25

Heterogeneous Computing

Howard Jay Siegel,[*] John K. Antonio,[*]
Richard C. Metzger,[†] Min Tan,[*] and Yan Alexander Li[*]

A single application task often requires a variety of different types of computation (e.g., operations on arrays versus operations on scalars). Existing supercomputers generally achieve only a fraction of their peak performance on certain portions of such application programs. This is because different subtasks of an application can have very different computational requirements that result in different needs for machine capabilities. In general, it is currently impossible for a single machine architecture to satisfy all the computational requirements of various subtasks in certain applications equally well [36]. Thus, a more appropriate approach for high-performance computing is to construct a heterogeneous computing environment.

25.1 Introduction

A *heterogeneous computing* (*HC*) system provides a variety of architectural capabilities, orchestrated to perform an application whose subtasks have diverse execution requirements. A *mixed-mode* HC system is a single parallel processing machine that is capable of operating in either the synchronous SIMD or asynchronous MIMD mode of parallelism, and can dynamically switch between modes at instruction-level granularity with generally negligible overhead [30]. A *mixed-machine* HC system is a heterogeneous suite of independent machines of different types interconnected by a high-speed network [81]. Unlike mixed-mode machines, switching execution among machines in a mixed-machine system requires measurable overhead because data may need to be transferred among machines. Thus, the mixed-machine systems considered in this chapter are assumed to have high-speed connections among machines that make decomposition at the subtask level feasible.

[*]Supported by Rome Laboratory under contract number F30602-94-C-0022 and by NRaD under subcontract number 20-950001-70. Some of the research discussed used equipment supported by the National Science Foundation under grant number CDA-9015696.
[†]Supported by AFOSR under RL JON 2304F2TK.

Another difference is that, in mixed-machine systems, the set of subtasks may be executed as an ordered sequence and/or concurrently on multiple machines. Mixed-machine HC has also been referred to as *metacomputing* [49, 51].

To fully exploit HC systems, a task must be decomposed into subtasks, where each subtask is computationally homogeneous, and different subtasks may have different machine architectural requirements. The subtasks are then assigned to and executed with the machines (or modes) that will result in a minimal overall execution time for the task. Currently, users typically must specify this decomposition and assignment. One long-term pursuit in the field of heterogeneous computing is to do this automatically.

Figure 25.1 shows a hypothetical example of an application program whose various subtasks are best suited for execution on different machine architectures, i.e., vector, SIMD, MIMD, data-flow, and special purpose [33]. Executing the whole program on a vector supercomputer only gives twice the performance achieved by a baseline serial machine. The vector portion of the program can be executed significantly faster. However, the non-vector portions of the program may only have a slight improvement in execution time due to the mismatch between each subtask's unique computational requirement and the machine architecture being used. Alternatively, the use of five different machines, each matched with the computational requirements of the subtasks for which it is used, can result in an execution 20 times as fast as the baseline serial machine.

A programming language used in an HC environment must be portable. To allow full flexibility of execution targets, the language must be compilable into efficient code for any machine in the mixed-machine suite or any mode available in a mixed-mode machine. Thus, ideally, this portable programming language must be machine/mode-independent and supply the compiler with the information it needs to produce efficient code for different target architectures and/or modes of parallelism. In this chapter, the existence of such a language is assumed. More about this topic is in Ref. [84].

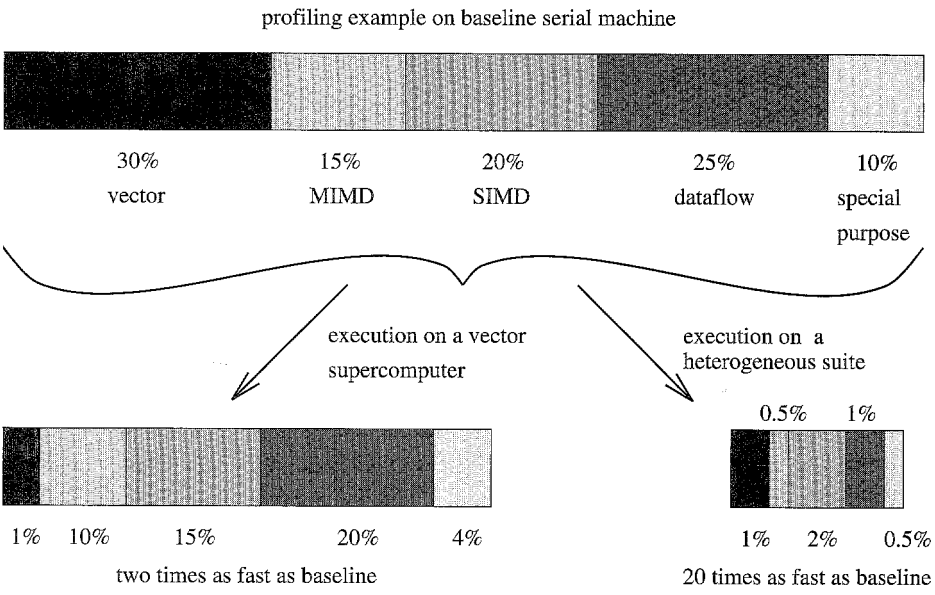


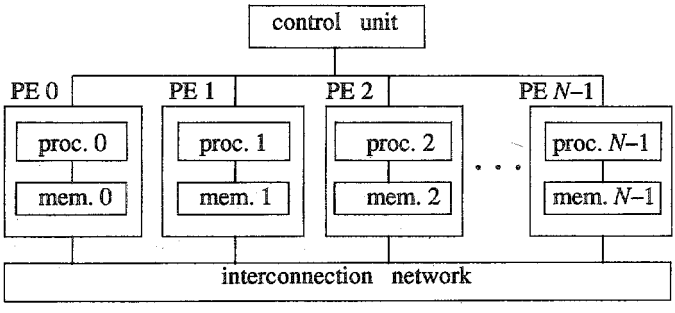
Figure 25.1 A hypothetical example of the advantage of using heterogeneous computing [33] in which the execution time for the heterogeneous suite includes inter-machine communications. Percentages are based on 100% being the total execution time on the baseline serial system but are not drawn to scale.

This chapter is a brief introduction to HC. In Section 25.2, mixed-mode systems are discussed. After Section 25.2, "HC system" will imply "mixed-machine system," as it is more commonly used in that way. Examples of existing mixed-machine systems are presented in Section 25.3. Section 25.4 describes some existing software tools for HC systems. A conceptual model for HC is introduced in Section 25.5. Existing literature that presents explicit frameworks for performing task profiling and analytical benchmarking, a stage in the conceptual model, is overviewed in Section 25.6. In Section 25.7, matching and scheduling techniques for selecting machines for each subtask based on certain cost metrics are overviewed. Finally, open problems in the field of HC are explored in Section 25.8.

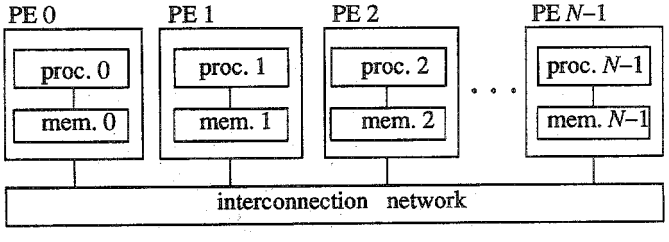
25.2 Mixed-Mode Systems

25.2.1 Trade-offs among SIMD, MIMD, and mixed-mode

Two types of parallel processing systems are the *SIMD* (single instruction stream, multiple data stream) machine and the *MIMD* (multiple instruction stream, multiple data stream) machine [31]. Figure 25.2a shows a distributed memory SIMD architecture in which each processor is paired with a memory module to form *N processing elements* (*PEs*). In SIMD mode, there is a single program and the control unit broadcasts instructions of this program in sequence to the *N* PEs. All enabled PEs execute the same instruction (broadcast by the control unit) at the same time, but each PE operates on data from its own local memory and registers. The interconnection network provides inter-PE communication.



(a)



(b)

Figure 25.2 (a) Distributed memory SIMD machine model and (b) distributed memory MIMD machine model

In an MIMD machine, each PE stores its own instructions and data. Distributed memory MIMD systems are typically structured like SIMD systems without the control unit (see Fig. 25.2b). Each PE executes its own program asynchronously with respect to the other PEs. The use of SIMD and MIMD machines is discussed further in Chapter 16.

There are many trade-offs between SIMD and MIMD machines. The advantages of SIMD mode include:

1. The single instruction stream and implicit synchronization of the PEs make SIMD programs easier to create, understand, and debug.
2. In SIMD mode, the PEs are implicitly synchronized at the instruction level. Explicit synchronization primitives may be required in MIMD mode and generally incur overhead.
3. In SIMD mode, if the PEs communicate through messages, during a given transfer, all enabled PEs send a message to distinct PEs, thereby implicitly synchronizing the "send" and "receive" commands and implicitly identifying the message. MIMD architectures require the overhead of message identification protocols and a scheme to signal when a message has been sent and received.
4. Control flow instructions and scalar operations that are common to all PEs (e.g., computing common local subimage data point addresses) can be executed on the *control unit* (*CU*) while the processors are executing other instructions (this is implementation dependent); this is referred to as *CU/PE overlap* [4, 50].
5. Only a single copy of the instructions needs to be stored in the system memory, thus possibly reducing memory cost and size, allowing for more data storage, and/or reducing communication between primary and secondary memory.
6. Cost is reduced by the need for only a single instruction decoder in the CU (versus one in each PE for MIMD mode).

The advantages of MIMD mode include:

1. MIMD allows different operations to be performed on different PEs simultaneously (i.e., multiple threads of control). Thus, MIMD can support both *functional* (*control*) *parallelism*, where separate and relatively independent processes or functions are assigned to and executed on different sets of processors simultaneously, and *data parallelism*, where the same set of instructions is applied to all the elements in a data set [58, 85]. SIMD is limited to data parallelism.
2. When executing conditional statements (e.g., "if-then-else") based on data local to PEs in MIMD mode, each PE can independently follow either decision path. In SIMD mode, all of the instructions for the "then" block must be broadcast, followed by all of the "else" block, with the appropriate PEs enabled for each block.
3. Consider a sequence of instructions each of whose execution time is data dependent (e.g., a sequence of "while" loops whose bounds are dependent on data local to PEs). In SIMD mode, a PE must wait until all the other PEs have completed an instruction before continuing to the next instruction, resulting in a "sum of maxs" effect:

$$T_{\text{SIMD}} = \sum_{\text{instrs}} \max_{\text{PEs}} (\text{instr. time})$$

MIMD mode allows each PE to execute the sequence of instructions independently, resulting in a "max of sums" effect (see Fig. 25.3):

$$T_{\text{MIMD}} = \max_{\text{PEs}} \sum_{\text{instrs}} (\text{instr. time}) \leq T_{\text{SIMD}}$$

4. MIMD machines do not need the SIMD instruction broadcasting hardware.

The trade-offs above are summarized from Refs. [13, 45, 69]. Because both modes have advantages, mixed-mode systems have been proposed. Various algorithm case studies have shown that the use of mixed-mode can outperform the use of a single-mode (e.g., Refs. [39, 67, 79]).

As a simple example, consider the bitonic sorting [7] of sequences on the mixed-mode PASM prototype [30], where L/N numbers are stored in each of N PEs and the numbers within a PE are sorted. The goal is to have each PE contain a sorted list of L/N elements, where each element in PE i is less than or equal to all of the elements in PE k, for $i < k$. The regular bitonic sorting algorithm for $L = N$ is modified for $L > N$ (see Fig. 25.4). An ordered merge is done between the local PE sequence X and the transferred sequence Y using local data conditional statements in merge(X, Y). The lesser half of the merged sequence is assigned the pointer X and the greater half is assigned the pointer Y. These pointers may be swapped by swap(X, Y), based on a precomputed data-independent mask.

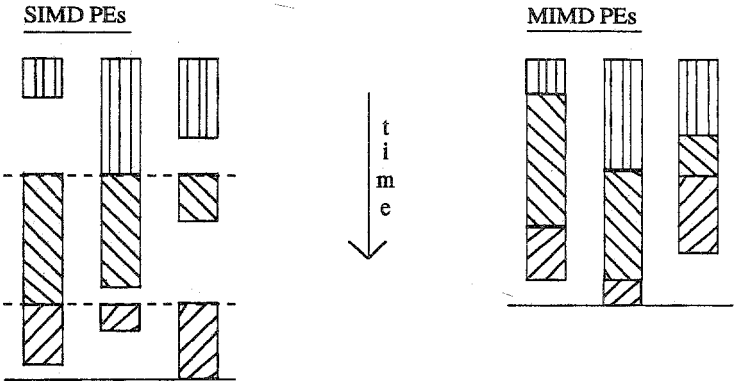


Figure 25.3 *Sum of maxs* vs. *max of sums* effects

---

for $k = 1$ to $\log_2 N$ do
  for $i = 1$ to k do
    { for $q = 1$ to L/N do
      { load $X[q]$ into network
        send to PE whose number differs in bit $(k - i)$
        $Y[q] \leftarrow$ network output }
    merge(X, Y)
    swap(X, Y) }

Figure 25.4 Bitonic sequence-sorting algorithm [30]

The ordered merge involves many comparisons, which can be more efficiently computed in MIMD mode. The innermost loop of the algorithm requires many network transfers, which are better performed in SIMD mode. In a mixed-mode implementation, the ordered merge and swap routines can be executed in MIMD mode, while the other operations are performed in SIMD mode. This approach has an advantage over pure SIMD or pure MIMD mode implementations because all comparisons are done in MIMD mode and all network transfers are done in SIMD mode. Additionally, there is opportunity in SIMD mode for CU/PE overlap. It is shown in Ref. [30] that there is a noticeable improvement in execution time for the mixed-mode implementation as a result of properties inherent to the modes of parallelism.

