Chapter 5

The VCE Execution Environment

The tools in the VCE software development suite help users to construct high performance applications. They allow an individual task to be implemented on various platforms with a set of executable files that all have the same input requirements and output characteristics. Applications can be constructed by indicating how tasks are to be interconnected and dispatched. The VCE run time system (the *execution suite*) manages the execution of these applications. It is responsible for determining what hardware resources are available and which of the available platforms should be used to host task executions. The execution suite is also responsible for providing communication facilities between application components and efficiently transporting intermediate data between tasks.

Executing large multistage applications on multiple high performance computers presents unique scheduling and load balancing challenges. Each application task may run for minutes or hours. Transient load conditions can change drastically during task execution and impede application performance. Migrating a task from one parallel computer to another during its execution is problematic. In order to address the difficulty of providing acceptable performance in an environment where process migration is not feasible, the VCE execution suite employs redundant scheduling techniques. If multiple suitable hosts are available to run a particular task, the VCE execution suite will dispatch the task on several host simultaneously. If the load on

some hosts increases, the task will complete in the time achieved by the least loaded host.

Of course, redundant scheduling requires specialized communication support. New Algorithms have been devised to provide flexible and efficient communication functions among redundantly executing task instances.

This chapter will provide a detailed description of the scheduling, dispatching and load balancing methods used to manage redundant execution (Section 5.1). It will also outline the methods used to provide transparent communication in the presence of redundant execution (Section 5.2).

5.1 VCE Scheduling Techniques

A VCE application consists of a set of tasks with associated synchronization and communication requirements. The tasks can be dispatched independently, as long as all synchronization constraints are observed and appropriate intertask communication mechanisms are provided. This section will explore techniques for scheduling and dispatching VCE tasks.

The goal of the VCE's scheduling techniques are to maximize application performance, improve hardware utilization and minimize interference with non-VCE workload (particularly on privately owned workstations).

Some of the factors which have been considered in the design of the VCE scheduling system include:

- The existence of many hosts to run each task.
- Preemptive process migration in a heterogeneous environment is problematic.
- Precise load measurement across a wide area network is difficult.
- The number and types of host systems participating in the VCE may change from one application run to the next and even during a single application execution.

In order to accommodate these factors a scheduling/dispatching protocol has been designed for the VCE. It involves scheduling redundant execution of task instances. A conservative load measurement policy is used to identify lightly loaded hosts for initial task placements. When task instances are placed on privately owned workstations they are run at low priority to minimize their impact on locally originated work.

While the VCE may benefit in the future from a large repertoire of process migration mechanisms (Section 3.5), current technology makes preemptive process migration between incompatible machines, and even between architecturally compatible parallel computers, impractical. Experimental and simulation studies discussed in Chapter 6 have shown that the redundant scheduling policy described here is suited to a broad range of VCE configurations.

The VCE has several characteristics which make it significantly different from more conventional distributed systems. These differences make it difficult to use an existing scheduling discipline. Instead, careful study of typical VCE applications and hardware configurations, along with analysis of simulated and prototype systems, was used to develop a scheduling protocol appropriate for use in the VCE.

While scheduling and load balancing in distributed systems have been studied thoroughly [26], some key assumptions of those research efforts do not hold for the Virtual Computing Environment. For instance, load balancing studies have traditionally assumed that the number of tasks to be scheduled is much larger than the number of processors available [32]. However, in the VCE this situation may be reversed. At any given time, there may be only a few Grand Challenge VCE applications running, each with a small number of tasks executing concurrently in any one phase of the application. Further, if a large number of computer centers, connected by a wide area network, are participating in the VCE, the number of possible hosts for each task can be quite large.

The scheduling mechanisms of the VCE have less control over the environment than other remote schedulers. Most scheduling schemes assume that all work on all processors is under the control of a single scheduling algorithm [15, 28, 32, 34, 36, 37, 84], which is not the case in the VCE. Computers participating in a VCE

configuration will remain available to accept tasks from other sources (usually local users). Most load fluctuations will be caused by increased or decreased demands of local users rather than by VCE activities. The VCE's goals of providing the best possible performance of VCE applications with the least impact to non-VCE users is less ambitious than the cited projects which manage all scheduling on (generally smaller) distributed systems and seek to approximate globally optimal scheduling.

Intelligent scheduling techniques [28, 84, 106] may be useful in the VCE. An algorithm based on neural networks has been proposed [106] which learns the characteristics of jobs after repeated runs and can use this knowledge to predict the effect of various scheduling options. Implementation of these approaches in the VCE is difficult. The large number of hosts in a VCE configuration makes collection of accurate state information problematic. Because the available hardware may be completely different each time an application is launched, inferences based on passed application behavior may be unreliable. Currently VCE research is concentrated on the use of redundant scheduling techniques, which have proven very successful (Chapter 6).

5.1.1 The VCE Execution Environment Components

The components of the VCE run time environment were discussed in Section 3.4.1. They include a scheduling/dispatching daemon running on each host node, group leaders which collect load statistics from the scheduling/dispatching daemons on architecturally compatible hosts, and an execution process which runs on a user's workstation. The execution process interprets application descriptions and collaborates with other components to schedule and dispatch tasks.

5.1.2 Elements of the Load Balancing Strategy

Dynamic load balancing and scheduling strategies can be characterized by the following four components [54]:

1. **The processor load measurement technique** - determines what quantities will be measured to gauge system load conditions.

2. **The information exchange policy** - indicates when machines exchange load information to detect imbalances.

3. **The transfer policy** - the criteria which governs when and what work should be moved from one machine to another.

4. **The location policy** - selects a machine where migrated work is to be placed.

This section will discuss how redundant task scheduling allows a simple and efficient implementation of each of these four components.

Processor Load Measurement

Objective

In any scheduling system, load measurement is needed to detect load imbalances. Systems that employ process migrations must insure that both the migrating task and the system overall will benefit from a migration. Timely and precise load measurements are often needed to achieve this objective. However, excessive overhead due to load measurement must be avoided.

Common Approaches

A survey of load measurement methods is included in [69]. Some possible load measures include: the number of tasks in a processor's run queue; the amount of available memory; the rate of CPU context switches; average run queue length over some time quantum; or the amount of idle CPU time during a time quantum. In practice, it is common to use some combination of these methods [9, 109].

Many systems which schedule remote executions on privately owned workstations use load measurement techniques geared towards distinguishing whether or not a machine is idle [6, 20, 21, 87]. "Idle" is usually defined as a very small average run queue length and no keyboard or mouse activity over some period of time [101].

Redundant Task Scheduling Approach

The main goal of load measurement in redundancy based scheduling is to identify lightly loaded machines where task instances should be placed. An extremely precise

load measurement technique is not needed for redundant task scheduling. If one instance of a task is placed on a heavily loaded machine, as a result of imprecise load measurement or bursty changes in the load on that computer, the result produced by another instance of the task running on a lightly loaded machine will be accepted. The overall performance of the application will be good in spite of poor allocation of one instance of a task. The scheduling approach that places tasks on completely idle machines [21] is overly conservative. Large parallel computers are rarely idle. A workstation being used to view e-mail or perform other casual processing can host background tasks, especially when they are run at low priority, without impacting the foreground user.

The scheduling goal of the VCE is to harness the excess capacity of lightly loaded machines without placing additional burdens on heavily loaded machines. A more formal definition of heavily and lightly loaded is needed. A machine is heavily loaded if its users notice poor performance. The load of such a machine is referred to as being over its *nuisance threshold*. A machine that would be below its nuisance threshold even after the addition of one more task or process is below its *adequate utilization threshold* and is considered lightly loaded (see Figure 27).

The load measurement routines for machines in the VCE employ an algorithm described in Figure 28. The algorithm computes a smoothed load measure ($Smoothed_load(t_i)$) as well as an instantaneous load measure ($Load(t_i)$). The instantaneous load measure is based on frequent snapshots of the local load conditions. Both the instantaneous and smoothed load measures can take any of the three values low, medium or high as indicated in Figure 27. When the smoothed load measure becomes low the group leader is notified that the machine is available to host remote tasks. When the smoothed load measure goes from low to medium the group leader must be informed that the machine is no longer in need of more work. A smoothed measure is used because the instantaneous measure may fluctuate due to the transient effects such as process initiation. It will also fluctuate when the load is very close to the threshold. The smoothed measure is not changed until several consecutive changes in the

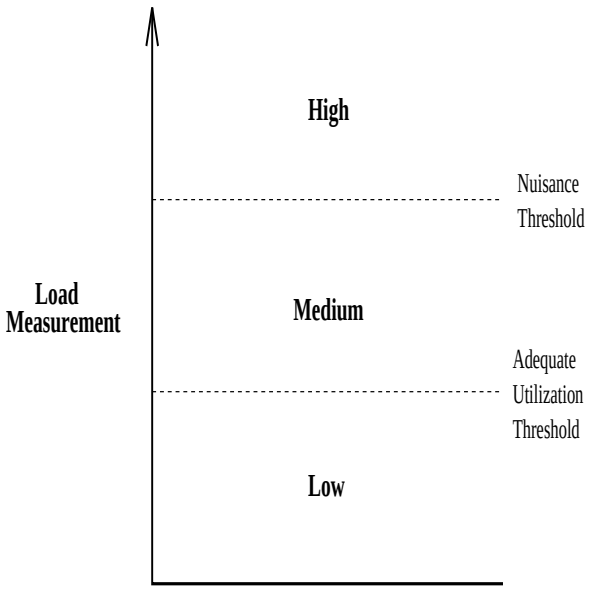

Figure 27: VCE Load Thresholds

instantaneous measure in the same direction are observed. Smoothing also allows for the introduction of a bias. That is, the criterion for decreasing the smoothed measure can be constructed to be more stringent then the criteria for raising it. This further decreases the probability that work will be sent to a machine whose load is approaching the nuisance threshold. The sampling interval and the criterion for translating between an instantaneous and smoothed load measure must be customized for different types of machines. Examples of values used in a workstation cluster are discussed in Chapter 6.

