Adaptive Data Placement for Distributed-Memory Machines

David K. Lowenthal
Gregory R. Andrews

Adaptive Data Placement for Distributed-Memory Machines[1]

David K. Lowenthal and Gregory R. Andrews

TR 94-35

Abstract

Programming distributed-memory machines requires paying careful attention to where the data is placed. This is because for efficiency, it is important to balance the computational load among the nodes and to minimize excess movement of data between the nodes during the computation. Most current approaches to data placement are static, requiring either the programmer or compiler to explicitly place or move the data. This paper describes a new adaptive approach. It is implemented in the Adapt system, which finds the best data placement at run time. When necessary, Adapt automatically changes the data placement in mid-application, so that it adapts to the needs of the application. Adapt can be used to simplify both programming in parallel languages such as HPF and in compilers for these languages. We test the performance of Adapt on three applications: Jacobi iteration, LU decomposition, and Dynamic Jacobi, which mimics the behavior of applications such as adaptive mesh refinement. Using Adapt, the first two attain performance very close to that of programs using the best static data placement; the third application cannot be analyzed statically and ran faster using Adapt than with any static data placement.

December 15, 1994

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

[1]This work was supported by NSF grants CCR-9108412 and CDA-8822652.

1 Introduction

Distributed-memory parallel computers are used to achieve scalable high performance computing. To execute programs on these machines, it is necessary to specify both what can be executed concurrently and when and how processes communicate. These two problems are largely independent. In this paper, we assume that processes have been specified—explicitly in the source program or inferred 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. Ultimately, on distributed-memory machines data placement and movement have to be realized by explicit message passing. However, a shared-memory programming model is generally agreed to be much simpler to use.

The goal of this work is to take a shared-memory programming model and then to determine data placements adaptively rather than requiring programmers or compilers to make such decisions. Most current approaches to data placement are static. They can generally be divided into two categories: using language primitives, such as the ones in HPF [HPF93], or inferring data placements in compilers, such as PARADIGM [GB93]. Language approaches involve the programmer in the choice of data placement, yet the best placement may be difficult or impossible for the programmer to determine. A compiler also may not be able to infer the best data placement; moreover, the complexity involved in inferring data placements greatly increases the size and complexity of the compiler.

In this paper we introduce a completely dynamic approach to data placement. We have created a prototype system, called Adapt, that has the following attributes:

- Data placement is determined at run time with a small amount of overhead.
- The data placement chosen by Adapt will change if needed in the middle of an application.
- Neither the programmer nor the compiler need be involved in the selection of the data placement.
- Programs written using Adapt can be ported to other distributed-memory machines with no source code changes.
- The system is small and simple, consisting of fewer than 1000 lines of C code.

Adapt starts with some initial data placement and monitors communication (data and synchronization messages) and computation to determine if there is a better placement. If so, it effects a change to the new placement. Then, Adapt continues to monitor the program, and if the characteristics of the application change, Adapt changes the placement again. The ability to change placements during execution is especially important for problems—such as adaptive mesh refinement [BO84]—for which the best data placement varies 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 outperforms programs using any static data placement on applications in which dynamically redistributing the data is important; the Adapt version of Dynamic Jacobi is faster than any program with a static placement. Even when good static placements exist, Adapt is competitive with programs that use them; Adapt versions of Jacobi iteration and LU decomposition are only slightly slower than their static counterparts.

The remainder of the paper is organized as follows. Section 2 gives an overview of data placement. Section 3 describes the implementation of Adapt, and Section 4 gives performance results.

