Application Development and Execution in a Virtual Computing Environment

Philip Rousselle, Rajesh Yadav, Salim Hariri*, Dongmin Kim,
Suresh Boddapati, and Paul Tymann
Northeast Parallel Architectures Center
Department of Electrical and Computer Engineering
Syracuse University,
Syracuse, NY 13244-4100

SCCS # 705

*Presenting and corresponding author, Deptartment of Electrical and Computer Engineering , 121 Link Hall, Syracuse University, Syracuse, NY 13244, hariri@cat.syr.edu, fax 315-443-2583

Abstract

As the size and complexity of high performance computing applications continue to increase, their computing, storage and connectivity requirements become more and more difficult to accommodate on any single computing platform. The Virtual Computing Environment (VCE) combines heterogeneous computing, storage and network resources into a flexible system for the execution of large scale applications. Current systems for managing distributed applications employ static allocation policies which are difficult to scale to large networks made up of geographically dispersed resources connected by high speed network technology (e.g. ATM). In this paper we present an application description framework which is used to develop large scale applications in the VCE. We also discuss two methods for scheduling tasks in the VCE: one emphasizes redundant task scheduling while the other emphasizes a more conventional use of load measurement techniques to select less loaded computers to run application tasks. Several example applications are included and used to demonstrate the performance of a Virtual Computing Environment which employs an IBM SP2 supercomputer, a cluster of DEC Alpha workstations, and a cluster of Sun and Silicon Graphics workstations interconnected by an ATM network.

1 Introduction

Grand Challenge applications require multiple computers, or even multiple supercomputers, to achieve acceptable performance. Some researchers are taking advantage of heterogeneous computing environments to achieve levels of performance that to date have not been possible. A climate simulation model described in [7] uses both SIMD and MIMD style parallel computers to achieve superlinear speedup. As applications that require multiple supercomputers and clusters of high performance workstations become more common, the need for systems to ease their development and automate their execution becomes clear. The Virtual Computing Environment *(VCE)* is a research effort which is investigating new techniques for developing and managing the execution of large scale applications which exploit high performance heterogeneous computing resources dispersed across a wide area network.

Previous attempts to harness the capacity of distributed resources and make them available to large applications are not suitable for managing the current generation of Grand Challenge applications. Utopia [10] allows a widely dispersed group of heterogeneous workstations to be used for remote task execution. However, Utopia is geared towards managing a network of privately owned Unix workstations. Tasks are placed on lightly loaded machines and remain there until they have completed. Users whose machines are used to host remote tasks may have to execute their own tasks remotely on other machines to realize good performance. Condor [6] also uses idle workstations to host long running applications and provides preemptive process migration. The process migration facilities in Condor require a high degree of homogeneity among the participants. Durra [1] provides many facilities which are useful for constructing large distributed applications. Durra is also oriented to a workstation environment and places more emphasis on fault tolerance than high performance computing.

New scheduling protocols must be devised to manage task execution in distributed supercomputing environments. Recent research has shown that for some tasks a cluster of DEC Alphas will provide comparable performance to an IBM SP2 [3]. If both these architectures are available at run time, the least loaded machine is preferable. Load measurement on supercomputers is more complex than for workstations. Workstations contain a single ready queue which can be sampled to gauge the system activity level [5]. In a supercomputer the load of each computing element must be sampled in order to compute an aggregate load. Comparing the loads of geographically dispersed supercomputers is complicated by communication delays and may be costly. Further, large tasks may have very long running times, so systems which emphasize precise load measurements in order to optimize initial task placement have no way to react to load changes during task execution. Conversely, the cost of migrating running tasks between supercomputers might be prohibitively expensive. Because load conditions are hard to measure and prone to

fluctuate, and process migration between high performance machines is problematic, redundant scheduling of performance critical tasks represents an interesting approach to guaranteeing good performance over a heterogeneous distributed computing environment. Several approaches to task scheduling are being investigated in the Virtual Computing Environment. Two that will be discussed in this paper are redundant task scheduling and selective initial task placement.

The Virtual Computing Environment has two principle components. The first is an application description framework which is used to describe the attributes of computing tasks and how these tasks can be arranged to construct large applications. The second is an execution environment which interprets application descriptions and executes them on a set of computers selected at runtime from machines available across a wide area network.

A prototype VCE has been constructed. It is being used to develop and run applications which exploit a variety of high performance architectures including an IBM SP2, and DEC Alpha, Sun and Silicon Graphics workstations connected to FDDI and ATM networks to investigate a variety of distributed systems issues including development methodologies and scheduling protocols.

The remainder of the paper is organized as follows. Section 2 gives an overview of the VCE execution environment. Section 3 discusses how applications are constructed from tasks using the application description framework. Section 4 details the scheduling and execution techniques used in the VCE. Section 5 describes the hardware configuration of a VCE prototype system, the applications running on it, and the performance results observed. Future work is proposed in Section 6.

