1998

The Design and Evaluation of a Virtual Distributed Computing Environment

Haluk Topcuoglu
Syracuse University

Salim Hariri
Syracuse University

Dongmin Kim
Syracuse University

Yoonhee Kim
Syracuse University

Xue Bing
Syracuse University

Baltzer Journals

The Design and Evaluation of a Virtual Distributed Computing Environment*

Haluk Topcuoglu[1] Salim Hariri[1] Dongmin Kim[1] Yoonhee Kim[1] Xue Bing[1]
Baoqing Ye[1] Ilkyeun Ra[1] Jon Valente[2]

[1]*Department of Electrical Engineering and Computer Science*
HPDC Laboratory
Syracuse University
Syracuse, NY 13244-4100
E-mail: {haluk, hariri}@cat.syr.edu

[2]*Rome Laboratory, Rome, NY 13441*

Current advances in high-speed networks such as ATM and fiber-optics, and software technologies such as the JAVA programming language and WWW tools, have made network-based computing a cost-effective, high-performance distributed computing environment. *Metacomputing*, a special subset of network-based computing, is a well-integrated execution environment derived by combining diverse and distributed resources such as MPPs, workstations, mass storage, and databases that show a heterogeneous nature in terms of hardware, software, and organization. In this paper we present the Virtual Distributed Computing Environment (VDCE), a metacomputing environment currently being developed at Syracuse University. VDCE provides an efficient web-based approach for developing, evaluating and visualizing large-scale distributed applications that are based on predefined task libraries on diverse platforms. The VDCE task libraries relieve end-users of tedious task implementations and also support reusability. The VDCE software architecture is described in terms of three modules: a) the Application Editor, a user-friendly application development environment that generates the Application Flow Graph (AFG) of an application; b) the Application Scheduler, which provides an efficient task-to-resource mapping of AFG; and c) the VDCE Runtime System, which is responsible for running and managing application execution and for monitoring the VDCE resources. We present experimental results of an application execution on the VDCE prototype for evaluating the performance of different machine and network configurations. We also show how VDCE can be used as a problem-solving environment on which large-scale, network-centric applications can be developed by a novice programmer rather than by an expert in low-level details of parallel programming languages.

Keywords: network computing; programming tools and environments; task scheduling.

*This research is supported by Rome Laboratory contract number F30602-95-C-0104.

1 Introduction

Grand challenge problems have computational and storage resource requirements that are beyond the capacities of a single computing environment. Additionally, emerging network technologies such as fiber-optic transmission facilities and the Asynchronous Transfer Mode (ATM) enable data to be transferred at the rate of a gigabit per second (Gbps). Since high-speed networks have become more common and provide low latency communication services that are close to those offered by massively parallel processors (MPPs), there is a growing interest in combining the computational and storage resources that are available over the wide area networks to build a new execution environment called *metacomputing* [14]. New software tools and techniques are required to utilize the metacomputing resources, which are not fully supported by existing parallel or distributed software. The heterogeneous and dynamic nature of a metacomputing environment limits the use of existing parallel computing tools; similarly, the existing distributed systems may not provide the high performance that is a key target in a metacomputing environment.

In this paper we present the design of the Virtual Distributed Computing Environment (VDCE) currently being developed at Syracuse University. VDCE [2, 3] provides an efficient mechanism to execute large-scale applications on distributed and diverse platforms. The main goal of the VDCE project is to develop an easy-to-use, integrated software development environment that provides software tools and middleware software to handle all the issues related to developing parallel and distributed applications, scheduling tasks onto the best available resources, and managing the Quality of Service (QoS) requirements.

VDCE is a three-tiered software architecture that consists of an Application Editor to assist in application development and specification, an Application Scheduler to perform transparent application scheduling and resource configuration, and a VDCE Runtime System to run and manage the application execution. The Application Editor is a web-based graphical user interface that helps users to develop parallel and distributed applications. In VDCE the application development process is based on a dataflow programming paradigm. The Application Editor generates its output in terms of an Application Flow Graph (AFG) in which the nodes represent task computations and links denote communication and/or synchronization among the nodes (tasks). The Application Editor provides menu-driven, functional building blocks of task libraries. A node of an AFG is a well-defined function or task selected from a given task library. VDCE provides a large set of task libraries grouped in terms of their functionality, such as matrix operations, Fourier analysis, C^3I (command, control, communication, and information) applications, etc.

VDCE provides a distributed runtime scheduler, the Application Scheduler, which provides efficient task-to-resource mapping of application flow graphs. The Application Scheduler uses performance prediction of individual tasks to achieve efficient resource allocations. The schedule decision is based on the task specifications (i.e., hardware/software requirements) in the application flow graph, locations and configurations of resources, and up-to-date resource loads. The VDCE Runtime System consists of two parts: the Control

Virtual Machine (CVM), and the Data Virtual Machine (DVM). The CVM is responsible for monitoring the VDCE resources, setting up the execution environment for a given application, monitoring the execution of the application tasks on the assigned computers, and maintaining the performance, fault tolerance, and quality of service (QoS) requirements. The DVM is responsible for providing low latency and high-speed communication and synchronization services for inter-task communications.

The rest of the paper is organized as follows. Section 2 is a summary of the related work. In Section 3 we present the design and implementation issues of the VDCE software architecture. Section 4 presents experimental results and evaluation of the current VDCE prototype. Concluding remarks and future work are given in Section 5.

2 Related Work

In this section we provide a review of related work on the software development process, followed by related work on metacomputing. The software development process of parallel and distributed applications can broadly be described in terms of three phases: a) application design and specification, b) application scheduling and resource configuration, and c) application execution and runtime.

In a well-integrated execution environment it is important to provide: a) an easy-to-use interactive user-interface to design and specify parallel distributed applications and, b) well-developed graphical utilities for visualization of results and program behavior. Generally, writing parallel/distributed programs overwhelms users due to the difficulty of explicitly expressing communication and synchronization among the computations [1]. A graph-based programming environment, in which a program is defined as a directed graph where nodes denote computations and links denote communication and synchronization between nodes, may be used to decrease the work of programmers. Currently, there are a few visual parallel programming languages and environments, such as Computationally Oriented Display Environment (Code) [4], Heterogeneous Network Computing Environment (HeNCE) [5], and Zoom [6]. To develop a Code or HeNCE application, a programmer first expresses the sequential computations in a standard language and then specifies how they are to be composed into a parallel program. Zoom is a hierarchical representation abstraction for describing heterogeneous applications. Zoom representation of an application can be translated into a HeNCE program for execution [5]. On the other hand, application development tools and environments are being modified to support web-based user interfaces, since the World Wide Web is becoming a low-cost, standard interface mechanism with which to access the computational resources that are distributed all over the world [22].

