Dyn-MPI: Supporting MPI on Non Dedicated Clusters[*]

D. Brent Weatherly
David K. Lowenthal
Mario Nakazawa
Dept. of Computer Science
The University of Georgia
Athens, GA 30602

Franklin Lowenthal
Dept. of CIS
California State University
Hayward, CA 94542

ABSTRACT

Distributing data is a fundamental problem in implementing efficient distributed-memory parallel programs. The problem becomes more difficult in environments where the participating nodes are not dedicated to a parallel application. We are investigating the data distribution problem in non dedicated environments in the context of explicit message-passing programs.

To address this problem, we have designed and implemented an extension to MPI called Dynamic MPI (Dyn-MPI). The key component of Dyn-MPI is its run-time system, which efficiently and automatically redistributes data on the fly when there are changes in the application or the underlying environment. Dyn-MPI supports efficient memory allocation, precise measurement of system load and computation time, and node removal. Performance results show that programs that use Dyn-MPI execute efficiently in non dedicated environments, including up to almost a three-fold improvement compared to programs that do not redistribute data and a 25% improvement over standard adaptive load balancing techniques.

1. INTRODUCTION

Distributed parallel architectures, such as clusters of workstations, deliver high performance and scalability. The most common way to write programs for distributed-memory machines is to use a message-passing library such as the Message Passing Interface (MPI) library [1], which is portable across a variety of distributed-memory machines.

One complication in writing MPI programs is that the nodes may be *non dedicated*, meaning that multiple users may be competing for nodes while a 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.

[*]This research was supported by NSF Grants CCR-9733063 and ITR-0081789. Email addresses for the first three authors are {dbrentw,dkl,nakazawa}@cs.uga.edu and the fourth author is flowenth@csuhayward.edu.

A non dedicated 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 minimizes communication and perfectly balances the load. In general, balancing the load on a non dedicated machine is difficult because the amount of CPU time (as well as physical memory) allocated to a parallel program fluctuates.

We have addressed this problem by designing and implementing what we call the Dynamic Message Passing Interface (Dyn-MPI), which is an extension to MPI. A programmer need only write a program using our extended MPI interface; when the application or the underlying environment changes, the Dyn-MPI run-time system redistributes data *automatically*.

In particular, Dyn-MPI contains the following novel components:

- an efficient, uniform memory allocation scheme for *both* dense and sparse matrices,
- a mechanism for determining accurately both system load and execution time, including situations in which the computation itself is unbalanced *and* a node is busy,
- a scheme that determines effective data distributions, improving upon relative power based methods [2], and
- a facility for *removing* nodes from the computation when their participation degrades performance.

Our experiments show that on a non dedicated cluster, Dyn-MPI determines distributions that result in up to almost a three-fold improvement compared to taking no action and a 25% performance improvement when physically removing nodes.

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

2. MODEL

Dyn-MPI is an extension of MPI. We chose the explicit message passing model because (1) it is the model that is most often used by computational scientists and (2) it is high-level enough to be portable and low-level enough to lead to an efficient program. Our other options were (1) the user writes a sequential program, and a parallelizing compiler is used (e.g., SUIF [3]), (2) a data parallel language like HPF [4], and (3) OpenMP with a software DSM for communication [5]. While these alternatives make programming as

```
// regular MPI initialization omitted
for (t = 0; t < num_iters; t++) {
  for (i = start_iter; i <= end_iter; i++) {
    for (j = 1; j <= N; j++)
      A[i][j] = F( B, i, j );
  }
  if (rank > 0)
    MPI_Send(B[start_iter], 1, MPI_DOUBLE, rank-1, ..... );
  if (rank < numprocs-1)
    MPI_Recv(B[end_iter + 1], 1, MPI_DOUBLE, rank+1, ..... );
}
```

Figure 1: Example MPI program.

```
// regular MPI initialization omitted
DMPI_init( num_processors, 1, 2, DMPI_BLOCK);
DMPI_register_dense_array("A", &A, 1, N, sizeof(double), MPI_DOUBLE );
DMPI_register_dense_array("B", &B, 1, N, sizeof(double), MPI_DOUBLE );
....
DMPI_init_phase(1, N, DMPI_NEAREST_NEIGHBOR );
DMPI_add_array_access("A", DMPI_WRITE, 1, 0);
DMPI_add_array_access("B", DMPI_READ, 1, 0);

for (t = 0; t < num_iters; t++) {
  start_iter = DMPI_get_start_iter();
  end_iter = DMPI_get_end_iter();

  if DMPI_participating() {

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

    rel_rank = DMPI_get_rel_rank( rank );
    if (rel_rank > 0)
      DMPI_Send(B[start_iter], 1, MPI_DOUBLE, rel_rank - 1, ..... );
    if (rel_rank < DMPI_get_num_active())
      DMPI_Recv(B[end_iter + 1], 1, MPI_DOUBLE, rel_rank + 1, ..... );
  }
}
```

Figure 2: Example Dyn-MPI program.

well as analysis easier they all result in less efficient programs. This is because MPI, as a lower-level model, avoids overheads that may manifest themselves in the higher-level models. For example, SUIF and HPF compilers may not be able to find the best parallelization for a sequential region of code, and software DSM systems have a request/reply model of communication that is less efficient than explicit messages.

