12-1-1994

Heterogeneous Computing

Howard Jay Siegel
Purdue University School of Electrical Engineering

John K. Antonio
Purdue University School of Electrical Engineering

Richard C. Metzger
Purdue University School of Electrical Engineering

Min Tan
Purdue University School of Electrical Engineering

Yan Alexander Li
Purdue University School of Electrical Engineering

Follow this and additional works at: http://docs.lib.purdue.edu/ecetr

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
HETEROGENEOUS COMPUTING

HOWARD JAY SIEGEL
JOHN K. ANTONIO
RICHARD C. METZGER
MIN TAN
YAN ALEXANDER LI

TR-EE 94-37
DECEMBER 1994
Heterogeneous Computing

to appear as a chapter in:

Handbook of Parallel and Distributed Computing


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

*Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285, USA
{hj, jantonio, mtan, yali}@ecn.purdue.edu

†Software Engineering Branch (C3CB)
Rome Laboratory
Griffiss AFB, NY 13441-5700, USA
metzgerr@sybil.se.rl.af.mil

December 1994

Purdue University
School of Electrical Engineering
Technical Report TR-EE 94-37
(also Rome Laboratory Technical Report DL-4-0250)

* Supported by Rome Laboratory under contract number F30602-94-C-0022 and by NRaD under contract number N68786-91-D-1799.

† Supported by AFOSR under RL JON 2304F2TK.
HETEROGENEOUS COMPUTING

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

*Purdue University
†Rome Laboratory

1 Overview
2 Mixed-Mode Machines
  2.1 Introduction
  2.2 PASM
  2.3 TRAC
  2.4 OPSILA
  2.5 Triton
  2.6 EXECUBE Chip
  2.7 Conclusions
3 Examples of Uses of Existing HC Systems
  3.1 Simulation of Mixing in Turbulent Convection
  3.2 Interactive Rendering of Multiple Earth Science Data Sets on the CASA Testbed
  3.3 Using VISTAnet to Compute Radiation Treatment Planning for Cancer Patients
4 Examples of Existing Software Tools and Environments
  4.1 Overview
4.2 Linda ............................................................................................................................. 25
4.3 p4 ................................................................................................................................... 26
4.4 Mentat ............................................................................................................................ 27
4.5 PVM. Xab. and HeNCE ................................................................................................. 30
5 A Conceptual Model for Heterogeneous Computing ...................................................... 35
6 Task Profiling and Analytical Benchmarking ..................................................................... 41
  6.1 Overview ....................................................................................................................... 41
  6.2 Definitions of Task Profiling and Analytical Benchmarking .......................................... 41
  6.3 Methodologies for Performing Task Profiling and Analytical Benchmarking ............... 42
  6.4 A Mathematical Formulation for Task Profiling and Analytical Benchmarking ............ 51
  6.5 Summary ...................................................................................................................... 52
7 Matching and Scheduling for HC Systems ....................................................................... 53
  7.1 Overview ....................................................................................................................... 53
  7.2 Characterizing Matching and Scheduling for HC Systems ............................................ 53
  7.3 Examples of Techniques and Formulations for Matching and Scheduling for HC Systems 55
  7.4 Summary ...................................................................................................................... 67
8 Conclusions and Future Directions ................................................................................... 69
Abstract

A heterogeneous computing (HC) system provides a variety of architectural capabilities, orchestrated to perform an application whose subtasks have diverse execution requirements. One type of heterogeneous computing system is a mixed-mode machine, where a single machine can operate in different modes of parallelism. Another is a mixed-machine system, where a suite of different kinds of high-performance machines are interconnected by high-speed links. To exploit such systems, a task must be decomposed into subtasks, where each subtask is computationally homogeneous. 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. Typically, users must specify this; decomposition and assignment. One long-term pursuit in the field of heterogeneous computing is to do this automatically. The field of HC is quite new, having been made possible by recent advances in high-speed inter-machine communication. This report is a brief introduction to HC.
HETEROGENEOUS COMPUTING

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

*Purdue University
†Rome Laboratory

1 Overview

A single application task often requires a variety of different types of computation (e.g., operations on arrays versus operations on scalars). Numerous application tasks that have more than one type of computational characteristic are now being mapped onto high-performance computing systems. 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 with its associated compiler, operating system, and programming tools to satisfy all the computational requirements of various subtasks in certain applications equally well [FrS93]. Thus, a more appropriate approach for high-performance computing is to construct a heterogeneous computing environment.

* Supported by Rome Laboratory under contract number F30602-94-C-0022 and by NRaD under contract number N68786-91-D-1799. 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.
A heterogeneous computing (HC) system provides a variety of architectural capabilities, orchestrated to perform an application whose subtasks have diverse execution requirements. One type of heterogeneous computing system is a mixed-mode machine, where a single machine can operate in different modes of parallelism. Another is a mixed-machine system, where a suite of different kinds of high-performance machines are interconnected by high-speed links. To exploit such systems, a task must be decomposed into subtasks, where each subtask is computationally homogeneous. 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. Typically, users must specify this decomposition and assignment. One long-term pursuit in the field of heterogeneous computing is to do this automatically. The field of HC is quite new, having been made possible by recent advances in high-speed inter-machine communication. This report is a brief introduction to HC.

In the most general case, an HC system includes a heterogeneous suite of machines, high-speed interconnections, interfaces, operating systems, communication protocols, and programming environments [KhP93]. HC is the effective use of these diverse hardware and software components to meet the distinct and varied computational requirements of a given application. Implicit in this concept of HC is the idea that subtasks with different machine architectural requirements are embedded in the applications executed by the HC system. The goal of HC is to decompose a task into computationally homogeneous subtasks, and then assign each subtask to the machine (or mode of parallelism) where it is best suited for execution.

Figure 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 [Fre91]. 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.

![Diagram showing execution examples on different systems](image)

**Figure 1:** A hypothetical example of the advantage of using heterogeneous computing [Fre91], where 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.

Two types of HC systems are mixed-mode machines and mixed-)machine systems [WaA94]. A **mixed-mode machine** 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 [FiC91]. A mixed-machine system is a heterogeneous suite of independent machines of different types interconnected by a high-speed network. 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 report are assumed to have high-speed connections among machines that make decomposition at the subtask level feasible. Another difference is that in mixed-machine systems, the set of subtasks may be executed as an ordered sequence and/or concurrently. Mixed-machine HC has also been referred to as metacomputing [KhP93].

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 report, the future existence of such a language is assumed. More about this topic is in [WeW94], where a collection of parallel programming languages are surveyed and various aspects of programming parallel systems from the perspective of supporting HC are addressed.

In Section 2, examples of mixed-mode machines are given and the mechanism of switching modes for each example is discussed. After Section 2, "HC system" will imply "mixed-machine system," as it is most commonly used in that way. Descriptions of and applications for example existing mixed-machine systems are presented in Section 3. Section 4 provides examples of existing software tools and environments for HC systems. A conceptual model for HC is introduced in Section 5. In this conceptual model, task profiling and analytical benchmarking are two steps necessary for characterizing an application program to automatically decompose it for processing on an HC system. Existing literature that presents explicit frameworks for performing task profiling and analytical benchmarking in the context of HC is overviewed in Section 6. Matching and scheduling are techniques for selecting machines for each subtask based on certain
cost metrics. In Section 7, some basic characteristics of matching and scheduling techniques are described and some existing formulations are reviewed. Finally, open problems in the field of HC are discussed in Section 8.
2 MIXED-MODE MACHINES

2.1 Introduction

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. An SIMD machine typically consists of N processors, N memory modules, an interconnection network, and a control unit [Fly66]. Figure 2(a) shows a distributed memory SIMD architecture in which each processor is paired with a memory module to form N processing elements (PEs). In the SIMD mode of parallelism, 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 on its own distinct data. The operand data for these instructions are fetched from the memory associated with each PE. The interconnection network provides inter-PE communication.

