Dome: Distributed object migration environment

Adam Beguelin Erik Seligman Michael Starkey

May 1994

CMU-CS-94-153

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Dome is an object based parallel programming environment for heterogeneous distributed networks of machines. This paper gives a brief overview of Dome. We show that Dome programs are easy to write. A description of the load balancing performed in Dome is presented along with performance measurements on a cluster of DEC Alpha workstations connected by a DEC Gigaswitch. A Dome program is compared with a sequential version and one written in PVM. We also present an overview of architecture independent checkpoint and restart in Dome.

This research was sponsored by the National Science Foundation and the Defense Advanced Research Projects Agency under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives.

Keywords: Heterogeneous parallel programming, load balancing, checkpoint and restart, Dome, PVM

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, 7], P4 [11], Linda [4], Isis [2], and Express [6] 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. (The tools generally support conversion from one machine's data format into another, thus providing support for heterogeneity.) We see these tools as useful but we strive for something higher level, something that can aid the programmer in parallel programming but also support load balancing and fault tolerance.

When using the previously mentioned tools, 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 multiuser computers connected by shared networks. This fact further burdens the programmer. Not only do the capacities of the machines differ because of heterogeneity but their capacities vary from moment to moment according to the load imposed upon them by multiple users. Failure is yet another consideration. If we are now using a number of machines to execute a long running program, the chances of a failure during program execution are increased, and therefore must be carefully considered.

Dome is our approach to the problems of parallel distributed computing in a heterogeneous networked environment. 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 failures.

Dome shares attributes with many other research projects. Gannon's pC++ [3, 9] is an attempt to extend C++ to a parallel programming language. High Performance Fortran (HPF) [8] is an emerging standard for writing distributed memory parallel Fortran programs. While we applaud efforts to develop language based mechanisms for expressing parallelism and data mapping in distributed memory machines, we are most interested in using existing languages and exploring object oriented mechanisms for parallel distributed computing. It is our hope that the knowledge gained in developing Dome can be used later in compilers that target heterogeneous networks.

LaPack++ [5] provides an object oriented interface to the LaPack routines for parallel linear algebra. Like LaPack++, Dome provides a library of parallel objects. Dome, however, is a more general paradigm, providing objects and features which are useful in a wide variety of parallel programming areas. We focus both on providing the objects themselves, and the tools, such as fault tolerance and load balancing, to make it easy and convenient to write efficient parallel programs.

This paper provides the motivation and overview of Dome including a preliminary performance evaluation of dynamic load balancing for vectors. We argue that programming in Dome is much easier than programming with low level primitives and that with dynamic load balancing good performance can be achieved in Dome.

2 Dome Programming

This section describes the Dome programming model. For discussion purposes we use a simple dot product program written using the Dome distributed vector class. Dome programs are written in C++ and the Dome objects make use of operator overloading to allow the programmer to easily express parallelism.

In order to achieve the goals of ease of programming, automatic load balancing, and architecture independent fault tolerance, Dome programs must have a certain structure. Dome programs are written using a single program multiple data (SPMD) style. At runtime the Dome program is replicated over a number of machines. Each of these replicated programs, which we call tasks, executes in parallel but on different sets of data. The distribution of the data and the extent upon which the programmer controls this distribution depends on the Dome object class the programmer uses. The default distribution of the Dome distributed vector class, dVector, is for the elements of the vector to be fragmented across the tasks in a block fashion. In this case each task contains approximately n/t elements where n is the total number of elements in a dVector and t is the number of tasks involved in the computation. Note that the programmer need not be concerned with the exact data mappings. Dome takes care of this automatically.

Let's examine the Dome dot product example in more detail (Figure 1). Keep in mind that copies of the same program are running in parallel, typically on all machines in the current virtual machine. The first Dome variable to be declared in the program is `tdome`; this is the Dome environment variable. It is used to keep track of Dome tasks and the layout of all Dome variables in a program. The program arguments are passed on as the parameters to the `tdome` constructor. This allows Dome to start multiple copies of the executable, in this case the dot product program. By default a Dome program will execute on all the processors in the virtual machine. This can be overridden by specifying a count to the Dome constructor indicating the number of tasks to be used.