In this section we first describe our computational model. Then, we describe how programs are written using Dyn-MPI. For illustrative purposes, we provide user code for a small, synthetic MPI program (Figure 1) and how it would be written in Dyn-MPI (Figure 2).

2.1 Computational Model

Our model is Single Program Multiple Data (SPMD), in which each node executes the same code but references a different subset of distributed data. We assume that applications are iterative and consist of one or more *phases*, which are sections of code comprised of computation followed by communication. Further, while we support N-dimensional arrays, we consider distributing only the first dimension. Our model assumes that applications use either a variable block distribution, in which a contiguous (but possibly unequal) set of iterations are assigned to each node, or a cyclic distribution, in which iterations are assigned to nodes in a modulo fashion. Figure 2 shows a single phase example in which the `i` loop is distributed. Lastly, we assume the existence of an outer loop, the *phase cycle*, that encloses all phases; in the example, this is the `t` loop.

2.2 Programming Model

Figure 2 shows a Dyn-MPI example. It should be noted that this is a degenerate example with little computation and therefore results in a large number of `DMPI` calls. Still, most likely Dyn-MPI would in general be the output of the preprocessor that translates an MPI program (e.g., Figure 1) to a Dyn-MPI program (e.g., Figure 2). This is discussed further below.

Dyn-MPI extends MPI by providing specialized facilities for memory allocation, communication, and node participation. All potentially redistributable arrays must be allocated through Dyn-MPI primitives. This allows Dyn-MPI to effect new data distributions automatically and efficiently. The two functions `DMPI_register_dense_array` and `DMPI_register_sparse_array` are used for this purpose. However, the user code must obtain explicitly the bounds of the partitioned loop of each phase in case these bounds change between different phase cycles. In Figure 2, functions `DMPI_get_start_iter` and `DMPI_get_end_iter` are used for this purpose.

Unlike their dense counterparts, sparse matrices have many different possible structures. In general, an automatic redistribution system that explicitly relocates data must know the underlying structure because sparse matrices have data *and* metadata. In our case, for ease of redistribution, Dyn-MPI stores sparse matrices in a vector of lists format; each element is a (data element/column id) pair. For user convenience, functions to access and update sparse matrices are provided by Dyn-MPI, though the user can access them directly if desired. These functions include an iterator to access each element of a sparse matrix as well as functions to get the next element, set the next element, advance the row, and move to the first element. Note also that the transformation between a customized sparse matrix format and the format used by Dyn-MPI is a straightforward one.

We assume that the communication necessary to maintain data dependencies is performed explicitly by the application code. Dyn-MPI uses Deferred Regular Section Descriptors (DRSDs) [6, 7] to describe array accesses; this enables Dyn-MPI to determine data ownership. DRSDs extend Regular Section Descriptors (RSDs) [8]; 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. A block distribution has one DRSD per node and a cyclic distribution has one or more.

Determining DRSDs for each array is the most challenging part in translating an MPI program to a Dyn-MPI one. While in Dyn-MPI the user currently specifies DRSDs, this step could be automated in many cases. For example, the work in [7] as well as our previous work in [6] modified SUIF [3] to generate DRSDs for each array reference; the translation here would be similar (see below).

Programs in Dyn-MPI need to use *relative ranks* rather than ranks, along with conditional execution using `DMPI_participating`. This is because Dyn-MPI may remove (and potentially later add back) non dedicated nodes from the computation. We denote this as *physically* dropping a node. Hence, ranks may vary over the course of a computation; for example, if node i sends a message to its left neighbor, this may be node $i-1$ at time t, but then change to node $i-2$ at time $t+\delta$. Relative ranks are shown in Figure 2. The alternative to this is to always assign a "removed" node a minimum amount of data, so that ranks, once determined, are static. We denote this as *logically* dropping a node. Our experiments show that the performance difference between logical and physical dropping can be significant (see Section 5).

2.3 Translation of MPI to Dyn-MPI

A transformation from an MPI program to an Dyn-MPI can be divided into two primary parts: the parts that are straightforward and the parts that require sophisticated analysis. Most of the transformation falls into the former category, which can be mostly implemented through a one-to-one replacement. This includes adding `DMPI_init`, `DMPI_register_dense_array`, `DMPI_register_sparse_array`, `DMPI_init_phase`, `DMPI_get_start_iter`, `DMPI_get_end_iter`, and `DMPI_participating`. Interface function `DMPI_init` is called once per program, and its parameters are easily determined from the number of processors and the desired data distribution. Either function `DMPI_register_dense_array` and `DMPI_register_sparse_array` is added for each multidimensional array that is used in a loop nest; which one is needed can be determined from array access patterns. Function `DMPI_init_phase` is called for each phase, and its parameters are mostly determined by just substituting in the loop bounds of that phase. Functions `DMPI_get_start_iter` and `DMPI_get_end_iter` are added before each partitioned loop, and these values become the new loop bounds. Finally, `DMPI_participating` encloses all code within the loop.

