Chapter 3

PARALLEL RENDERING SYSTEMS AND LOAD BALANCING SCHEMES

INTRODUCTION

Since many types of parallel rendering have been investigated [1, 2], classifying the various schemes is important to characterize the behavior of each. A parallel rendering system can be classified according to the method of task subdivision and/or by the hardware used to implement the scheme. Often, the choice of one influences the other.

Classifying by task subdivision refers to the method in which the original rendering task is broken into smaller pieces to be processed in parallel. Obviously, such subdivision strongly depends on the type of rendering employed. A task for rendering polygons will offer a different set of subdivision opportunities than a ray-tracing task. Also included in these decisions is the type of load balancing technique to employ.

Ultimately the rendering scheme is implemented within some sort of parallel environment. The system may run on parallel hardware (e.g., a general multiprocessor or specialized hardware) or in a distributed computing environment (a group of individual machines working together to solve a single problem). The advantages and disadvantages associated with each environment are discussed below.

In this chapter we studied about various parallel rendering systems and different load balancing schemes, which can be used for distributed image computing systems.

3.1. CLASSIFICATION OF PARALLEL RENDERING SYSTEMS BY TASK SUBDIVISION

In this section, we will look at two different types of rendering (polygon-based rendering and ray tracing) and various methods for subdividing the original task into subtasks for parallel processing. Although many subdivision techniques exist for each, we will focus on the schemes most widely used. For each technique, our goal is to subdivide the original task in such a way as to maximize parallelism, while not creating excessive overhead.
For polygon rendering, we often deal with a very large number of primitives (e.g., triangles), which can often be processed in a parallel manner. To handle this type of rendering, a graphics pipeline is usually employed (Fig. 3.1). Stages in this pipeline include geometry processing and rasterization.

Geometry processing comprises transformation, clipping, lighting, and other tasks associated with a primitive. A straightforward method for parallelizing geometry processing is to assign each processor a subset of primitives (or objects) in the scene to render. In rasterization, scan-conversion, shading, and visibility determination are performed. To parallelize this processing, each processor could perform the pixel calculations for a small part of the final image.

One way to view the processing of primitives is as a problem of sorting primitives to the screen since a graphics primitive can fall anywhere on or off the screen MoI [3]. For a parallel system, we need to distribute data across processors to keep the load balanced. Actually, this sort can occur anywhere in the rendering pipeline:

- during geometry processing (sort-first)
- between geometry processing and rasterization (sort-middle)
- during rasterization (sort-last)

The structure of the parallel rendering system is determined by the location of this sort. The following discussion follows that in MoI [3].
3.1.1.1. Sort-First

The main idea behind sort-first is to distribute primitives early (during geometry processing) in the rendering pipeline (Fig. 3.2). The screen is divided into regions of equal size (Fig. 3.3), and each processor (or renderer) is assigned a region. Each processor is responsible for all the pixel calculations that are associated with its screen region.

In an actual implementation, primitives would initially be assigned to processors in an arbitrary way. Each renderer then performs enough transformation processing to determine the screen region into which the primitive falls. If this region belongs to another processor, the primitive is sent over the interconnection network to that processor for rendering. After each primitive has been placed with the proper renderer, all of the processors can work in parallel to complete the final image.

With this method, each processor implements the entire rendering pipeline for its portion of the screen. Communication costs can be kept comparatively low compared with other methods if features such as frame coherence are properly exploited. For rendering a single frame, however, almost all the primitives will have to be redistributed after the initial random assignment. Some duplication of effort may occur if a primitive falls into more than one region, or if the results of the original geometry processing are not sent with the primitive when it is transmitted to the appropriate renderer. Also, the system is susceptible to load imbalance since primitives may be concentrated into particular regions or may simply take longer to
render. Both of these situations will cause the affected processor to consume more time in processing its screen region. Very few, if any, sort-first renderers have been built.