The VCE load measurement algorithm (Figure 28) can be customized for particular platforms. For instance, the measurement interval can be modified based on factors such as the desired precision and the frequency which load statistics are updated by the operating system. Also, the definition of low, medium, and high load levels can be adjusted by system administrators. Lowering these thresholds reduces the chance that non-VCE users will experience performance degradations and reduces the processing capacity available to VCE applications. Chapter 6 will discuss possible settings of the algorithm parameters for various platforms.

```
do forever

   sleep 1 Sample_Interval

   compute Load(t_i)

   if Load(t_i) = Smoothed_load(t_{i-1})
      then continue

   if Load(t_i) > Smoothed_Load(t_{i-k}) for k = 1...N
      then Smoothed_Load(t_i) = Load(t_i)

   if Load(t_i) < Smoothed_Load(t_{i-k}) for k = 1...M
      then Smoothed_Load(t_i) = Load(t_i)

   if Smoothed_Load(t_i) ≠ Smoothed_Load(t_{i-1})
      then notify group leader if a period
      of low load has begun or ended

end
```

Figure 28: Load Measurement Algorithm

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

Figure 29: Load Measurement Algorithm Example

An example of how the algorithm behaves is shown in Figure 29. The example uses a sampling interval of six seconds, an upward bias, N, of two, and a downward bias, M, of ten. These values are appropriate for a Unix workstation (see Section 6.1.3). At time 0 both the instantaneous and smoothed load measures indicate the load on the workstation is medium. The changes in the instantaneous load measure that occur between time 12 and time 30 are considered noise and are not reflected in the smoothed load measure. The ten consecutive low instantaneous readings beginning at time 36 cause the smoothed load measure to become low at time 90. The two consecutive instantaneous readings of medium at time 120 and 126 cause the smoothed load measure to return to medium.

The instantaneous load measure is derived from measurements of the amount of contention for system processor [69] and I/O facilities. Users will perceive poor performance if there is high contention for either the processor or I/O facilities. The instantaneous load measure reflects the higher of the two factors (processor and I/O).

Information Exchange Policy

Objective

The frequency of load information exchange is a central design parameter in dynamic task scheduling. Excessive exchanges can contribute to network congestion. Infrequent exchanges lead to decisions based on inaccurate data. Information exchange is especially important in systems that use process migration. Migrating a running process from one host to another requires significant processing from both the source and destination system. It can also generate a large amount of network traffic. In such an environment extensive information exchanges may be required to insure that costly process migrations will actually result in better overall system performance.

Common Approaches

A wide variety of information exchange policies have been discussed in the literature. Information exchanges can be performed periodically [9, 21] or only when a machine detects that its local load is extremely high or low [34]. Information may be

exchanged with neighboring nodes [84], randomly selected nodes [9, 34] or a central manager [21]. These parameters tend to be established experimentally [9] and are influenced by the amount of scalability desired (the neighbor based approach of [84] is extremely scalable, the centralized system of [21] is intended for use only in local area networks).

Redundant Task Scheduling Approach

The VCE information exchange policy is incorporated into the load measurement algorithm (Figure 28). Machines disclose their loads to their group leader only when a change is detected and persists for a significant period of time. A method for recovering from group leader failures will be presented in Section 5.2.7. Redundant scheduling reduces the need for frequent information exchanges because the penalty for a poor task placement decision is small.

Transfer Policy

Objective

In most distributed systems a transfer policy determines which processes are migrated and when they are migrated. The policy seeks to identify which migrations are most likely to improve workload balance. For instance, migrating a task that is about to complete is wasteful. Determining which migrations will be beneficial is difficult in a network based computing environment.

Common Approaches

Transfer policies are often the most complex part of a load balancing algorithm. Careful design of a transfer policy is essential because an incorrect transfer policy can degrade both system and application performance. For instance, transferring a task just before it completes is counterproductive. Thrashing behavior, where the same job is constantly transferred and is prevented from making progress, must also be prevented [9].

Many ways of characterizing transfer policies are possible. One of the most useful delineations is between sender-initiated and receiver-initiated policies [36]. With a sender-initiated policy a machine which, based on its estimation of the global system

state gleaned from the information exchange policy, determines that it is overloaded, will select a less loaded host to transfer work to. Receiver-initiated policies require lightly loaded nodes to seek out a heavily loaded machine to transfer work from. Because receiver-initiated policies are more efficient in heavily loaded systems and sender-initiated policies are better otherwise [36], research continues into hybrid systems which will provide good overall performance [15, 77].

Another important element of a transfer policy is whether it supports preemptive or non-preemptive transfers. In non-preemptive systems only newly arriving jobs or task queued for execution are candidates for transfer. Non-preemptive systems have less flexibility because fewer transfer options are available [109]. However, preemptive transfers are usually not possible in heterogeneous systems [10, 21].

Various mechanisms have been designed to insure stable transfer policies and prevent thrashing. Commonly a limit is placed on the number of times any single process can be transferred. In preemptive systems a a minimum time can be established for which a process must reside on a particular machine before it can be transferred [9]. This insures that a job will make some progress before each transfer and also prevents the transfer of small tasks.

Redundant Task Scheduling Approach

The VCE employs a non-premptive sender-initiated transfer policy. When a stage of an application becomes dispatchable the execution process obtains from the group leaders lists of lightly loaded machines, Task instances are then transferred to those machines for execution. A sender-initiated approach is used because the number of execution processes, and therefore the number of available tasks, is expected to be much smaller than the number of lightly loaded machines in a wide area network. It would not be practical for lightly loaded machines to attempt to see out execution processes.

The Virtual Computing Environment is inherently heterogeneous, so preemptive process transfers are often infeasable. Some of the characteristics of preemptive process transfer are retained nonetheless. If a task is placed on a machine and the load on the machine increases, the local operating system will reduce the resources allotted

to the task. That is, as the load on a time-shared system increases, the resources available to each task decrease. Since the slower instances of a task are cancelled when the fastest instance completes, the impact of VCE task instances on heavily loaded machines are further reduced.

Running VCE tasks at low priority on privately owned workstations further mitigates the need for a preemptive transfer mechanism. The resources consumed by a low priority task on a heavily loaded machine are minimal [8], so the behavior of the local operating system in starving a low priority VCE tasks as the local load increases has an effect similar to preemption.

Location Policy

Objective

Nodes in a distributed system must cooperate to determine the best location for each task. Once a task is selected for migration by the transfer policy, the location policy selects a target computer for the migration. To maximize the chances that a process will benefit from a migration, the receiving machine must have significantly less load than the sending machine.

Common Approaches

Location policies usually involve some type of threshold value for the difference between the load of the sending and receiving machines. The method for selecting candidate machines depends on the information exchange policy in force. In systems where machines communicate only with a fixed set of neighbors, a heavily loaded machine would transfer a job to its least loaded neighbor if the difference in their loads is significant [84]. Other systems employ random communication patterns [9]. In these systems an overloaded or underloaded machine would randomly probe other machines until a suitable transfer target was found [15].

Redundant Task Scheduling Approach

The location policies needed for redundant task management are simplified. They focus on selecting lightly loaded machines to host the initial placements of redundant task instances. The overhead associated with determining the optimal targets for

migrating running processes of an overloaded host is eliminated.

An execution process uses hints to estimate the computational requirements of the tasks to be placed and determines which group it would prefer to host each task. This decision is based on several factors such as what executables are available and each group's communication distance from the originating computer.

The execution process informs each group leader how many tasks it wishes to place in the group. For supercomputer or mainframe groups, which will rarely contain more than two or three members, the group leader will send to the execution process the address of each lightly loaded machine in the group. For workstation groups the execution process tells the group leader the maximum number of machines it can use. The workstation group leader provides the addresses of the requested number of lightly loaded machines, or the addresses of all lightly loaded machines if less than the maximum number are available.

If the execution process can achieve adequate redundancy within a single group, it will do so. If intergroup redundancy is needed the execution process will employ it.

To run a task on a machine provided by a group leader, the execution process contacts the daemon on the target machine directly and sends it the information necessary to establish an execution context. On privately owned workstations, the daemon's dispatcher starts the task with the lowest possible priority to insure it will only consume excess resources on the machine.

Once a task is placed, no further communication or cooperation is needed. Redundant task scheduling algorithms only need to identify a set of computers which are likely to perform well. By running instances of the task on several computers in this set the performance of the best machine is received without the burden of selecting one individual machine.

5.1.3 Overview of Scheduling Activities

Figures 30, 31 and 32 give an overview of the scheduling activities performed by each type of VCE component shown in Figure 20 (execution processes, group leaders and

group members).

5.1.4 Rationale

Most scheduling systems have one of two goals. They either: (1) manage all the tasks in a distributed system to improve overall performance [9, 67]; or (2) detect idle computers and make their resources available for running batch jobs or parallel programs [6, 21, 60]. The VCE is in the latter category, but its objectives are significantly different from past systems. The VCE seeks to harness available resources scattered across a wide area network and use them as a platform for running very large applications. Most existing systems do not scale well beyond a local area network and have little or no support for heterogeneous systems. Since wide area networks will contain an enormous variety of heterogeneous machines, the VCE supports an application model which allows the user to select a different set of candidate machines for each stage of an application. At run time the VCE attempts to locate machines suitable for each portion of the application. This approach may be a very attractive option for scientists and engineers whose applications require multiple high performance computers with disparate architectures. Accessing such machines remotely is the only way many users can exploit them, so a scheduling system which automatically locates underutilized system and manges the scheduling and dispatching of tasks on them may be valuable.

Because the goals of the VCE are distinctive, its scheduling approach has unique requirements. The scheduling system must avoid creating excessive traffic of scheduling information between widely separated machines. This is accomplished by having machines only disclose load information to group leaders. When A VCE application starts, the execution process contacts only those group leaders whose groups are architecturally suitable for the tasks being dispatched. Geographically near by groups are queried before more distant machines are considered. These measures further reduce network traffic.

```
for each dispatchable task

   determine preferred group

   obtain list of lightly loaded machines from group leader

   if not enough machines were provided
      then select another group

end

for each selected computer

   send dispatching information

end

for each completed task instance

   cancel redundant instances

   determine if any other task has become dispatchable

end
```

Figure 30: Execution Process Scheduling Activities

```
for each load update message received
   update list of lightly loaded machines
end
for each execution process placement request
   determine group capacity
   send list of available hosts to execution process
end
```

Figure 31: Group Leader Scheduling Activities

```
do forever

  sleep 1 Sample_Interval

  compute current Smoothed_Load (Figure 28)

  If Smoothed_Load changed from low to medium
    then notify group leader

  If Smoothed_Load changed from medium to low
    then notify group leader

end

dispatch tasks as directed by execution processes

terminate tasks as directed by execution processes

notify execution processes of task completions when they occur
```

Figure 32: Group Member Scheduling Activites

Redundant scheduling is the simplest way to achieve high performance in a heterogeneous environment. Process migration of jobs between parallel computers is not feasible. Canceling and restarting tasks is costly and can result in poor application performance. This is especially true when an application employs multiple communicating tasks in a single stage. Cancelling one task may require rollback of an entire set of tasks. By placing redundant instances of tasks on machines which are all lightly loaded when the tasks are dispatched, the chances of high performance are maximized.

