear speed-up. Solution of the same problem on five processing elements would give:

$$\text{Solution time} = \text{initial distribution} + \lceil \frac{24}{5} \rceil + \text{result collation}$$

This will be solved in even longer than the expected 4.8 *time units* as, in this case, one processing element is allocated 4 principal data items while the other four have to be apportioned 5. As computation draws to a close, one processing element will be idle while the four others complete their extra work. The solution time will thus be slightly more than 5 *time units*.

Unbalanced data driven

Differences in the computational effort associated with the principal data items will increase the probability of substantial processing element idle time if the simplistic balanced data driven approach is adopted. If the individual computation efforts differ, and are known *a priori*, then this can be exploited to achieve optimum load balancing.

The unbalanced data driven computational model allocates principal data items to processing elements based on their computational requirements. Rather than simply apportioning an equal number of tasks to each processing element, the principal data items are allocated to ensure that each processing element will complete its portion at approximately *the same time*.

For example, the complexity introduced into the ray tracing calculations by placing object into the scene, as shown in figure 2.7, will cause an increased computational effort required to solve the portions allocated to PE_1 and PE_2 in the balanced data driven model. This will result in these two processing elements still being busy with their computations long after the other processing element, PE_3, has completed its less computationally complex portion.

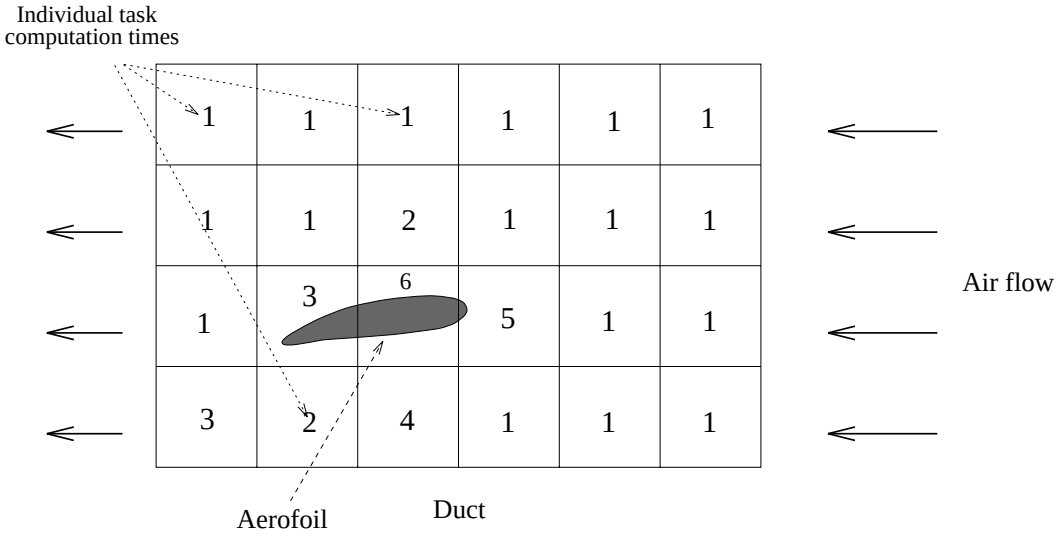


Figure 2.7: Unequal computational effort due to presence of objects in the scene

Should *a priori* knowledge be available regarding the computational effort associated with each principal data item then they may be allocated *unequally* amongst the processing elements, as shown in figure 2.8. The computational effort now required to process each of these unequal portions will be approximately the same, minimising any processing element idle time.

The sequential time required to solve the ray tracing with objects in the scene is now 42 *time units*. To balance the work load amongst the three processing elements, each processing element should compute for 14 *time units*. Allocation of the portions to each processing element in the unbalanced data driven model involves a preprocessing step to determine precisely the best way to subdivide the principal data items. The optimum compute time for each processing element can be obtained by simply dividing the

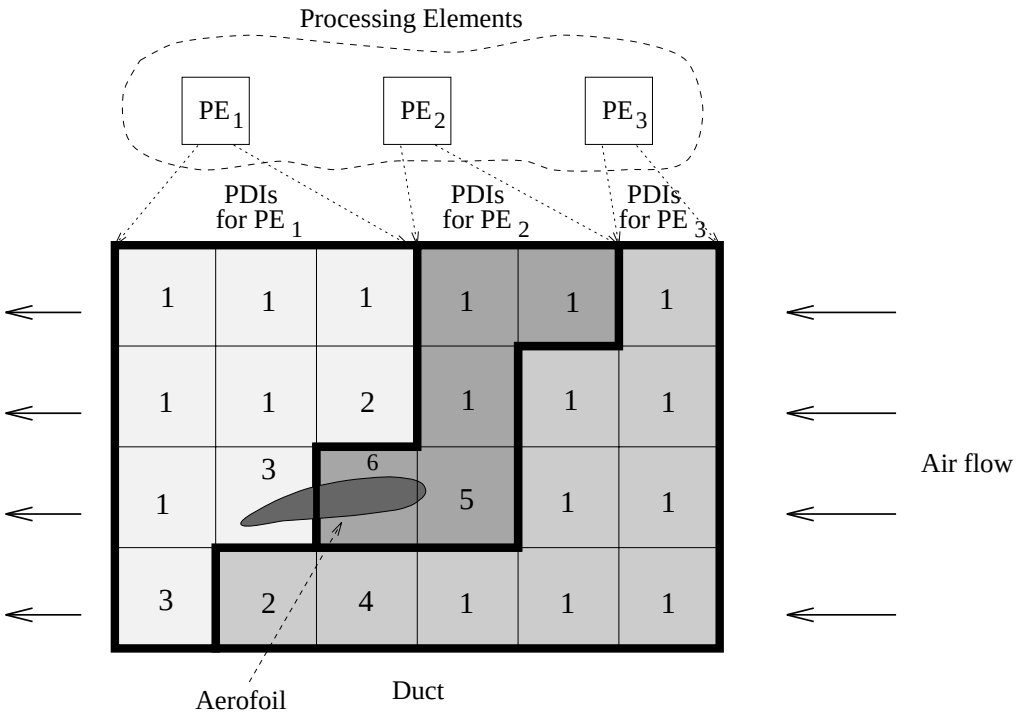


Figure 2.8: Unequal allocation of data items to processing elements to assist with load balancing

total computation time by the number of processing elements. If possible, no processing element should be allocated principal data items whose combined computation time exceeds this optimum amount. Sorting the principal data items in descending computation times can facilitate the subdivision.

The total solution time for a problem using the unbalanced data driven model is thus:

$$\begin{aligned}\text{Solution time} &= \text{preprocessing} + \text{distribution} \\ &\quad + \text{longest portion time} + \text{result collation}\end{aligned}$$

So comparing the naive balanced distribution from section 2.2.1

$$\begin{aligned}\text{Balanced solution time} &= \text{distribution} + 21 + \\ &\quad \text{result collation}\end{aligned}$$

$$\begin{aligned}&\text{Unbalanced solution time} = \\ &\quad \text{preprocessing} + \text{distribution} + 14 + \text{result collation}\end{aligned}$$

The preprocessing stage is a simple sort requiring far less time than the ray tracing calculations. Thus, in this example, the unbalanced data driven model would be significantly faster than the balanced model due to the large variations in task computational complexity.

The necessity for the preprocessing stage means that this model will take more time to use than the balanced data driven approach should the tasks have the same computation requirement. However, if there are variations in computational complexity and they are known, then the unbalanced data driven model is the most efficient way of implementing the problem in parallel.

2.2.2 Demand driven model

The data driven computational models are dependent on the computational requirements of the principal data items being known, or at least being predictable, before actual computation starts. Only with this knowledge can these data items be allocated in the correct manner to ensure an even load balance. Should the computational effort of the principal data items be unknown or unpredictable, then serious load balancing problems can occur if the data driven models are used. In this situation the demand driven computational model should be adopted to allocate work to processing elements evenly and thus optimise system performance.

In the demand driven computational model, work is allocated to processing elements *dynamically* as they become idle, with processing elements no longer bound to any particular portion of the principal data items. Having produced the result from one principal data item, the processing elements demand the next principal data item from some work supplier process. This is shown diagrammatically in figure 2.9 for the simple ray tracing calculation.

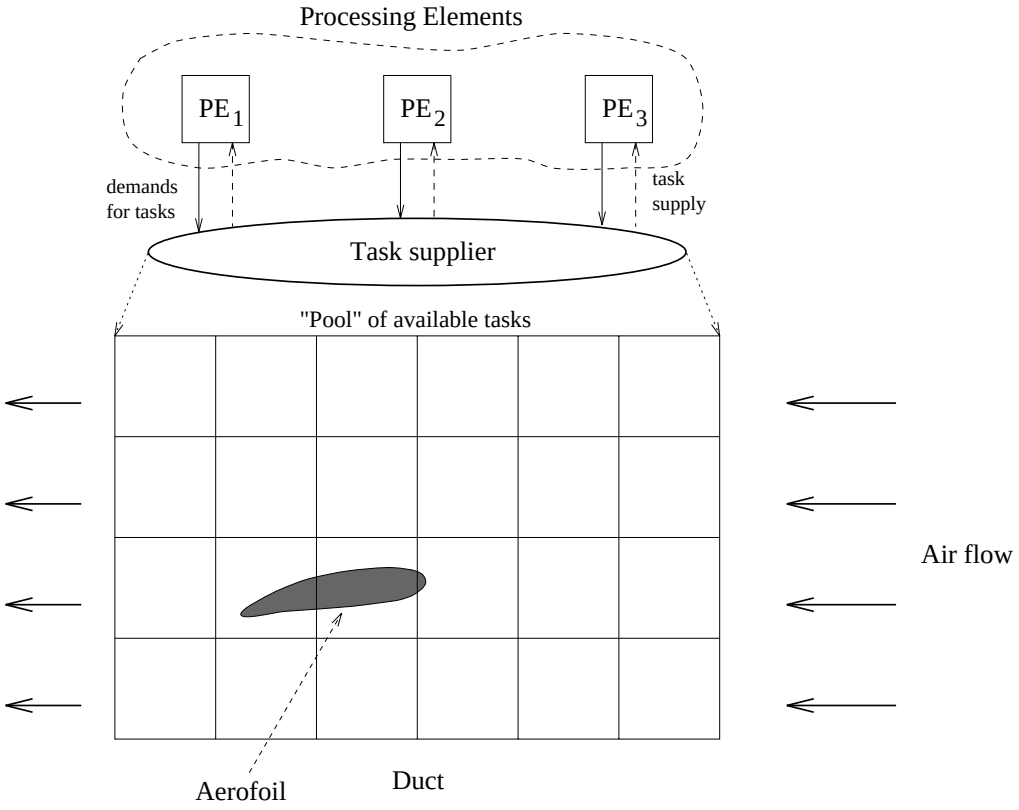


Figure 2.9: A demand driven model for a simple ray tracing calculation

Unlike the data driven models, there is no initial communication of work to the processing elements, however, there is now the need to send requests for individual principal data items to the supplier and for the supplier to communicate with the processing elements in order to satisfy these requests. To avoid unnecessary communication it may be possible to combine the return of the results from one computation with the request for the next principal data item.

The optimum time for solving a problem using this simple demand driven model is thus:

$$\text{Solution time} = \frac{2 \times \text{total communication time} + \text{total computation time for all PDIs}}{\text{number of PEs}}$$

This optimum computation time, $\frac{\text{total computation time for all PDIs}}{\text{number of PEs}}$, will only be possible if the work can be allocated so that all processing elements complete the last of their tasks at exactly the same time. If this is not so then some processing elements will still be busy with their final task while the others have completed. It may also be possible to reduce the communication overheads of the demand driven model by overlapping the communication with the computation in some manner. This possibility will be discussed later in section 2.3.

On receipt of a request, if there is still work to be done, the work supplier responds with the next available task for processing. If there are no more tasks which need to be computed then the work supplier may safely ignore the request. The problem will be solved when all principal data items have been requested and all the results of the computations on these items have been returned and collated. The dynamic allocation of work by the demand driven model will ensure that while some processing elements are busy with more computationally demanding principal data items, other processing elements are available to compute the less complex parts of the problem.

Using the computational times for the presence of objects in the scene as shown in figure 2.8, figure 2.10 shows how the principal data items may be allocated by the task supplier to the processing elements using a simple serial allocation scheme. Note that the processing elements do not complete the same number of tasks. So, for example, while processing elements 2 and 3 are busy completing the computationally complex work associated with principal data items 15 and 16, processing elements 1 can compute the less computationally taxing tasks of principal data items 17 and 18.

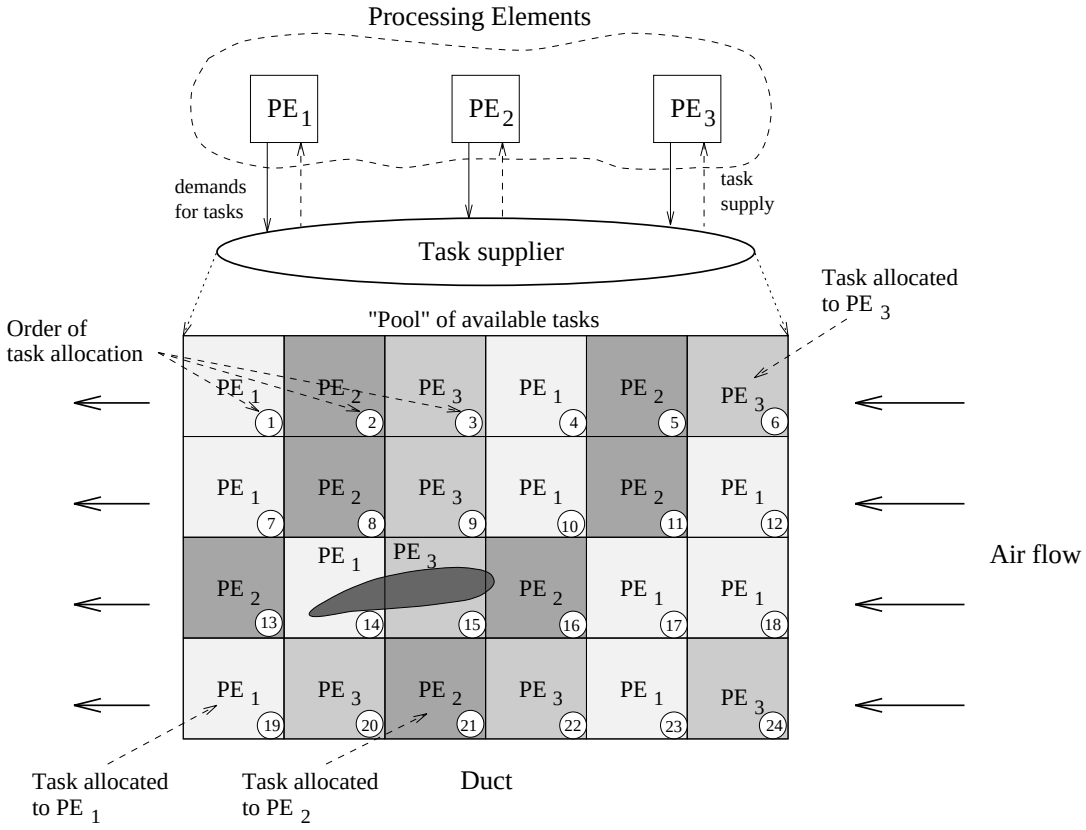


Figure 2.10: Allocation of principal data items using a demand driven model

The demand driven computational model facilitates dynamic load balancing when there is no prior knowledge as to the complexity of the different parts of the problem domain. Optimum load balancing is still dependent on all the processing elements completing the last of the work at the same time. An unbalanced solution may still result if a processing element is allocated a complex part of the domain towards the end of the solution. This processing element may then still be busy well after the other processing elements

have completed computation on the remainder of the principal data items and are now idle as there is no further work to do. To reduce the likelihood of this situation it is important that the computationally complex portions of the domain, the so called *hot spots*, are allocated to processing elements early on in the solution process. Although there is no *a priori* knowledge as to the exact computational effort associated with any principal data item (if there were, an unbalanced data driven approach would have been adopted), nevertheless, any insight as to possible hot spot areas should be exploited. The task supplier would thus assign principal data items from these areas first.

In the ray tracing example , while the exact computational requirement associated with the principal data items in proximity of the objects in the scene may be unknown, it is highly likely that the solution of the principal items in that area will more complex than those elsewhere. In this problem, these principal data items should be allocated first.

If no insight is possible then a simple serial allocation, as shown in figure 2.10, or spiral allocation, as shown in figure 2.11 or even a random allocation of principal data items will have to suffice. While a random allocation offers perhaps a higher probability of avoiding late allocation of principal data items from hot spots, additional effort is required when choosing the next principal data item to allocate to ensure that no principal data item is allocated more than once.

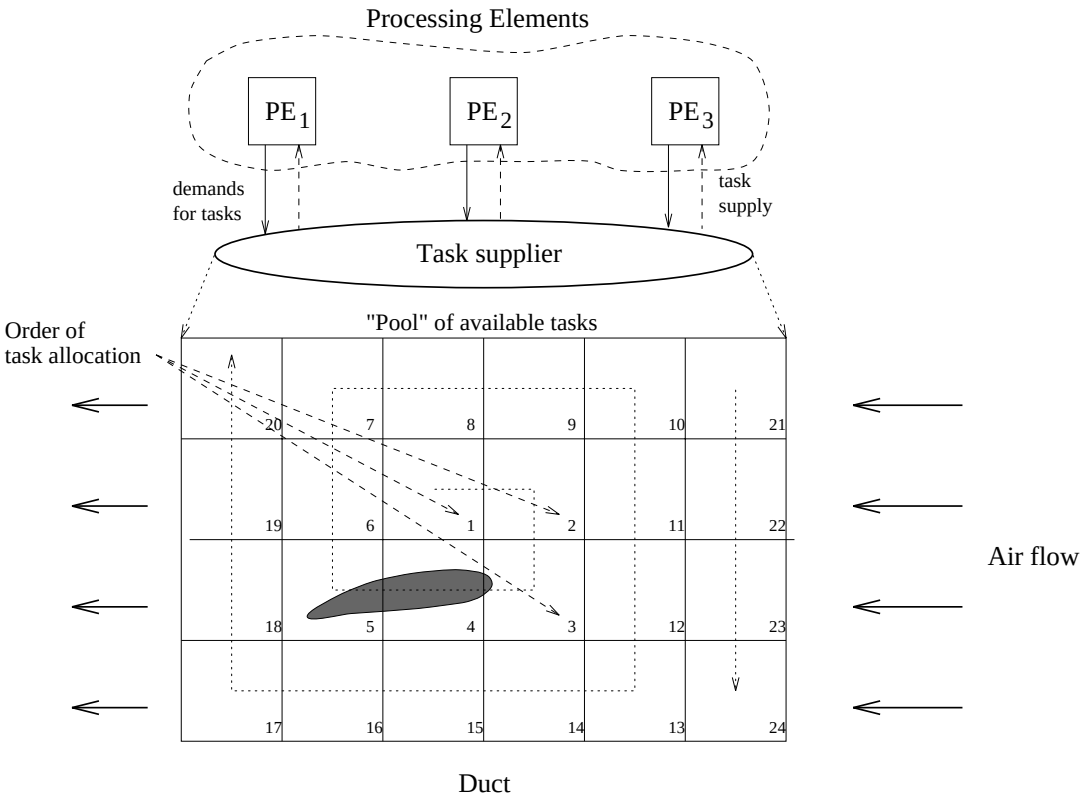


Figure 2.11: Allocation of principal data items in a spiral manner

As with all aspects of parallel processing, extra levels of sophistication can be added in order to exploit any information that becomes available as the parallel solution proceeds. Identifying possible hot spots in the problem domain may be possible from the computation time associated with each principal data item as these become known. If this time is returned along with the result for that principal data item, the work supplier can build a dynamic profile of the computational requirements associated with areas of the domain. This information can be used to adapt the allocation scheme to send principal data items from the possible hot spot regions. There is, of course, a trade off here between the possible benefits to load balancing in the early allocation of principal data items from hot spots, and the overhead that is introduced by the need to:

- time each computation at the processing elements;

- return this time to the work supplier;
- develop the time profile at the work supplier; and,
- adapt the allocation strategy to take into account this profile.

The benefits gained by such an adaptive scheme are difficult to predict as they are dependent on the problem being considered and the efficiency of the scheme implementation. The advice in these matters is always: "*implement a simple scheme initially and then add extra sophistication should resultant low system performance justify it.*"

2.2.3 Hybrid computational model

For most problems, the correct choice of computational model will either be one of the data driven strategies or the demand driven approach. However, for a number of problems, a hybrid computational model, exhibiting properties of both data and demand driven models, can be adopted to achieve improved efficiency. The class of problem that can benefit from the hybrid model is one in which an initial set of principal data items of known computational complexity may spawn an unknown quantity of further work.

In this case, the total number of principal data items required to solve the problem is *unknown* at the start of the computation, however, there are at least a known number of principal data items that must be processed first. If the computational complexity associated with these initial principal data items is unknown then a demand driven model will suffice for the whole problem, but if the computational complexity is known then one of the data driven models, with their lower communication overheads, should at least be used for these initial principal data items. Use of the hybrid model thus requires the computational model to be switched from data driven to demand driven mode as required.

2.3 Task Management

Task management encompasses the following functions:

- the definition of a task;
- controlling the allocation of tasks;
- distribution of the tasks to the processing elements; and,
- collation of the results, especially in the case of a problem with multiple stages.

2.3.1 Task definition and granularity

An *atomic element* may be thought of as a problem's lowest computational element within the sequential algorithm adopted to solve the problem. As introduced in section 2.1.2, in the domain decomposition model a single task is the application of this sequential algorithm to a principal data item to produce a result for the sub-parts of the problem domain. The task is thus the smallest element of computation for the problem within the parallel system. The *task granularity* (or *grain size*) of a problem is the number of atomic elements, which are included in one task. Generally, the task granularity remains constant for all tasks, but in some cases it may be desirable to alter dynamically this granularity as the computation proceeds. A task which includes only one atomic element is said to have the *finest granularity*, while a task which contains many is *coarser grained*, or has a *coarser granularity*. The actual definition of what constitutes a principal data item is determined by the granularity of the tasks.

A parallel system solves a problem by its constituent processing elements executing tasks in parallel. A *task packet* is used to inform a processing element which task, or tasks, to perform. This task packet may simply indicate which tasks require processing by that processing element, thus forming the lowest level of distributed work. The packet may include additional information, such as additional data items, which the tasks require in order to be completed.

To illustrate the differences in this terminology, consider again the simple ray tracing problem. The atomic element of a sequential solution of this problem could be to perform a single ray-object intersection

test. The principal data item is the pixel being computed and the additional data item required will be object being considered. A sequential solution of this problem would be for a single processing element to consider each ray-object intersection in turn. The help of several processing elements could substantially improve the time taken to perform the ray tracing.

The finest task granularity for the parallel implementation of this problem is for each task to complete one atomic element, that is perform one ray-object intersection. For practical considerations, it is perhaps more appropriate that each task should instead be to trace the complete path of a single ray. The granularity of each task is now the number of ray-object intersections required to trace this single ray and each pixel is a principal data item. A sensible task packet to distribute the work to the processing elements would include details about one or more pixels together with the necessary scene data (if possible, see Chapter 3).

To summarise our choices for this problem:

atomic element: to perform one ray-object intersection;

task: to trace the complete path of one ray (may consists of a number of atomic elements);

PDI: the pixel location for which we are computing the colour;

ADI: the scene data; and,

task packet: one or more rays to be computed.

Choosing the task granularity for the parallel implementation of a problem is not straightforward. Although it may be fairly easy to identify the atomic element for the sequential version of the problem, such a fine grain may not be appropriate when using many processing elements. Although the atomic element for ray tracing was specified as computing a single ray-object intersection in the above example, the task granularity for the parallel solution was chosen as computing the complete colour contribution at a particular pixel. If one atomic element had been used as the task granularity then additional problems would have introduced for the parallel solution, namely, the need for processors to to exchange partial results. This difficulty would have been exacerbated if, instead, the atomic element had been chosen as tracing a ray into a voxel and considering whether it does in fact intersect with an object there. Indeed, apart from the higher communication overhead this would have introduced, the issue of dependencies would also have to be checked to ensure, for example, that a ray was not checked against an object more than once.

As well as introducing additional communication and dependency overheads, the incorrect choice of granularity may also increase computational complexity variations and hinder efficient load balancing. The choice of granularity is seldom easy, however, a number of parameters of the parallel system can provide an indication as to the desirable granularity. The computation to communication ratio of the architecture will suggest whether additional communication is acceptable to avoid dependency or load balancing problems. As a general rule, where possible, data dependencies should be avoided in the choice of granularity as these imply unnecessary synchronisation points within the parallel solution which can have a significant effect on overall system performance.

2.3.2 Task distribution and control

The task management strategy controls the distribution of packets throughout the system. Upon receipt, a processing element performs the tasks specified by a packet. The composition of the task packet is thus an important issue that must be decided before distribution of the tasks can begin. To complete a task a processing element needs a copy of the algorithm, the principal data item(s), and any additional data items that the algorithm may require for that principal data item. The domain decomposition paradigm provides each processing element with a copy of the algorithm, and so the responsibility of the task packet is to provide the other information.

The principal data items form part of the problem domain. If there is sufficient memory, it may be possible to store the entire problem domain as well as the algorithm at each processing element. In this case, the inclusion of the principal data item as part of the task packet is unnecessary. A better method would be simply to include the identification of the principal data item within the task packet. Typically, the identification of a principal data item is considerably smaller, in terms of actual storage capacity, than the item itself. The communication overheads associated with sending this smaller packet will be significantly less than sending the principal data item with the packet. On receipt of the packet the processing element

could use the identification simply to fetch the principal data item from its local storage. The identification of the principal data item is, of course, also essential to enable the results of the entire parallel computation to be collated.

If the additional data items required by the task are known then they, or if possible, their identities, may also be included in the task packet. In this case the task packet would form an integral unit of computation which could be directly handled by a processing element. However, in reality, it may not be possible to store the whole problem domain at every processing element. Similarly, numerous additional data items may be required which would make their inclusion in the task packet impossible. Furthermore, for a large number of problems, the additional data items which are required for a particular principal data item may not be known in advance and will only become apparent as the computation proceeds.

A task packet should contain as a minimum either the identity, or the identity and actual principal data items of the task. The inability to include the other required information in the packet means that the parallel system will have to resort to some form of *data management*. This topic is described fully in Chapter 3.

2.3.3 Algorithmic dependencies

The algorithm of the problem may specify an order in which the work must be undertaken. This implies that certain tasks must be completed before others can commence. These dependencies must be preserved in the parallel implementation. In the worst case, algorithmic dependencies can prevent an efficient parallel implementation, as shown with the tower of toy blocks in figure 1.2. Amdahl's law, described in section 1.4, shows the implications to the algorithmic decomposition model of parallel processing of the presence of even a small percentage of purely sequential code. In the domain decomposition approach, algorithmic dependencies may introduce two phenomena which will have to be tackled:

- *synchronisation points* which have the effect of dividing the parallel implementation into a number of distinct stages; and,
- *data dependencies* which will require careful data management to ensure a consistent view of the data to all processing elements.

Multi-stage algorithms

Many problems can be solved by a single stage of computation, utilising known principal data items to produce the desired results. However, the dependencies inherent in other algorithms may divide computation into a number of distinct stages. The *partial results* produced by one stage become the principal data items for the following stage of the algorithm, as shown in figure 2.12. For example, many scientific problems involve the construction of a set of simultaneous equations, a distinct stage, and the subsequent solution of these equations for the unknowns. The partial results, in this case elements of the simultaneous equations, become the principal data for the tasks of the next stage.

Even a single stage of a problem may contain a number of distinct substages which must first be completed before the next substage can proceed. An example of this is the use of an iterative solver, such as the Jacobi method [22, 35], to solve a set of simultaneous equations. An iterative method starts with an approximate solution and uses it in a recurrence formula to provide another approximate solution. By repeatedly applying this process a sequence of solutions is obtained which, under suitable conditions, converges towards the exact solution.

Consider the problem of solving a set of six equations for six unknowns, $\mathbf{A}x = \mathbf{b}$. The Jacobi method will solve this set of equations by calculating, at each iteration, a new approximation from the values of the previous iteration. So the value for the x_i's at the n^{th} iteration are calculated as:

$$
\begin{aligned}
x_1^n &= \frac{b_i - a_{12}x_2^{n-1} - \ldots - a_{16}x_6^{n-1}}{a_{11}} \\
x_2^n &= \frac{b_i - a_{21}x_1^{n-1} - \ldots - a_{26}x_6^{n-1}}{a_{22}} \\
&\vdots \\
x_6^n &= \frac{b_i - a_{61}x_1^{n-1} - \ldots - a_{65}x_6^{n-1}}{a_{66}}
\end{aligned}
$$

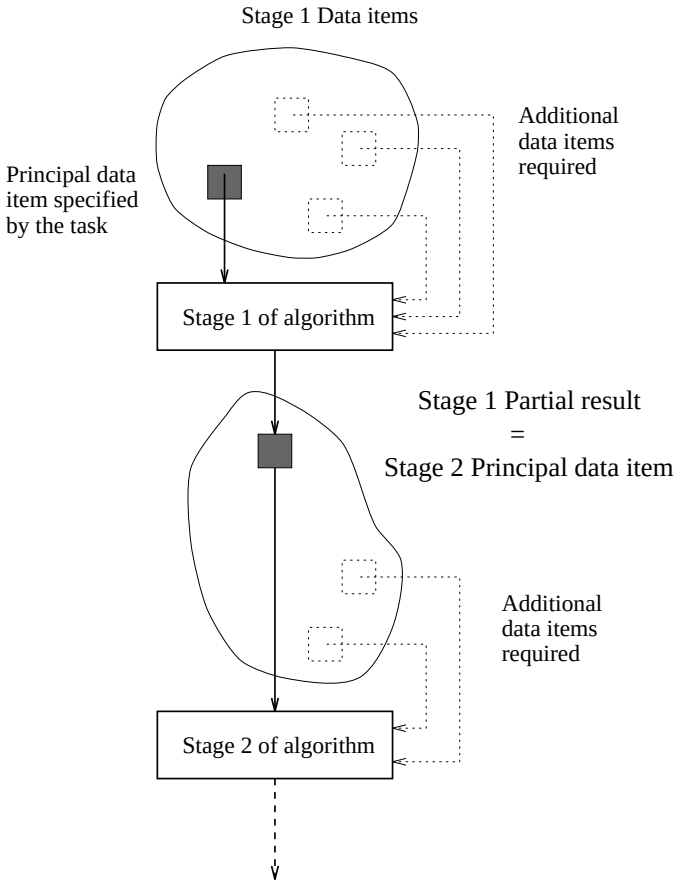


Figure 2.12: The introduction of partial results due to algorithmic dependencies

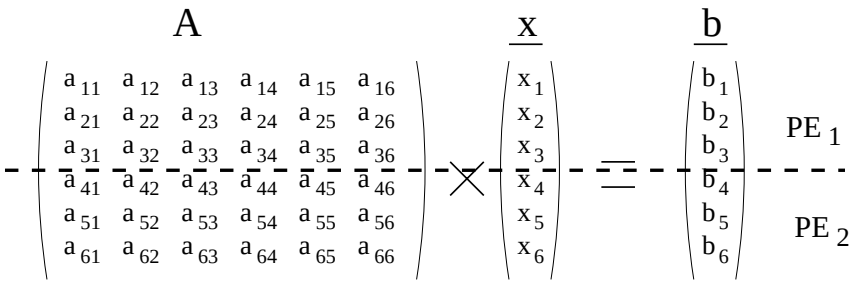


Figure 2.13: Solving an iterative matrix solution method on two processing elements

A parallel solution to this problem on two processing elements could allocate three rows to be solved to each processing element as shown in figure 2.13. Now PE_1 can solve the n^{th} iteration values $x_1^n,\ x_2^n$ and x_3^n in parallel with PE_2 computing the values of $x_4^n,\ x_5^n$ and x_6^n. However, neither processing element can proceed onto the $(n+1)^{st}$ iteration until both have finished the n^{th} iteration and exchanged their new approximations for the x_i^n's. Each iteration is, therefore, a substage which must be completed before the next substage can commence. This point is illustrated by the following code segment from PE_1:

```
PROCEDURE Jacobi() (* Executing on PE 1 *)
    Begin
        Estimate x[1] ... x[6]
        n := 0 (* Iteration number *)
        WHILE solution_not_converged DO
            Begin
                n := n + 1
                Calculate new x[1], x[2] & x[3] using old x[1] ... x[6]
                PARALLEL
                    SEND new x[1], x[2] & x[3] TO PE_2
                    RECEIVE new x[4], x[5] & x[6] FROM PE_2
            End
    End (* Jacobi *)
```

Data dependencies

The concept of dependencies was introduced in section 1.1.1 when we were unable to construct a tower of blocks in parallel as this required a strictly sequential order of task completion. In the domain decomposition model, data dependencies exist when a task may not be performed on some principal data item until another task has been completed. There is thus an implicit ordering on the way in which the task packets may be allocated to the processing elements. This ordering will prevent certain tasks being allocated, even if there are processing elements idle, until the tasks on which they are dependent have completed.

A linear dependency exists between each of the iterations of the Jacobi method discussed above. However, no dependency exists for the calculation of each x_i^n, for all i, as all the values they require, $x_j^{n-1},\ \forall j \neq i$, will already have been exchanged and thus be available at every processing element.

The Gauss-Seidel iterative method has long be preferred in the sequential computing community as an alternative to Jacobi. The Gauss-Seidel method makes use of new approximations for the x_i as soon as they are available rather than waiting for the next iteration. Provided the methods converge, Gauss-Seidel will converge more rapidly than the Jacobi method. So, in the example of six unknowns above, in the n^{th} the value of x_1^n would still be calculated as:

$$x_1^n = \frac{b_i - a_{12}x_2^{n-1} - \ldots - a_{16}x_6^{n-1}}{a_{11}},$$

but the x_2^n value would now be calculated by:

$$x_2^n = \frac{b_i - a_{21}\mathbf{x_1^n} - a_{23}x_3^{n-1} - \ldots - a_{26}x_6^{n-1}}{a_{22}}$$

Although well suited to sequential programming, the strong linear dependency that has been introduced, makes the Gauss-Seidel method poorly suited for parallel implementation. Now within each iteration no value of x_i^n can be calculated until all the values for $x_j^n,\ j < i$ are available; a strict sequential ordering of the tasks. The less severe data dependencies within the Jacobi method thus make it a more suitable candidate for parallel processing than the Gauss-Seidel method which is more efficient on a sequential machine.

It is possible to implement a hybrid of these two methods in parallel, the so-called "Block Gauss-Seidel - Global Jacobi" method. A processing element which is computing several rows of the equations, may use the Gauss-Seidel method for these rows as they will be computed sequentially within the processing element. Any values for x_i^n not computed locally will assume the values of the previous iteration, as in the Jacobi method. All new approximations will be exchanged at each iteration. So, in the example, PE_2 would calculate the values of x_4^n, x_5^n and x_6^n as follows:

$$x_4^n = \frac{b_i - a_{11}x_1^{n-1} - a_{12}x_2^{n-1} - a_{13}x_3^{n-1} - a_{15}x_5^{n-1} - a_{16}x_6^{n-1}}{a_{44}}$$

$$x_5^n = \frac{b_i - a_{11}x_1^{n-1} - a_{12}x_2^{n-1} - a_{13}x_3^{n-1} - a_{14}\mathbf{x_4^n} - a_{16}x_6^{n-1}}{a_{55}}$$

$$x_6^n = \frac{b_i - a_{11}x_1^{n-1} - a_{12}x_2^{n-1} - a_{13}x_3^{n-1} - a_{14}\mathbf{x_4^n} - a_{15}\mathbf{x_5^n}}{a_{66}}$$

2.4 Task Scheduling Strategies

2.4.1 Data driven task management strategies

In a data driven approach, the system controller determines the allocation of tasks prior to computation proceeding. With the unbalanced strategy, this may entail an initial sorting stage based on the known computational complexity, as described in section 2.2.1. A single task-packet detailing the tasks to be performed is sent to each processing element. The application processes may return the results upon completion of their allocated portion, or return individual results as each task is performed, as shown in this code segment:

```
PROCESS Application_Process()
    Begin
        RECEIVE task_packet FROM SC via R
        FOR i = start_task_id TO finish_task_id DO
            Begin
                result[i] := Perform_Algorithm(task[i])
                SEND result[i] TO SC via R
            End
    End (* Application_Process *)
```

In a data driven model of computation a processing element may initially be supplied with as many of its allocated principal data items as its local memory will allow. Should there be insufficient storage capacity a simple data management strategy may be necessary to prefetch the missing principal data items as computation proceeds and local storage allows. This is discussed further when considering the management of data in Chapter 3.

2.4.2 Demand driven task management strategies

Task management within the demand driven computational model is explicit. The work supplier process, which forms part of the system controller, is responsible for placing the tasks into packets and sending these packets to requesting processing elements. To facilitate this process, the system controller maintains a *pool* of already constituted task packets. On receipt of a request, the work supplier simply dispatches the next available task packet from this task pool, as can be seen in figure 2.14.

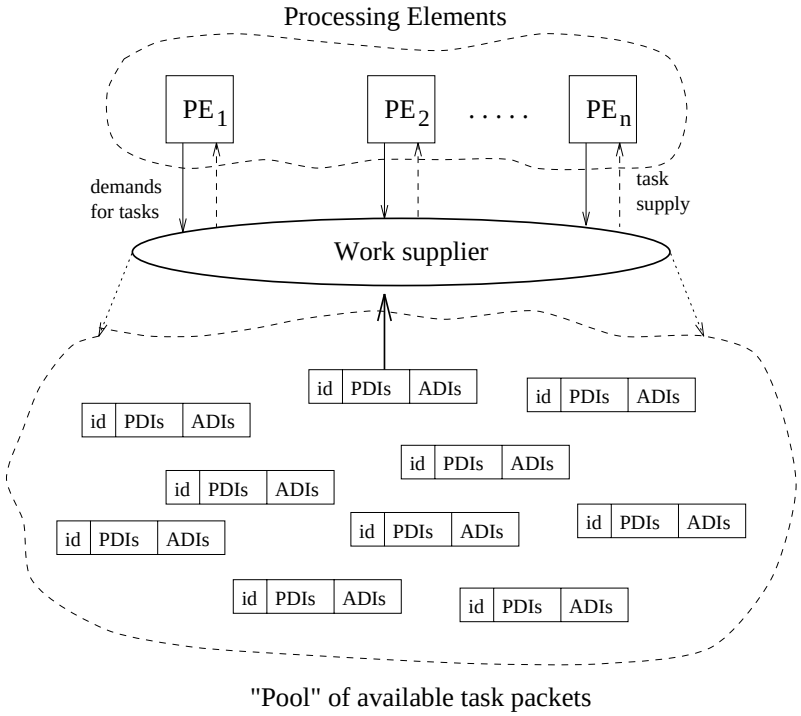


Figure 2.14: Supplying task packets from a task pool at the system controller

The advantage of a task pool is that the packets can be inserted into it in advance, or concurrently as the solution proceeds, according to the allocation strategy adopted. This is especially useful for problems that create work dynamically, such as those using the hybrid approach as described in section 2.2.3. Another advantage of the task pool is that if a hot spot in the problem domain is identified, then the ordering within the task pool can be changed dynamically to reflect this and thus ensure that potentially computationally complex tasks are allocated first.

More than one task pool may be used to reflect different levels of task priority. High priority tasks contained in the appropriate task pool will always be sent to a requesting processing element first. Only once this high priority task pool is (temporarily) empty will tasks from lower priority pools be sent. The multiple pool approach ensures that high priority tasks are not ignored as other tasks are allocated.

In the demand driven computational model, the processing elements demand the next task as soon as they have completed their current task. This demand is translated into sending a request to the work supplier, and the demand is only satisfied when the work supplier has delivered the next task. There is thus a definite delay period from the time the request is issued until the next task is received. During this period the processing element will be computationally idle. To avoid this idle time, it may be useful to include a buffer at each processing element capable of holding at least one task packet. This buffer may be considered as the processing element's own private task pool. Now, rather than waiting for a request to be satisfied from the remote system controller, the processing element may proceed with the computation on the task packet already present locally. When the remote request has been satisfied and a new task packet delivered, this can be stored in the buffer waiting for the processing element to complete the current task.

Whilst avoiding delays in fetching tasks from a remote task pool, the use of a buffer at each processing element may have serious implications for load balancing, especially towards the end of the problem solution. We will examine these issues in more detail after we have considered the realisation of task management for a simple demand driven system - the processor farm.

A first approach: The processor farm

Simple demand driven models of computation have been implemented and used for a wide range of applications. One realisation of such a model, often referred to in the literature, is that implemented by May and Shepherd [47]. This simple demand driven model, which they term a *processor farm*, has been used for solving problems with high computation to communication ratios. The model proposes a single system controller and one or more processing elements connected in a linear configuration, or chain. The structure of a processing element in this model is shown in figure 2.15.

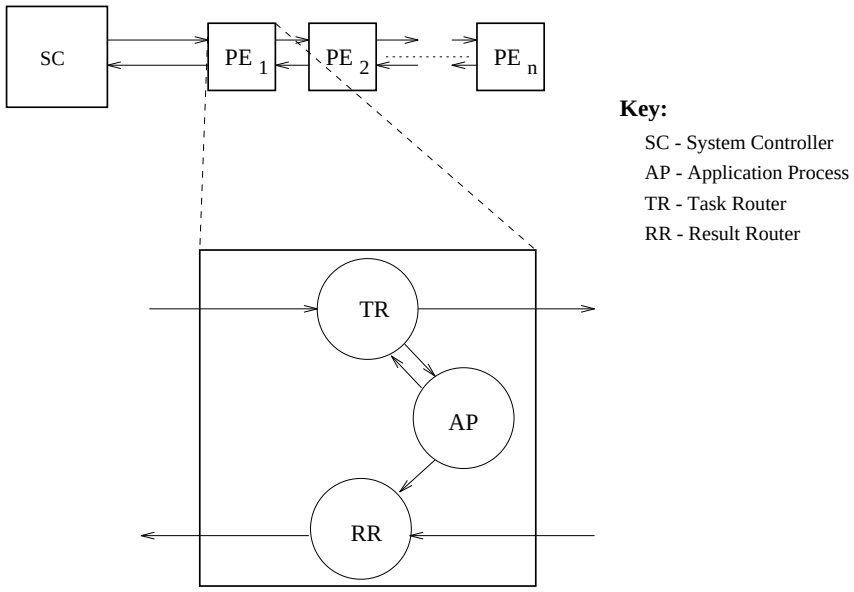


Figure 2.15: A processing element for the processor farm model

The application process performs the desired computation, while the communication within the system is dealt with by two router processes, the Task Router (TR) and the Result Router (RR). As their names suggest, the task router is responsible for distributing the tasks to the application process, while the result router returns the results from the completed tasks back to the system controller. The system controller contains the initial pool of tasks to be performed and collates the results. Such a communication strategy is simple to implement and largely problem independent.

To reduce possible processing element idle time, each task router process contains a single buffer in which to store a task so that a new task can be passed to the application process as soon as it becomes idle. When a task has been completed the results are sent to the system controller. On receipt of a result, the system controller releases a new task into the system. This synchronised releasing of tasks ensures that there are never more tasks in the system than there is space available.

On receipt of a new task, the task router process either:

1. passes the task directly to the application process if it is waiting for a task; or
2. places the task into its buffer if the buffer is empty; or, otherwise
3. passes the task onto the next processing element in the chain.

The processor farm is initialised by loading sufficient tasks into the system so that the buffer at each task router is full and each application process has a task with which to commence processing. Figure 2.16 shows the manner in which task requests are satisfied within a simple two processing element configured in a chain.

The simplicity of this realisation of a demand driven model has contributed largely to its popularity. Note that because of the balance maintained within the system, the only instance at which the last processing element is different from any other processing element in the chain is to ensure the `closedown_command`

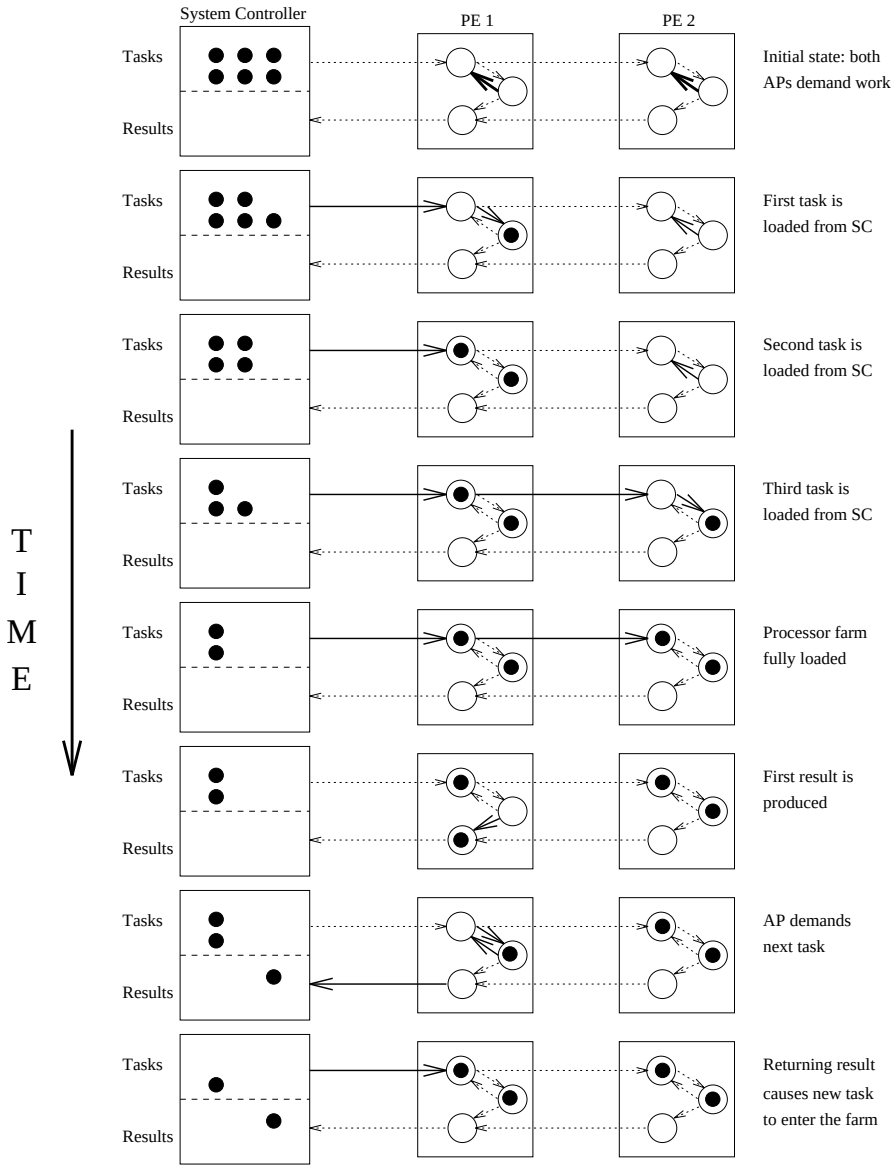


Figure 2.16: Task movement within a two PE processor farm

does not get passed any further. However, such a model does have disadvantages which may limit its use for more complex problems.

The computation to communication ratio of the desired application is critical in order to ensure an adequate performance of a processor farm. If this ratio is too low then significant processing element idle time will occur. This idle time occurs because the computation time for the application process to complete its current task and the task buffered at the task router may be lower than the combined communication time required for the results to reach the system controller plus the time for the new tasks released into the system to reach the processing element. This problem may be partially alleviated by the inclusion of several buffers at each task router instead of just one. However, without *a priori* knowledge as to the computation to communication ratio of the application, it may be impossible to determine precisely what the optimum number of buffers should be. This analysis is particularly difficult if the computational complexity of the tasks vary; precisely the type of problem demand driven models are more apt at solving. The problem independence of the system will also be compromised by the use of any *a priori* knowledge.

If the number of buffers chosen is too small, then the possibility of application process idle time will not be avoided. Provision of too many buffers will certainly remove any immediate application process idle time, but will re-introduce the predicament as the processing draws to a close. This occurs once the system controller has no further tasks to introduce into the system and now processing must only continue until all tasks still buffered at the processing elements have been completed. Obviously, significant idle time may occur as some processing elements struggle to complete their large number of buffered tasks.

The computation to communication ratio of the processor farm is severely exacerbated by the choice of the chain topology. The distance between the furthest processing element in the chain and the system controller grows linearly as more processing elements are added. This means that the combined communication time to return a result and receive a new task also increases. Furthermore, this communication time will also be adversely affect by the message traffic of all the intermediate processing elements which are closer to the system controller.

2.4.3 Task manager process

The aim of task management within a parallel system is to ensure the efficient supply of tasks to the processing elements. A Task Manager process (TM) is introduced at each processing element to assist in maintaining a continuous supply of tasks to the application process. The application process no longer deals with task requests directly, but rather indirectly using the facilities of the task manager. The task manager process assumes the responsibility for ensuring that every request for additional tasks from the application process will be satisfied immediately. The task manager attempts to achieve this by maintaining a local task pool.

In the processor farm, the task router process contains a single buffered task in order to satisfy the next local task request. As long as this buffer is full, task supply is immediate as far as the application process is concerned. The buffer is refilled by a new task from the system controller triggered on receipt of a result. The task router acts in a *passive* manner, awaiting replenishment by a new task within the farm. However, if the buffer is empty when the application process requests a task then this process must remain idle until a new task arrives. This idle time is wasted computation time and so to improve system performance the passive task router should be replaced by a “intelligent” task manager process more capable of ensuring new tasks are always available locally.

The task management strategies implemented by the task manager and outlined in the following sections are *active*, dynamically requesting and acquiring tasks during computation. The task manager thus assumes the responsibility of ensuring local availability of tasks. This means that an application process should *always* have its request for a task satisfied immediately by the task manager unless:

- at the start of the problem the application processes make a request before the initial tasks have been provided by the system controller;
- there are no more tasks which need to be solved for a particular stage of the parallel implementation; or,
- the task manager’s replenishment strategy has failed in some way.

A local task pool

To avoid any processing element idle time, it is essential that the task manager has at least one task available locally at the moment the application process issues a task request. This desirable situation was achieved in the processor farm by the provision of a single buffer at each task router. As we saw, the single buffer approach is vulnerable to the computation to communication ratio within the system. Adding more buffers to the task router led to the possibility of serious load imbalances towards the end of the computation.

The task manager process maintains a local task pool of tasks awaiting computation by the application process. This pool is similar to the task pool at the system controller, as shown in figure 2.14. However, not only will this local pool be much smaller than the system controller's task pool, but also it may be desirable to introduce some form of "status" to the number of available tasks at any point in time.

Satisfying a task request will free some space in the local task pool. A simple replenishment strategy would be for the task manager immediately to request a new task packet from the system controller. This request has obvious communication implications for the system. If the current message densities within the system are high and as long as there are still tasks available in the local task pool, this request will place and unnecessary additional burden on the already overloaded communication network.

As an active process, it is quite possible for the task manager to delay its replenishment request until message densities have diminished. However, this delay must not be so large that subsequent application process demands will deplete the local task pool before any new tasks can be fetched causing processor idle time to occur. There are a number of indicators which the task manager can use to determine a suitable delay. Firstly, this delay is only necessary if current message densities are high. Such information should be available for the router. Given a need for delay, the number of tasks in the task pool, the approximate computation time each of these tasks requires, and the probable communication latency in replenishing the tasks should all contribute to determining the request delay.

In a demand driven system, the computational complexity variations of the tasks are not known. However, the task manager will be aware of how long previous tasks have taken to compute (the time between application process requests). Assuming some form of preferred biased allocation of tasks in which tasks from similar regions of the problem domain are allocated to the same processing element, as discussed in section 2.4.5, the task manager will be able to build up a profile of task completion time which can be used to predict approximate completion times for tasks in the task pool. The times required to satisfy previous replenishment requests will provide the task manager with an idea of likely future communication responses. These values are, of course, mere approximations, but they can be used to assist in determining reasonable tolerance levels for the issuing of replenishment requests.

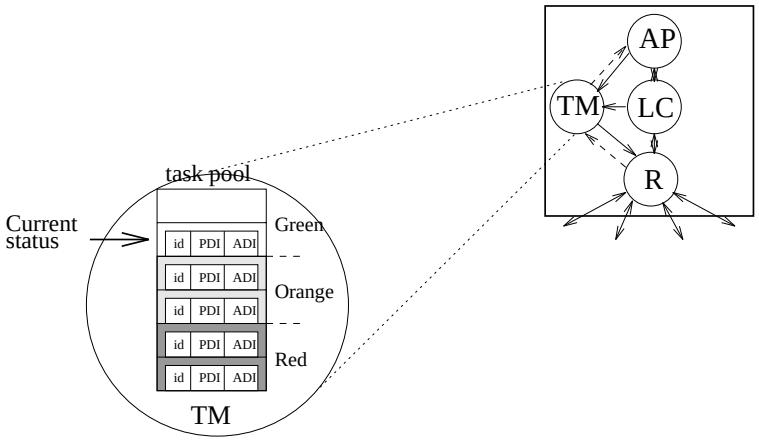


Figure 2.17: Status of task manager's task pool

The task manager's task pool is divided into three regions: *green*, *orange* and *red*. The number of tasks available in the pool will determine the current status level, as shown in figure 2.17. When faced with the need to replenish the task pool the decision can be taken based on the current status of the pool:

green: Only issue the replenishment request if current message traffic density is low;

orange: Issue the replenishment request unless the message density is very high; and,

red: Always issue the replenishment request.

The boundaries of these regions may be altered dynamically as the task manager acquires more information. At the start of the computation the task pool will be all red. The computation to communication ratio is critical in determining the boundaries of the regions of the task pool. The better this ratio, that is when computation times are high relative to the time taken to replenish a task packet, the smaller the red region of the task pool need be. This will provide the task manager with greater flexibility and the opportunity to contribute to minimising communication densities.

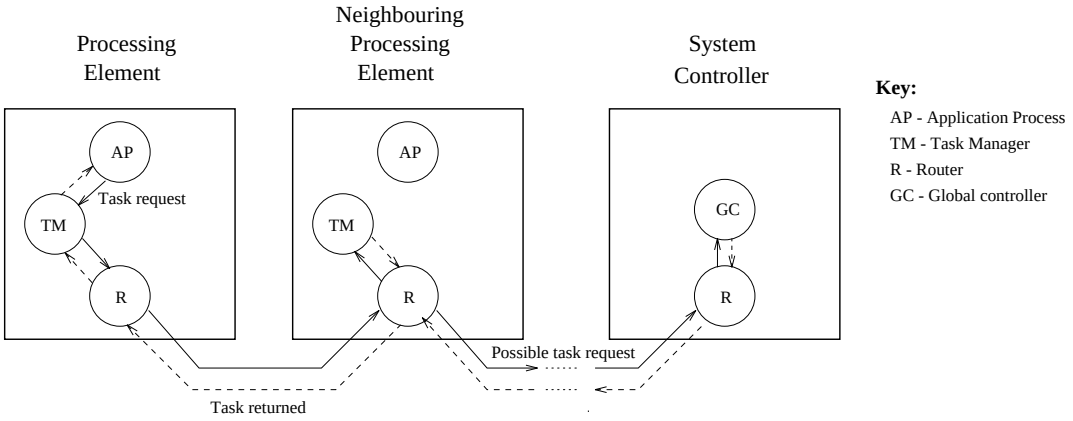


Figure 2.18: Task request propagating towards the system controller

2.4.4 Distributed task management

One handicap of the centralised task pool system is that all replenishment task requests from the task managers must reach the system controller before the new tasks can be allocated. The associated communication delay in satisfying these requests can be significant. The communication problems can be exacerbated by the bottleneck arising near the system controller. Distributed task management allows task requests to be handled at a number of locations remote from the system controller. Although all the tasks originate from the system controller, requests from processing elements no longer have to reach there in order to be satisfied.

The closest location for a task manager to replenish a task packet is from the task pool located at the task manager of one of its nearest neighbours. In this case, a replenishment request no longer proceeds directly to the system controller, but simply via the appropriate routers to the neighbouring task manager. If this neighbouring task manager is able to satisfy the replenishment request then it does so from its task pool. This task manager may now decide to in turn replenish its task pool, depending on its current status and so it will also request another task from one of its neighbouring task managers, but obviously not the same neighbour to which it has just supplied the task. One sensible strategy is to propagate these requests in a “chain like” fashion in the direction towards the main task supplier at the system controller, as shown in figure 2.18.

