Adaptive Data Placement for Distributed-Memory Machines

David K. Lowenthal
Gregory R. Andrews

Adaptive Data Placement for Distributed-Memory Machines[1]

David K. Lowenthal Gregory R. Andrews

TR 95-13

Abstract

Programming distributed-memory machines requires careful placement of data on the nodes. This is because achieving efficiency requires balancing the computational load among the nodes and minimizing excess data movement between the nodes. Most current approaches to data placement require the programmer or compiler to place data initially and then possibly to move it explicitly during a computation. This paper describes a new, adaptive approach to data placement. It is implemented in the Adapt system, which takes an initial data placement, efficiently monitors how well it performs, and changes the placement whenever the monitoring indicates that a different placement would perform better. Using Adapt can simplify the programming of parallel systems and simplify compilers for parallel languages such as HPF. In particular, Adapt frees the programmer from having to specify data placements, and it frees the compiler from having to do often complex analysis to determine a good placement. Moreover, Adapt supports a new "variable block" placement, which is especially useful for applications with nearest-neighbor communication but an imbalanced workload. For applications in which the best data placement varies dynamically, using Adapt can lead to much better performance than using any statically determined data placement. We present the performance of Adapt on three scientific applications that require different data placements.

December 4, 1995

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work replaces TR 94-35 and was supported by NSF grants CCR-941503 and CDA-8822652.

1 Introduction

Distributed-memory machines—including parallel computers and workstation clusters—are used to achieve scalable high performance computing. Programming these machines requires specifying both what can execute concurrently and when and how processes communicate. These two problems are largely independent. We assume that processes have already been specified—either by the programmer or by a compiler—and we consider the problems of how data is placed initially in the memories of the processors and how data moves during a computation.

The goal of this work is to determine data placements dynamically rather than requiring programmers or compilers to make such decisions. Most current approaches determine data placements statically. They can generally be divided into two categories: using language primitives, such as the ones in HPF [HPF93], or compiler analysis, such as the work reported in [AL93], [GB93], and [KK94]. Language primitives involve the programmer in the choice of data placement; unfortunately, the best placement may be difficult or impossible for the programmer to determine. Compiler analysis also may not be able to infer the best data placement; moreover, the difficulty of inferring placements greatly increases the size and complexity of the compiler.

This paper describes a completely dynamic approach to data placement. Our approach has been implemented in a prototype system called Adapt, which has the following attributes:

- Given some initial data placement, Adapt monitors the effect of the placement (with low overhead) and changes it to a better one if needed.
- Neither the programmer nor the compiler need be involved in the selection of the initial or new data placements.
- Adapt supports new data placements, those with variable sized blocks, that to our knowledge are not supported by current languages or compilers.
- Programs written using Adapt will run efficiently on machines and networks with varying ratios of processor speed to network speed.

Adapt is given (or chooses) some initial data placement and then monitors computation time and communication overhead and computes delays on each node to determine if a different placement would lead to a shorter completion time for the overall computation. When it finds a better placement, it changes to this new placement. Adapt continues to monitor the program, and if the characteristics of the application change, it changes the placement again. The ability to change placements during execution is especially important for problems—such as particle-in-cell codes [Har64]—for which the best data placement can vary over the course of the application [PAM94].

Adapt is currently implemented on a cluster of Sparc-1s and supports iterative scientific applications, which comprise a large subset of computational science applications. Performance on a network of workstations is such that Adapt can outperform programs using any statically determined data placement on applications in which the benefit of dynamically redistributing the data outweighs the overhead of redistribution. The Adapt version of a particle simulation ran over 10% faster than the program with the best statically determined placement when the particles tended to cluster. Even when good placements can be statically determined, Adapt is competitive with programs that use them; e.g., Adapt versions of Jacobi iteration and LU decomposition are only slightly slower than the best static counterparts, ranging from the best case of 1% slower to the worst case of 14% slower.

The remainder of the paper is organized as follows. Section 2 describes the data placement problem and the range of possible placements. Section 3 gives an overview of Adapt and its implementation. Section 4 presents performance results, and Section 5 discusses data placement methods and describes related work. Finally, Section 6 contains concluding remarks.

2 Framework for Data Placement

A data placement is a mapping of the data elements in a program to the memories of the nodes. There is some initial placement when a program begins execution; it can be changed, or *remapped*, in the middle of execution. The ideal data placement minimizes the overall completion time of an application. Because all nodes cooperate in order to complete an application, the completion time of the slowest node determines the completion time of the application.

Three factors affect the completion time of a node: computation time, communication overhead, and delay. Computation time is essentially the time spent executing application code, communication overhead is time spent executing low-level code that copies messages to and from the network, and delay is time spent waiting for other nodes to complete their computation or respond to a message. This section first describes our models of computation and communication and how a data placement affects computation time, communication overhead, and delays. Then we discuss the range of data placements and their relation to representative applications.