High levels of redundancy provide only minimal performance improvement beyond more reasonable levels of three or four instances per task. This will be seen in the experimental and simulation data of Chapter 6. The reasons for this are straight forward. Consider an execution process with one task to dispatch and a large group of identical initially idle hosts it can choose from to place redundant task instances. Given the assumption that all the hosts are identical, the performance of the task will only be influenced by differences in load conditions among the hosts. Since all the hosts are assumed to be idle when the placement decision is to be made, the execution process has no basis to expect better performance from any particular host, so the first task instance can be placed at random. Assuming that load fluctuations on the machines are independent, there is a 50% chance that a second host, randomly selected from the remaining machines, will perform better than the first. There is a only a 25% chance that a third machine will be superior to both of the first two. The probability of realizing a performance improvement continues to drop exponentially as the number of machines chosen increases. Of course, real environments are more complex and more variables, such as the amount of speedup achieved when a superior host is selected, must be considered. However, the general principle that the performance returns of redundant scheduling decline rapidly as redundancy levels increase, still holds true.

The largest shortcoming of redundant scheduling is its cost. Running several instances of a task increases processing requirements because all task instances will consume some CPU resources. Only the results of the fastest task instance are made

available to the user or used for further processing, so resources used by slower task instances do not benefit the application. Communication costs also are incurred when inputs must be duplicated and sent to each instance of the target task (Section 5.2). These costs are a further argument against high redundancy levels. For low redundancy levels, these costs are acceptable. Alternative methods for dealing with load fluctuations such as migrating an entire execution image [21] or disassembling, migrating and recompiling a task [102] also introduce considerable overheads.

5.2 Managing Communications Within the VCE

The VCE communication system provides a medium for tasks of VCE applications to transmit and receive partial results to and from other tasks participating in the same application. VCE tasks are expected to be large grain computational components. Even though the running times of such tasks are not dominated by continuous communication, the tasks may have to communicate large amounts of data from time to time (for example, when one stage of a large application completes and its successor begins). Managing these communications are complicated by two factors: (1) application tasks may be redundantly dispatched, so a single message could have multiple senders and multiple receivers, and (2) tasks of a single application could be running on systems which are geographically dispersed. The VCE manages these issues so as to achieve high transparency for both application designers and task implementers. As long as the task implementer realizes high granularity and (whenever possible) overlaps communication with computation operations, and the application designer correctly specifies the application, the VCE delivers high performance on the platforms available at run time.

In addition to supporting intertask communication, a variety of other communication activities are managed by the VCE. Scheduling/dispatching daemons disclose load information to their group leaders. Execution processes locate group leaders

and collaborate with them on task placement decisions. Once task placement decisions are made, the execution processes send dispatching information to the scheduling/dispatching daemons and receive completion notices and other information from them. All these communication operations have different characteristics. They differ on the basis of the proximity of senders and receivers, and the frequency and amount of data to be exchanged. In order to achieve high overall performance, different communication technologies are used for different communication activities. The design of the communication system provides high modularity, so that as new communication technologies are introduced they can be incorporated into the VCE with minimal effort.

5.2.1 VCE Application Communication Issues

The requirements of VCE application intertask communication differ from those of other distributed environments. For each application task the VCE may dispatch several redundant instances. This means that when a task produces an output data object it must be duplicated and sent to all instances of each task designated in the application description as a recipient of the object. When a slower instance of the sending task produces the same data object it must not be presented to the receiving task instances a second time. Further, the system must insure a proper level of consistency among intertask communications. If an application consists of two tasks which exchange data periodically, it is possible for the fastest instances of both tasks to run several exchanges ahead of slower instances. In such a situation the VCE's communication protocols must insure that all task instances receive their input in a coherent and transparent manner.

Before considering the methods used to support intertask communications it is important to examine the characteristics of such communications. Each VCE task represents a major, course grain, high performance component of a large scale (grand challenge) application. Communication among these tasks commonly take one of two

possible forms. Often intermediate results of an earlier application phase are communicated to later tasks. This was illustrated by the linear equation solver application described in Section 4.2.6. Alternatively, application components with different computational requirements may need to run on different machines and exchange data from time to time. The global climate simulation application described in [74], which is structured as separate global atmospheric and ocean models that communicate intermittently, employs this style of communication.

Most of the research on communication support for redundant processes concentrates on supporting broadcast communication [18, 27, 47, 107]. The broadcast model allows individual processes to join a *broadcast group*. Messages addressed to the group are received by all member processes. Different systems provide different levels of support [47]. Some simply guarantee that all messages sent to a group will be received by all group members. Others stipulate that messages sent to the group from a single source will be received by all members in the same order. The strongest level of support involves an assurance that all group members see all messages in the same order. Of course, systems that support stronger consistency criteria require greater complexity and suffer reduced performance [18]. Several techniques are commonly employed by these protocols. Time stamps or sequence numbers can be appended to messages by the sender so the receiving node can process them in the correct order. All broadcasts to a group can be routed through a leader node which forwards them to the member nodes. Or precedence histories can be included with messages to indicate the correct delivery order. Most broadcast protocols use some combination of these techniques.

The requirements of the VCE communication system are somewhat beyond the scope of broadcast protocols. These protocols are concerned with systems where message senders need to reach multiple receivers with a single message. That is, each message has a single source and multiple destinations. The challenge faced by the VCE is to manage communication among redundant task instances where a single message can have multiple senders as well as multiple receivers. This situation has been researched in the past as it relates to process replication in the fault tolerance

domain. In some fault tolerant systems process replication is used as an alternative to checkpointing and roll-back/recovery schemes [5, 31]. In such systems, as in the VCE, each application process can have multiple replicated copies running simultaneously. Of course, input to such a process presents similar issues to those encountered by the broadcast environments described above. However, output handling is more complex because each replicated process will generate identical output, only one copy of which can be presented to each instance of the receiving process. The most common approach to handling this situation is to designate one instance of each process as a leader or coordinator. Output can be routed through the leader which would discard redundant output [38] or the leader can be consulted for permission to send output, which would be denied if the output has already been sent [31]. In either case, performance is constrained by the dependence of all process instances on services provided by only one. These schemes also necessitate special procedures to allow the process instances to recover from the failure of the leader process.

The VCE uses a specially designed protocol to support communication among redundant tasks. The protocol, which is outlined in Section 5.2.3, exploits the unique properties of VCE applications and the VCE run time environment to allow efficient communication with a simpler protocol than would be required for a more general-purpose use.

5.2.2 Components of the VCE Intertask Communication System

The semantics of VCE application construction were outlined in Section 4.2. Task implementations use three simple functions (Section 4.2.5) to perform intertask communications. The execution process uses information in the task and application descriptions (Sections 4.2.4 and 4.2.6) to determine how these communications are to be routed. Each VCE task instance is supported by a VCE communication daemon. The communication daemon is dispatched by the scheduling/dispatching daemon before the task instance is dispatched (Figure 33). At compile time the VCE communication

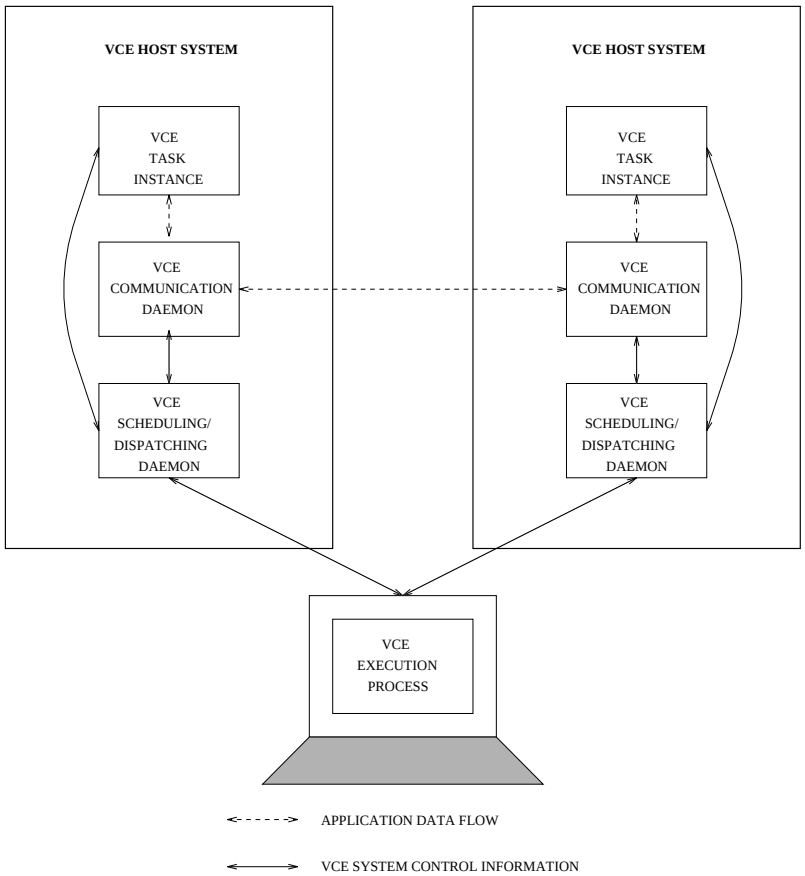

Figure 33: The Function of VCE Communication Daemons

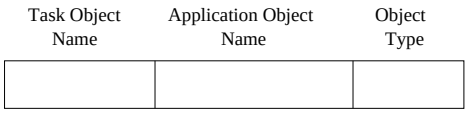

Figure 34: Input Data Table Entry Format

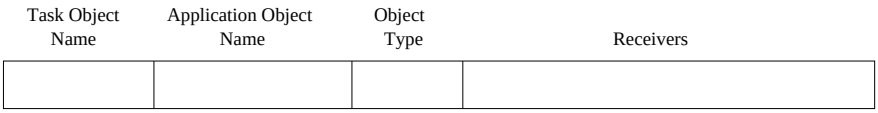

Figure 35: Output Data Table Entry Format

functions are replaced by routines which perform efficient interprocess communication, such as Unix shared segments [96], to exchange data with the communication daemon. The communication daemons do not run in the same address space as the task instance because the communication daemon, in order to ease overlapping of communication with computation, requires an independent thread of control. The communication daemon also needs to exercise memory management independent of the task instance.