Fig. 3.3: Image space subdivisions

3.1.1.2. Sort-Middle

In a sort-middle renderer, primitives are sorted and redistributed in the middle of the pipeline: between geometry processing and rasterization (Fig. 3.4). By this point, the screen coordinates of the primitives have been determined through transformation processing, but the primitives have not yet been rasterized. This point is a natural breaking position in the rendering pipeline.

Fig. 3.4: Sort-middle polygon rendering scheme

In an actual implementation, primitives are arbitrarily assigned to processors as before. The geometry processors perform transformation, lighting, and other processing on the primitives originally assigned to them, and then classify the primitives according to screen region. Screen-space primitives are then sent to the appropriate rasterizing processors, which have been assigned unique regions of the screen, for the remaining processing.
The sort-middle strategy is general and straightforward and has been implemented in both hardware (including Pixel-Planes 5 Fuch [4] and the SGI Reality Engine Acke [5]) and software Whit [6] Ells [7]. Like sort-first, however, this method is susceptible to load imbalance due to the uneven distribution of primitives across screen space. Communication times can be higher under certain conditions. Also, primitives that overlap regions may require some additional processing.

3.1.1.3. Sort-Last

Under the sort-last strategy, sorting is deferred until the end of the rendering pipeline (Fig. 3.5). Primitives are rasterized into pixels, samples, or pixel fragments, which are then transmitted to the appropriate processor for visibility determination.

![Fig. 3.5: Sort-last polygon rendering scheme](image)

In practice, primitives are initially distributed to processors in an arbitrary manner, as in the other methods. Each renderer then performs the operations necessary to compute the pixel values for its primitives, regardless of where those pixels may reside on the screen. These values are then sent to the appropriate processors according to screen location. At this point, the rasterizing processors perform visibility calculations and composite the pixels for final display.

As in sort-first, each processor implements the entire graphics pipeline for its primitives. While the overall technique is less prone to load imbalance, the pixel traffic in the final sort can be very high. Numerous rendering systems using the sort-last method have been constructed in various forms, including Evan [8] and Kub [9].

3.1.2. Ray Tracing

Ray tracing is a powerful rendering technique that can produce high-quality graphics images; however, this quality comes at a price of intensive calculation and long rendering times. Even relatively simple ray-traced animations can prohibitively expensive to render on a single processor. For longer, more complex animations, the
rendering time can be intractable. Fortunately, ray tracing is a prime candidate for parallelization since its processing is readily amenable to subdivision. Specifically, ray tracing inherently contains a large amount of parallelism due to the independent nature of its pixel calculations Whit [10]; therefore, most ray tracing rendering algorithms lend themselves to parallelization in screen space.

Other partitioning schemes are employed in ray tracing as well. Instead of dividing the image space, the object space can be split into smaller regions, or the objects themselves may be assigned to individual processors. These techniques are discussed more fully below.

3.1.2.1. Image Space Partitioning

Using this scheme, the viewing plane is divided into regions, each of which is completely rendered by an individual processor (Fig. 3.3). That is, for each pixel in a region, the processor assigned to that region computes its entire ray tree. While this technique is conceptually straightforward, the entire database of scene objects must be accessible to every processor.

The benefits of this approach are simplicity and low inter-processor communication as compared with other partitioning methods; the largest drawback is its limitation to multiprocessor architectures with significant local processor memory. Another potential problem is load imbalance, since image detail may be concentrated in certain regions of the screen. To combat this situation, the load-balancing algorithm may further subdivide complex regions to provide idle processors with additional tasks.

3.1.2.2. Object Space Partitioning

Here the 3-dimensional space where the scene objects reside is divided into sub-volumes, or voxels. Voxels may not be equally sized in order to achieve better load balancing. In the initialization phase of ray tracing, each voxel is parceled out to a particular processor. When rays are cast during rendering, they are passed from processor to processor as they travel through the object space. Each processor, therefore, needs only the scene information associated with its assigned voxels.