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, i.e., N PEs, an interconnection network, and multiple data streams [Fly66] (see Figure 2(b)). Each PE executes its own program asynchronously with respect to the other PEs. Thus, in contrast to the SIMD model, there are multiple threads of control (i.e., multiple programs). In both of the models in Figure 2, a PE processes data stored locally or received from another PE through the interconnection network. The use of SIMD and MIMD machines is discussed further in [SiW95].

SIMD machines and MIMD machines each have their own advantages when they are used to execute application programs. The advantages of SIMD mode include:

a) The single instruction stream and implicit synchronization of SIMD make programs easier to create, understand, and debug. Also, as opposed to MIMD architectures where common programs can be executed asynchronously, the user does not need to be concerned with the relative timings among the PEs.
Figure 2: (a) Distributed memory SIMD machine model. (b) Distributed memory MIMD machine model.

b) In SIMD mode, the PEs are implicitly synchronized at the instruction level. Explicit synchronization primitives, such as semaphores, may be required in MIMD mode, and generally incur overhead.

c) The implicit synchronization of SIMD mode also allows more efficient inter-PE communication. 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. The receiving PEs implicitly know when to read the message, who sent it, and why
it was sent. MIMD architectures require the overhead of identification protocols and a scheme to signal when a message has been sent and received.

d) Control flow instructions and scalar operations that are common to all PEs (e.g., computing common local subimage data point addresses) can be overlapped (i.e., executed concurrently) on the CU while the processors are executing instructions (this is implementation dependent); this is referred to as CU/PE overlap [ArN91, KiN91].

e) 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.

f) 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:

a) MIMD is very flexible in that different operations may be performed on the different PEs simultaneously (i.e., there are multiple threads of control). Thus, MIMD is effective for a much wider range of algorithms, including tasks that can be parallelized based on functionality (i.e., MIMD can exploit data parallelism and functional parallelism, while SIMD is limited to the former [Jam87]).

b) The multiple instruction streams of MIMD allow for more efficient execution of conditional statements (e.g., "if-then-else") because each PE can independently follow either decision path. In SIMD mode, when conditionals depend on data local to PEs, all of the instructions for the "then" block must be broadcast, followed by all of the "else" block. Only the appropriate PEs are enabled for each block.

c) MIMD's asynchronous nature results in a higher effective execution rate for a sequence of instructions each of whose execution time is data dependent (e.g., floating point operations on some processor architectures). 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 max's" effect: $T_{SIMD} = \sum_{PE} \max_{instrs} \text{ (instr. time)}$. MIMD mode allows each PE to execute the block of instructions independently, resulting in a "max of sum's" effect: $T_{MIMD} = \max_{PEs} \sum_{instrs} \text{ (instr. time)} \leq T_{SIMD}$ (see Figure 3).

d) MIMD machines do not have the added cost of a SIMD CU and the hardware for broadcasting instructions.

Figure 3: "Sum of max's" versus "max of sums" effects.

The trade-offs above are summarized from [BeS91]. The reader is referred to that paper and to [Jam87, SiA92] for more details and examples. Because both SIMD and MIMD modes have advantages, various mixed-mode machines have been proposed.

A mixed-mode machine, which can dynamically switch between the SIMD and MIMD modes of parallelism at instruction-level granularity, allows different modes of parallelism to be applied to execute various subtasks of an application program. Various studies have shown that the mode of parallelism has an impact on the performance of a parallel processing system, and a mixed-mode machine may outperform a single-mode machine with the same number of processors for a given algorithm (e.g., [GiW92, SaS93, UIM94]).
As an example of the use of a mixed-mode machine, consider the bitonic sorting \cite{Bat68} of sequences on the mixed-mode PASM prototype \cite{FiC91}. Assume there are \( L \) numbers and \( N = 2^n \) PEs, where \( L \) is an integer multiple of \( N \), that \( L/N \) numbers are stored in each PE, and that the \( L/N \) numbers within a PE are in sorted order. The goal is to have each PE contain a sorted list of \( L/N \) elements, where each of the elements 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 to accommodate the \( L/N \) sequence in each PE. As shown in Figure 4, an ordered merge is done between the local PE sequence \( X \) and the transferred sequence \( Y \) using local data conditional statements in \texttt{merge}(\( X, Y \)). The lesser half of the merged sequence is assigned the pointer \( X \) and the greater half is assigned the pointer \( Y \). The pointers to the two lists may be swapped by \texttt{swap}(\( X, Y \)), based on a precomputed data-independent mask.

\begin{verbatim}
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 }
      \texttt{merge}(\( X, Y \))
      \texttt{swap}(\( X, Y \))
    }
\end{verbatim}

\textbf{Figure 4}: Bitonic sequence-sorting algorithm \cite{FiC91}.

When choosing the mode of parallelism, the programmer must consider various characteristics of the algorithm. The ordered merge involves many comparisons, all of 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 rest of the operations, including network transfers, 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 potential in SIMD mode for overlapping operations done by the control unit (i.e., loop index variable increment and compare) with operations done by the PEs (i.e., the loop body). It is shown in [FiC91] that there is a noticeable improvement in execution time for the mixed-mode implementation. The mixed-mode results are shown to be the product 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 that allows the most appropriate mode to be selected at each step in the execution of a program. 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) [BeK91].

Very brief descriptions of four existing mixed-mode machines follow, emphasizing the particular mechanisms for implementing mode-switching during the execution of the application program. Readers can refer to the references provided for each system for detailed descriptions of the hardware organization and related issues.

2.2 PASM

PASM is a *Partitionable-SIMD/MIMD* system concept being developed as a design for a large-scale dynamically reconfigurable parallel processing system [SiS87, SiS95]. The PASM design concept is a distributed memory machine and can support at least 1024 PEs in the computational engine. 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 validating design concepts and studying issues related to the use of reconfigurable
parallel processing systems.

As a partitionable mixed-mode system, PASM can be dynamically reconfigured to form submachines of various sizes. Each submachine can independently perform mixed-mode parallelism. PASM uses a flexible multistage interconnection network for inter-PI! communication. Thus, PASM is dynamically reconfigurable along three dimensions: partitionability, mode of parallelism, and connections among PEs. To simplify the discussion, the additional hardware needed for partitioning is ignored, and a single control unit will be assumed.

The mechanism used by PASM to switch modes at instruction-level granularity is as follows. In SIMD mode, a PE fetches SIMD instructions by reading an instruction word from the SIMD instruction space of the PE's memory. This is only a logical address space because SIMD instructions are not physically located in the memory of the PEs. 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 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.

2.3 TRAC

The Texas Reconfigurable Array Computer (TRAC) is a partitionable mixed-mode parallel processing system, which was developed at University of Texas at Austin, in the USA [LiM87]. Its resources can be dynamically reconfigured to fit the structures of the applications. TRAC uses a Banyan interconnection network for inter-processor communication. TRAC 1.1, a shared memory machine, was an experimental prototype of the original paper design of TRAC 1.0. It consisted of four microprocessors that were connected to nine memory modules by an SW-
Banyan network with fan-out of three, spread of two, and two levels (see Figure 5).

--- data tree

------------- instruction tree

...data tree... instruction tree...

Figure 5: A task tree (instruction tree and data tree) of TRAC 1.1 [AlG89].

In TRAC 1.0, after configuring the Banyan network, several data trees connect data memories with their corresponding processors, and an instruction tree connects a specific program memory with processors. As shown in Figure 5, the dashed lines in the network illustrate two data trees, each connecting a processor at the top to a number of data memories at the bottom. The dotted lines illustrate an instruction tree that connects a single program memory to two processors that will work together in SIMD mode. 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.