Next, dVectors `v1`, `v2`, and `prod` are declared. The Dome dVector class supports templates, in this case the dVectors are made up of double precision

```
//  Dot Product
//  This is a simple example of a minimal Dome program.
//  It computes the dot product of two vectors.

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

// Dome includes
#include "dome.h"
#include "dVector.h"
#include "domeobj.h"

// The main program
int
main(int argc, char *argv[])
{
  const int count = 4; const int vector_size = 10240; double dp;

  // Create a dome environment.  The first step in any dome program.
  dome tdome(argc, argv);

  // Each of these vectors will be spread across all processes.
  dVector<double> v1(tdome, vector_size);
  dVector<double> v2(tdome, vector_size);
  dVector<double> prod(tdome, vector_size);


  // Assign the values to the vectors
  v1 = 1.0;   v2 = 3.14;

  // Compute the product, using the overloaded vector product operator.
  prod = v1 * v2;

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

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

Figure 1: A simple dot product program using Dome.

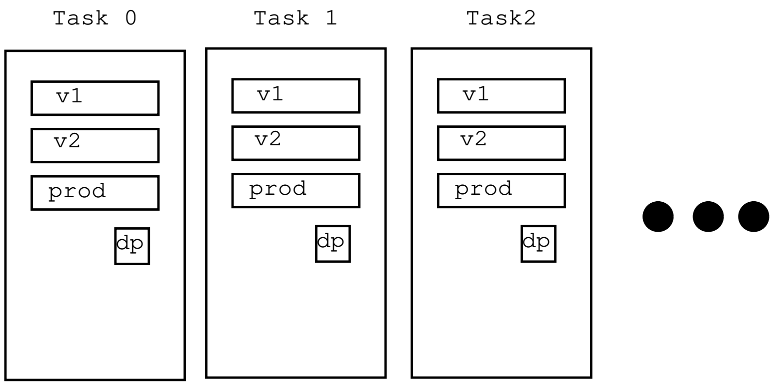

Figure 2: Multiple tasks executing the Dome dot product program.

floating point numbers. Notice that the Dome environment variable `tdome` is given in the declaration of the dVectors. The dimension of the dVectors is also given. These dVectors will be stored using a block distribution. By default the vectors will be automatically load balanced. The distribution and load balancing can be overridden by the programmer by using various options in the declaration.

The dot product program then assigns values to the vectors v1 and v2. These assignments of a scalar to a vector are done in parallel. For instance, all tasks in parallel perform the assignment on the elements in the block stored on that task. The assignment of $prod = v1 \times v2$ also happens in parallel. The elements of v1 and v2 are pairwise multiplied and the result is assigned to prod.

At this point in the program each task has a block of v1, v2, and prod. In order to compute the dot product we need to calculate a global sum on the prod vector. The dVector class contains a method called `gsum`. This method returns the global sum of all the elements in the vector. In the gsum method each task calculates the sum of its elements and then those partial sums are combined into a global sum. Each task receives a copy of the global sum, thus the scalar `dp` is the same in all tasks after the call to gsum.

The data layouts of the dot product program are depicted in Figure 2. Here we can see the vectors are block distributed and the scalars such as `dp` are replicated.

The equivalent PVM dot product program is several pages in length. If we add to this the load balancing and check-pointing facilities of Dome, the equivalent PVM program would be much longer. The point we would like to make here is that Dome programs are easier to write and the underlying support for load balancing and fault tolerance can be easily hidden from the programmer using object oriented techniques.

2.1 Extending Dome Classes

As with most C++ classes, Dome classes such as dVectors can be easily extended. For instance, a dVector of complex numbers can be easily created, as is shown in Figure 3. Here a complex class is defined with the normal operators that are needed for such a class. The only Dome specific methods that must be defined are `pvm_pk`, `pvm_upk`, and `dome_initialize`. The `pvm_pk` and `pvm_upk` functions tell Dome how to pack the class into a PVM message. For the complex number class this is simply a `pvm_pkfloat` for the real and imaginary parts of the complex number.

3 Dome Load Balancing

Dome programs typically execute on a heterogeneous collection of multiuser machines. In such an environment dynamic load balancing is very important. Not only can the native speeds of the machines be vastly different but the available performance of the machines can vary based on externally imposed loads. Our current approach to load balancing in Dome is to do periodic synchronization and data migration among the parallel tasks. This load balancing is completely transparent to the programmer.

We call the synchronization and data migration a load balancing phase. Likewise, the time between load balancing phases, when useful work is performed, is called a work phase. We do not attempt to predict the performance of work phases, rather we measure the performance of a task during its work phase and use this information for load balancing. Load balancing phases are triggered by the number of operations performed on Dome objects. For instance, a multiply of two dVectors is a single operation. After a certain number of operations have been performed, say 50, a load balancing phase begins. During a load balancing phase, tasks exchange information regarding their performance during the preceding work phase. Currently this performance information simply contains the elapsed time, measured by the system clock, that it took to perform the previous work phase. During the load balancing phase, Dome tasks communicate using a ring topology. Each task exchanges its performance information to its left and right neighbor in the ring. Based on this information, portions of the Dome objects are migrated. For instance, if a task is slower than its neighbors then it will send part of its Dome objects to the speedier neighbors. Note that this data movement is local, involving only neighbors. The advantages of this approach are that it reduces the amount of global synchronization among the tasks and it will converge to the right distribution using a relative measure of the load. The disadvantage is that it converges slowly.

We have built a tool which shows the changes in the distribution of dVectors