2 The VCE Execution Environment

A VCE execution environment is a group of heterogeneous machines connected by a high performance network [9] (Figure 1). A portion of the computing resources on each machine is allotted to execution of VCE applications. Each machine participating in the VCE hosts a VCE daemon to perform scheduling and dispatching of VCE work. A dispatchable unit in the VCE is a *task*. Each task corresponds to an executable file for one or more of the participating machines. The attributes of each task are defined in a *task description*. An application consists of one or more tasks. Each task is a step in the computation. Applications are described by files in an *application description* library.

To run an application, a user starts an *execution process* on their workstation and gives the name of an application in the application description library. The execution process reads the application description and any task descriptions it refers to. It then communicates with the VCE daemons on the machines that could be used to host application tasks to determine which machines are available. The execution process collaborates with the VCE daemons to arrange

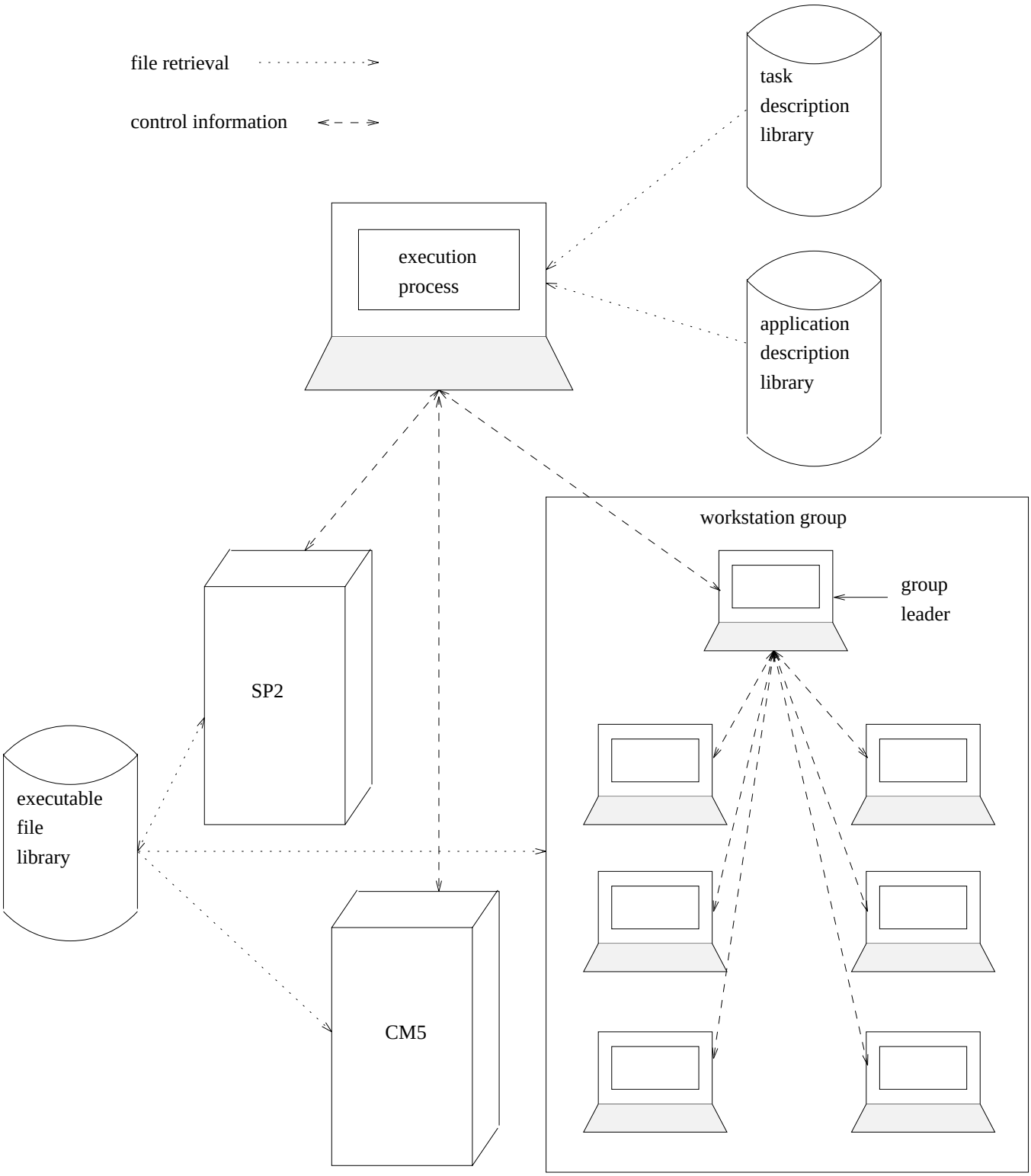

Figure 1: The VCE Execution Environment

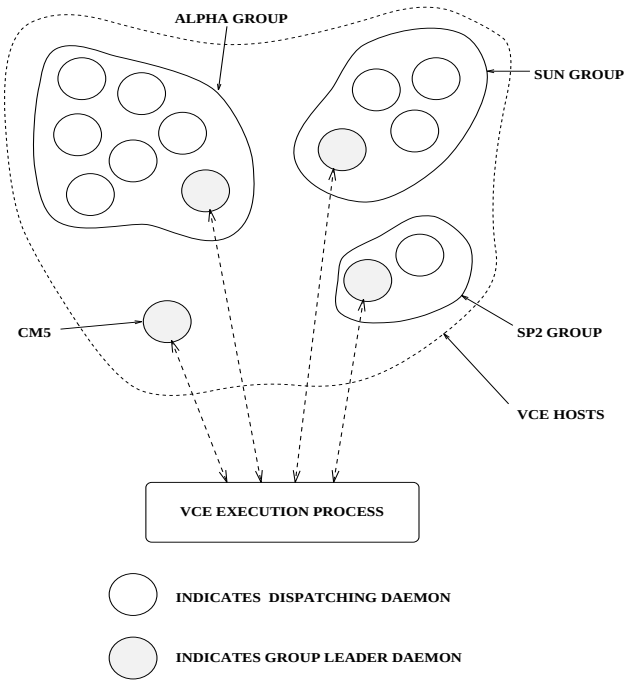

Figure 2: Organization of the VCE

execution of tasks as needed.

Some machines are organized into *groups* to reduce communication overhead and improve scalability. Specifically, whenever two or more machines are architecturally and binary compatible, they are organized as a group. One machine in each group acts as a group leader and communicates with the VCE execution process to manage and control the execution of an application. The group structure of the VCE allows an execution process to exchange scheduling information with group leaders only, rather than interfacing with every participating supercomputer and workstation. Machines within groups exchange load information with their group leaders so that when an execution process needs a machine to host an instance of a task, the group leader supplies the address of a lightly loaded machine. The relationship between VCE daemons and the execution process is illustrated in Figure 2.