2.4 OPSILA

OPSILA is a limited mixed-mode parallel machine built at University of Nice, in France [DuB88]. It runs with two different modes of parallelism, SIMD and SPMD. SPMD (single program - multiple data stream) mode is a special form of MIMD mode where all the PEs execute
the same program in an asynchronous fashion, each on its own data [DaG88]. OPSILA is composed of two parts: a central control unit and a computation unit with 16 PEs. Each PE is a processor associated with a memory bank (MB). A synchronous Omega/Benes interconnection network is used for inter-PE communication.

The central control unit consists of two processors: the scalar processor (SP) and the instruction processor (IP). In SIMD mode, the application program is stored entirely in the scalar memory (SM) of the central control unit and managed by the SP. The data are located in the vector memory (VM) of the computation unit. The IP broadcasts SIMD instructions to each PE. The PEs then execute the same instruction simultaneously, each on data from its own MB.

SPMD mode is initialized by the IP, which provides each PE the starting SPMD code address. In SPMD mode, the same program is duplicated in each MB. PEs cannot exchange information during SPMD mode. Data exchanges can only occur in SIMD mode via the synchronous Omega/Benes interconnection network. 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 end of the execution of the PE with the largest work load.

2.5 Triton

Triton is a mixed-mode SIMD/MIMD parallel processing system developed at University of Karlsruhe, in Germany [HeW93, PhW93]. It uses a generalized De Bruijn interconnection network for inter-PE communication. The Triton architecture is scalable up to 4096 nodes. The Triton/1 prototype will consist of 260 nodes (four are for fault tolerance). Each node consists of a processor/memory pair, a memory management unit, a numeric co-processor, a SCSI interface, and a network processor.

In SIMD mode, a single front-end processor produces the instruction stream for all PEs. If a PE is selected not 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 has to 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 two instructions. First, the program counter is set according to the location of the program to be executed in MIMD mode by a branch instruction. Second, the SIMD request bit for each PE is deactivated. Each PE then switches to MIMD mode and starts the execution of the code stored in the local memory. 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 the SIMD mode. Then each PE switches to SIMD mode and the next instruction is from the instruction stream broadcast by the front-end processor.

2.6 EXECUBE Chip

The EXECUBE chip is a building block for parallel processing systems that can support both the SIMD and MIMD modes of parallelism [Kog94]. Its current chip design consists of eight PEs. Each PE is a 16-bit CPU, associated with a 64KB memory module. 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 for accessing the memory system of each PE is through its CPU, the 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, with mode switching instructions included for changing modes between SIMD and MIMD. Those mode switching instructions are machine operation codes that activate special hardware functions. The mode switch from SIMD to MIMD is activated by executing an instruction to ”switch to MIMD mode” and 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 the SIMD execution is started.

2.7 Conclusions

Mixed-mode machines are one extreme form of HC, where two different modes of parallelism are available in one machine. This is in contrast to mixed-machine HC systems, where a suite of machines can provide different modes of parallelism by having each mode in a different machine. Both types of heterogeneous systems can support tasks that include some subtasks that execute faster in SIMD mode and others that execute faster in MIMD mode. Decomposing a task for mixed-mode execution is easier than mixed-machine because the same PEs are used for both modes and, in general, no data has to be moved as a result of a mode change. This eliminates two major problems in the use of mixed-machine HC: moving data among machines and determining machine loads.

The study of the design and use 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 [WaS94], was then extended for use as a heuristic for the mixed-machine case [WaA94].

Thus, mixed-mode machines are important for their advantages over single-mode machines and for their use in developing methodologies that may be adaptable for mixed-machine HC use. The emphasis of this report, however, is on mixed-machine systems. Therefore, for the rest of
the report, "HC system" by itself will imply a mixed-machine suite.
3 EXAMPLES OF USES OF EXISTING HC SYSTEMS

3.1 Simulation of Mixing in Turbulent Convection at the Minnesota Supercomputer Center

In [KIM93], the usefulness of a "metacomputer" developed at the Minnesota Supercomputer Center is demonstrated through a particular application involving the simulation of mixing in turbulent convection in three dimensions. "Metacomputer" is defined in [KIM93] to be a coordinated set of CPUs, I/O devices, mass storage, and graphical capabilities that are appropriately balanced for solving large-scale computational problems, and is equivalent to the term "HC system" defined in Section 1. 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 underlying physics and mathematics that govern the dynamics associated with simulating mixing in turbulent convection are not included here, but are overviewed in [KIM93]. 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. In the following paragraphs, a brief outline of how the required computations were decomposed and assigned to various machines in the system is given.

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 were calculated by solving a set of ordinary differential equations that are dependent on the velocity field solution computed in phase 1. Initially, 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 reported in [KIM93] 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 in the model) 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, the authors note that there is still much work to be done to improve the environment for developing HC applications. The authors state that there is a need for more vendor involvement, in addition to the need for more basic research in the areas of reliability, I/O software, interactivity, and distributed scheduling.

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 [SPR90]. The program established five gigabit testbeds to carry out research in different application areas, each with a different research focus, such as networking protocols, software development, and networking hardware. The research results from this program will contribute to the proposed National Research and Education Network.
(NREN) and ultimately to the National Information Infrastructure (NII). The NREN will link government, industry, and higher-education institutions involved in general research areas that can utilize the interconnected computational resources. In this and the next subsection, two applications that utilize the heterogeneous computing 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. In the future, these sites will be interconnected via SONET (synchronous optical network) connections operating at 2.488 gigabits per second; they are currently connected with lower speed connections [BeB93]. The computational resources of the testbed consists of various parallel and vector machines including an Intel Touchstone Delta, Thinking Machines' CM-5 and CM-200, CRAY Y-MP8/864, Y-MP/264, and Y-MP/232, and a number of workstations and specialized visualization engines.

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 [BeB93], 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. Network Express can be used on MIMD machines, vector machines, and other computers. Under Network Express, each machine is considered a node within the network. One node is chosen as a host, which manages a set of other nodes in the network.

As was done for the "simulation of mixing in turbulent convection" application described in the previous subsection, this application was decomposed based on functionality. Functional modules were identified and optimized for specific machines (and executed on those machines). Thus, when a functional module begins execution, it processes data sets that are completely resident on the machine where the module is executed. 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 (e.g., the CM-5 and CM-200 located at Los Alamos are both connected to a local HiPPI switch). 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 executing the application over this new high-speed HiPPI/SONET network to obtain new benchmark timings that will be compared with those of the current implementation.
3.3 Using VISTAnet to Compute Radiation Treatment Planning for Cancer Patients

VISTAnet is another network in the group of five gigabit testbeds mentioned in the last subsection. The VISTAnet testbed consists of several remote sites including 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) [StA93]. 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 [RoC92]. Recent improvements in the care of cancer patients are due in large part to the effective use of radiation treatment for attacking cancerous cells. Radiation is effective in treating the disease only if it is delivered to the tumorous cells in a high dose while sparing the nontumorous cells. To do this, 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.

Some types of cancer require that the radiation treatment planning take place in three dimensions to achieve maximum effectiveness. This three-dimensional type of planning 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 computational requirements of the application were decomposed in an attempt to take advantage of the strengths of the machines available in the testbed. 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-Planer; 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.
4 EXAMPLES OF EXISTING SOFTWARE TOOLS AND ENVIRONMENTS

4.1 Overview

A variety of software tools and environments have been implemented to assist programmers in developing applications to execute across a heterogeneous suite of computers. 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 and/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 [KaN93].

