# COVEN: A COMPUTATIONAL MODEL AND PROBLEM SOLVING ENVIRONMENT FRAMEWORK FOR SUPPORTING OPTIMIZATION OF PARALLEL APPLICATIONS

---

A Dissertation

Presented to

the Graduate School of

Clemson University

---

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

Computer Engineering

---

by

Nathan A. DeBardeleben

December 2004

Advisor: Dr. Walter B. Ligon, III

December 10, 2004

To the Graduate School:

This dissertation entitled “Coven: A Computational Model and Problem Solving Environment Framework for Supporting Optimization of Parallel Applications” and written by Nathan A. DeBardeleben is presented to the Graduate School of Clemson University. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy with a major in Computer Engineering.

Walter B. Ligon, III, Advisor

We have reviewed this dissertation
and recommend its acceptance:

Adam Hoover

Ron Sass

Pradip Srimani

Accepted for the Graduate School:

## ABSTRACT

As high performance computers continue to gain in acceptance and presence globally, more people are beginning to use them. This increase in use has brought parallel computing to groups seeking its performance benefits but who may not be trained in parallel computing. Problem solving environments (PSEs) have long been a software mechanism to ease programming complexity and are also gaining in popularity in the high performance computing realm. However, while PSEs have been useful in simplifying parallel program implementation, most lack the ability to optimize an application to perform with efficiency on the target architecture.

This work details a model of computation for parallel computing called the Coven model and the Coven PSE which implements the model. The Coven model makes it possible for the PSE to optimize applications implemented in it by understanding the application's computational structure, including data and control flow. The design and implementation of the Coven PSE is detailed and includes a suite of features that ease parallel programming by exploiting properties of the Coven model.

Three optimizations (multi-threading, dynamic load balancing, and checkpoint / recovery) are detailed and analyzed to study the performance benefits they provide to Coven applications. Using real applications such as the FFT, fractal generation, N-Body simulation, and a heat transfer simulation, this work examines both the performance improvement by employing these optimizations as well as the complexity of implementing the optimization outside of Coven. We conclude that Coven can provide these optimizations with almost no programming cost to the user and can be used to achieve performance gains. Most importantly, this work makes accessible these optimizations to users without the need for them to possess parallel computing expertise.

# ACKNOWLEDGMENTS

Thank you to my committee for their insights, Ron for his editorial assistance, and Walt for his guidance along the way. Thanks to my mother, Marian, for believing in me and my daughter, Natalia for inspiration and purpose. Most importantly, thank you to my wife, Michelle, whose undying patience and loving support were invaluable during this long journey.

TABLE OF CONTENTS

Table of Contents (Continued)

Page

Table of Contents (Continued)

Page

## LIST OF TABLES

LIST OF FIGURES

List of Figures (Continued)

List of Figures (Continued)

# CHAPTER 1

# INTRODUCTION

There have always been applications that perform poorly or fundamentally cannot execute on conventional computers. These applications are often limited by the resources of conventional computers including memory, disk, and processor speeds. As computers and their components have improved over the years, many of the applications that were previously unsolvable on commodity machines are so trivial that they now run on hand-held computers. This trend, however, does not mean that the set of applications that require advanced computers is shrinking. On the contrary, as more computing power is made available, scientists are finding that they can expand their algorithms to include more parameters or increase the data resolution to provide more realistic solutions, improved simulations, and achieve more accurate answers.

Even today on some of the world's fastest and most expensive supercomputers, many applications take days, weeks, or longer to execute. Often the goal is for these applications to execute in a matter of minutes. For example, while scientists have developed very accurate weather models that can predict tornado paths, the models take weeks to run on a multi-million dollar machine. This is far too slow to prove useful in saving lives or property. Examples like this can be found throughout computational science and engineering, as well as business, biomedicine, and many other areas.

Parallel computers are machines that aim to achieve high performance by bringing multiple processors together into a single computational entity. This approach contrasts with traditional methods that concentrate on increasing the speed of a single processor or other resource. There are numerous classes of parallel computers ranging from many individual machines connected together by a network, to several powerful processors sharing a massive centralized memory. Although the machines that reside at opposite ends of this spectrum strive to solve vastly different classes of problems, they share an important similarity. In particular, they all require special languages, tools, and environments to develop, execute, and debug programs.

For many years it appeared that with the arrival of each new supercomputer came a set of new tools and languages to learn. Invariably, legacy code had to be rewritten for the new system. This led to efforts in making languages or libraries portable across architectures. Libraries such as PVM [81] and MPI [36] provide APIs that are implemented on most systems and prove to be a much more portable way of moving applications between systems. While these interfaces have gone a long way in making parallel code easier to write, they still require a steep learning curve which is often a daunting fact to application domain scientists and engineers. Furthermore, achieving the best performance of these parallel machines often requires a very detailed understanding of many of its components, such as the operating system, network, memory, and storage system. For example, a code modification that results in improved performance in application runtime on one system often has the opposite effect on another.

The fundamental problem is that there are two separate classes of people in the parallel computing field: those who have applications that need the benefits from parallel computers, and those who understand how to correctly and efficiently implement code on parallel computers. We will term members from these two classes *application domain specialists* and *parallel computing specialists* respectively. Although there are many individuals that possess skills from both of these classes, the complexity of parallel computing often requires that an individual of one class learn the skills of the other class in order to reap the rewards of parallel computing.

With the introduction of Beowulf [80] clusters, more and more organizations are acquiring parallel computers. Beowulf clusters offer a low cost to performance ratio, generally through use of commodity hardware and open source software. This software includes operating systems, programming libraries, compilers, and debugging tools. One needs only to observe the fluctuation of the top list of the fastest supercomputers [35] to notice that parallel computing is becoming more prevalent. However, from the small university researcher who needs only sixteen nodes to the groups who need multi-million dollar machines, programming parallel computers still is difficult.

## 1.1 The Parallel Programming Problem

While parallel computer usage is increasing in amazing numbers all over the world, users are finding that in order to achieve the projected performance benefits of their system they must be skilled parallel programmers. Skillicorn and Talia [76] suggest a few reasons why parallel computers are complex to program. They note that while the execution time of a “sequential program changes by no more than a constant factor when it is moved from one uniprocessor to another,” this is not necessarily the case for parallel programs. Order of magnitude runtime differences are not uncommon when transporting parallel code between different machines. This is because the performance of the code is heavily dependent upon the underlying system, including the hardware architecture and operating system software. Furthermore, they state that it takes a “long time to understand the balance necessary between the performance of different parts of a parallel computer and how this balance affects (overall application) performance.” In particular, the dynamic relationship between processor and network interconnect greatly affects application performance. It is often difficult to achieve balance between processor and network capabilities, even for parallel computing specialists.

There are many approaches aimed at easing the task of parallel programming. Automatic parallelizing compilers can take sequential code and parallelize it; however the effectiveness of these compilers is largely limited to a relatively small class of applications. Programming models have emerged as a fairly productive way to achieve high performance on most architectures. MPI [36], for example, provides high level primitives which are implemented differently on each underlying system to achieve good performance. As with any high level “language”, there is a tradeoff between simplicity and performance. A talented and knowledgeable programmer could likely code a faster common parallel programming construct on a system in much the same way as a gifted assembly language programmer is able to achieve higher performance than a conventional compiler. Although this manual approach is effective, it is also extremely costly in man-hours and is usually considered not worthwhile in the long term.

## 1.2 Problem Solving Environments

Problem Solving Environments (PSEs) are one approach in addressing the programming complexity of parallel computers. They are gaining popularity, especially as more focus is being placed on making parallel computers accessible to those that need them the most. A PSE, simply put, is a software toolkit that assists an end-user in solving a particular problem. PSEs are intended to provide all the necessary tools a user needs, helping them at each stage of an application’s life cycle. This includes design, implementation, execution, debugging, data analysis, performance analysis, performance tuning, data visualization, and application maintenance. Many PSEs exist for sequential computers, but there are considerably fewer for parallel computing. Related work in the PSE field is outlined in Chapter 3.

While there exist a number of problem solving environments for specific application domains, little effort has been placed into general purpose problem solving environments, particularly in the realm of parallel computing. PSEs are generally used to reduce application development time through the use of modular programs, code reuse, and debugging tools. Parallel computing brings forth another constraint that is sometimes overlooked in sequential PSE development, namely performance. Programmers who look to parallel computing as an alternative for their code do so because the sequential world can no longer facilitate their application, either due to exorbitant program runtime (often on the order of days or weeks) or due to limited resources (such as memory or disk space). Therefore, PSEs for parallel computers must not only provide the set of features important to sequential PSEs but also assist in performance analysis and tuning as well as providing features that are important to parallel computing.

## 1.3 Parallel Computing Optimizations and Features

Simply implementing an application in parallel does not always provide the type of performance enhancement required. In these cases, code optimizations are often employed to see if they can potentially result in additional performance improvements. While PSEs, APIs, and frameworks are often used to help ease the task of parallel programming, optimizing parallel applications presents another level of complexity. Because of this difficulty, few environments exist that can assist a user in performing complex parallel computing optimizations.

There is a wide range of optimizations that are often employed to increase the performance of a parallel application. Examples include adjusting memory access patterns, prefetching data from a storage device, overlapping computation and communication, and balancing the workload distributed to each parallel process. Researchers are constantly in search of new optimizations and are trying to employ well known techniques in their applications to achieve as much performance as possible from their parallel computers. Due to the expense of parallel computers, the most important goal generally is to minimize an application's execution time.

In many parallel applications the processor must request data from some external device such as a storage system or another node through an interconnection network. Depending on the latency and bandwidth of the networking infrastructure, the processor may often stall waiting for the data to arrive. Researchers have turned to multitasking operating systems and multi-threading as a potential way to address this problem. With multi-threading, when a process (or thread) stalls waiting for data arrival, another task takes over the processor in an attempt to keep it busy. Writing multi-threaded applications is complex, requiring an understanding of libraries that vary from system to system and also of ways to partition an application into concurrently executing tasks.

Another common optimization deals with balancing the workload between each parallel task through careful distribution of the work. For some applications this can be done statically before the program runs while others require a more dynamic solution. The goal of load balancing is to minimize the time the parallel tasks stall waiting for other tasks. For some applications this stalling is periodic during program execution, while for others it only occurs once at program completion.

In a multiuser environment, parallel applications are often alloted a time window in which the code can run. After the time has elapsed, the program must be stopped so that another user's application can access the machine. Since many parallel applications run for extremely long periods of time, it is imperative that an application can stop its execution and then resume at a later time. The operation of stopping a program, saving its state, and resuming its execution from that state is commonly known as checkpointing. Checkpointing has other uses such as migrating programs from one cluster to another, saving progress for runtime data analysis, or recovering after application or hardware failure.

There are many potential optimizations available to parallel computing applications. While each optimization is applicable in certain situations, it is not always clear which optimizations will be the most beneficial to a particular application. Furthermore, recognizing the characteristics that warrant a particular optimization often requires expertise in parallel computing. The fact that optimization implementation is often challenging further hinders their usefulness to those without detailed parallel computing expertise. This in turn limits the potential performance that most application domain specialists can expect to receive from a parallel implementation of their application.

## 1.4 Proposed Solution

Operating systems have long been used to incorporate common functionality into a central, reusable location. If the functionality (or features) can be generalized so as to be usable by a wide range of applications, then the operating system can remove the complexity of implementing these features in applications. Generalizing and incorporating features helps provide a standard interface to those features and allows their implementation to be abstracted away from the applications that utilize them. This is particularly important with respect to functionality that is often reused.

In the world of modern operating systems, we find more and more functionality being incorporated into the OS or in tightly coupled API libraries. These include memory management, disk I/O, external device I/O, multi-threading, atomic locking, and many other features. In much the same way, PSEs can be seen as *middleware* — a term used to describe software that resides between an application and a lower layer such as an operating system that provides a meta-environment, or virtual operating system in which the application context executes.

Many PSEs are specialized to a specific class of applications. With specialized PSEs, the environment often has some knowledge of the application's structure due to assumptions with respect to that application domain. Conversely, those environments which allow a more general class of application to be implemented suffer from having little knowledge of the application's structure. This lack of knowledge limits the ability of the environment to make optimizations on the application.

This work details a new model of computation called the Coven model. The Coven model is implemented by the Coven PSE framework and enables a large class of applications to be implemented in the PSE and at the same time reveals key information about the structure of the application to the PSE. This information includes application data types, access patterns, control flow, trace information, data decomposition, and locality. Much like an operating system, this then allows the PSE middleware to incorporate a number of important functions and optimizations into the PSE itself.

Given this new model of computation, this work attempts to answer the following questions: *Does this model facilitate the encapsulation of common parallel computing features and optimizations? If so, do these features provide competitive performance compared to conventional, hand-coded approaches?* To answer these questions, the model is first explained in detail and is then followed by a description of a PSE that implements the model. Then, three common, powerful, yet complex to program parallel computing code optimizations are implemented in the PSE and utilize the model. Specifically, these optimizations are multi-threading, load balancing, and checkpoint/recovery. We analyze the performance of these optimizations utilizing a trace and debug tool which itself is encapsulated within the PSE. Furthermore, the model facilitates the implementation of this tool and a number of other features.

The PSE which implements this unique model of computation is called Coven. By employing the model to incorporate many common optimizations, features, and functionality into Coven, we show that the model can effectively and efficiently simplify many tasks of parallel application development. Furthermore, Coven is a PSE that not only addresses the needs of parallel program development, execution, data visualization, and performance analysis but also performance tuning and checkpointing. This work shows that, through the use of a new model of computation, a PSE can be used to leverage common optimizations and features while providing them in a way that is easily accessible by average users.

## 1.5 Outline

The rest of this thesis is devoted to proving the utility of this model of computation for PSEs. Chapter 2 presents the Coven model of computation and then introduces the Coven PSE that utilizes this model. Coven is shown with a sampling of the applications that have been created with it to date. A number of common features which are facilitated by the Coven model are provided. These include:

- garbage collection,
- data serialization,
- data partitioning,
- data joining,
- task virtualization,
- data communication between parallel tasks,
- trace analysis,
- and performance debugging.

We then study the types of applications that can be implemented within Coven and demonstrate its generality with a range of sample applications. The modular programming interface is presented and we show that existing application code requires little augmentation to be converted into a Coven application. Additionally, a graphical user interface complete with a drag-and-drop program creation is shown. The graphical tool includes a performance debugger and trace analyzer which utilizes the model to gather its statistics.

To study the power of the model we implement three common yet complex to program parallel computing optimizations. These optimizations are encapsulated within Coven and employ the model to provide the optimizations efficiently, elegantly, and with little effort on the part of the application programmer. The specific optimizations studied are multi-threading, load balancing, and application checkpoint and recovery.

In Chapter 4 the multi-threading optimization is presented. Utilizing multiple threads of control, tasks are able to concurrently execute on a compute node while sharing application data (state). Both a synthetic and a real application are implemented and analyzed to compare the performance of a Coven implementation versus a hand-coded parallel implementation. We show a performance improvement on the order of 25% to 30% for the applications tested. Furthermore, the implementations are studied to compare the programming complexity. We show that using Coven's multi-threading feature is trivial while implementing multi-threading in a parallel application by hand is error prone, tedious, and complex.

Chapter 5 studies the use of dynamic load balancing (DLB) within Coven. We present two different forms of DLB which are applicable in different types of applications. One of the DLB approaches is completely transparent to the application programmer and the other requires simply inserting a pre-built module into their data flow graph. Both approaches are analyzed and shown to provide a performance improvement of around 15% for the applications tested. We study the complexity involved in implementing DLB within a hand-coded parallel application and conclude that Coven can provide the optimization with minimal cost.

An application checkpoint and recovery optimization is analyzed in Chapter 6. This optimization is shown to provide checkpointing in Coven with minimal effort on the part of application programmer. Due to the unique properties of the Coven model presented in Chapter 2, unlike similar checkpointing systems Coven's implementation does not require users to "tag" data that must be checkpointed. This makes adding checkpointing to an existing application trivial.

In Chapter 7, we conclude with a discussion of unique contributions.

# CHAPTER 2

# COVEN

In this chapter we present a new model of computation for PSEs called the Coven model. We then present the Coven PSE framework which implements this model. The modular programming paradigm used by application programmers in Coven is discussed in Section 2.3. The implementation of the Coven model and the resulting PSE middleware is presented in Section 2.4. The graphical front-end is then described as well as how a user goes about implementing a Coven application. We conclude with an overview of the applications that have been written using Coven.

## 2.1 The Coven Model of Computation

A PSE or middleware environment for parallel computing can be designed so as to abstract away some common complexity from applications. However, the amount of functionality which can be abstracted from the application and incorporated into the environment itself is limited by the environment's lack of detailed information about the application's structure. If the application exposes some of its internal structure to the environment, then the PSE is able to encapsulate some of the application's functionality. This is exactly what the Coven model of computation provides and it does so with minimal change to the original application's code.

In the Coven model, a program is defined as a tuple $< I, D >$ where $I$ is a set of instruction components (modules), and $D$ is a set of data components. The $I$ components are simply program code, without any program state. The $D$ components are program *state*. In the simplest case, an $I$-component and a $D$-component is a sequential program. Under the Coven model, $I$ consists of a set of codes or modules. A target application program is decomposed into a number of modules. The critical aspect being that modules cannot contain state, thus all relevant data must be passed in or out of a module.

Whenever a module executes with a $D$-component, the result is termed a task. A program execution consists of a number of tasks executed in order based on dependencies between the tasks. If data output by one task is input by another, a data dependency exists between the tasks. Similarly, output dependencies and antidependencies can exist between tasks as per the standard definitions. The dependencies define a partial order between the tasks that must be satisfied for a correct execution.

The single $D$-component of a sequential program can be partitioned, replacing the original state with multiple smaller ones. The data contained in the original state may also be contained in the smaller ones. Some data may be distributed among the smaller partitions and other data may be replicated. When $D$-components are partitioned, the tasks that were defined for the sequential program must be redefined as multiple tasks. In this case, it becomes possible to execute more than one task at a time, potentially as many as there are partitions.

There are a great many data flow model extensions [52], most of which are slight variances on the central theme. The Coven model is similar in that much of the basic idea derives from data flow programming. A similarity also exists with process networks (PNs) where asynchronous messages are used to communicate data between components. These channels implement data flow but PNs are often considered awkward for specifying control logic.

One of the key concepts of the Coven model is the separation of an application's code from its state. State in terms of the model is some high level construct which maintains a portion of the application's data. One of the key differences of the Coven model is the ability of tasks to modify state. As state flows between modules ($I$ components), those components may add to, delete from, split, join, or do many other similar operations to the state. The ability for state to be split or joined is unique from the perspective of the model as well. This concept marries themes from parallel programming and enables many parallel optimizations. Another point is that a $D$-component generally contains more than one buffer of data. This means that as $D$ components are used as input to $I$ components, the $I$-component is able to pick and choose the data that are needed. While the differences are slight, they enable some of the optimizations and features which will be shown in later chapters.

Tasks may create new data in the state, remove data, replace data, and other similar operations. One key operation in the Coven model is that of state partitioning and joining. To partition a state involves creating new output states from an original input state. These output states may be strict partitions of the original state, duplicate portions of the state, or any combination thereof. Conversely, joining states takes as input an arbitrary number of states and reduces them to a single output state.

With state being partitionable into sub-states and the stateless feature of code (modules), the model allows more than one task to execute concurrently at a time. The advantage of this model is that by decoupling the state of an application from the code, the application unknowingly exposes information about its structure to the environment. The environment then is free to use this to optimize the application and provide features which can be encapsulated in the environment. This then alleviates the burden of implementing these optimizations and features in the application. There are potentially numerous benefits including quicker application development, less error prone applications, more reusable application code, and many others.

In developing the Coven model of computation we employ some data flow principles and apply them at a macro level but aim to cause the least augmentation to existing application code. In the realm of high performance computing, programmers are most used to the imperative (procedural) programming paradigm. Converting code to the data flow model requires a fundamental shift in the way applications are programmed. This is why the Coven model separates the state from the data to allow the code to remain relatively unchanged and at the same time utilize a component programming paradigm.

In the following chapters this model is studied to analyze its effectiveness. Section 2.2 presents Coven, a PSE that implements the Coven model of computation. Chapters 4, 5, and 6 study three complex parallel computing optimizations and how they can be encapsulated within Coven through utilization of the model. The study of these optimizations includes case studies of sample applications, performance analysis, and comparison with conventional hand-coded approaches.

## 2.2 The Coven Problem Solving Environment Framework

Coven is a problem solving environment framework for parallel computers that implements the Coven computational model. By abstracting the common ingredients of many PSEs and placing them into the Coven framework, custom PSE creation in a target problem domain is simplified. Applications are developed within a Coven PSE through the use of modular, component programming. Coven modules expose their interface, and through this the system is able to gather information about the underlying application. This information is then used to also abstract away the complexity of many common operations which would normally require a programmer to explicitly code.

At a high level, Coven is composed of three main components: a module library, a front-end, and a parallel runtime system/engine back-end [31]. Figure 2.1 gives a brief overview of the Coven system architecture. Users implement modular code utilizing a module library repository (Section 2.3). Many pre-built modules exist which can be used directly or modified to fit the needs of a particular application.

Interacting with the front-end, a Coven user implements his application as a data flow graph. Modules are selected from the library and interconnected to describe how data will flow between them. A graphical user interface (Section 2.5.1), or GUI, is available to assist in this task and provides a familiar drag-and-drop interface. A code generator

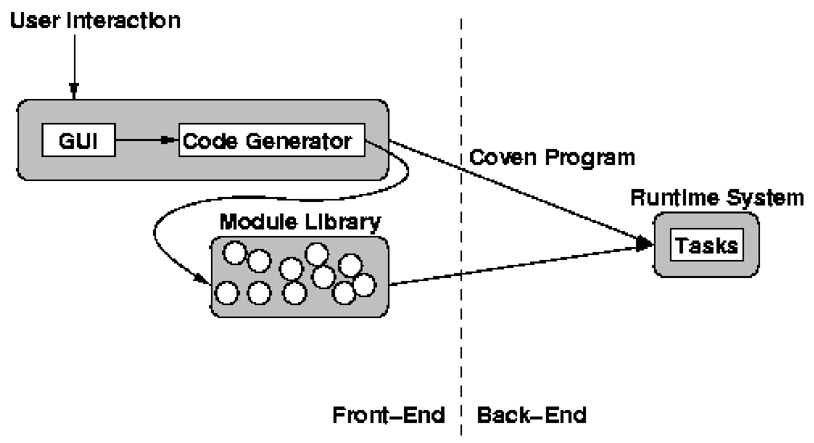


Figure 2.1: Architecture Overview

translates the data flow graph into a custom language (Section 2.5.3). Advanced users can program directly in this language if desired.

Once the data flow graph has been compiled, it is transferred with the chosen modules to the runtime system (Section 2.4). This component runs on a parallel computer back-end and executes the application and its modules in parallel. Each parallel process executes one Coven model task at a time with the parallelism coming from multiple processes each executing tasks on separate state partitions. Results can be sent back to the front-end for visualization and post mortem analysis.

## 2.3 Module Library

At the heart of writing Coven programs is decomposing applications into modules. Coven modules are pieces of C code that contain special directives describing the interface to the module. These directives are similar to a C function header and, while no automatic conversion system was built, it should be possible to translate from C source directly to a Coven module interface description.

With modular code design the programmer is able to insulate segments of code from others. This has the benefit of making the code more reusable. Much like a method in a library, a module (or collection of modules) can be used in many pieces of unrelated code. For example, a module can be implemented which multiplies a matrix by a vector to produce a vector easily by defining the matrix and vector as inputs and the vector as output. This module will find uses in very different applications.

Coven modules are contained in their own source file, generally with the `.cov` extension. A parser translates the module file into a C source file by replacing the Coven-specific directives with Coven library macro and method calls. These Coven calls include the use of many Coven internal data structures and a library API that we want to shield the user from having to understand. With this approach, the user can focus on describing the interface to their module. Additionally, by creating this layer it allows the Coven engine to change and as long as this layer remains the same, existing module code will continue to function.

Once translated into pure C, the modules files are transferred to the parallel computer and compiled into shared objects. Coven dynamically loads the modules at runtime, calls them in the order the user specified, and handles passing all data between modules. Since the modules are shared objects, Coven does not need to be recompiled each time a module file is changed. Additionally, it is possible to switch a module out at runtime for debugging purposes.

Modules can be broadly split into one of two categories: *application modules* and *system modules*. While there is no distinction in the implementation, it is helpful to categorize these classes of modules in this way to understand how Coven can facilitate collaboration between the application domain specialist and the parallel computing specialist. The following sections describe these classes of modules.

### 2.3.1 Application Modules

Application modules are those that deal with solving the implemented application and have little or nothing to do with parallel computing. The idea behind application modules is that they are easy to implement for the domain specialist and could run with or without parallel computing. This feature of Coven can be used to implement sequential code which runs entirely without the use of MPI or multiple processes.

With application modules, the user can specify parts of an algorithm without concern for how the data arrives at the algorithm. An example is the Normalized Difference Vegetation Index (NDVI) [22] algorithm, implemented as a Coven module and shown in Figure 2.2. The NDVI is a common satellite remote sensing algorithm. The module takes as input two arrays containing calibrated satellite channel data and produces as output the NDVI array. Notice that the module is self-contained in that the calibrated data arrays can be of any length, from any portion of any satellite image. This lends itself well to parallelism as at runtime this module may run on many parallel processors, each operating on a different portion of the calibrated data array.

### 2.3.2 System Modules

By contrast to application modules, system modules deal with issues related to partitioning data and transporting it between parallel processors. Programming this style of module often requires a more in depth understanding of the targeted architecture and some of Coven's more advanced functionality. Coven attempts to hide the complexity but to allow complete control; methods relating to data partitioning, distribution, and reassembly are provided to system module programmers.

Keeping with the modular programming approach, the goal of system modules is to allow a parallel computing specialist to design and implement code which handles efficient transportation of data to the application modules. Through well defined module

```
COVEN_Module sat_ndvi
  (
   input buffer float cal_c1[num_cal],
   input buffer float cal_c2[num_cal2],
   output buffer float ndvi[num_cal]
  )
{
        int i;

        for(i=0; i<num_cal; i++) {
            ndvi[i] = (cal_c2[i] - cal_c1[i]) /
                      (cal_c2[i] + cal_c1[i]);
            if(ndvi[i] > 0.7) ndvi[i] = 0.7;
        }

}
```

Figure 2.2: NDVI Calculation Application Module

interfaces, the parallel computing specialist needs only understand what data to provide to which modules and not be concerned with the science behind the module.

Figure 2.3 is a system module which distributes point mass bodies to parallel Coven processes. Rather than schedule one group of bodies on each processor, this module uses the input scalar `num_partitions_per_slave` to put multiple groups of bodies on each parallel node. The parallel computing specialist implemented the system module in this way in an attempt to gain benefits from multi-threading. Programming this module requires an understanding of a number of advanced Coven functions. Behind the scenes, Coven is placing the bodies into internal data structures and scheduling them to run on parallel Coven processes.

As with any modular programming approach, the goal is to make a module appear as a “black box.” This not only makes the module more usable in many applications, but also precisely specifies what the module does so that collaborative software development is simplified.

```
#include "medea.h"

COVEN_MODULE nbody_scatter_multiple
  (
   const int num_partitions_per_slave,
   inout buffer MEDEA_BODY my_bodies[total_num_bodies]
  )
{
        int num_slave_procs = COVEN_get_num_slave_processors();
        int num_data_chunks = num_slave_procs * num_partitions_per_slave;
        int target_num_bodies = (int)ceil((double)total_num_bodies /
          (double)num_data_chunks);

        int i, total_assigned_bodies;
        COVEN_size *array_of_blocks, *array_of_displacements;
        COVEN_Type body_type, scatter_type;

        COVEN_Type_contiguous(sizeof(struct Medea_Body), COVEN_BYTE,
          &body_type);

        total_assigned_bodies = 0;
        i = 0;

        array_of_blocks = (COVEN_size*)COVEN_malloc(num_data_chunks*
          sizeof(COVEN_size));
        array_of_displacements = (COVEN_size*)COVEN_malloc(num_data_chunks*
          sizeof(COVEN_size));

        while(total_assigned_bodies < total_num_bodies) {
            /* try and fit this many bodies */
            int num_bodies = target_num_bodies;
            /* starting at how many we've assigned so far */
            int start_body = total_assigned_bodies;

            total_assigned_bodies += num_bodies;
            /* if we've assigned more than we have, then back off by that
             * amount */
            if(total_assigned_bodies > total_num_bodies)
                num_bodies -= (total_assigned_bodies - total_num_bodies);

            /* now num_bodies is the number of bodies for this TPH to take */
            array_of_blocks[i] = num_bodies;
            array_of_displacements[i] = start_body;
            i++;
        }

        /* now that we have our displacements and blocks setup we can make
         * the MPI data type */
        COVEN_Type_indexed(num_data_chunks, array_of_blocks,
          array_of_displacements, body_type, &scatter_type);
        {
            struct COVEN_Collective_Action *col_action;
            struct COVEN_Distribution dist;

            col_action = (struct COVEN_Collective_Action*)malloc(1*
              sizeof(struct COVEN_Collective_Action));

            COVEN_Create_Scatter_Action(&(col_action[0]), scatter_type,
              "my_bodies");

            COVEN_Create_Blocked_Distribution(&dist, num_slave_procs,
              num_data_chunks / num_slave_procs);

            COVEN_Scatter_Distribution(num_data_chunks, col_action, 1, &dist);
            free(col_action);
        }

        COVEN_free(array_of_blocks);
        COVEN_free(array_of_displacements);
}
```

Figure 2.3: Data Distribution System Module

## 2.4 Runtime Engine

Coven's runtime engine is the component responsible for executing the user's application. Often referred to as a back-end component, the runtime engine executes on a parallel computer. Technically speaking, there is no requirement that the architecture be parallel, just that it have an MPI implementation.

Many individual components make up the runtime engine (also called the runtime system). Each component handles a portion of the task of executing a user's application, including such operations as: data encapsulation, module execution, work queues, memory maintenance, and transportation of data around the system. The runtime system is described in detail in the following sections but an overview first is helpful.

Figure 2.4 depicts a very detailed view of the entire Coven system. The portion outlined in black, marked *DRIVER*, represents the segment contained in the runtime engine. The figure describes how TPHs (units of work, Section 2.4.1) are created by the memory manager, put into input TPH queues for processing on the parallel processes (program sequencers), and how the program sequencers then process the TPH by passing it through modules. Notice the modules are loaded dynamically by the module loaded from a module library. Modules are placed in the module library after having been created for use in this program. The library may contain other general purpose modules including those from other applications. The TPHs interact with local memory and, upon completion, TPHs are sent back to a component (the reassembler) that reassembles the data, or at least completes the application.

These components, as well as some important concepts, are described in the following sections.

### 2.4.1 Tagged Partition Handle

Tagged Partition Handles (TPHs) are one of the core and most original concepts of Coven. TPHs are a completely internal data structure used by the Coven environment to implement

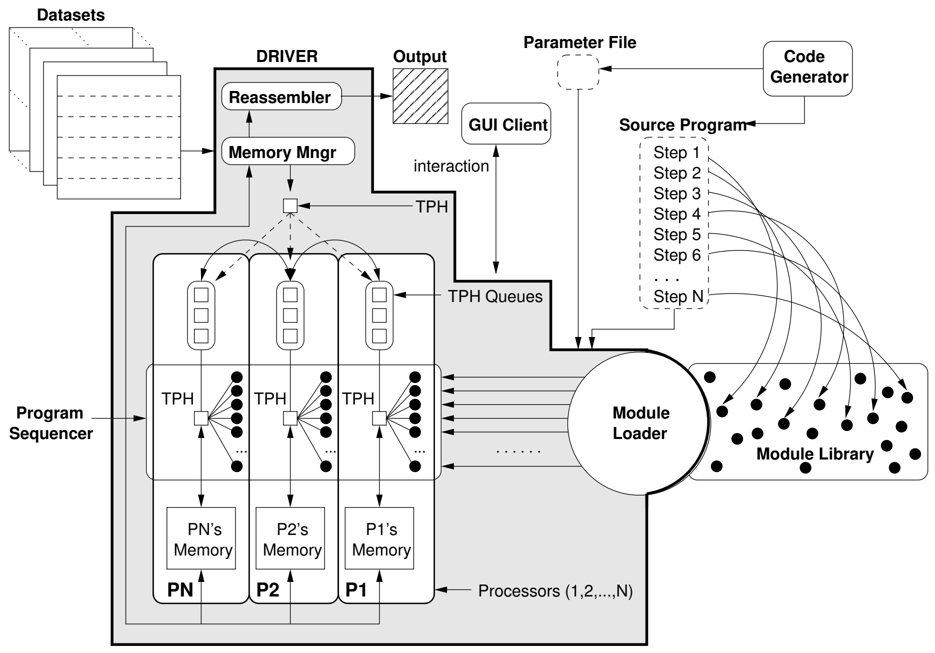


Figure 2.4: Detailed Architecture Overview

the stateless feature required by the Coven model. Transparency from the user means that implementation details and complexity can be hidden. Additionally, the TPH abstraction layer hides the complexity of the data structure behind an easy to use interface and allows a programmer to interact with the TPH transparently.

All data in Coven that passes between modules and Coven tasks is contained inside of a TPH. When a module reads data as input, it is reading it from a TPH. When it creates new output, it is creating it inside of a TPH. TPHs can contain any number of buffers (arrays) and scalars. All the common C data types are valid types in TPHs and users can even create their own data types and store them inside of TPHs.

TPHs can be thought of as a way to keep related information together. Imagine an image file that is split into four segments, each segment being contained within its own TPH. As those TPHs flow through the application's modules, new data is likely to be created by using the image file segment as input. This new information is attached to the original segment. This means that as a TPH moves through the system it keeps related information about that image segment together. This fact simplifies the concept

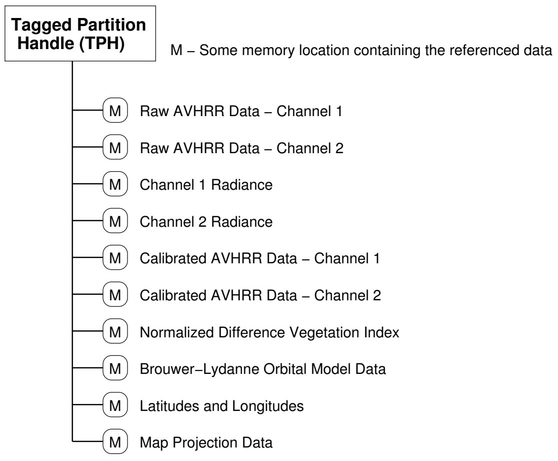

Tagged Partition Handle (TPH)
M – Some memory location containing the referenced data
M
Raw AVHRR Data – Channel 1
Raw AVHRR Data – Channel 2
Channel 1 Radiance
Channel 2 Radiance
Calibrated AVHRR Data – Channel 1
Calibrated AVHRR Data – Channel 2
Normalized Difference Vegetation Index
Brouwer–Lydanne Orbital Model Data
Latitudes and Longitudes
Map Projection Data

Figure 2.5: Tagged Partition Handle Example

of transferring TPHs through the system, as generally speaking, all needed information is self-contained within that TPH.

One can liken TPHs to packets in a network. Much like packets, TPHs contain arbitrary data and are wrapped up into a data structure that flows around the system. Header information details where the TPH (and packet) is destined as well as hints to what kind of information is contained within. Similarly, much as a packet flows around a network between hosts, TPHs pass between modules and parallel processes.