The sophisticated part of the transformation involves adding function `DMPI_add_array_access`. One instance of this function must be added for each array reference in a parallel loop. As described in the previous section, it represents a regular section descriptor. While this requires relatively sophisticated compiler techniques, we have implemented this for DSM-based programs in [6]. The primary difference is that RSDs in DSM programs can more easily be determined in a global sense, because the program uses a shared-memory model. Here, the data in the original (source) MPI program is already distributed, so some work would need to be done to convert a local view of a distributed array to a global view. This step is the reverse of the translation used by Fortran D compilers [9].

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 [4], for example. In this approach, the programmer annotates each array with its distribution. There has also been work in supporting a `REDISTRIBUTE` annotation. However, this places the burden on the programmer. In contrast, Dyn-MPI redistributes data automatically.

Compiler techniques to distribute data have also been studied extensively (e.g., [10, 11]). 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 [12, 13, 14, 15]. However, static analysis cannot possibly take into account run-time fluctuations in load between different nodes. Because Dyn-MPI uses run-time analysis, it can make better decisions about how to balance load dynamically.

Approaches employing a run-time system, such as AppLeS [16], SUIF-Adapt [6], CRAUL [2], and the CHAOS group [17] can use run-time information to find an efficient data distribution. This is especially effective in cases in which workload and communication characteristics of a program change at run time. In principle, these approaches show promise to solve the data distribution problem on non dedicated machines, in which multiple machine parameters are changing. In particular, through the Network Weather Service [18], AppLeS was able to determine when to avoid using a processor because its limited memory would cause (expensive) paging. Also, the CHAOS work as well as that in [19] 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 [20], but involve user intervention. Dyn-MPI is distinct from these systems in several ways. First, Dyn-MPI is integrated with MPI as opposed to implemented within a software distributed shared memory [21]. Second, Dyn-MPI performs analysis of whether to drop nodes based on communication/computation ratio, whereas the other approaches drop and add nodes purely based on whether another user is active. Third, Dyn-MPI handles applications with unbalanced computational loads such as particle simulation. Finally, Dyn-MPI supports automatic redistribution of sparse matrices.

The closest work to ours is Adaptive MPI, or AMPI [22, 23]. However, its approach is completely different than Dyn-MPI. AMPI creates several virtual MPI processors per physical processor and then, given different physical processor loads, migrates them as appropriate. In essence, this is a medium-grain approach to the non dedicated data distribution problem, whereas Dyn-MPI is a coarse-grain approach. CHARM [24], a medium-grain threads package, is used to implement AMPI. Another recent project, Tern [25], takes a similar approach to AMPI. The key difference is that a software

DSM-like for migrating thread stacks is used, and customized migration policies can be written by the user or compiler.

We argue that the key advantage of Dyn-MPI is that it leads to more efficient execution than AMPI or Tern. This is because of two reasons. First, the virtualization approach can incur overhead (as with fine-grain approaches) for reasons such as process creation, process management, and poor compiler optimization of fine-grain code [26]. As the number of redistributions increase, either the granularity must decrease or further load balancing becomes difficult. Second, and more importantly, fine-grain programs may have significantly more messages than their coarse-grain counterparts; for example, in a nearest neighbor communication pattern, it is necessary to send one message per boundary edge. This means that a fine-grain approach increases greatly the number of exchanged messages.

On the other hand, the key advantage of AMPI or Tern over Dyn-MPI is that the application requires less modification. In particular, Dyn-MPI requires the programmer to identify data that is to be considered for redistribution as well as write a program that continually updates loop bounds (because the bounds might change). It should be noted, however, that the previous section outlined how a Dyn-MPI program could be generated automatically.

4. IMPLEMENTATION

This section describes the implementation of Dyn-MPI, which consists of four primary components. First, Dyn-MPI needs to carefully allocate memory for efficient and automatic redistribution of dense and sparse matrices, balancing the tradeoff between minimizing the number of messages required and copying of data (Section 4.1). Second, Dyn-MPI must accurately measure both the load on the system and execution time (Section 4.2), but this is complicated by the fact that unpredictable operating system scheduling can result in unexpected measurements when using wallclock timers. Third, Dyn-MPI must determine ideal data distributions for a collection of nodes of varying available processing power (Section 4.3); however, the impact of communication makes this problem difficult. Fourth, redistribution must be effected automatically for both dense and sparse matrices, and node removal must be considered (Section 4.4).

