# An Adaptive Approach to Data Placement

David K. Lowenthal
Gregory R. Andrews
Department of Computer Science
The University of Arizona
Tucson, AZ 85721
{dkl,greg}@cs.arizona.edu

## Abstract

*Programming distributed-memory machines requires careful placement of data to balance the computational load among the nodes and minimize 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. 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. Adapt frees the programmer from having to specify data placements, and it can use run-time information to find better placements than compilers. Moreover, Adapt automatically supports a "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 better performance than using any statically determined data placement.*

## 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 [9], or compiler analysis, such as the work reported in [1], [5], and [12]. 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.

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 [6]—for which the best data placement can vary over the course of the application [17].

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 that use 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; e.g., Adapt versions of Jacobi iteration and LU decomposition are only slightly slower than the best static counterparts, ranging from a best case of 1% slower to a worst case of 13% 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, and Section 4 presents performance results. Finally, Section 5 describes related work and makes a few 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 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. 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.

We assume that any node can reference any data element. We also assume the *owner-computes* rule [8]. This means each data element has an “owner”, which is the only node that updates the element; however, other nodes may reference the element.

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

Between these extremes are several feasible data placements. Adapt considers two—variable block and striped. A *variable block* placement allocates a contiguous set of approximately the same number of data elements on each node. In applications such as Jacobi iteration, the block sizes are equal, because each matrix cell contains a data element, the workload is balanced, and the communication is regular (this placement is called `BLOCK` in HPF). On the other hand, particle-in-cell codes [6] often require unequal block sizes, because matrix cells can contain any number of data elements (HPF has no equivalent placement).

A *striped* placement method allocates data elements cyclically to the nodes (called `CYCLIC` in HPF). 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. 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.

## 3. Adapt and its Implementation

The current Adapt prototype is implemented in concert with the Distributed Filaments (DF) software kernel [4], which uses a DSM for communication. The Adapt system dynamically selects one of the data placements described in Section 2. It is given some initial data placement by the programmer or compiler (the current default is `BLOCK`) and then employs three steps to determine whether this placement is a good one or whether it should be changed. First, Adapt gathers information about the communication pattern and computation time for each loop body in the application. Next, it uses this information to determine which data placement is likely to minimize both communication overhead and delay. Finally, it effects the new placement (if necessary) and continues to monitor the computation in case the amount of computation or communication later changes. Below we discuss how Adapt monitors the computation, determines a placement, and changes the placement.

## 3.1. Adapt Monitoring

Adapt gathers information about the communication and computation in each loop body. Adapt monitors communication using DSM page faults and the DSM page table. In particular, the system counts the number of messages that each node sends and receives during one iteration of the application program. From these counts—and architecture-specific measures of the times it takes to send pages between nodes and to service page requests—Adapt estimates the time due to communication overhead and message delay on each node.

Adapt determines the communication pattern by inspecting the pattern of page faults on arrays in the page table. Currently, Adapt recognizes two patterns: *nearest-neighbor* and *broadcast*. In the nearest-neighbor 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 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 loops that exhibit the following characteristics: (1) in the first loop one node writes to a subset of pages of the array, (2) in the second loop each node has a distinct subset of exclusive-access pages of the array, and (3) in the second loop all nodes read the subset of pages that were written in the first loop.

Adapt gathers information about computation time by instrumenting the application code to obtain the time each node spends accessing the data elements it owns. These times are combined at the next barrier synchronization point to obtain the total computation time.

## 3.2. Adapt Algorithm

After gathering communication and computation information for one iteration of an application, Adapt uses it to choose a good data placement. In particular, given the total computation time $T$ and the number of nodes $P$, the ratio $T/P$ represents the amount of computation each node should perform for a perfectly balanced load. Adapt examines different ways that rows could be mapped to nodes to achieve this ideal load. This is done using a simple bin-packing procedure, which in turn depends on the communication pattern detected during the monitoring phase. When the communication pattern is nearest-neighbor, 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 $T/P$. Adapt also investigates multiple-bin packings if the load is not sufficiently balanced. When the communication pattern is broadcast, the type of packing depends on the workload. If the history of loop execution times shows a constant workload, the same procedure as the one above is used. On the other hand, if the execution times are changing, Adapt uses $n/P$ bins on each node to effect a `CYCLIC` style placement.

## 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 [4] 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 during execution, as described in Section 4). 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 monitoring phase and repeat the algorithm described above to determine the new (if any) best placement.

## 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 for which it is possible to determine a good data placement 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) [4] program without the Adapt subsystem. (Sequential programs were virtually identical to the one-node DF programs.) Below, we briefly describe the three applications and present the results of runs on 1, 2, 4, and 8 Sparc-1 nodes connected by an Ethernet.