Figure 2.5 depicts what a TPH might look like at some point during an application. The TPH shown in the figure represents one from a satellite remote sensing application. The *M* depicts a buffer attached to some memory location. Beside each buffer is the *tag* name. The TPH initially was empty, then was defined to constitute some portion of the image. As the application progressed, more and more buffers were created by modules. Those modules read previous buffers as input in order to calculate new output. Since inputs may be necessary to later modules, it is not possible to free a buffer once it is used but instead Coven must determine when the buffer will be last used. This is discussed in Section 2.4.5.

The TPH is the central and most important data structure in Coven. All application data is contained within TPHs and, through Coven's comfortable API, modules transparently manipulate data within TPHs. By encapsulating all application data within TPHs, Coven is able understand many properties of the data. These properties include data type, number of elements, size (in bytes), usage pattern, memory location, and others. Using this information, Coven is able to incorporate many common data operations and parallel application optimizations into the runtime engine itself. For instance, each buffer has stored with it the name, type, size (in bytes), and number of elements. This makes it easy for Coven to marshall it and send it elsewhere. Marshalling in Coven is the act of taking a TPH data structure containing many pointers and many separately allocated memory regions and folding it into a single, sequential stream of bytes which can be used to completely replicate the TPH. This is useful for sending the TPH across the network, or writing it to disk.

The TPH enables the environment to move, allocate, deallocate, save, and restore partitions of an application's state. The tagged nature of TPHs is not required by the Coven model and is employed to simplify the application building procedure [26]. Fundamentally, by having users describe their applications and data explicitly, Coven is able to determine the structure of the application. Then, optimizations and features common in parallel computing can be abstracted away from the user and placed into Coven itself, simplifying the implementation issue of applications and broadening the usability of these features and optimizations.

### 2.4.2 Program Sequencer

The Coven runtime engine executes each module sequentially. Parallelism comes from having multiple runtime engines each executing modules on different TPHs. This concept is often known as programming with parallel components. As each module is executed, a TPH (representing a portion of the data) is passed into the module. The TPH holds both input and output buffers. The Coven component responsible for calling the modules and passing the TPHs between them is called a *program sequencer*. Each parallel task executes a single program sequencer. Later, in Chapter 4 when multi-threading is discussed, each Coven Thread will be shown to execute its own program sequencer.

At startup, each program sequencer examines the Coven program file (data flow graph) and determines which modules it is assigned to process. Each then dynamically loads these modules and maintains pointers to them in an internal data structure.

Due to the sequential ordering of the modules, TPH flow is simplified. When a TPH arrives for processing, the program sequencer merely calls each module in sequence, passing the TPH into each module as an argument. Figure 2.6 represents a simplified view of a program sequencer. This example includes five modules (represented as circles) and at least four visible parallel tasks, each with their own program sequencer. Each program sequencer has a TPH (represented as a small square) flowing through it. In the first and second program sequencer, the TPH is being processed by a module. In the third program sequencer the TPH has just exited one module and is about to be passed into the next one in sequence. Finally, in the last program sequencer shown, the TPH is leaving the program sequencer as it is destined for another parallel task.

Coven users can use loop structures in the data flow graph and the program sequencer is responsible for implementing this functionality. A series of modules may be grouped together to form the body of a loop. The user specifies the loop parameters, such as start value, increment value, and end value. The program sequencer implements the loop, checking break conditions at the borders of the module group. Therefore, while the module order is sequential in nature, the program sequencer is able to move TPHs around in any arbitrary order.

### 2.4.3 TPH Queues

Each program sequencer may have assigned to it at any time a number of TPHs that are pending to be processed. Coven implements this through a FIFO queue of input (to be processed) TPHs. When idle, the program sequencer dequeues a TPH from this queue if

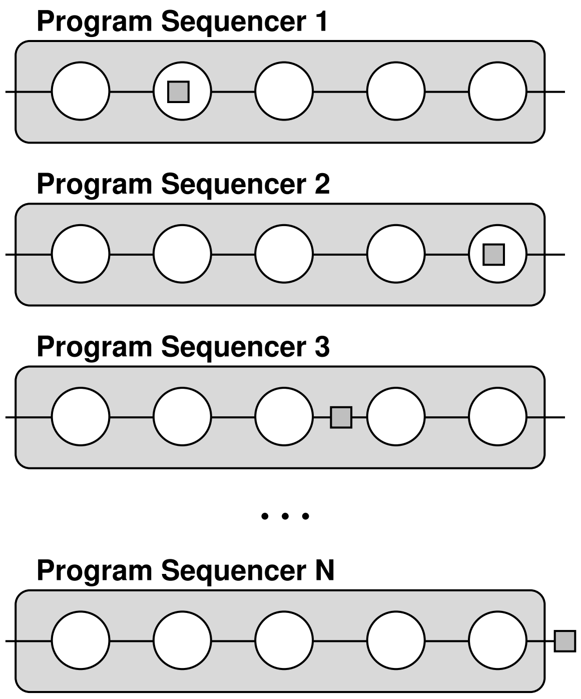


Figure 2.6: Program Sequencer

any are available. The TPH is then passed through modules as described in the previous section.

Generally speaking, once a TPH has been processed through all modules, it is scheduled to be sent to another parallel task. Usually this is some master process which coalesces all the data together into a final result. There are, however, other instances where TPHs may flow between parallel slave tasks. To address this, each program sequencer has its own output TPH queue as well. TPHs are placed there awaiting transfer to a destination process. Figure 2.7 depicts this queue system.

Coven's input queue is implemented through a collection of asynchronous MPI receive calls. A program sequencer issues a number of `MPI_Irecv` instructions, saving the request handle of each. It then goes to sleep waiting for any of the receive messages to arrive, signifying that a message is incoming. Messages usually contain marshalled (serialized) TPHs but may also contain control messages. Once a TPH is received and is demarshalled it is placed in the TPH input queue awaiting processing.

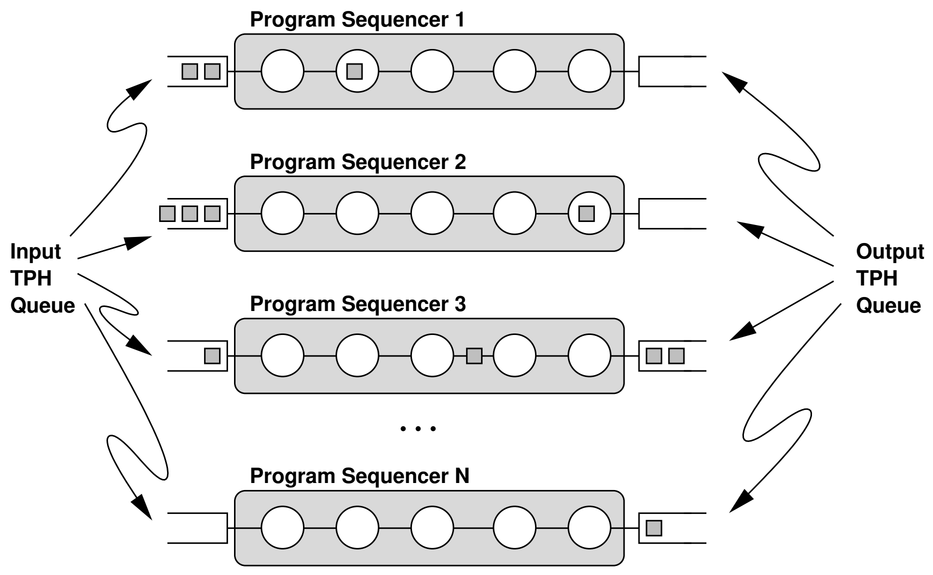


Figure 2.7: Program Sequencer With TPH Queues

The output queue is implemented similarly. As TPHs arrive in the output queue, an asynchronous MPI send is issued containing the TPH data and a control header. The program sequencer keeps track of this send operation just as it monitors the asynchronous receive calls for the input queue. Using the handle to the send operation, MPI notifies the program sequencer once the operation has completed. In MPI, this notification does not necessarily mean that a destination process has received the message but more specifically means that the data contained in the message may be deallocated.

### 2.4.4 Flow of Messages and TPHs

Figure 2.8 depicts Coven's message flow state diagram. Each program sequencer follows this same graph. The first thing to notice is that if there are no pending messages and no TPHs queued pending processing, then the program sequencer goes to sleep awaiting a message. This is implemented as an `MPI_Waitany` which releases the CPU and puts the program sequencer to sleep until a message arrives. Coven uses an `MPI_Waitany` because there are many asynchronous parallel receives it has started and needs to be alerted if any of them complete.

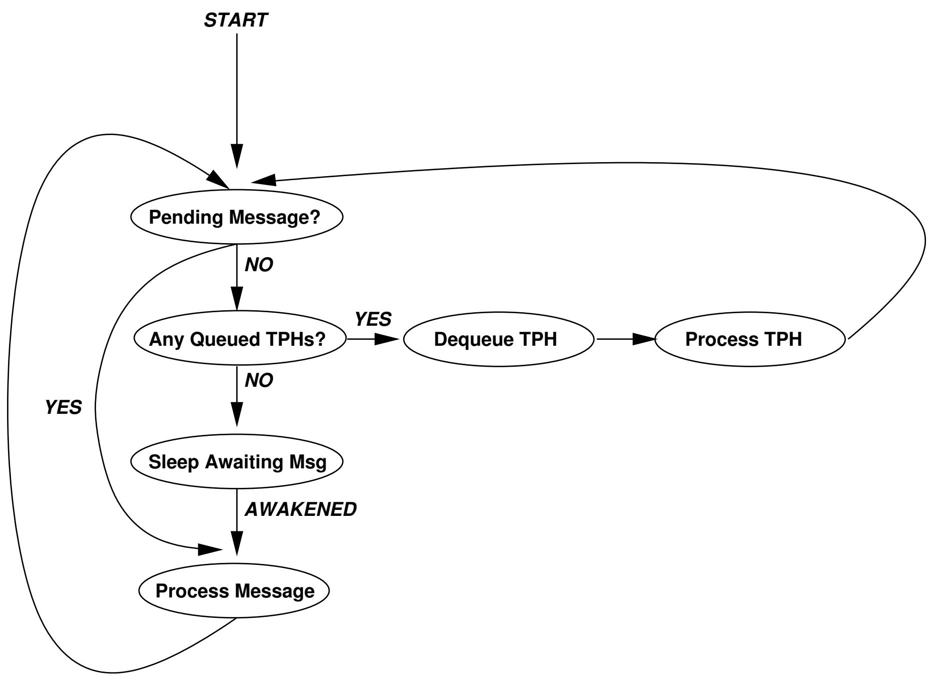


Figure 2.8: Message Flow State Transitions

This model causes every program sequencer to immediately go to sleep waiting on work to arrive. Therefore, the system needs an operation to kick start it. This is accomplished by having the master task inject an empty TPH into the first module in the system.

As long as there are pending messages incoming, Coven prioritizes processing of those messages over processing TPHs. Once there are no messages currently available for processing, Coven attempts to dequeue a TPH from the input TPH queue. If there are any TPHs available, the TPH is dequeued and processed. Afterwards, the program sequencer returns to the initial state where it checks for pending messages.

While Figure 2.8 depicts how Coven transitions between states with respect to message flow, it is an over simplification. The processing of a TPH generally results in the TPH being placed into an output queue, having been marked for transfer to another Coven task. This causes asynchronous MPI send operations of both control and data to the destination task. Coven then includes those messages in its list of those that it is waiting on for completion. Similarly, the Process Message state in Figure 2.8 includes a large number of operations. These include demarshalling incoming TPHs, inserting TPHs into the input queue, removing received TPHs from the output queue, and a number of “bookkeeping” operations.

### 2.4.5 Garbage Collector

As TPHs flow throughout the runtime engine they are passed into and out of modules. Those modules use input data contained in the TPH to create new output buffers in the TPH. This can lead to buffers that, once used, are never needed again for the lifetime of the program. Rather than provide a way for users to free buffers themselves, Coven employs a garbage collection mechanism.

Coven’s garbage collector determines the last point when each buffer will be used by examining the program source file, the arguments to each module, as well as the module interface. The garbage collector then recognizes when a buffer will be last used as input to a module and schedules it to be freed at that point.

Automatic garbage collection helps keep memory usage down while at the same time removing the burden from the programmer of having to free their own buffers. Furthermore, in a modular system where modules are intended to be self-contained, knowing when to free a buffer requires constructs at a higher level. The program source language (described in Section 2.5.3) is where a user specifies the modules to execute, parameters to pass them, and in what order. This is the level at which a user would be forced to deallocate unneeded memory. The Coven garbage collector can relieve a user of this burden by automating this process.

### 2.4.6 Intermodule Communication

While Coven simplifies the operation of partitioning work, distributing that work to parallel tasks, and reassembling that work, a vast majority of parallel applications must use additional communication to function properly. The runtime engine uses MPI internally,

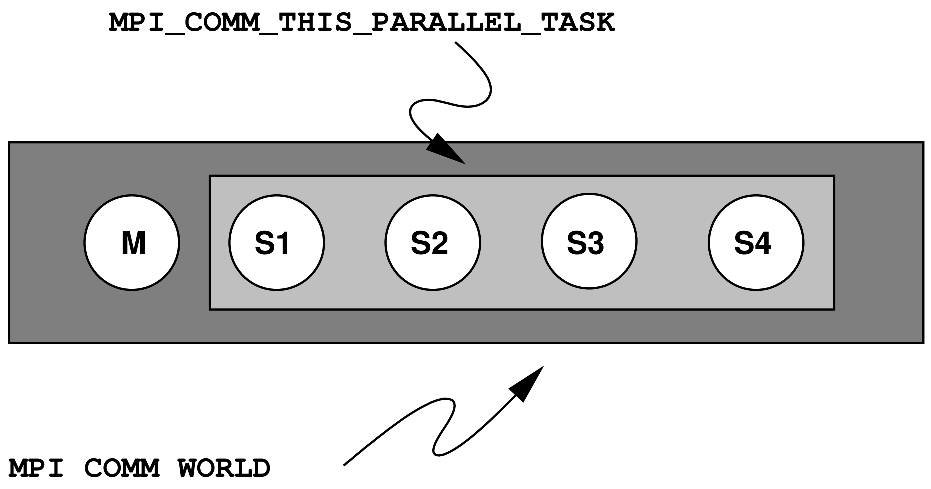


Figure 2.9: MPI Communicators

---

and the Coven modules are free to use MPI themselves. There are, however, several issues which module developers must be aware of.

Firstly, Coven operates under a *master / slave* model. Even for applications that do not need a sequential, master portion, Coven molds the user's application into this model. The result of this is that parallel operations must be aware that they are not the only MPI tasks in the system. To address this, Coven provides an MPI Communicator other than the default `MPI_COMM_WORLD` for use in parallel communication. This communicator, named `MPI_COMM_THIS_PARALLEL_TASK`, is structured so that each parallel task is a member of the group. Figure 2.9 provides an explanation of the organization of these communicators.

The use of this new communicator makes it easier for module writers to implement MPI code because it functions more like `MPI_COMM_WORLD` does in an application that is not forced into a master / slave model. Additionally, this makes it considerably easier to port existing MPI code to Coven modules, merely requiring changing `MPI_COMM_WORLD` references to `MPI_COMM_THIS_PARALLEL_TASK`.

### 2.4.7 Scatter and Gather

Coven provides a mechanism to break a TPH into an arbitrary number of TPHs and then distribute those TPHs to the parallel tasks. This operation in Coven is called a *scatter*. When scattering a TPH, the user can choose from simple, built-in data distribution patterns or describe their own, more complex, distribution. For this complex distribution, Coven employs a mechanism very similar to MPI data types. Users familiar with MPI user defined data types will find the interface very similar as well.

When $N$ TPHs are created by a scatter, by default each TPH contains $\frac{1}{N^{th}}$ of the initial data in the TPH. For instance, if a TPH contained several buffers with different size images contained in them then each TPH after the scatter would contain a separate portion of the data. Technically, Coven does not require that the data be split into equal, $\frac{1}{N^{th}}$ sizes. However, it is easier to understand conceptually.

Once each TPH is created by the scatter, they are distributed to the parallel tasks either using a round robin fashion or through a TPH to parallel task mapping. This mapping is discussed in more detail in Section 2.4.8. Arriving in a parallel task's input TPH queue, each TPH awaits processing.

It is generally useful to rejoin the parallel TPHs back into a single TPH. This is common in applications that divide the work and then need to reassemble it into a single solution before (for instance) writing the result to disk. In Coven, we call this rejoining of TPHs a *gather* operation. There are instances where gathering is not necessary, such as when the parallel TPHs handle writing the final solution to disk in parallel. This is still possible in Coven, but the functionality is provided to help alleviate the programming burden if it is necessary in an application.

The gather operation works essentially like the opposite of the scatter. A data partitioning pattern can once again be described and used to gather the data into a single TPH. It is important to note that this pattern may be different from the way it was initially scattered. Once gathered, the $N$ TPHs form one TPH and that TPH continues on from the gather module.

Both scatter and gather must be contained within Coven system modules. The Coven distribution provides several that are easily reused for an application or modified to suit an application's particular needs. The goal of these functions is to provide a drop-in solution for partitioning and rejoining parallel data so that the user does not need a detailed knowledge of how to do it. While Coven has many pre-built examples (blocked partitioning, blocked cyclic, etc.), the application programmer has the power to devise much more complex distribution patterns to suit their needs.

### 2.4.8 Virtualization

Coven encourages programmers to implement their Coven applications so as to be irrespective of the number of compute nodes where the application will eventually run. This makes the code easier to reuse and port between architectures with different configurations. Additionally, programming in this manner can lend itself to some of the optimizations we will explore in later chapters (such as dynamic load balancing). The user implements their code assuming a certain processor configuration and then maps that to the physical configuration that they chose to execute on. This is termed *virtualization* in Coven.

Dataparallel C [47] had a concept of what was termed *virtual processors*. The Dataparallel C program was written with directives which created virtual processors. These virtual processors were then mapped to physical processors. In many ways this simplifies the implementation and can improve performance of an application. For instance, it is very easy to change grain size by merely creating more or fewer virtual processors. This can ease implementation of applications that otherwise would have required out-of-core computation techniques.

There are other benefits, specifically with respect to load balancing. In a system where some of the processing elements execute faster than others, virtualization makes it easier to use a small grain size and schedule multiple data segments to run on the faster processors, leaving the slow processors only a few data segments. With this approach, a faster processor might get four times as much work as the slower. Coding this without using virtual processors would require modifying the application so that the code was designed for different size segments.

It may be more complex for some users to think about implementing code using the concept of virtual processors. Coven allows applications to be written in either manner. Both approaches (using virtual processors or not) are viable and useful in different circumstances. The idea is to make both available to Coven users so that they can choose.

One key problem with virtualization is how to address the issue of communication between virtual processors. Figure 2.10 depicts a dataset divided into 64 segments which are to be mapped onto four processors. The gray upper right-hand-side region is selected to be mapped to processor P3. In this example, the user has chosen 64 virtual processors. In Coven, each of the 64 segments will be represented by its own TPH.

The interesting case arises when the segments need to communicate with other segments. This is very common, for instance, in an application that needs to calculate its value based on values of its neighbors. Remember that, in this example, the 16 TPHs assigned to each parallel task end up in FIFO order in the TPH input queue. Therefore, it is likely that P3 will begin by computing segment 5 (from the figure). Segment 5 might then issue one send to 4, 6, and 13 (west, east, and south of itself respectively) and also a receive from those. The problem resides in the fact that segment 5's response from segment 6 (and 13 for that matter) cannot arrive until the TPH responsible for segment 6 is dequeued and processed.

To facilitate this, Coven allows programmers to communicate between virtual processors by simply specifying a TPH number to communicate with. These messages are implemented behind a Coven API which issues an asynchronous send to the target TPH. In the above example, when segment 5 posts a send to 6 (and 4 and 13) the TPH is marked

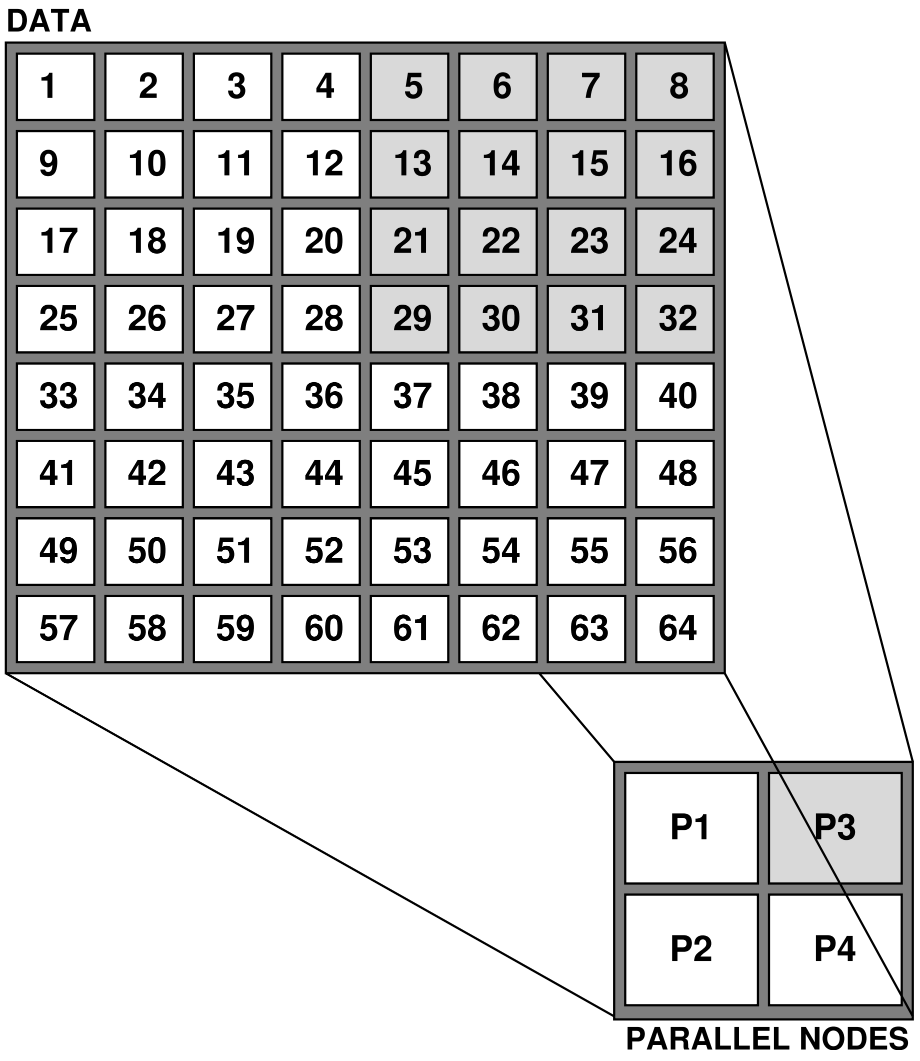


Figure 2.10: Virtualization Example

as unable to proceed until those messages arrive. Coven then places the TPH into a holding location, awaiting the conclusion of those communication transactions. In the example posed in Figure 2.10, TPH 6 would then be scheduled. This would post sends and receives to 5, 7, and 14. The exchange with TPH 5 would complete on both ends, but until each of the communications has completed, the TPH cannot proceed. This pattern continues until there are no more pending communications associated with each TPH.

The above scenario is complex and, in many cases, would result in poor performance if the choice of grain size is too small. However, the concept remains valid and the above reasons mentioned still make it useful in some instances. In particular, virtualizing in this way can make dynamic load balancing possible as will be shown in Chapter 5.

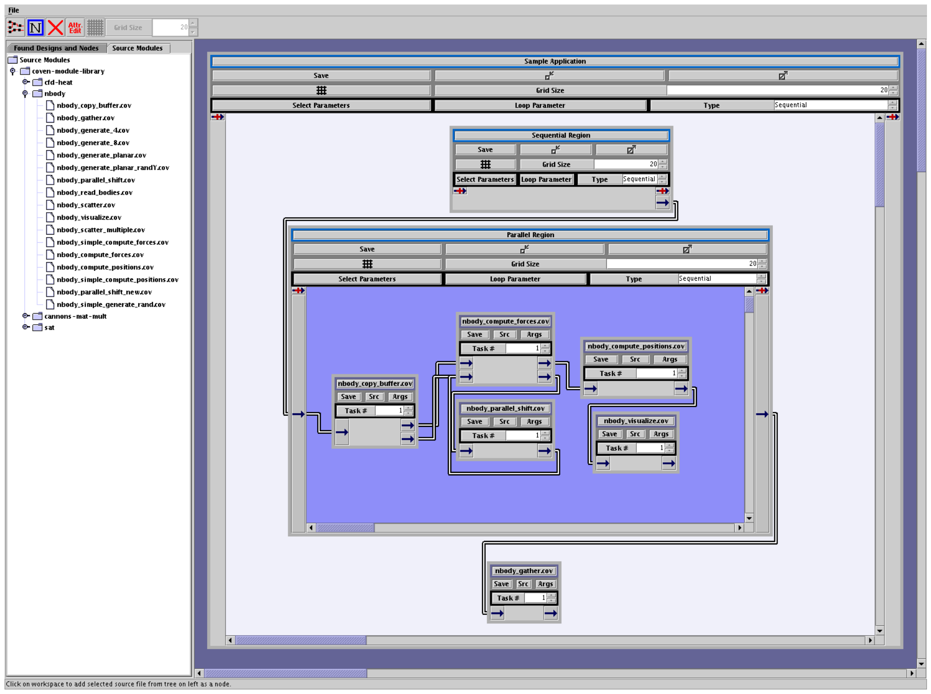
Finally, the scatter and gather functionality presented in Section 2.4.7 handles a great deal of the complexity involved with creating and rejoining virtual tasks. Specifically, the scatter Coven library call allows the user to specify a virtual processor topology and creates TPHs to map to that topology. Then, the distribution functionality maps those TPHs to physical processors — creating the mapping between virtual and physical processors.

## 2.5 Front-End

Coven's front-end system is used to construct a data flow graph, initialize, and start the parallel computation. Users can employ a graphical data flow editor or a custom programming language to design their applications. The following sections describe these components in detail.

### 2.5.1 Graphical User Interface

The easiest way to assemble Coven programs is to use the graphical user interface (GUI). Coven's GUI is written entirely in Java and is composed of approximately 60 interacting classes. Figure 2.11 is a snapshot of the GUI representing a Coven N-Body application.



Click on workspace to add selected source file from tree on left as a node.

Figure 2.11: Graphical User Interface

---

Users select Coven modules from a tree-based interface and drag and drop them onto the canvas. The modules are parsed as they are inserted and Coven automatically creates a graphical component to represent the module. This component includes input and output ports based on the module interface. Additionally, required module parameters (such as the gravitational constant, initial mass, and other constants) are input using a pop-up window. Modules are interconnected to define data flow by simply clicking on a pair of input and output ports. Coven then draws a line between the module's ports to express the flow of data.

Some segments of a Coven application are sequential, while others are parallel. Modules that execute on the master task are sequential while those that run on the slave tasks are parallel modules. This is represented in the GUI by placing modules into a container. In the screenshot (Figure 2.11), the large rectangle in the middle is one such container. This container has five modules in it. Containers can be thought of as black boxes, as they encapsulate modules and can be closed (or minimized) to simplify the view of the program.

The user employs the GUI to construct their application, defining data flow and setting initial parameters. Once the application is complete, the user clicks a ‘go’ button which transfers the job to the runtime engine to execute in parallel. This process involves first translating the code into a custom programming language and compiling it. This language is discussed in more detail in Section 2.5.3.

### 2.5.2 Arches

Coven uses the Arches framework for creating the GUI. Arches is a project which was developed to be a reusable framework for PSE creation. The project extended from the Clemson Environment for Computer Aided Application Design (CECAAD) [78, 77] project. The main idea behind Arches-based PSEs is that there is a persistent data structure (described below) that is acted on by several independent *agents*. The agents — or *tools* — are expertise-adding programs that annotate the data structure. A PSE-specific system generator is responsible for emitting an application (and, possibly, a run-time system) suitable for deploying on the target HPC system.

The agents are critical to extensible design. For example, profile information from a prior execution can be used by an agent to analyze a particular computation’s organization. The agent can suggest or effect the reorganization. While agents are superficially similar to compiler “passes,” this example points out a striking difference. Our model assumes that agents are semi-automatic; that is, agents have the option of interacting with the user.

The underlying data structure is a persistent, attributed data flow graph, which represents a program design. There is a single manager object which is used to create, save, and load data flow graphs. Each agent runs in its own thread, and the manager arbitrates access to the designs that are stored in shared memory. It is common, for example, to have an editor (implemented as an agent) running with an open design while another agent is

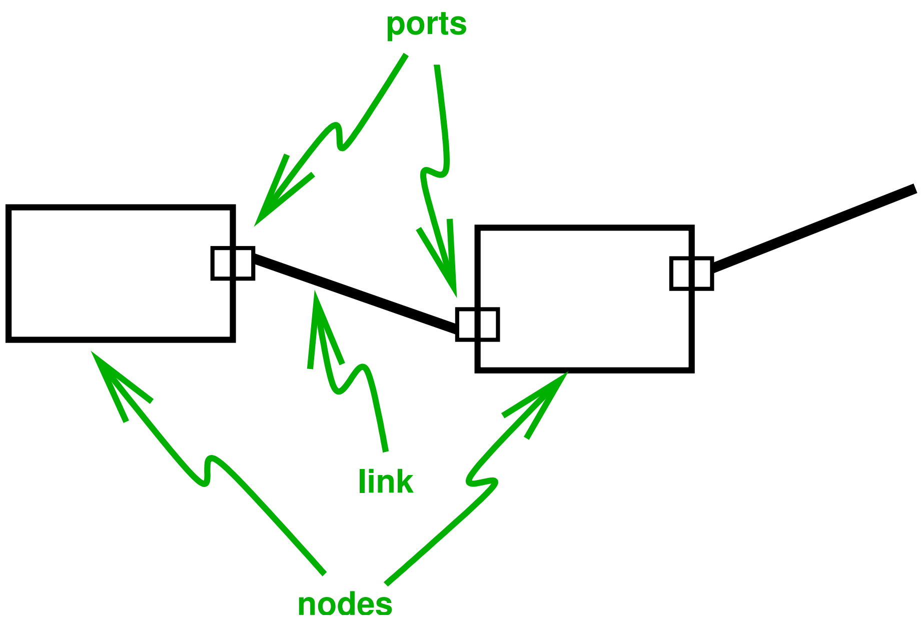


Figure 2.12: Arches Entities in Persistent Data Flow

transforming the same design. The manager ensures that the agents remain consistent. The graph itself consists of three primary entities: nodes, ports, and links, as shown in Figure 2.12. All entities have attribute tables that store persistent information. This includes information that may have been determined by previous agent invocations or from prior executions of the application. Attribute tables are themselves valid attributes and nested attribute tables are used extensively to organize agent-specific attributes. Similarly, there is a fourth entity called a design which extends node and is the algorithmic unit for holding a composition of nodes. Specifically, a design entity is a collection of nodes and links. Since it is an extension of node, it inherits the input and output ports of node. A more complete description of the relationships between the Arches classes, in UML notation, is shown Figure 2.13. Furthermore, because design descends from node, it can be used anywhere node can be used. Thus, complex designs can be built hierarchically.

In [28] we demonstrate how Arches has been used in Coven and RCADE. RCADE is a PSE for creation of FPGA-based reconfigurable computing applications. RCADE applications are composed of FPGA components implemented in a hardware definition lan-

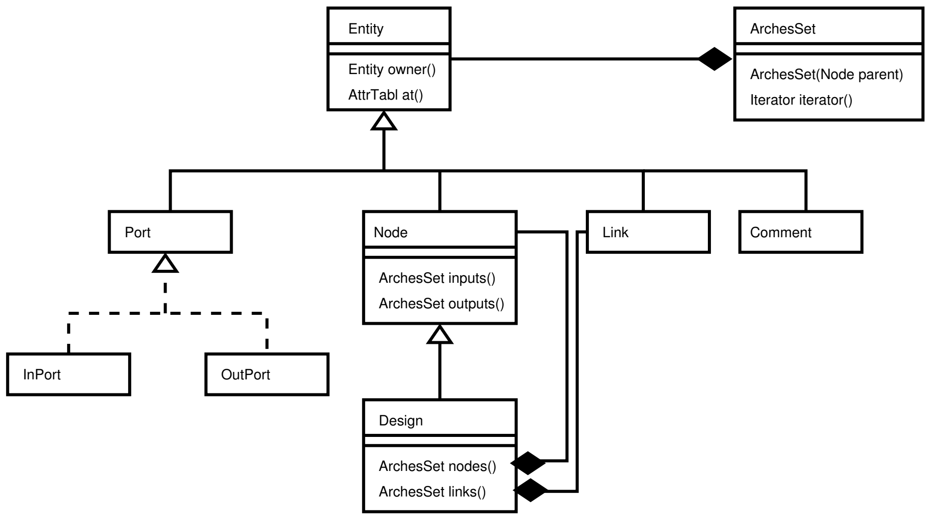


Figure 2.13: Arches Entities in Class Diagram

guage which are then synthesized. On the other hand, Coven applications are composed of modular C code which is compiled, dynamically loaded, and run by a parallel run-time engine using MPI on a parallel computer. These two PSEs are radically different in the problem domains they address but reuse much of the PSE infrastructure provided by Arches to ease PSE implementation.

### 2.5.3 Coven Language and Code Generator

A Coven user has two ways to enter their application; either through the graphical user interface, or using a custom source code language. The GUI was described previously in Section 2.5.1. When the user completes assembly of their application using the GUI, the data flow graph is translated into the custom source code language. Therefore, the GUI can be seen as an interface which simplifies creation of this text file that specifies the data flow.

The Coven language file lets the user specify the modules and their interfaces. Next, variables and constants are declared including their types and initial values. Finally, the data flow (call order) is declared using a C-like syntax that includes the arguments to each module. The multi-threading extension described in Chapter 4 augments this syntax to allow a user to specify which threads a module executes within. The Coven Program Writer's Guide [26] describes the Coven language in detail and includes an example.

Before the application is transferred to the runtime engine running on a parallel machine, the modules must be translated. Users program Coven modules in C, but specify the interface to the module using a custom set of directives. Coven translates this into a collection of C macro calls. In essence, this acts as an abstraction layer which helps to simplify module programming. The Coven Module Writer's Guide [25] describes how modules are programmed and explains these custom directives. The Coven Tutorial [27] explains how to use the GUI to generate the Coven language file, compile both the language file and modules, and begin execution of a parallel Coven job.

### 2.5.4 Performance Profiler

An important aspect of parallel programming is the ability to do performance analysis and tuning as well as simply observe the steps an application took at runtime. Debugging parallel programs is notoriously difficult. This is due to many factors including few quality open source parallel debuggers. Even using debug print statements can be misleading as the order in which they eventually propagate out cannot be assured in parallel. Additionally, most parallel computers do not use global clocks as the cost to synchronize them can be very high. This makes debugging through time-based print statements error prone as well. Since Coven is aimed at easing parallel application development, particularly for the non-parallel computing specialist, we must look for ways to assist the user in analyzing the execution of a Coven application.

While at this time Coven does not provide a debugger, it does have a trace analysis Arches agent. This trace analysis agent analyzes log files post-mortem from a Coven application and displays the information in a number of dynamic formats.

Because of the way that the Coven model causes the application to expose its structure to the environment, the Coven profiler is able to gather statistics with amazing detail.