```
#include <dVector.h>

class complex {
private:
  double real;
  double imaginary;
public:
  complex();
  ~complex();
  complex& operator+=(const complex& rhs);
  complex& operator=(const complex& rhs);
  complex operator/(const complex& rhs) const;
  complex operator*(const complex& rhs);
  complex operator-(const complex& rhs);
  complex operator+(const complex& rhs);
  int operator<(const complex& rhs);
  int operator>(const complex& rhs);
  int operator!=(const complex& rhs);

  friend ostream& operator<<(ostream& s, complex& a);
  friend ifstream& operator>>(ifstream& s, complex& a);
  friend int pvm_pk( complex *np, int cnt, int std );
  friend int pvm_upk( complex *np, int cnt, int std );
  friend void dome_initialize(complex& a);

};

main(int argc, char *argv[]){

  dome e(argc,argv);
  dVector<complex> v1(e,100);
}
```

Figure 3: Extending dVectors to handle complex numbers.

as they change over time. Figure 4 shows a screen dump of this tool. The upper window in Figure 4 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 are on dVector computations and not a more general measurement of load. However, this computation reflects some function of the load on the processor.

This approach to dynamic load balancing is straight forward and can be done relatively quickly. Eventually we plan on developing more sophisticated load balancing techniques, but first it is essential to understand how this simple approach performs.

3.1 Results of Load Balancing

In order to evaluate Dome and this approach to load balancing we have performed the following experiment. We have written a matrix multiply in Dome using dVectors. We compare this code to a sequential version of matrix multiply and a version written using PVM directly. The PVM program uses the same algorithm as the Dome program. Dome is implemented using PVM, therefore this comparison allows us to measure the overheads involved in the current implementation of Dome. The code for all three versions can be found in the appendices. It is important to note the Dome and sequential versions are the same length while the PVM version is 69% longer.

We ran the programs on the Alpha cluster at the Pittsburgh Supercomputing Center. This cluster is an isolated network of DEC Alpha workstations connected by a DEC Gigaswitch. (The Gigaswitch provides point to point FDDI between the workstations.) No other users were allowed access to the cluster during our experiments. We ran the matrix multiply codes while imposing various levels of additional load on one of the machines. The additional load imposed consisted of multiple copies of the sequential version of the matrix multiply program running in an infinite loop. We also adjusted the frequency of the load balancing.

Figure 5 shows the runtimes for the programs on the system with no additional load. All times shown are for 10 matrix multiplies, $C = A \times B$, where A is 3×262144 and B is 262144×3. The matrices are double precision. All programs were compiled with the DEC cxx compiler using optimization. PVM version 3.2.6 was used in the distributed cases. The **seq** data point indicates the runtime for the sequential matrix multiply. The **PVM** line indicates the runtimes for the PVM program, some speedup was achieved. The **lb** cases show the runtimes for the Dome program using various numbers of load balancing phases and no load balancing phase in the **no lb** case. For the runtimes in Figure 5 the machines are evenly loaded, Dome's load balancing hinders the