Most of the advantages of SIMD and MIMD modes can be realized with a mixed-mode architecture. Disadvantages of mixed-mode parallelism include higher hardware cost (because mixed-mode machines must have the hardware needed for both modes), more complicated use (because the mode switching ability adds another dimension of complexity for the programmer), and, when switching from MIMD to SIMD mode, some PEs may remain idle while they wait for the other PEs to reach the switch point (which they may not need to do if only MIMD mode was used) [12]. Very brief descriptions of five existing mixed-mode systems follow, emphasizing the mode-switching mechanism.

25.2.2 Partitionable SIMD/MIMD

PASM is a *PA*rtitionable-*S*IMD/*M*IMD system concept being developed as a design for a large-scale distributed memory dynamically reconfigurable parallel machine [70, 71]. PASM can be dynamically partitioned to form independent mixed-mode submachines of various sizes. PASM uses a fault-tolerant implementation of the flexible multistage cube network [68] (the Extra Stage Cube [1]), for inter-PE communication. Thus, PASM is dynamically reconfigurable along three dimensions: partitionability, mode of parallelism, and connections among PEs. A small-scale proof-of-concept prototype (30 processors, 16 PEs in the computational engine) has been built at Purdue University, in the USA. The prototype is a constantly evolving tool for studying the design and use of reconfigurable parallel machines.

Consider a single submachine. In SIMD mode, a PE fetches SIMD instructions by issuing a read to a reserved segment of the logical address space (that does not correspond to physical PE memory). Each memory access made by a PE's processor is monitored by the *instruction broadcast unit* (IBU). The IBU sends an SIMD instruction request to the control unit, and when all enabled PEs in a submachine have requested a new instruction, it is broadcast from a queue in the control unit. In MIMD, a PE fetches instructions from its local memory. A PE can switch from SIMD mode to an MIMD program located at some address A in its local memory by receiving a "branch to A" instruction in SIMD mode. Similarly, a PE can change from MIMD mode to SIMD mode by executing a branch to the logical SIMD instruction space. Such flexibility in mode switching allows mixed-mode programs to be written that change modes at instruction-level granularity with generally nominal overhead.

25.2.3 Texas Reconfigurable Array Computer

The *Texas Reconfigurable Array Computer* (TRAC) is a dynamically partitionable mixed-mode shared-memory parallel machine which was developed at the University of Texas at Austin, in the U.S.A. [57]. The TRAC prototype consisted of four microprocessors con-

nected to nine memory modules by an SW-Banyan network with fan-out of three, spread of two, and two levels (see Fig. 25.5). *Data trees* connect data memories with their corresponding processors. An *instruction tree* connects a specific program memory with processors to form a virtual SIMD machine. In MIMD mode, each processor can independently fetch its own instructions from a memory module associated with it. Mode switching between SIMD and MIMD is implemented by changing the source of the instructions for the processors.

25.2.4 OPSILA

OPSILA is a limited mixed-mode 16-PE prototype built at the University of Nice, in France [27]. It operates in SIMD and *SPMD* (single program - multiple data stream) mode, a special form of MIMD mode where all the PEs execute the same program in an asynchronous fashion, each on its own data [24]. A synchronous Omega network [53] (a member of the multistage cube family [68]) is used for inter-PE communication.

The central control unit consists of the *scalar processor* (*SP*) and the *instruction processor* (*IP*). In SIMD mode, the program is stored entirely in the *scalar memory* (*SM*) managed by the SP. The IP broadcasts SIMD instructions to the PEs. In SPMD mode, the same program is duplicated in each PE memory. SPMD mode is initialized by the IP, which provides each PE with the starting SPMD code address. The synchronization mechanism for initializing the SPMD mode and for returning to SIMD mode is a fork-join operation executed over the set of PEs. The transition from SPMD to SIMD mode is made in one machine cycle after the last PE executes the join. Inter-PE data transfers can only occur in SIMD mode.

25.2.5 Triton

Triton is a mixed-mode machine being developed at the University of Karlsruhe, in Germany [44, 64]. The Triton/1 prototype will consist of 260 nodes (four are for fault tolerance), though the Triton concept is scalable up to 4096 nodes. Each node consists of a processor/memory pair, a memory management unit, a numeric coprocessor, a SCSI interface, and a network processor. Triton uses a generalized De Bruijn network for inter-PE communication.

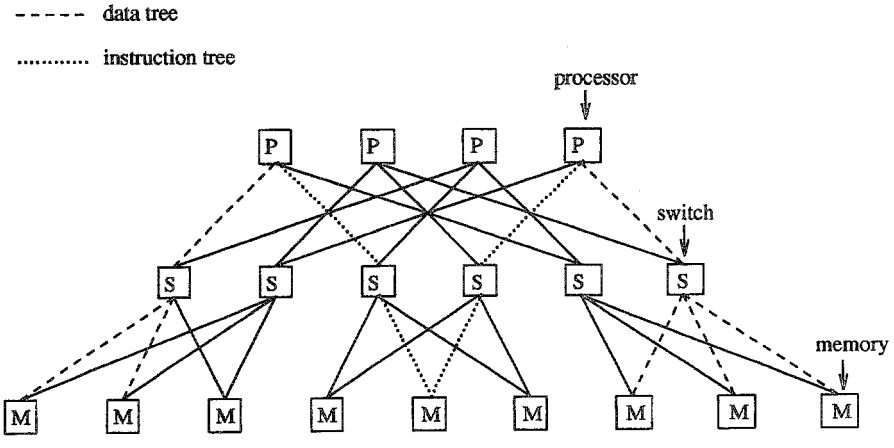


Figure 25.5 A task tree (instruction tree and data tree) of TRAC 1.1 [3]

In SIMD mode, a single front-end processor produces the instruction stream for all PEs. If a PE is not selected to execute an instruction, a local signal for the instruction stream is turned off and the corresponding PE is disabled. To switch to MIMD mode, the program must be downloaded to the local memory of the PEs. This is done via load instructions in SIMD mode. The switch from SIMD to MIMD mode is accomplished by setting the program counter to the MIMD program location and deactivating the SIMD request bit for each PE. To switch from MIMD to SIMD mode, the SIMD request bit for each PE is activated. The result of a global-wired-or operation of all PEs' SIMD request bits instructs the front-end processor to activate SIMD mode. Then each PE switches to SIMD mode and the next instruction is from the instruction stream broadcast by the front-end processor.

25.2.6 EXECUBE chip

The *EXECUBE chip* consists of eight 16-bit CPU mixed-mode PEs, each with a 64 kB memory module [52]. A hypercube interconnection network is used for inter-PE communication. This is all contained on a single chip developed by IBM Federal Systems Division, in the USA. A system with 64 EXECUBE chips (512 CPUs) has been constructed.

In SIMD mode, instructions are sent into each PE's instruction register by a separate controller via the SIMD broadcast bus. In MIMD mode, each PE obtains its own instructions from its local memory. Because the only way of accessing the memory system of each PE is through its CPU, MIMD instructions are sent and stored into participating PEs' local memory in SIMD mode via the SIMD broadcast bus. Arbitrary collections of PEs can be in either mode simultaneously. Mode switching instructions are machine operation codes that activate special hardware functions. By executing an instruction to "switch to MIMD mode," participating PEs begin execution at a specified address in local memory. After executing a switching instruction, the participating PEs stop fetching instructions from the SIMD broadcast bus and start to execute the instructions stored in local memory. A "switch to SIMD mode" instruction causes PEs to fetch instructions from the SIMD broadcast bus. A collective signal from the PEs is sent to the controller that sends SIMD instructions to each PE's instruction register. If any PE in the PE group that is changing to SIMD mode is still in MIMD execution, then the controller will wait until the collective signal from the PEs is set, at which point SIMD execution is started.

25.2.7 Conclusions

Mixed-mode machines are one extreme form of HC. Decomposing a task for mixed-mode execution is easier than for mixed-machine execution because three major problems in the use of mixed-machine HC are not present: moving data among machines, concurrent use of multiple machines, and determining machine loads. The study of mixed-mode machines provides valuable information about the trade-offs between SIMD and MIMD parallelism, explores the advantages and disadvantages of mixed-mode computation as a mode of parallelism, and establishes a relatively simpler environment for developing algorithm mapping techniques that may possibly be adapted to the mixed-machine arena. For example, a block-based mode selection methodology developed for mixed-mode machines, presented in Ref. [82], was then extended for use as a heuristic for the mixed-machine case [81] (see Section 25.7.3). Sections 25.3 through 25.7 focus on mixed-machine HC.

A variation on mixed-mode is a *mixed-component* HC system, where each separate component of a single machine represents one mode of parallelism, and two or more distinct modes are present in the machine. Examples are the SIMD/MIMD Image Under-

standing Architecture [83] and the SIMD/vector Cray-3/SSS Super Scalable System [23]. Mixed-component systems are outside the scope of this chapter.

25.3 Examples of Existing Mixed-Machine HC Systems

25.3.1 Simulation of mixing in turbulent convection at the Minnesota Supercomputer Center

In Ref. [51], the usefulness of an HC system developed at the Minnesota Supercomputer Center is demonstrated through a particular application involving the simulation of mixing in turbulent convection in three dimensions. The particular HC system developed consists of Thinking Machines' CM-200 and CM-5, a CRAY 2, and a Silicon Graphics VGX workstation, all interconnected over a high-speed *HiPPI* (high-performance parallel interface) network.

The required calculations for the simulation were divided into three phases: (1) calculation of velocity and temperature fields, (2) calculation of particle traces, and (3) calculation of particle distribution statistics and refinement of the temperature field. The velocity and temperature fields associated with the phase 1 calculations are governed by two second order partial differential equations. Three-dimensional cubic splines (over a grid of size $128 \times 128 \times 64$) were used to approximate the velocity and temperature fields in these equations, resulting in a linear system of equations for the unknown spline coefficients. A conjugate gradient method was applied to solve this system of equations. These computations were done on the CM-5. At each time step, the grid of $128 \times 128 \times 64$ spline coefficients were transferred to the CRAY 2, where the calculation of the particle traces were done.

The particle traces (phase 2) were calculated by solving a set of ordinary differential equations based on the velocity field solution from phase 1. This computation was attempted on the CM-200 by employing an Eulerian approach. Although this approach worked well for a two-dimensional instance of the problem, the same approach could not be used for the three-dimensional simulations because a prohibitive amount of memory was required. Instead, the three-dimensional simulations were implemented using a vectorized Lagrangian approach on the CRAY 2, which required substantially less memory than the parallel Eulerian scheme. The coordinates of the particles and the spline coefficients of the temperature field were then sent from the CRAY 2 to the CM-200.

The CM-200 was used to calculate statistics of the particle distribution and to assemble a three-dimensional temperature field from the associated spline coefficients (phase 3). A $256 \times 256 \times 128$ point temperature field file was produced from the $128 \times 128 \times 64$ grid of splines, representing a volume of eight million voxels (a *voxel* is a three-dimensional element). This file of voxels and the coordinates of the particles (one million particles were used) were then sent to an SGI VGX workstation, where they were visualized using an interactive volume renderer.

The application was successful in demonstrating the benefits of HC. However, in Ref. [51], it is noted that there is still much work to be done to improve the environment for developing HC applications.

25.3.2 Interactive rendering of multiple Earth science data sets on the CASA testbed

In 1990, the National Science Foundation (NSF), in conjunction with the Defense Advanced Research Projects Agency (DARPA), established a program to conduct research

in the area of networking at gigabit per second speeds [72]. In this and the next subsection, two applications that utilize the HC resources available on two of the testbeds are overviewed.

The CASA testbed interconnects several remote sites including the California Institute of Technology, San Diego Supercomputer Center, Jet Propulsion Laboratory (JPL), and Los Alamos National Laboratory. One of the applications developed on the CASA testbed involves interactive three-dimensional rendering of multiple Earth science data sets. Geology can be regarded as a "three-dimensional science," in the sense that both surface and subsurface data from the Earth are collected and studied. In the past, these two types of data were generally collected and analyzed separately. By making effective use of the computing and networking resources of the CASA testbed, researchers can construct a more complete image of the Earth's surface and subsurface, together, by combining multiple sets of data from various sources. The required processing and communication for merging these data sets should be fast enough to enable interactive manipulation of the associated image. According to Ref. [14], researchers can rotate, slice, zoom, and "fly over" a full-color view of the Earth's surface and subsurface while sitting at a workstation.

The software for the application is divided into three categories: (1) a collection of functionally distinct two-dimensional image processing modules that generate and/or manipulate color images and elevation data, (2) a rendering process that combines data and creates an electronic rendered image, and (3) the network and control software that coordinate the various processes. The two-dimensional modules are implemented using Network Express, which is a portable, message passing, programming environment developed by the ParaSoft Corporation. Initially, raw data sets are transferred to one of the two-dimensional functional modules for processing. The two-dimensional modules manipulate image and/or elevation data via a number of different algorithms. Most of the two-dimensional modules were developed for the CRAY Y-MP/232 at JPL and the CRAY Y-MP8/864 at the San Diego Supercomputer Center. Two of the two-dimensional modules were implemented on the CM-5 and CM-200 located at Los Alamos. Output from the two-dimensional modules are sent over the network to the three-dimensional rendering process, which was implemented on the Intel Touchstone Delta located at the California Institute of Technology.

In the current implementation of the CASA testbed, there are high-speed HiPPI connections only among machines located at a common geographical site. The current connections among the distributed sites, which utilize lower speed networks, will be upgraded by using HiPPI-SONET gateways to interconnect each site's local HiPPI network to a wide area high-speed SONET network. Future work includes experimenting with this application over this new high-speed HiPPI/SONET network.

25.3.3 Using VISTAnet to compute radiation treatment planning for cancer patients

VISTAnet is also in the group of gigabit testbeds mentioned in the last subsection. The VISTAnet testbed sites include the Center for Communications and Signal Processing at North Carolina State University, BellSouth, GTE, and three organizations within the University of North Carolina at Chapel Hill (the Graphics and Image Laboratory in the Department of Computer Science, the Microelectronics Systems Laboratory in the Department of Computer Science, and the Department of Radiation Oncology) [73]. The machines connected to the testbed include a CRAY Y-MP, a Pixel-Planes 5, a MasPar MP-1, and Silicon Graphics workstations.