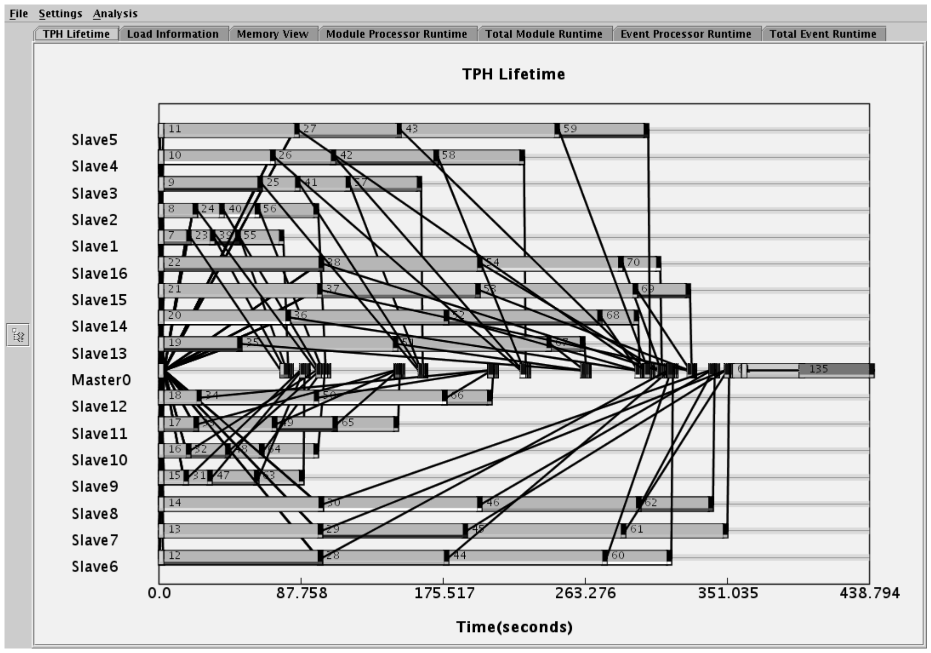


Figure 2.14: Profiler Screenshot - 16 Parallel Coven Tasks

TPHs are examined as they pass between modules and execute. The profiler is able to log statistics regarding the type, size, and usage information of data contained within TPHs. It is also able to log information about modules such as their runtime and CPU utilization.

During job execution, Coven's runtime engine logs trace information about the running job. The logging facility provided by MPE [7] is used for this purpose. MPE is a well known add-on to MPI which provides an automatic logging facility. Coven extends MPE's set of events that can be logged and adds Coven specific data to the log file. This allows the Coven profiler to access information such as memory usage, CPU usage/load, Coven thread statistics, module statistics, and more. Using this information, the profiler presents a number of different visualizations (or views) which the user can interact with to analyze trace and performance data. While there are many profiler views, Figure 2.14 depicts a single Gantt chart view of 16 processors performing an image processing algorithm in parallel.

The profiler has some built-in intelligence which analyzes a log file and attempts to suggest performance enhancements. These enhancements specifically are directed towards multi-threading (Chapter 4) and dynamic load balancing (Chapter 5). In the future, we hope to make the profiler more of an *expert agent*, suggesting a number of optimizations and possibly even automating the changes.

Throughout this thesis the profiler will be used to reveal performance problems, optimization suggestions, and then show improved performance after an optimization.

## 2.6 Sample PSEs and Applications

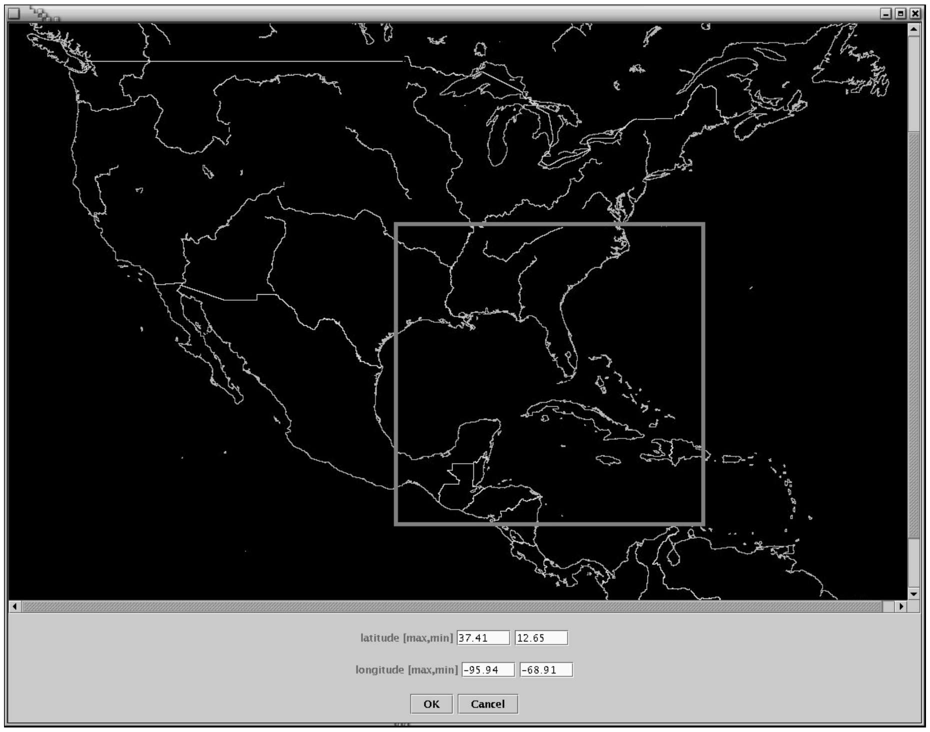
Coven has been used to create several different classes of applications. These applications were implemented to stretch the limits and test the capabilities of Coven. They demonstrate the versatility of design and help to argue against the numerous PSEs that exist that are only able to implement certain classes of applications.

In the next sections, we quickly introduce a few of the PSEs that Coven has been used to implement. This includes a description of the problem domain they target and a sampling of the Arches agents that are available to make their use easier.

### 2.6.1 CERSe

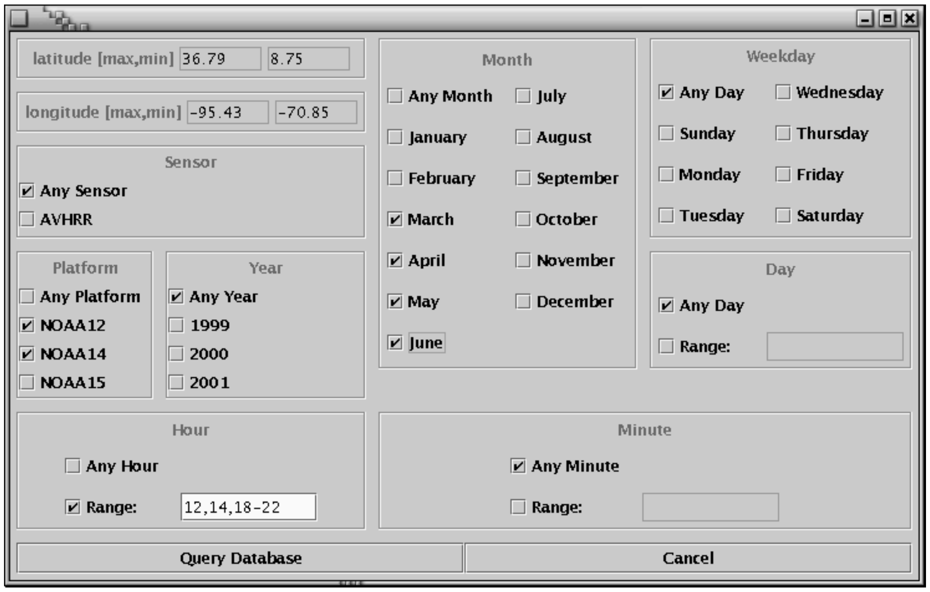
CERSe stands for the Component-based Environment for Remote Sensing [29, 30, 32] and is a PSE targeting satellite remote sensing and telemetry applications. Created for NASA, many AVHRR [22] (Advanced Very High Radiation Radiometer) algorithms have been implemented in CERSe. These algorithms include NDVI (Normalized Difference Vegetation Index) which calculates a relative “lushness” of the earth, SST (Sea Surface Temperature), and cloud detection / masking. Additionally, core AVHRR algorithms have been implemented, such as georectification, satellite calibration, and the Brouwer Lydanne orbital model.

One CERSe agent allows users to select a region of interest (Figure 2.15) over a map. Another agent (Figure 2.16) then interfaces with a relational database to let the user constrain the data by a number of parameters, such as which sensor, year, month, day, etc.


latitude [max,min] 37.41 12.65

longitude [max,min] -95.94 -68.91

Figure 2.15: CERSe Latitude and Longitude Selection Agent


latitude [max,min] 36.79 8.75

longitude [max,min] -95.43 -70.85

Sensor: [✓ Any Sensor] AVHRR

Platform: Any Platform [✓ NOAA12] [✓ NOAA14] NOAA15

Year: [✓ Any Year] 1999 2000 2001

Month: Any Month, January, February, [✓ March] [✓ April] [✓ May] [✓ June] July, August, September, October, November, December

Weekday: [✓ Any Day] Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday

Day: [✓ Any Day] Range:

Hour: Any Hour [✓ Range:] 12,14,18-22

Minute: [✓ Any Minute] Range:

Figure 2.16: CERSe Database Agent

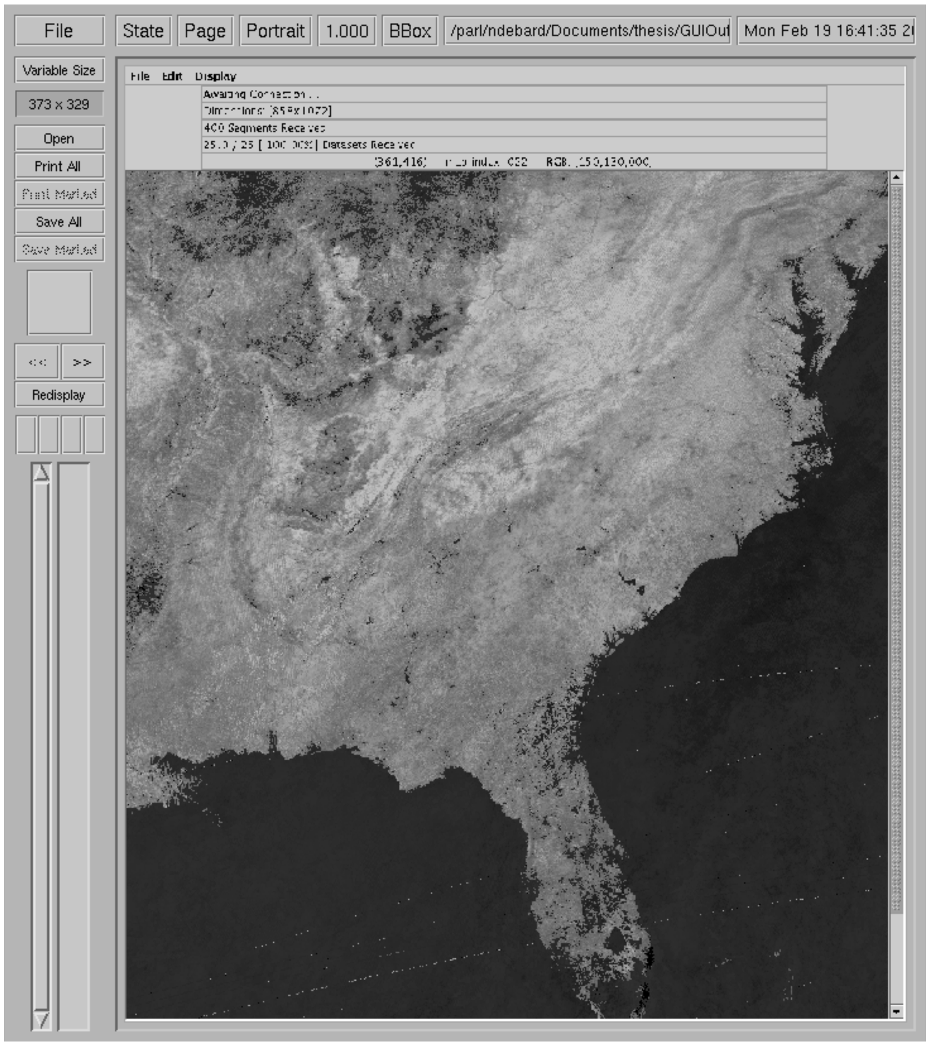
Satellite data files are rectangular, but due to the curvature of the earth, represent a non-rectangular region on the ground. Therefore, the images have to be georectified onto some chosen coordinate system. CERSe allows the user to select from half a dozen, and more could easily be implemented. When run in parallel, the data files are partitioned into segments by rows (also called *scanlines*). Each partition is represented by its own TPH. Processed sequentially, a TPH flows through the series of modules and the result is reassembled on the output. In many cases, the output is a composite image including many data files over the selected region and period of time. Each output file is composited into a single image using an algorithm, such as a pixel averaging. Figure 2.17 is a screenshot of a visualization agent that serves this purpose. The output interactively fills up this image and the image evolves over time as more and more data is composited into the final result.

CERSe originated as its own PSE and the limitations of that PSE spawned many of the concepts of Coven. Specifically, its inability to handle constructs such as loops, inter-process communication, and other similar functionality in Coven resulted in the creation of Coven. CERSe was then re-implemented in Coven to demonstrate that Coven truly was a superset of CERSe.

### 2.6.2 Medea

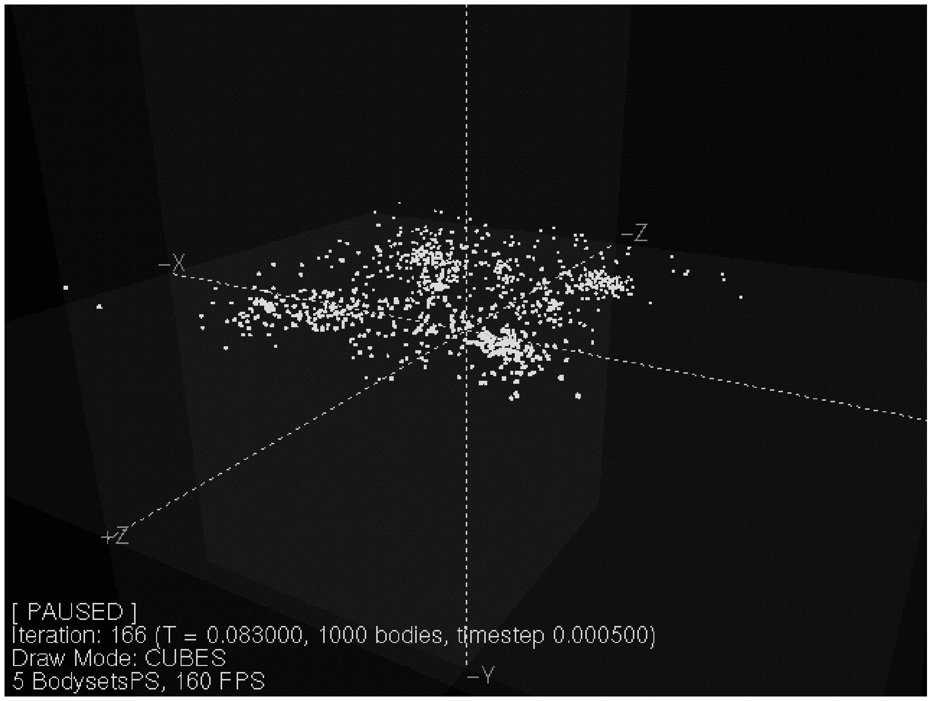
Coven was used to build an environment which targeted particle computations. The environment was named Medea and used to implement N-Body applications as well as one molecular dynamics program. Medea only has one custom agent which is used for visualization of the data. Figure 2.18 is a screenshot of that agent.

The agent is written in OpenGL, a high performance graphics library. This provides the user a full three dimensional view of the data, complete with the ability to move the view and zoom in or out to observe the data better. The visualization tool works in either realtime or post mortem mode. In realtime mode, a module is inserted into the application data flow graph which instructs data to be sent, in parallel, back to the visualization application



```
Awaiting Connection . .
Dimensions: [858x1072]
400 Segments Received
25.0 / 25 [ 100.00%] Datasets Received
(361,416)    map index 082    RGB. (150,130,000)
```

Figure 2.17: CERSe Output Visualization Agent



```
[ PAUSED ]
Iteration: 166 (T = 0.083000, 1000 bodies, timestep 0.000500)
Draw Mode: CUBES
5 BodysetsPS, 160 FPS
```

Figure 2.18: Medea OpenGL Visualization Agent

running on a specific port. In post mortem mode, the user inserts a module which writes the current state of the data to disk into a rolling replay file. This can then be used by the visualizer to replay the simulation graphically.

### 2.6.3 Hecate

The final PSE that Coven was used to create is named Hecate. This environment targets applications that are organized into an $n$-dimensional grid. It includes modules to help determine and communicate with neighbors, for applications such as was shown in Figure 2.10.

Hecate was used to create a complex fluid dynamics (CFD) application. This CFD program initializes a steel plate to a constant value, applies a steady new temperature to predefined locations on the plate, and then iterates forward until steady state has occurred. A visualization program written in OpenGL (Figure 2.19) helps the user visualize the computation. For the CFD application this data does not represent intermediate values (mean-

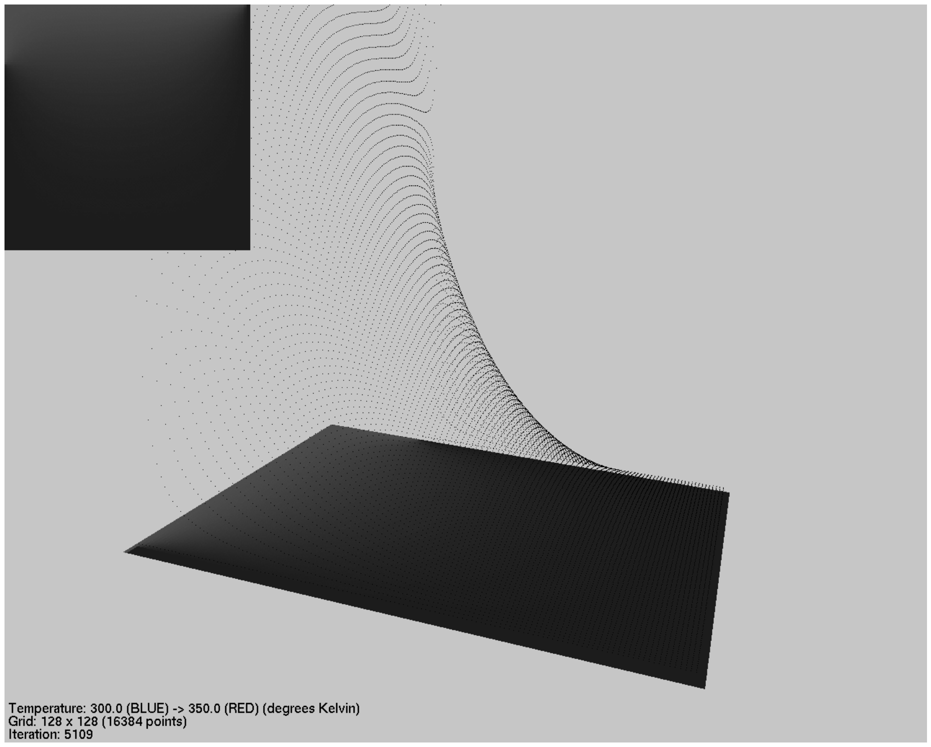

Temperature: 300.0 (BLUE) -> 350.0 (RED) (degrees Kelvin)
Grid: 128 x 128 (16384 points)
Iteration: 5109

Figure 2.19: Hecate OpenGL Visualization Agent

ing what the user sees does not correspond to how the steel plate would heat over time). Instead, it can be helpful in debugging and provides another way to visualize the data.

### 2.6.4 Additional Applications

Coven has been used to create a number of other applications. These include a fast Fourier transform (FFT), Cannon's matrix multiplication, a fractal generator application, and integer sort.

The FFT was implemented to demonstrate how simple it is to insert existing, legacy code into Coven. A popular FFT library, FFTW [40], was used and Coven modules were created which included only a few source code lines each with calls to the FFTW library. The application was trivial to build (since FFTW implemented all of the functionality) but it was possible to run it in Coven and get many of the benefits of the environment. These include multi-threading (as discussed in Chapter 4) and profiling tools.

Cannon's matrix multiplication is a relatively complex implementation of a common algorithm requiring a great deal of specialized exchange with different parallel tasks. This was easily translated into Coven modules.

One of the primary goals of Coven is to allow a wide range of applications to be implemented in it. With the wealth of environments which are specialized to one problem domain, we believe this goal to be important. The numerous, very different applications described in this chapter help to demonstrate the variability of the Coven PSE.

## 2.7 Summary

Coven is a large and complex problem solving environment composed of a graphical front end and a parallel runtime engine back end. Each piece is detailed with a number of their own components. The goal of Coven is to provide a toolkit where all phases of a parallel application's life cycle are addressed. These include development, debugging, performance analysis and tuning, as well as maintenance and reusability of code.

This chapter introduced Coven in as complete a form as was possible. Each aspect of Coven was aimed at either making the application easier to implement, perform better, or both. It is important to remember that Coven's target user is an application domain specialist with need for parallel computing performance.

Through the use of built-in Coven functionality, Coven helps to alleviate the need for problem domain specialists to be experts at parallel computing to achieve its performance gains. Additionally, when parallel computing expertise is required, the modular nature of Coven applications helps to separate the code which deals with the application from that which deals with the parallelism.

# CHAPTER 3
# RELATED WORK

During the 1990's the National Science Foundation held a series of workshops on Problem Solving Environments (PSEs). The goals of these workshops were to determine and address the needs of the PSE community. A definition of PSEs came out of the 1995 workshop and was given as:

> A problem solving environment (PSE) is a computer system that provides all the computational facilities necessary to solve efficiently a target class of problems. Moreover, PSEs use the language of the target class of problems, so users can solve them without specialized knowledge of the underlying computer hardware, software or algorithms. The facilities include advanced solution methods, automatic or semi-automatic selection of solution methods, and ways to easily incorporate novel solution methods. They also include facilities to automatically or semi-automatically select computing machines, to view or assess the correctness of solutions, to check the formulation of the problem posed, and to manage the overall computational process. Overall, PSEs are to be a framework that is all things to all people; they solve simple or complex problems, support rapid prototyping or detailed analysis, and can be used in introductory education or at the frontiers of science. [72]

Gallopoulos et al. [41] propose that there are many areas of PSEs that are open research problems. These include the areas of PSE architecture, kernel, interface technology, and scientific interface. The architecture of a PSE deals with how components are organized and how the system allows for growth and evolution (extensibility). The kernel pertains to the internal structure of the PSE more than to the structure of the algorithms that a user develops using it. Ways in which components interact with the PSE through data structures and protocols are issues of interface technology. The scientific interface involves the way in which users should communicate with the PSE (ease of use).

In [74], the authors point out that PSEs are by nature domain specific but that there is much infrastructure that is common across problem domains. While PSEs have historically been built from the ground up to solve a particular class of problems, little work has been done in building a reusable framework which is common to a wide range of problem domain specific PSEs. Coven specifically aims to provide this type of common infrastructure, including optimizations common to many potential problem domains.

## 3.1 The Common Component Architecture

Component-based software development is an evolutionary step from object-oriented software development. Much like an object-oriented approach, component software defines an interface to a piece of code that is often treated as a black box. The component is meant to be pluggable into a framework to cooperate with other components to form a program. Another important concept of component programming is the ability to wrap existing applications into a component and allow them to easily cooperate with other components.

The software industry defined a number of component standards including CORBA, Microsoft's COM, and JavaBeans. While some high performance computing researchers have found these standards acceptable in the HPC field, these standards were not built with HPC in mind and are often not completely appropriate. Some environment developers have extended these standards to specialize them to the HPC field where appropriate. The Common Component Architecture (CCA) Forum [1] was created by collaborators at many U.S. Department of Energy laboratories and universities to specifically address the need for a component standard for software development in the field of high performance computing.

The CCA was motivated by the need of researchers at physically disparate institutions to collaborate on software development. Often, these different researchers had legacy applications written in different languages and meant to run on certain HPC architectures. The CCA was brought about as a means to allow these applications to run as components and easily interoperate with other components without the need to worry about language, operating system, or architecture differences.

Each CCA component exposes its inputs and outputs using the *scientific interface definition language (SIDL)* [8]. The CCA defines a collection of programming interfaces which must be present in an application framework for the framework to be considered CCA compliant. These interfaces make it possible for CCA components to register themselves, find other components, and exchange data with other components. The CCA requires that *ports* are provided that marshall and unmarshall (demarshall) data going into and out of CCA components. This is used to translate data into a form that will be usable by the component and allow it to interact with other components written in different languages.

In [61] a hydrodynamics simulation toolkit is presented that employs the CCA for the component model. Utilizing existing hydrodynamics code in C and Fortran77, legacy applications were wrapped into CCA components. The result is a set of components which can be configured to create several different Structured Adaptive Mesh Refinement (SAMR) hydrodynamics applications. The authors found the CCA to be easy to implement and implement components with. Additionally, they found the overhead imposed by the CCA approach was negligible.

DCA [10] is a framework for distributed CCA component computing based on MPI. While the CCA has evolved extensively with respect to *direct connect* components, distributed components still present many challenges. Direct connect components are those running on a parallel machine that can be directly connected with each other and are similar to Coven modules. Distributed components, on the other hand, allow encapsulation of entire parallel programs in some cases which exchange data with other distributed components. The primary, well documented problem with this is referred to as the "$M \times N$

problem”. This problem deals with a parallel program running on $M$ processors communicating with another running on $N$ processors. This is increasingly more common in the Grid environment (discussed in Section 3.5). The DCA prototype framework proposes a solution to the $M \times N$ problem that relies on MPI communicator groups.

The CCA addresses the needs of the HPC community to merge applications from different languages and computing platforms. While it can certainly be used to develop homogeneous interoperating components, it is not clear how much overhead the approach will impose on application performance. In many cases, however, without the CCA many applications would have little hope of being componentized and used in a single application. Its importance is clear, as the HPC community has resoundingly adopted the standard. For a system like Coven, however, it may not be an approach that provides the kind of performance we seek. However, Coven has no support for interoperability between languages and disparate computing systems so there certainly are some things that Coven could gain by adopting the CCA.

At the time that Coven, and its predecessor CERSe, were being designed the CCA was not well accepted. While it was growing in acceptance, the standard was very much in fluctuation and we chose to use our own module standard rather than adopt the CCA. This approach was very common at the time. The rest of this chapter outlines related PSEs and frameworks, some of which use the CCA and some of which do not. At this time, the CCA has grown to be widely accepted and many frameworks, PSEs, and toolkits are emerging that adopt this standard. In many cases, older PSEs are being modified into new versions that use the CCA as well. Along these lines, a future advancement of Coven should be to look into using the CCA for Coven modules.

## 3.2 Domain Specific PSEs and Toolkits

The Extensible Computational Chemistry Environment (Ecce) [11, 74] is a very detailed and successful PSE for solving advanced computational chemistry problems. The Ecce

PSE provides many tools common to chemistry applications which are integrated through an *event-based component architecture*. This architecture uses a publish/subscribe system where components can subscribe (or listen) to certain data and as the data changes (is published) the subscribers are notified and can adjust accordingly. Ecce is a PSE for parallel computers and employs ParSoft [9], which is a suite of parallel computing libraries and tools. After a decade of use, Ecce has been improved to support Grid services including job management and monitoring.

Cactus [6, 5] is a problem solving environment initially designed to solve astrophysics problems. In Cactus, users develop modular code called *thorns*. The thorns are plugged into the Cactus *flesh*, which is essentially a driver for the application. Cactus runs on a wide variety of supercomputers including their operating systems, languages, and compilers. Cactus has support for interactive computational steering and monitoring as well as data visualization. Several parallel solvers and support for a few parallel I/O subsystems mean that Cactus relieves the user from being burdened with implementing these features. Furthermore, Cactus runs on the Grid and operates with the Globus Toolkit [37]. While initially aimed at solving applications whose data could be represented as an N-dimensional mesh, Cactus appears to have evolved into a more generic PSE. Still, its primary user base is in the astrophysics community and, within that community, Cactus is extremely successful.

The field of satellite remote sensing has always been a candidate for parallel computing due to the large number, size, and arrival rate of satellite data files. CERSe [29, 30, 32] (a project that Coven evolved from) focused on this field and provided a problem solving environment for implementing parallel remote sensing applications. A similar environment is the Open Source Software Image Map (or OSSIM) [69]. OSSIM is developed by the company ImageLinks and focuses on leveraging existing open source tools to provide a PSE for remote sensing, image processing, and Geographical Information Sciences (GIS) analysis. Like many other PSEs, OSSIM wraps existing applications into a form that allows interoperability between each other. In OSSIM, datasets are partitioned into *tiles* and passed down an *image chain* which executes the algorithm. The parallelism is entirely focused on data parallelism through partitioning of satellite data sets. As such, the usability of OSSIM outside of the image processing realm is questionable. Furthermore, while the goal of OSSIM is to run on parallel computers, parallelism was forced onto the PSE after it was already built. No concrete performance numbers exist yet as the project is still in development.

SCIRun [54] is a popular PSE toolkit that has been used to build two main environments: BioPSE and CSAFE. BioPSE is an environment aimed at biological systems such as studies of the brain, heart, and other similar applications. CSAFE is an environment aimed at fire detection using satellite imagery. Both BioPSE and CSAFE have a rich set of professional visualization tools which are truly impressive. Users of the environments implement code in SCIRun utilizing a shared memory paradigm. Modules are created and interconnected to form a program *network*. This application can be composed of computational and visualization components which interact to solve a problem. SCIRun (and its PSEs) also allows for computational steering and runs on a wide range of desktops and supercomputers.

Uintah [24] is a PSE from the same group who developed SCIRun [54] and BioPSE. While SCIRun targeted shared memory computers, Uintah addresses the need for a PSE in the distributed computing realm. Users of Uintah implement applications as CCA components and execute them within the Uintah environment. Uintah provides the user with performance and trace analysis tools. TAU [75] (a popular performance monitoring system) and XPARE [23] provide the basis for Uintah's performance analysis suite. With XPARE, users can track performance of an application through its development cycle. It can then be used to identify negative shifts in the performance of the application due to changes in the code.

SCIRun and Uintah both use their own component model. In an effort to conform with a growing trend in the community, the developers of SCIRun are creating the next version of SCIRun, SCIRun2 [91], which will employ the CCA component model. SCIRun2 is currently in the early stages of development, but the creators aim to not only support CCA components but also SCIRun (version 1), CORBA, and Microsoft COM components. Using a bridge system, SCIRun2 will allow these components to interoperate and be used in the same application. Support for distributed computing as well as parallel computing will be available. With distributed computing components, components can execute on separate machines but communicate together to solve an application in parallel. In contrast, parallel components will be the same components executing in parallel but operating on a different portion of the data. Since SCIRun2 is still in development, performance of the system can only be speculative. The developers believe that the PSE will provide competitive performance but the bridge technology may prove to be sufficiently complex and hinder application performance.

The Earth System Modeling Framework (ESMF) [49] is a collaborative project to build a complex PSE for implementing parallel earth science applications. ESMF collaborators are located at NASA, NOAA, the DOE, and many universities. It was seen as necessary as the share of algorithms and data between different institutions has historically been very difficult. Researchers have often developed large and complex code using tightly coupled software systems. ESMF attempts to provide a standards-based open-source solution where users develop components that can interact and are easily exchangeable. Relying on Fortran, C++, and MPI, ESMF provides a suite of classes which developers can employ to assist in parallelism. These include classes to grid data, distribute, reassemble, and communicate with other parallel tasks. As there are many important earth science codes in existence, ESMF encourages developers to wrap them into ESMF components. Additionally, both SPMD (like Coven) and MPMD applications are possible in ESMF. MPMD is particularly useful in task parallelism and helps utilization of existing earth science software. ESMF specifically targets earth science applications and has many helper classes which make implementation of this type of application easier.

## 3.3 General Purpose Frameworks and Programming Paradigms

Perhaps the closest framework/PSE to Coven is the Asynchronous Buffered Computation Design and Engineering Framework Generator (ABCDEFG) [21] or FG for short. FG is a framework from Dartmouth university that makes users program in C or C++ functions called *stages*. Stages are organized into a *pipeline* to create a program. Each pipeline is automatically placed into its own thread and FG handles all issues of thread maintenance (such as creation, destruction, atomic access to shared variables, etc). Additionally, FG manages the application's buffers as they pass through the pipeline and stores information about each buffer in a *thumbnail* record. This is similar to Coven's method of maintaining buffers inside of TPHs with detailed information about each buffer. Furthermore, the programming paradigm is very similar between the two projects. FG has built-in functionality to assist in looping (much like Coven does) by tagging a buffer as the *caboose*, or final buffer to go through a pipeline stage. The authors use the ChaMPIon/Pro MPI implementation, a commercial package that allows multiple MPI threads to make MPI calls. While FG uses true pthreads because of the availability of the ChaMPIon/Pro MPI implementation, Coven simulates threads with Coven Threads (discussed in Chapter 4). FG definitely has some similarities to Coven, particularly in the way programs are designed from a user's perspective. Coven, however, has additional features which do not appear to be present in FG. These include task virtualization, a graphical user interface, graphical profiling and trace debugger, dynamic load balancing, automatic program checkpointing, and others.

ParoC++ [71] is an object-oriented parallel programming environment with focus on data intensive computing. ParoC++ users implement their applications using a series of C++ classes. Through a mechanism called *passive data access*, ParoC++ tries to improve performance by overlapping computation and data retrieval (communication). Data often is distributed, and retrieving it can be costly and time consuming. ParoC++ has pre-built classes and methods for specifying a portion of the data that is required and the data is automatically retrieved from the distributed source. During this retrieval time, the application can continue execution. Only once the application can go no further without having the data does it potentially block awaiting data arrival. ParoC++ is mostly a collection of helper C++ classes which make programming data intensive applications simpler in a distributed environment. Coven has relatively few similarities to ParoC++ including being implemented in C, rather than C++, and having no specific optimizations for data intensive applications.

A similar framework is Dynamic Parallel Schedules (DPS) [42]. DPS provides a set of C++ classes that are extended, overloaded, or modified to perform parallel computation and communication. Users create applications by programming separate C++ classes that are combined into a directed acyclic graph (DAG). Similar to Coven, DPS has *split* and *merge* constructs. In DPS, the user can extend a split or merge class to create a data distribution pattern. DPS executes the code in separate threads, to attempt to achieve benefits from overlapping computation and communication. While some aspects of DPS are similar to Coven, these mainly are with respect to the programming paradigm. In particular, the projects share the concept of a component-based programming model where data is distributed in an SPMD fashion and computed on in parallel.

## 3.4 Miscellaneous

Condor [63, 84] is a project dating back into the early 1980's from the University of Wisconsin-Madison. It enables gathering collections of computers into *processor pools* and then utilizing these pools to execute parallel codes. Initially designed to efficiently use idle workstations at the university, Condor has evolved through Condor-G into a toolkit for the Computational Grid (see Section 3.5). As processors become idle, they add themselves to the processor pool. Condor has support for process migration so that executing applications may be halted and moved to other processors in the process pool. This is particularly common when a previously idle processor becomes unidle (for instance when a user sits down at his workstation in the morning).

In Condor, a *problem solver* provides a programming model, such as *master-worker* and *directed acyclic graph manager*. The problem solvers rely on the Condor agent to handle issues with respect to error handling, migration, and job execution and instead focus on the user's computation. In the master-worker problem solver, the master spawns work out to the workers that are provided to it by Condor. As workers appear and disappear, the problem solver must be able to adjust. The user is provided with a collection of C++ classes which must be extended to implement their specific functionality in the model.