After a parallel/distributed application is developed, the tasks of the application are assigned to the available resources. In the literature, although the task scheduling (or resource allocation) problem has been investigated extensively, most of the algorithms and systems are valid only for specific architectures and/or applications. One of the few research groups targeted on a general scheduling framework is the APPLeS [7] group. AP-

PLeS proposes *application-level scheduling* in which everything about the system is evaluated in terms of its impact on the application. APPLeS develops a customized schedule for each application by including user-specific, application-specific, system-specific, and dynamic information in its scheduling decision. The Network Weather Service component provides dynamic information. The Heterogeneous Application Template provides specific information about the structure of the application. User-supplied information is entered into the system with a user-specification file. There are resource management systems to provide load sharing and resource allocation, one of which, developed at the University of Wisconsin, is the Condor [24] project, a distributed batch system for sharing the workload of compute-intensive jobs in a pool of UNIX workstations connected by a network.

The application execution and runtime phase executes the developed and configured application and produces the required output. This stage integrates the assigned resources that will be involved in execution and supports inter-module communications, which are based on either a message-passing tool such as PVM [16], P4 [18], MPI [17], and NCS [19] or on a distributed shared memory (DSM) model. During the execution of the application this stage accepts data from different computing elements and combines them for proper visualization. It intercepts the error messages generated and provides proper interpretation. Some of these message-passing tools may be used in a metacomputing environment, although they were initially developed for parallel and distributed applications. In the first I-WAY metacomputing testbed, Nexus and MPI communication libraries were used within the prototype implementations of Globus communications.

In addition, there are a few projects targeted toward providing a metacomputing environment on diverse resources. The earliest metacomputer, the NCSA Metacomputer [20], was an integration of several MPPs, mass storage units, visualization and I/O devices. Globus [14] and Legion [21] are among the most recent projects targeted toward solving metacomputing problems. A low-level *toolkit* in the Globus environment provides mechanisms such as communication, authentication, and network information. These mechanisms can be used to construct higher-level metacomputing services such as parallel programming tools, schedulers, etc. On the other hand, Legion is a distributed-object metacomputing environment that is targeted to support a wide set of tools, languages, and programming models. The major objectives of the Legion project are site autonomy, an easy-to-use seamless computational environment, high performance via parallelism, security for users and resource owners, management and exploitation of resource heterogeneity, multiple language support and interoperability and fault tolerance. Additionally, there are several web-based metacomputing projects [22], that either use the JAVA programming language as the main computation language or provide a coordination medium based on WWW technologies or the JAVA language. There may be some drawbacks to these methods. First, they may not support the programs written in other languages such as C and Fortran. Second, they may support communication only between a server and a client, which restricts the execution of the candidate applications.

3 Overview of VDCE Software Architecture

The main design philosophy of VDCE is to provide a general software development environment to build and execute large-scale applications on a network of heterogeneous resources. VDCE is composed of geographically distributed computation sites (domains), as shown in Figure 1, each of which has one or more VDCE Servers. The words "site" and "domain" are used interchangeably in this paper. Each domain consists of several clusters, each of which includes heterogeneous resources in terms of type, speed, or the configuration. At each site the VDCE Server runs the server software, called *site manager*, which handles inter-site communications and bridges VDCE modules to the web-based site repository. The site manager is part of the Control Virtual Machine that was explained in Section 3.3.

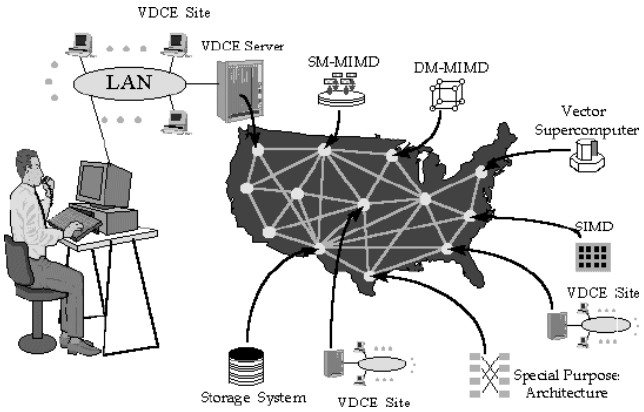

Figure 1: Virtual Distributed Computing Environment (VDCE)

The site repository consists of four different database tables. The *user-accounts table* is used to handle user authentication. In the user-accounts table, each VDCE user account is represented by a 5-tuple: user name, password, user ID, priority, and access domain type. The *resource-performance table* provides the resource (machine and network) performance attributes/parameters. These attributes are grouped into two parts: a) *static performance attributes* stored in the database once during the initial configuration of VDCE: host name, IP address, architecture type, operating system type, and total memory size, the computing weight (which will be described later in the Application Scheduler section) of each processor with respect to a base processor; and b) *dynamic performance attributes* that are updated periodically: CPU load, network latency, network bandwidth, and available memory size, number of processes, etc. The *task-performance table* provides performance characteristics for each task in the system and is used to predict the performance of the task on a given resource. Each task implementation is specified by some parameters: computation size, communication size, and required memory size. For each task in VDCE, the task-performance table includes an entry for the measured execution time of benchmarking the task per machine type as well as the CPU loads when the measurements are taken. In order to find the location of a task's exe-

cutable, VDCE stores location information of each task (i.e., the absolute path of the task executable) as well as other restrictions that might be related to the task execution for each host in the *task-constraints table*. Due to specific library requirements or other license restrictions, some task executables may reside only on a subset of the VDCE hosts.

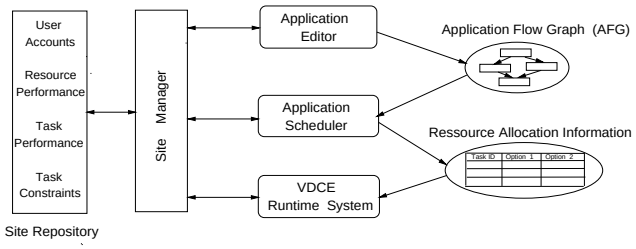

Figure 2: Interactions Among the VDCE Modules

The software development cycle for network applications can be viewed in terms of three phases: application development and specification phase, application scheduling and configuration phase, and execution and runtime phase. The functionality of these three phases is handled by the Application Editor, Application Scheduler, and VDCE Runtime System, respectively. Figure 2 shows the interaction of the VDCE modules within a site. In the following subsections we describe in detail the design and prototype implementation issues of the three main software modules.

3.1 Application Editor

The Application Editor is a web-based graphical user interface for developing parallel and distributed applications. The end-user establishes a URL connection to the VDCE Server software within the site (the *Site Manager*), which runs on a VDCE Server (see Figure 3). The Site Manager implementation is based on JAVA Web server technology, which uses servlets (i.e., server site JAVA applets) that relive the startup overheads and run on any platform. After user authentication (as shown in Figure 3), the Application Editor, which was implemented in JAVA, will be loaded into the user's local web browser so that the user can develop his/her application.

The Application Editor provides menu-driven task libraries that are grouped in terms of their functionality, such as the matrix algebra library, C^3I (command and control applications) library, etc. A selected task is represented as a clickable and draggable graphical icon in the active editor area. Each such icon includes the task name and a set of markers for logical ports. Color coding used in this visual representation helps to distinguish input ports from output ports. Operationally, the Application Editor can be in *task mode*, *link mode*, or *run mode*. In *task mode*, the user can select/add new tasks, and/or click/drag icons to position them conveniently in the active editor area. In *link mode*, the user can specify connections between tasks. In *run mode*, Editor submits the graph for execution and visualizes the performance and runtime characteristics of an ongoing computation.