This distributed task management strategy is referred to as a *producer-consumer* model. The application process is the initial consumer and its local task manager the producer. If a replenishment request is issued then this task manager becomes the consumer and the neighbouring task manager the producer, and so on. The task supplier process of the system controller is the overall producer for the system. If no further tasks exist at the system controller then the last requesting task manager may change the direction of the search. This situation may occur towards the end of a stage of processing and facilitates load balancing of any tasks remaining in task manager buffers. As well as reducing the communication distances for task replenishment, an additional advantage of this “chain reaction” strategy is that the number of request

messages in the system is reduced. This will play a major rôle helping maintain a lower overall message density within the system.

If a task manager is unable to satisfy a replenishment request as its task pool is empty, then to avoid "starvation" at the requesting processing element, this task manager must ensure that the request is passed on to another processing element.

A number of variants of the producer-consumer model are also possible:

- Instead of following a path towards the system controller, the "chain reaction" could follow a predetermined Hamiltonian path (the system controller could be one of the processors on this path).

  Aside: A Hamiltonian path is a circuit starting and finishing at one processing element. This circuit passes through each processor in the network once only.

  Such a path would ensure that a processing element would be assured of replenishing a task if there was one available and there would be no need to keep track of the progress of the "chain reaction" to ensure no task manager was queried more than once per chain.

- In the course of its through-routing activities a router may handle a task packet destined for a distant task manager. If that router's local task manager has an outstanding "red request" for a task then it is possible for the router to *poach* the "en route task" by diverting it, so satisfying its local task manager immediately. Care must be taken to ensure that the task manager for whom the task was intended is informed that the task has been poached, so it may issue another request. In general, tasks should only be poached from "red replenishment" if to do so would avoid local application process idle time.

2.4.5 Preferred bias task allocation

The preferred bias method of task management is a way of allocating tasks to processing elements which combines the simplicity of the balanced data driven model with the flexibility of the demand driven approach. To reiterate the difference in these two computational models as they pertain to task management:

- Tasks are allocated to processing elements in a predetermined manner in the balanced data driven approach.

- In the demand driven model, tasks are allocated to processing elements on demand. The requesting processing element will be assigned the next available task packet from the task pool, and thus no processing element is bound to any area of the problem domain.

Provided no data dependencies exist, the order of task completion is unimportant. Once all tasks have been computed, the problem is solved. In the preferred bias method the problem domain is divided into equal regions with each region being assigned to a particular processing element, as is done in the balanced data driven approach. However, in this method, these regions are purely *conceptual* in nature. A demand driven model of computation is still used, but the tasks are not now allocated in an arbitrary fashion to the processing elements. Rather, a task is dispatched to a processing element from its conceptual portion. Once all tasks from a processing element's conceptual portion have been completed, only then will that processing element be allocated its next task from the portion of another processing element which has yet to complete its conceptual portion of tasks. Generally this task should be allocated from the portion of the processing element that has completed the least number of tasks. So, for example, from figure 2.19, on completion of the tasks in its own conceptual region, PE_3 may get allocated task number 22 from PE_2's conceptual region. Preferred bias allocation is sometimes also referred to as *conceptual task allocation*.

The implications of preferred bias allocation are substantial. The demand driven model's ability to deal with variations in computational complexity is retained, but now the system controller and the processing elements themselves know to whom a task that they have been allocated conceptually belongs. As we will see in section 3.6, this can greatly facilitate the even distribution of partial results at the end of any stage of a multi-stage problem.

The exploitation of data coherence is a vital ploy in reducing idle time due to remote data fetches. Preferred bias allocation of tasks can ensure that tasks from the same region of the problem are allocated to the same processing element. This can greatly improve the cache hit ratio at that processing element.

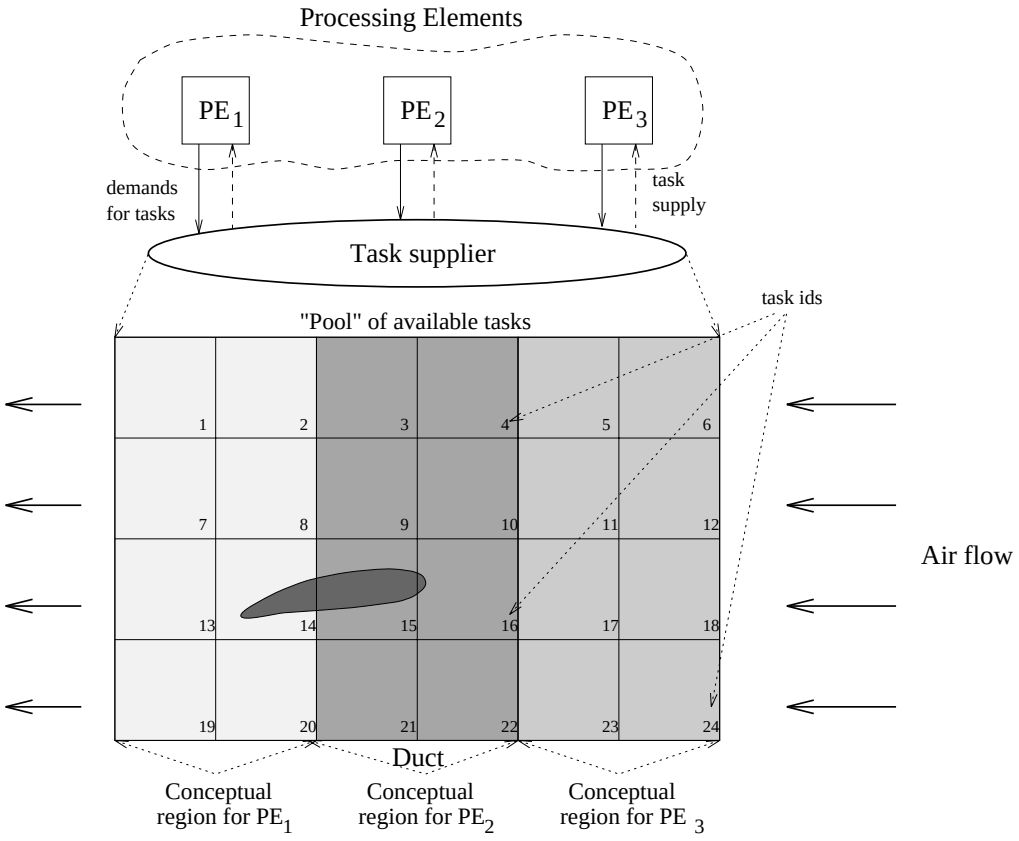


Figure 2.19: Partial result storage balancing by means of conceptual regions

Chapter 3

Data Management

The data requirements of many problems may be far larger than can be accommodated at any individual processing element. Rather than restricting ourselves to only solving those problems that fit completely within every processing element's local memory, we can make use of the combined memory of all processing elements. The large problem domain can now be distributed across the system and even secondary storage devices if necessary. For this class of application some form of data management will be necessary to ensure that data items are available at the processing elements when required by the computations.

Virtual shared memory regards the whole problem domain as a single unit in which the data items may be individually referenced. This is precisely how the domain could be treated if the problem was implemented on a shared memory multiprocessor system. However, on a distributed memory system, the problem domain is distributed across the system and hence the term **virtual**. Virtual shared memory systems may be implemented at different levels, such as in hardware or at the operating system level. In this chapter we will see how the introduction of a data manager process at each processing element can provide an elegant virtual shared memory at the system software level of our parallel implementation.

3.1 World Model of the Data: No Data Management Required

Not all problems possess very large data domains. If the size of the domain is such that it may be accommodated at every processing element then we say that the processing elements have a "world model" of the data. A world model may also exist if all the tasks allocated to a processing element only ever require a *subset* of the problem domain and this subset can be accommodated completely. In the world model, all principal and additional data items required by an application process will always be available locally at each processing element and thus there is no need for any data item to be fetched from another remote location within the system. If there is no requirement to fetch data items from remote locations as the solution of the problem proceeds then there is no need for any form of data management.

The processor farm described in section 2.4.2 is an example of a parallel implementation which assumes a world model. In this approach, tasks are allocated to processing elements in an arbitrary fashion and thus there is no restriction on which tasks may be computed by which processing element. No provision is made for data management and thus to perform any task, the entire domain must reside at each processing element.

Data items do not always have to be present at the processing element from the start of computation to avoid any form of data management. As discussed in section 2.3.1, both principal and additional data items may be included within a task packet. Provided no further data items are required to complete the tasks specified in the task packet then no data management is required and this situation may also be said to be demonstrating a world data model.

3.2 Virtual Shared Memory

Virtual shared memory provides all processors with the concept of a single memory space. Unlike a traditional shared memory model, this physical memory is distributed amongst the processing elements. Thus, a virtual shared memory environment can be thought of providing each processing element with a *virtual*

world model of the problem domain. So, as far as the application process is concerned, there is no difference between requesting a data item that happens to be local, or remote; only the speed of access can be (very) different.

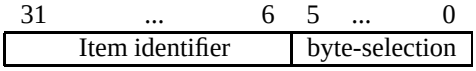
Virtual shared memory can be implemented at any level in the computer hierarchy. Implementations at the hardware level provide a transparent interface to software developers, but requires a specialised machine, such as the DASH system [43]. There have also been implementations at the operating system and compiler level. However, as we shall see, in the absence of dedicated hardware, virtual shared memory can also be easily provided at the system software level. At this level, a great deal of flexibility is available to provide specialised support to minimise any implementation penalties when undertaking the solution of problems with very large data requirements on multiprocessor systems. Figure 3.1 gives four levels at which virtual shared memory (VSM) can be supported, and examples of systems that implement VSM at that particular level.

Higher level	System Software	Provided by the Data Manager process
	Compiler	High Performance Fortran[36], ORCA[6]
	Operating System	Coherent Paging[45]
Lower level	Hardware	DDM [61], DASH [43], KSR-1 [38]

Figure 3.1: The levels where virtual shared memory can be implemented.

3.2.1 Implementing virtual shared memory

At the *hardware level* virtual shared memory intercepts all memory traffic from the processor, and decides which memory accesses are serviced locally, and which memory accesses need to go off-processor. This means that everything above the hardware level (machine code, operating system, etc.) sees a virtual shared memory with which it may interact in exactly the same manner as a physically shared memory. Providing this, so called, transparency to the higher levels, means that the size of data is not determined by the hardware level. However, in hardware, a data item becomes a fixed consecutive number of bytes, typically around 16-256. By choosing the size to be a power of 2, and by aligning data items in the memory, the physical memory address can become the concatenation of the "item-identifier" and the "byte selection". This strategy is easier to implement in hardware.



In this example, the most significant bits of a memory address locates the data item, and the lower bits address a byte within the item. The choice of using 6 bits as the byte selection in this example is arbitrary.

If a data structure of some higher level language containing two integers of four bytes each happened to be allocated from, say, address ...1100 111100 to ...1101 000100, then item ...1100 will contain the first integer, and item ...1101 will contain the other one. This means that two logically related integers of data are located in two physically separate items (although they could fit in a single data item).

Considered another way, if two unrelated variables, say `x` and `y` are allocated at addresses ...1100 110000 and ...1100 110100, then they reside in the same data item. If they are heavily used on separate processors, this can cause inefficiencies when the machine tries to maintain sequentially consistent copies of `x` and `y` on both processors. The machine cannot put `x` on one processor and `y` on the other, because it does not recognise `x` and `y` as different entities; the machine observes it as a single item that is shared between two processors. If sequential consistency has to be maintained the machine must update every write to `x` and `y` on both processors, even though the variables are not shared at all. This phenomenon is known as *false sharing*.

Virtual shared memory implemented at the *operating system level* also use a fixed size for data items, but these are typically much larger than at the hardware level. By making an item as large as a page of the operating system (around 1-4 KByte), data can be managed at the page level. This is cheaper, but slower than a hardware implementation.

When the *compiler* supports virtual shared memory, a data item can be made exactly as large as any user data structure. In contrast with virtual shared memory implementations at the hardware or operating

system level, compiler based implementations can keep logically connected variables together and distribute others. The detection of logically related variables is in the general case very hard, which means that applications written in existing languages such as C, Modula-2 or Fortran cannot be compiled in this way. However, compilers for specially designed languages can provide some assistance. For example, in High Performance Fortran the programmer indicates how arrays should be divided and then the compiler provides the appropriate commands to support data transport and data consistency.

Implementing virtual shared memory at the *system software* level provides the greatest flexibility to the programmer. However, this requires explicit development of system features to support the manipulation of the distributed data item. A *data manager* process is introduced at each processing element especially to undertake this job.

3.3 The Data Manager

Virtual shared memory is provided at the system software level by a data manager process at each processing element. The aim of data management within the parallel system is to ensure the efficient supply of data items to the processing elements. The data manager process manages data items just as the task manager was responsible for maintaining a continuous supply of tasks. Note that the data items being referred to here are the principal and additional data items as specified by the problem domain and not every variable or constant the application process may invoke for the completion of a task.

The application process now no longer deals with the principal and additional data items directly, but rather indirectly using the facilities of the data manager. The application process achieves this by issuing a data request to the data manager process every time a data item is required. The data manager process assumes the responsibility for ensuring that every request for a data item from the application process will be satisfied. The data manager attempts to satisfy these requests by maintaining a local data cache.

The data management strategies implemented by the data manager and outlined in the following sections are *active*, dynamically requesting and acquiring data items during computation. This means that an application process should *always* have its request for a data item satisfied immediately by the data manager unless:

- at the start of the problem the application processes make requests before any initial data items have been provided by the system controller;
- the data manager's data fetch strategy has failed in some way.

3.3.1 The local data cache

The concept of *data sharing* may be used to cope with very large data requirements [14, 23]. Data sharing implements virtual shared memory by allocating every data item in the problem domain an unique identifier. This allows a required item to be "located" from somewhere within the system, or from secondary storage if necessary. The size of problem that can now be tackled is, therefore, no longer dictated by the size of the local memory at each processing element, but rather only by the limitations of the combined memory plus the secondary storage.

The principal data item required by an application process is specified by the task it is currently performing. Any additional data item requirements are determined by the task *and* by the algorithm chosen to solve the problem. These additional data items may be known *a priori* by the nature of the problem, or they may only become apparent as the computation of the task proceeds.

To avoid any processing element idle time, it is essential that the data manager has the required data item available locally at the moment the application process issues a request for it. In an attempt to achieve this, the data manager maintains a local cache of data items as shown in figure 3.2. The size of this cache, and thus the number of data items it can contain, is determined by the size of a processing element's local memory.

Each data item in the system is a packet containing the unique identifier, shown in figure 3.2 as `id`, together with the actual data which makes up the item. The data items may be permanently located at a specific processing element, or they may be free to migrate within the system to where they are required. When a data manager requires a particular data item which is not already available locally, this data item must be fetched from some remote location and placed into the local cache. This must occur before the application

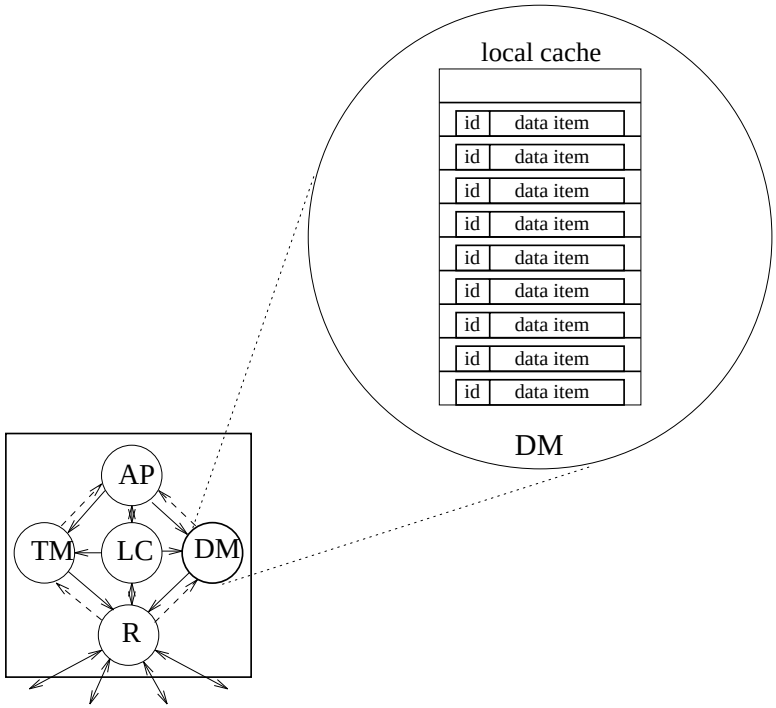


Figure 3.2: The local cache at the data manager

process can access the data item. The virtual shared memory of the system is thus the combination of the local caches at all the processing elements plus the secondary storage which is under the control of the file manager at the system controller.

In certain circumstances, as will be seen in the following sections, rather than removing the data item from the local cache in which it was found, it may be sufficient simply to take a copy of the data item and return this to the local cache. This is certainly the case when the data items within the problem domain are *read-only*, that is the values of the data items are not altered during the course of the parallel solution of the problem (and indeed the same would be true of the sequential implementation). This means that it is possible for copies of the same data item to be present in a number of local caches. Note that it is no advantage to have more than one copy of any data item in one local cache.

There is a limited amount of space in any local cache. When the cache is full and another data item is acquired from a remote location, then one of the existing data items in the local cache must be replaced by this new data item. Care must be taken to ensure that no data item is inadvertently completely removed from the system by being replaced in all local caches. If this does happen then, assuming the data item is read-only, a copy of the entire problem domain will reside on secondary storage, from where the data items were initially loaded into the local caches of the parallel system. This means that should a data item being destroyed within the system, another copy can be retrieved from the file manager (FM) of the system controller.

If the data items are *read-write* then their values may be altered as the computation progresses. In this case, the data managers have to beware of consistency issues when procuring a data item. The implications of consistency will be discussed in section 3.4.

As we will now see, the strategies adopted in the parallel implementation for acquiring data items and storing them in the local caches can have a significant effect on minimising the implementation penalties and thus improving overall system performance. The onus is on the data manager process to ensure these strategies are carried out efficiently.

3.3.2 Requesting data items

The algorithm being executed at the application process will determine the next data item required. If the data items were all held by the application process, requesting the data item would be implemented within the application process as an "assignment statement". For example a request for data item `i` would simply be written as `x := data_item[i]`. When all the data items are held instead by the data manager process, this "assignment statement" must be replaced by a request from the application process to the data manager for the data item followed by the sending of a copy of the data item from the local cache of the data manager to the waiting application process, as shown in figure 3.3.

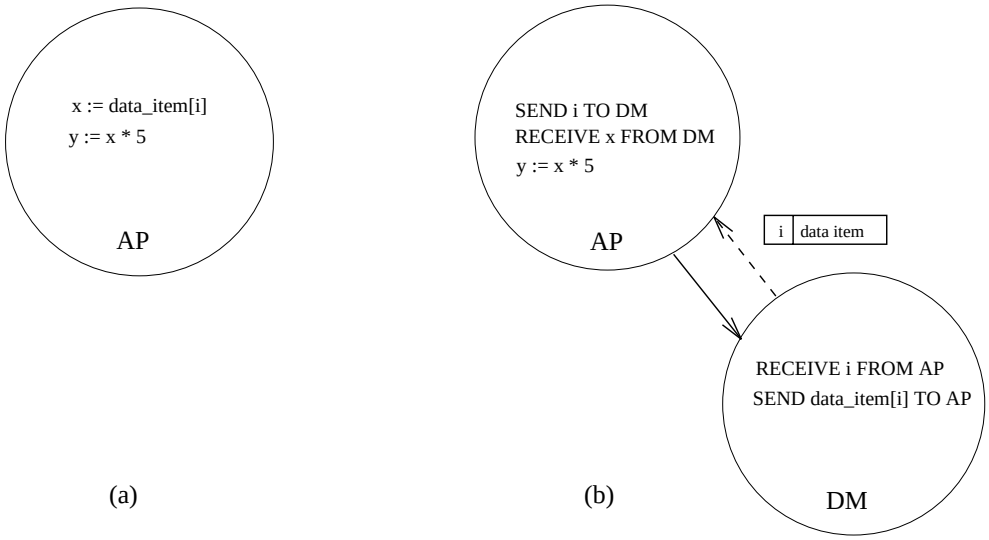


Figure 3.3: Accessing a data item (a) with, and (b) without a data manager

The data item's unique identifier enables the data manager to extract the appropriate item from its local cache. If a data item requested by the application process is available, it is immediately transferred, as shown in figure 3.4(a). The only slight delay in the computation of the application process will occur by the need to schedule the concurrent data manager and for this process to send the data item from its local cache. However, if the data item is not available locally then the data manager must "locate" this item from elsewhere in the system. This will entail sending a message via the router to find the data item in another processing element's local cache, or from the file manager of the system controller. Having been found, the appropriate item is returned to the requesting data manager's own local cache and then finally a copy of the item is transferred to the application process.

If the communicated request from the application process is asynchronous and this process is able to continue with its task while awaiting the data item then no idle time occurs. However, if the communication with the data manager is synchronous, or if the data item is essential for the continuation of the task then idle time will persist until the data item can be fetched from the remote location and a copy given to the application process, as shown in figure 3.4(b). Unless otherwise stated, we will assume for the rest of this chapter that an application process is unable to continue with its current task until its data item request has been satisfied by the data manager.

3.3.3 Locating data items

When confronted with having to acquire a remote data item, two possibilities exist for the data manager. Either it knows exactly the location of the data item within the system, or this location is unknown and some form of search will have to be instigated.

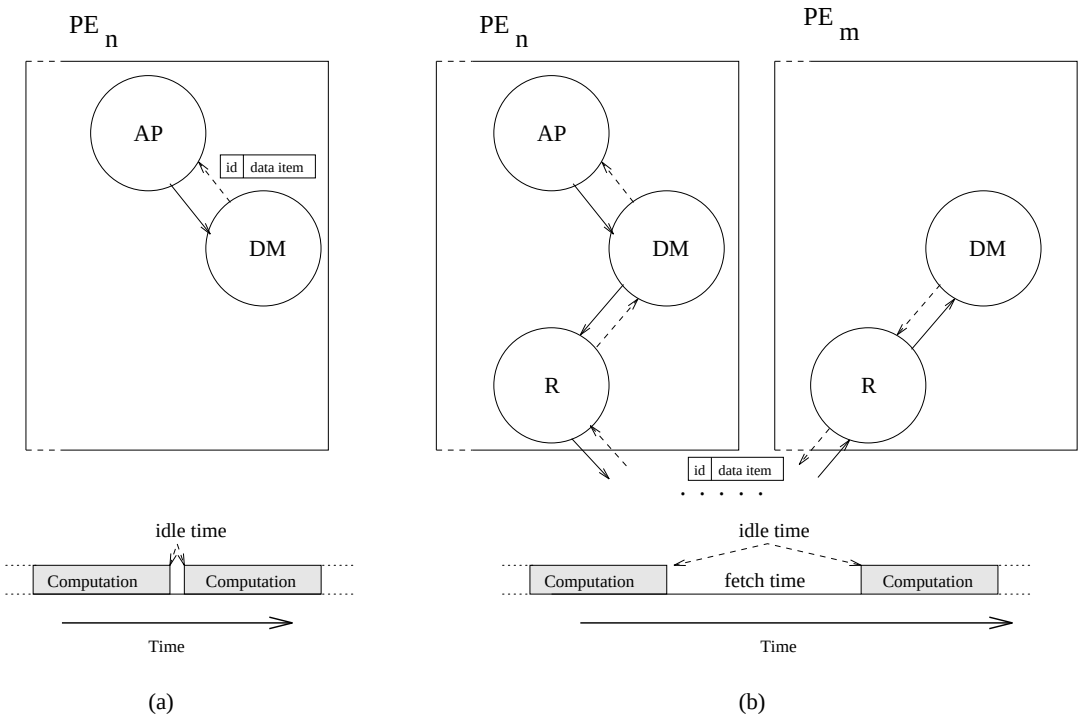


Figure 3.4: AP idle time due to: (a) Data item found locally (b) Remote data item fetch

Resident sets

Knowing the precise location of the requested data item within the system enables the data manager to instruct the router to send the request for the data item, directly to the appropriate processing element.

One of the simplest strategies for allocating data items to each processing element's local cache is to divide all the data items of the problem domain evenly amongst the processing elements before the computation commences. Providing there is sufficient local memory and assuming there are n processing elements, this means that each processing element would be allocated $\frac{1}{n}^{th}$ of the total number of data items. If there isn't enough memory at each processing element for even this fraction of the total problem domain then as many as possible could be allocated to the local caches and the remainder of the data items would be held at the file manager of the system controller. Such a simplistic scheme has its advantages. Provided these data items remain at their predetermined local cache for the duration of the computation, then the processing element from which any data item may found can be computed directly from the identity of the data item.

For example, assume there are twelve data items, given the unique identification numbers $1, \ldots, 12$, and three processing elements, PE_1, PE_2, and PE_3. A predetermined allocation strategy may allocate permanently data items $1, \ldots, 4$ to PE_1, data items $5, \ldots, 8$ to PE_2 and $9, \ldots, 12$ to PE_3. Should PE_2 wish to acquire a copy of data item 10, it may do so directly from the processing element known to have that data item, in this case PE_3.

It is essential for this simple predetermined allocation strategy that the data items are not overwritten or moved from the local cache to which they are assigned initially. However, it may be necessary for a processing element to also acquire copies of other data items as the computation proceeds, as we saw with PE_2 above. This implies that the local cache should be partitioned into two distinct regions:

- a region containing data items which may never be replaced, known as the *resident set*; and,
- a region for data items which may be replaced during the parallel computation.

The size of the resident set should be sufficient to accommodate all the pre-allocated data items, as

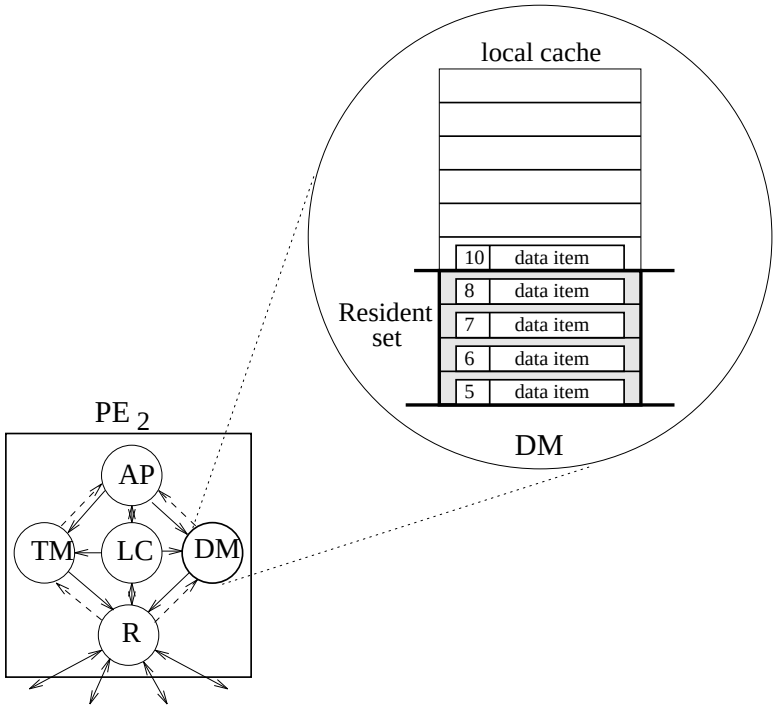


Figure 3.5: Resident set of the local cache

shown for PE_2 from the above example in figure 3.5. The remaining portion of the local cache will be as large as allowed by the local memory of the processing element. Note that this portion needs to have sufficient space to hold a minimum of *one* data item as this is the maximum that the application process can require at any specific point during a task's computation. To complete a task an application process may require many data items. Each of these data items may in turn replace the previously acquired one in the single available space in the local cache.

The balanced data driven model of computation is well suited to a simple pre-determined even data item allocation scheme. In this model the system controller knows prior to the computation commencing precisely which tasks are to be assigned to which processing elements. The same number of tasks is assigned to each processing element and thus the principal data items for each of these tasks may be pre-allocated evenly amongst the local caches of the appropriate processing elements. Similar knowledge is available to the system controller for the unbalanced data driven model, but in this case the number of tasks allocated to each processing element is not the same and so different numbers of principal data items will be loaded into each resident set. Note that the algorithm used to solve the problem may be such that, even if a data driven model is used and thus the principal data items are known in advance, the additional data items may not be known *a priori*. In this case, these additional data items will have to be fetched into the local caches by the data managers as the computation proceeds and the data requirements become known.

More sophisticated pre-allocation strategies, for example some form of hashing function, are possible to provide resident sets at each processing element. It is also not necessary for each data item to be resident at only one processing element. Should space permit, the same data item may be resident at several local caches.

The pre-allocation of resident sets allows the location of a data item to be determined from its unique identifier. A pre-allocated resident set may occupy a significant portion of a local cache and leave little space for other data items which have not been pre-allocated. The shortage of space would require these other data items to be replaced constantly as the computation proceeds. It is quite possible that one data item may be needed often by the same application process either for the same task or for several tasks. If this data item is not in the resident set for that processing element, then there is the danger that the data item will be replaced during the periods that it is not required and thus will have to be re-fetched when it is

required once more. Furthermore, despite being pre-allocated, the data items of a resident set may in fact never be required by the processing element to which they were allocated. In the example given earlier, PE_2 has a resident set containing data items $5, \ldots, 8$. Unless there is *a priori* knowledge about the data requirements of the tasks, there is no guarantee that PE_2 will ever require any of these data items from its resident set. In this case, a portion of PE_2's valuable local cache is being used to store data items which are never required, thus reducing the available storage for data items which are needed. Those processing elements that do require data items $5, \ldots, 8$ are going to have to fetch them from PE_2. Not only will the fetches of these data items imply communication delays for the requesting data managers, but also, the need for PE_2's data manager to service these requests will imply concurrent activity by its data manager which will detract from the computation of the application process.

The solution to this dilemma is not to pre-allocate resident sets, but to build up such a set as computation proceeds and information is gained by each data manager as to the data items most frequently used by its processing element. Profiling can also assist in establishing these resident sets, as explained in section 3.5.3. The price to pay for this flexibility is that it may no longer be possible for a data manager to determine precisely where a particular data item may be found within the system.

Searching for data at unknown locations

Acquiring a specific data item from an unknown location will necessitate the data manager requesting the router process to "search" the system for this item. The naive approach would be for the router to send the request to the data manager process of each processing element in turn. If the requested data manager has the necessary data item it will return a copy and then there is no need for the router to request any further processing elements. If the requested data manager does not have the data item then it must send back a `not_found` message to the router, whereupon the next processing element may be tried. The advantages of this *one-to-one* scheme is that as soon as the required data item is found, no further requests need be issued and only one copy of the data item will ever be returned. However, the communication implications of such a scheme for a large parallel system are substantial. If by some quirk of fate (or Murphy's law), the last processing element to be asked is the one which has the necessary data item, then one request will have resulted in $2 \times (number\ of\ PEs\ - 1)$ messages, a quite unacceptable number for large systems. Furthermore, the delay before the data item is finally found will be large, resulting in long application process idle time.

An alternative to this communication intensive *one-to-one* approach, is for the router process to issue a global broadcast of the request; a *one-to-many* method. A bus used to connect the processing elements is particularly suited to such a communication strategy, although, as discussed in section 1.2.1, a bus is not an appropriate interconnection method for large multiprocessor systems. The broadcast strategy may also be used efficiently on a more suitable interconnection method for large systems, such as interconnections between individual processors. In this case, the router issues the request to its directly-connected neighbouring processing elements. If the data managers at these processing elements have the required data item then it is returned, if not then these neighbouring processing elements in turn propagate the request to their neighbours (excluding the one from which they received the message). In this way, the requests propagates through the system like ripples on a pond. The message density inherent in this approach is significantly less than the *one-to-one* approach, however one disadvantage is that if the requested data item is replicated at several local caches, then several copies of the same data item will be returned to the requesting data manager, when only one is required.

For very large multiprocessor systems, even this *one-to-many* approach to discovering the unknown location of a data item may be too costly in terms of communication latency and its contribution to message density within the system. A compromise of the direct access capabilities of the pre-allocated resident set approach and the flexibility of the dynamic composition of the local caches is the notion of a *directory* of data item locations.

In this approach, it is not necessary to maintain a particular data item at a fixed processing element. We can introduce the notion of a *home*-processing element that knows where that data item is, while the data item is currently located at the *owner*-processing element. The home-processing element is fixed and its address may be determined from the identifier of the data item. The home processing element knows which processing element is currently owning the data item. Should this data item be subsequently moved and the one at the owner-process removed, then either the home-processing element must be informed as to the new location of the data item or the previous owner-processing element must now maintain a pointer to this

new location. The first scheme has the advantage that a request message may be forwarded directly from the home-processing element to the current owner, while the second strategy may be necessary, at least for a while after the data item has been moved from an owner, to cope with any requests forwarded by the home-processing element before it has received the latest location update.

Finally, it is also possible to do away with the notion of a home-processing element, by adding a hierarchy of directories. Each directory on a processing element "knows" which data items are present on the processing element. If the required data item is not present, a directory higher up in the hierarchy might know if it is somewhere nearby. If that directory does not know, yet another directory might if it is further away. This is much like the organisation of libraries: you first check the local library for a book, if they do not have it you ask the central library, and so on until you finally query the national library. With this organisation there is always a directory that knows the whereabout of the data item, but it is very likely that the location of the data item will be found long before asking the highest directory. (The Data Diffusion Machine [61] and KSR-1 [38] used a similar strategy implemented in hardware).

3.4 Consistency

Copies of *read-only* data items may exist in numerous local caches within the system without any need to "keep track" of where all the copies are. However, if copies of *read-write* data items exist then, in a virtual shared memory system, there is the danger that the data items may become *inconsistent*. The example in figure 3.6 illustrates this problem of inconsistency. Suppose that we have two processing elements PE_1 and PE_2, and a data item y with a value 0, that is located at processing element PE_1. Processing Element PE_2 needs y, so it requests and gets a copy of y. The data manager on processing element PE_2 decides to keep this copy for possible future reference. When the application at processing element PE_1 updates the value of y, for example by overwriting it with the value 1, processing element PE_2 will have a *stale* copy of y. This situation is called *inconsistent*: if the application running at processing element PE_1 requests y it will get the new value (1), while the application at processing element PE_2 will still read the old value of y (0). This situation will exist until the data manager at processing element PE_2 decides to evict y from its local memory.

The programming model of a physical shared memory system maintains only one copy of any data item; the copy in the shared memory. Because there is only one copy, the data items cannot become inconsistent. Hence, naive *virtual* shared memory differs from *physical* shared memory in that virtual shared memory can become inconsistent.

To maintain consistency all copies of the data items will have to be "tracked down" at certain times during the parallel computation. Once again tone-to-one or many-to-one methods could be used to determine the unknown locations of copies of the data items. If the directory approach is used then it will be necessary to maintain a complete "linked list" through all copies of any data item, where each copy knows where the next copy is, or it knows that there are no more copies. A consistency operation is performed on this list by sending a message to the first copy on the list, which then ripples through the list. These operations thus take a time linear in the number of copies. This is expensive if there are many copies, so it can be more efficient to use a tree structure (where the operation needs logarithmic time). (A combination of a software and hardware tree directory of this form is used in the LimitLESS directory [12].)

There are several ways to deal with this inconsistency problem. We will discuss three options: data items are kept consistent at all times (known as sequential consistency); the actual problem somehow copes with the inconsistencies (known as weak consistency); and finally, inconsistent data items are allowed to live for a well defined period (the particular scheme discussed here is known as release consistency).

3.4.1 Keeping the data items consistent

The first option is that the data manager will keep the data items consistent at all times. To see how the data items can be kept consistent, observe first that there are two conditions that must be met before a data item can become inconsistent. Firstly, the data item must be duplicated; as long as there is only a single copy of the data item, it cannot be inconsistent. Secondly, some processing element must update one of the copies, without updating the other copies. This observation leads to two protocols that the data manager can observe to enforce consistency, while still allowing copies to be made:

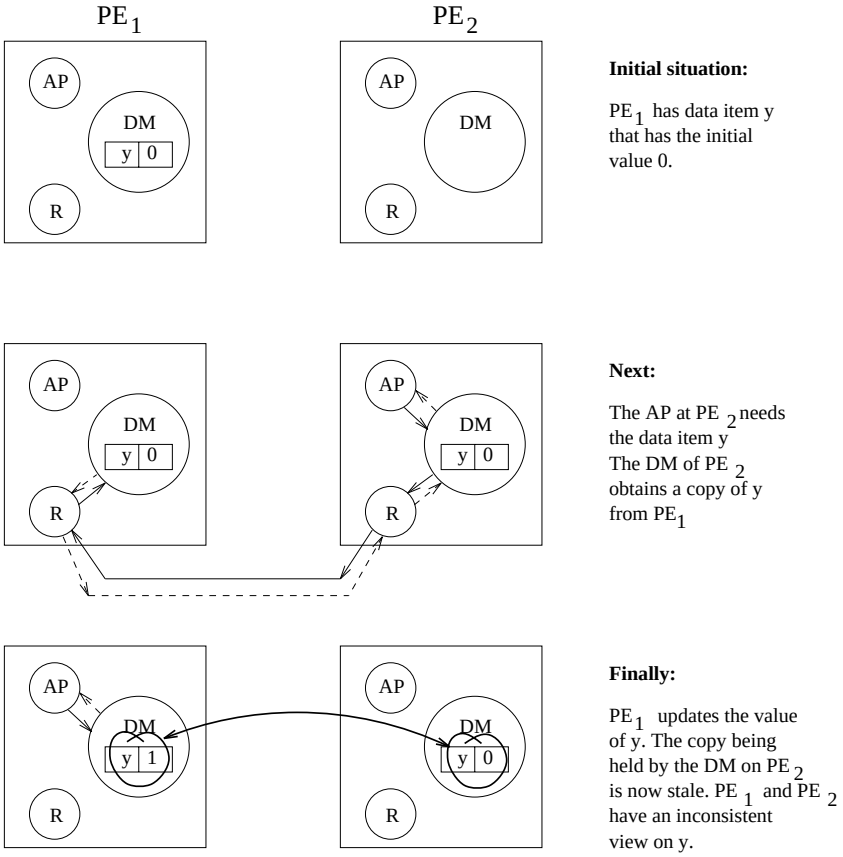


Figure 3.6: An example how an inconsistency arises. There are two processing elements, PE_1 and PE_2 and a data item y. PE_2 keeps a copy of y, while y is updated at PE_1.

1. Ensure that there is not more than a single copy of the data item when it is updated. This means that before a write all but one of the copies must be deleted. This solution is known as an *invalidating protocol*.

2. Ensure that all copies of the data item are replaced when it is updated. This solution is known as an *updating protocol*.

It is relatively straightforward to check that the invalidating option will always work: all copies are always identical, because a write only occurs when there is only a single copy. In the example, the copy of y at processing element PE_2 will be destroyed before y is updated on processing element PE_1.

For the updating protocol to be correct, the protocol must ensure that all copies are replaced "at the same time". Suppose that this is not the case: in the example the value on processing element PE_1 might be updated, while processing element PE_2 still has an old value for y. If the data managers running on processing elements PE_1 and PE_2 communicate, they can find out about this inconsistency. In order for the update protocol to work, the updating data manager must either ensure that no other data manager is accessing the data item while it is being updated, or that it is impossible for any communication (or other update) to overtake this update.

It is not easy to decide in general whether an invalidating or an updating protocol is better. Below are two examples that show that invalidating and updating protocols both have advantages and disadvantages. In both cases we assume that the problem is running on a large number of processing elements, and that there is a single shared data item that is initially replicated over all processing elements.

1. A task, or tasks, being performed by an application process at one processing element might require that the data item be updated at this data manager many times, without any of the other processing elements using it. An updating protocol will update all copies on all processing elements during every update, even though the copies are not being used on any of the other processing elements.

   An invalidating protocol is more efficient, because it will invalidate all outstanding copies once, whereupon the application process can continue updating the data item without extra communication.

2. Suppose that instead of ignoring the data item, all other processing elements do need the updated value. An invalidating protocol will invalidate all copies and update the data item, whereupon all other processing elements have to fetch the value again. This fetch is on demand, which means that they will have to wait on the data item.

   An updating protocol does a better job since it distributes the new value, avoiding the need for the other processing elements to wait for it.

There is a case for (and against) both protocols. It is for this reason that these two protocols are sometimes combined. This gives a protocol that, for example, invalidates all copies that have not been used since the last update, and updates the copies that were used since the last update. Although these hybrid protocols are potentially more efficient, they are unfortunately often more complex than a pure invalidating or updating protocol.

3.4.2 Weak consistency: repair consistency on request

The option to maintain sequential consistency is an expensive one. In general, an application process is allowed to proceed with its computation only after the invalidate or update has been completed. In the example of the invalidating protocol, all outstanding copies must have been erased and the local copy must have been updated before the application process can proceed. This idle time may be an unacceptable overhead. One of the ways to reduce this overhead is to forget about maintaining consistency automatically. Instead, the local cache will stay inconsistent until the application process orders the data manager to repair the inconsistency.

There are two important advantages of weak consistency. Firstly, the local cache is made consistent at certain points in the task execution only, reducing the overhead. Secondly, local caches can be made consistent in parallel. Recall for example, the updating protocol of the previous section. In a weakly consistent system we can envisage that every write to a data item is asynchronously broadcasted to all remote copies. Asynchronously means that the processing element performing the write continues whether the update has been completed or not. Only when a consistency-command is executed must the application process wait

until all outstanding updates are completed. In the same way, a weakly consistent invalidating protocol can invalidate remote copies in parallel. These optimisations lead to further performance improvement. The disadvantage of weak consistency is the need for the explicit commands within the algorithm at each application process so that when a task is being executed, at the appropriate point, the data manager can be instructed to make the local cache consistent.

3.4.3 Repair consistency on synchronisation: Release consistency

A weak consistency model as sketched above requires the programmer of the algorithm to ensure consistency at any moment in time. Release consistency is based on the observation that algorithms do not go from one phase to the other without first synchronising. So it suffices to make the local caches consistent during the synchronisation operation. This means that immediately after each synchronisation the local caches are guaranteed to be consistent. This is in general slightly more often than strictly necessary, but it is far less often than would be the case when using sequential consistency. More importantly, the application process itself does not have to make the local caches consistent anymore, it is done "invisibly".

Note that although invisible, consistency is only restored during an explicit synchronisation operation; release consistency behaves still very differently from sequential consistency. As an example, an application process at PE_1 can poll a data item in a loop, waiting for the data item to be changed by the application process at PE_2. Under sequential consistency any update to the data item will be propagated, and cause the application process at PE_1 to exit the loop. Under release consistency updates do not need to be propagated until a synchronisation point, and because it does not recognise that the polling loop is actually a synchronisation point the application process at PE_1 might be looping forever.

3.5 Minimising the Impact of Remote Data Requests

Failure to find a required data item locally means that the data manager has to acquire this data item from elsewhere within the system. The time to fetch this data item and, therefore, the application process idle time, can be significant. This *latency* is difficult to predict and may not be repeatable due to other factors, such as current message densities within the system. The overall aim of data management is to maximise effective processing element computation by minimising the occurrence and effects of remote data fetches. A number of techniques may be used to reduce this latency by:

Hiding the Latency: - overlapping the communication with the computation, by:

Prefetching - anticipating data items that will be required

Multi-threading - keeping the processing element busy with other useful computation during the remote fetch

Minimising the Latency: - reducing the time associated with a remote fetch by:

Caching & profiling - exploiting any coherence that may exist in the problem domain

3.5.1 Prefetching

If it is known at the start of the computation which data items will be required by each task then these data items can be *prefetched* by the data manager so that they are available locally when required. The data manager thus issues the requests for the data items *before* they are actually required and in this way overlaps the communication required for the remote fetches with the ongoing computation of the application process. This is in contrast with the simple fetch-upon-demand strategy where the data manager only issues the external request for a data item at the moment it is requested by the application process and it is not found in the local cache.

By treating its local cache as a "circular buffer" the data manager can be loading prefetched data items into one end of the buffer while the application process is requesting the data items from the other end, as shown in figure 3.7. The "speed" at which the data manager can prefetch the data items will be determined by the size of the local cache and the rate at which the application process is "using" the data items.

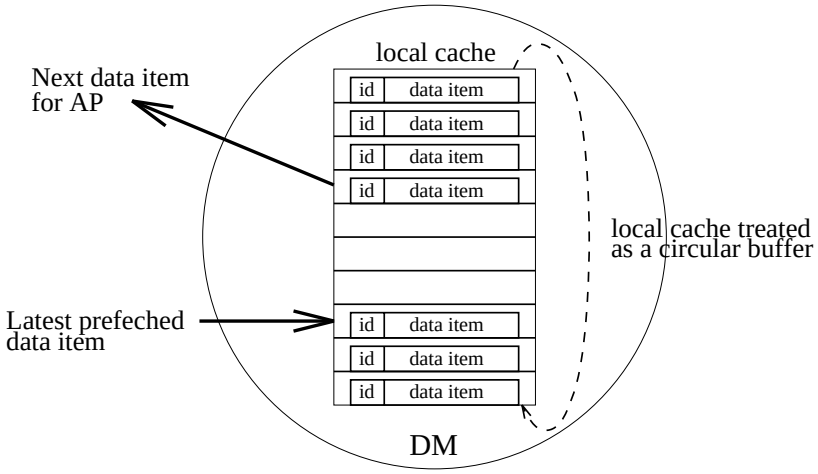


Figure 3.7: Storing the prefetched data items in the local cache

This knowledge about the data items may be known *a priori* by the nature of problem. For example, in a parallel solution of the hemi-cube radiosity method, the data manager knows that each task, that is the computation of a single row of the matrix of form factors, requires all the environment's patch data. The order in which these data items are considered is unimportant, as long as all data items are considered. The data manager can thus continually prefetch those data items which have yet to be considered by the current task. Note that in this problem, because all the data items are required by every task and the order is unimportant (we are assuming that the local cache is not sufficiently big to hold all these data items), those data items which remain in the local cache at the end of one task are also required by the subsequent task. Thus, at the start of the next task, the first data item in the local cache can be forwarded to the application process and prefetching can commence once more as soon as this has happened.

The choice of computation model adopted can also provide the information required by the data manager in order to prefetch. The principal data items for both the balanced and unbalanced data driven models will be known by the system controller before the computation commences. Giving this information to the data manager will enable it to prefetch these data items. A prefetch strategy can also be used for principal data items within the preferred bias task allocation strategy for the demand driven computation model, as described in section 2.4.5. Knowledge of its processing element's conceptual region can be exploited by the data manager to prefetch the principal data items within this region of the problem domain.

3.5.2 Multi-threading

Any failure by the data manager to have the requested data item available locally for the application process will result in idle time unless the processing element can be kept busy doing some other useful computation.

One possibility is for the application process to save the current state of a task and commence a new task whenever a requested data item is not available locally. When the requested data item is finally forthcoming either this new task could be suspended and the original task resumed, or processing of the new task could be continued until it is completed. This new task may be suspended awaiting a data fetch and so the original task may be resumed. Saving the state of a task may require a large amount of memory and indeed, several states may need to be saved before one requested data item finally arrives. Should the nature of the problem allow these stored tasks to in turn be considered as task packets, then this method has the additional advantage that these task packets could potentially be completed by another processing element in the course of load balancing, as explained in the section 2.4.4 on distributed task management.

Another possible option is multi-threading. In this method there is not only one, but several application processes on each processing element controlled by an application process controller (APC), as shown in figure 3.8. Each application process is known as a separate *thread* of computation. Now, although one thread may be suspended awaiting a remote data item, the other threads may still be able to continue. It

may not be feasible to determine just how many of these application processes will be necessary to avoid the case where all of them are suspended awaiting data. However, if there are sufficient *threads* (and of course sufficient tasks) then the processing element should always be performing useful computation. Note that multi-threading is similar to the Bulk Synchronous Parallel paradigm [60].

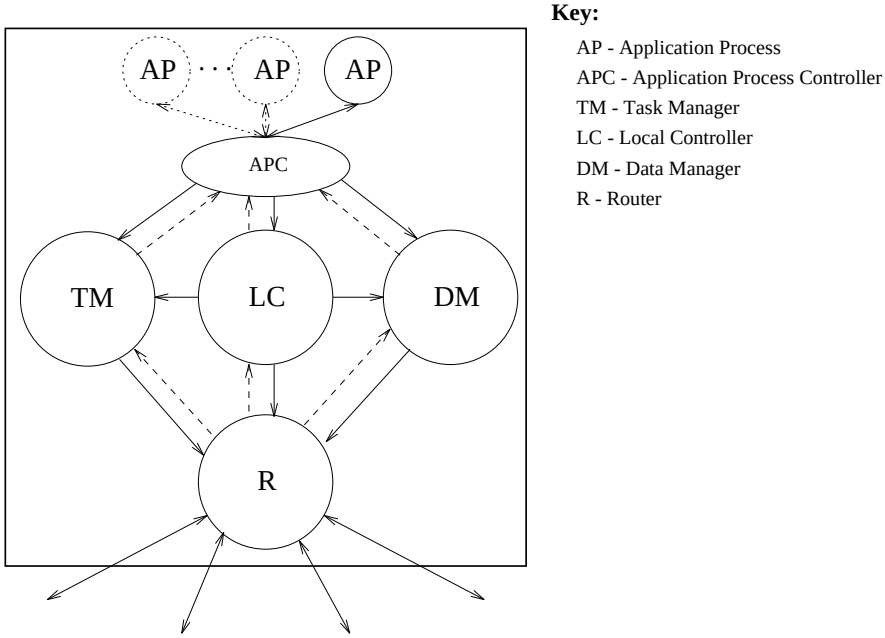


Figure 3.8: Several application processes per processing element

One disadvantage of this approach is the overhead incurred by the additional context switching between all the application processes and the application process controller, as well as the other system software processes: the router, task manager and the data manager, that are all resident on the same processor. A variation of multiple active threads is to have several application processes existing on each processing element, but to only have *one* of them active at any time and have the application process controller manage the scheduling of these processes explicitly from information provided by the data manager. When an application processes' data item request cannot be satisfied locally, that process will remain descheduled until the data item is forthcoming. The data manager is thus in a position to inform the application process controller to activate another application process, and only reactivate the original application process once the required data has been obtained. Note the application process controller schedules a new application process by sending it a task to perform. Having made its initial demand for a task to the application process controller (and not the task manager as discussed in section 2.4.2) an application process will remain descheduled until explicitly rescheduled by the application process controller.

Both forms of multi-threading have other limitations. The first of these is the extra memory requirements each thread places on the processing elements local memory. The more memory that each thread will require, for local constants and variables etc, the less memory there will be available for the local cache and thus fewer data items will be able to be kept locally by the data manager. A "catch 22" (or is that "cache 22") situation now arises as fewer local data items implies more remote data fetches and thus the possible need for yet more threads to hide this increase in latency. The second difficulty of a large number of threads running on the same processing element is the unacceptably heavy overhead that may be placed on the data manager when maintaining the local cache. For example, a dilemma may exist as to whether a recently fetched data item for one thread should be overwritten before it has been used if its "slot" in the local cache is required by the currently active thread.

Figure 3.9 shows results for a multi-threaded application. The graph shows the time in seconds to solve a complex parallel ray tracing problem with large data requirements using more than one application process per processing element. As can be seen, increasing the number of application processes per processing

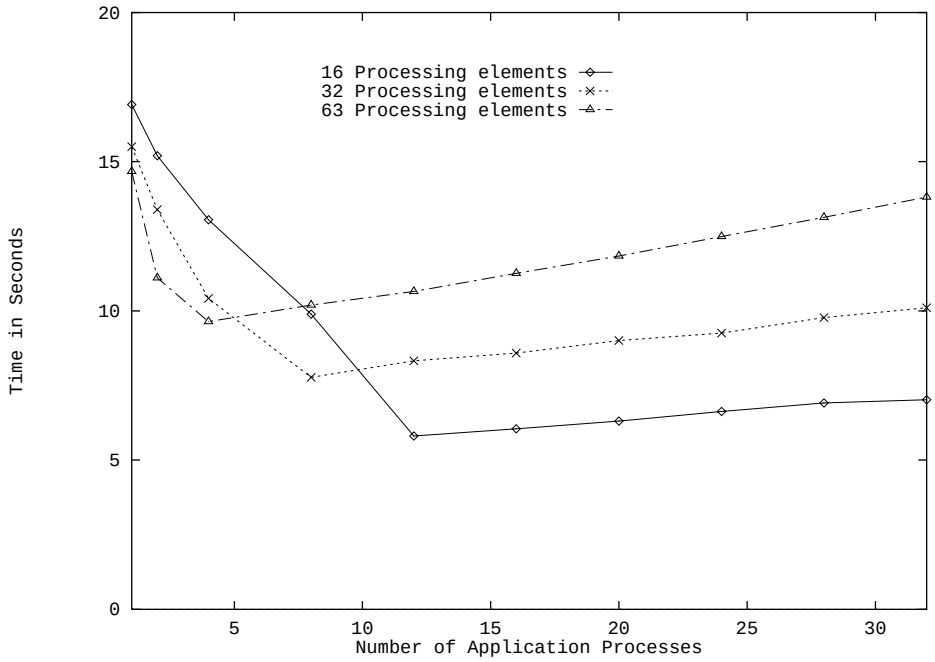


Figure 3.9: Problem solution time in seconds

element produces a performance improvement until a certain number of threads have been added. Beyond this point, the overheads of having the additional threads are greater than the benefit gained, and thus the times to solve the problem once more increase. The number of threads at which the overheads outweigh the benefits gained is lower for larger numbers of processing elements. This is because the more application processes there are per processing element, the larger the message output from each processing element will be (assuming an average number of remote fetches per thread). As the average distances the remote data fetches have to travel in larger systems is greater, the impact of increasing numbers of messages on the overall system density is more significant and thus the request latency will be higher. Adding more threads now no longer helps overcome communication delays, but in fact, the increasing number of messages actually exacerbates the communication difficulties. Ways must be found of dynamically scheduling the optimum number of application processes at each processing element depending on the current system message densities.

Despite these shortcomings, multi-threading does work well, especially for low numbers of threads and is a useful technique for avoiding idle time in the face of unpredictable data item requirements. Remember that multiple threads are only needed at a processing element if a prefetch strategy is not possible and the data item required by one thread was not available locally. If ways can be found to try and guess which data items are likely to required next then, if the data manager is right at least some of the time, the number of remote fetches-on-demand will be reduced. Caching and profiling assist the data manager with these predictions.

3.5.3 Profiling

Although primarily a task management technique, profiling is used explicitly to assist with data management, and so is discussed here. At the start of the solution of many problems, no knowledge exists as to the data requirements of any of the tasks. (If this knowledge did exist then a prefetching strategy would be applicable). Monitoring the solution of a single task provides a list of all the data items required by that task. If the same monitoring action is carried for all tasks then at the completion of the problem, a complete “picture” of the data requirements of all tasks would be known. Profiling attempts to predict the

data requirements of future tasks from the list of data requirements of completed tasks.

Any spatial coherence in the problem domain will provide the profiling technique with a good estimate of the future data requirements of those tasks from a similar region of the problem domain. The data manager can now use this profiling information to *prefetch* those data items which are *likely* to be used by subsequent tasks being performed at that processing element. If the data manager is always correct with its prediction then profiling provides an equivalent situation to prefetching in which the application process is never delayed awaiting a remote fetch. Note in this case there is no need for multi-threading.

A simple example of spatial coherence of the problem domain is in shown in figure 3.10. This figure is derived from figure 2.3 which showed how the principal data item (PDI) and additional data items (ADIs) made up a task. In figure 3.10 we can see that task i and task j come from the same region of the problem domain and spatial coherence of the problem domain has meant that these two tasks have three additional data items in common. Task k, on the other hand, is from a different region of the problem domain, requires only one additional data item which is not common to either task i or task j.