Charm [56, 57] is a language and runtime system for parallel application development and execution. Charm is implemented in C and the code implemented by users must also be in C. Charm++ is an extension of Charm which supports C++. In Charm, a *chare* is a construct which declares a computation and the data of that computation. Charm schedules chares in parallel and uses a message driver system to execute chares once all input data is available. A chare may produce data as output, and this likely causes a message to be sent to another chare elsewhere, possibly causing that chare to execute.

Like Coven, support for load balancing and packing of data is provided by Charm. Also like Coven, Charm employs queues where messages (data) are queued up waiting processing. Charm supports quiescence detection so as to detect load and adjust the number of control messages in order to take advantage of idle tasks as well as not interfere during a heavily loaded computation. Many Charm applications have been developed including VLSI/CAD applications, a Prolog compiler, parallel molecular dynamic program, parallel cell placement, and others. In [57] it is shown that on a single processor Charm imposes an overhead of less than 5% when compared to a sequential C implementation. Charm executes on a large number of parallel machines and has many features which simplify parallel application development. In many ways its goals and even some of the implementation approaches are similar to Coven.

Triana [83] is a programming environment for Java applications. Users program *units* which are modular pieces of Java code, conforming to a specific interface and utilizing a set of parent classes. Triana units are then interconnected to form a program. Each unit runs in its own Java thread and this is scheduled on a parallel node. Triana employs task parallelism, rather than executing the same unit on multiple processors where each processor has a segment of the data (data parallelism). The focus with Triana is on Grid services including spawning Triana applications on the Grid, steering computation remotely, and distributed visualization. Triana has been used to develop galaxy formation simulations, though it could likely be used in other similar application domains.

## 3.5 Grid Computing

Grid Computing is rapidly emerging as a powerful force in the parallel computing realm. The concept of Grid Computing is that distributed computing resources can be interconnected into large supercomputers. These are then available for use by other researchers. This has two main advantages: it allows researchers with smaller parallel computers to get access to larger parallel computers, and potentially it can result in better utilization of resources. Oftentimes a parallel computer might go unused for some period if it were not on the Grid. As part of the Grid, however, during this low utilization period the machine might be utilized by researchers around the globe. The whole Grid concept is built upon the idea of sharing.

As one might imagine, Grid Computing has introduced a massive number of challenges that must be addressed. These include running parallel jobs on machines whose nodes are distributed around the world, differences in operating systems, transportation of data, visualization, steering of computation, and many more. The Global Grid Forum (GGF) was created to attempt to steer the study of these problems and help encourage solutions and collaborative research. One of the most recent advancements is the Open Grid Services Architecture (OGSA) [38, 82]. The OGSA attempts to standardize some set of the Grid services so that researchers can begin building products that can interoperate.

Legion [44, 43, 51] and its predecessor systems have since the beginning focused on Grid applications and services. Legion attempts to provide a toolkit, programming framework, and runtime environment for Grid applications. Support for PVM, MPI, C, Fortran, C++, Java, and CORBA help the widespread use of Legion. By creating a virtual machine for Legion applications to run on, Legion helps to abstract away the underlying computing system. This is particularly important when the parallel nodes being utilized by an application may include several different operating systems and hardware architectures. Clearly, the Legion back end must be ported to many systems but the designers seem to have done this and there is a great deal of community support for the project. Legion evolved into Legion-G which was essentially a port of Legion to run on top of the Globus Toolkit [37]. Globus included a number of early Grid services, such as GridFTP and security protocols. With the emergence of OGSA, the Legion team has begun focusing on OGSI.NET, an attempt to utilize the .NET Framework (from Microsoft) in the Grid context.

The Padico [33] software infrastructure leverages existing technology to help solve many complex problems in the realm of Grid Computing. Wrapping legacy code, language interoperability, code maintainability, and security are all complex problems on the Grid. Padico employs the CORBA Component Model [45] to extend the distributed component programming concept of CCM and provide facilities for improved performance in the HPC realm. CCM has rich support for security, versioning (addressing maintainability), and interoperability across many operating systems, architectures, and languages. In CCM, components have facets (output ports) and receptacles (input ports) that specify what data is produced and required by the component, respectively. GridCCM extends the CCM further, adding concepts of parallelism and parallel components. CORBA is a very successful and well understood standard for creating software. As such, Padico benefits from this and is able to leverage CORBA's successes and bring it into the Grid Computing field. Support for wrapping legacy applications into CCM components is an important aspect of Padico and allows legacy parallel applications to be easily "ported" to Padico. Padico and GridCCM is most similar to the CCA but, unlike the CCA, GridCCM has support for parallel components. However, while the CCA is widely in use, at this time GridCCM is still a work in progress and so far GridCCM's performance is not promising.

The CCA Toolkit (CCAT) [15] is an implementation of the CCA specification designed to operate with Globus and other Grid frameworks. CCAT includes several means to design applications. In CCAT, the application design facilities are called *builders*. A graphical user interface, Python interface, Matlab interface, and a web-based interface provide many means for building applications in CCAT. Much like Coven's GUI, the CCAT graphical builder provides a familiar drag-and-drop interface for building applications by interconnecting components.

The Component Component Architecture (CCA) was not initially designed to be compliant with Grid standards. The OGSA standard does not directly propose a component standard for the Grid. XCAT3 [59] is a framework that merges the two standards, allowing CCA components to be accessible by portal-based Grid clients. This project was motivated by the successes and wide acceptance of both the CCA standard and the Grid computing movement. XCAT3 is implemented in Java and uses the Babel tool to generate Java interfaces for CCA components from the SIDL component specification. Then, XCAT3 uses the GSX toolkit to present the component ports and the component itself as Grid services. These two toolkits help XCAT3 merge the two standards.

## 3.6 Summary

The field of Problem Solving Environment research is very active, particularly with respect to high performance (parallel) computing. Users of these complex systems are seeking ways to incorporate software practices from the sequential world. These include software reuse, modularity, extensibility, and interoperability. Many of these are being addressed through the use of component-based (modular) software development. The HPC field, however, by its very nature is driven by the need for performance, something that often makes conventional sequential software practices and techniques not applicable in the parallel field.

Standards like the CCA have helped to unite the field in many respects and provide a common ground to develop PSEs. Similarly, the interest in Grid Computing has produced many standards and toolkits which PSE developers and designers are incorporating in next generation environments.

Much of the PSE research is focused on a particular problem domain. This usually comes about from assumptions and simplifications in the programming model which simplify application construction within that domain. A consequence often is that this approach limits widespread usability in different scientific computing fields.

In the following chapters we explore several common parallel computing optimizations and features that have been incorporated within Coven. We discuss their design, implementation, and study the impact on applications implemented in Coven. A study of related works in the field of these optimizations and features is left for the appropriate chapters that follow.

# CHAPTER 4

# MULTI-THREADING

Software, and parallel computing applications in particular, frequently have periods of execution where data must be accessed outside of the CPU. Disk, memory, and network access are all common examples of such operations. With processor speeds increasing, the devices which feed the CPU are struggling to keep up.

In parallel computing, network latency and bandwidth have long been major issues in parallel program performance. Similarly, data intensive applications, even those that use a parallel file system (such as PVFS [18, 60] or GPFS [55]) more often than not cause the processor to stall while waiting for data to arrive.

With parallel computing becoming more widely used in government, education, and business, there is an ever-increasing need to obtain better performance from parallel codes. Organizations often rent supercomputer access time and those who own them find that, due to the expense of the machine, they are sharing time and resources with other individuals who seek to achieve the massive program performance that these machines can provide.

Therefore, providing processors data at as fast a rate as it can consume is the goal and any progress in this direction is important. While much effort has gone into the process of speeding up disks and networks, there are other approaches. A common approach is to hide latency through the use of a multitasking operating system and schedule concurrently running processes. The idea being that while one process is waiting on data to arrive from (for example) some remote processor's memory, the other process is operating on data that has previously arrived. This is generally termed *multi-threading* or *asynchronous programming*.

Both approaches have strengths and weaknesses. While using fast networks and disks is an extremely simple solution (requiring often little to no program augmentation), it is extremely costly and still might not eliminate the offending latency. Multi-threading, on the other hand, is complex to implement well in application code but requires no special hardware or additional monetary costs. Multi-threading is also an effective mechanism to hide latency due to remote data access and interprocess communication by switching quickly between ready threads. Certainly there is a tradeoff between simplicity (high monetary cost) and complexity (low monetary cost).

Multi-threading is common and has great potential to be usable by a wide range of applications. As such, the optimization is a good candidate for incorporating within a PSE so that the PSE's applications can utilize the optimization. This chapter studies the addition of the multi-threading optimization within Coven and how the Coven model of computation both facilitates the easy implementation as well as results in a high performance optimization.

Since the Coven model separates the application code from the application state, it makes it simple to schedule tasks to execute in parallel. TPHs are moved around to combine with modules to form a task, and this task can execute anywhere in the system at any time. In the same manner, the Coven multi-threading implementation utilizes multiple threads of control concurrently executing on the same parallel node. These threads can be assigned any module and as TPHs arrive they form tasks which execute concurrently with other threads.

In Section 4.1.1 we study an optimization that transparently uses shared memory for state when necessary. This occurs when a TPH will be utilized by multiple threads sharing a central memory. The runtime engine is able to make this optimization by utilizing the Coven model of computation to determine how state will be used during the life cycle of an application. By analyzing where a TPH is destined, the runtime engine can efficiently utilize shared memory or local memory as appropriate.

Two case studies are performed to analyze the performance of the multi-threading optimization. The first experiment is detailed in Section 4.3 and is shown to provide an improvement ranging from about 25% to 28% on the system tested. This experiment is synthetic and designed to test the range of the improvement offered by multi-threading. The second experiment is presented in Section 4.4 and studies multi-threading the Fast Fourier Transform (FFT) utilizing a popular FFT library. The FFT is much less well balanced than the first experiment but performance improvement of as much as 18% was seen on the system tested. Each experiment includes a complexity analysis of a hand-coded implementation of the multi-threading optimization.

By utilizing the Coven model of computation, the multi-threading optimization is provided to Coven users with no augmentation to existing application code. This means that multi-threading can be experimented with in an application and studied to see if it will be beneficial without the cumbersome task of implementing it in each application. Additionally, we present a Coven expert system which analyzes trace performance data to suggest multi-threading as well as suggest which modules to schedule for concurrent execution. This expert system processes trace data which was automatically collected by Coven at runtime and includes state, module, and task information.

## 4.1 Multi-Threading Implementation

While Coven can work with any implementation of MPI, the environment under which this research was conducted used the implementation MPI Chameleon, or MPICH [67]. This version is free, open source, and relatively complete, conforming to most of the MPI2 specifications. One particular area which MPICH is not complete is with regard to threading. Threading in MPICH is allowed only if only one thread associated with a process issues any MPI calls. This is a very limited version of threading and was not acceptable for multi-threading in Coven.

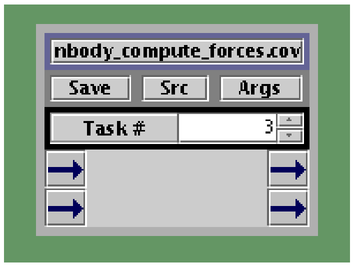


Figure 4.1: Choosing a Thread Number With GUI

To address this problem, Coven uses full weight MPI tasks and schedules them on the same processor rather than use conventional threads. This produces concurrently running processes on the same node,however the processes cannot access shared memory data. This concept of accessing shared data is outside the scope of Coven's programming model, so does not present an issue.

In Coven, this style of "threading" creates processes which we term *Coven Threads*. Each Coven Thread runs a program sequencer and looks much like a single threaded version of Coven. In fact, with the introduction of Coven Threads, the only difference between single and multi-threaded Coven programs is the number of Coven Threads. All modules are assigned a Coven Thread under which they will run. This is done using the graphical user interface (Figure 4.1) or the Coven program language (Figure 4.2). Figure 4.1 shows a graphical representation of a module named `nbody_compute_forces.cov`. This module has two inputs and two outputs (represented by arrows). The white box contains an integer which is the Coven Thread number under which this module will run.

When the runtime engine starts up, multiple MPI tasks are scheduled to run concurrently on the number of nodes the user specifies. Each of these tasks is a Coven Thread. Each Coven Thread looks at the program language specification and dynamically loads the modules which are scheduled to run under that thread. Figure 4.3 shows a sample threading scheme. In this figure, there are five slave processors being used to solve a problem in parallel. Each of the slave processors has three Coven Threads (depicted as different

```
. . .
   thread_A fftw_scatter(args)
   PARALLEL {
      thread_B fftw_comp_nd(args)
      thread_C fftw_transpose(args)
      thread_B fftw_comp(args)
      thread_C fftw_transpose(args)
   }
   thread_A fftw_gather(args)
. . .
```

Figure 4.2: Sample Coven Program Language Threading Scheme

colored circles) executing on it. On a processor, TPHs travel between Coven Threads as each thread is assigned separate modules to run.

Threads can communicate with other parallel instances of themselves. For example, in Figure 4.3, the slave Coven Thread 1 on processor 1 can communicate using MPI with the same thread running on any of the other processors. Coven's runtime engine creates special MPI Communicators to assist in this task. Since the Coven Threads in Figure 4.3 are actually full weight MPI tasks, the runtime engine initializes as an MPI program with seventeen parallel tasks. It then schedules the tasks in the order represented in the figure to create a thread-like environment.

The runtime engine handles transparently transferring data between Coven Threads. The user has defined a dataflow graph constructed of interconnected modules and these can be assigned in any threading scheme. The runtime engine handles moving data between Coven Threads on the same processor and other processors seamlessly. Since all data is encapsulated in TPHs, moving data is simple and occurs using the TPH queuing system.

During the course of testing multi-threading using MPICH1, it was found that MPICH1 performs poorly when multiple MPI tasks are scheduled to run concurrently. As mentioned in Section 2.4.3, Coven uses two TPH queues where it waits on incoming TPHs and outgoing TPHs. This is implemented by using a dynamically sized array of

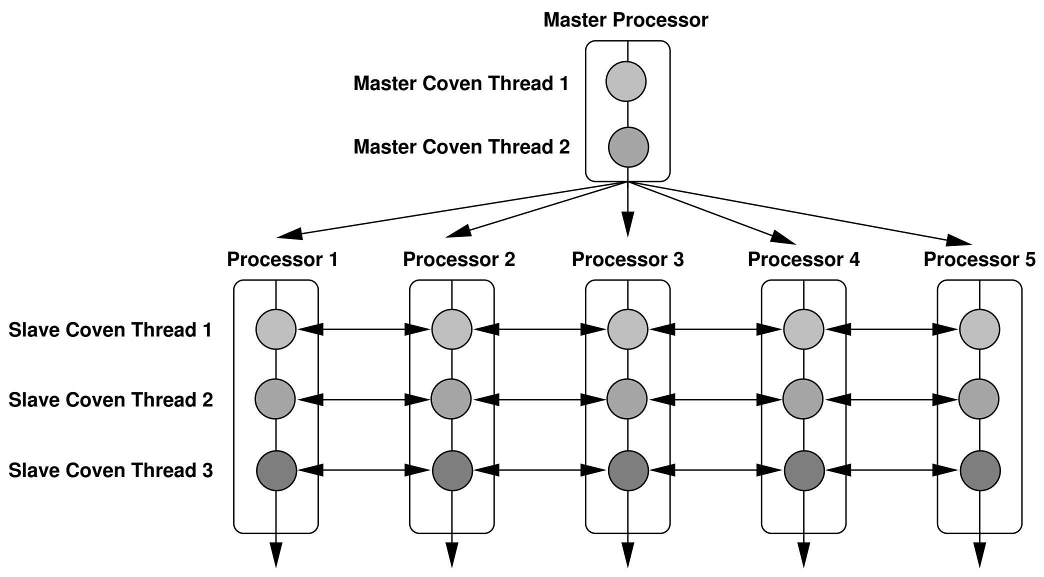


Figure 4.3: Example Coven Thread Organization

MPI_Request structs, which in this case are handles to pending asynchronous MPI transactions. Each Coven Thread then "sleeps" while waiting for any of these to complete using the call MPI_Waitany. While we might expect this operation to release the CPU while waiting for a transaction to complete, under MPICH1 this is not the case. Instead, the call repeatedly checks each element of the array for completion, amounting to a busy wait. The problem is that if multiple MPI tasks are scheduled on the same node, then each task is competing for the CPU. Ideally, a task waiting on communication to complete should release the CPU and idle waiting for notification of a completed message. This competition for the CPU keeps a Coven Thread with work to do from getting full access to the processor.

This is a well known problem with MPICH1 and is addressed by using MPICH2 [68]. At the time these experiments were conducted, MPICH2 was in beta testing and not complete. In particular, it lacked the shared memory extensions which improve performance of intercommunication MPI tasks running on the same processor.

### 4.1.1 Shared Memory

TPHs are transferred between Coven Threads using MPI. The runtime engine automatically takes the complex TPH data structure and copies the data contained in its many pointers and buffers into a sequential stream of bytes. This is termed marshalling in Coven. Once marshalled, the TPH is transferred using MPI and is unmarshalled on the receiving end back into a TPH for processing.

Experimentation showed that transferring TPHs between Coven Threads on the same processor resulted in poor performance. While the TPH source and destination threads were on the same processor, MPI was used to transfer the data. For large TPHs (on the order of five or more megabytes), this was found to slow performance down by a noticeable amount.

Figure 4.4 is a screenshot of the Coven profiler demonstrating the problems with transferring TPHs between Coven Threads on the same processor. Time is represented on the x-axis while the y-axis shows two separate Coven Threads. This simple example shows a single processor running two Coven Threads. These are represented by the two lines, labeled *Slave 1 / Thread 1* and *Slave 1 / Thread 2*. T For this application, a single TPH is passed back and forth between Thread 1 and Thread 2. The performance problem lies in the fact that after a module has completed (the end of a rectangle in the figure), the next module does not start for a large block of time. In this example, that transfer time is roughly seven seconds for a TPH which encapsulated 256MB of user data.

As MPICH2's shared memory functionality was not complete, an extension to Coven was added which provides shared memory between MPI tasks on the same processor. Through sharing memory, concurrently executing Coven Threads can exchange TPHs by simply exchanging a handle to where the TPH resides in the memory which is accessible by both parties.

Users declare a memory size that Coven can use for a shared memory region. Through the `shmget`, `shmat`, `shmctl` and `shmdt` calls, Coven creates and manages

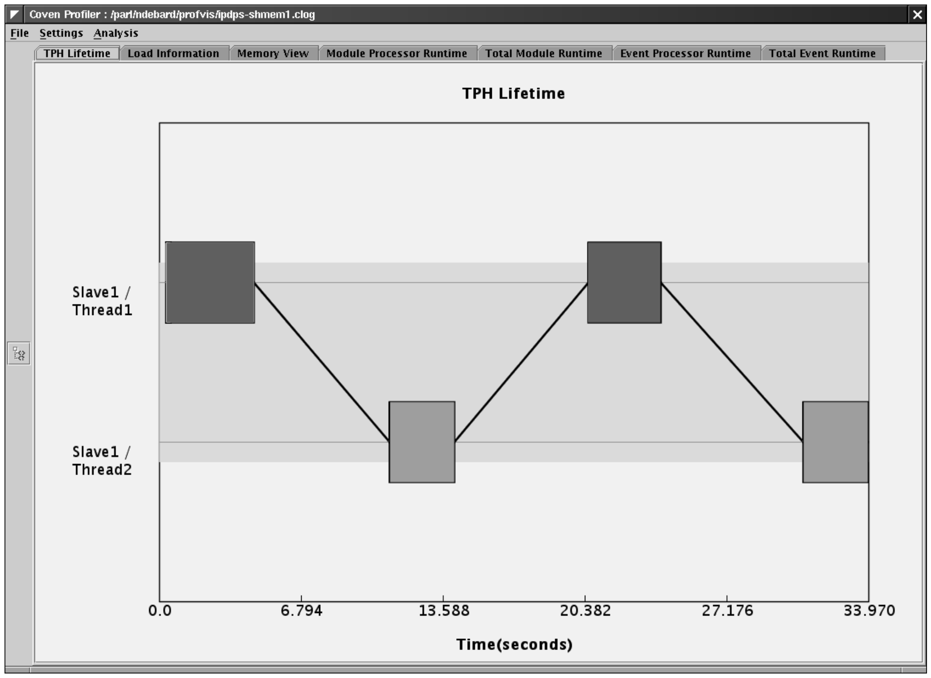


Figure 4.4: Without Shared Memory

this memory region. Access to the region is controlled by a semaphore, shared by all Coven Threads on the same processor. All interaction with shared memory is transparent to the user.

As modules create new data in a TPH, the runtime engine determines where the TPH will be destined for after it leaves the current Coven Thread. This can be determined by examining the dataflow graph. The runtime engine uses this information to decide what type of memory will be used for storing the data contained within a TPH: shared memory or local memory.

Coven employs a memory management system which essentially provides a malloc-style interface to manipulating the shared memory region. While this is used only by driver code, it greatly eases programming as the shared memory calls mentioned earlier merely allocate one large piece of memory. Coven imposes on top of this a memory management scheme that finds the first fit for a new memory allocation. This system is changeable and conforms to a simple interface. Therefore, enhancements to this scheme can be implemented and plugged into Coven with ease.

From a user's perspective it is essentially transparent where Coven is storing TPHs. While the user can set values for the requested maximum size of the shared memory region, Coven maintains a default which allocates a portion of the total memory. The point of using shared memory is that transporting TPHs between Coven Threads on the same processor is costly if they are large. This operation involves several copies from one Coven Thread's memory into the destination Coven Thread. The runtime engine automatically places buffers into local memory that are considered too small. There is certainly a tradeoff when using the shared memory system as it requires a synchronization with the other Coven Threads on a processor. This is only beneficial in the case where the memory contained in it is considered large. This size, of course, varies from system to system and Coven allows a user to specify it or use the default if he is unsure.

With the shared memory extensions built into Coven, the same application depicted in Figure 4.4 was re-run. Figure 4.5 is the profiler screenshot of the application using the built-in shared memory extensions. The time to transfer a TPH between Coven Threads running on the same processor is reduced to a barely measurable amount. In this simple example, it causes the program runtime to decrease from 34 seconds to 14 seconds.

This shared memory extension to Coven allows the runtime engine to transfer TPHs between Coven Threads on the same processor rapidly. Without this ability, performance improvements from multi-threading due to overlapping computation and communication would be overshadowed by merely transferring data between threads.

### 4.1.2 Expert System

The Coven profiler has a built-in expert system that offers suggestions to the end user in order to decrease the parallel program runtime. The expert system is a rule engine based on JESS (Java Expert System Shell) [39]. After analyzing runtime trace information, the

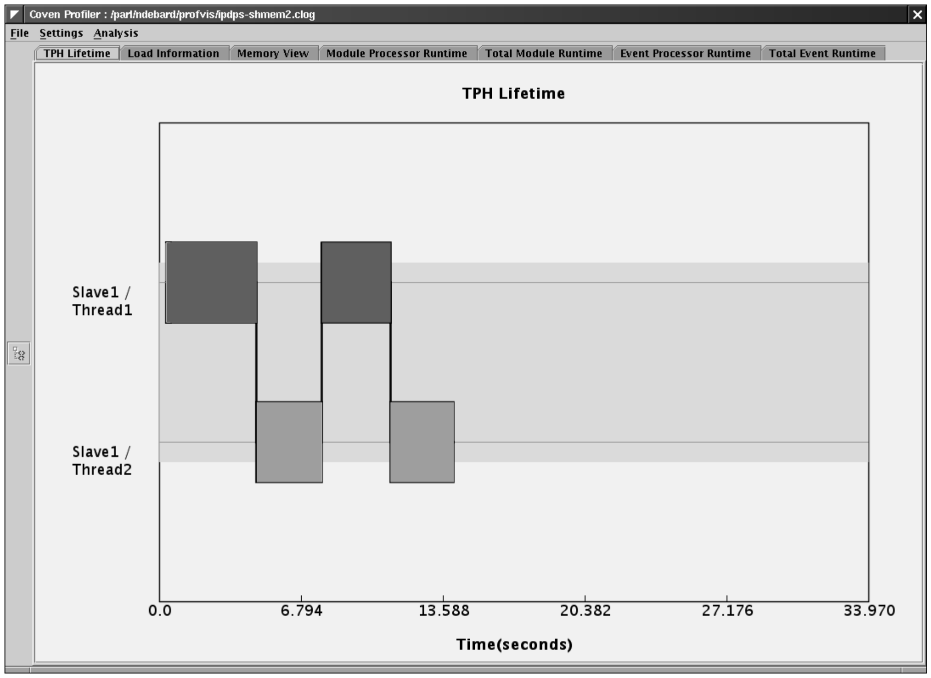


Figure 4.5: With Shared Memory

expert system is able to offer several possible performance improvements. Of interest to this chapter is its analysis of where multi-threading would be useful.

CPU load information is gathered with the executing Coven application. The expert system analyzes this information looking for modules which place a low load on the CPU and constitute a large enough portion of the total program runtime. These are then marked as candidates for appearing in another thread. We have empirically found on our system that a module loading the CPU less than 35% is a good candidate for multi-threading. Future work will entail determining this value for different systems automatically.

## 4.2 Working Environment

To analyze the performance effects of Coven's multi-threading capability two sample applications were implemented in Coven. These applications were run on a 16 node Beowulf cluster consisting of nodes with one 1GHz Pentium III processor, 1GB of RAM, and connected by Fast Ethernet. Each node of the cluster runs Debian Linux 3.0r2, Linux kernel

2.4.24, and MPICH2 0.93. All programs were compiled with GCC version 3.3.2 using maximum optimizations.

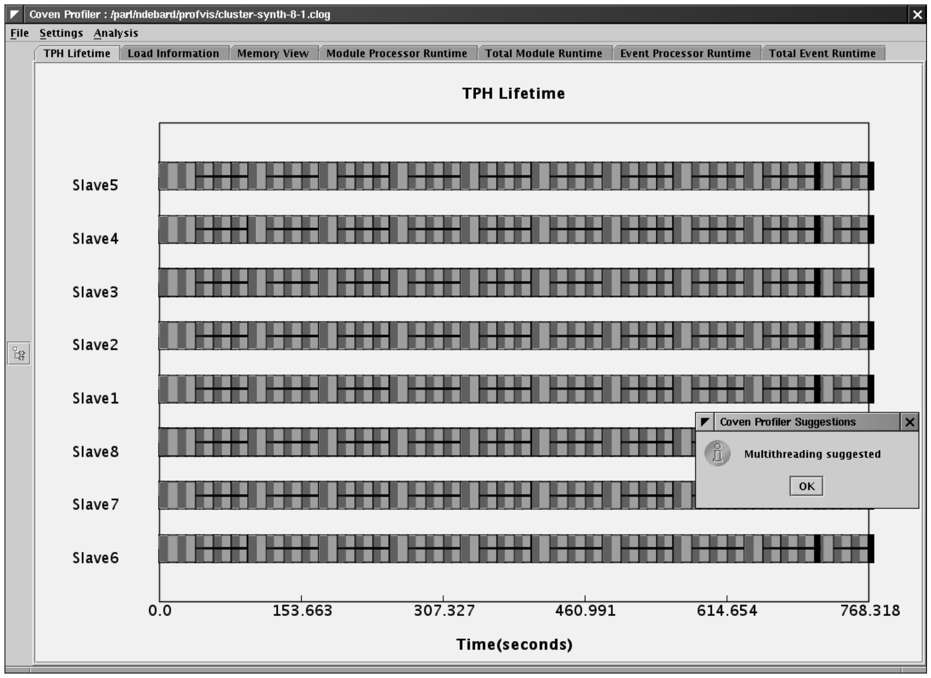
## 4.3 Synthetic Application Case Study

A synthetic application was constructed in an attempt to analyze several program variations and how they affect multi-threaded performance. This application has a computation phase followed by a network communication phase. Each phase is implemented as a module with many tunable parameters which can be altered without affect on the other modules. While this is not a realistic application it allows the study of several interesting effects. The program was designed so that multiple units of work were assigned to the same processor. In this way, when a thread completed work on one portion of work and handed it to another thread it could begin processing the next unit of work.

The CPU utilization of each module was analyzed using Coven's profiler. To determine the load each module placed on the CPU, Coven divides the total amount of time a module was scheduled on the CPU by the total amount of wall clock time that passed during the execution of the module. It was found that modules which load the CPU below about 35% were good candidates for placing into threads that run concurrently with other modules.

Additionally, the optimum performance was seen when a pipeline could be created where the modules which ran in the pipeline did so for approximately equal amounts of time. The synthetic application was tuned in this way and speedup was found to range from 25% to 28% as the number of processors was varied from two to sixteen.

Figure 4.6 shows a profiler screenshot of the single threaded version of this application running on eight processors. The screenshot represents the trace of the application's lifetime using a Gantt chart. On the x-axis time is represented and each parallel task is depicted on the y-axis. The alternating shaded blocks represent modules executing. Each block has a start time, end time, and a width which gives the total execution time for that

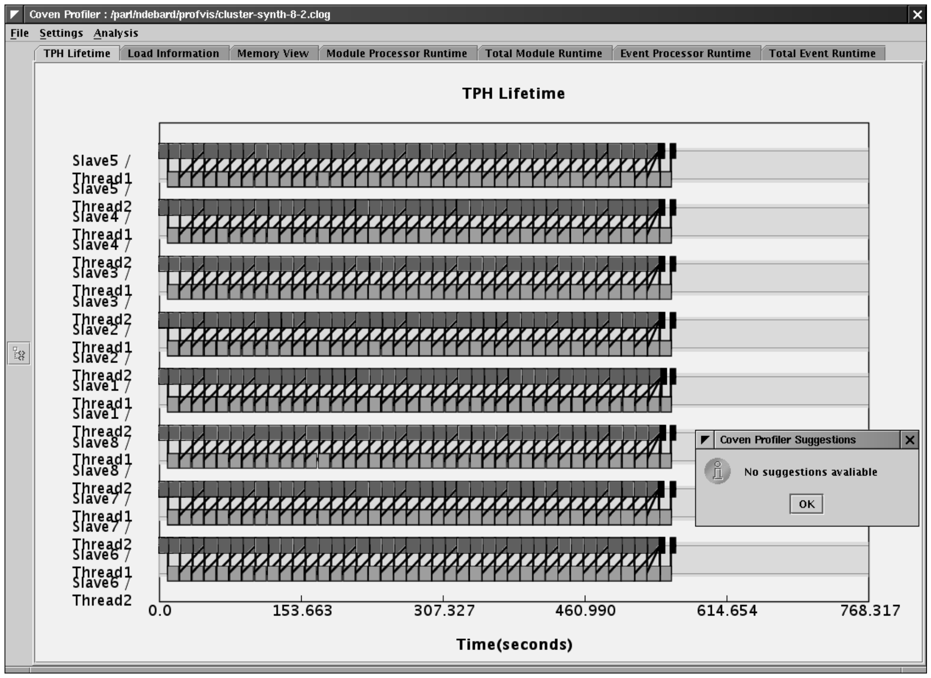


Multithreading suggested

Figure 4.6: Synthetic Application Single-threaded

module. In Figure 4.6, the darker shaded blocks are the computation module while the light shaded blocks are the communication module. The computation module alternates with the communication module and it can be seen that the blocks consume approximately the same amount of time. In Figure 4.6 the expert system is suggesting that "Multi-threading (is) suggested."

This application was then placed into multi-threaded mode by simply toggling a switch in the Coven GUI to place the communication module into a different thread from the computation module. A resulting profiler screenshot appears in Figure 4.7 and is scaled to be comparable with Figure 4.6. Since each of the eight parallel tasks are running in dual-threaded mode, the eight horizontal lines from the previous figure are now represented by 16 lines. In this case, the computation module (dark box) ran in one thread while the communication module (light box) ran in another thread. Data passed between the two threads in a regular pattern. The total runtime of the application dropped from 768 seconds to 554 seconds, an improvement of 28%.


Coven Profiler : /part/ndebard/profvis/cluster-synth-8-2.clog

Coven Profiler Suggestions

No suggestions avaliable

Figure 4.7: Synthetic Application Multi-threaded

---

The balance between execution time and CPU load of modules is important for determining under which threads modules should run and if the application will even benefit from multi-threading. For applications that exhibit extremely well balanced operations (such as depicted in this synthetic application), on this architecture Coven can provide a performance improvement ranging from approximately 25% to 28% with the use of multi-threading.

### 4.3.1 Non-Coven Comparison

A common concern with regard to PSEs is how much overhead the environment imposes on applications written within it. Due to the programming benefits present in PSEs, an overhead of as much as 15% is common in many PSEs. With Coven, we have found this overhead to be negligible — on the order of one to three percent for applications that are of sufficiently large grain size. This is true for most of the applications in this thesis. Chapter 6 outlines a fine grain application that does suffer more overhead due to Coven. To provide a performance and complexity comparison, we implemented the synthetic application without using Coven.

Multi-threading MPI programs present a great challenge. Part of this challenge lies in the normal challenges of threaded applications, such as the use of posix threads, condition variables, semaphores, and mutex locks. Additionally, most open source MPI libraries are not thread safe which makes utilizing the MPI library complex and cumbersome. For MPICH, thread safety can be maintained if at most only one thread ever makes an MPI call. The non-Coven multi-threaded synthetic application was implemented to allow for multiple threads of control where one thread was designated the *communication thread*.

This implementation requires and includes:

- a series of queues so that the threads can pass data between each other,
- condition variables so that the threads can release the CPU while waiting for data to arrive,
- awkward flow control as threads must hand all MPI communication tasks to one thread and await the results,
- mutex locks around the shared queues and other shared data structures,
- and a host of posix thread library calls.

Figure 4.8 depicts graphically this complicated series of intercommunicating components. An important thing to recognize from the figure is that the dark circles represent that actual application code. These are represented as functions in C and do the work of the application. These functions were literally taken directly from the Coven implementation, with just the Coven-specific header information changed to standard C style. These functions are the only pieces that are needed for the Coven version, as all the other operations depicted in Figure 4.8 are transparent to the Coven user and have been abstracted into the runtime engine.

