Dome: Parallel programming in a heterogeneous multi-user environment

José Nagib Cotrim Árabe[1] Adam Beguelin[2]
Bruce Lowekamp[3] Erik Seligman Mike Starkey[4]
Peter Stephan

April 1995
CMU-CS-95-137

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]Currently visiting CMU from Universidade Federal de Minas Gerais, Brasil, with support provided by CNPq.
[2]Joint appointment with the Pittsburgh Supercomputing Center.
[3]Partially supported by NSF Graduate Research Fellowship.
[4]Currently at IBM Canada Laboratory

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035. Partial support also was provided by ARPA under Contract Number DABT63-93-C-0054, and by CNRI.

Keywords: distributed, parallel, supercomputing, load balancing, check-pointing, fault tolerance, network, heterogeneous, object-oriented

Abstract

Writing parallel programs for distributed multi-user computing environments is a difficult task. The Distributed object migration environment (Dome) addresses three major issues of parallel computing in an architecture independent manner: ease of programming, dynamic load balancing, and fault tolerance. Dome programmers, with modest effort, can write parallel programs that are automatically distributed over a heterogeneous network, dynamically load balanced as the program runs, and able to survive compute node and network failures. This paper provides the motivation for and an overview of Dome, including a preliminary performance evaluation of dynamic load balancing for distributed vectors. Dome programs are shorter and easier to write than the equivalent programs written with message passing primitives. The performance overhead of Dome is characterized, and it is shown that this overhead can be recouped by dynamic load balancing in imbalanced systems. Finally, we show that a parallel program can be made failure resilient through Dome's architecture independent checkpoint and restart mechanisms.

1 Introduction

A collection of workstations can be the computational equivalent of a supercomputer. Similarly, a collection of supercomputers can provide an even more powerful computing resource than any single machine. These ideas are not new; parallel computing has long been an active area of research. The fact that networks of computers are commonly being used in this fashion is new. Software tools like PVM [1, 13, 14], P4 [5], Linda [6], Isis [2], Express [12], and MPI [16] allow a programmer to treat a heterogeneous network of computers as a parallel machine. These tools allow the programmer to partition a program into pieces which may then execute in parallel, occasionally synchronizing and exchanging data. Heterogeneity is supported through data conversion from one machine's format into another. These tools are useful, but there are further issues to be addressed. Namely, load balancing and fault tolerance mechanisms must be developed that will work well in a heterogeneous multi-user environment.

There are a wide variety of issues that a parallel programmer must deal with. When using most conventional parallel programming methods, one needs to partition the program into parallel tasks and manually distribute the data among those parallel tasks — a difficult procedure in itself. To further complicate matters, in most cases the target network of machines is composed of multi-user computers connected by shared networks. Not only do the capacities of the machines differ because of heterogeneity but their usable capacities also vary from moment to moment according to the load imposed upon them by multiple users. Heterogeneity is also evident in the underlying network. For instance, typical bandwidths in local area networks vary from 10 Mbit Ethernet to 800 Mbit HiPPI. Message latency can also vary greatly, particularly as ATM-based LANs [30] become commonplace. System failure is yet another consideration. If an application is using a large number of machines to execute for a long period of time, failures during program execution are likely. Processor heterogeneity complicates support for fault tolerance. The Distributed object migration environment (Dome) presented here addresses these parallel programming issues for heterogeneous multi-user distributed environments.

Dome provides a library of distributed objects for parallel programming that perform dynamic load balancing and support fault tolerance. Dome programmers, with modest effort, can write parallel programs that are automatically distributed over a heterogeneous network, dynamically load balanced as the program runs, and able to survive compute node and network failures. Thus, we provide both the objects and the tools needed to make it simple to write efficient distributed programs.

This paper provides the motivation for and an overview of Dome, including a preliminary performance evaluation of dynamic load balancing for vectors. We show that Dome programs are shorter and easier to write than the equivalent programs written with message passing primitives. The performance overhead of Dome is characterized, and it is shown that this overhead can be recouped by dynamic load balancing in imbalanced systems. Finally, we show that a parallel program can be made failure resilient through Dome's architecture independent checkpoint and restart mechanisms.

2 Related Work

Dome shares attributes with many other research projects. pC++ [3, 20] extends C++ to a parallel programming language. High Performance Fortran [18] is an emerging standard for writing distributed memory parallel Fortran programs. While language based mechanisms for expressing parallelism and data mapping in distributed memory machines are important, we are most interested in using existing languages and exploring object oriented mechanisms for parallel and distributed computing. Dome is written in C++. The flexibility of this language makes it easy to add parallel objects and operators to the language, giving us the ability to prototype ideas rapidly. Building these mechanisms into a compiler would be a much more difficult task. However, knowledge gained in developing Dome can be used in compilers that target heterogeneous networks.

LaPack++ [7] is an object oriented interface to the LaPack routines for parallel linear algebra. Like LaPack++, Dome provides a library of parallel objects. However, Dome does not focus on linear algebra but on objects which are of general use for many types of parallel programming. As a complete distributed programming system, Dome also provides features, such as dynamic load balancing and fault tolerance, which are not addressed by packages like LaPack++.

2.1 Related Load Balancing Work

In general, load balancing consists of effectively matching task requirements to the resources of a distributed computing system. There has been a considerable amount of theoretical work on the assignment problem for parallel and distributed computing. Most of this work addresses the problems of mapping tasks to processors, given a set of tasks whose requirements are known *a priori* and a compute system whose resources are also well known [4]. Most distributed multi-user systems have unpredictable loads, making these approaches impractical for general use.

Load balancing research in operating systems focuses on the similar mapping problems but where little is known about the tasks or the target system's capacities. Thus, heuristics play a large role. For instance, Eager et al. [10] compare heuristics for task placement and migration under various system loads. It is generally agreed upon that simple heuristics are best when scheduling independent tasks in multi-user distributed system [9]. In this case no assumptions are made about inter-task relationships. For parallel computing, inter-task relationships are very important when making load balancing decisions. As Wikstrom et al. point out, it is difficult to make load balancing of parallel programs pay off [31]. This reiterates Eager's thesis that simple strategies win and extends that idea to load balancing of parallel algorithms.