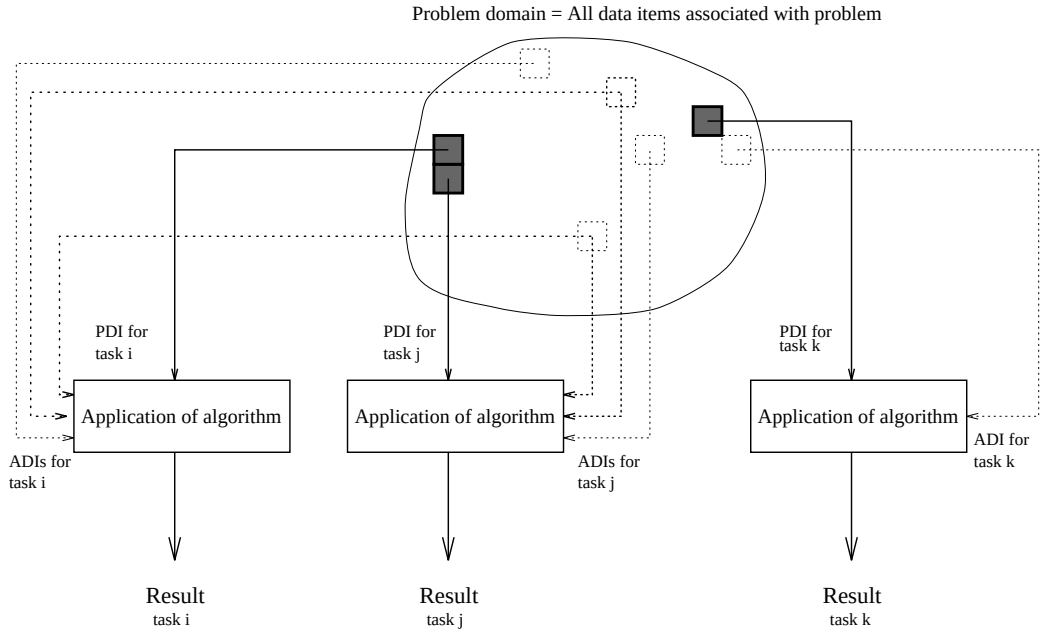


Figure 3.10: Common additional data items due to spatial coherence of the problem domain

Thus, the more successful the predictions are from the profiling information, the higher will be the cache-hit ratios. From figure 3.10 on page 65 we can see that if the completion of task i was used to profile the data item requirements for task j then, thanks to the spatial coherence of task i to task j in the problem domain, the data manager would have a 66% success rate for the additional data items for task j. However, a similar prediction for the additional data items for task k would have a 0% success rate and result in a 100% cache-miss, that is all the additional data items for task k would have to be fetched-on-demand.

3.6 Data Management for Multi-Stage Problems

In section 2.3.3 we discussed the algorithmic and data dependencies that can arise in problems which exhibit more than one distinct stage. In such problems, the results from one stage become the principal data items for the subsequent stage as was shown in figure 2.12. So, in addition to ensuring the application processes are kept supplied with data items during one stage, the data manager also needs to be aware as to how the partial results from one stage of the computation are stored at each processing element in anticipation of the following stage.

This balancing of partial result storage could be achieved statically by all the results of a stage being returned to the system controller. At the end of that current stage the system controller is in a position to

distribute this data evenly as the principal and additional data items for the next stage of the problem. The communication of these potentially large data packets twice, once during the previous stage to the system controller and again from the system controller to specific processing elements, obviously may impose an enormous communication overhead. A better static distribution strategy might be to leave the results in place at the processing elements for the duration of the stage and then have them distributed from the processing elements in a manner prescribed by the system controller. Note that in such as scheme the local cache of each processing element must be able to hold not only the principal and additional data items for the current stage, but also have space in which to store these partial results in anticipation of the forthcoming stage. It is important that these partial results are kept separate so that they are not inadvertently overwritten by data items during the current stage.

In a demand driven model of computation the uneven computational complexity may result in a few processing elements completing many more tasks than others. This produces a flaw in the second static storage strategy. The individual processing elements may simply not have sufficient space in their local cache to store more than their fair share of the partial results until the end of the stage.

Two dynamic methods of balancing this partial result data may also be considered. Adoption of the preferred bias task management strategy, as discussed in section 2.4.5, can greatly facilitate the correct distribution of any partial results. Any results produced by one processing element from another's conceptual portion, due to task load balancing, may be sent to directly to this other processing element. The initial conceptual allocation of tasks ensures that the destination processing element will have sufficient storage for the partial result.

If this conceptual allocation is not possible, or not desirable, then balancing the partial results dynamically requires each processing element to be kept informed of the progress of all other processing elements. This may be achieved by each processing element broadcasting a short message on completion of every task to all other processing elements. To ensure that this information is as up to date as possible, it is advisable that these messages have a special high priority so that they may be handled immediately by the router processes, by-passing the normal queue of messages. Once a data manager's local cache reaches its capacity the results from the next task are sent in the direction of the processing element that is known to have completed the least number of tasks and, therefore, the one which will have the most available space. To further reduce the possible time that this data packet may exist in the system, any processing element on its path which has storage capacity available may absorb the packet and thus not route it further.

Bibliography

[1] W. B. Ackerman. Data flow languages. In N. Gehani and A. D. McGettrick, editors, *Concurrent Programming*, chapter 3, pages 163–181. Addison-Wesley, 1988.

[2] G. S. Almasi and A. Gottleib. *Highly Parallel Computing*. Benjamin/Cummings, Redwood City, California, 2^{nd} edition, 1994.

[3] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS*, volume 30, Atlantic City, Apr. 1967. AFIPS Press, Reston, Va.

[4] M. Annaratone et al. Warp architecture and implementation. In *13^{th} Annual International Symposium on Computer Architecture*, pages 346–356, Tokyo, June 1986.

[5] J. Backus. Can programming be liberated from the von Neumann style functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[6] H. Bal. *Programming Distributed Systems*. Silicon Press, Summit, New Jersey, 1990.

[7] A. Basu. A classification of parallel processing systems. In *ICCD*, 1984.

[8] K. Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, 29(9):836–840, Sept. 1980.

[9] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Addison-Wesley, Wokingham, England, 1990.

[10] A. W. Burks. Programming and structural changes in parallel computers. In W. Händler, editor, *Conpar*, pages 1–24, Berlin, 1981. Springer.

[11] N. Carriero and D. Gelernter. *How to Write Parallel Programs*. MIT Press, Cambridge, Massachusetts, 1990.

[12] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the 4^{th} International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-IV*, pages 224–234, Apr. 1991.

[13] A. G. Chalmers. Occam - the language for educating future parallel programmers? *Microprocessing and Microprogramming*, 24:757–760, 1988.

[14] A. G. Chalmers and D. J. Paddon. Communication efficient MIMD configurations. In *4^{th} SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, 1989.

[15] P. Chaudhuri. *Parallel Algorithms: Design and analysis*. Prentice-Hall, Australia, 1992.

[16] B. Codenotti and M. Leonici. *Introduction to parallel processing*. Addison-Wesley, Wokingham, England, 1993.

[17] A. L. DeCegama. *The Technology of Parallel Processing: Parallel Processing Architectures and VLSI Design*. Prentice-Hall International Inc., 1989.

[18] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, Mar. 1989.

[19] V. Faber, O. M. Lubeck, and A. B. White Jr. Super-linear speedup of an efficient sequential algorithm is not possible. *Parallel Computing*, 3:259–260, 1986.

[20] H. P. Flatt and K. Kennedy. Performance of parallel processors. *Parallel Computing*, 12:1–20, 1989.

[21] M. J. Flynn. Some computer organisations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[22] C. F. Gerald and P. O. Wheatley. *Applied numerical analysis*. World Student Series. Addison-Wesley, Reading, MA, 5^{th} edition, 1994.

[23] S. A. Green and D. J. Paddon. A non-shared memory multiprocessor architecture for large database problems. In M. Cosnard, M. H. Barton, and M. Vanneschi, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing*, Pisa, 1988.

[24] H. A. Grosch. High speed arithmetic: The digital computer as a research tool. *Journal of the Optical Society of America*, 43(4):306–310, Apr. 1953.

[25] H. A. Grosch. Grosch's law revisited. *Computerworld*, 8(16):24, Apr. 1975.

[26] J. L. Gustafson. Re-evaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[27] D. R. Hartree. The ENIAC, an electronic computing machine. *Nature*, 158:500–506, 1946.

[28] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A quantitative approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[29] T. Hey. Scientific applications. In G. Harp, editor, *Transputer Applications*, chapter 8, pages 170–203. Pitman Publishing, 1989.

[30] D. W. Hillis. *The Connection Machine*. The MIT Press, 1985.

[31] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms*, chapter 1, pages 60–81. Adam Hilger, 1988.

[32] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms*. Adam Hilger, Bristol, 1988.

[33] M. Homewood, M. D. May, D. Shepherd, and R. Shepherd. The IMS T800 transputer. *IEEE Micro*, pages 10–26, 1987.

[34] R. M. Hord. *Parallel Supercomputing in MIMD Architectures*. CRC Press, Boca Raton, 1993.

[35] R. J. Hosking, D. C. Joyce, and J. C. Turner. *First steps in numerical analysis*. Hodder and Stoughton, Lonfon, 1978.

[36] HPF Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1), June 1993.

[37] K. Hwang. *Advanced Computer Architecture*. McGraw-Hill Series in Computer Science. McGraw-Hill, New York, 1993.

[38] KSR. *KSR Technical Summary*. Kendall Square Research, Waltham, MA, 1992.

[39] V. Kumar, A. Grama, A. Gupta, and G. Karyps. *Introduction to Parallel Computing*. Benjamin/Cummings, Redwood City, California, 1994.

[40] H. T. Kung. *VLSI array processors*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[41] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In Duff and Stewart, editors, *Sparse Matrix proceedings*, Philadelphia, 1978. SIAM.

[42] C. Lazou. *Supercomputers and Their Use*. Claredon Press, Oxford, revised edition, 1988.

[43] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan. 1993.

[44] T. Lewis and H. El-Rewini. *Introduction to parallel computing*. Prentice-Hall, 1992.

[45] K. Li. Ivy: A shared virtual memory system for parallel computing. *Proceedings of the 1988 International Conference on Parallel Processing*, 2:94–101, Aug. 1988.

[46] G. J. Lipovski and M. Malek. *Parallel Computing: Theory and comparisons*. John Wiley, New York, 1987.

[47] M. D. May and R. Shepherd. Communicating process computers. Inmos technical note 22, Inmos Ltd., Bristol, 1987.

[48] L. F. Menabrea and A. Augusta(translator). Sketch of the Analytical Engine invented by Charles Babbage. In P. Morrison and E. Morrison, editors, *Charles Babbage and his Calculating Engines*. Dover Publications, 1961.

[49] D. Nussbaum and A. Argarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):56–61, Mar. 1991.

[50] B. Purvis. Programming the Intel i860. *Parallelogram International*, pages 6–9, Oct. 1990.

[51] M. J. Quinn. *Parallel Computing: Theory and practice*. McGraw-Hill, New York, 1994.

[52] V. Rajaraman. *Elements of parallel computing*. Prentice-Hall of India, New Dehli, 1990.

[53] S. F. Reddaway. DAP - a Distributed Array Processor. In *1^{st} Annual Symposium on Computer Architecture*, 1973.

[54] R. M. Russel. The CRAY-1 computer system. *Communications of the ACM*, 21:63–72, 1978.

[55] J. E. Shore. Second thoughts on parallel processing. *Comput. Elect. Eng.*, 1:95–109, 1973.

[56] R. J. Swam, S. H. Fuller, and D. P. Siewiorek. 'Cm^*—A Modular, Multi-Microprocessor'. In *Proc. AFIPS 1977 Fall Joint Computer Conference 46*, pages 637–644, 1977.

[57] S. Thakkar, P. Gifford, and G. Fiellamd. The Balance multiprocessor system. *IEEE Micro*, 8(1):57–69, Feb. 1988.

[58] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins. Data driven and demand-driven computer architecture. *Communications of the ACM*, 14(1):95–143, Mar. 1982.

[59] A. Trew and G. Wilson, editors. *Past, Present and Parallel: A survey of available parallel computer systems*. Springer-Verlag, London, 1991.

[60] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[61] D. H. D. Warren and S. Haridi. The Data Diffusion Machine—a scalable shared virtual memory multiprocessor. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 943–952, Tokyo, Japan, Dec. 1988.

[62] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

Section II
Classification of Parallel Rendering Systems

Timothy A. Davis

Contents

Section II: Classification of Parallel Rendering Systems

Section II Classification of Parallel Rendering Systems

In the preceding chapters, we have reviewed the fundamental concepts of parallel processing and given some indication of how it might be effectively used in graphics rendering. Since many types of parallel rendering have been investigated [Gree91] [Whit92], classifying the various schemes is important to characterize the behavior of each. A parallel rendering system can be classified according to the method of task subdivision and/or by the hardware used to implement the scheme. Often, the choice of one influences the other.

Classifying by task subdivision refers to the method in which the original rendering task is broken into smaller pieces to be processed in parallel. Obviously, such subdivision strongly depends on the type of rendering employed. A task for rendering polygons will offer a different set of subdivision opportunities than a ray tracing task. Also included in these decisions is the type of load balancing technique to employ.

Ultimately the rendering scheme is implemented within some sort of parallel environment. The system may run on parallel hardware (e.g., a general multiprocessor or specialized hardware) or in a distributed computing environment (a group of individual machines working together to solve a single problem). The advantages and disadvantages associated with each environment are discussed below.

1 Classification by Task Subdivision

In this section, we will look at two different types of rendering (polygon-based rendering and ray tracing) and various methods for subdividing the original task into subtasks for parallel processing. Although many subdivision techniques exist for each, we will focus on the schemes most widely used. For each technique, recall that our goal is to subdivide the original task in such a way as to maximize parallelism, while not creating excessive overhead.

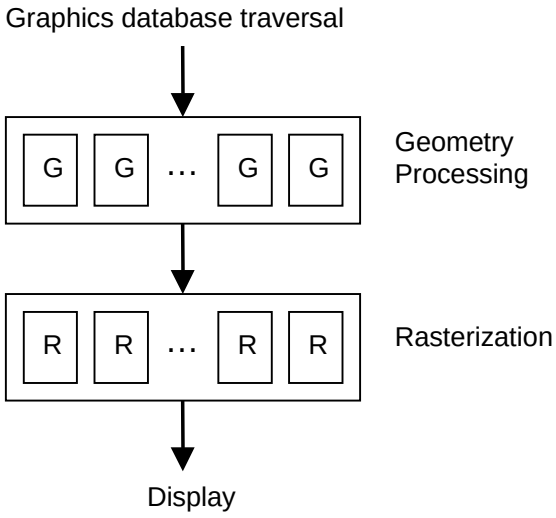


Figure 1 Polygon rendering pipeline

1.1 Polygon Rendering

For polygon rendering, we often deal with a very large number of primitives (e.g., triangles) which can often be processed in a parallel manner. To handle this type of rendering, a graphics pipeline is usually employed (see Figure 1). Stages in this pipeline include geometry processing and rasterization.

Geometry processing comprises transformation, clipping, lighting, and other tasks associated with a primitive. A straightforward method for parallelizing geometry processing is to assign each processor a subset of primitives (or objects) in the scene to render. In rasterization, scan-conversion, shading, and visibility determination are performed. To parallelize this processing, each processor could perform the pixel calculations for a small part of the final image.

One way to view the processing of primitives is as a problem of sorting primitives to the screen since a graphics primitive can fall anywhere on or off the screen [Moln94]. For a parallel system, we need to distribute data across processors to keep the load balanced. Actually, this sort can occur anywhere in the rendering pipeline:

- during geometry processing (sort-first)
- between geometry processing and rasterization (sort-middle)
- during rasterization (sort-last)

The structure of the parallel rendering system is determined by the location of this sort. The following discussion follows that in [Moln94].

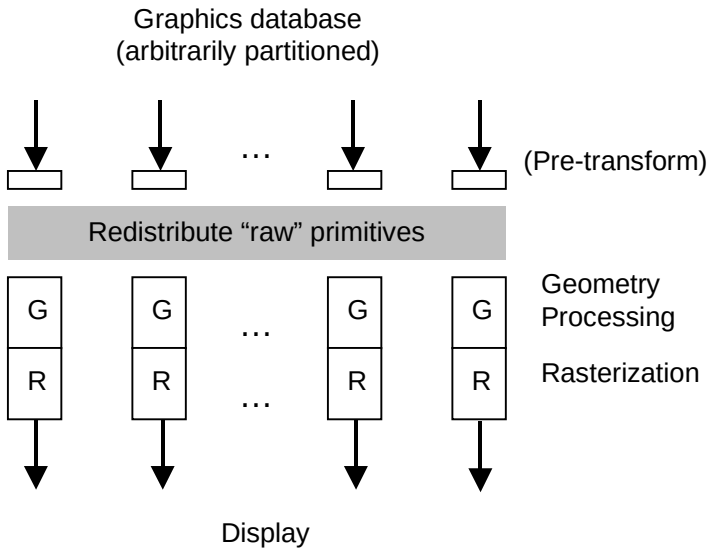


Figure 2 Sort-first polygon rendering scheme

1.1.1 Sort-First

The main idea behind sort-first is to distribute primitives early (during geometry processing) in the rendering pipeline (see Figure 2). The screen is divided into regions of equal size (see Figure 3), and each processor (or renderer) is assigned a region. Each processor is responsible for all the pixel calculations that are associated with its screen region.

In an actual implementation, primitives would initially be assigned to processors in an arbitrary way. Each renderer then performs enough transformation processing to determine the screen region into which the primitive falls. If this region belongs to another processor, the primitive is sent over the interconnection network to that processor for rendering. After each primitive has been placed with the proper renderer, all of the processors can work in parallel to complete the final image.

With this method, each processor implements the entire rendering pipeline for its portion of the screen. Communication costs can be kept comparatively low compared with other methods if features such as frame coherence are properly exploited. For rendering a single frame, however, almost all the primitives will have to be redistributed after the initial random assignment. Some duplication of effort may occur if a primitive falls into more than one region, or if the results of the original geometry processing are not sent with the primitive when it is transmitted to the appropriate renderer. Also, the system is susceptible to load imbalance since primitives may be concentrated into particular regions or may simply take longer to render. Both of these situations will cause the affected processor to consume more time in processing its screen region. Very few, if any, sort-first renderers have been built.

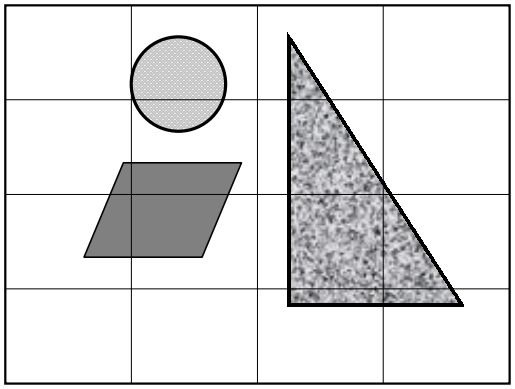


Figure 3 Image space subidivision

1.1.2 Sort-Middle

In a sort-middle renderer, primitives are sorted and redistributed in the middle of the pipeline: between geometry processing and rasterization (see Figure 4). By this point, the screen coordinates of the primitives have been determined through transformation processing, but the primitives have not yet been rasterized. This point is a natural breaking position in the rendering pipeline.

In an actual implementation, primitives are arbitrarily assigned to processors as before. The geometry processors perform transformation, lighting, and other processing on the primitives originally assigned to them, and then classify the primitives according to screen region. Screen-space primitives are then sent to the appropriate

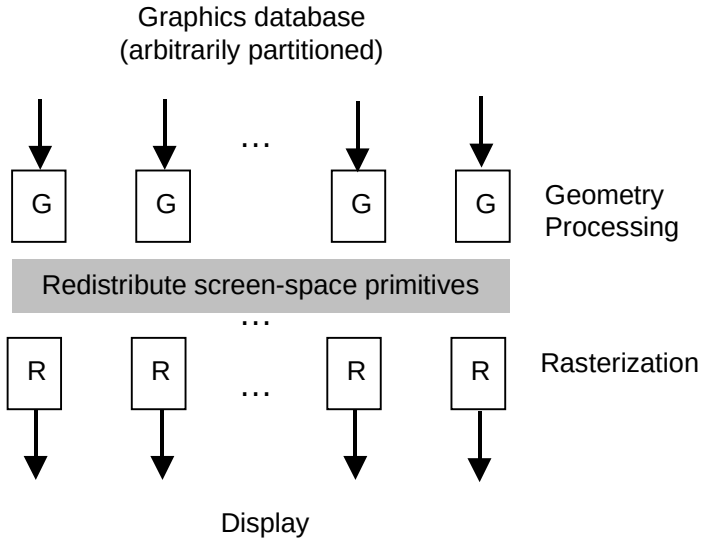


Figure 4 Sort-middle polygon rendering scheme

rasterizing processors, which have been assigned unique regions of the screen, for the remaining processing.

The sort-middle strategy is general and straightforward and has been implemented in both hardware (including Pixel-Planes 5 [Fuch89] and the SGI Reality Engine [Akel93]) and software [Whit94] [Ells94]. Like sort-first, however, this method is susceptible to load imbalance due to the uneven distribution of primitives across screen space. Communication times can be higher under certain conditions. Also, primitives that overlap regions may require some additional processing.

1.1.3 Sort-Last

Under the sort-last strategy, sorting is deferred until the end of the rendering pipeline (see figure 5). Primitives are rasterized into pixels, samples, or pixel fragments, which are then transmitted to the appropriate processor for visibility determination.

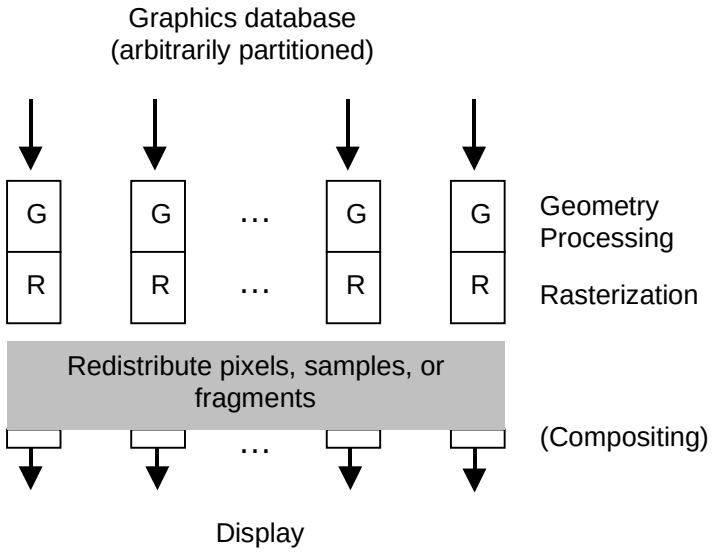


Figure 5 Sort-last polygon rendering scheme

In practice, primitives are initially distributed to processors in an arbitrary manner, as in the other methods. Each renderer then performs the operations necessary to compute the pixel values for its primitives, regardless of where those pixels may reside on the screen. These values are then sent to the appropriate processors according to screen location. At this point, the rasterizing processors perform visibility calculations and composite the pixels for final display.

As in sort-first, each processor implements the entire graphics pipeline for its primitives. While the overall technique is less prone to load imbalance, the pixel traffic in the final sort can be very high. Numerous rendering systems using the sort-last method have been constructed in various forms, including [Evan92] and [Kubo93].

1.2 Ray Tracing

Ray tracing is a powerful rendering technique that can produce high-quality graphics images; however, this quality comes at a price of intensive calculation and long rendering times. Even relatively simple ray-traced animations can prohibitively expensive to render on a single processor. For longer, more complex animations, the rendering time can be intractable. Fortunately, ray tracing is a prime candidate for parallelization since its processing is readily amenable to subdivision. Specifically, ray tracing inherently contains a large amount of parallelism due to the independent nature of its pixel calculations [Whit80]; therefore, most ray tracing rendering algorithms lend themselves to parallelization in screen space.

Other partitioning schemes are employed in ray tracing as well. Instead of dividing the image space, the object space can be split into smaller regions, or the objects themselves may be assigned to individual processors. These techniques are discussed more fully below.

1.2.1 Image Space Partitioning

Using this scheme, the viewing plane is divided into regions, each of which is completely rendered by an individual processor (see Figure 3). That is, for each pixel in a region, the processor assigned to that region computes its entire ray tree. While this technique is conceptually straightforward, the entire database of scene objects must be accessible to every processor.

The benefits of this approach are simplicity and low interprocessor communication as compared with other partitioning methods; the largest drawback is its limitation to multiprocessor architectures with significant local processor memory. Another potential problem is load imbalance, since image detail may be concentrated in certain regions of the screen. To combat this situation, the load balancing algorithm may further subdivide complex regions to provide idle processors with additional tasks.

1.2.2 Object Space Partitioning

Here the 3-dimensional space where the scene objects reside is divided into subvolumes, or voxels. Voxels may not be equally sized in order to achieve better load balancing. In the initialization phase of ray tracing, each voxel is parceled out to a particular processor. When rays are cast during rendering, they are passed from processor to processor as they travel through the object space. Each processor, therefore, needs only the scene information associated with its assigned voxels.

While this technique may not suffer from frequent load imbalance, it does incur costs in other ways. First, as new rays are shot, they must tracked through voxel space; this processing is not required for other schemes. Additionally, since potentially millions of rays are fired for each image, communication could be become excessive as rays enter and exit regions of object space during rendering.

1.2.3 Object Partitioning

This partitioning scheme parallelizes the rendering task by assigning each object to an individual processor. As in object space partitioning, rays are passed as messages between processors, which in turn test the ray for intersection with the objects they are assigned. Object partitioning also shares some of the benefits and detriments of object space partitioning. Specifically, the load may be fairly well-balanced, but the communication costs may be high due to the large amount of ray message traffic.

1.2.4 Load Balancing Scheme

Finally, parallel rendering schemes can be classified according to their load balancing method. Of course, the primary goal of any load balancing scheme is to distribute the work among processors as evenly as possible and thus exploit the highest degree of parallelism available in the application. Many different types of load balancing schemes exist, but each falls into one of two categories:

- *Static Load Balancing.* In this scheme, partitioning is performed up front and processors are assigned subtasks for the entire duration of the rendering process. In this way, overhead is minimized later in the rendering; however, a good deal of care must be taken to ensure that the load will be balanced. Otherwise, the algorithm will suffer from poor performance.

- *Dynamic Load Balancing.* With this scheme, some processing assignments are determined at the start, but later assignments are demand-driven. That is, when a processor determines that it needs more work to do, it will request a new assignment. In this way, processors will never be idle as long as more work is left to do. The key here is to distribute the load as evenly as possible without incurring excessive overhead.

1.2.5 Comments

Hybrid schemes have also been proposed, combining image space partitioning with object space partitioning [Bado94] and image space partitioning with object partitioning [Kim96]. When choosing a partitioning scheme, the architecture of the parallel machine should be considered. For instance, an image space partitioning algorithm will perform better on a MIMD machine than on a SIMD machine. In general, tradeoffs exist between the type of partitioning algorithm used and the architecture chosen.

2 Classification by Hardware

As previously stated, for some computationally intensive rendering tasks, parallel processing provides the only practical means to a solution. One way to perform parallel rendering is to use a single multiprocessor machine, such as a Thinking Machines CM-5, Intel Paragon, Cray T3E, or specialized parallel processor. In these machines, enormous computing power is provided by up to tens of thousands of processing elements able to access many gigabytes of memory and to work in concert through a high-speed interconnection network. Multiprocessors are the most powerful computers in the world and play an active role in solving Grand Challenge problems, such as weather prediction, fluid dynamics, and drug design [Hwan93].

An alternative to using traditional multiprocessor systems for parallel processing is to employ a network of workstations acting as a single machine. This approach, termed distributed or cluster computing, is conceptually similar to a multiprocessor, but each processing element consists of an independent machine connected to a network usually much slower than a multiprocessor interconnection network. While this network can be of any type (e.g., Ethernet, ATM) or topology, the computers connected to it are generally UNIX-based machines which support some type of distributed programming environment, such as Parallel Virtual Machine (PVM) [Geis94] or Message Passing Interface (MPI) [Grop94]. Many types of applications can benefit from distributed computing, including computation-intensive graphics tasks, such as ray tracing [Sung96].

This section focuses on past work that has been documented using traditional multiprocessors and clusters of machines to accomplish graphics rendering tasks, particularly in the area of ray tracing. Included in this discussion is relevant background concerning PVM and MPI, as well as motivation for using these systems in a clustered environment.

2.1 Parallel Hardware

Since rendering consumes such a large amount of computing resources and time, a good deal of effort has gone into exploring parallel solutions on multiprocessor machines. Some of the schemes proposed are designed to run on general-purpose parallel machines, such as the CM-5, while others rely on specialized hardware built especially for ray tracing rendering. A brief survey of these techniques appears in the following sections. Although current research continues in the design and implementation of parallel rendering systems, a flurry of activity in this area occurred in the late1980s and early 1990s, as reflected in many of the references.

2.1.1 General-Purpose Multiprocessors

In [Plun85], a vectorized ray tracer is proposed for the CDC Cyber 205. In a given execution cycle, rays awaiting processing are distributed to individual processors and ray-object calculations are

performed object by object in a lock-step SIMD fashion.

Similarly, [Crow88] implements a SIMD ray tracing algorithm, but for the Connection Machine (CM-2). Image subdivision is used with one pixel being assigned to each of the 16K processors to produce a 128x128 image, with ray-object intersections performed on an object by object basis. The algorithm proposed by [Schr92] also runs on a CM-2, but uses an object-space subdivision coupled with processor remapping capabilities to achieve dynamic load balancing.

Rounding out the SIMD field, [Goel96] describes a ray casting method developed on the MasPar MP-1 for volume rendering, another computation-intensive graphics application used for viewing complex structures in medical imaging and other forms of scientific visualization. To handle the large amount of data and processing involved, machines are assigned portions of the volume to render, which are composited to produce a final image. This system allows users to rotate a volume, magnify areas of interest, and perform other viewing operations.

In the MIMD category, [Reis97] employs an IBM SP/2 running an image-space partitioning scheme with dynamically adjustable boundaries to render frames of an animation progressively. In this form of rendering, termed progressive rendering, an image is initially rendered quickly at low resolution and progressively refined when little or no user interaction takes place. Progressive rendering is useful in interactive environments where frame generation rate is important. The goal of [Keat95] also involves progressive rendering, although their renderer makes use of object-space partitioning on the Kendall Square Research KSR1 machine.

Several research efforts have focused on the Intel iPSC machines as the architectural environment for implementing a parallel ray tracer. Interestingly, whether the partitioning scheme is image-based [Isle91] [Silv94], object space-based [Prio88] [Prio89] or a hybrid of the two [Akti94] [Bado94], the load balancing scheme is almost always of a static nature ([Isle91] also tests a dynamic scheme). This choice results from a concern that dynamic load balancing schemes produce a large number of messages, which in turn, may dramatically affect the performance of a distributed machine [Prio89].

In the area of transputer-based machines, [Gree90] uses an image subdivision technique combined with memory and cache local to each processor to deal with the many required accesses to the scene description database. Here, the granularity of parallelism is controlled through the size of the image subregion, which also relates directly to the effectiveness of the dynamic load balancing scheme. To render ray-traced animations, [Maur93] use a static object-space partitioning scheme on a system of 36 transputers. Progressive ray tracing and volume rendering on transputers is addressed by [Sous90] and [Pito93], respectively.

2.1.2 Specialized Multiprocessors

Probably one of the most noteworthy examples of a specialized multiprocessor for polygon rendering is the series of Pixel-Planes machines developed at UNC-Chapel Hill. The Pixel-Planes 4 machine [Eyle88] is a SIMD machine with three basic components: a host workstation, a graphics processor, and a frame buffer. Each of the customized processors is responsible for a column of display pixels. For its time, it provided good performance; however, the system used processors with slow clock speeds and did not provide effective load balancing.

Pixel-Planes 5 [Fuch89] provided some improvements over Pixel-Planes 4 by incorporating faster processors and employing a more flexible MIMD architecture. The system implemented a sort-middle algorithm, with each processor in charge of a particular region of the screen. To handle the communication, a ring architecture capable of handling eight messages simultaneously is employed. Ultimately, the ring network imposes a limit on scalability.

The PixelFlow machine [Moln92] was developed to overcome the limitations of the previous architectures through parallel image composition. Each individual processor works to create a full-screen image using only the primitives assigned to it. All of these images are collected and composited to form the final display.

For ray tracing, [Lin91] employs a specialized SIMD machine to perform stochastic ray tracing. The stochastic method adds extra processing to the ray tracing algorithm to handle antialiasing, an important aspect of any renderer. To overcome some of the inefficiencies found in other SIMD approaches, a combination of image space partitioning and object space partitioning is used. That is, a block of pixels is rendered by casting rays and using scene coherence to restrict the parts of object space which must be tested.

In [Gaud88] a special-purpose MIMD architecture using image space subdivision and a static load distribution is described. To overcome the problem of having the entire object database resident at each processor, a central broadcast processor issues data packets describing the object database cyclically. Here, the processors make requests for various pieces of the database, and only those parts are broadcast in a given cycle. Using a somewhat different approach, [Shen95] uses object space partitioning on clusters of processors, but each processor operates in a pipelined fashion, a scheme previously explored in the LINKS-1 architecture [Nish83].

One of the few multiprocessor architectures which allocates work based on object subdivision combined with image subdivision is proposed by [Kim96]. Each processor handles ray-object intersection tests with its assigned objects, which are spread across the object space. If the load becomes unbalanced, objects are dynamically transferred to other processors.

Other specialized multiprocessor machines of note are the Pixel Machine [Potm89] (useful for several types of rendering including ray tracing and the RayCasting Engine [Meno94] (specifically built for CSG modeling).

2.1.3 Distributed Computing Environments

Parallel rendering using distributed computing environments continues to grow in popularity, especially in the fields of entertainment and scientific visualization. Below are a few interesting examples.

Perhaps the most popular example is the Disney film, *Toy Story*, which used a network of 117 Sun workstations and the Pixar Renderman system to produce the animation [Henn96]. To generate its 144,000 individual frames, *Toy Story* required about 43 years of CPU time. If not for the many machines participating in the computation, the movie's production could not be realized. For some tasks, such as

applying surface textures, one machine was chosen as a server for the rest. For other tasks, such as final rendering, the machines were basically used independently to render individual frames.

Another entertainment application used a network of 40 Amiga machines to render special effects for the television series *SeaQuest* [Worl93]. Although the delivered product contained only two to three minutes of computer graphics per episode (3,600 to 5,400 frames of animation), the rendering activity was so time-intensive that the team struggled to deliver the graphics within its weekly deadline.

For the average user, some popular commercial animation packages (e.g., Alias/Wavefront, Maya, and 3D Studio) employ coarse-grain parallelism to allow rendering of individual frames of an animation across a network of machines. This technique can mean the difference between an animation being ready in hours or in days. POV-Ray has also been ported to run in a clustered environment; however, the parallelization scheme works on single images only.

Other computation-intensive graphics problems have taken advantage of the processing power of distributed computing, specifically volume rendering and virtual reality. Although real-time interaction in these systems is constrained by the relatively slow network connecting the machines, significant speedups have been reported using a network of IBM RS/6000 machines for volume rendering [Gier93] [Ma93], and a network of Sun Sparcstations and HP workstations for virtual environments [Pan96].

Distributed computing is also being applied to computer vision algorithms [Judd94], a field closely related to graphics. Here, researchers use PVM on a cluster of 25 Sun Sparcstations for an edge-detection algorithm. One remarkable result of their experiments is that they achieved superlinear speedup on the cluster over the sequential version. This result is due to the large aggregate memory of the clustered machines, which reduced the amount of paging as compared to the single processor.

Not nearly as much research has been conducted concerning ray tracing in distributed computing environments as in traditional multiprocessor machines. Perhaps this fact is due to the relatively recent introduction of PVM and MPI. Regardless of the reason, more advanced parallel ray tracing algorithms combined with a distributed computing environment remain a largely unexplored area. Several related projects are summarized below.

For single images, [Jeva89] uses a dynamic load balancing technique with spatial partitioning and a novel warp synchronization method. At the other end of the spectrum, [Ris94] applies a static load balancing scheme using object partitioning on a network composed of both sequential workstations and parallel computers. Surprisingly few systems use an image partitioning scheme in a distributed computing environment, even though it represents the technique with the highest potential for speedup [Clea86] and overcomes the problems of limited local memory that exist in traditional multiprocessors.

For animations, [DeMa92] describes the DESIRe (Distributed Environment System for Integrated Rendering) system, which incorporates a coarse-level dynamic load balancing scheme that distributes individual frames of an animation to networked workstations. The goal of [Stob88] is similar, except that the system is designed to run without affecting the regular users of the workstations. By stealing idle cycles from 22-34 workstations, a ray-traced animation lasting five minutes (7550 frames) was rendered in two months, although the overall task was estimated at 32 CPU-months.

The work presented in [Cros95] uses a relatively small (three-machine) distributed environment for ray tracing animations in virtual reality applications. Here, each of the machines has a special task

assigned to it according to its processing specialty. In order to achieve close to interactive rates, the system, which takes advantage of progressive refinement, is composed of fairly powerful individual processors connected by an ATM network.

2.2 Discussion of Architectural Environments

For parallel processing tasks, the fastest systems will generally be the specialized multiprocessor machines, since they are built with a specific task in mind. Next will be general-purpose multiprocessor machines. Although distributed environments may provide the same number of processors as a multiprocessor machine, computations will be performed more quickly on multiprocessors due to their high-speed interconnection networks. Even so, several factors have motivated a trend toward distributed computing.

First, and perhaps most importantly, not many organizations can afford a parallel machine, which can easily cost millions of dollars [Geis94]. Many sites, however, already have some type of network of computers. Second, multiprocessors often employ specialized or exotic hardware and software resources that significantly increase the complexity, and hence the cost, of the machine; conversely, great expense is rarely incurred to perform distributed computing because the network and the machines are usually already in place. Surprisingly, distributed computing has proven to be so cost-effective that networks of standard workstations have been purchased specifically to run parallel applications that were previously executed on more expensive supercomputers [Grop94].

Due to the fact that networks of workstations are loosely coupled, distributed computing environments allow the network to grow in stages and take advantage of the latest network technology. As CPUs evolve to faster speeds, workstations can be swapped out for the latest model. Such flexibility in network and processor choice is not usually available on a multiprocessor. Another consideration is system software: operating system interface, editors, compilers, debuggers, etc. A benefit of workstation platforms is that they remain relatively stable over time, allowing programmers to work in familiar environments. To use multiprocessor systems, developers may have to climb a steep learning curve.

Additionally, in a distributed computing networked environment, the interconnected computers often consist of a wide variety of architectures and capabilities. This heterogeneity leads to a rich variety of machine combinations and computing possibilities, which can be tailored to specific applications to reduce overall execution time. On the other hand, a multiprocessor machine does not spend processing time converting data between various machine types, as a distributed computing environment might.

Finally, while utilization and efficiency are extremely important in the multiprocessor world, users on a network of traditional machines rarely consider these issues. The results are under-utilized computers which spend much of their time idle. With distributed computing, some of those idle cycles can be put to good use without impacting the primary users of the machines.

2.3 Message-Passing Software for Distributed Computing Environments

To realize distributed computing, computers in a network must support some type of distributed programming environment that allows users to write parallel applications for networked machines. This programming environment should provide a common interface for developers to pass messages easily across various network types and between machines of differing architectures. Although many additional features are usually included, a distributed programming environment need only provide a minimum set of capabilities to be useful [Grop94]:

- First, some method must exist to start up and initialize the parallel processes on all participating machines. This procedure may be as simple as specifying each machine and an associated command in a static file, or spawning the processes directly within the program of the master process. Here, the master process refers to a user-initiated process responsible for delegating work and compositing results; conversely, slave processes perform only the work assigned to them and report results back to the master.

- Once start-up is complete, a process should be able to identify itself, as well as other processes running on the local machine or remote machines participating in the work. Such identification is useful for specifying the source and destination of transmitted messages.

- Since a distributed computing environment often consists of machines with widely varying architectures, message transmission must account for differing data formats so that all computers on the network understand the data exchanged between them. This capability is often built into the programming library, which first transforms the data into a common format that can be easily decoded on the receiver's side. For this reason, among others, a version of the distributed programming environment must exist for every type of machine architecture participating in the computation.

- Finally, once an application is complete, some way of terminating all the processes must be available.

Many distributed programming environments have received attention in the last five years, including p4 [Butl94], Express [Flow94], Linda [Carr94], and TCGMSG [Harr91]; however, by far the most popular systems are PVM and MPI. PVM was developed at Emory University and Oak Ridge National Laboratory and was first released in 1991. The MPI standard, an international effort, was introduced in 1993.

Both the PVM and MPI message-passing environments, freely available on the world-wide web, provide common interfaces for communication on both multiprocessors and networks of workstations. Both run on many different machines, allowing networked computers of diverse architectures to emulate a distributed-memory multiprocessor. Each of the machines in the network may be a single-processor or multiprocessor system.

A running process in either PVM or MPI can view a network of computers as a single, virtual machine, ignoring architectural details, or as a set of specialized processors with unique computational abilities. The master process runs on a single computer from which other tasks participating in the

computation are initiated. Processes, or tasks, roughly correspond to UNIX processes and operate independently and sequentially, performing both communication and computation. Multiple tasks can run on multiple machines, on a single machine, or a combination of the two.

PVM and MPI are not programming languages; rather, they provide libraries specifying the names, parameters, and results of Fortran and C routines used in message passing. Any program making use of these routines can be compiled with standard compilers by linking in the PVM or MPI library. Note that the developer controls the parallelism in the program by writing master and slave tasks and explicitly specifying the high-level message passing protocol between them. Both PVM and MPI support functional parallelism, in which each task is assigned one function of a larger process; data parallelism, in which identical tasks solve the same problem but for small subsets of the data; or a combination of either approach.

Bibliography

[Acke93] K. Ackeley, "RealityEngine Graphics," *Proceedings of SIGGRAPH 93* in *ACM Computer Graphics*, Aug. 1993, pp. 109-116.

[Akti94] M. Aktihanoglu, B. Ozguc, and C. Aykanat, "MARS: A Tool-based Modeling, Animation, and Parallel Rendering System," *The Visual Computer*, Vol. 11, No. 1, 1994, pp. 1-14.

[Bado94] D. Badouel, K. Bouatouch, and T. Priol, "Distributing Data and Control for Ray Tracing in Parallel," *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, 1994, pp. 69-77.

[Butl94] R. Butler and E. Lusk, "Monitors, Messages, and Clusters: the p4 Programming System," *Parallel Computing*, Vol. 20, No. 4, 1994, pp. 547-564.

[Carr94] N. Carriero, D. Gelernter, T. Mattson, and A. Sherman, "The Linda Alternative to Message-Passing Systems," *Parallel Computing*, Vol. 20, No. 4, 1994, pp. 633-655.

[Clea86] J. G. Cleary, G. Wyvill, and B. M Birtwistley, "Multiprocessor Ray Tracing," *Computer Graphics Forum*, Vol. 5, No. 1, 1986, pp. 3-12.

[Cros95] R. A. Cross, "Interactive Realism for Visualization Using Ray Tracing," *Proceedings of the 1995 IEEE Visualization Conference*, Atlanta, GA, 1995, pp. 19-26.

[Crow88] F. C. Crow, G. Demos, J. Hardy, J. McLaughlin, and K. Sims, "3D image Synthesis on the Connection Machine," *Proceedings of the International Conference on Parallel Processing for Computer Vision and Display* in *Parallel Processing for Computer Vision and Display*, P. M. Dew, T. R. Heywood, and R. A. Earnshaw (Eds.), Addison-Wesley, 1988, pp. 254-269.

[DeMa92] J. M. De Martino and R. Kohling, "Production Rendering on a Local Area Network," *Computers and Graphics*, Vol. 16, No. 3, 1992, pp. 317-324.

[Ells94] D. Ellsworth, "A New Algorithm for Interactive Graphics on Multicomputers," *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, pp. 33-40.

[Evan92] Evans and Sutherland Computer Corporation, *Freedom Series Technical Report*, Salt Lake City, Utah, October, 1992.

[Eyle88] J. Eyles, J. Austin, H. Fuchs, T. Greer, and J. Poulton, "Pixel-Planes 4: A Summary," *Advances in Computer Graphics Hardware II*, Springer-Verlag, Berlin, 1988, pp. 183-207.

[Flow94] J. Flower and A. Kolawa, "Express Is Not Just a Message Passing System-Current and Future Directions in Express," *Parallel Computing*, Vol. 20, No. 4, 1994, pp. 597-614.

[Fuch89] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *ACM Computer Graphics*, Vol. 23, No. 3, 1989, pp. 79-88.

[Gaud88] S. Gaudet, R. Hobson, P. Chilka, and T. Calvert, "Multiprocessor Experiments for High-Speed Ray Tracing," *ACM Transactions on Graphics*, Vol. 7, No. 3, 1988, pp. 151-179.

[Geis94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Network Parallel Computing*, MIT Press, Cambridge, MA, 1994.

[Gier93] C. Giertsen and J. Petersen, "Parallel Volume Rendering on a Network of Workstations," *IEEE Computer Graphics and Applications*, Vol. 13, No. 6, 1993, pp. 16-23.

[Goel96] V. Goel and A.Mukherjee, "An Optimal Parallel Algorithm for Volume Ray Casting," *The Visual Computer*, Vol. 12, No. 1, 1996, pp. 26-39.

[Gree90] S. A. Green and D. J. Paddon, "A Highly Flexible Multiprocessor Solution for Ray Tracing," *The Visual Computer*, Vol. 6, No. 2, 1990, pp. 62-73.

[Gree91] S. A. Green, *Parallel Processing for Computer Graphics*, MIT Press, Cambridge, MA, 1991.

[Grop94] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI – Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA, 1994.

[Harr91] R. J. Harrison, "Portable Tools and Applications for Parallel Computers," *International Journal of Quantum Chemistry*, Vol. 40, 1991, pp. 847-863.

[Henn96] M. Henne, H. Hickel, E. Johnson, and S. Konishi, "The Making of *Toy Story*," *IEEE COMPCON '96 Digest of Papers*, IEEE Computer Society Press, Los Alamitos, CA, 1996, 463-468.

[Hwan93] K. Hwang, *Advanced ComputerArchitecture – Parallelism, Scalability, Programmability*, McGraw-Hill, New York, 1993.

[Isle91] V. Isler and B. Ozguc, "Fast Ray Tracing 3D Models," *Computers and Graphics*, Vol. 15, No. 2, 1991, pp. 205-216.

[Jeva89] D. A. Jevans, "Optimistic Multi-Processor Ray Tracing," *New Advances in Computer Graphics (Proceedings of Computer Graphics International '89*, R. P. Earnshaw and B. M. Wyvill (Eds.), Springer-Verlag, Berlin, 1989, pp. 507-522.

[Keat95] M. J. Keates and R. J. Hubbold, "Interactive Ray Tracing on a Virtual Shared-Memory Parallel Computer," *Computer Graphics Forum*, Vol. 14, No. 4, 1995, pp. 189-202.

[Kim96] H-J. Kim and C-M Kyung, "A New Parallel Ray-Tracing System Based on Object Decomposition," *The Visual Computer*, Vol. 12, No. 5, 1996, pp. 244-253.

[Kubo93] Kubota Pacific Computer, *Denali Technical Overview*, version 1.0, Santa Clara, CA, March 1993.

[Lin91] T. T. Y. Lin and M. Slater, "Stochastic Ray Tracing Using SIMD Processor Arrays," *The Visual Computer*, Vol. 7, No. 4, 1991, pp. 187-199.

[Ma93] K-L. Ma and J. S. Painter, "Parallel Volume Visualization on Workstations," *Computers and Graphics*, Vol. 17, No. 1, 1993, pp. 31-37.

[Maur93] H. Maurel, Y. Duthen, and R. Caubet, "A 4D Ray Tracing," *Proceedings of Eurographics '91* in *Computer Graphics Forum*, Vol. 12, No. 3, 1993, pp. 285-294.

[Meno94] J. Menon, R. J. Marisa, and J. Zagajac, "More Powerful Solid Modeling through Ray Representations," *IEEE Computer Graphics and Applications*, Vol. 14, No. 3, 1994, pp. 22-35.

[Moln92] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Proceedings of SIGGRAPH 92* in *ACM Computer Graphics*, Vol. 26, No. 2, pp. 231-240.

[Moln94] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, pp. 23-32.

[Nish83] H. Nishimura, H. Ohno, T. Kawata, I. Shirakawa, and K. Omura, "LINKS-1: A Parallel Pipelined Multimicrocomputer System for Image Creation," *Proceedings of the 10th Symposium on Computer Architecture (SIGARCH)*, 1983, pp. 387-394.

[Pan96] Z. Pan, J. Shi, and M. Zhang, "Distributed Graphics Support for Virtual Environments," *Computers and Graphics*, Vol. 20, No. 2, 1996, pp. 191-197.

[Pito93] P. Pitot, "The Voxar Project," *IEEE Computer Graphics and Applications*, Vol. 13, No. 1, 1993, pp. 27-33.

[Plun85] D. J. Plunkett and M. J. Bailey, "The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed," *IEEE Computer Graphics and Applications*, Vol. 8, No. 5, 1985, pp. 52-60.

[Potm89] M. Potmesil and E. M. Hoffert, The Pixel Machine: A Parallel Image Computer," *Proceedings of SIGGRAPH '89* in *ACM Computer Graphics*, Vol. 23, No. 3, 1989, pp. 69-78.

[Prio88] T. Priol and k. Bouatouch, "Experimenting with a Parallel Ray-Tracing Algorithm on a Hypercube Machine," *Proceedings of Eurographics '88*, D. A. Duce and P. Jancene (Eds.), North-Holland, 1988, pp. 243-259.

[Prio89] T. Prioland K. Bouatouch, "Static Load Balancing for Parallel Ray Tracing on a MMD Hypercube," *The Visual Computer*, Vol. 5, Nos. 1 and 2, 1989, pp. 109-119.

[Reis97] A. Reisman, C. Gotsman, and A. Schuster, "Parallel Progressive Rendering of Animation Sequences at Interactive Rates on Distributed-Memory Machines," *Proceedings of the 1997 IEEE Parallel Rendering Symposium*, Phoenix, AZ, 1997, pp. 39-47.

[Ris94] P. Ris and D. Arques, "Parallel Ray Tracing Based upon a Multilevel Topological Knowledge Acquisition of the Scene," *Proceedings of Eurographics '94* in *Computer Graphics Forum*, Vol. 13, No. 3, 1994, pp. 221-232.

[Schr92] P. Schroder and S. M Drucker, "A Data Parallel Algorithm for Raytracing of Heterogeneous Databases," *Proceedings of Computer Graphics Interface '92*, Vancouver, British Columbia, 1992, pp. 167-175.

[Shen95] L-S. Shen, E. F. Deprettere, and P. Dewilde, "A Parallel Image-Rendering Algorithm and Architecture Based on Ray Tracing and Radiosity Shading," *Computers and Graphics*, Vol. 19, No. 2, 1995, pp. 281-296.

[Silv94] C. T. Silva and A. E. Kaufman, "Ray Parallel Performance Measures for Volume Ray Casting," *Proceedings of the 1994 IEEE Visualization Conference*, Washington, D. C., 1994, pp. 196-203.

[Sous90] A. A. Sousa, A. M. C. Costa, and F. N. Ferreira, "Interactive Ray-Tracing for Image Production with Increasing Realism," *Proceedings of Eurographics '90*, North-Holland, 1990, pp. 449-457.

[Stob88] A. Stober, A. Schmitt, B. Neidecker, H. Muller, T. Maus, and W. Leister, "*'Occursus Cum Novo'* – Tools for Efficient Photo-Realistic Computer Animation," *Proceedings of Eurographics '88*, D. A. Duce and P. Jancene (Eds.), North-Holland, Amsterdam, 1988, pp. 31-41.

[Sung96] K. Sung, J. L. J. Shiuan, and A. L. Ananda, "Ray Tracing in a Distributed Environment," *Computers and Graphics*, Vol. 20, No. 1, 1996, pp. 41-49.

[Whit92] S.Whitman, *Multiprocessor Methods for Computer Graphics Rendering*, Jones and Bartlett Publishers, Boston, 1992.

[Whit94] S.Whitman, "A Task Adaptive Parallel Graphics Renderer," *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, pp. 41-48.

[Whit80] T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, Vol. 23, No. 6, 1980, pp. 343-349.

[Worl93] L. World, "Low-End System Animates the Depths in *SeaQuest*," *IEEE Computer Graphics and Applications*, Vol. 13, No. 6, 1993, p. 93.

Section III
Interactive Ray Tracing

Erik Reinhard

Contents

III.1 Introduction

It is our belief that in the near future ray tracing will become the *de facto* standard for interactive rendering. For certain applications, multi-processor ray tracing already out-performs hardware based solutions [5, 19, 29, 30]. Whereas hardware rendering becomes ever more complex to keep up with todays demands, ray tracing is easily implemented in software, scales sub-linearly with scene-complexity and (nearly) linearly with the number of processors used. These are indeed favorable conditions.

Research has shown that using current high-end general purpose hardware, interactive ray tracing is possible for moderately complex scenes and scales well to reasonably large numbers of processors, albeit using fairly small image sizes (typically 512 by 512 pixels). One of the key features of such interactive ray tracing systems is that the inherent parallelism of ray tracing is exploited as much as possible. In other words, it is paramount that the complexity of the algorithm needs to be kept as low as possible. A simple algorithm is better than a complex one. Small data structures are better than large ones.

The choice of hardware is also quite important. High level choices such as interconnection network and memory topology can make or break an interactive ray tracing project. While shared-memory architectures are said to not scale beyond a certain number of processors due to the buss-architecture commonly employed, we believe that with current technology the latency and network throughput experienced in distributed memory systems just is not good enough for the purposes of interactive ray tracing. So this section of the course will be exclusively targeted at shared memory architectures. In fact, the interactive ray tracer described here is implemented on a Silicon Graphics Origin 2000 and results in the following chapters are shown using a Silicon Graphics Origin 3800. Both are shared memory machines with a (cache-coherent) non-uniform memory access (ccNUMA) architecture. Hardware selection and its implications are further discussed in section III.1.1

Another key feature is that at the code level the algorithm needs to be optimized as far as possible. The particular features of the hardware on which the algorithm is going to run, need to be exploited as much as possible. Low level optimization is an extremely important aspect of interactive ray tracing. Given the choice of hardware, in Chapter III.2 we show which issues should be considered to optimize the ray tracing process. We would also like to refer to Siggraph 2001 course 13 which contains a more elaborate discussion of these issues [25].

After optimizing the ray tracer, one may want to extent the feature set of the ray tracer. Walking through a scene at interactive rates is useful, but being able to interact with it is even more useful. While ray tracing is traditionally good at rendering static scenes, we show that it is possible to render animated scenes, or even interact with the scene in real-time. This requires some basic modifications to the ray tracing algorithm (in particular the spatial subdivisions need a tweak) that are easy to implement, incur a small performance penalty, but allow animation to take place interactively and give the user the ability to manoeuvre objects. Chapter III.3 shows the details.

A second wish users will have once they've implemented their interactive ray tracer, is that it would be great if it could deal with higher scene complexity and/or larger image sizes. As the ray tracer is already fully optimized, it is unlikely that further low-

level tweaks are going to significantly improve performance. A high-level optimization is required. One approach would be to try and implement techniques that allow results from previous frames to be reused. There are a number of techniques in existence that can be employed. Reusing previous results involves displaying pixels at a higher frame-rate than new pixels can be produced. Storing pixels in a 3D point cloud and reprojecting the points for each new frame is one such method. Several choices need to be made to optimize such reprojection techniques, including strategies to decide which pixels to render for the next frame, whether frames are going to be used in the first place or whether the algorithm is to operate in asynchronous mode (frameless rendering), and whether the point reprojection is also going to operate in parallel or not. Chapter III.4 presents an implementation and discusses its merits and weaknesses.

III.1.1 Hardware considerations

As discussed briefly above, the choice of hardware is important. Ray tracing can be implemented to perform interactively, but the first thing that appears to be required is some form of parallelism. Without it, the options are extremely limited. Assuming that parallel hardware is available, the next question to ask is whether this hardware is suitable or not.

We limit the discussion about what hardware to use to general purpose hardware and leave dedicated hardware solutions to others. General purpose hardware can generally be grouped into two broad categories: distributed and shared memory architectures. Putting this distinction before any others is no coincidence: its impact on the performance of our interactive ray tracer will be profound. Whereas distributed memory architectures (either as parallel machines or as networks of workstations) can be cheap, they do exhibit a few disadvantages that are more or less important depending on the application. The first of these is the fact that fetching a data item from a remote processor's local memory can be a couple of orders of magnitude slower than a local memory access, especially on clusters of workstations. Rendering large scenes using distributed memory architectures would therefore require the scene database to be replicated. Even then, after each frame is rendered, the pixel data needs to be transferred to a framebuffer which is typically not distributed. It therefore requires communication of all pixel data to one processor which holds the frame buffer. Given the relatively high latency and low throughput which one can expect from distributed memory systems, this is unlikely to be a fruitful approach. One solution would be to research the use of distributed frame buffers, which may make distributed memory interactive ray tracing feasible.

As far as we are aware, currently there are no distributed memory systems with distributed framebuffers commercially available. This is likely to change in the future, though. When this happens, the above arguments will have to be re-evaluated.

As a result, a solution with the necessary latency and throughput figures, is currently available by using shared memory systems. Although presented to the user as containing just one contiguous block of memory, these machines usually employ caching mechanisms to further speed-up memory accesses. Hence, there is still a distinction between local and remote memory accesses, albeit that the time figures for these accesses are much better than for distributed memory machines.

