A-MPI: Supporting MPI on a Nondedicated Cluster of Workstations

by

D. Brent Weatherly

(Under the direction of David Lowenthal)

Abstract

Distributing data is one of the fundamental problems in implementing efficient distributed-memory parallel programs. The problem becomes more difficult in environments where the participating nodes (processors) are not dedicated to a parallel application. Such environments increase the difficulty of the data distribution problem, which is to determine an assignment of data elements to each node to minimize completion time. We are investigating this problem in the context of explicit message-passing programs.

We have designed and implemented an extension to the popular Message Passing Interface (MPI) that efficiently supports adaptive programs by providing the necessary infrastructure to redistribute data dynamically. Our system, called the Adaptive Message Passing Interface (A-MPI), contributes (1) an efficient memory allocation mechanism, (2) techniques for accurately determining system load and computation time, and (3) a heuristic for determining efficient data distributions, including the removal of nodes whose participation degrades the performance of an application. Performance results show that programs that use A-MPI can produce significant improvements over previous load-balancing systems.

Index words: Load Balancing, MPI, Data Distribution, Parallel Programming

A-MPI: Supporting MPI on a Nondedicated Cluster of Workstations

by

D. Brent Weatherly

B.S., The University of Georgia, 1994

A.B., The University of Georgia, 1995

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2002

© 2002
D. Brent Weatherly
All Rights Reserved

A-MPI: Supporting MPI on a Nondedicated Cluster of Workstations

by

D. Brent Weatherly

Approved:

Major Professor: David Lowenthal

Committee: John Miller
Eileen Kraemer

Electronic Version Approved:

Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
August 2002

Acknowledgments

First and foremost, I must acknowledge and thank my wife Kim, who provided both the encouragement and financial support for me to return for this degree. Without her having made a great deal of sacrifices, I would have never accomplished this goal. I must also thank David Lowenthal for teaching me all about parallel systems, for believing that I was up to the challenge of this project, for his wisdom and guidance, and for making the entire process enjoyable (mostly). I'd like to thank John Miller for teaching me a great deal about databases and for offering me research opportunities early on, even though these did not work out as hoped. Finally, I must thank Eileen Kraemer for getting me into the "swing" of things (Java pun) when I first started and for teaching me about human-computer interaction; I utilize this practical knowledge almost every day.

Table of Contents

Page

Acknowledgments iv

List of Figures vii

List of Tables x

Chapter

1 Introduction 1

2 Overview 4

2.1 Programming Model 4

2.2 Computational Model 6

2.3 A-MPI Strategy 8

3 Related Work 14

4 Implementation 16

4.1 Memory Allocation 16

4.2 Load Determination and Computation Timing 19

4.3 Work Distribution 22

4.4 Physical Redistribution 29

4.5 Post-redistribution Monitoring 30

4.6 Physical Removal of Nodes 31

5 Performance 34

5.1 Memory Allocation 35

5.2 Synthetic nearest-neighbor application 37

5.3 Synthetic reduce-broadcast application 42

5.4 Redistribution Costs 46

5.5 Node Removal 49

5.6 Unbalanced Computations 51

6 Conclusion and Future Work 54

Bibliography 56

List of Figures

2.1 Example User Code (Sequential Computation) 5

2.2 Example User Code (Initialization) 6

2.3 Example User Code (Computation) 7

2.4 Data ownership between 2 nodes for a shared array. The left-hand side image shows the ownership for phase 0. The right-hand side image for phase 1. Note that communication is necessary between phases to arrive at the distribution for phase 1. 8

2.5 Updated communication code using relative ranks 12

4.1 Comparison of memory allocation methods. Note: shaded elements must be reallocated. 18

4.2 Memory allocation tree 19

4.3 Time determination with `gethrtime` on a single-phase application with competing processes present. All measurements are in milliseconds and denote the wall-clock time to execute each of the 8 iterations of the phase. The boxes in bold denote those iterations that are affected by the execution of the competing processes. 21

4.4 Graph of workload percentages for a node with 1 competing process in a 2-node configuration. Each point indicates the percentage of work given to the loaded node. The top line shows the fixed percentage determined by a naive distribution, the bottom line the ideal percentage determined experimentally. Notice that as the amount of communication increases, the ideal workload percentage decreases. 24

4.5 Pseudocode for determining ideal work proportions 25

4.6 Graph of workload percentages based on the ratio of computation to communication for a node with 1 competing process in a 2-node configuration. The line labelled *Experimental Average* shows the measured results from the micro-benchmark, while *linear trend* denotes a linear function that smooths the experimental results. 26

4.7 Successive Balancing 27

4.8 `BLOCK` example of iteration and data distributions. The DRSDs describe, for a phase, {start, end, step}. The arrows denote the communication of data elements `2` and `3` from node P_0 to P_1 during redistribution. Note: data ownership reflects two read accesses: (`A[i]`) and (`A[i+1]`), where `A` is the array and `i` is the iteration number. 30

4.9 Example of global communication involving removed nodes. P_7 is physically removed and does not participate in the Send In. It does, however, participate in the Send Out. 32

5.1 Graph of memory allocation, shifting, and update costs for a three-dimensional array of size 256 (square matrix). We use a log scale for the y-axis because of the varying orders of magnitude of the measurements. All times are in milliseconds. 35

5.2(a) Graphs of results of the synthetic nearest-neighbor tests when one processor has 1 or 2 competing processes. The top row shows 2-node graphs, the middle row 4-node graphs, and the bottom row 8-node graphs. The load configuration is shown in parentheses (e.g., (2-{0}) means 2 competing processes on one node and none on the remaining nodes). 39

5.2(b) Graphs of results of the synthetic nearest-neighbor tests when one processor has 3 or 4 competing processes. See Figure 5.2(a) for layout details. 40

5.2(c) Graphs of results of the synthetic nearest-neighbor tests when two or more processors have 1 or more competing processes. We show two examples each for a 4-Node configuration and an 8-node configuration. 43

5.3 Graphs of the synthetic reduce-broadcast tests. The left-hand side column shows the results when the root node is loaded. The right-hand side column shows the results when a non-root node is loaded. 44

5.4 Comparison of the execution of Jacobi iteration. The results are for a 4-node configuration, and the arrays are doubles of dimension 2048x2048. Also, *Period* denotes the execution of Jacobi, *Grace Period* denotes the monitoring period of A-MPI, and *Redist* denotes redistribution. 46

5.5 Results of SOR tests when (1) a node has one, two, or three competing processes and a distribution that includes the loaded node is used (labelled *1 CP*, *2 CP*, or *3 CP*) and (2) the loaded node is removed (labelled *Drop*). Note that successive balancing is used to determine distributions that include the loaded node. Also, the times shown are for average phase cycle execution time after redistribution. Configurations of 8 and 16 nodes were used for arrays of dimension 512x512 and 1024x1024. 49

5.6 Results of the particle simulation tests where the computation is unbalanced. The left graph compares the results when redistribution does and does not take place. The right graph shows the effectiveness of using 5 phase cycle iterations (compared with 1 iteration) to determine true, unloaded iteration times before A-MPI determines a new distribution. All results are for an 8-node configuration with a grid dimension of 256x256. The label *Part* denotes the degree of imbalance as determined by the number of particles in each grid cell in the top half of the rows owned by P_0 52

List of Tables

5.1 Table of memory allocation, shifting, and update costs. The arrays are three-dimensional of the size shown for Dim. All times are in seconds. . . . 36

5.2 Table of redistribution costs when the loaded node is removed. *Load Redist Cost* shows the redistribution time when the node is loaded at the time it is removed; *Unload Redist Cost* the time when the node is not loaded when removed. One distributed, two-dimensional array of the dimension shown is redistributed each time (i.e. 512x512). All times are in seconds. 48

Chapter 1

Introduction

Distributed parallel architectures, such as clusters of workstations, deliver high performance and scale better than shared-memory machines. Two programming models exist for distributed-parallel architectures: shared variables, through a software distributed shared memory, and explicit message-passing, generally through libraries. Though a shared variable model is easier to program, usually for best performance one must use explicit message passing. One popular message-passing library is the Message Passing Interface (MPI) library, which is portable across a variety of distributed-memory machines.

While writing MPI applications is challenging, one typical simplifying assumption is that the nodes on the target machine are dedicated to the application. However, many computing environments are instead *nondedicated*, meaning that multiple users may be competing for nodes while the parallel application is executing. These users can be running sequential or parallel programs. Such computing environments are often found in national labs or department-wide clusters.

A nondedicated target increases the difficulty of the *data distribution problem*, which is to determine an assignment of the elements of each data structure to each node to minimize completion time. An optimal data distribution satisfies two conflicting goals: communication is minimized and computational load is balanced. In general, balancing the load on a nondedicated machine is more difficult because the amount of CPU time allocated to a parallel program fluctuates throughout its lifetime.

We have addressed this problem by designing and implementing what we call the Adaptive Message Passing Interface (A-MPI), which is an extension to MPI. A programmer need only write an A-MPI program; the A-MPI run-time system combines several novel features that serve as an infrastructure for writing *fully automatic*, *adaptive* message passing programs.

- A-MPI implements a 2-D projection memory allocation method that supports efficient redistribution, balancing the goals of minimizing both the number of communication messages and the copying of data.

- A-MPI implements techniques for accurately determining both system load and execution time, which involves obtaining unloaded iteration execution times.

- A-MPI uses micro-benchmarks and a heuristic, called *successive balancing*, to determine effective distributions. These often differ from the naive distribution, which ignores the effects of competing processes on communication and scheduling.

- A-MPI removes nodes from the computation when their communication overhead exceeds their computational benefit.

Our experiments show that our 2-D projection memory allocation method is considerably more efficient for redistribution than an allocation method that allocates memory in a single, contiguous chunk. In addition, a distribution determined by successive balancing results in up to 28% performance improvement over a distribution that considers *only* system load. Also, when node removal is necessary, physical removal of a node results in up to 56% performance improvement over simply giving the node the minimum amount of work.

The remainder of this thesis is organized as follows: Chapter 2 gives background information, and Chapter 3 describes related work. Chapter 4 details our implementation. Chapter 5 presents the results of performance tests. Finally, Chapter 6 gives some concluding remarks and suggestions for future work.

Chapter 2

Overview

In this chapter we describe the programming and computational models supported by our system. We then discuss the strategy for our implementation.

2.1 Programming Model

We use a distributed-memory programming model that is implemented with MPI (Message Passing Interface). Our goal is to extend the existing model of message-passing programs so that our system is both effective and easy to use. However, building a load-balancing system on this model requires us to address two key issues: memory allocation and communication. We provide the user code for a sample application for reference throughout this section: Figure 2.1 shows the sequential computation, Figures 2.2 shows the A-MPI initialization portion, and 2.3 shows the parallel computation.

We require that memory for all shared data be allocated by A-MPI. This allows us to effect new data distributions efficiently and automatically without requiring additional instrumentation to user code. For efficiency, we support both a *local view* of the data, where memory is only allocated for the data that is actually accessed by each node, and a *global view*, where memory is allocated for each entire shared array. However, in this thesis we focus exclusively on the local view of data. To support automatic redistribution, `MPI_AMPI_init_array` replaces the use of `calloc` or `malloc` to allocate memory (note its use in Figure 2.2). If redistribution takes place,

```
void compute() {
  for (pc = 0; pc < num_cycle_iters; pc++) {
      // COMPUTATION
      for (i = 1; i <= N; i++) {
          for (j = 1; j <= N; j++)
            A[i][j] = F( B, i, j );
      }
  }
}
```

Figure 2.1: Example User Code (Sequential Computation)

user data is moved transparently. However, the user code must obtain explicitly the bounds of the distributed (outermost) loop of each phase. In the sample code of Figure 2.3, the functions `MPI_AMPI_get_start_iter` and `MPI_AMPI_get_end_iter` are used for this purpose.

On the other hand, because we use a distributed-memory programming model, we assume that the communication necessary to maintain data dependencies is performed explicitly by the application code. Figure 2.4 shows an example of a shared array whose ownership differs in phase 0 and phase 1. The application must explicitly perform the communication of the ownership changes shown (i.e., node P_1 must send half of its data to P_0). The alternative is for A-MPI to perform this communication automatically. However, this would instead impose a shared-memory programming model, which is beyond the scope of our work. This decision does impact how distributions of iterations are determined (see Section 2.3.2).

```
int main( int argc, char **argv ) {
  // regular MPI initialization omitted

  MPI_AMPI_init( num_processors, 1, 2, MPI_AMPI_BLOCK, MPI_AMPI_LOCAL);

  MPI_AMPI_init_array("A", &A, 1, N, sizeof(double), MPI_DOUBLE );
  MPI_AMPI_init_array("B", &B, 1, N, sizeof(double), MPI_DOUBLE );

  MPI_AMPI_init_phase(1, N, MPI_AMPI_NEAREST_NEIGHBOR );

  MPI_AMPI_add_array_access("A", MPI_AMPI_WRITE, 1, 0);
  MPI_AMPI_add_array_access("B", MPI_AMPI_READ, 1, 0);

  MPI_AMPI_init_done();

  compute();  // computation code
}
```

Figure 2.2: Example User Code (Initialization)

2.2 Computational Model

Our system supports the Single Program Multiple Data (SPMD) computational model, in which each node executes the same code but references a different subset of distributed data. SPMD is a relaxation of the Single Instruction Multiple Data (SIMD) model. We assume that applications are iterative and consist of one or more *phases*, which are sections of code comprised of computation followed by communication (or vice-versa). Further, the iterations of the outermost phase loop are distributed among the available nodes; thus, shared data is distributed at the level of the first dimension for each array. Our model assumes that applications use either a variable block distribution, where a contiguous (but possibly unequal) set of iterations are assigned to each node, or a cyclic distribution, where iterations are assigned

```
void compute() {
  for (pc = 0; pc < num_cycle_iters; pc++) {
    MPI_AMPI_start_cycle();
      start_iter = MPI_AMPI_get_start_iter();
      end_iter = MPI_AMPI_get_end_iter();

      // COMPUTATION
      for (i = start_iter; i <= end_iter; i++) {
        MPI_AMPI_start_timer( i );
          for (j = 1; j <= N; j++)
            A[i][j] = F( B, i, j );
        MPI_AMPI_end_timer( i );
      }

      // COMMUNICATION
      if (rank > 0)
        MPI_Send(B[start_iter], 1, MPI_DOUBLE, rank - 1, ..... );
      if (rank < num_processors)
        MPI_Send(B[end_iter], 1, MPI_DOUBLE, rank + 1, .....  );

      if (rank < num_processors)
        MPI_Recv(B[end_iter + 1], 1, MPI_DOUBLE, rank + 1, ..... );
      if (rank > 0)
        MPI_Recv(B[start_iter - 1], 1, MPI_DOUBLE, rank - 1, ..... );
    MPI_AMPI_end_cycle();
  }
}
```

Figure 2.3: Example User Code (Computation)

to nodes in a modulo fashion. Figure 2.3 shows a single phase example. Note that the `i` loop is distributed. Lastly, we assume the existence of an outer loop, the *phase cycle*, that encloses all phases. In the example, the `pc` loop is the phase cycle loop.