When a communication daemon is instantiated it receives from the execution process (by way of the scheduling/dispatching daemon) an *input data table* and an *output data table*. The input data table indicates, for each object that the task instance will receive as input, the name and type that the object was declared with in the task description and the name which it was overridden with in the application description (Figure 34). The output data table contains equivalent information. Additionally, an output data table entry also contains the addresses of all the communication daemons supporting tasks which the output object is to be forwarded to (Figure 35).

5.2.3 Managing Communication Among Concurrent Redundant Tasks

Section 5.1 described how the VCE uses redundant task scheduling to improve application performance. In an environment where the same output object could be

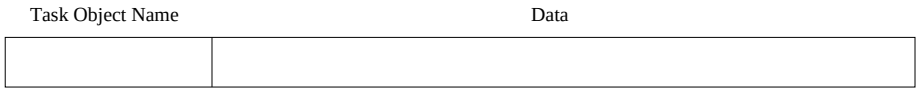

Figure 36: Task Instance Output Message Format

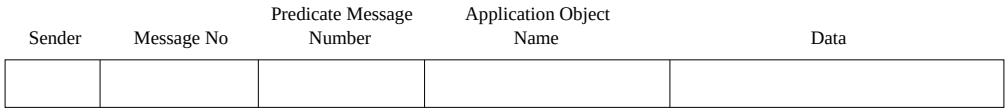

Figure 37: Format of Data Sent Between Communication Daemons

produced by several redundant instances of a single task, the processing required of communication daemons are more complex than simply forwarding output and receiving input on behalf of a task instance. For instance, it is the communication daemon's responsibility to insure that input is presented to each task instance in a coherent manner. Not all the instances of a single task will progress at the same rate. When there are two or more communicating tasks, and each task may have multiple instances, intertask communication must be managed so as to avoid *presentation of duplicate input* and *presentation of input which violates coherence.*

Detecting Duplicate Input

Presentation of duplicate input is a straight forward problem. When a task instance uses the **VCE_send()** function to make a data object available to other tasks, the communication daemon forwards copies of the object to all task instances which require the object as input (as indicated in the application description). At some later time another redundant instance of the same task running on a slower machine (due either to load conditions or processor limitations) may produce the same output object. In the VCE communication system such duplicate messages are detected and discarded by the receiving communication daemon. The algorithm for detecting duplicate data is described below.

When the **VCE_send()** function is executed it builds a data structure which contains the name and contents of the object being output (Figure 36). This data

structure is presented to the communication daemon through shared memory [96] or some other interprocess communication method. The communication daemon uses the output data table to determine what the object name has been overridden with in the current application and where task instances which will receive this object are located. The communication daemon also keeps a count of how many objects it has sent to each receiver. This count enables receivers to detect duplicate messages. The format of a message used to exchange a data object between instances of different tasks is shown in Figure 37. The *Sender* field is filled in with the name from the application description of the application stage producing the output. The *Message No.* field holds the total count of messages sent by this application stage (task) to the receiver. When two instances of the same task produce the same output, the receiving communication daemon will see the same *Sender* and *Message No.* on each. By logging the *Sender*s and *Message No.*'s on input objects as they arrive, duplicate input can be detected and discarded. This methodology implies that extraneous messages are going to be sent across platforms only to be discarded as they arrive. A method for limiting such extraneous traffic will be presented below.

Maintaining Communication Coherence

Redundant communication can lead to incoherence in the following situation. Suppose an application has two concurrently executing tasks, A and B, where A sends an object (a request) to B and receives another object (a result) in return. When the fastest instance of A sends its first request and the fastest instance of B returns a result, the result may arrive at a slower instance of task A before it has made its first request. The communication daemon for the slower instance of task A must not present the result object from task B until the task instance has sent the corresponding request. This is resolved using the *Predicate Message Number* field shown in Figure 37. In this example, the communication daemon for the slower instance of task A would find a 1 in this field of the message sent from task B. The 1 indicates that task B received one object from some instance of task A before sending the message. Since the communication daemon knows that the instance of task A it is

supporting has not sent any objects to task B, it will buffer the input until it can be presented in a coherent fashion. This also enables the communication daemon to discard extraneous output rather than sending it. Since the communication daemon for the slower instance of task A knows that task B has already received one object from a faster instance of task A, when its task instance trys to send an object to task B the communication daemon can discard the duplicate output rather than sending it.

The Communication Daemon Algorithm

When a communication daemon receives input it must determine if the input can be presented to the task instance immediately or must be buffered. Additionally, if the communication daemon can determine that an output object being produced by the task instance it is supporting has already been sent by a faster instance of the same task, then the output can be discarded rather than being sent. The processing required of a communication daemon in managing the communications of the task instance it is supporting (called the *local task instance* for clarity) is summarized in Figure 38 and discussed below.

The communication daemon must maintain one (possibly empty) input buffer for each task sending input to the local task instance. The processing shown in Figure 38 pertains to the maintainence of one such buffer. When input is received by a communication daemon, the *Predicate Message Number* field of the message is checked to determine whether the input may be dependant on any output not yet generated by the local task instance. If not, then the input can be presented to the local task instance or discarded if another copy of the input was previously received from a faster task instance. When a newly received input object is dependant on output data not yet produced by the local task instance, it is added to the appropriate input buffer. Whenever output is produced by the local task instance, the input buffers associated with each task receiving the output are examined. If any predicates associated with buffered inputs are satisfied by the output object, those inputs can be presented. When an output object satisfies the predicate of a queued input message, the output

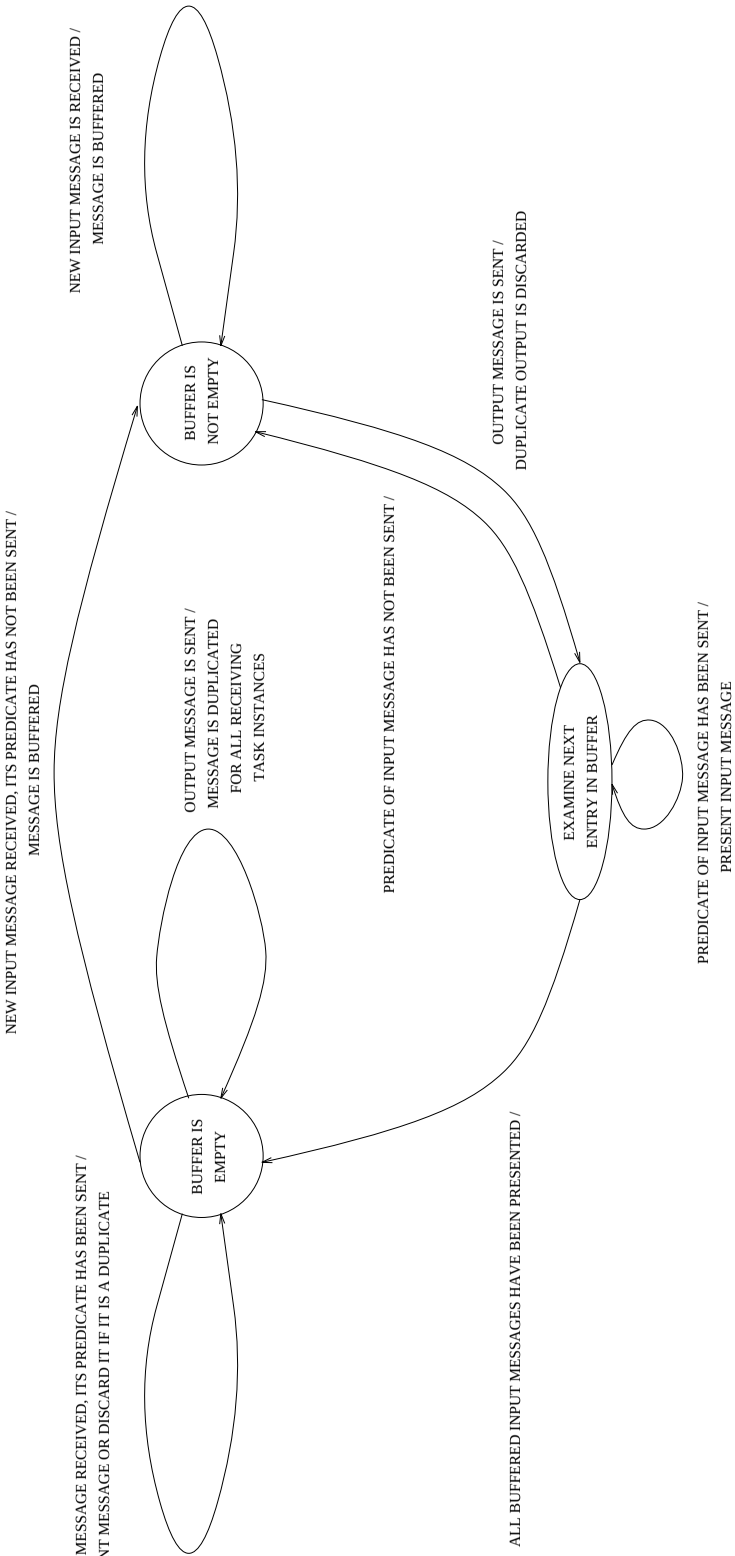

Figure 38: An Algorithm for Managing Communication Among Concurrent Tasks

must be a duplicate of an object produced by a faster instance of the local task. In this case the output object is discarded rather than being sent.

Limitations of the Communication Daemon Algorithm

The algorithm discussed above is not a general purpose algorithm for communication among replicated processes. It is specifically intended to meet the most common needs of VCE applications with maximum efficiency and minimal complexity. The purpose of intertask communication is to allow application components to transmit intermediate results to each other. The VCE application model (Chapter 4) supposes that tasks will use the **VCE_receive()** function to wait for a set of inputs to become available. Then they will use these inputs to perform a computation and invoke the **VCE_send()** function to return the results. In such an environment, the order in which input objects are received is not consequential. That is, if a task needs object A and object B to perform its computation, the application will function correctly even if some instances of the task receive A first while others receive B first. Therefore, the algorithm does not include complex processing to impose such consistency [18].