Our computational model is Single Program Multiple Data (SPMD) [HKT92], in which each process executes the same code but references a different subset of the data elements We also require iterative computations, because placing data dynamically depends on having computations that exhibit repeated access patterns, giving a run-time data placement method the opportunity to detect these patterns and use them to make a placement choice. (Iterative computations comprise a large subset of scientific applications.) Hence, each process contains one or more loops, with each loop consisting of a sequence of application code/barrier pairs[1]. Data placement remappings are made only at the last barrier point in a loop, because that is the one point at which information exists about every code segment in the loop.

We assume that any node can reference any data element. We also assume the *owner-computes* [HKT92] rule, which means each data element has an "owner" node and that is the only node that will update the element; however, other nodes may reference the element.

The data placement affects both computation time and communication overhead. Because of the owner-computes rule, the number of data elements each node owns determines the time it spends computing. Communication (message passing) occurs when a node accesses a non-local data element while updating a local data element; this, along with where the data elements are placed, determines the communication overhead on a node. This communication can be implemented implicitly (e.g. by a distributed shared memory) or explicitly (e.g. by a compiler). In either case, communication results in overhead to transfer the data from the application process to the network, and from the network to the application process on the receiving machine.

Both the computation time and communication overhead on a node can lead to delay. If the total computation and overhead between the nodes is unbalanced, they will finish at different times, causing the early finishing nodes to block at a barrier point waiting for the others to finish. Also, after a node transfers a message to the network, it may need to block waiting for the reply to that message. The key for a good data placement is balancing the computation between the nodes—to minimize synchronization delay—while also minimizing the number of messages—to minimize communication overhead and message delay.

The elements of a data structure can be placed on the nodes in numerous ways. However, the goals of simultaneously balancing computational load and minimizing communication often conflict,

[1] A barrier is a synchronization point at which all processes must arrive before any proceed.

Application	Jacobi iteration	LU decomposition	Particle Simulation	ADI
Computation	Balanced	Unbalanced	Unbalanced	Balanced
Locality	Important	Not important	Important	Important
Best Placement	`BLOCK`	`CYCLIC`	Variable Blocks	`BLOCKCYCLIC(x)`
Static approaches work?	Yes	Yes	No	Maybe

Figure 1: Summary of application characteristics.

as there is an interaction between the two. For example, one placement extreme is to put all data elements on one node; this will minimize communication (there is none), but it also maximizes load imbalance (all other nodes are idle), which leads to large delays at barrier points. The other extreme is to assign elements randomly to nodes; this will (probabilistically) balance the load, but the lack of spatial locality will most likely lead to a large amount of communication.

Below we describe a feasible set of data placements and a representative application for each. (Figure 1 summarizes the applications.) The best placement is dependent on both application and machine characteristics. Application characteristics include data access patterns and computation required to update each element, and machine characteristics include processor, memory, and communication speeds. For explanatory purposes, we restrict our attention to the placement of the elements of a single two-dimensional 16×16 array onto 4 nodes.

One alternative is use a *block* placement, which places a logically contiguous set of approximately the same number of data elements on each node. This mapping, called `BLOCK` in HPF, could distribute just one dimension of a matrix—in which case elements in the same row or column are mapped to the same node—or it could distribute both dimensions. (For an n dimensional array, up to n dimensions of the array could be distributed.) For example, a one-dimensional mapping places a group of 4 contiguous rows on each node, whereas two-dimensional mapping places a square 8×8 subarray on each node[2]. Contiguous placements tend to work well for stencil-based applications such as Jacobi iteration, because such applications have spatial and temporal locality, a balanced workload, and regular communication between neighboring nodes.

Some applications that have locality and a regular "nearest neighbor" communication pattern do not have a balanced workload, such as particle-in-cell codes [Har64]. It is still usually best to use a contiguous mapping for such applications, but the mapping may need to assign each node a different number of elements in order to balance the workload. In particular, the number of elements per node should match the distribution of particles. For example, if many more particles reside in the top rows than in the middle or bottom rows, then a possible one-dimensional mapping would be to map rows 0 through 2 to node 0, rows 3 through 8 to node 1, and so on. We will refer to these placements as *variable block*[3]. If the workload is fairly well balanced by such a placement, performance should be quite reasonable. Particle-in-cell is also an application where remapping can be beneficial, because the particles move dynamically through the array, and hence the distribution of particles can vary dramatically over time.

Another placement method is to *stripe* data across the nodes. For example, a one-dimensional, striped mapping, called `CYCLIC` in HPF, maps rows 0, 4, 8, and 12 to node 0; rows 1, 5, 9, and 13 to node 1; and so on. Striped placements can handle problems with changing workloads well, because if the amount of work per element decreases within the computation, a striped placement balances the load without a need for remapping. (Using a variable-block mapping, such as the one described

[2] Whether it is better to distribute one dimension or two is dependent on problem size, communication latency, and communication bandwidth. In general, a one-dimensional mapping results in fewer messages than a two-dimensional mapping, but it communicates more data per message.