4.1 Memory Allocation

Memory allocation in Dyn-MPI balances the need to maximize data locality and minimize the number of redistribution messages. The key challenge is that redistribution causes data sizes to change across nodes, which can cause repeated and costly memory allocation and deallocation. Dyn-MPI uses a nearly uniform allocation scheme for both dense and sparse matrices; these are discussed in turn below.

4.1.1 Dense Matrices

The memory allocation scheme used by Dyn-MPI for dense matrices is to project all arrays onto two dimensions using a vector of vectors style allocation: the first is the outermost dimension of the original *n* dimension array, while the second consists of *extended rows* of the remaining elements (the product of the remaining *n*-1 dimensions). This allows nodes (1) to communicate entire extended rows with a single message as well as (2) to reuse memory where possible by allowing a newly allocated pointer in the first dimension to point to already existing memory. Figure 3 shows an example of the contiguous allocation compared to our projection method. In the former, all data must shift (a complete reallocation) if a new node is sent new data. In the projection method, a node performs a copy of only the top level vector (which is the size of the first dimension) and then allocates space for the new data.

Our memory allocation method performs significantly better than contiguous allocation. This is because contiguous allocation can cause excessive disk accesses due to complete reallocation of large data structures. More details on an experimental comparison of these methods can be found in [27].

4.1.2 Sparse Matrices

Dyn-MPI must know the format of a sparse matrix so that it can make sure to redistribute data *and* metadata when necessary. The memory allocation scheme used by Dyn-MPI for sparse matrices is as similar as possible to that of dense matrices. The only differences are that (1) the extended row is a linked list instead of a vector, and (2) both the element and the column id are stored. Dyn-MPI could even remove (1) above and store a sparse matrix row in a vector, but this would require reallocation of the row if its size ever changes.

4.2 Load Determination and Computation Timing

In order to determine when redistribution is necessary, Dyn-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 check system load at every phase cycle and redistribute if any change is detected, as was done in [17].

Determining system load is a non-trivial problem. Previous research utilized `vmstat` to determine the number of active processes on a node [18], the essential information being the number of processes on the running, ready or blocked queues. However, our experience is that this method is unreliable; processes that have voluntarily relinquished the processor because they are blocked at a receive are not reported by `vmstat`. We have created an application called `dmpi_ps` that uses `ps` to determine active processes and accounts for only those processes that are in either a running or ready state; furthermore, the monitored application is *automatically* included. We configure `dmpi_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, Dyn-MPI must carefully determine the load on all nodes and the true, unloaded execution time of iterations in order to determine ideal data distributions. Our solution allows for a *grace period* during which 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. Per-iteration times are needed for applications such as particle simulation, where the iterations themselves are nonuniform. It is important to understand that we *must* obtain unloaded execution times for each iteration, or the resulting distribution chosen will likely be inefficient—this is because we will not know if an iteration is slow due to the application itself or another process on the system.

Dyn-MPI handles this by choosing between two timing mechanisms (during the grace period) to determine the true unloaded execution time for iterations. The first is information from `/PROC`, and the second is wallclock timers using `gethrtime`. The advantage of using `/PROC` is that it avoids counting time by other processes. Due to the limited granularity of `/PROC`, the latter is used when iterations have execution times less than 10ms. However, because `gethrtime` may include execution of other processes, we must measure over several phase cycle iterations and take the minimum—this removes potential spikes caused by context switches in the middle of an iteration.