The algorithm does not support all possible intertask communication patterns. Figure 39 shows five possible patterns. In Figures 39a, 39b, 39c and 39d the VCE communication algorithm does guarantee that no instance of any task will receive input that may be dependent on output it has not yet generated. However, this assurance does not hold for the communication pattern of Figure 39e. That is, the algorithm cannot insure coherent communication in the presence of indirect recursive communication. In Figure 39e TASK A may send data to TASK B which would generate a data object to be consumed by TASK C. If, as a result of this, TASK C returns data to TASK A, some instances of TASK A might see the final result before they have produced the output that generated it. This occurs because the message sent by the communication daemons for TASK C to the communication daemons for TASK A would contain a zero in the *Predicate Message Number* field (Figure 37). So it would appear (incorrectly) to the communication daemons supporting TASK A that the input from TASK C is not related to any prior output of TASK A.

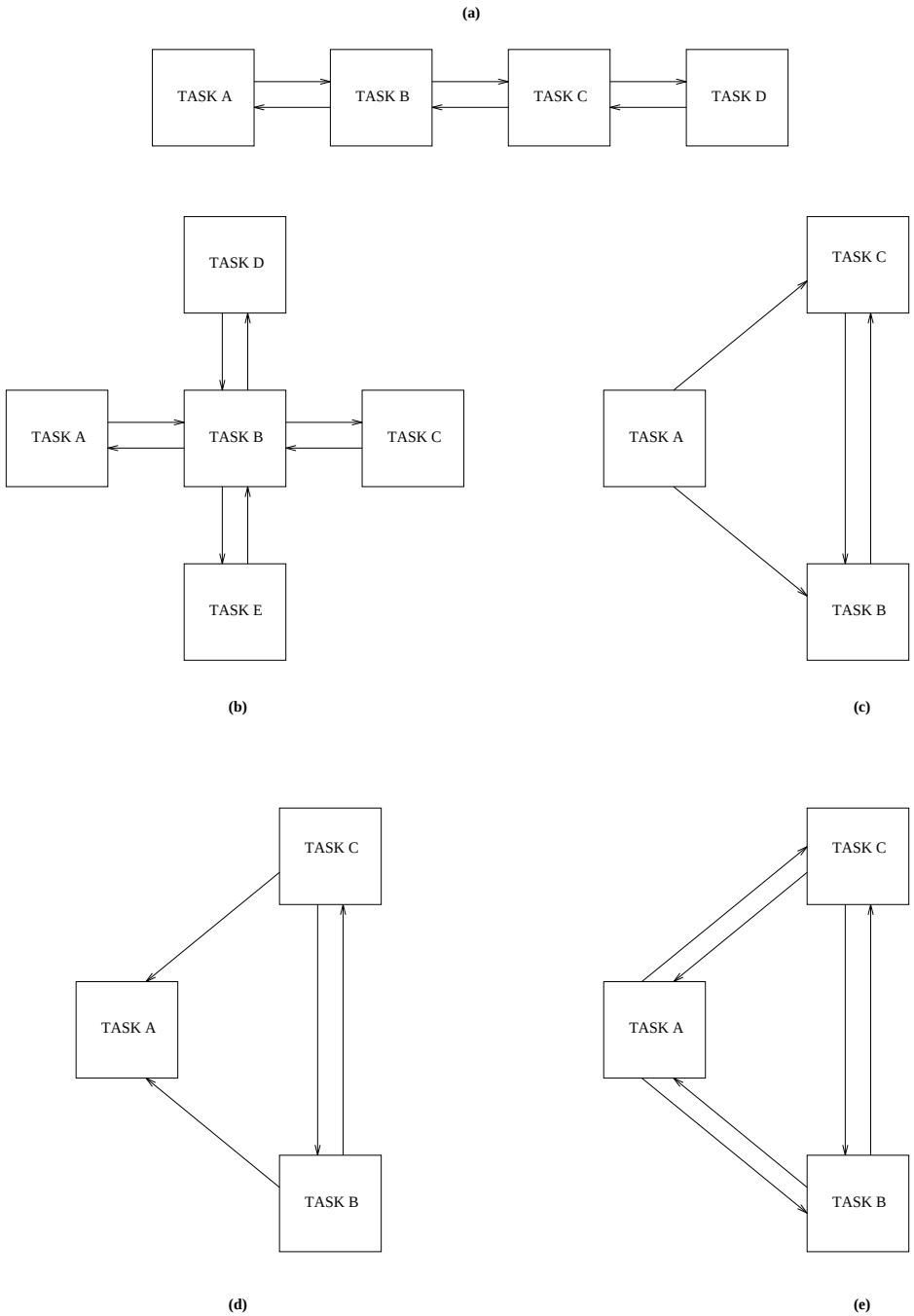

Figure 39: Intertask Communication Pattern Examples

This limitation of the algorithm could be overcome by extending the *Predicate Message Number* field shown in Figure 37 to contain a precedence graph [47]. This data structure would identify all the input received by a task before it generated a data object. Further, the precedence graph would incorporate information included in the precedence graphs of all those prior inputs. Essentially, it would identify all the data that a particular object could have any dependence on. On receiving an input object, the communication daemon would parse the precedence graph and determine if its task instance had not yet created some object that the newly received object might have been derived from. If so, the input object would be buffered until its precedents were satisfied.

It is unlikely that such an elaborate communication algorithm will be warranted. An application which employs indirect recursive communication is likely to be poorly structured and error prone. The software engineering tools in the VCE's software development suite, which are used to formulate the application description, will detect such a situation, alert the user to its presence, and help him or her to replace it with a more well structured application design.

Capabilities of the Algorithm

In spite of the limitations mentioned above, the communication algorithm of Figure 38 is well suited for managing communication among concurrent VCE tasks. One advantage of the algorithm is that a communication daemon supporting a particular task instance does not need to communicate with any communication daemon supporting instances of the same task, as would be required by the more robust algorithms of [31] and [38]. A communication daemon does not need to have any information about the number of redundant task instances dispatched or their locations. Since task instances may be dispatched on geographically dispersed platforms, requiring communication daemons to consult with each other before broadcasting their output could be costly. Also, the chief objective of redundant scheduling is to enable the application to progress at the rate of the fastest machine available. Making one communication daemon the "leader" and requiring it to coordinate all the input and

output for a set of task instances [31] is likely to introduce a costly bottleneck.

5.2.4 Managing Communication Among Predecessor and Successor Tasks

The previous section discussed the methods used by VCE communication daemons to manage communications among concurrently executing application components (tasks). A different protocol must be used to accommodate the exchange of objects between tasks in one application stage and tasks in later stages (Figure 26). This situation is evident in an application description when the **synchronization** specifications indicate that one stage cannot begin until another completes, and the output of the earlier stage serves as input to the later stage. When such a situation is detected by an execution process, it informs the communication daemons supporting the earlier stage that the output objects are to be buffered. This is indicated by placing a special value in the *Receivers* field of the output table entry (Figure 37) corresponding to each output object to be buffered.

Whenever any instance of a VCE task completes, all other redundant instances of the task are terminated. The communication daemons supporting those task instances are also terminated and any (possibly incomplete) data maintained in their buffers is discarded. This means that the output objects needed by the later application stages are maintained only by the communication daemon of the task instance that ran to completion. When all the synchronization constraints of the successor stage are satisfied, the execution process determines where its task instances are to be placed. After the successor task instances and their communication daemons are instantiated, the execution process sends the communication daemon holding the buffered object(s) an updated output table indicating where they are to be sent. Once all the buffered objects have been delivered to the successor tasks the surviving communication daemon of the predecessor task instance is also terminated.

This procedure further minimizes the amount of extraneous message traffic produced by redundant dispatching. In the case of predecessor to successor communications only one communication daemon transmits output to the successor task instances, regardless of the number of predecessor task instances that were dispatched.

5.2.5 Managing Communication Among VCE System Components

Sections 5.2.1, 5.2.2, 5.2.3 and 5.2.4 dealt with techniques for managing the communication of data objects among the various tasks that make up a VCE application. Execution of VCE applications also requires exchange of various types of control information among the components of the VCE itself. Providing efficient mechanisms to manage this communication also contributes to the robustness of the VCE design.

Figures 40 and 41 show schematicly how different VCE communication activities exploit different technologies. Table 3 summarizes the characteristics of these communication activities. The gist of the information in Table 3 is that communication among communication daemons is the situation that most requires the use of high performance communication technology.

The scheduling methods discussed in Section 5.1 minimize the amount of load information that must be transmitted across wide area networks. This information is gathered by an execution process only when it is needed to make a task placement decision. Once the execution process has selected machines to host instances of a task, it relays dispatching information to the scheduling/dispatching daemons on those machines. The dispatching information includes the location of the executables that correspond to the tasks to be run. Task executables may be copied to remote machines ahead of time by the software development suite to avoid a delay at run time. The execution process may also look ahead in an application description to determine where candidate machines to host subsequent application stages are available. If some of these machines do not have local copies of the requisite executables, the execution process can transmit them as earlier application stages are running. This