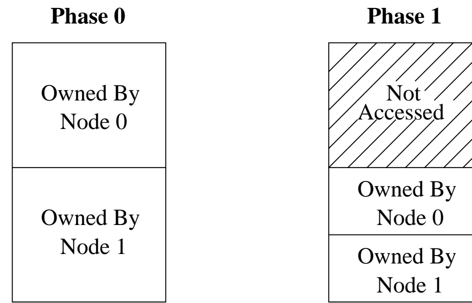

Figure 2.4: Data ownership between 2 nodes for a shared array. The left-hand side image shows the ownership for phase 0. The right-hand side image for phase 1. Note that communication is necessary between phases to arrive at the distribution for phase 1.

2.3 A-MPI Strategy

In this section, we highlight the key decisions that were made in the design of A-MPI. We first describe the motivation and strategy for determining *work proportions* (Section 2.3.1) that result in good performance. We then describe our models for determining data ownership and distributions of iterations (Section 2.3.2). Finally, we discuss the motivation and strategy for node removal (Section 2.3.3).

2.3.1 Work Proportions

Work proportions describe distributions by assigning a percentage of work to each node. A-MPI calculates work proportions using the number of *competing processes* on each node and the execution time of the iterations of the outermost loop of each phase, where we assume a competing process is compute-bound. Our motivation is to support both *balanced* and *unbalanced* computations. We define a balanced computation as one in which the iterations have an approximately equal cost. On the other hand, if the computation is nonuniform, we classify the computation as unbalanced.

Most work in this area considers only balanced computations; iterations are distributed according to the *relative power* of nodes. The schemes for determining relative processor power are either *wallclock-based* or *percentage-based*. In a wallclock-based scheme [IRSD99, LA96, RD01], each participating node measures the time taken for each node to execute the same code, denoted T_i. The relative power of node i is then $max_j(T_j)/T_i$.

A percentage-based scheme estimates the percentage of time that the parallel application is actually executing. Each node uses the statistics kept by the OS (CPU utilization and the number of ready processes) to estimate the CPU percentage for the application. The relative power is then calculated by dividing the CPU percentage of a node by the minimum CPU percentage among all nodes.

Using the relative power obtained by either scheme, the iterations (and data) are assigned to the nodes such that the load is balanced. In particular, if node i is twice as powerful as node j, then node i receives twice as much work as node j. Unfortunately, these schemes do not support unbalanced computations because they do not consider the actual cost of each iteration when distributing work. Furthermore, our tests have shown that these schemes are impractical in general for balanced computations as well because they do not account for the communication that takes place.

Our solution is to model the execution of a variety of applications when competing processes are present and use this information to determine *ideal* work proportions based on predicted behavior. In addition, A-MPI obtains the true, unloaded execution time for the iterations of the outermost loop of each phase. Used together, A-MPI distributes work in ideal proportions based on the actual cost of each iteration. Hence, we support both balanced and unbalanced computations. Sections 4.2 and 4.3 contain additional details.

2.3.2 Work Distribution and Data Ownership

The decision to use a distributed-memory programming model requires us to distribute work according to a single, global distribution of iterations and effect the *initial* data ownership for the entire phase cycle whenever redistribution takes place. This differs from a *global data distribution* [ML01], which assigns one distribution to each phase and follows from the requirement that all communication be performed by the application. If A-MPI creates different iteration distributions for different phases, data may change ownership during the phase cycle without the necessary communication being performed by the application. On the other hand, if the user *intends* for data to change ownership during the phase cycle, we must effect the initial ownership of the phase cycle to ensure that the explicit communication is correct. For example, if the data distribution shown in Figure 2.4 is determined during redistribution, A-MPI must schedule the appropriate communication such that the ownership in phase 0 is effected. This will ensure that the inter-phase communication scheduled by the user works properly. In general, we assume a multiple-reader, single-writer sharing model, so initial ownership is determined as the first node (over all phases) that updates data or the one or more nodes that access read-only data.

A-MPI uses Deferred Regular Section Descriptors (DRSDs) [ML01] to describe array accesses. These DRSDs enable A-MPI to determine data ownership. DRSDs were created in order to extend Regular Section Descriptors (RSDs) [HK91]; both express an array reference in terms of the start, end, and step. However, DRSDs defer the computation of the bounds of these descriptors until run-time. We extend the solution for block *and* cyclic distributions by specifying a list of DRSDs for each node and each phase, where a block distribution has one list and a cyclic distribution has one or more. However, whereas originally a compiler automatically generated DRSDs, we currently require the user to specify the access type and the

sequential step and offset for every array access of every phase (see Figure 2.2, `MPI_AMPI_add_array_access`). A-MPI then generates the DRSDs using this information and the iteration distributions. In future work, we could modify the MPI compiler to generate DRSDs automatically.

2.3.3 Removal of Nodes

Depending on the amount of computation and communication, as well as the number of available processors, a parallel application may perform better when loaded nodes are removed from the computation. However, there are two ways that a node can be removed: *logical removal* and *physical removal*. As we will discuss, logical removal is simpler but in general physical removal is more efficient.

Logical removal of nodes is the simplest solution in that it requires no changes to any communication routines. To logically remove a node, the load-balancing system assigns the minimum amount of work to the loaded nodes (one iteration). When the DRSDs are calculated, the start and end will specify the same iteration. The reason that *some* work is required is that the removed nodes continue to participate in all communication—the node must therefore own some data. For example, if a removed node participates in a nearest neighbor exchange of shared data, its neighbors will schedule communication to send data to and receive data from the removed node. For large computations, one iteration is essentially no work, and hence, the node is logically removed. Unfortunately, our tests have shown that communication and/or scheduling can severely impact the performance of the application if nodes are loaded (see Section 5.2). Because of this, it is best to physically remove loaded nodes when removal is appropriate.

Physical removal is not just a matter of terminating the application on the node, because we may want it to rejoin the computation if the competing processes terminate. Our solution is to (1) remove the node from any parallel computation, (2)

```
if MPI_AMPI_participating() {
  // COMPUTATION
  ....No Changes

  // COMMUNICATION
  rel_rank = MPI_AMPI_get_rel_rank( rank );
  if (rel_rank > 0)
    MPI_AMPI_Send(B[start_iter], 1, MPI_DOUBLE, rel_rank - 1, ..... );
  if (rel_rank < MPI_AMPI_get_num_active())
    MPI_AMPI_Send(B[end_iter], 1, MPI_DOUBLE, rel_rank + 1, .....  );

  if (rel_rank < MPI_AMPI_get_num_active())
    MPI_AMPI_Recv(B[end_iter + 1], 1, MPI_DOUBLE, rel_rank + 1, ..... );
  if (rel_rank > 0)
    MPI_AMPI_Recv(B[start_iter - 1], 1, MPI_DOUBLE, rel_rank - 1, ..... );
}
```

Figure 2.5: Updated communication code using relative ranks

remove the node from any neighbor communication, and (3) allow removed nodes to *receive only* during global communication. In this way, removed nodes *do* continue execution of sequential sections [EAS+97] and global communication in order to maintain program correctness; however, they do not delay participating nodes (see Section 4.6).

We introduce the concept of *relative rank* as an extension of the use of processor rank in MPI programs. With A-MPI, only the participating processors are given a relative rank. Figure 2.5 shows the modified computation section of the sample application. The function `MPI_AMPI_participating` allows nodes to execute or skip the parallel section depending on whether or not they have been removed. In addition, we modify the appropriate MPI communication routines to support relative ranks. In the code, the relative rank is maintained via the function `MPI_AMPI_get_rel_rank` and is used in the A-MPI communication routines

`MPI_AMPI_Send` and `MPI_AMPI_Recv`. Since the relative rank can change as a result of redistribution, the updated value *must* be used every phase cycle.

Chapter 3

Related Work

There have been three primary approaches to data distribution: language annotations, compiler analysis, and run-time adaptation. We discuss them in turn.

One way to distribute data is to provide language annotations and allow the programmer to choose the distribution using application-specific knowledge. This is the approach taken by HPF [HPF94], which was motivated by many others' work [HKT91, RSW91, ZBG88]. In this approach, the programmer annotates each array with its distribution.

Compiler techniques to distribute data have also been studied extensively (e.g., [LC90, Soc91, BFKK91, GB93, RN95]). The basic idea behind compiler-based systems is to analyze the source code to determine the communication pattern and then choose a `block`- or `cyclic`-based distribution that balances the load. There has also been research on compilers that can generate dynamic data distributions; these include [AL93, PB95, KK98, PB95, GAL96].

Approaches employing a run-time system, such as ALEXI [Who91], Dame [CC96], CHAOS [HMS+95], AppLeS [SWB97], SUIF-Adapt [ML01, LA96], CRAUL [RD01, ID98], and the CHAOS group [ESS96] can use run-time information to find an efficient data distribution. This is especially effective in cases where workload and communication characteristics of a program change at run time. In principle, these approaches show promise to solve the data distribution problem, because multiple machine parameters are changing. In particular, through the Network

Weather Service [WSH99], AppLeS was able to determine when to avoid using a processor because its limited memory would cause (expensive) paging as well as when the computation/communication was low. Also, the CHAOS work as well as that in [SLGZ99] can drop nodes when required (e.g., users log back on to their workstation), as well as add them back when conditions change. Other methods to remap data at run time have been studied [MS94], but involve user intervention.

Techniques to predict execution time based on micro-benchmarks (a training set) were pioneered by Balasundaram et al [BFKK91]. In this work, execution times are measured for an applicable set of computation and communication operations, enabling a compiler to predict the behavior of applications and optimize accordingly.

Chapter 4

Implementation

The implementation of A-MPI addresses several fundamental issues. First, A-MPI needs to carefully allocate memory for efficient redistribution, balancing the tradeoffs between minimizing (1) the number of messages required for internode communication and (2) copying of data (Section 4.1). Second, A-MPI must accurately measure both the load on the system and the execution time of iterations (Section 4.2), but this is complicated by the fact that system load varies, which can result in unexpected measurements using wallclock timers. Third, A-MPI must determine ideal work proportions for a collection of nodes of varying available processing power (Section 4.3); however, the impact of communication makes this problem difficult. Fourth, a single global distribution must be found in multiple phase programs (Section 4.4), and the redistribution must be effected automatically. Fifth, A-MPI must monitor the application after redistribution (Section 4.5) and remove the loaded nodes if the application performs poorly (Section 4.6).

4.1 Memory Allocation

A-MPI's memory allocation mechanism balances (1) the need to maximize data locality and minimize the number of redistribution messages with (2) the desire to eliminate excess memory allocation and deallocation, which we call *memory reuse*[1].

[1]Memory reuse is typically associated with memory management. We are redefining the term for our purposes.

Balancing these issues is difficult because maximizing data locality requires allocating memory for all data elements in a contiguous block of memory. However, while this method allows nodes to exchange data with a single message, it also necessitates complete memory reallocation whenever redistribution takes place; data that have not changed ownership must be copied into the new memory explicitly. The left-hand side image of Figure 4.1 shows a redistribution example for a node using the all-contiguous method. In the example, the node's ownership increases from 6 elements to 9 elements. A neighbor sends 3 elements that are copied into the new memory automatically by MPI. However, the other elements do not change ownership; hence, they must be copied manually (6 updates).

On the other hand, one could allocate memory using a vector of vectors approach, which would facilitate memory reuse. However, all rows would thus be allocated separately, compromising data locality and either (1) requiring a message for every row or (2) adding overhead for data copying if the data is packed into a single message. For large, multi-dimensional arrays, the overhead of paging and message passing would far outweigh the benefit of memory reuse. Our solution is to project all arrays onto a two-dimensional array: the first dimension is the outermost dimension of the original n dimension array, while the second dimension consists of *extended rows* of the remaining elements (the product of the remaining n-1 dimensions). This solution allows nodes to communicate entire extended rows with a single message and allows memory to be reused where possible by allowing a newly allocated pointer in the first dimension to point to existing memory. The right-hand side image of Figure 4.1 shows an example of memory reuse assuming the 2-D projection method. Instead of copying every element, as in the previous example, the node merely updates the pointer from the outermost dimension.