4.3 Ideal Data Distributions

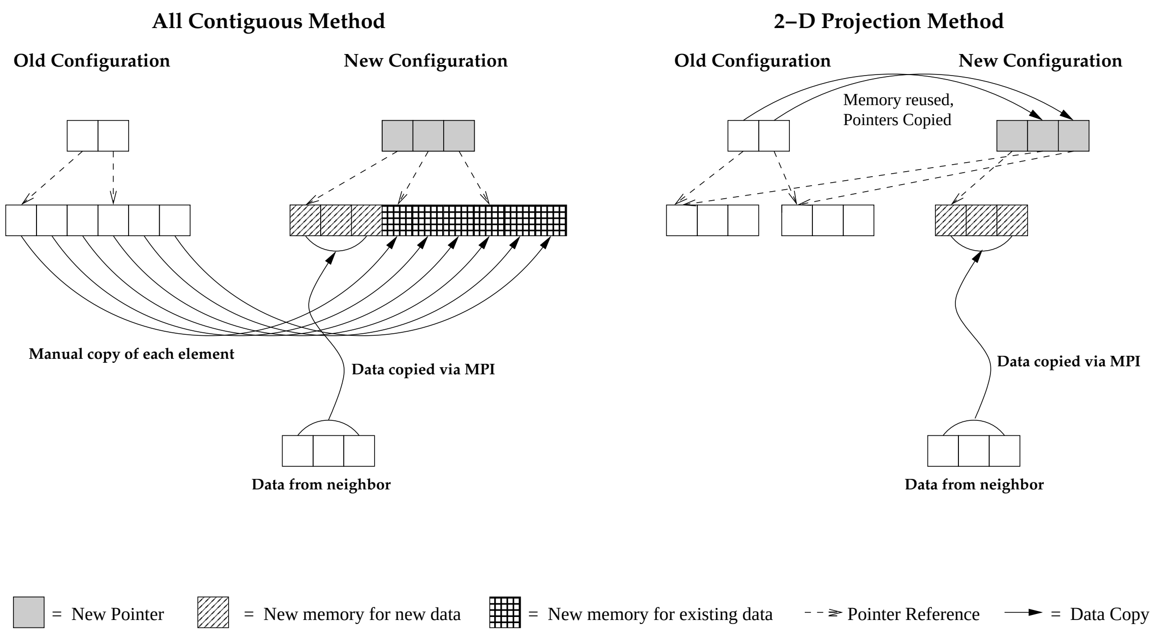

Figure 3: Comparison of memory allocation methods. Shaded elements must be reallocated. Note that in the 2-d projection method, the vectors that hold actual data are contiguous memory for dense matrices and linked lists for sparse matrices.

Given system load and iteration times, the traditional way to determine work proportions is based on the relative power [2] of each node. Our experiments show that this can result in poor distributions by not accounting for communication. We attribute this behavior to the fact that distributing iterations does not take into consideration the computation portion of communication (i.e., communication requires *some* use of the CPU). We found that the best distributions often differed from the relative power based ("naive") distribution.

As a result, our approach is to determine effective distributions by executing micro-benchmarks. We executed several synthetic programs for different computation to communication ratios. To extend the two-node model to multiple nodes, we use an algorithm called *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 change to the iteration assignment to the unloaded nodes. Further details are available in [27].

4.4 Data Redistribution and Node Removal

To effect the new distribution, each node must (1) determine data ownership, (2) deallocate memory that is no longer needed, (3) allocate the required memory for new data owned, (4) update pointers for data that does not change ownership, and (5) schedule communication for data that does. The key to determining how to schedule communication is the DRSDs, as they indicate precisely what data needs to be communicated. In particular, rows must be acquired if they are not local to a node, but the DRSD indicates that these rows are needed—this is borrowed from the techniques used in the Fortran D compiler [9].

As discussed above, our choice to represent a sparse matrix as a vector of lists results in little difference between redistributing dense and sparse matrices. In particular, the only added complexities when redistributing sparse matrices are that (1) when a row is sent from node to another, it must be packed into a vector, and (2) the row must be unpacked on receipt and converted to a list. It should be noted that the cost of this uniformity between dense and sparse matrices is in efficiency; a list is more expensive to traverse than a vector. However, we note that one solution is for users to copy the data to and from the format used by Dyn-MPI to their own, using the format of Dyn-MPI *only* when measurements are needed and redistribution is performed. As redistribution is likely infrequent, this cost will likely be amortized over the entire computation.

After redistribution, Dyn-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. This is performed during a post-redistribution grace period (currently ten phase cycle iterations), which 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 (which we *can* predict with high accuracy, because there is no unpredictability due to loaded nodes). If we predict that the unloaded configuration is best, the loaded nodes are physically removed from the computation. This is why Dyn-MPI requires relative ranks to account only for those nodes that are participating; Dyn-MPI assists by re-assigning relative ranks whenever nodes are removed.

The complication for physical removal of nodes involves keeping the nodes current on all global state information. For example, if the factors necessary to terminate an application are reached by an appropriate node and a termination message is sent to all other nodes, removed nodes must receive this message. At the same time, we do not want the participating nodes to be delayed by removed nodes or reach an incorrect state due to erroneous data having been received from removed nodes. We modified global communication routines so that removed nodes do *not* participate in the *send in*

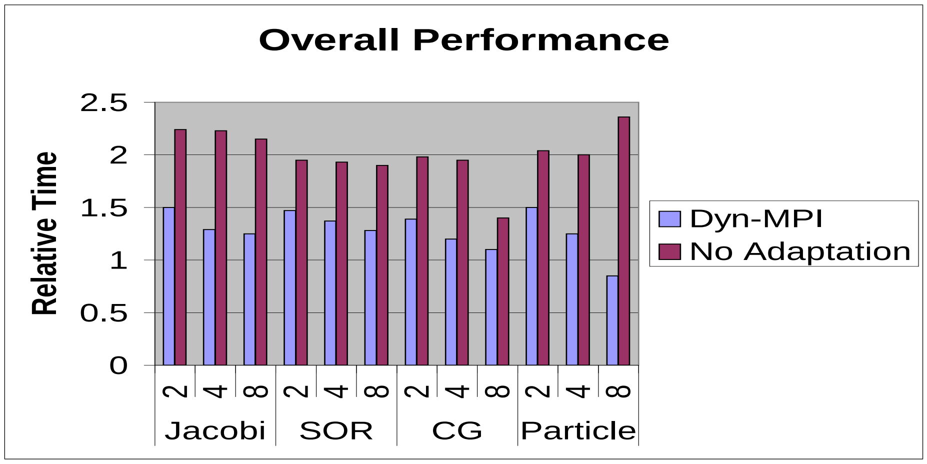

Figure 4: Results for our four applications: Jacobi iteration, SOR, CG, and particle simulation. Each program was run on a system with one non dedicated node, with the competing process introduced on the 10th iteration. All times are relative to the corresponding version with *all* nodes dedicated, so smaller bars are better.

phase, but do participate in the *send out*.