[3] HPF does not support such placements.

in the particle example, would balance the load temporarily, but as the workload decreases periodic remapping would be needed.)

However, striped placements have fairly poor spatial locality, so they are typically useful only when the amount of communication in an application is (relatively) independent of the data placement. LU decomposition is an example of an application with a changing workload and a placement-independent communication pattern.

A compromise between the contiguous and striped mappings is to combine the two methods and stripe contiguous regions onto each processor. An example of such a one-dimensional mapping, where 2 contiguous sets of elements are striped across the nodes, would be placing rows 0, 1, 8, and 9 on node 0; rows 2, 3, 10, and 11 on node 1; and so on. We denote this placement `BLOCKCYCLIC(2)`[4]. (The parameter refers to the size of blocks mapped to each node, not the number of blocks.)

This kind of placement is useful when the parallelism is not perfect; i.e. the application has dependencies that prevent full parallelization, usually resulting in pipelined code. In such applications, such as the second sweep in Alternate Direction Integration [McM86], a node must wait for data from other nodes before starting to perform computation. Contiguous placements would exacerbate this delay, and a fully striped placement would cause excess communication. Hence, a combination placement can be a good compromise.

3 Adapt and its Implementation

The Adapt system chooses data placements that attempt to minimize the completion time of an application. Adapt is given some initial data placement by the programmer or compiler and then employs three steps. First, it acquires information at run-time about the code segments that are in loops. Next, it uses this information to choose a data placement that tries to minimize both communication overhead and delay by balancing the computation and minimizing the number of messages. Finally, it effects the new placement and continues to monitor the computation in case the workload changes. Regardless of the initial placement (the default is currently `BLOCK`), Adapt when necessary finds a better placement. Below we discuss how Adapt monitors the computation (Section 3.1), determines a placement (Section 3.2), and effects and continues to monitor this placement (Section 3.3).

The current implementation of Adapt is implemented in concert with the Distributed Filaments (DF) software kernel [FLA94]. It supports multi-dimensional arrays and distributes only the first dimension (rows). It relies on instrumenting a distributed shared memory (DSM) (for monitoring) and requires that code segments in loops have the same communication pattern. We describe possible extensions to Adapt in Sections 5 and 6.

3.1 Adapt Monitoring

Adapt gathers information about the communication pattern and computation time for each code segment. Communication in Adapt is performed implicitly by an underlying DSM provided by DF. Adapt uses this DSM to determine the communication pattern and number of messages. It also uses the UNIX `gettimeofday()` command to estimate execution times.

The system must determine communication overhead, computation time, and both synchronization delay and message delay. The next subsection discusses how Adapt determines communication overhead and message delay. Subsection 3.1.2 describes how the system estimates computation

[4]The HPF equivalent to this is `CYCLIC(2)`, but we use the Fortran D notation of `BLOCKCYCLIC(2)` because it more clearly denotes the combination mapping.

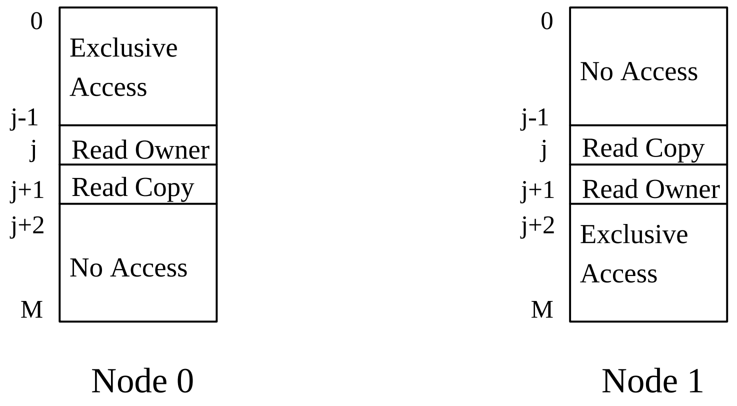

Figure 2: Portion of page table showing nearest-neighbor communication pattern. The edge sharing is detected by observing that pages j and $j+1$ are readable by both nodes, with each node owning one page.

time. Adapt does not need to measure synchronization delay, as the maximum delay (which is all that matters) is implicit in the completion time of the slowest node.

3.1.1 Communication Monitoring

Adapt monitors communication using DSM page faults and the DSM page table. It estimates C_i, the time due to communication overhead and message delay on node i, using the following formula[5]:

$$C_i = m_i * D + s_i * S$$

The factors m_i and s_i refer to the number of messages node i sends and services, respectively. Adapt instruments the DSM to count m_i and s_i. On the other hand, the system determines D and S statically by running isolated tests for each new architecture (alternatively, Adapt could obtain them by monitoring the appropriate DSM routines, which would make the system more self-contained). The delay, D, is the round-trip message time, which is the cumulative time for a page request message to travel to a remote node, be serviced, and the page reply to travel back to the requesting node. The service time, S, is the time a node spends servicing a page request.

