HETEROGENEOUS DISTRIBUTED COMPUTING

One of the biggest challenges in high-performance computing is that as machine architectures become more advanced to obtain increased peak performance, only a small fraction of this performance is achieved on many real application sets because a typical application may have various subtasks with different architectural requirements. When such an application is executed on a given machine, the machine spends most of its time executing subtasks for which it is unsuited. With the recent advances in high-speed digital communications, it has become possible to use collections of different high-performance machines in concert to solve computationally intensive application tasks. This article describes the issues involved in using such a *heterogeneous computing* (HC) suite of machines to solve application tasks.

A hypothetical example of an application that has various subtasks that are best suited for different machine architectures is shown in Fig. 1 (based on Ref. 1). The example executes for 100 time units on a baseline serial machine. The application consists of four subtasks: the first is best suited for execution on a single instruction stream, multiple data streams (SIMD) parallel machine, the second is best suited for a distributed-memory multiple instruction streams, multiple data streams (MIMD) parallel machine, the third is best

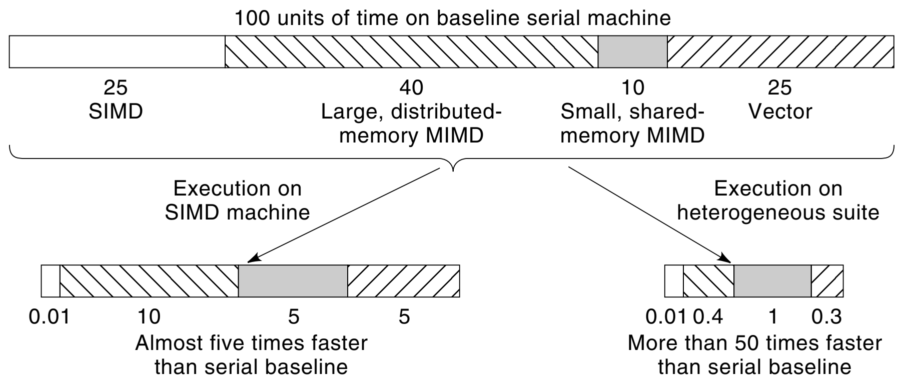

Figure 1. Hypothetical example of the advantage of using a heterogeneous suite of machines, where the heterogeneous suite time includes intermachine communication overhead (based on Ref. 1). Not drawn to scale.

suited for a shared-memory MIMD machine, and the fourth is best suited for execution on a vector (pipelined) machine.

Executing the whole application on an SIMD machine may improve the execution time of the SIMD subtask from 25 to 0.01 time units and the other subtasks to varying extents. The overall improvement in execution time may only be about a factor of 5 because other subtasks may not be well suited for an SIMD machine. Using four different machines that match the computational requirements for each of the individual subtasks can result in an overall execution time that is better than the baseline serial execution time by more than a factor of 50. If the subtasks depend on any shared data, then intermachine data transfers need to be performed when multiple machines are used. Hence, data transfer overhead has to be considered as part of the overall execution time on the HC suite. For example, in Fig. 1 the time for executing on the vector machine must include any time needed to get data from the other machines.

There are many types of HC systems. This article focuses on *mixed-machine HC* systems (2), where a heterogeneous suite of independent machines is interconnected by high-speed links to function as a *metacomputer* (3). *Mixed-mode HC* refers to a single parallel processing system, whose processors are capable of executing in either the synchronous SIMD or asynchronous MIMD mode of parallelism, and can switch between modes at the instructional level with negligible overhead (4). PASM, TRAC, OPSILA, Triton, and EXECUBE are examples of mixed-mode HC systems that have been prototyped (4).

One way to exploit a mixed-machine HC environment is to decompose an application task into subtasks, where each subtask is computationally well suited to single-machine architecture, but different subtasks may have different computational needs. The subtask may have data dependencies among them. Once the subtasks are obtained, each subtask is assigned to a machine (*matching*). Then the subtasks and intermachine data transfers are ordered (*scheduling*). It is well known that finding a matching and scheduling (*mapping*) that minimizes the overall completion time of the application is generally, NP-complete (5). Currently, programmers must manually specify the task decomposition and the assignment of subtasks to machines. One long-term pursuit in the field of heterogeneous computing is to automate this process.