The process of building a VDCE application with the Application Editor can be di-

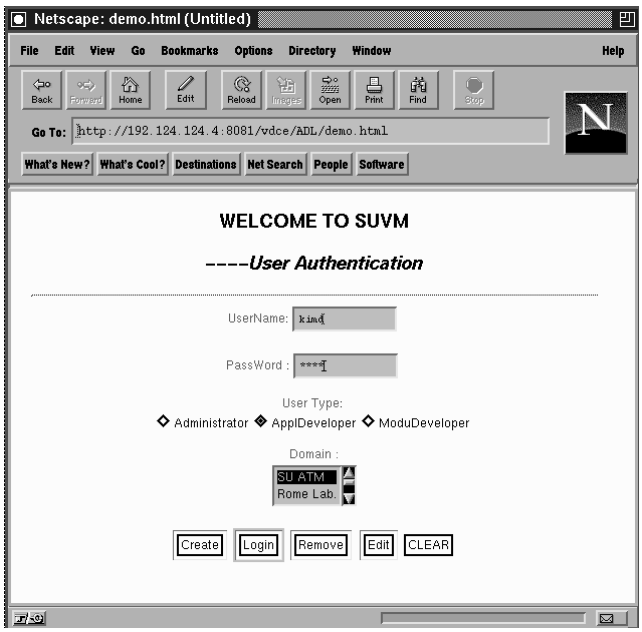

Figure 3: VDCE Authentication Window

vided into two steps: building the application flow graph (AFG), and specifying the task properties of the application. The application flow graph is a directed acyclic graph, $G = (T, L)$, where T is the set of tasks in the application and L is a set of directed links among tasks. A directed link (i, j) between two tasks, T_i and T_j, of the application indicates that T_i must complete its execution before T_j begins to run. Figure 4 shows the application flow graph of a Linear Equation Solver (based on LU Decomposition) developed using the Application Editor. In this application, the problem is to find the solution vector x in an equation $Ax = b$, where A is a known $N \times N$ matrix and b is a known vector. With LU Decomposition, any matrix can be decomposed into the product of a lower triangular matrix L and upper triangular matrix U. Once LU Decomposition is solved, the solution vector, x, is derived with $x = U^{-1}(L^{-1}b)$. To construct the flow graph of this application, the user creates nodes by selecting LU_Decomposition, Matrix_Inverse(2), and Matrix_Multiply(2) tasks from the Matrix_Operations menu.

After the application flow graph is generated, the next step in the application development process is to specify the properties of each task. A double click on any task icon generates a popup panel that allows the user to specify optional preferences such as computational mode (sequential or parallel); domain type (Syracuse University or Rome Lab); cluster type (HPDC cluster, CAT cluster, TOP cluster; Rome Lab Cluster); communication type (P4, socket, MPI, DSM, NCS, PVM); thread type (none, pthread, qthread, cthread), communication protocol type (TCP/IP, ATM); machine type (SUN SPARC, RS6000, Pentium PC, HP) and the number of processors to be used in a parallel implementation of a given task (see the right part of Figure 4). In this figure, for the MULT task of the Linear Equation Solver the user has selected the parallel execution mode using

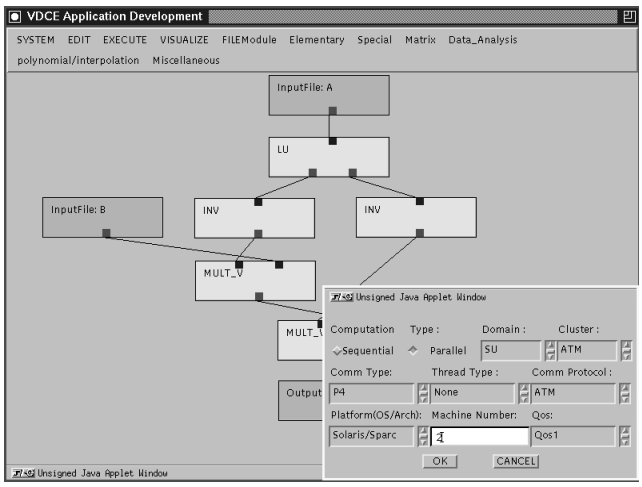

Figure 4: Building the Linear Equation Solver Application with the Application Editor

two nodes of Sun SPARC machines interconnected by an ATM network. When the task properties are specified the user may either submit the application for execution in the VDCE or store the application flow graph for future use.

3.2 Application Scheduler

The main function of the Application Scheduler module in VDCE is to interpret the application flow graph and to assign the current best available resources for running application tasks in order to minimize the total execution time in a transparent manner. This module is based on application-based scheduling framework [7, 8] that is currently being implemented. VDCE provides distributed scheduling in a wide-area system in which each site consists of its own Application Scheduler running on the VDCE server. The Application Scheduler has two scheduling algorithms explained at the following pages: *site scheduler algorithm* and *host selection algorithm*. The schedule of an AFG is determined by the VDCE server at the local site, which runs the site scheduler algorithm, and a set of selected remote sites that execute the Host Selection Algorithm. Table 1 gives the meanings of the symbols used in the algorithms.

The *site scheduler algorithm* and *host selection algorithm* are based on the *list scheduling* [9, 10, 11] heuristic. In list scheduling each node (task) of the graph is assigned a priority and stored in an ordered list. In this paper node and task terms are used interchangeably. Whenever a processor is available for execution the highest priority task in the list is assigned to this processor. This process is repeated until all nodes of the graph are covered. The difference among the list scheduling heuristics is the way in which they assign priorities to nodes. The different priority assignment methods lead to different selection orders that result in different schedules.

We use the level of each node to determine its priority [10]. The *level* of a node is

Table 1
Symbols and their meanings

Symbol	Meaning
AFG	Application Flow Graph.
$Site\_List$	The list of sites that will be part of the scheduling process.
S_{local}	The site that has received the application execution request.
S_{remote}	The set of selected k neighbor sites of S_{local}.
$BW(S_i, S_j)$	The network bandwidth between sites S_i and S_j.
$LT(S_i, S_j)$	The network latency between sites S_i and S_j.
$Pred\_Time(task_i, S_j)$	The *best* predicted execution time of $task_i$ at S_j.
$pred\_time(task_i, P_j)$	The predicted execution time of $task_i$ on P_j.
$EST(task_i, S_j)$	The earliest start time of $task_i$ at site S_j.
$EFT(task_i, S_j)$	The earliest finish time of $task_i$ at site S_j.
$Predecessor(task_i)$	The set of nodes that are immediate predecessor of $task_i$.
$Exec\_time(task_i, P_{test})$	The measured execution time of $task_i$ on P_{test} for the trial run.
$C\_Load(P_j)$	The recent CPU load of P_j.
$M\_Load(P_{test})$	The CPU load of P_{test} at the time of the trial run.
$Weight(P_j)$	The computing weight of P_j with respect to a base processor.

defined by the length of the longest path from the node to a terminal (or exit) node. The *length* of a path in the task graph is measured by the summation of all node weights and edge weights along the path. The node weight is the predicted execution time of the task, and edge weight is the predicted intertask communication time. Some of the previous works do not consider the edge weight when calculating the level of a node. For a node weight, we use the execution time of the task (node) on a predefined base-processor within the site. The weight of an edge between task i and task j is measured by dividing the data size to be sent from task i to task j, $D(i, j)$, to a base communication-link bandwidth, BW_{base}. We assume that each AFG has only one root node and one exit node.