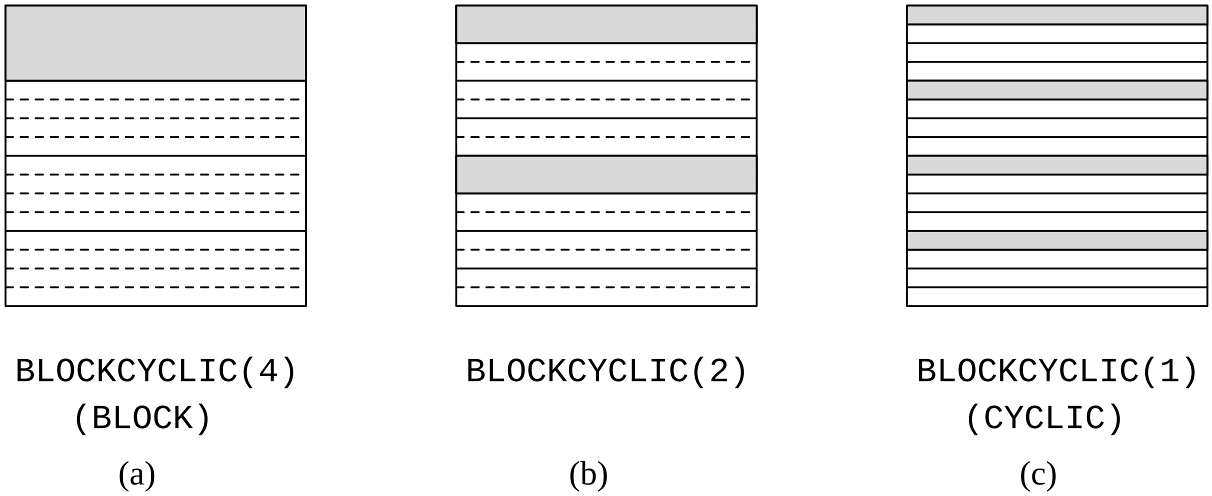

Figure 1: Different data placements in Adapt for the case of $n = 16$ and $p = 4$. Three different `BLOCKCYCLIC` placements are shown. The rows that are placed on node 0 are shaded. We will refer to (a) as `BLOCK` and (c) as `CYCLIC`.

Section 5 describes related work. Finally, Section 6 gives a few concluding remarks and discusses future work.

2 Overview of Data Placement

We assume there are p nodes, numbered $0, 1, \ldots, p-1$. We make two simplifying assumptions in this paper: arrays have only two dimensions, and elements in the same row are placed on the same node (one-dimensional placements)[1]. Each node owns a subset of the data and needs to communicate with other nodes to access any non-local data.

The data placement problem is to divide each data structure in a program among the nodes so that the total completion time is minimized. In other words, given a set of possible data placements $\mathcal{D}$, we wish to minimize $T(d)$, where d ranges over $\mathcal{D}$. In order to minimize $T(d)$, there are two overheads that should be minimized: communication and synchronization delay. Communication overhead results from nodes needing access to non-local data, while synchronization delay results from nodes completing their work at different rates. Attempts to minimize both overheads simultaneously often conflict, as discussed below.

The extremes of data placements for an array are `SEQUENTIAL`, in which the entire array is placed on one node, and `RANDOM`, in which the points of the array are randomly placed on the nodes. The `SEQUENTIAL` placement will minimize communication (there is none) but will have the maximum load imbalance (all other nodes are idle). On the other hand, the `RANDOM` placement will (probabilistically) balance the load, but will likely incur a large amount of communication. The more reasonable extremes are `BLOCK` and `CYCLIC`, as shown in Figure 1. These two placements are really a special case of the general `BLOCKCYCLIC` placement. We will refer to a *strip* as n contiguous data elements of an array. The `BLOCK` placement assigns one group of n/p contiguous strips to each node and the `CYCLIC` placement assigns strips to nodes in a round-robin fashion. The `BLOCK` placement has the most locality and is thus the best for minimizing communication, while

[1]The techniques presented here are valid for arrays of any dimension and for multi-dimensional placements. Also, one could distribute columns rather than rows, e.g., in a Fortran compiler.

Application	Jacobi iteration	LU decomposition	Dynamic Jacobi
Computation	Balanced	Unbalanced	Unbalanced
Locality	Important	Not important	Important
Best Placement	`BLOCK`	`CYCLIC`	Varies
Can static approaches work?	Yes	Yes	No

Figure 2: Summary of application characteristics.

the `CYCLIC` placement has less locality but is the best for balancing the load. The intermediate placements (e.g. `BLOCKCYCLIC(2)`, which assigns two groups of $n/2p$ contiguous strips to each node in a round-robin fashion) are compromises.

In this paper we choose the set $\mathcal{D}$ to be all possible `BLOCKCYCLIC` placements (which includes all regular placements supported by Fortran D [HKK+91], for example). The best placement in this set depends on the application. For applications with roughly the same amount of computation on all parts of the matrix and for which locality is important, the `BLOCK` placement will be the best because of its good locality. On the other hand, for applications with a varied amount of computation on the matrix and for which locality is not as important, the `CYCLIC` placement will be the best because of its good balance. Finally, the best placement for applications in which both locality is important and the amount of computation varies are dependent on several factors, including processor and communication speed.