Dispatching is accomplished when a VCE daemon on a host machine, under the supervision of an execution process, fetches an executable file and begins execution.

3 Application Description Framework

An application description language is presented in [1] which is used to construct fault tolerant Ada applications on a heterogeneous workstation network. The VCE application description framework uses a similar syntax to describe applications constructed for the heterogeneous VCE

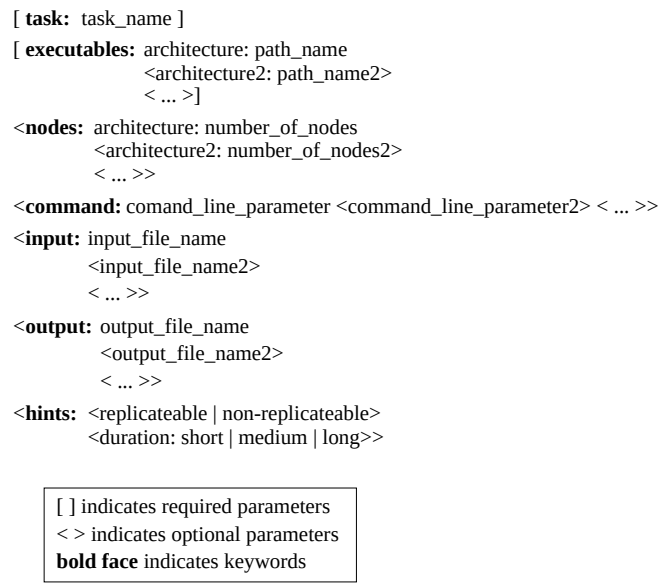

Figure 3: VCE Task Description Syntax

environment.

The VCE application description framework allows a user to specify the structure of a distributed application and give the VCE enough information about its characteristics to efficiently dispatch it on available hardware.

The two central constructs of the application description framework are *tasks* and *applications*. *Task descriptions* are the building blocks of VCE applications. *Application descriptions* completely specify how a set of tasks can be dispatched in a heterogeneous computing environment to perform useful work.

The complete syntax for task descriptions are given in Figures 3.

Figure 4 shows task descriptions for some common linear algebra operations. Each task is implemented on three different platforms. The *Nodes* specification gives a default value for how many nodes of each architecture are to be assigned to the task. The number of nodes is chosen by the task implementor to produce efficient granularity in common applications.

Figure 5 shows how the tasks of Figure 4 can be used to construct a linear equation solver application.

The default *nodes* specification given in the task description can be overridden in the application description. The application designer may have more information about the problem size and can tailor the number of nodes to achieve better granularity. The *input* and *output* keywords specify the data objects produced and consumed by each task. The *command* keyword is used to pass command line arguments to the task. The *hints* keyword is used to alert the execution process to attributes of the task which should be considered for scheduling purposes.