While this technique may not suffer from frequent load imbalance, it does incur costs in other ways. First, as new rays are shot, they must tracked through voxel space; this processing is not required for other schemes. Additionally, since potentially millions of rays are fired for each image, communication could be become excessive as rays enter and exit regions of object space during rendering.

3.1.2.3. Object Partitioning

This partitioning scheme parallelizes the rendering task by assigning each object to an individual processor. As in object space partitioning, rays are passed as
messages between processors, which in turn test the ray for intersection with the objects they are assigned. Object partitioning also shares some of the benefits and detriments of object space partitioning. Specifically, the load may be fairly well balanced, but the communication costs may be high due to the large amount of ray message traffic.

3.1.2.4. Load Balancing Scheme

Finally, parallel-rendering schemes can be classified according to their load balancing method. Of course, the primary goal of any load-balancing scheme is to distribute the work among processors as evenly as possible and thus exploit the highest degree of parallelism available in the application. Many different types of load balancing schemes exist, but each falls into one of two categories:

**Static Load Balancing**: In this scheme, partitioning is performed up front and processors are assigned subtasks for the entire duration of the rendering process. In this way, overhead is minimized later in the rendering; however, a good deal of care must be taken to ensure that the load will be balanced. Otherwise, the algorithm will suffer from poor performance.

**Dynamic Load Balancing**: With this scheme, some processing assignments are determined at the start, but later assignments are demand-driven. That is, when a processor determines that it needs more work to do, it will request a new assignment. In this way, processors will never be idle as long as more work is left to do. The key here is to distribute the load as evenly as possible without incurring excessive overhead.

We may further classify the Load Balancing Methods in following three categories:

**Sender or receiver initiated**: An algorithm is said to be sender-initiated if a local host makes a determination as to where a generated task or arriving task is to be executed. The queues of ready jobs tend to form at the target PE's. Job transfer decisions are made at task arrival time. In a receiver initiated process, a server or target PE determines which jobs at different sources, it will process. The ready jobs tend to queue at the source PE's. With job transfer decisions made at task completion time.

**Static or Adaptive**: In the former, the transfer and placement policies are based on the information of the average behavior of the system. These algorithms do not depend on the current state of the system at the time of decision-making. Random and round robin placement policies are examples of static decision making processes. Adaptive schemes react to changes in the system state. The complexity of these schemes is an obvious drawback, as they require runtime load information and state collection activities. Threshold scheduling are examples of this.
Central or Distributed information and placement decisions: There are three categories:

Centralized policies: A PE designed as a central scheduler, receives updated load information from all other hosts and assembles them into a load vector. For the placement policy, when a host decides to transfer a job, it sends a request to the central scheduler who selects the target host using the load vector and informs the source host of its choice. These policies reduce system overhead due to the load distribution information.

Distributed policies: Here each host autonomously constructs its own local load vector by collecting the load info from the other PE's. The placement decisions are made locally. These policies speed up the process but generate large communication overhead due to load info distribution.

Mixed policies: A combination of central/distribution policies can be used. For example, we have a central load information policy along with distributed placement policy.

The major drawback for dynamic load balancing is the runtime overhead for load info distribution, making placement decisions and transferring a job to a target host. The goal is to achieve less overhead, by using effective load index measures and efficient load distribution methods. Hierarchical system organizations with local load info distribution and local load balancing policies seem to be ideal for more efficiency.

3.1.2.5. Hybrid Scheme

Hybrid schemes have also been proposed, combining image space partitioning with object space partitioning Bado [11] and image space partitioning with object partitioning Kim [12]. When choosing a partitioning scheme, the architecture of the parallel machine should be considered. For instance, an image space-partitioning algorithm will perform better on a MIMD machine than on a SIMD machine. In general, tradeoffs exist between the type of partitioning algorithm used and the architecture chosen.

