Dome: Parallel Programming in a Distributed Computing Environment

José Nagib Cotrim Árabe, Adam Beguelin, Bruce Lowekamp, Erik Seligman, Mike Starkey, and Peter Stephan
Email: {arabe,adamb,lowekamp,eriks,starkey,pstephan}@cs.cmu.edu
School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The Distributed object migration environment (Dome) addresses three major issues of distributed parallel programming: ease of use, load balancing, and fault tolerance. Dome provides process control, data distribution, communication, and synchronization for Dome programs running in a heterogeneous distributed computing environment. The parallel programmer writes a C++ program using Dome objects which are automatically partitioned and distributed over a network of computers. Dome incorporates a load balancing facility that automatically adjusts the mapping of objects to machines at runtime, exhibiting significant performance gains over standard message passing programs executing in an imbalanced system. Dome also provides checkpointing of program state in an architecture independent manner allowing Dome programs to be checkpointed on one architecture and restarted on another.

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, 15], P4 [5], Linda [7], Isis [2], Express [14], and MPI [16] allow a programmer to treat a heterogeneous network of computers as a parallel machine. These tools are useful, but for efficient and practical use, load balancing and fault tolerance mechanisms must be developed that will work well in a heterogeneous multi-user environment.

When using most conventional parallel programming methods, one needs to partition the program into tasks and manually distribute the data among those tasks—a difficult procedure in itself. To further complicate matters, 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; bandwidths in local area networks can vary from 10 Mbit Ethernet to 800 Mbit HiPPI, and message latency can vary greatly. System failure is yet another consideration. If a long-running application is executing on a large number of machines, failures during program execution are likely. Processor heterogeneity complicates support for fault tolerance. The Distributed object migration environment (Dome) addresses these parallel programming issues for heterogeneous multi-user distributed environments.

Dome provides a library of distributed objects for parallel programming that allows programmers to 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 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. We show that a parallel program can be made failure resilient through Dome’s architecture independent checkpoint and restart mechanisms.
2. 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 use PVM for process control and communication. The Dome library uses operator overloading to allow the application programmer simple manipulation of Dome objects and to hide the details of parallelism. When an object of one of these classes is instantiated, it is automatically partitioned and apportioned within the distributed environment, and computations using this object are performed in parallel across the nodes of the current PVM virtual machine.

When a program using the Dome library is run, Dome first creates the processes which constitute the distributed program using a single program multiple data (SPMD) model. 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. Dome keeps track of these processes and the existence and distribution of all Dome variables in the program. Dome maintains global checkpointing and load balancing information which can be controlled by the user through input parameters.

A Dome class generally represents a large collection of similar and related data elements. Those elements are partitioned and distributed among the processes of the distributed program when an object is instantiated. 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. Finally, dynamic indicates that the elements are initially distributed evenly, but the data is reapportioned among the processors periodically through dynamic load balancing performed at given intervals. The user may indicate the particular method for partitioning a given Dome object when that object is declared.

A Dome operation 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. 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 4.

3. Dome programming

Figure 1 illustrates the simplicity of programming with Dome objects. This program performs a standard inner product operation on a pair of vectors.

The standard argc and argv parameters to main 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 Dome to spawn remote copies of the program with the same argument list that the user specified originally.

Next, 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 and is used so that the variable can be included in a Dome checkpoint. Three dVector objects, vector1, vector2, and prod, are also declared as vectors of 10240 doubles. By default the vectors will be distributed among the processes using the dynamic distribution, initially assigning approximately 10240/p elements of each vector, where p is the total number of processes, to each process.

The program next assigns the values 1.0 and 3.14 to each of the elements of vector1 and vector2 respectively. The statement which follows, prod = vector1 * vector2, performs two Dome operations, an element-wise multiplication of the vectors vector1 and vector2 and the assignment of the result to the distributed vector prod. All of these operations are performed in parallel on the elements of the distributed vectors assigned to each processor.

The gsum method causes each processor in the distributed program to calculate a local sum of the elements of the vector prod. The local sums are combined to complete the standard inner product calculation which is assigned to the scalar value dp on all processors.

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 4 and 5 respectively.

This simple program demonstrates that 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 lengthier and more difficult to write, and similar load balancing and checkpointing would complicate it further.

Dome provides all of the common operator primitives for manipulation of its distributed objects. In addition, there are constructs available which allow the user great latitude in defining operations on the Dome objects. These include methods which allow the user to define functions to be applied to the elements of Dome objects, access and manipu-
int main(int argc, char *argv[]) {
    dome_init(argc, argv);
    dScalar<int> vector_size = 10240;
    dScalar<double> dp = 0.0;
    dVector<double> vector1(vector_size);
    dVector<double> vector2(vector_size);
    dVector<double> prod(vector_size);

    vector1 = 1.0; vector2 = 3.14;
    prod = vector1 * vector2;
    dp = prod.gsum();
    cout << "Dot product is " << dp << 'n';
}

Figure 1. Dome inner product program.

late a subset of the elements through ranges and masks, and customize the datatype of the object via Dome’s templated C++ classes.