Task Descriptions

task: LU_decompisition
executables: SUN: LU_decompisition.sun
SP2: LU_decompisition.sp2
ALPHA: LU_decompisition.alpha
nodes: SUN: 4
SP2: 4
ALPHA: 4

task: inverse_matrix
executables: SUN: inverse_matrix.sun
SP2: inverse_matrix.sp2
ALPHA: inverse_matrix.alpha
nodes: SUN: 4
SP2: 4
ALPHA: 4

task: matrix_multiply
executables: SUN: matrix_multiply.sun
SP2: matrix_multiply.sp2
ALPHA: matrix_multiply.alpha
nodes: SUN: 4
SP2: 4
ALPHA: 4

Figure 4: Task Descriptions of Linear Algebra Operations

An application description consists of a number of *subtask* specifications. Each subtask refers to a task description which implements the desired function. The *nodes, command, input, output* and *hints* keywords in the subtask specification override the values given in the task description. The task implementor can place common default values in the task description which can be altered as needed in particular application descriptions.

The **synchronization** directive of an application subtask provides a mechanism for describing a variety of control sequences. For instance, tasks can be initiated after other specified tasks have begun or after other tasks have completed. Specifying that one task is to start only after others has started allows for server tasks to be initiated before their clients proceed.

The application description framework allows application structure to be expressed in application and task description files rather than in the application source code. Intertwining these operations with the application's computations and I/O can lead to excessive code complexity and hamper code reusability.

4 Scheduling Considerations

Scheduling methodologies for the the VCE are an area of active research. Two protocols being investigated are: 1) *Redundant Execution* where each task is run on several available computers (up to a certain maximum) and the result produced by the fastest instance of the task is returned

Application Description

subtask: LU_decompose
task: LU_decompisition
input: A.matrix
output: L.matrix , U.matrix
command: A.matrix L.matrix U.matrix

subtask: inverse_matrix_1
task: inverse_matrix
input: L.matrix
output: L^{-1} .matrix
command: L.matrix L^{1} .matrix
synchronization: after LU_decompse completes

subtask: inverse_matrix_2
task: inverse_matrix
input: U.matrix
output: U^{-1}.matrix
command: U.matrix U^{-1}.matrix
synchronization: after LU_decompse completes

subtask: first_matrix_multiply
task: matrix_multiply
input: L^{-1} .matrix , B.vector
output: temp.vector
command: L.matrix B.vector temp.vector
synchronization: after inverse_matrix_1 completes

subtask: second_matrix_multiply
task: matrix_multiply
input: U^{-1}.matrix , temp.vector
output: X.vector
command: U.matrix temp.vector X.vector
synchronization: after first_matrix_multiply completes
after inverse_matrix_2 completes

Application Graph

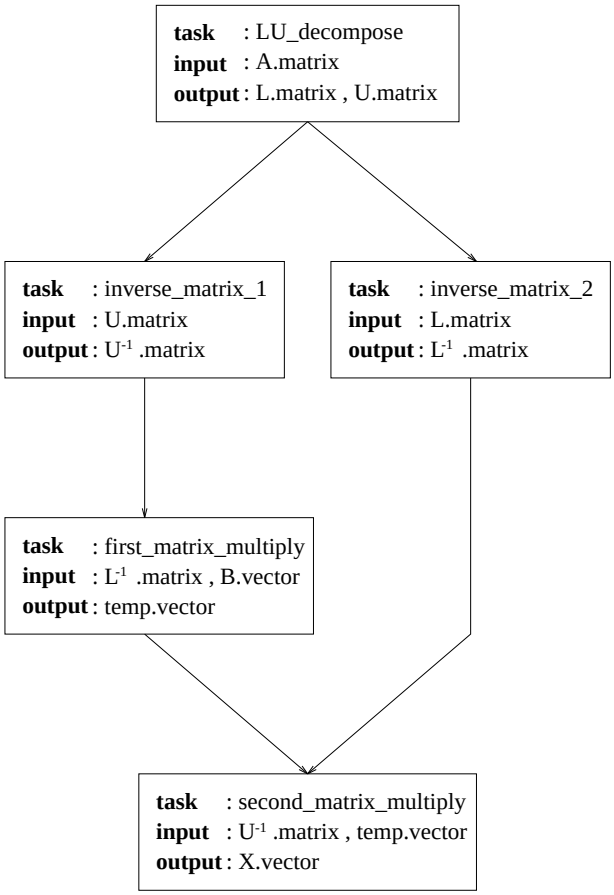

Figure 5: Application Description of a Linear Equation Solver

to the user or made available for subsequent computations; and 2) *Selective Initial Placement* where the load on each machine is monitored and one instance of each task is placed on the least loaded machine.

4.1 Redundant Task Scheduling

Two different approaches to redundancy are used in the VCE: 1) *Explicit Redundancy* where the user supplies executable files for the same task to run on multiple platforms which can be dispatched simultaneously; and 2) *Transparent Redundancy* where one of the executable files supplied by the user is dispatched on two or more members of a single group.