5. PERFORMANCE

This section details the performance of Dyn-MPI using four programs. Jacobi iteration and Red-Black SOR both can be used to solve partial differential equations. Conjugate Gradient, denoted CG, is from the NAS suite [28]; it solves an unstructured sparse linear system. Finally, particle simulation is a scaled down version of MP3D [29], which is from the Splash suite [30].

Section 5.1 presents overall performance of all of these programs in a non dedicated environment. Next, we use Jacobi iteration, SOR, and particle simulation to demonstrate three different features of Dyn-MPI, respectively: multiple redistribution points (Section 5.2), node removal (Section 5.3), and unbalanced computations (Section 5.4). We have also performed a number of synthetic tests to tune our redistribution scheme; these results are available in a related technical report [27].

Most 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. A few tests (Section 5.3) were run on a cluster of Sun Ultra-Sparcs. All user programs are compiled with the `-O2` option. For competing processes, we use programs that execute an infinite loop.

5.1 Overall Results

This section presents the overall performance of all four applications. For Jacobi iteration and SOR, we used an input size of 2048×2048 and iterated 250 times. CG used a size of $14,000 \times 14,000$, and particle simulation used 256×256 cells with an initial distribution of one or two particles per cell, for 200 time steps. For the particle experiment, one node had twice as many particles than each of the others.

Figure 4 displays the results for each program on 2, 4, and 8 nodes. As a baseline, we ran a version of the program with *no* competing processes; in other words, all nodes are dedicated. All results are normalized to this version. We also ran two versions in which we started a competing process on one node on the 10th iteration. One of those versions uses Dyn-MPI and therefore adapts, and the other does not use Dyn-MPI and so never adapts. It should be noted that both of these versions will in general have inferior performance to the version with all nodes dedicated. However, there are rare cases where this may not be true (see below).

Figure 4 shows that in general, the results are encouraging. Dyn-MPI successfully redistributes data to adapt to the competing process, improving the execution time by almost a factor of three compared to no adaptation. The improvement averages 72%, and the slowdown of the Dyn-MPI programs compared to the dedicated version averages only 29%.

Consider in particular the 4-node CG experiment. The version with no competing process takes 37.5 seconds. With a competing process but without adaptation, the time increases to 73.0 seconds, almost a 100% increase. With the competing process and the Dyn-MPI version, the time goes up to only 45.1 seconds, which is a 20% increase. The distribution of data found by Dyn-MPI is to give each of the three unloaded nodes $2/7$ of the work and the loaded node (with one competing process) $1/7$. Hence, the ideal time would be to incur a percentage increase of $2/7-1/4$, which is approximately 12%, or about 4.5 seconds. In practice, Dyn-MPI must also incur overhead to redistribute the data, which in this case takes about 1 second. This means that the additional Dyn-MPI overhead consists of about 2 seconds, which can be explained by unpredictability of the loaded node. Still, the overall Dyn-MPI overhead is quite low.

Finally, our particle simulation experiment has twice as many particles on the node that also has the competing process. The results show that the Dyn-MPI version actually outperforms the version with no competing process. This is because in the Dyn-MPI version, adaptation occurs more quickly. (For more time steps, the Dyn-MPI version would have inferior performance.)

5.2 Multiple Redistribution Points

This section shows the effect of Dyn-MPI on an application with multiple redistributions. Figure 5 breaks down the overall execution time of Jacobi iteration on 4 nodes. 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 with 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. Two experiments were performed: *Short*