Adapt determines the communication pattern by inspecting the pattern of page faults on each array through the page table. Currently, Adapt recognizes two patterns: *nearest-neighbor* and *broadcast*. In the nearest-neighbor communication pattern, node i needs to communicate values with nodes $i+1$ and $i-1$. This pattern occurs on an array when (1) each node has a distinct subset of exclusive-access pages of the array and (2) neighboring nodes have read access to consecutive sets of pages of the array, with each node owning one set. A page table showing nearest-neighbor communication between 2 nodes is shown in Figure 2.

A broadcast pattern means that one node writes a value, there is a barrier synchronization point, and then all nodes read the value. Adapt detects a broadcast pattern on an array if there are a pair of code segments that exhibit the following characteristics: (1) in the first segment one node writes to a subset of pages of the array, (2) in the second segment each node has a distinct subset of exclusive-access pages of the array, and (3) in the second segment all nodes read the subset of pages that were written in the first segment (see Figure 3).

[5]This formula is actually pessimistic, because a node could be servicing another node's message while waiting for a reply. In addition, some of the delay can be eliminated by overlapping communication and computation, as is done for example in [vCGS92, FLA94].

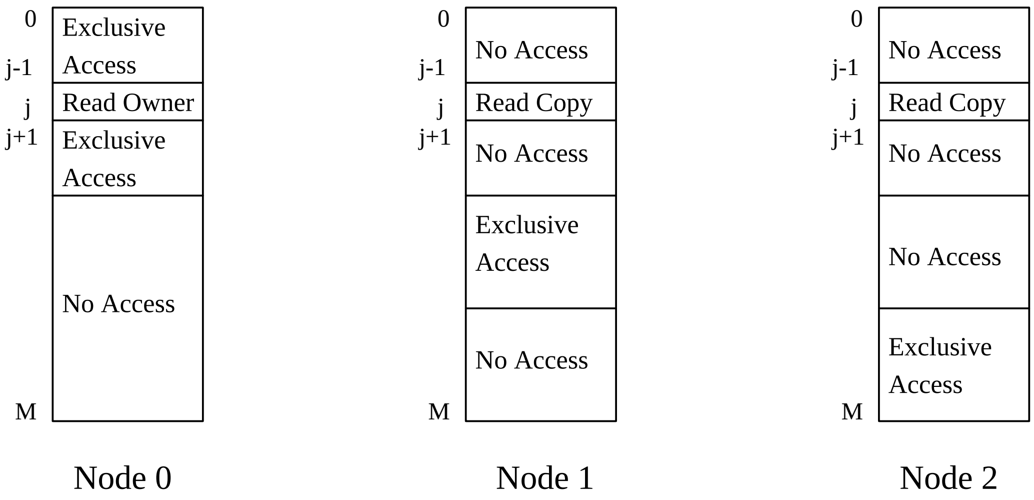

Figure 3: Portion of page table showing broadcast communication pattern. All nodes read page j, with node 0 the owner.

3.1.2 Computation Monitoring

Adapt instruments the code to obtain the time a node spends computing the data elements it owns. The system uses rows as its unit of placement (i.e., it only distributes the first dimension), because the granularity of elements is too small, and using any unit larger than rows would rule out fully-striped placements. Each node computes the times spent accessing its rows; these times are combined at the barrier synchronization point to obtain the total computation time T.

3.2 Adapt Algorithm

Adapt uses the communication and computation information to choose a placement. Adapt maps rows to nodes. Given the total time T and the number of nodes P, T/P represents the amount of computation each node should perform for a perfectly balanced load. Adapt maps the rows to the nodes by a simple bin-packing procedure that is dependent on the communication pattern.

If the communication pattern is nearest neighbor, each node starts with one bin. Adapt packs the bins so that each bin contains *consecutive* rows and the estimated total time on the node is as close as possible to T/P. This forms a variable block placement[6] as described in the particle simulation example in Section 2. Even the best possible contiguous single-bin packing may not sufficiently balance the load. Therefore Adapt also investigates contiguous multiple-bin packings. In this case the system packs the first bin on each node up to but not exceeding T/P. This results in leftover rows in general, which are packed (contiguously) into the extra bins on each node to bring the total time closer to T/P. Adapt adds bins until the load is sufficiently balanced.

Adapt takes into account that this better balancing of workload causes extra communication. In particular, for the nearest-neighbor pattern, each additional bin results in 2 more messages per node. For each number of bins, Adapt estimates the completion time on each node; the overall completion time is the completion time of the slowest node (this is how synchronization delay is accounted for) The system chooses the placement with smallest of these completion times (see example below).