Transparent redundancy is used when two or more members of a group are lightly loaded. Under such conditions transparent redundancy further improves the fault tolerance and performance of the application and allows better hardware utilization. Transparent redundancy is also used extensively in workstation groups where each machine is intended primarily for the use of a single user (ie. its "owner"). Such machines are frequently idle or under utilized for long periods of time [8]. A VCE task will only dispatched on privately owned machine if the machine's average ready queue length is below 1.0. This indicates that the workstation has some idle capacity. The tasks are dispatched with the lowest possible priority so that if the locally originated load of the machine increases, interference from the VCE task will be negligible. By redundantly executing a single task on several workstations, the probability that at least one will deliver good performance is high.

Redundant task scheduling has several benefits. Fault tolerance is achieved because the failure of a single machine does not lead to failure of the entire application. In fact, redundancy eliminates the need for checkpointing and error recovery schemes which can complicate task implementation and reduce performance. Redundancy also improves performance by allowing the application to proceed at the rate of the fastest machine participating. These benefits are realized without the overhead of precise load measurement routines or process migration techniques.

4.2 Selective Initial Task Placement

A coarse grain load measurement algorithm has been implemented in the VCE. For supercomputers, the run queue length of each computing node is sampled periodically and the average run queue length is computed. Experiments are performed on each supercomputer to determine how high this number can get before users begin to notice degradations in performance. This load level is considered the machine's *nuisance threshold*. When the average run queue length is below one, indicating idle computing capacity in the machine, the machine is considered to be under the *adequate utilization threshold*. At any instant, a machine's load relative to these

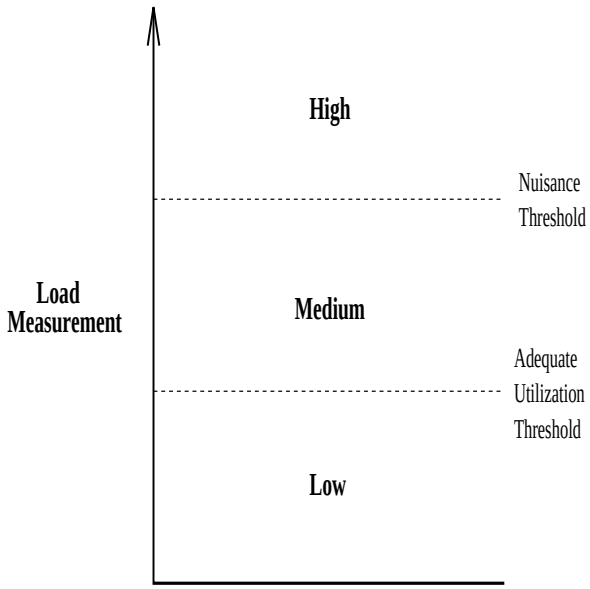

Figure 6: VCE Load Thresholds

```
do forever
    sleep 1 Sample_Interval
    compute Load(t_i)
    if Load(t_i) = Smoothed_load(t_{i-1})
     then continue
    if Load(t_i) > Smoothed_load(t_{i-k}) for k = 1...N
     then Smoothed_load(t_i) = Load(t_i)
    if Load(t_i) < Smoothed_load(t_{i-k}) for k = 1...M
     then Smoothed_load(t_i) = Load(t_i)
end
```

Figure 7: Load Measurement Algorithm

thresholds can be used to determine if the load is *low, medium* or *high* (Figure 6).

Experience in the VCE has shown that taking a single instantaneous reading of the load measure when a placement decision is to be made is problematic. Task initiation is delayed while the nodes of all candidate machines are polled, and transient spikes and dips in load condition can lead to poor placement decisions. To eleviate both these problems, one process on each supercomputer or workstation cluster periodically polls the nodes and computes a smoothed load average. This value can be queried by an execution process when a task placement decision needs to be made.

Figure 7 shows the algorithm used to calculate a machine's smoothed load measurement. An instantaneous load measure is computed periodically. If this instantaneous load measure is below or above the smoothed load measure for several consecutive intervals, the smoothed load measure is changed accordingly. The sampling interval, and the constants N and M, which determine how quickly the smoothed load measure reacts to changing load conditions, can be customized for each machine. Currently, the Unix *rstat()* function is used to gather load statistics. Because the *rstat()* values are updated by Unix every five seconds, six second sampling intervals are used. The constants N and M are set to 2 and 10, respectively. This causes the smoothed load measure to be increased quickly when load conditions are raised but to decrease slowly as the load falls. This bias is used to maximize the probability that machines reporting low load conditions are actually underutilized.

An example of how the algorithm behaves is shown in Figure 8. At time 0 both the instantaneous and smoothed load measures indicate the load on the machine is medium. The changes in the instantaneous load measure that occur between time 12 and time 30 are considered noise and are not reflected in the smoothed load measure. The ten consecutive low instantaneous readings beginning at time 36 cause the smoothed load measure to become high at time 90. The two consecutive instantaneous readings of medium at time 120 and 126 cause the smoothed load measure to return to medium.