Site Scheduler Algorithm

In this algorithm, the next step after initializing the Task_List with level values of AFG nodes is to select a set of remote sites that will be part of the scheduling process and that may possibly be part of the execution process. If the *update_request* flag is true, it indicates that one or more sites in the S_{remote} have high network traffic (or down). In this case, the remote sites are selected according to the network bandwidth between the remote site and the local site (shown in steps 4–8). Otherwise, the previously stored set is used. Then, AFG and $Task\_List$ are multicast to the involved sites for bidding, after which the $Host\_Selection\_Algorithm$ is executed at each site (step 12).

The Site Scheduler Algorithm receives the bidding from each site for each task in AFG (step 12), i.e., the best available processor, and the predicted execution time on the best available processor. Step 14 assigns the root task to the site that minimizes the predicted execution time. Step 19 calculates the earliest start time (EST) of the current task ($task_i$) at each site (S_j). To obtain the EST value of $task_i$, the summation of the

earliest finish time (EFT) and the communication cost is calculated for each immediate predecessor task of $task_i$ in the graph. The EFT of a task at a site is calculated by the summation of its EST value and the predicted execution time of the task at the current site (step 20). As shown in step 22, the best site of a node is the one that minimizes the EFT value. The best available site for the current task is determined at each iteration of the while-loop from step 16 to step 25. For an application flow graph $AFG(v, e)$ with v nodes and e edges the while-loop takes $O(v)$ to compute the EST value of a node on a site (steps 15 and 16). We assume AFG to be a dense graph in which the number of edges are proportional to $O(v^2)$. Since there are v nodes in AFG and k sites involved in the scheduling process, the while-loop takes $O(kv^2)$ time; hence the time complexity of the site scheduler algorithm is $O(kv^2)$, since the while-loop is the dominant part. The value of k will be much smaller than v; thus the worst case complexity of the algorithm is $O(v^2)$.

Site_Scheduler_Algorithm(AFG)

Step 1 Compute the *level* for all nodes in AFG.

Step 2 Initialize $Task\_List$ according to a non-increasing order of node *level*.

Step 3 Read S_{remote} list and the *update_request* flag from resource performance table.

Step 4 **If** *update_request* flag is true **then**

Step 5 Select k nearest neighbor sites of S_{local} that maximize the network bandwidth and store them in a set, S_{remote}.

Step 6 $update\_request \leftarrow false$.

Step 7 Update S_{remote}, and *update_request* in the resource performance table.

Step 8 **endif**

Step 9 $Site\_List \leftarrow S_{local} \bigcup S_{remote}$

Step 10 **For** each site $S_j \in Site\_List$ **do**

Step 11 Send AFG and $Task\_List$ for bidding.

Step 12 $\{Pred\_Time(task_i, S_j), Best\_Resource(task_i, S_j)\} \leftarrow$
$Host\_Selection\_Algorithm(Task\_List) \quad \forall\, task_i \in Task\_List.$

Step 13 **endfor**

Step 14 $Resource\_Alloc\_Table(task_1) \leftarrow S_m$, such that:
$Pred\_Time(task_1, S_m) \leftarrow \min\{Pred\_Time(task_1, S_i)\},\ \forall\, S_i \in Site\_List.$

Step 15 Remove $task_1$ from the $Task\_List$.

Step 16 **while** $Task\_List$ is not empty **do**

Step 17 $task_i \leftarrow$ the first task in $Task\_List$.

Step 18 **For** each site, S_j, in the $Site\_List$ **do**

Step 19 $EST(task_i, S_j) \leftarrow \max\left\{EFT(task_k, S_m) + (LT(S_m, S_j) + \frac{D(k,i)}{BW(S_m, S_j)})\right\}$
$\forall\, task_k \in Predecessor(task_i)$, such that:
$S_m \leftarrow Resource\_Alloc\_Table(task_k)$.

Step 20 $EFT(task_i, S_j) \leftarrow EST(task_i, S_j) + Pred\_Time(task_i, S_j)$.

Step 21 **endfor**

Step 22 Select $Best\_Site$, such that:
$EFT(task_i, Best\_Site) \leftarrow min\{EFT(task_i, S_j)\},\ \forall S_j \in Site\_List.$

Step 23 $Resource\_Alloc\_Table(task_i) \leftarrow Best\_Resource(task_i, Best\_Site)$

Step 24 Remove $task_i$ from the $Task\_List$.

Step 25 **endwhile**

Step 26 Multicast the $Resource\_Alloc\_Table$ to the relevant sites.

Host Selection Algorithm

The Host Selection Algorithm determines the task assignments of AFG tasks on the available processors within each site. The calculation of the EST is similar to the previous algorithm. In this algorithm, base communication-link bandwidth, BW_{base}, is considered for all connections within a site (step 4). Additionally, the latency within a site is negligible if it is compared with the latency between the different sites. The communication cost between a task and its immediate predecessor is zero if they are scheduled to the same processor. The core of the Host Selection Algorithm is the performance prediction phase. The execution time prediction of a task on a a given resource is based on the current load of the processor, load of the test processor at the time of trial run, measured execution time for the trial run, and computing weights (step 5).

The measured execution time and the load value for the trial runs are retrieved from the task-performance table, as explained in the Site Repository section of this paper. $Weight(P_j)$ is the computing weight [12, 13] of processor P_j with respect to the base-processor at the site. To calculate the weight of each processor, trial runs of a set of task implementations are executed on each processor. The ratio of average execution time of the trial runs on a processor P_i to the average execution time on the base-processor gives the computing power weight of P_i. In step 6, the EFT value is the summation of the EST and the predicted execution time. For each task, the processor that minimizes the EFT value is selected as the best resource in this site. An iteration of the while loop takes $O(pv)$ times, where v is the number of nodes in AFG and p is the number of processor in the $Processor\_List$. Thus the time complexity of the Site Scheduler Algorithm is $O(pv^2)$.

Host_Selection_Algorithm(Task_List)

Step 1 **while** $Task\_List$ is not empty **do**

Step 2 $task_i \leftarrow$ the first task in $Task\_List$.

