The Virtual Computing Environment

Philip Rousselle
Syracuse University

Paul Tymann
Syracuse University

Salim Hariri
Syracuse University

Geoffrey C. Fox
Syracuse University

The Virtual Computing Environment

Philip Rousselle, Paul Tymann, Salim Hariri, and Geoffrey Fox

Northeast Parallel Architectures Center
Department of Electrical and Computer Engineering
Syracuse University
Syracuse, NY 13244-4100

Abstract

A network of supercomputers and high-performance workstations appears to be the only reasonable way to provide adequate computing resources for the Grand Challenge problems of the next century. Such a collection of computers and supporting software environments is called a virtual computing environment (VCE). This paper describes the motivation and goals of the VCE project, followed by a description of the system. The paper concentrates on the runtime aspects of the VCE, and concludes with a discussion of a small prototype system that has been built using the Isis distributed toolkit.

1 Why Build A Virtual Computer?

Advances in software technology have produced tools and environments for the development of parallel and distributed applications. These advances have resulted in the proliferation of a large number of different architectural classes like SIMD computers, MIMD computers, vector computers, etc., where each class represents a set of different trade-offs in design decisions. Each architectural class is tuned to deliver maximum performance to a specific set of applications, however, none of the existing computer systems are general enough to address all classes of applications and provide the desired performance levels.

Applications that require the resources of a supercomputer are quickly reaching the point where it will be difficult to find a single computing platform on which they may be executed. The grand challenge problems of 1995 and beyond will require a memory capacity of 1000GB and system speeds of 1 Tflop [6]. Given the current pace of technological advancement it is unlikely that a single supercomputer will be able to provide the resources required for these problems. A network of supercomputers and high performance workstations appears to be the only reasonable way to provide adequate computing resources for these large applications.

Programmers developing parallel applications are faced with the daunting task of selecting the correct architecture, finding the appropriate software tools, and must handle many low-level issues such as load balancing. The situation today is similar to that of the pre-compiler era of computing. Before the first FORTRAN compilers were available, programmers could find the tools required to build their applications, however they were forced to program in assembly language and had to deal with low-level machine issues in addition to the task of developing their programs. The first high-level languages made programming easier by providing an abstract model of computing which hid the details of the low-level machine and enabled the programmer to concentrate on the application. The same type of software support is required for parallel and distributed computing.

2 Design Philosophy

The main design philosophy being used in creating the VCE is to utilize existing programming paradigms, programming languages, computer architectures, and physical machines, to provide high performance computing for a variety of applications. By using existing technology whenever possible system designers will be able to focus on providing a new computing environment rather than reimplementing basic functions. Running existing tools and system software will eliminate compatibility problems and will make it easier to use existing programs as modules in a virtual computer application.

This approach differs from previous distributed operating system projects which saught to address similar issues. While implementing distributed processing functions like communication protocols and file systems at the kernel level improves performance, the distributed operating system approach also has drawbacks. Developing any operating system is a complex task. The resources required to design, implement, deploy and maintain a new operating system are considerable. Compatibiliity with existing systems and portability over a range of hardware platforms are difficult to acheive. Since facilities like LANs and distributed file systems are becoming commonplace the design philosophy of the VCE will be to exploit these technologies fully.

3 The Virtual Computing Environment

3.1 Overview

The VCE will allow users to develop and run applications that require more resources than can be reasonably provided by a single computer. A VCE application is broken down into functional components called *tasks*, which are represented visually using a *task graph*. For every task in the application, there is a corresponding node in the task graph. The task graph defines the input, output, and function of each task. The nodes in the task graph are connected by arcs which define the communication and synchronization relationships among the tasks.

The VCE will use the task graph to manage the development, compilation, and execution of an application. An application begins as a simple task graph. The programmer uses the software development tools provided by the VCE to annotate the task graph so that it contains information regarding the type of computation, and the high-level source code required by the computation. When a VCE application is run the task graph will be used to select the machines on which the application will run, prepare binaries, and manage execution.