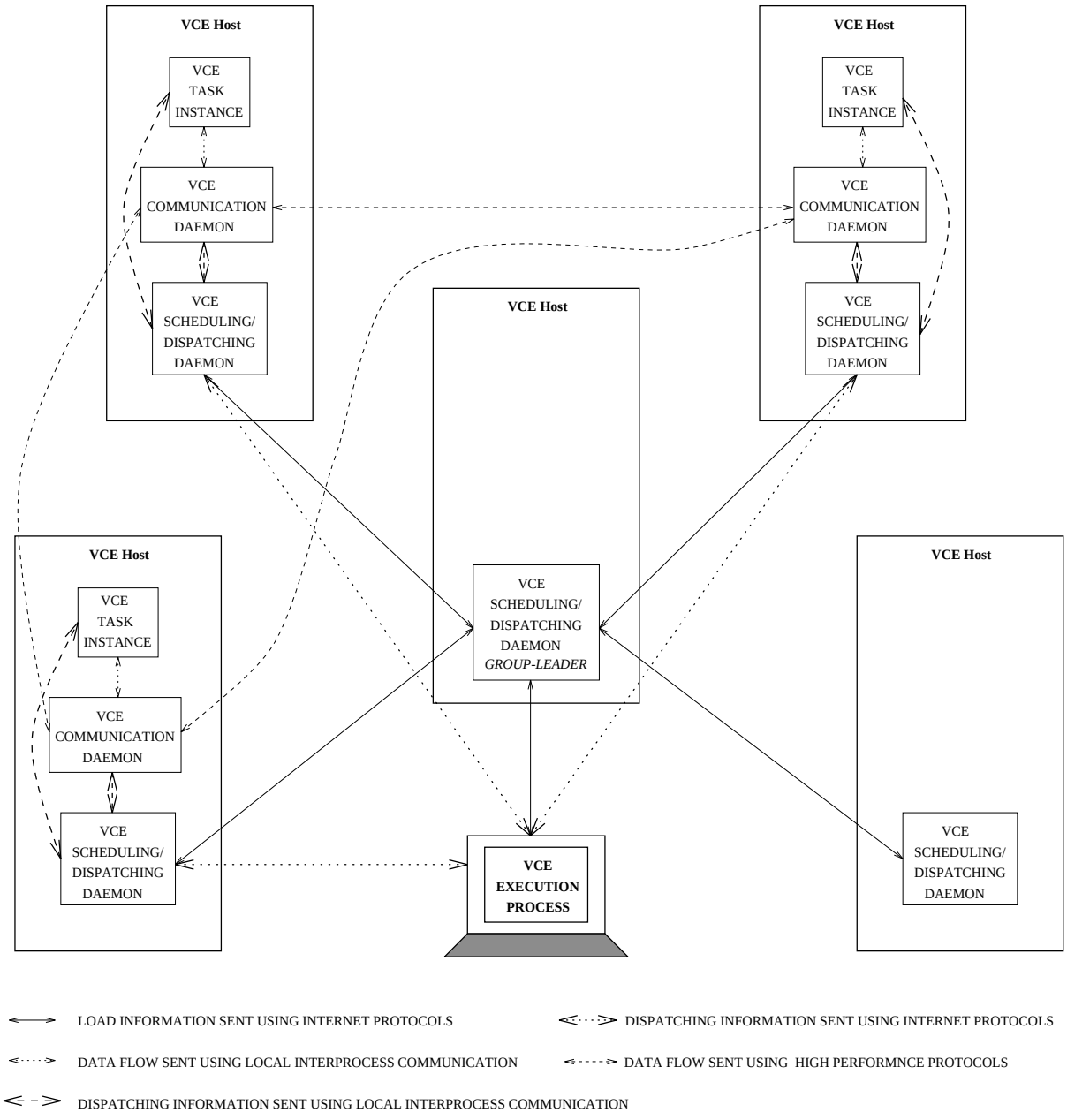

Figure 40: Components of the VCE Communication System

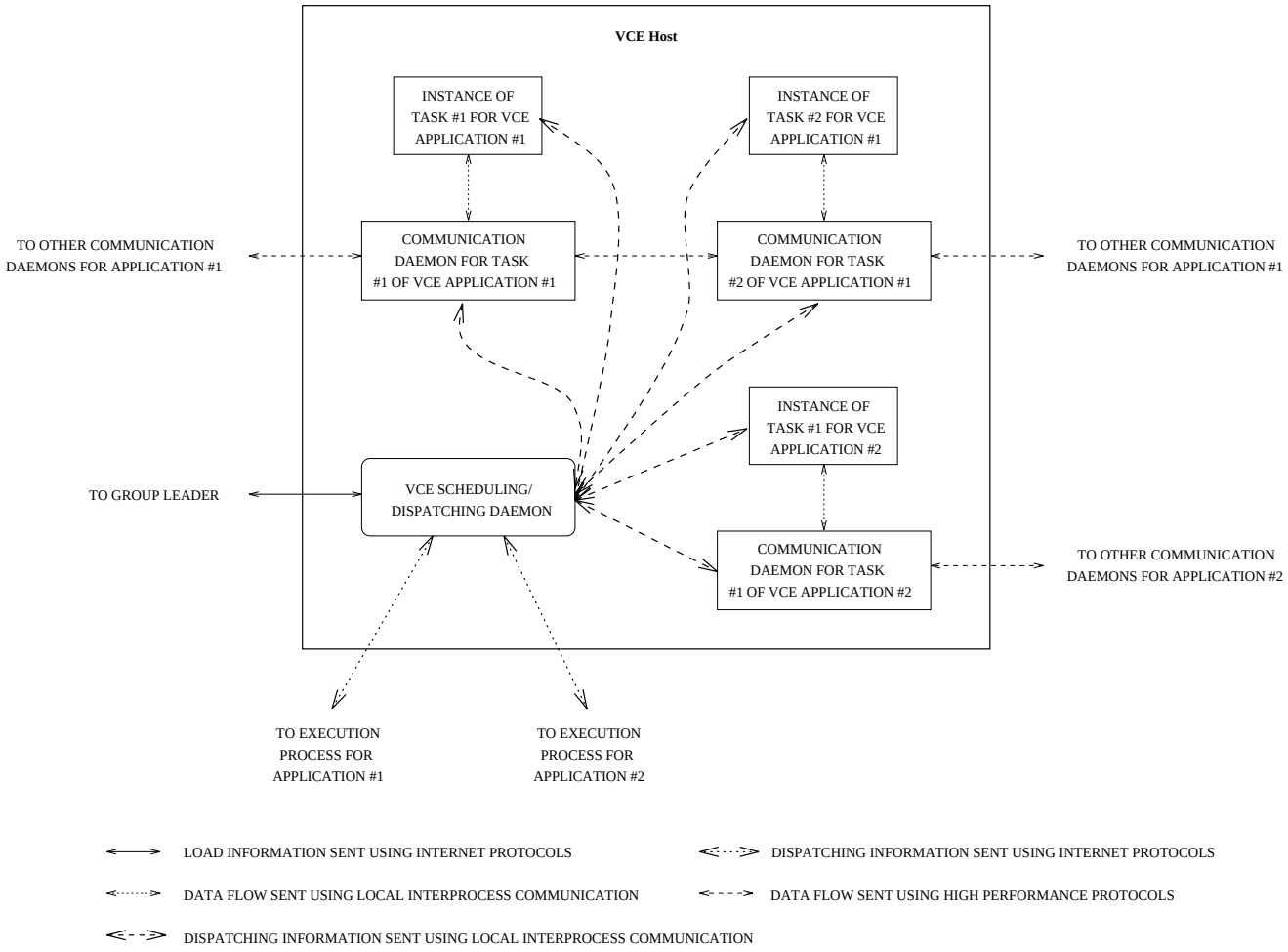

Figure 41: Communication to Support Multiple VCE Tasks on a Single Host

Table 3: Communication Characteristics of VCE Components

Parties	Geographical Seperation	Frequency	Size
Group Leaders and Scheduling Daemons	Small	Less than twice a minute	10's of bytes
Execution Processes and Group Leaders	Great	At the beginning of each application stage	10's of bytes
Execution Processes and Scheduling Daemons	Great	At the beginning and end of each application stage	100's of bytes
Scheduling Daemons and Communication Daemons	None	At the beginning and end of each application stage	100's of bytes
Communication Daemons with Each Other	Great	Intermittently or continuously	Terabytes

affords an opportunity for overlapping VCE communication activities with application processing.

Executables for several different types of communication daemons reside on the local file systems of each platform. These communication daemons differ in the types of protocols they employ. The execution process has considerable information about the properties of the communications that will be carried out by each communication daemon. For instance, it knows the number and location of task instances that will be exchanging data and the size of the objects being communicated. The synchronization specifications of the application stages also indicate whether peer-to-peer or predecessor-to-successor communications will be performed. The execution process can use all this information to select which of the available communication daemon implementations will give the best results for a particular task instance.

The VCE simplifies the process of enabling applications to exploit advances in communication technology. If a new high performance protocol or channel becomes available, VCE applications can be enabled to use it by implementing a new communication daemon and updating the execution processes with information on when use of the new daemon would be beneficial. Modifications to task implementations, task descriptions, and application descriptions would not be required.

The VCE design is not inordinately sensitive to the performance of communication protocols used to carry control information between group leaders, scheduling/execution daemons and execution processes. Each stage of a VCE application is a course grain long running computation. Delays on the order of a few seconds for the execution process to gather load information, select task hosts and disperse dispatching directives will not represent a critical performance issue as long as the dispatched tasks have running time on the order of minutes of hours.

5.2.6 Managing Location Information in the VCE

In all distributed systems, except those that are so static that machine addresses can be hard coded into system components, a mechanism must be provided for system

components to locate each other. In the VCE, in spite of the fact that participating machines may be spread across a wide area network, efficient mechanisms for managing location information have been established.

Section 5.2.2 discussed *Output Data Tables* (Figure 35) which are supplied to communication daemons and indicate how the locations of other communication daemons which are to receive their output. Communications daemons also communicate with a scheduling daemon and the task instance they are supporting. Both these components reside on the same machine as the communication daemon so dynamic location mechanisms are not needed.

Scheduling/dispatching daemons must divulge their load status to their group leaders. When a VCE scheduling/dispatching daemon is installed on a machine a file is placed on the local file system which includes a list of possible group leaders. The scheduling/dispatching daemon attempts to contact these machines starting at the top of the list. When one is found to be hosting the group leader this fact is noted for future reference. In many cases group members will have access to a common file system which simplifies administration of the group leader list. Methods for recovering from failures or withdrawal of scheduling/dispatching daemons or group leaders will be discussed in Section 5.2.7.

The most difficult location problem involves how execution processes can locate group leaders at run time. For each task in a VCE application, the execution process must select a group, or set of groups, to host it. To make this decision the execution process must have access to information about the location and architectural capabilities of each group participating in the VCE. This information is maintained in a series of World Wide Web servers geographically dispersed across the network. When an execution process begins, it contacts the nearest such server and down loads the information. When a task placement decision is to be made, the execution process consults this list and determines which groups would be most suitable for each task. Group leaders of suitable groups are contacted, starting with those that are geographically closest, to determine their capacity in light of their current transient load conditions. The execution process continues contacting group leaders until the

desired redundancy level is attained.

This method for locating group leaders requires that they register with the World Wide Web servers, and provide the required architectural information, when they are ready to begin fielding requests from execution processes.

5.2.7 Managing Dynamic Changes in the VCE

A Virtual Computing Environment is inherently dynamic. Machines may become unavailable because of hardware failures, network failures or decisions by local system administrators to temporarily remove resources from the VCE. Several strategies are employed by the VCE to cope with this dynamism.