For parallel programs the source of load imbalances can be both internal and external. Internal imbalances occur because the work distribution among the parallel tasks changes as the program executes. Iterative algorithms are a good example of this phenomenon [25]. External load imbalances are the result of sharing CPU and network resources. Dome uses simple load balancing strategies to address both internal and external load imbalance. This work differs from operating systems approaches to load balancing in that the tasks have intricate intercommunication dependencies and tend to be long running. It also differs from most parallel computing load balancing techniques in that external system load is a major consideration.

2.2 Related Checkpointing Work

Most checkpointing libraries for distributed systems focus on checkpointing in a homogeneous environment, using system-specific techniques to efficiently capture consistent memory images from each process. Among the recent ones are Li, Naughton, and Plank's [22, 23, 26], which is designed to minimize the checkpointing overhead on multicomputers; Silva and Silva's [28], which takes into account the latency between failure occurrence and detection; and Leon, Fisher, and Steenkiste's [21], which is designed specifically for programs written in PVM. In most work on checkpointing for distributed systems, the primary focus is on attempting to minimize the cost of each checkpoint. In [11], however, Elnozahy, Johnson, and Zwaenepoel have suggested that checkpointing is generally an inexpensive operation. Thus, performance of the checkpointing mechanism is not our focus. Rather, maximizing user-transparency and architecture independence is of much greater concern. Dome uses an object-oriented paradigm and an implementation in non-machine-dependent C++ code to achieve these objectives even in the face of heterogeneity.

An architecture independent package has also been developed by Silva, Veer, and Silva [29], who have created a purely library based system where the user is responsible for inserting calls to specify the data to be saved and perform the checkpoints. Another system related to ours was developed by Hofmeister and Purtilo [19]. As in Dome, they use a preprocessing mechanism for saving the state of distributed programs. While their main concern is dynamic program reconfiguration rather than checkpoint and restart, their preprocessing method is similar to the one we are using.

Finally, Duda [8] has analyzed the expected runtime of a distributed program with checkpointing, assuming failures are a Poisson process. Since the main purpose of checkpointing is to reduce the total expected runtime in the presence of failures, we have tried to relate our performance observations to his analysis, rather than merely reporting the overhead of the individual checkpoints.

3 Dome Architecture

Dome was designed to provide application programmers a simple and intuitive interface for parallel programming. It is implemented as a library of C++ classes which uses PVM for its process control and communication. When an object of one of these classes is instantiated, it is automatically partitioned and apportioned within the distributed environment. Therefore, computations using this object are performed in parallel across the nodes of the current PVM virtual machine.

The Dome library uses operator overloading to allow the application programmer simple manipulation of Dome objects and to hide the details of parallelism. In designing the interfaces to Dome objects, care has been taken to provide a simple and intuitive programming paradigm for the application programmer.

When a program using the Dome library is run, a Dome environment is created. Initially, the Dome environment controls the creation of the multiple processes which constitute the distributed program. Dome uses a single program multiple data (SPMD) model to perform the parallelization of the program. In the SPMD model the user program is replicated in the virtual machine, and each copy of the program, executing in parallel, performs its computations on a subset of the data in each Dome object. The number of copies of the program that are spawned by the Dome environment defaults to one per node in the virtual machine but can be controlled by the user with a parameter passed to the Dome environment on initialization. The Dome environment then keeps track of these Dome processes and the existence and distribution of all Dome variables in the program. Global checkpointing and load balancing information is also maintained within the Dome environment and can be controlled by the user through input parameters.

A Dome class generally represents a large collection of similar and related data elements, a vector for example. When a Dome object is created, the elements of that object are partitioned and distributed among the processes of the distributed program. Dome offers a few different possibilities for the method of data partitioning. The **whole** directive indicates that all elements of the given object are replicated at all of the distributed processes. **Block** distribution indicates that the data elements of the Dome object are to be evenly divided among the processes in contiguous blocks. Finally, **dynamic** indicates that the initial distribution is the same as in the **block** distribution, but the data is reapportioned among the processors periodically based on dynamic load balancing performed at given intervals. This dynamic redistribution of data is discussed further in Section 5. The user may indicate the particular method for partitioning a given Dome object when that object is declared. The default partitioning is **dynamic**. Figure 1 illustrates the data distribution of a program using Dome over a virtual machine consisting of four nodes.

It is useful to discuss the concept of a Dome operation. A Dome operation

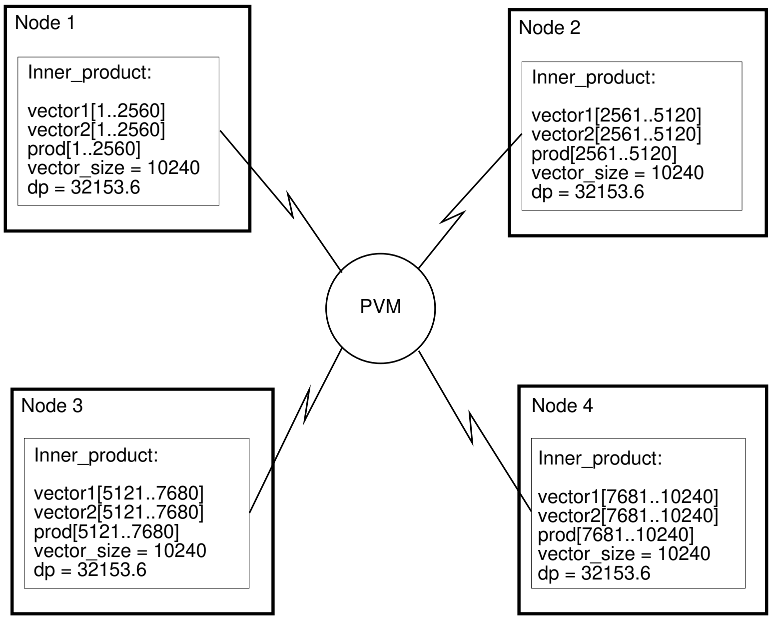

Figure 1: SPMD model of the Dome program *Inner_product* executing on four nodes in a PVM virtual machine. The contents of the distributed vectors *vector1*, *vector2*, and *prod* have been divided evenly among the processes in the distributed program. The scalar variables, *vector_size* and *dp*, are replicated at each process. The code for this program is given in Figure 2.