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). However, as suggested in [CaG92], the tuple space abstraction of Linda makes it an attractive choice for HC systems as well. The tuple space acts to loosely connect processes that 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, which is called the tuple space. Additionally, tuples can be removed, read, and evaluated from the tuple space. There are two types of tuples: process tuples that incorporate executable code and data tuples that are passive, ordered collections of data items [BuL93]. Although the current version of Linda does not support concurrent utilization (i.e., interaction) among machines in an HC system, Linda programs are portable across a range of architecture types. Issues that must be resolved in order to extend the present version of Linda for concurrent use among machines in
an HC system are outlined and discussed in [CaG92].

4.3 p4

p4 is a set of parallel programming tools designed to support portability across a wide range of multicomputer/multiprocessor architectures [BuL92, BuL93, BuL94]. p4 includes high-level operations built on top of low-level system-dependent primitives. These high-level operations allow certain procedure calls for a given system to be replaced with the equivalent p4 calls. The p4 functions are implemented by utilizing the lower-level system-specific set of 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.

The architectures supported by p4 can be divided into three distinct classes. The first class is shared memory multiprocessors (e.g., the Alliant FX/8). In general, the method of communication for shared memory architectures is through the use of the global memory space. Using this method of communication requires that shared data be protected from unsafe concurrent access. p4 provides monitor data types for encapsulating shared data and controlling access. The second class of architectures supported by p4 is the class of distributed memory systems that implement communication through message passing. The members of this class are distributed memory multiprocessor machines and groups of workstations that communicate over a network [BuG93]. The third class of architectures supported by p4 is the class consisting of called "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 then defined by creating a progroup
file, which defines how many programs are to be executed, the names of the programs, and where they are to be executed. The procs group file gives the user the flexibility to experiment with different configurations and types of machines.

In addition to facilitating code portability in an HC environment, p4 also helps the user understand and analyze the behavior of the program's execution. This is accomplished using a utility called ALOG, which creates a log of time-stamped events captured during program execution. ALOG consists of a set of macros that can be used to instrument C or FORTRAN programs. These macros record various events during execution and then dump the associated information to a file on disk (i.e., log) upon program completion or memory exhaustion. This event log can then be used as an input file for a graphical tool called Upshot [HeL91]. 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. In this particular application, p4 was used 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.

4.4 Mentat

Overview

Mentat is an object-oriented parallel processing system designed to provide a layer of abstraction between the user's application and the hardware and system software used to execute the application. Mentat consists of run time support facilities and language abstractions that provide a clear separation between the user and the physical systems [GrW94]. This separation 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 [Gri93]. In this medium-grain dataflow model, programs are characterized as directed graphs. The vertices of the graph 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 facilitates data hiding and other 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 [Gri93].

The Mentat system consists of two major parts. The first is the MPL programming language, which is used to express the high-level abstractions of parallelism within the application. The second is Mentat's run time system (RTS).

MPL

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 is similar to a C structure. The Mentat class instance consists of objects (e.g., local and member variables), their procedures, and a thread of control [GrW94].

In MPL, the standard object-oriented notions of data encapsulation and method encapsulation have been extended to include "parallelism encapsulation" [Gri93]. MPL supports two types of parallelism encapsulation: intraobject parallelism encapsulation, where the implementation (i.e., sequential or parallel) of a member function is hidden from the user, and interobject parallelism encapsulation, where the parallelism among member-function invocations is also hidden from the user. For interobject parallelism encapsulation, 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 [Gri93]. The MPL compiler maps MPL programs onto the dataflow model by translating the MPL programs into C++ programs with embedded calls to the Mentat run 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) [Gri93], 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.

RTS (Run Time System)

The RTS, which initially supported execution on homogeneous parallel machines, has been extended to support HC systems. The RTS supports Mentat's macro-dataflow model via a portable virtual macro-dataflow machine. The virtual macro-dataflow machine provides support routines that perform run time data dependence detection, program graph construction, program graph execution, scheduling, communication, and synchronization [Gri93, GrW94]. The virtual macro-dataflow machine contains two inner components: a set of machine-independent components and libraries, and a set of machine-dependent components. One of the important features of the virtual macro-dataflow machine is that it can be ported to any supported machine in the HC system by changing only the machine-dependent components. This low-level
portability allows the user to port the application source code to any machine in the supported network and have the code execute without source code 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.

4.5 PVM, Xab, and HeNCE

Overview

In this subsection, the PVM (Parallel Virtual Machine) system and two tools that support development of applications using PVM are overviewed. The first of the supporting tools is Xab (X-window Analysis and Debugging), which provides run time monitoring of PVM programs [Beg93]. The second supporting tool is HeNCE (Heterogeneous Network Computing Environment), which provides a high-level PVM-based environment for constructing parallel programs via directed acyclic graphs [BeD93].

PVM

PVM is a software system that enables a collection of heterogeneous computers to be used as a coherent, flexible, and concurrent computational resource [BeD93, Sun90, Sun92]. The PVM package consists of two major parts. The first part includes system level daemons, called pvmds, which reside on each computer in the HC system. The second part is 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. The messages 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. Synchronizations 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 and/or be among processes on different machines.

The second part of the PVM package is a library of interface routines. Applications developed with PVM must be linked with this library. 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
supporting tools described in the next two subsections (Xab and HeNCE) aid the user in monitoring and developing PVM applications based on any of these access modes.

Xab

Xab is a tool developed for the run time monitoring of PVM programs [Bed93, Beg93]. The Xab tool gives the user direct feedback on what PVM functions the program is executing and how the program is performing in a heterogeneous 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. The instrumented calls generate events that can be displayed during program execution. The instrumentation takes place by replacing PVM calls with instrumented Xab calls. 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 event messages from the instrumented PVM calls and formats them into human-readable form. These formatted event messages can be sent either to a file or to the Xab display process. At the display process, formatted messages are received from admon and displayed in an X-window. The window displays each event captured during the execution of the program. The user can single-step through these events or allow Xab to replay the events continuously in real-time.

HeNCE

HeNCE 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 [Bed92, Bed93, Sun92]. In HeNCE, the programmer explicitly specifies the parallelism for an application by drawing a
directed graph, where nodes in the graph represent subtasks written in either FORTRAN or C. The arcs in the graph represent dependencies and flow control. In addition to subtask nodes and dependency arcs, there are 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. Each cost entry is a positive integer; the higher the value the higher the cost of executing a subtask on the associated machine. The meaning of the cost parameters are defined by the user (e.g., estimated execution times or utilization costs in terms of dollars). At run time, HeNCE uses the cost matrix to estimate the most cost effective machine on which to execute each subtask.

Once the graph has been specified and the cost matrix has been defined, the HeNCE tool configures a "virtual machine" using PVM constructs. The machines that make up this virtual machine are a subset of those defined in the cost matrix. After the virtual machine is configured, HeNCE begins execution of the program. Each node in a HeNCE 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). The subtasks execute in three phases. First they obtain those parameter values needed to begin execution. These parameters are obtained from predecessors of each node. If the immediate predecessors do not have all the required parameters for a node, earlier predecessors are checked until all required parameters are found. The second phase is the actual execution of the subtask. Finally, a node finishes execution and passes the needed parameters onto descendant nodes before exiting.

HeNCE can trace the execution of the heterogeneous application. The captured trace information can be displayed in real-time or replayed 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. These visualizations can be used in different ways. They can enable the programmer
to detect bottlenecks in the application by displaying the states of the application components while the application is executing. Alternatively, the trace animation can be used for performance tuning. After viewing the program's behavior, 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.
5 A CONCEPTUAL MODEL FOR HETEROGENEOUS COMPUTING

A conceptual model for the automatic assignment of subtasks to machines in an HC environment is shown in Figure 6. This model builds on the one presented in [FrS93]. In Figure 6, the rectangles contain actions or procedures to be performed as part of the conceptual model. The ellipses show the information used and/or created by action blocks. Figure 6 is referred to as a "conceptual" model because no complete automatic implementation currently exists. As stated earlier, automatic decomposition and assignment is a long-term goal in the field of HC.