In some cases, an application is a collection of independent tasks, instead of the precedence-constrained set of subtasks considered in the previous discussion. For such cases, the matching and scheduling problem considers minimizing the completion time of the overall *metatask* consisting of all the tasks in the application.

This article summarizes information from various projects that cover different aspects of HC research. This is not an exhaustive survey of the literature. Each section of this article illustrates the concepts involved by describing a few representative techniques or systems.

In the next section, some HC application case studies are described. The section on example HC environments and tools discusses various software systems that are available to manage an HC suite of machines. Different ways of categorizing HC systems are presented in the taxonomies section. The conceptual model section provides a block diagram that illustrates what is involved in automatically mapping an application onto an HC system. Techniques for characterizing applications and representing machine performance are briefly examined in the section on task profiling and analytical benchmarking. Methods for using these characterizations to obtain an assignment of the subtasks to machines and to order the subtasks assigned to each machine is explored in the section on matching and scheduling.

EXAMPLES OF HC APPLICATION STUDIES

Simulation of Mixing in Turbulent Convection

An HC system at the Minnesota Supercomputer Center demonstrated the usefulness of HC through an application involving the three-dimensional simulation of mixing and turbulent convection (6). The system developed for this HC application consists of a Thinking Machines Corporation (TMC) SIMD CM-200 and MIMD CM-5, a vector CRAY 2, and a Silicon Graphics Inc. (SGI) VGX workstation, all communicating over a high-speed *high-performance parallel interface* (HiPPI) network.

The necessary simulation calculations were divided into three phases: (1) calculation of velocity and temperature fields, (2) calculation of particle traces, and (3) calculation of particle distribution statistics with refinement of the temperature field. The calculation of velocity and temperature fields associated with phase 1 is governed by two second-order partial differential equations. To approximate the field components in these equations, three-dimensional cubic splines (over a grid of size $128 \times 128 \times 64$) were used. The result was a linear system of equations representing the unknown spline coefficients. The system of equations for the spline coefficients was solved by applying a conjugate gradient method.

These conjugate gradient computations were performed on the CM-5. At each time interval, the grid of 128 × 128 × 64 spline coefficients was then sent to the CRAY 2, where phase 2 was performed.

The calculation of particle traces (phase 2) involved solving a set of ordinary differential equations based on the velocity field solution from phase 1. This calculation was performed by using a vectorized Lagrangian approach on the CRAY 2. Once they were computed, the coordinates of the particles and the spline coefficients of the temperature field were transferred from the CRAY 2 to the CM-200.

Phase 3 used the CM-200 to calculate statistics of the particle distribution and to assemble a three-dimensional temperature field, based on the spline coefficients received from phase 2. The 128 × 128 × 64 grid of splines was used to generate a file containing a 256 × 256 × 128 point temperature field, representing a volume of eight million voxels (a *voxel* is a three-dimensional element.) Then the voxels and the coordinates of the particles (one million particles were used) were sent to the SGI VGX workstation. The SGI VGX workstation visualized the results by using an interactive volume renderer. Although the simulation successfully demonstrated the benefits of HC, Klietz et al. (6) noted that much work is still required to improve the environment for developing more HC applications.

Collision of Galaxies on the I-Way

A metacomputer consisting of a TMC MIMD CM-5, Cray MIMD T3D, IBM MIMD SP-2, and SGI Power Challenge was used to carry out a very large simulation of colliding galaxies (7). The objective of this grand challenge was to harness the power of a collection of parallel machines to address the following questions: (a) What is the origin of the large-scale structure of the universe, and (b) How do galaxies form? The simulation was performed by solving an n-body dynamics problem and a gas dynamics problem. The n-body problem was solved using the *self-consistent field* (SCF) method. The gas dynamics problem was solved by the *piecewise parabolic method* (PPM).