We will discuss three applications: Jacobi iteration, LU decomposition, and a program we call Dynamic Jacobi, which we use to mimic the behavior of algorithms such as adaptive mesh refinement [BO84]. The characteristics of these applications are summarized in Figure 2. The first two have regular data access patterns and loop structures, so the programmer or compiler can determine the best data placement. However, a data placement for Dynamic Jacobi cannot be determined statically because the best one is dependent on input values. Our adaptive approach works for all three applications. At run time, it monitors data access patterns and computation time and determines the best data placement.

Adaptively determining a data placement involves acquiring information about a phase of an application and then changing the placement for the next phase based on this information. Thus, an adaptive approach can only be used on repeating phases, which we will call iterative phases. Many scientific kernels and applications are comprised of iterative phases, such as PDE solvers, linear algebra factorization routines, and molecular dynamics particle simulations. In each case, a phase consists of computation followed by a barrier synchronization point.

A good adaptive data placement strategy has several requirements. Most importantly, it must find the best data placement with a high degree of accuracy and a small amount of overhead. Second, as iterative phases may be short lived, the best data placement must be found as quickly as possible. Finally, after a data placement is selected, the strategy must make sure to detect when changing characteristics might lead to a better data placement in the middle of an application; however, it must do so with minimal overhead.

3 Implementation

We have developed Adapt, an adaptive data placement scheme. Adapt chooses a data placement based on minimizing the completion time of the application. Figure 3 shows (a) a general strategy for adaptively placing data and (b) the specific implementation used by Adapt. By default, Adapt initially uses the `BLOCK` placement; however, a compiler using Adapt can generate code using

1. Choose an initial placement.
2. Monitor the program until a communication pattern is detected.
3. Determine communication overheads and execution times.
4. Determine the best placement and broadcast this to all nodes.
5. Effect this placement.
6. Continue to monitor the application to determine if there is a better placement.

1. Initially use the `BLOCK` placement.
2. Monitor the program until a pattern of page faults is detected.
3. On each node i and for each placement d, compute the overhead due to communication, $C(d,i)$, and the execution time due to computation, $E(d,i)$.
4. Collect all times and for each placement d, estimate the overall execution time on each node i using $(C(d,i) + E(d,i))$. Then take the largest such time for each d and minimize these times over all possible placements in $\mathcal{D}$.
5. Change the data each node accesses to cause page faults that will effect the new placement.
6. Time only the overall computation and communication on each iteration. If there is either a large variance in computation times or an increase in the communication times, repeat 2-5.

Figure 3: General algorithm for adaptive data placement (left) and specific algorithm used by Adapt (right).

a different initial placement (it might obtain better knowledge through static analysis). It is important to note that *neither* the programmer nor the compiler has to choose this initial placement; Adapt will start with the `BLOCK` placement and then determine the best placement in $\mathcal{D}$, the set of `BLOCKCYCLIC` placements.

Adapt monitors each iteration. When it recognizes a communication pattern, it creates a table. For each node i and each data placement d in $\mathcal{D}$, this table contains the communication overhead $(C(d,i))$ and the time to execute the computation $(E(d,i))$ on this iteration. Adapt then computes $(C(d,i) + E(d,i))$, the estimate of the total execution time on each node. Then, for each d, the completion time $T(d)$ is the largest total execution time over all nodes, i.e. $T(d) = \max_{i=0}^{p-1}(C(d,i)+E(d,i))$. This is because all nodes must wait at a synchronization point until every node has arrived. The best data placement is determined by choosing the smallest $T(d)$ in $\mathcal{D}$. An illustrative example using the above formulas to determine the best placement for Figure 1 is given below. The completion time $T(d)$ is computed for each placement using the sum of communication and computation times estimated by each node (for this example, these times are arbitrary).