An example of Adapt's algorithm described above is shown below for a nearest-neighbor communication pattern, with $N = 8$ and $P = 2$. Below we show the sample computation times accessing

[6] The algorithm is intended to run quickly and produce a good packing, not an optimal one.

each row and denote the communication overhead as γ.

Row	1	2	3	4	5	6	7	8
Time	2	2	6	5	1	4	2	2

The total time in this example is 24 units, so each node would compute 12 time units for a perfect balance of work. The table below shows the estimated completion time using both one and two bins on each node. Bins are denoted by [ ], with the numbers inside indicating which rows that bin contains.

Bins/Node	Node 0	Node 1	Comm. Overhead	T_0	T_1	Completion Time
1	[1-3]	[4-8]	γ	10	14	$14+\gamma$
2	[1-3], [8]	[4-7], [ ]	2γ	12	12	$12+2\gamma$

In this example, one can see that the first placement leads to a completion time of $14+\gamma$ units (again, the completion time is the time of the slowest node), and the second $12+2\gamma$ units. In this example, if γ is 2, the two placements give the same completion time. If γ is less than 2, the second placement is preferable, and if γ is greater than 2, the first placement is better. Of course, γ is machine dependent.

If the communication pattern is broadcast, the number of messages is constant over the placements considered by Adapt; each node needs to access a specific set of rows, independent of the rows mapped to the node. Consequently, the communication overhead is constant over any number of bins. Adapt uses a history of execution time to make the decision of how to pack the bins. If the computation times on a nodes have been relatively constant over several iterations, the same algorithm is used as in the nearest-neighbor case (start with one bin per node, and potentially add bins for leftover rows). Differing computation times on a node over several iterations indicate that the workload is increasing or decreasing. In this case Adapt uses a different bin-packing procedure: if the problem size is N, each node starts with N/P bins (instead of one) and with capacity of each bin T/N (instead of T/P). Adapt effects a placement resembling `CYCLIC` by placing rows in the nodes' bins in a round-robin manner. In the above example, there would be 4 bins on each node and a capacity per bin of 3. This will find an appropriate placement for applications such as LU decomposition with a decreasing workload (see Section 2).

3.3 Changing the Data Placement

Once a new data placement has been chosen, Adapt changes the data placement by reparameterizing the code so that each node accesses different data. When a node accesses data it does not own, page faults result; the underlying DSM then implicitly moves the data. The Filaments package provides a simple and efficient mechanism for generating a new code parameterization (see [FLA94] for details); however, any generation method will do. After a placement has been changed, Adapt continues to monitor the application to detect when a different placement might be better. (This can happen when characteristics change in the middle of a loop, as described in Section 4). Instead of monitoring page faults and timing the computation to update rows, Adapt uses a much more coarse-grain monitoring: it gathers only the overall computation and communication times on each node during each iteration. A large variance in the computation times suggests an imbalanced load, which might require a placement that better balances the load. An increase in the communication times suggests excess communication, which might require a placement with more locality. If either is detected, Adapt notifies the nodes before the start of the next iteration. All nodes then re-enable the fine-grain monitoring (time each row, etc.) and repeat the algorithm described above to determine the new (if any) best placement.

Number of Nodes	1	2	4	8
Adapt Time (sec)	189	104	55.2	32.0
DF Time, `BLOCK` (sec)	188	104	54.6	30.4
DF Time, `BLOCKCYCLIC`($N/2P$) (sec)	188	107	57.5	33.0

Figure 4: Jacobi iteration, 512×512, $\epsilon = 10^{-3}$, 70 iterations.

4 Performance

This section reports the performance of Adapt on three programs: Jacobi iteration, LU decomposition, and particle simulation. Jacobi iteration and LU decomposition are examples of applications in which a good data placement can be determined statically. Particle simulation, on the other hand, requires run-time support both to determine a good placement and possibly to change the placement during the computation.

For each application we developed a program using Adapt. For an accurate comparison, we also developed a Distributed Filaments (DF) [FLA94] program without the Adapt subsystem. The DF program uses a statically determined data placement. For each application we present the results of the DF program with the *best* statically determined data placement and compare it to the Adapt program.

Below, we briefly describe the three applications and present the results of runs on 1, 2, 4, and 8 nodes. (The one-node Adapt programs have only a few extra conditionals compared to the one-node DF programs, so their times were virtually identical.) All tests were run on a network of 8 Sparc-1s connected by a 10Mbs Ethernet. They use the `gcc` compiler with the `-O` flag for optimization. The execution times reported are the median of at least three test runs, as reported by `gettimeofday`. The tests were performed when the only other active processes were Unix daemons.

4.1 Jacobi Iteration

Laplace's equation in two dimensions is the partial differential equation $\nabla^2(\Phi) = 0$. Given boundary values for a region, its solution is the steady-state values of interior points. These values can be approximated numerically by using a finite difference method such as Jacobi iteration, which repeatedly computes new values for each point, then tests for convergence.