While redundant scheduling is used in the VCE primarily as a load balancing methodology, it also imparts certain fault tolerant characteristics on VCE applications. When a machine hosting a VCE task instance fails, it is equivalent to the machine becoming infinitely slow. That is, instances of the same task running on other machines will complete and furnish results to concurrent tasks, subsequent tasks, or the user. Therefore, no special actions are needed on the part of an execution process to recover from a node failure.

Group leaders can minimize the impact of node failures with some rudimentary processing. The scheduling and dispatching algorithms discussed in Section 5.1 stated that when the load status of a machine changes, that is, when the load on a lightly machine increases or the load on a heavily loaded machine decreases, it notifies its group leader of the change. The group leader maintains a list of lightly loaded machines. If a heavily loaded machine fails, no action on the part of the group leader is needed. Since the machine was not on the group leader's list of available machines, no attempt will be made to dispatch VCE work on it. When a group leader is contacted by an execution process, the group leader provides a list of machines that VCE tasks can be dispatched on. Before the group leader places a machine on the list it contacts the scheduling/dispatching daemon on that machine to verify that it is still operating. If it is not, it is not included in the list returned to the execution process and it is

removed from the list of lightly loaded (available) machines.

Lightly loaded machines send test messages to the group leader at random intervals to verify that it has not failed or withdrawn from the VCE. If the group leader does not respond, the scheduling/dispatching daemon that detects it consults the group leader list and begins contacting these machines. The top most machine which has a functioning scheduling/dispatching daemon becomes group leader and contacts the World Wide Web servers to correct their group leader lists.

Bibliography

[1] Adler, R. M., *Emerging Standards for Component Software*, Computer, Vol. 28, No. 3, pp. 68-76, March 1995.

[2] Advanced Visual Systems Inc., **AVS Developer's Guide (Release 4.0)**, Part No. 320-0013-02, Revision B, Advanced Visual Systems Inc., Waltham Mass., May 1992.

[3] Ahmad, I., Ghagoor, A., and Mehrotra, K., *A Decentralized Task Scheduling Algorithm and its Performance Modeling for Computer Networks*, Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing, pp. 314-321, December 1991.

[4] Ahmad, I., Ghagoor, A., Mehrotra, K., Mohan, C. K., and Ranka, S., *Performance Modeling of Load Balancing Models Using Neural Networks*, Concurrency: Practice and Experience, Vol. 6, No. 5, pp. 393-409, August 1994.

[5] Allia, M., Arlat, J., Fabre, J.-C., Kanoun, K., and Powell, D., *Dependability Evaluation of the Saturation Technique*, Proceedings of the 5th Annual European Computer Conference (CompEuro '91), pp. 437-441, May 1991.

[6] Anderson, T. E., Culler, D. E., and Patterson, D. A., *A Case for NOW (Networks of Workstations)*, IEEE Micro, Vol 15, No. 1, pp. 54-64, February 1995.

[7] Armstrong, R., and Macfarlane, J., *The Use of Frameworks for Scientific Computation in a Parallel Distributed Environment*, Proceedings of the Third International Symposium on High Performance Distributed Computing, San Francisco, California, pp. 15-25, August 1994.

[8] Bach, M. J., **The Design of the Unix Operating System**, Prentice-Hall Inc., 1986.

[9] Barak, A., Guday, S.,and Wheeler, R. G., **The MOSIX Distributed Operating System, Load Balancing for UNIX**, Lecture Notes in Computer Science, Springer-Verlag, Vol. 672, 1993.

[10] Barak, A., Laden, O., and Yarom, Y., *The NOW MOSIX and its Preemptive Process Migration Scheme*, Bulletin of the Technical Committee on Operating Systems and Application Environments, IEEE Computer Society, Vol. 7, No. 2, pp. 5-11, June 1995.

[11] Barak, A., and Wheeler, R. G., *MOSIX: An Integrated Multiprocessor UNIX*, Proceedings of the Winter 1989 USENIX Conference, San Diego, CA, pp. 101-112, February 1989.

[12] Barbacci, M. R., Doubleday, D. L., Weinstock, C. B., and Wing, J. M., *The Durra Language and Runtime Environment: Tools for PMS Level Programming*, **Decentralized Systems**, Proceedings of the IFIP WG 10.3 Working Conference, pp. 25-27, North Holland, 1990.

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

[14] Barbacci, M. R., and Wing, J. M., *Specifying Functional and Timing Behavior for Real-Time Applications*, Proceedings of Parallel Architectures and Languages Europe (PARLE) Volume 2. Springer-Verlag, Lecture Notes in Computer Science, Vol 259, pp. 124-140, June 1987.

[15] Benmohammed-Mahieddine, K., Dew, P. M., and Kara, M., *A Periodic Symmetrically-Initiated Load Balancing Algorithm for Distributed Systems*, Proceedings of the 14th International Conference on Distributed Computing Systems, pp. 484-492, June 1994.

[16] Besana, V., De Luca, J., Giard, E., and Reger, J., **Experiences with SOMobjects: Distributed System Object Model (DSOM)**, IBM International Technical Support Centers, Document No. GG24-4165-00, November 1993.

[17] Betz, M. *Interoperable Objects*, Dr. Dobb's Journal, Vol. 19, No. 11, pp. 18-32, October 1994.

[18] Birman, K. P., Schiper, A., and Stephenson, *Lightweight Causal and Atomic Group Multicast*, ACM Transactions on Computer Systems, Vol 9, No 3. pp. 272-299, August 1991.

[19] Birman, K., Cooper, R., Joseph, T., Kane, K. and Schmuck, F., **The Isis System Manual.**

[20] Blumofe, R. D., and Park, D. S., *Scheduling Large-Scale Parallel Computations on Networks of Workstations*, Proceedings of the Third International Symposium on High Performance Distributed Computing, San Francisco CA., pp. 96-105, August 1994.

[21] Bricker, A., Litzkow, M. and Livny, M., *Condor Technical Summary*, Computer Science Department, University of Wisconsin - Madison, January 1992.

[22] Butler, R., and Lusk, E., *User's Guide to the p4 Programming System*, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439-4801, April 1994.

[23] Carriero, N., and Gelernter, D., *Linda in Context*, Communications of the ACM, Vol. 32, No. 4, pp. 444-459, April 1989.

[24] Carriero, N., Gelernter, D., Kaminsky, D., Westbrook, J., *Adaptive Parallelism with Piranha*, Department of Computer Science, Yale University, New Haven, Connecticut, YALE/DCS/RR-954, February 1993.

[25] Carriero, N., Gelernter, D., Mattson, T. G., and Sherman, A. H., *The Linda Alternative to Message-Passing Systems*, Parallel Computing, Vol. 20, No. 4, April 1994.

[26] Casavant, T. L., and Kuhl, J. G., *A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems*, IEEE Transactions on Software Engineering, Vol. 14, No. 11, pp. 141-154, November 1988.

[27] Chang, J., and Maxemchuck, N. F., *Reliable Broadcast Protocols*, ACM Transactions on Computer Systems, Vol. 2, No. 3, pp. 251-273, August 1984.

[28] Chen, J., Xie, L., and Sun, Z.-X., *An Intelligent Network Manager for Distributed Task Scheduling*, Proceedings of the 11th International Conference on Computer Communication, pp. 493-499, October 1992.

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

[30] Cheriton, D.R., *The V Distributed System,* Communications of the ACM, pp. 314-333, Vol. 31, No. 3, 1988.

[31] Chiu, J-F, and Chiu, G-M, *Process-Replication for Fault-Tolerance and Performance Improvement in Distributed Computing Systems*, Proceedings of the Third International Symposium on High Performance Distributed Computing, San Francisco, California, pp. 236-243, August 1994.

[32] Chu, W. W., Holloway, L. J., Lan, M., and Efe, K., *Task Allocation in Distributed Data Processing*, Computer, Vol 13, No. 11, pp. 57-69, November 1980.

[33] 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, 1992.

[34] Dandamudi, S., *Performance Impact of Scheduling Discipline on Adaptive Load Sharing in Homogeneous Distributed Systems*, Proceedings of the 15th International Conference on Distributed Computing Systems, pp. 484-492, June 1995.

[35] Deremer, F., and Kron, H. H., *Programming-in-the-Large Versus Programming-in-the-Small*, IEEE Transactions on Software Engineering, Vol. 2, No. 2, June 1974.

[36] Eager, D. L., Lazowska, E. D., and Zahorjan, J., *A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing*, Performance Evaluation, Vol. 6, pp. 53-68, March 1986.

[37] Eager, D. L., Lazowska, E. D., and Zahorjan, J., *Adaptive Load Sharing in Homogeneous Distributed Systems*, IEEE Transactions on Software Engineering, Vol. 12, No. 5, pp. 662-675, May 1986.

[38] Elnozahy, E. N., and Zwaenepoel, W., *Replicated Distributed Processes in Manetho*, Proceedings of the Conference on Fault-Tolerant Computing Systems, pp 222-230, 1991.

[39] Efe, K., and Schaar, M. A., *Performance of Co-Scheduling on a Network of Workstations*, Proceedings of the 13th International Conference on Distributed Computing Systems, pp. 525-531, May 1993.

[40] Fabre, J. C., *Improving Fault Tolerance in Distributed Systems: The Saturation Approach*, Proceedings of the 5th Annual European Computer Conference (CompEuro '91), pp. 634-641, May 1991.

[41] Farrell, W. E., Fox, W. Given, J. W., MacRitchie, B., and Wang, J., *Data Processing on Networks of Workstations*, Bulletin of the Technical Committee

on Operating Systems and Application Environments, IEEE Computer Society, Vol. 7, No. 2, pp. 35-42, June 1995.

[42] Fox, G. C., et. al., **Solving Problems on Concurrent Processors, Volume 1,** Prentice Hall, 1988.

[43] Fox, G. C., **The Architecture of Problems and Portable Parallel Software Systems,** Northeast Parallel Architectures Center, Technical Report SCCS-134, Syracuse University, July 1991.

[44] Fox, G. C., and Furmanski, W., *The Use of National Information Infrastructure and High Performance Computers in Industry*, Proceedings of the 2nd International Workshop on Massively Parallel Processing Using Optical Interconnections, October 1995.