3.2. CLASSIFICATION PARALLEL RENDERING SYSTEMS BY HARDWARE

As previously stated, for some computationally intensive rendering tasks, parallel processing provides the only practical means to a solution. One way to perform parallel rendering is to use a single multiprocessor machine, such as a Thinking Machines CM-5, Intel Paragon, Cray T3E, or specialized parallel processor. In these machines, enormous computing power is provided by up to tens of thousands of processing elements able to access many gigabytes of memory and to work in concert through a high-speed interconnection network. Multiprocessors are the most
powerful computers in the world and play an active role in solving Grand Challenge problems, such as weather prediction, fluid dynamics, and drug design Hwan [13].

An alternative to using traditional multiprocessor systems for parallel processing is to employ a network of workstations acting as a single machine. This approach, termed distributed or cluster computing, is conceptually similar to a multiprocessor, but each processing element consists of an independent machine connected to a network usually much slower than a multiprocessor interconnection network. While this network can be of any type (e.g., Ethernet, ATM) or topology, the computers connected to it are generally UNIX-based machines which support some type of distributed programming environment, such as Parallel Virtual Machine (PVM) Geis [14] or Message Passing Interface (MPI) Grop [15]. Many types of applications can benefit from distributed computing, including computation-intensive graphics tasks, such as ray tracing (Image Computing) Sung [16].

This section focuses on past work that has been documented using traditional multiprocessors and clusters of machines to accomplish graphics rendering tasks, particularly in the area of ray (Image) tracing. Included in this discussion is relevant background concerning PVM and MPI, as well as motivation for using these systems in a clustered environment.

3.2.1. Parallel Hardware

Since rendering consumes such a large amount of computing resources and time, a good deal of effort has gone into exploring parallel solutions on multiprocessor machines. Some of the schemes proposed are designed to run on general-purpose parallel machines, such as the CM-5, while others rely on specialized hardware built especially for ray tracing rendering. A brief survey of these techniques appears in the following sections. Although current research continues in the design and implementation of parallel rendering systems, a flurry of activity in this area occurred in the late 1980s and early 1990s, as reflected in many of the references.

3.2.1.1. General-Purpose Multiprocessors

In Plun [17], a vectorized ray tracer is proposed for the CDC Cyber 205. In a given execution cycle, rays awaiting processing are distributed to individual processors and ray-object calculations are performed object by object in a lock-step SIMD fashion.

Similarly, Crow [18] implements a SIMD ray tracing algorithm, but for the Connection Machine (CM-2). Image subdivision is used with one pixel being assigned to each of the 16K processors to produce a 128x128 image, with ray-object intersections performed on an object-by-object basis. The algorithm proposed by Schr [19] also runs on a CM-2, but uses an object-space subdivision coupled with processor remapping capabilities to achieve dynamic load balancing.
Rounding out the SIMD field, Goe [20] describes a ray casting method developed on the MasPar MP-1 for volume rendering, another computation-intensive graphics application used for viewing complex structures in medical imaging and other forms of scientific visualization. To handle the large amount of data and processing involved, machines are assigned portions of the volume to render, which are composited to produce a final image. This system allows users to rotate a volume, magnify areas of interest, and perform other viewing operations.

In the MIMD category, Reis [21] employs an IBM SP/2 running an image-space partitioning scheme with dynamically adjustable boundaries to render frames of an animation progressively. In this form of rendering, termed progressive rendering, an image is initially rendered quickly at low resolution and progressively refined when little or no user interaction takes place. Progressive rendering is useful in interactive environments where frame generation rate is important. The goal of Keat [22] also involves progressive rendering, although their renderer makes use of object-space partitioning on the Kendall Square Research KSR-1 machine.

Several research efforts have focused on the Intel iPSC machines as the architectural environment for implementing a parallel ray tracer. Interestingly, whether the partitioning scheme is image-based Isie [23] Silv [24], object space-based Prio [25] Prio [26] or a hybrid of the two Akti [27] Bado [11], the load-balancing scheme is almost always of a static nature Isie [23] also tests a dynamic scheme). This choice results from a concern that dynamic load balancing schemes produce a large number of messages, which in turn, may dramatically affect the performance of a distributed machine Prio [26].