To show that Dome is useful for more complicated programs, we have implemented several real applications including a molecular dynamics water simulation [25] and a multidimensional integration nuclear physics application [6].

4. 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 mentions an approach to load balancing based on the interconnection network.

4.1. Load balancing techniques

Load balancing involves assigning 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. Many metrics can be used in attempting to capture and predict the performance of a particular processor: processor speed, available memory, length of the current run queue, percentage of idle time in the recent past, number of recent network interrupts, and others. Dome simply uses the actual rate at which the processors have been executing the program to estimate near term future performance for that same program.

When a program begins execution in the parallel virtual machine, all dynamically distributed Dome objects are initially distributed evenly among the 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. Presently, an initial load balancing phase is performed after the first few Dome operations. This early load balancing phase captures the initial load conditions and performance of the 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, depending on the load balance interval, because the fastest machines have to wait at the first synchronization point, potentially remaining idle for a long time. 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 can be set by the user.

Another major issue is whether the load balancing decision should be global or local. 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. The master broadcasts the new distribution mapping, and the tasks exchange data. 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 from scaling well to a large number of machines. Another option is to have processors simply exchange control data locally with their neighbors. 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.

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. We have developed a separate package, called ECO, which deals with the issue of network load balancing by automatically determining network topology, partitioning the network into subnets, and establishing optimized communication patterns. More details on ECO can be found in [21].


4.2. Initial timing results

In order to evaluate the Dome approach to load balancing, we have written a matrix multiply program using dVectors (mmndome). This implementation was compared with three other versions of matrix multiply: a sequential version (mmseq), a PVM version that uses the same algorithm as the Dome program (mmpvm), and another PVM version 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 overhead of the Dome implementation, as well as the overhead of load balancing. It is also worth mentioning that while the Dome program has roughly the same size as the sequential version the PVM implementations are almost double that size and must deal explicitly with data distribution, gathering, and synchronization.

The results given here are for 240 matrix multiplies, where the matrices are of sizes \( \frac{3 \times 262144}{n_0} \times \frac{262144 \times 3}{n_0} \) double precision elements. Three experiments were performed:

1. Unloaded, balanced system: exclusive use of 6 DEC Alpha workstations interconnected by Ethernets.
2. Imbalanced, stable system: same environment as before, but one machine was artificially loaded.
3. Production system (loaded, unstable): 6 DEC Alpha workstations interconnected by a DEC Gigaswitch run under normal production conditions, scheduled under DQS at various hours of the day and days of the week. 45 runs for each experimental data point were performed.

The mmndome program 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 using different numbers of machines, varying matrix sizes, and different numbers of load balancing phases. The results presented here for the case of six machines are representative of the overall behavior of the programs.

Figure 2 shows a comparison of the times obtained for experiment 2. Load balanced cases performed well, being 35% faster than mmpvm and 13% faster than mmpvmopt. The figure also shows that the overhead of Dome without load balancing is quite high in this case. Finally, as expected, there is not much difference between doing one and multiple load balancing phases since the system is stable.

Figure 3 shows the results under production conditions. In this case, Dome with load balancing is only 10% slower than mmpvmopt. On the other hand, Dome with load balancing presents a significant gain of 44% over the equivalent PVM implementation (mmpvm).

5. Checkpointing

While performing large computations on a network of workstations offers many advantages, handling failures on one or more of the nodes in the cluster becomes an important concern. As the number of workstations in a cluster increases, the chance of a failure during a particular computation increases exponentially. Thus, it is vital that some kind of fault tolerance mechanism be incorporated into any system designed for extended execution on a workstation clus-
Checkpoint preprocessor has also been completed and has for this application. A preliminary implementation of the head is low enough to provide a good expected runtime when checkpointing is performed very frequently, the over-

lication, md collected using this package with a molecular dynamics ap-

package has been completed, and benchmarks have been usable high level checkpointing features for Dome.

A fault tolerance package for use with Dome can be im-

plemented at various levels of programming abstraction. At a high level the application programmer can call a set of C++ methods to checkpoint a program’s data structures and to restart that program from the checkpointed data. This method provides a fault tolerance package which is highly portable since it uses no machine-dependent constructs in creating a checkpoint. With this method the user is respon-

sible for writing a program with a structure that is sim-

ple enough that the program counter and stack need not be saved. Many scientific applications meet this requirement. A second high level method uses a preprocessor to insert most of the checkpointing calls automatically. The use of a preprocessor offers the same advantages of portability while helping to reduce the work required of the application pro-

grammer. Calls to the checkpointing methods are inserted around any function call that may lead to a checkpoint and where control flow enters and exits these functions. These calls save the stack information during a checkpoint and re-

store the stack upon restart. In general the expansion in code size is small and linear. Finally, a low level checkpoint-

ing package in which the system periodically saves the pro-

gram’s memory image upon interrupt could be used. Low level methods are very simple to use, requiring no additional work of the programmer, and the program can be restored from the saved checkpoints easily. Neither the low level fault tolerance packages nor the checkpoint files that they produce, however, are generally portable to other platforms.