The SCF code was parallelized so that if the entire calculation contains N particles and the computer has P processors, each processor evolves N/P particles. Each processor computes the contribution of its particles to the global gravitational field. These partial results were summed through a parallel reduction operation. After summing, the expansion coefficients were computed and broadcast to the processors. Then the processors use this information to reconstruct the global gravitational field and evaluate the gravitational acceleration of the particles.

The computation for each time step in the SCF requires 36,280 FLOP/s per particle. The particles were distributed so that the computation time per time step was approximately equivalent across machines. For example, 40,960 particles per processor on the CM-5 and 57,600 particles per processor on the T3D yielded a well-balanced load. A speed of 2.5 GFLOP/s was obtained for the CM-5 and T3D suite with 6,307,840 particles and the machines executing concurrently. The results obtained through the distributed simulation were viewed by using a distributed visualization system. The SGI Power Challenge was also used for solving the n-body problem by using the SCF code.

The PPM code was executed in parallel on an IBM SP2 machine in single program, multiple data streams (SPMD) mode. The PPM algorithm was computationally intensive and has a high computation-to-communication ratio. This code obtains nearly 21.2 MFLOP/s per node on the IBM SP2.

EXAMPLES OF HC ENVIRONMENTS AND TOOLS

This section overviews examples of software environments and tools that exist or are being developed for HC systems. These examples are implemented at several different levels from the high-level management framework of SmartNet to the low-level Globus Toolkit. The functionalities described here evolve and change rapidly; the descriptions here are based on the references given. Other tools include Fafner (8), Legion (9), Linda (10), Mentat (11), Ninf (12), and p4 (13).

SmartNet

SmartNet is a mapping framework that is employed for managing jobs and resources in a heterogeneous computational environment (14,15). SmartNet enables users to execute jobs on a network of different machines as if the network were a single machine. SmartNet supports a *resource management system* (RMS) that accepts requests for mapping a job or a sequence of jobs. The jobs are assigned to the machines in the suite by the mapping algorithms built into SmartNet. Traditionally, RMSs use opportunistic load-balancing schemes in which a job is assigned to the machine that becomes available first. However, SmartNet uses a multitude of more sophisticated algorithms to assign jobs to machines. SmartNet's goal is to optimize the mapping criteria in an HC environment, but these criteria are flexible, allowing SmartNet to adapt to many different situations and environments.

SmartNet exploits a variety of information resources to map and manage the applications within its heterogeneous environment. It considers (1) how well the computational capabilities of each machine match the computational needs of each application; (2) machine loading and availability; and (3) time for any needed intermachine data transfers. SmartNet also considers the current state of other resources, such as the intermachine communication network, before the mapping algorithms assign jobs to machines to account for the shared usage of all resources.

SmartNet uses a variety of optimization criteria to perform its mapping. Two currently implemented optimization criteria are (1) maximizing throughput by minimizing the expected completion time of the last job and (2) minimizing the average expected run time for each job. The mapping engine built into SmartNet uses a set of different heuristics for searching the space of possible maps to find the best one, as defined by the optimization criteria. Several heuristics have been implemented. They include algorithms based on greedy strategies with varying execution time complexities and algorithms based on evolutionary programming strategies. The mapper is modular and is designed to implement any algorithm that satisfies relatively simple interfacing requirements. The SmartNet mapping engine considers the heterogeneity present in both the network of machines and the user tasks.

One of the advantages of SmartNet is that it does not constrain the user to a particular programming language or require a special wrapper code for legacy programs. SmartNet only requires the user to provide a description of the time complexity of each program. SmartNet demonstrates that the performance of a metacomputer is enhanced by considering both the machine loading and heterogeneity in coordinating the execution of user programs. Thus, SmartNet provides a global, general-purpose, scalable, and tunable resource management framework for HC systems. SmartNet was designed and developed at a Naval laboratory (NRaD) and is operational at several research laboratories.