When a task is eligible to be dispatched, the execution process obtains the current smoothed load measure of each candidate system. The task is dispatched on the least loaded platform. The order in which machines are listed in the task description is used to indicate the task implementors preference of execution platforms. If the lowest load measure is reported by two or more machines (that is, there is a tie), the precedence expressed in the task description is observed.

Time t	Instantaneous Load Measure $Load(t_i)$	Smoothed Load Measure $Smoothed\_load(t_i)$
0	Medium	Medium
6	Medium	Medium
12	High	Medium
18	Medium	Medium
24	Low	Medium
30	Medium	Medium
36	Low	Medium
42	Low	Medium
48	Low	Medium
54	Low	Medium
60	Low	Medium
66	Low	Medium
72	Low	Medium
78	Low	Medium
84	Low	Medium
90	Low	Low
96	Low	Low
102	Low	Low
108	Medium	Low
114	Low	Low
120	Medium	Low
126	Medium	Medium
132	Medium	Medium

Figure 8: Load Measurement Algorithm Example

5 Validation

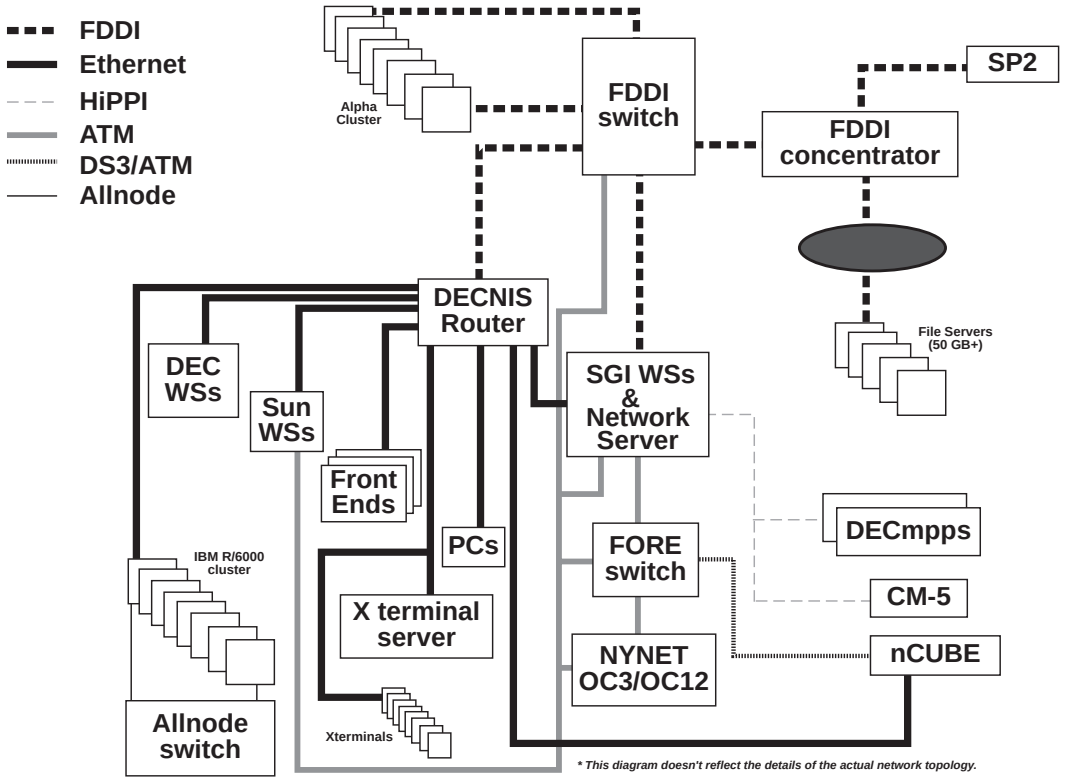

Figure 9: Computing Environment at NPAC

Several supercomputers and high performance workstation clusters at the Northeast Parallel Architectures Center have been included in a VCE prototype system. The NPAC environment consists of a wide set of *state-of-the-art* multi-computer systems (see Figure 9). The platforms used are briefly described below:

IBM SP-2: The SP-2 consists of a cluster of 16 RISC/6000 370 nodes interconnected by a crossbar switch (Allnode). Each node runs at a clock rate of 62.5 MHz.

ALPHA/FDDI: The ALPHA/FDDI configurations consists of 8 DEC ALPHA workstations interconnected by a high performance (100 Mbps) backbone composed of dedicated, switched FDDI segments. The ALPHA nodes have a clock rate of 150 MHz.