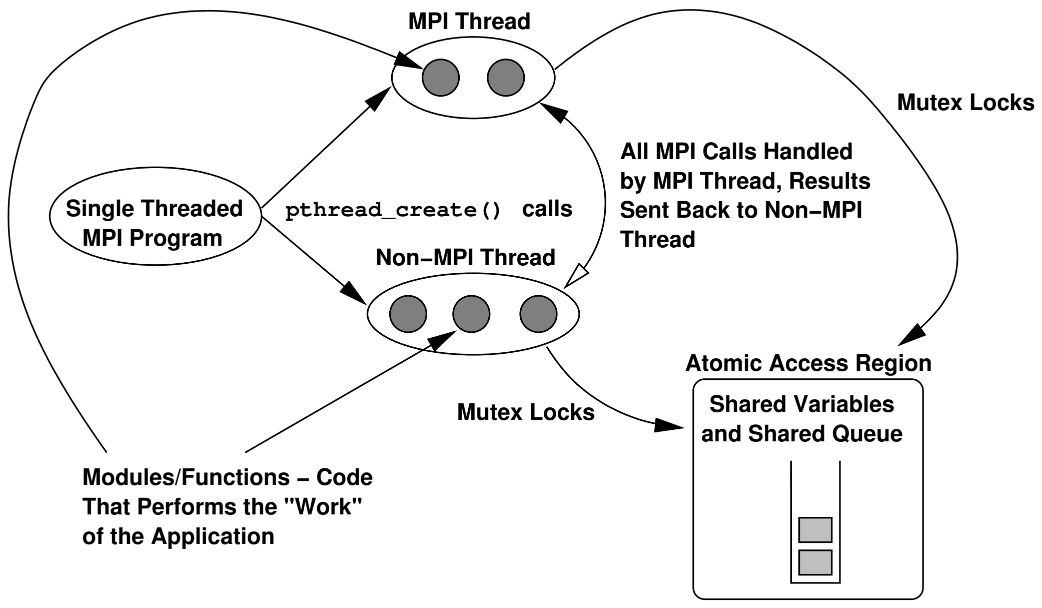

MPI Thread
Mutex Locks
All MPI Calls Handled by MPI Thread, Results Sent Back to Non–MPI Thread
Single Threaded MPI Program
pthread_create() calls
Non–MPI Thread
Atomic Access Region
Shared Variables and Shared Queue
Mutex Locks
Modules/Functions – Code That Performs the "Work" of the Application

Figure 4.8: Non-Coven Multi-threaded Application Diagram

A second approach would be to use more MPI tasks than necessary, scheduling them to run concurrently on the same node to emulate threading. This would allow each task to make MPI calls freely, but would also require a series of queues between MPI “threads” using asynchronous MPI calls and CPU-releasing wait calls.

Either implementation involves substantial programming complexity, especially when considering one target Coven user is a domain specialist with more expertise in his problem domain than in programming. Coven implements something similar to the second approach, but all the complexity is hidden from the user.

The non-Coven, multi-threaded, synthetic application using the first approach (with POSIX threads) was found to be approximately one to two percent faster than the Coven version. The code complexity is considerably larger, however. Our version included approximately 400 lines of multi-threading code. While the modules were taken directly from Coven, this additional multi-threading code was tightly-coupled and would require additional work to generalize it to be usable in a wide range of applications.

No profiling screenshot is presented here as the MPE log files are unaware of the non-MPI threads. This is due to the fact that only the communication thread was ever able to make MPI calls. Profiling and debugging of this type of application are therefore complicated further with this approach. We argue that Coven provides these features to any application implemented in it without need to retrofit multi-threading on top of each existing parallel application.

## 4.4 2D Fast Fourier Transform

For a second application the 2D Fast Fourier Transform (FFT) was chosen. The 2D-FFT is used in many applications and is often considered representative of workloads that operate on matrices that are distributed across the nodes of a parallel machine.

In an additional effort to demonstrate the simplicity of porting an existing application to Coven, an existing FFT library was linked in to Coven and a series of modules were created using it to perform a 2D-FFT. The Fastest Fourier Transform in the West (FFTW) [40] was developed by MIT and is considered one of the fastest FFT implementations available. The FFTW is open source and portable, unlike many vendor implementations. Complex double-precision floating-point numbers are used in the version considered here.

This algorithm is composed of the following four steps:

- compute the 1D-FFT for each row
- transpose the matrix (redistribution of data)
- compute the 1D-FFT for each row
- transpose the matrix (redistribution of data)

These steps were translated directly into Coven modules, each containing no more than 3 lines of calls to FFTW libraries. This results in a port of the FFTW MPI code to Coven modules and demonstrates the ease of wrapping existing code in Coven modules.

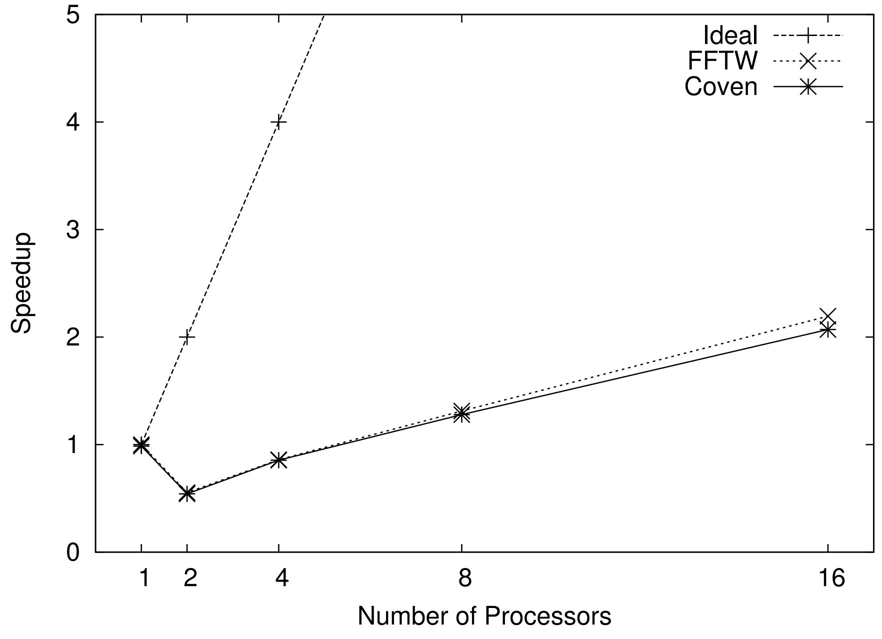


Figure 4.9: FFT Speedup — Coven vs. FFTW

The FFTW provides an MPI parallel implementation and this was used as a comparison application in these tests.

Firstly, a relatively small $4,000 \times 4,000$ 2D-FFT was processed in parallel to compare the FFTW native implementation versus FFTW wrapped in Coven modules. Coven imposed an overhead between 1% to 5% of the total runtime for this application. Figure 4.9 shows the speedup curve for this application.

Next, a large $10,000 \times 10,000$ 2D-FFT was considered. The resulting parallel portion of this FFT is too large to run on a small number of nodes on our system and so 8 and 16 nodes were chosen for testing. First, a single FFT was processed using FFTW and also a port of the application running in Coven. The results are shown in the first two rows of Table 4.1. Coven imposed an overhead of 6.6% in the 8 processor case and 5.1% in the 16 processor case.

Next, three FFTs of $10,000 \times 10,000$ size were processed. The timings for the single FFT case using FFTW were tripled to get the resulting time in row 3 of Table 4.1. Three separate instances of FFTW were then spawned concurrently in an attempt to let the

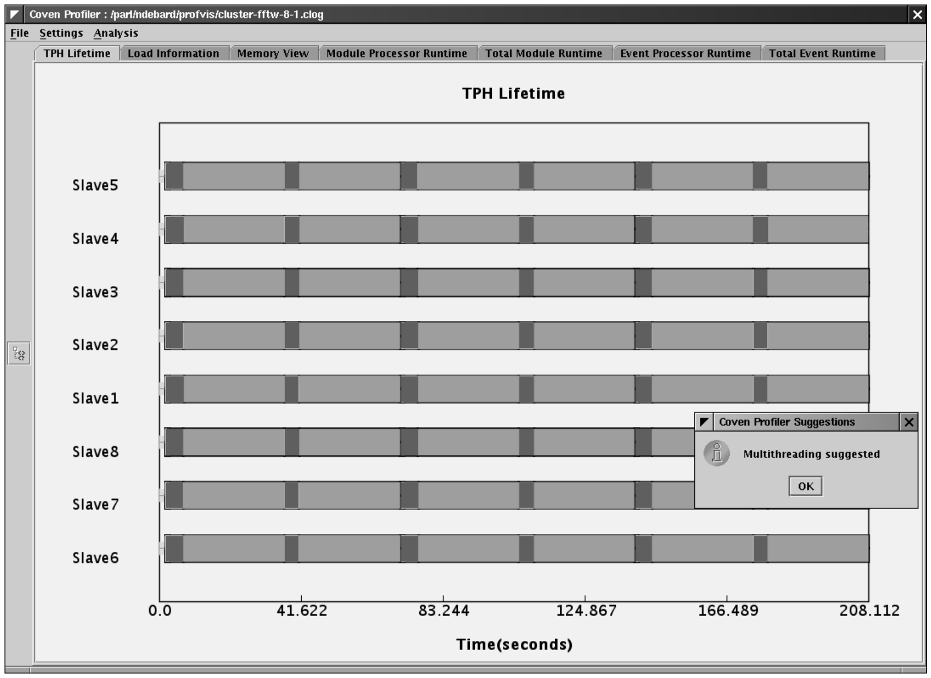
Table 4.1: 10,000 x 10,000 Element FFT Timings

| | 8 Procs. | 16 Procs. |
|---|---|---|
| 1 FFT — FFTW | 63.7s | 42.9s |
| 1 FFT — Coven | 68.2s | 45.2s |
| 3 FFTs — FFTW | | |
| Back-to-Back | 191.1s | 128.7s |
| Concurrent | 173.2s | 111.6s |
| Improvement | 9.3% | 13.3% |
| 3 FFTs — Coven | | |
| 1 Thread | 204.0s | 137.1s |
| 2 Threads | 161.1s | 105.2s |
| Improvement Over Back-to-Back FFTW | 15.7% | 18.2% |
| Improvement Over Concurrent FFTW | 7.0% | 5.7% |

operating system try and overlap computation and communication. The results are shown in row 4 of Table 4.1 which produce an improvement of between 9.3% and 13.3% over running 3 FFTs of this size back to back using FFTW.

Finally, Coven was used to process the 3 FFTs. First, in the single threaded case row 6 of Table 4.1 shows that Coven performs more poorly than either approach using FFTW. However, when 2 threads were used in Coven, rows 7 and 8 of Table 4.1 show an improvement of between 15.7% and 18.2% over the back-to-back execution of 3 FFTs in parallel and between 5.7% and 7.0% over concurrent execution of 3 FFTs using FFTW.

Concurrently running multiple FFTW MPI parallel programs seems like a simple solution to process multiple FFTs but it has some drawbacks. For instance, while the operating system does an acceptable job of scheduling three FFTs concurrently, it is unlikely to handle 10 with the same efficiency. Furthermore, insufficient memory problems are likely to surface with this many large FFTs. Coven, on the other hand, uses a queue system described in Section 2.4.3. With this approach, the user can set it so that no more than 3 FFTs are being processed concurrently. Therefore, as one exits the system and the mem-



Multithreading suggested

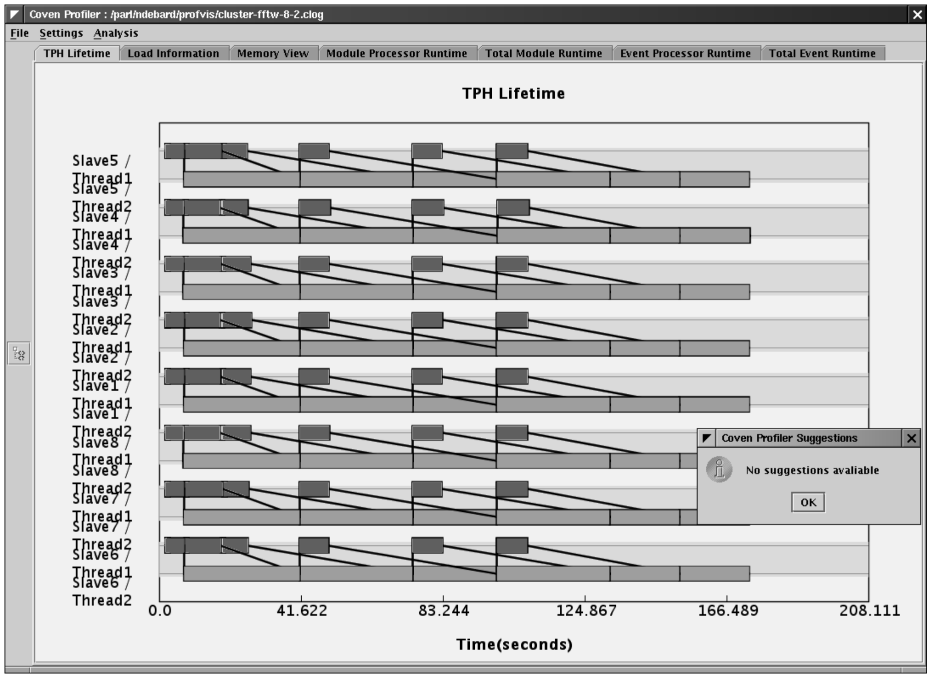
Figure 4.10: FFT Application Single Threaded

ory associated with it is freed, Coven can begin processing another FFT. The FFTW MPI implementation could be augmented to do something similar, however once an application is written in Coven, the programmer gets these features automatically.

The FFTW Coven application did not perform as well as the synthetic application. The reason for this is that the computation phase for this size 2D-FFT takes 5 seconds, while the communication phase takes nearly 30 seconds. This 1:6 ratio is not nearly as balanced as was the synthetic application. It is this reason that the FFTW Coven application does not approach the higher performance increases that are possible when using Coven's multi-threaded capabilities.

Figure 4.10 contains a screenshot of the profiler for the 8 processor, single threaded version of this Coven application. The small blocked regions are the computation modules while the much wider regions are the communication modules.

Figure 4.11 contains a screenshot of this application running in multi-threaded mode with Coven on 8 processors. The figure is scaled for easy comparison with Fig-



No suggestions avaliable

Figure 4.11: FFT Application Multi-threaded

ure 4.10. After the first FFTs computation module, the FFT is handed to the second thread which begins a parallel transpose. The transpose operation on the multi-threaded version takes between 33 and 34 seconds which is around 10% longer than the single threaded version. However, during this time the other two FFTs complete their 1-D computation. This overlap allows the application to achieve the performance gain described above.

Through careful scheduling of which modules execute concurrently, the multi-threaded Coven version of the FFTW application was able to perform better than the naive execution of multiple FFTW MPI programs concurrently. Rather than have 3 FFTs competing for the CPU to perform computations concurrently, Coven lets one FFT get to the communication phase before it lets another into the computation phase.

## 4.5 Related Work

Multi-threading has been utilized in a wide range of computing systems and programming environments. We next look at several related multi-threading projects and how they relate to Coven's multi-threading implementation.

EARTH (Efficient Architecture for Running THreads) [50] is a multi-threaded program execution model which aims to support latency tolerance through fine-grained parallelism. Using EARTH synchronization operations, code is partitioned into threads. Each process consists of an *Execution Unit* (EU) and a *Synchronization Unit* (SU). The EU executes the thread and the SU handles operations requested by the thread. The MANNA (Massively parallel Architecture for Numerical and Nonnumerical Applications) multiprocessor platform uses EARTH, a RISC processor for the EU and a custom piece of hardware for the SU. The MANNA system uses Threaded-C and custom EARTH primitives to aid in multi-threaded programming. Unlike Threaded-C, Coven multi-threading applications are normal C modules without any threading API. As such, it is simple to convert existing applications into Coven modules and trivial to experiment with different threading schemes to study the potential performance improvements.

Tian et al. [86] studied the affect of multi-threading applications on the EARTH-MANNA system, particularly how sensitive performance was to data locality. They defined measures to quantify the sensitivity of data locality to performance. These measures were the Multi-Processor Locality Sensitivity Indicator (MLSI) and the Locality Sensitivity Indicator (LSI). Using a synthetic benchmark and a Matrix-Multiplication program, they found that on the EARTH-MANNA system, the multi-threaded programs could achieve as much as 20% better performance than their single threaded counterpart. They concluded that for performance-tuning, programmers and compilers must expend significant effort to improve data partitioning in an attempt to better cut down on the latencies involved with accessing remote data. Multi-threading provides an easier solution where computation and communication can be overlapped and the system can more easily tolerate communication and synchronization latencies.

A multi-threaded Fast Fourier Transform (FFT) was studied in [85] using the EARTH-MANNA system. Using special EARTH primitives and taking advantages of the MANNA hardware, they were able to design a complex threading scheme which proved to be capable of providing high performance. In particular, they achieved near linear speedup when using very fine-grained multi-threaded parallelism and very good performance improvements when using a more coarse-grained approach. These results were obtained using an EARTH-MANNA simulator called SEMi and were closely tied into the EARTH-MANNA hardware and software primitives. Furthermore, the simulation assumed there was no operating system to multitask between other operations. This multi-threaded FFT study was performed in a simulator of a system with a custom software and hardware system. The multi-threaded FFT experiment detailed in this chapter was performed on a conventional Beowulf cluster and was written in standard C.

The Asynchronous Buffered Computation Design and Engineering Framework Generator (ABCDEFG) [21], or FG for short, is a parallel computing framework which focuses on multi-threading. In FG, each module (called a stage) is executed inside of its own thread. Threads are implemented using the POSIX thread interface, and FG handles all aspects of thread maintenance so the user is relieved of the burden. The most widely used, open source MPI implementation is MPICH [67]. MPICH is currently not thread safe, and this chapter will discuss how Coven's multi-threading deals with this. In FG, however, the developers turned to a commercial MPI implementation, ChaMPIon/Pro. While similar to Coven in some ways, FG places every module inside its own thread. In Coven, the user is given the choice to group related modules into similar threads. Where in Coven a 20 module application might use two or three threads, an FG approach would have 20. This can be a major hindrance to the operating system and potentially could cause slow downs passing data back and forth between threads when the modules have little chance of being able to run asynchronously.

A similar C++ framework that places modular user code into their own threads is DPS [42]. While the specified goal of this approach is to achieve performance improvements from overlapping computation and communication, neither DPS nor FG (mentioned previously) have been used to demonstrate an actual performance improvement using multi-threading. In particular, this would amount to demonstrating a parallel application that can achieve benefits from asynchronous operations as well as showing how the frameworks perform in single threaded mode.

## 4.6 Summary

The multi-threading extension to Coven provides a simple way to achieve benefits from overlapping computation, communication, and external I/O. While not all applications can benefit from this performance enhancement, it is provided with Coven and can be turned on by simply specifying which thread to run each module within.

A real application was tested and shown to be able to achieve a good improvement in performance by using multi-threading. Through a benchmark with many tunable parameters, it has been shown the range of performance that an application can hope to gain from multi-threading. While these particular results are specific to the environment on which the tests were run, they demonstrate the real possibility of improving performance through the use of multi-threading. The fact that multi-threading can improve application performance even in the realm of parallel computing is nothing new. What is new is the ability to abstract away many of the complexities with multi-threading and place them into the environment itself. This expands the ease of implementing multi-threaded applications in particular due to the lack of coupling the threading code with the application code.

The Coven model of computation simplifies the multi-threading implementation in many ways. Transferring data between threads in a hand-coded system can be complex, requiring the use of queues, shared memory, and atomic locks around the data. In Coven, application state is encapsulated within TPHs and Coven already employs TPH queues. Furthermore, by analyzing where a partition of state will travel during its lifetime, Coven efficiently utilizes shared memory for data that will flow between threads sharing a memory. This is empowered by the state model and its property of separating application state from code.

Work is ongoing in studying if a smart agent (in this case, the Coven profiler) can examine trace information from a program run and automatically suggest a threading scheme that is likely to improve performance.

# CHAPTER 5

# LOAD BALANCING

Many types of parallel computing applications exhibit workload characteristics which cause a variable amount of work to be done by each parallel task. This presents a problem because usually there exist at least one point where some set of all parallel tasks must synchronize. Commonly this is the end of the program, or every iteration. Regardless, this presents time where parallel tasks are idle, having completed their work, and waiting for other tasks to catch up. This is termed a *load balancing* problem.

## 5.1 Introduction

With the cost of parallel computing systems and computing cycles so high, developers seek to fully utilize the resources at their disposal. Not only do applications with load balancing problems fail to use all available resources, but this by definition means there exists a potential performance improvement if the application could be adapted to a more balanced workload.

There exists a class of applications that the amount of computation to be performed is dependent on the data. In these applications, it is difficult (sometimes impossible) to predict how much work will be required to compute a partition of the data. Therefore, these applications present major load balancing problems. Examples include ray tracing, N-Body simulations, structured adaptive mesh refinement (SAMR), and many others.

While data-dependent workload applications like these are common load balancing applications, load balancing can be applied to any application when run on a heterogeneous parallel computer. Even the most naturally parallel program can, and usually does, have less than optimal utilization on a heterogeneous computer. It is not uncommon in clusters of workstations (COWs) to have machines which are of a wide range of processing power. The clustering tool Condor [63, 84] (discussed in Chapter 3) creates just this type of heterogeneous cluster, piecing together different speed workstations as they are idle.

Load balancing solutions can be divided into one of two broad categories: static load balancing (SLB) and dynamic load balancing (DLB). In SLB, data is preprocessed before assigning to a parallel task. An algorithm then balances the work to be done between the tasks.

SLB is generally performed on a single processor and sometimes is not included in the performance results of an application. This can make direct comparisons between the two approaches difficult. Using a SLB approach on a heterogeneous parallel computer is complex as it is difficult to balance based on the work to be done while taking into consideration differences in processors, memories, networks, and disk systems.

In DLB, the application is left to run and at runtime the parallel tasks exchange units of work in an attempt to better balance their load. Two popular approaches for DLB are random stealing (RS) and random pushing (RP) [87]. With random stealing, a parallel task requests more work once it has completed all it was assigned. The request is often implemented as a broadcast, where other parallel tasks respond when they are able by transferring a portion of their work to the requester. With random pushing, parallel tasks have a maximum amount of pending work that they are willing to keep and once that number is reached, they send additional work to another random parallel task. Since RS waits until there is no work left to process before seeking out additional work, there is a period of time while waiting for new work to arrive when the parallel task sits idle. RP does not suffer from this, but under high workloads it saturates the network with pushes to tasks which already have work to do.

While static load balancing is useful in many types of applications, those applications generally are of a class where the operations are both well defined (regular) and well known. One approach to achieving SLB in a general purpose environment is through gathering and assimilating trace data from multiple application runs. This approach is similar to SPEEDES [88, 89] and could employ an expert system to recognize how the different runs varied. Performing SLB in a general purpose application can be complex without a detailed understanding of both the application structure and the system on which it will run. Due to Coven's model of computation, the application structure is provided to the environment which would facilitate SLB. At this time, we have chosen not to focus on static load balancing and instead look at a more dynamic approach (DLB) in hopes of it being more applicable to a wider range of applications.

This chapter presents two dynamic load balancing optimizations available to Coven applications. The first DLB algorithm transparently moves TPHs between parallel tasks as less loaded tasks run out of TPHs to process. This is enabled by the Coven model which separates application state from code and makes it simple to move state partitions to execute wherever is deemed most useful. We present a sample application with load imbalances, employ Coven's random stealing DLB algorithm, and then analyze the results. We find that the random stealing algorithm results in the expected improvement in performance and processor utilization.

There are some applications which cannot directly benefit from Coven's random stealing DLB algorithm. For applications of this type we present an explicit DLB algorithm for Coven. This less transparent DLB scheme merely requires inserting a single, pre-built system model into a user's application. We study an application experiencing regular and then random external load which causes the load to change dynamically at runtime. Coven's second DLB algorithm gathers runtime statistical information about the TPHs and uses this to determine the current load. Coven then determines if an imbalance exists and instructs specific parallel tasks what other tasks to offload some of their work to. This is facilitated by the Coven model because related state can be maintained together and that state tagged with its runtime load statistics. State partitions can then be independently analyzed to determine how they affect the overall load of the system.

## 5.2 Approach

Recall that in Coven, all units of work are encapsulated in an internal data structure, transparent to the application programmer, called a Tagged Partition Handle (TPH). This structure is indirectly controlled by a user through system modules which partition, scatter, gather, and reform TPHs. Each parallel Coven Thread operates on a single TPH at a time, while maintaining queues of TPHs to be processed and to be sent on. We call these input and output queues. After processing a TPH the program sequencer returns to its input queue to retrieve the next one. TPHs arrive in this input queue from another Coven Thread which sends the TPH on to another program sequencer to process.

By employing the TPH queuing system, we are able to readily identify how many units of work are present on each parallel Coven Thread. This enables Coven to determine and monitor the amount of work scheduled for computation throughout the system. Given the work queuing system, the most relevant DLB algorithms are random pushing and random stealing. Random pushing was found to be unstable in environments with high workload [87] while random stealing (RS), on the other hand, proves to be much more stable and regular. Coven's DLB approach focuses on using random stealing to allow parallel tasks to request additional work when they are lightly loaded.

While similar algorithmically to the work done by Nikolopoulos [87] on random stealing, Coven brings this technology to a parallel problem solving environment. Unlike many load balancing approaches which target a very specific application domain, Coven has been used to implement a wide range of applications with varying workload characteristics. With this approach, dynamic load balancing becomes available to *all* applications written in Coven. The application programmer receives this benefit for no effort on their part, as Coven already enforces a data partitioning scheme which places data into bundled units of work (TPHs).

Random stealing requires each parallel task to be assigned more than one portion of the data. This requires splitting data into more partitions than there are parallel tasks.

Coven's model of computation addresses this point and the split/join features of the Coven PSE detailed in Chapter 2 facilitate this. Partitioning of data into more segments than there are parallel tasks has some advantages. For instance, this approach increases the rate at which results appear at the output of the algorithm. While the total results should come through no faster (and often a tad slower), this approach does lend itself well to load balancing. The idea being, if the work is divided into more segments than there are tasks, then faster tasks can "steal" work from slower tasks if the need arises.

This chapter illustrates two types of dynamic load balancing inside of Coven. Both approaches rely on the concept of partitioning data into more segments than there are parallel tasks. This presents an interesting problem when sending messages between segments which may be queued for processing on another (or even the same) parallel task. To address this problem, Coven's virtualization feature (Section 2.4.8) makes it easy to send and receive data by specifying which state partition to communicate with.

The first DLB approach presented in this chapter is useful in algorithms which are non-iterative. It utilizes random stealing by having a Coven task send out a "steal request" message once it runs out of TPHs (work) to process. Heavily loaded tasks can then choose to respond by sending a portion of their work to be processed by the stealer.

The second DLB approach presented in this chapter is less automatic. It requires the user to insert a Coven system module which attempts to do load balancing. This approach uses a more sophisticated load balancing algorithm which analyzes TPH throughput through each parallel task and employs a centralized algorithm to determine better TPH balance through all parallel tasks.

The following sections explain the rationale for these dynamic load balancing algorithms as well as the changes to Coven which were required. Experiments are provided which analyze the performance and study the usefulness of each DLB approach.

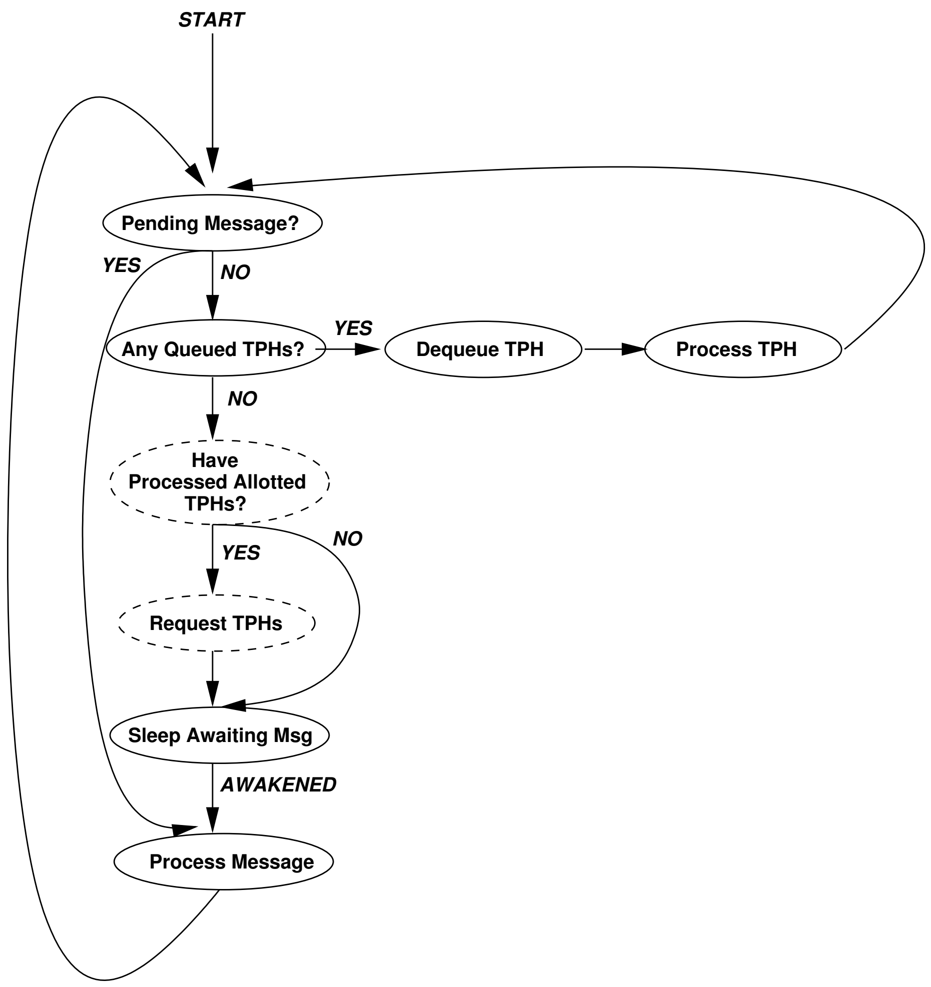


Figure 5.1: Message Flow State Transitions With Random Stealing

## 5.3 Random Stealing Implementation

In Chapter 2, Figure 2.8 introduced how messages and TPHs flow through Coven. To handle dynamic load balancing through random stealing, the state diagram is augmented as shown in Figure 5.1. Figure 5.1 includes the new states surrounded by dotted lines. Once a program sequencer realizes that there are no incoming messages and no TPHs queued for processing, the random stealing algorithm is initiated.

Coven's random stealing implementation involves the use of several control messages which are exchanged between the parallel tasks in order to communicate information about the DLB operation. These control messages are briefly defined below and assume that Task $A$ is requesting work and Task $B$ is responding to the request.

**Steal Request** message sent from Task $A$ to all other parallel tasks signifying that additional work is requested

**Steal Response** message sent from Task $B$ to a requesting Task $A$ signifying that $N$ TPHs (pieces of work) are being sent

**Steal Response Notification** message sent from Task $B$ to all tasks except $A$ (and itself) signifying that Task $B$ has fulfilled $N$ requests from Task $A$

Each parallel task is allotted a certain number of TPHs defined by the application and the user's specification. Each task is required to process that number of TPHs before requesting additional work. This addresses a race condition where tasks have no work initially and flood the network with steal requests before their allotted work has arrived. Only once a task has processed their allotted number of TPHs may it request additional work from other, heavily loaded tasks.

Coven uses a load balancing *score board* to keep track of which tasks have requested to steal work and which tasks have responded to their requests. Each Coven task maintains their own score board. The score board is implemented on each task as an integer array with each element corresponding to the load balancing requests from a parallel Coven task. Each element in the score board is referred to as a *slot*.

As a steal request arrives at Coven task $B$ from Coven task $A$, $B$ increments a counter in the score board slot corresponding to task $A$. Coven processes control messages before dequeuing and operating on any queued TPHs. In this manner, we assure that all arrived steal request messages have been incorporated into the score board before the receiving task begins processing another TPH. Once task $B$ has received any pending DLB control messages, it then analyzes the score board at its TPH input work queue. Task $B$ then attempts to evenly distribute its TPH input queue between all tasks which have positive slot values in its score board. For each task that $B$ sends TPHs to it issues a steal response notification to all the other tasks. Additionally, it decrements the slot value in the score board corresponding to the task that it transferred work.

When a steal response notification arrives at a task $C$ that message signifies that some task $D$ has fulfilled a request for another task $E$. Task $C$ then decrements the slot value in its score board corresponding to slot $E$. This may cause the slot value to become negative and can occur when a steal response notification arrives before the initial steal request message.

This score board is required in order to deal with potential race conditions which were seen to occur empirically. The following series of steps would be possible if not for the use of a slot value whose positive or negative status signifies the number of unfulfilled steal requests.

1. Task $A$ sends a *steal request* to all other tasks.
2. Task $B$ receives the *steal request* message, sends a *steal response* and some TPHs to Task $A$, and a *steal response notification* message to all other tasks notifying that Task $B$ has responded to Task $A$'s request.
3. Task $C$ receives the *steal response notification* message from Task $B$ before the *steal request* message from Task $A$.
4. Task $C$ receives the *steal request* from Task $A$ and sends some TPHs to Task $A$.
5. Task $A$ is now likely overloaded and the total load is likely imbalanced.

In the above simple scenario, Task $C$ would have assigned the value of $-1$ in its score board in the slot for Task $A$ when it received the request fulfilled message from Task $B$. Then, when the request from Task $A$ arrived, this value would have been incremented by one to $0$. Task $C$ would then have chosen not to send any TPHs.

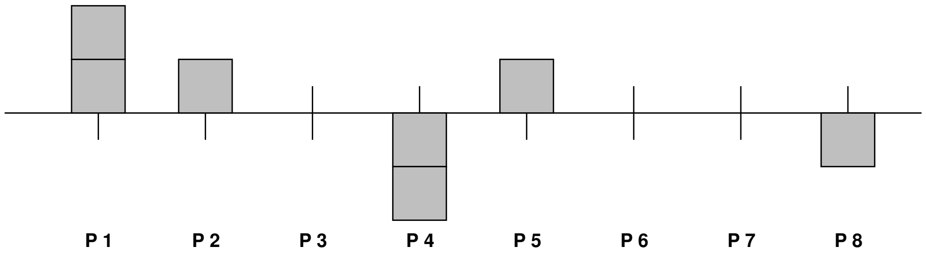


Figure 5.2: Load Balancing Score Board

Figure 5.2 depicts graphically how the score board might appear in an example with more tasks. In this figure, processes are represented on the x-axis ranging from `P1` through `P8`. This figure might represent a snapshot of the score board as seen from `P6`'s perspective at some point during the application.

When sending TPHs to a stealer, Coven employs an algorithm which attempts to evenly divide number of TPHs between all the stealers. In the previous example (Figure 5.2), the number of queued TPHs will be divided between `P1`, `P2`, `P5` (those slots with positive values), and the sender (`P6`). The sender will then dequeue, marshall, and transfer the necessary number of TPHs to each stealer, send out request fulfilled messages to all other tasks, and then decrement the value of the appropriate element in its score board.

To study the random stealing implementation we next present an experiment that benefits from the optimization. We analyze the performance improvement it provides and compare the complexity (from a user's perspective) of using the optimization within the environment with the difficulty of implementing DLB by hand.

## 5.4 Random Stealing Case Study

To study the Coven implementation of the random stealing DLB optimization we next present an application with imbalanced workloads. This application is the calculation of the Mandelbrot fractal set which is commonly used as an example of an application that needs DLB. Figure 5.3 is a sample image of a fractal generated by this application. A fractal is a geometric shape that generally has the property of self similarity, meaning bits

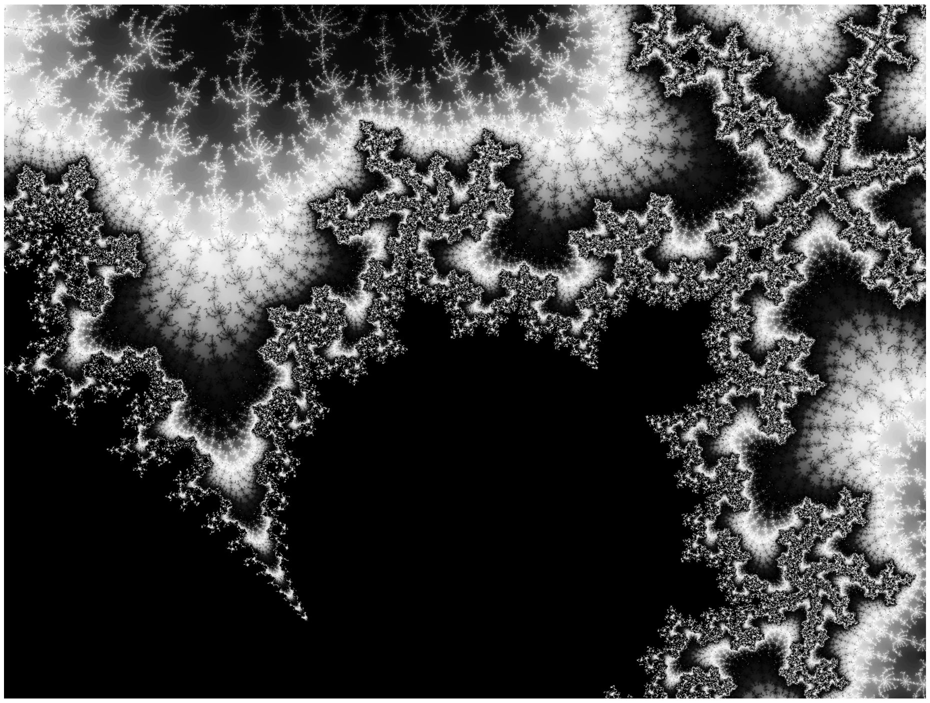


Figure 5.3: Sample Fractal Image

of the fractal look like the whole. Additionally, they generally are independent of scale and look similar regardless of how close one zooms in.