Placement (d)	$C(d,0)+E(d,0)$	$C(d,1)+E(d,1)$	$C(d,2)+E(d,2)$	$C(d,3)+E(d,3)$	$T(d)$
`BLOCK`	$2+2$	$4+4$	$3+8$	$8+5$	13
`BLOCKCYCLIC(2)`	$3+1$	$6+3$	$5+6$	$9+3$	12
`CYCLIC`	$5+1$	$7+3$	$9+5$	$11+2$	14

The best placement in this case would be `BLOCKCYCLIC(2)` as it leads to completion in 12 time

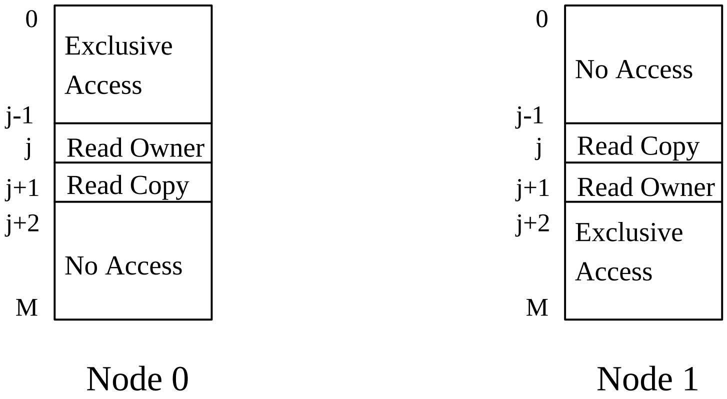

Figure 4: Portion of page table showing nearest-neighbor communication pattern. The edge sharing is detected by observing that pages j and $j+1$ are read-shared, with each node owning one page.

units. After effecting the change in placement (if necessary), Adapt then continues monitoring in case the behavior of the application changes.

Below we discuss how Adapt obtains the communication overhead (Section 3.1) and the execution times (Section 3.2). In Section 3.3 we discuss how Adapt effects a change in the data placement and how it monitors the computation after this change to allow the data placement to adapt to the needs of the application.

The Adapt system currently runs on a network of homogeneous, isolated Sparc-1s[2]. It is built on top of Distributed Filaments (DF) [FLA94], a software kernel that implements a shared-memory programming model on distributed-memory machines. The shared-memory programming model is implemented by a distributed shared memory; when an attempt is made to access a remote memory location, a page fault is incurred, and the page that is needed is brought to the faulting node. Thus, the data accesses in essence place the data.

3.1 Communication Overhead

All communication monitoring in Adapt is performed implicitly as a result of page faults. Several quantities are needed to estimate communication overhead. First, we need the number of page faults, $f(d,i)$, that occur on an iteration on node i using placement d. This is obtained through a counter in the page fault handler. Second, we need the amounts of time taken to acquire a remote page and service a remote request. These numbers can be estimated by monitoring the appropriate handlers or by running isolated tests for each new architecture; currently, Adapt uses the latter. Then the communication overhead for one iteration using placement d can be obtained by multiplying the number of page faults by the sum of the overheads.

$$C(d,i) = f(d,i) * \sum \text{overheads per fault}$$

The above method will estimate the communication overhead of the initial placement used (`BLOCK`). However, Adapt needs to determine $C(d,i)$ for all placements in set $\mathcal{D}$. A brute force method (with very high overhead) is to try all placements and actually execute the computation to determine how many page faults will occur with each placement. Instead, once Adapt recognizes a communication pattern, it computes all $C(d,i)$ using information obtained on only one iteration.

[2]Adapt could be implemented in a multi-user environment with appropriate heuristics to determine whether differing computation times are due to load imbalance or to other activity on the system.

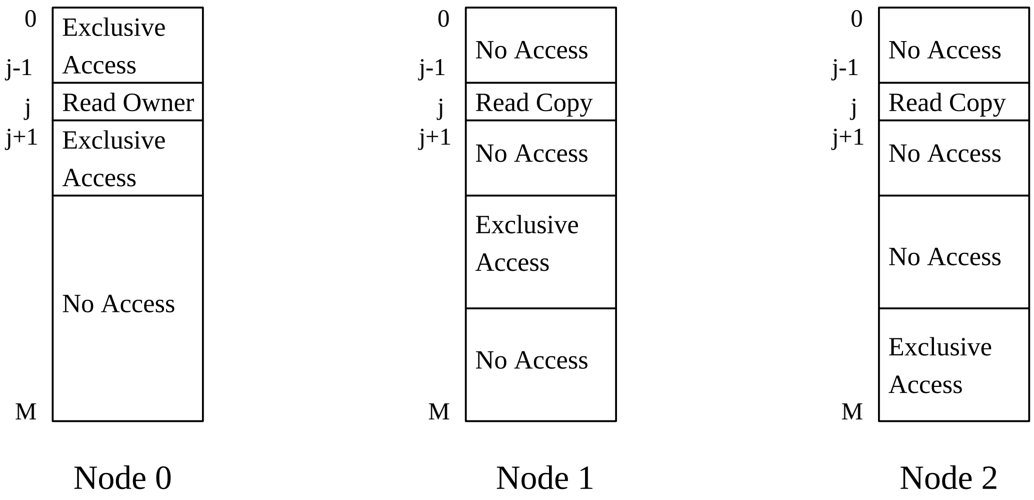

Figure 5: Portion of page table showing broadcast communication pattern. All nodes read-share page j, with node 0 the owner.