is a function performed on one or more Dome objects. A single Dome operation usually causes a function to be applied in parallel to all of the elements of that object. Some Dome operations involve a synchronization of the processes of the distributed program, but most do not. Consider the term *vector1* + *vector2* in which *vector1* and *vector2* have been declared as Dome distributed vectors (`dVectors`) of the same dimension. The + operation on vectors causes an elementwise addition of the contents of the two distributed vectors to be performed in parallel. No communication is necessary between the processes of the distributed program to perform this operation. The concept of a Dome operation is important for load balancing because the intervals at which a load balancing phase is performed are determined by a given number of completed Dome operations. This is discussed fully in Section 5.

4 Dome Programming

To illustrate the simplicity of programming with Dome objects, consider the example program in Figure 2. This program performs a standard inner product operation on a pair of vectors.

The program includes header files for two of the Dome classes, distributed vectors and distributed scalars (the `dVector` and `dScalar` classes). The entry point to `main` accepts the standard `argc` and `argv` parameters. These arguments are passed to the `dome_init` routine because they can contain user parameters to the Dome environment such as the number of copies of this program to run in parallel, the method and frequency of load balancing, and checkpointing information. It also allows the Dome environment to spawn remote copies of the program with the same argument list that the user specified originally.

Next, several Dome variables are declared. Two `dScalar` objects, `vector_size` and `dp`, are declared and initialized. The `dScalar` class replicates the variables at all of the processes of this distributed program. `dScalar` variables differ from normal C++ variables solely in that when the `dScalar` variable is declared it is registered with the Dome environment. It can then be included in a Dome checkpoint whereas normal C++ variables will not be checkpointed. Three `dVector` objects, `vector1`, `vector2`, and `prod`, are also declared in the example program. Each vector is composed of 10240 double precision floating point numbers. By default the vectors will be distributed among the processes using the **dynamic** distribution. Each of the processes in the distributed program will initially be assigned n/p elements of each vector, where n is the total number of elements in the vector and p is the total number of processes.

The example program next assigns the values 1.0 and 3.14 to each of the elements of the vectors `vector1` and `vector2` respectively. These assignments are done in parallel in the distributed program, each process making the scalar assignment to the vector elements which have been assigned to that processor. The statement which follows, `prod = vector1 * vector2`, performs two Dome operations, the vector multiplication and the vector assignment. Each of these operations, like the scalar assignment just described, is performed in parallel on the elements of the distributed vectors assigned to each processor. An element-wise multiplication of the vectors `vector1` and `vector2` is performed and then the values are assigned to the distributed vector `prod`.

The elements of the entire distributed vector `prod` must now be added to complete the standard inner product calculation. The `gsum` method is used to perform this addition. This method causes each processor in the distributed program to calculate a local sum of the elements assigned to that processor. It then forces a synchronization of all of the processes to exchange the local sums. These are added to reach the final result which is assigned to the scalar value `dp`.

Automatic load balancing and architecture independent checkpointing can be performed on the distributed data objects declared. Although not necessary or particularly useful in a small program like the example given, these features offer powerful advantages to complex distributed programs, as will be discussed in Sections 5 and 6 respectively.

This simple program demonstrates the power of Dome. Distributed programs are easy to write using Dome objects. Most of the details of program parallelism, load balancing, and architecture independent checkpointing are hidden from the programmer. An equivalent program to perform a distributed standard inner product operation using PVM primitives would be much more complicated to write and would be several pages in length. If similar load balancing and checkpointing were added to the PVM program it would be even longer.

As seen in the example program in Figure 2 the Dome library uses templated C++ classes. This allows for a wide range of user extensibility and customization. In the example program scalar values of integers and doubles as well as vectors of doubles were declared, but user defined classes can also be used in the templated Dome classes.

5 Load Balancing

The object oriented architecture of Dome hides data placement and communication from the programmer. This makes it possible for Dome to alter data mappings and communication patterns dynamically during program execution in response to changes in the execution environment. This section addresses load balancing based on observed processor speed and briefly describes an approach to load balancing based on observed communication performance.

```
//  Simple Dome program to compute the standard inner product of two vectors.

#include <stdlib.h>                             // C++ includes
#include <stream.h>

#include "dScalar.h"                            // Dome includes
#include "dVector.h"

int main(int argc, char *argv[])
{
  dome_init(argc, argv);                        // Initialize the Dome environment.

  dScalar<int> vector_size = 10240;             // These scalars will be replicated
  dScalar<double> dp = 0.0;                     // at all processes.

  dVector<double> vector1(vector_size);         // These vectors will be
  dVector<double> vector2(vector_size);         // distributed across
  dVector<double> prod(vector_size);            // all processes.

  vector1 = 1.0;   vector2 = 3.14;              // Assign values to the vectors.

  prod = vector1 * vector2;                     // Compute product using overloaded
                                                // vector product operator.

  dp = prod.gsum();                             // Compute sum of all elements of prod.

  cout << "The dot product is " << dp << '\n'; // Print the result.
}
```

Figure 2: Program to find the standard inner product of two vectors written using Dome.

5.1 Processor Based Load Balancing

Load balancing involves mapping work to processors such that all the work is completed in the shortest amount of time. In a multi-user system, processor loads can change frequently; therefore, prediction of actual execution speeds is an integral part of load balancing. One could utilize any number of metrics when attempting to capture and predict the performance of a particular processor: processor speed, amount of available memory, length of the current run queue, percentage of idle time in the recent past, number of recent network interrupts, and others. Indeed, various authors have utilized some of these parameters to characterize the performance of processors, deriving load indices that are used to make decisions on scheduling and the distribution of work among the nodes of a network. Rather than using metrics of this kind to predict the performance of a Dome program, we simply use the actual rate at which the processors have been executing the Dome program. The recent past performance in executing a Dome program is assumed to indicate near term future performance for that same program.