While they tend to require more work from the user, the high level fault tolerance methods mesh very well with the Dome system since they allow for easy portability to any system that supports PVM and C++. Furthermore, since Dome applications execute in a heterogeneous environment, it is vital that checkpoints created on one architecture are usable on others. Thus, we have concentrated on developing usable high level checkpointing features for Dome.

An implementation of a high level checkpoint and restart package has been completed, and benchmarks have been collected using this package with a molecular dynamics ap-

lication, md, that uses Dome. Timings indicate that even when checkpointing is performed very frequently, the over-

head is low enough to provide a good expected runtime for this application. A preliminary implementation of the checkpoint preprocessor has also been completed and has been successfully tested on Dome programs.

The portability of our checkpointing package has also been demonstrated by running md with checkpointing on both DEC Alpha and on SGI workstations. Furthermore, the portability of the checkpoints themselves has also been suc-

cessfully accomplished by restarting md on Alpha workstations from checkpoints created on the SGI workstations. For more detailed information on Dome’s checkpointing meth-

ods, see [23].

6. Related work

Although addressing similar aspects, Dome is not a re-

source management system like Condor [20], LoadLeveler [8], and DQS [10]. These systems also deal with load balancing, fault tolerance, parallelism, and heterogeneity. However, they operate on a different level, overseeing the distribution and placement of processes rather than data over the network.

Dome shares attributes with many other research projects. pC++ [3] extends C++ to a parallel programming language. High Performance Fortran [17] is an emerging standard for writing distributed memory parallel Fortran programs. While language based mechanisms for express-

ing parallelism and data mapping in distributed memory machines are important, we are most interested in using ex-

isting languages and exploring object oriented mechanisms for parallel and distributed computing. Knowledge gained in developing Dome can, however, be used in compilers that target heterogeneous networks.

LaPack++ [9] is an object oriented interface to the La-

Pack routines for parallel linear algebra. Like LaPack++, Dome provides a library of parallel objects. However, Dome focuses on objects of general use and provides fea-
tures like dynamic load balancing and fault tolerance that are not addressed by LaPack++.

6.1. Related load balancing work

There has been a considerable amount of theoretical work on matching task requirements to resources 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 computing system whose resources are also well known [4]. Most dis-

tributed multi-user systems have unpredictable loads, making these approaches impractical for general use.

Load balancing research in operating systems focuses on 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. [12] 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 [11]. 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. However, Dome’s load balancing strategy still agrees with Eager’s thesis that simple strategies are most effective and extends that idea to load balancing of parallel algorithms.

For parallel programs the source of load imbalances can be both internal, where the work distribution among the parallel tasks changes as the program runs, and external, which result from the 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.

Finally, in order to perform load balancing, resource management systems and schedulers in operating systems, in general, have to make use of load indices based on available system resources [13, 19]. Dome uses a direct approach for load measurement, namely, the timing of execution progress in each participating processor. This is a very simple way to capture both processor speed and amount of external and internal load present in each machine.

6.2. Related checkpointing work

Most checkpointing libraries for distributed systems address checkpointing in a homogeneous environment using system-specific techniques to capture consistent memory images from each process efficiently. Plank and Li’s [22], for example, is designed to minimize the checkpointing overhead on multicomputers. An architecture independent package has also been developed by Silva, Veer, and Silva [24], who have created a 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 [18]. As in Dome, they use a preprocessing mechanism for saving the state of distributed programs.

7. Future work

The Dome system is undergoing very active development. We are adding new classes and working with both computer science and computational science researchers to develop more production quality Dome applications in several domains. This is an important step towards showing that Dome is general enough to express a wide array of parallel algorithms.

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. This work will mesh well with the automatic network partitioning described in Section 4.1 and [21]. It is also possible that runtime metrics of communication performance can be used to develop a virtual topology for load balancing and collective communications.

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 and are built upon C++ and PVM they allow the system to be very portable. To date Dome has been ported to eight platforms: DEC Alpha OSF/1, IBM RS/6000 AIX, HP HPUX, Intel Paragon, Sparc SunOS, SGI Irix, DEC Ul- trix, and Cray C90. Current plans are for the Dome system to be released in 1996.

Preliminary measurements show that the runtime overhead of Dome programs is reasonable. Furthermore, this overhead can be overcome by load balancing in an imbalanced system, resulting in performance gains of up to 44%. 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.

References