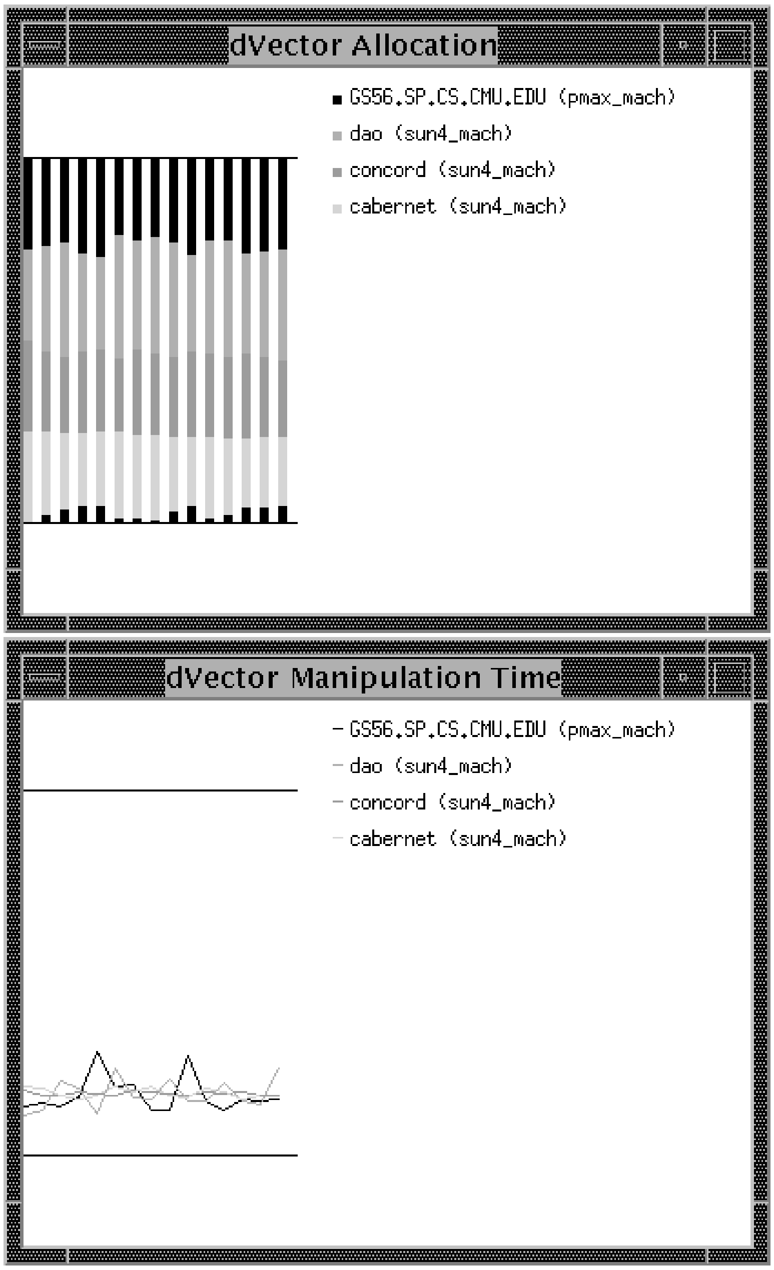

Figure 4: dVector mappings and machine loads over time.

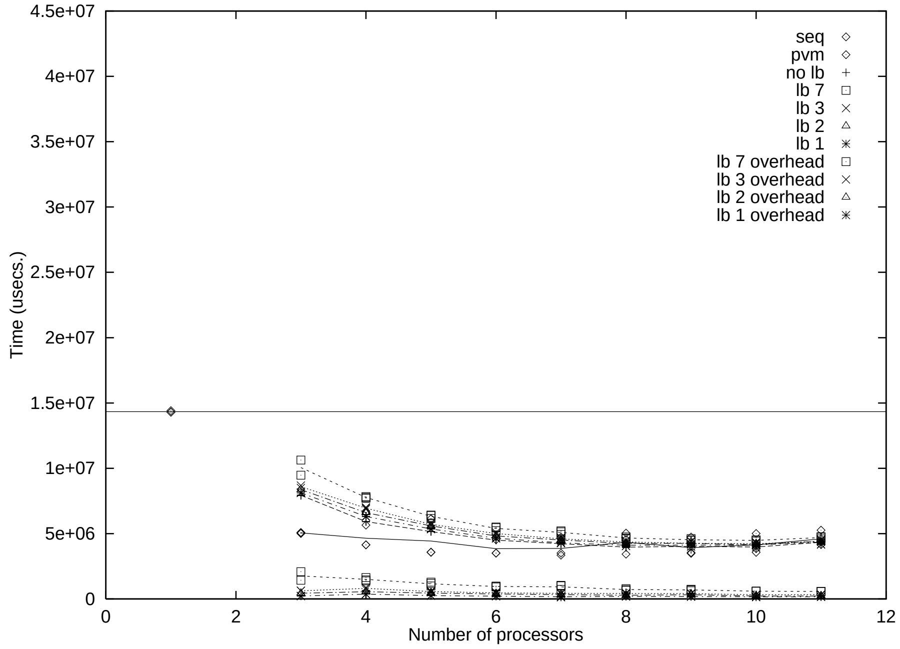

Figure 5: Timing results in a balanced system.

computation slightly due to the cost of periodic synchronization to determine that the system is balanced. All the Dome values are very close except for the **lb 7** case. Note that there is some load balancing taking place as shown by the **overhead** data. The overhead includes the time for each task to send the number of elements it has and its time per element to its neighbors and to do the data movement specified by the resulting calculations. The closeness of the **lb** values demonstrates that at the very least, the load balancing code can absorb its own overhead.

Figure 6 displays the results of running the programs while one extra loader process is running on one of the workstations (again this loader process is the sequential matrix multiply in an infinite loop). As expected, the sequential runtime has roughly doubled. The PVM runtimes are slightly longer than in the no load case. The Dome runtimes have also increased as have the different values for the load balancing overhead. This increase in the overhead is cancelled in most of the load balancing cases by the redistribution of the dVector and, although difficult to see in this figure, some of the load balanced runtimes are quicker than with no load balancing.

Finally, Figure 7 shows the runtimes where two loader processes are running

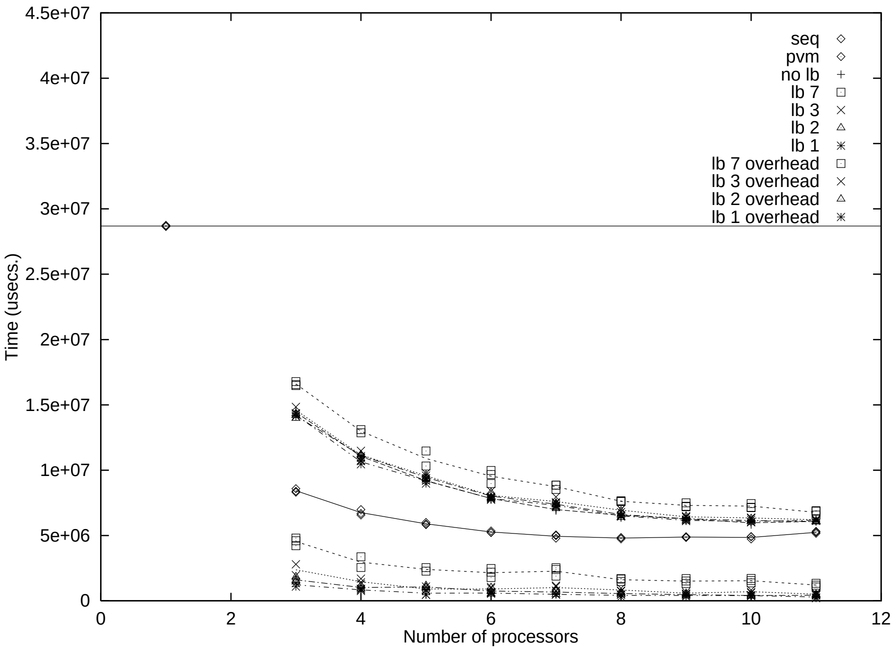

Figure 6: Timing results with one loader process.

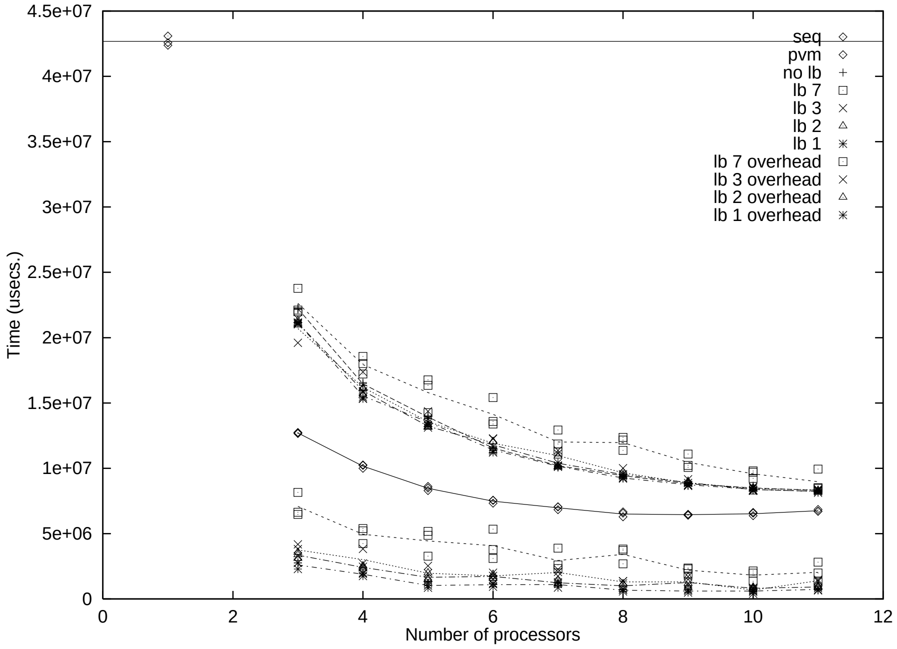

Figure 7: Timing results with two loader processes.

on one of the processors. Again as expected, the sequential program is about three times slower than the no load case. Both the PVM and Dome versions are slower than their counterparts in the two previous figures. The load balancing is now starting to consistently show improvement over the non-load balanced case. The **lb 1** case is consistently faster than the **no lb** case. The runtime has become much more erratic for the frequently load balanced Dome program.

The distribution of the elements of a dVector on five processors with different amounts of load is shown in Figure 8. Note that there is some data movement even in the "balanced" case. The effects of having extra load on processor 0 is clear. The reduction in the number of elements on this processor causes the other processors to increase their share of the dVector. The number of elements on these other processors is also maintained in balance so that the order of these processors in terms of their number of elements is preserved and the number of elements on each is in approximately the same ratio as in the balanced case.

These results are surprising in two ways. The overhead of using Dome is greater than expected and our load balancing scheme cannot overcome this overhead greatly. We already use some time (and space) saving memory management techniques; however, we believe that we can do more in this area. We