Ideas and lessons learned from SmartNet are used in designing and implementing the DARPA/ITO Quorum Program project called *Management System for Heterogeneous Networks* (MSHN). MSHN is a collaborative research effort among Naval Postgraduate School (NPS), NOEMIX, Purdue University, and University of Southern California (USC). The technical objective of the MSHN project is to design, prototype, and refine a distributed resource management system that leverages the heterogeneity of resources and tasks to deliver the requested qualities of service.

NetSolve

NetSolve is a client-server-based application designed to provide network access to remote computational resources for solving computationally intense scientific problems (16). The machines participating in a NetSolve system can be on a local or geographically distributed HC network.

For a given problem, a NetSolve client (i.e., an application task) sends a request to a NetSolve agent (residing in the same or different machine). Then the NetSolve agent selects a resource for the problem based on the size and nature of the problem. There can be several instantiations of NetSolve agents and clients. Every machine in a NetSolve system runs a NetSolve computational server for access to the machine's scientific packages. The NetSolve system can be accessed from a variety of interfaces, including MATLAB, Java, shell scripts, C, and FORTRAN. NetSolve can also be called in a blocking or nonblocking fashion, so that computations can be performed concurrently on the client system, thus improving performance.

NetSolve uses load balancing to improve system performance. For every machine in the NetSolve system, the execution time for a given problem is estimated. This estimate is used to determine the hypothetical best machine on which to execute the problem. This execution time estimate is based on several factors, including size of the data, size of the problem, complexity of the algorithm, network parameters, and machine characteristics.

To maintain accurate information on system performance, each instance of an agent maintains a value of the workload from every other server. A new workload value is conditionally broadcast at regular intervals, that is, if the value is outside a defined range, then the server broadcasts the value. This allows maintaining accurate system information, without needlessly burdening the network with the same workload value.

NetSolve has capabilities for handling fault tolerance at several different levels. Servers generally handle failure detection. Clients minimize side effects from service failures by maintaining lists of computational servers. Future work includes increasing the number of interfaces, improved load balancing, and allowing user-defined functions.

PVM AND HeNCE

Parallel Virtual Machine (PVM) is a software environment that enables utilizing an HC system as a single, connected, flexible, and concurrent computational resource (17,18). The PVM software package consists of system-level daemons, called *pvmds,* which reside on each machine in the HC system, and a library of PVM interface routines.

The pvmds are responsible for providing services to both local processes and remote processes executing on other machines in the HC system. By considering the entire set of pvmds collectively, a virtual machine is formed. This virtual machine allows viewing the HC system as a single metacomputer. The pvmds provide three major services: process and virtual machine management, communication, and synchronization. Process and virtual machine management issues include computational unit scheduling and placement, configuration and inclusion of remote computers into the virtual machine, and naming and addressing of resources. Communication is performed with asynchronous message passing, allowing a sending process to continue execution without waiting for a receive acknowledgment. The synchronization among processes provided by the pvmds is accomplished with barriers or other techniques. Multiple processes can be synchronized, including synchronization of processes that are executing on a local machine and processes that are executing remotely.

The PVM system also provides a library of interface routines. Applications access platforms in the HC system via library calls embedded within imperative procedural languages, such as C or FORTRAN. The library routines and the pvmds (resident on each machine) interact to provide communication, synchronization, and process management services. A single pvmd provides the requested service, or the service is provided by a group of pvmds in the HC system working in concert.

The *heterogeneous network computing environment* (HeNCE) is a tool that aids users of PVM in decomposing their application into subtasks and deciding how to distribute these subtasks to the machines currently available in the HC system (17). HeNCE allows the programmer to explicitly specify the parallelism for an application by creating a directed graph, where nodes represent subtasks (written in either FORTRAN or C) and arcs represent precedence constraints and flow dependencies. HeNCE also has four types of control constructs: conditional, looping, fan out, and pipelining.

