Journal of Parallel and Distributed Computing **62**, 1248–1271 (2002)
doi:10.1006/jpdc.2002.1838

Predicting the Cost and Benefit of Adapting Data Parallel Applications in Clusters

Jon B. Weissman

Department of Computer Science and Engineering, University of Minnesota, Twin Cities, 200 Union Street S.E., Minneapolis, Minnesota 55455
E-mail: jon@cs.umn.edu

Received January 26, 2001; accepted February 20, 2002

This paper examines the problem of adapting data parallel applications in a shared dynamic environment of PC or workstation clusters. We developed an analytic framework to compare and contrast a wide range of adaptation strategies: dynamic load balancing, migration, processor addition and removal. These strategies have been evaluated with respect to the cost and benefit they provide for three representative parallel applications: an iterative jacobi solver for Laplace's equation, gaussian elimination with partial pivoting, and a gene sequence comparison application. We found that the cost and benefit of each method can be predicted with high accuracy (within 10%) for all applications and show that the framework is applicable to a wide variety of parallel applications. We then show that accurate prediction allows the most appropriate method to be selected dynamically. Performance improvement for the three applications ranged from 25% to 45% using our adaptation library. In addition, we dispel the conventional wisdom that migration is too expensive, and show that it can be beneficial even for running parallel applications with non-trivial communication. © 2002 Elsevier Science (USA)

Key Words: cluster computing; distributed computing; parallel processing.

1. INTRODUCTION

Shared cluster networks of PC workstations offer an attractive platform for parallel applications due to their cost performance ratio. The cluster environment can offer both high performance and high throughput if resources are allocated effectively and managed efficiently. One of the impediments to achieving high performance in clusters is that resources may not be fully under control of the individual application. Networks, computers, and disks, are often shared among competing applications. In this environment, parallel programs may be contending for resources with other programs and may be subject to resource fluctuation during execution. If such applications wish to continue to achieve high performance in the face of resource sharing, they must adapt to changing resource availability. The need for adaptivity raises a set of questions: When should the application adapt? How should the application adapt? What is the cost of adaptation? What is the benefit of adaptation?

This is a complex set of questions that depends on the structure of the application, problem parameters such as problem size, the type of resources allocated, and system parameters such as machine and network performance. This paper examines adaptivity for high-performance data parallel applications due to CPU sharing in commodity clusters and workstation networks. We have focused on SPMD data parallel applications because they represent a large and important class of scientific applications. We also studied applications with a variety of communicating structures including peer-to-peer local communication and global master–slave communication. To simplify presentation of our model, it is assumed that the cluster resources are homogeneous. While heterogeneous clusters can be handled with a few simple modifications to our model, they are the subject of future work. Application adaptivity in response to CPU sharing was achieved by the following techniques: dynamic load balancing, migration, processor addition, and processor removal. Prior work established that performance prediction for non-adaptive SPMD applications can be done accurately [23, 24]. In this paper, we extend our earlier models to include the cost and benefit of adaptation.

We found that the cost and benefit of each adaptation technique can be characterized and predicted with high accuracy for all data parallel applications. This cost is dependent on application parameters such as problem size and system parameters such as the amount of resource fluctuation and network performance. The most appropriate adaptive technique is also shown to depend on these parameters. In addition, several optimized migration schemes were developed and shown to be competitive with dynamic load balancing. These results provide part of the answers to the *how* and *cost* questions concerning adaptation. The answer to the question of *when* to adapt is more complex. An application should adapt when it can improve its performance in response to a change in resource availability either due to sharing or to exploit new resource opportunities. An application can utilize our cost models to determine when to adapt by comparing the predicted benefit to the predicted cost. The results indicate the answer to *when* also depends on problem size, the amount of resource fluctuation, and when the necessary adaptations occur. We also showed that a run-time system can automatically select among the adaptation methods using these performance prediction models and achieve greater performance on a shared cluster.

2. RELATED WORK

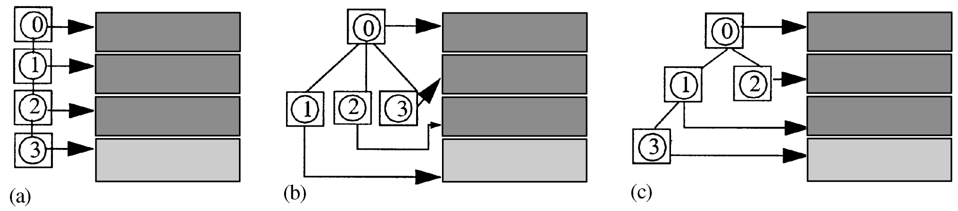
Techniques for adapting data parallel applications in response to dynamic resource fluctuation have been well studied. Both migration [4, 12, 13, 22, 27] and dynamic load balancing [1, 5, 9, 12, 14, 15, 18, 26] strategies have been proposed. The majority of these systems focus on a single technique with particular attention paid to the implementation mechanism. Several studies have affirmed that there is a benefit to exploiting shared resources in a network environment and analyzed the conditions that make it most beneficial [1]. In our work, we determine when it is beneficial to use a set of resources and then analyze when adaptation is necessary to continue to reap benefits. Other work has presented a general framework for adaptation for several classes of programs, both parallel and non-parallel [12]. For parallel applications, the focus was on threaded parallel programs in which the grain of adaptation is a thread. In our work, the grain of adaptation is based on the data itself. In other work, several adaptive methods (migration and load balancing) for parallel PVM programs running in a network environment were implemented and compared [4]. The focus of this system was on the efficient implementation of each adaptive method as opposed to a cost-driven analysis of when and how to adapt.

Little research exists on quantitative comparison of different adaptation mechanisms for performance. However, a quantitative model for performance-based redistribution similar to our work was developed [17]. The results show that the cost and benefit of dynamic load balancing can be predicted for a single application. In addition, much of the research in migration is geared to OS mechanisms [2, 6, 19], reclaiming desktop resources [13, 27], or space-sharing for parallel jobs [16]. It is generally believed that migration is too expensive for high-performance applications [7]. However, our results show that this conventional wisdom may be false as confirmed by others [11]. In general, our work shows that different adaptation methods are appropriate in different circumstances and provides an analytic framework for comparison. Further, the framework can be realized in a practical implementation.

Other related work has focused on performance prediction of parallel applications in clusters and Grids [3, 8, 20, 21]. In particular, the AppLeS project [3, 8] and MicroGrid simulation environment [20] have developed application performance prediction models with similar accuracy to our prior [23] and current work. This paper extends our prior work to performance prediction relating to the cost and benefit of adaptation which is different from those other projects. The impact of contention was investigated in [8] and accurate prediction models were developed. Our work also considers the impact of CPU contention, but then goes further to decide how to adapt in response. Adapting master–slave parallel programs in response to CPU load has been investigated by a number of researchers, including for the Complib gene sequence comparison application, that we also study here [21]. Our version of Complib used static scheduling initially (as in [21]), but then used dynamic load balancing instead of self-scheduling to perform the adaptation in clusters and Grids. If the number of adaptive events is small, then the DLB approach is superior since it only takes action when the load fluctuates, in contrast, self-scheduling incurs overhead for each work distribution event.

3. APPLICATION MODEL AND ADAPTIVE METHODS

In the SPMD application model, the data domain is decomposed across a set of *workers* that are implemented as address-space disjoint processes that communicate via message-passing (Fig. 1a–c). Shared-memory is not assumed, but a shared-memory version of our application and adaptation model can be easily constructed.



```
(d) 1. receive piece of the problem and neighbor addresses
    2. iterate until done
       3. communicate to/from neighbors
       4. check if adaptation event is triggered
          4a. if so, initiate adaptation
       5. compute/update local data domain
          5a. complete adaptation if initiated
       6. perform convergence check periodically (optional)
    7. end iterate
```

FIG. 1. SPMD model. Three communication topologies are shown (1-D, master–slave, and tree) in which the data domain is decomposed across four machines or workers (numbered 0...3) each given a chunk of the data. The data domain need not be a grid or matrix despite its appearance above. Workers communicate via message passing. In (d), pseudo-code for the SPMD worker loop including adaptation detection and action is also shown.

The applications under consideration are iterative with alternating computations and communications. This is the most common model that fits many parallel applications. We have experimented with three data parallel applications each with a very different communication topology: an iterative jacobi solver for a 2-D Laplace's equation decomposed by row (1-D communication), gaussian elimination with partial pivoting also decomposed by row (master–slave communication), and a parallel gene sequence comparison application (tree communication), described fully in Section 4.1. In each case, a main program (not shown) creates a number of parallel workers across a set of processors within a cluster. The workers each compute on a portion of the data domain.