This can be accomplished by inspecting the pattern of faults on each array. (Adapt knows what pages are allocated to each array.) Currently, Adapt recognizes two patterns: *nearest-neighbor* and *broadcast*. (We are working on adding new patterns, such as *butterfly*.)

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 (i) each node has a distinct subset of exclusive-access pages of the array and (ii) pairs of nodes (that are neighbors) read-share consecutive sets of pages of the array, with one node owning each set. The degenerate case with 2 nodes is shown in Figure 4. In this case using `BLOCKCYCLIC(x/2)` will result in twice as many page faults and services as using `BLOCKCYCLIC(x)`. Thus, the communication overhead doubles with each `BLOCKCYCLIC` placement that has less locality.

On the other hand, as shown in Figure 5, Adapt detects a broadcast pattern on an array if each node has (i) a distinct subset of exclusive-access pages of the array and (ii) a common subset of read-shared pages of the array. A broadcast pattern means that one node writes a value, there is a barrier synchronization point, and then all nodes read the value. In this case the number of page faults is independent of the data placement, so the communication overhead is constant over all `BLOCKCYCLIC` placements.

3.2 Determining Execution Times

The execution time due to computation on node i with placement d can be determined by timing the start and end of an iteration, then subtracting out any time the node spends performing other tasks, such as communicating. Instead, Adapt determines *all* $E(d, i)$ in one iteration. Adapt achieves this by timing the execution of each strip (row) (Adapt uses `gettimeofday` for all timings). Because the strip is the finest unit of placement, by timing each strip Adapt can determine execution times due to computation for each node and for each placement.

Consider again Figure 1(a). Let R_j denote the time to execute code accessing row j. When program execution begins (using `BLOCK`), node 0 will execute code that accesses rows 0 through 3, node 1 rows 4 through 7, and so on. So $E(\texttt{BLOCK}, 0) = \sum_{j=0}^{3} R_j$ and $E(\texttt{BLOCK}, 1) = \sum_{j=4}^{7} R_j$ (and so on). Now, with an assignment of `BLOCKCYCLIC(2)`, node 0 would execute code that accesses rows 0 and 1 and rows 8 and 9. Because all computation on each node is timed, Adapt can estimate $E(\texttt{BLOCKCYCLIC(2)}, 0)$ using the formula $\sum_{j=0}^{1} R_j + \sum_{j=8}^{9} R_j$. Adapt must have access to all times from all nodes; code accessing rows 8 and 9 was timed on node 2, not node 0. For scalability purposes, each node computes a $p \times (log(n/p)+1)$ table (there are $log(n/p)+1$ different `BLOCKCYCLIC` placements for a problem of size n using p nodes) with its contribution to the execution time for each placement. This table is sent along with the barrier synchronization message. Then the tables are combined to produce $E(d, i)$ for all d and all i.

3.3 Changing Data Placements

Once a data placement has been chosen, Adapt must effect this placement. This simply involves changing the data each node accesses to cause page faults that effect the new placement. After a placement has been chosen, Adapt also continues to monitor the application to detect when a different placement might be better. (This can happen when applications characteristics change in the middle of a phase, as shown in Section 4). Instead of timing the execution of computation accessing each row, Adapt times only the overall computation and communication times of each iteration; each node sends these times along with each barrier synchronization message. 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, the nodes are notified before the start of the next iteration. All nodes then enable the fine-grain monitoring (time the computation on each row again, etc.) and repeat the algorithm described above to determine the new (if any) best placement.

4 Performance

This section reports the performance of three programs: Jacobi iteration, LU decomposition, and a new one we call Dynamic Jacobi. Jacobi iteration and LU decomposition are examples of applications in which the best data placement can be determined statically. Dynamic Jacobi, on the other hand, models applications that need run-time support to determine the best placement.