Another advantage of such systems is that in theory, programming is much simpler than programming on a distributed memory architecture. On shared memory architectures, a sequential program may run unaltered. If the performance is unsatisfactory, the inner loops can be parallelized incrementally. There is no need to parallelize the full code in one go. After each incremental step, the program could be tested for correctness and performance. Also, within the shared memory paradigm, loops can be parallelized without knowledge about where the data resides that is accessed within the loop. These are facilities that make programming on a shared-memory architecture relatively straightforward.

In practice, however, ease of programming is strongly related to the level of optimization desired. As ray tracing is extremely computationally intensive, we cannot afford to waste computer cycles and we therefore have to pay careful attention to performance optimization. Unfortunately, this negatively affects ease of programming. On the hardware architecture described in Appendix IIIA, the memory is physically distributed, although it is presented to the programmer as one block of contiguous memory, having a single address space. This means that it is faster to access some parts of the memory than it is to access other parts of the memory. Thus, there is a performance gain to be had from anticipating where data is located. Although the parallelized program will work without such knowledge, its performance will not be optimal.

We have now stated our case for shared memory architectures. It is motivated by the nature of our application, which requires extremely fast communication, just to get the pixels to the screen. For other applications, the choice of hardware may be different and for coarse grain applications, it may be extremely cost effective to choose distributed memory machines. If, for example, the lighting simulation is to include diffuse interreflection, the cost of rendering a single frame may be prohibitively expensive. Using a distributed memory system is then an attractive alternative to turn a nearly intractable problem into a feasible one.

Whereas we've so far considered memory architectures in general, the interactive ray tracer described in this course is actually implemented on a particular shared memory system: a Silicon Graphics Origin 2000. In Appendix IIIA we describe this architecture in some detail, as knowledge of this hardware allows our code to be optimized further at a later stage. Note also that the results presented in the following chapters were obtained on a faster Silicon Graphics Origin 3800. From a programmer's point of view, the differences are small, although the algorithms run around 1.7 times faster than on the Origin 2000.

III.2 Interactive ray tracing

In this section we focus on the basics of interactive ray tracing. The general approach is to keep the algorithm as simple as possible. In practice this means that a master-slave configuration is employed. The slaves produce pixels, which are displayed by a single display thread, which also doles out new tasks.

The display thread uses double buffering. While one set of pixels is being displayed, another pixel array is filled by the renderers. Once the new frame is complete, the display thread swaps the two pixel arrays and displays the new frame.

III.2.1 Organization of the algorithm

For the renderers, a task consists of rendering a number of neighboring pixels. The task size is chosen such that the pixel data in a single task fits on a cache line. Because the pixels are close to each other in screen space, their associated rays are likely to intersect the same objects, thus improving cache performance by minimizing the amount of object data that needs to be fetched from memory.

New tasks are doled out starting from the top of the image. Tasks near the bottom of the image, which are computed later during the current frame, consist of fewer pixels than the ones near the top. This ensures a good load balance, while also keeping the total number of tasks executed per frame small. The latter is important, because there is inevitably some overhead associated with each task and fewer tasks may improve cache coherence.

III.2.2 Frameless rendering

An alternative strategy is to abandon the concept of frames and switch to frameless rendering [4, 8, 19, 33], where pixels are displayed as soon as they are rendered. Here, a task consists of a single ray, which for each processor is chosen randomly. Although the latency between computing a pixel and displaying it is reduced, the time required to render the equivalent of a complete frame is increased. Thus, from a performance point of view, this is a less successful organization of the algorithm. Cache coherence is preserved less well and extra overhead is incurred. However, when scene complexity increases, frame-based rendering would slow down to the point where the frame rate would be too slow for practical purposes. Although frameless rendering is in fact slightly slower than frame-based rendering, the effect of randomizing the order in which pixels are traced, and splatting the pixels on screen as soon as results become available, gives the impression of smooth movement long after frame-based rendering ceased to be effective. In Chapter III.4 this technique is compared with other mechanisms to increase the production of pixels in an interactive ray tracing context. This technique is discussed further in Section III.2.4.

III.2.3 Tracing a single ray

Tracing each ray is optimized in the usual way by employing a spatial subdivision. While the exact type of spatial subdivision is not fixed, in our system it is either a grid,

which may be nested to accommodate local scene complexity, or a hierarchical spatial subdivision such as a bounding volume hierarchy or an octree. More elaborate schemes are possible, but we have obtained good results with the above spatial subdivisions. We will therefore not go into great detail describing them, with the exception of the discussion in Chapter III.3 where object animation has a direct impact on the organization of any spatial subdivision.

While the above discusses some important design decisions, a simple screen space subdivision combined with a standard spatial subdivision does not explain why this algorithm is capable of achieving interactive rates. The performance gain for this type of rendering on SGI Origin 2000 and 3800 architectures lies in the low level optimization employed. This is the topic of the following sections.

III.2.4 Low level optimization

In this section, we discuss simple ways to optimize data structures. Efficiency can be gained by optimizing data access patterns as well as by ensuring that data structures fit on a single cache line as much as possible.

Ray tracing in general can be quite efficient in terms of data access patterns. Neighboring primary rays have a reasonable probability of intersecting the same objects. Hence, if neighboring rays are traced one after the other, chances are that the objects fetched for the first ray, still reside in the local cache when tracing the second ray. Such data coherence can for example be exploited by tiling the screen into sufficiently large tiles and assign these tiles as tasks to processors. Although this improves cache coherence, such tiling approaches reduce the number of tasks available per frame. By making tiles too large, load balancing issues may appear. In the interactive ray tracer a trade-off is reached by assigning large tiles at the start of the frame and reducing tile sizes when the frame progresses. This ensures high cache efficiency throughout most of the computations associated with a frame, while at the same time the workload is well balanced since the processors are likely to finish their work for the current frame roughly at the same time.

In the case of frameless rendering, results are displayed as soon as they become available. This is in contrast with standard frame-based rendering where the results of the current frame are displayed as soon as the complete frame is finished. Frameless rendering therefore has the advantage that most of the results are displayed quicker than in frame-based rendering of the same scene. However, to achieve fluidity, task assignment will have to be randomized. Because of this, cache coherence is effectively destroyed and the time taken to render the equivalent of a complete frame would therefore take longer than it would take to compute a single frame using frame-based rendering. One advantage of frameless rendering is that for slightly more complex scenes, the animation or walk-through is perceived more fluid and responsive where in frame-based rendering the update rate of the display would become distractingly slow. Frameless rendering is compared to other optimization strategies in Chapter III.4.

Assuming frame-based rendering for now, the data structures employed can be optimized by noting that on the Origin 2000, a cache line is 128 bytes long. When a cache invalidation occurs, a whole cache line is swapped out, to be replaced by a new one. If two different data items reside on a single cache line, and one gets invalidated, by

necessity the other data item will also be removed from the cache. This may incur a performance penalty that can be fairly easily avoided.

Many data structures routinely employed for ray tracing can be cast in the form of an array of structs. If this is the case, then the size of each of these structs should be considered. In general, it is recommended that the size of these structures is as small as possible. One could ensure that cache lines are completely filled by making the size of these structures a power of two. Those data structures that are crucial to performance can be artificially increased in size to be multiples of 128 bytes by adding a character array with the required size to make the whole struct 128 bytes long. As a result, these padded data items can never be removed from the cache because another item on the same cache line is invalidated. Such padding is therefore an important weapon in the arsenal of the programmer, but should nonetheless be used with care. Before and after padding such data structures, one should profile the result to establish if the effect was beneficial or not.

Finally, source code in general can be optimised by minimising the number of branch instructions that occur. This is especially the case for branch instructions that are located within inner loops. With the advent of branch prediction facilities, branching is less detrimental to performance than it once was, but still should be considered an opportunity for optimization.

III.2.5 Profiling

Other non-trivial reasons for suboptimal performance may be determined using profiling and can result from unoptimized code, unoptimized memory access patterns and perhaps even a lack of understanding of the underlying architecture. As it can be difficult to predict where unnecessary performance penalties are incurred, a thorough analysis of the implementation can be useful. Rather than giving a detailed analysis of the profiling steps undertaken to make the interactive ray tracer faster, we would like to refer to appendices IIIA and IIIB which describe the SGI Origin 2000 architecture in some detail and provide a tutorial on profiling on these machines.

III.2.6 Memory and CPU placement

As it is realized that even in architectures that are presented to the programmer as shared memory configurations, the memory is in fact physically distributed, performance can be gained by matching memory with processors. Normally, memory is allocated using system calls, which means that the operating system decides which part of the address space to use for each particular memory allocation. Therefore, unhappy memory allocations may occur where the process requesting the memory is physically located far away from the memory.

If it is known in advance that a particular data structure will be predominantly accessed by a particular process, it may be advantageous to actively place the memory and the processors on fixed locations, rather than leaving this placement up to the operating system. For such an undertaking to be successful, knowledge of the machine's physical architecture is necessary. For the Silicon Graphics Origin 2000, the architecture is described in Appendix IIIA. Briefly summarizing: the machine consists of

nodes which are interconnected using hubs. Each node contains a block of memory and two processors (four on the Origin 3800). Each processor has a separate primary and secondary cache. Memory access is therefore fastest if the data item requested is located in a local cache. Second fastest are memory accesses within the node. Memory that is located with other nodes are slowest to access.

If an algorithm causes many cache misses and if profiling has revealed that this is not easily fixable, then mapping the data structures that cause the cache misses to specific nodes may improve performance. One way to achieve this is to use **mmap** to place memory on a node close to the requesting process and can be achieved using:

```
devzero_fd   = open ("/dev/zero", O_RDWR);
local_memory = (localmemory_t *) mmap (0, sizeof (localmemory_t),
                                       PROT_READ | PROT_WRITE,
                                       MAP_PRIVATE | MAP_LOCAL,
                                       devzero_fd, 0);
```

For further information, we would like to refer to the *mmap* manual page. A less involved method for placing memory with specific nodes is described in the *dplace* manual page.

If the mmap mechanism is used for memory placement, it may be advantageous to also pin processes to specific CPU's. This can be achieved with the **sysmp** and **sproc** commands (see the *sysmp* and *sproc* manual pages). To spawn a new process on a specific processor, the followinf statements may be used:

```
sysmp (MP_MUSTRUN, cpu);
pid = sproc (render_process, PR_SALL);
```

III.3 Animation and interactive ray tracing

A fully optimized ray tracer which allows interactive walk-throughs is attractive over other real-time rendering algorithms because it allows a large set of effects to be rendered which are more difficult or even impossible to obtain using graphics hardware. In addition, ray tracing scales sub-linearly in the number of objects due to the use of spatial subdivisions. It also scales sub-linearly in the number of pixels rendered, provided cache coherency can be exploited fully.

To make interactive ray tracing more attractive, we have looked into ways to enable objects to be manipulated in real-time ([23], reproduced with permission in Appendix IIIC). In the following we assume that animation paths are not known prior to the rendering, and so updates to the scene need to be achieved in real-time with as little overhead as possible. In addition it is important that the effect of time-varying scenes on the performance of the renderer is as small as possible.

Changing the coordinates of an object in real-time is not particularly difficult to achieve, so we will not address this issue in any detail. However, an object's change in location, size or rotation does imply that after the transformation, the object may occupy a different portion of space. In the absence of a spatial subdivision to speed up the intersection tests, this would not constitute a problem.

However, the current speed of the hardware, combined with the number of computations required to ray trace an image, does not allow us to do away with spatial subdivisions altogether. Additionally, spatial subdivisions are usually built as a pre-process to rendering. The cost of building a spatial subdivision is not negligible. Hence, spatial subdivisions are required to obtain interactive frame-rates, but at the same time they are not flexible enough to accommodate time-varying data.

In this section we describe a simple adaptation to both grid and octree spatial subdivisions which caters for a small number of animated objects. These objects can either be animated according to pre-defined motion splines or they can be picked up by the user and placed elsewhere in the scene. Animating all objects at the same time in a complex scene is not yet possible. It would require rebuilding the entire spatial subdivision for each frame and this is too costly to achieve using current technology. Focusing on just a small number of objects to be animated/manipulated allows the design of spatial subdivisions which can be incrementally updated after each frame.

In the following sub-sections the basic idea is explained (Section III.3.1) and results are shown (Section III.3.2). We would also like to refer to Appendix IIIC which includes a full publication regarding this subject. The results presented in this chapter are obtained using an SGI Origin 3800, while appendix IIIC contains older results using an SGI Origin 2000.

III.3.1 Algorithm

In this section modifications to grid and octree spatial subdivisions are discussed. The octree is a hierarchical extension to the grid. We assume the reader is familiar with these spatial subdivisions [1, 6, 9, 11, 13, 15, 17, 18, 28, 32].

Grid spatial subdivisions for static scenes, without any modifications, are already useful for animated scenes, as traversal costs are low and insertion and deletion of

objects is reasonably straightforward. Insertion and deletion are considered basic operations necessary for the animation of objects. The general approach is to remove an object from the spatial subdivision, modify its coordinates and the re-insert the object into the acceleration structure. Insertion is usually accomplished by mapping the axis-aligned bounding box of an object to the voxels of the grid. The object is inserted into all voxels that overlap with this bounding box. Deletion can be achieved in a similar way.

However, when an object moves outside the extent of the spatial subdivision, the acceleration structure would normally have to be rebuilt. As this is too expensive to perform repeatedly, we propose to logically replicate the grid over space. If an object exceeds the bounds of the grid, the object wraps around before re-insertion. Ray traversal then also wraps around the grid when a boundary is reached. In order to provide a stopping criterion for ray traversal, a logical bounding box is maintained which contains all objects, including the ones that have crossed the original perimeter. As this scheme does not require grid re-computation whenever an object moves far away, the cost of maintaining the spatial subdivision will be substantially lower. On the other hand, because rays now may have to wrap around, more voxels may have to be traversed per ray, which will slightly increase ray traversal time.

During a pre-processing step, the grid is built as usual. We will call the bounding box of the entire scene at start-up the 'physical bounding box'. If during the animation an object moves outside the physical bounding box, either because it is placed by the user in a new location, or its programmed path takes it outside, the logical bounding box is extended to enclose all objects. Initially, the logical bounding box is equal to the physical bounding box. Insertion of an object which lies outside the physical bounding box is accomplished by wrapping the object around within the physical grid, as depicted in Figure III.1 (left).

As the logical bounding box may be larger than the physical bounding box, ray traversal now starts at the extended bounding box and ends if an intersection is found or if the ray leaves the logical bounding box. In the example in Figure III.1 (right), the ray pointing to the sphere starts within a logical voxel, voxel (0, -2), which is mapped to physical voxel (0, 2). The logical coordinates of the sphere are checked and found to be outside of the currently traversed voxel and thus no intersection test is necessary. The ray then progresses to physical voxel (1, 2). For the same reason, no intersection with the sphere is computed again. Traversal then continues until the sphere is intersected in logical voxel (4, 2), which maps to physical voxel (0, 2).

Objects that are outside the physical grid are tagged, so that in the above example, when the ray aimed at the triangle enters voxels (0, 2) and (1, 2), the sphere does not have to be intersected. Similarly, when the ray is outside the physical grid, objects that are within the physical grid need not be intersected. As most objects will initially lie within the physical bounds, and only a few objects typically move away from their original positions, this scheme speeds up traversal considerably for parts of the ray that are outside the physical bounding box.

When the logical bounding box becomes much larger than the physical bounding box, there is a tradeoff between traversal speed (which deteriorates for large logical bounding boxes) and the cost of rebuilding the grid. In our implementation, the grid is rebuilt when the length of the diagonals of the physical and logical bounding boxes

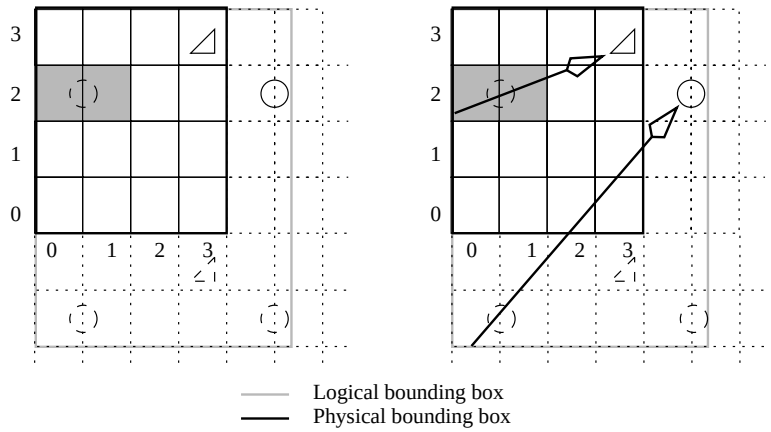


Figure III.1: Grid insertion (left). The sphere has moved outside the physical grid, now overlapping with voxels (4, 2) and (5, 2). Therefore, the object is inserted at the location of the shaded voxels. The logical bounding box is extended to include the newly moved object. Right: ray traversal through extended grid. The solid lines are the actual objects whereas the dashed lines indicate voxels which contain objects whose actual extents are not contained in that voxel.

differ by a factor of two. This heuristic aims to provide a trade-off between traversal speed and the frequency with which the spatial subdivision needs to be re-generated.

Hence, there is a hierarchy of operations that can be performed on grids. For small to moderate expansions of the scene, wrapping both rays and objects is relatively quick without incurring too high a traversal cost. For larger expansions, rebuilding the grid will become a more viable option.

This grid implementation shares the advantages of simplicity and cheap traversal with commonly used grid implementations. However, it adds the possibility of increasing the size of the scene without having to completely rebuild the grid every time there is a small change in scene extent.

The cost of deleting and inserting a single object is not constant and depends largely on the size of the object relative to the size of the scene. The size of an object relative to each voxel in a grid influences how many voxels will contain that object. This in turn negatively affects insertion and deletion times. Hence, it would make sense to find a spatial subdivision whereby the voxels can have different sizes. If this is accomplished, then insertion and deletion of objects can be made independent of their sizes and can therefore be executed in constant time. Such spatial subdivisions are not new and are known as hierarchical spatial subdivisions. Octrees, bintrees and hierarchical grids are all examples of hierarchical spatial subdivisions. However, normally such spatial subdivisions store all their objects in leaf nodes and would therefore still incur non-constant insertion and deletion costs. We extend the use of hierarchical grids in such a way that objects can also reside in intermediary nodes or even in the root node for objects that are nearly as big as the entire scene.

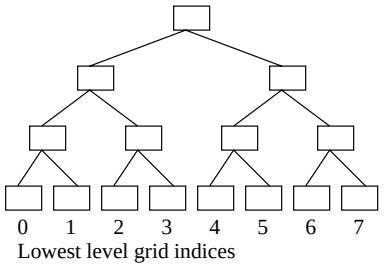
Because such a structure should also be able to deal with expanding scenes, our

efforts were directed towards constructing a hierarchy of grids (similar to Sung [28]), thereby extending the functionality of the grid structure presented in the previous section. Effectively, the proposed method constitutes a balanced octree.

Object insertion now proceeds similarly to grid insertion, except that the grid level needs to be determined before insertion. This is accomplished by comparing the size of the object in relation to the size of the scene. A simple heuristic is to determine the grid level from the diagonals of the two bounding boxes. Specifically, the length of the grid's diagonal is divided by the length of the object's diagonal, the result determining the grid level. Insertion and deletion progresses as explained above.

The gain of better control over insertion time is offset by a slightly more complicated traversal algorithm. Hierarchical grid traversal is effectively the same as grid traversal with the following modifications. Traversal always starts at a leaf node which may first be mapped to a physical leaf node as described earlier in this section. The ray is intersected with this voxel and all its parents until the root node is reached. This is necessary because objects at all levels in the hierarchy may occupy the same space as the currently traversed leaf node. If an intersection is found within the space of the leaf node, then traversal is finished. If not, the next leaf node is selected and the process is repeated.

This traversal scheme is wasteful because the same parent nodes may be repeatedly traversed for the same ray. To combat this problem, note that common ancestors of the current leaf node and the previously intersected leaf node, need not be traversed again (Figure III.2). If the ray direction is positive, the current voxel's number can be used to derive the number of levels to go up in the tree to find the common ancestor between the current and the previously visited voxel. For negative ray directions, the previously visited voxel's number is used instead. Finding the common ancestor is achieved using simple bit manipulation, as detailed in Figure III.3.



Grid Index	Levels between leaf and previously checked common ancestor
0	4 (root)
1	1
2	2
3	1
4	3
5	1
6	2
7	1

Figure III.2: Hierarchical grid traversal. Assuming that ray traversal starts at node 0 and goes in positive direction, then after each step, the common ancestor is found *n* levels above the leaf node as indicated in the table.

As the highest levels of the grid may not contain any objects, ascending all the way to the highest level in the grid is not always necessary. Ascending the tree for a particular leaf node can stop when the largest voxel containing objects is visited.

This hierarchical grid structure has the following features. The traversal is only marginally more complex than standard grid traversal. In addition, wrapping of objects

```
bitmask = (raydir_x > 0) ? x : x + 1
forall levels in hierarchical grid
{
  cell = hgrid[level][x>>level][y>>level][z>>level]
  forall objects in cell
    intersect(ray, object)
  if (bitmask & 1)
    return
  bitmask >>= 1
}
```

Figure III.3: Hierarchical grid traversal algorithm in C-like pseudo-code. The bitmask is set assuming that the last step was along the x-axis.

in the face of expanding scenes is still possible. If all objects are the same size, this algorithm effectively defaults to grid traversal. Insertion and deletion times are much better controlled than for the interactive grid[1].

III.3.2 Results

The grid and hierarchical grid spatial subdivisions were implemented using an interactive ray tracer [19], which runs on an SGI Origin 3800 with 32 processors and 16GB of main memory[2]. Each processor is an R12k running at 400Mhz and manages an 8MB secondary cache. We have chosen to use 30 processors for rendering and one extra thread to take care of user input, displaying the frames, and also for updating and rebuilding the spatial subdivision when necessary (one processor remained unused by our application to allow for system processes to run smoothly). The reason to include the scene update routines with the display thread is that querying the keyboard and displaying the images takes very little time. The remainder of the time to calculate a frame could therefore easily be spent animating objects. In addition, it is important that the scene updates are completed within the time to compute a new frame, as longer update times would either cause delays or result in jerky movement of objects. As the frame rate depends on both the scene complexity and the number of processors that participate in the calculation, the time to update the scene is dependent on both of these parameters.

For evaluation purposes, two test scenes were used. In each scene, a number of objects were animated using pre-programmed motion paths. The scenes as they are at start-up are depicted in Figure III.21 (top, Appendix IIIC). An example frame taken during the animation is given for each scene in Figure III.21 (bottom, Appendix IIIC). All images were rendered at a resolution of 512^2 pixels.

Traversal performance - static scenes

The performance penalty incurred by the new grid and hierarchical grid implementations are assessed by comparing these with a standard grid implementation. The stan-

[1]Note that this also obviates the need for mailbox systems to avoid redundant intersection tests.

[2]Note that the original work, presented in Appendix IIIC, reported results obtained on a slower Origin 2000.

dard grid data structure consists of a single array of object pointers. This design allows better cache efficiency on the SGI Origin series. Finally, we have also implemented a hierarchical grid with a higher branching factor. Instead of subdividing a voxel into eight children, here nodes are split into 64 children (4 along each axis).

From here on we will refer to the new grid implementation as ‘interactive grid’ to distinguish between the two grid traversal algorithms. As all these spatial subdivision methods have a user defined parameter to set the resolution (voxels along one axis and maximum number of grid levels, respectively), various settings are evaluated. The overall performance is given in Figure III.4 and is measured in frames per second.

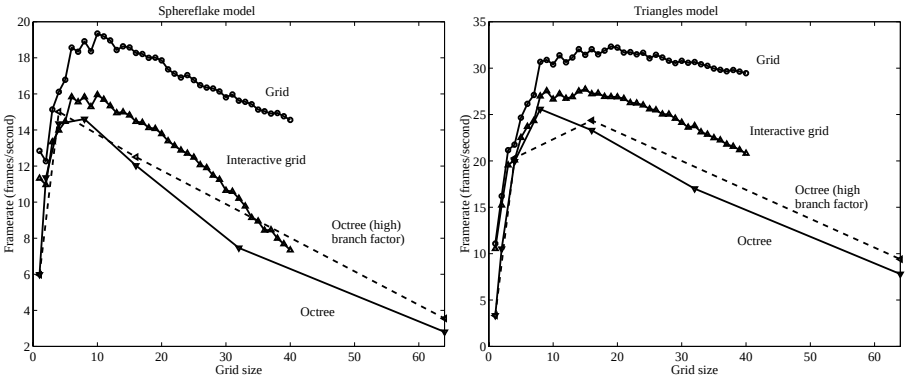


Figure III.4: Performance (in frames per second) for the grid, the interactive grid and the hierarchical grids for two static scenes.

The extra flexibility gained by both the interactive grid and hierarchical grid implementations results in a somewhat slower frame rate. This is according to expectation, as the traversal algorithm is a little more complex and the Origin’s cache structure cannot be exploited as well with either of the new grid structures. The graphs in Figure III.4 should be compared to our previous results given in Figure III.19 in Appendix IIIC. For the hierarchical grid with the higher branching factor, the observed frame rates are very similar to the hierarchical grid.

Object update rate - dynamic scenes

The object update rates were slightly better for the sphereflake and triangle scenes, because the size differences between the objects matches this acceleration structure better than both the interactive grid and the hierarchical grid.

The non-zero cost of updating the scene effectively limits the number of objects that can be animated within the time-span of a single frame. However, for both scenes, this limit was not reached. For each of these tests, the hierarchical grid is more efficiently updated than the interactive grid, which confirms its usefulness.

The size difference between different objects should cause the update efficiency to be variable for the interactive grid, while remaining relatively constant for the hierarchical grid. In order to demonstrate this effect, both the ground plane and one of the

triangles in the triangle scene was interactively repositioned during rendering. Similarly, in the sphereflake scene one of the large spheres and one of the small spheres were interactively manipulated. The update rates for different size parameters for both the interactive grid and the hierarchical grid, are presented in Figure III.5. Comparing the grid size of 16 for the interactive grid with the size parameter of 4 for the interactive grid in this figure, shows that for similar numbers of voxels (at the deepest level of the hierarchical grid) along each axis, the update rate varies much more dependent on object size for the interactive grid than for the hierarchical grid. Hence, the hierarchical grid copes much better with objects of different sizes than the interactive grid. Dependent on the number of voxels in the grid, there is one to two orders of magnitude difference between inserting a large and a small object. For larger grid sizes, the update time for the ground plane of the triangles scene is roughly half a frame. This leads to visible artifacts when using the interactive grid, as during the update the processors that are rendering the next frame temporarily cannot intersect this object (it is simply taken out of the spatial subdivision). In practice, the hierarchical grid implementation does not show this disadvantage.

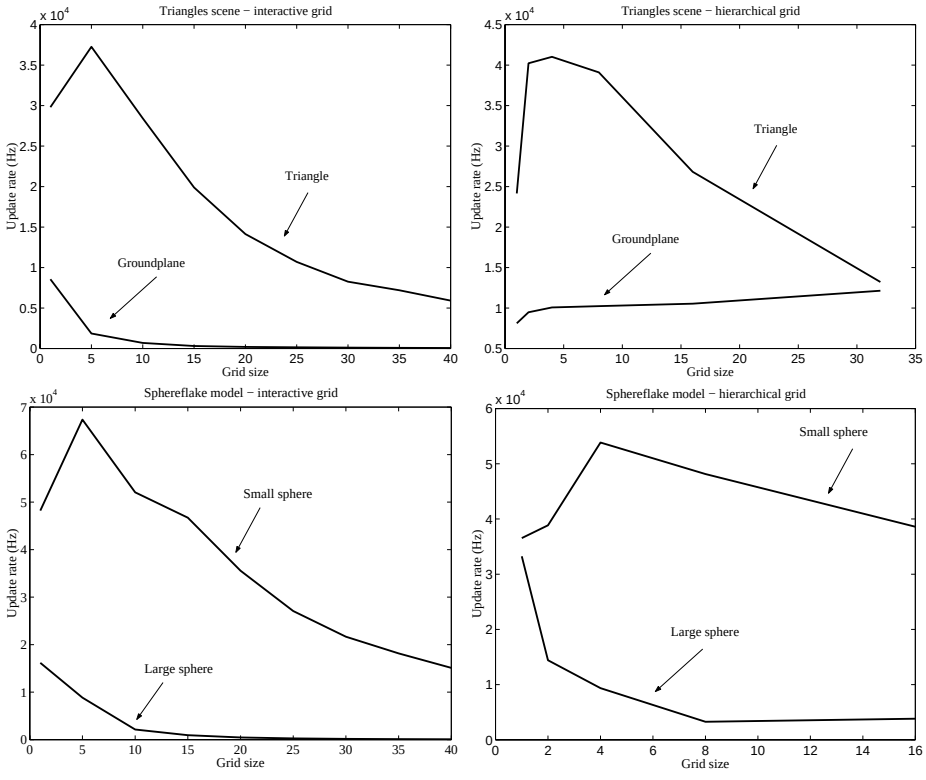


Figure III.5: Update rate as function of grid size for the interactive and hierarchical grids.. We compare the update rates for a small as well as a large object in both the triangles model (top) and the sphereflake model (bottom).

The time to rebuild a spatial subdivision from scratch is expected to be considerably higher than the cost of re-inserting a small number of objects. For the triangles scene, where 200 out of 201 objects were animated, the update rate was still a factor of two faster than the cost of completely rebuilding the spatial subdivision. This was true for both the interactive grid and the hierarchical grid. A factor of two was also found for the animation of 81 spheres in the sphereflake scene. When animating only 9 objects in this scene, the difference was a factor of 10 in favor of updating. We believe that the performance difference between rebuilding the acceleration structure and updating all objects is largely due to the cost of memory allocation, which occurs when rebuilding. The cost of rebuilding the spatial subdivision will become prohibitive when much larger scenes are rendered.

Traversal cost - dynamic scenes

In the case of expanding scenes, the logical bounding box will become larger than the physical bounding box. The number of voxels that are traversed per ray will therefore on average increase. This is the case in the triangles scene. The variation over time of the frame rate is given in Figure III.6. In this example, the objects are first stationary. At some point the animation starts and the frame rate drops because the scene immediately starts expanding. For the sphereflake scene, the animated objects do not cause the scene to expand, and therefore no drop in framerate is observed.

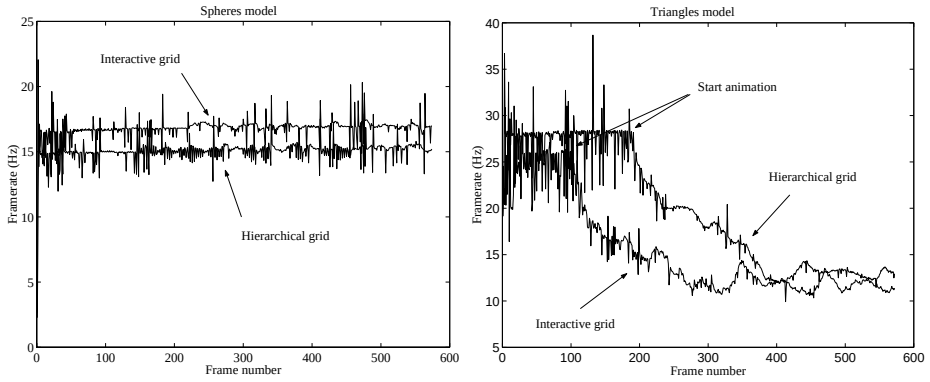


Figure III.6: Framerate as function of time for the triangles scene and the sphereflake scene. Note that the sphereflake scene does not expand over time and therefore starting the animation does not appreciably affect the framerate.

Animating clusters of objects

For many applications it will be necessary to animate clusters of objects in a coherent manner. For example, if a teapot such as depicted in Figure III.7, needs to be repositioned, it would not make sense to individually move each of its 25,000 individual

polygons. Encapsulating the teapot within its own spatial subdivision will improve rendering time but will not improve insertion and deletion time, as after moving the teapot, all its polygons would still require updating. Here, the use of instancing provides a good solution as only the transformation matrix specifying where the teapot is positioned in space will need to be updated. For this example, updating the spatial subdivision as well as the transformation matrix can be performed around 2400 times per second. The benchmark for this scene resulted in a frame rate of 12.1 fps[3].

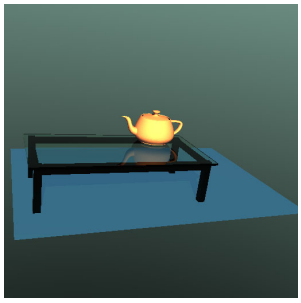


Figure III.7: Example of instancing. Moving the teapot requires a cheap update of a transformation matrix.

Finally, Figure III.22 shows that interactively updating scenes using drag and drop interaction is feasible.

III.3.3 Discussion

When objects are interactively manipulated and animated within a ray tracing application, much of the work that is traditionally performed during a pre-processing step becomes a limiting factor. Especially spatial subdivisions which are normally built once before the computation starts, do not exhibit the flexibility that is required for animation. The insertion and deletion costs can be both unpredictable and variable. We have argued that for a small cost in traversal performance flexibility can be obtained and insertion and deletion of objects can be performed in a well controlled amount of time.

By logically extending the (hierarchical) grids into space, these spatial subdivisions deal with expanding scenes rather naturally. For modest expansions, this does not significantly alter the frame rate. When the scenes expand a great deal, rebuilding the entire spatial subdivision may become necessary. For large scenes this may involve a temporary drop in frame rate. For applications where this is unacceptable, it would be advisable to perform the rebuilding within a separate thread (rather than the display thread) and use double buffering of the scene to minimize the impact on the rendering threads.

[3]Result obtained on a 32-node SGI Origin 2000.

III.4 Sample reuse techniques

As argued in previous chapters, interactive Whitted-style ray tracing has recently become feasible on high-end parallel machines [16, 19]. However, such systems only maintain interactivity for relatively simple scenes or small image sizes, due to the brute-force nature of these approaches. While keeping the algorithm as simple as possible is an important factor for their succes, reasonably straightforward extensions have been deviced to improve visual appearance for much larger image sizes and scene complexities. After a brief overview, one such system is explored further in this chapter.

By reusing samples instead of relying on brute force approaches, the limitations in scene complexity and image size can be overcome. There are several ways to reuse samples. All of them require interpolating between existing samples as the key part of the process. First, rays can be stored along with the color seen along them. The color of new rays can be interpolated from existing rays [3, 14]. Alternatively, the points in 3D where rays strike surfaces can be stored and then woven together as displayable surfaces [24]. This method was designed to display course results by a display processor while new samples are created by a rendering back-end which can consist of one or more renderers. As new results become available to the display processor, the image is refined and redisplayed. Finally, stored points can be directly projected to the screen, and holes can be filled in using image processing heuristics [31]. All techiques that re-use samples rely on the fact that the reprojection step is much cheaper than the generation of new samples and are therefore typically employed in cases where sample generation is too slow for creating interactive results. In the case of Simmons' work, this occurred because the lighting simulation is too complex for interactive display [24]. Walter's point reprojection algorithm is directed towards interactive display of scenes that are too complex to display interactively otherwise.

Another method to increase the interactivity of ray tracing is *frameless rendering* [4, 8, 19, 33]. Here, a master processor farms out single pixel tasks to be traced by the slave processors. The order in which pixels are selected is random or quasi-random. Whenever a renderer finishes tracing its pixel, it is displayed directly. As pixel updates are independent of their display, there is no concept of frames. During camera movements, the display will deteriorate somewhat, which is visually preferable to slow frame-rates in frame-based rendering approaches. It can therefore handle scenes of higher complexity than brute force ray tracing, although no samples are reused.

The main thrust of this chapter is the use of parallelism to increase data reuse and thereby increase allowable scene complexity and image size without affecting perceived update rates. The remainder of this chapter uses the *render cache* of Walter et al. [31] and applies to it the concept of frameless rendering. By distributing this algorithm over many processors we are able to overcome the key bottleneck in the original render cache work. We demonstrate our system on a variety of scenes and image sizes that have been out of reach for previous systems. The work described in this chapter is currently under submission for the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics [22].

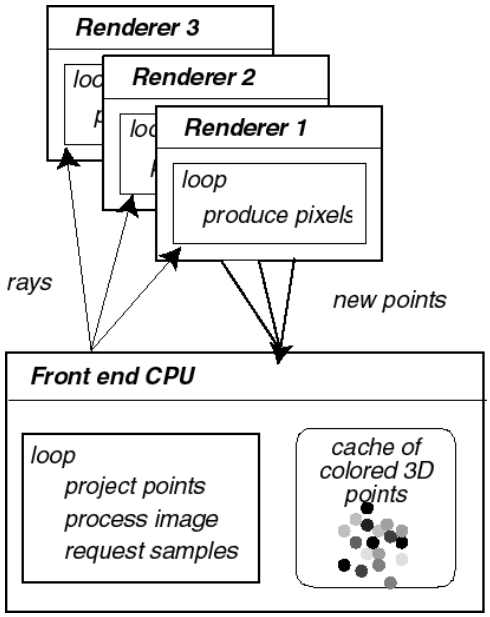


Figure III.8: *The serial render cache algorithm [31].*

III.4.1 Render cache algorithm

The basic idea of the render cache is to save samples in a 3D point cloud, and reproject them when viewing parameters change [31]. New samples are requested all over the screen, with most samples concentrated near depth discontinuities. As new samples are added old samples are eliminated from the point cloud.

The basic process is illustrated in Figure III.8. The front-end CPU handles all tasks other than tracing rays. Its key data structure is the cache of colored 3D points. The front end continuously loops, first projecting all points in the cache into screen space. This will produce an image with many holes, and the image is processed to fill these holes in. This filling-in process uses sample depths and heuristics to make the processed image look reasonable. The processed image is then displayed on the screen. Finally, the image is examined to find "good" rays to request to improve future images. These new rays are traced by the many CPUs in the "rendering farm". The current frame is completed after the front end receives the results and inserts them into the point cloud.

From a parallel processing point of view, the render cache has the disadvantage of a single expensive display process that needs to feed a number of renderers with sample requests and is also responsible for point reprojection. The display process needs to insert new results into the point cloud, which means that the more renderers are used, the heavier the workload of the display process. Hence, the display process quickly becomes a bottleneck. In addition, the number of points in the point cloud is linear in image size, which means that the reprojection cost is linear in image size.

The render cache was shown to work well on 256x256 images using an SGI Origin 2000 with 250MHz R10k processors. At higher resolutions than 256x256, the front end has too many pixels to reproject to maintain fluidity.

III.4.2 Parallel render cache

Ray tracing is an irregular problem, which means that the time to compute a ray task can vary substantially depending on depth complexity. For this reason it is undesirable to run a parallel ray tracing algorithm synchronously, as this would slow down rendering of each frame to be as slow as the processor which has the most expensive set of tasks. On the other hand, synchronous operation would allow a parallel implementation of the render cache to produce exactly the same artifacts as the original render cache. We have chosen responsiveness and speed of operation over minimization of artifacts by allowing each processor to update the image asynchronously.

Our approach is to distribute the render cache functionality with the key goal of not introducing synchronization, which is analogous to frameless rendering. In our system there will be a number of renderers which will reproject point clouds and render new pixels, thereby removing the bottleneck from the original render cache implementation. Scalability is therefore assured.

We parallelize the render cache by subdividing the screen into a number of tiles. A random permutation of the list of tiles could be distributed over the processors, with each renderer managing its set of tiles independently from all other renderers. Alternatively, a global list of tiles could be maintained with each processor choosing the tile with the highest priority whenever it needs a new task to work on. While the latter option may provide better (dynamic) load balancing, we have opted for the first solution. Load balancing is achieved statically by ensuring that each processor has a sufficiently large list of tiles. The reason for choosing a static load balancing scheme has to do with memory management on the SGI Origin 3800, which is explained in more detail in Section III.4.3 and Appendix IIIA.

Each tile has associated with it a local point cloud and an image plane data structure. The work associated with a tile depends on whether or not camera movement is detected. If the camera is moving, the point cloud is projected onto the tile's local image plane and the results are sent to the display thread for immediate display. No new rays are traced, as this would slow down the system and the perceived smoothness would be affected. This is at the cost of a degradation in image quality, which is deemed more acceptable than a loss of interaction. It is also the only modification we have applied to the render cache concept.

If there is no camera movement, a depth test is performed to select those rays that would improve image quality most. Other heuristics such as an aging scheme applied to the points in the point cloud also aid in selecting appropriate new rays. Newly traced rays are both added to the point cloud and displayed on screen. The point cloud itself does not need to be reprojected.

The renderers each loop over their allotted tiles, executing for each tile in turn the following main components:

1. Clear tile Before points are reprojected, the tile image is cleared.

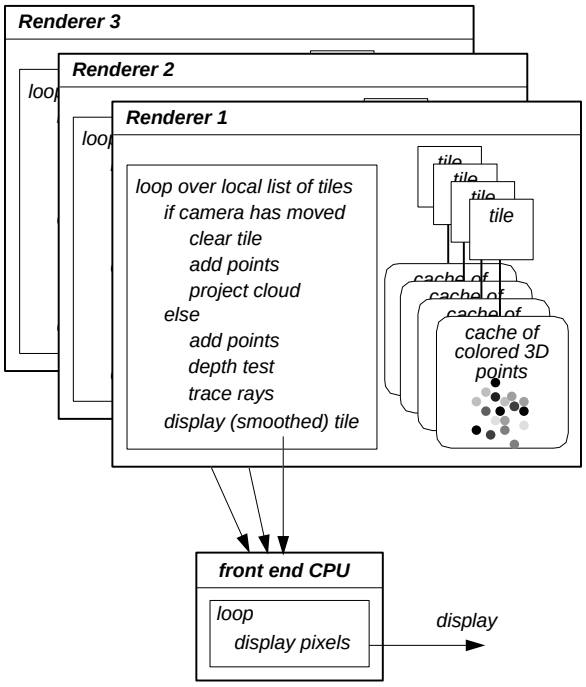


Figure III.9: *The parallel render cache algorithm.*

2. Add points Points that previously belonged to a neighboring tile but have been projected onto the current tile are added to the point cloud.

3. Project point cloud The point cloud is projected onto the tile image. Points that project outside the current tile are temporarily buffered in a data structure that is periodically communicated to the relevant neighboring tiles.

4. Depth test A depth test is performed on the tile image to determine depth discontinuities. This is then used to select new rays to trace.

5. Trace rays The rays selected by the depth test function, are traced and the results added to the local point cloud.

6. Display tile The resulting tile is communicated to the display thread. This function also performs hole-filling to improve the image's visual appearance.

If camera movement has occurred since a tile was last visited, items 1, 2, 3 and 6 in this list are executed for that tile. If the camera was stationary, items 1, 2, 3 and 6 are executed. The algorithm is graphically depicted in Figure III.9

While tiles can be processed largely independently, there are circumstances when interaction between tiles is necessary. This occurs for instance when a point in one tile's point cloud projects to a different tile (due to camera movement). In that case, the point is removed from the local point cloud and is inserted into the point cloud

associated with the tile to which it projects. The more tiles there are, the more often this would occur. This conflicts with the goal of having many tiles for load balancing purposes. In addition, having fewer tiles that are larger causes tile boundaries to be more visible.

As each renderer produces pixels that need to be collated into an image for display on screen, there is still a display process. This display thread only displays pixels and reads the keyboard for user input. Displaying an image is achieved by reading an array of pixels that represents the entire image, and sending this array to the display hardware using OpenGL. When renderers produce pixels, they are buffered in a local data structure, until a sufficient number of pixels has been accumulated for a write into the global array of pixels. This buffering process ensures that memory contention is limited for larger image sizes.

Finally, the algorithm shows similarities with the concept of frameless rendering, in the sense that tiles are updated independently from the display process. If the size of the tiles is small with respect to the image size, the visual effect is like that of frameless rendering. The larger the tile size is chosen, the more the image updating process starts to look like a distributed version of the render cache.

III.4.3 Implementation details

The parallel render cache algorithm is implemented on a 32 processor SGI Origin 3800. While this machine has a 16 GB shared address space, the memory is physically distributed over a total of eight nodes. Each node features four 400 MHz R12k processors and one 2 GB block of memory. In addition each processor has an 8 MB secondary cache. Memory access times are determined by the distance between the processor and the memory that needs to be read or written. The local cache is fastest, followed by the memory associated with a processor's node. If a data item is located at a different node, fetching it may incur a substantial performance penalty.

A second issue to be addressed is that the SGI Origin 3800 may relocate a rendering process with a different processor each time a system call is performed. Whenever this happens, the data that used to be in the local cache is no longer locally available. Cache performance can thus be severely reduced by migrating processes.

These issues can be avoided on the SGI Origin 3800 by actively placing memory near the processes and disallowing process migration. This can, for example, be accomplished using the *dplace* library or the mmap system call (see also Section III.2.6). Associated with each tile in the parallel render cache is a local point cloud data structure and an image data structure which are mapped as close as possible to the process that uses it. Such memory mapping assures that if a cache miss occurs for any of these data structures, the performance penalty will be limited to fetching a data item that is in local memory. As argued above, this is much cheaper than fetching data from remote nodes. For this reason, using a global list of tiles as mentioned in the previous section is less efficient than distributing tiles statically over the available processors.

Carefully choreographing the mapping of processes to processors and their data structures to local memory enhances the algorithm's performance. Cache performance is improved and the number of data fetches from remote locations is minimized.

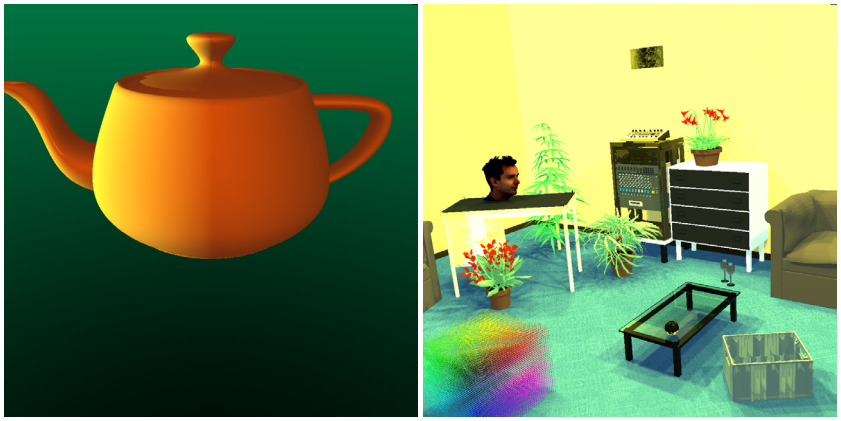


Figure III.10: *Test scenes. The teapot (top) consists of 32 bezier patches, while the room scene consists of 846,563 primitives and 80 point light sources.*

III.4.4 Measuring scalability

The main loops of the renderers consist of a number of distinct steps. During each iteration a subset of these steps is executed dependent on whether camera movement has occurred or not (see Section III.4.2). Standard speed-up measurements would under these circumstances produce unreliable results, since the measured speed-up would depend on how often the user moves the camera. The user cannot be expected to move the camera in exactly the same way for each measurement.

For this reason each of the steps making up the complete algorithm are measured separately. To assess scalability, the time to execute each step is measured, summed over all invocations and processors and subsequently divided by the number of invocations and processors. The result is expressed in events per second per processor, which for a scalable system should be independent of the number of processors employed. Hence, using more processors would then not alter the measurements. In case this measure varies with processor count, scalability is affected.

If the number of events per second per processor drops when adding processors, sublinear scalability is measured, whereas an increase indicates super-linear speed-up for the measured function. Also note that the smaller the number, the more costly the operation will be. Using this measure provides better insight into the behavior of the various parts of the algorithm than a standard scalability computation would give, especially since only a subset of the components of the render cache algorithm is executed during each iteration.

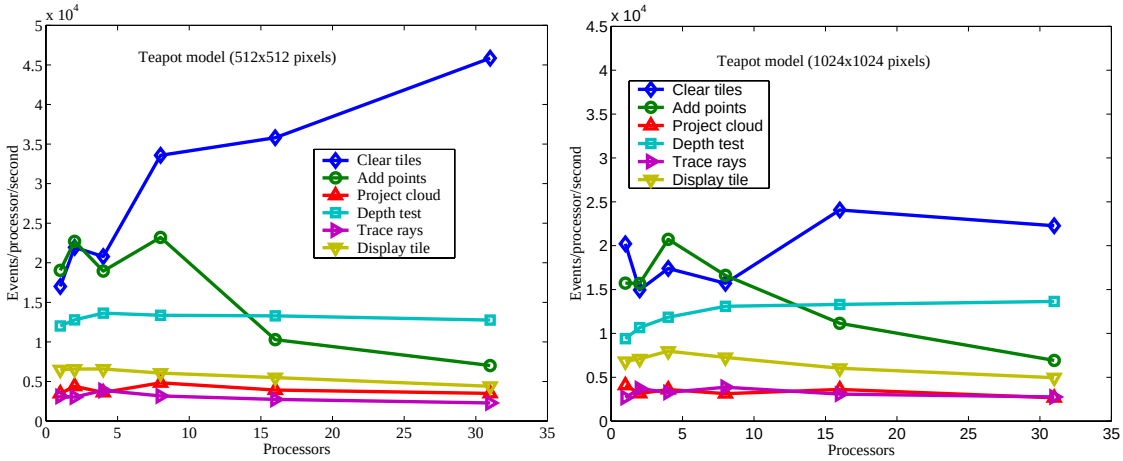


Figure III.11: *Scalability of the render cache components for the teapot scene rendered at* 512^2 *pixels (left) and* 1024^2 *pixels (right). Negative slopes indicate sub-linear scalability, whereas horizontal lines show linear speed-ups.*

III.4.5 Results

Our implementation uses the original render cache code of Walter et al [31][4]. Two test scenes were used: a teapot with 32 bezier patches[5] and one point light source, and a room scene with 846,563 geometric primitives and area light sources approximated by 80 point light sources (Figure III.10). For the teapot scene, the renderer is limited by the point reprojection algorithm, while for the room scene, tracing new rays is the slowest part of the algorithm. The latter scene is of typical complexity in architectural applications and usually cannot be interactively manipulated.

In the following subsection, the different components making up the parallel render cache are evaluated (Section III.4.5), the performance as function of task size is assessed (Section III.4.5) and the parallel render cache is compared with other methods to speed up interactive ray tracing (Section III.4.5).

Parallel render cache evaluation

The results of rendering the teapot and room models on different numbers of processors at a resolution of 512^2 and 1024^2 pixels are depicted in Figures III.11 and III.12.

While most of the components making up the algorithm show horizontal lines in these graphs, meaning that they scale well, the "Clear tiles" and "Add point" components show non-linear behavior. Clearing tiles is a very cheap operation which appears to become cheaper if more processors are used. Because more processors result in each

[4]The original code has since been improved (Walter, personal communication) but we have not ported that improved code. However, we expect that any improvements to the serial code would transfer to our parallel version since the serial code runs essentially as a black box.

[5]These bezier patches are rendered directly using the intersection algorithm from Parker et. al [19].

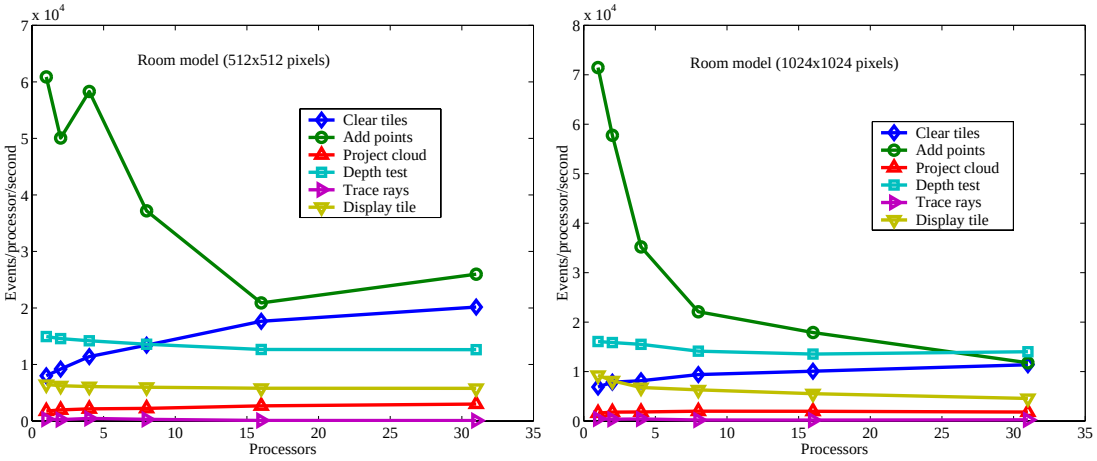


Figure III.12: *Scalability for the room scene, rendered at* 512^2 *pixels (left) and* 1024^2 *pixels (right). Horizontal lines indicate linear scalability, whereas a fall-off means sub-linear scalability.*

processor having to process fewer tiles, this super-linear behavior may be explained by better cache performance. This effect is less pronounced for the 1024^2 pixel renderings, which also points to a cache performance issue as here each processor handles more data.

The "Add point" function scales sub-linearly with the number of processors. Because the total number of tiles was kept constant between runs, this cannot be explained by assuming that different numbers of points project outside their own tile and thus have to be added to neighboring tiles. However, with more processors there is an increased probability that a neighboring tile belongs to a different processor and may therefore reside in memory which is located elsewhere in the machine. Thus projecting a point outside the tile that it used to belong to, may become more expensive for larger numbers of processors. This issue is addressed in the following section.

Note also that despite the poor scalability of "Add points", in absolute terms its cost is rather low, especially for the room model. Hence, the algorithm is bounded by components that scale well (they produce more or less horizontal lines in plots) and therefore the whole distributed render cache algorithm scales well, at least up to 31 processors (see also Section III.4.5). In addition, the display of the results is completely decoupled from the renderers which produce new results and therefore the screen is updated at a rate that is significantly higher than rays can be traced and is also much higher than points can be reprojected. This three-tier system of producing new rays at a low frequency, projecting existing points at an intermediate frequency and displaying the results at a high frequency (on the Origin 3800 at a rate of around 290 frames per second for 512^2 images and 75 frames per second for 1024^2 images, regardless of number of renderers and scene complexity) ensures a smooth display which is perceived as interactive, even if new rays are produced at a rate that would not normally

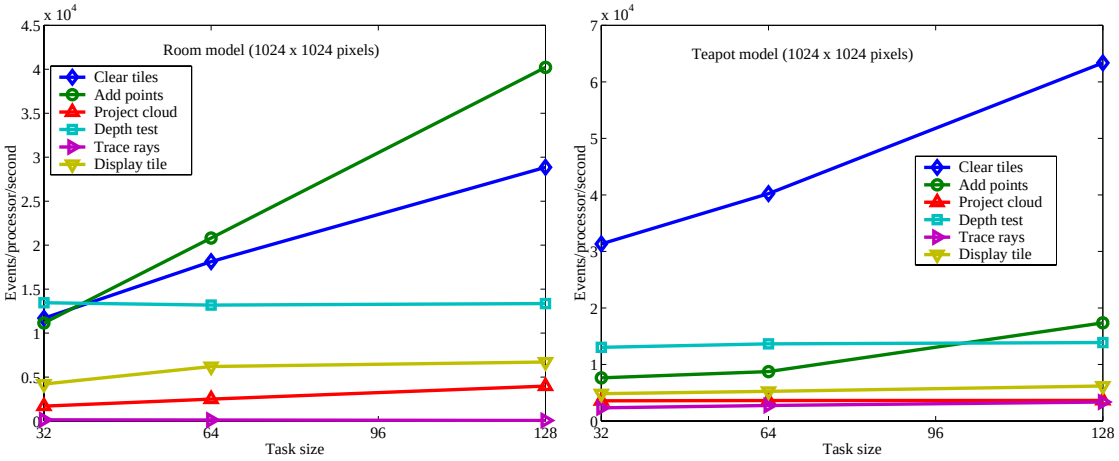


Figure III.13: *Scalability for the room model (left) and teapot scene (right) as function of tile size (32^2, 64^2 and 128^2 pixels per tile). The image size is 1024^2 pixels and for these measurements 31 processors were used. These graphs should be interpreted the same as those in Figures III.11 and III.12.*

allow interactivity.

By abandoning ray tracing altogether during camera movement, the system shows desirable behavior even when fewer than 31 processors are used. For both the room scene and the teapot model, the camera can move smoothly if 4 or more processors are used. During camera movement, the scene deteriorates because no new rays are produced and holes in the point cloud may become visible. During rapid camera movement, tile boundaries may become temporarily visible. After the camera has stopped moving, these artifacts disappear at a rate that is linear in the number of processors employed. We believe that maintaining fluid motion is more important than the temporary introduction of some artifacts, which is why the distributed render cache is organized as described above.