[45] Fox, G. C., Furmanski, W., Chen, M., Rebbi, C., and Cowie, J. H., *WebWorks: Integrated Programming Environment Tools for National and Grand Challenges*, Northeast Parallel Architectures Center, Technical Report sccs-0715, June 1995.

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

[47] Garcia-Molina, H., and Spauster, A., *Ordered and Reliable Multicast Communication*, ACM Transactions on Computer Systems, Vol. 9, No. 3, pp. 242-271, August 1991.

[48] Geist, A., Beguelin, A., Dongarra, J., Weicheng, J., Manchek, R., and Sunderam, V., **PVM 3 User's Guide and Reference Manual**, Oak Ridge National Laboratory, Document No. ORNL/TM-12187, September 1994.

[49] Gelernter, D., and Carriero, C., *Coordination Languages and their Significance*, Communications of the ACM, Vol. 35, No. 2, pp. 97-107, February 1992.

[50] Ghafoor, A., and Ahmad, I., *An Efficient Model of Dynamic Task Scheduling for Distributed Systems*, Proceedings of the 14th Annual International Computer Software and Applications Conference, pp. 442-447, November 1990.

[51] Graham, I. S., **The HTML Sourcebook**, John Wiley & Sons Inc., 1995.

[52] Grimshaw, A. S., *The Mentat Runtime System: Support for Medium-Grain Parallel Computation*, Proceedings of the Fifth Distributed Memory Computing Conference, Los Alamitos, CA., Vol II, pp. 1064-1073, 1990.

[53] Grimshaw, A. S., *Easy-to-Use Object-Oriented Parallel Processing with Mentat*, Computer, Vol. 26, No. 5, May 1993.

[54] Hariri, S., *High Performance Distributed Systems: Concepts and Designs*, Artech House, in preparation.

[55] 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.

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

[57] Heidelberger, P., and Lavenberg, S. S., *Computer Performance Evaluation Methodology*, IEEE Transactions on Computers, Vol. 33, No. 12, pp. 1195-1220, December 1984.

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

[59] Jacob, B., **SOMobjects: A Pratical Introduction to SOM and DSOM**, IBM International Technical Support Centers, Document No. GG24-4357-00, July 1994.

[60] Ju, J., Xu, G., and Tao, J., *Parallel Computing Using Idle Workstations,* Operating Systems Review, Vol. 27, No. 3, 1993.

[61] Kaashock, M. F., and Tanenbaum, A. S., *Group Communication in the Amoeba Distributed Operating System*, Proceedings of the 11th International Conference on Distributed Computing Systems, pp. 222-230, May 1991.

[62] Kernighan, B. W., and Ritchie, M. R., **The C Programming Language - Second Edition**, Prentice Hall, 1988.

[63] Kramer, J., *Configuration Programming - A Framework for the Development of Distributable Systems,* The Proceedings of the IEEE International Conference on Computer Systems and Software Engineering (CompEuro 90), pp. 374-384, 1990.

[64] Kramer, J., Magee, J., and Finkelstein, A., *A Constructive Approach to the Design of Distributed Systems*, Proceedings of the 10th International Conference on Distributed Computing Systems, pp. 580-587, June 1990.

[65] Kramer, J., Magee, J., and Sloman, M., *Configuration Support for System Description, Construction and Evolution*, **Managing Complexity in Software Engineering**, pp. 247-260, edited by R. J. Mitchell, Peter Pereginus Ltd., 1990.

[66] Kremien, O., and Kramer, J., *Methodical Analysis of Adaptive Load Sharing Algorithms*, IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 6, pp. 747-760, November 1992.

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

[68] Krueger, P., and Livny, M., *A Comparison of Preemptive and Non-Preemptive Load Distributing*, Proceedings of the 8th International Conference on Distributed Computing Systems, pp. 123-130, June 1988.

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

[70] Leffler, S. J., McKusick, M. K., Karels, M. J., and Quaterman, J. S., **The Design and Implementation of the 4.3BSD Unix Operating System**, Addison-Wesley, 1989.

[71] Litzkow, M. and Solomon, M., *Supporting Checkpointing and Process Migration Outside the Unix Kernel,* Proceedings of the Winter 1992 USENIX Conference, pp. 283-290, 1991.

[72] Liu, R.-C., *Analysis on the Effects of System Parameters on Load Balancing*, Information Sciences, Vol. 81, No. 1-2, pp. 37-54, November 1994.

[73] Lord, H. D., *Improving the Application Development Process with Modular Visualization Environments*, Computer Graphics, Vol. 29, No. 2, May 1995.

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

[75] MacDougall, M. H., **Simulating Computer Systems - Techniques and Tools**, MIT Press, 1987.

[76] Mattson, T. G., *Programming Environments for Parallel Computing: A Comparison of CPS, Linda, P4, PVM, POSYBL, and TCGMSG*, Proceedings of the 27th Annual Hawaii International Conference on Systems Science, Vol. II, pp. 586-594, January 1994.

[77] Mirchandaney, R., Towsley, D., and Stankovic, J. A., *Analysis of the Effects of Delays on Load Sharing*, IEEE Transactions on Computers, Vol. 38, No. 11, pp. 1513-1525, November 1989.

[78] Mullender, S.J., and van Rossum, G., *Amoeba: A Distributed Operating System for the 1990s,* Computer, pp. 44-53, Vol. 23, No. 5, 1990.

[79] Munetomo, M., Takai, Y., and Sato, Y., *A Genetic Approach to Dynamic Load Balancing in a Distributed Computing System*, Proceedings of the 1st IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence, Vol. 1, pp. 418-421, June 1994.

[80] Mutka, M. W., *Estimating Capacity for Sharing in a Privately Owned Workstation Environment,* IEEE Transactions on Software Engineering, Vol. 18, No. 4, 1992.

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

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

[83] Parasoft Corporation, *Express Reference Manual,* 1988.

[84] Park, C., and Kuhl, J. G., *A Fuzzy-Based Distributed Load Balancing Algorithm for Large Distributed Systems*, Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems, pp. 366-273, April 1995.

[85] Peterson, L., Hutchinson, N., O'Malley, S., and Rao, H., *The x-Kernel: A Platform for Accessing Internet Resources,* Computer, pp. 23-33, Vol. 23, No. 5, May 1990.

[86] Prieto-Diaz, R., and Neighbors, J.M., *Module Interconnection Languages,* The Journal of Systems and Software, No. 6, pp. 307-334, 1986.

[87] Rehmet, G. M., **Remora - Implementing Adaptive Parallelism on a Heterogeneous Cluster of Networked Workstations**, Master's Thesis, Rhodes University, South Africa, September 1994.

[88] Rinaldo, F., and Wolbers, S., *Loosely Coupled Parallel Processing at Fermilab*, Computers in Physics, Vol. 7. No. 2, pp. 184-190, March 1993.

[89] Rosenberry, W., Kenney, D., and Fisher, G., **Understanding DCE**, O'Reily & Associates, 1992.

[90] Rousselle, P., Hariri, S., and Tymann, P, *Redundant Task Scheduling in a Heterogeneous Computing Environment*, Northeast Parallel Architectures Center, Technical Report SCCS-658, October 1994.

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

[92] Rousselle, P., Yadav, R., Hariri, S., Kim, D., Boddapati, S., and Tymann, P., *Application Development and Execution in a Virtual Computing Environment*, Northeast Parallel Architectures Center, Technical Report SCCS-705, April 1995.

[93] Schaar, M., Efe, K., and Delcambre, L. M., *An Analytical Model for Predicting Performance in a Heterogeneous System*, Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Computing Systems, pp. 328-335, May 1991.

[94] Shoja, G.C., *A Distributed Facility for Load Sharing and Parallel Processing Among Workstations,* Journal of Systems Software, pp. 163-172, Vol. 14, No. 3, 1991.

[95] Simpson, R. A., and Harget, A. J., *A Simulation Study to Determine the Importance of Load Balancing Algorithms for Loosely Coupled Distributed Systems*, IFIP Transactions on Computer Science and Technology, IFIP 13th World Computer Congress, Vol. A-51, pp. 69-74, August 1994.

[96] Stevens, W. R., **UNIX Network Programming**, Prentice Hall, 1990.

[97] Stroustrup, B., **The C++ Programming Language - Second Edition**, Addison-Wesley, 1991.

[98] Stroustrup, B., **The Design and Evolution of C++**, Addison-Wesley, 1994.

[99] V. S. Sunderam, **PVM: A Framework for Parallel Distributed Computing**, Oak Ridge National Laboratory, 1991.

[100] Tannenbaum, A. S., **Modern Operating Systems**, Prentice-Hall, 1992.

[101] Theimer, M., and Lantz, K., *Finding Idle Machines in a Workstation-Based Distributed System,* IEEE Transactions on Software Engineering, pp. 1444-1458, Vol. 15, No. 11, 1989.

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

[103] Thornborrow, C., Wilson, A. J. S., and Faigle, C., *Developing Modular Application Builders to Exploit MIMD Parallel Resources*, Proceedings of Visualization '93, pp. 134-141, San Jose CA, October 1993.

[104] 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.

[105] Walker, B., Popek, G., English, R., Kline, C., and Thiel, G., *The LOCUS Distributed Operating System,* Proceedings of the 1983 SIGOPS Conference, pp. 49-70.

[106] Wang, C. -J., Krueger, P. E., and Liu, M. T., *Intelligent Job Selection for Distributed Scheduling,* Proceedings of the Thirteenth International Conference on Distributed Computing Systems, pp. 517-524, May 1993.

[107] Wood, M. D., *Replicated RPC Using Amoeba Closed Group Communication*, Proceedings of the 13th International Conference on Distributed Computing Systems, pp. 222-230, 1993.

[108] Zhou, S., *A Trace-Driven Study of Dynamic Load Balancing*, IEEE Transactions on Software Engineering, Vol. 14, No. 9, pp. 1327-1341, September 1988.

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

[110] Zhu, W., and Goscinski, A., *Low Cost Load Balancing Algorithms*, Proceedings of the 27th Annual Simulation Symposium, pp. 226-235, April 1994.

Biographical Data

Name:	Philip Rousselle
Date and Place of Birth:	September 10, 1962 Buffalo, New York
Elementary & High School:	Buffalo Public Elementary School #84 Graduated 1976
	Kensington High School Buffalo, New York Graduated 1980
College:	SUNY College at Buffalo BS (Info Systems Management), 1983
	Union College Schnectady, New York MS (Computer Science), 1989
Employment	International Business Machines Inc Endicott NY