The Mandelbrot set is a set of complex numbers defined by Equation 5.1. In Equation 5.1, $\mathbb{C}$ is the set of all complex numbers and $Z_n$ is a recursive function defined by Equations 5.2 and 5.3.

$$M = \left\{ c \in \mathbb{C} \mid \lim_{n \to \infty} Z_n \neq \infty \right\} \tag{5.1}$$

$$Z_0 = c \tag{5.2}$$

$$Z_{n+1} = Z_n^2 + c \tag{5.3}$$

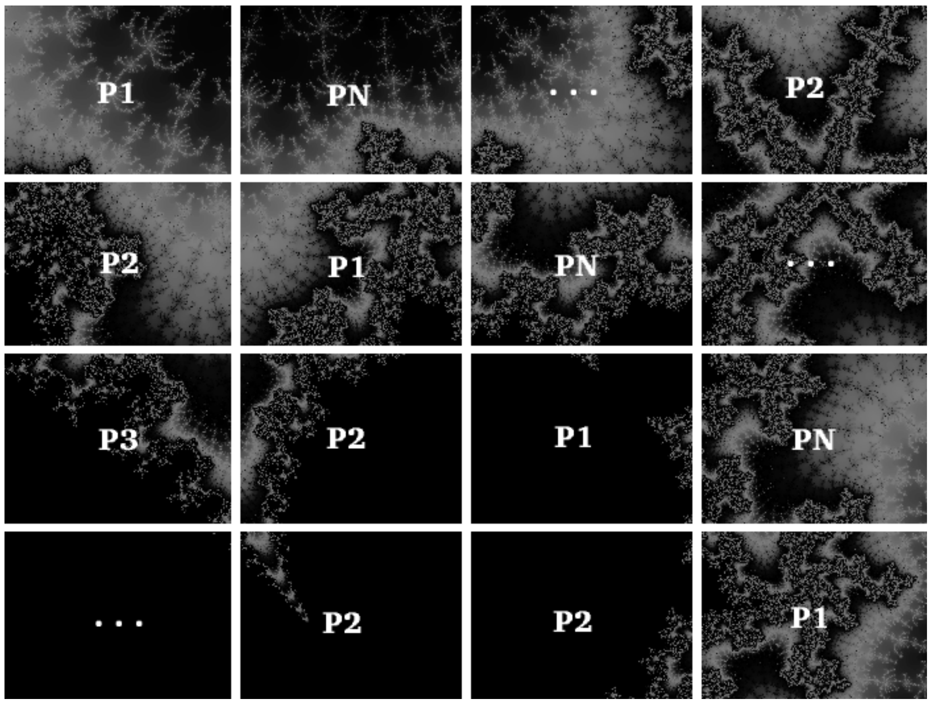


Figure 5.4: Sample Fractal Image Partitioning

If the value of $Z_n(c)$ is not infinite when $n$ approaches infinity, then $c$ belongs to the set [2].

Like most image processing applications, the fractal is calculated by dividing the image into regions. Each region is distributed to a parallel task which is responsible for computing a portion of the fractal over that region. This particular application is *naturally parallel*, meaning there is no communication required between the parallel tasks except (in this case) to reconstitute the solution into a final image.

Figure 5.4 depicts one partitioning of this fractal region. In this example, the image is partitioned into four segments in the X and four segments in the Y, creating a total of 16 segments. This is then distributed among N parallel tasks using the round-robin fashion depicted in the figure.

Calculating the Mandelbrot set requires that an association be made between pixel coordinates and complex numbers. Then, for each pixel we determine if the corresponding complex number belongs to the Mandelbrot set by evaluating $Z_n$ from Equation 5.3. Since we cannot loop to infinity in practice, we must choose some maximum value to bound the loop — we call this the $depth$. In [2], it is proven that if “the absolute value of $Z$ ever gets bigger than $2$ it will never return to a place closer than $2$ but will rapidly escape to infinity.” This means that the recursive $Z_n$ function can abort if, for a given complex number, the function’s value increases beyond $2$.

For each complex number (related to the pixel coordinate), the number of iterations required to determine if that number is part of the Mandelbrot set is variable. This means that it is essentially impossible to determine how many computations will be required to calculate the Mandelbrot set over a set of pixels. This makes it impossible to balance the work of calculating the fractal by partitioning the image into equal sizes. For these reasons, calculating the Mandelbrot set is a prime example of an application that has load imbalances.

We next present the Coven modules that implements the Mandelbrot set algorithm. We then analyze the application’s runtime behavior, showing the effects of the load imbalance and then evaluate the use of Coven’s random stealing optimization to reduce the imbalance. Coven’s profiler is employed to graphically depict the load imbalances as well as the more balanced work load after optimization.

### 5.4.1 Module Decomposition

The fractal algorithm was decomposed into a series of only a few Coven modules. These are shown in the Coven data flow graph editor in Figure 5.5. Three modules `frac_scatter`, `frac_gather` and `frac_write` reside in the sequential design. The parallel portion of this application is only one module, `frac_comp`.

The application requires the user to set parameters for the image dimensions and how detailed (the *depth* mentioned in the previous section) the fractal will be. The image is partitioned into NX by NY segments, where NX and NY are also input parameters. The scatter module then creates TPHs each assigned to a different partition and then scatters the

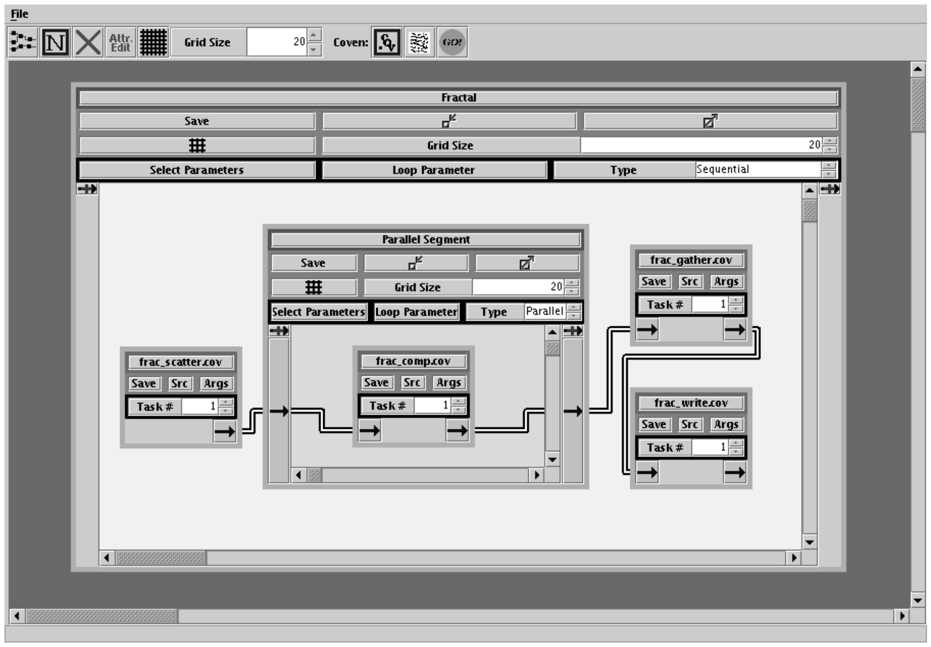


Figure 5.5: Fractal Data Flow Graph

TPHs in a round-robin fashion to as many parallel slave tasks as requested. Each parallel task then computes the fractal over that region, executing the `frac_comp` module. Results are then gathered and written to an image PNG file in the final two modules.

Figure 5.6 is a source code listing of the fractal parallel computation module. There are five constants used as input to the module; these are set by the user before the application runs. A four-tuple of input integers represents the region information. The module creates an output buffer of necessary size and names it `subdata`. The body of the `frac_comp` module is three nested `for` loops which compute the fractal over the subimage. Notice the most inner `for` loop which counts up to `depth` but can short circuit execution if a condition is met. This part represents the portion of the code that is data dependent and causes each point in the image to take a variable amount of time to compute.

The fractal is a mathematical entity which is believed to be found in nature. One of its properties is that it is infinitely "zoomable," meaning that as one delves deeper and deeper into the image created by the Mandelbrot set, it continues to expand. The fractal algorithm goes through a `for` loop of configurable max depth in search of a solution. If no

```
#include <coven.h>
#include <unistd.h>
#include <math.h>
#include <stdlib.h>

COVEN_Module frac_comp
  (
        const int depth;
        const int tot_width;
        const double ymax;
        const double xmax;
        const double xmin;

        input int start_width;
        input int end_width;
        input int start_height;
        input int end_height;

        output buffer unsigned int subdata[(end_width - start_width) *
                                           (end_height - start_height)];
  )
{
        int index, i, j, k;
        unsigned int z;
        double x, y, c_re, c_im, re, im, re2, im2;

        for(i=start_height; i<end_height; i++) {
            y = ymax - ((xmax - xmin)/(double)tot_width) * (double)i;
            for(j=start_width; j<end_width; j++) {
                index = (i - start_height) * (end_width -
                        start_width) + (j - start_width);
                x = ((double)j/(double)tot_width) * (xmax - xmin) + xmin;

                re = x;
                im = y;
                re2 = re * re;
                im2 = im * im;
                c_re = x;
                c_im = y;

                for(z=0; z<depth; z++) {
                    im = 2.0 * re * im + c_im;
                    re = re2 - im2 + c_re;
                    re2 = re * re;
                    im2 = im * im;
                    if((re2 + im2) > 4.0) break; // solution found, short circuit loop
                }

                if(z == depth) z = UINT_MAX;

                subdata[index] = z;
            }
        }
}
```

Figure 5.6: Fractal Computation Module

solution is found, the pixel is marked undefined (or black in the image). The `for` loop short circuits and aborts once a value has been found, and the number of iterations it takes to find the value corresponds with the color in the image. A small maximum depth the `for` loop can go results in faster fractal generation, but considerably less detailed. Therefore, since for each point in the image the fractal algorithm makes a variable number of computations, it exhibits major load imbalances.

We next examine how the fractal application performs in Coven with and without dynamic load balancing.

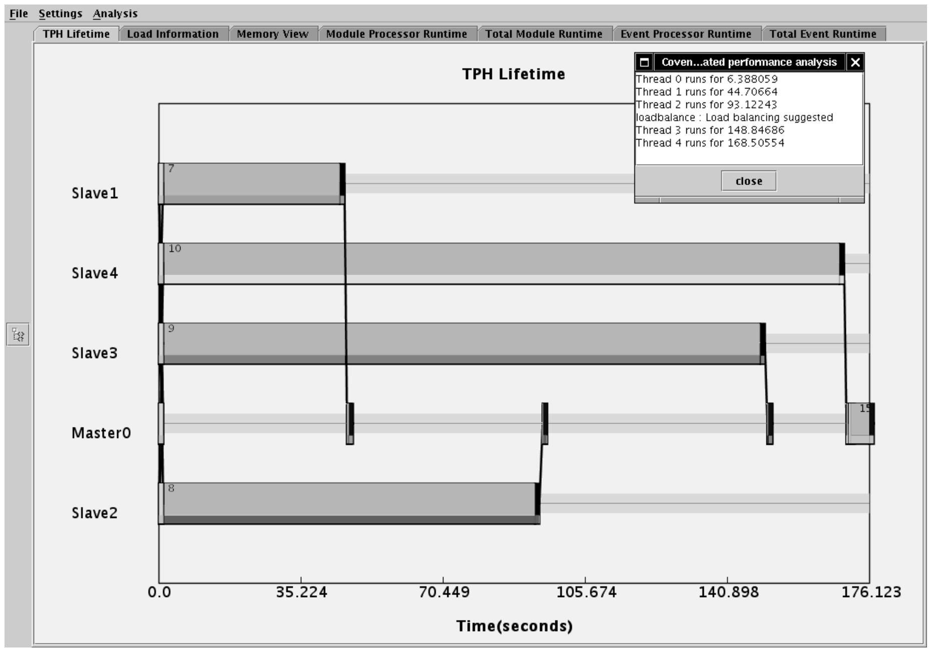
### 5.4.2 Performance Without Load Balancing

The fractal application outlined in the previous sections was used to generate a large, detailed fractal image (the same one shown in Figure 5.3). The final image was $5000 \times 3759$ and over $8.5$ megabytes in size. The application was executed in parallel, but shown here on merely four processors for simplification and easy visualization of performance data. This helps to illustrate the need for load balancing and the issues addressed in this chapter.

First, in order to analyze the load imbalances of the application we ran it with one partition of the image assigned to each parallel task. Figure 5.7 is a screenshot of the Coven profiler visualization depicting the time it took to process each partition. In the figure, time is represented on the X-axis, ranging from program start (zero) in this case up to 176.123 seconds. The Y-axis represents each parallel task. In this case, there are four parallel slave tasks and a master controlling task, which is being used to distribute and coalesce the fractal data. The rectangular boxes in the figure represent individual portions of the fractal image, similar to as was shown in Figure 5.4 except there are only four large partitions of the fractal, rather than many small ones. Each portion is described entirely by a single TPH. The boxes are numbered to correspond to unique internal TPH identifiers, and colored using a unique mapping between color and identification number.

Table 5.1, column two (original†) presents additional performance data from this application. The percent utilization is the sum of the time each parallel task spent unidle divided by the total application runtime. The average wait time is the average amount of time each parallel task was idle waiting on new data or for the application to complete. For this experiment, we found a utilization of 64.11% and an average wait time of 62.23 seconds.

In this example, since the fractal image was partitioned into only one segment per parallel task, dynamic load balancing is completely prohibited with Coven. The screenshot includes a dialog box overlayed from the Coven expert system which, after analyzing the data, has determined that load balancing might provide a performance improvement. Note



```
Thread 0 runs for 6.388059
Thread 1 runs for 44.70664
Thread 2 runs for 93.12243
loadbalance : Load balancing suggested
Thread 3 runs for 148.84686
Thread 4 runs for 168.50554
```

Figure 5.7: Four Processor Fractal No DLB Possible

Table 5.1: Fractal Performance Comparison

| | orig. no LB† | partitioned no LB⋆ | RS DLB◊ |
|---|---|---|---|
| run time (secs) | 176.12 sec | 181.74 sec | 154.61 sec |
| percent utilization | 64.11% | 62.53% | 73.51% |
| avg. wait (secs) | 62.23 sec | 67.81 sec | 40.56 sec |

† From Figure 5.7 — original application, no LB

⋆ From Figure 5.8 — original application with partitioning, no LB

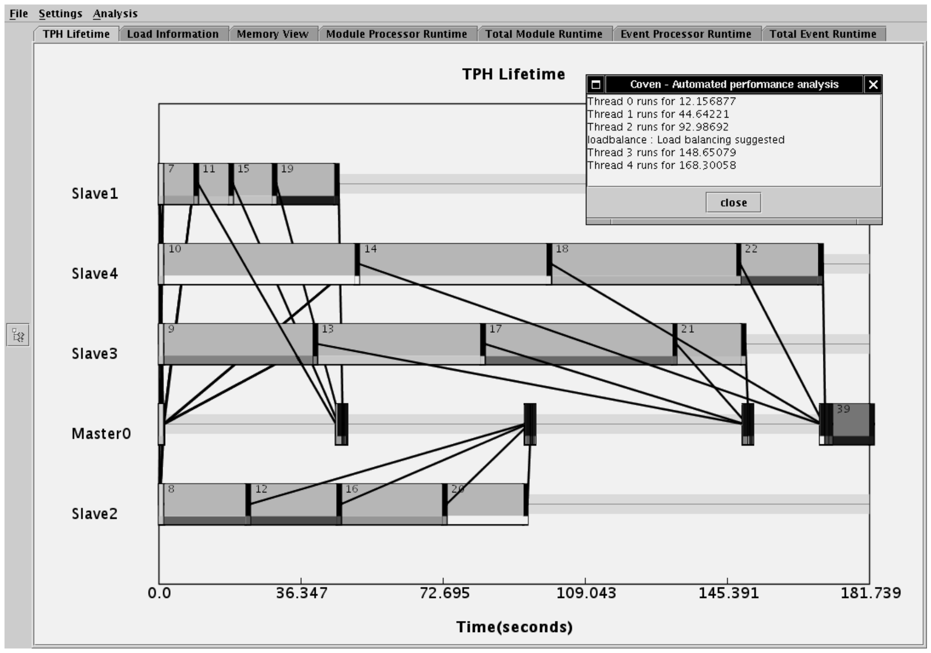
◊ From Figure 5.9 — application using random stealing DLB that the runtimes for each parallel task have a great deal of variance. Before load balancing can be performed, the application data must be partitioned into more segments, so that Coven's random stealing algorithm can move around TPHs to better balance the load.

Next, in order to measure the overhead of using multiple smaller partitions we repartitioned the data and reran the application. The repartitioning scheme created many smaller partitions which were scheduled so that each parallel task was responsible for multiple partitions. The total amount of work each task was assigned remained the same, but the work was now split into smaller packets.

Figure 5.8 depicts this version of the application with four partitions assigned to each parallel task, totaling 16 partitions. In this example, dynamic load balancing is still not enabled. Notice that the runtime increases from 176 seconds to 181 seconds, an increase of about 3%. This can be accounted for with the increased decomposition of the data into more segments than there are parallel tasks. In this case, this type of decomposition results in a worse runtime. However, this decomposition scheme facilitates load balancing, and the goal is to show that after using DLB, the runtime is cut to less than 176 seconds (the default case shown in Figure 5.7).

As previously stated, the amount of computation of each segment varies and therefore load balance cannot be assured. Notice in Figure 5.8 the parallel task on the top (*Slave 1*) processes its alloted four pieces very quickly while *Slave 4* (the one below it) is the last to finish, taking nearly four times as long. By looking at the post mortem analysis of this application, it is clear that *Slave 1* and *Slave 2* (on the bottom) could have been assigned additional work to do.

Table 5.1, column three (original partitioned$\star$) presents additional performance data from this application. Notice that when compared with the original† results from column two, the utilization decreased and the average wait time increased. This is due to the overhead incurred by splitting the data into smaller partitions but not using load balancing.



```
Thread 0 runs for 12.156877
Thread 1 runs for 44.64221
Thread 2 runs for 92.98692
loadbalance : Load balancing suggested
Thread 3 runs for 148.65079
Thread 4 runs for 168.30058
```

Figure 5.8: Four Processor Fractal Without DLB

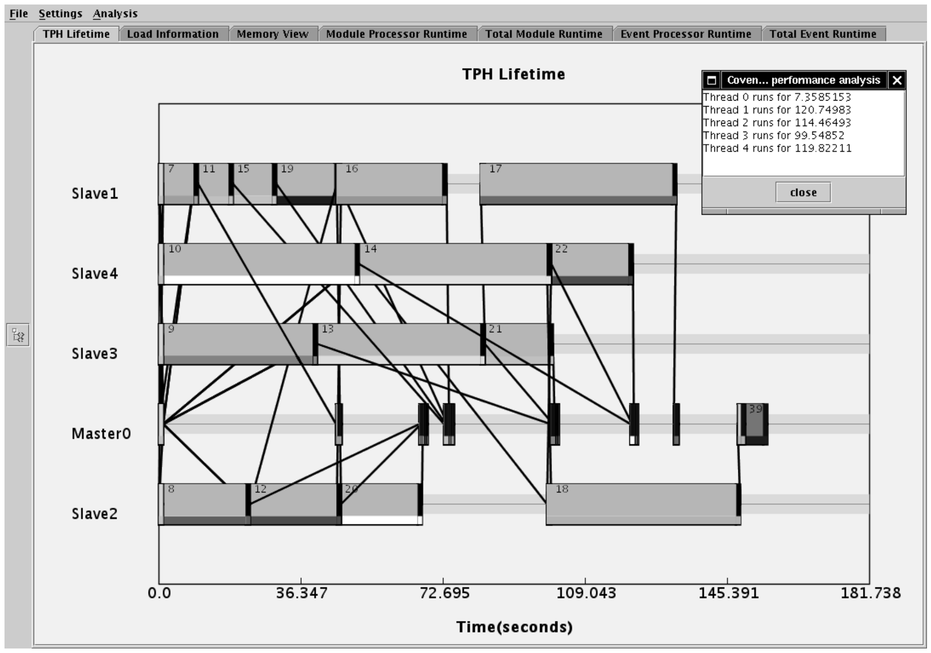
---

In Figure 5.8 the profiler's expert system dialog box states "(l)oad balancing suggested." This is determined using a heuristic and the relative runtimes of each parallel task. This dialog box also prints the total time the CPU was utilized by each parallel task. It is also easy to tell from here that tasks 1 and 2 were idle for a large portion of the program.

### 5.4.3 Performance With Random Stealing DLB

The same fractal application with the same input parameters was then run using Coven's dynamic load balancing (DLB) random stealing algorithm. Each task was required to have received and handled their allotted TPHs, in this case four. By handled we mean either process that segment themselves or send it to another task for processing, through a random stealing operation.

In random stealing, the outcome will vary greatly depending on the specific global state of the system when one task issues a steal request. Figure 5.9 is a screenshot of the Coven profiler doing post mortem analysis on the fractal application after dynamic load balancing had been enabled. Notice that *Slave 1* processed its allotted four portions and



```
Thread 0 runs for 7.3585153
Thread 1 runs for 120.74983
Thread 2 runs for 114.46493
Thread 3 runs for 99.54852
Thread 4 runs for 119.82211
```

Figure 5.9: Four Processor Fractal With Random Stealing DLB

then sent out a steal request. The first parallel task to receive the request was *Slave 2*, as *Slave 2* finished processing its TPH before the other slaves. *Slave 2* recognized that it had two TPHs left, #16 and #20 (see Figure 5.8 for evidence of the TPHs *Slave 2* was allotted). Half of its work (one TPH) was then transferred to *Slave 1*.

*Slave 2* itself then requested additional work, since after processing TPH #20 it was idle. There was a relatively long idle period while the task waited for work to arrive, in this case from *Slave 4*. During this time, *Slave 1* requested even more addition work and this time its request was fulfilled by *Slave 3*. Notice that *Slave 2* requested additional work before *Slave 1*'s second request but *Slave 3* satisfied *Slave 1*'s request rather than *Slave 2*. This is because when *Slave 3* was ready to handle steal requests, it had requests from both *Slave 1* and *Slave 2* and it randomly chose one to satisfy.

Figure 5.9 has been scaled to make it easily comparable to the non-load balanced version in Figure 5.8. The total runtime of the DLB version was 154.55 seconds, which is approximately 12.25% faster than the version where DLB was not possible (Figure 5.7), and 15% faster than the previous example (Figure 5.8). Notice in the expert system dialog box there is no suggested optimization. Furthermore, the runtimes for each parallel task are considerably more balanced, having a great deal less variation. This results in a better utilization of the computing resources through faster application runtime.

In the dynamic load balanced version of the application shown in Figure 5.9, the more heavily loaded parallel tasks (*Slave 4* and *Slave 3*) were able to hand off a portion of their work to other tasks, in particular, bringing *Slave 1* more into balance with the other parallel tasks by having that task process 50% more work than was allotted to it.

Table 5.1, column four (random stealing$\Diamond$) presents additional performance data from this application. Not only does the total runtime decrease to 154.61 seconds, but the percent utilization increases and the average wait time decreases. This results in quicker application runtime as well as less processor idle time and better utilization of the computational resources.

Dynamic load balancing through random stealing is a transparent way users can achieve better load balancing in their Coven applications. It requires that an application be partitioned into more TPHs than necessary and this granularity definitely can affect application performance. A too fine granularity would mean more time was spent handling transportation of TPHs and bookkeeping rather than performing the computation, however, too coarse of a granularity would prohibit proper load balancing.

### 5.4.4 Non-Coven Performance and Complexity Comparison

A non-Coven implementation of the fractal application in parallel using random stealing would likely result in the same results seen elsewhere in this thesis; the performance would be slightly (1% to 3%) better than the Coven version but the programming complexity would be much larger. Instead of implementing the non-Coven version, it was instead decided to implement a version of the fractal application that used a different type of load balancing.

This new form of load balancing comes from where each application is assigned only a single piece of the fractal, though the fractal is still segmented into many more pieces than there are parallel tasks. When a task completes processing its segment, it sends a request to the “master” task which then sends over another segment to work on. This is a fairly common method of dynamic load balancing, specifically being the method of choice for MPI Povray [3], a parallel image rendering application.

There are advantages and disadvantages, of course, with this form of DLB as well. One advantage is that a segment is only sent to a parallel task when it will be executed there, as opposed to random stealing where a segment can be moved from task to task responding to dynamic load. One major disadvantage is that the task that hands out work becomes a bottleneck, particularly being a performance concern as the number of parallel tasks goes up. Another disadvantage is true of either the random stealing or this other DLB approach — the code to implement either is non-trivial.

A non-Coven random stealing implementation would be particularly complex to implement. It would require workload queues and only when no additional incoming work was available would a task being processing an element. The system would need to use both data message (to exchange fractal segments) and control messages (to exchange random stealing requests).

The master distribute load balancing algorithm is a little simpler to implement. Each task merely goes to sleep waiting for data to arrive. Once they receive that data, they process it, and send it back for reassembly. Upon sending it back, the master recognizes that the task is no longer busy and sends another segment. This continues until there are no segments left to distribute. While this version is certainly simpler to implement than a non-Coven random stealing application, the argument still stands that the Coven version of the fractal application has *no* dynamic load balancing code present at all. Any addition of DLB to the non-Coven version is purely extra code that would have been handled by the environment in a Coven version of the application.

As with the other non-Coven applications, the lack of a profiling library makes it difficult to analyze the trace information. While MPE and Jumpshot provide some view of application runtime, the fact that the fractal segments have no representation in MPE makes it difficult to establish what is going on. This is just another example of a feature that is transparently provided by Coven.

The non-Coven version of the master distribute DLB algorithm was implemented. The exact same four processor test was performed as was seen in the previous sections. The hand coded version performed about 7% faster than the Coven implementation. This is a bit better than the other hand coded implementations seen elsewhere in this thesis, but a portion of that performance is due to the particular decomposition of the fractal. To demonstrate this fact, the application was divided into a very small grain size. Instead of four segments per processor the fractal region was divided into twenty segments per processor. This version performed better in the Coven implementation due to the fact that when random stealing was initiated, many segments were transferred at once to the stealer. The master distribute DLB version, however, overloaded the master task with many requests for small segments which were rapidly processed.

Obviously this version could be optimized further to perhaps transport multiple segments together, or even use a mixture of random stealing and master distribute DLB. The point still remains that the hand coded version used the exact code from the Coven modules, but additionally required a great deal of handshaking, control, and DLB code. The advantage of Coven is that these operations, common to many parallel applications, have been incorporated into the environment so that the user can steer and control them without having to implement the features.

### 5.4.5 Discussion

Clearly dynamic load balancing through random stealing can be useful to Coven applications. A sample application was presented which shows an imbalanced workload and how through the use of Coven's built-in random stealing functionality, the program runtime was cut by a good amount. An important fact to recognize is that a user is able to try random stealing in Coven by simply toggling a flag.

There are many instances of the fractal application where the workload can be balanced, through different parameter choice or different processor configuration. While these configurations would result in little or no benefit from DLB, the user can try it out for "free." Clearly, implementing random stealing by hand (as discussed in Section 5.4.4) can be very complex, especially when compared to the ease of toggling random stealing on or off in the Coven version.

While this style of DLB is very successful inside of Coven, we want to study another more complicated approach. This first version of DLB is useful mainly in naturally parallel applications, or applications whose parallel tasks do not require much (or any) communication. While a large class of problems conforms to this, many problems do not and we want to make DLB possible to those less naturally parallel applications implemented in Coven.

While this study looked only at a homogeneous computing system, it is also applicable to parallel computers where the nodes vary in performance. This would result in the time it takes to process each TPH (fractal segment) varying not only based on the fractal data, but also on hardware limitations. Additionally, in a multitasking machine, parallel nodes may be competing for the processor with other tasks. This would additionally augment the time it takes to process a TPH. While this aspect was not explored in this study, the next study of less naturally parallel applications will address this issue.

## 5.5 Explicit Load Balancing Implementation

While the random stealing algorithm described previously is useful in a large class of applications, iterative applications who are processing a set of data over and over again have a more difficult time taking advantage of Coven's random stealing implementation. This is due to the fact that the RS algorithm initializes once a parallel task has no TPHs left to process. In an iterative operation, especially one that uses inter-task communication, load balancing issues can arise due to slow parallel tasks while each task still has TPHs left to process.

Therefore, we implemented what we termed explicit load balancing inside of Coven. The user initiates this by placing a system module into his data flow graph at the end of an iteration. The following sequence of events describes how this system module works:

1. Each TPH, in sequence, arrives at the DLB system module
2. Once a TPH arrives, the DLB system module calls COVEN_Barrier_Balance()
3. This function halts the TPH, stopping it from proceeding (looping back around)
4. The TPH is placed into a holding queue
5. Once all TPHs arrive at a parallel task, that task communicates with a leader task, informing the leader the wall clock time each TPH spent executing modules
6. An algorithm is performed on the leader which determines if the system is in balance within some threshold
7. If not, the leader instructs the chosen parallel tasks to transfer work to other chosen parallel tasks in an effort to improve balance
8. The TPHs are released from their queues, reloop, and this occurs again the next iteration

The user can configure how often load balancing occurs, such as every N iterations. While this DLB algorithm is costly (requiring a complete synchronization, global communication, and then TPH reshuffling), it is often more costly to remain in an imbalanced state — particularly for long running applications.

As with random stealing, explicit load balancing requires the data be partitioned into more TPHs than there are processors. This, of course, is so that the TPHs can be shifted around from more loaded processes to less loaded ones. Additionally, since explicit load balancing specifically is designed to address instances where modules communicate with other parallel modules, virtualization is required. Process virtualization was explained in Section 2.4.8.

By using virtualization, modules (processing TPHs) can communicate with other parallel modules by specifying which segment of the data they are trying to communicate with, rather than which parallel task. This is important in explicit DLB, since Coven will be moving TPHs between processors. Without being able to specify which data partition the user wanted to communicate with, they would have to *know* where that partition resided and use MPI communication operations to communicate with a specific MPI task. Since Coven is handling the DLB for the user, it must also provide a way for them to communicate with a TPH without specific knowledge of where that TPH resides. This functionality is all provided in Coven and is utilized in the following example.

## 5.6 Explicit Load Balancing Case Study

An N-Body application was developed in Coven as a demonstration application for explicit load balancing. This application uses a Coven system module which handles the dynamic load balancing. Figure 5.10 is a code listing of this extremely short system module. This is intended as a module that can be dropped into any existing application to achieve explicit DLB. The application must be iterative, as described in the previous section.

The N-Body application is a program which models point-masses in three dimensional space and how they interact due to gravitational forces. There are N individual point-masses (bodies) which are divided between the parallel tasks. The N-Body application is iterative, moving forward in timesteps. In the most complete case of N-Body, for each iteration each body must compute the forces applied on it by each other body. This requires in a parallel system that each body eventually reach each parallel task. Other N-Body algorithms, such as Barnes Hut, attempt to improve on the performance by assuming

```
COVEN_MODULE load_balance
  (
   input int max_iterations,
   input int current_iteration
  )
{
        if(current_iteration != max_iterations) {
            COVEN_Barrier_Balance();
        }
}
```

Figure 5.10: Explicit Load Balancing Module

that bodies far away have no force affect on each other. With this assumption, it is not necessary for each body's forces to be computed with each other body's forces and, therefore, not all bodies will need to be transported to each parallel task.

For this example, the application was implemented in such a way that the bodies were divided into groups and those groups were passed around in a ring until each parallel task had received them. After receiving a portion of the bodies, each task computes the partial forces those bodies apply to the bodies it is responsible for. Only after every task has computed all partial forces does the application have a global solution for the current timestep. Time then advances to the next iteration.

Figure 5.10 depicts the simple Coven system module which performs the explicit DLB for this application. This module was available in Coven's module library and merely dropped in after the last module to execute in an iteration. This version of the DLB module tries to perform load balancing every iteration except the last.

### 5.6.1 Performance Without Load Balancing

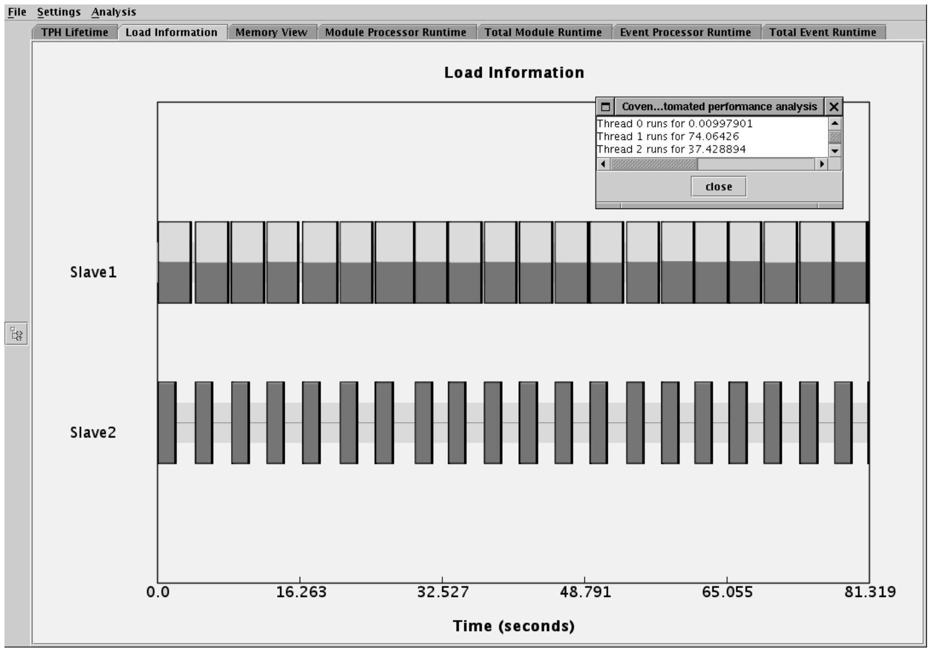
The N-Body application is perfectly balanced because there are a predefined number of calculations that are required to compute the new force applied to each body. In an effort to experiment with load imbalances with this type of application we introduce external load.

We define external load as some force other than the application which causes the application to execute with an imbalance. Examples of external load are hardware differences between parallel nodes, and load imposed due to external sources such as concurrently executing applications. While both causes will exhibit very similar behavior (with respect to Coven DLB mechanisms), we chose to study how the N-Body application performs when competing for resources with another long-running application.