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

Once the directed graph and cost matrix are specified, HeNCE uses PVM constructs to configure a subset of the machines defined in the cost matrix as a virtual machine. Then HeNCE initiates execution of the program. Each subtask in the graph is realized by a distinct process on some machine in the HC system. The subtasks communicate by sending parametric values necessary for executing of a given subtask. These parametric values are specified by the user for each subtask. Parametric values needed to begin execution of a subtask are obtained from predecessor subtasks. If the set of immediate predecessor subtasks does not have all of the required parameters for a subtask to begin execution, earlier predecessor subtasks are checked until all of the required parameters are located. Once all of the parameters are found, the subtask is executed, and the appropriate parameters are passed onto descendant subtasks. HeNCE traces the execution of the application for the display in real time or replay later.

Globus Metacomputing Infrastructure Tool Kit

The Globus project (19,20) defines a set of low-level mechanisms that provide basic HC infrastructure requirements, such as communication, resource allocation, and data access. These low-level mechanisms are part of the Globus metacomputing infrastructure tool kit, and are used to implement higher level HC services (e.g., mappers and parallel programming tools).

Each component in the tool kit defines an interface and an implementation for any HC environment. The interfaces allow higher level services to invoke that component's mechanisms. The implementation uses low-level instructions to realize these mechanisms on the different systems occurring within HC environments. Presently, the Globus tool kit consists of six components: (1) The *communication* component provides a wide range of communication methods, including message passing, remote procedure call, distributed shared memory, and multicast. (2) The *resource location, allocation,* and *process creation* module provides mechanisms for expressing application resource requirements and identifying resources suitable for these requirements; scheduling these resources after they have been located; and initiating the computation. The process creation includes initialization of executables, starting an executable, passing arguments, integrating the new process into the rest of the computation, and process termination. (3) In the *unified resource information service* component, a mechanism is provided for posting and receiving real-time information about the HC environment. (4) The *data access* module is responsible for providing high-speed access to remote data and files. (5) The heartbeat monitor module performs fault detection. Finally, (6) The *authentication interface* module provides basic authentication mechanisms for validating the identity of users and resources.

The modules of the Globus tool kit define an abstract HC system. The definition of this HC system simplifies development of higher level applications by allowing HC programmers to think of geographically distributed, heterogeneous collections of resources as unified entities. It also allows developing a range of alternative infrastructures, services, and applications. The stated long-term goal of the Globus project is to address the problems of configuration and performance optimization in HC environments. To accomplish this goal, the Globus project is designing and constructing a set of higher level services layered on the Globus tool kit. These higher level services would form an adaptive wide area resource environment (AWARE).

TAXONOMIES OF HETEROGENEOUS COMPUTING

One of the first classifications of HC systems provided in Watson et al. (21) divides systems into *mixed-machine HC* systems or *mixed-mode HC* systems. These two categories were defined earlier in this article. Mixed-machine HC systems denote *spatial* heterogeneity, whereas mixed-mode HC systems denote *temporal* heterogeneity. Recently, researchers have further refined this classification to obtain different schemes.

In Ekemecic et al. (22), a taxonomy called the EMMM = *e*xecution *m*ode, *m*achine *m*odel (EM^3) is presented for HC systems. In this scheme, HC systems are categorized in two orthogonal directions. One direction is the *execution mode* of the machine, which is defined by the type of parallelism supported by the machine. For example, high-performance computing architectures are often specialized to support either MIMD, SIMD, or vector execution modes. The heterogeneity based on this criterion is temporal or spatial. The second categorization is the *machine model,* which is defined as the machine architecture and machine performance. For example, Sun Sparc CY7C601 and Intel i860 are considered different architectures. In addition, two CPUs of the same type but driven by different speed clocks provide different machine performance and hence are considered different machine models. The heterogeneity based on this criterion is always spatial in nature.