Figure 4.2 shows an example of allocating a four dimensional (2x2x2x4) array. The first dimension, the leftmost block in the figure, is the distributed dimension and

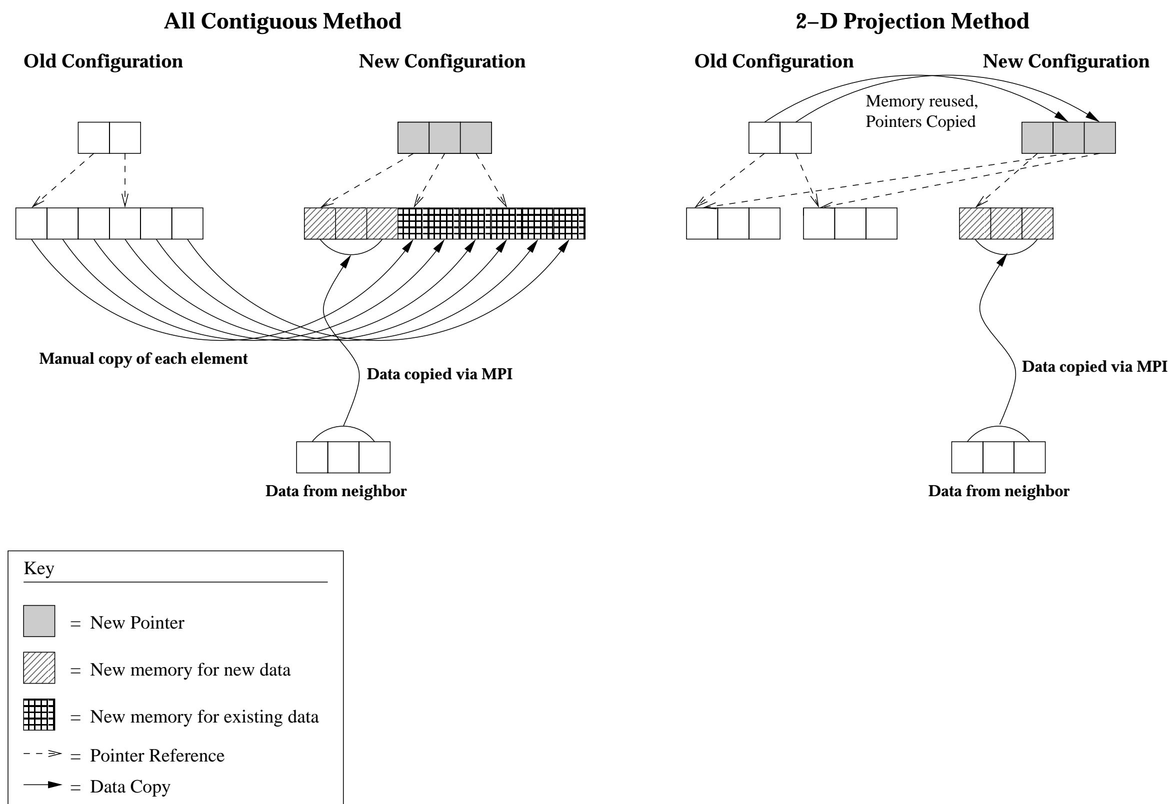

Figure 4.1: Comparison of memory allocation methods. Note: shaded elements must be reallocated.

is allocated separately. The figure then shows two examples of extended rows, organized as *memory trees*. A memory tree is a generalization of the most compact way to dynamically allocate a two-dimensional array. To create the memory tree, A-MPI first works forward in dimensionality (i.e., from 2, 3, etc.), calculating the product of the last n-1 dimensions and storing the total sizes calculated at each dimension. When finished, the total number of calculated elements of the n-1 dimensions is used to determine the size of the contiguous chunk of memory for each row. In the figure, the bottommost blocks of 16 elements constitute the contiguous chunk of memory for the example. A-MPI then works backwards in dimensionality (i.e., from n-2, n-3,

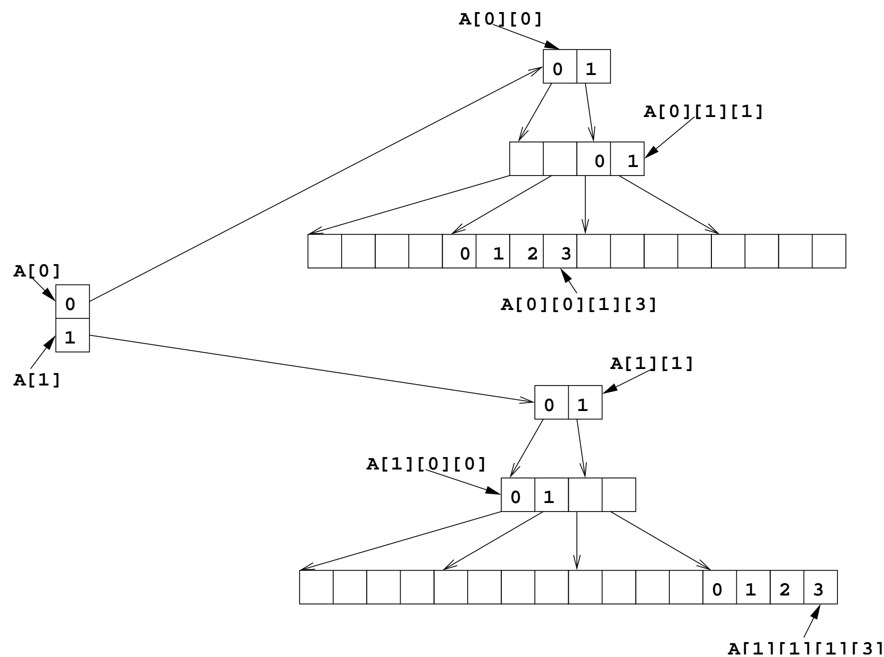

Figure 4.2: Memory allocation tree

etc.) allocating memory for the indirection pointers using the sizes stored previously and pointing them to the proper elements in the data chunk. After all memory for data and indirection pointers is allocated, the address of the topmost indirection block is returned and assigned to the appropriate pointer in the first dimension.

4.2 Load Determination and Computation Timing

In order to determine when redistribution is necessary, A-MPI must monitor all nodes to detect load changes and the execution time of the phase cycle to determine if the load is balanced. Our policy is to (1) check system load every phase cycle and redistribute if any change is detected and (2) measure the execution time of the phase cycle at regular intervals and redistribute if the performance degrades past a

tolerance. We currently check for a load imbalance every 20 phase cycle iterations, but this interval is arbitrary and can be changed to suit a particular application.

Determining system load is a non-trivial problem. Previous work in this area utilized `vmstat` to determine the number of active processes on a node [WSH99], the essential information being the number of processes on the running, ready or blocked queues. However, our tests have proven this method to be unreliable because `vmstat` cannot account for processes that are blocked on a receive; these processes have voluntarily relinquished the processor but are not on the queues on which `vmstat` reports. As a result, a monitored application may or may not be included when `vmstat` reports system load, and a poor distribution results if the measurement is incorrect. We have created an application called `ampi_ps` that uses `ps` to determine active processes, and we account for only those processes that are in either a running or ready state. Our policy is to *always* exclude the monitored application. Hence, if it is on the running queue, it is not counted. We configure `ampi_ps` to run on each node as a daemon process that updates every second.

Once the redistribution process is invoked by a change in load, A-MPI must carefully determine the load on all nodes and the true, unloaded execution time of iterations in order to calculate ideal work proportions. Our solution allows for a grace period, called the *monitoring period*, where the application continues execution for five phase cycle iterations after a change in load is detected. During this time, processor load and accurate iteration times are determined.

A-MPI calculates processor load as the median load from all measurements taken during the monitoring period. This method is necessary because a single measurement may reflect a temporary spike, which could result in a sub-optimal distribution when the load stabilizes. As mentioned above, A-MPI takes a measurement every second using `ampi_ps`.

	Cycle Iter 1	Cycle Iter 2	Cycle Iter 3	Cycle Iter 4	Cycle Iter 5	Minimum
1	8.55	8.55	8.33	8.37	8.37	8.33
2	8.08	8.08	8.00	8.01	8.01	8.00
3	8.06	8.06	**208.04**	8.04	8.02	8.02
4	**208.08**	8.08	8.07	8.05	8.07	8.05
5	8.03	8.04	8.04	**408.06**	8.04	8.03
6	8.05	**407.99**	7.96	8.02	**167.99**	7.96
7	8.05	8.04	8.03	8.04	8.05	8.03
8	8.07	8.04	**208.07**	8.05	8.04	8.04

Figure 4.3: Time determination with `gethrtime` on a single-phase application with competing processes present. All measurements are in milliseconds and denote the wall-clock time to execute each of the 8 iterations of the phase. The boxes in bold denote those iterations that are affected by the execution of the competing processes.

During the monitoring period, A-MPI uses two timing mechanisms to determine the true unloaded execution time for iterations: information from `/PROC` and clock timers using `gethrtime`. Our solution is to use `/PROC` for iterations with execution times greater than 10ms. Otherwise, the minimum time measured by `gethrtime` for each iteration is used. Both mechanisms are necessary because they each have drawbacks that the other overcomes: `/PROC` is not accurate below 10ms but reflects only the execution time of a process, whereas `gethrtime` is accurate to the microsecond but includes the time that a process spends in every state (i.e. running, blocked or ready). The use of `gethrtime` can thus be inaccurate when competing processes are present because iteration times may include the time that the application is de-scheduled during interleaved execution. Figure 4.3 shows an example of the time measured for 8 iterations for each phase cycle of the monitoring period. The bold boxes indicate incorrect iteration times.

Fortunately, `gethrtime` can be used for a 10ms iteration (on Solaris) if multiple readings are taken. Because 10ms is much less than the minimum quantum (40ms), a

context switch can only take place at a *minimum* of every 4 iterations of each phase. Hence, we know that some iterations are not interrupted during each phase cycle, ensuring that at least some measurements will be accurate. Furthermore, because of changes in scheduling, the probability is low that the same iteration of a phase is context switched on consecutive phase cycle iterations. As a result, we can expect to obtain an accurate measurement for every phase iteration after some number of phase cycle iterations. In our experiments, taking the minimum time measured over five phase cycle iterations with `gethrtime` successfully determined the unloaded execution time for all iterations. The right-hand side image of Figure 4.3 shows the correctly measured time for each iteration (8 ms).

4.3 Work Distribution

Once system load and iteration times are known, A-MPI calculates the proportion of work to give to each node based on the load on the processors and the ratio of computation to communication in the phase. We use data from two-node, single-phase benchmark tests and a heuristic to support multiple nodes and phases. This model is necessary because communication must be considered when balancing the computation.

4.3.1 Micro-benchmarks

The traditional way to determine work proportions is to base it solely on the relative powers of each processor. For example, for a two-node configuration with one competing process on P_0 (load = 2) and no competing processes on P_1 (load = 1), P_1 is twice as powerful as P_0, and the naive distribution gives 1/3 the work to P_0 and the remaining 2/3 to P_1. Unfortunately, this method can result in sub-optimal distributions by not accounting for the communication that takes place. Our tests

have shown that optimal distributions often differ from naive distributions and are application dependent. As a result, our approach is to simulate the execution of real applications in order to develop a generic model that determines ideal work proportions. Hence, our model determines distributions that often differ from the naive distribution. This section has two parts: in Section 4.3.1 we discuss the micro-benchmarks that were used to determine the ideal proportions of work for two-node, single-phase configurations and unloaded communication times. We detail in Section 4.3.2 an algorithm called *successive balancing* to extend the model to support multiple nodes and then explain how A-MPI supports multi-phase applications.

A-MPI uses a two-node, single-phase model to calculate work proportions for a variety of programs. We use this model because it is straightforward, and furthermore, enumerating every configuration is impossible. The challenge is understanding how competing processes affect an application that alternates periods of computation and communication. As an example, Figure 4.4 shows the results of a two-node test with 1 competing process on one of the nodes for a fixed computation time (200ms). The y-axis shows the ideal percentage of work given to the loaded node. As the communication increases, the ideal percentage decreases. We attribute this behavior to the fact that distributing iterations (work) does not take into consideration the computation portion of communication (i.e., communication requires *some* use of the CPU). Hence, communication can be affected by competing processes, and this slowdown becomes more of a concern as the amount of communication increases. We have found that given a number of competing processes, similar ratios of computation to communication for different applications result in similar ideal work proportions regardless of the actual values of computation and communication.

As a result, our main benchmark determines an ideal percentage of work based only on ratio and load. For this reason we test a variety of ratios in order to support different applications. This also allows our model to scale as well, for the *two-*

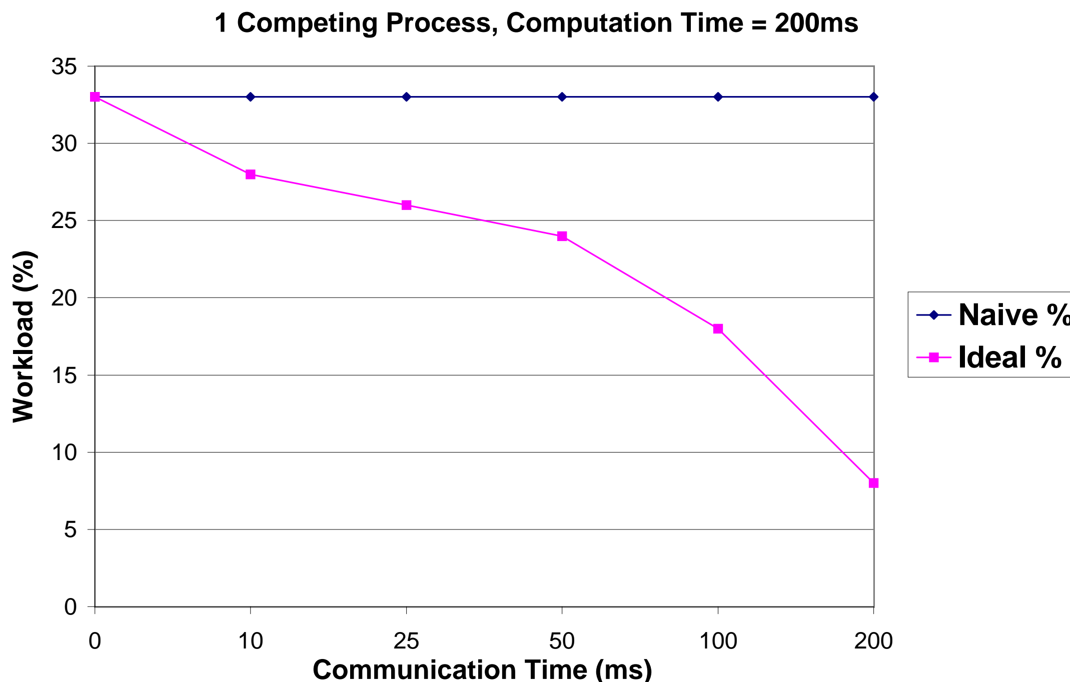

Figure 4.4: Graph of workload percentages for a node with 1 competing process in a 2-node configuration. Each point indicates the percentage of work given to the loaded node. The top line shows the fixed percentage determined by a naive distribution, the bottom line the ideal percentage determined experimentally. Notice that as the amount of communication increases, the ideal workload percentage decreases.

node ratio of computation to communication varies with the number of available processors—the more nodes available, the less work given to any two nodes.