Step 3 **For** each available processor, P_j, in the $Processor\_List$ **do**

Step 4 $EST(task_i, P_j) \leftarrow \max\left\{EFT(task_k, P_m) + Comm\_Cost(task_k, task_i)\right\}$
$\forall\, task_k \in Predecessor(task_i)$ such that:
$P_m \leftarrow Best\_Resource(task_k)$ and
$$Comm\_Cost(task_k, task_i) \leftarrow \begin{cases} \frac{D(k,i)}{BW_{base}} & P_m \neq P_j \\ 0 & otherwise \end{cases}$$

Step 5 $$Pred\_Time(task_i, P_j) \leftarrow \frac{C\_Load(P_j)}{M\_Load(P_{test})} \times Exec\_Time(task_i, P_{test}) \times \frac{Weight(P_{test})}{Weight(P_j)}$$

Step 6 $EFT(task_i, P_j) \leftarrow EST(task_i, P_j) + Pred\_Time(task_i, P_j)$

Step 7 **endfor**

Step 8 $Best\_Resource(task_i) \leftarrow P_k$, such that:
$EFT(task_i, P_k) \leftarrow min\{EFT(task_i, P_j)\}, \quad \forall P_j \in Processor\_List.$

Step 9 **endwhile**

Step 10 Return $Pred\_Time(task_i, Best\_Resource)$ and $Best\_Resource(task_i)$ to S_{local} for each task.

3.3 VDCE Runtime System

The VDCE Runtime System sets up the execution environment for a given application and manages the execution to meet the hardware/software requirements of the application. The VDCE Runtime System separates control and data functions by allocating them to the Control Virtual Machine (CVM) and Data Virtual Machine (DVM), respectively. CVM measures the loads on the resources (hosts and networks) periodically and monitors the resources for possible failures. CVM daemons control the execution of the application tasks on the assigned resources based on the performance and quality of service requirements. Application visualization (real-time or post-mortem) services are provided by CVM. DVM provides an execution environment for a given VDCE application by binding tasks so that they can interact and communicate efficiently. DVM supports socket-based point-to-point connections for inter-task communications.

Control Virtual Machine (CVM)

The functionality of CVM is provided by the following four processes: Site_CVM, Local_CVM, Monitor, and Cluster Manager (see Figure 5). Each VDCE machine runs a Local_CVM process and a Monitor daemon. Additionally, one of the machines within each cluster executes the Cluster Manager process. Each site (domain) has a Site_CVM process located at the VDCE Server machine. The main functions of the stated CVM processes are given below:

- *Retrieving Resource Performance Parameters.* VDCE resources are periodically monitored to collect up-to-date values of processor and network parameters that were given in the Site Repository subsection of this paper. The *Monitor* daemon of each machine periodically measures the up-to-date parameters every 30 seconds and updates its fields at the Cluster leader machine shown in Figure 5. The Cluster Manager daemon gathers the parameters of machines within the cluster in a table and periodically forwards the table to the Site_CVM every 60 seconds. In the future implementation the Cluster Manager will be modified to send only the workloads of the resources that have changed considerably from the previous measurement. The workload of a resource is significantly changed if the up-to-date measurement is higher or lower than the summation of the previous measurement and the width of the confidence interval [15].

- *Updating the Site Repository.* The Site_CVM periodically updates the resource-performance table at the site repository with the parameters that are collected from Cluster Managers. The execution time and load measurement of benchmarking runs

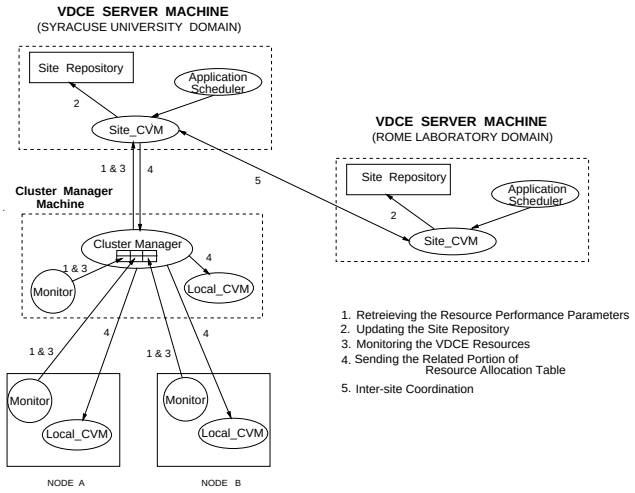

Figure 5: Interactions Among the Control Virtual Machine Components

of tasks are stored at the task-performance table.

- *Monitoring the VDCE Resources.* When a Monitor daemon of a processor stores its parameters, it reads the random number that was generated by the Cluster Manager and updates its alive_check field with this value. Every 60 seconds the Cluster Manager compares its alive_check field with each cluster machine's alive_check field. The machines with a different value are marked as *down*; others are marked *alive*. After the comparison, the Cluster Manager assigns a new random number for its alive_check field. The monitor information is forwarded to the Site_CVM with the resource parameters to be stored at the site repository. The machines that are marked as *down* at the resource-performance table are not selected by the Application Scheduler.

- *Sending the Related Portion of the Resource Allocation Table.* After the resource allocation table is generated by the Application Scheduler, the Site_CVM multicasts it to the Cluster Managers that will be involved in the execution. If a machine in a cluster is assigned for a task execution, the Cluster Manager sends an execution request message and related parts of the resource allocation table to the Local_CVM of the machine.

- *Inter-site Coordination.* As explained in Section 3.2, the Application Scheduler at the local site selects a subset of remote sites and multicasts the application flow graph to these sites. The remote sites run the *Host Selection Algorithm* locally and transfer the mapping decisions to the sender site. The inter-site coordination and message transfer are handled by Site_CVMs.

- *Initialize the Application Execution Environment.* After the Local_CVM receives an execution request message from the Cluster Manager, it activates the DVM. The DVMs on the assigned machines set up the application execution environment by

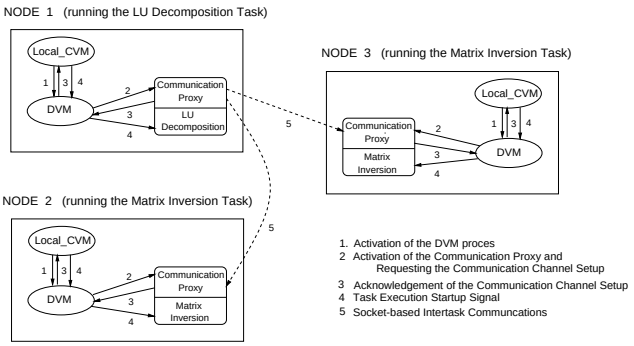

Figure 6: Setting Up the Application Execution Environment

starting the task executions and creating point-to-point communication channels for inter-task data transfer. Figure 6 shows the part of the execution environment of the Linear Equation Solver application discussed in Section 3.1. Machine 1 will execute the LU_Decomposition task, which is followed by the execution of Matrix_Inversion tasks on Machine 2 and Machine 3. When all the required acknowledgments are received, an execution startup signal is sent to start the application execution.

- *Managing the application execution.* The Local_CVM monitors the application execution on the assigned machines and maintains the performance, fault tolerance, and QoS requirements of the application tasks. If the current load on any of these machines is more than a predefined threshold value, the Local_CVM terminates the task execution on the machine and sends a task rescheduling request to the Site_CVM through the Cluster Manager.

Data Virtual Machine (DVM)

DVM is a socket-based, point-to-point communication system for inter-task communications. Therefore, any machine that supports socket programming can be part of VDCE. As shown in Figure 6, the DVM activates the communication proxy and sends the resource allocation information, including the socket number, IP address for target machine, etc., that will be used for the communication channel setup. After the setup is completed successfully, the communication proxy sends an acknowledgment to the Local_CVM. The execution startup signal is sent to start the task executions.