For those who would prefer a more accurate display at the cost of a slower system response, it would be possible to continue tracing rays during camera movement. Although the render cache then behaves differently, the scalability of the separate components, as given in Figures III.11 and III.12, would not change. However, the fluidity of camera movement is destroyed by an amount dependent on scene complexity.

Task size

In section III.4.2 it was argued that the task size, i.e. the size of the tiles, is an important parameter which defines both speed and the occurrence of visual artifacts. The larger the task size, the better artifacts become visible. However, at the same time, the reprojections that cross tile-boundaries are less likely to occur, resulting in higher performance. In Figure III.13 the scalability of the parallel render cache components as function of task size is depicted. Task sizes range from 32^2 pixels to 128^2 pixels

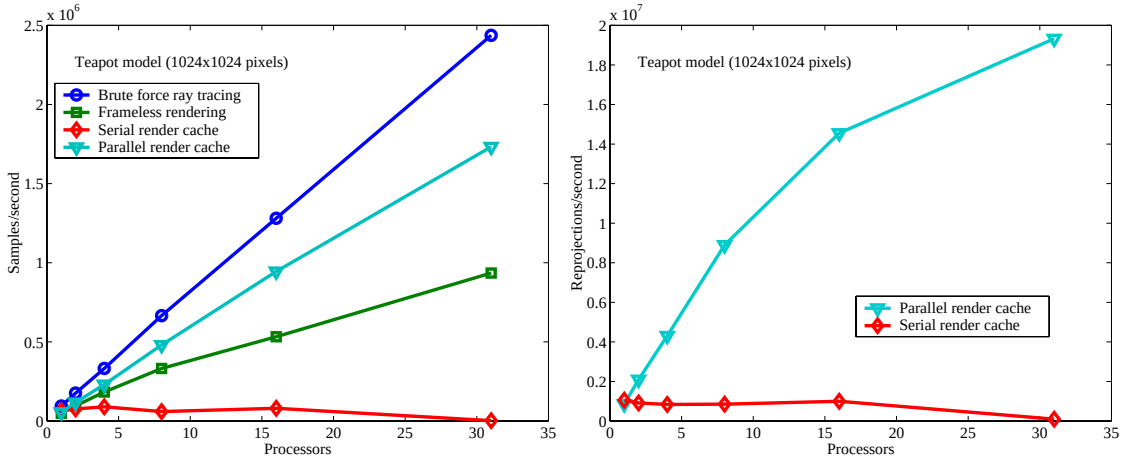


Figure III.14: *Samples per second (left) and point reprojections per second (right) for the teapot model.*

and the measurements were all obtained using 31 processors on 1024^2 images. Larger tile sizes are thus impossible, as the total number of tasks would become smaller than the number of processors. Task sizes smaller than 32^2 pixels resulted in unreasonably slow performance and were therefore left out of the assessment.

As in the previous section, the "Add points" and "Clear tile" components show interesting behavior. As expected, for larger tasks, the "Add points" function becomes cheaper. This is because the total length of the tile boundaries diminishes for larger task sizes, and so the probability of reprojections occurring across tile boundaries is smaller.

The "Clear tile" component also becomes less expensive for larger tiles. Here, we suspect that resetting one large block of memory is less expensive than resetting a number of smaller blocks of memory.

Although Figure III.13 suggests that choosing the largest task size as possible would be appropriate, the artifacts visible for large tiles are more unsettling than for smaller task sizes. Hence, for all other experiments presented in this paper, a task size of 32^2 pixels is used, which is based on an assessment of both artifacts and performance.

Comparison with other speed-up mechanisms

In this section, the parallel render cache is compared with other state-of-the-art rendering techniques. All make use of the interactive ray tracer of Parker et. al. [19], either as a back-end or as the main algorithm. The comparison includes the original render cache algorithm [31], the parallel render cache algorithm as described in this paper, the interactive ray tracer (rtrt) without reprojection techniques and the interactive ray tracer using the frameless rendering concept [19]. In the following we will refer to the original render cache as "serial render cache" to distinguish it from our parallel render

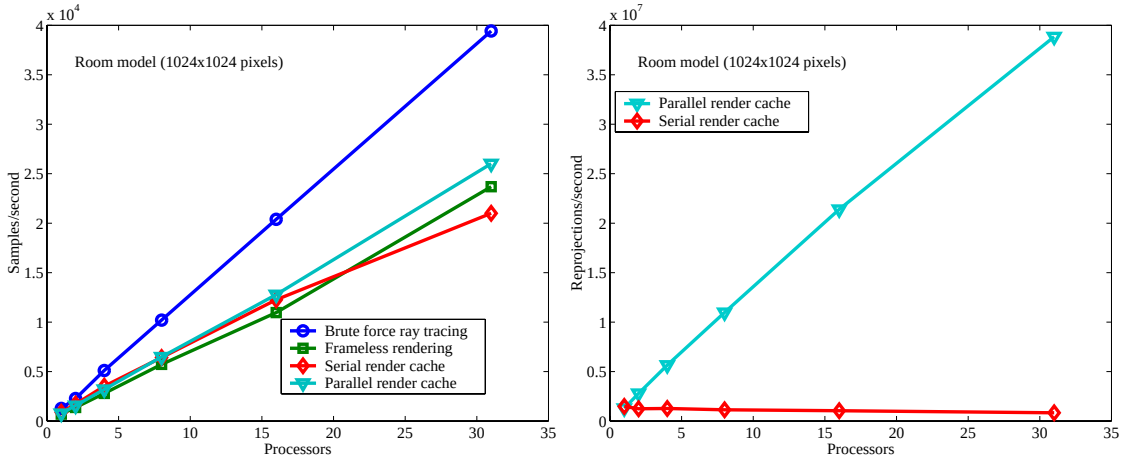


Figure III.15: *Samples per second (left) and point reprojections per second (right) for the room scene.*

cache implementation. All renderings were made using the teapot and room models (Figure III.10) at a resolution of 1024^2 pixels.

The measurements presented in this section consist of the number of new samples produced per second by each of the systems and the number of points reprojected per second (for the two render cache algorithms). These numbers are summed over all processors and should therefore scale with the number of processors employed. The results for the teapot model are given in Figure III.14 and the results for the room model are presented in Figure III.15.

The graphs on the left of these figures show the number of samples generated per second. All the lines are straight, indicating scalable behavior. In these plots, steeper lines are the result of higher efficiency and therefore, the real-time ray tracer would be most efficient, followed by the parallel render cache. The frameless rendering concept looses efficiency because randomizing the order in which pixels are generated destroys cache coherence. The parallel render cache does not suffer from this, since the screen is tiled and tasks are based on tiles. The serial render cache appears to perform well for complex scenes and poorly for simple scenes. For scenes that lack complexity, the point reprojection front-end becomes the bottleneck, especially since the image size chosen causes the point cloud to be quite large. Thus, the render cache front-end needs to reproject a large number of points for each frame and so constitutes a bottleneck.

Although the parallel render cache does not produce as many new pixels as the real-time ray tracer by itself does, this loss of efficiency is compensated by its ability to reproject large numbers of points, as is shown in the plots on the right of Figures III.14 and III.15. The point reprojection component of the parallel render cache shows good scalability, and therefore the goal of parallelizing the render cache algorithm is reached. The point reprojection part of the serial render cache does not scale because it is serial in nature.

III.4.6 Discussion

While it is true that processors get ever faster and multi-processor machines are now capable of real-time ray tracing, scenes are getting more and more complex while at the same time frame sizes still need to increase. Hence, Moore's law is not likely to allow interactive full-screen brute-force ray tracing of highly complex scenes anytime soon.

Interactive manipulation of complex models is still not possible without the use of sophisticated algorithms that can efficiently exploit temporal coherence. The render cache is one such algorithm that can achieve this. However, for it not to become a bottleneck itself, the render cache functionality needs to be distributed over the processors that produce new samples. The resulting algorithm, presented in this paper, shows superior reprojection capabilities that enables smooth camera movement, even in the case where the available processing power is much lower than would be required in a brute force approach. It achieves this for scene complexities and image resolutions that are not feasible using any of the other algorithms mentioned in the previous section.

While smoothness of movement is an important visual cue, our algorithm necessarily produces other artifacts during camera motion. These artifacts are deemed less disturbing than jerky motion and slow response times. The render cache attempts to fill small holes after point reprojection. For larger holes, this may fail and unfilled pixels may either be painted in a fixed color, or can be left unchanged from previous reprojections. Either approach causes artifacts inherent to the algorithm and is present both in the original render cache and in our parallel implementation of it.

The parallel render cache produces additional artifacts due to the tiling scheme employed. During camera movement, tile boundaries may temporarily become visible, because there is some latency between points being reprojected from neighboring tiles and this reprojection becoming visible in the current tile. A further investigation to minimize these artifacts is in order, which we reserve for future work. Currently, the parallel render cache algorithm is well suited for navigation through highly complex scenes to find appropriate camera positions.

It has been shown that even with a relatively modest number of processors, the distributed render cache can produce smooth camera movement at resolutions typically sixteen times higher than the original render cache. The system as presented here scales well up to 31 processors. Its linear behavior suggests that improved performance is likely beyond 31 processors, although if this many processors are available, it would probably become sensible to devote the extra processing power to produce more samples, rather than increase the speed of reprojection.

III.5 Summary and discussion

In this chapter we have explained some of the issues involved in implementing an interactive ray tracer. Using general purpose hardware, currently shared memory architectures appear the most attractive solution, since distributed memory architectures do not allow pixel data to be communicated to the display quickly enough. Distributed memory systems are better suited for coarse grain applications, such as ray tracing with diffuse interreflection [21].

Modern shared memory systems often have a single shared address space, but the memory is still physically distributed. Knowing the architecture for which code is written, may help reduce memory access bottleneck. Mapping memory close to the processes that most often access this memory, can be advantageous, especially if the volume of data that is read from memory is large compared with the local cache size. In that case, cache misses will cause local main memory to be read rather than remote main memory.

Other optimizations that are important include cache optimization. Important data structures should fit on a single cache line. Smaller data structures can be padded to occupy a single cache line. This has the advantage that when a cached item is swapped out because it has not been used for a while, it can not affect other data on the same cache line that is still in use.

While the basic interactive ray tracer described in this chapter is based on such low level optimizations, algorithmic extensions to this basic ray tracer can be employed to allow much larger images to be computed or have scenes of much higher complexity rendered interactively. Point reprojection techniques do not allow rays to be produced quicker, but rely on frame coherence by reusing samples computed for previous frames. For complex scenes, reprojection of existing results is much faster than tracing new rays and so point reprojection allows for smooth movement between camera points for scenes that are too complex for other algorithms to smoothly advance from one camera position to the next.

In this chapter extensions to spatial subdivisions are discussed as well. Normally, spatial acceleration structures are built as a preprocess and are therefore not flexible enough to accomodate interactively placed or moved objects. Extending grid and octree data structures to enable user interaction with the scene interactively have a modest impact on speed of rendering which is acceptable given their ability to allow objects to be moved within and even outside the extent of the scene.

Interactive ray tracing is now feasible and for certain types of application, such as interactive rendering of the visible female data set [20], it is a better choice than other forms of rendering, including z-buffer techniques. With increasing available computational power, the range of applications for interactive ray tracing is likely to grow and become possible on cheaper hardware.

Acknowledgements

We would like to thank Brian Smits, Chuck Hansen and Pete Shirley for their help, support and involvement in the projects that are described in these course notes. In addition, we thank Bruce Walter, Steven Parker and George Drettakis for their render cache source code and John McCorquodale for valuable discussions regarding processor and memory placement issues. Thanks also to Silicon Graphics and Springer Verlag for their friendly cooperation and allowing us to republish appendices IIIA to IIIC. This work was supported by NSF grants 97-96136, 97-31859, 98-18344, 99-77218, 99-78099 and by the DOE AVTC/VIEWS.

Appendices

Appendices IIIA and IIIB contain extracts from the following SGI technical report, which are reproduced with permission from Silicon Graphics.

> David Cortesi and Jeff Fier, *Origin2000 and Onyx2 Performance Tuning and Optimization Guide*, Silicon Graphics technical report, 1998, available from http://techpubs.sgi.com:80/

Appendix IIIC contains the following paper, which is reproduced with permission from Springer Wien - New York:

> Reinhard, E., Smits, B., Hansen, C., *Dynamic Acceleration Structures for Interactive Ray Tracing*. in: Rendering Techniques 2000 (Proceedings of the Eurographics Workshop in Brno, Czech Republic, 2000). pp299-306. Wien - New York: Springer. 2000

IIIA SGI Origin 2000

In this section, the SGI Origin 2000 architecture is described, along with the R10k processor and its facilities for profiling. This will constitute basic knowledge for those who want to write optimized code on this hardware platform. The following subsections are extracted from the SGI document "Origin2000 and Onyx2 Performance Tuning and Optimization Guide" [7].

IIIA.1 The MIPS R10000 processor

The R10000 has a two-level cache hierarchy. Located on the microprocessor chip are a 32 KB, two-way set associative level-1 instruction cache and a 32 KB, two-way set associative, two-way interleaved level-1 (L1) data cache. Off-chip is a two-way set associative, unified (instructions and data) level-2 (L2) cache. This secondary cache may range in size from 512 KB to 16 MB; the size of the secondary cache in the Origin 2000 is 4 MB for 195 MHz systems, and 1 MB for 180 MHz systems. The L1 instruction cache uses a line size of 64 bytes, while the L1 data cache has a line size of 32 bytes. The line size of the L2 cache may be either 64 or 128 bytes; in the Origin 2000 it is 128 bytes. Both the L1 data cache and the L2 unified cache employ a least recently used (LRU) replacement policy for selecting in which set of the cache to place a new cache line.

IIIA.2 Origin 2000 layout

To understand how the Origin2000's scalable shared memory multiprocessor (S2MP) architecture works, we first look at how the building blocks of an Origin system are connected. This is diagrammed in figure III.16. This figure represents Origin systems ranging from 2 to 16 processors. We start by considering the two-processor system in the upper left-hand corner. This is a single Origin 2000 node. It consists of one or two processors, memory, and a device called the hub. The hub is the piece of hardware that carries out the duties that a bus performs in a bus-based system; namely, it manages each processor's access to memory and I/O. This applies to accesses that are local to the node containing the processor, as well as to those that must be satisfied remotely in multi-node systems.

The smallest Origin systems consist of a single node. Larger systems are built by connecting multiple nodes. A two-node system is shown in the upper middle of the figure. Since information flow in and out of a node is controlled by the hub, connecting two nodes means connecting their hubs. In a two-node system this simply means wiring the two hubs together. The bandwidth to local memory in a two-node system is double that in a one-node system: the hub on each of the two nodes can access its local memory independently of the other. Access to memory on the remote node is a bit more costly than access to local memory since the request must be handled by both hubs. A hub determines whether a memory request is local or remote based on the physical address of the data accessed.

When there are more than two nodes in a system, their hubs cannot simply be wired together. In this case, additional hardware is required to control information

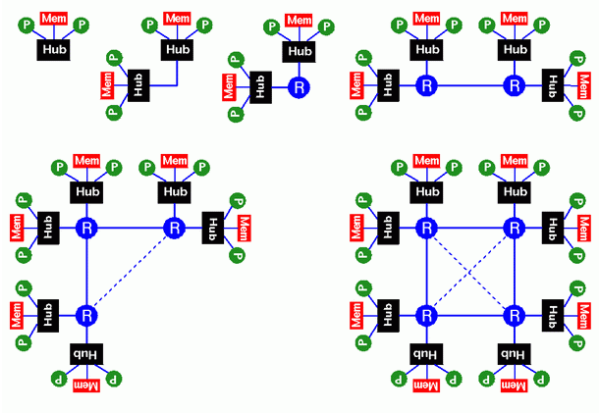


Figure III.16: Building blocks of the SGI Origin 2000.

flow between the multiple hubs. The hardware used for this in Origin systems is called a router. A router has six ports, so it may be connected to up to six hubs or other routers. In a two-node system, one may employ a router to connect the two hubs rather than wiring them directly together; this is shown adjacent to the other two-node configuration in the figure. These two different configurations behave identically, but because of the router in the second configuration, information flow between the two hubs takes a little more time. The advantage, though, of the configuration with the router is that it may be used as a basic building block from which to construct larger systems.

In the upper right corner of the figure, a four-node — or, equivalently, eight-processor — system is shown. It is constructed from two of the two-node-with-router building blocks. Here, the connection between the two routers allows information to flow and, hence, the sharing of memory between any pair of hubs in the system. Since a router has six ports, it is possible to connect all four nodes to just one router, and this one-router configuration can be used for small systems. That is a special case, and in general the two-router implementation is used since it conveniently scales to larger systems.

Two such larger systems are shown on the lower half of the figure; these are 12- and 16-processor systems, respectively. From these diagrams you can begin to see how the router configurations scale: each router is connected to two hubs, routers are then connected to each other forming a binary n-cube, or hypercube, where n, the dimensionality of the router configuration, is the base-2 logarithm of the number of routers. For the four-processor system, n is zero, and for the eight-processor system, the routers form a linear configuration, and n is one. In both the 12- and 16-processor systems, n is two and the routers form a square; for the 12-processor system, one corner of the square is missing. Larger systems are constructed by increasing the dimensionality of the router configuration and adding up to two hubs with each additional router. Systems with any number of nodes may be constructed by leaving off some corners of the n-dimensional hypercube. We will see these larger configurations later.

The key thing here is that the hardware allows the physically distributed memory of the system to be shared, just as in a bus-based system, but since each hub is connected to its own local memory, memory bandwidth is proportional to the number of nodes. As a result, there is no inherent limit to the number of processors that can be used effectively in the system. In addition, since the dimensionality of the router configuration grows as the systems get larger, the total router bandwidth also grows with system size (proportional to n2n, where n is the dimensionality of the router configuration). Thus, systems may be scaled without fear that the router connections will become a bottleneck.

To allow this scalability, however, one nice characteristic of the bus-based shared memory systems has been sacrificed; namely, the access time to memory is no longer uniform: it varies depending on how far away the memory being accessed is in the system. The two processors in each node have quick access through their hub to their local memory. Accessing remote memory through an additional hub adds an extra increment of time, as does each router the data must travel through. But several factors combine to smooth out these nonuniform memory access (NUMA) times:

1. The hardware has been designed so that the incremental costs to access remote memory are not large. The choice of a hypercube router configuration means that the number of routers information must pass through is at most n + 1, where n is the dimension of the hypercube; this grows only as the logarithm of the number of processors. As a result, the average memory access time on even the largest Origin system is no more than the uniform memory access time on a Power Challenge 10000 system. We'll see a detailed table of these costs later.

2. The R10000 processors operate on data that are resident in their caches. If programs use the caches effectively, the access time to memory, whether it is local or remote, is unimportant since the vast majority of memory accesses are satisfied from the caches.

3. Through operating system support or programming effort, the memory accesses of most programs can be made to come primarily from local memory. Thus, in the same way that the caches can make local memory access times unimportant, remote memory access costs can be reduced to an insignificant amount.

4. The R10000 processors can prefetch data that are not cache resident. Other work can be carried out while these data move from local or remote memory into the cache; thus the access time can be hidden.

The architecture of the Origin 2000 system, then, provides shared memory hardware without the limitations of traditional bus-based designs.

IIIB Profiling on the SGI Origin 2000

The hardware counters in the R10000 CPU make it possible to profile the behavior of a program in many ways without modifying the code. The software tools are perfex, which runs a program and reports exact counts of any two selected events from the R10000 counters. Alternatively, it time-multiplexes all 32 countable events and reports extrapolated totals of each. Perfex is useful for identifying what problem (for example, secondary data cache misses) is hurting the performance of your program the most. (see timex for simple timing functionality.) The Perfex functions are also available as callable library functions in libperfex. Similarly, for speedshop, the ssapi library is available. Speedshop (actually, the ssrun command), which runs a program while sampling the state of the program counter and stack, and writing the sample data to a file for later analysis. You select the timebase for the sampling and the particular type of information to be sampled. SpeedShop is useful for locating where in your program the performance problems occur. Prof, which analyzes a Speedshop data file and displays it in a variety of formats. Dprof, which, like Speedshop, samples a program while it is executing but records memory access information as a histogram file. It identifies which data structures in the program are involved in performance problems. Use these tools to find out what constrains the program and which parts of it consume the most time. Through the use of a combination of these tools, it is possible to identify most performance problems.

The profiling tools depend for most of their features on the R10000's performance counter registers. These on-chip registers can be programmed to count hardware events as they happen, for example, machine cycles, instructions, branch predictions, floating point instructions, or cache misses. There are only two performance counter registers. Each can be programmed to count machine cycles or 1 of 15 other events, for a total of 32 events that can be counted (30 of which are distinct). The specific events are summarized in Table III.1, which can be obtained by using the command perfex -h.

The counters are 64-bit integers. When a counter overflows, a hardware trap occurs. The kernel can preload a counter with $2^{64} - n$ to cause a trap after n counts occur. The profiling tools use this capability. For example, the command **ssrun -gi_hwc** programs the graduated instruction counter (event 17) to overflow every 32 K counts. Each time the counter overflows, ssrun samples the program counter and stack state of the subject program. The reference page *r10k_counters(5)* gives detailed information on how the counters can be accessed through the /proc interface. This is the interface used by the profiling tools. The interface hides the division of events between only two registers and allows the software to view the counters as a single set of thirty-two 64-bit counters. The operating system time-multiplexes the active counters between the events being counted. This requires sampling and scaling, which introduce some error when more than two events are counted. In general, it is better to access the counters through the profiling tools. A program that uses the counter interface directly cannot be profiled using perfex or using ssrun for any experiment that depends on counters. When a program must access counter values directly, the simplest interface is through libperfex, documented in the *libperfex* reference page.

0	Cycles	16	Cycles
1	Instructions issued to functional units	17	Instructions graduated
2	Memory data access (load, prefetch, sync, cacheop) issued	18	Memory data loads graduated
3	Memory stores issued	19	Memory data stores graduated
4	Store conditionals issued	20	Store conditionals graduated
5	Store conditionals failed	21	Floating point instructions graduated
6	Branches decoded	22	Quadwords written back from L1 cache
7	Quadwords written back from L2 cache	23	TLB refill exceptions
8	Correctable ECC errors on L2 cache	24	Branches mispredicted
9	L1 cache misses (instruction)	25	L1 cache misses (data)
10	L2 cache misses (instruction)	26	L2 cache misses (data)
11	L2 cache way mispredicted (instruction)	27	L2 cache way mispredicted (data)
12	External intervention requests	28	External intervention request hits in L2 cache
13	External invalidate requests	29	External invalidate request hits in L2 cache
14	Instructions done (formerly, virtual coherency condition)	30	Stores, or prefetches with store hint, to CleanExclusive L2 cache blocks
15	Instructions graduated	31	Stores, or prefetches with store hint, to Shared L2 cache blocks.

Table III.1: *Hardware counters of the R10000 processor.*

IIIB.1 Performance analysis using perfex

The simplest profiling tool is **perfex**, documented in the *perfex* reference page. It runs a subject program and records data about the run, similar to **timex**:

% perfex [options] command [arguments]

The subject program and its arguments are given. **perfex** sets up the counter interface and forks the subject program. When the program ends, **perfex** writes counter data to standard output. **perfex** gathers its information with no modifications to your existing program. Although this is convenient, the data obtained come from the entire run of the program. If you only want to profile a particular section of the program, you need to use the library interface to perfex, *libperfex(3)*. To use this interface, insert a call to initiate counting into your program's source code and another to terminate it; a third call prints the counts gathered. The program must then be linked with the libperfex library:

% cc -o program -lperfex

Since you can use SpeedShop to see where in a program various event counts come from, libperfex is not described in detail. More information can be found in its reference page.

IIIB.2 Absolute counts of one or two events

Use **perfex** options to specify what is to be counted. You can specify one or two countable events. In this case, the counts are absolute and accurate. For example, the command

% perfex -e 15 -e 21 adi2

runs the subject program and reports the exact counts of graduated instructions and graduated floating point instructions. You use this mode to explore specific points of program behavior.

IIIB.3 Statistical counts of all events

When you specify option -a (all events), **perfex** multiplexes all 32 events over the program run. Each count is active 1/16 of the time and then scaled by 16. The resulting counts have some statistical error. The error is small (and the counts sufficiently repeatable) provided that the subject program runs in a stable execution mode for a number of seconds. When the program runs for a short time, or shifts between radically different regimes of instruction or data use, the counts are less dependable and less repeatable. Nevertheless, **perfex -a** usually gives a good overview of program operation. Here is the **perfex** command line and output applied to a sample program called adi2:

% perfex -a -x adi2

```
WARNING: Multiplexing events to project totals--inaccuracy possible.

 Time:      7.990 seconds
 Checksum:  5.6160428338E+06
 0 Cycles.....................................................   1645481936
 1 Issued instructions........................................    677976352
 2 Issued loads...............................................    111412576
 3 Issued stores..............................................     45085648
 4 Issued store conditionals..................................            0
 5 Failed store conditionals..................................            0
 6 Decoded branches...........................................     52196528
 7 Quadwords written back from scache.........................     61794304
 8 Correctable scache data array ECC errors...................            0
 9 Primary instruction cache misses...........................         8560
10 Secondary instruction cache misses.........................          304
11 Instruction misprediction from scache way prediction table..         272
12 External interventions.....................................         6144
13 External invalidations.....................................        10032
14 Virtual coherency conditions...............................            0
15 Graduated instructions.....................................    371427616
16 Cycles.....................................................   1645481936
17 Graduated instructions.....................................    400535904
18 Graduated loads............................................     90474112
19 Graduated stores...........................................     34776112
20 Graduated store conditionals...............................            0
21 Graduated floating point instructions......................     28292480
22 Quadwords written back from primary data cache.............     32386400
23 TLB misses.................................................      5687456
24 Mispredicted branches......................................       410064
25 Primary data cache misses..................................     16330160
26 Secondary data cache misses................................      7708944
27 Data misprediction from scache way prediction table........       663648
28 External intervention hits in scache.......................         6144
29 External invalidation hits in scache.......................         6864
30 Store/prefetch exclusive to clean block in scache..........      7582256
31 Store/prefetch exclusive to shared block in scache.........         8144
```

The -x option requests that **perfex** also gather counts for kernel code that handles exceptions, so the work done by the OS to handle TLB misses is included in these counts.

IIIB.4 Analytic output with the -y option

The raw event counts are interesting, but it is more useful to convert them to elapsed time. Some time estimates are simple, for example, dividing the cycle count by the machine clock rate gives the elapsed run time (1645481936 / 195 MHz = 8.44 seconds). Other events are not as simple and can only be stated in terms of a range of times. For example, the time to handle a primary cache miss varies depending on whether the needed data are in the secondary cache, in memory, or in the cache of another CPU. Analysis of this kind can be requested using **perfex -a -x -y**. When you use -a , -x, and -y, **perfex** collects and displays all event counts, but it also displays a report of estimated times based on the counts. Here is an example, again, of the program adi2:

% perfex -a -x -y adi2

```
WARNING: Multiplexing events to project totals--inaccuracy possible.

 Time:      7.996 seconds
 Checksum:  5.6160428338E+06

                                                                 Based on 196 MHz IP27
                                                                              Typical      Minimum      Maximum
   Event Counter Name                                          Counter Value  Time (sec)   Time (sec)   Time (sec)
================================================================================================================
 0 Cycles.....................................................   1639802080     8.366337     8.366337     8.366337
16 Cycles.....................................................   1639802080     8.366337     8.366337     8.366337
26 Secondary data cache misses................................      7736432     2.920580     1.909429     3.248837
23 TLB misses.................................................      5693808     1.978017     1.978017     1.978017
 7 Quadwords written back from scache.........................     61712384     1.973562     1.305834     1.973562
25 Primary data cache misses..................................     16368384     0.752445     0.235504     0.752445
22 Quadwords written back from primary data cache.............     32385280     0.636139     0.518825     0.735278
 2 Issued loads...............................................    109918560     0.560809     0.560809     0.560809
18 Graduated loads............................................     88890736     0.453524     0.453524     0.453524
 6 Decoded branches...........................................     52497360     0.267844     0.267844     0.267844
 3 Issued stores..............................................     43923616     0.224100     0.224100     0.224100
19 Graduated stores...........................................     33430240     0.170562     0.170562     0.170562
21 Graduated floating point instructions......................     28371152     0.144751     0.072375     7.527040
30 Store/prefetch exclusive to clean block in scache..........      7545984     0.038500     0.038500     0.038500
24 Mispredicted branches......................................       417440     0.003024     0.001363     0.011118
 9 Primary instruction cache misses...........................         8272     0.000761     0.000238     0.000761
10 Secondary instruction cache misses.........................          768     0.000290     0.000190     0.000323
31 Store/prefetch exclusive to shared block in scache.........        15168     0.000077     0.000077     0.000077
 1 Issued instructions........................................    673476960     0.000000     0.000000     3.436107
 4 Issued store conditionals..................................            0     0.000000     0.000000     0.000000
 5 Failed store conditionals..................................            0     0.000000     0.000000     0.000000
 8 Correctable scache data array ECC errors...................            0     0.000000     0.000000     0.000000
11 Instruction misprediction from scache way prediction table..          432     0.000000     0.000000     0.000002
12 External interventions.....................................         6288     0.000000     0.000000     0.000000
13 External invalidations.....................................         9360     0.000000     0.000000     0.000000
14 Virtual coherency conditions...............................            0     0.000000     0.000000     0.000000
15 Graduated instructions.....................................    364303776     0.000000     0.000000     1.858693
17 Graduated instructions.....................................    392675440     0.000000     0.000000     2.003446
20 Graduated store conditionals...............................            0     0.000000     0.000000     0.000000
27 Data misprediction from scache way prediction table........       679120     0.000000     0.000000     0.003465
28 External intervention hits in scache.......................         6288     0.000000     0.000000     0.000000
29 External invalidation hits in scache.......................         5952     0.000000     0.000000     0.000000

Statistics
================================================================================
Graduated instructions/cycle..................................     0.222163
Graduated floating point instructions/cycle...................     0.017302
Graduated loads & stores/cycle................................     0.074595
Graduated loads & stores/floating point instruction...........     5.422486
Mispredicted branches/Decoded branches........................     0.007952
Graduated loads/Issued loads..................................     0.808696
Graduated stores/Issued stores................................     0.761099
Data mispredict/Data scache hits..............................     0.078675
Instruction mispredict/Instruction scache hits................     0.057569
L1 Cache Line Reuse...........................................     6.473003
L2 Cache Line Reuse...........................................     1.115754
L1 Data Cache Hit Rate........................................     0.866185
L2 Data Cache Hit Rate........................................     0.527355
```

```
Time accessing memory/Total time.....................................     0.750045
L1--L2 bandwidth used (MB/s, average per process).....................   124.541093
Memory bandwidth used (MB/s, average per process).....................   236.383187
MFLOPS (average per process)..........................................     3.391108
```

"Maximum," "minimum," and "typical" time cost estimates are reported. Each is obtained by consulting an internal table which holds the maximum, minimum, and typical costs for each event, and multiplying this cost by the count for the event. Event costs are usually measured in terms of machine cycles, and so the cost of an event generally depends on the clock speed of the processor, which is also reported in the output. The maximum value in the table corresponds to the worst-case cost of a single occurrence of the event. Sometimes this can be a pessimistic estimate. For example, the maximum cost for graduated floating point instructions assumes that every floating point instruction is a double-precision reciprocal square root since it is the most costly R10000 floating point instruction. Because of the latency-hiding capabilities of the R10000, the minimum cost of virtually any event could be zero since most events can be overlapped with other operations. To avoid simply reporting minimum costs of zero, which would be of no practical use, the minimum time reported by **perfex -y** corresponds to the best-case cost of a single occurrence of the event. The best-case cost is obtained by running the maximum number of simultaneous occurrences of that event and averaging the cost. For example, two floating point instructions can complete per cycle, so the best case cost is 0.5 cycles per floating point instruction. The typical cost falls somewhere between minimum and maximum and is meant to correspond to the cost you see in average programs. **perfex -y** prints the event counts and associated cost estimates sorted from most costly to least costly. Although resembling a profiling output, this is not a true profile. The event costs reported are only estimates. Furthermore, since events do overlap with one another, the sum of the estimated times will usually exceed the program's run time. This output should only be used to identify which events are responsible for significant portions of the program's run time and to get a rough idea of what those costs might be. In the example above, the program spends a significant fraction of its time handling secondary cache and TLB misses. To make a significant improvement in the run time of this program, the tuning measures need to concentrate on reducing those cache misses. In addition to the event counts and cost estimates, **perfex -y** also reports a number of statistics derived from the typical costs. The meaning of many of the statistics is self-evident, for example, Graduated instructions/cycle. Below is a list of those statistics whose definitions require more explanation:

Data mispredict/Data scache hits The ratio of the counts for data misprediction from scache way prediction table and secondary data cache misses.

Instruction mispredict/Instruction scache hits The ratio of the counts for instruction misprediction from scache way prediction table and secondary instruction cache misses.

L1 Cache Line Reuse The number of times, on average, that a primary data cache line is used after it has been moved into the cache. It is calculated as graduated loads plus graduated stores minus primary data cache misses, divided by primary data cache misses.

L2 Cache Line Reuse The number of times, on average, that a secondary data cache line is used after it has been moved into the cache. It is calculated as primary data cache misses minus secondary data cache misses, divided by secondary data cache misses.

L1 Data Cache Hit Rate The fraction of data accesses that are satisfied from a cache line already resident in the primary data cache. It is calculated as 1.0 - (primary data cache misses divided by the sum of graduated loads and graduated stores).

L2 Data Cache Hit Rate The fraction of data accesses that are satisfied from a cache line already resident in the secondary data cache. It is calculated as 1.0 - (secondary data cache misses divided by primary data cache misses).

Time accessing memory/Total time The sum of the typical costs of graduated loads, graduated stores, primary data cache misses, secondary data cache misses, and TLB misses, divided by the total program run time. The total program run time is calculated by multiplying cycles by the time per cycle (inverse of the processor's clock speed).

L1–L2 bandwidth used (MB/s, average per process) The amount of data moved between the primary and secondary data caches, divided by the total program run time. The amount of data moved is calculated as the sum of the number of primary data cache misses multiplied by the primary cache line size and the number of quadwords written back from primary data cache multiplied by the size of a quadword (16 bytes). For multiprocessor programs, the resulting figure is a per-process average since the counts measured by **perfex** are aggregates of the counts for all the threads. Multiply by the number of threads to get the total program bandwidth.

Memory bandwidth used (MB/s, average per process) The amount of data moved between the secondary data cache and main memory, divided by the total program run time. The amount of data moved is calculated as the sum of the number of secondary data cache misses multiplied by the secondary cache line size and the number of quadwords written back from secondary data cache multiplied by the size of a quadword (16 bytes). For multiprocessor programs, the resulting figure is a per-process average since the counts measured by **perfex** are aggregates of the counts for all the threads. Multiply by the number of threads to get the total program bandwidth.

MFLOPS (MB/s, average per process) The ratio of the graduated floating point instructions and the total program run time. Note that although a multiply-add carries out two floating point operations, it only counts as one instruction, so this statistic may underestimate the number of floating point operations per second. For multiprocessor programs, the resulting figure is a per-process average since the counts measured by **perfex** are aggregates of the counts for all the threads. Multiply by the number of threads to get the total program rate.

These statistics give you a quick way to identify performance problems in your program. For example, the cache hit-rate statistics tell you how cache friendly your

program is. Since a secondary cache miss is much more expensive than a cache hit, the L2 Data Cache Hit Rate needs to be close to 1.0 to indicate that the program is not paying a large penalty for the cache misses. Values of ± 0.96 and above indicate good cache performance. Note that, for the above example, the rate is 0.53, further confirmation of the cache problems in this program.

IIIB.5 Using SpeedShop

The purpose of profiling is to find out exactly where a program is spending its time, that is, in precisely which procedures or lines of code. Then you can concentrate your efforts on the (usually small) areas of code where there is the most to be gained. Profiling using the SpeedShop package supports these methods:

- Sampling, in which the subject program is frequently interrupted; the program counter (PC) and stack are recorded on each interruption. The more frequently the PC is found in a particular procedure, the more execution time that procedure costs.
- SpeedShop can sample on a variety of time bases: the system timer or any of the R10000 performance counters. Ideal counting, in which a copy of the subject program binary is modified with trap instructions at the end of each basic block. During execution, the exact number of uses of each basic block is counted.
- Exception trace, not really a profiling method, records only floating point exceptions and their locations.

The SpeedShop package has three parts:

- ssrun performs experiments and collects data.
- prof processes data and prepares reports.
- The ssapi interface allows you to insert caliper points into a program to profile specific sections of code or phases of execution.

These programs are documented in the following reference pages: *speedshop* documents the types of experiments, as well as a number of environment variables you can set. *ssrun* documents specific options of ssrun. You need both **speedshop(1)** and **ssrun(1)** to run an experiment. *prof* documents the report types and the options you use to get them. *ssapi* documents the three library calls you can use.

IIIB.6 PC sampling profiling

The accuracy of sampling depends on the time base that sets the sampling interval. In each case, the time base is the independent variable and the program state is the dependent variable. Select from the sampling methods summarized in Table III.2. Each time base finds the program PC more often in the code that consumes the most units of that time base:

ssrun Option		Time Base		Comments
-usertime		30 ms timer		Fairly coarse resolution; experiment runs quickly and output file is small; some bugs noted in speedshop(1).
-pcsamp[x]	-fpcsamp[x]	10 ms timer	1 ms timer	Moderately coarse resolution; functions that cause cache misses or page faults are emphasized. Suffix x for 32-bit counts.
-gi_hwc	-fgi_hwc	32771 insts	6553 insts	Fine-grain resolution based on graduated instructions. Emphasizes functions that burn a lot of instructions.
-cy_hwc	-fcy_hwc	16411 clocks	3779 clocks	Fine-grain resolution based on elapsed cycles. Emphasizes functions with cache misses and mispredicted branches.
-ic_hwc	-fic_hwc	2053 icache miss	419 icache miss	Emphasizes code that doesn't fit in L1 cache.
-isc_hwc	-fisc_hwc	131 scache miss	29 scache miss	Emphasizes code that doesn't fit in L2 cache. Should be coarse-grained measure.
-dc_hwc	-fdc_hwc	2053 dcache miss	419 dcache miss	Emphasizes code that causes L1 cache data misses.
-dsc_hwc	-fdsc_hwc	131 scache miss	29 scache miss	Emphasizes code that causes L2 cache data misses.
-tlb_hwc	-ftlb_hwc	257 TLB miss	53 TLB miss	Emphasizes code that causes page faults.
-gfp_hwc	-fgfp_hwc	32771 fp insts	6553 fp insts	Emphasizes code that performs heavy FP calculation.
-prof_hwc		user-set		Hardware counter and overflow value from environment variables.

Table III.2: *Sampling methods.*

- The time bases that reflect actual elapsed time (-usertime, -pcsamp, -cy_hwc) find the PC more often in the code where the program spends elapsed time. The time may be spent in that code because it is executed a lot, or it might be spent there because those instructions are processed slowly owing to cache misses, memory contention, or failed branch prediction. Use these to get an overview of the program and to find major trouble spots.
- The time bases that reflect instruction counts (-gi_hwc, -gfp_hwc) find the PC more often in the code that actually performs the most instructions. Use these to find the code that could benefit most from algorithmic changes.
- The time bases that reflect data access (-dc_hwc, -sc_hwc, -tlb_hwc) find the PC more often in the code that has to wait for its data to be brought in from another memory level. Use these to find memory access problems.
- The time bases that reflect code access (-ic_hwc, -isc_hwc) find the PC more often in the code that has to be fetched from memory when it is called. Use these to pinpoint functions that could be reorganized for better locality, or to see when automatic inlining has gone too far.

It is easy to perform an experiment. Here is the application of an experiment to program adi2:

% ssrun -fpcsamp adi2

```
 Time:      7.619 seconds
 Checksum:  5.6160428338E+06
```

The output file of samples is left in a file with the default name of ./command.experiment.pid. In this case the name was adi2.fpcsamp.4885. It is often more convenient, however, to dictate the name of the output file. You can do this by putting the desired filename and directory in environment variables. Using this csh script you can run an experiment, passing the output directory and filename on the command line, for example

% ssruno -d /var/tmp -o adi2.cy -cy_hwc adi2

```
ssrun -cy\_hwc adi2 ..............................
 Time:      9.644 seconds
 Checksum:  5.6160428338E+06
.............................. ssrun ends.
-rw-r--r--    1 guest    guest      18480 Dec 17 16:25 /var/tmp/adi2.cy
```

IIIB.7 Using prof

Regardless of which time base you use for sampling, you display the result using **prof**. By default, **prof** displays a list of procedures ordered from the one with the most samples to the least:

% prof adi2.fpcsamp.4885

```
------------------------------------------------------------------------
-------
Profile listing generated Sat Jan  4 10:28:11 1997
    with:       prof adi2.fpcsamp.4885
------------------------------------------------------------------------
-------

samples   time    CPU    FPU   Clock   N-cpu  S-interval Countsize
   8574   8.6s R10000 R10010 196.0MHz   1      1.0ms     2(bytes)

Each sample covers 4 bytes for every  1.0ms ( 0.01% of 8.5740s)


------------------------------------------------------------------------
-------
  -p[rocedures] using pc-sampling.
  Sorted in descending order by the number of samples in each procedure.
  Unexecuted procedures are excluded.
------------------------------------------------------------------------
-------

samples   time(%)      cum time(%)      procedure (dso:file)

   6688   6.7s( 78.0)  6.7s( 78.0)         zsweep (adi2:adi2.f)
    671  0.67s(  7.8)  7.4s( 85.8)         xsweep (adi2:adi2.f)
    662  0.66s(  7.7)    8s( 93.6)         ysweep (adi2:adi2.f)
    208  0.21s(  2.4)  8.2s( 96.0)       fake_adi (adi2:adi2.f)
    178  0.18s(  2.1)  8.4s( 98.1)         irand_ (/usr/lib32/libftn.so:../../libF77/rand_.c)
    166  0.17s(  1.9)  8.6s(100.0)          rand_ (/usr/lib32/libftn.so:../../libF77/rand_.c)
      1 0.001s(  0.0)  8.6s(100.0)      __oserror (/usr/lib32/libc.so.1:oserror.c)

   8574   8.6s(100.0)  8.6s(100.0)          TOTAL
```

This profile indicates that you should focus on the routine zsweep, since it consumes almost 80% of the run time of this program. For finer detail, use the -heavy option. This supplements the list of procedures with a list of individual source line numbers, ordered by frequency:

```
------------------------------------------------------------------------
-------
  -h[eavy] using pc-sampling.
     Sorted in descending order by the number of samples in each line.
     Lines with no samples are excluded.
------------------------------------------------------------------------
-------

samples   time(%)      cum time(%)      procedure (file:line)

   3405   3.4s( 39.7)  3.4s( 39.7)         zsweep (adi2.f:122)
   3226   3.2s( 37.6)  6.6s( 77.3)         zsweep (adi2.f:126)
    425  0.42s(  5.0)  7.1s( 82.3)         xsweep (adi2.f:80)
    387  0.39s(  4.5)  7.4s( 86.8)         ysweep (adi2.f:101)
    273  0.27s(  3.2)  7.7s( 90.0)         ysweep (adi2.f:105)
    246  0.25s(  2.9)    8s( 92.9)         xsweep (adi2.f:84)
    167  0.17s(  1.9)  8.1s( 94.8)         irand_ (../../libF77/rand_.c:62)
    163  0.16s(  1.9)  8.3s( 96.7)       fake_adi (adi2.f:18)
    160  0.16s(  1.9)  8.5s( 98.6)          rand_ (../../libF77/rand_.c:69)
```

```
    45 0.045s(  0.5)  8.5s( 99.1)       fake_adi (adi2.f:59)
    32 0.032s(  0.4)  8.5s( 99.5)         zsweep (adi2.f:113)
    21 0.021s(  0.2)  8.5s( 99.7)         zsweep (adi2.f:121)
    11 0.011s(  0.1)  8.6s( 99.8)         irand_ (../../libF77/rand_.c:63)
     6 0.006s(  0.1)  8.6s( 99.9)          rand_ (../../libF77/rand_.c:67)
     4 0.004s(  0.0)  8.6s(100.0)         zsweep (adi2.f:125)
     1 0.001s(  0.0)  8.6s(100.0)         ysweep (adi2.f:104)
     1 0.001s(  0.0)  8.6s(100.0)         ysweep (adi2.f:100)
     1 0.001s(  0.0)  8.6s(100.0)       __oserror (oserror.c:127)

  8574   8.6s(100.0)  8.6s(100.0)          TOTAL
```

From this listing it is clear that lines 122 and 126 warrant further inspection. Even finer detail can be obtained with the -source option, which lists the source code and disassembled machine code, indicating sample hits on specific instructions.

IIIB.8 Ideal time profiling

The other type of profiling is called ideal time, or basic block, profiling. Basic block is a compiler term for a section of code that has only one entrance and one exit. Any program can be decomposed into basic blocks. To obtain an ideal profile, **ssrun** copies the executable program and modifies the copy to contain code that records the entry to each basic block. Not only the executable itself but also all dynamic shared objects (DSOs; for more information, see *dso(5)*) that it links to are copied and instrumented. The instrumented executable and libraries are statically linked and run:

% ssrun -ideal adi2

```
Beginning libraries
        /usr/lib32/libssrt.so
        /usr/lib32/libss.so
        /usr/lib32/libfastm.so
        /usr/lib32/libftn.so
        /usr/lib32/libm.so
        /usr/lib32/libc.so.1
Ending libraries, beginning "adi2"
 Time:      8.291 seconds
 Checksum:  5.6160428338E+06
```

The number of times each basic block was encountered is recorded. The output data file is displayed using **prof**, just as for a sampled run. The report ranks source and library procedures from most to least used:

% prof adi2.ideal.4920

```
  Prof run at: Sat Jan  4 10:34:06 1997
 Command line: prof adi2.ideal.4920


         285898739: Total number of cycles
          1.45867s: Total execution time
         285898739: Total number of instructions executed
             1.000: Ratio of cycles / instruction
               196: Clock rate in MHz
            R10000: Target processor modeled


------------------------------------------------------------
```

```
Procedures sorted in descending order of cycles executed.
Unexecuted procedures are not listed. Procedures
beginning with *DF* are dummy functions and represent
init, fini and stub sections.
------------------------------------------------------------

       cycles(%)  cum %     secs    instrns    calls procedure(dso:file)

  68026368(23.79)  23.79     0.35   68026368 32768 xsweep(adi2.pixie:adi2.f)
  68026368(23.79)  47.59     0.35   68026368 32768 ysweep(adi2.pixie:adi2.f)
  68026368(23.79)  71.38     0.35   68026368 32768 zsweep(adi2.pixie:adi2.f)
  35651584(12.47)  83.85     0.18   35651584 2097152 rand_(./libftn.so.pixn32:../../libF77/rand_.c)
  27262976( 9.54)  93.39     0.14   27262976 2097152 irand_(./libftn.so.pixn32:../../libF77/rand_.c)
  18874113( 6.60)  99.99     0.10   18874113     1 fake_adi(adi2.pixie:adi2.f)
     11508( 0.00)  99.99     0.00      11508     5 memset(./libc.so.1.pixn32:/slayer_xlv0/ficussg-nov05/work/irix/lib/libc/libc_n32_M4/strings/bzero.s)
      3101( 0.00)  99.99     0.00       3101    55 __flsbuf(./libc.so.1.pixn32:_flsbuf.c)
      2446( 0.00) 100.00     0.00       2446    42 x_putc(./libftn.so.pixn32:../../libI77/wsfe.c)
      1234( 0.00) 100.00     0.00       1234     2 x_wEND(./libftn.so.pixn32:../../libI77/wsfe.c)
      1047( 0.00) 100.00     0.00       1047     1 f_exit(./libftn.so.pixn32:../../libI77/close.c)
      1005( 0.00) 100.00     0.00       1005     5 fflush(./libc.so.1.pixn32:flush.c)
       639( 0.00) 100.00     0.00        639     4 do_fio64_mp(./libftn.so.pixn32:../../libI77/fmt.c)
       566( 0.00) 100.00     0.00        566     3 wrt_AP(./libftn.so.pixn32:../../libI77/wrtfmt.c)
       495( 0.00) 100.00     0.00        495     6 map_luno(./libftn.so.pixn32:../../libI77/util.c)
       458( 0.00) 100.00     0.00        458    14 op_gen(./libftn.so.pixn32:../../libI77/fmt.c)
       440( 0.00) 100.00     0.00        440     9 gt_num(./libftn.so.pixn32:../../libI77/fmt.c)
       414( 0.00) 100.00     0.00        414     1 getenv(./libc.so.1.pixn32:getenv.c)
                                    .
                                    .
                                    .
```

The -heavy option adds a list of source lines, sorted by their consumption of ideal instruction cycles. An ideal profile shows exactly and repeatedly which statements are most executed and gives you an exact view of the algorithmic complexity of the program. An ideal profile does not necessarily reflect where a program spends its time since it cannot take cache and TLB misses into account. Consequently, the results of the ideal profile are startlingly different from that of the PC sampling profile. These ideal results indicate that zsweep should take exactly the same amount of run time as ysweep and xsweep. These differences can be used to infer where cache performance issues exist. On machines without the R10000's hardware profiling registers, such comparisons are the only profiling method available to locate cache problems.

IIIB.9 Operation counts

Since ideal profiling counts the instructions executed by the program, it can provide all sorts of interesting information about the program. Already printed in the standard **prof** output are counts of how many times each subroutine is called. In addition, you may use the -op option to **prof** to get a listing detailing the counts of all instructions in the program. In particular, this will provide an exact count of the floating point operations executed:

% prof -op adi2.ideal.4920

```
 Prof run at: Wed Jan 15 14:42:54 1997
Command line: prof -op adi2.ideal.4920


         285898739: Total number of cycles
          1.45867s: Total execution time
         285898739: Total number of instructions executed
             1.000: Ratio of cycles / instruction
               196: Clock rate in MHz
            R10000: Target processor modeled
```

```
-------------------------------------------------------
pixstats summary
-------------------------------------------------------
           56590456: Floating point operations (38.796 Mflops @ 196 MHz)
          105500230: Integer operations (72.3265 M intops @ 196 MHz)

                                .
                                .
                                .
```

Note that this is different from what you get using **perfex**. The R10000 counter #21 counts floating point instructions, not floating point operations. As a result, in a program that executes a lot of multiply-add instructions — each of which carries out two floating point operations — **perfex**'s MFLOPS statistic can be off by a factor of two. Since **prof -op** records all instructions executed, it counts each multiply-add instruction as two floating point operations, thus providing the correct tally. The MFLOPS figure it calculates, however, is based on the ideal time; to calculate floating point performance, divide the number of floating point operations counted by **prof -op** by wall clock time. Either method of profiling, PC sampling or ideal, can be applied to multiprocessor runs just as easily as it is applied to single-processor runs; each thread of an application maintains its own histogram, and the histograms may be printed individually or merged in any combination and printed as one profile.

IIIB.10 Gprof

One limitation of the **prof** output for either PC sampling or ideal time is that the information reported contains no information about the call hierarchy. That is, if the routine zsweep in the above example were called from two different locations in the program, you would not know how much time results from the call at each location; you would only know the total time spent in zsweep. If you knew that, say, the first location was responsible for the majority of the time, this could affect how you tune the program. For example, you might try inlining the call into the first location, but not bother with the second. Or, if you wanted to parallelize the program, knowing that the first location is where the majority of the time is spent, you might consider parallelizing the calls to zsweep there rather than trying to parallelize the zsweep routine itself. SpeedShop provides two methods of obtaining hierarchical profiling information. The first method, which is called gprof, is used in conjunction with the ideal time profile. To obtain the gprof information for the above example, simply add the flag -gprof to the **prof** command:

% prof -gprof adi2.ideal.4920

```
 Prof run at: Wed Jan 15 16:52:09 1997
Command line: prof -gprof adi2.ideal.4920


        285898739: Total number of cycles
         1.45867s: Total execution time
        285898739: Total number of instructions executed
            1.000: Ratio of cycles / instruction
              196: Clock rate in MHz
           R10000: Target processor modeled
```

```
------------------------------------------------------------
Procedures sorted in descending order of cycles executed.
Unexecuted procedures are not listed. Procedures
beginning with *DF* are dummy functions and represent
init, fini and stub sections.
------------------------------------------------------------

        cycles(%)  cum %     secs    instrns    calls procedure(dso:file)

  68026368(23.79)  23.79     0.35   68026368 32768 xsweep(adi2.pixie:adi2.f)
  68026368(23.79)  47.59     0.35   68026368 32768 ysweep(adi2.pixie:adi2.f)
  68026368(23.79)  71.38     0.35   68026368 32768 zsweep(adi2.pixie:adi2.f)
  35651584(12.47)  83.85     0.18   35651584 2097152 rand_(./libftn.so.pixn32:../../libF77/rand_.c)
  27262976( 9.54)  93.39     0.14   27262976 2097152 irand_(./libftn.so.pixn32:../../libF77/rand_.c)
                                                .
                                                .
                                                .

          All times are in milliseconds.
-------------------------------------------------------------------------------
          NOTE: any functions which are not part of the call
                graph are listed at the end of the gprof listing
-------------------------------------------------------------------------------

                                    self             kids        called/total    parents
 index          cycles(%)          self(%)          kids(%)  called+self     name     index
                                    self             kids        called/total    children

   [1]  285895481(100.00%)         57( 0.00%)  285895424(100.00%)     0       __start [1]
                                      50          285895369         1/1         main [2]
                                       3                  0         1/1         __istart [112]
                                       2                  0         1/1         __readenv_sigfpe [113]

-------------------------------------------------------------------------------

                                      50          285895369         1/1         __start [1]
   [2]  285895419(100.00%)         50( 0.00%)  285895369(100.00%)     1       main [2]
                                18874113          267020445         1/1         fake_adi [3]
                                    205                606         5/5         signal [44]

-------------------------------------------------------------------------------

                                18874113          267020445         1/1         main [2]
   [3]  285894558(100.00%)   18874113( 6.60%)  267020445(93.40%)     1       fake_adi [3]
                                68026368                  0     32768/32768     zsweep [4]
                                68026368                  0     32768/32768     ysweep [5]
                                68026368                  0     32768/32768     xsweep [6]
                                35651584           27262976     2097152/2097152     rand_ [7]
                                      28              13486         2/2         s_wsfe64 [9]
                                      26                5368        2/2         e_wsfe [17]
                                      22                2610        1/1         do_fioxr4v [25]
                                      22                2610        1/1         do_fioxr8v [24]
                                      23                2428        1/1         s_stop [28]
                                     114                  44        2/2         dtime_ [68]

-------------------------------------------------------------------------------

                                68026368                  0     32768/32768     fake_adi [3]
   [4]   68026368(23.79%)   68026368(100.00%)          0( 0.00%) 32768       zsweep [4]

-------------------------------------------------------------------------------

                                68026368                  0     32768/32768     fake_adi [3]
   [5]   68026368(23.79%)   68026368(100.00%)          0( 0.00%) 32768       ysweep [5]

-------------------------------------------------------------------------------

                                68026368                  0     32768/32768     fake_adi [3]
   [6]   68026368(23.79%)   68026368(100.00%)          0( 0.00%) 32768       xsweep [6]

-------------------------------------------------------------------------------
                                                .
                                                .
                                                .
```

This produces the usual ideal time profiling output, but following that is the hierarchical information. There is a block of information for each subroutine in the program. A number, shown in brackets (e.g., [1]), is assigned to each routine so that the information pertaining to it can easily be located in the output. Let's look in detail at the block of information provided; we'll use fake_adi [3] as an example. The line beginning with the number [3] shows, from left to right, the:

- Number of cycles consumed by this routine and the routines it calls (its descen-

dants)

- Number of cycles spent inside the routine, but not in any of its descendants
- Number of cycles spent in its descendants
- Total number of times the routine was called
- Name of the routine, fake_adi [3]

Above this line are lines showing which routines fake_adi [3] was called from. In this case, it is only called from one place, main [2], so there is just one line (in general, there would be one line for each routine which calls fake_adi [3]). This line shows

- The proportion of the cycles spent inside fake_adi [3] as a result of the call from main [2];
- The proportion of time spent in fake_adi [3]'s descendants as a result of the call from main [2];
- How many calls there are to fake_adi [3] in main [2] / the total number of calls to fake_adi [3] from all laces in the program

Since fake_adi [3] is only called once, all the time in it and its descendants is the result of this one call. Below the line beginning with the number [3] are all the descendants of fake_adi [3]. For each descendant you see:

- The proportion of the descendant's cycles spent inside it.
- The proportion of the descendant's cycles spent in its descendants (i.e., fake_adi [3]'s grandchildren).
- How many calls to the descendant there are in fake_adi [3] / the total number of calls to the descendant from all places in the program.

This block of information allows you to determine not just which subroutines but which paths in the program are responsible for the majority of time. The only limitation is that gprof reports ideal time, so cache misses are not represented.