To examine the cost and benefit of adaptivity on SPMD data parallel applications in shared clusters, we implemented a library of adaptive techniques. In parallel, we also developed an analytic model for characterizing the performance of the adaptive methods and the applications to be described later. Library calls were inserted to detect and initiate adaptation actions within the applications (Fig. 1d, lines 4, 4a, 5a). An adaptation event was triggered from an external detector process and sent to all workers. The library implements the following adaptation techniques: *migrate*, *migrate-non-block*, *migrate-stream*, *migrate-ghost*, *add-processor*, *remove-processor*, and *dynamic-load-balance*. Migration involves remote worker or process creation followed by the transmission of the old worker's data to the new worker. Migration events do not migrate the entire process image since we initiate migration events only at predictable points in the worker's execution and transmit the necessary state. This state includes the portion of the data domain contained within the worker and iteration indices. Non-blocking migration allows the remote process creation to be initiated before the computation step in a non-blocking fashion which permits the process creation and the computation of a single iteration to be overlapped. The migrate-stream method takes this a step further by streaming ahead a portion of the data to the newly created worker thereby providing some parallelism in the computation step for a particular iteration (Fig. 2). This requires that the new worker is created first unlike non-blocking migration. We set the amount of data to stream to be half of the data held by a worker. More optimal strategies that determine the most appropriate amount to stream are the subject of future work.

Ghosting allows the re-use of an existing worker on a processor that was removed or migrated earlier. When ghosting is enabled, migration and processor removal will allow the process image to persist for the worker but in a sleep state. This will reduce the overhead of remote process creation significantly. Adding and removing processors is conceptually simple but requires data transmission to maintain load balance and patching the communication topology. Finally, dynamic load balancing collects load indices, determines a re-distribution of data that rebalances the workers, and initiates the data transmission. Our dynamic load balancing implementation is based on Quinn's model [14].

The adaptation action was initiated by a single processor or worker (4a): for migration and removal it is the worker being migrated or removed; for an add or dynamic load balancing action it is worker 0 by convention. Most of the adaptive techniques are well-known and their implementations straightforward. For all of these techniques, we must insure that load balance is maintained. If workers are added, removed, migrated, and explicitly load balanced, data must be moved between the workers to maintain load balance. The amount of data transmitted is a large component of the adaptation overhead. For migration, we simply transmit the entire data domain from the old worker to the new worker. If the new worker

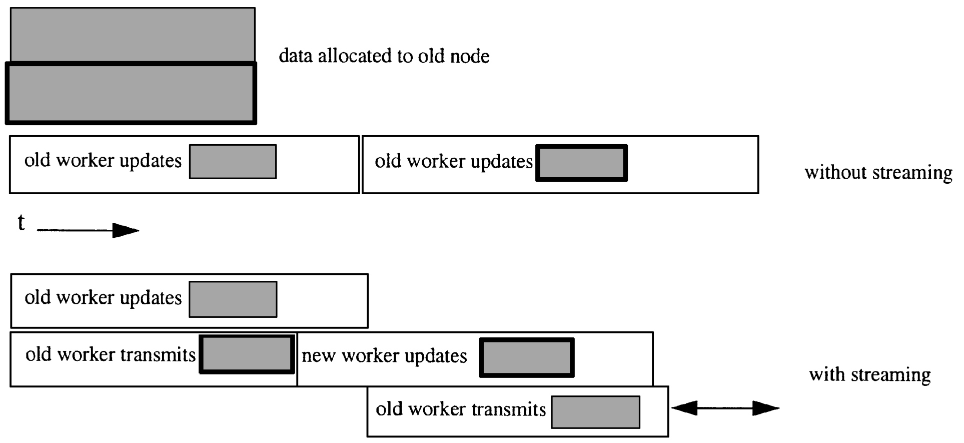


FIG. 2. Streaming. An old worker is migrating its data to a new worker. In the first case, the old worker updates both halves of its data and then transmits the updated region. In the second case, the old worker "streams ahead" a portion of the data, thus enabling both the old and new workers to update different parts of the data domain in parallel. The double-arrow indicates the potential performance pickup.

machine is more or less powerful than the old worker machine, then a re-balance may be required. We handled this by detecting the load imbalance as a separate event and responding to it.

Ghosting and streaming are designed to reduce the overhead of migration. Ghosting is the re-use of a worker processor that was previously active. A ghost is created when a current worker is migrated or is simply removed. When the worker is finished, it releases its memory and goes to sleep consuming little or no resources (5a), but the process image remains. At a later point in time, this worker may be selected as the destination of a migrated worker or to add a new worker. While not all systems will allow ghosts to linger, the use of ghosts can greatly reduce adaptation overhead since it can prevent expensive dynamic process creation overhead. Streaming is an optimization that is useful when the amount of computation (5) is large. In streaming, the migrator first streams ahead a fraction of its data (typically half) to the new worker before updating the data. At this point, the migrator and migratee each update half of the data in parallel (for one iteration). When the migrator is finished, it sends its updated half to the new worker. The advantage of streaming is that the new worker need not have to wait for the computation of the data domain within the migrator in one iteration (5).

To evaluate the cost and benefit of these adaptive methods, a performance prediction framework has been developed for data parallel applications such as in Fig. 1. This model is applicable to a wide variety of data parallel applications. The model is based on the following terms (N is the data domain or problem size, data elements are E_{size} bytes, P is the # of processors, i is the current iteration, and DM is the amount of data that must be moved at iteration i to achieve load balance, it will be defined shortly):

$T_{\text{comm}}\ (N, P, j, i)$ is the cost of the ith communication step (3) for jth worker,
$T_{\text{comp}}\ (N, P, j, i)$ the cost of the ith computation step (5) for jth worker,
$T_{\text{exec}}\ (N, P, j, i)$ the total cost of the ith iteration for jth worker,
T_{pc} the cost of creating a remote process or worker,
T_{gc} the cost of waking up a ghost process or worker,
$T_{\text{dm}}\ (B)$ the cost of transferring B bytes of data, and
T_{nn} is the cost of establishing new worker neighbors.

The term T_{comp} includes any memory overhead required as part of the computation. After a response to an adaptive event, load balance will be restored and T_{exec} will be the same for all workers. For regular applications such as the jacobi solver, T_{comm} and T_{comp} are the same for each iteration, but for less regular applications such as gaussian elimination the amount of computation and communication are a function of the current iteration. Using these terms, equations that capture the overhead for each adaptive method are defined (all methods except dynamic load balancing include T_{nn} since the communicating neighbors will change for at least one of the workers):

$$T_{\text{migrate}}(N, P, i) = T_{\text{pc}} + T_{\text{dm}}(E_{\text{size}}\ \text{DM}(N, P, i)) + T_{\text{nn}},$$

migrate includes process creation and data movement;

$$T_{\text{migrate-non-block}}(N,P,i) = \max\{0, T_{\text{pc}} - T_{\text{comp}}(N,P,i)\} + T_{\text{dm}}(E_{\text{size}}\,\text{DM}(N,P,i)) + T_{\text{nn}},$$

migrate-non-block overlaps process creation and computation;

$$T_{\text{migrate-ghost}}(N,P,i) = T_{\text{gc}} + T_{\text{dm}}(E_{\text{size}}\,\text{DM}(N,P,i)) + T_{\text{nn}},$$

migrate-ghost includes ghost creation and data movement;

$$T_{\text{migrate-stream}}(N,P,i) = T_{\text{pc}} - T_{\text{comp}}(N,P,i)/2 + T_{\text{dm}}(E_{\text{size}}\,\text{DM}(N,P,i)) + T_{\text{nn}},$$

migrate-stream includes process creation and data movement; it streams ahead $\frac{1}{2}$ of the data to allow parallel execution in the inner loop;

$$T_{\text{add}}(N,P,i) \max\{T_{\text{dm}}(E_{\text{size}}\,\text{DM}(N,P,i)), T_{\text{pc}}\} + T_{\text{dm}}(E_{\text{size}}\,\text{DM}(N,P+1,i)) + T_{\text{nn}},$$

add includes two data movement steps: first, each worker sends a fraction of their data to worker 0 (in parallel with this step, a process is created for the new worker), second: worker 0 then distributes the collected data to the new worker;

$$T_{\text{remove}}(N,P,i) = 2T_{\text{dm}}(E_{\text{size}}\,\text{DM}(N,P,i)) + T_{\text{nn}},$$

remove includes two data movement steps: first, the exiting worker's data is sent to a worker 0, second: this data is then distributed across each remaining worker;

$$T_{\text{dlb}}(N,P,i) = 2T_{\text{dm}}(E_{\text{size}}\,\text{DM}(N,P,i)),$$

the cost of dynamic load balancing is the cost of moving data from overloaded to underloaded workers and is done in three steps: first, the load indices of each worker are collected by worker 0 (this cost is negligible and is omitted), second, this worker computes the re-distribution and collects the data from each overloaded worker, third, worker 0 transmits data to each underloaded worker to achieve load balance.