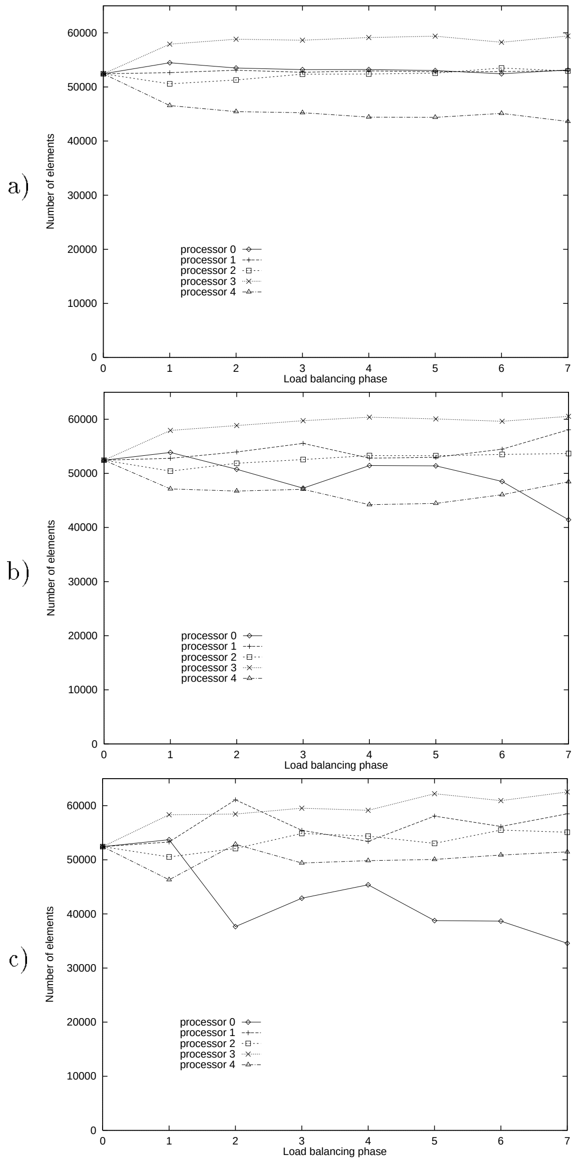

Figure 8: Distribution of vectors at different load balancing phases for a) a balanced system, b) one loader process and c) two loader processes.

are also unsure of the differences that compiler optimization is causing between the Dome and PVM implementations. In Dome, we specify multiplication between corresponding elements in the dVector producing a product dVector, this is followed by the gsum operation that sums these elements. The PVM implementation produces the multiplication of an element of each dVector and accumulates the sum in one C++ statement. This may produce better register usage by the compiler. Probably an even bigger factor in the Dome overhead is due to an increase in the number of cache misses. The dVector multiply requires all the elements from three dVectors, the two sources and the result, to be accessed one element at a time. Following this, the gsum operation requires the entire result dVector to be traversed. Since the portion of these dVectors that is resident on each single processor is quite large (approximately 680 Kbytes in the 3 processor case), it cannot fit in the cache in its entirety. The PVM program just performs the gsum and multiply at the same time on the two dVectors and accumulates the sum causing a lot fewer cache misses.

We also believe that the load balancing should be able to consistently improve on the runtime to a greater extent than it currently does. By reducing the load balancing costs, the runtime benefits will be more evident. We can reduce the costs by decreasing the number of messages used in a load balance phase and by doing more careful timing of the computation phases. More careful timing will also allow more accurate decisions to be made during the load balance phases.

Clearly more work needs to be done with respect to the efficiency of the object oriented approach taken in Dome and to the effectiveness of the dynamic load balancing implementation.

4 Architecture Independent Checkpoint and Restart

As we increase the number of machines and the average runtime of our computations, it is becoming clear that any practical parallel programming environment will have to provide some sort of failure tolerance. When designing Dome's fault tolerance features, there are a number of issues that we have considered. First, the reduction in the mean time between failures makes it increasingly necessary to use a checkpoint-based strategy rather than starting over after each failure. When more than one process is involved in the computation, in addition, we must take care to coordinate the states of the multiple checkpoints. Furthermore, in a heterogeneous environment, architecture independence is very important; if a process on a Cray C90 fails, the chances of quickly finding another C90 to start it on are slim.