When a program begins execution in the parallel virtual machine, Dome makes no assumptions about the current loads at each node. All dynamically distributed Dome objects are initially distributed evenly among all participating processors. Dome operations are instrumented with timers, which measure the amount of time each processor spends doing computation. During the load balancing phase the Dome program synchronizes, and these times are compared. The load balancing is performed by remapping data based on the time taken by each task during the last computational phase. The synchronization is straightforward given the SPMD structure of Dome applications. Thus, Dome load balancing does not require a complicated interrupt scheme. Presently, an initial load balancing phase is performed after the first few Dome operations of program execution. This early load balancing phase captures the initial load conditions of the participating processors. Our experiments have shown that the absence of this initial phase can increase total running time by a factor from 1.25 to 1.50 on imbalanced systems. This happens because the fastest machines have to wait for the first synchronization point, remaining idle for a long time. The experiments also have shown that the very short initial period of timing is a good predictor of the load distribution in the machines. After this initial redistribution of work, the load balance phases are triggered upon the completion of a predetermined number of Dome operations. This count is fixed for the duration of the program execution, and it can be set by the user. Future Dome implementations may vary this parameter based on the results of a load balancing phase; if several load balancing phases have shown the workload to be evenly balanced then it is reasonable to increase the time between load balancing phases.

For load balancing purposes Dome considers that the participating machines are interconnected in a virtual topology, either linear or ring. The reason to use these simple topologies is twofold: first, it greatly simplifies the management of the objects that are distributed among the machines; second, by allowing data movement to occur only between neighbors, it avoids arbitrary data traffic that may overload the network.

Another major issue is whether the load balancing decision should be done locally or globally. In a global remapping the final data layout will exactly reflect the most recent performance measurements. In this scheme all tasks send their most recent computational times to a designated master task that calculates the ideal data distribution. Then the master broadcasts the new distribution mapping, and the tasks begin to exchange data. Note that the global communication involves only control information; the real data movement is done only between neighbors. Although the control information exchanged is small, this remapping may be costly because it may result in a large amount of data movement. Also, a central point of control prevents this scheme to scale well with a large number of machines. Another option is to have processors simply exchange control data locally, i.e., with their neighbors in a ring. This local load balancing option will not result in a globally optimal data mapping after each load balancing phase, but it is scalable to a large number of processors and requires less data remapping. Dome currently implements both of these methods. There is, in fact, an entire spectrum of choices for load balancing between the two extremes described here, and future versions of Dome will include more points along this continuum. Actual measurements made on a virtual machine of less than ten processors show that global load balancing performs better than local. Therefore, scalability is not an issue for networks of this size.

We have built a tool which shows the changes in the distribution of dVectors over time. Figure 3 reproduces this tool's display. The upper window in Figure 3 shows the dVector mapping while the lower window shows the loads of the processors over the same time period. Notice that vectors can be load balanced "off the ends" where data at the ends of the vector can move around to the process handling the other end of the vector. The loads are indicated by the amount of time spent computing on dVectors. Thus, it is an indication of how fast the processors perform dVector computations.

5.2 Initial Timing Results

In order to evaluate the Dome approach to load balancing we have written a matrix multiply program using `dVectors` *(mmdome)*. This implementation was compared with three other versions of matrix multiply: a sequential version *(mmseq)*, a version written in PVM that uses the same algorithm of the Dome program *(mmpvm)*, and another version using PVM that takes advantage of its lower level primitives to produce better register usage by the compiler and fewer cache misses *(mmpvmopt)*. Since Dome is implemented using PVM, this comparison allows us to measure the overheads involved in the Dome implemen-

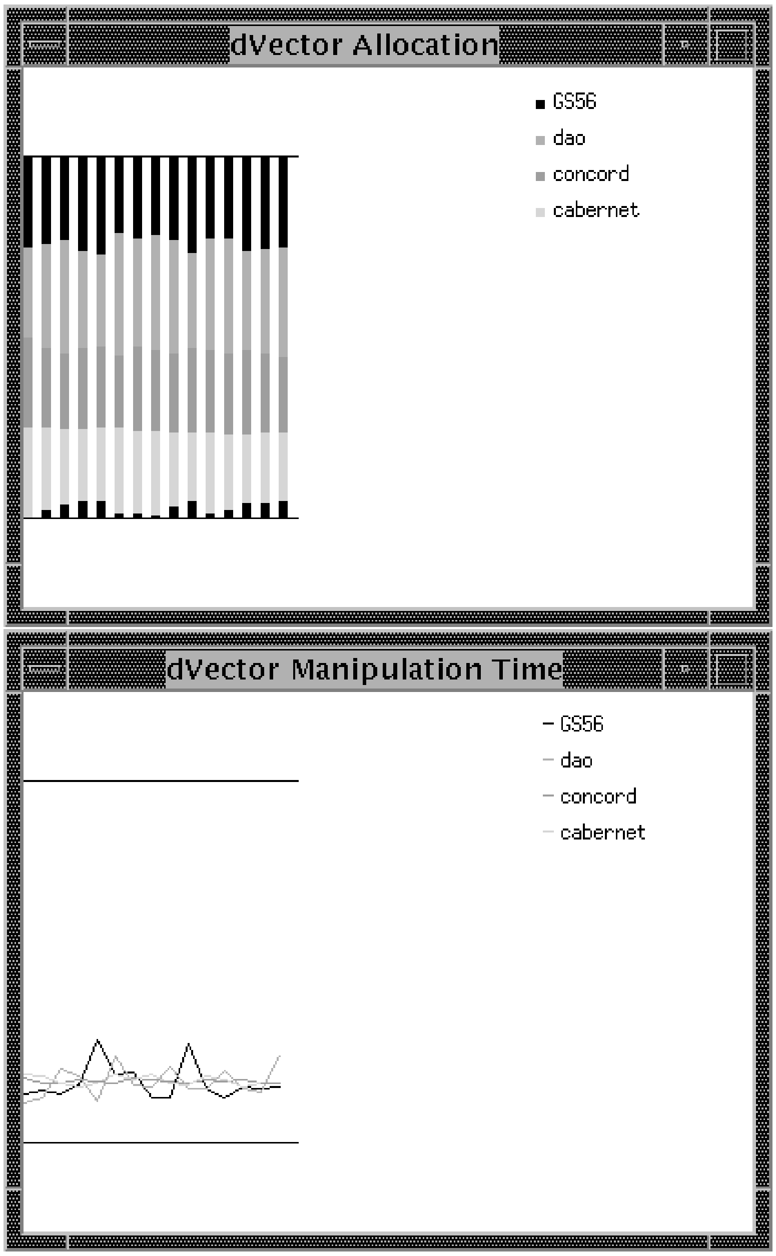

Figure 3: dVector mappings and machine loads over time.