Streaming has the effect of reducing the effective cost of computation by streaming ahead a portion of the data. This overall cost reduction is captured by the minus term in the overhead. It is possible that $T_{\text{migrate-stream}}$ can be negative indicating a net improvement in performance if T_{comp} is sufficiently large. All of these methods depend on $\text{DM}(N,P,i)$, the amount of data that must be moved to achieve load balance. We defined this function in terms of the individual load indices $L(j,i)$ of the jth worker (processor) at iteration i, and the current amount of data $D(j,i)$ held by each worker (processor) as follows (a load value of 1.0 is unloaded or idle):

$$\text{DM}(N,P,i) = \sum_{j=1}^{P} \left| D(j,i) - \frac{N}{L(j,i)L_{\text{normal}}} \right|, \qquad \text{where } L_{\text{normal}} = \sum_{j=1}^{P} \left(\frac{1}{L(j,i)} \right).$$

Finally, we determined the benefit to performing an adaptive method A (T_{exec} is the execution time without performing the adaptation, T^{*}_{exec} is the execution time if the adaptation is performed, and T_{method} is the cost of method A, e.g., if $A = \text{dlb}$, then $T_{\text{method}} = T_{\text{dlb}}$):

$$T_{\text{benefit}}(A, N, P, i) = \sum_{k>i} T_{\text{exec}}(N, P, j, k) - \sum_{k>i} T^{*}_{\text{exec}}(N, P, j, k) + T_{\text{method}}(N, P, i).$$

This adaptation model is applicable to the two dominant models of shared cluster computing: (1) space-shared clusters and (2) networks of shared PCs or workstations. In pure space-sharing, adaptation due to external load is not an issue, but opportunistic adaptation (adding and possibly removing processors) would be. In a time-shared cluster or networks of shared PCs or workstations, competing workload will be submitted by other users or the resource owner. This is captured by our load measure L, and several of our adaptive methods are useful in this context (migration, DLB, and possible removal). In both models, we focused on CPU contention and taking adaptive action is response. Contention for communication and disk resources may also be present, but our prior work with communication contention [25] for compute-intensive data parallel applications indicates that the impact of communication contention is likely to be minimal.

4. RESULTS

Our experimental environment was a shared Unix cluster containing 10 UltraSparc workstations (166 MHz, 64 MB memory) connected by a dual network of 10 Mbps (non-switched) ethernet and 155 Mbps ATM. Each node was running Solaris 5.7. We ran experiments both when the cluster was dedicated to us, and when other users were running. These modes allowed us to experiment with space-shared clusters and networks of shared PCs or workstations. We implemented our applications and adaptive library in C with hand-coded TCP-IP communications to compare the cost/benefit of each adaptive method quantitatively. Workers were created with the Unix *rsh* command.

4.1. Applications

We applied the adaptation methods to three distinct parallel applications, an iterative jacobi solver, gaussian elimination with partial pivoting, and gene sequence comparison, to test the performance prediction models and to gain insight into the questions posed in Section 1.0. In all applications, the cost of updating the neighbors T_{nn} was a local operation and patching the communication topology was very inexpensive relative to the other components of the overhead (under 5 ms for both ATM and ethernet) and was omitted from the cost equations. We first determined the cluster-specific cost constants by running a few test programs that performed communication and process creation ($T_{\text{nn}}, T_{\text{pc}}, T_{\text{gc}}, T_{\text{dm}}$). We then ran each application on three problem sizes using three processor configurations to determine the application-specific constants for these cost equations. For T_{comm}, latency and per-byte costs were determined by the communication overhead for transmitting a 1 byte and M byte message within the application, respectively. This provided sufficient accuracy. In general, the procedure for a new application consists of first determining the T_{comm} and T_{comp} equations and then running several problem instances (three or more) to determine the new cost constants.

4.1.1. STEN. STEN is a five-point stencil iterative jacobi solver for solving Laplace's equation $u_{xx} + u_{yy} = 0$ on the unit square. Using finite-differences, a 2-D grid is imposed over this domain with the grid points u_{ij} related in the following way: $-u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} + 4u_{i,j} = 0,\ i,j = 1,\ldots,N$. A grid of size N produces a linear system that contains N^2 equations corresponding to N^2 interior grid points. The SPMD implementation of STEN divides the $N \times N$ grid into a number of contiguous rows and assigns them to workers as depicted in Fig. 1a. The resulting communication topology of STEN is a 1-D in which workers transmit border rows north or south during each iteration. In STEN, the size of each data point E_{size} is 8 bytes. The number of iterations of STEN is normally unknown since it runs until convergence. For convenience, we have added an iteration parameter that allows us to control the experiments. For the stencil application in the UltraSparc cluster, we determined

$$\begin{aligned} T_{\text{nn}} &\approx 0 \qquad \text{(and is ignored)}, \\ T_{\text{comm}}(N,P,j,i) &\approx 5 + 0.00219 \times 8\text{N ms} \qquad \text{[for ethernet]}, \\ &\approx 3 + 0.00010 \times 8\text{N ms} \qquad \text{[for ATM]}, \end{aligned}$$

$\frac{5}{3}$ ms are the latency and 0.00219/0.0001 are the per byte transfer costs, respectively, for a 1-D communication; $8N$ is the message size;

$$T_{\text{comp}}(N,P,j,i) \approx 0.000263 \times 5ND(j,i)L(j,i) \text{ ms},$$

5 floating point operations per element in the stencil code; the constant 0.000263 is the cost to perform the update of a single element on an unloaded Sparc;

$$T_{\text{exec}}(N,P,j,i) = T_{\text{comm}}(N,P,j,i) + T_{\text{comp}}(N,P,j,i),$$

$$T_{\text{pc}} \approx 330 \text{ ms},$$

process creation cost for a worker binary of size $\sim$ 50 kB;

$$\begin{aligned} T_{\text{gc}} &\approx 10 \text{ ms} \\ T_{\text{dm}}(M) &\approx 1 + 0.00103M \text{ ms} \qquad \text{[for ethernet]}, \\ &\approx 1 + 0.00009M \text{ ms} \qquad \text{[for ATM]}, \end{aligned}$$

1 ms is the latency and 0.00103/0.00009 are the per byte transfer costs, respectively, for a point-to-point communication.

Because STEN is regular both in computation and communication, the cost functions do not depend on i. The units of $D(j,i)$ are the number of rows of the grid.

4.1.2. GE. Gaussian elimination with partial pivoting (GE) is perhaps the most well-known direct method for solving a linear system of equations of the form, $Ax = b$, where A is an $N \times N$ coefficient matrix, b is a right-hand vector, and x is the solution vector. GE is a floating-point numeric computation that contains two phases, forward reduction and back substitution. The forward-reduction phase reduces the matrix to an upper-triangular form and is dominant with $O(N^3)$ complexity, while back substitution solves the upper-triangular system and has $O(N^2)$ complexity. The results we present are for the more dominant forward-reduction phase. In the parallel implementation of GE, the implementation performs an initial row-cyclic decomposition of the matrix to give load balance since the amount of computation decreases for rows further down in the matrix. The tasks are arranged in a master–slave broadcast topology for pivot exchange (Fig. 1b). In GE, the size of each matrix value E_{size} is also 8 bytes. The number of iterations of GE is N. For the GE in the UltraSparc cluster, we determined

$$T_{\text{nn}} \approx 0 \qquad \text{(and is ignored)},$$

$$\begin{aligned} T_{\text{comm}}(N, P, j, i) &\approx 1.14 + 0.00114 \times 8\,(N - i)P \text{ ms} \qquad \text{[for ethernet]}, \\ &\approx 0.5 + 0.000137 \times 8\,(N - i)P \text{ ms} \qquad \text{[for ATM]}, \end{aligned}$$

1.14/0.5 ms are the latency and 0.00114/0.000137 are the per byte transfer costs, respectively, for a broadcast communication; $8(N - i)$ is the message size at the ith iteration and P determines the total amount of messages (hence the amount of data);

$$T_{\text{comp}}(N, P, j, i) \approx 0.000335(N - i)D(j, i)L(j, i) \text{ ms},$$

for each row held by a worker, the number of entries modified is $(N - i)$; the constant 0.000335 is the cost to perform the reduction of a single element on an unloaded Sparc;

$$\begin{aligned} T_{\text{exec}}(N, P, j, i) &= T_{\text{comm}}(N, P, j, i) + T_{\text{comp}}(N, P, j, i), \\ 6T_{\text{pc}} &\approx 330 \text{ ms}, \end{aligned}$$

process creation cost for a worker binary of size $\sim$ 50 kB;

$$\begin{aligned} T_{\text{gc}} &\approx 10 \text{ ms}, \\ T_{\text{dm}}(M) &\approx 1 + 0.00103 \times M \text{ ms} \qquad \text{[for ethernet]}, \\ &\approx 1 + 0.00009 \times M \text{ ms} \qquad \text{[for ATM]}, \end{aligned}$$

1 ms is the latency and 0.00103/0.00009 are the per byte transfer costs, respectively, for a point-to-point communication.