Some preliminary work has been done on the general problem of checkpoint and restart of PVM programs [10]. In general, checkpoint and restart schemes work as follows. Periodically, process state is saved to a checkpoint file. If a failure occurs then the process can be restarted from a saved checkpoint. Leon's and most other current methods work by saving the entire process state (i.e., a core image) to the checkpoint file. Dome attempts to address the problem from a high level by saving only the Dome specific data structures. (By overloading our Dome checkpoint operation, the programmer may also cause ordinary non-Dome data structures to be included in the checkpoints.) If each object provides a checkpoint method then all of the Dome data structures can be easily checkpointed. The checkpoint methods are implemented in such a way as to store the data in an architecture independent format such as XDR. Upon restart the checkpoint is valid on any architecture, assuming the proper restart functions have been implemented.

Although we do not discuss it in detail here, we have demonstrated architecture independent checkpoint and restart with Dome dVector objects on a real application, in this case a molecular dynamics code developed at the Pittsburgh Supercomputing Center.

5 Conclusions and Future Work

This paper presents a brief introduction to Dome (distributed object migration environment) and an initial evaluation of a dynamic load balancing scheme implemented in Dome. Dome addresses the issues of ease of programming, dynamic load balancing, and architecture independent checkpoint and restart. We show that Dome programming is easier than message passing for parallel programming.

Our initial evaluation of Dome's performance shows considerable overhead. The load balancing heuristic we currently use for dVectors cannot compensate for this overhead. However, in many cases the heuristic is successful in masking the additional cost of load balancing. Although these results may be seen as discouraging, we view them simply as an indication that the problem is non-trivial; as we reduce the costs of the basic dVector operations, we expect to see the results of load balancing improve.

Our next step with regards to Dome is to optimize our implementation by carefully studying the memory usage and message passing overheads involved in executing a Dome program with dynamic load balancing. We are also continuing our exploration of the architecture independent checkpoint and restart facilities.

We are convinced the Dome paradigm is a good one. We are continuing our efforts toward developing a rich set of objects which efficiently support parallel programming in a shared heterogeneous environment.

6 Availability

Dome is implemented in C++ and uses PVM extensively, both of which are widely available. We plan on making Dome available on a variety of platforms. We will be distributing Dome to the community and will be providing limited support. Contact the authors for a current copy of the software.

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

[5] 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.

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

[7] 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.

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

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

[10] Juan Leon, Allan L. 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.

[11] Ewing Lusk, Ross Overbeek, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

A Sequential Matrix Multiply Code