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

Below, we briefly describe the three applications and present the results of runs on 2 and 4 nodes. (The one-node Adapt program has only a few extra conditionals compared to the one-node DF program, so their times were virtually identical and are not reported.) All tests were run on a network of 4 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.

Nodes	2	4
Adapt Time (sec)	161	100
DF Time, `BLOCK` (sec)	160	99
DF Time, `BLOCKCYCLIC(2)` (sec)	171	110

Figure 6: Jacobi iteration, 256×256, $\epsilon = 10^{-3}$, 360 iterations

Nodes	2	4
Adapt Time (sec)	99.3	68.1
Adapt Time, starting with `CYCLIC` (sec)	93.3	62.8
DF Time, `CYCLIC` (sec)	91.1	62.5
DF Time, `BLOCK` (sec)	118	94

Figure 7: LU decomposition, 512×512

Jacobi iteration is an example of an application that has a nearest-neighbor communication pattern (see Section 3) and a load that is completely balanced. In particular, each node needs to communicate only with its neighbors to exchange edges and the same 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 additional load-balancing benefit.

The execution times for three versions of Jacobi iteration are shown in Figure 6. The Adapt program (as always) initially uses `BLOCK`; after recognizing nearest-neighbor communication and the balanced load, Adapt determines that `BLOCK` is in fact best. The difference between this program and the DF program that uses `BLOCK` is exactly the overhead of Adapt's monitoring. The DF program that uses `BLOCKCYCLIC(2)` is worse than the DF program that uses `BLOCK` due to the doubling of communication overhead.

4.2 LU Decomposition

LU decomposition is used to solve the linear system $Ax = b$. We decompose A into lower- and upper-triangular matrices, such that $A = LU$. Then $Ax = b$ becomes $Ax = LUx = b$, and 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. On each iteration, each 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 four versions of LU decomposition are shown in Figure 7. After recognizing a broadcast communication pattern and the imbalanced load, Adapt determines that the best placement is `CYCLIC`. The difference between this program and the DF program that uses `CYCLIC` is primarily the cost of changing the data placement at run time, which results from extra page faults to effect the change of the data placement. This is shown by the time of the Adapt program using an initial `CYCLIC` placement; the time for this program is almost the same as the DF program using `CYCLIC`. To show the effects of an imbalanced load, we include the DF program using `BLOCK`, which exhibits severe tail-end load imbalance. Consequently, its performance is much worse than the other programs.

Nodes	2	4
Adapt Time (sec)	137	89.4
DF Time, `BLOCK` (sec)	178	103
DF Time, `BLOCKCYCLIC(2)` (sec)	141	94.1

Figure 8: Dynamic Jacobi, 256×256

4.3 Dynamic Jacobi

Dynamic Jacobi (DJ) is a parameterized program that simulates the behavior of applications for which the best data placement can be determined only at run time or for which the best data placement changes over the course of the application (or both). DJ is similar to Jacobi iteration except that an auxiliary matrix, read in and referenced at run time, determines how much computation should be performed at each point. Thus static analysis cannot determine the best data placement. Dynamic Jacobi is intended to model applications such as adaptive mesh refinement (AMR) [BO84] and MP3D [McD88]. AMR, for example, is similar to Jacobi iteration except that when the convergence rate of certain regions is slow, a finer grid is generated to cover this region. This increases the amount of computation in these regions. We can model this with DJ by increasing the computation performed at certain regions (we do not actually generate the finer grids).

The parameters we used for DJ are such that adaptive mesh refinement is modeled. The DJ program performs 256 iterations. On the first 128 iterations, it performs exactly the same computation as Jacobi iteration. On the last 128 iterations we modeled an adaptive refinement of the bottom half of the grid. These points perform 5 relaxations per iteration[3]. As a result, the best data placement for the first 128 iterations is `BLOCK`, just as in Jacobi iteration. On the last 128 iterations, the best data placement is `BLOCKCYCLIC(2)`, which completely balances the load. The `BLOCK` placement will cause nodes that work on the top half of the matrix to have very little work compared to nodes that work on the bottom half, and placements with less locality than `BLOCKCYCLIC(2)` will cause excess communication.

The execution times for three versions of DJ are shown in Figure 8. After recognizing nearest-neighbor communication, the Adapt program determines that the `BLOCK` placement is best because the load is balanced, just as in Jacobi iteration. Then, after detecting the large variance in computation times during the last 128 iterations, Adapt determines that the `BLOCKCYCLIC(2)` placement is best and changes to use it. The regular DF programs cannot change the data placement, so the DF program using `BLOCK` performs poorly on the last 128 iterations (due to load imbalance) and the DF program using `BLOCKCYCLIC(2)` performs poorly on the first 128 iterations (due to excess communication).