On the other hand, for a thread-based programming environment, the Data Manager consists of three threads that are initiated by the communication proxy: send thread, receive thread, and compute thread. After the communication channel is established, the send and receive threads are activated for data transfer and the compute thread performs the task execution. The control transfer between the Local_CVM and the DVM (or any other control transfer on the same machine) are based on an inter-process communication mechanism (i.e., pipes or shared-memory paradigm). The data transfer among the communication proxies (or between send and receive threads for multithreaded systems) uses a socket-based, message-passing mechanism.

Since user tasks can be programmed in various message-passing tools, the VDCE Runtime System supports multiple message-passing libraries such as P4, PVM, MPI, NCS. Additionally, the VDCE Runtime System provides data conversions that might be needed when an application execution environment includes heterogeneous machines. The VDCE Runtime System provides several user-requested services such as I/O service, console service, and visualization service. A user can request these services while developing his/her application with the Application Editor. I/O Service provides either file I/O or URL I/O for the inputs of the application tasks. The user can suspend and restart the application execution with the console service. The VDCE visualization service provides both real-time and post-mortem visualizations. There are three types of visualizations provided in VDCE:

- Application Performance Visualization: The execution time of tasks in an application is visualized.
- Workload Visualization: Up-to-date workload information on VDCE resources is visualized.
- Comparative Visualization: VDCE makes it possible for an end user to experiment and evaluate his/her application for different combinations of hardware and software medium by providing the comparative performance visualization.

4 VDCE Testbed: Experimental Results and Discussion

The current VDCE prototype consists of two sites, one at Syracuse University and the other at Rome Laboratory, that are connected by the NYNET ATM Wide Area Network, as shown in Figure 7. Each site or domain has a VDCE server, a Site Repository and several computing clusters. At the Syracuse University site there are three computing clusters: HPDC, CAT, and TOP. The HPDC cluster consists of several ATM switches and ATM concentrators that connect high-performance workstations and PCs at a rate of 155 and 25 Mbps, respectively (URL:http//www.atm.syr.edu). The TOP and CAT clusters have SUN SPARCs, SUN IPXs and IBM RS6000s that are connected to the ATM cluster through the Ethernet. The Rome Lab site consists of three clusters that include SUN, Digital, and HP workstations.

In this section we discuss and evaluate the performance of the current VDCE prototype in implementing two important tasks: 1) The use of VDCE as an evaluation tool for the parallel implementations of the VDCE library tasks using different numbers of workstations, and different networks to connect them (e.g, ATM or Ethernet); and 2) The use of VDCE as a problem-solving environment for large-scale VDCE applications.

4.1 Experiment 1: Using VDCE as a Parallel Evaluation Tool

In this experiment we used the matrix multiplication (MULT) task as a running example to show the use of the VDCE for experimentation and to evaluate the performance

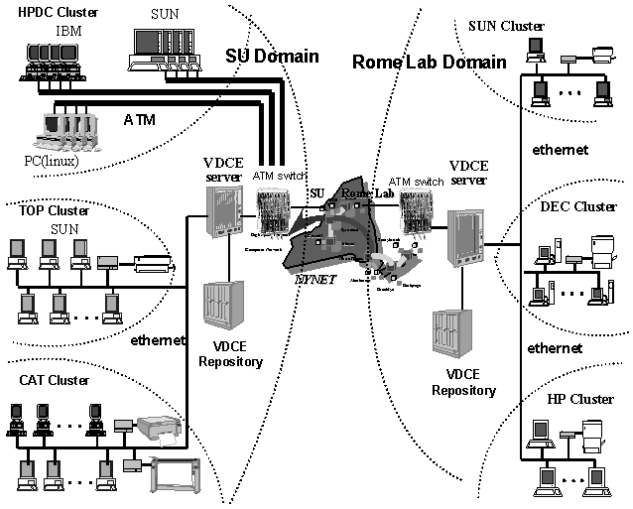

Figure 7: The configuration of the VDCE Testbed

of different configurations when the number of computers, network types, and problem sizes are changed. We compared the time and effort required to perform such tasks with and without using the VDCE. We benchmarked the sequential and parallel algorithms of Matrix multiplication(MULT) based on various machine and network configurations and problem sizes. The parallel implementation of MULT ($A \times B = C$) task is based on the host-node programming model. The master process distributes the rows of matrix A evenly among the processes (where each process runs on one workstation) while all the slave processes receive the entire B matrix. Each slave process computes its part of result matrix C and sends it back to the host process.

The VDCE provides a web-based, user-friendly interface that allows a novice programmer to experiment with and evaluate different parallel configurations of each VDCE task in minutes. We argue that performing similar evaluation tasks is almost impossible for novice programmers and requires hours and even days to be performed by an expert programmer using parallel processing and message passing and visualization tools. With VDCE, once a task library is registered to the VDCE site repository, any VDCE user can use that task or any existing VDCE task by just clicking on the task name in the Application Editor. Once the task is selected, the user can click on one button to determine the problem size, the number of computers to be involved in the computation, and the network to be used to connect them. Selecting the VDCE task and specifying how it will be implemented can be done in a few minutes. Once that is done, the task configuration can be run and its execution time visualized immediately without any effort other than clicking on the execute and visualize buttons.

Figure 8 shows the execution times of the VDCE-based, matrix multiplication algorithm for 512×512 and 1024×1024. The result for p4-based implementation of the same multiplication algorithm is given in Figure 9. The experiments were done for one, two

and four Sun SPARCs that are connected by an IP/ATM network. We also evaluated the performance of MULT task on a heterogeneous cluster of four SUN SPARCs and four IBM RS6000 workstations. The objective of such an evaluation is to provide users with a better understanding of the performance of parallel processing algorithms when there is a change in problem size, number of nodes, or network type. As an example, for the p4-based, matrix multiplication algorithm, we can determine from Figure 9 that eight nodes provide the best performance among the test cases.