Unlike STEN, the cost functions do depend on the iteration i. In addition, the amount of data that is transferred (e.g., M in T_{dm}) also depends on i. This dependence is not shown in the generic DM function shown earlier. In the case of GE, DM was modified to reflect an optimized transfer based on i (i.e., for a data transfer at iteration i, there is no reason to transmit 0's in the upper triangle of the matrix for columns $< i$). As with STEN, the units of $D(j, i)$ are number of rows of the matrix.

4.1.3. CL. Complib (CL) is a biology application that classifies protein sequences that have been determined by DNA cloning and sequencing techniques. CL compares a source library of sequences to a target library of sequences. Comparing protein or DNA sequences is a string matching problem over strings of base pairs (A, C, G, T). In this paper, a set of DNA sequence libraries initially randomized for load balance have been used with the Fasta (FA) comparison algorithm [10]. In the parallel implementation of CL, the target library is decomposed across the workers in a load balanced fashion. Each worker compares all of the sequences it is assigned to a sequence in the source library during a single iteration. During each iteration, a new source sequence is sent to each worker. The number of iterations is known and is equal to the number of source sequences. The workers are arranged in a tree with the leaves performing the computation which generates a comparison score for the current source based on the target sequences (Fig. 1c). The amount of computation to do a comparison (by a single worker) is linearly dependent on the size of the current source sequence and the size of its stored target sequences. The data elements of CL are the target sequences of varying size. N is the total number of target sequence blocks and $D(j,i)$ is the number of target sequence blocks stored with the jth worker. Each target sequence block is 5000 bytes and the target sequences are allocated in multiples of this block size ($D(j,i)5000$ is the number of target bytes stored with the worker). The source sequences are also of varying size (given by seq(i) for the ith sequence). The amount of data transferred in a single iteration includes the source sequence sent to each worker and the results sent back up the tree. For CL in the UltraSparc cluster, we determined

$$\begin{aligned} T_{\mathrm{nn}} &\approx 0 \qquad \text{(and is ignored)},\\ T_{\mathrm{comm}}(N,P,j,i) &\approx 1.14P + 0.00130(\mathrm{seq}(i) + 180D(j,i))\ \mathrm{ms} \qquad \text{[for ethernet]},\\ &\approx 1.0P + 0.000538P(\mathrm{seq}(i) + 180D(j,i)) \qquad \text{[for ATM]}, \end{aligned}$$

$1.14/1.0P$ ms are the latency and 0.0013/0.000538 are the per byte transfer costs, respectively, for a tree communication—here the latency depends on P unlike the 1-D and broadcast of STEN and GE; the message size includes the source sequence sent to the worker and the comparison score which is 180 bytes for each target sequence stored with the worker;

$$T_{\mathrm{comp}}(N,P,j,i) \approx 0.00000424D(j,i)5000\mathrm{seq}(i)\,L(j,i)\ \mathrm{ms}$$

cost depends on the number of target sequences stored with worker and the size of the current source sequence; 5000 is the size of an assigned sequence block, the constant 0.00000424 is the cost to perform an integer comparison between a single byte of a target and source sequence on an unloaded Sparc;

$$T_{\mathrm{exec}}(N,P,j,i) = T_{\mathrm{comm}}(N,P,j,i) + T_{\mathrm{comp}}(N,P,j,i),$$

$$T_{\mathrm{pc}} \approx 550\ \mathrm{ms},$$

process creation cost for a worker binary of size $\sim$ 100 kB;

$$T_{\mathrm{gc}} \approx 10\ \mathrm{ms},$$

$$\begin{aligned} T_{\mathrm{dm}}(M) &\approx 1 + 0.00103M \text{ ms} \qquad &&\text{[for ethernet]}, \\ &\approx 1 + 0.00009M \text{ ms} \qquad &&\text{[for ATM]}, \end{aligned}$$

1 ms is the latency and 0.00103/0.00009 are the per byte transfer costs, respectively, for a point-to-point communication.

Like GE, the amount of computation and communication per iteration is data dependent in CL and depends on the size of the current source sequence, seq(i), that is being compared to the target library.

4.2. Experimental Results

For the experiments, we initiated a number of adaptive events at different iterations within the applications (Table 1). For all experiments, we start with an initial set of $P = 3$ processors. The cluster was under our control for these initial experiments. To simulate resource sharing, we ran a competing sequential program on one or more nodes. The amount of CPU used by this program was controllable for our experiments. For each application, we ran four problem sizes on both ethernet and ATM networks within the cluster (Table 2). The first question we investigated was how well our analytic model predicted the costs of adaptation. The first set of data indicates that the framework can accurately model the cost of adaptation for all of the adaptation methods and all of the applications within 10%, and often within 5% for both ethernet and ATM configurations (Fig. 3, Tables 3 and 4). In this paper, all data presented are the average of five or more data values (unless otherwise stated). We show a few representative problem instances for each application, and cumulative average performance over all application instances and methods (Tables 3 and 4). In the tables, the percent deviation from the predicted value is shown. In the graphs, *basic* is a run with no adaptations and is shown as a baseline, *load* is a run with a load spike generated for one of the workers, and non- is non-blocking migration. Load simulated the introduction of a competing sequential program on one of the nodes. The results substantiate our claim that the cost of adaptation can be accurately predicted. Also, observe that the migration optimizations of ghosting, streaming, and non-blocking can further reduce the adaptation overhead relative to blocking migration. This was observed for all problem instances. The ghosting data contains an initial blocking migration (to create a ghost), so its actual cost is the plotted cost minus the cost of a blocking migration. Overall, overhead estimation achieved similar accuracy on ethernet (3.8%) and ATM (5.2%). For the methods, accuracy was also high ($<10\%$) although there was more variance observed on ethernet as compared to ATM. Similarly, the applications had comparable accuracy: STEN (4.4%), GE (5.2%), and CL (4.2%).

The second question is how well the analytic model predicted the benefit of adaptation. The next set of data indicates that the benefit of adaptation can also be predicted with the same high accuracy as above (Fig. 4, Tables 5 and 6). Again, we present results for representative problem instances and cumulative average performance (Tables 5 and 6). We show the total elapsed time which includes the

TABLE 1

Adaptation Parameters

Adaptive event	Iteration	Comments
Leave-1	20	Worker #1 leaves at this iter
Leave-2	20, 100	Workers #1 and #2 leave at these iters, respectively
Migrate	10	Migrate to an unloaded processor
Migrate-stream	10	Migrate-stream to an unloaded processor
Migrate-ghost	100	Ghost was done following migrate event (which was needed to create the ghost) at iter 10
Migrate-non-block	10	Migrate-non-block to an unloaded processor
Load	10	(1 worker is given $L[j] = 2.0$; double the load)
dlb	100	dlb was done following load event at iter 10 (1 worker is given $L[j] = 2.0$; double the load)
Add-1	10 (GE), 20 (STEN), 20 (CL)	1 idle processor is added at these iters, respectively, for each appln
Add-2	100 (GE), 50 (STEN), 30 (CL)	2 idle processors ...
Add-3	220 (GE), 85 (STEN), 50 (CL)	3 idle processors ...
Add-4	330 (GE), 140 (STEN), 60 (CL)	4 idle processors ...
Add-5	380 (GE), 220 (STEN), 70 (CL)	5 idle processors ...