IIIB.11 Usertime profiling

gprof only reports ideal time. To get hierarchical profiling information that accurately accounts for all the time in the program, the way PC sampling does, use usertime profiling. For this type of profiling, the program is sampled during the run. At each sample, the location of the program counter is noted and the entire call stack is traced to record which routines have been called to get to this point in the program. From this, a hierarchical profile is constructed. Since unwinding the call stack is an expensive operation, the sampling period is for usertime profiling is relatively long: 30 milliseconds. usertime profiling is performed with the following command:

% ssrun -usertime adi2

The output is written to a file called ./adi2.usertime.pid, where pid is the process ID for this run of adi2. The profile is displayed using **prof** just as for PC sampling and ideal time profiling:

% prof adi2.usertime.19572

The output is as follows:

```
------------------------------------------------------------------------
-------
Profile listing generated Wed Jan 15 16:57:10 1997
    with:       prof adi2.usertime.19572
------------------------------------------------------------------------
-------

        Total Time (secs)     : 9.99
        Total Samples         : 333
        Stack backtrace failed: 0
        Sample interval (ms)  : 30
        CPU                   : R10000
        FPU                   : R10010
        Clock                 : 196.0MHz
        Number of CPUs        : 1


------------------------------------------------------------------------
-------

index  %Samples     self descendents  total        name
[1]    100.0%    0.00        9.99     333          __start
[2]    100.0%    0.00        9.99     333          main
[3]    100.0%    0.09        9.90     333          fake_adi
[4]     80.8%    8.07        0.00     269          zsweep
[5]      7.5%    0.75        0.00      25          xsweep
[6]      6.9%    0.69        0.00      23          ysweep
[7]      3.9%    0.12        0.27      13          rand_
[8]      2.7%    0.27        0.00       9          irand_
```

The information is less detailed than that provided by gprof, but when combined with gprof, you can get a complete hierarchical profile for all routines which have run long enough to be sampled.

IIIC Dynamic Acceleration Structures for Interactive Ray Tracing

Acceleration structures used for ray tracing have been designed and optimized for efficient traversal of static scenes. As it becomes feasible to do interactive ray tracing of moving objects, new requirements are posed upon the acceleration structures. Dynamic environments require rapid updates to the acceleration structures. In this paper we propose spatial subdivisions which allow insertion and deletion of objects in constant time at an arbitrary position, allowing scenes to be interactively animated and modified.

IIIC.1 Introduction

Recently, interactive ray tracing has become a reality [16, 19], allowing exploration of scenes rendered with higher quality shading than with traditional interactive rendering algorithms. A high frame-rate is obtained through parallelism, using a multiprocessor shared memory machine. This approach has advantages over hardware accelerated interactive systems in that a software-based ray tracer is more easily modified. One of the problems with interactive ray tracing is that previous implementations only dealt with static scenes or scenes with a small number of specially handled moving objects. The reason for this limitation is that the acceleration structures used to make ray tracing efficient rely on a significant amount of preprocessing to build. This effectively limits the usefulness of interactive ray tracing to applications which allow changes in camera position. The work presented in this paper is aimed at extending the functionality of interactive ray tracing to include applications where objects need to be animated or interactively manipulated.

When objects can freely move through the scene, either through user interaction, or due to system-determined motion, it becomes necessary to adapt the acceleration methods to cope with changing geometry. Current spatial subdivisions tend to be highly optimized for efficient traversal, but are difficult to update quickly for changing geometry. For static scenes this suffices, as the spatial subdivision is generally constructed during a pre-processing step. However, in animated scenes pre-processed spatial subdivisions may have to be recalculated for each change of the moving objects. One approach to circumvent this issue is to use 4D radiance interpolants to speed-up ray traversal [2]. However, within this method the frame update rates depend on the type of scene edits performed as well as the extent of camera movement. We will therefore focus on adapting current spatial subdivision techniques to avoid these problems.

To animate objects while using a spatial subdivision, insertion and deletion costs are not negligible, as these operations may have to be performed many times during rendering. In this paper, spatial subdivisions are proposed which allow efficient ray traversal as well as rapid insertion and deletion for scenes where the extent of the scene grows over time.

The following section presents a brief overview of current spatial subdivision techniques (Section IIIC.2), followed by an explanation of our (hierarchical) grid modifications (Sections IIIC.3 and IIIC.4). A performance evaluation is given in Section IIIC.5, while conclusions are drawn in the final section.

IIIC.2 Acceleration Structures for Ray Tracing

There has been a great deal of work done on acceleration structures for ray tracing [13]. However, little work has focused on ray tracing moving objects. Glassner presented an approach for building acceleration structures for animation [12]. However, this approach does not work for environments without *a priori* knowledge of the animation path for each object. In a survey of acceleration techniques, Gaede and Günther provide an overview of many spatial subdivisions, along with the requirements for various applications [10]. The most important requirements for ray tracing are fast ray traversal and adaptation to unevenly distributed data. Currently popular spatial subdivisions can be broadly categorized into bounding volume hierarchies and voxel based structures.

Bounding volume hierarchies create a tree, with each object stored in a single node. In theory, the tree structure allows $O(\log n)$ insertion and deletion, which may be fast enough. However, to make the traversal efficient, the tree is augmented with extra data, and occasionally flattened into an array representation [26], which enables fast traversal but insertion or deletion incur a non-trivial cost. Another problem is that as objects are inserted and deleted, the tree structure could become arbitrarily inefficient unless some sort of rebalancing step is performed as well.

Voxel based structures are either grids [1, 9] or can be hierarchical in nature, such as bintrees and octrees [11, 27]. The cost of building a spatial subdivision tends to be $O(n)$ in the number of objects. This is true for both grids and octrees. In addition, the cost of inserting a single object may depend on its relative size. A large object generally intersects many voxels, and therefore incurs a higher insertion cost than smaller objects. This can be alleviated through the use of modified hierarchical grids, as explained in Section IIIC.4. The larger problem with spatial subdivision approaches is that the grid structure is built within volume bounds that are fixed before construction. Although insertion and deletion may be relatively fast for most objects, if an object is moved outside the extent of the spatial subdivision, current structures would require a complete rebuild. This problem is addressed in the next section.

IIIC.3 Grids

Grid spatial subdivisions for static scenes, without any modifications, are already useful for animated scenes, as traversal costs are low and insertion and deletion of objects is reasonably straightforward. Insertion is usually accomplished by mapping the axis-aligned bounding box of an object to the voxels of the grid. The object is inserted into all voxels that overlap with this bounding box. Deletion can be achieved in a similar way.

However, when an object moves outside the extent of the spatial subdivision, the acceleration structure would normally have to be rebuilt. As this is too expensive to perform repeatedly, we propose to logically replicate the grid over space. If an object exceeds the bounds of the grid, the object wraps around before re-insertion. Ray traversal then also wraps around the grid when a boundary is reached. In order to provide a stopping criterion for ray traversal, a logical bounding box is maintained which contains all objects, including the ones that have crossed the original perimeter. As this scheme does not require grid re-computation whenever an object moves far away, the

cost of maintaining the spatial subdivision will be substantially lower. On the other hand, because rays now may have to wrap around, more voxels may have to be traversed per ray, which will slightly increase ray traversal time.

During a pre-processing step, the grid is built as usual. We will call the bounding box of the entire scene at start-up the 'physical bounding box'. If during the animation an object moves outside the physical bounding box, either because it is placed by the user in a new location, or its programmed path takes it outside, the logical bounding box is extended to enclose all objects. Initially, the logical bounding box is equal to the physical bounding box. Insertion of an object which lies outside the physical bounding box is accomplished by wrapping the object around within the physical grid, as depicted in Figure III.17 (left).

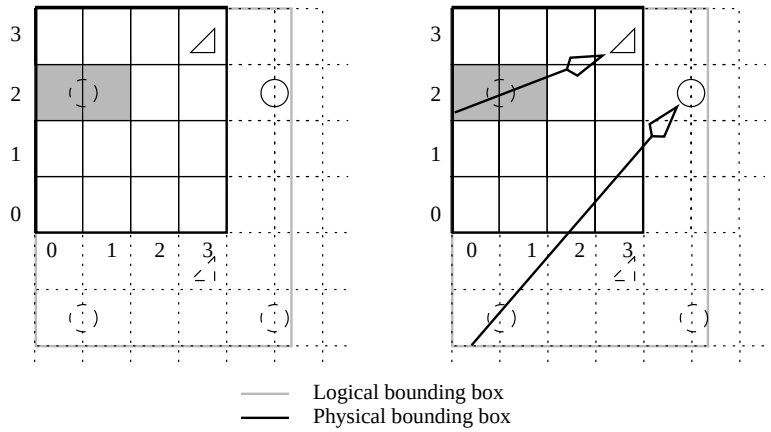


Figure III.17: Grid insertion (left). The sphere has moved outside the physical grid, now overlapping with voxels (4, 2) and (5, 2). Therefore, the object is inserted at the location of the shaded voxels. The logical bounding box is extended to include the newly moved object. Right: ray traversal through extended grid. The solid lines are the actual objects whereas the dashed lines indicate voxels which contain objects whose actual extents are not contained in that voxel.

As the logical bounding box may be larger than the physical bounding box, ray traversal now starts at the extended bounding box and ends if an intersection is found or if the ray leaves the logical bounding box. In the example in Figure III.17 (right), the ray pointing to the sphere starts within a logical voxel, voxel (-2, 0), which is mapped to physical voxel (0, 2). The logical coordinates of the sphere are checked and found to be outside of the currently traversed voxel and thus no intersection test is necessary. The ray then progresses to physical voxel (1, 2). For the same reason, no intersection with the sphere is computed again. Traversal then continues until the sphere is intersected in logical voxel (4, 2), which maps to physical voxel (0, 2).

Objects that are outside the physical grid are tagged, so that in the above example, when the ray aimed at the triangle enters voxels (0, 2) and (1, 2), the sphere does not have to be intersected. Similarly, when the ray is outside the physical grid, objects that are within the physical grid need not be intersected. As most objects will initially

lie within the physical bounds, and only a few objects typically move away from their original positions, this scheme speeds up traversal considerably for parts of the ray that are outside the physical bounding box.

When the logical bounding box becomes much larger than the physical bounding box, there is a tradeoff between traversal speed (which deteriorates for large logical bounding boxes) and the cost of rebuilding the grid. In our implementation, the grid is rebuilt when the length of the diagonals of the physical and logical bounding boxes differ by a factor of two.

Hence, there is a hierarchy of operations that can be performed on grids. For small to moderate expansions of the scene, wrapping both rays and objects is relatively quick without incurring too high a traversal cost. For larger expansions, rebuilding the grid will become a more viable option.

This grid implementation shares the advantages of simplicity and cheap traversal with commonly used grid implementations. However, it adds the possibility of increasing the size of the scene without having to completely rebuild the grid every time there is a small change in scene extent. The cost of deleting and inserting a single object is not constant and depends largely on the size of the object relative to the size of the scene. This issue is addressed in the following section.

IIIC.4 Hierarchical grids

As was noted in the previous section, the size of an object relative to each voxel in a grid influences how many voxels will contain that object. This in turn negatively affects insertion and deletion times. Hence, it would make sense to find a spatial subdivision whereby the voxels can have different sizes. If this is accomplished, then insertion and deletion of objects can be made independent of their sizes and can therefore be executed in constant time. Such spatial subdivisions are not new and are known as hierarchical spatial subdivisions. Octrees, bintrees and hierarchical grids are all examples of hierarchical spatial subdivisions. However, normally such spatial subdivisions store all their objects in leaf nodes and would therefore still incur non-constant insertion and deletion costs. We extend the use of hierarchical grids in such a way that objects can also reside in intermediary nodes or even in the root node for objects that are nearly as big as the entire scene.

Because such a structure should also be able to deal with expanding scenes, our efforts were directed towards constructing a hierarchy of grids (similar to Sung [28]), thereby extending the functionality of the grid structure presented in the previous section. Effectively, the proposed method constitutes a balanced octree.

Object insertion now proceeds similarly to grid insertion, except that the grid level needs to be determined before insertion. This is accomplished by comparing the size of the object in relation to the size of the scene. A simple heuristic is to determine the grid level from the diagonals of the two bounding boxes. Specifically, the length of the grid's diagonal is divided by the length of the object's diagonal, the result determining the grid level. Insertion and deletion progresses as explained in the previous section.

The gain of constant time insertion is offset by a slightly more complicated traversal algorithm. Hierarchical grid traversal is effectively the same as grid traversal with the following modifications. Traversal always starts at a leaf node which may first

be mapped to a physical leaf node as described in the previous section. The ray is intersected with this voxel and all its parents until the root node is reached. This is necessary because objects at all levels in the hierarchy may occupy the same space as the currently traversed leaf node. If an intersection is found within the space of the leaf node, then traversal is finished. If not, the next leaf node is selected and the process is repeated.

This traversal scheme is wasteful because the same parent nodes may be repeatedly traversed for the same ray. To combat this problem, note that common ancestors of the current leaf node and the previously intersected leaf node, need not be traversed again. If the ray direction is positive, the current voxel's number can be used to derive the number of levels to go up in the tree to find the common ancestor between the current and the previously visited voxel. For negative ray directions, the previously visited voxel's number is used instead. Finding the common ancestor is achieved using simple bit manipulation, as detailed in Figure III.18.

```
bitmask = (raydir_x > 0) ? x : x + 1
forall levels in hierarchical grid
{
  cell = hgrid[level][x>>level][y>>level][z>>level]
  forall objects in cell
    intersect(ray, object)
  if (bitmask & 1)
    return
  bitmask >>= 1
}
```

Figure III.18: Hierarchical grid traversal algorithm in C-like pseudo-code. The bitmask is set assuming that the last step was along the x-axis.

As the highest levels of the grid may not contain any objects, ascending all the way to the highest level in the grid is not always necessary. Ascending the tree for a particular leaf node can stop when the largest voxel containing objects is visited.

This hierarchical grid structure has the following features. The traversal is only marginally more complex than standard grid traversal. In addition, wrapping of objects in the face of expanding scenes is still possible. If all objects are the same size, this algorithm effectively defaults to grid traversal. Insertion and deletion can be achieved in constant time, as the number of voxels that each object overlaps is roughly constant[6].

IIIC.5 Evaluation

The grid and hierarchical grid spatial subdivisions were implemented using an interactive ray tracer [19], which runs on an SGI Origin 2000 with 32 processors. For evaluation purposes, two test scenes were used. In each scene, a number of objects were animated using pre-programmed motion paths. The scenes as they are at start-up are depicted in Figure III.21 (top). An example frame taken during the animation is given for each scene in Figure III.21 (bottom). All images were rendered on 30 processors at a resolution of 512^2 pixels.

[6]Note that this also obviates the need for mailbox systems to avoid redundant intersection tests.

To assess basic traversal speed, the new grid and hierarchical grid implementations are compared with a bounding volume hierarchy. We also compared our algorithms with a grid traversal algorithm which does not allow interactive updates. Its internal data structure consists of a single array of object pointers, which improves cache efficiency on the Origin 2000.

From here on we will refer to the new grid implementation as ‘interactive grid’ to distinguish between the two grid traversal algorithms. As all these spatial subdivision methods have a user defined parameter to set the resolution (voxels along one axis and maximum number of grid levels, respectively), various settings are evaluated. The overall performance is given in Figure III.19 and is measured in frames per second.

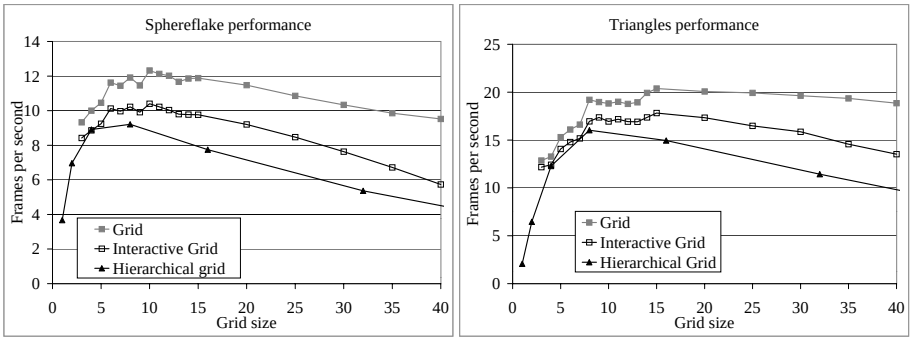


Figure III.19: Performance (in frames per second) for the grid, the interactive grid and the hierarchical grid for two static scenes. The bounding volume hierarchy achieves a frame rate of 8.5 fps for the static sphereflake model and 16.4 fps for the static triangles model.

The extra flexibility gained by both the interactive grid and hierarchical grid implementations results in a somewhat slower frame rate. This is according to expectation, as the traversal algorithm is a little more complex and the Origin’s cache structure cannot be exploited as well with either of the new grid structures. The graphs in Figure III.19 show that with respect to the grid implementation the efficiency reduction is between 12% and 16% for the interactive grid and 21% and 25% for the hierarchical grid. These performance losses are deemed acceptable since they result in far better overall execution than dynamically reconstructing the original grid. For the sphereflake, all implementations are faster, for a range of grid sizes, than a bounding volume hierarchy, which runs at 8.5 fps. For the triangles scene, the hierarchical grid performs at 16.0 fps similarly to the bounding volume (16.4 fps), while grid and interactive grid are faster.

The non-zero cost of updating the scene effectively limits the number of objects that can be animated within the time-span of a single frame. However, for both scenes, this limit was not reached. In the case where the frame rate was highest for the triangles scene, updating all 200 triangles took less than 1/680th of a frame for the hierarchical grid and 1/323th of a frame for the interactive grid. The sphereflake scene costs even less to update, as fewer objects are animated. For each of these tests, the hierarchical

grid is more efficiently updated than the interactive grid, which confirms its usefulness.

The size difference between different objects should cause the update efficiency to be variable for the interactive grid, while remaining relatively constant for the hierarchical grid. In order to demonstrate this effect, both the ground plane and one of the triangles in the triangle scene was interactively repositioned during rendering. The update rates for different size parameters for both the interactive grid and the hierarchical grid, are presented in Figure III.20 (left). As expected, the performance of the hierarchical grid is relatively constant, although the size difference between ground plane and triangle is considerable. The interactive grid does not cope with large objects very well if these objects overlap with many voxels. Dependent on the number of voxels in the grid, there is one to two orders of magnitude difference between inserting a large and a small object. For larger grid sizes, the update time for the ground plane is roughly half a frame. This leads to visible artifacts when using an interactive grid, as during the update the processors that are rendering the next frame temporarily cannot intersect this object (it is simply taken out of the spatial subdivision). In practice, the hierarchical grid implementation does not show this disadvantage.

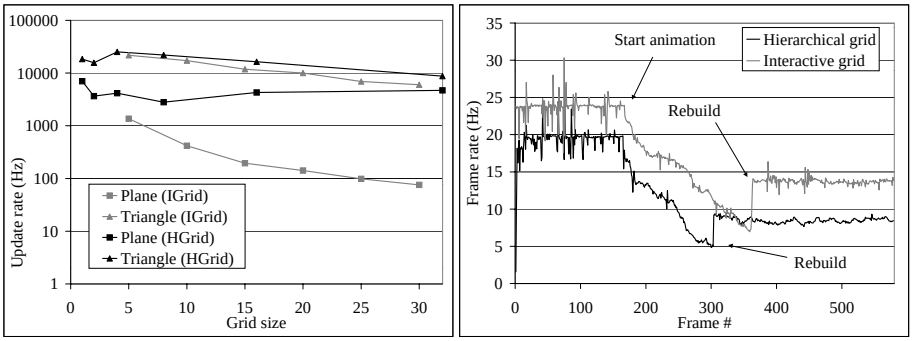


Figure III.20: Left: Update rate as function of (hierarchical) grid size. The plane is the ground plane in the triangles scene and the triangle is one of the triangles in the same scene. Right: Frame rate as function of time for the expanding triangle scene.

The time to rebuild a spatial subdivision from scratch is expected to be considerably higher than the cost of re-inserting a small number of objects. For the triangles scene, where 200 out of 201 objects were animated, the update rate was still a factor of two faster than the cost of completely rebuilding the spatial subdivision. This was true for both the interactive grid and the hierarchical grid. A factor of two was also found for the animation of 81 spheres in the sphereflake scene. When animating only 9 objects in this scene, the difference was a factor of 10 in favor of updating. We believe that the performance difference between rebuilding the acceleration structure and updating all objects is largely due to the cost of memory allocation, which occurs when rebuilding.

In addition to experiments involving grids and hierarchical grids with a branching factor of two, tests were performed using a hierarchical grid with a higher branching factor. Instead of subdividing a voxel into eight children, here nodes are split into 64 children (4 along each axis). The observed frame rates are very similar to the hierar-

chical grid. The object update rates were slightly better for the sphereflake and triangle scenes, because the size differences between the objects matches this acceleration structure better than both the interactive grid and the hierarchical grid.

In the case of expanding scenes, the logical bounding box will become larger than the physical bounding box. The number of voxels that are traversed per ray will therefore on average increase. This is the case in the triangles scene[7]. The variation over time of the frame rate is given in Figure III.20 (right). In this example, the objects are first stationary. At some point the animation starts and the frame rate drops because the scene immediately starts expanding. At some point the expansion is such that a rebuild is warranted. The re-computed spatial subdivision now has a logical bounding box which is identical to the (new) physical bounding box and therefore the number of traversed voxels is reduced when compared with the situation just before the rebuild. The total frame rate does not reach the frame rate at the start of the computation, because the objects are more spread out over space, resulting in larger voxels and more intersection tests which do not yield an intersection point.

Finally, Figure III.22 shows that interactively updating scenes using drag and drop interaction is feasible.

IIIC.6 Conclusions

When objects are interactively manipulated and animated within a ray tracing application, much of the work that is traditionally performed during a pre-processing step becomes a limiting factor. Especially spatial subdivisions which are normally built once before the computation starts, do not exhibit the flexibility that is required for animation. The insertion and deletion costs can be both unpredictable and variable. We have argued that for a small cost in traversal performance flexibility can be obtained and insertion and deletion of objects can be performed in constant time.

By logically extending the (hierarchical) grids into space, these spatial subdivisions deal with expanding scenes rather naturally. For modest expansions, this does not significantly alter the frame rate. When the scenes expand a great deal, rebuilding the entire spatial subdivision may become necessary. For large scenes this may involve a temporary drop in frame rate. For applications where this is unacceptable, it would be advisable to perform the rebuilding within a separate thread (rather than the display thread) and use double buffering to minimize the impact on the rendering threads.

[7] For this experiment, the ground plane of the triangles scene was reduced in size, allowing the rebuild to occur after a smaller number of frames.

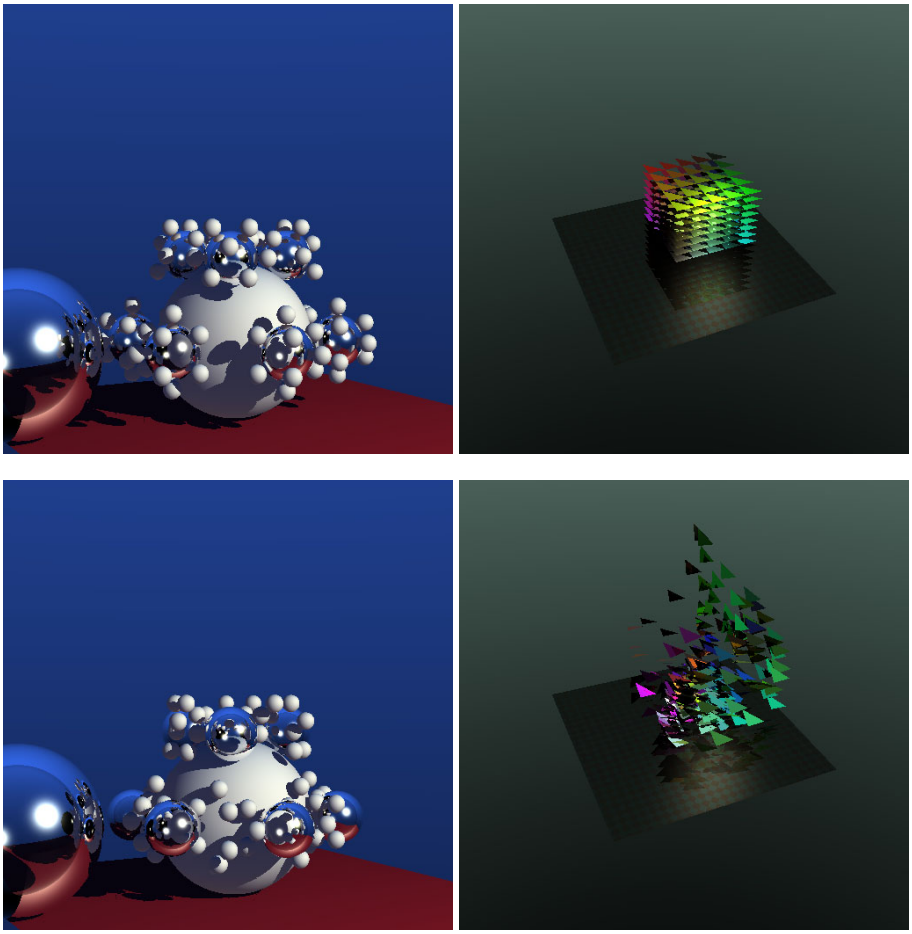


Figure III.21: Test scenes before any objects moved (top) and during animation (bottom).

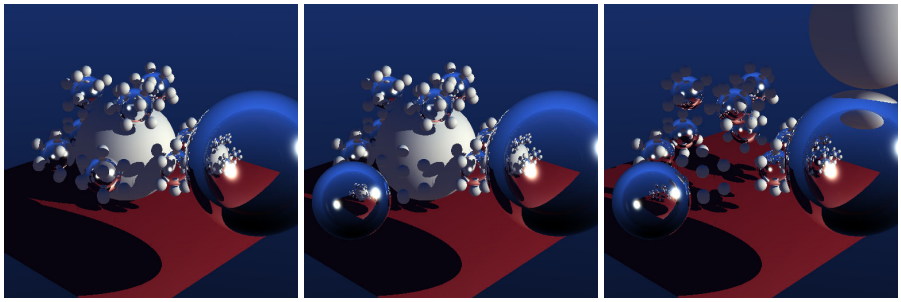


Figure III.22: Frames created during interactive manipulation.

Bibliography

[1] J. AMANATIDES AND A. WOO, *A fast voxel traversal algorithm for ray tracing*, in Eurographics '87, Elsevier Science Publishers, Amsterdam, North-Holland, Aug. 1987, pp. 3–10.

[2] K. BALA, J. DORSEY, AND S. TELLER, *Interactive ray traced scene editing using ray segment tree*, in Rendering Techniques '99, D. Lischinski and G. W. Larson, eds., Eurographics, Springer-Verlag Wien New York, 1999, pp. 31–44.

[3] ——, *Radiance interpolants for accelerated bounded-error ray tracing*, ACM Transactions on Graphics, 18 (1999), pp. 213–256.

[4] G. BISHOP, H. FUCHS, L. MCMILLAN, AND E. J. SCHER ZAGIER, *Frameless rendering: Double buffering considered harmful*, Computer Graphics, 28 (1994), pp. 175–176. ACM Siggraph '94 Conference Proceedings.

[5] A. CHALMERS, T. DAVIS, AND E. REINHARD, eds., *Practical Parallel Rendering*, AK Peters, 2002.

[6] J. G. CLEARY AND G. WYVILL, *Analysis of an algorithm for fast ray tracing using uniform space subdivision*, The Visual Computer, (1988), pp. 65–83.

[7] D. CORTESI AND J. FIER, *Origin2000 (tm) and onyx2 (tm) performance tuning and optimization guide*, tech. rep., Silicon Graphics, 1998. Available from http://techpubs.sgi.com:80/.

[8] R. A. CROSS, *Interactive realism for visualization using ray tracing*, in Proceedings Visualization '95, 1995, pp. 19–25.

[9] A. FUJIMOTO, T. TANAKA, AND K. IWATA, *ARTS: Accelerated ray tracing system*, IEEE Computer Graphics and Applications, 6 (1986), pp. 16–26.

[10] V. GAEDE AND O. GÜNTHER, *Multidimensional access methods*, ACM Computing Surveys, 30 (1998), pp. 170–231.

[11] A. S. GLASSNER, *Space subdivision for fast ray tracing*, IEEE Computer Graphics and Applications, 4 (1984), pp. 15–22.

[12] A. S. GLASSNER, *Spacetime ray tracing for animation*, IEEE Computer Graphics and Applications, 8 (1988), pp. 60–70.

[13] A. S. GLASSNER, ed., *An Introduction to Ray Tracing*, Academic Press, San Diego, 1989.

[14] G. W. LARSON AND M. SIMMONS, *The holodeck ray cache: An interactive rendering system for global illumination in non-diffuse environments*, ACM Transactions on Graphics, 18 (October 1999), pp. 361–368.

[15] J. D. MACDONALD AND K. S. BOOTH, *Heuristics for ray tracing using space subdivision*, The Visual Computer, 6 (1990), pp. 153–166.

[16] M. J. MUUSS, towards real-time ray-tracing of combinatorial solid geometric models, in Proceedings of BRL-CAD Symposium, June 1995.

[17] K. NAKAMARU AND Y. OHNO, *Breadth-first ray tracing utilizing uniform spatial subdivision*, IEEE Transactions on Visualization and Computer Graphics, 3 (1997), pp. 316–328.

[18] K. NEMOTO AND T. OMACHI, *An adaptive subdivision by sliding boundary surfaces for fast ray tracing*, in Proceedings of Graphics Interface '86, M. Green, ed., May 1986, pp. 43–48.

[19] S. PARKER, W. MARTIN, P.-P. SLOAN, P. SHIRLEY, B. SMITS, AND C. HANSEN, *Interactive ray tracing*, in Symposium on Interactive 3D Computer Graphics, April 1999.

[20] S. PARKER, M. PARKER, Y. LIVNAT, P.-P. SLOAN, C. HANSEN, AND P. SHIRLEY, *Interactive ray tracing for volume visualization*, in IEEE Transactions on Visualization and Computer Graphics, July-September 1999.

[21] E. REINHARD, A. CHALMERS, AND F. W. JANSEN, *Sampling diffuse interreflection within a hybrid scheduling ray tracer*, Journal of Parallel and Distributed Computing Practices, 3 (2001), pp. 11–19.

[22] E. REINHARD, P. SHIRLEY, AND C. HANSEN, *Parallel point reprojection*, in IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics, 2001, pp. 29–35.

[23] E. REINHARD, B. SMITS, AND C. HANSEN, *Dynamic acceleration structures for interactive ray tracing*, in Proceedings of the 11th Eurographics Workshop on Rendering, Brno, Czech Republic, June 2000, pp. 299–306.

[24] M. SIMMONS AND C. SÉQUIN, *Tapestry: A dynamic mesh-based display representation for interactive rendering*, in Proceedings of the 11th Eurographics Workshop on Rendering, Brno, Czech Republic, June 2000, pp. 329–340.

[25] P. SLUSALLEK, P. HANRAHAN, S. PARKER, H. PFISTER, T. PURCELL, AND E. REINHARD, *Interactive ray tracing*, August 2001. Siggraph course #13.

[26] B. SMITS, *Efficiency issues for ray tracing*, Journal of Graphics Tools, 3 (1998), pp. 1–14.

[27] J. SPACKMAN AND P. WILLIS, *The SMART navigation of a ray through an oct-tree*, Computers and Graphics, 15 (1991), pp. 185–194.

[28] K. SUNG, *A DDA octree traversal algorithm for ray tracing*, in Eurographics '91, W. Purgathofer, ed., North-Holland, sept 1991, pp. 73–85. European Computer Graphics Conference and Exhibition; held in Vienna, Austria; 2-6 September 1991.

[29] I. WALD AND P. SLUSALLEK, *State of the art in interactive ray tracing*, in Eurographics STAR - State of the Art reports, 2001, pp. 21–42.

[30] I. WALD, P. SLUSALLEK, C. BENTHIN, AND M. WAGNER, *Interactive rendering with coherent ray tracing*, Computer Graphics Forum, 20 (2001), pp. 153–164.

[31] B. WALTER, G. DRETTAKIS, AND S. PARKER, *Interactive rendering using the render cache*, in Rendering Techniques '99, D. Lischinski and G. W. Larson, eds., Eurographics, Springer-Verlag Wien New York, 1999, pp. 19–30.

[32] K.-Y. WHANG, J.-W. SONG, J.-W. CHANG, J.-Y. KIM, W.-S. CHO, C.-M. PARK, AND I.-Y. SONG, *Octree-r: An adaptive octree for efficient ray tracing*, IEEE Transactions on Visualization and Computer Graphics, 1 (1995), pp. 343–349.

[33] E. S. ZAGIER, *Defining and refining frameless rendering*, Tech. Rep. TR97-008, UNC-CS, July 1997.

Section IV
The “Kilauea” Massively Parallel Ray Tracer

Toshi Kato

The "Kilauea" Massively Parallel Global Illumination Renderer

Toshiaki Kato
Square USA Honolulu Studio

Hitoshi Nishimura, Tadashi Endo, Tamotsu Maruyama,
Jun Saito, Motohisa Adachi

Abstract

Kilauea, a revolutionary parallel renderer which has been developed at Square USA, will be described. The goal of the R&D effort was to create a renderer which is able to render extremely complex scenes with the consideration of global illumination. The renderer makes massive use of multiple Linux PCs which are networked together to form a cluster of rendering servers. This course note will illustrate the methods that were used to stabilize the parallel renderer and optimize the parallel performance and final rendering speed in order to meet the requirements of the production pipeline. We believe that the topics covered here will be useful as a guideline for implementing parallel renderers.

1 Overview of the Kilauea Research Project

1.1 What is the Kilauea Research Project?

The Kilauea Research Project took place in the R&D division of Square USA's Honolulu Studio from March 1998 till March 2002, to design and implement a ground-breaking high-end renderer. The development started fully from scratch – from all architectural design to the actual coding. The name of the renderer developed in this project is Kilauea.

Two ultimate goals were set when the Kilauea Research Project began. The Kilauea renderer aimed to render:

1. High quality images using global illumination;
2. Extremely complex and large scenes.

The global illumination computation methodology is an actively debated subject and various ideas are proposed[2][9][3]. Kilauea chose a ray tracing based global illumination methodology because this offers one robust algorithm which handles diverse scene types, primitives, and visual effects such as motion blur, while allowing the overall system architecture to be clean and flexible. Kilauea is designed to compute global illumination using final gathering and photon mapping, which are based on Monte Carlo ray tracing.

However, ray tracing is inherently expensive, and tends to require all scene data to be expanded into the memory. Thus, the algorithm has an evident disadvantage in rendering huge and complex scenes. To achieve the goals of Kilauea, our approach is to take advantage of parallel processing. While the cost of PC hardware is dropping, its performance is steadily increasing. The evolution of cost performance is happening not just to inside PCs, but also to the network connecting them. Kilauea obtains enormous computational resources at minimum expense by massively clustering cost-efficient PCs running Linux.

1.2 Characteristics of Kilauea

Kilauea renders an image by the interaction of processes running on multiple machines. Essentially, Kilauea is a parallel ray tracer with its basis on message passing. The rendering computation is best described as the flow of data, where each Kilauea process sends data back and forth.

This type of parallel renderers is far more complex compared to sequential renderers, and have numerous implementation issues specific to the nature of parallel computing. Please refer to SIGGRAPH 2001 course #40[7] for the whole picture of the Kilauea architecture. This course note aims to focus on various ideas, principles, and implementation tips and tricks collected in the development process to improve Kilauea's speed, stability, and usability as a parallel renderer.

The improvements to Kilauea presented here should be valuable as a guide to parallel rendering or parallel processing in general, even if there is no preliminary knowledge on Kilauea's architecture.

2 Scene construction improvements

This section describes various ideas implemented to improve the scene construction in Kilauea.

2.1 Improving object distribution

How Kilauea distributes the scene data to multiple machines and techniques to improve its performance are explained here.

2.1.1 Reading in scene data and rendering

Kilauea can assign WTask (task to read in the scene) and ATask (task to hold geometry data and compute ray tracing) to separate machines. Depending on the size of the scene, the number of machines to run these tasks can be adjusted. If the scene data is small enough to fit in the memory of one machine, increasing the number of ATasks will increase the rendering speed accordingly. Multiple ATasks may also be grouped together to hold extremely large scenes which cannot be stored in the memory of one machine. These task configurations can be mixed together. For example, if the scene is too large and needs to be shared among two machines and four machines are available, the rendering speed can be doubled by using two sets of two machines. If six machines are available, use three sets of two machines to triple the speed, and so on. This is the principal mechanism to accomplish one of Kilauea's goals to render complex and huge scenes. In order to initiate such distributed renderings, the scene data must first be transferred properly to all machines. This is an unavoidable overhead caused by the essential design of Kilauea and many efforts are done to minimize it.

2.1.2 Scene data distribution: first implementation

Kilauea uses an original scene data format called *ShotData*. One of its characteristics is that it is an incremental format storing only the difference from the previous frame. WTask constructs the current frame data from the previous frame data and the difference data newly read in from ShotData. WTask then sends the current frame data to available ATasks. After receiving the scene data, ATasks start constructing acceleration grid for speeding up ray tracing. When multiple ATasks are used for speed-up, WTask sends exact same copies of data to all ATasks. Even if the scene does not fit in one machine and it is shared among multiple ATasks, these ATasks can be considered as one group, and WTask sends exact same copies to all ATask groups (figure 1).

The scene data is divided into multiple data packets of constant size and are sent from WTask to ATask in series. In most cases, acceleration grid construction at ATask takes longer than data transmission. The queue for receiving data will overflow if WTask keeps on sending data without waiting for them to be processed. To avoid this, ATask requests WTask to postpone sending

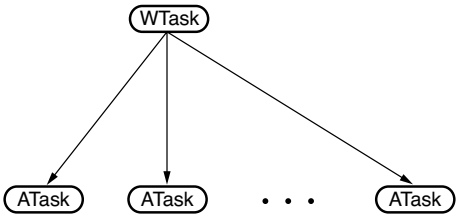


Figure 1: Data distribution

data when the data queue size exceeds a certain size. When one WTask needs to send data to multiple ATasks, WTask sequentially schedules the send requests to handle multiple send requests. WTask creates multiple threads to process data send requests from ATasks. Even when all ATasks requested WTask to wait, WTask keeps on creating send data packet until the memory exhausts.

2.1.3 Scene data distribution: improved version

The mechanism in the previous section works reasonably well when the number of ATasks is small. However, as the number of ATasks increases, the number of threads in WTask for sending data must also be increased. If WTask needs to send data to a large number of ATasks and the number of send threads are relatively small, the first data packet sent to the first ATask will be processed way before the packet is sent to the last ATask, and the first ATask needs to wait until all ATasks are ready to receive the second data packet.

However, the number of threads cannot be increased unlimitedly without considering the network bandwidth. Blindly increasing the number of send threads will simply saturate the network, without improving the overall performance. Therefore, the number of send threads must be limited by the network bandwidth. But then this will cause many ATasks to wait for data.

Here, notice that WTask is sending the same data to all ATasks. Because of this property, ATasks which received the data can also send the data to other ATasks which have not received the data yet. This is achieved by setting a constant limit to the number of ATasks to which WTask directly sends data, and passing the routing information along with the scene data. Because ATasks can now actively participate in the duty of WTask other than data packet generation, the CPU load and network traffic is distributed evenly throughout the render farm. Another advantage of this method is that CPUs on multi-CPU systems are utilized better than the previous method. Because acceleration grid construction is essentially difficult to parallelize, it can only make use of one CPU. Now that ATasks are also responsible for sending the scene data packets, CPUs not running the acceleration grid construction have jobs to do.

Determining the optimal data routing path holds the key to maximizing the performance in this method. The simplest yet effective routing is the binary/ternary tree distribution, as shown in figure 2. WTask is the root of the tree, and ATasks are the tree nodes. Each node is allowed to have two to three sub nodes.

Simple binary tree distribution improves the situation to the satisfactory level. However, one of the problems with the method is that deeper tree levels suffer from the time lag in data distribution. Apparently, the scene data take longer to get copied to the nodes at deeper levels. To a certain extent, the lag also happens to the nodes at the same level because even if the send operations are multi-threaded, the packets will eventually be serialized at low level system network layer. In the current implementation, other than the adjustment of the number of sub nodes allowed at a certain tree level according to the network bandwidth, partial tree depth control and additional routing path creation are performed to minimize the lag.

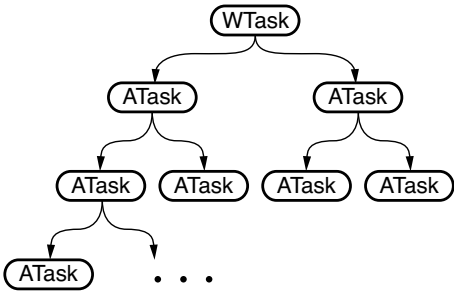


Figure 2: Data transfer in binary tree

2.1.4 Object distribution: first implementation

As previously mentioned, Kilauea can distribute a scene to several machines in order to render extremely complex and large scenes. This section discusses the strategy to split the scene data to multiple parts to be distributed.

The first implementation was the most straightforward one, where each triangle is randomly distributed (figure 3).

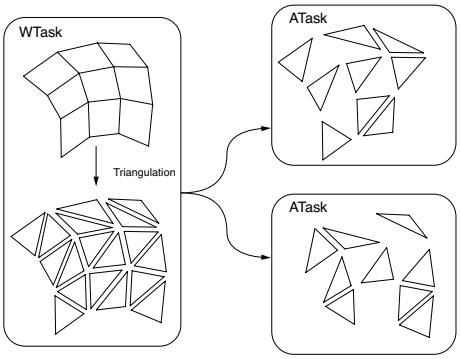


Figure 3: Random primitive distribution

All objects in the acceleration grid are stored as individual triangles transformed to the world coordinate system. The objects read from the file must be transformed at some point, and in the first implementation, WTask did all the transform of the local coordinate system to the world coordinate system, then these objects were transferred to ATask. This maximizes the randomness in the distribution of triangles to improve the parallel ray tracing performance. On the other hand, because the data is expanded before the transfer, the data size of the WTask to ATask transfer will increase. Primitives other than polygonal objects such as NURBS and subdivision surfaces also had to be tessellated before the transfer, and this initial implementation was evidently a naive, wasteful one.

2.1.5 Object distribution: improved version

Object-based random scene distribution is another straightforward approach over distribution based on triangles or similar primitives. When the scene consists of many relatively simple objects, scheduling the distribution to ATasks based on objects will suffice. This approach apparently fails in an extreme case where the scene consists of one extremely complex object and another simple

object. When the scene is distributed to two ATasks, one will have millions of triangles whereas the other only has a few triangles. Objects over certain complexity must be adequately partitioned and then distributed (figure 4). With this method, only vertices at the border of the partitioned pieces are duplicated and the partitioned pieces can be sent to ATasks while keeping the shared vertices. This considerably reduces the memory requirement compared to the previous triangle-based distribution.

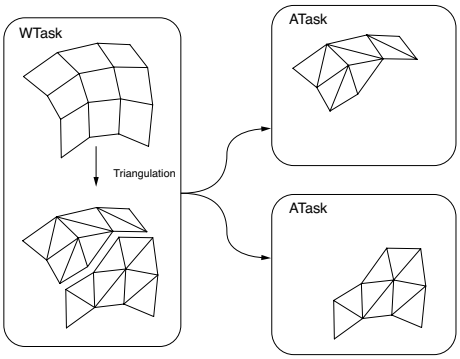


Figure 4: Random partition distribution

The instantiated objects may be expanded at WTask to increase the memory requirements and communication overhead. When WTask performs the transform from the object local coordinate system to the world coordinate system, all instantiated objects must be expanded. To avoid the load, this operation is instead executed at ATasks. One of the features made possible by this idea is the cycle animation. This is a mechanism to efficiently render crowd animation by instantiating objects including animation composition. This feature exhibits its full strength when the instantiated objects are expanded in ATasks.

2.2 Improvements to acceleration grid

Previous section dealt with how the scene data is read from the file and distributed to machines. This section explains how the distributed scene data are processed to prepare for ray tracing.

2.2.1 Acceleration grid in Kilauea

Kilauea uses a hierarchical uniform grid for the acceleration grid. When the scene must be distributed across multiple ATasks, grid cell divisions must be a fixed number for the entire scene, instead of being computed independently for each acceleration grid. This is necessary in order to avoid problems that occur due to numerical imprecision (more on this in [7]).

2.2.2 Leaf node rule

Hierarchical uniform grid is a data structure that divides space into uniform pieces. Cells in the grid may also be uniformly divided grids. When a primitive is added to a cell that already contains primitives over a threshold, a new hierarchy is created by subdividing the cell. The primitives in the subdivided cell are moved to a smaller cell at the bottom of the hierarchy. This method where primitives are added only at the bottom of the hierarchy is referred to as the *leaf node rule*. The advantage of this method is that intersection detection can be performed efficiently by starting from the cell that contains the ray origin and traversing only the leaves of the tree (figure 5).

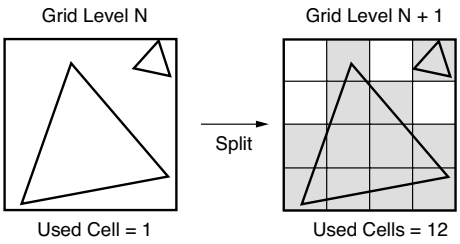


Figure 5: Leaf node rule

2.2.3 Motion blur object

Almost every parameter in Kilauea may change over time. There are three cases where motion blur is required when rendering an object: when the object is either moving or deforming, and when the camera is moving. Of these cases, there is nothing the acceleration grid needs to do when the camera is moving; the rays shot from the camera simply change over time, and the individual ray tracing is handled in a normal way. Object deformation and motion can actually be optimized individually, but here they are handled in the same way. When a primitive is moving within a frame, its path creates a volume. When adding such primitives to the acceleration grid, the primitive is added to every cell that include this volume (figure 6).

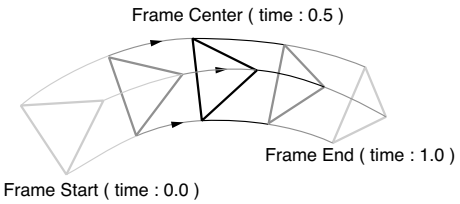


Figure 6: Motion blur object

2.2.4 Improved motion blur object

There are two issues that need to be dealt with in order to handle motion blur objects efficiently. One is the methodology for optimizing the size and path of the volume to be added to the acceleration grid, and the other is handling the intersection detection with the volume. When the object is moving over a large distance, the volume of the path of the object also increases in proportion. This means that such objects need to perform intersection detection with more rays. This is a problem when computing whether a ray has intersected with the object at a specific point in time; extra intersection computation must be done even though most of the volume has no chance of intersecting with a ray.

To get around this problem, Kilauea subdivides the object path volume into several subvolumes based on the camera shutter release time. When a subvolume is added to the grid, the time information for that subvolume (start/end time) is added as well. If the time for a frame is between 0.0 to 1.0 and divided into four segments, the first subvolume specifies the volume between 0.0 to 0.25, second subvolume 0.25 to 0.5, and so on. Rays contain time information when being rendered with motion blur. During ray tracing, the time information of the ray and the object volume inside the cell can be compared to efficiently perform intersection detection at a specified time. For example, if the ray specified the time 0.3 in the above example, only the second subvolume from 0.25 to 0.5 need to be used for intersection calculation (figure 7).

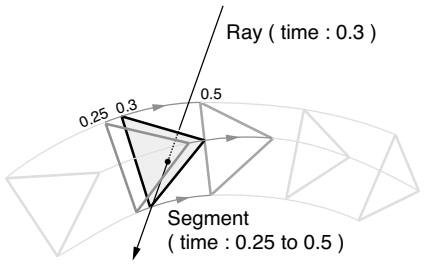


Figure 7: Optimized intersection calculation

The method of dividing the volume into subsections is also effective for optimizing the size and path of the volume registered in the acceleration grid. When a primitive is moving along a complex curve, describing its path as a single volume is inefficient. Such volumes can be optimized by creating a minimal volume for each subsection of time.

2.2.5 Obsoleting leaf node rule

Grid level of the acceleration grid increases when many primitives are added to the same region. A motion blur object that travels a large distance has a large volume that include many acceleration grid cells. Therefore, to add such primitives into a region that employs the leaf node rule, large numbers of leaf node cells will be required. For example, say that a single cell is subdivided into 4×4×4 sub grids. If the volume of the path of some primitive completely encloses that cell, that primitive will be registered into all 64 cells of the sublevel grid. Even triangles without motion blur has the possibility of using a maximum of 16 subgrid cells in the worst case. Initial implementation of Kilauea's memory management system had a tendency to use too much memory in such situations, and increase in memory usage were disproportionate to the increase in grid levels. Memory was being used less effectively as scene complexity increased, and in the worst case memory was being exhausted by one level of increase in the grid level even though almost no memory was being used at the previous level. One workaround for this problem is to limit the grid depth, but this increases the number of primitives registered in a single cell, significantly decreasing the intersection detection efficiency of ray tracing.

To allow objects that cover many cells such as motion blur objects, tree nodes were modified to allow adding of primitives at non-leaf nodes. This kept the primitive vs. cell usage ratio down to a certain level. In the previous example, when the motion volume of a primitive completely covers the cell and that cell needs to be subdivided, the primitive is not re-registered into all the subgrid cells but rather left in the current level. In other words, nodes in the grid trees maintain their functions as a cell even at the non-leaf levels, and at the same time function as a subtree parent (figure 8).

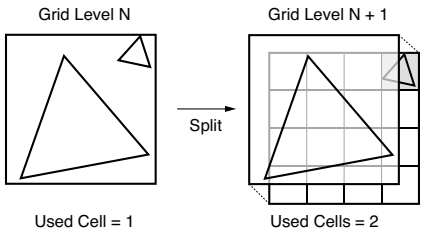


Figure 8: Multi-level rule

This suppressed the memory usage increase to the minimal level, even when the scene contains objects with large motion. Also, by allowing primitives to be registered in non-leaf nodes, large primitives are added to a level close to the root and smaller primitives are added to a level close to the leaf. This has the effect of the primitive size being reflected in the tree depth to a certain degree, eliminating the extreme memory usage increase depending on the grid tree level specification.

2.2.6 Multi-level traversal

With the elimination of leaf node rule from the acceleration grid, the ray tracing methodology must be changed as well. Basically, the traversal starts from the largest cell that include the ray's origin and contain a primitive, going down to the lead nodes in order. Intersection detection is done at the intermediate nodes if there is a primitive registered. If the leaf node traversal hits a grid boundary, then the traversal backs up to the previous level, and similar computation is repeated.

When the two acceleration grid methodologies are compared, the grid that doesn't use the leaf node rule has an overhead of few percent in terms of ray tracing efficiency assuming that the limit on tree depth is the same. This is because more intersection calculation being performed due to the primitives that cover a large number of cells being registered at the higher grid level. However, because memory usage was improved to reduce the memory footpriny by as much as 80% for motion blur objects and 20% for static objects, it is possible to create a tree with greater depth with the same memory size, improving the overall ray tracing efficiency.

2.2.7 Issues on implementing other primitives

Multi-level rule grid without the leaf node rule also has advantages when handling primitives that need to be adaptively tessellated at rendering time. Examples of such primitives are non-polygonal primitives such as NURBS surfaces and primitives with displacement maps. Only the rough bounding volumes of these primitives are known prior to tessellation. Therefore these primitives tend to use the acceleration grid cells in a similar way as the motion blur objects. Also, these primitives use a massive amount of memory if the data after tessellation is retained, requiring caching as needed. Efficient caching can be done by creating a separate small grid that register only the post-tessellation primitives, and adding and removing that primitive as a single item to/from the parent grid.

If such subgrid registration were to be performed on a grid that employs a leaf node rule, problems such as the rapid increase in memory consumption occur depending on the subgrid size to be registered and level of subdivision of the parent grid. This is similar to the problems with the previously mentioned motion blur objects. Multi-level grid can handle these cases in a flexible way.

3 Threading improvements

Kilauea is structured as a complex, multi-level pipeline. All tasks are described as messages, and these messages flow through various stages in the pipeline. At each stage, the messages are processed to execute their responsible computation. Implementation of this pipeline is closely tied to how the parallel processing is performed. This determines the overall system performance, as well as the extensibility and the ease of code maintenance and debugging.

This section describes the implementation of pipeline processing, which is the heart of Kilauea.

3.1 Kilauea pipeline synopsis

As previously mentioned, Kilauea executes extremely complex pipeline. Let us have a look at an example of Kilauea pipeline processing with the case of a simple rendering (figure 9).

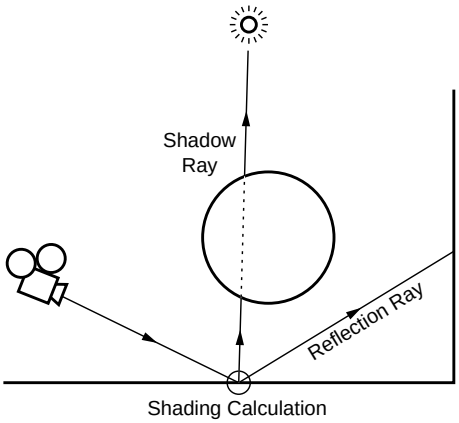


Figure 9: Simple rendering

The ray emitted from the camera traverses through space to detect the intersection with an object. If it hits an object, a shading color is computed on the surface. If the object surface has a specular property, a reflection ray is shot and another ray tracing starts. There will be a similar shading computation involved at the intersection of the reflection ray. In some cases, the shadow ray shooting may be necessary to detect if the surface is in the shadow. There may be a case where multiple reflection rays, instead of only one, are shot from the surface.

The above procedure is executed for all samples in all pixels of the image to render the whole image. The rendering computation is in turn enormous amount of space traversal computations and shading computations at the intersection of rays and objects.

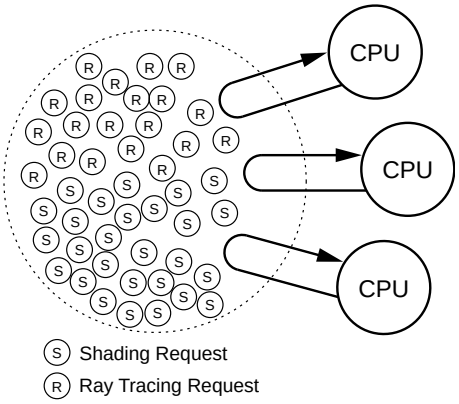


Figure 10: Unit of parallel processing

In Kilauea, the unit in parallel processing are based on ray tracing and shading. Every ray tracing computation is independently parallel processed from each other. Every shading is also independently parallel processed from each other. A ray tracing and a shading computation are also independent.

Because enormous amount of ray tracing and shading requests make up a typical rendering, these tasks can be distributed to CPUs of multiple machines to be processed in parallel and increase the computation speed. Because one parallel processing unit is a very small computation, all CPUs can receive almost equal amount of work to process, resulting in a near-optimal load balancing

(figure 10).

To implement these procedures, the data structures to represent such small parallel processing unit are created. To be more specific, ray tracing request to represent one ray tracing, and shading request to represent shading computation at object surfaces are defined. These data structures are passed as messages between computation tasks. The data structures have all the information needed for the computation, and thus each computation task can take them as inputs to execute the computation.

This data only represents the information for one computational task. For the rendering of the entire image, more data structures are necessary. The basic computation flow of Kilauea is that a data item is stored in a queue, the method to process it takes out one data item at a time, process it, and finally attach the result to another queue. Processing one stage is equivalent to some computation getting executed, and relevant data structures getting updated accordingly. The processed data structures are output to new data structures. For example, the computational result of the ray tracing request data structure is placed in the ray tracing result data structure. Similarly, the result for a shading request is placed in the shading result data structure. In a typical rendering, the shading color computation is invoked according to the result of ray tracing, so the ray tracing result is converted to the shading request. The camera ray computation typically goes through data structure changes as shown in figure 11. In reality, this data processing is further broken down to smaller data processing, and the actual steps in the computation is way more complex. One of the characteristics of Kilauea is that a certain computation consists of this multi-procedural pipeline.

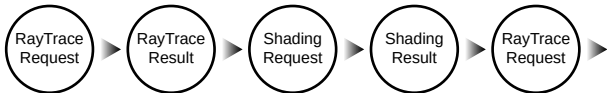


Figure 11: Data processing flow

3.2 Pipeline procedures and parallel processing

The target execution environment of Kilauea is multi-CPU systems. This section will explain the execution of this multi-level pipeline from the viewpoint of the execution efficiency on multi-CPU systems.

Describing all computations as a single procedure is possible. This corresponds to implementing all computations in a single sequential process. However, this will not improve performance on multi-CPU systems because the task itself is not structured to be processed by multiple CPUs at the same time.