A major application focus for this testbed has been the computation of radiation treatment planning for cancer patients [66]. Radiation is effective in treating the cancer only if it is delivered to the tumorous cells in a high dose while sparing the nontumorous cells. The physician must determine the number of treatment beams to be used, the beam angles and shapes, the time the beam is to be activated, and which custom filters to use to alter the beam. This process is know as *radiation treatment planning* and in the past was carried out in only two spatial dimensions; however, some types of cancer require that the planning take place in three dimensions to achieve maximum effectiveness. This requires advanced modeling of human anatomy (rendered from tomography scans) as well as three-dimensional modeling of the radiation beam (i.e., the treatment plan). In the application, the treatment plan model is superimposed onto the anatomical model. One of the objectives is to provide a visualization of these models that can be rotated, zoomed, and/or modified interactively.

The CRAY Y-MP was demonstrated to be ideal for radiation dose calculation and interpolation throughout the entire model. The Pixel-Planes 5 machine (which contains a quarter-million custom one-bit processors) is designed for rendering images and is used for shading and merging large amounts of image data. The physician interacts with the system via a medical workstation hosted on a Silicon Graphics 340 VGX. From this workstation, the physician can modify the treatment plan based on the current dosage patterns and can adjust the view by rotating the image. When an image viewpoint is adjusted, the new viewpoint information is sent to the Pixel-Planes 5, which renders the otherwise unchanged data according to the new viewing angle and presents the new image to the physician at the workstation. If the treatment plan is modified, the new treatment plan information is sent to the CRAY Y-MP, which computes the new three-dimensional dose distribution and sends the information to the Pixel-Planes 5 for rendering.

In the future, a MasPar MP-1 will be integrated into the application and will receive the three-dimensional dose distribution generated by the CRAY Y-MP. With this information, the MP-1 will be used to compute a statistical analysis of the treatment plan in relation to the anatomical data. This computed information will provide the physician with a quantitative measure of merit for each treatment plan.

25.4 Examples of Software Tools for Mixed-Machine HC Systems

25.4.1 Overview

A variety of software tools and environments have been implemented to assist programmers in developing applications to execute on a mixed-machine HC system. A common feature among most of the existing tools is that they create a layer of abstraction between programmers and the suite of machines. Some also provide explicit constructs needed to express synchronization and communication among tasks within the application. The following subsections discuss examples of software tools that exist or are being developed for HC systems. The functionalities of most of the tools described in this section tend to evolve and change rapidly; the descriptions here are based on the references given. A survey of distributed queueing and clustering systems, some of which can be applied to HC, is given in Ref. [46].

25.4.2 Linda

Linda was originally implemented for *homogeneous* computing environments such as shared memory parallel computers (e.g., the Sequent Symmetry), distributed memory

computers (e.g., the Intel iPSC/2), and local area networks (e.g., a network of workstations). As suggested in Ref. [19], it is an attractive choice for HC systems as well. In Linda, processes communicate via persistent objects called *tuples*, and not through transient events such as message passing or procedure calls. A process can generate a tuple and place it in a globally shared collection of tuples, the *tuple space*. Tuples can be also removed, read, and evaluated from the tuple space. *Process tuples* incorporate executable code and *data tuples* are passive, ordered collections of data items [16]. Although the current version of Linda does not support concurrent interaction among machines in an HC system, the issues that must be resolved to do this are outlined and discussed in Ref. [19].

25.4.3 p4

p4 is a set of parallel programming tools designed to support portability across a wide range of architectures [16–18]. p4 includes high-level operations that allow certain procedure calls to be replaced with the equivalent p4 calls that are implemented by utilizing system-specific procedures. The long-term goal of this project is to allow a single program to be written for an entire class of systems (e.g., message passing) without requiring the explicit utilization of constructs of the specific system (e.g., Intel Paragon versus nCUBE 2) in the source code. The p4 function library is linked with the source code to provide functions for message passing, shared memory monitoring, process management, debugging, and language interfacing.

Three classes of architectures are supported by p4. The first consists of shared memory multiprocessors (e.g., the Alliant FX/8). p4 provides monitor data types for encapsulating shared data and controlling access. The second consists of distributed memory systems that implement communication through message passing, specifically distributed memory multiprocessors and groups of workstations that communicate over a network [15]. The third consists of "communicating clusters," which can include multiprocessor machines that communicate via shared-memory and/or through the exchange of messages. Therefore, p4 can support communication within and among both shared-memory and message-passing machines.

The process of executing a p4 program begins with the user compiling the code for the desired set of machines. The configuration of the system is specified by a *procgroup file*, which defines the number, names, and target machines of the programs to be executed. This allows the user to experiment with different configurations and machines. p4 also has a utility called ALOG, which creates a log of time-stamped events captured during program execution, and can be used with C or FORTRAN. This event log can then be used as an input file for a graphical tool called Upshot [43]. With Upshot, the log file can be examined in detail to detect computational and/or communication bottlenecks.

The developers of p4 stress that it is not an "abstract tool" and that various components of p4 evolved through the development of real applications. As an example, p4 was used in developing a piezoelectric crystal simulation program to coordinate the computations and communications among an Intel Touchstone Delta, the graphical output on a Stardent Titan, and a Solbourne workstation (which was used as an I/O server). Current and future research directions for p4 include the implementation of Linda with p4 to provide a single high-level programming model.

25.4.4 Mentat

Mentat consists of execution time support facilities and language abstractions that provide a clear separation between the user's application and the target machine [41]. This separa-

tion is achieved by using an object-oriented language to specify parallelism within the application and compiler technology to handle many of the tedious and time consuming bookkeeping tasks. Mentat combines a medium-grain dataflow computation model with the object-oriented programming paradigm to produce a system that facilitates hierarchies of parallelism [40]. Programs are characterized as directed graphs. The vertices represent computational elements (e.g., class member functions) and the edges model data dependencies between these elements. The idea behind Mentat is to allow the programmer to express the problem in a C++ based language, called *MPL* (Mentat programming language), which provides many popular features of the C++ language. Mentat uses the dataflow model to exploit the inherent medium-grain parallelism of the program; in addition, the programmer can specify those C++ classes which are themselves of sufficient computational complexity to warrant parallel execution [40].

The use of object-oriented programming languages, such as MPL, masks much of the underlying complexity from the user and is the basis for "separating" the user from the various machines in the HC system. The basic unit of computation in MPL is the Mentat class instance, which consists of objects (e.g., local and member variables), their procedures, and a thread of control [41]. In MPL, the standard object-oriented notions of data encapsulation and method encapsulation have been extended to include "parallelism encapsulation" [40]. MPL supports two types of parallelism encapsulation that are hidden from the user: intraobject (within a member function) and interobject (among member-function invocations). For interobject, it is the responsibility of the MPL compiler to ensure that data dependencies between invocations are satisfied and that communication and synchronization are handled correctly [40]. The MPL compiler maps MPL programs onto the dataflow model by translating the MPL programs into C++ programs with embedded calls to the Mentat execution time system. These C++ programs are then compiled by the host C++ compiler resulting in executable object code.

A distinguishing feature of MPL is its implementation of a construct called *rtf* (return-to-future) [40], which is analogous to the "return" function commonly found in imperative languages such as C. The rtf construct allows Mentat member functions to return values to successor nodes in the macro-dataflow graph. These returned values are forwarded to all member functions (of the successor nodes) that are dependent on the result. The rtf function differs from a standard return in three ways. First, a member function may "rtf a value" from a Mentat-object member function that has not completed execution. Second, the execution of rtf indicates only that the associated values are ready (additional computation may be carried out after the rtf call). Finally, depending on the program's data dependency structure, rtf may not return data to its caller. In particular, if the caller does not use the resulting values locally, then the caller does not receive a copy of the values.

The *RTS* (run time system) of Mentat, which initially supported execution on homogeneous parallel machines, has been extended to support HC systems. The RTS uses a *virtual macro-dataflow machine* that provides support routines to perform execution time data dependence detection, program graph construction, program graph execution, scheduling, communication, and synchronization [40, 41]. The virtual macro-dataflow machine contains a set of machine-independent components and libraries and a set of machine-dependent components. The virtual macro-dataflow machine can be ported to any supported machine in the HC system by changing only the machine-dependent components, which allows the user to port the application source code to any supported machine without changes.

The RTS has been implemented for several platforms including a network of Sun workstations, the Silicon Graphics Iris, and the Intel iPSC/2. Matrix multiplication and Gaussian elimination programs have been coded in MPL and executed on a network of eight Sun

workstations and a 32 node iPSC/2. While MPL improved the ease of use of the HC system, it was indicated that the performance may not be as good as hand-coded versions that use send and receive protocols. Thus, there is a trade-off between ease of use and some performance degradation. Future work includes the implementation of several optimizations for the MPL compiler.

25.4.5 PVM, Xab, and HeNCE

PVM (parallel virtual machine). The *parallel virtual machine* (*PVM*) is a software system that enables a mixed-machine HC system to be used as a coherent, flexible, and concurrent computational resource [11, 74, 75]. The PVM package includes system level daemons, called *pvmds*, which reside on each computer in the HC system, and a library of PVM interface routines.

The pvmds provide services to both local processes and remote processes on other platforms in the HC system. Together, the entire collection of pvmds form what is called a "virtual machine" by enabling the HC system to be viewed as a single "meta-computer." Two of the major services provided by the pvmds are communication and synchronization. Processes communicate via the use of messages, which are exchanged asynchronously so that a sending process may continue execution without waiting for an acknowledgment from the receiving process. The other major service provided is the synchronization among processes, which can be accomplished by using barriers or by using event rendezvous. The synchronizations may be among multiple processes that are executing on a local machine or may be among processes on different machines.

The second part of the PVM package is a library of interface routines. Applications to be executed on one or more computing platforms in the HC system are able to access these platforms via library calls embedded in imperative procedural languages such as C or FORTRAN. The library routines interact with the pvmd (resident on each machine) to provide services such as communication, synchronization, and process management. The pvmd may provide the requested service alone or in cooperation with other pvmds in the HC system.

From the user's point of view, the PVM system can be conceptualized as a three-level hierarchy. At the uppermost layer, which is the interface to the programmer, is the concept of an *instance* (or process), which is the basic unit of computational abstraction in PVM. Applications developed with PVM generally consist of several instances (possibly executing concurrently) that cooperate across machine boundaries. The middle layer is defined as the virtual machine layer. The virtual machine layer consists of the pvmds that reside on the machines of the HC system. The lowest layer is the actual set of machines in the HC system.

The computational resources in the HC system may be accessed using three different modes: (1) the transparent mode, in which instances are automatically located at the most appropriate sites based upon a user-specified cost matrix, (2) the architecture-dependent mode, in which the user can indicate specific architecture types on which particular instances are to execute, and (3) the low-level mode, in which particular machines may be specified by the user. The PVM tools described next can be used in any of these access modes.

Xab (X-window analysis and debugging). *Xab* (X-window analysis and debugging) is a tool developed for the execution time monitoring of PVM programs [8, 11]. The Xab tool

gives the user direct feedback on what PVM functions the program is executing and how the program is performing in an HC environment. Xab consists of three parts: the Xab library, which contains instrumented PVM routines that are linked to the user's code, a special monitoring process called *admon*, which receives trace messages from the library routines, and a front-end process, which graphically displays trace events.

Xab monitors a user's program by instrumenting calls to the PVM library. Each instrumented call not only performs its intended PVM function, but also sends an Xab event message to the admon process. (The Xab event message is itself a PVM message.) An Xab event message generally includes an event type, a time stamp, and event-specific information.

The admon process receives Xab event messages to be sent either to a file or to the Xab display process, which displays each event captured in an X-window. The user can single-step through these events or allow Xab to replay the events continuously in real-time.

HeNCE (heterogeneous network computing environment). *HeNCE* (heterogeneous network computing environment) aids users of PVM in decomposing their application into subtasks and deciding how to allocate these subtasks onto the available machines in the HC system [9–11, 75]. In HeNCE, the programmer explicitly specifies the parallelism for an application by drawing a directed graph, where nodes represent subtasks written in either FORTRAN or C, and arcs represent dependencies and flow control. There are also four types of control constructs: conditional, looping, fan-out, and pipelining.

The user must specify a cost matrix, which represents the cost of executing each subtask on each machine in the HC system. The meaning of the cost parameters are defined by the user (e.g., estimated execution times or utilization costs in terms of dollars). At execution time, HeNCE uses the cost matrix to estimate the most cost effective machine on which to execute each subtask.

Given the graph and the matrix, HeNCE configures a subset of the machines defined in the cost matrix as a "virtual machine" using PVM constructs. Then HeNCE begins execution of the program. Each node in the graph is realized by a distinct process on some machine. The nodes communicate with each other by sending parameter values needed for execution of a given node, which are specified by the user for each node (subtask). A node obtains parameter values needed to begin execution from predecessors nodes. If the immediate predecessors do not have all the required parameters for a node, earlier predecessors are checked until all required parameters are found. Then the node (subtask) is executed and passes the appropriate parameters onto descendant nodes.

HeNCE can trace the execution of the application for display in real-time or replay later. The trace tool displays active machines in the network as icons whose colors change depending on whether they are computing or communicating. The tool also displays the user's directed graph and dynamically illustrates paths of execution. This can enable the programmer to detect bottlenecks in the application by displaying the states of the subtasks during execution. The trace animation can also be used for performance tuning, i.e., the programmer can reallocate subtasks across the machines in the HC system and tune the application's behavior to match the environment for subsequent executions of the application.