execution time, adaptation benefit, and adaptation cost all together. Overall, prediction accuracy is similar for ethernet (2.1%) and ATM (1.7%). Similarly, the applications had comparable accuracy: STEN (1.6%), GE (2.3%), and CL (1.8%). The results for adding processors is particularly important because it represents the benefit/cost of performing opportunistic adaptation. It helps answer the following questions. If resources become available during the course of execution, is there a benefit to dynamically adding them to the running application? At what point does

TABLE 2

Application Instances

Application	
STEN	$N = 400, 600, 800, 1000$ ($N \times N$ grid)
GE	$N = 400, 600, 800, 1000$ ($N \times N$ matrix)
CL	$(T = 50, S = 100), (T = 100, S = 100), (T = 200, S = 100), (T = 200, S = 200)$ T is the number of target sequence blocks and S is the number of source sequences (S also determines the number of iterations)

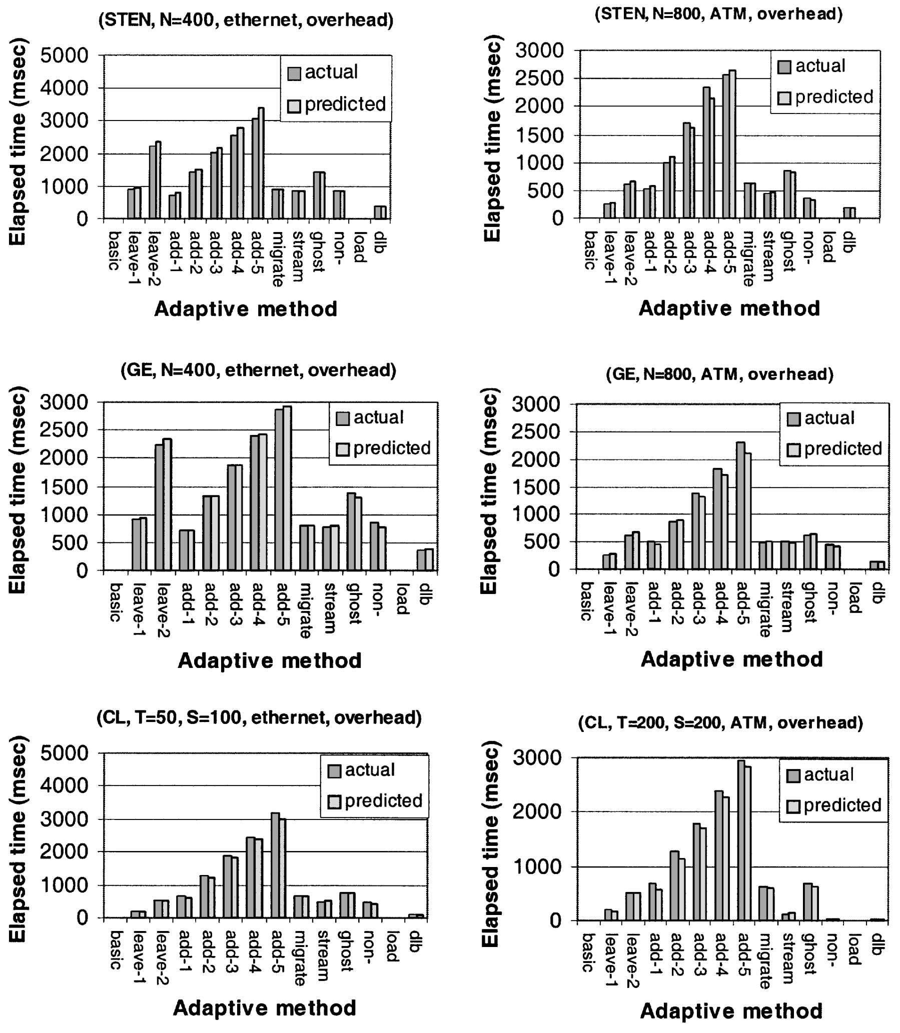


FIG. 3. Predicting adaptation overhead for STEN, GE, and CL on ethernet and ATM.

TABLE 3

Result Summary: Overhead Accuracy (Ethernet)

Adaptive method	STEN	GE	CL	Overall
Add	0.069	0.047	0.034	0.050
Leave	0.051	0.043	0.017	0.037
Migrate	0.008	0.009	0.010	0.009
Migrate-stream	0.019	0.025	0.059	0.034
Migrate-ghost	0.015	0.031	0.015	0.020
Migrate-non-block	0.037	0.046	0.073	0.052
dlb	0.034	0.055	0.089	0.059
Overall	0.042	0.039	0.032	0.038

TABLE 4

Result Summary: Overhead Accuracy (ATM)

Adaptive method	STEN	GE	CL	Overall
Add	0.049	0.040	0.039	0.043
Leave	0.056	0.089	0.028	0.057
Migrate	0.050	0.072	0.047	0.056
Migrate-stream	0.045	0.066	0.089	0.067
Migrate-ghost	0.023	0.046	0.043	0.037
Migrate-non-block	0.033	0.068	0.059	0.053
dlb	0.017	0.078	0.063	0.053
Overall	0.046	0.066	0.053	0.052

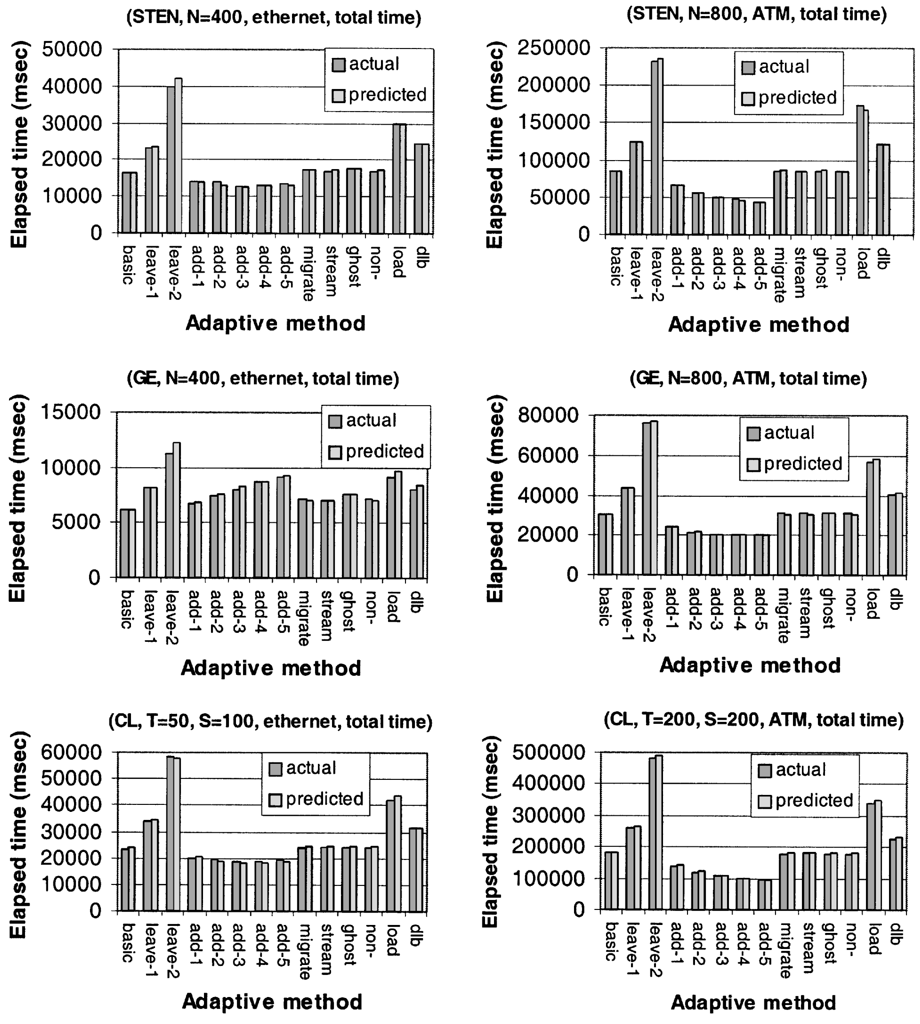


FIG. 4. Predicting total execution time for STEN, GE, and CL on ethernet and ATM. This includes the prediction of adaptation cost and benefit.