In the area of transputer-based machines, Green [28] uses an image subdivision technique combined with memory and cache local to each processor to deal with the many required accesses to the scene description database. Here, the granularity of parallelism is controlled through the size of the image subregion, which also relates directly to the effectiveness of the dynamic load-balancing scheme. To render ray-traced animations, Maur [29] use a static object-space partitioning scheme on a system of 36 transputers. Progressive ray tracing and volume rendering on transputers is addressed by Sous [30] and Pito [31], respectively.

3.2.1.2. Specialized Multiprocessors

Probably one of the most noteworthy examples of a specialized multiprocessor for polygon rendering is the series of Pixel-Planes machines developed at UNC-Chapel Hill. The Pixel-Planes 4 machine Eyle [32] is a SIMD machine with three basic components: a host workstation, a graphics processor, and a frame buffer. Each of the customized processors is responsible for a column of display pixels. For its time, it provided good performance; however, the system used processors with slow clock speeds and did not provide effective load balancing.
Pixel-Planes 5 Fuch [4] provided some improvements over Pixel-Planes 4 by incorporating faster processors and employing a more flexible MIMD architecture. The system implemented a sort-middle algorithm, with each processor in charge of a particular region of the screen. To handle the communication, ring architecture capable of handling eight messages simultaneously is employed. Ultimately, the ring network imposes a limit on scalability.

The PixelFlow machine Moln [33] was developed to overcome the limitations of the previous architectures through parallel image composition. Each individual processor works to create a full-screen image using only the primitives assigned to it. All of these images are collected and composited to form the final display.

For ray tracing, Lin [34] employs a specialized SIMD machine to perform stochastic ray tracing. The stochastic method adds extra processing to the ray-tracing algorithm to handle anti-aliasing, an important aspect of any renderer. To overcome some of the inefficiencies found in other SIMD approaches, a combination of image space partitioning and object space partitioning is used. That is, a block of pixels is rendered by casting rays and using scene coherence to restrict the parts of object space, which must be tested.

In Gaud [35] a special-purpose MIMD architecture using image space subdivision and a static load distribution is described. To overcome the problem of having the entire object database resident at each processor, a central broadcast processor issues data packets describing the object database cyclically. Here, the processors make requests for various pieces of the database, and only those parts are broadcast in a given cycle. Using a somewhat different approach, Shen [36] uses object space partitioning on clusters of processors, but each processor operates in a pipelined fashion, a scheme previously explored in the LINKS-I architecture Nish [37].

One of the few multiprocessor architectures which allocates work based on object subdivision combined with image subdivision is proposed by Kim [12]. Each processor handles ray-object intersection tests with its assigned objects, which are spread across the object space. If the load becomes unbalanced, objects are dynamically transferred to other processors.

Other specialized multiprocessor machines of note are the Pixel Machine Potm [38] (useful for several types of rendering including ray tracing and the RayCasting Engine Men [39] (specifically built for CSG modeling).

### 3.2.1.3. Distributed Computing Environments

Parallel rendering using distributed computing environments continues to grow in popularity, especially in the fields of distributed image computing, entertainment and scientific visualization. Below are a few interesting examples.
Perhaps the most popular example is the Disney film, *Toy Story*, which used a network of 117 Sun workstations and the Pixar Renderman system to produce the animation Henn [40]. To generate its 144,000 individual frames, *Toy Story* required about 43 years of CPU time. If not for the many machines participating in the computation, the movie's production could not be realized. For some tasks, such as applying surface textures, one machine was chosen as a server for the rest. For other tasks, such as final rendering, the machines were basically used independently to render individual frames.