25.5 A Conceptual Model for Automatic Mixed-Machine HC

A conceptual model for automatic task decomposition, matching of subtasks to machines, and scheduling of subtasks in a mixed-machine HC environment is shown in Fig. 25.6. It

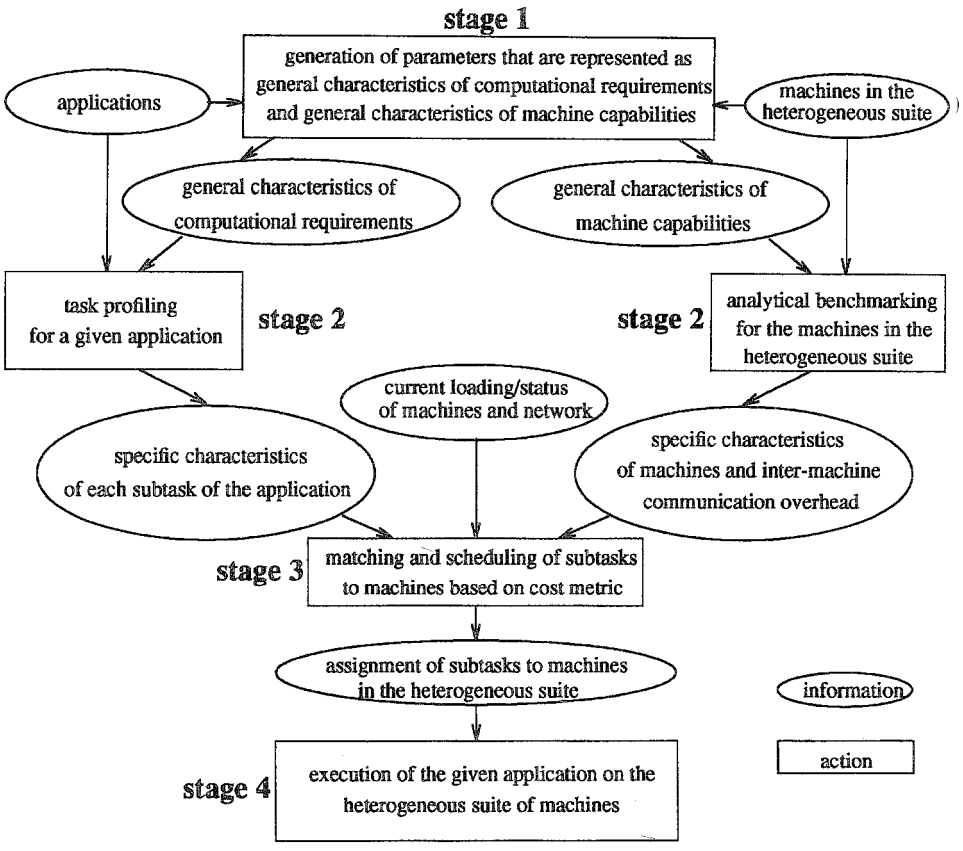


Figure 25.6 Conceptual model of the automatic assignment of subtasks to machines in an HC environment

builds on the one presented in Ref. [36] and is referred to as a "conceptual" model because no complete automatic implementation currently exists. Automatic implementation of HC is a long-term goal that is important for encouraging and facilitating the use of HC, and for improving HC system performance.

In stage 1, using information about the expected types of application tasks and about the machines in the HC suite, a set of parameters is generated that is relevant to both the computational requirements of the applications and the machine capabilities of the HC system. These parameters define the multidimensional decision space to be used for describing and matching subtasks and machines. For each parameter, computational requirements and machine architecture features are derived. For example, considering the parameter "floating point operations," the computational requirements of the application tasks to be quantified are the number and types of the floating point operations needed to perform the calculation. The architecture features of the machines in the HC suite to be quantified are the speeds for these different types of floating point operations. Irrelevant parameters are excluded. For example, if the given applications have no floating point operations, then it is not necessary to evaluate the machine capabilities for executing floating point operations; if there is no vector machine available in the suite, vectorizable code may be excluded from the set of the computational requirements considered.

The total number of parameters enumerated determines the complexity of this automation problem. The chosen parameters evolve dynamically when new types of applications and/or new types of machines are added.

In stage 2, *task profiling* decomposes the application task into subtasks, each of which is computationally homogeneous. The computational requirements for each subtask are then quantified. The term often used for this step in the literature is code profiling. The reason for using task profiling in this chapter instead is that, to identify the types of computational requirements present in a specific task, both the code and data upon which the specified HC system will operate must be profiled. *Analytical benchmarking* is used in stage 2 to quantify how effectively each of the available machines in the suite performs on each of the types of computations being considered. Existing literature that presents explicit methodologies for performing task profiling and analytical benchmarking in the context of HC is reviewed in Section 25.6 of this chapter.

One of the functions of stage 3 is to use the information from stage 2 to derive, for a given application, the estimated execution time of each subtask on each machine in the HC suite [55] and the inter-machine communication overhead associated with each assignment of subtasks to machines. In stage 3, these static results and the dynamic information about the current loading and "status" of the machines and inter-machine network are used to generate an assignment of the subtasks to machines in the HC system based on certain cost metrics. The "status" could include whether the machines/network are fully or partially functioning due to faults, and when other tasks using the machines/network are expected to complete. The most common cost metric for HC is to minimize the overall execution time (including the inter-machine communication time) of a given application task on a particular HC system. Another problem is to find the most appropriate suite of heterogeneous machines for a given collection of applications, such that the cost of the corresponding HC system is minimized for a given set of execution time constraints [32]. Section 25.7 of this chapter presents a variety of techniques in the literature for matching subtasks and machines.

Stage 4 is the execution of the given applications on the HC suite. Because the loading/status of the machines/network may change, sometimes it is necessary to reselect machines for certain subtasks by reactivating stage 3. Techniques for the migration of a subtask from one type of machine to another in the middle of execution present a difficult problem; one approach is described in Ref. [5].

Automatic HC is a relatively new field. The task profiling, analytical benchmarking, and matching and scheduling techniques discussed in Sections 25.6 and 25.7 are representative frameworks that require further research before they are practical tools.

25.6 Task Profiling and Analytical Benchmarking

25.6.1 Overview

To execute a task on a mixed-machine HC system, the task must be decomposed into a collection of subtasks, where each subtask is a homogeneous code *block*, such that the computations within a given code block have similar processing requirements (e.g., see Refs. [21, 32, 35, 49, 75, 80]). These homogeneous code blocks are then assigned to different types of machines to minimize the overall execution time. In some cases, it is better not to use the best-matched machine because of the overhead involved in any inter-machine data transfer that may be needed. Thus, it is important to know the degree to which a code block matches each machine.

This section presents example methodologies for the task profiling and analytical benchmarking that comprise stage 2 of the conceptual model in Section 25.5. These methodologies make many simplifying operating assumptions and are frameworks, rather than fully implemented systems. Further research is needed before they are practical tools that can provide the quantitative results needed for matching and scheduling (stage 3).

25.6.2 Definitions of task profiling and analytical benchmarking

Task profiling is a method used to quantify the types of computations that are present in the application program [32]. Task profiling divides the source program into homogeneous code blocks based on the types of computations required. The definition of the set of code-types is based on the features of the machine architectures available and the computational requirements of the applications being considered for execution on the HC system. This is done in stage 1 of the conceptual model in Section 25.5.

Analytical benchmarking is a procedure that provides a measure of how well each of the available machines in the heterogeneous suite performs on each of the given code-types [32]. Together, the task profiling and analytical benchmarking steps provide the information needed for the matching and scheduling step, which is described in Section 25.7. The performance of a particular kind of machine on a specific code-type is a multivariable function. The parameters (i.e., variables) for this performance function can include the requirements (e.g., data precision) of the application, the size of the data set to be processed, the algorithm to be applied, the programmer's and compiler's efforts to optimize the program, and the operating system and architecture of the machine that will execute the specific code-type [38].

There are a variety of mathematical formulations, collectively called *selection theory*, that have been proposed to choose the appropriate machine for each code block. Many (e.g., Refs. [21, 48, 80]) define analytical benchmarking as a method of measuring the optimal speedup a particular kind of machine can achieve compared to a baseline system when the best-matched code-type for that machine is executed. The ratio between the actual speedup and the optimal speedup defines how well a code block is matched with each machine type, and the actual speedup, in general, is less than the optimal speedup.

25.6.3 Methodologies for performing task profiling and analytical benchmarking

A comparison between traditional benchmarking and analytical benchmarking. One of the most widely used traditional benchmarking techniques is to execute a set of well-studied programs on a machine (e.g., see Refs. [22] and [26]), using the total execution time as the final measure to compare that specific machine's performance with that of others. But in the context of HC, only code blocks, rather than a whole program, are executed on a specific type of computer. Traditional benchmarking techniques do not reflect the performance of a particular kind of machine on a specific code-type. The problem with these traditional benchmarking techniques is that they are not *analytical*.

The techniques for analytical benchmarking should not only be able to show the overall execution time of a specific kind of machine on a certain type of code, but should also be able to predict future capabilities when new types of machines and/or new types of applications are added [33]. As introduced in Ref. [33], the goal of analytical benchmarking is

to construct a class of relatively basic benchmarking programs for each type of computer available in the HC suite. A set of benchmarking programs can be used to derive the performance metrics of the system for a range of conditions, such as the size of the input data file and the type of calculations required. This is in contrast to the usual benchmarking program, whose result is just the execution time.

Parallel assessment window system. *Parallel assessment window system* (*PAWS*) is an experimental platform capable of performing machine and application evaluations for task profiling and analytical benchmarking. It consists of four tools: the application characterization tool, the architecture characterization tool, the performance assessment tool, and the interactive graphical display tool [63].

The *application characterization tool* transforms a given program written in Ada into IF1, an acyclic graphical language that illustrates the program's data dependencies. In IF1, basic operations, such as addition and multiplication, are represented by simple nodes, and complex constructs, such as conditional branches and loops, are represented by compound nodes. By grouping sets of nodes and edges into functions and procedures, the application characterization tool can describe the execution behavior of a given program at various levels.

The *architecture characterization tool* partitions the architecture of a specific type of machine into four categories: computation, data movement and communication, I/O, and control. Each category can be repeatedly partitioned into subsystems until the subsystems in the lowest level can be described by raw timing information. This hierarchical organization of architectural parameters for a specific machine provides a detailed model for determining the operational behavior of each subsystem. The raw timing information of each leaf node of the tree can be user specified or obtained by low-level benchmarking.

The *performance assessment tool* obtains information from the architecture characterization tool and generates timing information for operations on a given machine upon request. Timings for primitive operations are stored within the architecture characterization tool; the performance assessment tool uses these to determine timings for more complicated operations (e.g., complex floating point multiplication).

Two sets of performance parameters for an application, parallelism profiles and execution profiles, are generated by the performance assessment tool. *Parallelism profiles* represent the applications' theoretical upper bounds of performance (e.g., the maximal number of operations that can be parallelized). *Execution profiles* represent the estimated performance of the applications after they have been partitioned and mapped onto one particular machine. Both parallelism and execution profiles are produced by traversing the applications' task-flow graph and then computing and recording each node's performance and statistically based execution time estimates. The objectives of parallelism and execution profiles are very similar to those of task profiling and analytical benchmarking. The *interactive graphical display tool* is the menu-driven user interface for accessing the other tools.

Distributed heterogeneous supercomputing management system. In Ref. [38], a framework called the *distributed heterogeneous supercomputing management system* (*DHSMS*) introduces a systematic methodology for performing both task profiling and analytical benchmarking. The basic approach in DHSMS is to generate a *USC* (universal set of codes) for task profiling. The USC can also be viewed as a standardized set of benchmarking programs used in analytical benchmarking. Because the method of generat-

ing a USC is architecture-driven, these programs can provide information about a machine's hardware features.

The construction of a USC is based on an architecture-dependent hierarchical structure that is a detailed architectural characterization of machines available in an HC system, and is similar to the hardware organization generated by the architectural characterization tool in PAWS. At the highest level of this hierarchical structure, the modes of parallelism for classifying machine architectures are selected. At the second level, finer architectural characteristics, such as the organization of the memory system, can be chosen. This hierarchical structure is organized in such a way that the architectural characteristics at any level are choices for a given category, e.g., type of interconnection network used.

To generate a USC, DHSMS assigns a code-type to each path from the root of the hierarchical structure to a leaf node. Every such path defines a set of architectural features corresponding to the nodes traversed by that path. Mathematically, a USC is defined as a set of code-types $C = \{C_i\}$, where $1 \le i \le K$, and K is the total number of paths from the root to a leaf node. In this proposed framework, conceptually each C_i represents the type of code ideally suited for the architectural features indicated by the ith path. Thus, K is also the number of code-types in C. Let $v_0(j)$ be the size of the parallelism (e.g., maximum possible number of concurrent threads of execution) in the given code block S_j, and let $v_i(j)$ $(1 \le i \le K)$ be a real number between 0 and 1 that indicates how well the code block S_j is matched with the code-type C_i. Then, a *task profiling vector* V_j for a given code block S_j is defined as $V_j = [v_0(j), v_1(j), v_2(j), \ldots, v_K(j)]$. The objective of task profiling in DHSMS is to estimate V_j for each S_j.

The element $v_0(j)$ that quantifies the size of parallelism for code block S_j is very important. Benchmarking results for supercomputers show that the size of parallelism can affect the choice of machines used to achieve the best performance on certain programs [22, 26]. As an example, consider the study in Ref. [33] where the performances of a SIMD machine and a vector machine on SAXPY code (i.e., matrix-vector calculation of the form $S = AX + Y$) are evaluated and compared. Even for a code block that is perfectly matched with the vectorizable code-type, the SIMD machine outperforms the vector machine on vectors with length (the size of the parallelism) longer than the optimal length for the vector machine. Hence, the suggestion in Section 25.5 that the term "task profiling" be used instead of code profiling is very appropriate, because both code and data must be considered.

The task profiling process must be repeated for each application. A fine-grained task profiling, with all levels of architectural features incorporated into the hierarchical structure of machine characteristics mentioned above, will certainly generate a more accurate task profiling vector V_j, but the overhead associated with it increases significantly. Alternatively, a coarse-grained task profiling, which chooses only a few levels of architectural features in the corresponding hierarchical structure, can result in relatively low overhead, but the information obtained from task profiling may not be accurate enough for the subsequent procedures of matching and scheduling. Thus, there is a trade-off between the accuracy of the task profiling and the complexity of the overhead incurred, and this choice can be user-specified [86].

Let $b_q(n)$ be the speedup that machine q can achieve compared to a baseline system by executing optimally matched benchmarking programs with the size of parallelism equal to n. Then, analytical benchmarking can be formally defined as a vector $B(n) = [b_q(n)]$, $q = 1, 2, \ldots, M$, where M is the number of machines available in the HC suite.