ATM LAN: This configuration consists of 2 SUN SPARCstation IPXs and 3 SGI workstation communicating over an ATM Local Area Network using ATM FORE ASX-100 switch. SUN IPX nodes operate on an approximately 40 MHz clock. Host computers are connected to the ATM

switch through Fore's 200 series adaptors. Each adaptor has a dedicated Intel i960 processor (running at 25 MHz) to support segmentation and reassemble functions and to manage data transfer between the adaptor and the host computer. The 200 series adaptors also have special hardware for AAL CRC and special-purpose DMA hardware.

5.1 Applications

Three distributed applications have been implemented with the VCE prototype and are being used to test the runtime support.

5.1.1 Linear Equation Solver

A linear equation solver constructed from a set of general purpose linear algebra tasks (shown in Figure 5) was discussed in Section 3.

5.1.2 Multi-target Tracking System

The Multi-target tracking system demonstrates the multi target tracking capabilities that is required by a Battle Management Command Control and Communication System [4]. The multi target tracker, shown in Figure 10, is designed to provide an estimation of launch vehicle parameters for individual targets/missiles in multi-target scenarios. The tracker receives input in terms of launch sites for missiles or targets. The launch sites are specified in terms of latitudes and longitudes. This information is fed to two focal plane tracking (FPT) modules (2 dimensional tracking) at 5 second intervals. The focal plane tracking modules process this data using kinematic filtering algorithms and track pruning and prediction algorithms. The output of this module is an initial prediction of trajectories of launched missiles. This data is then fed to a three dimensional tracking system which uses the data from the two focal plane tracking modules to prune duplicate tracks (if any), extend existing tracks, prune bad tracks and initiate new tracks. The output of the system is a list of target trajectories.

5.1.3 JPEG Compression/Decompression

JPEG (Joint Photographic Experts Group) is an emerging standard for image compression. JPEG standard aims to be generic and can support a wide variety of applications for continuous-tone images. We have used data parallel programming model to implement a distributed JPEG algorithm on a cluster of workstations. In this implementation half of the computers participate in compression of an image file while the second half reconstruct the compressed image. The image to be compressed is divided into N/2 equal parts (where N denotes the number of processors)

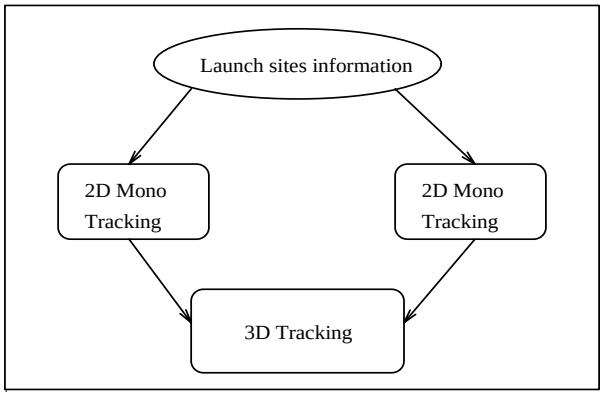

Figure 10: Multi-Target Tracker System

by the master process and then shipped to one half of the processors. Each processor performs the sequential JPEG compression algorithm on its portion of the image. After compression the processors send the compressed image to another set of N/2 processors which perform the decompression. Once decompression is done, the results are sent back to the master process which combines them into one image. Consequently, this algorithm involves five stages *viz.* distribution of uncompressed image , compression of the image, transmission of compressed image, decompression of the image, and finally combining the decompressed images.

5.2 Performance

Of the three VCE applications described in in Section 5.1 the linear equation solver is most representitive of the kind of application that will benefit from development and excution in the VCE. It is constructed from tasks which could be reused in other applications and for large data sets it has a long running time. So the developer benefits from the application description framework and the user benefits from the automatic runtime scheduling and dispatching capabilities.

A series of performance tests using the linear equation solver application were performed to determine under what conditions running an application in the VCE with the scheduling protocols described in Section 4 would be better than running it in a single computer. The IBM SP2 and the DEC ALPHA workstation cluster have similar performance characteristics [3], so often a user who has an application that could run on either platform would simply choose one at random. Observations in [3] suggest that even a moderate difference in load conditions at runtime can effect performance enough to impose a considerable penalty for choosing the wrong machine. Performance experiments done with the VCE support these findings and show that both the scheduling protocols described in Section 4 will reduce turnaround times in many situations.

In order to obtain controlled testing conditions, testing was performed on the DEC ALPHA cluster and the IBM SP2 when transient load conditions were low. A load generating utility

Table 1: Performance of the Linear Equation Solver Application

	Load	Execution time in seconds
SP2	Moderate	1153
	Low	849
ALPHA	Moderate	1440
	Low	732
Reduntant Scheduling	SP2-Moderate ALPHA-Low	741
	SP2-Low ALPHA-Moderate	868
Selective Scheduling	SP2-Moderate ALPHA-Low	744
	SP2-Low ALPHA-Moderate	886

was used to slightly elevate the load on one or the other as the tests were performed. The load generator allows load conditions to be varied between *low* and *moderate*. It places two dummy processes on each node in the configuration for a randomly determined period to simulate an interval of high activity. These *busy* periods are alternated with randomly determined *idle* periods where no artificial load is placed on the machine. When *low* load is specified, the busy periods vary from four to eight minutes while the idle periods are between 20 and 30 minutes. To simulate *moderate* load, idle periods are between four and eight minutes and busy periods are from 20 to 30 minutes.