For simplicity of visualization, we show the N-Body application running on only two parallel processes and for a very short number of iterations. These choices make visualizing the behavior of the application easier and it is clear how the application would perform under longer runs.

To demonstrate the need for load balancing under these conditions, the N-Body application was executed while one of the processors was heavily loaded with another application. This other application caused the Coven parallel task to only get access to the CPU about 50% of the time. Therefore, this portion ran half as fast as the other parallel task. Figure 5.11 is a screenshot of the Coven profiler which depicts this application. The first process (*Slave 1*) is the one competing for CPU. *Slave 2* can be seen to be idle for a large portion of the runtime, due to delays waiting on the other parallel task.

The rectangles represent Coven modules where the left edge depicts the start of the module and the right edge depicts when the module exited. CPU load is depicted by shading the rectangles. Notice that the rectangles for *Slave 1* are shaded half full, meaning that the modules executing under that parallel task were receiving only 50% access to the CPU. On the other hand, the modules for *Slave 2* are completely shaded meaning that this parallel task had complete access to its processor. The rectangle widths for the second parallel task are smaller and this time spends much time idling (depicted by empty space between rectangles when no modules were executing). This is due to the task having to wait for the slower, externally loaded, *Slave 1*. The dialog box overlayed on top of the screenshot confirms this, showing that the second task was executing only 37.4 seconds



```
Thread 0 runs for 0.00997901
Thread 1 runs for 74.06426
Thread 2 runs for 37.428894
```

Figure 5.11: N-Body Without Load Balancing With Constant External Load

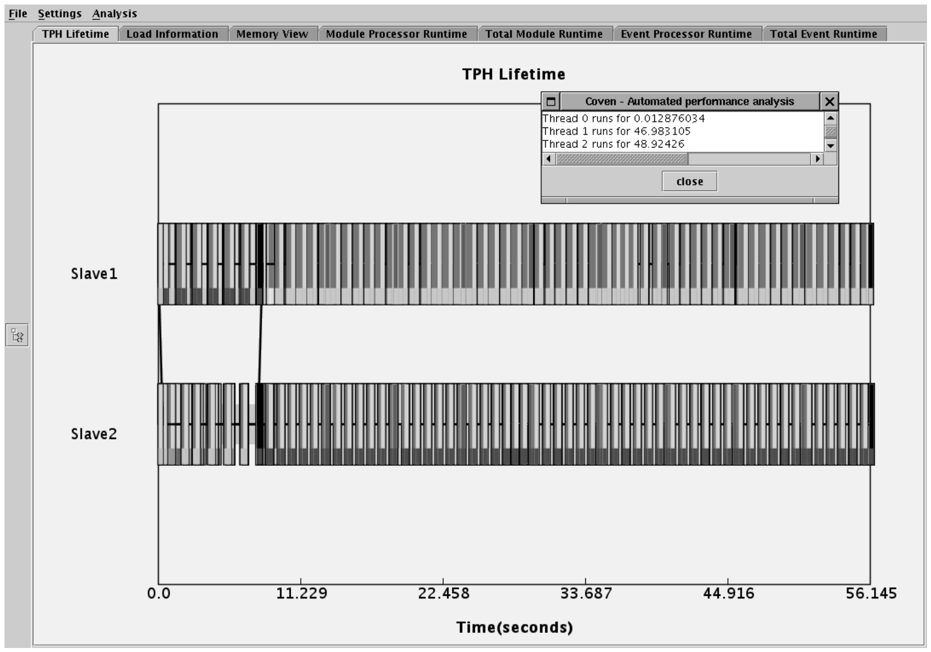
---

compared to the first task which executed for 74.1 seconds. This amounts to the second task running for only about 50% of the time that the first task was.

### 5.6.2 Performance With Load Balancing

Once again, before DLB is possible, the application must be partitioned into more pieces than there are parallel nodes. The groups of bodies are contained within TPHs which can then be moved around at will. Using a Coven scatter operation and a distribution pattern (both described in Chapter 2), the bodies were easily partitioned into three segments per processor. Coven's DLB functionality was then enabled in an effort to move a portion of the work away from the parallel task that was competing for CPU.

Figure 5.12 shows the profiler screenshot for this application. Notice that initially, the second task was idle for a portion of the work. This was because, again, the first task was competing for CPU with an external process. In the screenshot, each TPH is uniquely shaded. Notice that when compared with Figure 5.11, the rectangle widths are considerably



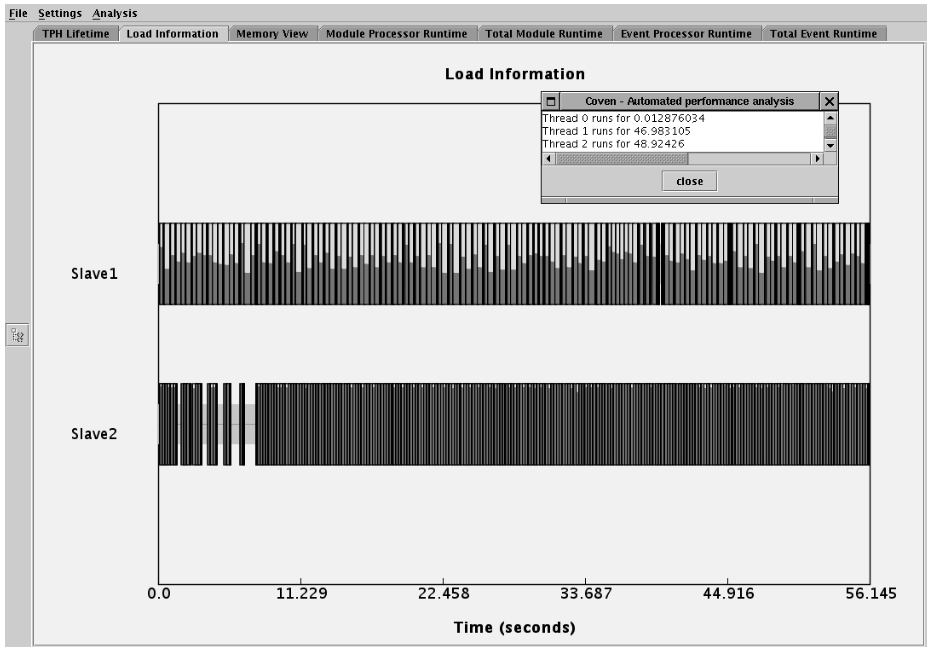
```
Thread 0 runs for 0.012876034
Thread 1 runs for 46.983105
Thread 2 runs for 48.92426
```

Figure 5.12: N-Body Explicit Load Balancing With Constant External Load

smaller. This is due to the data being partitioned into one third smaller pieces on each task and the modules then are able to process each partition about a third as fast.

The DLB module was called at around 9 seconds in the application. At this time, Coven recognized an imbalance between the two parallel tasks due to *Slave 1*'s external load and instructed *Slave 1* to send some work to *Slave 2*. A black line at around 9 seconds depicts the transfer of one TPH from *Slave 1* to *Slave 2*. Notice in Figure 5.12 the dark shaded TPH that was assigned to *Slave 1* initially is moved to *Slave 2* around 9 seconds and then resides there throughout the lifetime of the application. After the TPH is moved the first task was processing two TPHs (portions of the bodies) while the second task has four TPHs. Throughout the rest of the application's life cycle, no additional TPHs were transferred, signifying that the balance had been achieved.

Not only did the application runtime decrease by 31%, but (as seen from the dialog box overlay) the tasks reached a balance between their individual runtimes. When compared with the previous example (without DLB, Figure 5.11), *Slave 1*'s runtime went from 74.06 seconds to 46.98 seconds and *Slave 2*'s runtime went from 37.43 seconds to



```
Thread 0 runs for 0.012876034
Thread 1 runs for 46.983105
Thread 2 runs for 48.92426
```

Figure 5.13: CPU Load of Application in Figure 5.12

48.92 seconds. To visualize the varying load, Figure 5.13 is presented. Notice the shaded rectangles for *Slave 1* that depict the approximately 50% CPU utilization throughout the application.

### 5.6.3 Random External Load Example

Finally, to study the flexibility of Coven's explicit DLB algorithm we provide an example where the load placed on *both* processors varies over time. Figure 5.14 is a screenshot of this example. Notice the numerous lines representing TPHs being moved between processors as the system attempts to deal with fluctuations produced by concurrently running applications.

Figure 5.15 presents a zoomed-in version of the previous screenshot with an analysis of CPU load. Each task can be seen to be loaded differently over time as other programs compete for the processor. Finally, Figure 5.16 depicts the same zoom but with a view of how the TPHs are moving between processors due to the previously shown load.

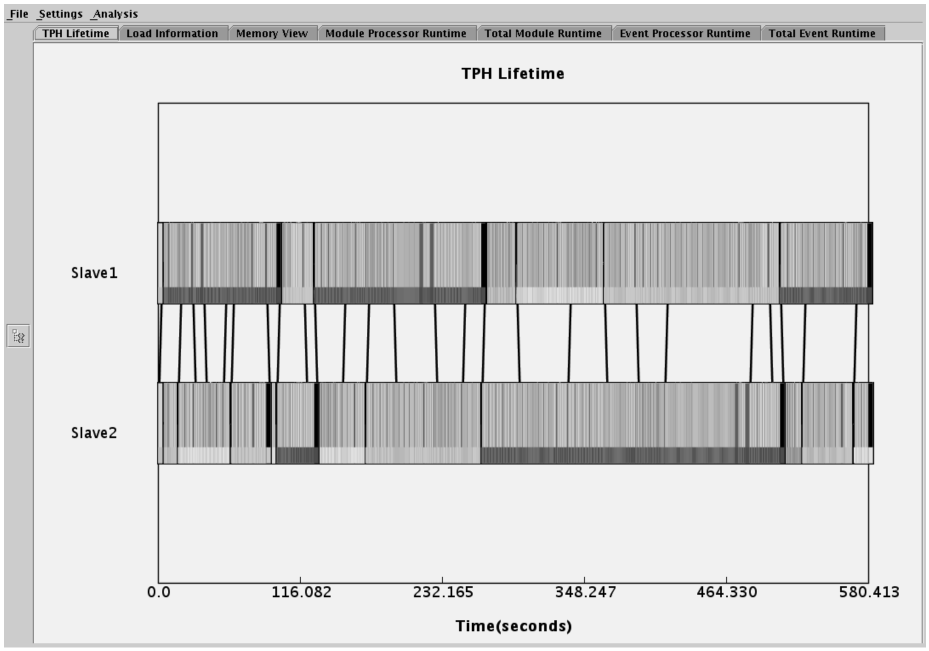


Figure 5.14: N-Body Example With Random External Load and DLB

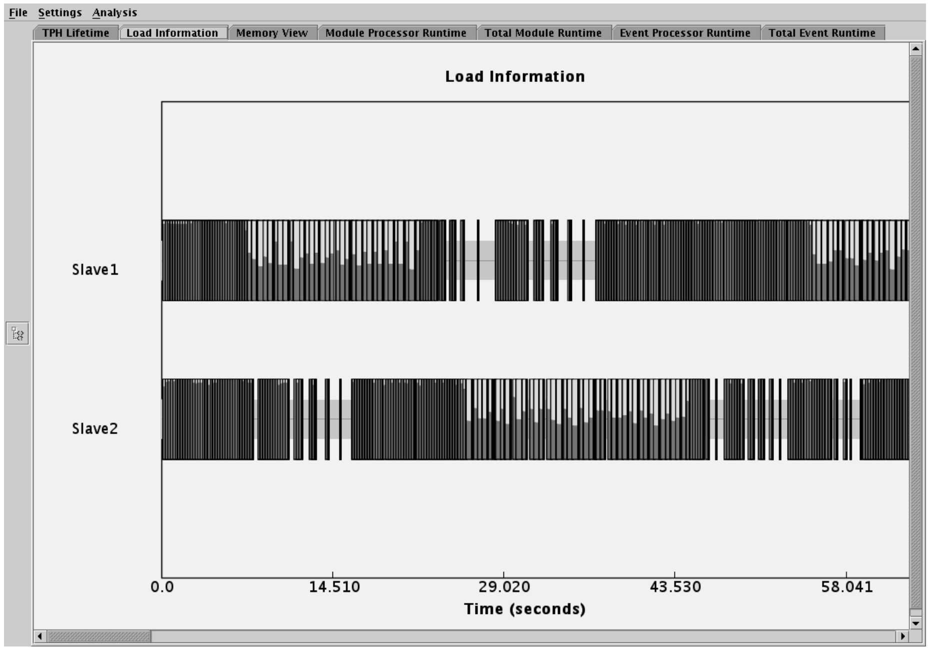


Figure 5.15: Zoomed in for CPU Load Detail of Figure 5.14

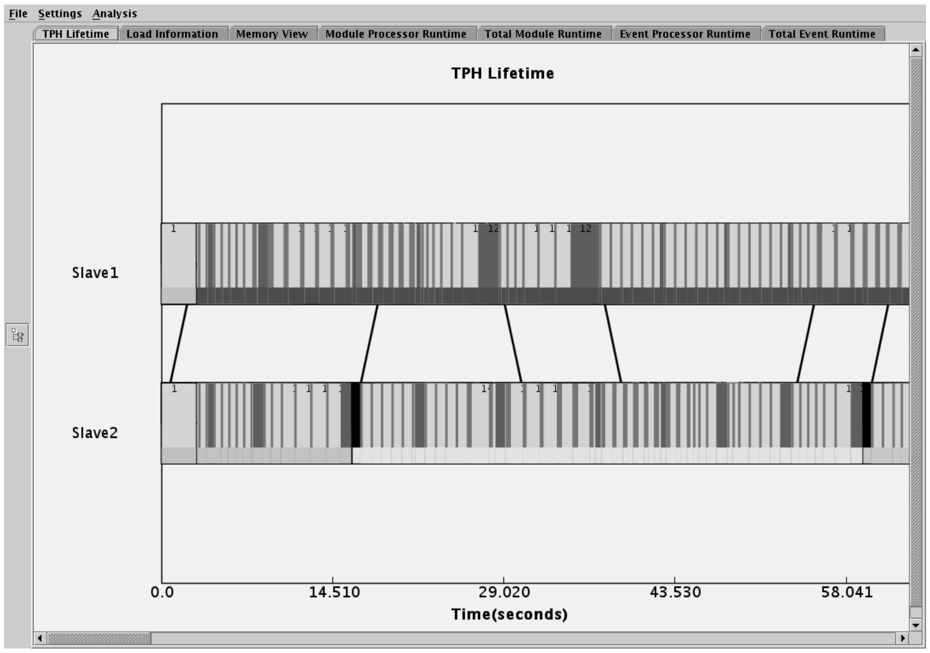


Figure 5.16: Zoomed in for Detail of Figure 5.14

This example demonstrates how Coven's explicit DLB mechanism responds dynamically to changes in the system. Here, each parallel task was competing for the CPU with other processes, at times getting no more than 50% of the processor. When a task began to run slowly, TPHs were migrated off to other tasks which were not experiencing as much load due to competition for resources. The randomness of this example helps to illustrate the versatility of this approach and demonstrates not only how Coven helps with DLB under random loads, but also makes clear how Coven would assist in DLB on heterogeneous parallel computers.

### 5.6.4 Non-Coven Performance and Complexity Comparison

A hand coded version of the N-Body, explicit DLB application was not implemented. The explicit DLB algorithm would be the same as described in Section 5.5. Throughout this thesis it has been shown that a Coven implementation performs approximately 1% to 3% slower than a hand coded version which implements the same algorithm.

As with previous hand coded versions, this version would include a great deal of code that is present in Coven's runtime engine. Besides all the previously stated reasons for using a Coven implementation over a hand coded version, it is important to reiterate that a hand coded version here would likely be tightly coupled to the N-Body implementation. This would cut down on the reusability of the explicit DLB code and subsequent applications implemented in this manner would need to decouple the DLB code from the N-Body application in order to modify it to the new application.

## 5.7 Related Work

Research in load balancing optimizations can be found in many different areas. These include parallel file systems, load balancing specific applications, environments, and libraries.

In a similar study to Nieuwpoort's [87], Blumofe and Leiserson [12] look at work stealing versus work sharing in the context of a multi-threaded environment. In work sharing, whenever a processor generates a new thread it attempts to find an underutilized processor to transfer the thread to. With work stealing, however, the initiative to seek out more work is left to the processor that is underutilized. The authors show that work sharing saturates the network with DLB operations before it is really necessary. Furthermore, in a dynamically changing environment, the early transfer of work to an initially underutilized processor may not prove useful later in the application should that processor begin to be loaded by some external force (for instance, another program contending for resources).

Scheuermann et al. [73] developed a DLB parallel file system to minimize the average wait time for requests by balancing the requests across all the disks. Data is moved at runtime from *hot* (very frequently accessed) disks to *cold* (less frequently accessed) disks in an effort to adapt to the current application being run.

Boukerche et al. [14] studied dynamic load balancing on parallel simulations using CPU-queue length data. Queue length data is compiled by sending a token from processor to processor in a virtual ring topology. Once complete, a global load balancer determines when to move which process to what destination. This is a form of process migration to achieve load balancing.

Mills et al. [70] used a cluster of workstations (COWs) and a relatively different type of load balancing to try and evenly balance a parallel eigensolver on heterogeneous and time shared machines. Their code adjusts the preconditioning phase of the eigensolver so that it preconditions for a certain length of time, rather than to some iteration. Therefore, if a processor is too loaded then it will not take up the other processors' time by preconditioning fully. This value of time is adjusting dynamically during computation and is related to the time it takes the fastest processor to complete a phase. As processors become more or less loaded in a time shared, heterogeneous environment then the preconditioning time value adjusts itself accordingly.

Zoltan [34] is a project that provides a DLB library that any application can use. The application must provide callback functions that Zoltan can call to tell it how many objects (elements, particles, etc.) are on a processor, the coordinates and/or connectivity of the objects, and the computation load of the objects. Zoltan collects this data and makes decisions about how to rebalance load but does not actually do the rebalancing itself. It can be used with any application and supports both geometric and graph-based partitioning algorithms. Another project, DRAMA [64], functions in much the same way but is limited to finite element applications. Due to this restriction, the interface is easier to use than Zoltan but obviously is useful to solve less types of problems. The goal of these projects is to provide a means so that load balancing code can be separated from application code. This makes it easier to add to many applications as the load balancing code is less tightly coupled than in most custom load balancing solutions.

In [20] dynamic load balancing strategies are studied in the context of a parallel linear system solver. DLB is achieved by redistributing rows of the matrices from overloaded processes to underloaded ones. The authors propose four distinct DLB approaches all using different types of asynchronous communication and distributed redistribution of rows. Their results showed that all approaches worked about as equally well, and DLB definitely was beneficial to improving performance in their system.

Hamdi and Lee [46] created a system for dynamic load balancing of data parallel applications. In this system, a master hands out work to slaves. This task periodically asks the slaves how loaded they are and adjusts the amount of work destined for each slave accordingly. In this way, the system dynamically adjusts to varying workloads. This research, like many other load balancing research, was conducted on time shared clusters of heterogeneous workstations.

Nieuwpoort, Keilmann, and Bal [87] developed a Java programming environment for solving divide and conquer applications on wide area networks arranged hierarchically. They studied a series of load balancing algorithms that took into account different optimizations, such as communication, idle time, WAN communication, LAN communication, and latency hiding. These algorithms all performed differently under different conditions and they outline ways to choose which would best fit each work environment.

Henrichs [48] created a new programming model in which programs were augmented with new keywords. These keywords essentially allow for the creation of virtual processes. Multiple virtual processes are scheduled on the same processor. The context for this research was metacomputing, where heterogeneous machines and supercomputers are connected together. The goal of the research was to allow for programmers to identify pieces of code that would better perform on a certain architecture and have the underlying system handle the details. For instance, if one part of the code uses a type of operation that is more optimized on a Cray, then that part can run on the Cray while, concurrently, something else is running on a Paragon. The researchers looked into load balancing schemes for this complex system but did not accomplish much.

SPEEDES [88, 89, 90] is an environment for parallel discrete event simulations. Wilson, et al. created a central load analyzer which determines load on parallel SPEEDES tasks. This operation requires user intervention to describe the “complexity factor” of each event on a scale of 1 to 10. The load analyzer then determines which (if any) tasks are too loaded and then takes care of routing their work to another task. While this is most certainly DLB, SLB was looked at in SPEEDES as well. For SLB, load data from previous runs of SPEEDES on a particular architecture was saved and, before another run, the data was statically load balanced. They found generally good results but that, of course, it didn’t account for applications where the load varies due to changes in the algorithm or in the system.

## 5.8 Summary

Load balancing problems arise in many parallel applications and an environment which intends to ease parallel program implementation and use should attempt to address this issue. We have presented two different styles of dynamic load balancing (DLB) which are easily used within Coven. The first method, random stealing, initiates when a Coven task runs out of TPHs (work) to process. When this occurs, it informs other tasks that it is idle and waits for potential work to arrive. With the second method, explicit DLB, the user inserts a DLB system module into their data flow graph. This module then analyzes TPH throughput on each task and, if the load is deemed imbalanced, instructs certain Coven tasks to offload TPHs to a less loaded Coven task.

Both of these optimizations were shown to be easy to add to an existing Coven application and improve performance over leaving the system imbalanced for the applications tested. Hand-coded versions which included DLB were shown to be complex, but (as expected) perform slightly better than Coven. This is expected in a PSE and is due to the overhead imposed by the environment on the application. This overhead was shown to be less than 10%, well within common expectations of PSE overhead.

# CHAPTER 6
# CHECKPOINTING

Checkpointing is a feature common to parallel applications, particularly large simulations, and involves saving the state of a program to disk. This has many uses such as recovering after a failure, intermediate analysis of data, or migration. While there has been much research in checkpointing, it still remains complex to implement in generic applications.

## 6.1 Introduction

The study of checkpointing often includes related topics, such as fault tolerance and task migration. Fault tolerance is the idea that an application should be able to recover from a hardware or software fault. These include compute nodes failing, being disconnected, network disconnections, and numerous potential software issues. While fault tolerance in itself has not necessarily been a primary focus for the parallel computing community, the present trends towards clusters of commodity workstations has reduced the mean-time-between-failures (MTBF) of these systems. As such, fault tolerance is becoming even more important.

The study of task migration involves the concept of stopping a task mid-execution and moving it to another processing element. There are many reasons to do this, including load balancing and freeing up a node that needs to be shut down or reallocated to some other purpose.

These related topics often include application checkpointing at some point in their implementation. For fault tolerance, often periodic checkpoints are performed (either to disk or a large memory) and if a node is found to have failed, the work that node was doing is merely spawned off to a backup reserve node. Similarly, task migration generally involves saving state of an application to disk, transferring this state to another processing element, and then restarting the application on this new processing element.

Broadly speaking, there are three main forms of checkpointing: non-transparent, semi-transparent, and transparent. Non-transparent checkpointing means that the checkpointing facilities are entirely evident to the programmer. Generally speaking, this means that the application handles all issues regarding marking data for being saved, saving state, recovering from a fault, etc. This approach is obviously difficult to implement and often tightly coupled with a particular application. Moreover, non-transparent (obtrusive) checkpointing is usually done by hand, without the assistance of any helper environment or library. This is specifically the case that we are trying to avoid with Coven, as do other checkpoint environments and libraries.

Semi-transparent checkpointing relies on hiding most of the checkpointing difficulty behind an API and/or an external checkpointer helper application. With this approach, the user must generally specify information about their program through source code modifications. These include tagging which data must be saved, what points in the computation a checkpoint is possible, and other similar constraints.

Transparent checkpointing refers to an approach where the checkpointing facility is able to checkpoint/restart code without any assistance from the programmer. This generally means it requires no source code modifications or tips as to when a checkpoint is possible. Transparent checkpointing is the most desirable, as it is usable by the widest range of individuals for the least effort. One particular problem with transparent checkpointing is that it is usually implemented using assistance from the operating system and is often tightly coupled with that architecture. Transparent checkpointing, since it has no understanding about the application being checkpointed, requires that the entire system state be saved so that it can be recovered. This usually results in massive checkpoint files, having register dumps, snapshots of the heap, local and global variables, etc. While easiest to use, its usefulness may be limited to homogenous computing systems and limit portability because of the tight coupling to the operating system. Transparent checkpointing is often called system-level checkpointing or core-image checkpointing.

Checkpointing has been a very active topic in the field on parallel and distributed computing over the last several decades (see Section 6.5 for related work). The usefulness of checkpointing is likely to increase, as larger and larger machines are being constructed with very low MTBF. For example, the 8000 node Google cluster claims to have a MTBF of merely 36 hours. Therefore, a PSE designed to aid implementation and expand the user base of parallel computers should aim to provide checkpoint and recovery features.

This chapter presents Coven's checkpoint and recovery feature. We explore how the Coven model of computation, particularly the separation of state and code, facilitates checkpointing. This property makes it possible for Coven to keep track of all state in the system. Coven then is able to signal a global checkpoint and write each TPH to disk. Coven's computational model further makes it trivial to reload these TPHs on recovery, insert them into queues, and begin processing again.

We study the use of checkpointing in a complex fluid dynamics application and show its effectiveness at recovery after failure. We show that unlike most existing checkpointing systems, Coven's model makes it transparent for the user to tag which data must be checkpointed. This feature makes checkpointing unobtrusive to the user and requires barely any application code modifications. We show that the checkpointing feature of Coven is a logical stepping stone to task migration. Task migration is explored and we study how the Coven model can further simplify this powerful optimization.

## 6.2 Approach

Section 2.4.5 detailed a feature of Coven called the *garbage collector*. This runtime engine component analyzes the data flow graph and TPHs to determine when a data value will be last used. It then frees the memory associated with that data automatically. This feature plays an important roll in making checkpointing within Coven as transparent as the system-level checkpoints, but save the minimum amount of application data like in manual checkpointing systems. The following facts facilitate checkpointing in Coven:

- All application data in Coven is maintained within TPHs
- Coven knows the location of all TPHs at all times
- Coven's garbage collector frees data at the point where it is last used (no down-stream modules will need it)

There are many potential problems with checkpointing running applications. These include, but are not limited to, the use of static variables, open file pointers, and remote procedure call (RPC) handles. While some related checkpointing suites attempt to address aspects of these (in particular, open file pointers by saving the file name and file offset), the general approach is to disallow these features or ignore them when it comes to checkpointing. This certainly augments the checkpoint recovery operation of some applications as they would then require startup code to move values back in static memory, open files, and restart RPC handles. While Coven's checkpoint/recovery system could be augmented to allow for these features, we chose to focus on the core functionality at this time and leave these issues for future work.

A user initiates a checkpoint in Coven by placing the Checkpoint system-module into their data flow graph at the point where they wish checkpointing to occur. Figure 6.1 depicts this module. The user can easily modify this module to checkpoint at other intervals, such as periodically based on a wall clock.

The `COVEN_Checkpoint` method is implemented similarly to the explicit dynamic load balancing method from Section 5.6, Figure 5.10. A global synchronization is required to perform the checkpoint in Coven. While this is not necessarily required by some checkpoint implementations, for many reasons it is with Coven. The most obvious reason that many checkpoint systems require a global synchronization is so that all tasks agree they have reached the same point before checkpointing. One example of this is a race

```
COVEN_MODULE checkpoint
  (
   input int max_iterations,
   input int current_iteration,
   input int checkpoint_interval
  )
{
        if(current_iteration != max_iterations &&
           current_iteration % checkpoint_interval == 0)
        {
            COVEN_Checkpoint();
        }
}
```

Figure 6.1: Checkpoint Module

condition that can occur if two tasks A and B are both checkpointing and communicating. If A reaches the checkpoint before B, checkpoints, reloops (in the case of an iterative application), and transfers more data to B all before B reaches its checkpoint, then task B's checkpoint may contain data from a communication with A before B had itself relooped. Often this can be addressed through the use of clever logic. The way Coven handles messages makes this very difficult to get around without a global synchronization. This global synchronization makes checkpointing within Coven expensive (roughly between 10ms and 10 seconds depending on problem size and number of parallel tasks). This expense varies with the amount of data to be checkpointed, due to disk write times, as well as the number of parallel tasks, due to the global synchronization. The following section details some empirical results seen using Coven's checkpointing feature and the overhead incurred.

Figure 6.2 is a diagram explaining the global synchronization operation of Coven's checkpoint feature. In this example, there are $N$ tasks which must synchronize to be certain all TPHs are accounted for. Once each task has reached the barrier synchronization, Coven knows that no other TPHs are being transferred and that the location of all TPHs is now known. These TPHs may reside in input or output queues and are written to the checkpoint

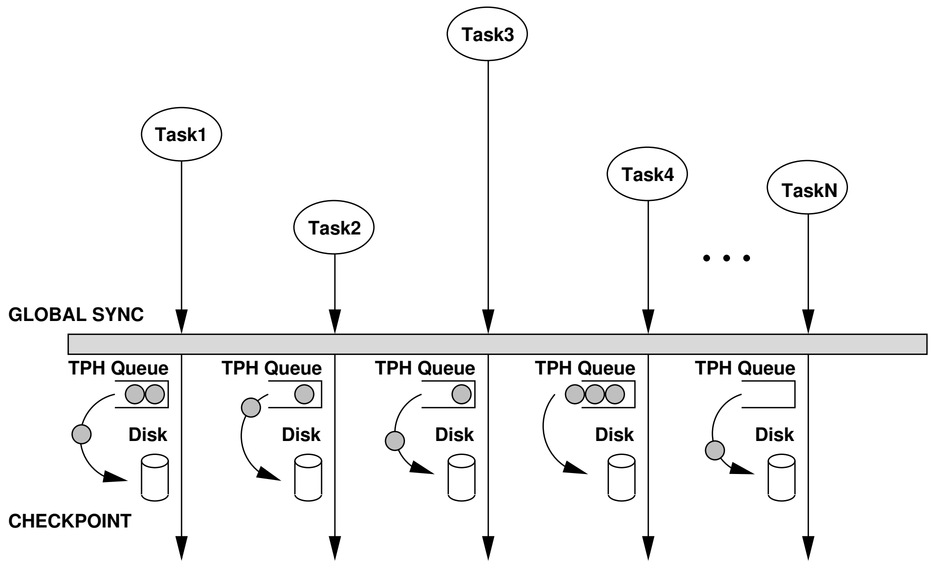


Figure 6.2: Checkpoint Global Synchronization

file so that they can be reloaded in the same order. Each TPH is then written to disk and the task continues.

Coven is equipped to read checkpoint files and restart an application at that point. This can either be done at program execution by supplying a flag and a link to the checkpoint files to load, or at runtime. If initiated at runtime, Coven abandons all TPHs in the system by freeing all memory associated with them and then loads the checkpoint files. All modules are dynamically loaded, and since all program state is contained within TPHs, Coven merely injects the TPHs back into their queues in the appropriate order. Each Coven task then recognizes the arrival of new TPHs (work) and begins processing them. TPHs are marked with their progress through the data flow graph and are reinserted into that graph where they left off.

It is important to note that the checkpoint module (Figure 6.1) only synchronizes every `checkpoint_interval` iteration. If the periodicity of checkpointing is too small, then the overhead due to checkpointing will become a greater portion of the application runtime. This requires a careful balance, saving state often enough so that too much work does not have to be redone but at the same time not decreasing application performance too much due to checkpointing too frequently.

In the next section we describe the application chosen to test checkpointing within Coven. Then we study how this application performs and demonstrate fault recovery using checkpointing.

## 6.3 Finite Difference Application

A finite difference application was chosen for the checkpoint/recovery case study. This application was selected due to its use of the MPI processor topology functionality, something not previously demonstrated in Coven. For this application, a steel plate is set to an initial temperature and then at the start time another temperature is applied to one side of the plate. The simulation iterates as the heat transfers into the plate, calculating values of grid points based on the value of neighbor points on the previous iteration. Figure 6.3 is a OpenGL visualization agent which plots the values of the computation. The two-dimensional plate is shaded to represent the temperature and the third-dimension shows the temperature value as height.

The implementation of this simulation requires the plate be gridded and then the grid partitioned between the parallel tasks. The plate is represented by a collection of grid points, such as $512 \times 512$. Data decomposition is determined by using a MPI process topology where the number of processors in the $X$ and $Y$ dimensions are specified. The application is designed so that it can only operate on power of two size number of processors as well as a power of two number of elements in each grid partition.

The initialization phase of the simulation determines the number of iterations which will be required. This value can be calculated based on the chosen size of the grid. Iteratively, each parallel process exchanges boundary regions with its neighbors and then calculates the updated values of its grid points based on the values of the previous iteration. Since the number of iterations for this application vary based on the grid size and the grid

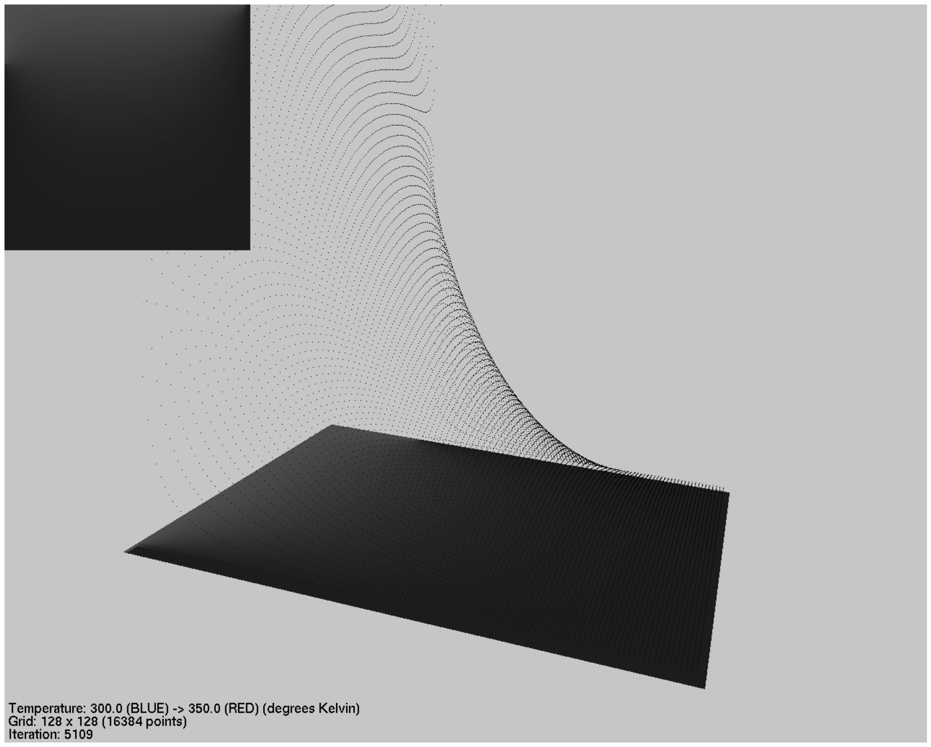

Temperature: 300.0 (BLUE) -> 350.0 (RED) (degrees Kelvin)
Grid: 128 x 128 (16384 points)
Iteration: 5109

Figure 6.3: Finite Difference OpenGL Visualization

size affects the data size it makes comparing parallel runs difficult except when the grid size is held constant.

A hand-coded MPI version of this application was provided by Dr. Richard S. Miller, Mechanical Engineering, Clemson University. This version was then adapted into Coven modules. With large grid sizes, the number of iterations required to perform the simulation can be more than 100,000 iterations. Maintaining the data in memory on our working cluster with 1GB of RAM requires pairing a large grid with large number of parallel tasks. As such, the time each task executes each module is relatively small when compared with the number of iterations which are performed. Many modules in this example run for only a few milliseconds.