```
//  Adam Beguelin
//  Jan 1994
//  Sequential Matrix multiply, C = A * B

// multiply an MxN by an NxL matrix to get an MxL matrix
// The matrix C is replicated on all processors
// The rows of A are dVectors
// The cols of B are dVectors

// Regular C includes
extern "C" {
#ifndef __DECCXX
extern long random(void);
#else
extern long random();
#endif
}

// C++ includes
#include <stdio.h>
#include <strings.h>
#include <assert.h>
#include <stdlib.h>
#include <stream.h>
#include <math.h>
#include <fstream.h>
#include "debug.h"

// Dome include files

#include "dTimer.h"

const int MAXM = 3;
const int MAXN = 262144; // .25 MB
const int MAXL = 3;

#ifndef __DECCXX
  // Declare some vectors.
  double a[MAXM][MAXN];
  double b[MAXN][MAXL];

  // result matrix
```

```
  double c[MAXM][MAXL];
#endif

// The main program

int
main(int argc, char *argv[])
{
  int nloops=10;
  int m,n,l;
  double dot_product;

  // Get the input parameters
  if (argc != 4) {
    cerr << "usage: smult <m> <n> <l>\n";
    return -1;
    }

  m = atoi(argv[1]);
  n = atoi(argv[2]);
  l = atoi(argv[3]);

#ifdef __DECCXX
  // Declare some vectors.
  double a[MAXM][MAXN];
  double b[MAXN][MAXL];

  // result matrix
  double c[MAXM][MAXL];
#endif


  // a timer
  dTimer t;

  int i,j,k;

  // Generate the inputs
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        a[i][j] =  double(random() % 100)/10.0;
 for (i = 0; i < n; i++)
    for (j = 0; j < l; j++)
        b[i][j] =  double(random() % 100)/10.0;
```

```
  t.start();

  while(nloops--) {
  for (i = 0; i < m; i++) {
    for (j = 0; j < l; j++) {
        for (k = 0, c[i][j] = 0.0; k < n; k++)
            c[i][j] += (a[i][k] * b[k][j]);
        }
    }
  }

  t.stop();

  cout << "\nTime " << t.elapsed() << "\n";


}
```

B Dome Matrix Multiply Code

```
//  Adam Beguelin
//  Jan 1994
//  Matrix multiply, C = A * B

// multiply an MxN by an NxL matrix to get an MxL matrix
// The matrix C is replicated on all processors
// The rows of A are dVectors
// The cols of B are dVectors

// Regular C includes
extern "C" {
#ifndef __DECCXX
extern long random(void);
#else
extern long random();
#endif
}

// C++ includes
#include <stdio.h>
#include <strings.h>
#include <assert.h>
#include <stdlib.h>
#include <stream.h>
#include <math.h>
#include <fstream.h>
#include "debug.h"

// Dome include files
#include "dome.h"
#include "dVector.h"
#include "dTimer.h"
#include "domeobj.h"

const int MAXM = 100;
const int MAXL = 100;

// The main program

int
main(int argc, char *argv[])
{
```

```
 int nloops=10;
 int nproc;
 int m,n,l;
 double dot_product;

 // Get the input parameters
 if (argc != 5) {
   cerr << "usage: mmult <nproc> <m> <n> <l>\n";
   return -1;
   }

 nproc=atoi(argv[1]);
 m = atoi(argv[2]);
 n = atoi(argv[3]);
 l = atoi(argv[4]);

 // Create a dome environment.  The first step in any dome program.
 dome tdome(argc,argv,nproc);

 // Declare some distributed dVectors.
 // Each of these vectors will be spread across all processes.
 dVector<double> *a[MAXM]; // Rows of a are dVectors
 dVector<double> *b[MAXL]; // Cols of b are dVectors

 // result matrix
 double c[MAXM][MAXL];

 // a timer
 dTimer t;

 int i,j;

 // Get the input from cin.
 for (i = 0; i < m; i++) {
   a[i] = new dVector<double>(tdome, n);
   *a[i] = double(random() % 100)/10.0;
   }
for (i = 0; i < l; i++) {
   b[i] = new dVector<double>(tdome, n);
   *b[i] = double(random() % 100)/10.0;
   }

 t.start();
```

```
  while(nloops--) {
  for (i = 0; i < m; i++) {
    for (j = 0; j < l; j++) {
        c[i][j] = (*a[i] * *b[j]).gsum();
        }
    }
  }

  t.stop();

  cout << "\nTime " << t.elapsed() << "\n";