TABLE 5

Result Summary: Elapsed Time Accuracy (Ethernet)

Adaptive method	STEN	GE	CL	Overall
Add	0.021	0.040	0.024	0.028
Leave	0.031	0.024	0.010	0.022
Migrate	0.009	0.031	0.016	0.019
Migrate-stream	0.008	0.009	0.018	0.012
Migrate-ghost	0.008	0.018	0.018	0.015
Migrate-non-block	0.011	0.016	0.005	0.011
Load	0.026	0.070	0.027	0.041
dlb	0.004	0.043	0.009	0.019
Overall	0.014	0.032	0.018	0.021

the overhead of adding resources begin to diminish the overall performance of the application? We will return to this question shortly.

The third question of interest is the sensitivity of the adaptation method to different load conditions: which adaptation method is best? We simulated load by the arrival of a competing sequential application whose CPU utilization (i.e., load) we controlled ($0.1 = 10\%, 0.2 = 20\%, 1.0 = 100\%$ of its running time). For this study, the competing application arrived approximately at the midpoint iteration of each application and the adaptation response was initiated 10 iterations later. The results indicate that migration was very competitive with dynamic load balancing and in fact preferable, particularly as the load imbalance increases. We show results for a single problem size for each application, STEN and GE ($N = 800$), and CL (target = 100 sequence blocks and source library = 100 sequences) (FIG. 5). As a baseline, we also show the results of having no load induced (basic) and not adapting even if load is initiated (load-no-dlb). At high loads, the comparative performance of migration and dynamic load balancing is not surprising as the increased load serves to decrease the overall compute power of the cluster, but at low loads it was more

TABLE 6

Result Summary: Elapsed Time Accuracy (ATM)

Adaptive method	STEN	GE	CL	Overall
Add	0.026	0.014	0.022	0.021
Leave	0.013	0.011	0.013	0.012
Migrate	0.004	0.013	0.017	0.011
Migrate-stream	0.016	0.019	0.017	0.017
Migrate-ghost	0.014	0.010	0.017	0.010
Migrate-non-block	0.004	0.015	0.021	0.013
Load	0.038	0.037	0.028	0.034
dlb	0.009	0.014	0.011	0.011
Overall	0.018	0.014	0.019	0.017

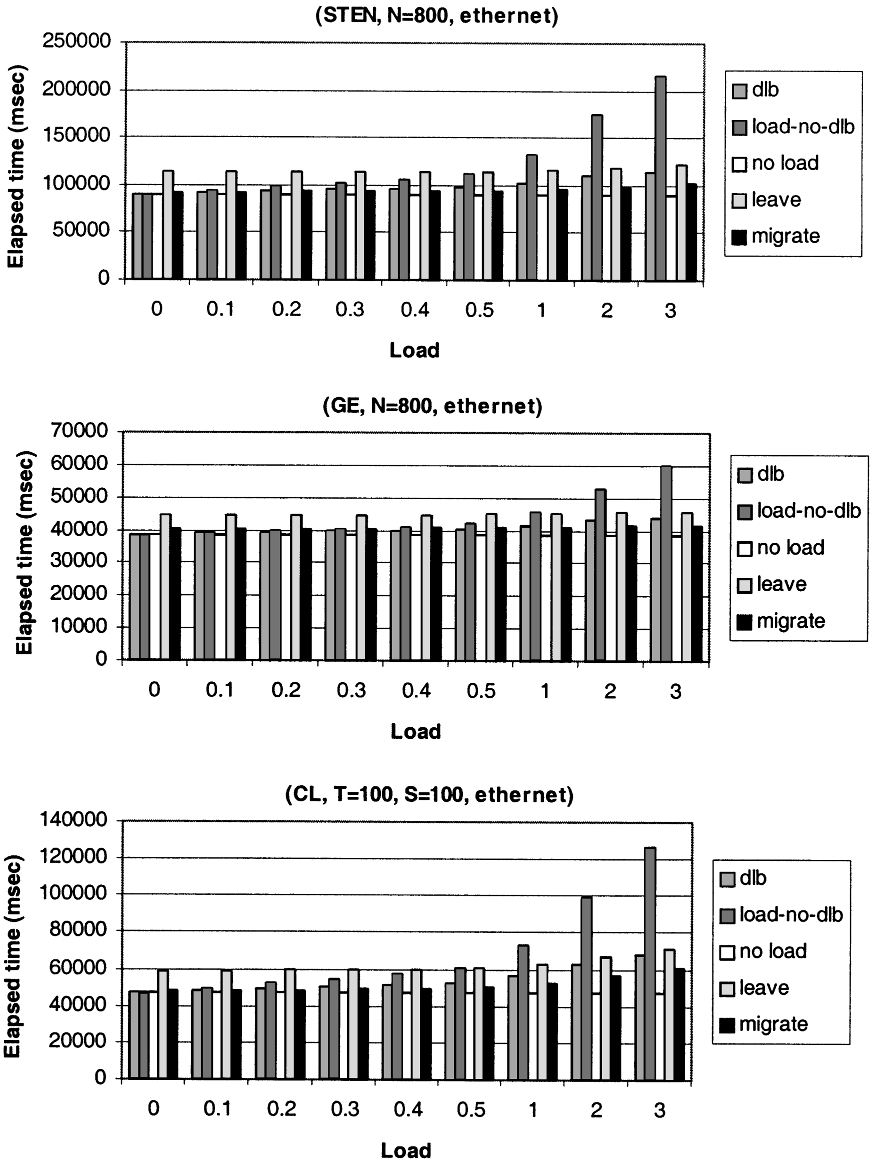


FIG. 5. Comparative performance sensitivity to amount of load imbalance.

surprising to see the penalty for dynamic load balancing to be so easily amortized. In addition, under very light load, the best action may be to take no adaptive action at all. The results also indicate that while migration appears to always be beneficial relative to removing a processor, this distinction begins to narrow significantly under high load. Under high loads if a free processor does not exist, the results suggest that it may be preferable to simply vacate the processor. These trends were observed for all problem sizes that we ran on both ATM and ethernet.

These results, however, depend on when the load and adaptation events are initiated (in this case in the middle of each execution). The fourth question is the

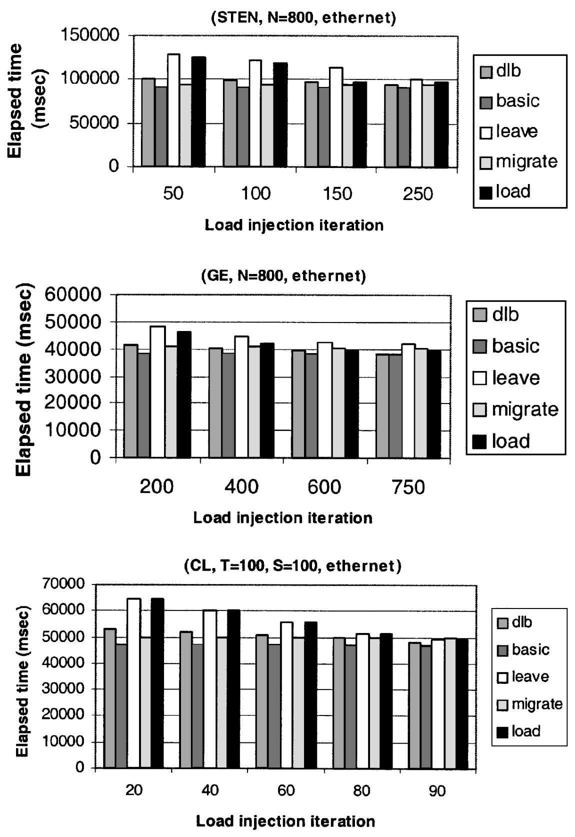


FIG. 6. Comparative performance sensitivity to adaptation method temporal placement.

sensitivity of the overall performance of the adaptation method to *when* these events occur. For this study, we held the amount of load imbalance constant (0.5% or 50% performance degradation on one node), but varied when the load was spiked (i.e., when the competing application was run). The adaptation response was initiated 10 iterations following the detection of load increase. The results show that the comparative performance depends on when the load spike occurs (Fig. 6). The results are shown for an instance of each application, but the same pattern was observed for other problem sizes on ATM and ethernet. As the load spike occurs later, migration becomes an increasingly expensive option relative to dynamic load balancing or to simply taking no adaptive action at all. These results confirm our intuition that the best response to a load event depends on the degree of load imbalance and when it is detected.