How to assign procedures to each CPU in a multi-CPU system is extremely important to keep it constantly busy. In the case of Kilauea, the procedures are broken down into multi-level pipeline, and thus the assignment of these stages determines the efficiency in CPU usage.

In reality, the algorithm used in each stage of the pipeline and the actual execution environment can be considered separately, and the optimal parallel processing design varies from case to case. The Kilauea Research Project experimented on diverse parallel processing methodologies and they are explained below.

3.3 First threading methodology: threads as stages

The implementation of the parallel pipeline processing has gone through a drastic change in the course of four years of development. The idea used in the latest version is completely opposite from the first one.

In the early development stage after the initial architectural design of the multi-level pipeline, one thread was simply assigned to process one pipeline stage (figure 12). This idea expresses the pipeline algorithm in the most straightforward form. We had already decided to implement threading on multi-CPU using Pthread, and Pthread standard has the priority boosting mechanism to change the execution priority of each thread. Our first intention was to use this priority boost to control the thread execution in high precision.

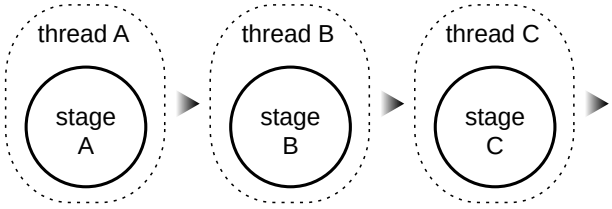


Figure 12: Threads as stages

The pipeline stages were designed with the focus on the following:

1. parallel processing elements
2. algorithmic structure

The first one mainly corresponds to the case where one machine sends data to multiple machines. After some sequential computation, its result may have to be sent to multiple machines. This communication can be written sequentially by processing the transmission of data to each machine one by one, but parallel processing is logically more intuitive in this case. Thus, data transmission stage is separated, and each stage only handles the transmission to the machine it is responsible for. By structuring the stages in this way, algorithms are expressed in a more intuitive way and the slow-down by such causes as blocking I/Os in each thread can be investigated independent of each other

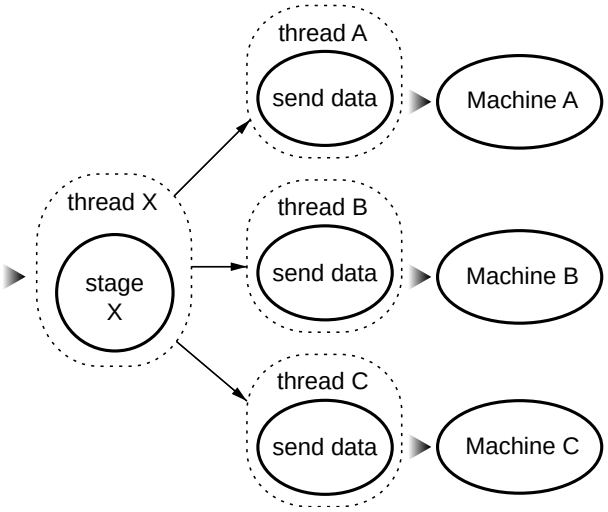


Figure 13: Parallel Processing Stage Structure

The second one intends to design the stages using the algorithmic structure, and the premise for this distinction is often quite vague. The expected benefit of this methodology is that by separating

a sequential operation into multiple stages, a speed-up may be obtained by their parallel execution. The stages were designed at the logical breaks in the computation. By designing the stages this way, we believed that if the pipeline depth, that is, the number of stages, were greater than the number of processors available, the benefits of parallel processing can be obtained easily.

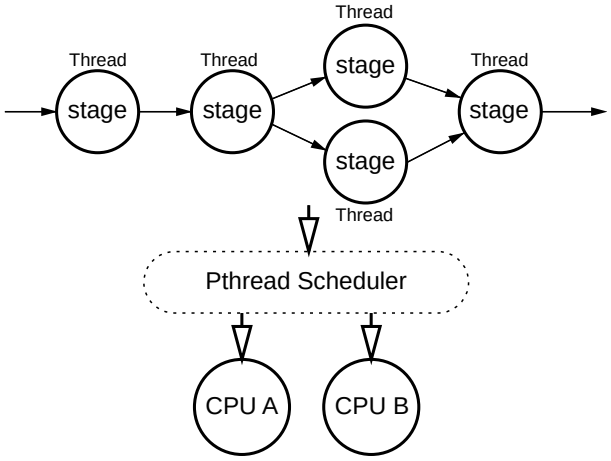


Figure 14: Threads Assignment to CPUs

As a result of this design, a very large number of stages were created. The result of a single computation is obtained by passing through these stages in turn. Because each stage is handled by separate CPUs, the benefit of parallel processing can be obtained.

3.4 Problems with the first implementation

In the end, we were unable to obtain the expected parallel processing performance. Although the computation itself was executed correctly, we were unable to utilize fully all of the CPUs of a multi-CPU machine.

The following factors contributed to this problem.

3.5 Unbalanced loads of stages

In order to efficiently execute a pipeline composed of multiple stages, the load at each stage must be close to equal. If the computation load at a certain stage is excessively high, intuitively this stage becomes a bottleneck, leading to reduced performance (figure 15).

The load of each stage must be estimated and designed such that there are they are balanced. The estimation itself is difficult, and completely balancing the load is inherently impossible. As a result, the load between the stages inevitable become unbalanced.

As the computation is executed, there are cases where the changes in the situation break down the balance, causing one stage of the pipeline to become overloaded. An example of this is the shading computation, whose load depends on the type of shading being performed. As the execution progresses and the shading type on the surface hit by the ray changes, the shading computation itself changes. This has the effect of changing the load of a particular stage within the pipeline. Without exception, every image rendering task contains this uncertainty factor. This shows that there is a stage whose load is uncertain until execution time, and thus load-balancing cannot be done statically prior to execution.

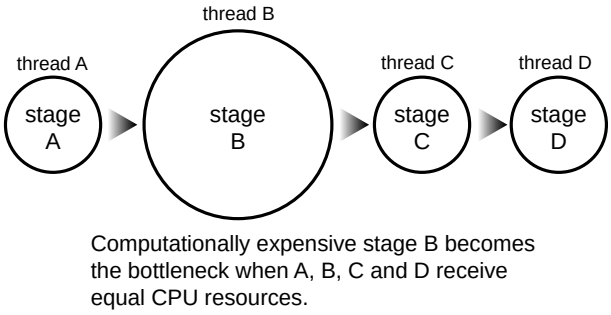


Figure 15: Load of stages

3.6 Execution order of stages

Execution order of threads cause problems as well. Thread schedulers attempt to switch threads intelligently when there are more threads than there are CPUs. The order in which CPU time is allocated to threads is dependent on the scheduling policy internal to the threads themselves, and this policy is very difficult to control directly from outside. This problem can also be illustrated in an intuitive way. Suppose there are three threads A, B, and C, and each thread processes one stage of a pipeline. Assume that the pipeline is executed in the order of A > B > C (figure 16).

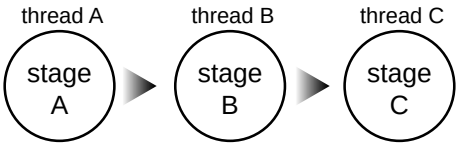


Figure 16: Stage A, B and C

If there are more than three CPUs, and each thread is constantly ready to be processed by the CPU, then this pipeline is operating in an ideal way, and maximum performance is possible (figure 17).

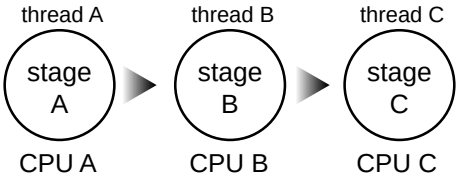


Figure 17: Number of CPUs is enough

If only two CPUs are available, however, one of the threads do not have a CPU available at any time. Thus, only a part of the pipeline is operating at any time (figure 18).

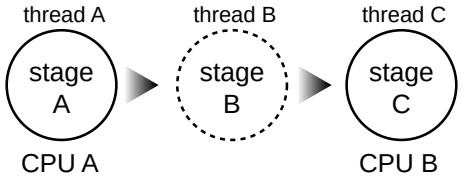


Figure 18: Number of CPUs is not enough

Ideally, the computation is performed in the order A > B > C, and the thread execution should be scheduled in such way. Unfortunately, the default execution scheduling of Pthreads will not behave this way. As a result, queue structures between each thread will function to absorb this scheduling problem. The queue must have enough depth to be able to hold the messages no matter what order thread A, B, and C are executed in (figure 19).

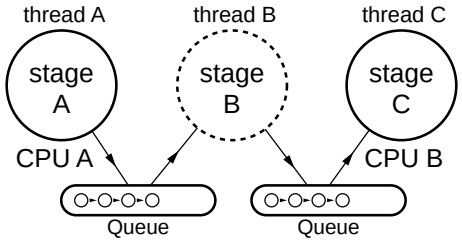


Figure 19: Queue between stages

If there are enough data in the queues, no fatal problems will occur no matter what order A, B, and C are executed in. However, the time required for a message to pass through stage A and finish at stage C will increase. In other words, the latency will tend to increase. Kilauea is able to diligently continue executing even in high-latency situations, but ideally the latency should be short. A breakthrough was necessary.

3.7 Dynamic assignment of thread priority

The problem is complicated further when the load of stages A, B, and C, change dynamically. Suppose that the load of thread B doubled at some point. Prior to the change the CPU power was balanced by allocating exactly 1/3 of the power to each thread (figure 20). This changes so that threads A and C are allocated 1/4 each, and thread B 1/2 (figure 21).

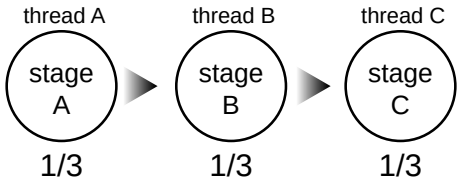


Figure 20: Balanced load

If execution were to continue with equal allocation of CPU time to each thread, A and C will be processed quickly as result. In other words, the output queue of A will grow while the input

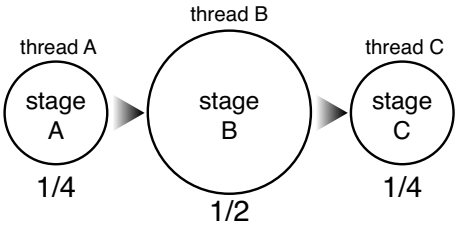


Figure 21: Unbalanced load

queue of C will shrink. If this continues to an excess, A will need a large amount of memory in order to write its output into the queue. When the memory request exceeds a certain threshold, the CPU will then become busy in order to handle the memory allocation. This reduces the CPU power available to handle the original computation, reducing the processing speed significantly (figure 22).

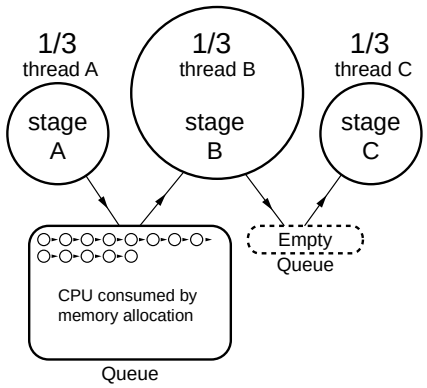


Figure 22: CPU consumed for memory allocation

Such problems occurred frequently in the initial experimentation stages of Kilauea. The most effective solution to this problem were to put the thread to sleep once the output queue size reaches a certain threshold (figure 23). More on this topic can be found in [7].

Using this technique forces a thread whose execution is progressing too quickly to be paused, allocating that CPU power elsewhere. This is very effective at circumventing worst-case situations, but unfortunately does not help to increase overall performance. This method is basically a very passive solution where a thread gives up its allocated CPU time to other threads. A method that fine-tunes the thread scheduling directly is desirable.

Pthread allows a thread's execution priority to be changed through a mechanism called priority boosting. We can consider an improvement where in the previous example the priority of thread B is boosted above the other threads, maintaining the optimal execution efficiency. However, changing the priority of threads in this way is effective when the load of the thread is already known, but it is difficult to control when the load itself changes dynamically and measuring the changes precisely is difficult. This made controlling the pipelines inside Kilauea using priority boost complex and difficult. Despite several experiments, we were unable to achieve good results.

Based on these findings, we needed to consider a more sophisticated way of parallelizing the pipelines, especially the thread scheduling.

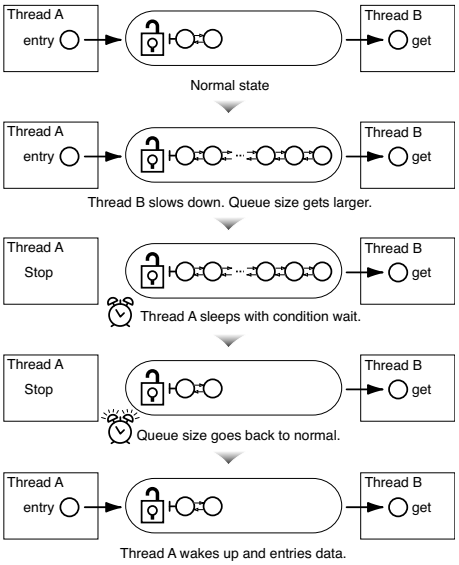


Figure 23: Auto Cruising

3.8 Other problems related to threading

Other than the problems caused by varying loads of stages and thread scheduling, threading by stages raised other problems.

Fundamentally, the pipeline consists of stages with queues in between. Every stage needs to access the queue frequently. Under a multi-CPU environment, stages running within different threads could be executing on different CPUs. This means that they really are executing in parallel in a real timeline. In order to protect the coherency of the data inside the queue in such situations, the queue itself clearly needs to be implemented in a multi-thread safe manner. For example, locking/unlocking mechanisms for mutual exclusion and condition waiting mechanism to allow the thread to run efficiently must be implemented.

Factors that cause problems in terms of efficiency are the lock collisions and race conditions which occur when a thread that was waiting due to the condition lock mechanism is woken up. When multiple threads are frequently accessing the queue, handling lock collisions become an issue that can't be ignored. In our case, we have made improvements in places where lock collisions were reducing performance significantly by using try-locks and other experiments. Additionally, we have created special queues that avoid race conditions as much as possible and optimized its implementation. However, performance improved by only a few percent. Despite the improvement, it was clear that we solve this problem at a more fundamental level. In other words, modify the implementation so that it can execute without locks in the first place.

Allocating stages to threads also meant that a change in the thread code has a direct impact on the number of threads created and their execution order. This means that the trial and error process of merging dividing threads to change the number of threads involves an extremely complicated coding. Because this part is a very important part the determine the final computation efficiency, the difficulty of this work became a huge obstacle. We needed to somehow be able to easily perform the various trial and error process.

3.8.1 Consideration of threading by stages

With these considerations in mind, fundamental improvements in executing the pipeline as threads were implemented.

The following goals for improving threading were set.

- Dynamically follow the change in load of stages
- Full control in stage execution schedule
- Minimize locks on queues between stages
- Minimize latency
- Fast and stable execution
- Ease of debugging and profiling

After making major changes with the above goals in mind, the current Kilauea pipeline engine was implemented. The goals were almost fully met, and runs very stably compared to the previous version.

The details of the current implementation will be explained.

3.9 Improved threading methodology

Let us look at the problem from a completely different perspective. Solving the scheduling problem while making every stage of the pipeline into a separate thread requires the deep intervention into the thread implementation itself, which is undesirable. Also, this will not fundamentally solve various problems associated to threads such as locking overhead.

3.9.1 Thread as an engine

Under the new idea, stage execution and threads are considered completely separately. A threads is treated as an engine for computation, and stages inside the pipeline as a job that the engine must process. The stages are connected by a queue data structure.

A thread is an engine for computation, so under a multi-CPU environment, the number of engines launched are as many as the number of CPUs — i.e., launch two engine threads to execute the computation on a dual CPU machine. Kilauea restricts these threads to handle only the main pipelines, however, so in the end a Kilauea process executing on a given machine will launch number of CPUs and additional three to four threads. These extra threads mostly handle tasks such as processing the MPI layer communication or exchanging control commands with the master Kilauea process. What is important is that the main image rendering pipeline are executed with the number of threads restricted to the number of CPUs available.

Say that some pipeline is composed of three stages A, B, and C. An engine thread will try to execute all of these stages. Inside Kilauea there exist many messages to be executed by this pipeline. If multiple engine threads are launched under a multi-CPU environment, these threads will be able to handle these messages in parallel (figure 24).

3.9.2 Execution of stages

Every pipeline stage was implemented as a class, and each stage executed its task by passing messages around. This made modifications very easy to do. An engine thread need only call the stages of the pipeline in turn. If only the top-level structures needed modification, the internal implementation did not need to be touched. Each stage were implemented as a part of a pipeline that read the input message contents, perform computation on them, and output the result. The various stage execution can be looked at from two perspectives. This can also be considered to be the stage execution scheduling problem.

1. In what order will the stages be executed

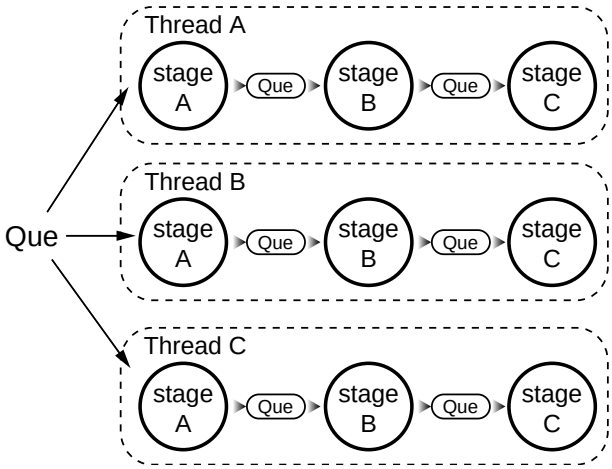


Figure 24: Threads as engine

2. How long each stage will be executed

In what order the stages will be executed can be controlled with a very clear policy. This is because all the stages have a clear order in which they should be executed, and the stages are divided based on that information, making complete control possible.

The second problem of determining how long each stage will be executed can also be solved by watching the input and output queue depth of a stage and controlling the stages so that the queue depth are optimal.

After some thought, it is also possible to think of an implementation that solves both of these problems at once. Each stage of the pipeline will be executed at a higher priority than the previous one (figure 25). When the engine executes each of the stages with this policy, the internal queue will only contain a single item. Therefore, execution will proceed with the smallest latency.

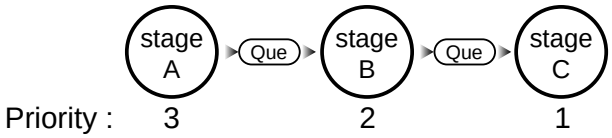


Figure 25: Priority of stages

In the example above with three stages A, B, and C, the stages are executed in the order $A > B > C$, but their priorities are $C > B > A$. In other words, the engine thread executes the C stage with the highest priority, then B, and finally A. By implementing the stage scheduling in this way, messages sent to A will reach C as fast as possible, achieving minimum latency as a result.

Depending on the computation involved at each stage, sometimes handling individual messages as a group of multiple messages, rather than actually handling them individually, is more efficient. In such cases, modifying the stage scheduling while considering this characteristic is not difficult at all. Kilauea dynamically switches the scheduling algorithm depending on the situation. This feature is an extremely important element in determining the rendering efficiency, and fine-tuning via thorough analysis yields worthwhile performance improvements.

3.9.3 Independency of engines

Stages A, B, C exist within each engine thread, and a queue data structure exists between each stage. In the initial implementation of Kilauea, such queue structures required a multi-thread safe mechanism. With the integration of the stages into single engine thread, thread safety is no longer needed. This frees the stages from any of the problems specific to threads such as locking/unlocking or lock collisions. However, the parts of the engine thread that retrieves messages from outside must be thread safe, with the attention to the possibility of multiple engine threads being launched simultaneously. From the perspective of the entire system, however, this is only the entry and exit points of the pipeline, which allows the logical structure to be much simpler, resulting in faster execution (figure 26).

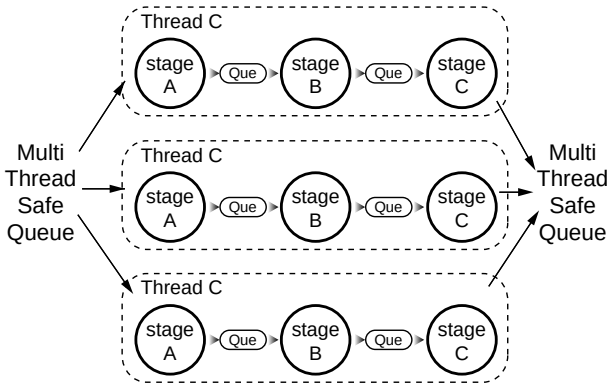


Figure 26: Independency of engines

The multiple engine thread that will be launched will have a dependency as far as the input and output queues are concerned, but they have no dependency whatsoever as far as the execution of their internal processing goes. Compared to the previous thread execution environment, the length of real time that one thread can execute without having any dependency on other threads has increased significantly. This means that the independency between the threads are extremely high, and once an engine thread starts operating on a message, that computation will be performed totally independently inside the thread for a long amount of time. This is a very effective trait in terms of parallel processing efficiency, allowing an ideal distribution of computation to multiple CPUs.

As a result, threads are able to continue executing rapidly in a stable way, greatly increasing the computation efficiency.

3.9.4 Debugging and profiling environment

Debugging and profiling environment acquire significant rewards from the new threading strategy as well. The computation within a single engine thread is a sequential program, which allows developers to take advantage of this characteristic to debug effectively.

Whenever the developers suspect a problem inside the engine thread, a program to isolate this thread from Kilauea is used to debug under a single engine state. This program is single threaded, which makes it a lot easier to debug using ordinary debugging methodology.

Also, the optimization of the logic made by profiling and analyzing this program is directly applicable to the actual Kilauea runtime environment. And because the engine threads are highly independent of each other, the results of the optimization is multiplied by the number of CPUs available when running in a multi-CPU environment. Unfortunately, we are using only dual-processor machines, so we must perform tests on systems with large number of CPUs, such as

16-CPU systems, to verify whether Kilauea can achieve similar results on such systems. However, even in such environment, the new threading strategy will perform better than the previous one, without a doubt.

3.9.5 Results of the improvements

As a result of these improvements, Kilauea's main pipeline processing efficiency has increased by a factor of seven to ten compared to previous versions. Also, CPU load is sustained at almost 100% on all CPUs in a multiprocessor system, showing that multiple CPUs are being used fully. We also verified that the system runs in a very stable way under any kind of scene.

4 Multi-pass rendering

Kilauea renders images by reassembling the information from multi-pass computations. How Kilauea executes this multi-pass rendering is explained in this section.

4.1 Rendering stages

Currently, Kilauea executes the rendering computation in four main stages:

0) Scene data construction;

1) Photon tracing;

2) Final gather estimation;

3) Final rendering.

Each stage is completely independent of each other, and the rendering is executed in the order from 0 to 3. However, it is possible, for example, to interrupt the computation in the middle of the final gather estimation stage and recompute it, or after the final rendering, start over from the photon tracing (figure 27).

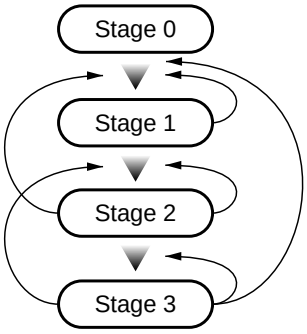


Figure 27: Rendering stages

This feature is made possible by Kilauea's basic design that it stays resident after computing one frame, which is especially effective when adjusting the materials of objects in the scene. Kilauea's resident design allows users to repeatedly re-render while tuning parameters with high responsiveness. The computation precision while rendering to adjust shaders or to get the final image is the same. Therefore, the qualities obtained during the shader adjustment process and the final rendering process match exactly, enhancing the productivity.

Each rendering stage is designed and implemented with full attention to parallel processing in mind. At the rendering stages, appropriate parallel processing approaches are taken depending on their problem types, and the resulting overall parallel performance is satisfactory.

The specifics of each rendering stage are explained in the following sections.

4.2 Stage 0: Scene data construction

Kilauea receives the scene data from Maya frontend in ShotData format (refer to [7]). This data is written as the incremental data, so the scene to be rendered must be reconstructed in the system. Also, such scene data must be converted to acceleration grid structure in order to perform ray tracing. The details of the scene data construction stage has already been explained in section 2.1 and 2.2.

The computation in this stage is the most difficult to parallelize compared to the other stages. The parallel processing methodology that Kilauea ultimately adopted is very different from the ideas used in the other stages. This is because the computation itself is inherently not suitable for parallel processing. However, as far as the performance of distributing the scene data is concerned, Kilauea has achieved the performance and stability required for the production use, and the algorithm has undergone many improvements, as previously mentioned.

From the perspective of generating images, the sequence of computation from reading scene data, reconstructing the scene based on the incremental data, distributing of the scene, and constructing the acceleration grid structure can be grouped together as the "scene data construction" stage.

In Kilauea, internal scene data and the acceleration grid data converted from it remain inside the system without being freed, even after the rendering of one frame is complete. Therefore, in the next rendering operation, such data do not need to be updated, nor does the scene need to be read again. Viewed this way, stage 0 "scene data construction" can be considered as executed only once when computing a new frame.

4.3 Stage 1: Photon tracing

When the scene data assembly stage completes, Kilauea can start executing the photon tracing. In this rendering stage, basically large numbers of photons are shot from the light sources, and then their behavior is observed in the scene using conventional ray tracing. In the end, photon map data is constructed and irradiance precomputation is performed on this photon map data (refer to [6]).

4.3.1 Parallel processing

The photon tracing computation shoots enormous amount of photons from the light sources, and the photons are traced through their paths. The tracing algorithm is algorithmically the same as ray tracing. Kilauea uses its ray tracing engine for photon tracing as well.

Fundamentally, photons are completely independent of each other, and therefore the path tracing computation can be performed independently. This implies that photon tracing is extremely suitable for parallel processing.

Kilauea executes the photon tracing in parallel. In other words, path tracings on the massive number of photons that exist in the scene are executed completely independently on multiple CPUs (figure 28. If a photon hits an object, then the photon behavior on that object surface is computed. The shading computations for photons hitting this surface is also independent from each other, and thus executed in parallel. This parallel mechanism is not just restricted to photon tracing, but to every ray tracing computation.

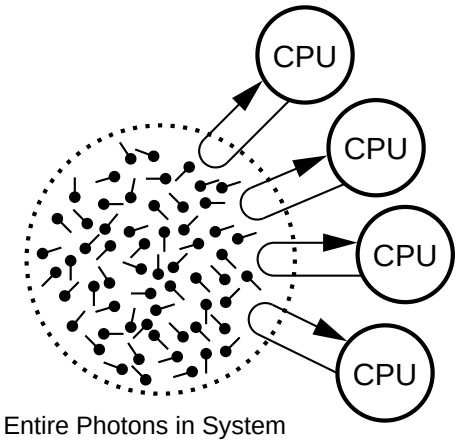


Figure 28: Parallel photon tracing

4.3.2 Computation result

Photons that end up being stored on some surfaces actually get stored in the kd-tree data structure.

If all photons to be stored completely fit in one machine, then all of the photon data will be stored completely on that machine. In the end all photon map data on all machines becomes identical. This is the simplest case.

If all photon data cannot be stored in one machine, the one photon map is stored across multiple machines. The actual photon look-up will access the photon data distributed across multiple machines in a clever way, performing the lookup as if the data were a single photon map. Please refer to [6] for the details on the ideas used for accessing the photon map data distributed across multiple machines.

In our experience, distributing the photon map is a very special case. In almost all cases, a photon map is small enough to be stored in one machine.

In the actual photon tracing stage, all photons are independently computed on multiple CPUs on multiple machines. To look at it from another perspective, each CPU will be performing a photon tracing on the subset of the entire photons. At the same time, there is no way to know about the result of a photon tracing that another CPU computed. Photon tracing is a completely independent operation. Taking advantage of this characteristic, an implementation that has an extremely high level of parallel processing independency and at the same time exert maximum parallel processing performance is possible.

However, to finally construct the photon map data, photon data other than the ones that one CPU computed will be required. When considering the options for how to store the photon map data inside Kilauea, if the photon map fits inside one machine, then collecting all the photons from other CPUs will be necessary. Exactly which photons must be collected will vary from machine to machine. In order to construct an identical photon map data on each machine at the end, somewhat complex photon data exchange will be necessary.

Kilauea handles this part as follows:

1. Store the photon tracing results that was computed locally into the machine's local photon map.

2. Distribute the photon tracing results to all other machines holding other photon maps.

3. When the photon tracing results are received from other machines, treat the data as if it were computed locally, and store it in the local photon map.

4. When all photon maps have finished computation, photon map on each machine should have an identical copy. Start the irradiance precomputation on these photon maps.

Simply put, the algorithm distributes the local photon tracing results to all other machines (figure 29).

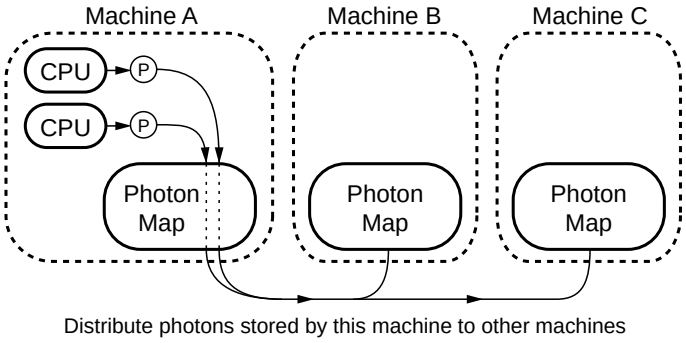


Figure 29: Photon distribution

When photon maps need to be shared across multiple machines, a more complex distribution processing is required, but the basic idea that the locally computed photon trace results are distributed to others remains.

4.3.3 Independency from other stages

Basically, in every image generation, photon tracing stage is required immediately after the scene data construction stage. There are at times where photon map reconstruction is not necessary when computing the next frame. For example, if only the camera is moving, then it is possible to use the photon map generated previously. Such requests are controlled by whether this photon tracing stage is executed or not.

When interactively creating the materials, photon map reconstruction may be desirable. For example, to get a rough idea of an image the number of photons to be used will be cut down to 1/10th, increasing the number of photons in steps.

Also, the number of photons required to get enough precision in the final rendering can be speculated once all photons are shot, so the trial-and-error process only needs to be repeated a few times to obtain the near-optimal number of photons.

To handle such requests, Kilauea's ability to recompute the photon tracing stage without restraint is invaluable.

4.4 Stage 2: Final gather estimation

After the stages up to photon tracing is complete, Kilauea can then proceed to execute the final gather estimation computation.

Initially, this stage did not exist in Kilauea. The computation performed in the final gather estimation stage was previously performed inside the final rendering stage entirely. However, the final rendering stage turned out to be too slow in many cases, and a separate stage to speed it up was necessary.

4.4.1 Motivation

Thorough analysis of the final gather values over images confirmed that the changes in the final gather values are smooth in most areas. This observation implies that the sampling rate required

for accurate final gather values and the sampling rate required for the direct illumination, are extremely different. With this in mind, the final gather estimation stage aims to compute only the final gather values efficiently. Greg Ward's irradiance cache[9] is the starting point of this final gather estimation algorithm.

4.4.2 Mechanism

There are situations where computing final gather values in a separate stage will not make a difference compared to computing them in the final rendering stage. In some cases it is even advantageous to compute final gather values in the final render stage. In the current implementation of Kilauea, the emphasis is on computing final gather values in an efficient manner. If for some reason the final gather values cannot be computed or the computation is terminated in certain regions, they will be computed in a normal way at the final rendering stage. Therefore, no harm will be done to the final rendered image.

The final gather values generated in this stage are computed with the attention to the screen space coordinates of the object surface. In the end, they are computed as on-screen pixel values. This allows Kilauea to avoid computing invisible final gather values as much as possible. If the image is simple in terms of global illumination, the computation cost will be low. On the other hand, computation cost will be high for a complex scene. The final gather values are not computed at a smaller unit than a single pixel. There are cases where final gather values need to be computed at a finer resolution than a pixel. However, the goal of this stage is to compute final gather values in an efficient manner, so such computation will be terminated and resolved at the final rendering stage.

Please refer to [8] for more detailed explanation regarding the final gather estimation stage.

4.4.3 Parallel processing

The final gather estimation stage will of course be computed in parallel in Kilauea. Since the computation will proceed in a manner that is dependent of screen space coordinates, the parallel computation also focuses more on screen space, compared to the methods used in the photon tracing or final rendering stage. The current implementation divides the screen into small rectangular buckets, and individual buckets are computed independently (figure 30).

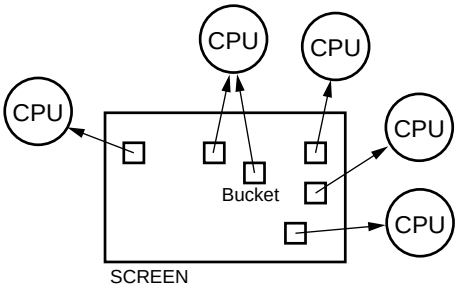


Figure 30: Parallel processing by bucketing

The final gather estimation computation inside each region needs to perform final gathering, so in this sense Kilauea's normal parallel ray tracing computation is used.

The computational cost of final gather estimation stage depends on how the scene appears within the bucket of the screen each processor is in charge of computing. Therefore the total computation cost varies between different regions. At times, when a certain region is very complex, only one CPU in charge of this region may be caught up, while other CPUs complete their computation, causing CPUs to idle and waste resources when viewed as a whole rendering system.

However, in the experiments, no fatal load balancing problem occurred unless the bucket size was extremely large. When the bucket size is reasonable, the processing time for a bucket is short enough to keep the load adequately balanced.

4.4.4 Final gather estimation results

When the final gather estimation for one bucket completes, final gather values of this bucket are stored per pixel. If for some reason final gather estimation failed or terminated for a pixel, then a flag denoting that it must be computed in the final stage is stored. These values are recorded as on-screen pixel data. The final gather values computed in this stage will be referenced in the final rendering stage. However, the final rendering stage has no information about how each pixel was computed, what sampling method was used, on what CPU, in what order. The sampling order may be somehow fixed beforehand, but doing so disables Kilauea's flexible screen space sample scheduling in the final rendering stage. To avoid such limitation, current implementation of Kilauea distributes the locally computed final gather values in the bucket to all other machines. This is the same idea as distributing the photon tracing results. In the end, all machines will store one final gather value per pixel for the entire screen (figure 31).

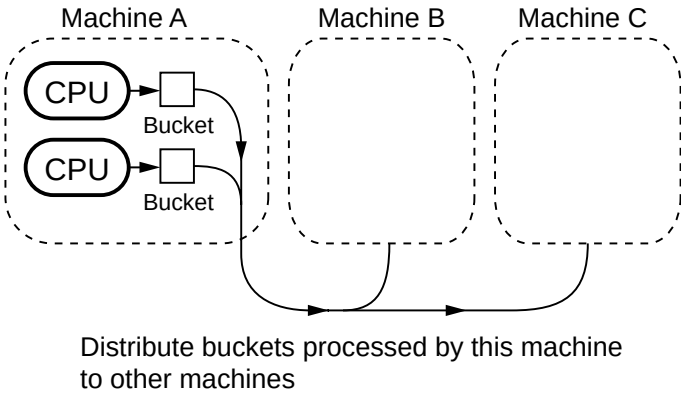


Figure 31: Distribution of bucket results

Currently, the data size transmitted in order to send a 1024×768 image is around 8 to 10 Mbytes. Because the time required for transferring this data is about 1 or 2 seconds using 100 BaseT network, this overhead is almost negligible. The goal of final gather estimation is to complete the estimation computation in around two to three minutes, so at present the network transfer time does not inflict any major problems

However, because the resulting data size of the final gather estimation stage depends on the screen size, greater image size leads to memory deficiency. Also, this data will be referenced in the final rendering time, but not all of the values need to be accessed simultaneously.

The access pattern depends on how the final rendering stage will sample the screen.

If the final rendering stage samples the screen in a completely random pixel order, the final gather values sampling also require that the database be sampled in a random order. However, if the screen were to be rendered using some rule, such as sample the screen from the bottom left to the upper right in scanline units, then the final gather values will also be sampled using this rule, following the sampling order of the screen pixels. Currently, it is possible to freely choose the final rendering pixel processing order from among several preset scheduling types. The ones currently being used in real usage all sample the screen according to a set rule, and do not perform random accesses on the entire screen. Usually the access concentrates to some area on the screen (32×32 pixels, for example).

For this reason, there is no need to keep the final gather estimation results resident on memory. Instead, the results are output to a file called *IAB* (Irradiance Array Binary). If Kilauea efficiently reads the IAB files in small parts, then Kilauea should achieve good processing speed. As a result, when the final gather estimation stage completes, IAB files are created on all machines as local data, which contains all the final gather values of the image as screen pixel values. In the end, identical IAB files will be created on each machine's local disk. The size of this file is approximately 10Mbytes for a 1024×768 image (figure 32).

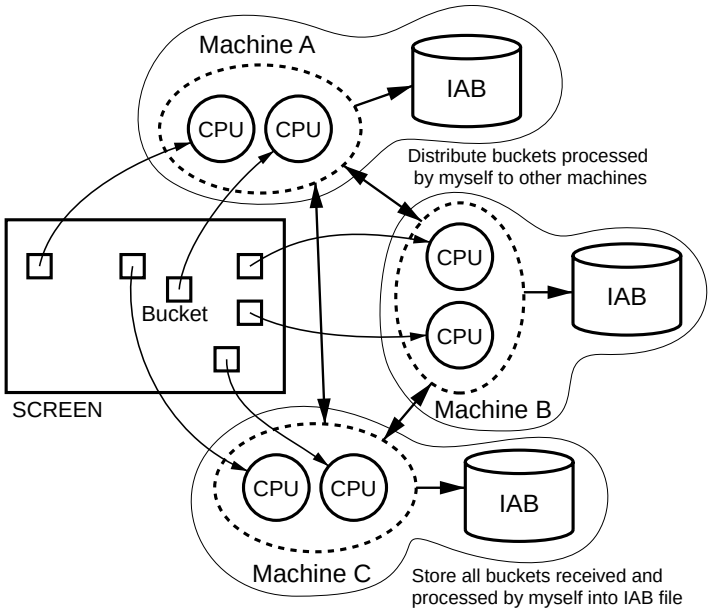


Figure 32: IAB local file data

4.4.5 Independency from other stages

As previously mentioned, the final gather estimation stage must be performed after the photon tracing stage. There are times when repeatedly executing this stage only is useful.

Final gather value calculation requires the information of the complete scene. This is understandable because the final gather values include distinguishing elements of global illumination, such as color bleeding. Because of this, when the surface attributes of one object in the scene is changed, the change does not only affect the color of the object surface but affect the whole environment around it as well. There is no need to take such situations into account in a direct illumination rendering, but must always be taken into account when using global illumination. Therefore, when an object surface attribute has been changed, the final gather estimation stage must be run again. In the same sense, the photon tracing stage must actually be executed again as well.

Also, allowing to compute this stage repeatedly without restraint is important in trial and error procedure to efficiently tune the final gather estimation parameters.

4.5 Stage 3: Final rendering

Once the final gather estimation stage is complete, it is finally possible to start the final rendering stage.

Final rendering stage generate the image using the stored photon map data and final gather values in IAB format that were computed in the previous stages. This stage basically executes all the elements of the rendering. In the current implementation of Kilauea, this stage performs the following computations:

1. Ray tracing
2. Shading
3. Final gathering
4. Photon look-up

Kilauea computes these tasks in parallel, using multiple machines. The basic unit of parallel processing is the ray tracing and the shading computation on the object surface. Enormous amount of ray tracings required to generate an image are independent of each other, and computed in parallel. Surface color computation will be performed on the object surface that was found as a result of the ray tracing, and here parallel processing will be performed independently per sample on multiple CPUs as well. Please refer to [7] on how the ray tracing is actually performed in parallel and the specifics on the shading computation themselves.

The following sections discusses how the values computed in stages 1 and 2 are referenced in this stage.

4.5.1 Referencing final gather values

Kilauea is a parallel processing renderer, but this does not change the basic image generation methodology from ordinary sequential ray tracers. Rays are shot from the camera, intersection with an object in the scene is computed, and if there is an intersection, the shading color at the surface is determined. During this computation, if a shadow ray tracing is required to compute the shadow, then another ray is shot toward the light to do a shadow determination. If a specular reflection or refraction is required, then an another ray is shot and the object surface color is determined once the results are returned. Additionally, an irradiance value is required on the object surface. This is the global illumination element on the object surface, which is typically faked by the ambient value in a direct illumination renderer. But in Kilauea the irradiance value is determined through light simulation. Specifically, the irradiance value is the sum of final gather values (irradiance from diffuse surfaces) and the caustic value (irradiance from specular surfaces) at a point on a surface.

The caustic value on the object surface is determined by referencing the caustic photon map at that point. This photon map has already been generated at rendering stage 1 (figure 33). Please refer to [7] for the specifics on the look-up, and refer to [8] for the global illumination mechanism.

The final gather estimation values computed in rendering stage 2 is referenced to obtain the final gather values. Final gather values are stored within IAB files, and an identical IAB file for the entire image has already been distributed to every machine. In other words, by looking up in the local IAB file, whether a final gather value has already been computed or not is determined.

As already explained, IAB files store final gather values in screen space coordinates. The final gather value at a certain point on an object is computed by first projecting the point to the screen space, and then looking up the corresponding coordinates in the IAB file. The look up operation returns one of two possible results:

1. Estimation completed normally, and a final gather value exists.
2. Estimation did not complete for some reason or another, and final gather value was not computed for this pixel.

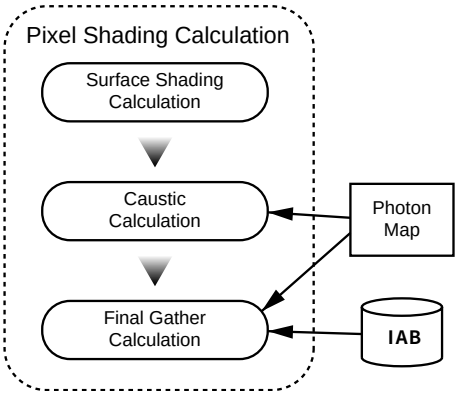


Figure 33: Pixel shading calculation

In the first situation, the final gather value for this surface has already been computed in rendering stage 2, so this value is simply used in the proceeding shading computation. This allows the computation to skip the final gather computation on that sample for the surface altogether, contributing greatly to computation speed-up.

In the second case, however, final gather values were not computed in the rendering stage 2, so final gathering must be done on that object surface to compute the final gather values.

In other words, the more pixels can be computed in the stage 2 final gather estimation stage, the less final gathering computation performed in the final rendering stage, reducing the total computation time.

The computational cost of final gather values in a pixel varies depending on the state of that pixel. Through experimentation, obtaining final gather values from the final gather estimation is known to be more efficient than actually doing the final gathering in an average case. This is because the final gather estimation stage tries to compute the final gather values on screen with the minimum number of samplings by considering the smooth change in the irradiance values, whereas the final rendering stage unconditionally performs final gathering for every sampling.

One important goal of the final rendering stage is to sample the direct illumination values, and the screen space sampling is performed in parallel at per sampling basis. The optimal sampling pattern here is very different from the sampling pattern optimal to the final gather estimation stage. Incorporating the sampling pattern optimal for the final gather estimation stage in the final rendering stage is possible, but this is not worthwhile for its high complexity and reduced parallel performance. Hence, the final gather estimation stage is separated from the final rendering stage.

Accesses to the IAB files complete within a single machine. If that machine was a multi-CPU machine, however, there will be simultaneous accesses from multiple CPUs to that file. But since this file is basically read-only, there is no need to implement anything complex. The current implementation structures IAB files as small tiles, and create a CPU local cache that can contain multiple tiles, and manage the cache independently for each CPU (figure 34). Tiles inside the cache update the timestamp every time it is accessed, and the least recently used tiles are thrown out first when the cache is full. Experiments show that accesses to IAB files are a very small percentage of the computation in proportion to the total computation in the final rendering stage, and satisfactory performance is achieved with a cache smaller than 512Kbytes.

4.5.2 Final gathering

If the final gather values on an object surface cannot be retrieved from the IAB data, Kilauea computes that value in the traditional way. This is as follows:

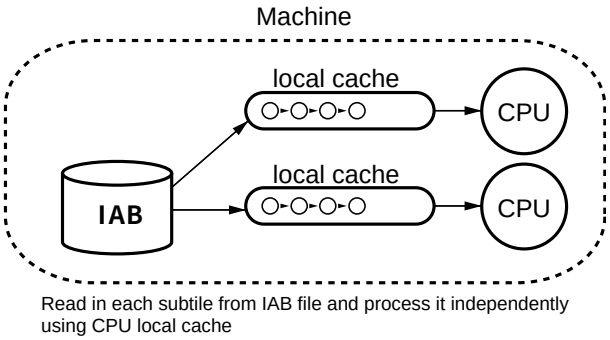


Figure 34: Accessing to IAB file

1. Start the final gather from that point
2. When the final gather ray hits an object, perform a photon map look-up at that point

There are many options regarding how the final gather will be performed. For example, shooting a specified number of rays randomly, or unconditionally shoot n×n rays at a specified resolution. In Kilauea, this is solely a problem of how the shader for the object surface is written. To put it in another way, users can implement final gathering in any way at the shader level.

The final gather rays being shot may hit an another object after performing a ray tracing. If it hits, then a photon lookup is performed at that point. Please refer to [7] regarding this photon lookup operation.

For the specifics on how global illumination should be implemented using photon maps, please refer to [8].

4.5.3 Final image

When all the samples have been taken from all the pixels on screen, the computation for that frame is finished.

If the user intends to adjust the materials of this scene at this point, he may proceed to tweak some parameters and start a rendering again. Depending on the parameter and its global illumination effect, the rendering may need to start over from stage 1 or 2.

If this is the final rendering, the image is saved to the disk and proceeds to the next frame, from stage 0.

4.6 Sampling efficiency and multi-pass rendering

Many levels of computation are required to render a single image. As explained so far, Kilauea divides the rendering into four stages. The following is the summary of four stages and their objectives.

1. Stage0: Scene data assembly

   Refresh the scene and perform any necessary pre-computation

2. Stage 1: Photon tracing

   Sample the scene from the light source

3. Stage 2: Final gather estimation

   Sample the scene from the camera, while performing sampling optimized for global illumination.

4. Stage 3: Final rendering

   Again, sample the scene from the camera, but this time optimized for direct illumination.

Here, stage0 is considered to be the scene assembly stage, that is the preprocessing, and is required unconditionally. Also, stage 1 is required when using photon map method.

Stages 2 and 3 can be worked with in terms of computation speed — i.e.; sampling rate. Here I would like to make clear some of the key points in implementing these parts.

4.6.1 Final gather values and complexity inside the pixel

The changes in final gather values of a very smooth object surface tend to have smooth regions as well. This is what allows the final gather estimation stage to be independent. But realistically, if multiple objects are visible within a single pixel, the estimation computation is costly, and a major amount of computation must be invested to obtain enough precision.

Final gather estimation algorithm attempts to fill as many pixels as possible with final gather values computed by a simple interpolation. In other words, if the changes are smooth enough, final gather sample points are placed only sparsely and compute the in-between values via interpolation later. On the other hand, if the values are changing significantly, more final gather sample points are placed.

The problem in implementing an algorithm like this is determining where to stop the sampling rate at. In our case we use the pixel size of the screen. In other words, if the final gather values change greatly between single pixel units, then we are unable to get any benefits of the final gather estimation. If the values are changing rapidly at a single pixel level, then getting a final gather value using supersampling and final gather is smaller in cost in terms of computation cost.

The following explanation should explain why doing the computation at the final rendering stage is better if the changes are occurring at a single pixel level.

Kilauea's final gather algorithm used in the final rendering stage is a simple Monte Carlo algorithm, which means that it contains a very high frequency noise. Also, it can be noted that pixels where final gather estimation didn't function has more information within that pixel, and therefore its looks are complex. For complex looking pixels such as this, high-quality final gathering isn't necessary. in most cases, because of the characteristic of the human eye, even if there were some noise it wouldn't be a problem because the pixel itself is already complex by nature. This means that for areas where the internal structure of the pixel is complex, we can achieve acceptable quality even if the final gather sampling rate is dropped. Currently our goal is to achieve an average quality of 65×65 final gather rays in the final gather estimation stage. But in the final rendering stage, similar quality can be achieved for areas with complex pixels with 27×27 or even 15×15 rays without problems.

This is why we don't perform computation finer than 1 pixel resolution in the final gather estimation stage.

For these reasons, the final gather estimation stage actually checks the complexity of each pixel beforehand. First we render the primary ray only and generate a mask that shows how the scene looks and how complex each pixel is. Based on this mask information, final gather estimation is executed only on areas where the pixel internals are simple enough. At this time, areas where the pixel internals have been determined to be complex will be processed using final gather + photon map at the final rendering stage, and at that time the sampling rate used will be significantly lower than the quality used in the final gather estimation stage. (see [8] for the details on final gather estimation itself.)

4.6.2 Pixel sampling in final rendering

We have already explained that IAB data will be used to transfer the final gather values between the stage 2 final gather estimation stage and stage 3 final rendering stage. There actually is one

another data structure that will be used in the final stage. That is the data used to describe the complexity inside each pixel.

Final gather estimation stage only performs its computation on areas where the pixel internals are simple enough, but in order to do that it processes the primary ray and store many types of information internally. By also recording the primary ray shading result at this time, it is possible to record the direct illumination changes at pixel units, which allow the final rendering stage to reference this data (refer to [8]). Current Kilauea implementation takes this direct illumination color information and the final gather estimation computation results when determining the on-screen sampling pattern in the final rendering stage.

For example, suppose that a 4×4 sampling will ultimately be required. Usually in Kilauea the image is computed by shooting 4×4 rays for each pixel in a straightforward way. Using the information computed in stage 2, it finds an area where point sampling is acceptable for each pixel, and create a screen sampling schedule that changes 16 rays into one ray. By doing this, it allows Kilauea to dramatically reduce the number of rays shot and the shading computation cost required to generate the final image.

This is done in the following method. based on the direct illumination information created in stage 2, we can determine which pixels have a greater color change from the adjacent pixels, based on simple lookup. From this information, it is easy to find pixels where direct illumination can be simplified to point sampling.

Also by looking at the final gather estimation values, we can determine which pixels already have the final gather values, and we can skip the final gather value computation for those pixels during the final rendering stage. This means that no supersampling needs to be done to compute the final gather values during the final rendering stage, and as a result for pixels where the final gather values have already been determined, point sampling is sufficient.

From these findings, we can determine for some pixel whether the pixel can use point sampling or need to do super sampling at the final image generation time, and as a result allow us to improve the computation speed significantly.

In other words, for simple cases like a simple flat object with a large area and its surface is a single color, We can guarantee a satisfactory quality by just point sampling the pixels. Numbers of such areas tend to increase as the resolution of the image to be rendered increases, so this is a very powerful characteristic when considering a production pipeline.

As a result, it is now possible to plan the sampling for the final image efficiently by rendering in multiple passes.

5 Kalapana – render farm control

Kilauea is a massively parallel renderer, designed to run separate processes across multiple machines which cooperate to compute the images. Naturally, some kind of batch system to automate dispatching, controlling, and shutting down of Kilauea processes on proper machines with proper parameters is demanded. The batch system specifically designed for the Kilauea renderer is called *Kalapana*.

5.1 Kalapana objectives

Here are the key features of Kalapana.

1. Make Kilauea look like one huge process

   Each Kilauea process is a separate entity, though it depends on each other for the rendering computation. Ideally, the creation and destruction of these Kilauea processes would be much easier to maintain if it were like a single process spanning over multiple machines. Kalapana achieves this by hiding all booting, shutdown, and signal handling (SIGHUP, SIGKILL, etc) from users.

2. Distributed design

   Kalapana does not rely on one powerful central server to maintain all job information. Rather, job information is distributed across machines under the control of Kalapana. In a sense, any machine can turn into the central job server. This distributed solution is more robust and risk-free compared to the central server solution, where the failure of the central server directly affects the entire system.

3. Fault-tolerant mechanism

   Because Kilauea runs on many machines at the same time, the chance to encounter the hardware or OS-level system instability multiplies. Kilauea has a built-in retry mechanism within sample scheduler to complete the rendering, but if some error beyond the control of the sample scheduler occurs, Kalapana shuts down and reboots the Kilauea processes, and starts the rendering from the current frame.

4. Resource control

   Kalapana keeps track of various system load information to find out which machines are idle, busy, reserved by someone, etc. The load monitoring system also turned out to be a crucial tool in the development process for troubleshooting and for checking if Kilauea is appropriately load balanced.

5.2 Kalapana implementation

Kalapana is a fully multi-threaded batch system written in C++ supporting Linux and IRIX. It consists of four programs: *host server*, *info server*, *client*, and *executor*.

- Host server

  Each render farm machine runs one Kalapana host server. A Kalapana client always connects to the host server running on the same machine for any transactions. Host servers are responsible for booting Kalapana executor, which starts a job process and watches its execution status. Host servers wake up every one second to gather system information and send them to info servers.

- Info server

  Info servers manage system information sent from host servers. In a typical render farm setup, two info servers keep track of the entire render farm system information to distribute the load.

- Client

  Kalapana client is the interface between users and Kalapana system. Users send various requests such as dispatching jobs and querying system information through the client.

- Executor

  Kalapana executors are responsible for actually booting job processes and watching their execution status. Executors act as the bridge between host servers and job processes, mainly to avoid multi-thread incompatibility issues with some system calls.

All Kalapana executables listed above have the standard Kalapana binary command engine built in, and they communicate with other Kalapana executables using this command protocol.

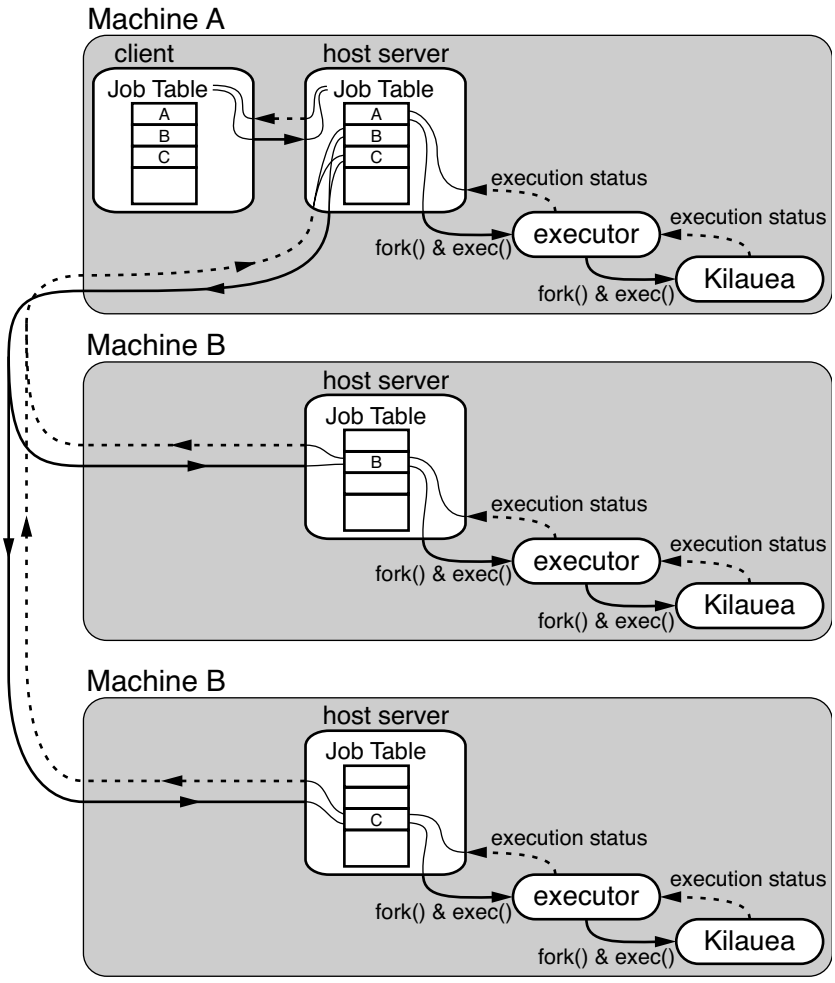


Figure 35: Job dispatch mechanism in Kalapana

5.2.1 Job dispatch mechanism

Figure 35 illustrates how a Kilauea rendering job is dispatched from the client. The client on machine A first sends the job dispatch request along with the corresponding job table to the host server running on the same machine. This host server then scans through the job table, and send job execution request to hosts A, B and C, as listed in the table. The host servers on A, B, and C receive the job execution requests and invoke Kalapana executors, and then these Kalapana executors finally starts the job process, Kilauea in this case.