Jacobi iteration is an example of an application that has a nearest-neighbor communication pattern and a load that is perfectly balanced. In particular, each node needs to communicate only with its neighbors to exchange edges, and the same amount of computation is performed on each point of the matrix on each iteration. Hence, the best data placement for this application is `BLOCK`, as all placements with less locality incur more communication with no load-balancing benefit.

The execution times for the three versions of Jacobi iteration are shown in Figure 4. The Adapt program initially uses `BLOCK` by default; after recognizing nearest-neighbor communication, Adapt runs the bin-packing algorithm, which reproduces the `BLOCK` placement. (In some cases the bin-packing algorithm does not quite produce `BLOCK`, because small variances in row execution times lead to mapping some nodes one more or one fewer row.) The difference between this program and the DF program that uses `BLOCK` is small because the placement Adapt chooses is virtually identical to the initial placement, so remapping consumes very little (if any) time. As an example of the effects of excess communication, the DF program that uses `BLOCKCYCLIC`($N/2P$) (2 contiguous groups of $N/2P$ rows mapped to each node) is worse than the DF program that uses `BLOCK` due to the doubling of communication overhead (again, see the figure).

Number of Nodes	1	2	4	8
Adapt Time (sec)	173	107	77.2	—
DF Time, `CYCLIC` (sec)	172	95.0	68.2	—
DF Time, `BLOCK` (sec)	172	111	84	—

Figure 5: LU decomposition, 512×512.

4.2 LU Decomposition

LU decomposition is used to solve the linear system $Ax = b$. First, decompose A into lower- and upper-triangular matrices, such that $A = LU$. Then $Ax = b$ becomes $Ax = LUx = b$, so the solution, x, is obtained by solving two triangular systems $Ly = b$ and $Ux = y$.

LU decomposition is an example of an application in which the load is not balanced. After a row is pivoted, it is never accessed again; on iteration i, only an $(n-i+1)$ by $(n-i+1)$ submatrix is accessed. The workload decreases by one row on each iteration. On each iteration, every node must read the pivot row (row i), which is written by the owner of row i. Communication is constant over all data placements. For these reasons, the best data placement for this application is `CYCLIC`.

The execution times for three versions of LU decomposition are shown in Figure 5. Near the beginning of the computation, the work is evenly balanced, as most rows are still active. Thus, after recognizing a broadcast communication pattern, Adapt packs the bins in the variable block manner (see Section 3), just as in Jacobi iteration. However, Adapt quickly detects imbalanced load, re-enabling the fine-grain monitoring. At this point Adapt also detects a decreasing workload and packs the bins in a cyclic manner (again, see Section 3). The difference between this program and the DF program that uses `CYCLIC` is primarily the cost of the extra page faults necessary to change the data placement at run time. There is also minimal load imbalance on the initial iterations where the Adapt version uses a variable block placement. (If Adapt used `CYCLIC` as the default placement instead of `BLOCK`, the remapping time would be very low, and the performance of the Adapt version would be better.) To show the effects of an imbalanced load, we also ran a DF program using `BLOCK`, which exhibits severe tail-end load imbalance. Consequently, its performance is much worse than the `CYCLIC` version.

The eight node test for LU decomposition is omitted. Neither the Adapt nor DF version of the program sped up relative to the 4 node test because the problem size is not large enough. (Our experimental cluster of workstations could not handle a larger matrix due to its limited memory.) However, we expect the performance of Adapt relative to the DF `CYCLIC` program to be about the same on eight node tests.

4.3 Particle Simulation

Our particle simulation program models the behavior of MP3D [McD88]. (The following explanation is paraphrased from [PWG91].) MP3D solves rarefied fluid flow problems, studying the flow of molecules through a rectangular tunnel. Molecules move through the tunnel and at times collide with other molecules — the program computes their new locations using statistically determined probabilities. When exiting the tunnel, the molecules re-enter at the opposite end. The two main data structures are a three-dimensional space array and a list of particles. Our version of particle simulation mimics the behavior of MP3D but is greatly simplified for experimental purposes. For example, we use a two-dimensional grid of space cells. Also, the movement of particles is parameterized to facilitate experimentation. Although our implementation simplifies the physics involved, the computational structure is the same as MP3D.

Number of Nodes	1	2	4	8
Adapt Time (sec)	69.4	40.1	29.8	23.5
DF Time, `BLOCK` (sec)	69.1	47.5	38.4	32.4
DF Time, `BLOCKCYCLIC(`$N/2P$`)` (sec)	69.1	47.0	39.1	25.3
DF Time, `BLOCKCYCLIC(`$N/4P$`)` (sec)	69.1	48.5	34.2	26.2
DF Time, `BLOCKCYCLIC(`$N/8P$`)` (sec)	69.1	46.5	39.3	42.6

Figure 6: Particle Simulation, load imbalanced, grid 64×64, 150 particles.