Since migration was established to be a competitive adaptation method, the next question we investigated was the comparative performance (overhead) of the different migration methods and their sensitivity to problem size (Fig. 7). The results show that the optimized migration strategies provided improvement over standard migration with respect to overhead. In addition, the best method depends on the problem size. In almost all cases, ghost-based migration was the obvious choice but this depends on the existence of a ghost worker from a prior migration or processor

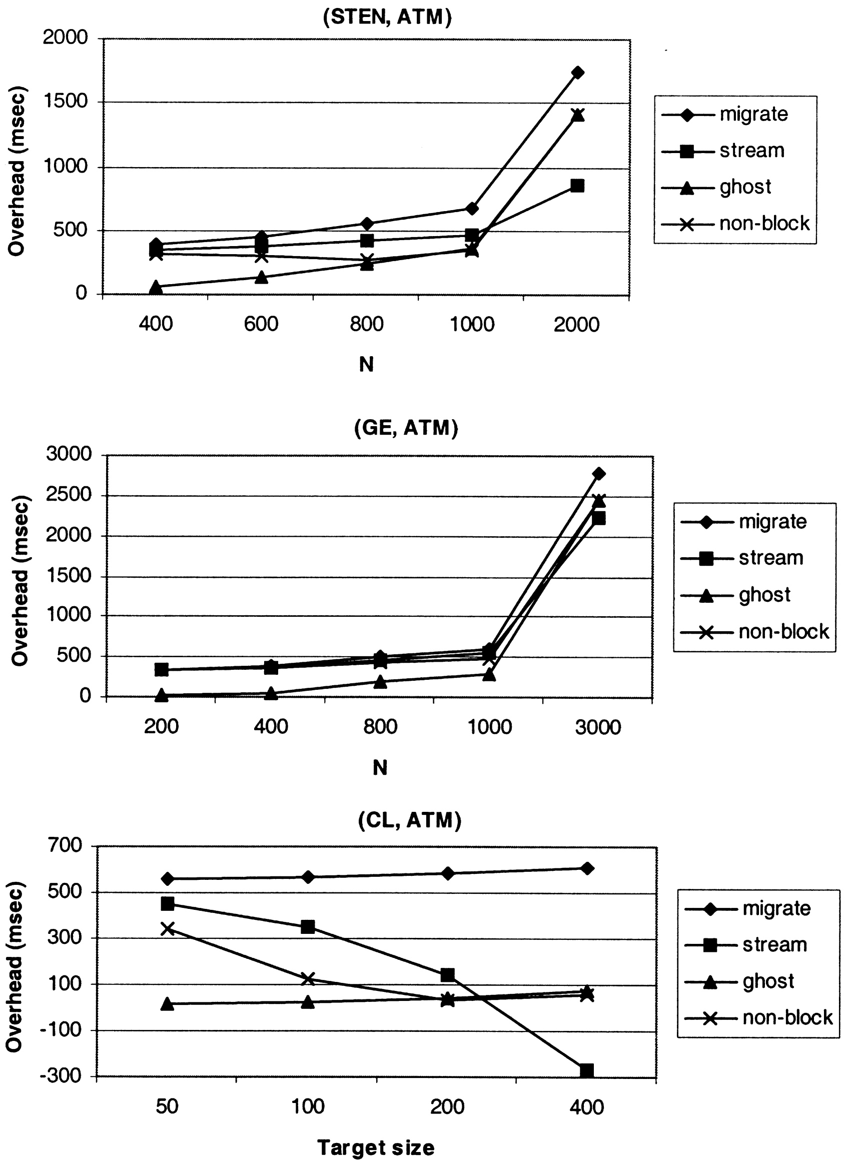


FIG. 7. Adaptation overhead sensitivity to application problem size.

removal (leave) event. However, not all systems will allow ghosts to linger. Following ghost, non-blocking migration was the second best as it can hide the process creation overhead when $T_{\mathrm{pc}} < T_{\mathrm{comp}}$. For larger problems when $T_{\mathrm{comp}} > T_{\mathrm{pc}}$, streaming becomes the best because it reduces the overhead by an amount proportional to T_{comp}. As T_{comp} grows, the benefit of streaming may actually lead to a reduction in overall execution time! This is seen for CL the most computationally intensive of the applications. For large target sizes, T_{comp} was so large that streaming can actually reduce the total elapsed time due to the parallelism provided by streaming. In this case, the effective overhead of streaming is actually negative in value. A negative value is certainly possible given our formulation of $T_{\text{migrate-stream}}$. For large values of T_{comp}, the performance of ghost migration and non-blocking was nearly identical. This is not surprising as once T_{pc} is fully overlapped then this is equivalent to a zero cost process creation. For CL, given a target size $\leqslant 200$, non-blocking migration is best (assuming no ghost workers); above this, streaming is best. For very large problems, streaming even outperformed the use of ghost workers. However, it is likely that performance can be further improved by combining ghosts with streaming.

Our prior results have established that high prediction accuracy for adaptation cost and benefit is feasible, but that method performance was sensitive to the degree of load imbalance, temporal occurrence of adaptive events, and problem size. The next logical step is to show how an adaptive run-time system could utilize these models to improve application performance by intelligent method selection. Such a run-time system could use these cost models to determine if an adaptive response is in the best interest of the application, and which method is most appropriate. We have implemented such a scheme within a run-time system used by the STEN and CL applications and examined automated adaptive method selection in response to two events: (1) discovery of a new processor and (2) the presence of external CPU load.

The discovery of a new processor can be exploited by opportunistic adaptation. The earlier results indicated that the benefit of adding workers depends on the problem size, when the adaptations occur, and the number of iterations. In many data parallel applications, the current iteration relative to the total number of iterations will be known or can be estimated as was the case with GE, STEN, and CL. However, there is always a point of diminishing returns both due to the overhead of adding processors and the ability of the application to exploit additional parallelism. As the problem size increases, the benefit of additional processors increases because these problems typically run longer, can better amortize the adaptation overhead, and contain more parallelism. We ran several problem instances and announced the availability of free nodes periodically during the application execution. Our library added free nodes when their benefit was predicted to exceed their cost (via the cost functions). We repeated the experiments (by announcing free nodes at the same point in time), and had the system choose $1, 2, \ldots, 7$ additional nodes (Fig. 8). Our scheme always chose the appropriate number of nodes, and also outperformed greedy adaptation which would always grab a free processor (i.e., 7). We observed that our adaptive library avoided new nodes when the benefit was too little given the cost, or when the new node appeared late in the execution (i.e., the current iteration was near the final iteration).

In the next set of experiments, we examined the impact of adaptation due to load events. In these experiments, the runs were performed over the course of a week in which other users were on the cluster. Load detectors were installed on each node and when a load event was detected, the adaptive library either migrated (by selecting any of the different migration versions), performed a DLB, or took no action based on the predicted cost/benefit. We compared this automated selection strategy against one that always performed a migration, always performed a DLB, or always took no action, when the load event was detected. We ran experiments using each adaptive strategy one after the other and then repeatedly to ensure that

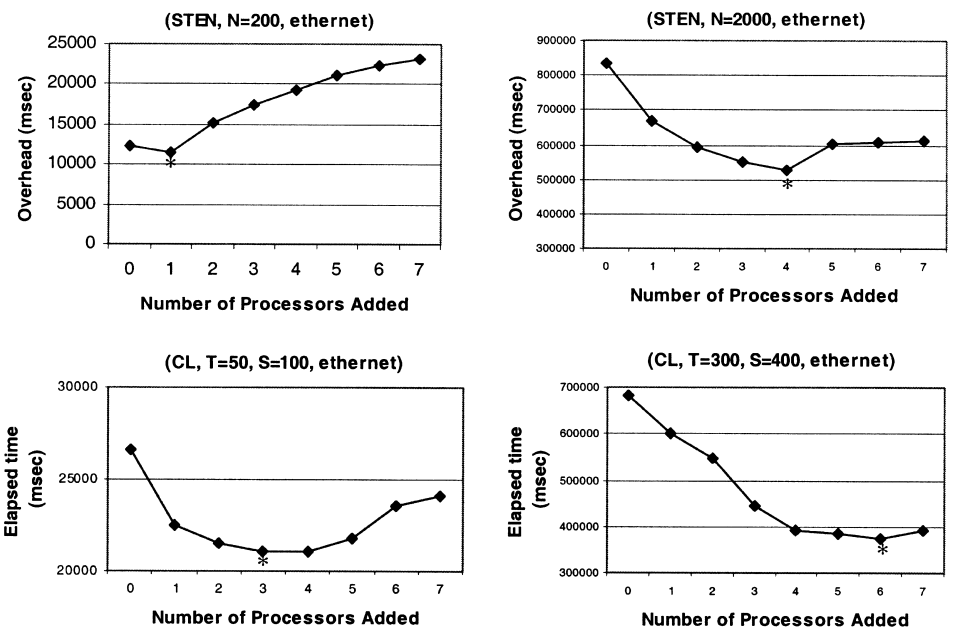


FIG. 8. Adaptive method selection for opportunistic adaptation. The number of processors marked * is the point picked by automated selection.

each technique was run in similar load conditions. The results indicated that automated selection outperformed any single adaptive technique (Fig. 8, Table 7). In our traces, we observed that automated selection was able to take no action for low-load events or for events that occurred near the end of the application as opposed to migrate- or DLB-only which always took action in response to a load event. In addition, for very high loads, our selection strategy migrated when DLB-only did not, and for moderate loads it performed a DLB when migration-only did not. Thus, it exploited the advantages of each approach. For the larger problem instances, automated selection also correctly determined that streaming would be the best migration strategy in terms of overhead (as predicted from Fig. 7).