In stage 1 of the conceptual model in Figure 6, a set of descriptive parameters is generated that is represented as the general characteristics of 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. Information about the expected types of application tasks to be executed and about the machines that currently exist in the heterogeneous suite are used to generate these parameters. For each parameter, a corresponding computational requirement and a corresponding machine architecture feature 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 floating point operations needed to perform the calculation. The architecture feature of the machines in the heterogeneous suite to be quantified is the speed for these different types of floating point operations.

A particular parameter is included for further consideration in the following stages of this conceptual model only if both the related computational requirements and the architecture features exist. 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 in stage 2. As another example, if there is no vector machine available in the heterogeneous suite, vectorizable code may be excluded from the set of the computational requirements; that needs to be considered.
After stage 1, a collection of corresponding features of the application tasks and machines in the heterogeneous suite can be enumerated. As stated above, these features determine the dimensions of this automatic assignment problem for the given applications and the given HC system. Each of these dimensions represents a specific parameter, which characterizes computational requirements and the related machine capabilities, that needs to be considered in the rest of the stages of this conceptual model. The total number of features enumerated determines the complexity of this automatic assignment problem. An important aspect of the chosen parameters is that they evolve dynamically when new types of applications and/or new types of machines are added.

In stage 2, two characterization steps, task profiling and analytical benchmarking, are used to quantify these corresponding features and transform them into concrete quantitative data. Task profiling is a method used to identify the types of computational requirements that are actually present in a specific application program. The task is decomposed into computationally homogeneous subtasks, and the computational requirements for each subtask are determined. The term often used for this characterization step in the existing literature is code profiling. The reason for using task profiling in this report 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 a procedure that provides a measure of how effectively each of the available machines in the heterogeneous suite performs on each of the types of computations being considered.

Only the computational requirements and the machine capabilities that are included in the collection of corresponding features from stage 1 are identified and evaluated by task profiling and analytical benchmarking. Recall the example above, if no vector machine is available, then task profiling does not need to search for vectorizable code in each application program. If no floating point operations are performed, then it is not necessary for analytical benchmarking to estimate the machine capabilities for those types of operations. Existing literature that presents explicit methodologies for performing task profiling and analytical benchmarking in the context
of HC is reviewed in Section 6 of this report.

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 heterogeneous suite and the inter-machine communication overhead associated with each possible assignment of subtasks to machines. In stage 3, these results and the 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 such items as 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 interesting 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 [Fre89]. Section 7 of this report presents a variety of techniques available in the existing literature for selecting a machine for each subtask based on certain cost metrics.

Stage 4 of this conceptual model is the execution of the given applications on the heterogeneous suite of machines in the HC system. Because the loading of the machines and network in the HC system may change and some faults may occur, sometimes it is necessary to reselect machines for certain subtasks of the application program. Under such circumstances, the current loading and status of the machines and network are updated and stage 3 is reactivated to decide the new assignment of subtasks. Finding techniques for the actual migration of a subtask from one type of machine to another in the middle of execution is a difficult problem; one approach is described in [ArS94].

It is important to note that the mathematical formulation and automation of the intelligent assignment of subtasks to a heterogeneous suite of machines connected by high-speed links
stage 1

generation of parameters that are represented as
general characteristics of computational requirements
and general characteristics of machine capabilities

applications

machines in the
heterogeneous suite

general characteristics of
computational requirements

general characteristics of
machine capabilities

stage 2

task profiling
for a given application

specific characteristics
of each subtask of the application

current loading/status
of machines and network

specific characteristics
of machines and inter-machine
communication overhead

stage 2

analytical benchmarking
for the machines in the
heterogeneous suite

stage 3

matching and scheduling of subtasks

to machines based on cost metric

assignment of subtasks to
machines in the heterogeneous suite

stage 4

execution of the given application on the
heterogeneous suite of machines

Figure 6: Conceptual model of the automatic assignment of subtasks to machines in an HC environment.
are two relatively new fields in HC. Thus, most of the automatic methods that have been proposed for stages 2 and 3 of the conceptual model are frameworks that require further research before they are completely working systems. The task profiling, analytical benchmarking, and matching and scheduling techniques discussed in Sections 6 and 7 of this report are representative frameworks.
6 TASK PROFILING AND ANALYTICAL BENCHMARKING

6.1 Overview

Executing a given task by using an HC system requires identifying and profiling the subtasks in the application code. The basic approach for this, as is described in the literature, is to decompose the overall task 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., [ChE93, FrC90, Fre89, KhP93, Sun92, WaK92]). That is, the concept of a subtask discussed in Section 5 is represented as a homogeneous code block when considering the actual implementation of the applications. These homogeneous code blocks are then assigned to different types of machines to minimize the overall execution time. In general, the goal is to assign each homogeneous code block to the best-matched machine type. 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 how well a code block and machine match with each other even when they do not form the optimal pairing. Also, communication overhead must be considered, as indicated as an input to stage 3 of the conceptual model in Section 5. This section presents example methodologies for task profiling and analytical benchmarking.

6.2 Definitions of Task Profiling and Analytical Benchmarking

Task profiling is a method used to identify the types of computations that are actually present in the application program and quantify how effectively each type can be executed on a particular kind of machine [Fre89]. 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, as discussed in Section 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 [Fre89]. Together, the task profiling and analytical benchmarking steps provide the information needed for the matching and scheduling step, which is described in Section 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 problem domain, 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 [GhY93].

There are a variety of mathematical formulations, collectively called selection theory, that have been proposed to choose the appropriate machine for each code block of the application program. Many of these mathematical formulations (e.g., [ChE93, KhP92, WaK92]) 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.

6.3 Methodologies for Performing Task Profiling and Analytical Benchmarking

Overview

There are only a few papers in the literature that provide specific methodologies for performing task profiling and analytical benchmarking in the context of HC. These papers are the focus of this subsection.

A Comparison between Traditional Benchmarking and Analytical Benchmarking

There are a variety of benchmarking techniques used today for evaluating and comparing the performance of different computers. One of the most widely used methods is to execute a set of
well-studied programs on a machine (e.g., [CoH91, DoM87]), 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. The overall execution time cannot illustrate the true comparative performance of a given machine when it is used for applications suited for an HC environment [Fre89]. Such traditional benchmarking techniques do not reflect the individual contributions of several underlying factors to the performance of a particular kind of machine on a specific code-type. These factors can include the mode of the parallelism, hardware architecture, compiler, operating system, I/O capacity, etc. [GhY93]. 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 of an HC environment when new types of machines and/or new types of applications are added [Fre91]. As introduced in [Fre91], the goal of analytical benchmarking is to construct a class of relatively basic benchmarking programs for each type of computer available in the heterogeneous suite. A set of benchmarking programs can be used to derive the performance metrics of the system for a range of conditions. Thus, each performance metric is a function associated with a set of parameters, 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 [PeG91].
Through the **application characterization tool**, PAWS transforms a given program written in Ada into a graph that illustrates the program's data dependencies. **IF1**, an acyclic graphical language, is used to generate the intermediate graphical form of the program. 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** in PAWS 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 further partitioned into subsystems until the subsystems in the lowest level are fine enough to be enumerated and characterized by raw timing information. **PAWS stores** this hierarchical organization of subsystems in a tree data structure. The raw timing **information** of each leaf node of the tree can be obtained by low-level **benchmarking**. This hierarchical organization of architectural parameters for a specific machine provides a detailed model for determining the operational behavior of each subsystem. This facilitates analytical benchmarking in evaluating the execution time of a particular **kind** of machine when it is used to execute a specific type of code.

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). The user provides the machine performance data for the architecture **characterization** tool and the parameters that define the primitive operations to be used by the performance assessment tool.