Figure 4.5 shows pseudocode for the benchmark. The main idea is to test different numbers of competing processes (currently 1 to 5) and a variety of ratios in order to create a function that predicts the ideal percentage for any ratio and a particular load. To ensure that our model is not biased by a single test, we tested different amounts of communication and computation for a given ratio. Each test was then run five times in order to establish an expected (median) result. The actual experiment assigns varying proportions of work to the loaded and unloaded processors to find the percentage that results in the minimum execution time. Thus, for

```
for number_competing = 1 to 5 {
    start_competers(num_competing)
    for ratio = 1 to 10 step .25 {
        for comm_time = (10ms, 25ms, 50ms, 100ms, 200ms, 300ms) {
            total_comp_time = comm_time * ratio
            for repeat = 1 to 5 {
                for percentage = "naive" downto 0 {
                    if loaded
                        comp_time = percentage * total_comp_time
                    else
                        comp_time = (100-percentage) * total_comp_time
                    start_timer()
                        compute(comp_time)
                        communicate(comm_time)
                    end_timer()
                    update_min(end_time - start_time)
                }
                store_min(repeat)
            }
            find_median(store_min, comm_time)
        }
    }
    calculate_avg_percentage(find_median)
}
```

Figure 4.5: Pseudocode for determining ideal work proportions

each ratio we arrive at many similar percentages; the average of these is used as the ideal percentage. Finally, we graph the ideal percentage (y-axis) based on the ratio (x-axis) and calculate a linear trend function for each load. Figure 4.6 shows an example graph for 1 competing process and ratios ranging from 1 to 10. In the graph, the line labelled *Experimental Average* shows the measured results from the micro-benchmark, while *Linear Trend* denotes the linear function. These functions are called *workload functions* and take as input the ratio of computation to communication.

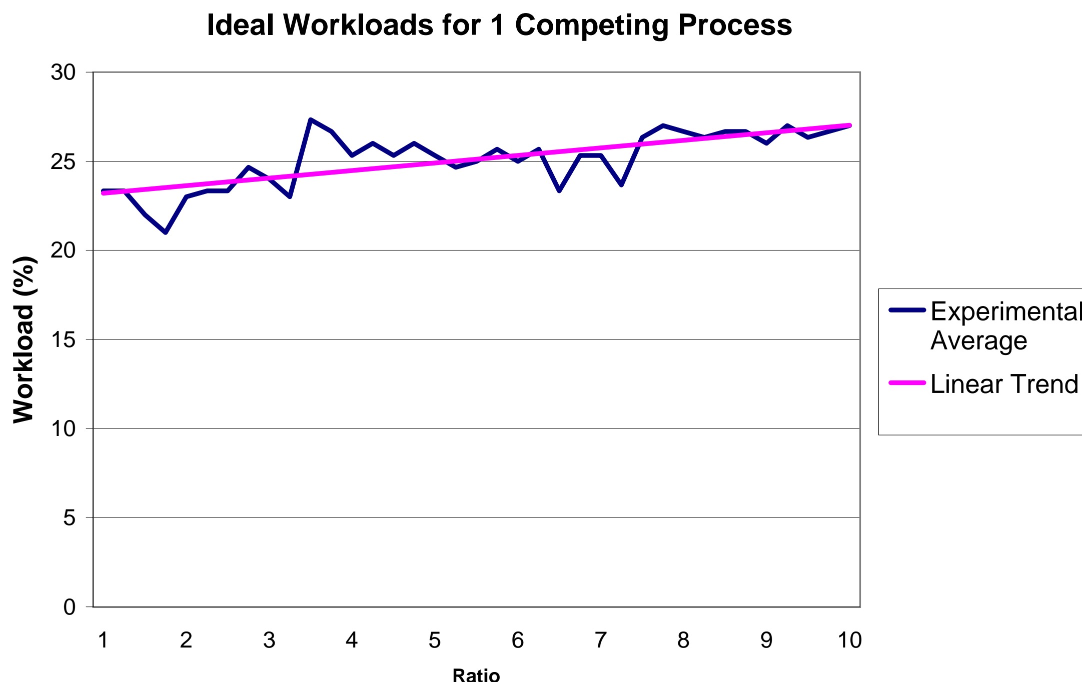

Figure 4.6: Graph of workload percentages based on the ratio of computation to communication for a node with 1 competing process in a 2-node configuration. The line labelled *Experimental Average* shows the measured results from the micro-benchmark, while *linear trend* denotes a linear function that smooths the experimental results.

To use the workload function, A-MPI must know the ratio of computation to communication for the phase. The computation time can be determined by summing the unloaded iteration times. However, obtaining unloaded communication time dynamically is beyond the scope of this thesis because it is impacted by many factors (number of nodes, load on other nodes, network load, etc.) Instead, we have performed benchmark tests to model the supported communication types. Using this information, we create trend functions that allow A-MPI to predict communication time based on communication type and message size.

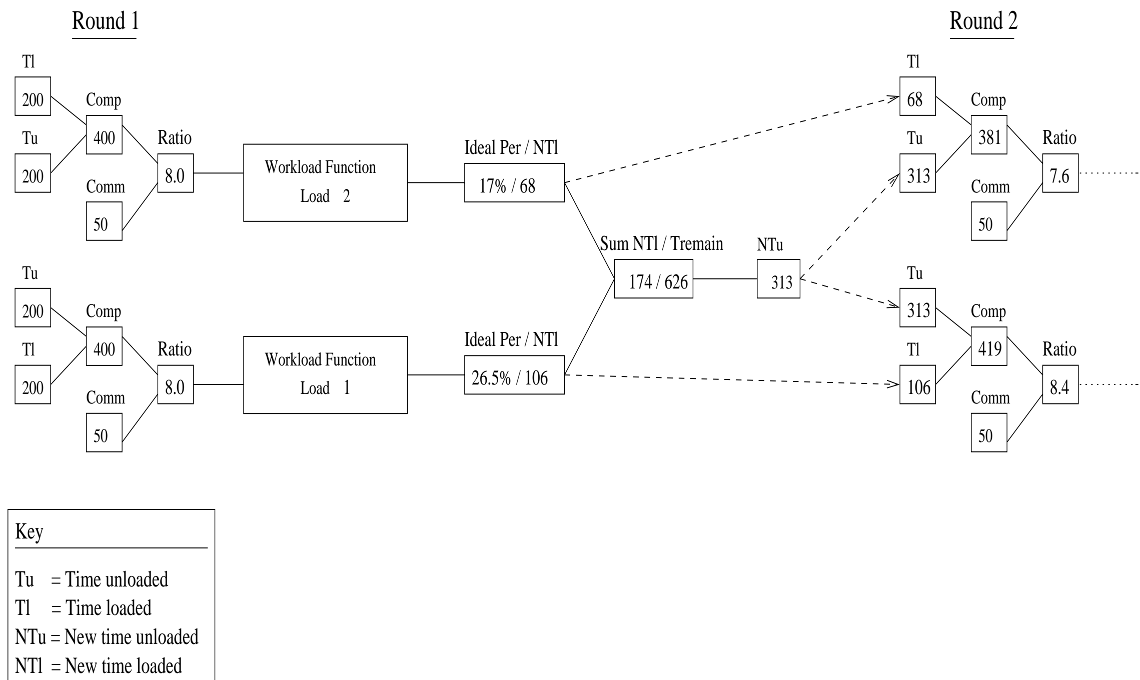

Figure 4.7: Successive Balancing

4.3.2 Successive Balancing

To extend the two-node, single-phase model to multiple nodes, A-MPI uses *successive balancing*. This determines the percentage of work to assign to nodes between pairs of loaded and unloaded nodes, thus reducing a multi-node problem to several two-node problems. Essentially, successive balancing entails one or more balancing *rounds*, where the workload assignment is calculated for the loaded nodes and the remaining computation balanced among the unloaded nodes. Balancing continues until a round results in little or no change to the iteration assignment to the unloaded nodes.

Figure 4.7 shows a 4-node example of successive balancing for a synthetic phase with 800ms of computation and 50ms of communication. In the example, one node (P_0) has 2 competing processes and another (P_1) has 1. Each round entails pairing the

loaded nodes with an unloaded node and summing their assigned computation times (note, in round 1, A-MPI assumes an equal distribution, so each node gets 200ms of work). Next, A-MPI calculates the ratio of computation to communication and uses the workload function (created as described in Section 4.3.1) for the appropriate load to determine the ideal percentage of work to assign to the loaded node. In round 1, both ratios are 8.0, and the workload function returns 17% and 26.5% for P_0 and P_1 respectively. A-MPI then calculates the new work assignment for the loaded nodes using the ideal percentage and each pair's computation time (in the example, 17% of 400ms gives P_0 68ms and 26.5% of 400ms gives P_1 106ms). The new loaded times are subtracted from the total computation time of the phase to determine the work that will be evenly distributed among the unloaded nodes. In the example, the unloaded nodes are assigned a total of 626ms, which averages to 313 ms for the two unloaded nodes. Round 2 follows in the same fashion, except that the new computation times from round 1 are used as the starting computation times. Successive balancing is complete when the difference between the starting and ending computation assignments for a round is within a tolerance.

To extend this method to multi-phase programs, A-MPI selects the phase that contributes most to the total execution time of the program and uses the single-phase model above. This is a straightforward approach, and further research is needed to determine its effectiveness. An alternative is to solve the global distribution problem, but this would necessitate a change to the programming model (as discussed in Section 2.3.2). Another approach is to use successive balancing for each phase to determine the minimum and maximum proportions that can be given to each node. One could then iteratively adjust these percentages (between the minimum and maximum) and predict the execution time based on the percentage of work given to the unloaded nodes. In so doing, the ideal proportions can be found exhaustively. However, the goal of our model is to ensure that all loaded nodes reach communication

points before the unloaded ones; based on our experiments, this results on average in the fastest execution. If these percentages are adjusted so that loaded nodes arrive late at communication points, the scheduling changes such that the application is more affected by the competing processes.

4.4 Physical Redistribution

Once the ideal proportion of work for each node is determined, A-MPI sums the times for iterations over all phases to create a single, global list of iteration times for the entire phase cycle. Typically, creating this list is simple because each phase has identical sequential loop descriptions. However, we support the general case where iterations may not exist in all phases. To this end, we treat the iterations of each phase as a set and create the global list as the union of these sets. Using this list, A-MPI either assigns a block of iterations (block distribution) or a number of sequences (cyclic distribution) based on the percentage of total work. Figure 4.8 shows an example of redistribution for a two-node, block configuration with 4 iterations. The figure shows the iteration assignments (as specified by the DRSDs) for both before and after redistribution.

To effect the new distribution, each node must (1) determine data ownership, (2) free memory that changes ownership, (3) allocate the required memory for new ownership, (4) update pointers for data that does not change ownership, and (5) schedule communication for data that does. Note that A-MPI's memory allocation method makes redistribution more efficient by minimizing the amount of memory allocated and allowing memory reuse. On the other hand, our method adds overhead if memory is successively freed and allocated. However, our tests have shown this results in negligible overhead, as the available blocks are automatically reused without the memory management system having to make any expensive system

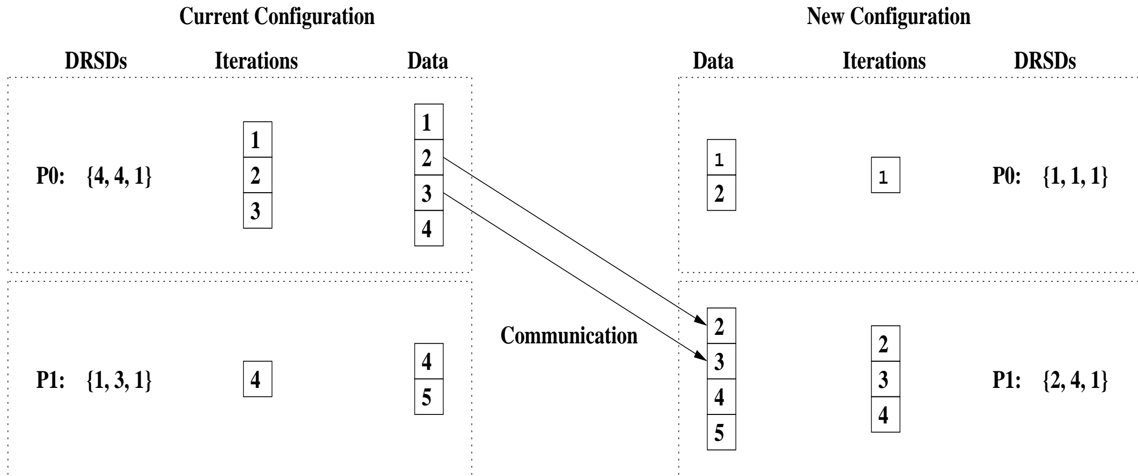

Figure 4.8: `BLOCK` example of iteration and data distributions. The DRSDs describe, for a phase, {start, end, step}. The arrows denote the communication of data elements `2` and `3` from node P_0 to P_1 during redistribution. Note: data ownership reflects two read accesses: (`A[i]`) and (`A[i+1]`), where `A` is the array and `i` is the iteration number.

calls. The middle of Figure 4.8 shows the data ownership for the current and new distributions, as well as the communication necessary to effect the redistribution.