Thus far, we have not presented any Coven applications with this fine a granularity. As expected, Coven imposes an overhead on the application with the many function calls, TPH manipulations, and queue operations. This performance degradation was between 5% and 15% and varied with the grid granularity.

## 6.4 Checkpoint/Recovery Case Study

The finite difference application described in the previous section was used as a sample program to demonstrate application checkpoint and recovery within Coven. Since Coven uses a global synchronization before writing all TPHs to disk, the time it takes to perform a checkpoint varies depending on the number of parallel tasks and the size of the data grid.

Experiments were run on this application using varying grid sizes and processors. For the larger grids, the memory requirements were large enough that a small number of processors was impossible to use without using swap space or out of core computation. Checkpoint cost was measured as compared to not using the module at all. Therefore, included in the cost was the module call, global synchronization, writing all TPHs to disk, queue operations, and releasing the TPHs to resume processing. It was seen that checkpoint cost varied between 10ms and 10 seconds, a large range. The faster times resulted from very small grid sizes (such as $16 \times 16$ grid points) on only 2 parallel tasks. The longer times were seen for much larger grid sizes (such as $512 \times 512$ grid points) and for 8 or 16 parallel tasks. While this checkpoint time seems large, considering it in relation to the total program runtime is important. The time to perform one checkpoint was found to be between 1% and 2% of total application runtime for this application. Comparatively, this is small. This number does not account for the fact that generally multiple checkpoints need to be performed in long running applications to effectively cut down on the cost to restart.

In an effort to demonstrate Coven's checkpoint and recovery feature the following experiment was performed. The finite difference application was run without checkpointing and with two checkpoints using a grid size of $512 \times 512$ and using eight processors. Figure 6.4 presents these results as the first two bars. The checkpoint version ran about

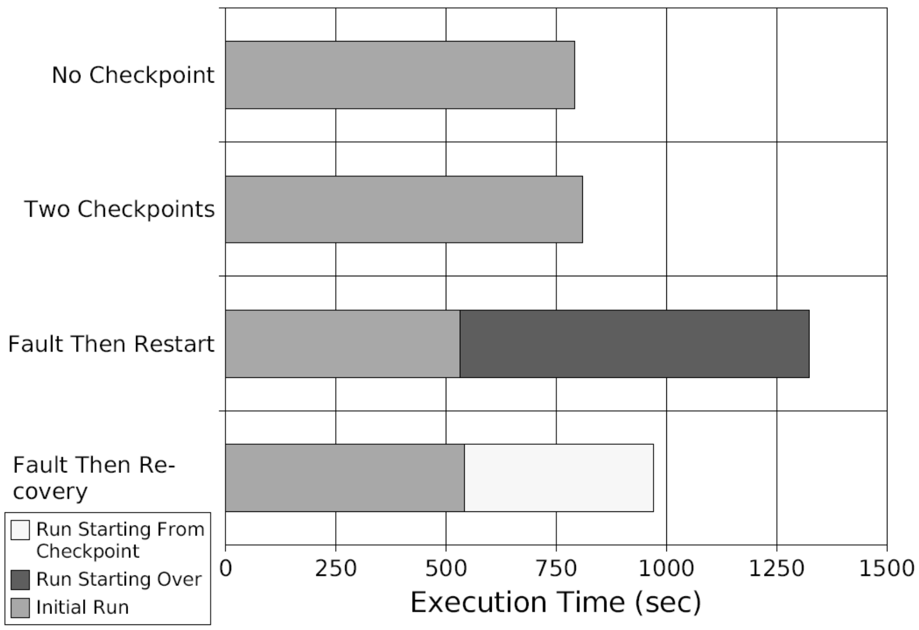


Figure 6.4: Checkpoint/Recovery Example

18 seconds slower than the version without a checkpoint. Each checkpoint operation took about 9 seconds for this data size.

For this data size, the application needs $104,857$ iterations to complete. The checkpoints were run every $50,000$ iterations (so at $50,000$ and $100,000$ iterations). For the next test, we caused the checkpointing version to fault around iteration $70,000$. The third bar in Figure 6.4 presents the runtime of this operation (as the first rectangle) combined with the cost of rerunning the entire application. The resulting time is 1324.83 seconds and depicts the total runtime that would have occurred if the application had faulted at $70,000$ iterations and was rerun from the start, without checkpointing.

For the final test we provide an example using checkpointing. Once again, at $70,000$ iterations a fault occurred but a checkpoint had been performed at $50,000$ iterations. The application was restarted from this point, having lost $20,000$ iterations and then allowed to run to completion. This results in a total runtime of 971.33 seconds.

There is certainly a tradeoff between the time it takes to perform a checkpoint and the time a user is willing to lose due to a restart. For example, if a checkpoint were being performed every $1,000$ iterations in our example, we would have lost a mere $1,000$ iterations and restarted from iteration $69,000$. However, the cost involved with this many checkpoints would have, in this applications case, caused the program runtime to effectively double.

Something commonly seen in checkpoint algorithms is the ability to checkpoint based on time, rather than iteration count. With this approach, a checkpoint can be made (for example) every hour. Then, at worst case only an hour's worth of work would be lost if a recovery was required.

### 6.4.1 Non-Coven Performance and Complexity Comparison

The original MPI finite difference application was augmented to do checkpointing. For this application, checkpointing was very easy as there are only a few very large arrays and a handful of global variables.

At the end of an iteration, each parallel task determines if it is time to checkpoint using the same method as was used in the Coven version of the application. A simple `MPI_Barrier` operation proceeds a function which handles the checkpointing. This function opens separate checkpoint files and dumps the values of the arrays and global variables and then the simulation continues.

For a grid size of $512 \times 512$ and running on 8 processors the Coven checkpoint operation consumed roughly 9 seconds. This version consumed a mere 1.5 seconds. While this is considerably faster, there are several things to consider here. The implementation of this is simplified here due to the small number of large arrays, which makes clear precisely what needs to be saved. The implementation is also coupled to the finite difference application and would require changes to be ported to new applications. Furthermore, as changes occur in the finite difference program, they would have to be reflected in the checkpoint code.

Coven's implementation, while slower due to the overhead imposed by TPH manipulation, numerous function calls, and several queue operations, is more portable to other applications. In Coven, it would be trivial to checkpoint TPHs that had been virtualized, where a hand-coded version would require its own queue or holding area to hold data until synchronized. As with the other hand-coded applications presented, it is faster than the Coven implementation but is coupled to the application which reduces its reusability.

The following section examines some of the related work in the field of checkpointing and the related topics of fault tolerance and task migration.

## 6.5 Related Work

There are a number of core-image checkpointing systems which include CoCheck [79], CLIP [19] and many others. Generally speaking, each of these environments makes checkpointing both transparent and easy but are limited to a particular operating environment and have large checkpoint files. The large checkpoint files have several limitations. Not only do they require a long time to collect and write to storage, they similarly take a long time to migrate and restart. Specifically, this can hinder task migration (through checkpointing) in order to achieve a better load balance. This is particularly true when attempting to load balance fine grain applications.

Coven's checkpointing features, described in the following sections of this chapter, are not tied to a specific operating system. While not transparent, Coven's implementation is not particularly obtrusive. This is achieved through Coven's ability to understand what data is being passed around the system (remember that all data is represented in TPHs and Coven has a total knowledge of all data in all TPHs). Rather than focusing on the related work in core-image checkpointing systems, we now look at some semi-transparent environments.

Li [62] presented a checkpoint algorithm for the DEC Firefly shared memory computer. The algorithm works by employing a separately running program which, upon checkpoint initialization, copies the memory space to be checkpointed into a buffer area and then writes the checkpoint to disk. Once the data is buffered, the application breaks free from the checkpoint barrier and then, concurrently, the checkpoint program performs its task. The algorithm relies heavily on heap and page manipulation, most of which is specialized code for the Firefly but could likely be ported to other similar platforms. Additionally, the algorithm only functions on a shared memory architecture where it has complete knowledge and view of the entire memory of the application to be checkpointed. Since this approach uses an externally running checkpoint program that hooks into an application, it is transparent to use in any application.

ParaSol [66] is a parallel discrete event (PDE) simulator (mentioned in Chapter 3). In [65], a state-saving algorithm for Parasol is presented. In this algorithm, parallel Parasol threads work on the simulation but due to the nature of the problem domain it is common for causality errors to occur. Causality errors occur when a thread at simulation time $t_n$ receives a late transaction with timestamp $t_p < t_n$. The computation must then be rolled back and restarted from a previously saved checkpoint such that the new time is less than $t_p$. Threads must then checkpoint their state periodically and the overhead involved in this operation is shown to be as high as 18% on a SPARC 5 cluster and 29% on an Intel Paragon. While costly, this type of checkpointing is required by the PDE simulator. Parasol provides the user with transparent checkpoint and recovery mechanism. This is facilitated by the fact that Parasol can be only used to implement PDEs and, due to that constraint, the system is aware of a great deal about the underlying application.

Bronevetsky [17, 16] introduces a software layer which assists in checkpointing MPI applications. This layer resides between the MPI layer and the application, making it usable by any application and any MPI implementation. Rather than use an expensive system-level checkpoint approach, this system has users add source code into their application which assists the checkpointer at runtime. A pre-compiler analyzes the code as well and inserts additional code to describe the state that needs to be saved. This helps the user in that the added coding complexity is relatively minimal, requiring just a few library calls in proper places. One advantage of Bronevetsky's approach is that it is able to checkpoint simple MPI point-to-point communication (something commonly available in similar systems) but can also handle complex MPI collective operations.

Starfish [4] is another MPI checkpointing project which provides a user level API assisting the user in controlling checkpoint and recovery. Starfish does a simplistic checkpoint for trivial MPI applications or, through the API, sends messages to the parallel tasks instructing them to handle their own checkpointing at appropriate times. While for simple applications Starfish may be sufficient, more complex applications are given little assistance except for being notified when the system has reached a checkpoint and that it should be now performed.

MPICH-V [13] is an adaptation of the MPI Chameleon implementation that focuses on providing support for volatile hardware. The authors note that with ever decreasing mean-time-between-failure (MTBF), MPI applications more than ever need to be able to handle hardware and software failures. These failures make the parallel nodes *volatile*, by their definition. MPICH-V has facilities for redundancy and checkpoint/restart including task cloning and migration. MPICH-V uses a distributed logging feature where messages are logged into a memory buffer and remain there until disk-based checkpoint occurs. This has the usual drawbacks of having difficulties handling very communication intensive applications and, potentially, scalability concerns. While MPICH-V appears to transparently deal with fault tolerance issues with regard to the MPI portion of a parallel application, it is unclear how application data is stored. Presumably, this is left up to the user to address or is deemed unnecessary, assuming that the application can regenerate its data from what it is sending in messages. While this may be true for some applications, many others would require additional checkpoint assistance, such as system-level checkpointing of memory.

Dynamite [53] is a project for PVM that adds a dynamic load balancing feature through process migration. The migration is essentially transparent to the user and is implemented in user space. A set of functions are provided that users can call in their application to force a task to be migrated to another processor. Much like the PVM daemon which runs concurrently with each PVM task, Dynamite uses a monitoring task which interfaces with the PVM daemon on each node. This monitoring task gathers statistical information and makes decisions about load balancing, though how this decision is made is not entirely clear. One use of process migration in this form is to free nodes that need to be taken down for service or expand the computation to include more newly available nodes. Presumably, the authors are implying each parallel node is overloaded with PVM tasks. Dynamite is implemented for a specific version of PVM, Linux or Solaris, libc, and glibc. As new versions of these systems come out, the authors note that Dynamite must be ported. Additionally, considering MPI has become considerably more widely used than PVM, Dynamite would likely need porting to MPI to be widely usable.

CUMULVS [58] is an infrastructure for parallel scientific programs and supports checkpoint and recovery operations. CUMULVS achieves this by making the user identify which data must be saved to achieve a consistent checkpoint. Additionally, the user marks places in his code (through source modifications) where checkpoints may occur. CUMULVS requires that no messages be currently in transit, having all been resolved. Since the user chooses what data must be saved, CUMULVS is able to write checkpoint files which are considerably smaller than system-level checkpoints. With each segment of data that is saved, information regarding its type, name, storage allocation, and decomposition is saved as well. This makes it possible to translate it from one system to another, which allows CUMULVS to checkpoint, migrate, and restart in a heterogeneous computing environment.

## 6.6 Summary

Checkpointing was implemented in Coven as an additional feature that is both common and essential in long running parallel applications. Since all user data is encapsulated within

TPHs and Coven has complete knowledge of the way that data is organized, Coven's checkpointing is practically transparent to the user. Furthermore, due to the garbage collector feature of Coven, only data that will be needed later in the computation is saved during a checkpoint. These aspects combine to make checkpointing in Coven unobtrusive and produce minimal size checkpoint data files.

While checkpointing has been studied for many years, the solutions generally are at either end of a spectrum ranging from simple (in terms of program augmentation to use) but costly (in terms of checkpoint file size) to obtrusive but inexpensive. Due to the nature of the Coven programming model and the way in which data is encapsulated, Coven's checkpoint and recovery feature possesses the benefits of both approaches with little of the drawbacks.

The checkpointing feature is a stepping stone to other optimizations and features which could be implemented in Coven. For instance, migration is a common companion to checkpointing and involves moving an application from one set of machines to another. The ability to halt an application, save its state, and recover from that point is an integral part of migration. It should be possible to easily move from a set of machines with $N$ processors to a set of machines with the same number of processors.

A more complex, but equally interesting operation involves migrating to a parallel computer with a *different* number of processors than the application was checkpointed on. Due to the structure of TPHs and the scatter/gather functionality, it should be possible to take $N$ TPHs and combine them into $M$ (where $M < N$) larger TPHs in order to run $M$ processors. Similarly, each TPH should be able to be split into $Q$ (where $Q > N$) smaller TPHs in order to run on $Q$ processors. This would allow a Coven application to shrink and grow as it was migrated to different size parallel computers. This may additionally have importance in systems where resources are constantly changing and new nodes are rapidly becoming available while others are disappearing (such as Condor). While migration support is not explicitly provided by Coven at this time, checkpointing plays an important first step in that feature and this chapter has presented the implementation and demonstration of its usefulness in an application.

# CHAPTER 7
# CONCLUSION

Generally speaking, parallel programming is still difficult for application domain specialists who are often skilled at sequential Fortran programming. Not only is parallel implementation daunting, but almost as important is parallel application performance tuning. Users have long sought to properly utilize computational resources and this is unlikely to change in the near future.

A PSE for high performance computing needs to assist and ease parallel programming. This can occur through many means including programming models and languages, easing application porting such as by wrapping legacy code, and providing stock reusable parallel computation “kernels.” While many PSEs address or aim to address these issues, we believe a HPC PSE must also assist in performance optimization as well as provide important features. These optimizations and features should be designed and built so that they are reusable by a wide range of applications using the environment.

The Coven PSE aims to ease parallel programming and performance optimization. Through a unique method of encapsulating the data passed between modules, the environment is able to maintain knowledge about the data and how the modules intend to use the data. This allows Coven to garner knowledge about the application it is executing thereby being able to ease the programming load and complexity to the user.

We have presented the Coven model of computation for parallel application environments. The benefits of the model have been detailed and an example environment which employs the model, the Coven PSE, was implemented and presented. Due to the way the Coven model causes an application to expose its structure to the environment, many features were able to be incorporated into the PSE. We have shown through three experiments that advanced parallel computing optimizations and features can be incorporated in the environment and this process is facilitated by the Coven model.

The first two experiments studied the optimizations of multi-threading and dynamic load balancing. These optimizations are usable by Coven applications with little effort on the part of the application programmer. Both optimizations are important to the parallel computing field and are found in many real applications.

Chapter 4 presented Coven's multi-threading optimization, studied two applications which benefit from multi-threading, and analyzed the performance improvement seen due to multi-threading. In Coven, multiple threads of control are created and Coven schedules tasks to execute concurrently within those threads. Data passes seamlessly between threads and modules — the user only has to specify which thread each module should execute within. All issues of thread creation, communication, and use of shared memory between threads are handled by Coven for the user. Implementing multi-threading in an existing parallel application by hand is considered complex and Chapter 4 studied the difficulties involved in this process. Using Coven, multi-threading is trivial as the complexity is encapsulated in the environment. The Coven computation model enables multi-threading through its strict separation between application state and code. This enables Coven to freely schedule tasks to execute within separate threads of control while transporting the application data between threads in the system.

Chapter 5 presented two different dynamic load balancing optimizations which were incorporated within Coven. The two DLB optimization algorithms are "random stealing" and "explicit load balancing" and are usable in different classes of applications. Random stealing is most applicable in applications which are naturally parallel and non-iterative. The second algorithm, explicit load balancing, uses real time data throughput analysis to determine load imbalances and is most useful in iterative applications. Each algorithm was presented and studied using different types of applications. Performance results were analyzed and both algorithms were shown to improve the overall utilization of the system as well as cut down on application runtime. The Coven model establishes a separation between the state of an application and the code. This fact makes it possible to seamlessly move application data between parallel processors to attempt to create a more balanced workload. Furthermore, since the Coven model causes an application to expose its structure to the environment, the Coven PSE is able to transparently serialize and move data from one node to another without any user intervention.

Finally, in Chapter 6, we presented a checkpoint and recovery feature of Coven which is incorporated in the environment and usable by any Coven application. By simply inserting a pre-built system model into a user's dataflow graph, an existing Coven application is able to checkpoint. Unlike conventional checkpointing schemes which require a programmer to explicitly tag data that needs to be saved, the Coven model allows the environment to inherently know each piece of data that needs to be saved. Saving an application's state to disk is trivial as the Coven model enables the environment to know where all application data is at all times and exactly how to save it. Furthermore, recovery after fault is easy because the application's state can be simply reloaded and the system restarted where it left off. The separation of state and application code as well as a built-in work queuing system simplifies this.

## 7.1 Contributions

Throughout this work, our contributions are:

- the Coven model of computation for parallel applications,
- the implementation of a PSE that employs this model,
- the implementation of three built-in optimizations and features which are enabled by the model,
- and the study and analysis of these optimizations.

The Coven model is unique in the way it separates an application's state from the code. This was shown to enable an implementing environment to understand details about the application's structure without the application having to explicitly express it. Programming in an environment which implements the Coven model is easy as existing legacy code can be readily inserted as-is into Coven modules. Furthermore, once the environment has been made aware of the application's structure, it is possible to incorporate many common features, facilities, and optimizations directly within the environment. This in turn simplifies implementing applications using the environment as many common tasks are provided transparently and the programmer does not have to implement them in each application they write.

The Coven PSE is a working environment which has been used to implement a wide range of applications. It includes graphical tools for application building as well as performance / trace analysis. Many mundane and common place operations from parallel computing have been encapsulated within the Coven PSE. These include memory management, garbage collection, shared memory utilization, data transportation, application profiling, task virtualization, data partitioning, and data distribution. Each of these features are available to all Coven applications and make application building simpler as the programmer can focus more on the science and less on the tasks which make the application run efficiently on a parallel system.

The optimizations presented are well understood and common in many parallel computing applications. While some are present in related environments, we know of no environment that incorporates this many optimizations. Furthermore, the Coven model of computation is unique and enables these optimizations to be implemented elegantly.

## 7.2 Future Work

There are many future research directions within Coven which may be addressed. Up to this point, Coven has been used as a research tool and has no real end-users. Users would prove invaluable in demonstrating the implementation of real applications using the environment. Additionally, the recent widespread acceptance of the CCA component model and Grid computing mean that future Coven focus needs to look at these technologies.

Further optimizations are likely facilitated with the Coven model. Of particular interest is task migration through the use of checkpoint and recovery. Coven provides facilities to merge and split TPHs and it is possible that these could be used to migrate transparently between machines with different numbers of processing nodes. The performance profiler may prove to be useful in automatically recognizing performance problems and suggesting appropriate optimizations. These include data granularity size problems, memory leaks, and issues with module granularity. Already we have begun to demonstrate the ability of the profiler to recognize opportunities for multi-threading and dynamic load balancing. A potential future feature might be the ability of the profiler to not only recognize the problem, but automate fixing of it. Similarly, using a benchmark suite the profiler may be able to tune Coven's runtime engine for a particular platform and a particular application.

Regardless of the future directions taken in Coven, we feel the contributions have been made. In particular, we have demonstrated that through a unique model of parallel computation and a centralized data encapsulation technology it is possible to abstract the complexity of implementing three common parallel computing optimizations and features. Data flow programming through the use of parallel components has been popular for many years and component programming is beginning to gain even more acceptance. Many systems do little more than provide the “glue” between parallel components and modules. By having the programmer expose his application's structure transparently by employing a new computational model, the Coven PSE is able to optimize that application to perform better. This is a step beyond existing PSE work and makes the approach presented in this thesis unique.

## BIBLIOGRAPHY

[1] Common component architecture forum. http://www.acl.lanl.gov/cca-forum, 2001.

[2] The mandelbrot set. http://www.students.tut.fi/ warp/Mandelbrot/, 2004.

[3] MPIPovray: Distribute povray using MPI message passing. http://www.verrall.demon.co.uk/mpipov/, 2004.

[4] Adnan Agbaria and Roy Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 31. IEEE Computer Society, 1999.

[5] Gabrielle Allen, Werner Benger, Tom Goodale, Hans-Christian Hege, Gerd Lanfermann, Andr Merzky, Thomas Radke, Edward Seidel, and John Shalf. The cactus code: A problem solving environment for the grid. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, pages 253–260, August 2000.

[6] Gabrielle Allen, Tom Goodale, Gerd Lanfermann, Thomas Radke, Edward Seidel, Werner Benger, Hans-Christian Hege, Andre Merzky, Joan Mass, and John Shalf. Solving Einstein's equations on supercomputers. *IEEE Computer*, pages 52–58, 1999.

[7] Anthony Chan and William Gropp and Ewing Lusk. User's Guide for MPE: Extensions for MPI Programs. http://www-unix.mcs.anl.gov/mpi/mpich/docs/mpeman/mpeman.htm.

[8] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the Eighth International Conference on High Performance Distributed Computing*, pages 115–124, August 1999.

[9] Environmental Molecular Sciences Laboratory at Pacific Northwest National Laboratory (PNNL). ParSoft: Parallel computing libraries and tools software. http://www.emsl.pnl.gov/docs/parsoft/.

[10] Felipe Bertrand and Randall Bramley. DCA: a distributed CCA framework based on MPI. In *Ninth IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 80–89, Santa Fe, New Mexico, April 2004.

[11] G. D. Black, K. L. Schuchardt, D. K. Gracio, and B. Palmer. The extensible computational chemistry environment: A problem solving environment for high performance theoretical chemistry. In *Computational Science - ICCS*, pages 122–131, 2003.

[12] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[13] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. Mpich-v: toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18. IEEE Computer Society Press, 2002.

[14] Azzedine Boukerche and Sajal K. Das. Dynamic load balancing strategies for conservative parallel simulations. In *Proceedings of the eleventh workshop on Parallel and distributed simulation*, pages 20–28. IEEE Computer Society, 1997.

[15] Randall Bramley, Kenneth Chiu, Shridhar Diwan, Dennis Gannon, Madhusudhan Govindaraju, Nirmal Mukhi, Benjamin Temko, and Madhuri Yechuri. A component based services architecture for building distributed applications. In *Ninth IEEE International Symposium on High Performance and Distributed Computing*, pages 51–, 2000.

[16] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of mpi programs. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94. ACM Press, 2003.

[17] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Collective operations in application-level fault-tolerant mpi. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 234–243. ACM Press, 2003.

[18] P.H. Carns, W.B Ligon III, R.B. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000.

[19] Yuqun Chen, James S. Plank, and Kai Li. Clip: a checkpointing tool for message-passing parallel programs. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–11. ACM Press, 1997.

[20] Peter Christen. A parallel iterative linear system solver with dynamic load balancing. In *Proceedings of the 12th international conference on Supercomputing*, pages 7–12. ACM Press, 1998.

[21] Thomas H. Cormen and Elena Riccio Davidson. FG: A framework generator for hiding latency in parallel programs running on clusters. White Paper at http://www.cs.dartmouth.edu/FG/.

[22] Arthur P. Cracknell. *The Advanced Very High Resolution Radiometer*. Taylor and Francis, 1997.

[23] J. Davison de St. Germain, Alan Morris, Steven G. Parker, Allen D. Malony, and Sameer Shende. Integrating performance analysis in the uintah software development cycle. In *International Symposium on High Performance Computing (ISHPC-IV)*, School of Computing, University of Utah, may 2002.

[24] J.D. de St. Germain, J. McCorquodale, S.G. Parker, and C.R. Johnson. Uintah: A massively parallel problem solving environment. In *Ninth IEEE International Symposium on High Performance and Distributed Computing*, pages 33–41. IEEE, Piscataway, NJ, Nov 2000.

[25] Nathan DeBardeleben. Coven module writer's guide version 1.1. ftp://ftp.parl.clemson.edu/pub/coven/docs/coven-modguide.pdf, 2004.

[26] Nathan DeBardeleben. Coven program writer's guide version 1.2. ftp://ftp.parl.clemson.edu/pub/coven/docs/coven-pgmguide.pdf, 2004.

[27] Nathan DeBardeleben. Coven tutorial version 1.3. ftp://ftp.parl.clemson.edu/pub/coven/docs/coven-tutorial.pdf, 2004.

[28] Nathan DeBardeleben, Walter B. Ligon III, and Ron Sass. Arches: An infrastructure for PSE development. In *Ninth IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 120–128, Santa Fe, New Mexico, April 2004.

[29] Nathan A. DeBardeleben. The Component-based Environment for Remote Sensing. Master's thesis, Clemson University, 2001.

[30] Nathan A. DeBardeleben, Walter B. Ligon III, Sourabh Pandit, and Dan C. Stanzione Jr. CERSe - a Tool for High Performance Remote Sensing Application Development. In *Science and Data Processing Workshop 2002*, February 2002.

[31] Nathan A. DeBardeleben, Walter B. Ligon III, Sourabh Pandit, and Dan C. Stanzione Jr. Coven - a framework for high performance problem solving environments. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 2002)*, pages 291–298, Edinburgh, Scotland, UK, July 2002. IEEE Computer Society.

[32] Nathan A. DeBardeleben, Walter B. Ligon III, and Dan C. Stanzione Jr. The Component-based Environment for Remote Sensing. In *Proceedings of the 2002 IEEE Aerospace Conference*, March 2002.

[33] Alexandre Denis, Christian Perez, Thierry Priol, and Andres Ribes. Padico: a component-based software infrastructure for grid computing. In *Proceedings of the 2003 International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[34] Karen Devine, Bruce Hendrickson, Erik Boman, Matthew St. John, and Courtenay Vaughan. Design of dynamic load-balancing tools for parallel applications. In *Proceedings of the 14th international conference on Supercomputing*, pages 110–118. ACM Press, 2000.

[35] Jack Dongarra, Hans Meuer, and Erich Strohmaier. Top500 supercomputer sites. http://www.top500.org.

[36] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report CS-93-213 (revised), University of Tennessee, April 1994.

[37] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[38] Ian Foster, Carl Kesselman, Jeffry M. Nick, and Steven Tuecke. Grid services for distributed system integration. *IEEE Computer*, 35(6):37–46, June 2002.

[39] Ernest Friedman-Hill. JESS: java expert system shell. http://herzberg.ca.sandia.gov/jess, 2004.

[40] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[41] Efstratios Gallopoulos, Elias Houstis, and John R. Rice. Computer as thinker/doer: Problem solving environments for computational science. *IEEE CS & E*, 1(2):11–23, Summer 1994.

[42] Sebastian Gerlach and Roger D. Hersch. DPS - dynamic parallel schedules. In *Proceedings of the Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'03)*, pages 15–24, 2003.

[43] Andrew Grimshaw, Adam Ferrari, Fritz Knabe, and Marty Humphrey. Wide-area computing: Resource sharing on a large scale. *IEEE Computer*, 32(5):29–37, May 1999.

[44] Andrew Grimshaw and Wm. A. Wulf. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[45] Object Management Group. CORBA components. http://www.omg.org/technology/documents/formal/components.htm.

[46] Mounir Hamdi and Chi-Kin Lee. Dynamic load balancing of data parallel applications on a distributed network. In *Proceedings of the 9th international conference on Supercomputing*, pages 170–179. ACM Press, 1995.

[47] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIND Computers*. The MIT Press, 1991.

[48] Jorg Henrichs. Optimizing and load balancing metacomputing applications. In *Proceedings of the 12th international conference on Supercomputing*, pages 165–171. ACM Press, 1998.

[49] Chris Hill, Cecelia Deluca, V. Balaji, Max Suarez, and Arlindo da Silva. The architecture of the earth system modeling framework. *Computing in Science and Engineering*, 6(1), 2004.

[50] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xin-Min Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Laurie J. Hendre, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank S. Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchu Zun. A design study of the EARTH multiprocessor. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 59–68. ACM Press, June 1995.

[51] Marty Humphrey. From legion to legion-g to ogsi.net: Object-based computing for grids. In *Proceedings of the IPDPS NFS Next Generation Software Workshop*, Nice, France, April 2003.

[52] Christopher Hylands, Edward Lee, Jie Liu, Xiaojun Liu, Stephen Neuendorffer, Yuhong Xiong, Yang Zhao, and Haiyang Zheng. Overview of the ptolemy project. Technical memorandum no. ucb/erl m03/25, University of California, Berkeley, July 2003.

[53] K. A. Iskra, F. van der Linden, Z. W. Hendrikse, B. J. Overeinder, G. D. van Albada, and P. M. A. Sloot. The implementation of dynamite: an environment for migrating pvm tasks. *SIGOPS Oper. Syst. Rev.*, 34(3):40–55, 2000.

[54] C.R. Johnson, S. Parker, D. Weinstein, and S. Heffernan. Component-based problem solving environments for large-scale scientific computing. *Journal on Concurrency and Computation: Practice and Experience*, (14):1337–1349, 2002.

[55] T. Jones, A. Koniges, and R. K. Yates. Performance of the IBM general parallel file system. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 673–681, May 2000.

[56] L. V. Kalé, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM Parallel Programming Language and System: Part I – Description of Language Features. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[57] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II – The Runtime system. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[58] James Arthur Kohl and Philip M. Papadopoulas. Efficient and flexible fault tolerance and migration of scientific simulations using cumulvs. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 60–71. ACM Press, 1998.

[59] Sriram Krishnan and Dennis Gannon. XCAT3: A framework for CCA components as OGSA services. In *Ninth IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 90–97, Santa Fe, New Mexico, April 2004.

[60] R. Latham, N. Miller, R. B. Ross, and P. H. Carns. A next-generation parallel file system for linux clusters. *LinuxWorld Magazine*, January 2004.

[61] Sophia Lefantzi, Jaideep Ray, and Habib N. Najm. Using the common component architecture to design high performance scientific simulation codes. In *Proceedings of the 2003 Parallel and Distributed Processing Symposium*, April 2003.

[62] K. Li, J. F. Naughton, and J. S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 79–88. ACM Press, 1990.

[63] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[64] Bart Maerten, Dirk Roose, Achim Basermann, Jochen Fingberg, and Guy Lonsdale. DRAMA: A library for parallel dynamic load balancing of finite element applications. In *European Conference on Parallel Processing*, pages 313–316, 1999.

[65] Edward Mascarenhas, Felipe Knop, Reuben Pasquini, and Vernon Rego. Checkpoint and recovery methods in the parasol simulation system. In *Proceedings of the 29th conference on Winter simulation*, pages 452–459. ACM Press, 1997.

[66] Edward Mascarenhas, Felipe Knop, and Vernon Rego. Parasol: a multithreaded system for parallel simulation based on mobile threads. In *Proceedings of the 27th conference on Winter simulation*, pages 690–697. ACM Press, 1995.

[67] MCS Division, Argonne National Laboratory. MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich.

[68] MCS Division, Argonne National Laboratory. MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich2.

[69] Ken Melero. Open Source Software Image Map Documentation. http://www.ossim.org, 2001.

[70] Richard Tran Mills, Andreas Stathopoulos, and Evgenia Smirni. Algorithmic modifications to the Jacobi-Davidson parallel eigensolver to dynamically balance external cpu and memory load. In *Proceedings of the 15th international conference on Supercomputing*, pages 454–463. ACM Press, 2001.

[71] Tuan-Anh Nguyen and Pierre Kuonen. ParoC++: a requirement-driver parallel object-oriented programming language. In *Proceedings of the Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'03)*, pages 25–33, 2003.

[72] John R Rice. Scalable Scientific Software Libraries and Problem Solving Environments. Technical report, Department of Computer Science, Purdue University, 1996.

[73] Peter Scheuermann, Gerhard Weikum, and Peter Zabback. Data partitioning and load balancing in parallel disk systems. *The VLDB Journal*, 7(1):48–66, 1998.

[74] Karen Schuchardt, Brett Didier, and Gary Black. Ecce - a problem solving environment's evolution toward grid services and a web architecture. *Concurrency and Computation: Practice and Experience*, 14:1221–1239, 2002.

[75] Sameer Shende and Allen D. Malony. Integration and application of the tau performance system in parallel java environments. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, University of Oregon, Eugene, Oregon, June 2001.

[76] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[77] Daniel C. Stanzione Jr. *Problem Solving Environment Infrastructure for High Performance Computer Systems*. PhD thesis, Clemson University, December 2000.

[78] Daniel C. Stanzione Jr. and Walter B. Ligon III. Infrastructure for high performance computer systems. In et al. Jose Rolim, editor, *IPDPS 2000 Workshops, LNCS 1800*, pages 314–323. ACM/IEEE, Springer-Verlag, May 2000.

[79] Georg Stellner. Cocheck: Checkpointing and process migration for mpi. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 526–531. IEEE Computer Society, 1996.

[80] Thomas Sterling. *Beowulf Cluster Computing with Linux*. The MIT Press, 2001.

[81] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[82] Domenico Talia. The open grid services architecture: Where the grid meets the web. *Internet Computing, IEEE*, 6(6):67–71, November–December 2002.

[83] Ian Taylor, Matthew Shields, and Ian Wang. Distributed P2P computing within Triana: A galaxy visualization test case. In *Proceedings of the 2003 International Parallel and Distributed Processing Symposium*, 2003.

[84] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[85] Parimala Thulasiraman, Kevin B. Theobald, Ashfaq A. Khokhar, and Guang R. Gao. Multithreaded algorithms for the fast fourier transform. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 176–185. ACM Press, 2000.

[86] Xin-Min Tian, Shashank Nemawarkar, Guang R. Gao, and Herbert Hum. Data locality sensitivity of multithreaded computations on a distributed-memory multiprocessor. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, page 37. IBM Press, 1996.

[87] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 34–43. ACM Press, 2001.

[88] Linda F. Wilson and David M. Nicol. Automated load balancing in speedes. In *Proceedings of the 27th conference on Winter simulation*, pages 590–596. ACM Press, 1995.

[89] Linda F. Wilson and David M. Nicol. Experiments in automated load balancing. In *Proceedings of the tenth workshop on Parallel and distributed simulation*, pages 4–11. IEEE Computer Society, 1996.

[90] Linda F. Wilson and Wei Shen. Experiments in load migration and dynamic load balancing in speedes. In *Proceedings of the 30th conference on Winter simulation*, pages 483–490. IEEE Computer Society Press, 1998.

[91] Keming Zhang, Kostadin Damevski, Venkatanand Venkatachalapathy, and Steven Parker. SCIRun2: A CCA framework for high performance computing. In *Ninth IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 72–79, Santa Fe, New Mexico, April 2004.