Two **sets** of performance parameters for an application, parallelism profiles and execution profiles, are generated by the performance assessment tool using the information provided by the application characterization 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 interactive graphical display tool is the user interface for accessing all the other tools in PAWS. It has been implemented as a hierarchical menu-driven system. The main menu allows the user to select the other three PAWS tools. Windows containing information for each of these three tools can be opened simultaneously.

The terms "task profiling" and "analytical benchmarking" are not used in PAWS. However, the objectives of parallelism and execution profiles are very similar with those of these two characterization steps.

Distributed Heterogeneous Supercomputing Management System

In [GhY93], a framework called the Distributed Heterogeneous Supercomputing Management System (DHSMS) is proposed for managing an HC environment. DHSMS introduces a systematic methodology for performing both task profiling and analytical benchmarking. The basic approach in DHSMS is to generate a Universal Set of Codes (USC) for task profiling. The USC can also be viewed as a standardized set of benchmarking programs used in analytical benchmarking. Because the method of generating USC is architecture-driven, the benchmarking programs based on USC can provide information about the hardware features of machines in an HC system.

The construction of a USC in DHSMS is based on an architecture-dependent hierarchical structure. This hierarchical structure 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 archi-
tectural 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_i\}, where 1 ≤ i ≤ K and K is the total number of paths from the root of the hierarchical structure 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 i-th path of the hierarchical structure. Thus, K is also the number of code-types available in C. A task profiling vector \(V_j\) for a given code block \(S_j\) is defined as \(V_j = [v_0(j), v_1(j), v_2(j), ..., v_K(j)]\). \(v_0(j)\) is the size of the parallelism (e.g., maximum possible number of concurrent threads of execution) in the given code block \(S_j\). \(v_i(j)\) (1 ≤ i ≤ K) is a real number between 0 and 1 that indicates how well the code block \(S_j\) is matched with the code type \(C_i\). The objective of task profiling in DHSMS is to estimate \(V_j\) for each \(S_j\).

There are two points that need to be emphasized in this methodology for performing task profiling. First, in the task profiling vector \(V_j\), 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 [CoH91, DoM87]. As an example, consider the study in [Fre91] 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 vector machine on vectors with length longer than the optimal length for the vector machine. Task profiling must, therefore, consider the size of the parallelism (in the above case, vector length) for each code block with inherent parallelism. Hence, the suggestion in Section 5 that the term "task profiling" be used instead of code profiling is very appropriate, because both code and data must
be considered.

Second, the task profiling process must be repeated for each given 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 [YaG93]. This trade-off is largely dependent on the number of levels of the hierarchical structure being selected in DHSMS, and this choice can be user-specified.

In DHSMS, the proposed USC is not only used as a set of code-types, but can be viewed as a standard set of architecture-dependent benchmarking programs in the following sense. Analytical benchmarking can be formally defined as a vector $B(n) = [b_q(n)]$, $q = 1, 2, ..., M$, where $M$ is the number of machines available in the heterogeneous suite. The variable $b_q(n)$ is 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 $n$. Conceptually, this optimally matched benchmarking program belongs to one of the code-types $C_i$ in USC. Thus, $C_i$ is associated with a benchmarking program that optimally matches the $i$-th path of the machine architectural hierarchical structure.

Because $B(n)$ only estimates the execution time that each machine spends on its best matched code-type, the inter-machine communication overheads of the application program are not evaluated. This kind of benchmarking technique is categorized as computation benchmarking in DHSMS. There are two other kinds of benchmarking techniques in DHSMS, I/O benchmarking and network-interface profiles. 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. Both types of benchmarking techniques are necessary for accurate matching and scheduling in an HC system discussed in stage 3 of the conceptual model.

**I/O benchmarking** and network-interface profiles are defined by a vector of length M, which is called the communication overhead vector $D(a_m) = [d_1(a_m), d_2(a_m), \ldots, d_M(a_m)]$. Each element $d_q(a_m)$ of $D(a_m)$ represents the destination-independent expected I/O and network-interface overhead of machine q, when there are a units of data transmitted through the m-th edge of the data dependence graph of the original program. In reality, the amount of data being transmitted through the network may not be deterministic, in which case some stochastic performance measures are required.

By systematically applying the task profiling and analytical benchmarking techniques described above, DHSMS can generate a code-flow graph (CFG) for the subsequent procedures of matching and scheduling. The process 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. By using the information generated by task profiling, a task profiling vector $V_j$ is assigned to each code block $S_j$ in TFG, forming an intermediate 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 $e_q (1 \leq q \leq M)$ is the estimated computation time of code block $S_j$ on machine q and is a function of $V_j$ and $B(n)$.

In the resulting CFG, each communication link m between two code blocks in the original TFG is associated with a communication overhead matrix $D^*(a_m) = \{d^*_{p,q}(a_m)\}, 1 \leq p, q \leq M$ (in [GhY93], an asterisk is used to distinguish the communication overhead matrix D from the communication overhead vector $D$). The element $d^*_{p,q}(a_m)$ represents the expected I/O and network-interface overhead, when there are a units of data transmitted between machine p and
machine q. The data format conversion overhead also can be added to $d^*_{p,q}(a_m)$. The $M \times M$ matrix $D'(a)$ is assumed to be symmetric along the diagonal. Each element $d^*_{p,q}(a_m)$ is a function of both $d_p(a_m)$ and $d_q(a_m)$, where $1 \leq p, q \leq M$, and $p \neq q$. The resulting CFG contains detailed information about machine-dependent execution time, I/O performance, and the inter-machine communication overhead associated with each code block in the TFG. The final CFG, can be used in matching and scheduling.

The USC introduced in DHSMS is machine-dependent (i.e., depends on the characteristics of the machines in the HC system), but is not application-dependent because there is no characterization of the given applications involved during the construction of the USC. However, the efficient management of an HC system requires a detailed analysis of both the architectures of the machines and the structures of the applications. In [YaG93], two techniques called augmented task profiling and augmented analytical benchmarking, are proposed to characterize the applications as well as the machines available in the corresponding HC system. 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 adopt the above mathematical formulation for task profiling and analytical benchmarking (e.g., [ChE93, WaK92, NaY94]). Subsection 6.4 presents that mathematical formulation in detail.

The parametric task profiling and parametric analytical benchmarking proposed in [YaK94] adopt different mathematical formulations for these two characterization steps. The goal of [YaK94] is to predict the execution of a task on a single machine. At first, a set of parameters 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 for HC presented in Section 5. Then formally, in parametric task profiling, the computational 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 size of \( V_t \) is \( P \), where \( P \) is the cardinality of the parameter (operation) set. Each \( v_i \) (for \( 1 \leq i \leq P \)) of \( V_t \) represents the operation count for parameter \( i \). The handling of data-dependent loop parameters and conditionals is not included in this formulation.

In parametric analytical benchmarking, a parametric computation benchmarking vector \( B^m = [b_{m1}, b_{m2}, \ldots, b_{mp}] \) is also defined, where \( P \) is the cardinality of the parameter set also. Each \( b_{mi} \) (for \( 1 \leq i \leq P \)) represents the execution time of machine \( m \), when that specific kind of machine is used to execute one occurrence of parameter \( i \).

A computation estimation vector for a given application task \( t \) is defined as \( E_t^{\text{comp}} = [e_1^{\text{comp}}, e_2^{\text{comp}}, \ldots, e_M^{\text{comp}}] \), where \( M \) is the number of machines available in the HC system. The element \( e_m^{\text{comp}} \) (for \( 1 \leq m \leq M \)) represents the estimated computational time of task \( t \) on machine \( m \), where

\[
e_m^{\text{comp}} = \sum_{i=0}^{P} v_i b_{mi}. \quad v_i \text{ and } b_{mi} \text{ are obtained from parametric task profiling and parametric analytical benchmarking, respectively.}
\]

Although parametric task profiling and parametric analytical benchmarking adopt a mathematical formulation that is different from the one presented in Subsection 6.4, this methodology for performing these two characterization steps is still compatible with the conceptual model presented in Figure 6. Parametric task profiling is defined as a procedure to estimate the
computational requirement of the application task and parametric analytical benchmarking is defined as a method to evaluate the machine capability of the specific HC system as discussed in Section 5.