At the highest level, the VCE consists of two major components: the Software Development Module (SDM), and the Execution Module (EXM). Each module, in turn, is broken down into several layers as shown in Figure 1.

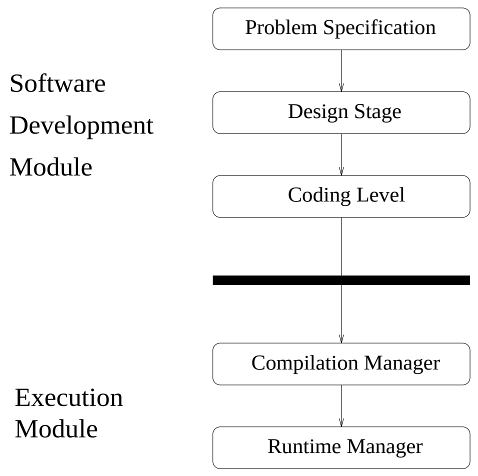

Figure 1: The Virtual Computing Environment

3.1.1 The Software Development Module

The SDM provides the tools required to produce a high-level description of a distributed application. The primary purpose of this module is to develop, test and evaluate the performance of the application. The SDM consists of three layers, each of which is responsible for attaching specific information to the task graph. The information contained in the completed task graph will include: Implementation language, Input requirements, Hardware requirements, User supplied information, and Outputs.

The *problem specification* layer is responsible for extracting the requirements of the problem to be solved and formalizing its functional flow. This is accomplished by creating the initial task graph.

The *design stage* is responsible for analyzing the computational needs and the existing dependencies for each task in the task graph. The analysis performed by the design stage is based on Fox's [3] work on the architecture of problems and portable parallel software systems. The parallel software design methodology used in the design stage concentrates on the architecture of the problem and not the machine. The machine architectural features will be captured by other lower stages. There are three broad classes of problem architectures: synchronous, loosely synchronous, and asynchronous, which describe the temporal (time or synchronization) structure of the problem. The temporal structure of the problem is analogous to the hardware classification into SIMD and MIMD. Other classes that capture the nature of the task, such as graphic or interactive, will be used to assist the lower layers in determining the best way to map tasks onto available machines.

The graph produced by the design stage feeds into the *coding level*. In this stage, the application is parallelized using architecture independent languages that allow tasks to be implemented using functional, data parallelism or both. The software tools and languages to code and parallelize the application at this level will be based on emerging standards (High Performance Fortran [4], High Performance C++ [1], etc.). Support for architecture independent communication between tasks will be provided via standard communication libraries (based on standards such as MPI [5, 14]) and the semantics of object-oriented method invocation.

As the task graph is annotated in the various layers of the SDM, the user will be able to record information about the nature of the application that will be useful to the compilation and runtime managers. These *hints* will allow the execution module to do extra optimization. For instance, suppose a particular application has three functionally parallel modules and the user expects one to run much longer than the combined running times of the other two. If the system is aware of this, dispatching of the longer job can be given higher priority so opportunities for parallel execution will be maximized.

3.1.2 The Execution Module

The primary purpose of the EXM module is to run VCE applications. As in most systems this consists of two distinct steps: preparing executables, and managing the execution of the application. The EXM runtime manager is a distributed application whose components are running on each of the machines in the VCE network. These components accept both local and remote execution requests from VCE users.

The *compilation manager* will be responsible for preparing the executable code for each component of the application. The compilation manager will use standard compilers to generate machine code, and maps the architecture independent computation and communication requirements of VCE tasks to machines that are actually available in the VCE network. Through the use of a simple database, maintained by VCE software, the compilation manager determines which are the best machines on which to run each task. The appropriate compilers are then invoked to prepare executables for each of the selected machines.