The results of performance tests run using the linear equation solver application to solve a system of 1024 equations are shown in Table 1. The table shows the degradation which results when just two other processes per node are competing for computing resources. Neither redundant nor selective scheduling will have a significant impact on performance when the loads on all the candidate machines are roughly equivalent. That is, if all machines are heavily loaded, or all machines are lightly loaded, there is little a scheduling system can do to improve performance. However, when some candidate machines are less active than others, both the scheduling approaches studied tend to prevent the performance loss that could be encountered with a random placement.

Redundant scheduling yields slightly better performance than selective scheduling. If there is a significant load change after a task is placed with selective scheduling performance will degrade. Redundant scheduling insures that the best possible performance will be obtained because all

candidate machines run the tasks and the result produced by the fastest machine is returned to the user or made available for subsequent computations. Of course, redundant scheduling incurs higher costs for the application because the computations performed by some machines are discarded. These costs are somewhat offset by eliminating the overhead required for gathering load measurements.

Both of the scheduling protocols insulate the user from the effects of poor random placements without imposing the considerable overheads associated with process migration or checkpointing techniques.

In production Virtual Computing Environments it is expected that the choice between redundant and selective task scheduling would be based on the nature and frequency of VCE jobs to be executed. If VCE applications are submitted only occasionally, or tend to be of short duration then the use of redundant scheduling would be preferred. If many large VCE jobs are to be executed then selective scheduling would provide lower costs and preserve improved performance.

6 Future Work

The VCE will provide a testbed for investigating many issues relating to the efficient use of heterogeneous distributed supercomputing environments. Facilities to allow several geographically remote supercomputers connected to an ATM wide area network to participate in the VCE are already in place. Extensions to the scheduling and dispatching capabilites of the VCE execution environment to exploit these machines are being implemented.

Further research on scheduling protocols is underway. Methods for effectively determining when lightly loaded remote machines should be used rather than heavily loaded local ones are being developed. Issues of load measurement and redundant scheduling across a wide area network are being explored.

Design of an intertask communication system which will facilitate communication among redundantly scheduled components of an application has begun. Current applications use the P4 system [2] for ommunication. A more robust system, which will allow communication requirements to be specified in task and application descriptions, will ease application design and implementation.

Other aspects of the application description framework will also undergo further development. The syntax will be expanded to permit conditional and iterative task execution. A graphical user interface which will insulate the user from the details of the application description syntax will be developed.

References

[1] Barbacci, M. R., Weinstock, C. B., Doubleday, D. L., Gardner, M. J., and Lichota, R. W., *Durra: a structure description language for developing distributed applications,* Software Engineering Journal, pp. 83-94, March 1993.

[2] Butler, R., and Lusk, E., *User's Guide to the p4 Programming System*, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439-4801

[3] Hariri, S., Park, S. -Y., Reddy, R., Subramanyan, M., Yadav, R., Fox, G., Parashar, M., *Software Tool Evaluation Methodology,* to appear in The Proceedings of the Fifteenth International Conference on Distributed Computing Systems, May 1995.

[4] Hariri, S., Yadav, R., Thiagarajan, B., Park, S. Y., Subramanyan, M., and Reddy, R., "A Concurrent Multi Target Tracker: Benchmarking and Portability", International Conference on Parallel Processing (ICPP), August 1994.

[5] Kunz, T., *The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme,* IEEE Transactions on Software Engineering, Vol. 17, No. 7, pp. 725-730, July 1991.

[6] Litzkow, M. and Solomon, M., *Supporting checkpointing and process migration outside the UNIX kernel,* Proceedings of the Winter 1992 USENIX Conference, pp. 283-290, 1991.

[7] Mechoso, C. R., Farrara, J. D., Spahr, J. A., *Running a Climate Model in a Heterogeneous, Distributed Computer Environment,* Proceedings of the Third IEEE International Symposium on High Performance Distributed Computing, pp. 79-84, August 1994.

[8] Mutka, M. W., and Livny, M., *The Available Capacity of a Privately Owned Workstation Environment,* Performance Evaluation, pp. 269-284, Vol. 12, No. 4, July 1991.

[9] Rousselle, P., Tymann, P., Hariri, S., and Fox, G., *The Virtual Computing Environment,* Proceedings of the Third IEEE International Symposium on High Performance Distributed Computing, pp. 7-14, August 1994.

[10] Zhou, S., Zheng, X., Wang, J. and Delisle, P., *Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems,* Software - Practice and Experience, pp. 1305-1336, Vol. 23, No. 12, December 1993.