tation, as well as the overhead for load balancing. It is also worth mentioning the difference in complexity and size of the source code between the Dome implementation and the two PVM implementations. While the Dome program has roughly the same size as the sequential version, the PVM implementations are almost double that size and have to deal explicitly with data distribution, gathering and synchronization.

All results shown here were obtained for 240 matrix multiplies, $C = A \mathtt{x} B$, where A and B are of size 786432 double precision elements. Three experiments were performed.

1. Unloaded, balanced system: using 6 DEC Alpha workstations interconnected by an Ethernet at the School of Computer Science at CMU. The experiment had exclusive use of the machines.

2. Imbalanced, stable system: same environment as before, but one of the machines was artificially loaded, while the others remained unloaded for the duration of the experiment.

3. Production system (loaded, unstable): using 6 DEC Alpha workstations of the Pittsburgh Supercomputer Center SuperCluster (also DEC Alpha workstations) interconnected by a DEC Gigaswitch (FDDI point-to-point connection between workstations). These experiments were run under normal production conditions at various hours of the day and different days of the week. A total of 45 runs for each experimental data point was performed.

The Dome program *(mmdome)* was executed without any load balancing *(no_lb)*, and with 1, 2 and 3 load balancing phases (*lb1, lb2, lb3*, respectively). Other experiments were also performed for a different number of machines (from 4 to 8), various sizes of the matrices and various numbers of load balancing phases (i.e., different intervals between load balancing phases). The results presented here for the case of six machines are representative of the overall behavior of the programs.

Figure 4 shows a comparison of the times obtained for the imbalanced, stable experiment (experiment 2). We can see that the load balanced cases perform well, being 35% faster than the *mmpvm* version and 13% faster than the optimized PVM version *(mmpvmopt)*. The figure also shows that the overhead of Dome without load balancing is quite high *(mmdome-no_lb)* in this case. Finally, as expected, there is not much difference between doing one and more than one load balancing phases, since the system is stable.

Figure 5 shows the results in the SuperCluster at PSC under production conditions. In this case, Dome with load balancing is only 10% slower than the hand-optimized PVM implementation. On the other hand, Dome with load balancing presents a significant gain of 44% over the equivalent PVM implementation *(mmpvm)*.

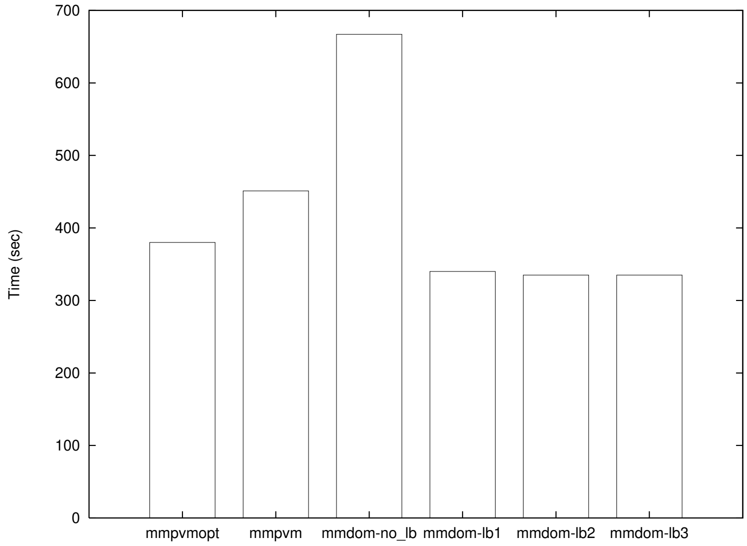

Figure 4: Times for imbalanced, stable system

The table shown in Figure 6 summarizes the results of the three sets of experiments. The numbers for the first experiment indicate that the overhead of unnecessary load balancing is around 6%, as is the case in an unloaded, balanced system. Future versions of Dome will recognize the stability of the system and will not perform load balancing under these conditions, virtually eliminating this overhead. The other two experiments show the advantage of load balancing. In the second experiment, *mmdome* outperforms even the optimized implementation of the PVM version *(mmpvmopt)*.

5.3 Network Load Balancing

In addition to balancing the workload based on the characteristics of the processors, it is equally important to consider the characteristics and topology of the interconnection network. Although this can have a large impact on overall performance, it is typically overlooked, in part because it is difficult to measure network characteristics in a system independent manner.

Dome attempts to deal with the issue of network load balancing using a three stage process. The first step is to characterize the network in a platform independent manner. The second step is to use this information in assigning Dome objects to processors in such a way that the remapping overhead is minimized. The third step is to choose specific communication patterns for collective communication operations which make efficient use of the interconnection network.