$B(n)$ does not evaluate the inter-machine communication overheads of the application program and is categorized as *computation benchmarking* in DHSMS. *I/O benchmarking*

estimates the I/O overhead of a given architecture as a performance metric that is a function of the amount of data being transmitted through the I/O subsystem. *Network-interface profiles* estimate the overhead of the network due to the protocols for communication and media access. Let $d_q(a_m)$ be the destination-independent expected I/O and network-interface overhead of machine q, when there are a_m units of data transmitted through the mth edge of the data dependence graph of the original program. Then, I/O benchmarking and network-interface profiles are defined by the communication overhead vector $D(a_m) = [d_1(a_m), d_2(a_m), \ldots, d_M(a_m)]$. If the exact value of a_m is not known at compile time, some stochastic performance measures are required.

DHSMS begins with a *task-flow graph* (*TFG*), which provides the execution time of each code block S_j on a baseline system and the amount of data transferred between code blocks due to data-dependencies. A task profiling vector V_j is assigned to each code block S_j in the TFG, forming an intermediate *code-flow graph* (*CFG*). The length of V_j and the complexity of task profiling each depend on the number of levels of the hierarchical structure selected by the user. In the final CFG, each code block S_j in the intermediate CFG is associated with an estimated computation time vector $E_j = [e_1, e_2, \ldots, e_M]$, where e_q is the estimated computation time of code block S_j on machine q and is a function of V_j and $B(n)$.

Let $d^*_{p,q}(a_m)$ be the expected I/O and network-interface overhead when there are a_m units of data transmitted between machine p and machine q (this is a function of $d_p(a_m)$ and $d_q(a_m)$). Then, in the final CFG, each communication link m between two code blocks in the original TFG is associated with a communication overhead matrix $D^*(a_m) = [d^*_{p,q}(a_m)]$, $1 \le p, q \le M$. (In Ref. [38], an asterisk is used to distinguish the communication overhead *matrix D* from the communication overhead *vector D*.) The data format conversion overhead also can be added to $d^*_{p,q}(a_m)$. The $M \times M$ matrix $D^*(a_m)$ is assumed to be symmetric along the diagonal. The final CFG can be used in matching and scheduling.

Extensions to the DHSMS approach are presented in Ref. [86]. Two techniques, *augmented task profiling* and *augmented analytical benchmarking*, are proposed to characterize the applications (as well as the machines available in the corresponding HC system) during the construction of the set of code-types. The new augmented approach is a two level framework that combines both fine-grained and coarse-grained characterization techniques. This framework of task profiling and analytical benchmarking is based on generating a representative set of templates (*RST*) that can characterize the execution behavior of the programs at variant levels of details.

Parametric task profiling and parametric analytical benchmarking. In the above two methodologies for performing task profiling and analytical benchmarking, a task profiling vector is defined as a function that maps each combination of the subtasks in the application program and the elements in the set of code-types to a real number in the range [0, 1]. This real number quantifies the degree of the match between the specific subtask and the code-type. Analytical benchmarking is defined as a method of measuring the optimal speedup a certain kind of machine can achieve compared to a baseline system when the best-matched code-type for that machine is executed. By combining the results from the above two characterization steps as discussed in DHSMS, the estimation of the execution times of the subtasks on the available machines in the HC system can be obtained. Most of the selection theories of HC use similar mathematical formulation for task profiling and analytical benchmarking (e.g., see Refs. [21, 59, 80]). Section 25.6.4 presents that mathematical formulation in detail.

The goal of Ref. [87] is to predict the execution time of a task on a single machine. The parametric task profiling and parametric analytical benchmarking proposed in Ref. [87] adopt different mathematical formulations for these two characterization steps in PAWS and DHSMS, but is still compatible with the conceptual model of Section 25.5. First, a set of parameters of cardinality P is defined such that each parameter represents a distinct category of low-level operations performed in a task. This step corresponds to stage 1 of the conceptual model. Let v_i be the operation count for parameter i for a given task. Then, in parametric task profiling, the task profiling of stage 2 is defined as a parametric task profiling vector $V_t = [v_1, v_2, \ldots, v_P]$ for an application task t. The handling of data-dependent loop parameters and conditionals is not included in this formulation.

Let $b_{mi}\ (1 \le i \le P)$ be the execution time of machine m, when that specific kind of machine is used to execute one occurrence of parameter i. Then, in parametric analytical benchmarking, a parametric computation benchmarking vector

$$B^m = [b_{m1}, b_{m2}, \ldots, b_{mP}]$$

is defined.

Let the estimated computational time for a given task on machine m be

$$e_m^{\text{comp}} = \sum_{i=1}^{P} v_i b_{mi}$$

Then, a *computation estimation vector* for a given application task t is

$$E_t^{\text{comp}} = [e_1^{\text{comp}}, e_2^{\text{comp}}, \ldots, e_M^{\text{comp}}]$$

where M is the number of machines available in the HC system. Thus, E_t^{comp} can be used to select a machine for task t.

In Ref. [25], a prototype software system called Automatic Heterogeneous Supercomputing (*AHS*) is introduced. AHS uses a method similar to the V_t and B^m vectors in Ref. [87] to predict execution time. It differs from Ref. [87] in several ways. Data-dependent loop parameters and conditional branch probabilities are approximated by constant values. AHS can use information about the current load on a machine to appropriately weight the expected execution time to account for the load. AHS can estimate the execution time of a specific application program on a group of networked sequential UNIX machines (it is not limited to a single machine). The inter-machine data transfers are handled by asynchronous communication through a UDP (user datagram protocol) socket. AHS can generate the code for inter-machine communication automatically. A proof-of-concept functioning AHS prototype for the MasPar MP-1 and some Unix-based workstations has demonstrated the usefulness of this approach.

25.6.4 A general mathematical formulation for task profiling and analytical benchmarking

One possible general mathematical formulation for task profiling and analytical benchmarking can now be presented. Let CS be a code space spanned by C, where $C = \{C_i\}$ $(1 \le i \le K)$ is a set of code-types generated as dimensions for task profiling and analytical

benchmarking. CS is a K-dimensional space, where K is the number of code-types in C. The contents of C depend on the characteristics of the applications as well as the machine architectures in a given HC system. For example, in DHSMS [38], a USC is generated to be C, where C is a set of code-types for characterizing the architectures of machines in the corresponding HC system. In Refs. [87] and [25], the code-types are individual machine instructions.

Let $S = \{S_j\}$ be a set of computationally homogeneous code blocks (subtasks) generated by decomposing a given application program. After task profiling, for each code block S_j, a K-dimensional vector $\Omega(j) = [\Omega_1(j), \Omega_2(j), \ldots, \Omega_k(j)]$ is generated, where $\Omega_1(j)$ is a real number in the interval [0, 1] that quantifies the degree of match between S_j and the ith dimension of the code space CS.

Let $R = \{m_i\}$ be a set of machines in the HC system. A computation cost-coefficient vector $T = \{t_i\}$ can also be defined, where t_i is the maximal speedup a machine i can achieve compared to a baseline system when it executes the best-matched code-type. The purpose of analytical benchmarking is to estimate t_i as a function of a set of parameters, such as types of operations and length of data vectors.

The amount of communication overhead depends on many factors, such as the bandwidth of the memory channels of the source and destination machines, the topology and bandwidth of the interconnection network, and the complexity of the data format conversion. Let $\delta^*_{r,s}(a)$ represent the expected communication overhead incurred when there are a units of data transmitted from machine r to machine s [48]. Then, a communication cost-coefficient matrix $B^*(a) = [\delta^*_{r,s}(a)]$ is also part of analytical benchmarking. It is possible for $B^*(a)$ to be affected during execution time by network usage of other tasks.

The above formulation is based on the ideas presented in several papers [21, 33, 48, 59, 80]. Methods for automatically determining C, S, Ω, T, and B^* are still largely open problems.

25.6.5 Summary

Example methodologies for the task profiling and analytical benchmarking stage were presented. The information generated by this stage is used in the matching and scheduling stage, discussed in the next section.

25.7 Matching and Scheduling for Mixed-Machine HC Systems

25.7.1 Overview

For mixed-machine HC systems, *matching* involves deciding on which machine(s) each code block should be executed and *scheduling* involves deciding when to execute a code block on the machine to which it was mapped [77]. Matching and scheduling constitute stage 3 of the conceptual model in Section 25.5.

Mapping and scheduling research for general parallel and distributed computing systems, which are closely related to matching and scheduling problems for HC systems, has focused on how to effectively execute multiple subtasks across a network of sequential processors (e.g., see Refs. [6, 20, 60]). In such an environment, load balancing can be an effective way to improve response time and throughput. Although some of these existing mapping and scheduling concepts and techniques can be (and have been) applied to

matching and scheduling for HC systems, there is a fundamental distinction between mapping and scheduling subtasks for a network of sequential processors (e.g., a network of workstations) and matching and scheduling subtasks for an HC system consisting of various types of parallel computers (e.g., MIMD, SIMD, and vector). In the latter case, a subtask may execute most effectively on a particular type of parallel architecture and matching subtasks to machines of the appropriate type is a more important factor than merely balancing the load among all machines in the suite.

This section describes some basic characteristics of matching and scheduling for HC systems and overviews some existing techniques and formulations for matching and scheduling. Although some of the proposed techniques make simplifying assumptions that may be difficult to justify in practice, the body of work reviewed represents solid research that is being conducted as important first steps in a relatively new field.

25.7.2 Characterizing matching and scheduling for HC systems

In HC systems, the total execution time of a task depends on the matching and scheduling techniques used as well as the local mapping and local scheduling employed on each machine in the HC system. *Local mapping* involves the assignment of a code block and its associated data to the processors/memories of a given parallel architecture. Formulating and solving local mapping problems for specific types of parallel architectures is a subject of extensive research within the parallel processing community (e.g., see Ref. [62]). The choice of the local mapping will impact the execution time of a block, which influences matching/scheduling decisions [21, 59]. *Local scheduling* is typically performed by the individual operating system of each machine to decide when to execute multiple jobs that are assigned to run on that machine. Matching/scheduling techniques for HC systems often assume that load information, such as start time and percentage of cycles available, can be obtained from local schedulers [6].

In a broad sense, matching and scheduling problems can be viewed as resource management problems consisting of three main components: consumers, resources, and policy [20]. In the context of HC systems, the *consumers* are represented by the code blocks, which are identified by task profiling. The *resources* include the suite of computers, the network(s) that interconnect these computers, and the I/O devices. The *policy* is the set of rules used by the matcher/scheduler to determine how to allocate resources to consumers based on knowledge of the availability of the resources and the suitability of the available resources for each consumer.

Matching/scheduling policies are generally designed to optimize an objective function subject to a set of constraints. Minimizing the overall execution time under a cost constraint or minimizing cost under a performance constraint are two commonly used formulations for HC systems [21, 32, 80]. Cost can be defined in different ways, including as a weighted sum of execution times for the machines in an existing HC system, or as the total system price (in terms of dollars) for prospective purchases to build an HC suite. Execution time can be estimated through the task profiling and analytical benchmarking techniques discussed in Section 25.6, or from user supplied information or empirical measurements based on typical input data sets [55]. The I/O time and network delay among machines can also be incorporated in the formulation (e.g., Refs. [38, 81]). Once the objective function and constraints are defined, the associated matching/scheduling problem can be solved. In many cases, matching and scheduling problems are NP-complete, thus heuristics and approximation algorithms are often used in practice to obtain solutions (e.g., Ref. [78]).

Matching/scheduling techniques (i.e., policies) can be classified as either static or dynamic. *Static* refers to the case where the decisions of where/when to execute the various code blocks of the given task are made at compile time, and information about the code blocks (e.g., code types and execution time estimates) are available. Either no information on the load of the machines in the HC system is used, or statistically-based models and/or assumptions for these loads may be incorporated. *Dynamic* matching/scheduling decisions are made at execution time, utilizing static and execution time information, such as measured load. Dynamic techniques can either be non-preemptive assignments or can allow dynamic reassignments. They can be adaptive or non-adaptive, depending on whether feedback about the effectiveness of the matching/scheduling policy is used to modify the policy itself.

In the next subsection, some of the existing matching and scheduling techniques and formulations are reviewed. This is not an exhaustive review; however, it demonstrates the range of issues involved.

25.7.3 Examples of techniques and formulations for matching and scheduling for HC systems

Block-based SIMD/SPMD mode selection technique and its extension. In Ref. [82], a *block-based mode selection (BBMS)* technique is proposed that uses static source code analysis of data-parallel program behavior to assign each code block to SIMD mode or SPMD mode in a single mixed-mode machine (e.g., PASM [71]). BBMS is used as a basis for a heuristic for machine selection for SIMD/SPMD mixed-machine systems (i.e., a network of SIMD and MIMD machines) in Ref. [81].

In the mixed-mode BBMS framework developed in Ref. [82], the application program is assumed to be written in a mode-independent language (e.g., Ref. [61]). In a mode-independent language, *operations* represent the most explicit level at which program representation is identical for each mode of parallelism. Task profiling is done by dividing the program into code blocks. *Code blocks* are identified by their leading statements, called *leaders*. The first statement in a program is a leader and any statement that is a target of a branch at the machine code level is a leader, any statement following a conditional branch at the machine code level is a leader [2]. In addition, any statement requiring a synchronization or an inter-PE data transfer and the statement that follows it are leaders.

After the code blocks are defined, the program is transformed into a *flow analysis tree*, whose structure represents the scope levels within the program. The root of the tree represents the scope of the whole program. The non-leaf nodes represent control and data-conditional constructs. Code blocks are represented by leaf nodes of the tree. An example program segment and its associated flow analysis tree are shown in Fig. 25.7.

It is assumed that leaf blocks (i.e., code blocks) are executed either completely in SIMD or completely in SPMD mode, and mode changes are allowed only at inter-block boundaries. Also, the leaf blocks are executed in an ordered sequence (from left to right) as they appear in the flow analysis tree. Thus, the schedule for executing the code blocks is static and is defined by the program itself. If a block is to be executed more than once, such as in a loop, then the mode of parallelism for that block is the same for all loop iterations. Each iteration of a loop body must begin and end execution in the same mode of parallelism (but can change modes within the body). All blocks that are descendants of a data-conditional construct are implemented in the same mode of parallelism (this is to avoid complex execution time bookkeeping overhead, and is independent of BBMS).