4.5 Post-redistribution Monitoring

After redistribution, A-MPI continues to monitor the application in order to determine if the new distribution is effective or if loaded nodes need to be removed from the computation. Ideally, we would (1) compare the actual run-time of the current configuration with the predicted run-time where the node with the most competing processes is removed and then (2) redistribute, if necessary. The subsequent monitoring period would then compare the run-time of the new configuration to that of a configuration where the next loaded node is removed, and so on. However, this would require A-MPI to accurately predict the execution time of configurations that

include loaded nodes, which our tests have shown to be extremely difficult. On the other hand, we *can* accurately predict the execution time of a configuration consisting of only unloaded nodes, and A-MPI uses this configuration to determine if the current configuration is performing poorly. As a result, A-MPI monitors each node in the current configuration during a post-redistribution grace period (currently ten phase cycle iterations). This grace period allows each node to determine the average execution time for a single phase cycle iteration. The maximum time of these averages among all nodes is then compared to the predicted unloaded execution time for a configuration consisting only of unloaded nodes. If we predict that the unloaded configuration is best, the loaded nodes are physically removed from the computation.

4.6 Physical Removal of Nodes

If A-MPI determines that the loaded nodes should be removed from the computation altogether, the load balancing process takes place again. However, successive balancing is unnecessary, as the work is redistributed equally among the unloaded nodes. To this end, A-MPI uses relative ranks to account only for those nodes that are participating. These relative ranks are assigned in an incremental manner that assigns consecutive ranks to the unloaded, participating nodes. Hence, a node's neighbors may change when nodes are removed.

The complication for physical removal of nodes involves keeping the nodes current on all global state information. By global state information we mean any information exchanged via global communication. For example, if the factors necessary to terminate an application are reached by an appropriate node and a termination signal is sent to all other nodes, removed nodes must receive this signal. At the same time, we do not want the participating nodes to be delayed by removed nodes or reach an

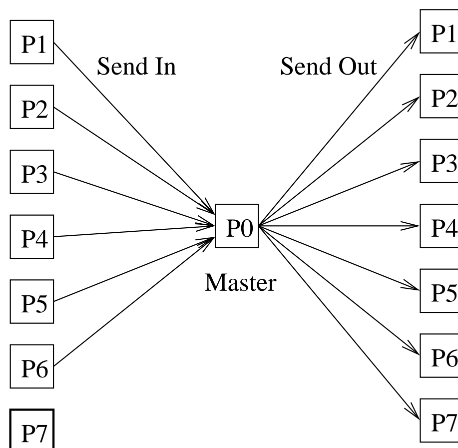

Figure 4.9: Example of global communication involving removed nodes. P_7 is physically removed and does not participate in the Send In. It does, however, participate in the Send Out.

incorrect state due to erroneous data having been received from removed nodes. To this end, we modify the appropriate global communication routines so that removed nodes participate in the *send out* of a global communication but not in the *send in*. Figure 4.9 shows an example of global communication. Note that node P_7 is physically removed from the computation and therefore does not send in. However, it does receive from the master node (P_0) during the send out.

Finally, we discuss the problem with essentially removing the loaded nodes from the inherent synchronization of global communication. Because physically removed nodes participate in the send out, they *cannot* race ahead of the execution of other nodes; removed nodes will block if they reach a global communication before the other nodes. However, it is possible for physically removed nodes to fall behind the execution of other nodes. As a result, the removed nodes may become one or more phase cycle iterations (and thus global messages) behind the participating nodes. This might happen if the execution time of the entire phase cycle is less than the

time that a removed node is de-scheduled while competing processes execute. This is unlikely to be a problem because in this scenario a removed node is essentially a communication-bound skeleton process [EAS+97] and will be given the highest priority by the scheduler. However, if this were to occur, the concern might be that the network buffer of the removed node could overflow. Fortunately, MPI automatically prevents this problem by blocking a sending node before the network buffer of a destination node overflows; this will allow removed nodes to catch up.

Chapter 5

Performance

This chapter details the performance of A-MPI. First, we compare the performance of A-MPI's memory allocation mechanism with that of the all-contiguous method (Section 5.1). Second, we show detailed performance results from a synthetic application (Section 5.2, where the computation is a spin loop and the communication is a nearest neighbor exchange. This application allows us to model a variety of different applications by specifying the amount of sequential computation and communication in order to test different ratios. Next, we report measurements from an additional synthetic application where the communication is a reduce/broadcast (Section 5.3). We then show the cost of redistribution with measurements from Jacobi iteration (Section 5.4). Next, we examine node removal closely and show results using a Red/Black Successive Over-Relaxation application (Section 5.5). Finally, we show measurements using a particle simulation to prove that our system supports unbalanced computations (Section 5.6).

Unless stated otherwise, all tests were run on 2, 4, or 8 processor configurations, where the computing environment consists of 550 MHz Pentium-III Xeon CPUs and a switched 100Mbs Ethernet network. All users programs are compiled with the `-O2` option. In addition, we use a block distribution for all examples. Where applicable, we use a simple application that infinitely loops as a competing process.

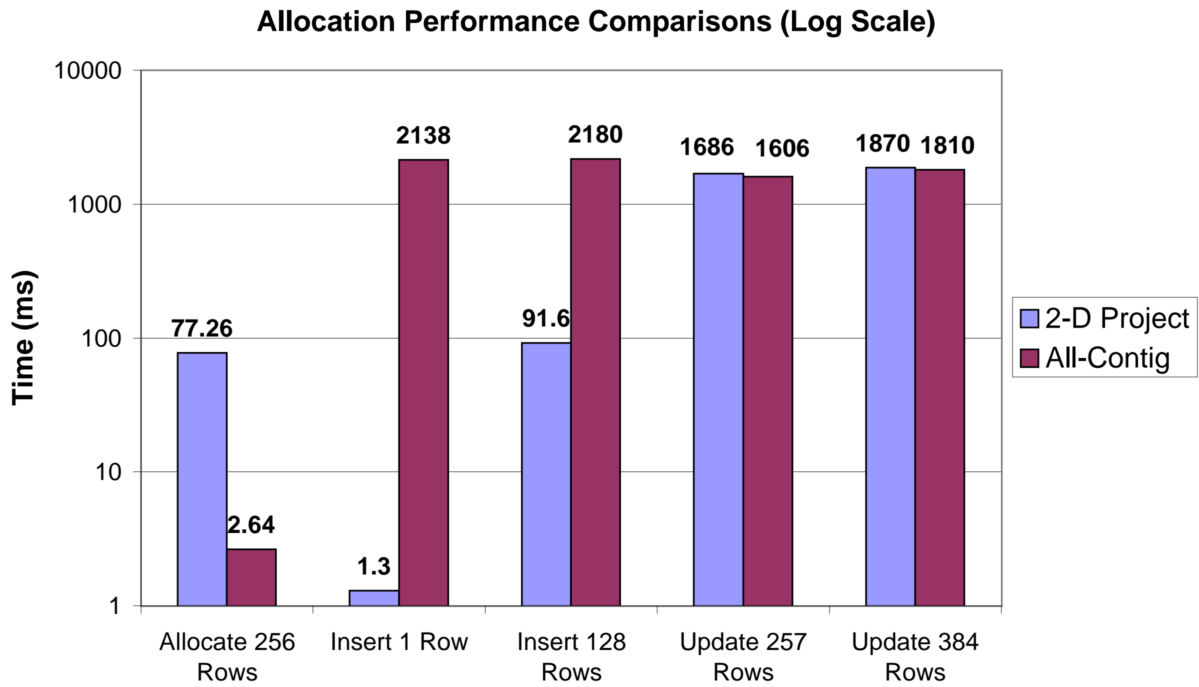

Figure 5.1: Graph of memory allocation, shifting, and update costs for a three-dimensional array of size 256 (square matrix). We use a log scale for the y-axis because of the varying orders of magnitude of the measurements. All times are in milliseconds.

5.1 Memory Allocation

As discussed in Section 4.1, A-MPI uses a 2-D projection method for memory allocation. We demonstrate the efficiency of this technique by comparing the performance of the 2-D projection method with the all-contiguous method. Figure 5.1 graphs the results (in milliseconds) of allocating, shifting, and updating a three-dimensional array of size 256 (square matrix). Table 5.1 shows these results (in seconds) and includes the measurements for arrays of size 128 and 360 as well. We used a stand-alone test application that allocates the necessary memory, performs a shift, and then updates every element of the matrix. For each array size, we *simulate* redistri-

Dim	Method	# Rows	Time to allocate	Time to shift	Time to update
128	2-D Project	1	0.0144	0.0004	0.0866
128	All-Contig	1	0.0008	0.2664	0.0789
128	2-D Project	64	0.0141	0.0094	0.1376
128	All-Contig	64	0.0008	0.2687	0.1334
256	2-D Project	1	0.0773	0.0013	1.686
256	All-Contig	1	0.0026	2.138	1.606
256	2-D Project	128	0.0776	0.0916	1.870
256	All-Contig	128	0.0027	2.180	1.810
360	2-D Project	1	0.3402	0.0029	4.335
360	All-Contig	1	0.0052	223.2	326.1
360	2-D Project	180	0.3304	0.5439	87.85
360	All-Contig	180	0.0052	145.3	261.9

Table 5.1: Table of memory allocation, shifting, and update costs. The arrays are three-dimensional of the size shown for Dim. All times are in seconds.

butions where either 1 row or half of the array is inserted at the beginning of the array; however, no actual communication takes place.

From the results, it can be seen that the 2-D projection method is always considerably faster than the all-contiguous method for data shifting. As a result, A-MPI can effect a new distribution more efficiently with the 2-D projection method than the all-contiguous method. Of note is the matrix of dimension 360. Notice from the table that the shift takes 223 seconds. By itself, this matrix is not out-of-core. However, by reallocating the entire matrix (plus one row) in order to copy every element, the application *becomes* out-of-core during redistribution.

On the other hand, the results also show that our method allocates and updates slightly slower than the all-contiguous method for most cases. Fortunately, the absolute time for allocation is small, and allocation only takes place once for any array. However, the update cost as a result of lost data-locality is unavoidable with the 2-D projection method. This cost is modest, though and is on the same order of magnitude as that of the all-contiguous method. For example, for the matrix of

dimension 256, the table shows that the absolute times for the 2-D projection and all-contiguous methods are 1.686s and 1.606s respectively (4.7% worse) for the 1 row test and 1.870s and 1.810s (3.2% worse) for the 128 row test. In fact, since most of the computation for supported applications is in nested loops, the use of an optimizing compiler should result in nearly equivalent update times. On the other hand, the example matrix of dimension 360 again shows how the 2-D projection method is useful. Once the all-contiguous application becomes out-of-core during redistribution, it will remain so until the memory pages for the old matrix are reclaimed. Thus the update time is much greater than that of the 2-D projection method for this example (e.g., for the 1 row test, the absolute times are 4.335s and 326.1s).

It can be argued that our method will eventually degrade the performance of an application with small matrices based on the accrued slowdown over many phase cycle iterations. However, A-MPI was designed as a tool for computing environments that change somewhat frequently, making efficient redistribution important. The results show that the slowdown associated with redistribution using the all-contiguous method is greater than that of updating with the 2-D projection method. As a result, as long as redistribution takes place occasionally, the speedup obtained during redistribution by the 2-D projection method should outweigh any slowdown incurred during data access.

5.2 Synthetic nearest-neighbor application

In order to show the performance of A-MPI for a variety of ratios of computation to communication, we use a synthetic, single-phase application. The computation portion of the program is simply a spin loop where the number of iterations is calculated according to the appropriate amount of computation time. Similarly, the communication is a nearest-neighbor exchange where the message size is calculated according

to the communication time. We tested 7 different ratios, each using different amounts of computation and communication. For each test, we start the desired number of competing processes on the appropriate nodes on the fifth phase cycle iteration. The monitoring period begins on the 10th iteration, with actual redistribution taking place on the 15th iteration. We then calculate the average execution time for a phase cycle.

We first ran tests on each processor configuration with 1, 2, 3, or 4 competing processes on one of the nodes. Figures 5.2(a) and 5.2(b) show the results for each of these tests. Each graph compares the performance of the synthetic application *after* redistribution using (1) the naive distribution (labelled Naive), (2) successive balancing (labelled A-MPI), (3) logical removal of the loaded node (labelled Log. Drop), and (4) physical removal of the loaded node (labelled Phys. Drop).

The key observation that we draw is that, as the ratio of computation to communication *decreases*, the likelihood of removing loaded nodes increases. On the other hand, as the ratio increases, a distribution (either Naive or A-MPI) that includes all nodes results in better performance. In addition, the results also show that removing nodes becomes more likely when (1) there are more processors available or (2) the number of competing processes increases. Thus, it is equally important to be able to find ideal distributions or remove loaded nodes, and the figure shows that in almost all cases our implementation (either using successive balancing or physically removing loaded nodes) results in the best performance.