The *runtime manager* will be responsible for managing the execution of a VCE application. The basic service provided by this level is selecting the "best" machines on which to run the various tasks, loading the corresponding binaries, and initiating execution. The selection of target machines is based on current load and availability. While the application is running this layer will migrate tasks to less loaded machines, and provide fault tolerance, if required or requested by the user.

4 Runtime Support

The execution module uses the task graph produced by the software design module to run a virtual application. Several important issues need to be addressed by the runtime system, some of these issues are discussed below.

4.1 Compilation Issues

Part of the power provided by the VCE lies in its ability to dynamically select machines on which to run tasks, the ability to migrate processes from one architecture to another, and support for fault tolerance. Most systems that provide similar features do so for homogeneous environments, the VCE is designed to use heterogeneous platforms which makes these features harder to support. A single VCE task may be able to be executed on several different hardware platforms, each requiring different binaries.

The compilation manager must select the machine, or machines, on which each task should be run. Once the selection has been made the standard compilation tools on the target machine will be used to create the executable version of the task. At this level all the machines participating in the VCE are divided into classes. These classes are the low-level counterparts of the problem architecture classes used by the design stage. For example, the synchronous class of problems maps easily to most SIMD style machine. So a possible machine class might be SIMD which would contain machines like the CM5 and the MasPar MP-1.

There is no guarantee that every task will map to exactly one machine. In fact in most cases several different machines may be used to execute a particular task. In this case the compilation manager prepares executable images for all possible machines. The choice of which machine will actually be used will be made by the runtime manager. By preparing all possible executables before an application is actually run, the runtime manager will be able to move a given task among various machine architectures without the need to compile a task while the application is running.

4.2 Communication

Communication between tasks will take place either through primitives defined in the MPI or via object-oriented method invocation semantics. The compilation manager will provide a number of different libraries that will map MPI to communication tools available in the system. In addition to providing a map between the abstract and low-level communications primitives, these libraries will provide the runtime manager with the ability to monitor, redirect, and move connections between tasks.

A channel is used to connect VCE tasks. A channel is a logical transport medium that connects possibly many tasks sending and receiving messages. Channels are distinct from the tasks that are connected to them, and thus readily support messaging directed to groups and/or single tasks without requiring that clients use different forms of message addressing in the two cases. In channel-based systems, clients may be unaware of whether messages are being received by groups or individuals. The runtime system may split channels, interposing other tasks between senders and receivers to deal with issues such as authentication or data conversion. Channels will be connected to tasks through *ports*. The runtime system will be responsible for the creation, placement, and destruction of ports.

Applications designed using the object-oriented paradigm can be viewed as collections of independent entities called objects. Every object provides one or more services, *methods*, that can be requested by a client. At an abstract level communication between objects is done by message passing. A client prepares a message that identifies the service required, along with the appropriate parameters, and sends the message to the server object. Upon receipt the server processes the request and sends a reply back to the client. A method definition defines the interface used between the two objects.

In systems where objects reside on a single processor, communication is typically implemented using standard procedure calling conventions. For objects residing on different processors, the communication between objects could be implemented using message passing. In this case the client object and a *server proxy* would be placed on one processor, and the server object and a *client proxy* on the other. The role of the proxy is to receive messages, translate information into architecture independent form, and forward the result to the corresponding proxy on the other processor, as shown in Figure 2.

Proxies will be generated by the compilation manger when needed, using a tool such as the IDL compiler provided by the Object Management Group [11]. The generated code will use VCE channels to exchange information with proxies running on other machines. Since both styles of programming will use the VCE communication system, the runtime system will be able to migrate both types of processes, and communication between object-oriented and data parallel tasks will be possible.