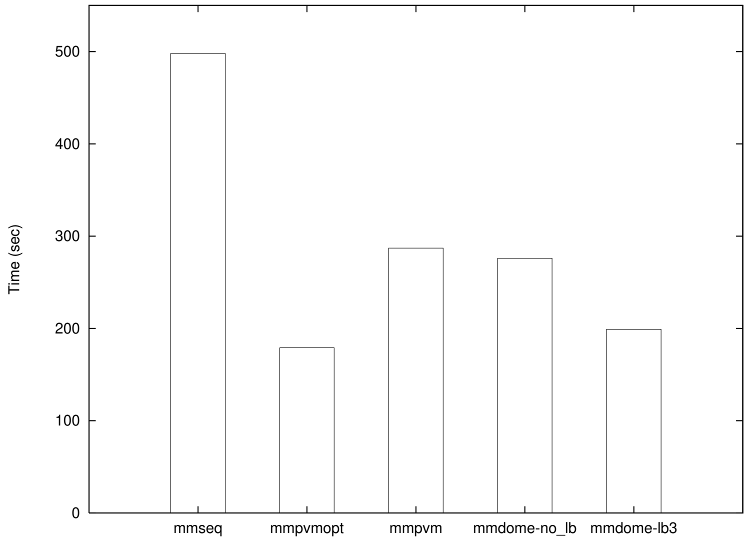

Figure 5: Times for production system (loaded, unstable)

Characterization of the network is done by measuring the round-trip time for message exchanges between processors. All exchanges are done with the recipient having a blocking receive posted for the message prior to its arrival, so that the resulting times include as little overhead as possible, given the message passing interface. Messages are exchanged in both an application-induced congested pattern, where as many exchanges as possible are occurring simultaneously, and an uncongested pattern, with one exchange occurring at a time. The data from these measurements is used to approximate the bandwidth of the network and whether it is shared or switched. This information is then used to divide the network into clusters of machines which are mutually closer to each other than any other machine on the network. Generally, these clusters

Matrix Multiply — Mean time in seconds					
Experiment	mmseq	mmpvmopt	mmpvm	mmdome-nolb	mmdome-lb3
1	655	109	171	174	184
2	1964	380	451	671	335
3	498	179	287	276	199

Figure 6: Comparison of results for the 3 experiments

correspond to machines on a single Ethernet strand or on a single switch.

A number of simplifying assumptions are made which allow the network characterization problem to be solved off line. It is assumed that the user runs the characterization program on all accessible machines, so that the processors used in computation are always a subset of those machines. The ambient network load, i.e., the network traffic not caused by Dome, is assumed to be relatively similar on all portions of the network. Therefore, while it may alter the available bandwidths, it will not change the relative bandwidths of the network components. While this second assumption may not be accurate in practice, it has proven to be a reliable heuristic.

Since load balancing communication in Dome is routed using a ring, the optimal assignment of tasks to nodes is ordered along the minimum communication time circuit, which corresponds to the solution of the travelling salesperson problem. Since this problem is NP-complete, an exact solution is not feasible for large numbers of processors, but observation leads one to the simple heuristic that the nodes in the closely connected clusters should be linked together, followed by those in network-wise nearby clusters.

Due to its SPMD model, the communication used in Dome is collective; a typical program requires several operations, including reduction, all-to-all broadcast, one-to-all broadcast, and scatter/gather. The information obtained through network characterization is used to choose the optimal local and global communication patterns. To allow for general use of this technique, a collective communication library is being developed. This library will be modeled after the MPI collective communication specification and will be used in the implementation of Dome classes.

6 Checkpointing

While performing large computations on a network of workstations offers many advantages, it also introduces some new problems, including the possibility of failure on some subset of the nodes. As the number of workstations in a cluster increases, the chance that one of them will fail during a particular computation increases exponentially. For example, if we have a workstation with a mean time between failures of 16 days, a one day computation may have a 94% chance of a successful run, but on a cluster of ten such machines, there is only a 54% ($.94^{10}$) chance that the computation will be complete before one of them fails. Thus, it is vital that some kind of fault tolerance mechanism be incorporated into any system designed for extended execution on a workstation cluster.

6.1 Abstraction Levels for Checkpointing

There are several levels of programming abstraction at which one could implement a fault tolerance package in Dome. At the highest level, we have provided

```
main(argc,argv)
{
   // dome_init decides if we are starting the program from
   // scratch or restarting from a saved checkpoint, based on argv.
   dome_init(argc,argv);

   // declare variables
   dScalar<int> integer_variable;
   dScalar<float> float_variable;
   dVector<int> vector_of_ints;
   dVector<float> vector_of_floats;
   // etc.

   // initialization code-- user must skip if in restart mode.
   if (!dome_env.is_restarting())
        execute_user's_initialization_code(...);

   // main loop
   while (!loop_done(...)) {  // loop_done is a function of dome vars
        do_computation(...);
        dome_env.checkpoint();
        }
}
```

Figure 7: Program structure needed for high level checkpointing without preprocessing.

a set of C++ methods which can be called to checkpoint the program's data structures. The user is responsible for writing a program with a simple enough structure, such as in Figure 7, that the program counter and stack do not need to be saved. Another method we have developed is to do the checkpointing with C++ code, but to have a preprocessor insert calls to save and restore the stack and program counter, saving the user some work. Figure 8 shows a sample program fragment before and after preprocessing, illustrating how the stack and program counter can be saved and restored in high level code. The expansion in code size will in general be small and linear. Finally, we could use a general, truly transparent low level checkpointing package such as the one described in [21], so the user has little work to do. The system would save a memory image periodically upon interrupt, from which it could restore the state later. In this case, determining a consistent state is a major issue since messages may be in transit, while in the higher level methods the knowledge of the program structure makes this problem much simpler. The advantages and disadvantages of each level of checkpointing are described in Figure 9.

While high level fault tolerance tends to require more work from the users, it is an important feature in Dome, since one of our major goals is for Dome to be easily portable to any system that supports PVM and C++. Furthermore, it is vital that checkpoints created on one architecture be usable on others. Thus, we have concentrated on developing usable high level checkpointing features for Dome.

6.2 Checkpointing Results

We have completed an implementation of high level checkpointing. It has been tested on *md*, a molecular dynamics application we have written based on a CM-Fortran program developed at the Pittsburgh Supercomputing Center. Our timings show that even in the case of frequent failures, our checkpointing overhead is low enough to provide a good expected run time for this application, assuming a Poisson failure model. This result is plotted in Figure 10, which shows the expected runtime we calculate for various mean times between failures based on the checkpointing times we observed, in an approximately 26 minute run, and the expected runtime formula calculated in [8]. It is interesting to note that if we put in so many checkpoints that a failure every 3 minutes will not even double our expected runtime, rather than incurring the thousands-of-times expected cost without checkpointing, we only suffer a 3% cost to our runtime in the failure free case.