When a Kilauea job is submitted and accepted at a host, this host is marked as reserved by the user who submitted the job. The job may be rejected if another user had already reserved it. The client must choose another host to run Kilauea in this case.

In the figure 35, the client on machine A is the *job master client*, responsible for managing job execution. The execution status of Kilauea processes are constantly monitored, and the information in job master client get updated promptly. When the executor catches the termination signal of the Kilauea process, the status is sent back to the host server, then to the job master client. When the job master client receives the signal of one of Kilauea processes, it proceeds to clean up all processes and release hosts reserved by this job. Depending on the type of received signal and the user preference, the client may attempt to restart the job. More on this fault tolerance is discussed in the next section.

Note that the job master client can run on any hosts. Unlike the central server strategy where all job information is stored in one powerful master server, job information of Kalapana are dispersed throughout the render farm. The weakness of the central server strategy is that everything fails when the master server fails. Kalapana's distributed design significantly reduces such risk.

5.2.2 Fault-tolerant mechanism

Because a typical Kilauea rendering job tends to use large number of machines, it is more susceptible to hardware and software instabilities than conventional renderers. Two fault-tolerant mechanisms exist in order to save Kilauea renderings from such unfortunate accidents.

The first mechanism is built into Kilauea. The sampling scheduler in Kilauea keeps track of every one sample request. When a sample request gets lost somehow and the result does not back after a certain period of time, it is rescheduled. This gives Kilauea another chance to complete the rendering even when one of the worker machine freezes. If the rescheduled samples does not get processed for a certain period of time, then Kilauea gives up and aborts, hoping that next fault-tolerant mechanism will remedy the situation.

The second mechanism involves Kalapana. Upon receiving the abnormal termination signal of one of Kilauea processes, the job master client first sends kill signal to all remaining job processes. After the confirmation of the entire job termination, the client may proceed to resubmit the job. The check-pointing and restart mechanism in Kilauea allows the restarted job to automatically pick up the rendering from the frame where it left off.

Users may set the maximum number of restarts per frame or per sequence to avoid wasting CPUs with jobs which always fail.

5.2.3 Obtaining host information – from info servers

One of the key features of Kalapana is the render farm resource management. Figure 36 illustrates how info servers maintain the render farm system information.

Host servers running on all machines wake up every one second to send the *heartbeat packets* to info servers. The heartbeat packets are basically just Kalapana commands with system information embedded to update the host list information on info servers. The system information included in the packet are Since the reliability of each packet is insignificant, it is sent in UDP to minimize the communication overhead. The packet includes information such as CPU usage (user and system),

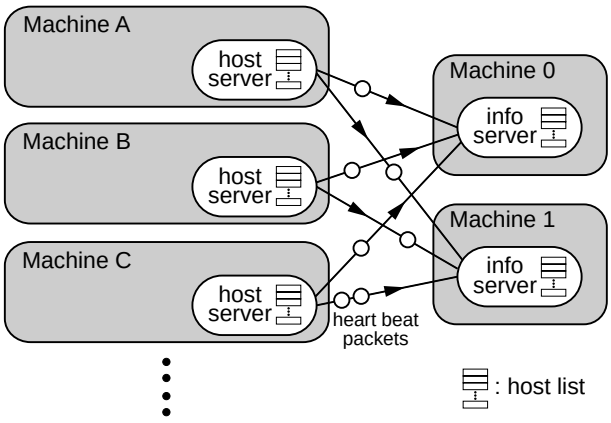


Figure 36: Host information management mechanism

CPU model, CPU speed, memory usage (allocated, cached), total memory size, network usage, disk access, OS version, reserve status, etc.

Heartbeat packets are literally used to keep track of whether a host is alive or not. After a long time of inactivity, info servers mark the host as offline. Users are not allowed to submit jobs to offline hosts. If a job process was running on the host detected as offline, the job master client may proceed to terminate the entire job, if it is instructed by the user to do so.

The step-by-step execution of the Kalapana command *lsHost*, a command to retrieve the host information from info servers, is explained in figure 37.

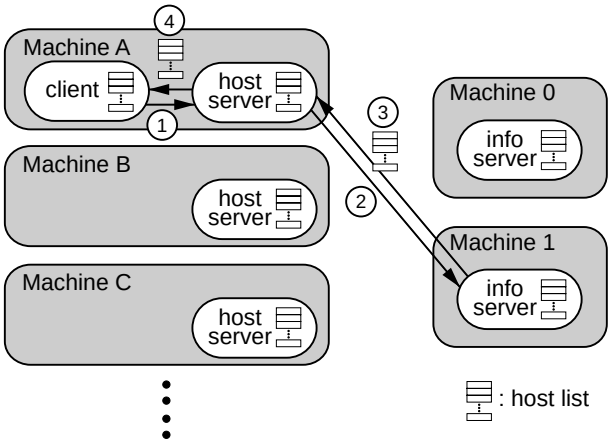


Figure 37: Host information retrieval

1. Client sends host info retrieval command to host server
2. Host server sends host info retrieval command to info server
3. Info server sends back host info update command to host server
4. Host server sends back host info update command to client

At step 2, the host server may send the command to either one of info servers. For every query from the host server to the info server, an internal counter is incremented to decide which info server to query next. This distributes the load on info servers.

One might wonder why the client does not directly query the info servers for the system information. Please notice that more than one client may run on a certain host. Because clients always communicate through the host server running on the same host, the host server may act as a proxy for these clients to reduce the network traffic. This is another aspect of Kalapana's distributed design.

5.2.4 Obtaining host information – without info servers

The previous section explained how the client retrieves all host information from info servers. However, users often want to only monitor the hosts running their job processes. Kalapana has an efficient mechanism to retrieve only the necessary host information, without querying info servers.

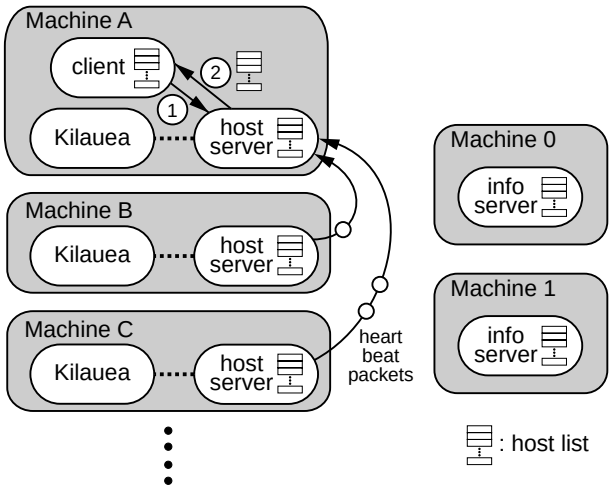


Figure 38: Host information retrieval without info servers

Figure 38 illustrates the case where machine A is the job master and Kilauea job processes are running on machines A, B, and C. Upon sending heartbeat packets to info servers, the host server also directs them to the master host server when job processes are running on this host server. This keeps the host information at the master host always up to date. Thus, to retrieve the host information of the hosts running jobs, the client only has to retrieve the information from the host server. This mechanism effectively reduces the network traffic and CPU load, especially when frequent update of the host information is needed.

5.3 Hawkeye – render farm resource monitor

Collecting system information in Kalapana is crucial for effectively managing computational resources of the entire render farm. Users may need to keep their eyes on how well a Kilauea rendering is performing, and manually tune parameters accordingly to maximize the resource usage. Also, Kilauea developers highly demanded a tool to verify if newly implemented load balancing and parallel processing schemes are performing as expected, and if not, find out where is the bottleneck at a glance. This was the main motivation for developing *Hawkeye*, the render farm resource monitor. The following sections go over how Hawkeye was successfully built on top of the Kalapana system

in a short amount of development time by integrating Ruby[1] scripting language.

5.3.1 Hawkeye development strategy

Hawkeye is essentially a special version of Kalapana client which retrieves the host information every certain interval and displays it in an easy-to-see, real-time manner. The special UI can be written in C++ just like Kalapana itself, but since building various interfaces suitable for diverse situations was planned, we came to the conclusion that a scripting language should be integrated into the Kalapana client to speed up the development. By making Kalapana's features and data accessible to a scripting language, major improvement in the development productivity is expected.

Scripting language integration is nothing new in this project. Kilauea renderer itself integrates Tcl interpreter for scripting and for use as a command processor. *Bufd*, the image viewer to receive pixel information from Kilauea, also integrates Tcl to provide a command processor and Tk GUI. Hence, we were fully aware of the advantages in combining C/C++ and scripting languages integration and it was a natural decision to integrate a scripting language in Kalapana also. The question is, which scripting language should Kalapana use?

The main objective here is to speed up the development more than C++, so the language should be a higher level language than C++, whose design leans toward more to machine than human. The natural access to C++ objects from the scripting language is desirable, and thus the language must be object-oriented. Also, the language should have powerful data structures and readily available libraries to prepare various interfaces. With these conditions taken into account, the language of the choice for Kalapana integration is Ruby.

Ruby is an object-oriented dynamic language with garbage collection, fully equipped with handy data structures and libraries such as text processing, networking, and GUI toolkits. Compared to other popular scripting languages, Ruby stands out from the crowd because of its consistent object-oriented design, highly dynamic nature, existence of closures (called iterators in Ruby) built into the language design, and easy-to-use extension API.

To summarize, the objectives in integrating Ruby into Kalapana for Hawkeye development are:

1. Seamless access to Kalapana functions and data from Ruby
2. Maximize the advantages of both scripting and compiled languages
3. Speed up development

5.3.2 Ruby integration into Kalapana client

When using a scripting language with C++, there are usually choices of either making the C++ program available from the scripting language, or integrate the scripting language interpreter in the C++ program. Kalapana chose to integrate Ruby, because Kalapana makes use of special malloc() developed for Kilauea to minimize memory fragmentation. If the Ruby interpreter run as the main program to call the Kalapana module, then malloc() would not be overridden and neither Kalapana nor Ruby would benefit from the special memory manager.

Making C++ classes available to Ruby is very easy. Many good documents for creating Ruby extensions and understanding Ruby internals already exist, so the detail will not be covered here. Although extending Ruby is intuitive and easy, making large number of C++ classes available to Ruby is tedious and time consuming. For the Ruby-Kalapana integration, a program to automatically create the bridge between C/C++ and various scripting languages called *SWIG*[2] was used to speed up the process.

[1] http://www.ruby-lang.org/
[2] http://www.swig.org/

5.3.3 Ruby and multi-thread safety

The Ruby interpreter is not thread safe. Kalapana is, however, fully multi-threaded. Full attention must be paid so that the interpreter will not be accessed by more than one thread at a time. One approach is to properly lock every operation to the Ruby interpreter. This is not so easy to accomplish, because a thread-safe wrapper must be written for all Ruby APIs used in the program. Ruby-Kalapana guarantees the thread safety by redesigning the threading structure and making sure that only one thread accesses the interpreter.

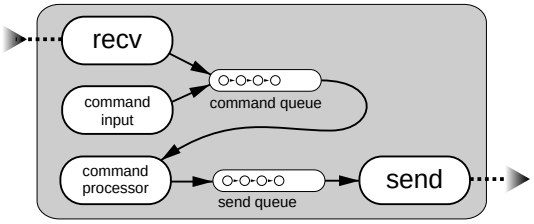


Figure 39: Kalapana client thread structure

Normally, Kalapana client runs four threads (figure 39). Their responsibilities are:

1. Send packet;
2. Receive packet;
3. Process command queue;
4. Accept command input.

Threads 3 and 4 may access the Ruby interpreter at the same time. Ruby-Kalapana runs in a special client mode where 3 and 4 are processed in the same native thread. The method to process one command queue is made available to Kalapana, and in the Ruby interpreter, a Ruby thread (non-native user-level thread) is created to invoke the command queue processing method whenever the queue is available. In other words, native thread 3 is replaced by Ruby thread to avoid the multi-thread issue (figure 40).

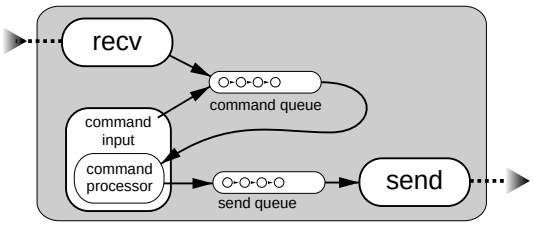


Figure 40: Ruby-Kalapana thread structure

5.3.4 Existing Hawkeye interfaces

Three Hawkeye interfaces are developed on top of Ruby-Kalapana.

1. Curses Hawkeye

   This is a text-based lightweight load monitor intended for use on text consoles and less powerful machines.

2. GTK Hawkeye (figure 41)

Figure 41: Screenshot of GTK-Hawkeye. CPU0 usage, CPU1 usage, and memory usage are displayed for each host.

   Using a graphical interface provided by GTK[3], hundreds of render farm load information is displayed as charts in real-time, updated every one second. The chart display allows users to grasp how the load changed over past 64 seconds. Because of Ruby's iterators, writing signal handers for GTK widgets become very intuitive and easy to read.

3. Hawkeye::Web (figure 42)

   Hawkeye::Web is an HTTP server to allow users to monitor and manipulate the render farm using their favorite web browsers on their favorite platform. The server uses an excellent Ruby library called *WEBrick*[4], which is literally a building block for developing custom network servers. The HTTP server built by WEBrick takes advantage of another wonderful Ruby library called *ERb*[5], which allows embedding of Ruby code into HTML. For example, a stripped down HTML code below shows how Ruby is actually embedded.

```
<html><body>
Current time is <%=Time.new%>.
claire01 CPU0 usage is <%=Kalapana.hostHash["claire01"].cpu[0].usage%>.
</body></html>
```

[3]http://www.gtk.org
[4]http://www.notwork.org/ipr/webrick/
[5]http://www2a.biglobe.ne.jp/ seki/ruby/erb.html

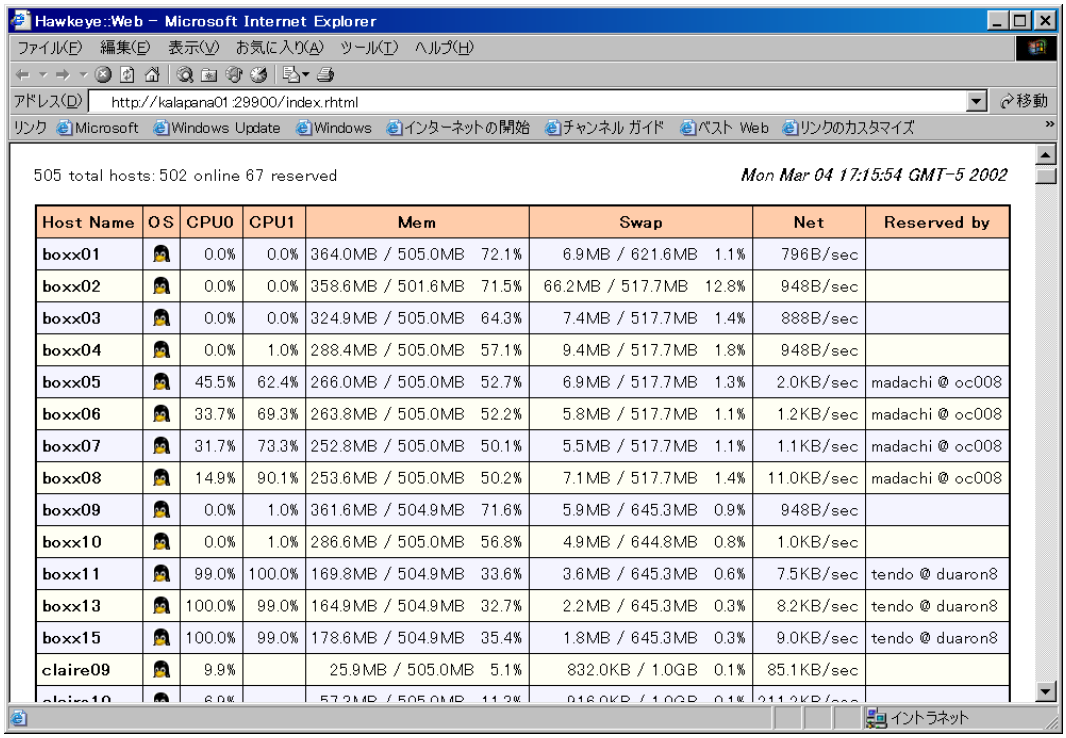


Figure 42: Screenshot of Windows IE accessing Hawkeye::Web

When the above HTML file is accessed, the server evaluates the embedded Ruby code and replace the text with the result of the Ruby code evaluation. In the example above, current time and claire01's CPU usage should be displayed.

The development of the initial version of Hawkeye::Web took only one hour, due to this extremely helpful Ruby-embedded HTML feature.

Please note that the development of Ruby-Kalapana and three interfaces for Hawkeye took only about one month, and most of the development time was spent on optimizing the real-time display speed of GTK Hawkeye.

5.4 Achievements of Kalapana and Hawkeye

Kalapana succeeds to be a distributed solution to the render farm control. Its distributed design minimizes the risk for failure, and also reduces network traffic and CPU load in various cases. Kalapana allows users to see distributed Kilauea processes as single process to make proper booting and shutdown much easier to control. Kalapana takes an important role in the fault-tolerance of the Kilauea rendering to vastly increase the chance of completing animation sequence renderings.

Hawkeye provides an easy-to-use, real-time mean to monitor the render farm resources at a glance to investigate the bottleneck quickly and make sure that the load is equally balanced. GTK, web, and Curses interfaces are available to meet specific needs.

Ruby integration into multi-threaded C++ program is not difficult at all and definitely worthwhile for boosting development speed. Using Ruby-Kalapana, Kalapana host list and job table are made accessible from Ruby array and hash. As a result, complex set operations to host list and job table can be expressed in a very short, intuitive code, resulting in vastly increased productivity. Ruby's iterators are extremely helpful in writing more readable code, as proven especially while

developing GTK Hawkeye. The dynamic nature of Ruby also greatly helped in speeding up the development by reducing the code test cycle. Convenient libraries such as WEBrick and ERb significantly reduced the development cost.

Hawkeye achieved to combine the fast, low-level backend engine implemented in C++ with high-level, flexible interface scripted in Ruby to maximize the strengths of both languages.

6 User interface between Maya and Kilauea

Maya is used for creating scene data rendered by Kilauea. User interface for controlling Kilauea from within Maya aims to integrate the Kilauea rendering seamlessly into Maya so that from the user's perspective, the user simply presses the "Render" button and appears as if Maya is performing the rendering.

6.1 UI implementation using Maya and its problems

The Maya scene data is converted into a Kilauea-specific file format and then the image is rendered from this data. Kilauea itself does not depend on a specific application for data creation. However, the production workflow at Square USA is built around Maya, and the goal was to be able to easily integrate Kilauea into this workflow.

As opposed to Maya which runs on each artist's desktop machine, Kilauea is designed to run on a Linux-based render farm over the network, managed by Kalapana (see section 5). For this reason, some form of communication mechanism is necessary between Maya running on the desktop and Kilauea running on the render farm. This communication needs to be bidirectional to exchange data and commands between Maya and Kilauea.

Initially, Maya plug-in MEL commands implemented the link between Maya and Kilauea directly using sockets. This plug-in mainly provides three features:

1. Set up rendering environment
2. Output Kilauea scene data and configuration data
3. Startup and shutdown Kilauea through Kalapana

The rendering environment describes the various data that will be required when Kilauea starts the rendering process. In addition to parameters required in ordinary renderers such as the screen field-of-view, they also describe parameters required for global illumination.

Kilauea scene data is referred to as KIS (Kilauea Incremental Shotdata) files, which will be described in detail later. Kilauea configuration data is a Tcl script file.

This workflow of launching Kilauea, sending the scene data to Kilauea, and rendering the image has been tested for its effectiveness under the production environment, and some flaws were found.

Maya plug-in modules do not run in multiple threads, which limits the plug-ins to work as a part of Maya's sequential process, that is, as a part of a single thread. This resulted in limitations such as not being able to continue modeling work in Maya while the commands are being executed.

During the production testing, users demanded to monitor Kilauea's execution status such as the progress of the rendering job or any error conditions. While it is possible to perform the monitoring outside of Maya, many test users requested to integrate the feature within Maya. In order to implement this directly in Maya, MEL commands must be issued frequently, or at least periodically. This solution causes the undesirable side-effect of pausing Maya every time the command is executed.

Focusing on the interactivity of sending commands directly from Maya to Kilauea raised another problem with the strong dependency between Maya and Kilauea. For example, because the startup and shutdown of Kilauea through Kalapana was controlled directly from Maya, sometimes

it became impossible to control neither Maya nor Kilauea, if for some reason the commands to Kilauea deadlocked.

The rendering flow between Maya and Kilauea was reorganized based on these problems found in the production testing. As a result, some of the functions were separated from Maya; commands are no longer sent directly from Maya, and the task of launching, shutting down, and monitoring Kilauea are removed from Maya.

6.2 Management of rendering jobs

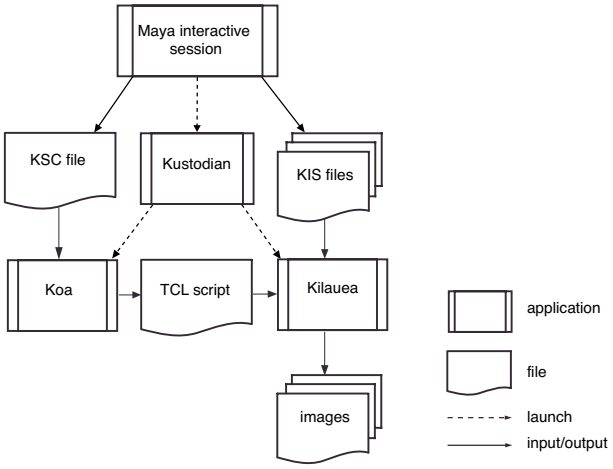


Figure 43: Kilauea workflow from Maya interactive session

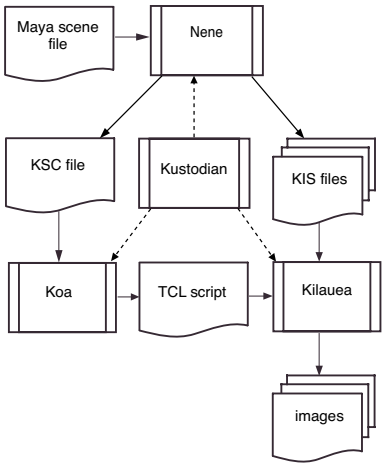


Figure 44: Kilauea workflow from Maya scene file using Nene

The current Kilauea workflow is shown in the figure 43. The task of launching, shutting down, and monitoring Kilauea has been removed from Maya into an application called Kustodian (see section 6.3). Likewise, the generation of rendering configuration file to be read by Kilauea (Tcl script in figure 43) is taken care of by an application called Koa (see section 6.5). Kustodian

communicates with both Maya and Kilauea, and is controllable from within Maya as a part of Maya's GUI.

In addition, an application called Nene was developed to convert Maya scene files (both Maya ASCII and Maya binary) to KIS files. Although exporting KIS files directly from Maya is still possible, using Nene as a stand-alone converter allows batch processing of KIS files, as shown in figure 44.

Running Nene on a separate machine also has the advantage of decreasing the load of the host Maya is running on.

6.3 Kustodian

Initial implementation to directly control Kilauea from Maya suffered from interruption of Maya operations and troubles occurring when Kilauea became unstable. *Kustodian* intends to separate the task to start-up and manage Kilauea jobs from Maya. Using Kustodian, job control becomes completely independent from Maya, which means that there is no effect on the Kilauea jobs even if Maya is working on something else, and conversely, no harm is done to Maya if the Kilauea execution becomes unstable. This allows the users to continue on with their Maya operations without worrying about the Kilauea job in progress. In the previous implementation, users were unable to specify the CPUs to be used for the rendering job, nor launch multiple rendering jobs simultaneously. When Kustodian receives a rendering job from Maya, it automatically assigns the necessary number of CPUs, based on the information from a database of rendering CPUs, checking for the currently running machines. Users need only specify the number of CPUs desired, without considering the resource usage in the render farm. Additionally, the ability to simultaneously execute and manage multiple jobs improved the usability.

When Kustodian receives the rendering command from Maya, it extracts the KSC (Kilauea Scene Configuration) file name from the command and analyzes its contents. Based on this information, Tcl file for Kilauea is generated, through a program named *Koa* (explained in section 6.5). In the next step, the execution status of machines currently registered in the database is queried to Kalapana, and Kustodian reserves the requested number of machines, excluding the ones currently reserved or offline. After the machines are reserved, the display program is launched, and Kilauea is launched via Kalapana. Once the Kilauea process is launched, the management of Kilauea itself is handled by Kalapana. Kustodian constantly checks the log output from Kilauea, and displays the rendering progress and any error status on screen. Once the rendering job completes, Kustodian releases the CPUs used for the job from the database, allowing the next job to use them.

Rendering commands from Maya are registered into Kustodian's internal database as rendering jobs. A rendering job is divided into small stages such as the analysis of KSC files and reservation of CPUs. A rendering job is treated as one object, and automatically saved to a file at every stage. If for some reason Kustodian's execution terminates, pending jobs will automatically be restored when Kustodian is re-launched. In addition, jobs are saved as files unless explicitly deleted, allowing them to be reexecuted at any time. The ability to resume a job is one of the most important feature of Kustodian, making this a big advantage when managing a job that takes a long time to render.

Kustodian is developed using Ruby and GTK. The use of scripting language was planned from the beginning, for the extensibility and maintainability. As already described in the previous section on Kalapana, Ruby is an excellent object-oriented dynamic scripting language to speed up the development. GTK is the toolkit of choice, because of its popularity, variety of available widgets, and ease of use from Ruby.

Kustodian is structured as a client/server application. The client and server communicates through a socket. Maya calls the Kustodian command which is the client, sending the KSC filename (see section 6.5) to the Kustodian server which actually controls the rendering job. Kustodian command is simple, consisting of the KSC file and a few options. This makes it easy to send a rendering command from Maya to the Kustodian server. If necessary, other tools can start a rendering job easily.

When the Kustodian command is called, it checks for a running Kustodian server, and will launch one if it isn't already running. Also, the Kustodian server is designed so only one copy is running on a single machine. Therefore, the caller does not need to think about the Kustodian server at all.

By managing the rendering jobs through Kustodian, Kilauea and Maya were separated effectively, making possible a flexible development and work environment.

6.4 KIS file

KIS is an acronym for Kilauea Incremental Shotdata. KIS files contain all the camera, light, and renderable object data from Maya. A single KIS file may contain all the objects for all the frames, or multiple KIS files may contain a specific object for a specific frame.

One important feature of KIS files is that, as its name implies, it is an incremental data format. This is to say that the data of the first frame contains all the data of the scene, but only the difference information is stored in the file for frame two and later. To avoid the problem of reading all the proceeding frames in sequence just to get the last frame, KIS optionally allows to set the full dump frame every several frames.

The main reason for using an incremental data format is above all that and in conjunction with other compression schemes, the file size can be greatly reduced.

6.5 KSC file

In addition to the KIS file that describes the scene to be rendered, Kilauea requires a Tcl script that controls the scene rendering as well. Using Tcl scripts allow the rendering parameters to be manipulated in a flexible way, but the disadvantage of this method is that users must understand Tcl scripting and Kilauea functions to be called. Actually, users did not need to write Tcl files themselves, instead letting Maya generate the Tcl files and send to Kilauea. However, Kilauea functions and parameters constantly undergo additions and modifications, so export tools constantly needed to take this into account. This also caused compatibility problems between Tcl files generated from different tools. Further, Running an older version of Tcl files on a newer version of Kilauea caused problems with parameter mismatch. To solve these problems, a metafile format called "KSC" is created to pass information from Maya to Kustodian, and developed a tool "Koa" to generate a Tcl file from this format. KSC stands for "Kilauea Scene Configuration". KSC absorbs the difference between the version differences in the Tcl files. Programs that previously generated Tcl programs no longer generate Tcl files directly, instead generating a KSC file and execute Koa to generate the Tcl file. By doing so, it is possible to unify the version of Tcl files generated by the various tools. KSC file format itself is a very simple, extensible format, and developing other programs to output it is not a problem at all.

Koa, like Kustodian, is written in Ruby. In addition to the same reasons Ruby was used for Kalapana and Kustodian, Ruby simplifies the coding of text processing using regular expressions, which is crucial in the processing of KSC files and generation of complex Tcl files.

The introduction of KSC file format and Koa accomplished absorbing the version differences in Kilauea Tcl commands. This improved the version control maintainability in developing Kilauea plug-ins, resulting in more stable workflow.

6.6 Achievement of the new UI

By deploying Kustodian, Nene, and Koa together, the usability of Kilauea + Maya and the overall system stability including Maya has increased significantly.

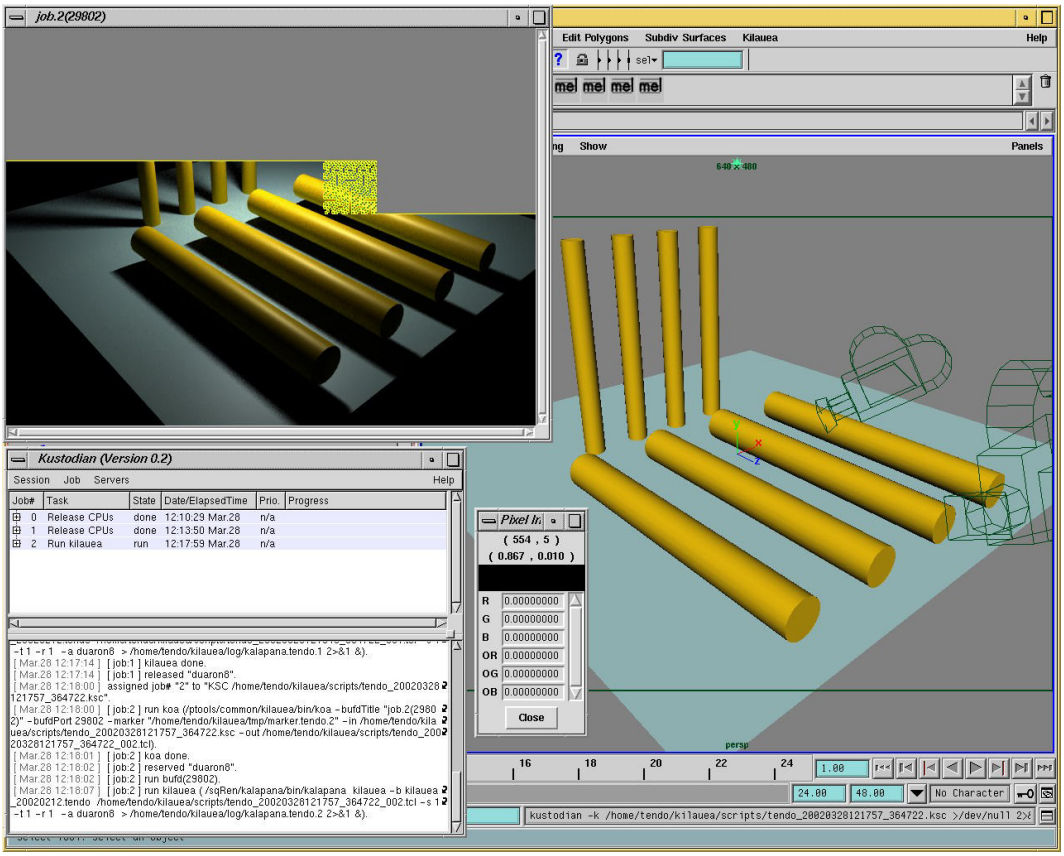


Figure 45: Kilauea rendering in progress from Maya

7 Results

7.1 Summary of the project

Kilauea is a vast system. Brief explanation of the scale, special features of the project, etc. are given here.

7.1.1 Scale

The following data outline the overall scale of the system.

- Duration of the development: 4 years (Mar 1998 to Feb 2002)
- Number of developers: 7
- Main programming language: C++
- Integrated interpreters: Tcl, Ruby
- Source code management: cvs
- Number of lines in the Kilauea renderer itself: 758,329
- Number of lines in the testsuite program: 187,041
- Total number of lines: 945,370
- Number of internal modules in Kilauea: 46
- Number of module-related testsuite programs: 239
- Number of task types during execution of Kilauea: 11
- Total number of test scenes: 80+

7.1.2 Features

The internals of Kilauea are divided according to their function. Kilauea is divided into several groups of logically independent processes. Such group is referred to as module. Kilauea now consists of 46 modules.

Interdependency between the modules is kept as low as possible. The concept of message passing greatly aids in maintaining the low interdependency. Such separation of the modules was thoroughly carried out to allow smooth development even when different groups of developers are assigned to work on each module. Developers can concentrate on perfecting the module that they are assigned to, and each module is designed so that it can be tested independently as a testsuite by a small program. Currently, 46 modules of Kilauea are created as local library and linked upon compilation. There exist 239 testsuite programs that are used to test one module or several modules at the same time.

Conducting tests repeatedly to make sure that each module is thoroughly independent and perfecting each module's functionality by utilizing testsuite program is the least requirement for assuring the stability of such enormous rendering system as Kilauea. Therefore, it becomes important that when a new module is installed or modification is made to a module, accompanying testsuite is almost always updated or appended with a new program. Kilauea turned out to be a system comprising of more than 750,000 lines in the end. However, as long as the developers adhere to the same testing policy, the system can grow even larger and still have the same level of stability.

#	FG est. (hr:min:sec)	Render (hr:min:sec)	All (hr:min:sec)
1	0:32:17	1:14:39	1:51:24
2	0:15:27	0:35:37	0:53:58
3	0:10:15	0:23:30	0:37:40
4	0:07:36	0:17:22	0:29:02
5	0:06:14	0:14:04	0:24:17
6	0:05:09	0:11:41	0:20:51
7	0:04:26	0:10:01	0:18:29
8	0:04:12	0:09:13	0:17:49
9	0:03:43	0:08:02	0:16:32
10	0:03:11	0:07:09	0:15:00
11	0:03:04	0:06:36	0:14:21
12	0:02:49	0:06:00	0:13:08
13	0:02:36	0:05:35	0:12:41
14	0:02:29	0:05:10	0:12:14
15	0:02:17	0:04:47	0:11:37

Table 1: Town house: timing results

Before the distribution of the official version of Kilauea, thoroughly confirming that all important functionalities are not broken using a pre-defined set of testsuite scenes is important. At the moment, there are 80 types of test scenes that an official version of Kilauea must pass before it is released.

7.1.3 Parallel performance timing results

- Test case 1: Town house

  Figure 46, the town house scenery, is small enough to be stored in one machine, and Kilauea will distribute identical copies of scene data to all machines participating in the rendering. Three rendering times were recorded while changing the total number of machines:

  A) All: total time from booting Kilauea to shutting down Kilauea

  B) FG Est: final gather estimation stage

  C) Render: final rendering stage

  For this experiment, maximum of 15 machines with dual Pentium III 800 MHz, 512 Mbyte memory, and 100 BaseT Ethernet are used.

  Table 7.1.3 shows the timing results, and figure 47 is the plot of A), the total rendering time including Kilauea initialization and clean up. Unfortunately, the initialization stages such as booting tasks and reading in the scene data are essentially difficult to take the advantage of parallel processing or parallelization is not thoroughly considered yet. Due to this, the graph in figure 47 gradually falls off the optimal linear performance as the number of machines increases.

  Figure 48 is the plot of B), the final gather estimation stage. This stage performs parallel computation by dividing up the screen space into rectangular buckets. It achieves adequate parallel performance, though the varying processing time for each tiles starts to dominate and affect the load balancing as the number of machines increases. By decreasing the size of each bucket, better scalability may be attained.

Figure 46: Town house: 732,058 triangles, one directional light, one sky light, four area lights in the rooms, 650,000 photons emitted, 885,315 photons stored

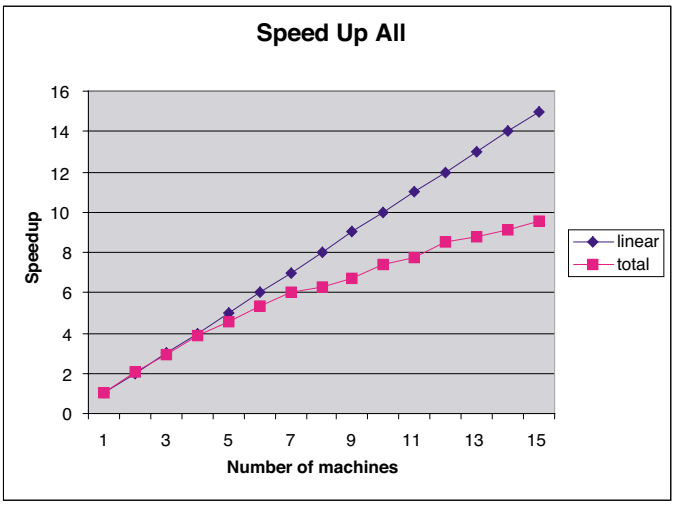


Figure 47: Town house: number of machines vs. overall rendering time

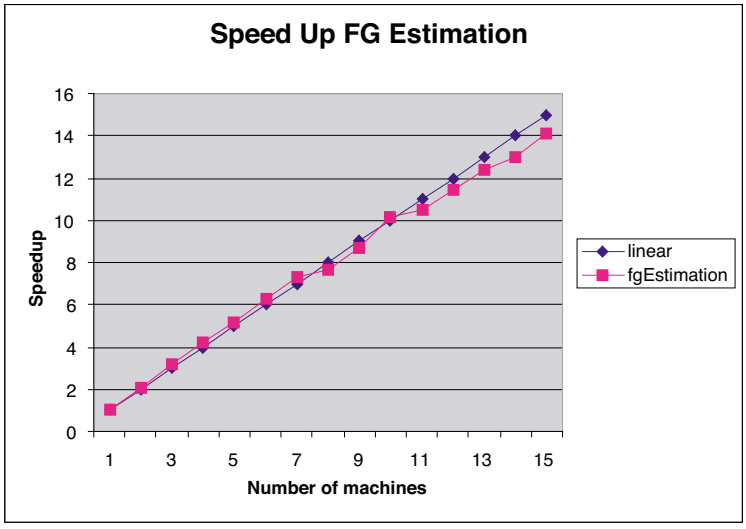


Figure 48: Town house: number of machines vs. final gather estimation time

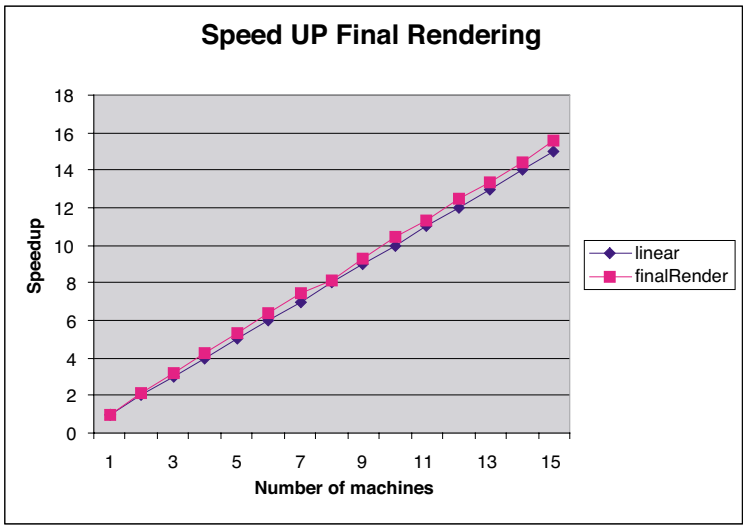


Figure 49: Town house: number of machines vs. final rendering time

Figure 49 is the plot of C), the final rendering stage. This stage exhibits a superlinear scalability, as the plots exceed the optimal values. This is speculated to be caused by caching at various levels.

The final gather estimation stage and the final rendering stage both exhibit satisfactory parallel performance, meaning that the more machines invested, the more performance increase achieved correspondingly. These two stages are the most repeated ones during the course of adjusting surface materials and lighting, and their optimal scalability is a huge benefit in the production work.

Only 15 machines were available to conduct this experiment, and there is no actual data for using more machines. In theory, the network will be the bottleneck and the performance will saturate at some point as the number of machines increases. In this test case, the only data transferred through network are the final sampling results, and this overhead is very small. Therefore, parallel rendering using more than 50 machines should not be a problem at all.

- Test case 2: Fiat 500L×32

#	FG est. (hr:min:sec)	Render (hr:min:sec)	All (hr:min:sec)
2	0:13:12	2:24:45	2:43:06
4	0:06:23	1:12:52	1:23:27
6	0:04:13	0:46:31	0:54:28
8	0:03:08	0:34:49	0:41:25
10	0:02:26	0:28:45	0:34:46
12	0:02:01	0:23:44	0:29:28

Table 2: Fiat 500L×32: timing results

This test intends to show the parallel performance of Kilauea when the scene data is shared among multiple machines. The cars in figure 50 are exactly same, but for the purpose of this experiment, they are intentionally duplicated, not instantiated, to make the scene larger. Because one machine cannot hold this much data, the scene is shared among two machines.

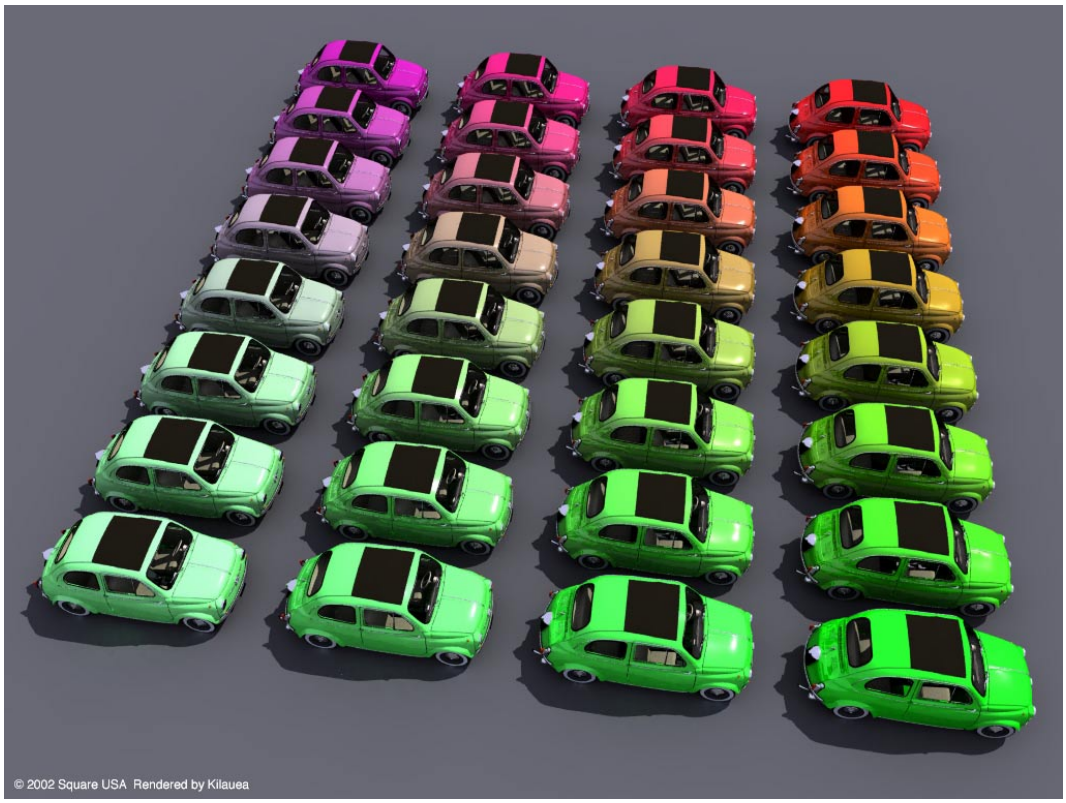


Figure 50: Fiat 500L×32: 2,994,752 triangles, one directional light, one sky light, 700,000 photons emitted, 756,492 photons stored

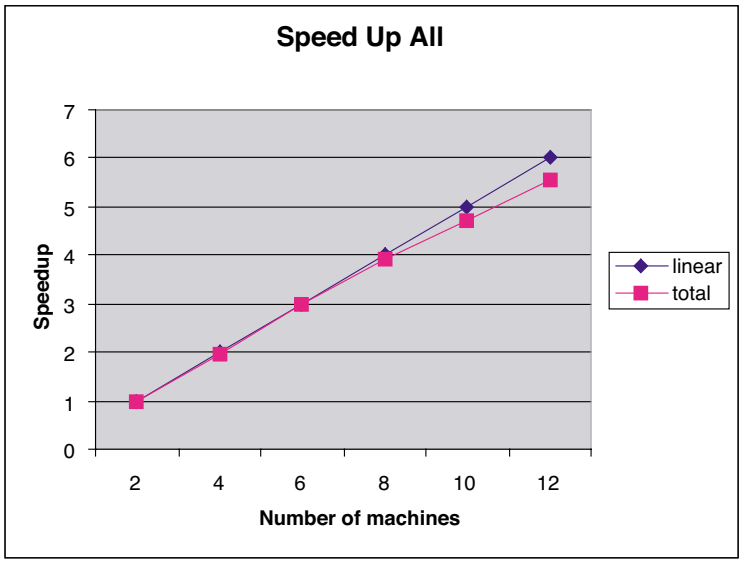


Figure 51: Fiat 500L×32: number of machines vs. overall rendering time

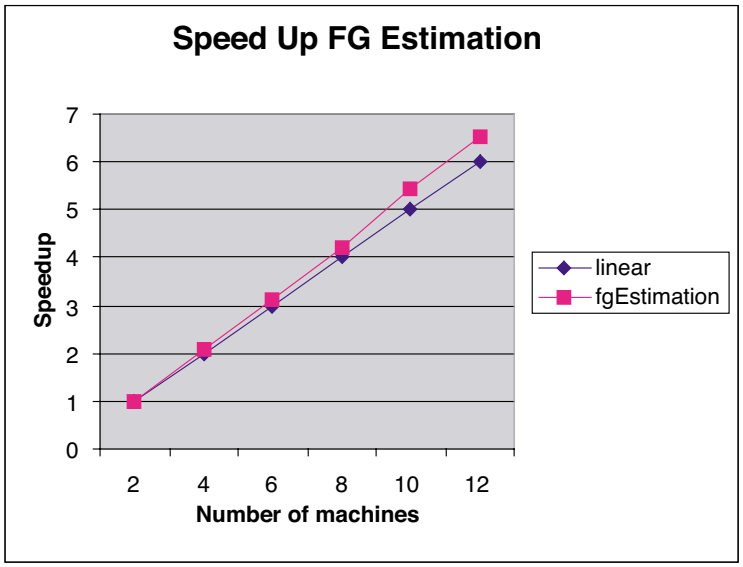


Figure 52: Fiat 500L×32: number of machines vs. final gather estimation time

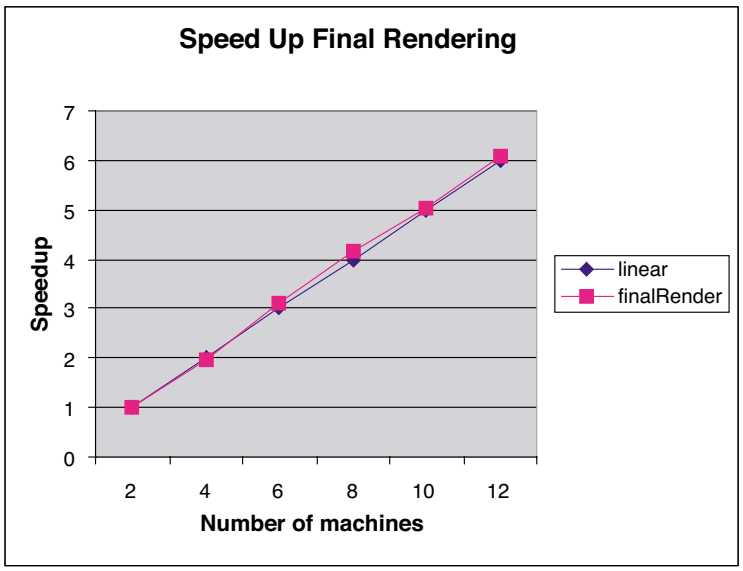


Figure 53: Fiat 500L×32: number of machines vs. final rendering time

The same timing tests will be conducted. Because the minimum number of machines to hold the scene is two, the number of machines will be increased by the multiple of two. The results are shown in table 7.1.3.

Figure 51 is the plot for the entire rendering, including Kilauea initialization and clean-up. Notice that the performance drop is not as drastic compared to the previous case. This is because the final rendering stage is slow and thus dominates the timing results, compared to initialization stages which are not-so-well parallelized.

Figure 52, showing the final gather estimation stage, is now also exhibiting superlinear behavior. The scalability of the final rendering stage is also superb, as shown in figure 53. Overall, the performance of the rendering stage increases as more machines are invested accordingly.

7.1.4 Discussion on parallel performance

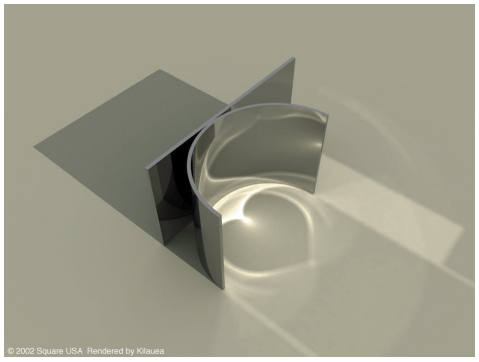
So far, the final gather estimation stage and the final rendering stage show excellent scalability. However, the parallel rendering performance will eventually saturate because the network communication starts to be the bottleneck as the number of participating machines are increased either to speed up the rendering or to store larger scenes. Some of the solutions to remedy the situation are:

1. Suppress network usage
2. Use network with broader bandwidth

One idea to achieve the first solution is to implement communication compression. Currently, the compression is performed on certain data communication only. Applying compression to all streaming data in Kilauea may somewhat improve the network traffic.

The second solution is to simply make use of the faster network such as Gigabit Ethernet, whose price is rapidly dropping lately. This will push the saturation point even further away.

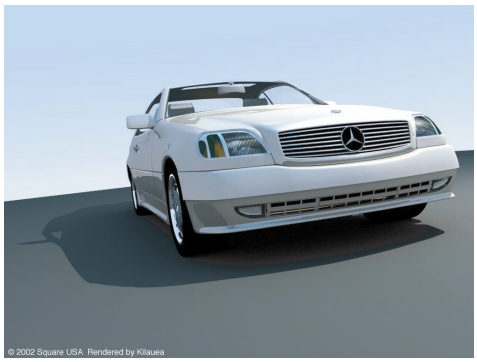
7.2 Other sample images



Caustic K

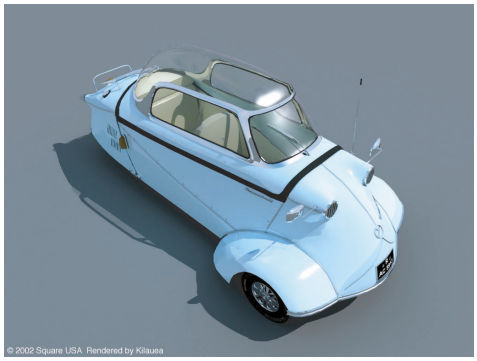
Hikone

FALoader



M600

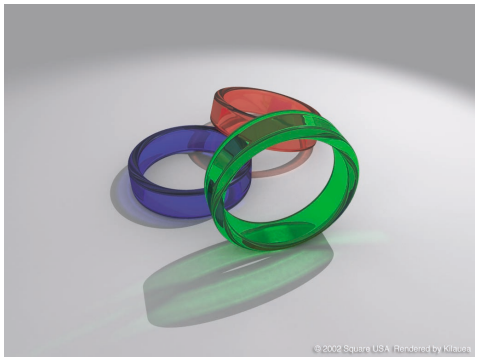
Fiat 500L



Messerschmidt

Panzer

Sponza 2



Rings

Vase

Sponza 1

Victorian House

Figure 54: Town house at night

8 Conclusions

As stated in the beginning, there were two objectives of the Kilauea Research Project: 1) rendering of high quality images with the consideration of global illumination, and 2) ability to render extremely complex scenes. Kilauea, as it exists now, can satisfactorily calculate high quality scenes with full global illumination. By distributing the task among multiple machines, extremely large scenes can be rendered. In this sense, we have achieved the original two goals of the project.

Furthermore, Kilauea is already at the level where it is stable enough to be used in the production environment. Issues that may arise during the production work are taken into account as well. For example, need for incorporation of various shaders is handled by providing shader API by C++. The daemon that is responsible for dispatching, controlling and shutting down of the processes on the Kilauea render farm is also at the level of testing for commercial viability. Although much work has been accomplished, more improvements are needed to speed up the rendering further. Overheads still exist for handling Kilauea's intrinsic processes such as parallel processing, message passing and distribution of scenes among multiple machines. If a scene is sufficiently simple, processing speed of Kilauea is admittedly slower than conventional direct illumination renderers without such overheads. It is regretful that the project is terminated at this point prematurely with still much room for optimization in reducing the overheads, before the project's full potential is reached.

However, if the overheads were reduced to a permissible level, scenes can be rendered with a flexible degree of processing power as needed. The concept of adding more machines when one machine is inadequate to render a scene is a very effective means to increase processing power, especially considering the low-cost and high-performance PCs readily available in today's market, and even more so in the future. On the contrary, the concept will take direct advantage of upcoming technological advancements such as those in computer hardware, networking technology etc., and will not be an impediment in the future. In that sense, Kilauea has succeeded in establishing a core architecture that will endure for a very long time in the future.

It is clear that the need for rendering high quality, extremely complex scenes will increase in the CG productions in the future. Although there may only be a handful of information and lessons learned from the project, those are sure to be invaluable in contributing to the future of rendering technology.

9 Acknowledgements

We would like to express our utmost gratitude to Kazuyuki Hashimoto, Kaveh Kardan, and all members of Square USA R&D division for provided us with numerous invaluable advice and ideas over the course of four years in the Kilauea Research Project. Furthermore, this course note would not have completed without the assistance of Junichi Kimura of Square USA R&D division.

We would also like to express our very best gratitude to Alan Chalmers of the University of Bristol for advice on parallel rendering, and Henrik Wann Jensen of Stanford University for advice on global illumination and photon mapping.

Sponza model was provided by the courtesy of Marko Dabrovic. Many testsuite models are from De ESPONA 3D Encyclopedia 2000 Edition.

References

[1] Alan Chalmers et al. “Practical Parallel Rendering”. ISBN: 1-56881-179-9, A K Peters, 2002.

[2] Cindy Goral, Kenneth Torrance, Donald Greenberg, Bennet Battaile. “Modeling the Interaction of Light Between Diffuse Surfaces”. *Computer Graphics (SIGGRAPH ’84 Proceedings)*, volume 18, number 3, pages 213-222, July 1984, Minneapolis, Minnesota.

[3] Henrik Wann Jensen, Niels Jorgen Christensen. “Photon Maps in Bidirectional Monte Carlo Ray Tracing of Complex Objects”. Computers & Graphics, volume 19 (2), pages 215-224, March 1995.

[4] Henrik Wann Jensen. “Realistic Image Synthesis Using Photon Mapping”. ISBN: 1-56881-140-7, A K Peters, 2001.

[5] SIGGRAPH 2000 Course Note. “A Practical Guide to Global Illumination Using Photon Maps”.

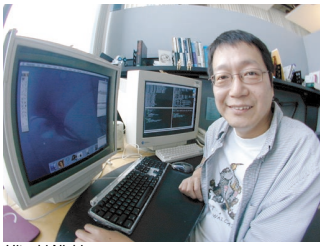
[6] SIGGRAPH 2001 Course Note. “A Practical Guide to Global Illumination Using Photon Mapping”.

[7] SIGGRAPH 2001 Course Note. “Parallel Rendering and the Quest for Realism: The ’Kilauea’ Massively Parallel Ray Tracer”.

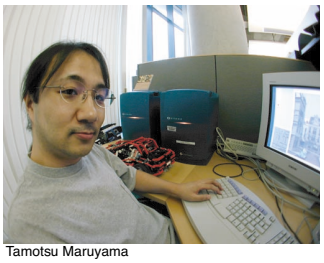
[8] SIGGRAPH 2002 Course Note. “Photon Map in Kilauea”.

[9] Gregory Ward, Francis Rubinstein, Robert Clear. “A Ray Tracing Solution for Diffuse Interreflection”. *Computer Graphics (SIGGRAPH ’88 Proceedings)*, volume 22, number 4, pages 213-222, August 1988, Atlanta, Georgia.


Toshi Kato


Hitoshi Nishimura


Tadashi Endo


Tamotsu Maruyama


Jun Saito


Motohisa Adachi