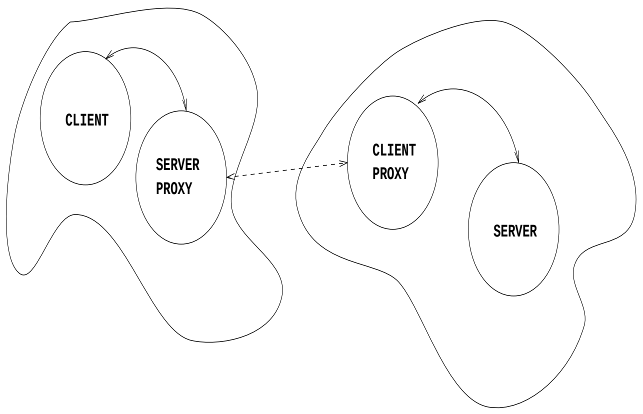

Figure 2: Communication via Proxies

4.3 Task Placement

The scheduling mechanisms used in the machine level will seek to achieve two, sometimes conflicting, goals:

- Schedule work on idle machines to maximize hardware utilization.
- Execute tasks on the best available platform. Where best available platform means the computer best equipped to satisfy the task's requirements (processor requirements, architecture requirements, file requirements, etc.)

The execution layer scheduling algorithms will tend to give preference to schedules that maximize overall resource utilization (and therefore maximize system throughput) rather than schedules that optimize the performance of any single job. For example, suppose the execution layer has two tasks. The first task can only run on a particular Unix workstation (call it machine A) because of that machine's architecture. The second task can run on any Unix workstation, but will run fastest on machine A. In this situation the execution layer should run the first task on machine A. Even if there are no other idle Unix workstations available the second job should be made to wait because it can be used to occupy a workstation if one becomes idle before the first job completes on machine A. An additional prioritization scheme will also be needed to prevent starvation of tasks. That is, as a task waits to be dispatched its priority will be increased to insure it will eventually be dispatched even if that results in a globally suboptimal schedule. Authorized users will be able to modify the priorities of particular applications.

Krueger [9] showed that processor utilization rates in a distributed system could be improved significantly through careful use of non-preemptive process placement techniques. Such techniques may not be adequate to support the virtual computing environment however. Virtual computing applications may provide enough tasks to consume all the excess capacity in a system. The philosophy of Clark [2], Ju [8], and Krueger [9], is to suspend (or drastically reduce the local dispatching priority of) remotely initiated tasks when resource requirements of locally initiated processes increase. Execution of remote tasks is resumed when activity of locally initiated tasks diminishes. This approach reduces the frequency of process migrations and yields high resource utilization without requiring an efficient process migration mechanism. This may not be the case in the virtual computing environment. If a virtual machine task is suspended to allow execution of local tasks, initiation of other tasks dependent on the output of the suspended task could be delayed. This ripple effect could adversely effect system throughput. Opportunities for increasing throughput could be missed if it is not possible to move a process from a less suitable machine to a more suitable machine when one becomes available.

4.4 Load Balancing

The ability of the execution layer to maximize resource utilization and system throughput will be restricted if task placement decisions must be made before task initiation and cannot be altered later. To achieve the system objectives will require both:

- Load balancing techniques that determine which tasks should be running on which machines.
- Remote execution mechanisms to dispatch and relocate tasks on the machines best able to accommodate them.

Many papers have been written on both these issues, but the value of some of the current research is undermined by its lack of generality. Many remote execution systems are built to support a particular load balancing philosophy [9, 10, 13]. They also tend to make restrictive assumptions about the homogeneity of the underlying network and the nature of the tasks being migrated. For instance, Litzkow [10] describes a system that can migrate long running, compute intensive, non-communicating jobs across a homogeneous network. Such a system would not be robust enough to be provide the functionality required by the execution layer.

The literature on load balancing does not address the remote execution problem. Load balancing algorithms are often validated on homogeneous systems using tasks that are easily migrated (like parallel Monte Carlo simulations [13] or big batch jobs [10]). Load balancing algorithms are rarely tested on the kind of heterogeneous networks running heterogeneous applications that are common in high performance distributed computing (and that will typify the execution layer environment).