Another entertainment application used a network of 40 Amiga machines to render special effects for the television series *Sea Quest* Wor [41]. Although the delivered product contained only two to three minutes of computer graphics per episode (3,600 to 5,400 frames of animation), the rendering activity was so time-intensive that the team struggled to deliver the graphics within its weekly deadline.

For the average user, some popular commercial animation packages (e.g., Alias/Wavefront, Maya, and 3D Studio) employ coarse-grain parallelism to allow rendering of individual frames of an animation across a network of machines. This technique can mean the difference between an animation being ready in hours or in days. POV -Ray has also been ported to run in a clustered environment; however, the parallelization scheme works on single images only.

Other computation-intensive graphics problems have taken advantage of the processing power of distributed computing, specifically volume rendering and virtual reality. Although real-time interaction in these systems is constrained by the relatively slow network connecting the machines, significant speedups have been reported using a network of IBM RS/6000 machines for volume rendering Gier [42] Ma [43], and a network of Sun SPARCstation's and HP workstations for virtual environments Pan [44].

Distributed computing is also being applied to computer vision algorithms, a field closely related to graphics. Here, researchers use PVM on a cluster of 25 Sun SPARCstation's for an edge-detection algorithm. One remarkable result of their experiments is that they achieved super-linear speedup on the cluster over the sequential version. This result is due to the large aggregate memory of the clustered machines, which reduced the amount of paging as compared to the single processor.

Not nearly as much research has been conducted concerning ray tracing in distributed computing environments as in traditional multiprocessor machines. Perhaps this fact is due to the relatively recent introduction of PVM and MPI. Regardless of the reason, more advanced parallel ray tracing algorithms combined with a distributed computing environment remains a largely unexplored area. Several related projects are summarized below.

For single images, Jeva [45] uses a dynamic load balancing technique with spatial partitioning and a novel warp synchronization method. At the other end of the
spectrum, Ris [46] applies a static load-balancing scheme using object partitioning on a network composed of both sequential workstations and parallel computers. Surprisingly few systems use an image partitioning scheme in a distributed computing environment, even though it represents the technique with the highest potential for speedup Clea [47] and overcomes the problems of limited local memory that exist in traditional multiprocessors.

For animations, DeMa [48] describes the DESIRE (Distributed Environment System for Integrated Rendering) system, which incorporates a coarse-level dynamic load-balancing scheme that distributes individual frames of an animation to networked workstations. The goal of Stob [49] is similar, except that the system is designed to run without affecting the regular users of the workstations. By stealing idle cycles from 22-34 workstations, a ray-traced animation lasting five minutes (7550 frames) was rendered in two months, although the overall task was estimated at 32 CPU-months.

The work presented in Cros [50] uses a relatively small (three-machine) distributed environment for ray tracing animations in virtual reality applications. Here, each of the machines has a special task assigned to it according to its processing specialty. In order to achieve close to interactive rates, the system, which takes advantage of progressive refinement, is composed of fairly powerful individual processors connected by an ATM network.

3.2.2. Discussion of Architectural Environments

For parallel processing tasks, the fastest systems will generally be the specialized multiprocessor machines, since they are built with a specific task in mind. Next will be general-purpose multiprocessor machines. Although distributed environments may provide the same number of processors as a multiprocessor machine, computations will be performed more quickly on multiprocessors due to their high-speed interconnection networks. Even so, several factors have motivated a trend toward distributed computing.

First, and perhaps most importantly, not many organizations can afford a parallel machine, which can easily cost millions of dollars Geis [14]. Many sites, however, already have some type of network of computers. Second, multiprocessors often employ specialized or exotic hardware and software resources that significantly increase the complexity, and hence the cost, of the machine; conversely, great expense is rarely incurred to perform distributed computing because the network and the machines are usually already in place. Surprisingly, distributed computing has proven to be so cost-effective that networks of standard workstations have been purchased specifically to run parallel applications that were previously executed on more expensive supercomputers Grop [15].