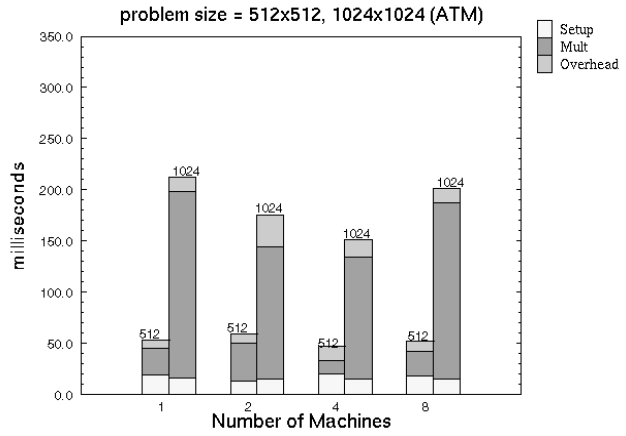

Figure 8: Execution Time of Matrix Multiplication Task Using VDCE

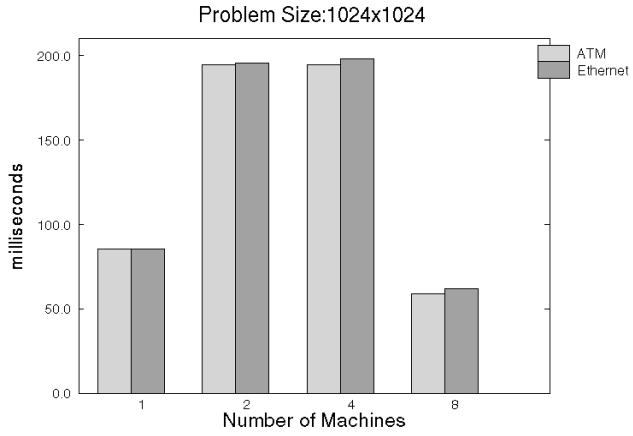

Figure 9: Execution Time of Matrix Multiplication Task Using p4

Table 2 compares the times required to develop, compile, execute, and visualize a Matrix Multiplication task using p4 and VDCE for a 1024×1024 problem size with four

Table 2
The performance comparison of matrix multiplication task for each software phase

Phase	p4	VDCE
Design and development	862 min. (431 lines)	2.10 min.
Compilation	7.01 sec.	0 sec.
Runtime setup	0.980 sec.	0.015 sec.
Task execution	0.194 sec.	0.136 sec.
Visualization and evaluation	1890sec.	0.095 sec.

nodes. In the design and implementation phase, it takes around 862 minutes for a parallel programming expert to develop a p4-based multiplication program from scratch if we assume that programming speed is two minutes per line. If the programmer has no experience with p4, he/she will spend more time to learn about it and to develop an application. For VDCE, even if the user does not have any knowledge about parallel programming, but wants to run the application in parallel, the only thing he/she needs to do is to choose the parallel option in the application design window of the Application Editor. Additionally, he/she can easily define the I/O for a task using the Application Editor. The total time for developing a VDCE MULT application is 2.10 minutes.

There is no compilation time in VDCE after the VDCE MULT application is designed. The location of the executable for MULT task on the selected resource is provided in the resource allocation information, which is retrieved from the task constraints table. The executable is then linked to the I/O module. In the p4 version the MULT program takes 7.01 seconds for compilation. The runtime setup time in VDCE is for the CVM to transfer the activation and resource allocation information to DVM and to wait for the acknowledgment, which takes 15 milliseconds for the MULT task on the selected resource. For a p4 application, the user creates a configuration file, i.e., procgroup file, and manually links it to the p4 application which takes 980 milliseconds. VDCE runs the application automatically with the "Execute Application" button and generates the results in the selected output file. The execution time of MULT task is 136 milliseconds when it is executed on four nodes over the ATM. The execution time is 194 milliseconds using a p4 program with the same configuration.

VDCE provides dynamic and post-mortem visualization of the application. A VDCE user monitors the load of all machines dynamically in the domain and he/she can consider the load information to select an appropriate machine and/or a cluster. In addition, the execution time of each module within an application is visualized in VDCE. It takes 95 milliseconds to invoke the VDCE visualization window for the MULT task. If a p4 user wants to visualize the execution time to compare its performance with others, it is necessary to use another graphic tool. The visualization and evaluation time depends on which tool is used; as an example, "gnuplot" takes 1890 seconds.

Table 3
Performance comparison of linear equation solver application for each software phase [1]

	p4			VDCE		
Phase	LU	INV	MULT	LU	INV	MULT
Design and development	838 min. (419 lines)	1314 min. (657 lines)	862 min. (431 lines)	2.10 min.	1.57 min.	2.30 min.
Compilation	6.45 sec.	8.10 sec.	7.01 sec.	0 sec.	0 sec.	0 sec.
Runtime setup	1.200 sec.	1.580 sec.	0.980 sec.		0.043 sec [2]	
Task execution	0.386 sec.	0.556 sec.	0.194 sec.	0.801 sec.	1.360 sec.	0.140 sec
Application execution		1.691 sec.			1.451 sec.	
Application visualization		3200 sec.			0.140 sec.	

4.2 Experiment 2: Using VDCE as a Problem Solving Environment

In this experiment we demonstrated how the VDCE can enable a novice programmer to develop large-scale parallel and distributed applications running on geographically distributed heterogeneous resources. Implementing such applications is currently a challenging programming problem and time consuming for experts on parallel and distributed programming tools. A distributed application can be viewed as an Application Flow Graph (AFG), where its nodes denote computational tasks and its links denote the communications and synchronization between these nodes. Without an application development tool, a developer or development team must apply much effort and time to develop a distributed application from scratch. The VDCE provides a web-based interface to enable users to develop, configure, execute, and visualize such a distributed application in a few minutes. However, to perform the same tasks in a non-VDCE case, the user or team developers need to develop techniques to interact and communicate the modules running on different computers, and they need to develop or integrate techniques to run and manage the execution of the distributed application, as well as collect and visualize the required performance results.

To solve these difficulties, VDCE provides an integrated problem solving environment to enable novice users to develop large-scale, complex, distributed applications using VDCE tasks. The Linear Equation Solver (LES) application has been selected as a running example. Figure 4 shows the AFG of Linear Equation Solver, which consists of an LU Decomposition (LU) task, two Matrix Inversion (INV) tasks and Matrix Multiplication (MULT) tasks. The problem size for this experiment is 1024×1024 using four nodes, which are SUN SPARCs and IBM RS6000 machines that are connected by an ATM network.

Table 3 compares the timing of several software phases for a Linear Equation Solver application using p4 and VDCE. When a user has enough knowledge about parallel programming and the p4, he/she will spend 838 minutes for an LU task, 1314 minutes for

[1] The last two rows of the table are for the total time of the application.
[2] It is the total setup time for a VDCE-based linear equation solver application.

an INV task, and 862 minutes for MULT task. The total time to develop the application for a non-VDCE version is approximately 3014 minutes, (i.e., around 50 hours). Using VDCE, a novice user spends around six minutes to develop such an application. There is no compile time for VDCE, but a p4 application needs 21 seconds for compilation. The VDCE setup time for a Linear Equation Solver application is 43 milliseconds. The p4 user should create all procgroup files and launch them in order, which takes around eight seconds.

Since the VDCE is based on the data flow model and executes tasks automatically, there may be overlap among task executions that causes the total execution time of the VDCE application, including the setup time, to be less than the summation of all individual task execution times. In our experiment with the Linear Equation Solver application, the total execution time of p4 parallel execution using four nodes is 1691 milliseconds. A VDCE-based execution with the same configuration takes 1451 milliseconds, which outperforms the p4 by 16%.

5 Conclusion

We have presented the design and evaluation of the Virtual Distributed Computing Environment (VDCE) being developed at Syracuse University. The VDCE consists of three main modules: Application Editor, Application Scheduler, and VDCE Runtime System. The Application Editor provides users with all the software tools and library functions required to develop a VDCE application. The main function of the Application Scheduler is the initial task-to-resource mapping and any necessary dynamic rescheduling. The VDCE Runtime System is based on the Control Virtual Machine (CVM) and the Data Virtual Machine (DVM). CVM provides a seamless interconnection of the resources and monitors the resources. DVM enables a high-performance communication medium among the application tasks.