First, we specifically consider the situation where all nodes remain in the computation. Balancing the load with successive balancing results in up to 28% improvement (Graph: *4 Nodes, 3*-{*0*}, Config: *Ratio=14*) over using the naive distribution. In fact, the naive distribution performs better in only 3 cases out of 84 tests (see *2 Nodes, (3*-{*0*}*)* and *4 Nodes, (3*-{*0*}*)*, *Ratios 160* and *300*) for an average improvement of 5.6%. Hence, we conclude that using a naive distribution is often a poor

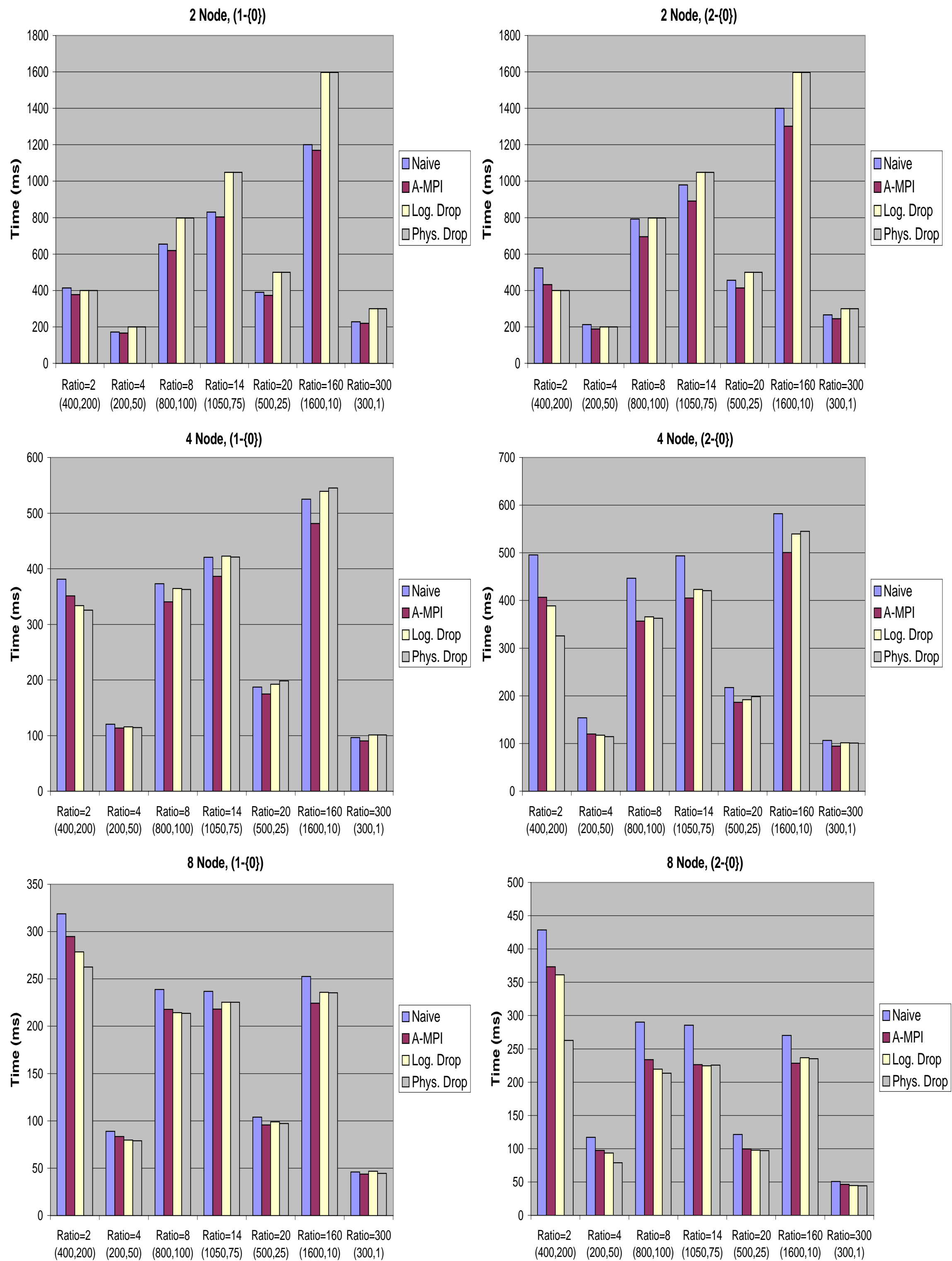

Figure 5.2(a): Graphs of results of the synthetic nearest-neighbor tests when one processor has 1 or 2 competing processes. The top row shows 2-node graphs, the middle row 4-node graphs, and the bottom row 8-node graphs. The load configuration is shown in parentheses (e.g., (2-{0}) means 2 competing processes on one node and none on the remaining nodes).

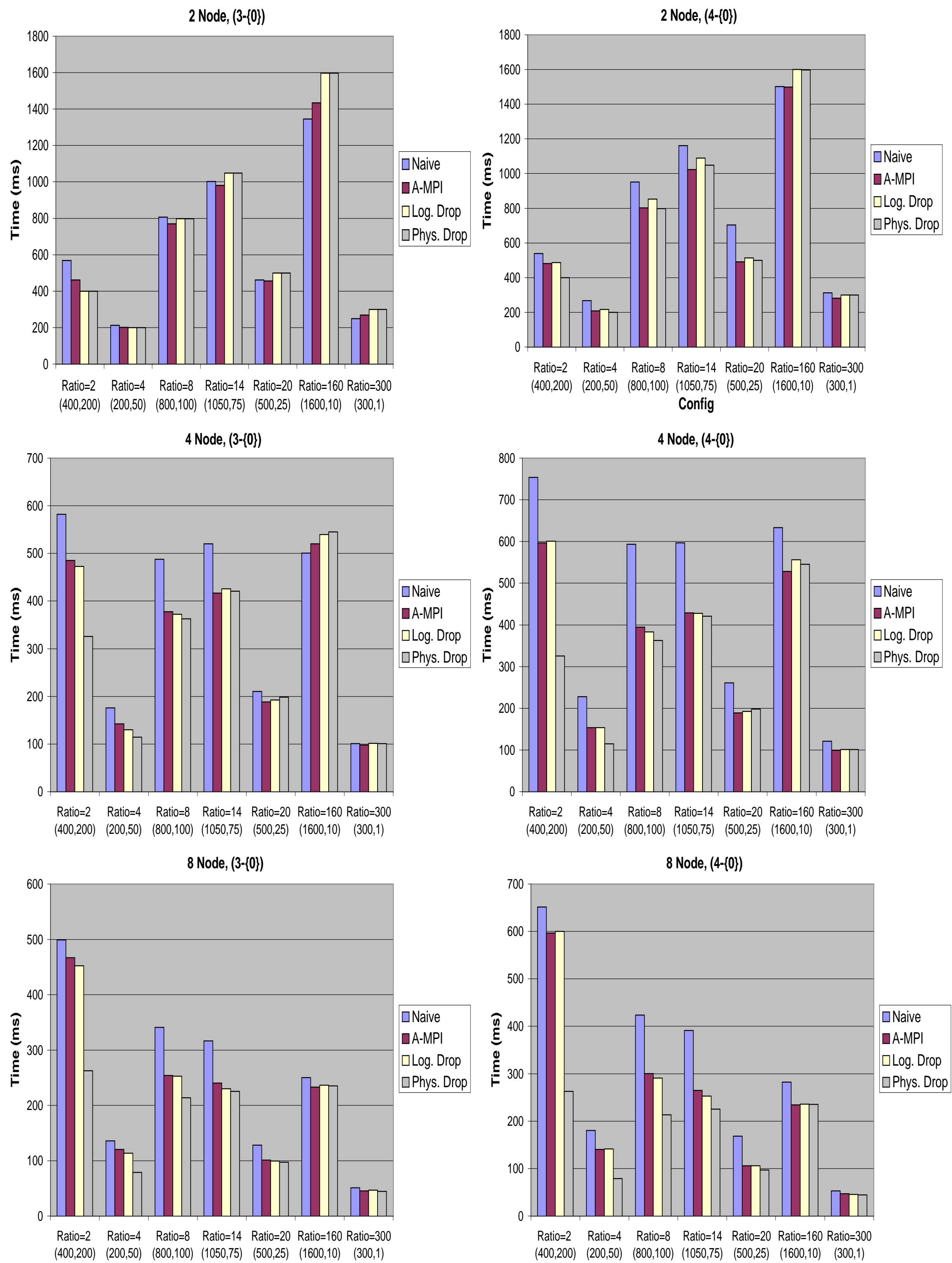

Figure 5.2(b): Graphs of results of the synthetic nearest-neighbor tests when one processor has 3 or 4 competing processes. See Figure 5.2(a) for layout details.

decision because too much work is given to a loaded node. This supports our contention that loaded nodes must reach communication points before unloaded nodes. While a naive distribution may create a scenario where all nodes finish a computation section at the same time (thus minimized), this may actually lead to poor performance in the communication.

Examining more closely one specific case where the naive redistribution outperforms successive balancing (*Ratio=160*, *4 Nodes, (3-{0})*), we found that the work distribution across the 4 nodes were: (30.77%, **7.69%**, 30.77%, 30.77%) for naive and (32.08%, **3.75%**, 32.08%, 32.08%) for A-MPI, where the percentages in bold are the proportions given to the loaded node. Notice that the proportion given to the loaded node by A-MPI is half as much as that of the naive distribution. Hand-coding A-MPI to test different distributions where the loaded node percentage ranged from 3.75 to 7.69 resulted in no better performance than the naive distribution. We conclude that the communication in this case *does not* negatively impact the computation and thus, the workload function for 3 competing processes finds an ideal percentage that is too small (i.e., too little work is given to the loaded node). In future work, we will extend the test ratios of the micro-benchmark in order to support these cases.

When loaded nodes are removed, the results show that in general it is best to physically remove nodes rather than logically remove them. This is because a logically removed node continues to participate in all communication and can slow down other nodes; if the ratio of computation to communication is small or there are several competing processes, communication involving loaded nodes negatively impacts the application. The difference can be significant: the 8 node example with 4 competing processes (Figure 5.2(b)) shows an improvement ranging from 56% for the *ratio=2* case to 10.8% for the *ratio=14* case for physically removing over logically removing the loaded node. Note that there are cases when logical removal is the same or slightly better than physical removal. However, this is rare and only

occurs in cases where the loaded nodes remain in the computation (e.g., *4 Nodes, 2-{0}*, *Ratio=20*). There are no cases where logical removal of the loaded nodes outperforms physical removal *when removal is best*.

Figure 5.2(c) shows four examples of the synthetic application where two or more nodes have one or more competing processes. We show these examples as evidence that A-MPI can handle a variety of configurations. In all cases, the best performance is the result of either balancing the load with successive balancing or physically removing the loaded nodes.

In summary, out of 112 experiments of the synthetic nearest-neighbor application, our strategy resulted in the best performance in 106 of them either using successive balancing or physically removing loaded nodes. Further comparison shows that in the 74 cases where all nodes remain in the computation, the distribution derived by successive balancing outperformed the naive distribution in 66 of 74 cases for an average improvement of 10.7%. In 2 of the 74 cases, the performance difference is less than 1% and is virtually equal. The naive distribution outperformed successive balancing in the remaining 6 cases, but the average improvement is only 4.3%. Of the 38 cases where removal is best, physical removal outperforms logical removal in 36 cases for an average improvement of 17.8%. The other 2 cases resulted in essentially equal performance.

5.3 Synthetic reduce-broadcast application

Next, we show the performance of A-MPI with a second synthetic application. Instead of a nearest-neighbor exchange, we test a reduce-broadcast communication. Because the test results show similar trends to that of the synthetic nearest-neighbor application, we show only the tests where one node has a single competing process.

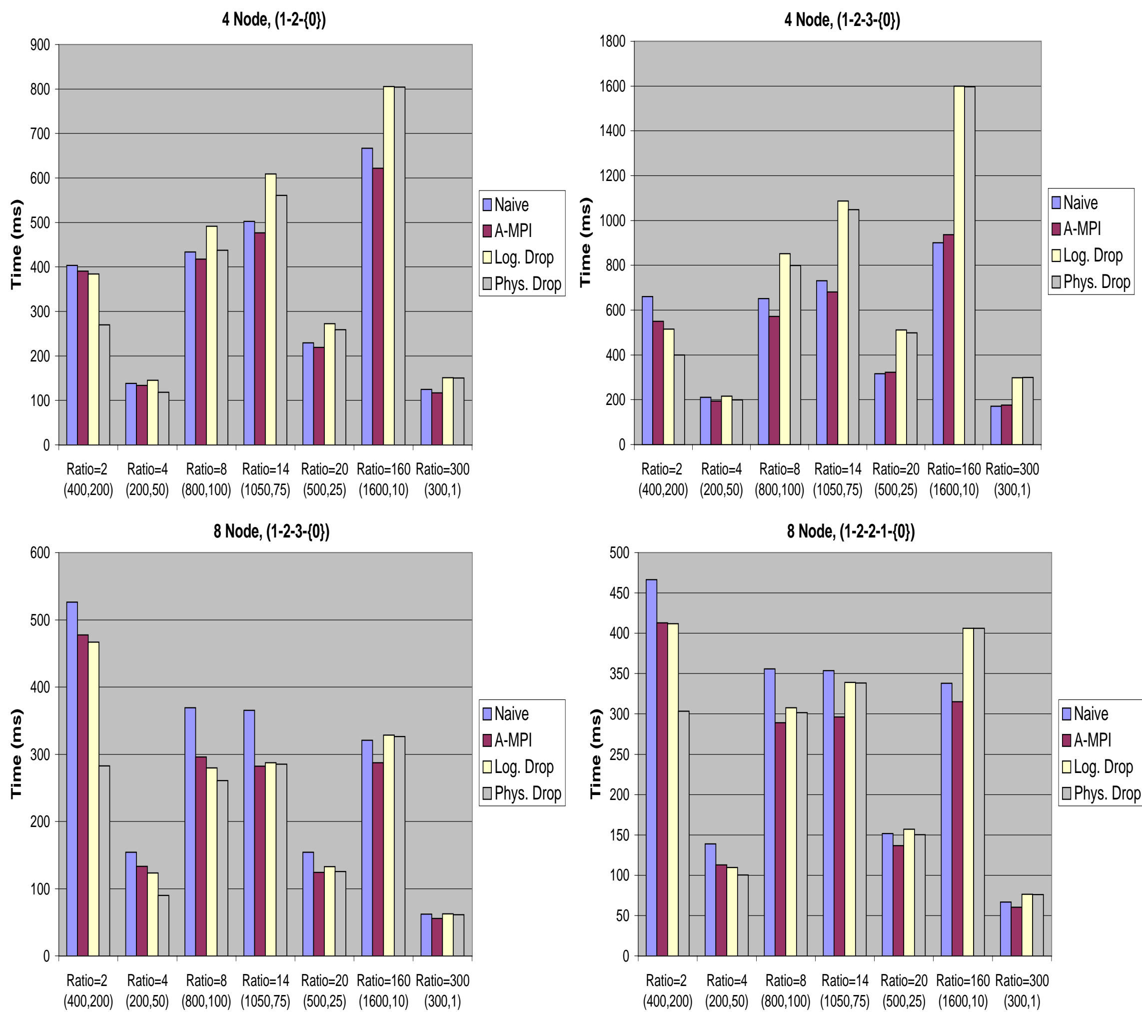

Figure 5.2(c): Graphs of results of the synthetic nearest-neighbor tests when two or more processors have 1 or more competing processes. We show two examples each for a 4-Node configuration and an 8-node configuration.