Due to the fact that networks of workstations are loosely coupled, distributed computing environments allow the network to grow in stages and take advantage of
the latest network technology. As CPUs evolve to faster speeds, workstations can be swapped out for the latest model. Such flexibility in network and processor choice is not usually available on a multiprocessor. Another consideration is system software: operating system interface, editors, compilers, debuggers, etc. A benefit of workstation platforms is that they remain relatively stable over time, allowing programmers to work in familiar environments. To use multiprocessor systems, developers may have to climb a steep learning curve.

Additionally, in a distributed computing networked environment, the interconnected computers often consist of a wide variety of architectures and capabilities. This heterogeneity leads to a rich variety of machine combinations and computing possibilities, which can be tailored to specific applications to reduce overall execution time. On the other hand, a multiprocessor machine does not spend processing time converting data between various machine types, as a distributed computing environment might.

Finally, while utilization and efficiency are extremely important in the multiprocessor world, users on a network of traditional machines rarely consider these issues. The results are underutilized computers, which spend much of their time idle. With distributed computing, some of those idle cycles can be put to good use without impacting the primary users of the machines.

3.2.3. Message-Passing Software for Distributed Computing Environments

To realize distributed computing, computers in a network must support some type of distributed programming environment that allows users to write parallel applications for networked machines. This programming environment should provide a common interface for developers to pass messages easily across various network types and between machines of differing architectures. Although many additional features are usually included, a distributed programming environment need only provide a minimum set of capabilities to be useful Grop [15]:

First, some method must exist to start up and initialize the parallel processes on all participating machines. This procedure may be as simple as specifying each machine and an associated command in a static file, or spawning the processes directly within the program of the master process. Here, the master process refers to a user-initiated process responsible for delegating work and compositing results; conversely, slave processes perform only the work assigned to them and report results back to the master.

Once start-up is complete, a process should be able to identify itself, as well as other processes running on the local machine or remote machines participating in the work. Such identification is useful for specifying the source and destination of transmitted messages.
Since a distributed computing environment often consists of machines with widely varying architectures, message transmission must account for differing data formats so that all computers on the network understand the data exchanged between them. This capability is often built into the programming library, which first transforms the data into a common format that can be easily decoded on the receiver's side. For this reason, among others, a version of the distributed programming environment must exist for every type of machine architecture participating in the computation.

Finally, once an application is complete, some way of terminating all the processes must be available.

Many distributed programming environments have received attention in the last five years, including p4 Butl [51], Express Flow [52], Linda Carr [53], and TCGMSG Harr [54]; however, by far the most popular systems are PVM and MPI. PVM was developed at Emory University and Oak Ridge National Laboratory and was first released in 1991. The MPI standard, an international effort, was introduced in 1993.

Both the PVM and MPI message-passing environments, freely available on the worldwide web, provide common interfaces for communication on both multiprocessors and networks of workstations. Both run on many different machines, allowing networked computers of diverse architectures to emulate a distributed-memory multiprocessor. Each of the machines in the network may be a single-processor or multiprocessor system.

A running process in either PVM or MPI can view a network of computers as a single, virtual machine, ignoring architectural details, or as a set of specialized processors with unique computational abilities. The master process runs on a single computer from which other tasks participating in the computation are initiated. Processes, or tasks, roughly correspond to UNIX processes and operate independently and sequentially, performing both communication and computation. Multiple tasks can run on multiple machines, on a single machine, or a combination of the two.

PVM and MPI are not programming languages; rather, they provide libraries specifying the names, parameters, and results of Fortran and C routines used in message passing. Any program making use of these routines can be compiled with standard compilers by linking in the PVM or MPI library. Note that the developer controls the parallelism in the program by writing master and slave tasks and explicitly specifying the high-level message passing protocol between them. Both PVM and MPI support functional parallelism, in which each task is assigned one function of a larger process; data parallelism, in which identical tasks solve the same problem but for small subsets of the data; or a combination of either approach.
REFERENCES