5 Related Work

One common approach to data placement is to provide language-level primitives, such as `BLOCK` and `CYCLIC`. This is the approach used by HPF [HPF93], Fortran D [HKK+91], and Dino [RSW91], among others. The compiler uses the programmer's specification of the data placement to determine which data each node owns and when and how each node communicates. This approach has two advantages. First, the programmer does not have to become involved with unnecessary, low-level details of the application, such as determining starting and ending rows and columns for each

[3] Most likely, after the slow converging regions are smoothed on the finer grid, the characteristics change back to those similar to Jacobi iteration. Adapt would then switch back to `BLOCK` in the same manner.

node. Second, the compiler can generate a program with efficient communication, because it knows exactly how the data is placed. However, language primitives hinder both programming ease and program portability. In many complex programs, the programmer may not know the best data placement. Also, when moving the program to a new architecture with a different ratio of computation to communication speed, the programmer may need to change the use of the data placement primitives to achieve good performance.

Another approach to data placement is to build a compiler to analyze the program code and automatically determine a placement. The PARADIGM compiler [GB93] analyzes the whole program, and based on the structure of the loops and array accesses, determines the best data placement for each array. PARADIGM achieves this by formulating constraints on each array in the program and then combining the constraints consistently. The Crystal compiler is similar [LC90]. Many people have worked on methods for compiling iterative loops, such as Socha [Soc91]. Balasundaram et al. [BFKK91] implemented a static performance estimator that uses training sets. The compiler approach to data placement has several advantages. Most importantly, the programmer does not have to get involved in placing data. Also, programs need not be changed to run efficiently on new architectures. However, this approach also has several drawbacks. There are problems for which the best data placement can be determined only at run time, such as mesh refinements and particle codes [PAM94]. Furthermore, procedure calls, pointers, and loop bounds based on run-time values can hinder the compiler's effort to determine the best data placement for problems otherwise amenable to static analysis. Finally, the complexity of the compiler also increases. Given that parallelizing compilers already have to infer parallelism and generate communication, this complexity can be significant.

A third approach, used by Wholey in the ALEXI system [Who91], is to select the data placement at run time given compiler support. This is done by basing a static cost model for language primitives on the cost of machine primitives, and then using a hill-climbing heuristic executed at run time to determine the best from a set of data placements. Choosing a placement at run time eliminates difficulties caused by procedure calls, pointers, and run-time loop bounds. Furthermore, the ALEXI system will always choose the best data placement in its set because the language is explicitly parallel with well-defined communication costs for each statement. ALEXI has the same basic philosophy as Adapt: the best time to determine data placement is at run time. One significant difference between the systems is that Adapt allows a data placement to change over the course of an application. Furthermore, the explicitly-parallel programming model of ALEXI is completely different from that of Adapt; as Adapt uses a shared-memory programming model, without significant compiler support there is no way to determine the communication and computation costs of statements. Adapt also does not depend on a machine cost model.

6 Conclusion

We have presented an approach to adaptive data placement, implemented in the Adapt system, which places data at run time and allows the placement to adapt to the needs of the application. The performance of Adapt is almost as good as the performance of static schemes on problems for which a placement can be determined by the programmer or compiler. Moreover, the performance of Adapt is superior to any static scheme for problems that are impossible to analyze at compile time. Adapt is implemented on a cluster of Sparc-1s in less than 1000 lines of code, which makes it much simpler than the analysis required by compilers that attempt to infer data placements. Adapt supports a larger class of problems than any compiler approach, and it requires no help from the programmer in determining a data placement.

Adapt currently supports iterative scientific applications. To test scalability, we intend to run

Adapt programs on larger machines, including 8- and 16-node clusters. We are also developing more application programs, including those containing several phases, to provide a better understanding of their common characteristics. We will add new communication patterns to those currently recognized by Adapt, such as butterfly and replicated patterns. In addition, we are investigating the application of the Adapt model to compilers that explicitly generate communication rather than using a DSM.

References

[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, 1991.

[BO84] M.J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(484):482–512, 1984.

[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*, Tokyo, Japan, July 1993.

[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.

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

[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*, October 1990.

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

[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, 1994.

[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.

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

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