To provide the most robust possible execution environment and provide the greatest possible flexibility for whatever workload balancing scheme is to be employed, the execution layer should implement a variety of process migration schemes. The following list outlines four process migration approaches that will be considered for implementation in later versions of the VCE prototype.

- **Process migration through redundant execution.** Dispatch the same task on several idle machines. If one of those machines gets busy with other work then kill the incarnation of the redundant task on that machine. This achieves process migration with low overhead because killing a task and using an already running redundant copy avoids the communication overhead of moving a process and its state information over the network.
- **Process migration through checkpointing.** Migratable jobs checkpoint regularly. To migrate a job kill it and start it somewhere else by instantiating the new incarnation from the checkpoint record. This is expensive and may require the cooperation of the task involved.
- **Process migration the old-fashioned way.** To migrate a job we dump the contents of the address space, copy it to a new machine and restart it. This has many drawbacks, one being that it requires homogeneity [10].
- **Process migration through recompilation.** This is very expensive but may be very robust. It is only discussed in one paper [12] and may be difficult to implement.

The execution layer should have several of these techniques in its repertoire. Which of these will be used for any particular migration will depend on the state of the system and the characteristics of the task(s) involved.

4.5 Processor Utilization

Two related concepts that will be employed by the execution layer to achieve high processor utilization are *free parallelism* and *anticipatory processing.*

Free parallelism is based on the observation that when parallel processes are running on otherwise idle machines, efficiency is not a relevant measure of parallel performance, only speed-up needs to be considered. If 100 idle machines are available and the only way to use them is to distribute a single application over all 100 machines to realize a 10% speed-up, it is still worth doing because the 10% speed-up comes for "free". Many distributed computing environments contain significant numbers of idle or nearly idle computers most of the time. In these environments the execution layer should make every possible effort to get idle processors doing useful work.

Anticipatory processing is a method for using idle processors to increase system throughput even when there are no dispatchable VCE tasks available to exploit them. It involves using idle workstations to perform processing that may or may not be required in the future. For example, suppose there is a VCE application consisting of two modules where the second cannot start until the first completes. If there are lots of idle resources in the network they can be used to do things that may help the second module run faster when it is ready to go. If the second module is a C program we could compile it on one machine of each different architecture in the network so that, at run time, we will have our choice of where to dispatch it (*anticipatory compilation*). If the second module requires input files other than the ones produced by its predecessor module, we could use idle resources to replicate those files at many sites that may be candidates to host the second module when it becomes dispatchable.

5 Implementation

Machines with a wide variety of capabilities will participate in the Virtual Computing Environment. A VCE daemon running on each machine contributes to global scheduling and remote execution functions. All of the machines participating in the VCE will be divided into groups, where the members of the group share similar architectural features. In a typical heterogeneous environment there might be a MIMD group, a SIMD group and a workstation group. One machine within the group acts as group leader and coordinates the activities of the group. Fault-tolerance of the group leader will be achieved through redundancy and error recovery mechanisms.