HC systems are classified by counting the number of *execution modes* (EM) and the number of *machine models* (MM). The four categories proposed in Ref. 22 are (1) single execution mode, single machine model (SESM), (2) single execution mode, multiple machine model (SEMM), (3) multiple execution mode, single machine model (MESM), and (4) multiple execution mode, multiple machine model (MEMM). Fully homogeneous systems make up the SESM class. HC systems composed of different architectures (or clock speeds) with the same execution mode are in the SEMM class. Both the SEMM and MEMM classes are mixed-machine systems, but only the MEMM class includes different execution models and mixed-mode machines. The MESM corresponds to mixed-mode systems, that is, temporal heterogeneity. HC systems composed of different architectures, where some of the machines use different execution models, fall into the MEMM class.

In the classification provided in Eshagian (23), HC systems are grouped into (1) system heterogeneous computing (SHC) and (2) network heterogeneous computing (NHC). SHC is further divided into multimode SHC and mixed-mode SHC. *Multimode SHC* systems perform computations in both SIMD and MIMD modes simultaneously and exhibit spatial heterogeneity in a single machine. *Mixed-mode SHC* systems which switch execution between the SIMD and MIMD modes of parallelism, exhibit temporal in a single machine. The NHC systems are divided into multimachine NHC and mixed-machine NHC. *Multimachine NHC* denotes homogeneous distributed computing systems and *mixed-machine NHC* indicates heterogeneous distributed computing systems.

A CONCEPTUAL MODEL OF HETEROGENEOUS COMPUTING

In the examples featured in the application studies section, the programmer specified the machine assignment for each program segment and initial data item. One of the long-term goals of HC research is to develop software environments that automatically find a near-optimal mapping for an HC program expressed in a machine-independent high-level language. Performing the mapping automatically has the following benefits: (1) an increase in portability because the programmer need not be concerned with the composition of the HC suite, (2) easier use of the HC system, and (3) the possibility of deriving better mappings than the user can with ad hoc methods. Although no such environment exists today, many researchers are working toward developing an environment to automatically and efficiently perform the mapping of subtasks to machines in an HC suite. A conceptual model for such an environment using a dedicated HC suite of machines is described in Fig. 2 (based on Refs. 24 and 25).

For stage 1, information about the type of each application task and each machine in the HC suite is used to generate a set of parameters relevant to both the computational characteristics of the applications and the machine architectural features of the HC system. Categories for computational requirements and categories for machine capabilities are derived from this set of parameters.

Stage 2 consists of two components, *task profiling* and analytical benchmarking. Task profiling decomposes the application task into subtasks, where each subtask is computationally homogeneous. Usually, different subtasks have different computational needs. The computational requirements of each subtask are quantified by profiling the code and data. *Analytical benchmarking* quantifies how effectively each of the machines available in the suite performs on each of the types of computations required. The components of stage 2 are discussed further in the next section.

Stage 3 requires the information available from stage 2 to derive the estimated execution time of each subtask on each machine in the HC suite, along with the associated intermachine communication overheads. Then these statically derived results are incorporated with initial values for machine loading, intermachine network loading, and status parameters (e.g., machine/network faults) to perform the matching and scheduling of subtasks to machines. The result is an assignment of subtasks to machines and an execution schedule based on certain cost metrics (e.g., minimizing the overall execution time for all tasks.) Matching and scheduling in HC systems is examined in more detail later in this article.

Stage 4 is the execution of the given application. If a dynamic matching and scheduling system is employed, the subtask completion times and loading/status of the machines/network are monitored. The monitoring process is necessary because the actual computation times and data transfer times may be input-data-dependent and deviate from the static estimates. This information is used to reinvoke the matching and scheduling of stage 3 to improve the machine assignment and execution schedule. Automatic HC is a relatively new field. Preliminary frameworks for task profiling, analytical benchmarking, and mapping have been proposed,. However, further research is needed to make this conceptual model a reality (2,24).