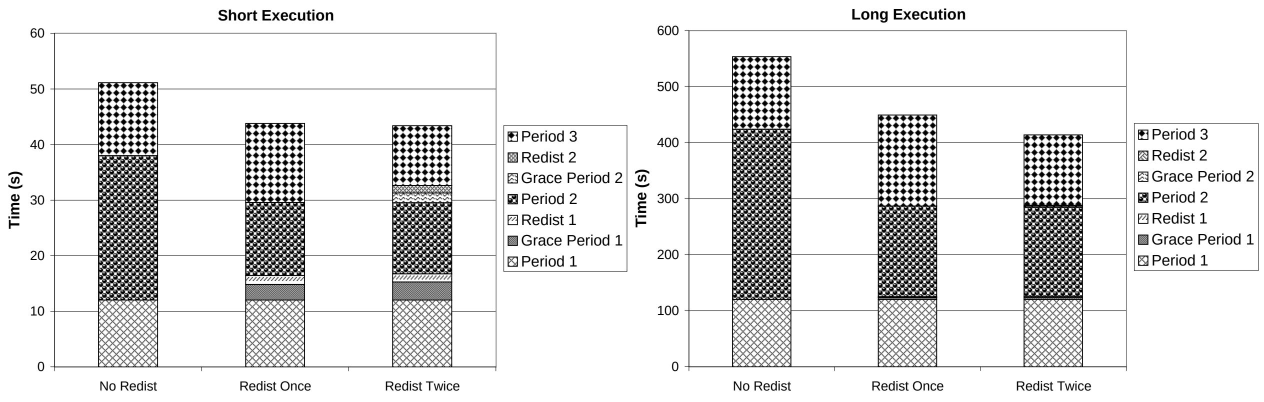

Figure 5: Comparison of the execution of Jacobi iteration. The results are for a 4-node configuration, and the arrays are doubles of dimension 2048x2048. Also, *Period* denotes the execution of Jacobi, *Grace Period* denotes the monitoring period of Dyn-MPI, and *Redist* denotes redistribution.

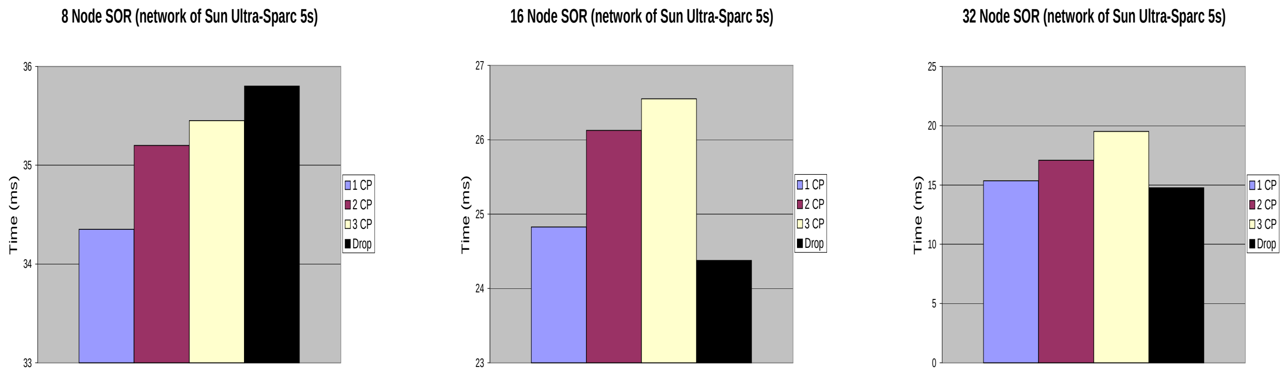

Figure 6: Results of SOR tests when (1) a node has one, two, or three competing processes, and a distribution that includes the loaded node is used (labeled *1 CP*, *2 CP*, or *3 CP*) and (2) the loaded node is removed (labeled *Drop*). Configurations of 8, 16, and 32 nodes were used for 1024×1024 arrays. For readability, the first two graphs do not start at 0, so the differences in the rightmost graph are more pronounced than the ones in the two leftmost graphs.

Execution (period = 50 iterations) and *Long Execution* (period = 500 iterations).

The left-hand side graph clearly shows the speedup that Dyn-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 negates 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% over no redistribution. In this case, redistribution is less than 1% of the total execution time.

5.3 Node Removal

In order to demonstrate the applicability of physical node removal, we now show the results from Red/Black SOR. This program has a smaller ratio of computation to communication than Jacobi, making node removal likely when competing processes are introduced.

We ran tests on a cluster of Sun Ultra-Sparc 5 machines (360 Mhz), with 8, 16 and 32 nodes. Figure 6 shows the results for two-dimensional arrays of size 1024. 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*). It is clear that dropping a node becomes more important with smaller computation/communication ratios, which occurs as the number of nodes increases. Dropping is always worse on 8 nodes, moderately better on 16 nodes (2, 7, and 8% on 1, 2, and 3 CPs), and significantly better on 32 nodes (4, 14, and 25%). This shows that as the number of nodes increases, the benefit of removal (when the computation/communication ratio is low) will also increase.

5.4 Unbalanced Computations

Finally, we show results from an additional particle simulation experiment in order to establish that Dyn-MPI supports unbalanced computations by accurately determining unloaded iteration times. Figure 7 compares the execution time after redistribution when the grace period (GP) is either 1 or 5 phase cycle iterations. The small number of particles used means that each iteration is less than

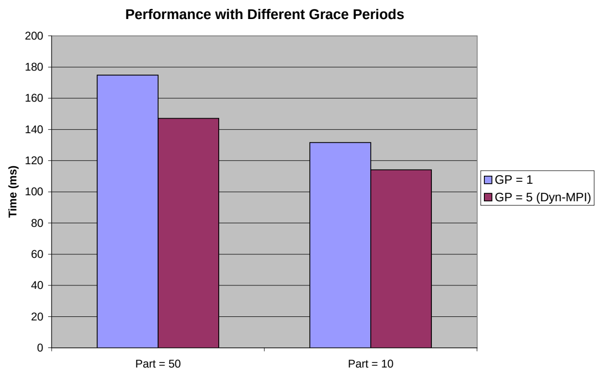

Figure 7: Results of a particle simulation with an 8-node configuration and a grid dimension of 256x256. Grace periods of 1 and 5 iterations are shown. The label *Part* denotes the degree of imbalance as determined by the number of particles in each grid cell in the top half of the rows owned by P_0.