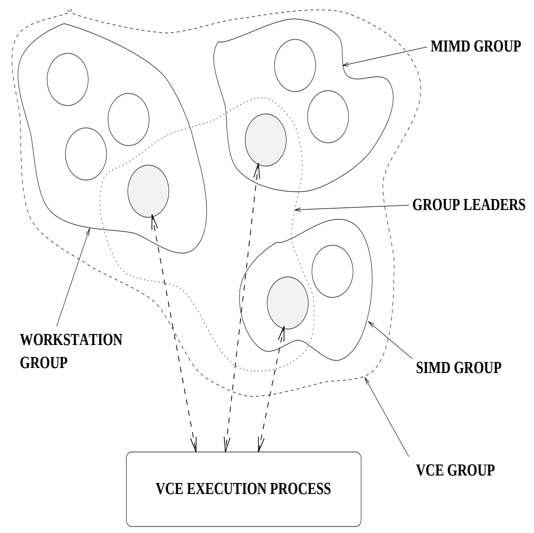

Figure 3: Runtime Bidding Mechanism

Execution of a VCE application will utilize an execution program running on the user's workstation. After identifying the groups that contain the types of machines required to run the application, the execution program sends a request message to each group leader (see Figure 3). The request message contains a list of the resources required from each group for a given VCE application. Once the request is received by the group leader, it is sent to each machine in the group. Each machine, based on current load and availability, sends a "bid" back to the group leader indicating that it can satisfy part of the request. Each bid includes the current load of the bidding machine. The group leader collects the bids, determines which are the "best" processors to allocate to the application, and then sends a reply back to the execution program. If there are insufficient resources within a group a message to that effect is returned to the execution program.

Once the bidding process has been completed, the execution program sends to the selected machines the programs and data files that make up the application and specifies the required communication patterns.

Each group leader and its members, might have several VCE applications underway at any one time. Each group manages its workload employing the load balancing, process migration and remote execution techniques mentioned earlier.

In the current implementation of the VCE these groups are object-code compatible and applications are described at runtime in terms of object (rather than source) modules. A small prototype of the remote scheduling/dispatching facility has been completed. The user can present the system with a description of a distributed application and the system will make a placement decision for each constituent module and arrange for its execution. The input to the system is a script:

```
ASYNC       2 "/apps/snow/collector.vce"
WORKSTATION 1 "/apps/snow/usercollect.vce"
SYNC        1 "/apps/snow/predictor.vce"
LOCAL         "/apps/snow/display.vce"
```

which describes a distributed application. The script shown above corresponds to a weather forecasting application. The first line of the script requests two instantiations of a data collector program on machines with asynchronous architectures. The third line requests remote execution of a predictor program on a synchronous computer. The LOCAL directive identifies a program to run on the local workstation after the remote executions have begun.

As VCE development proceeds, the vocabulary supported in the application description will become more powerful. For instance constructs like "ASYNC 5-" to indicate five or less remote instances are required, "SYNC 5,10" to indicate between five and 10 remote instances and so on. Conditional statements and statements describing the communication requirements of the application will also be added. Ultimately, a sophisticated job control language derived from these preliminary dispatching directives will be used by the upper levels of the VCE to communicate runtime dispatching requirements to the lower levels.

The prototype scheduler/dispatcher has been constructed using the Isis Distributed Toolkit [7]. It consists of two programs: a scheduling/dispatching daemon that runs in each workstation authorized to host remote executions, and an execution program that executes applications on behalf of a local user.

The prototype runs on a homogeneous workstation LAN and implements the workstation group pictured in Figure 3. The scheduling/dispatching daemons are organized as an Isis process group. The first instance of the scheduler/dispatcher program to come on-line assumes the role of group leader and coordinates the activities of the group. Isis provides error notification functions which are used to allow the oldest surviving member of the group to assume the role of group leader in case the group leader fails. Machines can enter or leave the group at any time.

When the prototype encounters a WORKSTATION directive in an application description it sends its machine requirements to the scheduling/dispatching daemons. The group leader fields this request and translates it into a broadcast to all the scheduling/dispatching daemons to disclose their state. Any daemon that is not already excessively loaded and can run remote jobs sends its load description to the group leader. If the group leader receives fewer responses than needed a failure indication is sent to the execution program, otherwise the group leader sends the execution program a list of the Isis addresses of the least loaded processors available for remote execution. The execution program then sends a path specification of the program to be executed to each daemon on the list. When an application terminates, the execution program notifies all machines working on the application to terminate.