  // delete the vectors
  for (i = 0; i < m; i++) delete a[i];
  for (j = 0; j < l; j++) delete b[j];
  delete [] a ;
  delete [] b ;
}
```

C PVM Matrix Multiply Code

```
//  Adam Beguelin
//  Jan 1994
//  Matrix multiply, C = A * B
//  PVM Version

// multiply an MxN by an NxL matrix to get an MxL matrix
// The matrix C is replicated on all processors
// The rows of A are dVectors
// The cols of B are dVectors

// Regular C includes
extern "C" {
#ifndef __DECCXX
extern long random(void);
#else
extern long random();
#endif
extern int pvm_errno;
}

// C++ includes
#include <stdio.h>
#include <strings.h>
#include <assert.h>
#include <stdlib.h>
#include <stream.h>
#include <math.h>
#include <fstream.h>
#include <pvm3.h>
#include "dTimer.h"
#include "pvm3pack.h"
#include "debug.h"

extern int alb_spawn(char *, char **, int , int *);

const int MTAG = 1971;
const int MAXM = 100;
const int MAXL = 100;


// The main program
int
```

```
main(int argc, char *argv[])
{
  int nloops=10;
  int count;
  int m,n,l;
  double dot_product;

  dTimer t;

  // PVM vars
  int *tids;
  int mytid, mygid, parent;
  int len, mtag, info;

  // Register with PVM
  if ((mytid = pvm_mytid()) < 0) { pvm_perror("pmult"); return -1; }
  pvm_setopt(PvmRoute, PvmRouteDirect);
  if ((mygid = pvm_joingroup("pmult")) < 0) { pvm_perror("pmult"); return -1; }
  parent = pvm_parent();

  // Get the input parameters
  if (mygid == 0) {
    if (argc != 5) {
        cerr << "usage: pmult <count> <m> <n> <l>\n";
        return -1;
        }
    count=atoi(argv[1]);
    m = atoi(argv[2]);
    n = atoi(argv[3]);
    l = atoi(argv[4]);
    }

  // Spawn the PVM tasks
  if (mygid == 0) {
    tids = new int[count];
    tids[0] = mytid;
    info = alb_spawn("pmult", (char **)0, count-1, tids+1);
    if (info != (count-1)) {
        cerr << "Spawn failed\n" << tids[1] ;
        pvm_lvgroup("pmult");
        pvm_exit();
        delete [] tids;
        return -1;
        }
```

```
  pvm_initsend(PvmDataDefault);
  pvm_pk(&count); pvm_pk(&m); pvm_pk(&n); pvm_pk(&l);
  pvm_mcast(tids, count, MTAG);
  }
else {
  pvm_recv(parent, MTAG);
  pvm_upk(&count); pvm_upk(&m); pvm_upk(&n); pvm_upk(&l);
  }

// Declare some distributed dVectors.
// Each of these vectors will be spread across all processes.
double *a[MAXM];  // Rows of a are dVectors
double *b[MAXL];  // Cols of b are dVectors

// result matrix
double c[MAXM][MAXL];

double tmp;
int i,j,p;

// Generate the vectors
len = n/count;
if (mygid < (n % count)) len++;
for (i = 0; i < m; i++) {
  a[i] = new double [len];
  tmp = (random() % 100) / 10.0;
  for (j=0; j < len; j++) a[i][j] = tmp;
  }
for (i = 0; i < l; i++) {
  b[i] = new double [len];
  tmp = (random() % 100) / 10.0;
  for (j=0; j < len; j++) b[i][j] = tmp;
  }

// Do the multiply

t.start();

while(nloops--) {
for (i = 0; i < m; i++) {
  for (j = 0; j < l; j++) {
      c[i][j] = 0.0;
      // Calculate the partial sum
      for (p = 0; p < len; p++)
```

```
            c[i][j] += a[i][p] * b[j][p];
        // Use a unique message tag for each elt of c
        mtag = i*m+j;
        if (mygid == 0) {
            for (p = 1; p < count; p++) {
                pvm_recv(-1, mtag);
                pvm_upk(&tmp);
                c[i][j] += tmp;
                }
            pvm_initsend(PvmDataDefault);
            pvm_pk(&c[i][j]);
            pvm_mcast(tids, count, mtag);
            }
        else {
            pvm_initsend(PvmDataDefault);
            pvm_pk(&c[i][j]);
            pvm_send(parent, mtag);
            pvm_recv(parent, mtag);
            pvm_upk(&c[i][j]);
            }
        }
    }
  }
  t.stop();

  cout << "\nTime " << t.elapsed() << "\n";

  // delete the vectors
  for (i = 0; i < m; i++) delete [] a[i];
  for (j = 0; j < l; j++) delete [] b[j];

  delete [] tids;
  pvm_lvgroup("pmult");
  pvm_exit();
  return 0;
}
```