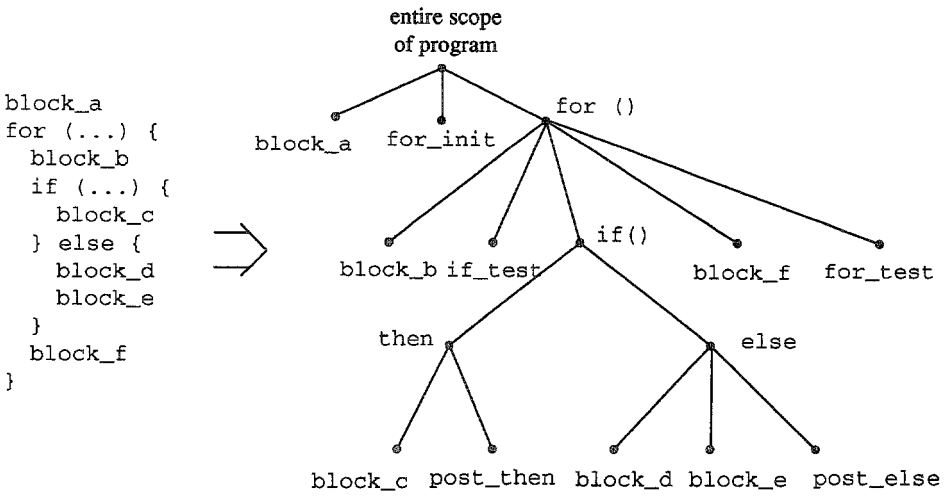


Figure 25.7 Example program segment and its associated flow-analysis tree [82]

Execution time estimates are assumed to be known (e.g., based on the results of analytical benchmarking) for the leaf blocks in both SIMD and SPMD modes, and are denoted by T_l^{SIMD} and T_l^{SPMD} for the lth leaf block, respectively. It is also assumed that the number of iterations for each looping construct and the probability that a PE executes the "then" clause of each data conditional construct are known or estimated at compile time (e.g., through compiler directives). In general, the information associated with sibling nodes at each level of the tree is combined to determine the minimum execution times for starting in each mode and ending in each mode.

Figure 25.8 shows how mode selection for a sequence of sibling code blocks is transformed into a multistage optimization graph. The parameters C^{SIMD} and C^{SPMD} repre-

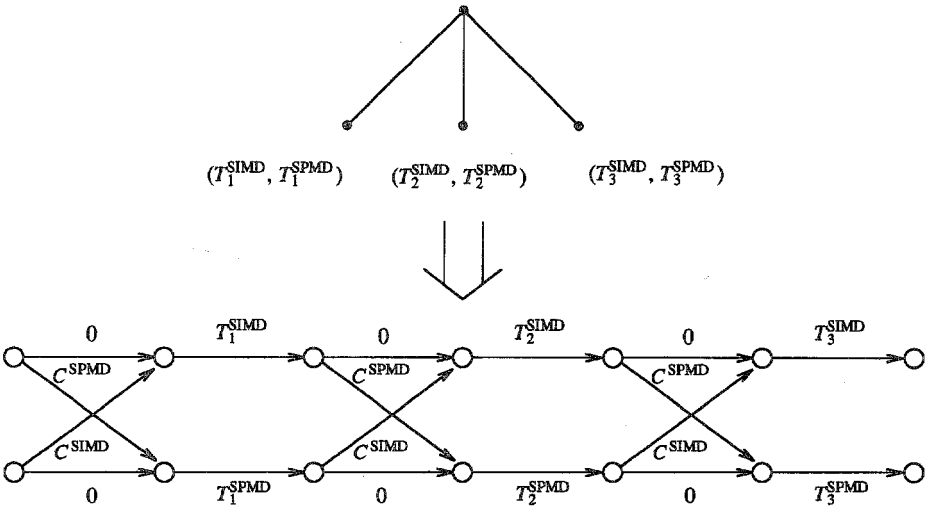


Figure 25.8 Transformation from flow-analysis tree to multistage optimization graph [82]

sent the times for switching to SIMD and SPMD modes, respectively. From the multistage optimization graph, four shortest (in terms of time) paths, corresponding to the four minimum execution times mentioned earlier, are determined. The algorithm for the multistage optimization problem reduces a sequence of three stages to two stages by determining the shortest four paths associated with all possible starting and ending mode choices (starting at the first stage and ending at the third stage). This is repeated until only the initial and final stages remain.

If the parent node is a looping construct, then the known (or approximated) number of iterations is utilized to estimate the total time for the loop. If the parent node is a data conditional construct, then the known (or approximated) probability of executing the "then" clause is used to estimate the total time for the data conditional. The time of the shortest of the four paths at the root is the optimal mixed-mode execution time. The mode assignments corresponding to this path are then made.

Using BBMS as a heuristic for machine selection in a mixed-machine system with two machines is considered in Ref. [81]. The time to switch execution from one machine to the other depends on the time to transfer the required data between machines. Thus, in contrast to the assumed constant time associated with switching modes in a mixed-mode machine, the time of switching execution from one machine to another is dependent on which machine(s) contain the data sets that are required to execute the next block, which depends on the machine choices made for executing the previous blocks, and on the size of the data set to be transferred. A given machine may contain a data set because it was initially loaded there, it was received from another machine, or it was generated by that machine.

Consider a program segment consisting of a sequence of blocks $S_0, S_1, S_2, \ldots$ (subtasks). For each machine, there is an associated execution time that is assumed to be known for each block. The data structures used by each block are assumed to be known and are stored in a *data use* (*DU*) table, denoted by DU_i for block S_i. Each DU_i entry is labeled as read, create, or modify.

Each data structure is assigned a *cost* attribute, which corresponds to the time required to transfer the data structure between the two machines (to simplify the presentation, this cost is assumed to be machine independent). A *location* attribute is used to track the availability of each data structure for each machine. A *data location* (*DL*) table stores these as the cost and location attributes for each data structure. If the data structure is on one machine only, then the cost to transfer the data structure to the other machine is tabulated; if the data structure is located on both machines, then a cost of zero is used. DL_i is used to denote the state of the data location table just before executing block S_i.

Figure 25.9 shows example DU and DL tables for a program segment, where blocks S_0 and S_1 are assigned machine Y, and block S_2 is assigned to machine X (this assignment is arbitrary). T_i^X is the time required to execute block S_i on machine X.

Given the information specified above, the goal is to find an assignment of blocks to machines that results in the minimum overall execution time. In Ref. [81], this problem is transformed into a multistage optimization problem similar to the one used in Ref. [82]. Each time the graph is reduced, a separate DL table is updated for each of the four aggregate paths generated in the reduction step (see Fig. 25.10). Because the time to switch between machines depends on past machine selections, the proposed approach may not always produce optimal assignments. For example, the algorithm may make a machine assignment for a given block that will either require a later block to read a large data structure from the other machine or use a machine that is not well suited for that block. However, simulation studies of program behaviors indicate that the proposed approach, which has a polynomial time complexity, typically produces assignments with overall exe-

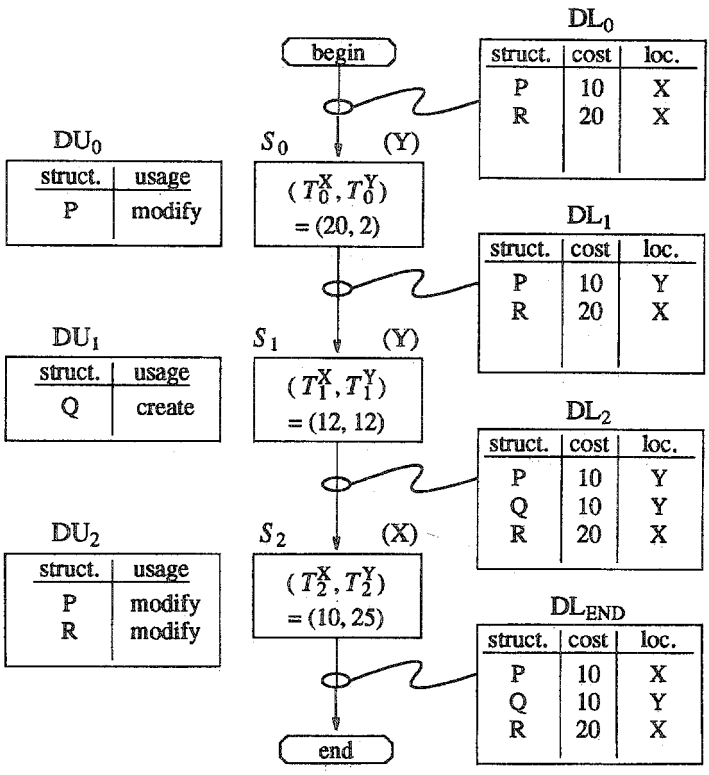


Figure 25.9 Simplified model of parallel program behavior with an arbitrary choice of machine for each code block [81]

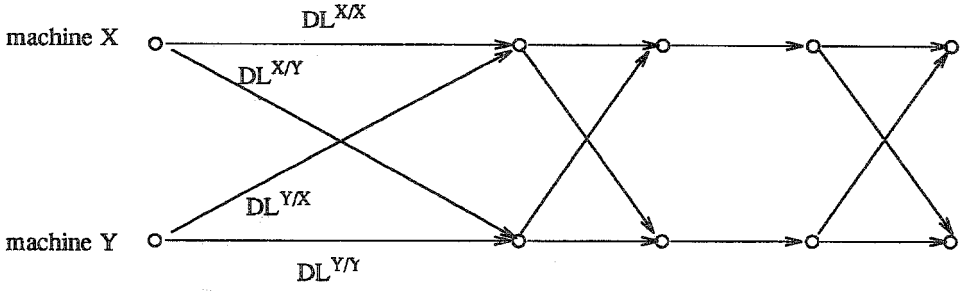


Figure 25.10 Heuristic building on the multistage technique [81]

cution times that are within 1 percent of the optimal assignments, which are determined using an exhaustive search that has an exponential time complexity. This research is currently being extended to more than two machines.

Optimal selection theory and its extensions. A mathematical programming formulation for selecting an optimal heterogeneous configuration of machines for a given set of prob-

lems under a fixed cost constraint, known as *Optimal Selection Theory* (*OST*) [32, 33], is overviewed in this subsection. An extension of OST, called *Augmented Optimal Selection Theory* (*AOST*) [80], is presented (in considerable detail) to illustrate the various components of the mathematical model. Two other extensions of OST, *Heterogeneous Optimal Selection Theory* (*HOST*) [21] and *Generalized Optimal Selection Theory* (*GOST*) [59] are also reviewed.

In the OST framework [32, 33], the application is assumed to consist of a set of non-overlapping *code segments* that are totally ordered in time. Thus, the total execution time of the application is equal to the sum of the execution times of all its code segments. These code segments are identified by task profiling such that each segment is homogeneous in computational requirements. A code segment is defined to be *decomposable* if it can be partitioned into different code blocks that can be executed on different machines of the same type concurrently. A nondecomposable code segment is a code block. The OST formulation assumes, for simplicity, linear speedup when a decomposable code segment is executed on multiple copies of a best matched machine type, and that there are always a sufficient number of machines of each type available. Information about the code blocks and machines is assumed known, as was the case for the methodologies described in Section 25.6. It is noted in Ref. [33] that integer programming techniques can be used with the OST formulation to solve the problem of minimizing the execution time of the application under a fixed dollar cost constraint to purchase the machines that will compose the HC suite, or minimizing the purchase cost under a fixed execution time constraint.

AOST [80] augments OST by incorporating the performance of code segments for all available machine choices (not just the best matched machine type) and by considering non-uniform decompositions of code segments. The issue of considering all available choices of machines is important in practice because the best matched machine may be unavailable. Different machine types are considered, and each machine type may include different models (e.g., the SIMD machine type may include multiple copies of Thinking Machine's CM-2 and/or MasPar's MP-1). Unlike the OST formulation, the number of available machines for each type is limited. For ease of presentation and without loss of generality, the case of having only one model (perhaps multiple copies) for every machine type is considered here.

The optimal speedup $\theta[\tau]$ with respect to a baseline sequential system (e.g., a Sun SPARCstation 5), is assumed to be estimated by analytical benchmarking based on the best matched code type for each machine type τ. For an M-machine suite, for each code segment j, an M-tuple is assumed to be known from task profiling:

$$\omega[j] = (\pi[1,j], \pi[2,j], \ldots \pi[M,j])$$

where $0 \leq \pi[\tau,j] \leq 1$ is an indicator of how well code segment j can be matched with machine type τ. Let S be the set of $|S|$ non-overlapping code segments of the application task. Let μ be the number of different machine types to be considered.