Of course, the next question one should ask is what are realistic values for the mean time between failures. In a run as small as our experiment, we do not expect a failure at all, but it should be noted that our results would scale up, since we found the cost per checkpoint to be a constant for a given problem size. Actually, for *md*, the computation complexity grows faster than the data size, so the checkpoint cost compared to the total run time will go down. Therefore,

```
f() {
  dScalar<int> i;
  do_f_stuff;
  g(i);
  next_statement;
  ...
}


g(dScalar<int> i) {
  do_g_stuff_1;
  dome_env.checkpoint();
  do_g_stuff_2;
}
```

```
  f() {
    dScalar<int> i;

*   if (dome_env.is_restarting()) {
*        next_call=dome_env.get_next_call();
*        if (next_call == ``g1'') goto g1;
*        ...
*    }

    do_f_stuff;

*   dome_env.push(``g1'');
* g1:
    g(i);
*   dome_env.pop();

    next_statement;
    ...
  }

   g(dScalar<int>& i) {
*   if (dome_env.is_restarting())
*       goto restart_done;

    do_g_stuff_1;
    dome_env.checkpoint();

* restart_done:
    do_g_stuff_2;
  }
```

Figure 8: Program fragment before and after preprocessing. Lines added by the preprocessor are marked with a '*'.

Level	Transparency	Portability
High	very weak	very portable code and checkpoints
High with Preprocessing	almost transparent (but interferes with debugging)	very portable code and checkpoints
Low	completely transparent	code and check-points not portable

Figure 9: Comparison of levels of checkpointing.

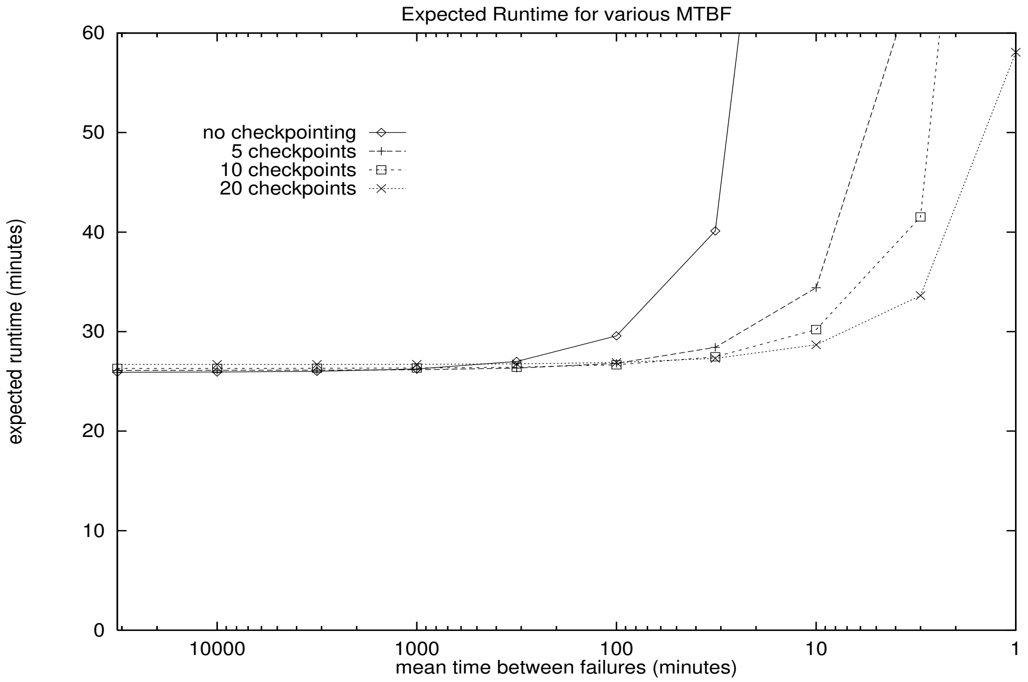

Figure 10: Expected runtime for *md* vs. mean time between failures.

we would expect the graph in Figure 10 to remain valid for longer runs, so the units could be hours or days rather than minutes. In fact, the figure would overestimate the cost of checkpointing in runs with larger data sets.

In an experiment on the Internet measuring typical failure rates, Long, Carroll, and Park [24] found that, depending on the system, the mean time between failures tended to be between 12 and 20 days. If we take 16 days as a rough estimate, a cluster of five machines is likely to have a failure an average of every 3.2 days. Looking at the graph as if the units were days rather than minutes, realistic for some large simulations, and estimating for a cluster of five machines, we can see that a properly chosen checkpoint interval can result in less than a 25% increase in expected runtime over its ideal value. Without checkpointing such a program would take several millenia to run to completion.

We have demonstrated the portability of our checkpointing code by running *md* with checkpointing on both DEC Alpha and on SGI workstations. In addition, the portability of the checkpoints themselves has been demonstrated by restarting *md* on the Alphas from checkpoints created on the SGIs. Our experiments will continue to demonstrate the portability of our system on additional architectures, though we do not expect this to be much of an issue since our checkpointing methods use only high level code and generate checkpoint files in an architecture independent format.

We have also completed a preliminary implementation of the preprocessor and demonstrated that it works on Dome programs. We are currently in the process of developing more complex applications for use in performance tests. For more detailed information on Dome's checkpointing methods, see [27].

7 Future Work

The Dome system is undergoing very active development. We are adding new classes and developing more Dome applications. We are collaborating with both computer science and computational science researchers to develop production quality Dome applications, allowing us to demonstrate Dome in several domains. This is an important step to showing that Dome is general enough to express a wide array of parallel algorithms. We also plan to add parallel programming structures to Dome, such as general task parallelism, futures [17], and pipelining.

With respect to load balancing, there is still much work to do. Adaptation of the load balancing frequency based on runtime characteristics may be fruitful. Also, a further exploration of the spectrum of choices between local and global load balancing is important. This work will mesh well with the automatic network partitioning described in Section 5.3. It is also possible that runtime metrics of communication performance can be used to develop a virtual topology for load balancing and collective communication operations.