We have successfully implemented a proof-of-concept prototype that supports all major components of the VDCE architecture. We are currently working on extending the current prototype in several ways: a) develop and implement an application programming interface (API) that enables users to add VDCE library tasks; b) add more sites to increase the computing services offered by VDCE; and c) develop and integrate mobile computing technology into VDCE so that users can access VDCE resources using mobile hosts and mobile interconnection networks.

Acknowledgments

We would like to thank Elaine Weinman for proofreading this manuscript.

References

[1] J. C. Browne, S. Hyder, J. Dongarra, K. Moore, P. Newton, Visual programming and debugging for parallel computing, IEEE Parallel and Distributed Technology, 3(1) (1995) 75–83.

[2] H. Topcuoglu, S. Hariri, W. Furmanski, J. Valente, I. Ra, D. Kim, Y. Kim, X. Bing, B. Ye, The software architecture of a virtual distributed computing environment, in *Proceedings of Sixth IEEE International Symposium on High Performance Distributed Computing*, 1997, pp. 40–49.

[3] H. Topcuoglu and S. Hariri, A global computing environment for networked resources, in *Proceedings of International Conference on Parallel Processing*, 1997, pp. 493–496.

[4] P. Newton, J. C. Browne, The CODE 2.0 graphical parallel programming language, in *Proceedings of ACM International Conference on Supercomputing*, 1992.

[5] R. Wolski, C. Anglano, J. Schopf, F. Berman, Developing heterogeneous applications Using Zoom and HeNCE, in *Proceedings of the Forth Heterogeneous Computing Workshop*, 1995.

[6] C. Angalano, J. Schopf, R. Wolski, F. Berman, Zoom: a hierarchical representation for heterogeneous applications, technical report cs95-451, Computer Science Department, University of California at San Diego, 1995.

[7] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, Application-level scheduling on distributed heterogeneous networks, in *Proceedings of Supercomputing 96*, 1996.

[8] J. Weissman, A. Grimshaw, A federated model for scheduling in wide-area-systems, in *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, 1996, pp. 542–550.

[9] T.L. Adam, K. Chandy, and J. Dickson, A comparison of list scheduling for parallel processing systems, Communication of ACM, 17 (1974) 685–690.

[10] H. El-Rewini, H. Ali, T. Lewis, Task scheduling in multiprocessing systems, IEEE Computer, 28(12) (1995) 27–37.

[11] Y. Kwok, I. Ahmad, Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors, IEEE Transactions on Parallel and Distributed Systems, 7 (1996) 506–521.

[12] Y. Yan and X. Zhang, An effective and practical performance prediction model for parallel computing on nondedicated heterogeneous NOW, Journal of Parallel and Distributed Computing, 38 (1996) 63–80.

[13] M. Zaki, W. Li and M. Cierniak, Performance impact of processor and memory heterogeneity in a network of machines, in *Proceedings of the Forth Heterogeneous Computing Workshop*, 1995.

[14] I. Foster and C. Kesselman, Globus: a metacomputing infrastructure toolkit, in *Proceedings of the Workshop on Environment and Tools for Parallel Scientific Computing*, 1996.

[15] H. Casanova and J. Dongarra, Netsolve: a network server for solving computational science problems, in *Proceedings of Supercomputing 96*, 1996.

[16] A. Beguelin, J. Dongara, A. Geist, R. Manchek, and V. Sunderam, User Guide to PVM, Oak Ridge National Laboratory and Department of Mathematics and Computer Science, Emory University, 1993.

[17] Message Passing Interface Forum, MPI: A message-passing interface standard, version 1.0 May 1994.

[18] R. Butler and E. Lusk, User's guide to the p4 programming system, Mathematics and Computer Science Division, Argonne National Laboratory.

[19] S. Park, S. Hariri, Y. Kim, J.S. Harris, and R. Yadav, NYNET communication system (NCS): a multithreaded message passing tool over ATM network, *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, 1996, pp. 460–469.

[20] L. Smarr and C. Catlett, Metacomputing, Communications of the ACM, 35, 6, (June 1992) 44–52.

[21] A. Grimshaw and W. Wulf, Legion - A View from 50,000 Feet, *Proceedings of Fifth IEEE International Symposium on High Performance Distributed Computing*, 1996, pp. 89–99.

[22] K. Dincer, *World-Wide Virtual Machine: A Metacomputing Environment Integrating World-Wide Web and High Performance Computing and Communication Technologies,* Ph.D. Thesis, Syracuse University, 1997.

[23] J. Gehring and A. Reinefeld, MARS - A framework for minimizing the job execution time in a metacomputing environment, Future Generation Computing Systems, (1996).

[24] Mike Litzkow, Miron Livny Experience with the condor distributed batch system, in *IEEE Workshop on Experimental Distributed Systems*, 1990.

Haluk Topcuoglu is a Ph.D. candidate in Electrical Engineering and Computer Science Department at Syracuse University. He received B.S. and M.S. degree in 1991 and 1993 respectively in computer engineering from Bogazici University, Istanbul, Turkey. His research interests include task scheduling techniques in heterogeneous environments, metacomputing issues, and high-performance parallel and distributed systems. Mr. Topcuoglu has authored or coauthored 6 technical papers in the area of parallel and distributed computing. He is a member of ACM, IEEE, IEEE Computer Society, and Phi Beta Delta.

Dongmin Kim is a Ph.D. candidate in Electrical Engineering and Computer Science Department at Syracuse University, New York. He received an M.S. degree in computer science from Syracuse University, and a B.S. degree in mathematics from Hanyang University, Seoul, Korea in 1991.

Yoonhee Kim received an M.S. degree in computer science from Syracuse University, New York in 1996. She is currently a Ph.D. candidate in Electrical Engineering and Computer Science Department at Syracuse University in 1996. Her research interest includes distributed and heterogeneous computing systems, mobile computing and software architecture.

Baoqing Ye is a Ph.D candidate in Electrical Engineering and Computer Science Department at Syracuse University. She received an M.S. degree in Engineering from TsingHua University, China in 1994, and B.S. in Computer Science from GuiLin Elec. Industry University, China in 1991. She worked in a software engineer position at Beijing Goma Tech Co. (Tandem Computer's Branch) in China between 1994 and 1995.

Xue Bing is a Ph.D. candidate in Electrical Engineering and Computer Science Department at Syracuse University. She was department director at Jiangsu Institute of Computing Technology in China for the 1992-1996 period. She received an M.S. in Computer Science and Engineering department at University of Nanjing Technology in 1984 and B.S. in Computer Science at East China Institute of Technology in 1982.

Ilkyeun Ra received B.S. degree in computer science from the Sogang University, Korea in 1985, and M.S. degree in computer science from the University of Colorado at Boulder in 1994. Mr. Ra is a Ph.D. student in Electrical Engineering and Computer Science Department at Syracuse University. His research interests include distributed shared memory, high performance parallel and distributed systems, metacomputing and high-

speed communication protocols.