However, we must consider the fact that a reduce-broadcast is a global communication; a particular node, the root node, is responsible for receiving messages from all the other nodes, performing operations on the data, and sending the result(s) to the other nodes. Thus, we show tests where (1) the root node is loaded and (2) a non-root node is loaded.

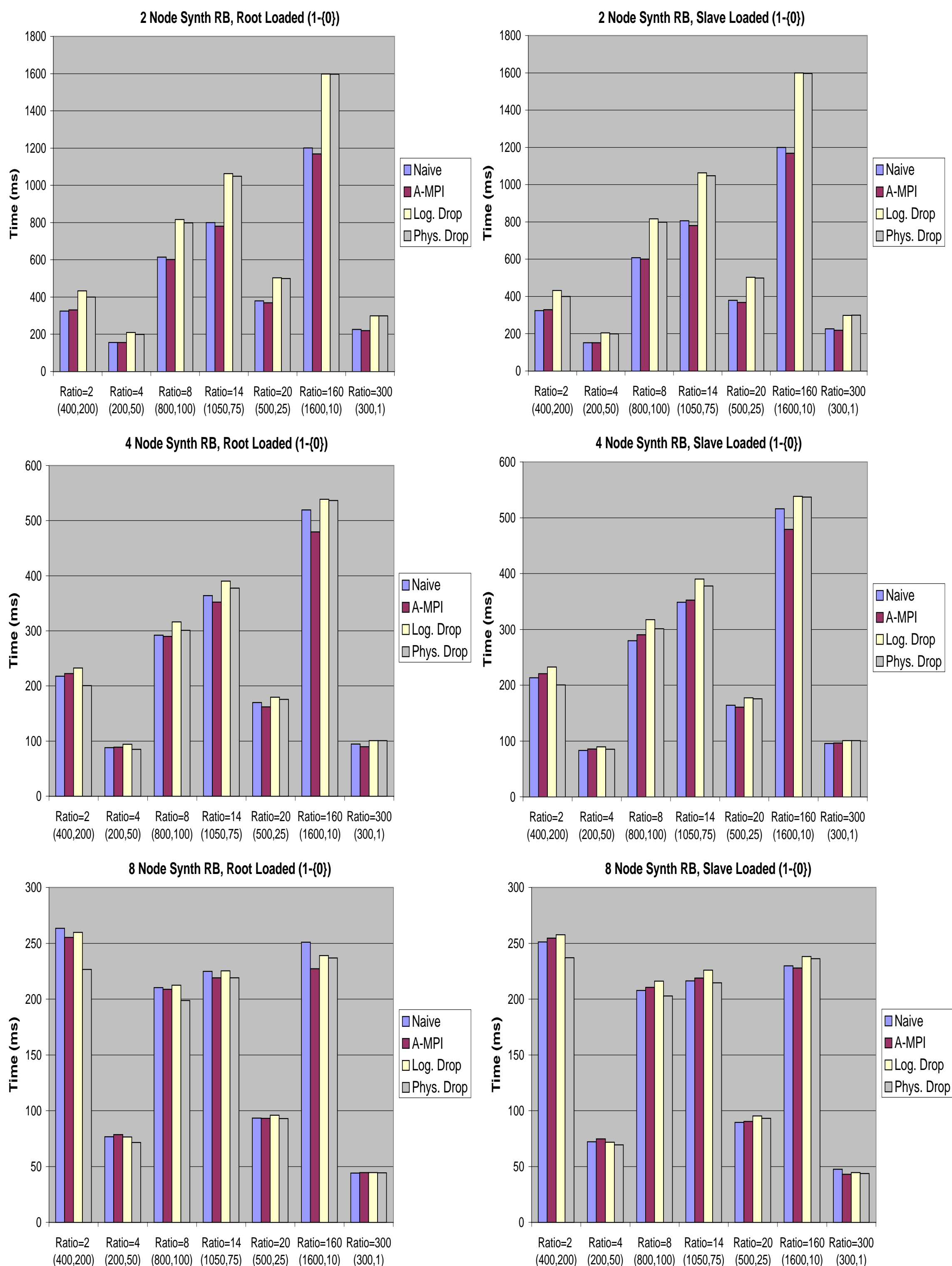

Figure 5.3: Graphs of the synthetic reduce-broadcast tests. The left-hand side column shows the results when the root node is loaded. The right-hand side column shows the results when a non-root node is loaded.

Figure 5.3 shows the graphs of our measurements. The left-hand side column displays the tests where the competing process is run on the root node. Comparing these results to the results of the one competing tests of Figure 5.2(a) (from the previous section) reveals many similarities. If the ideal distribution with all nodes is best for a nearest-neighbor test, then the ideal distribution with all nodes is also best for the corresponding reduce-broadcast test. Likewise, if removing the node is best for a nearest-neighbor test, then removing is best for the reduce-broadcast test. Interestingly, for the ratios between 8 and 20, the absolute difference between Naive and A-MPI is much less for the reduce-broadcast tests than the nearest-neighbor tests. However, the results for the large ratios are almost identical. A closer inspection reveals that for the large ratio tests, the absolute times for the reduce-broadcast tests are very similar to the nearest-neighbor tests. On the other hand, as the ratio gets smaller, the absolute times for the reduce-broadcast tests are less than the times for the nearest-neighbor tests. We believe that for small ratios of computation to communication, reduce-broadcast communications are not as affected by load as a corresponding nearest-neighbor communication (time-wise).

The last point becomes even more evident when we examine the graphs where a non-root node is loaded (the right-hand side column of Figure 5.3). In several cases for the 4 and 8 node tests, the naive distribution results in the best performance. Comparing the loaded root and loaded non-root graphs, we see that the distributions chosen by A-MPI result in approximately equal execution times for all cases. However, naive distributions where the non-root node is loaded result in better performance in many cases. Clearly, the presence of the competing process on the non-root node has less of an impact on the reduce-broadcast (and thus the application) than it does when it is on the root node. In the future, we can make special provisions for this case.

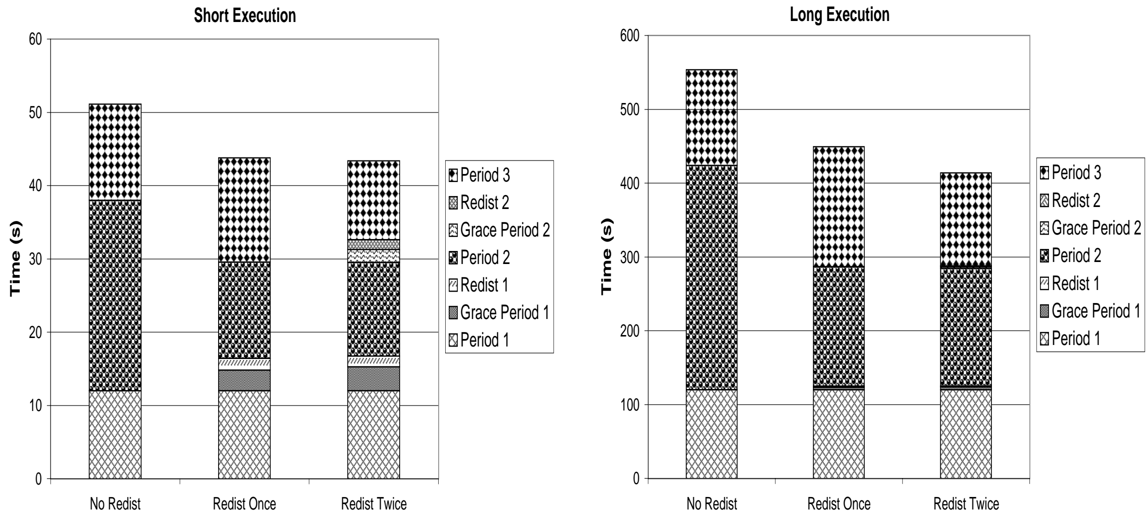

Figure 5.4: Comparison of the execution of Jacobi iteration. The results are for a 4-node configuration, and the arrays are doubles of dimension 2048x2048. Also, *Period* denotes the execution of Jacobi, *Grace Period* denotes the monitoring period of A-MPI, and *Redist* denotes redistribution.

5.4 Redistribution Costs

We now show the cost of redistribution using Jacobi iteration. Our version of Jacobi is a 2-phase application; the first phase performs a two-dimensional computation of Laplace's equation and a nearest-neighbor edge exchange, and the second phase performs an array swap and a reduce-broadcast to test convergence. The first phase has the longest execution time, so A-MPI performs successive balancing for the first phase. We first show results of entire executions in order to compare the overall performance when load balancing does and does not take place. We then focus on the actual cost of redistribution by showing results for several problem sizes.

Figure 5.4 breaks down the overall execution time of Jacobi iteration. This test is meant to show the effectiveness of A-MPI when competing processes are either introduced on a node or terminate; either way, A-MPI must adapt to changes in

available processing power. In addition, we wish to show application performance when we include the cost of redistribution.

Each graph of the figure shows the results of three tests: *No Redist*, *Redist Once*, and *Redist Twice*. Execution is separated into three periods of a fixed number of iterations, and for each test, the first period executes without competing processes. At the end of the first period, we introduce a single competing process on one node. At this point, no action is taken for the *No Redist* test, while redistribution takes place for the other two. We then execute the second period in the presence of the competing process. At the end of the second period, we terminate the competing process. Here, redistribution only takes place for the *Redist Twice* test. We then execute the third and final period. We show results for a 4-node configuration for arrays of doubles with dimension 2048x2048. The figure shows two graphs: *Short Execution* (period = 50 iterations) and *Long Execution* (period = 500 iterations).

The left-hand side graph clearly shows the speedup A-MPI obtains when redistributing after the competing process is initiated. Overall, Jacobi executes 16.7% faster if we redistribute after the first period; this includes the results of *Redist Once* and *Redist Twice*. Notice, however, that there is very little performance improvement (less than 1%) if we redistribute after the second period as well. For short executions, the cost of redistribution cancels out the speedup obtained (redistribution is 6.4% of the total execution time for the *Short Execution*, *Redist Twice* test). However, this is not the case for the long execution test (right-hand side graph). In this case, it is worthwhile to redistribute after the second period; the improvement is 7.9% over redistributing only once and 25.2% if we never redistribute. In this case, redistribution is less than 1% of the total execution time.

We now examine the cost to physically redistribute. Table 5.2 shows the results of six examples. The tests were set up in two- and four-node configurations. In each test, the computation begins with an equal distribution of iterations to each node.

# Nodes	Dim	Load Redist Cost	Unload Redist Cost
2	512	0.1851	0.1013
4	512	0.1312	0.0705
2	1024	0.6167	0.4028
4	1024	0.2836	0.2471
2	2048	2.434	1.632
4	2048	1.009	0.7363

Table 5.2: Table of redistribution costs when the loaded node is removed. *Load Redist Cost* shows the redistribution time when the node is loaded at the time it is removed; *Unload Redist Cost* the time when the node is not loaded when removed. One distributed, two-dimensional array of the dimension shown is redistributed each time (i.e. 512x512). All times are in seconds.

A competing process is started on one of the nodes and redistribution takes place such that the loaded node is removed. The numbers for *Load Redist Cost* in the table reflect this redistribution time. For comparison we run identical tests, except that the removed node is not actually loaded. The numbers for *Unload Redist Cost* reflect this redistribution time.

As is expected, the cost of redistribution is proportional to the amount of data that is communicated. Though the redistribution process has many steps, the time required to communicate the data that changes ownership dominates the total time. Of note is the fact that doubling the number of processors from 2 to 4 greatly reduces the redistribution cost (e.g., 51% improvement from 2 to 4 processors for *Unload Redist Cost, Dim=2048*). We attribute the improvement with 4 nodes to the fact that (1) the overall communication is reduced because the removed node owns less data and (2) some communication takes place in parallel during the data shift.