C style descriptions of the execute and group leader programs are given below:

```
void execute(void) {
  openExecutionScriptForReading();
  while(!eof(ScriptFile)) {
    readLineFromScriptFile();
    SendRequestToSpecifiedGroup();
    ReceiveReply();
    if (AllocError()) Terminate();
  }

  for(i=0; i<NUMGROUPS; i++)
    SendExecutionInfoToGroup();
  StartExecution();
  WaitForApplicationTermination();
  SendTerminateMessage();
}

void groupLeader(void) {
  struct RequestMsg request;
  struct BidReply bids[NUMINGRP];
  int reps=0;

  while(1) {
    receiveRequest(&request);
    bcastRequestToGroup(&request);
    for (reps=0; reps<NUMINGRP; reps++)
      insertReplyIntoList();
    sortBidsByLoad();

    if (reps<request.numNeeded)
      returnAllocError();
    else
      returnBids(bids);
  }
}
```

The prototype uses Isis *bcast* and *reply* primitives for communication between the execution program, group leaders, and group members. If several execution programs have requests outstanding at the same time, Isis will construct different threads for each request. In the current prototype of the scheduler/dispatcher, applications effect communication themselves (using Isis or PVM for instance). Later, an MPI library will be added and requisite communication support will be provided in the application description semantics to allow the VCE to perform communication redirection as needed for redundant execution, process migration and checkpointing.

While the overall design of the initial scheduler/dispatcher has been established, some points (like techniques for providing security and isolating the system from application errors) have not been finalized. All design issues are being investigated thoroughly to insure that the prototype scheduler/dispatcher will be robust enough to serve as the foundation of the VCE runtime environment.

6 Future Work

Future development of the VCE will focus on increasing the number of architectures supported and exploring advanced heterogeneous process migration techniques which will enhance the system's load balancing and fault tolerance capabilities.

References

[1] Cheng, D.Y., *A Survey of Parallel Programming Languages and Tools*, RND-93-005, NASA Ames Research Center, March 1993.

[2] Clark, H. and McMillin, B., *DAWGS a distributed compute server utilizing idle workstations,* Journal of Parallel and Distributed Computing, pp. 175-186, Vol.14, No.2, February 1992.

[3] Fox, G.C., *The Architecture of Problems and Portable Parallel Software Systems,* Northeast Parallel Architectures Center, Syracuse University.

[4] Fox, G.C., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C., and Wu, M., *Fortran D Language Specifications*, Northeast Parallel Architectures Center, Syracuse University, 1990.

[5] Hariri, S., Park, J., Yu, F., Parashar, M., and Fox, G., *A Message Passing Interface for Parallel and Distributed Computing*, Proceedings of the Second International Symposium on High Performance Distributed Computing, pp. 84-91, July 1993.

[6] Hwang, K., and Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw-Hill, Second Edition.

[7] Isis Distributed Systems Inc., *The Isis Distributed Toolkit Version 3.0 User Reference Manual*, Ithaca NY, 1992.

[8] Ju, J., Xu, G., and Tao, J., *Parallel Computing Using Idle Workstations,* Operating Systems Review, pp. 87-96, Vol. 27, No. 3, July 1993.

[9] Krueger, P., and Cawla, R., *The Stealth Distributed Scheduler,* Proceedings of the 11th IEEE International Conference on Distributed Computing Systems, May 1991.

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

[11] The Object Management Group, *The Common Object Request Broker: Architecutre and Specification*, OMG Document Number 93.12.43, December 1993.

[12] Theimer, M., and Hayes, B., *Heterogeneous Process Migration by Recompilation*, Proceedings of the 11th IEEE International Conference on Distributed Computing Systems, pp. 18-25, May 1991.

[13] Waldspurger, C.A., Hogg, T., Huberman, B.A., Kephart, J.O., and Stornetta, W.S., *Spawn: a distributed computational economy,* IEEE Transactions on Software Engineering, pp. 103-117, Vol.18, No.2, 1992.

[14] Walker, D., Standards for message passing in a distributed memory environment. *ORNL/TM-12147*, August 1992.