In [DiC93], a prototype software system called Automatic Heterogeneous Supercomputing (AHS) is introduced. AHS uses a method similar to the $V_l$ and $B^m$ vectors in [YaK94] to predict execution time. It differs from [YaK94] 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. AHS can estimate the execution time of a specific application program on a group of networked sequential UNIX machines. The inter-machine data transfers are handled by asynchronous communication through a UDP socket. AHS can generate the code for inter-machine communication automatically. A proof-of-concept functioning AHS prototype has determined the usefulness of this approach.

6.4 A Mathematical Formulation for Task Profiling and Analytical Benchmarking

A mathematical formulation for task profiling and analytical benchmarking can now be presented in unambiguous terms. Let CS be a code space spanned by C, where $C = \{C_i\}$ $(1 \leq i \leq K)$ 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 [GhY93], 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. As an example, in [YaK94] and [DiC93], the code types are individual machine instructions.

Let $S = \{S_j\}$ be a set of computationally homogeneous code blocks generated by decomposing a given application program. After task profiling, for each code block $S_j$, a K-dimensional vector $\Omega(j) = [\Omega_1(j), \Omega_2(j), ..., \Omega_K(j)]$ is generated, where $\Omega_i(j)$ is a real number in the interval $[0, 1]$ that quantifies the degree of match between $S_j$ and the i-th dimension of the
Let \( R = \{ m_k \} \) be a set of machines in the HC system. A computation cost-coefficient vector \( T = \{ t_k \} \) can also be defined, where \( t_k \) is the maximal speedup a machine \( k \) can achieve compared to a baseline system when it executes the best matched code-type. The purpose of analytical benchmarking is to estimate \( t_k \) 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. A communication cost-coefficient matrix \( B^* (a) = \{ \delta^*_{r,s}(a) \} \), where the variable \( \delta^*_{r,s}(a) \) represents the expected communication overhead incurred when there are \( a \) units of data transmitted from machine \( r \) to machine \( s \) [KhP92], is also part of analytical benchmarking. It is possible for \( B^* (a) \) to be impacted during execution time due to network usage by other tasks.

The above formulation is based on the ideas presented in several papers [ChE93, Fre89, KhP92, NaY94, WaK92]. Methods for automatically determining \( C, S, \Omega, T, \) and \( B^* \) are still largely open problems.

### 6.5 Summary

Definitions and example methodologies for performing task profiling and analytical benchmarking were presented in this section. Also, a mathematical formulation for these two characterization steps was given. As mentioned earlier, these formulations make many simplifying operating assumptions. Further research is needed before these formulations are practical tools that can provide the quantitative results needed in subsequent matching and scheduling techniques, examples of which are presented in the next section.
7 MATCHING AND SCHEDULING FOR HC SYSTEMS

7.1 Overview

For 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. Mapping and scheduling problems for parallel and distributed computing systems, which are closely related to matching and scheduling problems for HC systems, have been studied extensively in the past. Much of the work in mapping and scheduling for parallel and distributed systems has focused on how to effectively execute multiple subtasks across a network of sequential processors (e.g., see [AtB92, CaK88, NiH81]). 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, the subtasks can be characterized based on "type of parallelism" present in each subtask to account for the fact that certain types of subtasks may execute most effectively on a particular type of parallel architecture. In general, matching subtasks to machines of the appropriate type(s) 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.

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 onto the processors/memories of a given parallel architecture. Formulating and solving local map-
ping problems for specific types of parallel architectures is a subject of extensive research within
the parallel processing community ([NoT93] is a recent thorough review on this subject). The
choice of the local mapping will impact the execution time of a block, which influences
matching/scheduling decisions [ChE93, NaY94]. Local scheduling is typically performed by the
individual operating system of each machine in the HC system 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 [AtB92].

In a broad sense, matching and scheduling problems can be viewed as resource management problems consisting of three main components: consumers, resources, and policy [CaK88]. 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 [ChE93, Fre89, WaK92]. Cost can be defined in different ways, including as a weighted sum of execution times for each machine in an existing HC system, or as the total system price (in terms of dollars) for prospective purchases. Execution time can be estimated through the analytical benchmarking and task profiling techniques discussed in Section 6, or from empirical measurements based on typical input data sets. The I/O time and network delay among machines can also be incorporated in the formulation, e.g., see [GhY93, WaA94]. 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., [TaN93]).

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 run time, utilizing static information as well as information available only at run time, 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 on 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 for HC systems are reviewed. This is not a complete review of research done in the area; it is presented to demonstrate the range of issues involved and overview some of the approaches proposed for solving matching and scheduling problems for HC systems.

7.3 Examples of Techniques and Formulations for Matching and Scheduling for HC Systems

Block-Based SIMDISPMD Mode Selection Technique and its Extension

An SIMD/SPMD environment, such as a single mixed-mode machine (e.g., P'ASM [SiS95]) or an SIMDISPMD mixed-machine system (i.e., a network of SIMD and MIMD machines), represents a special class of HC systems. In [WaS94], 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 mixed-mode machine. BBMS is used as a basis for a heuristic for machine selection for SIMDISPMD mixed-machine systems in [WaA94]. In the remainder of this subsection, the application of BBMS for mixed-mode machines is overviewed first, followed by its extension to mixed-machine systems.
In the framework developed in [WaS94], the application program is assumed to be written in a mode-independent language. In a mode-independent language, operations represent the most explicit level at which program representation is identical for each mode of parallelism. Mode-independent languages make it possible to utilize the most appropriate parallel execution mode (machine) for each block of a given program.

In the BBMS framework, 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, 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, and, 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 Figure 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 part of (i.e., descendants of) a data-conditional construct are implemented in the same mode of parallelism.

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 \( T^{SIMD}_i \) and \( T^{SPMD}_i \) for the \( i \)-th leaf block. It is also assumed that the number of iterations for each loop-
Figure 7: Example program segment and its associated flow-analysis tree [WaS94].

The program segment is:

```c
block-a
for (...) {
    block_b
    if (...) {
        block-c
    } else {
        block-d
        block-e
    }
    block-f
}
```

The flow-analysis tree is shown in the diagram. The entire scope of the program is represented at the top, followed by the for loop `for ()` and the nested blocks and conditional statements.

The information associated with sibling nodes at each level of the tree is combined to determine the minimum execution times for starting and ending in SIMD, starting and ending in SPMD, starting in SIMD and ending in SPMD, and starting in SPMD and ending in SIMD. These four times are determined by using a multistage optimization algorithm. Traversing the flow analysis tree using a depth first traversal, the deepest levels of the tree are combined first, and higher levels are combined until only the root node remains. Then the parallel mode for each segment of the program is assigned.

Figure 8 shows how the problem of selecting the best modes of execution for a sequence of sibling code blocks is transformed into a multistage optimization graph. The parameters $C_{SIMD}$ and $C_{SPMD}$ represent 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.

Figure 8: Transformation from flow-analysis tree to multistage optimization graph [WaS94].

If the parent node is a looping construct, then the (assumed) information for the number of iterations is utilized to estimate the total time for the loop. If the parent node is a data conditional construct, then the (assumed) information for the 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. For more details, refer to [WaS94].