10ms. Hence `gethrtime` (rather than `/PROC`) must be used. As a result, the timings will be affected by context-switching. 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 of the latter. Using *GP = 5* (the default in Dyn-MPI) results in 13% and 16% improvements over using *GP = 1* for 10 and 50 particles (labeled *Part = 10* and *Part = 50*), respectively.

6. SUMMARY

This paper has described our approach to supporting efficient, message-passing programs in environments in which nodes are not dedicated and/or the computation is not balanced. We have designed and implemented the Dynamic Message Passing Interface (Dyn-MPI) to meet these goals. The Dyn-MPI run-time system combines several novel features, including efficient memory allocation supporting automatic redistribution of dense and sparse matrices, accurate determination of system load and execution times, selection of an effective data distribution, and removal of nodes when their participation is detrimental to performance. Moreover, this is done *automatically*. Performance results showed that our system automatically adapts to changes in system load and application behavior with low overhead, resulting in up to almost a three-fold improvement compared to taking no action. Also, when physical node removal is necessary, the resulting performance improvement is up to 25%.

In the future, we would like to expand Dyn-MPI to support competing parallel applications. This will require changes to our load determination technique such that the probability that an application is computing (as opposed to blocking) is considered. Also, we need to investigate methods to accurately predict execution time when nodes are loaded. This will enable us to consider distributions in which *some* loaded nodes are removed, instead of considering only the removal of all of them.

7. REFERENCES

[1] Message Passing Interface Forum MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.

[2] Umit Rencuzogullari and Sandhya Dwarkadas. Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations. In *Eighth ACM PPOPP*, pages 72–81, June 2001.

[3] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Chau-Wen Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

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

[5] Y. Charlie Hu, Honghui Lu, Alan L. Cox, and Willy Zwaenepoel. OpenMP for networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, 2000.

[6] Donald G. Morris and David K. Lowenthal. Accurate data redistribution cost estimation in software distributed shared memory systems. In *Eighth ACM PPOPP*, pages 62–71, June 2001.

[7] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proc. of ASPLOS VII*, pages 186–197, 1996.

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

[9] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communication of the ACM*, 35(8):66–80, August 1992.

[10] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An static performance estimator to guide data partitioning decisions. In *Proceedings of the Third PPOPP*, pages 213–223, April 1991.

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

[12] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of PLDI '93*, pages 112–125, June 1993.

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

[14] Daniel J. Palermo and Prithviraj Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. In *Proceedings of the*

8th LCPC, August 1995.

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

[16] 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.

[17] Guy Edjlali, Gagan Agrawal, Alan Sussman, Jim Humphries, and Joel Saltz. Runtime and compiler support for programming in adaptive parallel environments. *Scientific Programming*, 6(2):215–227, 1997.

[18] 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.

[19] Alex Scherer, Honghui Lu, Thomas Gross, and Willy Zwaenepoel. Transparent adaptive parallelism on NOWs using OpenMP. In *Seventh ACM PPOPP*, pages 96–106, May 1999.

[20] Bongki Moon and Joel Saltz. Adaptive runtime support for direct simulation monte carlo methods on distributed memory architectures. In *Scalable High Performance Computing Conference*, pages 176–183, May 1994.

[21] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4), November 1989.

[22] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Adaptive load balancing for MPI programs. In *International Conference on Computational Science*, pages 108–117, San Francisco, CA, May 2001.

[23] Orion Lawlor, Milind Bhandarkar, and L. V. Kale. Adaptive MPI. TR 02-05, University of Illinois, 2002.

[24] L. V. Kale and Sanjeev Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *OOPSLA '93*, pages 91–108, September 1993.

[25] Jian Ke and Evan Speight. Tern: Migrating threads in an MPI runtime environment. Technical Report CSL-TR-2001-1016, Cornell, November 2001.

[26] Gregory W. Price and David K. Lowenthal. A comparative analysis of fine-grain threads packages. *Journal of Parallel and Distributed Computing (to appear)*.

[27] D. Brent Weatherly, David K. Lowenthal, and Franklin Lowenthal. Dyn-MPI: Supporting MPI on non dedicated clusters (extended version). Technical Report 03-003, University of Georgia, January 2003.

[28] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. RNR-91-002, NASA Ames Research Center, August 1991.

[29] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *Computer*, 26(7):42–50, July 1993.

[30] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.