The maximum number of independent code blocks into which code segment j can be decomposed for concurrent execution on machines of type τ is defined as $v[\tau, j]$ and is assumed to be known. Let $\beta[\tau]$ = number of machines of type τ available (or possible to purchase). Thus, the number of code blocks into which code segment j can be decomposed is $\gamma[\tau, j] = \min(v[\tau, j], \beta[t])$. Assume that on the baseline system $p[j]$ = fraction of time spent executing code segment j relative to the overall execution time of S, and $p[j, i]$ = fraction of time spent executing code block i relative to the execution time of code segment j. Thus,

$$\sum_{j=1}^{|S|} p[j] = 1 \quad \text{and} \quad \sum_{i=1}^{\gamma[\tau,j]} p[j,i] = 1, \quad \text{for all } \tau, j$$

The *available parallelism* of a code segment is defined to be the minimum number of processors that results in the optimal execution time with respect to its assumed machine model. Let $\Lambda[\tau, j]$ denote the utilization factor when executing a code segment (or block) j on a machine of type τ. $\Lambda[\tau, j] = 1$ if the available parallelism of code segment j with respect to machine type τ is greater than or equal to the number of processors within machine type τ; otherwise $\Lambda[\tau, j]$ = (available parallelism)/(total number of processors) < 1. Thus, the expected actual speedup of code segment j on machine τ is $q[\tau] \times \pi[\tau,j] \times \Lambda[\tau,j]$. The execution time of a decomposable code segment is the longest execution time among all its code blocks executing on the selected machines. The relative execution time for code segment j on machine type τ, i.e., the time to execute code segment j on a machine of type τ divided by the time to execute the entire task on the baseline machine, is given by

$$\lambda[\tau,j] = \max_{1 \le i \le \gamma[\tau,j]} \left\{ \frac{p[j] \times p[j,i]}{\theta[\tau] \times \pi[\tau,j] \times \Lambda[\tau,i]} \right\}$$

Code segment j is assumed to be executed on machines of type $\tau[j]$, $1 \le \tau[j] \le \mu$, for each $1 \le j \le |S|$. Thus, for a given matching of code segments to machine types (i.e., $\tau[j]$ values), the relative execution time of S is given by:

$$\mathrm{ET}\{\tau[1], \tau[2], \ldots, \tau[|S|]\} = \sum_{j=1}^{|S|} \lambda[\tau[j], j]$$

Given the overall cost constraint, H, and the cost of a machine of type τ, $h[\tau]$, AOST is formulated as:

$$\min_{\substack{1 \le [\tau,j] \le \mu \\ 1 \le j \le |S|}} \mathrm{ET}\{\tau[1], \tau[2], \ldots \tau[|S|]\}$$

subject to

$$\sum_{\tau=1}^{\mu} \left(\max_{1 \le j \le |S|} \gamma[\tau,j] \right) \times h[\tau] \le H$$

HOST [21] extends AOST by incorporating the effects of various local mapping techniques and allowing concurrent execution of mutually independent code segments on different types of machines. The "Hierarchical Cluster-M" model [28] is discussed in Ref. [21] as a way to simplify the matching process by exploiting the hierarchically clustered structure of both the system architecture and the application's communication graph.

In the formulation of HOST, it is assumed that a particular application task is divided into *subtasks*. Subtasks are executed serially. Each subtask may consist of a collection of code segments (as defined earlier) that can be executed concurrently. A code segment consists of homogeneous parallel instructions. Each code segment is further decomposed into several code blocks that can be executed concurrently on machines of the same type. The execution time of a subtask is equal to the longest execution time among all code segments in that subtask. Similarly, the execution time of a code segment is equal to the longest execution time among all code blocks in that segment. The underlying mathematical formulation of HOST is similar to (and a generalization of) that of AOST.

GOST [59] generalizes OST and its extensions to include tasks modeled by general dependency graphs. In GOST, it is assumed that there are ω different machine types and an unlimited number of machines in each type. Different machine models are treated as different types.

In GOST, the most basic code element is a *process*, which corresponds to a block or a nondecomposable code segment (as defined by AOST). It is assumed that an application task consists of several processes modeled by a dependency graph, which could be generated by task profiling. Each node η_i of the graph represents a process and has a number of weights corresponding to the execution times of that process on each machine type for each mapping available on that machine. An edge of the graph represents dependencies between two processes that require communication. Each edge (η_i, η_j) has a number of weights (communication times)—one for each reasonable communication path between each possible pair of host machines for processes η_i and η_j. The weights for nodes and edges are assumed to be derivable from analytical benchmarking. The objective is to determine the optimal matching/scheduling in which each process node in the dependency graph is assigned one machine type and a start time, and the completion time of the whole application is minimized using polynomial time heuristics.

Other formulations and solution techniques. SmartNet is a real-time look-ahead/look-back, near optimal scheduler/planner for HC systems that is being designed and developed at NRaD (a Naval Laboratory) [34]. SmartNet's goal is to minimize the total time to execute a task set, taking into account the times to compute each task i on each machine j, as well as the latency time for any needed data transfers involved in computing task i on machine j. It is assumed that each machine may have previously scheduled tasks either executing or awaiting execution. The objective is not to minimize the compute time for any specific task, but rather to maximize the overall throughput. SmartNet is currently operational and performs the functions discussed above. Various extensions to SmartNet are under development.

In Ref. [78], the HC system is represented by an architecture graph, in which the nodes represent the machines and the edges represent the interconnections among the machines. The application task, which is also modeled with a graph, uses nodes to represent the interacting code blocks and edges to represent data communication dependencies among the code blocks. It is assumed that the bandwidth of each link and the interface overhead between each pair of machines are known. It is also assumed that the computation time of each code block on each machine and the amount of communication required between each pair of code blocks are known. Mapping interacting code blocks of the given application task to machines in the HC system is done by assigning each code block to a machine (i.e., node in the architecture graph). The objective is to minimize the completion time of the whole program. An initial mapping is assumed at the beginning of the search. The basic ac-

tions of the proposed graph-based search are called moves. An example of a move is swapping the current locations of two code blocks. Three types of heuristics are used for attempting to find the optimal mapping. Simulations on randomly generated models are conducted to compare the solution quality and execution times among the three approaches.

Another graph-based method for representing problems for automatically matching code blocks to machines in an HC environment is presented in Ref. [54]. In this work, a "generalized virtual fully-connected architecture graph" is proposed as the machine abstraction and a "Meta Graph" is proposed as the abstraction for the task. In the architecture graph, each node represents a machine in the HC system and contains various machine characteristics. Each edge represents the virtual communication link between every pair of machines, and includes information such as connectivity (i.e., direct versus indirect), connection bandwidth, physical distance, and node-pair heterogeneity (i.e., data-reformatting requirements). In the Meta Graph, the nodes represent code blocks, and edges represent control and data flows between code blocks. Classical list scheduling [65] is augmented to utilize the node-pair heterogeneity representation and is used in simulations on randomly generated problems to match code blocks to machines. Based on several hundred simulations, an average improvement of approximately 70% is obtained from this implementation over the regular weighted graph implementation (i.e., without the node-pair heterogeneity information).

A crossover strategy for assigning tasks on a simple HC system consisting of two machines is proposed in Ref. [56]. It is assumed that the two machines work in a client/server mode. The proposed strategy is used by the client to decide when the speedup of executing a subtask on the server can compensate for the communication/interface overhead involved. When deemed to be beneficial, a remote procedure call is used to execute this subtask on the server. Two experiments were conducted on an actual HC system consisting of a Sun workstation, which functioned as the client, and a Thinking Machines CM-200, which operated as the server. The first experiment was an implementation of the "maximum subvector problem," which involves finding the maximum sum of elements of any contiguous subvector of a given real input vector. The second experiment was based on an implementation of the shallow weather prediction benchmark [76]. The proposed crossover strategy was shown to make the correct choice for executing these applications (i.e., executing entirely on the client or using both the client and the server). In the first application, using both the client and the server was shown to be the proper choice provided that the vector size was larger than a critical value. For the second application, the choice was to always use (only) the client because of high communication requirements.

25.7.4 Summary

Some existing matching and scheduling techniques for stage 3 of the conceptual model were overviewed. More research is needed to integrate all of the stages of the conceptual model into a practical system.

25.8 Conclusions and Future Directions

Although the underlying goal of HC is straightforward—to support computationally intensive applications with diverse computing requirements—there are a great many open problems that need to be solved before HC can be made available to the average applications programmer in a transparent way. Many (possibly even most) need to be addressed just to facilitate near-optimal practical use of mixed-machine HC systems in a "visible" (i.e., user

specified) way. Below is a brief informal discussion of some of these open problems for mixed-machine HC systems; the first few also apply to mixed-mode HC systems. While this list is far from exhaustive, it will convey the types of issues that need to be addressed. Others may be found in Refs. [49, 75].

Implementation of an automatic HC programming environment, such as that envisioned in Section 25.5, will require a great deal of research for devising practical and theoretically sound methodologies for each component of each stage. A general open question that is particularly applicable to stages 1 and 2 of the conceptual model is: "What information should (must) the user provide and what information should (can) be determined automatically?" For example, should the user specify the subtasks within an application or can this be done automatically? Future HC systems will probably not completely automate all of the steps in the conceptual model. A key to the future success of HC hinges on striking a proper balance between the amount of information expected from the user (i.e., effort) and the level of performance delivered by the system.

To program an HC system, it would be best to have one or more machine-independent programming languages that allow the user to augment the code with compiler directives. The programming language and user specified directives should be designed to facilitate (1) the compilation of the program into efficient code for any of the machines in the suite, (2) the decomposition of tasks into homogeneous subtasks, (3) the determination of computational requirements of each subtask, and (4) the use of machine-dependent subroutine libraries.

Along with programming languages, there is a need for debugging and performance tuning tools that can be used across an HC suite of machines. This involves research in the areas of distributed programming environments and visualization tools.

Operating system support for HC is needed. This includes techniques applicable at both the local machine level and at the system-wide network level.

Ideally, information about the current loading and status of the machines in the HC suite and the network that is linking these machines should be incorporated into the matching and scheduling decisions. Many questions arise here: what information to include in the status (e.g., faulty or not, pending tasks), how to measure current loading, how to effectively incorporate current loading and status information into matching and scheduling decisions, how to communicate and structure the loading and status information in the other machines, how often to update this information, and how to estimate task/transfer completion time.

There is much ongoing research in the area of inter-machine data transport. This research includes the hardware support required, the software protocols required, designing the network topology, computing the minimum time path between two machines, and devising rerouting schemes in case of faults or heavy loads. Related to this is the data reformatting problem, involving issues such as data type storage formats and sizes, byte ordering within data types, and machines' network-interface buffer sizes.

Another area of research pertains to methods for dynamic task migration between different parallel machines at execution time. Current research in this area involves how to move an executing task between different machines and determining how and when to use dynamic task migration for load rebalancing or fault tolerance.

Lastly, there are policy issues that require system support. These include what to do with priority tasks, what to do with priority users, what to do with interactive tasks, and security.

Thus, although the uses of existing HC systems demonstrate the significant benefit of HC, the amount of effort currently required to implement an application on an HC system

can be substantial. Future research on the above open problems will improve this situation and allow HC to realize its inherent potential.

Acknowledgments

We thank K. H. Casey, H. G. Dietz, R. F. Freund, A. Ghafoor, A. S. Grimshaw, R. Gupta, E. L. Lusk, R. W. Quong, J. Rosenman, J. M. Siegel, V. S. Sunderam, and J. Yang for their valuable comments. We especially thank J. M. Siegel for her careful reading of the manuscript.

25.9 References

1. Adams III, G. B., and H. J. Siegel. 1982. The extra stage cube: a fault tolerant interconnection network for supersystems. *IEEE Transactions on Computers,* Vol. C-31, No. 5, 443–454.
2. Aho, A., R. Sethi, and J. D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Reading, Mass.: Addison-Wesley.
3. Almasi, G. S., and A. Gottlieb. 1989. *Highly Parallel Computing.* Redwood City, Calif.: Benjamin/Cummings.
4. Armstrong, J. B., M. A. Nichols, H. J. Siegel, and L. H. Jamieson. 1991. Examining the effects of CU/PE overlap and synchronization overhead when using the complete sums approach to image correlation. *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing,* 224–232.
5. Armstrong, J. B., H. J. Siegel, W. E. Cohen, M. Tan, H. G. Dietz, and J. A. B. Fortes. 1994. Dynamic task migration from SPMD to SIMD virtual machines. *Proceedings of the 1994 International Conference on Parallel Processing,* Vol. II, 160–169.
6. Atallah, M. J., C. L. Black, D. C. Marinescu, H. J. Siegel, and T. L Casavant. 1992. Models and algorithms for coscheduling compute-intensive tasks on a network of workstations. *J. Parallel and Distributed Computing,* Vol. 16, No. 4, 319–327.
7. Batcher, K. E. 1968. Sorting networks and their applications. *Proceedings of the AFIPS 1968 Spring Joint Computer Conference,* 307–314.
8. Beguelin, A. L. 1993. Xab: a tool for monitoring PVM programs. *Proceedings of the Workshop on Heterogeneous Processing,* 92–97.
9. Beguelin, A., J. Dongarra, G. A. Geist, R. Manchek, K. Moore, R. Wade, J. Plank, and V. Sunderam. 1992. *HeNCE: A User's Guide, Version 1.2.*Oak Ridge, Tenn.: Oak Ridge National Laboratory, Engineering Physics and Mathematics Division, Dec. 1992, 28 pp.
10. Beguelin, A., J. Dongarra, G. A. Geist, R. Manchek, and V. Sunderam. 1991. *A User's Guide to PVM: Parallel Virtual Machine.* Technical Report ORNLITM-11826. Oak Ridge, Tenn.: Oak Ridge National Laboratory, Engineering Physics and Mathematics Division, 13 pp.
11. Beguelin, A., J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. 1993. Visualization and debugging in a heterogeneous environment. *IEEE Computer,* Vol. 26, No. 6, 88–95.
12. Berg, T. B., S.-D. Kim, and H. J. Siegel. 1991. Limitations imposed on mixed-mode performance of optimized phases due to temporal juxtaposition. *J. Parallel and Distributed Computing,* Vol. 13, No. 2, 154–169.
13. Berg, T. B., and H. J. Siegel. 1991. Instruction execution trade-offs for SIMD vs. MIMD vs. mixed-mode parallelism. *Proceedings of the Fifth International Parallel Processing Symposium,* 301–308.
14. Bergman, L., H-W. Braun, B. Chinoy, A. Kolawa, A. Kuppermann, P. Lyster, C. R. Mechoso, P. Messina, J. Morrison, D. Stanfill, W. St. John, and S. Tenbrink. 1993. *CASA Gigabit Testbed 1993 Annual Report: A Testbed for Distributed Supercomputing.* Technical Report CCSF-33. Pasadena, Calif.: Caltech Concurrent Supercomputing Facilities, California Institute of Technology, 68 pp.
15. Butler, R., W. Gropp, and E. Lusk. 1993. Developing applications for a heterogeneous computing environment. *Proceedings of the Workshop on Heterogeneous Processing,* 77–83.
16. Butler, R. M., A. L. Leveton, and E. L. Lusk. 1993. p4-Linda: a portable implementation of Linda. *Proceedings of the Second International Symposium on High Performance Distributed Computing,* 50–58.
17. Butler, R. M., and E. L. Lusk. User's guide to the p4 parallel programming system. *Technical Report ANL-92117.* Argonne, Ill.: Argonne National Laboratory, Mathematics and Computer Science Division, 34 pp.
18. Butler, R. M., and E. L. Lusk. 1994. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing,* Vol. 20, 547–564.