The results establish the feasibility of predicting the cost/benefit of adaptation and exploiting it for improved application performance in small-to-medium size clusters and shared workstation networks. We believe that the majority of our results will be valid in larger clusters (e.g., prediction accuracy should not depend on cluster size). In clusters of small-to-medium size, adaptation based on the use of a single worker for data collection and transmission is feasible. However, for larger cluster systems more distributed data collection and transmission schemes are needed within the adaptive methods.

5. SUMMARY

Exploiting the full power of shared commodity clusters and workstation networks for executing parallel applications requires the ability to adapt to changing resource

TABLE 7

Prediction-Based Adaptation

Adaptive method	STEN ($N = 200$)	STEN ($N = 2000$)	CL ($T = 100,\ S = 400$)	CL ($T = 300,\ S = 400$)
Prediction-based	14 137	1 019 537	238 972	740 201
Migrate-only	16 776	1 072 418	246 037	755 459
DLB-only	15 075	1 111 583	277 375	748 426
No action	19 246	1 631 781	424 863	1 182 726

availability. We examined adaptivity to the most common form of resource fluctuation, CPU sharing, for three representative SPMD applications. We investigated answers to the following four questions: When should the application adapt? How should the application adapt? What is the cost of adaptation? What is the benefit of adaptation? To answer these questions, we developed a series of quantitative performance prediction models that were general enough to accurately characterize the behavior of three distinct parallel applications on two different network types (ATM and ethernet).

Collectively, the results indicate that the cost and benefit of adaptation can be characterized with high accuracy and that different methods perform best under different situations. We found that surprisingly, migration was very competitive with dynamic load balancing. In addition, we showed that cost functions can be used to predict the best adaptation method as a function of problem size, amount of load imbalance, and temporal placement of adaptation events. We then implemented a run-time system that made adaptive method selections automatically based on these models. We showed that superior performance can be achieved as compared to more monolithic adaptation strategies that are commonly implemented. Experience with three applications that differ in structure, communication topology, and degree of regularity suggests to us that these results are likely to be applicable to a wide variety of SPMD data parallel applications. Future work includes extending our models to heterogeneous clusters and larger cluster systems. We also plan to apply the performance models to the problem of parallel job scheduling in which jobs can dynamically shrink and grow. In this context, predicting the cost/benefit of adding and removing processors will be a very useful input to a space-sharing parallel job scheduler.

ACKNOWLEDGMENT

This work was sponsored in part by the Army High Performance Computing Research Center (AHPCRC) under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement DAAD19-01-2-0014.

REFERENCES

1. A. Acharya, G. Edjlali, and J. Saltz, The utility of exploiting idle workstations for parallel computation, *in* "Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modelling of Computer Systems," Seattle, WA, 1997.
2. Y. Artsy and R. Finkel, Designing a process migration facility: The Charlotte experience, *IEEE Comput.* **4**(1), (September 1989), 22–28.
3. F. Berman *et al.*, Application-level scheduling on distributed heterogenous networks, *in* "Proceedings of Supercomputing '96," Pittsburgh, PA, November 1996.
4. J. Casas *et al.*, Adaptive load migration systems for PVM, *in* "Proceedings of Supercomputing," Washington, DC, 1994.
5. M. Colajanni and M. Cermele, DAME: An environment for preserving the efficiency on data parallel computations on distributed systems, *IEEE Concurrency* **5**(1), (January 1997), 41–55.
6. F. Douglis and J. Ousterhout, Process migration in the sprite operating system, *in* "Proceedings of the 7th International Conference on Distributed Computer Systems," Berlin, Germany, 1987.
7. D. Eager, E. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing, *in* "Proceedings of the 1988 ACM SIGMETRICS International Conference on Measurement and Modelling of Computer Systems," Madison, WI, 1988.
8. S. M. Figueira and F. Berman, Modeling the effects of contention on the performance of heterogeneous applications, *in* "Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing," Syracuse, NY, August 1996.
9. D. Gelernter *et al.*, Adaptive parallelism and piranha, *IEEE Comput.* **28**(1), 1995, 46–49.
10. A. S. Grimshaw, E. A. West, and W. R. Pearson, No pain and gain!—Experiences with mentat on biological application, *Concurrency: Pract. Exp.*, **5**(4) (July 1993), 309–328.
11. M. Harchol-Balter and A. B. Downey, Exploiting process lifetime distributions for dynamic load balancing, *ACM Trans. Comput. Syst.* **15**(3) (1997), 253–285.
12. J. K. Hollingsworth and P. J. Keleher, Prediction and adaptation in active harmony, *in* "Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing," Chicago, IL, July 1998.
13. M. J. Litzkow *et al.*, Condor—A hunter of idle workstations, *in* "Proceedings of the 8th International Conference on Distributed Computing Systems," San Jose, CA, June 1988.
14. N. Nedeljkovic and M. J. Quinn, Data parallel programming on a network of heterogeneous workstations, *Concurrency: Pract. Exp.* **5**(4) (June 1993), 257–268.
15. J. Pruyne and M. Livny, Parallel processing on dynamic resources with Carmi, *in* "Proceedings of the Workshop on Job Scheduling for Parallel Processing," Santa Barbara, CA, 1995.
16. S. Setia *et al.*, Supporting dynamic space-sharing on clusters of non-dedicated workstations, *in* "Proceedings of the 17th International Conference on Distributed Computer Systems," Baltimore, MD, 1997.
17. S. Shao, R. Wolski, and F. Berman, Modeling the cost of redistribution in scheduling, *in* "Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing," Minneapolis, MN, 1998.
18. B. S. Siegell and P. Steenkiste, Automatic generation of parallel programs with dynamic load balancing, *in* "Proceedings of the Third IEEE International Symposium on High Performance Distributed Computing," San Francisco, CA, 1994.
19. J. Smith, A survey of process migration mechanisms, *Oper. Syst. Rev.* **22**(3) (July 1988).
20. H. J. Song *et al.*, The MicroGrid: A scientific tool for modelling computational grids, *in* "Proceedings of SC2000," Dallas, TX, November 2000.
21. N. Spring and R. Wolski, Application level scheduling of gene sequence comparison on metacomputers, *in* "Proceedings of the 12th ACM International Conference on Supercomputing," Melbourne, Australia, July 1998.

22. B. Weissmane *et al.*, Efficient fine-grain migration with active threads, *in* "Proceedings of the Joint 12th International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing," Orlando, FL, March 1998.
23. J. B. Weissman, Prophet: Automated scheduling of SPMD programs in workstation networks, *Concurrency: Pract. Exp.* **11**(6) (May 1999), 301–321.
24. J. B. Weissman, Gallop: The benefits of wide-area computing for parallel processing, *J. Parallel Distributed Comput.* **54**(2) (November 1998), 183–265.
25. J. B. Weissman and X. Zhao, Run-time support for scheduling parallel applications in heterogeneous NOWs, *in* "Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing," Portland, OR, August 1997.
26. M. J. Zaki *et al.*, Customized dynamic load balancing, *in* "Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing," Suracuse, NY, 1996.
27. V. Zandy, B. Miller, and M. Livny, Process hijacking, *in* "Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing," Redondo Beach, CA, 1999.

JON B. WEISSMAN received the B.S. from Carnegie-Mellon University in 1984, and the M.S. and Ph.D. from the University of Virginia in 1989 and 1995, respectively, all in computer science. He has been an assistant professor of computer science at the University of Minnesota since 1999. He was an active member of the Mentat and Legion research groups while at the University of Virginia. His current research interests are in resource management in parallel and distributed systems and Grid computing. He is the architect of two software systems, Prophet and Gallop, that provide automated scheduling support in Grid systems. His research is supported by NSF and the AHPCRC.