Nodes	1	2	4	8
Adapt Time (sec)	69.4	39.4	24.3	16.3
DF Time, `BLOCK` (sec)	69.1	38.7	22.1	14.6

Figure 7: Particle Simulation, load balanced, grid 64×64, 150 particles.

We distribute the space array to the nodes as in [MSH+95] (as opposed to distributing the particles to the nodes). Each space cell contains a pointer to a list of particles contained in the cell. In each time step, the program updates the positions of each particle and collides particles that reside in the same grid cell.

This application is representative of programs where a good data placement depends on information available only at run time and different placements might be better at different time steps of the computation. The amount of computation at each grid cell depends on how many particles are in that cell, and the initial distribution of the particles is read in at run time. Thus, static analysis cannot in general determine a good data placement. Furthermore, if particles cluster in certain regions of the grid, the data placement may need to change to balance the load better.

We implemented two versions of our particle simulation. In the first, the application tended to move the particles to the upper region of the grid[7]. Figure 6 shows the execution times for this program. The Adapt version performs the best in this case, because when more particles cluster near the top, Adapt remaps the space array to balance the number of particles (for this particular program Adapt performed three remappings). We tested several DF programs with different data placements; using larger block sizes exacerbates the load imbalance, and using smaller block sizes causes excess communication[8]. None of the DF programs perform as well as the Adapt version, which uses a variable block placement. In the second version, the particles do not cluster; hence, as expected a simple `BLOCK` placement works well; as in the Jacobi iteration results, the Adapt program runs slightly slower in this case (see Figure 7).

5 Discussion and Related Work

Data placement can be supported by language-level primitives, compilers, or (less commonly) run-time systems. With language primitives, the programmer annotates each array with a placement (e.g. [HPF93, HKK+91, RSW91, ZBG88, CMZ92, TCF94]). The advantage of using language primitives is that the programmer has full control over the program. However, the programmer might not know the best placement; even if the programmer does, the best placement might change

[7] The clustering of particles is not contrived; this kind of clustering can occur in practice [Har64].

[8] The execution times using very small blocks can cause so many messages that the network bandwidth is exceeded, slowing the program down even more than expected.

when executing the program on a new architecture.

With a compiler-based approach, the compiler infers a placement for each array in the source code by inspecting loops and array accesses (e.g. [GB93, LC90, BFKK91, HA90, LC91, KLS90, Soc91]). Hence, the programmer need not be involved in placing data. On the other hand, a compiler may not be able to infer the best placement, and compiler approach increases its complexity greatly.

With a run-time system approach, such as Adapt ALEXI [Who91], and the inspector/executor [SMC91][9], data-placement decisions are made during execution. This approach can produce good placements for a larger class of applications because of the increased information available at run time, but it incurs additional overhead to do so.

This section first compares the relative merits of the language, compiler, and run-time data placement methods. We do so by revisiting application classes and explaining which method works best for each. At the end we describe ALEXI in more detail.

Jacobi iteration and LU decomposition are application kernels—they represent large classes of applications but are not themselves complex programs. These kernels are characterized by regular data access and computation patterns, and hence they are simple enough both for programmers to understand and compilers to analyze in order to choose a good data placement. Adapt can also easily analyze the communication pattern and computation load, so it too can choose a good placement. This causes some execution-time overhead due to monitoring and possible remapping. For simple kernels that can be easily analyzed, the language or compiler approach is generally better than the run-time approach.

Particle simulation represents another class of application with regular data access patterns but unpredictable workloads. Programmers and compilers cannot determine the best placement in general without knowing both the initial distribution of the particles and the ways in which the particles will move. Neither is usually known in advance, so unless the programmer or compiler happens to pick a placement that balances the load, performance will suffer. The programmer therefore must customize the program for each different data set. With Adapt no program modification is necessary. For particle simulation and similar applications, the run-time method is generally better than the language or compiler method.

Jacobi, LU, and particle simulation are kernels of applications. A key issue is how well the different approaches to data placement perform on larger applications that are composed of several kernels or that have several iterative phases. With the language-based method, the programmer has to examine a large program and predict how data structures will be used in different parts of the program. Generally an array will be updated in one part of the program and read later in another part; it is possible that the best placement is different in each part of the program. Whether a remapping or a combination placement is best requires the programmer not only to have extensive understanding of the application, but also of the underlying hardware [KK94]. With the compiler-based method, determining whether to remap requires extensive analysis. (Some work has been done in this area, such as [AL93] and [KK94].) Furthermore, larger programs are more likely to contain procedures and aliases, hindering a compiler's effort.

Adapt works on some of these types of "multi-kernel" applications. Currently it can remap data between loops, but not within loops containing code segments with different communication patterns. (These can be handled by the approach used in [KK94].) If extended, Adapt could find data placements for some of these applications, but certain types of remappings, such as transpose, would pose a problem. On balance, however, the best placement strategy for large applications

[9]The inspector/executor model is similar to Adapt in the sense that it monitors a loop to determine communication patterns. However, it is concerned with optimizing communication, whereas the goal of Adapt is to determine data placements.

varies with the specific application; all three approaches have both advantages and drawbacks.