## 4.1. Jacobi Iteration

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

| 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 |

**Figure 1. Jacobi iteration,** $512 \times 512$**,** $\epsilon = 10^{-3}$

| Number of Nodes | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Adapt Time (BLOCK) (sec) | 547 | 322 | 210 | 185 |
| Adapt Time (CYCLIC) (sec) | 547 | 305 | 190 | 165 |
| DF Time, CYCLIC (sec) | 544 | 303 | 189 | 164 |

**Figure 2. LU decomposition,** $800 \times 800$**.**

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 two versions of Jacobi iteration are shown in Figure 1. The Adapt program initially uses BLOCK by default; after recognizing nearest-neighbor communication, Adapt runs the bin-packing algorithm, which approximately reproduces the BLOCK placement. 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.

## 4.2. LU Decomposition

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 LU decomposition are shown in Figure 2. The first program, Adapt BLOCK, initially uses a block placement. 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 uses a variable block placement. However, the system quickly detects an imbalanced load, re-enabling the monitoring phase. At this point Adapt detects a decreasing workload and changes to a striped placement. The second program, Adapt CYCLIC, initially uses a cyclic placement. The difference between Adapt BLOCK and the DF CYCLIC program is the cost of the extra page faults necessary to change the data placement at run time and the overhead of initially using a variable block placement. In the Adapt CYCLIC program, these overheads are not present and the performance is very close to the DF program.

| 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, BC($n/2P$) (sec) | 69.1 | 47.0 | 39.1 | 25.3 |
| DF Time, BC($n/4P$) (sec) | 69.1 | 48.5 | 34.2 | 26.2 |
| DF Time, BC($n/8P$) (sec) | 69.1 | 46.5 | 39.3 | 42.6 |

**Figure 3. Particle simulation, grid** $64 \times 64$**, 150 particles. (BC is short for the Fortran D style BLOCKCYCLIC.)**

## 4.3. Particle Simulation

Our particle simulation program models the behavior of MP3D [15]. We use a two-dimensional grid of space cells and parameterize the movement of particles to facilitate experimentation. Although our implementation simplifies the physics involved, the computational structure is the same as MP3D.

This application is representative of programs where a good data placement depends on information that is 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 re-balance the load.

In our experiement, the application tended to move the particles to the upper region of the grid[1]. Figure 3 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.

## 5. Related Work and Conclusion

Data placement can be supported by language-level primitives, compilers, or (less commonly) run-time systems.

[1] This clustering is not contrived; it can occur in practice [6].

With language primitives, the programmer annotates each array with a placement (e.g. [9, 7, 18, 21, 3]). 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 so, the best placement might change 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. [1, 12, 5, 2, 10, 14, 13, 19]). Hence, the programmer need not be involved in placing data. However, a compiler may not be able to infer the best placement, especially for a dynamic computation.

With a run-time system approach, such as Adapt, ALEXI [20], and CHAOS [11], 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. Other methods to remap data at run time have been studied [16], but involve user intervention.

We have presented an approach to data placement that allows the placement to adapt to the needs of the application. Adapt supports a larger class of problems than compiler approaches and it requires no help from the programmer in determining a data placement. 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 or require run-time support.

## 6. Acknowledgements

Vince Freeh provided assistance on several technical and written aspects of this work. This research was supported by NSF grants CCR-9415303 and CDA-8822652.

## References

[1] 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, June 1993.

[2] 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, Apr. 1991.

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

[4] V. W. Freeh, D. K. Lowenthal, and G. 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, Nov. 1994.

[5] 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.

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

[7] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng. An overview of the Fortran-D programming system. Report TR91121, CRPC, Mar. 1991.

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

[9] High Performance Fortran language specification. Oct. 1993.

[10] D. E. Hudak and S. 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, Sept. 1990.

[11] Y. Hwang, B. Moon, S. D. Sharma, R. Ponnusamy, R. Das, and J. H. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed-memory machines. *Software—Practice and Experience*, 25(6):597–621, June 1995.

[12] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. Technical Report CRPC-TR94498-S, Rice University, Dec. 1994.

[13] 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, Feb. 1990.

[14] 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, Aug. 1991.

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

[16] B. Moon and J. Szltz. Adaptive runtime support for direct simulation monte carlo methods on distributed memory architectures. In *Proceedings of the Scalable High Performance Computing Conference*, pages 176–183, May 1994.

[17] D. S. Pande, D. P. Agrawal, and J. Mauney. Compiling functional parallelism on distributed-memory systems. *IEEE Parallel and Distributed Technology*, 1(1):64–76, Apr. 1994.

[18] M. Rosing, R. Schnabel, and R. Weaver. The Dino parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, Sept. 1991.

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

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

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