Optimal machine selection in a mixed-machine system consisting of two machines is considered in [WaA94]. The time to switch execution from one machine to the other is assumed to depend 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 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 \((S_0, S_1, S_2, \cdots)\), where each block is to be executed on one of the two machines. For each machine, there is an associated execution time that is assumed to be known for each block. It is assumed that a collection of data structures are used to execute the sequence of blocks and for each block, a subset of these data structures is used. The data structure requirements for each block are assumed to be known and are stored in a data use (DU) table denoted by \(DU_i\) for block \(S_i\). For each data structure listed in a DU table, one of three usage types is tabulated: 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 (for clarity of presentation, this cost is assumed to be independent of the source of the transfer). 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. The value of the tabulated cost attribute depends on the location(s) of the 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 9 shows example DU and DL tables for a program segment consisting of three blocks. In the figure, blocks \(S_0\) and \(S_1\) are assigned to
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 $X$.

<table>
<thead>
<tr>
<th>Block</th>
<th>Machine X</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1^X$</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>$T_2^X$</td>
<td>10</td>
<td>25</td>
</tr>
</tbody>
</table>

**Figure 9:** Simplified model of parallel program behavior with an arbitrary choice of machine for each code block [WaA94].

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 [WaA94], this problem is transformed into a multistage optimization problem similar to the one used in [WaS94]. Each time the graph is reduced, a separate DL table is kept for each of the four aggregate paths gen-
erated in the reduction step (see Figure 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 execution times that are less than 1% more than 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.

![Heuristic building on the multistage technique](image)

**Figure 10:** Heuristic building on the multistage technique [WaA94].

**Optimal Selection Theory and its Extensions**

A mathematical programming formulation for selecting an optimal heterogeneous configuration of machines for a given set of problems under a fixed cost constraint, known as Optimal Selection Theory (OST) [Fre89, Fre91], is overviewed in this subsection. An extension of OST, called
**Augmented** Optimal Selection Theory (AOST) [WaK92], is presented (in considerable detail) to illustrate the various components of the mathematical model. Two other extensions of OST, Heterogeneous Optimal Selection Theory (HOST) [ChE93] and Generalized Optimal Selection Theory (GOST) [NaY94] are also reviewed.

In the OST framework, 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 there are always a sufficient number of machines of each type available. Various *information* about the code blocks and machines is assumed known, as was the case for the methodologies described in Section 6. It is noted in [Fre91] 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 cost under a fixed execution time constraint. The solution from the OST framework shows the existence of an optimal suite of heterogeneous supercomputers for a given problem set under a fixed cost constraint.

AOST 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.

In the formulation of AOST, five machine types are considered: vector, SIMD, MIMD, scalar, and special purpose. 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-
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 details of dealing with more than one model per machine type are described in [WaK92].

The optimal speedup $\theta[\tau]$ with respect to a baseline sequential system (e.g., a VAX machine), is assumed to be estimated by analytical benchmarking based on the best matched code type for each machine type $\tau$. For each code segment $j$, a five-tuple is assumed to be known from task profiling: $\omega[j] = (\pi[vector, j], \pi[SIMD, j], \pi[MIMD, j], \pi[scalar, j], \pi[special, j])$, where $0 \leq \pi[\tau, j] \leq 1$ is an indicator of how well code segment $j$ can be matched with machine type $\tau$. Let $S$ be the set of non-overlapping code segments of the application task. Let $\mu$ 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 $\tau$ is defined as $v[\tau,j]$, and is assumed to be known. Let $\beta[\tau]$ = number of machines of type $\tau$ available (or possible to purchase). Therefore, the actual number of code blocks into which code segment $j$ can be decomposed is defined by $\gamma[\tau, j] = \min(v[\tau, j], \beta[\tau])$. Assume on the baseline system, $p[j] = \frac{\text{fraction of time spent executing code segment } j \text{ relative to the overall execution time of } S}$, and $p[j, i] = \frac{\text{fraction of time spent executing code block } i \text{ relative to the execution time of code segment } j}{\text{of code segment } j}$, thus $p[j, i] = 1$, 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 running a code segment (or block) $j$ on a machine of type $\tau$. $\Lambda[\tau, j] = 1$ if the available parallelism of code segment $j$ with respect to machine type $\tau$ is not less than the number of processors within machine type $\tau$; otherwise $\Lambda[\tau, j] = \frac{\text{available parallelism}}{\text{total number of processors}}$. Thus, the expected actual speedup of code segment $j$ on machine $\tau$ is $\theta[\tau] \times \pi[\tau, j] \times \Lambda[\tau, j]$. 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 $\tau$ is given by:

$$\lambda[\tau,j] = \max_{1 \leq i \leq |\tau[j]|} \{ p[j] \times p[j,i] / (\theta[\tau] \times \pi[j] \times \Lambda[\tau,i]) \}.$$ 

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

$$ET[\tau[1], \tau[2], ..., \tau[|\sigma|]] = \sum_{j=1}^{\sigma} \lambda[\tau[j], j].$$

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

$$\min_{1 \leq \tau[j] \leq \mu, 1 \leq j \leq |\sigma|} \{ ET[\tau[1], \tau[2], ..., \tau[|\sigma|]] \}$$

subject to $\sum_{\tau=1}^{\mu} (\max_{1 \leq j \leq |\sigma|} \gamma[\tau,j]) \times h[\tau] \leq H$.

HOST 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 [EsF92] is discussed in [ChE93] 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 \textbf{HOST} is similar to (and a natural generalization of) that of AOST.

\textbf{GOST} generalizes OST and its extensions to include tasks modeled by general dependency graphs. In \textbf{GOST}, it is assumed that there are \(\omega\) different machine types and an unlimited number of machines in each type. Different machine models are treated as different types.

In \textbf{GOST}, the most basic code element is a process, which corresponds to a block or a non-decomposable 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 \(\eta_i\) 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 \((\eta_i, \eta_j)\) has a number of weights (communication times), one for each reasonable communication path between each possible pair of host machines for processes \(\eta_i\) and \(\eta_j\). 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 algorithms.

\textbf{Other Formulations and Solution Techniques}

In [Tan93], the problem of mapping interacting code blocks of a given application task to machines in an HC system is studied. The HC system is represented by an architecture graph, in which the nodes represent the machines and the edges represent the \textbf{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 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 actions 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.

In [LeP93], another graph-based method for representing problems for automatically matching code blocks to machines in an HC environment is presented. 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 [Po188] 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).

In [Li93], a crossover strategy for assigning tasks on a simple HC system consisting of two machines is proposed. 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 running 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 [Swa84]. 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.

7.4 Summary

Some existing matching and scheduling techniques for HC systems were overviewed in this section. All of these frameworks, which are applicable to stage 3 of the conceptual model of Section 5, assume that information from stage 2 of the conceptual model is available and given. 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. More research is needed to integrate all of the stages of the conceptual model into a practical system. Specific research challenges for HC are discussed in the next section.
**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 heterogeneous computing 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 real heterogeneous suites in a "visible" (i.e., user specified) way. Below is a brief informal discussion of some of these open problems; it is far from exhaustive, but it will convey the types of issues that need to be addressed. Others may be found in [KhP93, Sun92].

Implementation of an automatic HC programming environment, such as envisioned in Section 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 (a) the compilation of the program into efficient code for any of the machines in the suite, (b) the decomposition of tasks into homogeneous subtasks, and (c) 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 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. This could be used to rebalance loads or if a fault occurs. 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 balancing.

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.

In conclusion, there is clearly a gap between the state-of-the-art in practical HC computing (briefly illustrated in Sections 2 through 4) and automating all of the steps characterized by the conceptual model of Section 5 (and discussed in Sections 5 through 7). In particular, stages 1 through 3 of the conceptual model are typically done entirely by the user, while some aid is provided for the user for stage 4 by existing tools and environments. Thus, although the uses of existing HC systems demonstrate the significant potential 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 make HC more viable.

References


47 pp.