Adapt monitors and remaps data at run time. A related approach is ALEXI [Who91], which employs a static cost model for language primitives—based on the cost of machine primitives—and then uses a hill-climbing heuristic executed at run time to determine a good data placement. ALEXI does not allow data placements to change over the course of the application. Like Adapt, ALEXI eliminates difficulties caused by procedure calls, pointers, and run-time loop bounds. Furthermore, the ALEXI system will always choose the best data placement from among those it considers (it does not use run-time measurements), because the language it models has explicit parallelism and specific communication costs for each statement in the language. ALEXI has the same basic philosophy as Adapt: the best time to determine data placement is at run time. However, Adapt allows a data placement to change over the course of an application, uses a shared-memory programming model, and does not require *a priori* information on the costs of statements or machine primitives.

6 Conclusion

We have presented an approach to data placement that allows the placement to adapt to the needs of the application. The performance of Adapt is very reasonable on applications for which a good placement can be statically determined by the programmer or compiler. More importantly, the performance of Adapt can be superior to any static scheme for problems that are impossible to analyze at compile time. Furthermore, Adapt supports a larger class of problems than compiler approaches and it requires no help from the programmer in determining a data placement.

We are working on improving the Adapt implementation. First, we are investigating the possibility of integrating Adapt into a system that provides a shared-memory model but an explicit message-passing implementation (such as HPF). The only information that Adapt obtains from the distributed shared memory is the number of messages and the communication pattern. If implemented in concert with a compiler, Adapt would have access to that information, because a compiler must insert compile- or run-time code to send and receive messages from designated nodes. A key attribute of Adapt is that it times computation, whereas computational workload can only be approximated in a compiler. Integrating Adapt with a compiler would allow the support of pipelined applications, where the main issue is the size of the blocks to be communicated. We will also add new communication patterns to those currently recognized by Adapt, such as butterfly and replicated patterns, and work on tuning some of its parameters, such as the load imbalance threshold. To test scalability, we intend to run Adapt programs on larger machines, including 16- and 32-node clusters.

References

[AL93] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation*, pages 112–125, 1993.

[BFKK91] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 213–223, April 1991.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[FLA94] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *First Symposium on Operating Systems Design and Implementation*, pages 201–212, November 1994.

[GB93] M. Gupta and P. Banerjee. PARADIGM: A compiler for automated data distribution on multicomputers. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, pages 357–367, July 1993.

[HA90] David E. Hudak and Santosh G. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings 1990 International Conference on Supercomputing, ACM SIGARCH Computer Architecture News*, pages 187–200, September 1990.

[Har64] Francis H. Harlow. The particle-in-cell computing method for fluid dynamics. In Bernie Alder, editor, *Methods in Computational Physics*, pages 319–343. Academic Press, Inc., 1964.

[HKK+91] Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, and Chau-Wen Tseng. An overview of the Fortran-D programming system. Report TR91121, CRPC, March 1991.

[HKT92] S. Hiranandani, K. Kennedy, and C.W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[HPF93] High Performance Fortran language specification. October 1993.

[KK94] Ken Kennedy and Ulrich Kremer. Automatic data layout for High Performance Fortran. Technical Report CRPC-TR94498-S, Rice University, December 1994.

[KLS90] K. Knobe, J. Lukas, and G. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[LC90] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 424–432, October 1990.

[LC91] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(4):213–221, August 1991.

[McD88] Jeffrey D. McDonald. A direct particle simulation method for hypersonic rarified flow. Technical Report 411, Stanford University, March 1988.

[McM86] F. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[MSH+95] Shubendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient support for irregular applications on distributed-memory machines. In *Fifth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 68–79, July 1995.

[PAM94] Dantosh S. Pande, Dharma P. Agrawal, and Jon Mauney. Compiling functional parallelism on distributed-memory systems. *IEEE Parallel and Distributed Technology*, 1(1):64–76, April 1994.

[PWG91] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Department of Electrical Engineering and Computer Science, Stanford University, April 1991.

[RSW91] Matthew Rosing, Robert Schnabel, and Robert Weaver. The Dino parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, September 1991.

[SMC91] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[Soc91] David Grimes Socha. *Supporting fine-grain computation on distributed memory parallel computers*. PhD thesis, University of Washington, Seattle, WA 98195, July 1991.

[TCF94] Rajeev Thakur, Alok Choudhary, and Geoffrey Fox. Runtime array redistribution in HPF programs. In *Proceedings of Scalable High Performance Computing Conference 94*, pages 309–316, May 1994.

[vCGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Eric Schauser. Active Messages: a mechanism for intergrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[Who91] Skef Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991.

[ZBG88] H.P. Zima, H.J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6(6):1–18, January 1988.