19. Carriero, N., D. Gelernter, and T. G. Mattson. 1992. Linda in heterogeneous computing environments. *Proceedings of the Workshop on Heterogeneous Processing,* 43–46.
20. Casavant, T. L., and J. G. Kuhl. 1988. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Software Engineering,* Vol. 14, No. 2, 141–154.
21. Chen, S., M. M. Eshaghian, A. Khokhar, and M. E. Shaaban. 1993. A selection theory and methodology for heterogeneous supercomputing. *Proceedings of the Workshop on Heterogeneous Processing,* 3, 15–22.
22. Conte, T. M., and W. W. Hwu. 1991. Benchmark characterization. *IEEE Computer,* Vol. 24, No.l, 48–56.
23. Cray Corp. 1994. The Cray-3/Super Scalable System, preliminary specifications. Cray Computer Corporation, 10 pp.
24. Darema, F., D. A. George, V. A. Norton, and G. F. Pfister. 1988. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing,* Vol. 7, 11–24.
25. Dietz, H. G., W. E. Cohen, and B. K. Grant. 1993. Would you run it here... or there? (AHS: automatic heterogeneous supercomputing). *Proceedings of 1993 International Conference on Parallel Processing,* Vol. II, 217–221.
26. Dongarra, J., J. L. Martin, and J. Worlton. 1987. Computer benchmarking: paths and pitfalls. *IEEE Spectrum,* 38–43.
27. Duclos, P., F. Boeri, M. Auguin, and G. Giraudon. 1988. Image processing on a SIMD/SPMD architecture: OPSILA. *Proceedings of the 9th International Conference on Pattern Recognition,* 14–17.
28. Eshaghian, M. M., and R. F. Freund. 1992. Cluster-M paradigms for high-order heterogeneous procedural specification computing. *Proceedings of the Workshop on Heterogeneous Processing,* 47–49.
29. Parasoft Corporation. 1990. *Express User's Guide Version 3.0.* Pasadena, Calif.: Parasoft Corporation.
30. Fineberg, S. A., T. L. Casavant, and H. J. Siegel. 1991. Experimental analysis of a mixed-mode parallel architecture using bitonic sequence sorting. *J. Parallel and Distributed Computing,* Vol. 11, No. 3, 239–251.
31. Flynn, M. J. 1966. Very high-speed computing systems. *Proceedings of the IEEE,* Vol. 54, No. 12, 1901–1909.
32. Freund, R. F. 1989. Optimal selection theory for superconcurrency. *Proceedings of Supercomputing '89,* 699–703.
33. Freund, R. F. 1991. SuperC or distributed heterogeneous HPC. *Computing Systems in Engineering,* Vol. 2, No. 4, 349–355.
34. Freund, R. F. 1994. The challenges of heterogeneous computing. *Proceedings of the Parallel Systems Fair at the 8th International Parallel Processing Symposium,* 84–91.
35. Freund, R. F., and D. S. Conwell. 1990. Superconcurrency: a form of distributed heterogeneous supercomputing. *Supercomputing Review,* Vol. 3, No. 10, 47–50.
36. Freund, R. F., and H. J. Siegel. 1993. Heterogeneous processing. *IEEE Computer,* Vol. 26, No. 6, 13–17.
37. Geist, G. A., M. T. Heath, B. W. Peyton, and P. H. Worley. 1990. *User's Guide to PICL: A Portable Instrumented Communication Library.* Technical Report ORNLITM-11616. Oak Ridge, Tenn.: Oak Ridge National Laboratory, Engineering Physics and Mathematics Division, 22 pp.
38. Ghafoor, A., and J. Yang. 1993. Distributed heterogeneous supercomputing management system. *IEEE Computer,* Vol. 26, No. 6, 78–86.
39. Giolmas, N., D. W. Watson, D. M. Chelberg, and H. J. Siegel. 1992. A parallel approach to hybrid range image segmentation. *Proceedings of the 6th International Parallel Processing Symposium,* 334–342.
40. Grimshaw, A. S. 1993. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer,* Vol. 26, No. 5, 39–51.
41. Grimshaw, A. S., J. B. Weissman, E. A. West, and E. Loyot. 1994. Meta Systems: an approach combining parallel processing and heterogeneous distributed computing systems. *J. Parallel and Distributed Computing,* Vol. 21, No. 3, 257–270.
42. Haddad, E. 1993. Load distribution optimization in heterogeneous multiple processor systems. *Proceedings of the Workshop on Heterogeneous Processing,* 42–47.
43. Herrarte, V., and E. Lusk. 1991. *Studying Parallel Program Behavior with Upshot.* Technical Report ANL-91115. Argonne, Ill.: Argonne National Laboratory, Mathematics and Computer Science Division.
44. Herter, C. G., T. M. Warschko, W. F. Tichy, and M. Philippsen. 1993. Triton/1: a massively parallel mixed-mode computer designed to support high level languages. *Proceedings of the Workshop on Heterogeneous Processing,* 65–70.
45. Jamieson, L. H. 1987. Characterizing parallel algorithms. In *The Characteristics of Parallel Algorithms,* L. H. Jamieson, D. B. Gannon, and R. J. Douglass, eds. Cambridge, Mass.: MIT Press, 65–100.
46. Kaplan, J. A., and M. L. Nelson. 1993. A comparison of queueing, cluster and distributed computing systems. *NASA Technical Memorandum 109025.* Langley Research Center, Hampton, Va.: National Aeronautics and Space Administration, 47 pp.

47. Karpovich, J. K., M. Judd, W. T. Strayer, and A. S. Grimshaw. 1993. A parallel object-oriented framework for stencil algorithms. *Proceedings of the Second International Symposium on High Performance Distributed Computing,* 34–41.
48. Khokhar, A., V. K. Prasanna, M. Shaaban, and C. L. Wang. 1992. Heterogeneous supercomputing: problems and issues. *Proceedings of Workshop on Heterogeneous Processing,* 3–12.
49. Khokhar, A., V. K. Prasanna, M. Shaaban, and C. L. Wang. 1993. Heterogeneous computing: challenges and opportunities. *IEEE Computer,* Vol. 26, No. 6, 18–27.
50. Kim, S.-D., M. A. Nichols, and H. J. Siegel. 1991. Modeling overlapped operation between the control unit and processing elements in an SIMD machine. *J. Parallel and Distributed Computing,* Vol. 12, No. 4, 329–342.
51. Klietz, A. E., A. V. Malevsky, and K. Chin-Purcell. 1993. A case study in metacomputing: distributed simulations of mixing in turbulent convection. *Proceedings of the Workshop on Heterogeneous Processing,* 101–106.
52. Kogge, P. M. 1994. EXECUBE—a new architecture for scalable MPPs. *Proceedings of 1994 International Conference on Parallel Processing,* Vol. I, 77–84.
53. Lowrie, D. H. 1975. Access and alignment of data in an array processor. IEEE Transactions on Computers, Vol. C-24 (December), 1145–1155.
54. Leangsuksun, C., and J. Potter. 1993. Problem representations for an automatic mapping algorithm on heterogeneous processing environments. *Proceedings of the Workshop on Heterogeneous Processing,* 48–53.
55. Li, Y. A., J. K. Antonio, H. J. Siegel, M. Tan, and D. W. Watson. 1995. Estimating the distribution of execution times for SIMD/SPMD mixed-mode programs. *Proceedings of Heterogeneous Computing Workshop,* 35–46.
56. Lilja, D. J. 1993. Experiments with a task partitioning model for heterogeneous computing. *Proceedings of the Workshop on Heterogeneous Processing,* 29–35.
57. Lipovski, G. J., and M. Malek. 1987. *Parallel Computing: Theory and Comparisons.* New York: John Wiley & Sons.
58. MasPar Computer Corp. 1993. *Data-Parallel Programming Guide.* Sunnyvale, Calif.: MasPar Computer Corporation.
59. Narahari, B., A. Youssef, and H. A. Choi. 1994. Matching and scheduling in a generalized optimal selection theory. *Proceedings of the Heterogeneous Computing Workshop,* 3–8.
60. Ni, L. M., and K. Hwang. 1981. Optimal load balancing strategies for a multiple processor system. *Proceeding of the 1981 International Conference on Parallel Processing,* 352–357.
61. Nichols, M. A., H. J. Siegel, and H. G. Dietz. 1993. Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler. *IEEE Trans. on Parallel and Distributed Systems,* Vol. 4, No. 2, 222–234.
62. Norman, M. G., and P. Thanisch. 1993. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys,* Vol. 25, No. 3, 263–302.
63. Pease, D., A. Ghafoor, I. Ahmad, D. L. Andrews, K. Foudil-Bey, T. E. Karpinski, M. A. Mikki, and M. Zerrouki. 1991. PAWS: a performance evaluation tool for parallel computing systems. *IEEE Computer,* Vol. 24, No. 1, 18–29.
64. Philippsen, M., T. Warschko, W. F. Tichy, and C. Herter. 1993. Project Triton: towards improved programmability of parallel machines. *Proceedings of the 26th Hawaii International Conference on System Sciences,* 192–201.
65. Polychronopoulos, C. D. 1988. *Parallel Programming and Compilers.* Norwell, Mass.: Kluwer Academic Publishers.
66. Rosenman, J., and T. Cullip. 1992. High-performance computing in radiation cancer treatment. *CRC Critical Reviews in Biomedical Engineering,* Vol. 20, Nos. 5–6, 391–402.
67. Saghi, G., H. J. Siegel, and J. L. Gray. 1993. Predicting performance and selecting modes of parallelism: a case study using cyclic reduction on three parallel machines. *J. Parallel and Distributed Computing,* Vol. 19, No. 3, 219–233.
68. Siegel., H. J. 1990. *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* ed. 2. New York: McGraw-Hill.
69. Siegel, H. J., J. B. Armstrong, and D. W. Watson. 1992. Mapping computer-vision-related tasks onto reconfigurable parallel processing systems. *IEEE Computer,* Vol. 25, No. 2, 54–63.
70. Siegel, H. J., T. Schwederski, J. T. Kuehn, and N. J. Davis IV. An Overview of the PASM Parallel Processing System. In *Computer Architecture,* D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, DC, 1987, 387–407.
71. Siegel, H. J., T. Schwederski, W. G. Nation, J. B. Armstrong, L. Wang, J. T. Kuehn, R. Gupta, M. D. Allemang, D. G. Meyer, and D. W. Watson. 1995. Chapter 25, The design and prototyping of the PASM reconfig-

urable parallel processing system. To appear in *Parallel Computing: Paradigms and Applications,* 2nd ed., A. Y. Zomaya, ed. London: International Thomson Computer Press, 78–114.
72. 1990. Special report: gigabit network testbeds. *IEEE Computer,* Vol. 23, No. 9, 77–80.
73. Stevenson, D., and K. Antell, eds. 1993. *VISTAnet Annual Report,* 138 pp.
74. Sunderam, V. S. 1990. PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience,* Vol. 2, No. 4, 315–339.
75. Sunderam, V. S. 1992. Design issues in heterogeneous network computing. *Proceedings of the Workshop on Heterogeneous Processing,* revised edition, 101–112.
76. Swartzrauber, P. N. 1984. *The Shallow Benchmark Weather Prediction Program.* National Center for Atmospheric Research.
77. Tan, M., J. K. Antonio, H. J. Siegel, and Y. A. Li. Scheduling and data relocation for sequentially executed subtasks in a heterogeneous computing system. *Proceedings of the Heterogeneous Computing Workshop,* 109–120.
78. Tao, L., B. Narahari, and Y. C. Zhao. 1993. Heuristics for mapping parallel computations to heterogeneous parallel architectures. *Proceedings of the Workshop on Heterogeneous Processing,* 36–41.
79. Ulrey, R. R., A. A. Maciejewski, and H. J. Siegel. 1994. Parallel algorithms for singular value decomposition. *Proceedings of the 8th International Parallel Processing Symposium,* 524–533.
80. Wang, M., S.-D. Kim, M. A. Nichols, R. F. Freund, H. J. Siegel, and W. G. Nation. 1992. Augmenting the optimal selection theory for superconcurrency. *Proceedings of the Workshop on Heterogeneous Processing,* 13–22.
81. Watson, D. W., J. K. Antonio, H. J. Siegel, and M. J. Atallah. 1994. Static program decomposition among machines in an SIMD/SPMD heterogeneous environment with nonconstant mode switching costs. *Proceedings of the Heterogeneous Computing Workshop,* 58–65.
82. Watson, D. W., H. J. Siegel, J. K. Antonio, M. A. Nichols, and M. J. Atallah. 1994. A block based mode selection model for SIMD/SPMD parallel environments. *J. Parallel and Distributed Computing,* Vol. 21, No. 3, 271–288.
83. Weems, C. C., E. M. Riseman, and A. R. Hanson. 1992. Image understanding architecture: exploiting potential parallelism in machine vision. *IEEE Computer,* Vol. 25, No. 2, 65–68.
84. Weems, C. C., G. E. Weaver, and S. G. Dropsho. 1994. Linguistic support for heterogeneous parallel processing: a survey and an approach. *Proceedings of the Heterogeneous Computing Workshop,* 81–88.
85. Wilson, G. V. 1993. A glossary of parallel terminology. *IEEE Parallel and Distributed Technology,* Vol. 1, No. 1, 52–67.
86. Yang, J., I. Ahmad, and A. Ghafoor. 1993. Estimation of execution times on heterogeneous supercomputer architecture. *Proceedings of the 1993 International Conference on Parallel Processing,* Vol. I, 219–225.
87. Yang, J., A. Khokhar, S. Sheikh, and A. Ghafoor. 1994. Estimating execution time for parallel tasks in heterogeneous processing (HP) environment. *Proceedings of the Heterogeneous Computing Workshop,* 23–28.