There are several ways in which we plan to make the fault tolerance preprocessor more powerful. We will give it the ability to automatically transform

C++ variables that will be in scope at the time of a checkpoint into Dome variables, so they will be saved without any special effort from the user. In addition, we plan to experiment with techniques for inserting the checkpoint calls automatically, again saving the user additional effort. These and other improvements will help to make the preprocessing method nearly as user transparent as low level checkpointing, while maintaining architecture independence.

Finally, we have implemented a prototype parallel I/O system for use in Dome programs. This prototype has been demonstrated in the context of conventional files systems and with the Scotch parallel file system [15]. There are many issues related to parallel file systems and parallel program I/O that can be addressed by Dome.

8 Concluding Remarks

This paper shows that Dome provides mechanisms that effectively address three critical issues of parallel programming in a distributed computing system: ease of programming, dynamic load balancing, and architecture independent checkpointing. Because Dome's mechanisms are at the language level they allow the system to be very portable. In fact, to date Dome has been ported to seven platforms: DEC Alpha OSF/1, HP HPUX, Intel Paragon, Sparc SunOS, SGI Irix, DEC Ultrix, and Cray C90. The code changes among these ports is minimal, on the order of several lines of code, demonstrating that the system's goals can be achieved while portability is maintained.

Preliminary measurements show that the runtime overhead of Dome programs is reasonable. Furthermore, this overhead can be recouped by load balancing in an imbalanced system. Dome's approach to checkpointing allows programs to achieve good runtimes even when failures are present. Dome's combination of solutions make it uniquely suited for parallel programming in a heterogeneous multi-user distributed environment.

9 Acknowledgements

We would like to acknowledge William Young, Brad Keister, and Stan Simon for their help on the Dome project. William Young has participated in many useful discussions and provided the CM-Fortran molecular dynamics code on which our Dome program is based. Brad Keister has the distinction of being our first Dome user besides ourselves. Finally, Stan Simon contributed to coding of several Dome classes.

References

[1] A. Beguelin, J. Dongarra, A. Geist, and V. Sunderam. Visualization and

debugging in a heterogeneous environment. *IEEE Computer*, 26(6):88–95, June 1993.

[2] Kenneth Birman and Keith Marzullo. Isis and the META project. *Sun Technology*, pages 90–104, Summer 1989.

[3] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Supercomputing 93*, 1993.

[4] Sahid H. Bokhari. *Assignment Problems in Parallel and Distributed Computing*. Kluwer Academic Publishers, 1987.

[5] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[6] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, pages 323–357, September 1989.

[7] J. Dongarra, R. Pozo, and D. Walker. An object oriented design for high performance linear algebra on distributed memory architectures. In *Proc. OON-SKI Object Oriented Numerics Conf.*, pages 257–264, Sun River, Oregon, April 1993.

[8] Andrzej Duda. The effects of checkpointing on program execution time. *Information Processing Letters*, 16:221–229, June 1983.

[9] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5), May 1986.

[10] D. Eager, E. Lazowska, and J. Zahorjan. A comparison of receiver initiated and sender initiated dynamic load sharing. *Performance Evaluation*, 6(1), April 1986.

[11] Elmootazbella Elnozahy, David Johnson, and Willy Zwaeneopoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47. IEEE Computer Society Press, 1992.

[12] J. Flower, A. Kolawa, and S. Bharadwaj. The express way to distributed processing. *Supercomputing Review*, pages 54–55, May 1991.

[13] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[14] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[15] G. Gibson, D. Stodolsky, F. Chang, W. Courtright II, C. Demetriou, E. Ginting, M. Holland, Q. Ma, L. Neal, R. H. Patterson, J. Su, R. Youssef, and J. Zelenka. The scotch parallel storage system. In *Compcon'95*, pages 403–410, San Francisco, March 1995.

[16] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI.* MIT Press, 1994.

[17] R. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[18] High Performance Fortran Forum. *High Performance Fortran Language Specification*, January 1993. Version 1.0 DRAFT.

[19] Christine Hofmeister and James Purtilo. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. Technical Report UMIACS-TR-92-120, University of Maryland, November 1992.

[20] Jenq Kuen Lee and Dennis Gannon. Object oriented parallel programming experiments and results. In *Supercomputing 91*, pages 273–282, 1991.

[21] Juan Leon, Allan Fisher, and Peter Steenkiste. Fail-safe pvm: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, February 1993.

[22] Kai Li, Jeffrey Naughton, and James Plank. Real-time, concurrent checkpoint for parallel programs. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88. Association for Computing Machinery, 1990.

[23] Kai Li, Jeffrey Naughton, and James Plank. Checkpointing multicomputer applications. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 2–11. IEEE Computer Society Press, 1991.

[24] D.D.E. Long, J.L. Carroll, and C.J. Park. A study of the reliability of internet sites. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 177–186. IEEE Computer Society Press, 1991.

[25] D. Nicol and J. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Transactions on Computers*, 37(9), September 1988.

[26] James Plank and Kai Li. ickp: A consistent checkpointer for multicomputers. *IEEE Parallel and Distributed Technology*, pages 62–66, Summer 1994.

[27] Erik Seligman and Adam Beguelin. High level fault tolerance in distributed programs. Technical Report CMU-CS-94-223, Carnegie Mellon University, December 1994.

[28] Luis Silva and João Silva. Global checkpointing for distributed programs. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 155–162. IEEE Computer Society Press, 1992.

[29] Luis Silva, Bart Veer, and João Silva. Checkpointing SPMD applications on transputer networks. In *Proceedings of the Scalable High Performance Computing Conference*, pages 694–701, May 1994.

[30] R. Vetter. ATM Concepts, architectures, and protocols. *Communications of the ACM*, 38(2), February 1995.

[31] M. Wikstrom, G. Prabhu, and J. Gustafson. Myths of load balancing. In D. Evans, G. Joubert, and H. Liddell, editors, *Parallel Computing '91*, pages 531–49, London, September 1991.