It is important to note the effect of load on redistribution. For the 2-node tests, the loaded redistribution is as much as 83% slower than the unloaded redistribution (*Dim=512*). The slowdown is reduced in general for 4 nodes, but the redistribution

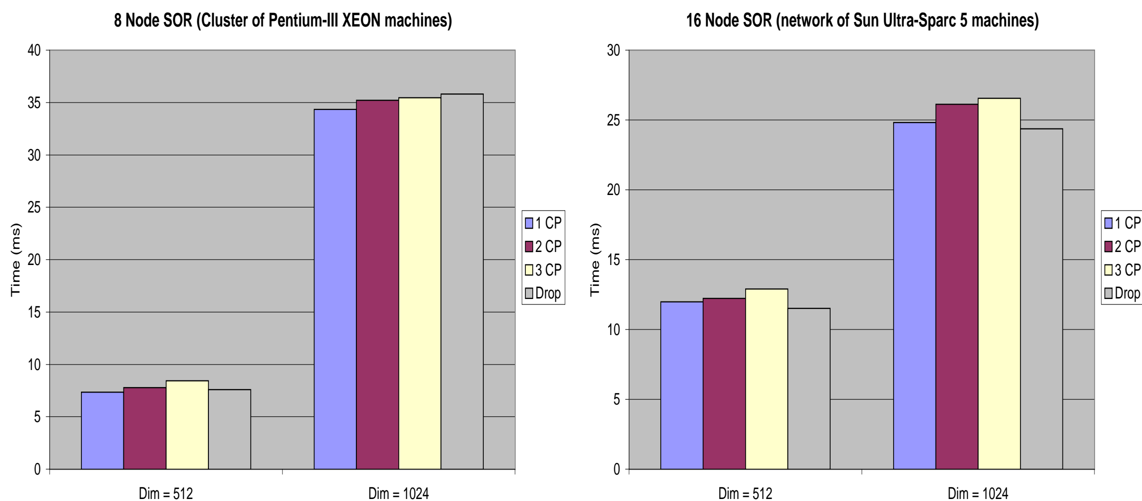

Figure 5.5: Results of SOR tests when (1) a node has one, two, or three competing processes and a distribution that includes the loaded node is used (labelled *1 CP*, *2 CP*, or *3 CP*) and (2) the loaded node is removed (labelled *Drop*). Note that successive balancing is used to determine distributions that include the loaded node. Also, the times shown are for average phase cycle execution time after redistribution. Configurations of 8 and 16 nodes were used for arrays of dimension 512x512 and 1024x1024.

is still affected. However, from an absolute standpoint these times are small (on the order of hundreds of milliseconds), and as we have shown, a small, one-time penalty is usually a much better choice than the performance degradation incurred if redistribution does not take place.

5.5 Node Removal

In order to demonstrate the applicability of physical node removal, we now show the results from Red/Black SOR. In our version, the computation is performed in two passes over half of the elements of the array, where each pass is a separate phase. The updates are performed in-place, which results in good cache performance with the use of a single array for all accesses. As a result, the ratio of computation to

communication is small for certain problem sizes, making node removal likely when competing processes exist.

We first ran tests using an 8-node configuration. Then, to demonstrate that increasing the number of processors further increases the likelihood of node removal, we ran the same tests using 16 nodes[1]. Figure 5.5 shows the results for two-dimensional arrays of size 512 and 1024 using 8 (left-hand side) and 16 (right-hand side) nodes. Each graph shows the average phase cycle execution time of SOR after redistribution (using successive balancing) when one, two, or three competing processes (*1 CP*, *2 CP*, or *3 CP*) are introduced on a single node. In addition, each graph shows the results of physically removing the loaded node (*Drop*).

The results shows that node removal is usually the best decision for the problem size of 512 for both configurations. The only case where node inclusion outperforms node removal is the 8-node, one competing process test. In all other cases, node removal is best, with an improvement of up to 11% (16-nodes, *3 CP*) over node inclusion. However, for the problem size of 1024, the results for the 8- and 16-node configurations differ. With 8 nodes, node inclusion outperforms node removal regardless of the number of competing processes, but the percentage difference is small (max of 4% with *1 CP*). On the other hand, with 16 nodes, node removal is always best, with an improvement ranging from 2% to 8%.

Based on the previous results from the synthetic nearest neighbor tests (Figures 5.2(a) and 5.2(b)), these results are as expected. This is because the ratio of computation to communication is approximately 12 and 50 (8 nodes) and 10 and 30 (16 nodes) for the arrays of size 512 and 1024 respectively[2]. Examining the 8-node

[1] For the 16-node tests, the computing environment consists of 16 Sun Ultra-Sparc 5 machines (360 Mhz).

[2] Differences in system architecture, processor speed, amount of memory and network infrastructure can all affect the ratio of computation to communication. Hence, the differing ratios for the 8- and 16-node configurations.

graphs with similar ratios (e.g. *Ratio=14* and *Ratio=20*), we see that the likelihood of node removal increases with the number of competing processes. Hence, we expect node removal to outperform node inclusion for a ratio of 12 when we increase the load. However, for a ratio of 50 (array of size 1024), we would not expect node removal unless the system was more heavily loaded (4 or more competing processes).

On the other hand, the synthetic tests show that scaling the number of processors from 2 to 8 also increases the likelihood or removal for any load (e.g. 1 competing process, *Ratio=20*). Hence, since node removal is usually best with 8 nodes and a ratio of 12, we expect it to always outperform node inclusion with 16 nodes and a ratio of 10. In addition, considering that the ratio of computation to communication is smaller on the 16-node cluster for the array of size 1024 (ratio of 30 versus 50), it is not surprising that node removal is also best in this case. We conclude that removal of loaded nodes is applicable to *all* parallel applications for some number of nodes.

5.6 Unbalanced Computations

We now show the performance of a particle dynamics application in order to (1) prove that A-MPI supports unbalanced computations and (2) demonstrate the effectiveness of our method for determining unloaded iteration times. In this application, 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 [SWG92]. This application is representative of programs where a good data distribution depends on information that is available only at run time, and different distributions might be better at different time steps of the computation. This is because the amount of

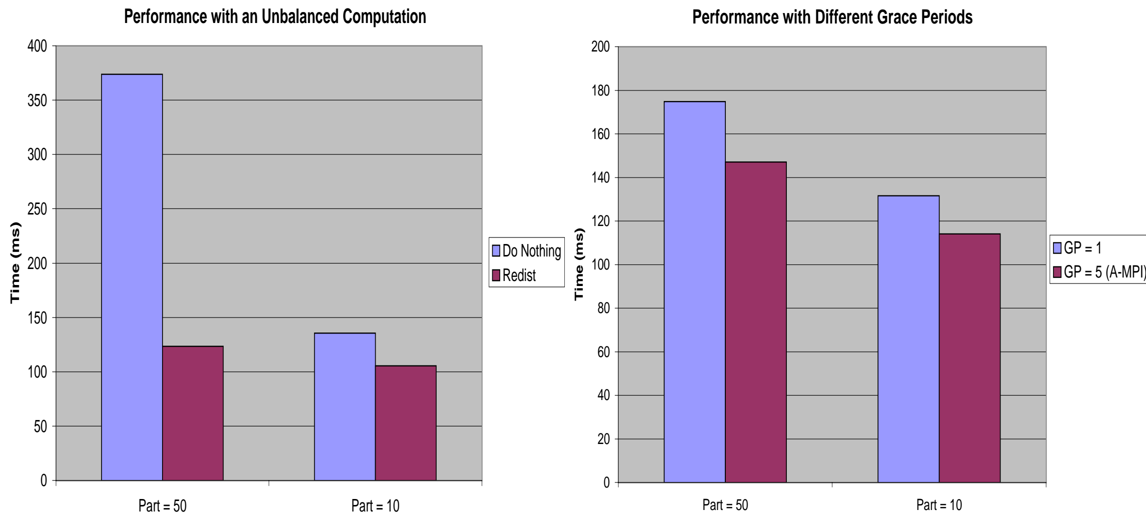

Figure 5.6: Results of the particle simulation tests where the computation is unbalanced. The left graph compares the results when redistribution does and does not take place. The right graph shows the effectiveness of using 5 phase cycle iterations (compared with 1 iteration) to determine true, unloaded iteration times before A-MPI determines a new distribution. All results are for an 8-node configuration with a grid dimension of 256x256. The label *Part* denotes the degree of imbalance as determined by the number of particles in each grid cell in the top half of the rows owned by P_0

computation at each grid cell depends on how many particles are in that cell. For our tests, we use an 8-node configuration and the dimension of the matrix is 256x256.

The left-hand side graph of Figure 5.6 compares the average phase cycle execution time of the particle simulation when redistribution does (labelled *Redist*) and does not (labelled *Do Nothing*) take place. We introduce no competing processes for this test, we simply show the importance of effecting a good distribution when the computation is unbalanced. To create an imbalance, we assign an equal number of grid rows to each node but place a greater number of particles (denoted *Part*) in each grid cell of the first half of the grid rows owned by the first node. For all other rows, we place a single particle in the first grid cell. The graph shows the results for $Part =$

50 and *Part = 10*. In both cases, the distribution determined by A-MPI outperforms the initial distribution (67% and 22% for *Part = 50* and *10* respectively).

The right-hand side graph of Figure 5.6 compares the execution time after redistribution when the monitoring period is either 1 (*GP = 1*) or 5 (*GP = 5*, the default for A-MPI) phase cycle iterations. For this test, we create a load imbalance as described above but in both cases, redistribution takes place to balance the workload. However, we then introduce a single competing process on one node and perform redistribution again. Because each iteration is less than 10ms, `gethrtime` must be used, and as a result, the timings will be affected by context-switching and any other performance fluctuations (such as cache hits or misses). Hence, if the grace period is only 1 phase cycle iteration, the timing data can be inaccurate. Comparing the results for *GP = 1* and *GP = 5* clearly shows the effectiveness when the grace period is 5 phase cycle iterations. Using 5 iterations to determine iteration times results in a 16% improvement (*Part = 50*) over using 1 iteration. This improvement is realized because the times are minimized over the grace period to determine an unbiased cost associated with the computation of each iteration. As a result, a distribution can be correctly created according to the ideal workload percentages. Similarly, for *Part = 10*, the improvement is 13%.

Chapter 6

Conclusion and Future Work

This thesis has described our approach to supporting efficient message-passing programs in distributed-memory environments where nodes are nondedicated and/or the computation is not balanced. We have designed and implemented Adaptive Message Passing Interface (A-MPI) as a programming infrastructure to meet these goals. We created a memory allocation mechanism to support efficient redistribution. We also developed novel techniques for determining system load and iteration execution times. Further, we have implemented a new method for balancing the computational load when available processing power varies and modified MPI communication routines to support the removal of nodes that degrade performance.

Performance results showed that our system automatically adapts to changes in system load and application behavior with low overhead. Further, the techniques we use outperformed traditional methods in most cases. In particular, distributions determined by successive balancing resulted in a performance improvement of up to 28% over naive distributions. In addition, physical removal of loaded nodes resulted in up to 56% improvement over logical removal. We further substantiated the effectiveness of our system with various experiments using well-known applications.

In the future, we would like to expand our system to support competing processes that are parallel applications. This will require changes to our load determination technique such that the probability that an application is actually computing is considered. We would also like to determine effective distributions for multi-phase

applications where the calculated ideal workloads of each phase differ. It may be that our current method is best (considering only the phase with the longest execution time), but this needs to be researched further. Lastly, we need to investigate methods to accurately predict execution time when nodes are loaded. This will enable us to consider distributions where *some* loaded nodes are removed, instead of considering only the removal of all of them.

Bibliography

[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, June 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.

[CC96] Michele Cermele and Michele Colajanni. Dynamic load balancing of distributed SPMD computations with explicit message-passing. TR RI.96.04, University of Rome, Tor Vergata, September 1996.

[EAS+97] G. Edjlali, G. Agrawal, A. Sussman, J. Humphries, and J. Saltz. Runtime and compiler support for programming in adaptive parallel environments. *Scientific Programming*, January 1997.

[ESS96] Guy Edjlali, Alan Sussman, and Joel Saltz. Interoperability of data parallel runtime libraries with Meta–Chaos. Technical Report CS-TR-3633 and UMIACS-TR-96-30, University of Maryland, Department of Computer Science and UMIACS, May 1996. A condensed version submitted to Supercomputing'96.

[GAL96] Jordi Garcia, Eduard Ayguade, and Jesus Labarta. Dynamic data distribution with control flow analysis. In *Supercomputing '96*, November 1996.

[GB93] M. Gupta and P. Banerjee. PARADIGM: A Compiler for Automated Data Distribution on Multicomputers. In *Proceedings of the 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, 1991.

[HKT91] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for FortranD on MIMD distributed-memory machines. pages 86–100, 1991.

[HMS+95] Yuan-Shin Hwang, Bongki Moon, Shamik D. Sharma, Ravi Ponnusamy, Raja Das, and Joel 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.

[HPF94] High Performance Fortran language specification. November 1994.

[ID98] Sotiris Ioannidis and Sandhya Dwarkadas. Compiler and run-time support for adaptive load balancing in software distributed shared memory systems. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-Time Systems for Parallel Computing*, pages 107–122, May 1998.

[IRSD99] S. Ioannidis, U. Rencuzogullari, R. Stets, and S. Dwarkadas. CRAUL: Compiler and run-time integration for adaptation under load. *Journal of Scientific Programming*, August 1999.

[KK98] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, 1998.

[LA96] David K. Lowenthal and Gregory R. Andrews. An adaptive approach to data placement. In *Proceedings of the 10th International Symposium on Parallel Processing*, pages 349–353, April 1996.

[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.

[ML01] Donald G. Morris and David K. Lowenthal. Accurately computing redistribution cost in distributed shared memory systems. In *Principles and Practice of Parallel Programming*, pages 62–71, June 2001.

[MS94] Bongki Moon and Joel Saltz. 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.

[PB95] Daniel J. Palermo and Prithviraj Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[RD01] Umit Rencuzogullari and Sandhya Dwarkadas. Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations. In *Eighth Conference on Principles and Practice of Parallel Programming (to appear)*, June 2001.

[RN95] J. Ramanujam and A. Narayan. Automatic data mapping and program transformations. In *Workshop on Automatic Data Layout and Performance Prediction*, June 1995.

[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.

[SLGZ99] Alex Scherer, Honghui Lu, Thomas Gross, and Willy Zwaenepoel. Transparent adaptive parallelism on NOWs using OpenMP. In *Seventh Conference on Principles and Practice of Parallel Programming*, pages 96–106, May 1999.

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

[SWB97] Gary Shao, Rich Wolski, and Fran Berman. Modeling the cost of redistribution in scheduling. In *Eighth SIAM Conference on Parallel Processing for Scientific Computation*, March 1997.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. Splash: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1), 1992.

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

[WSH99] Rich Wolski, Neil Spring, and Jim Hayes. Predicting the CPU availability of time-shared unix systems on the computational grid. In *Eighth High-Performance Distributed Computing Conference*, August 1999.

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