TASK PROFILING AND ANALYTICAL BENCHMARKING

Task profiling specifies the types of computations present in the application program by decomposing the source program into homogeneous *code blocks* based on computational requirements (26). The set of code types defined is based on the features of the machine architectures available and the processing requirements of the applications considered for ex-

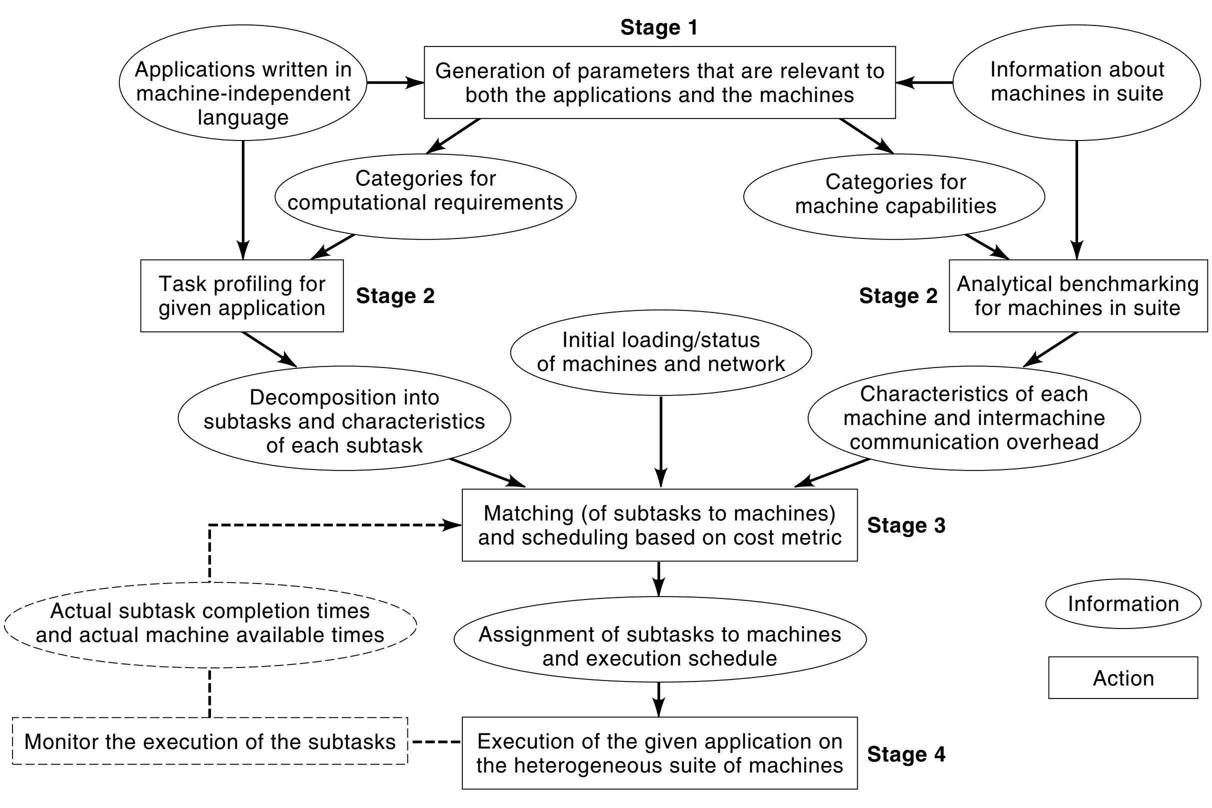

Figure 2. Model for integrating the software support needed for automating the use of heterogeneous computing systems (based on Refs. 24 and 25).

ecution on the HC system (phase 1 of the conceptual mode described in the previous section). This set of code types is a function of the application task code and the types and sizes of data sets it is to process. Task profiling is performed in stage 2 of the conceptual model presented in the previous section.

Analytical benchmarking provides a measure of how well each of the available machines in the heterogeneous suite performs on each of the given code types (26). In combination, task profiling and analytical benchmarking provide the necessary information for the matching and scheduling step (discussed in the next section). The performance of a particular code type on a specific kind of machine is a multivariable function. The variables within this performance function include the following: the requirements of the application (e.g., data precision), the size of the data set to be processed, the algorithm to be applied, programmer and compiler efforts to optimize the program, and the operating system and architecture of the machine that executes the specific code type (27).

Selection theory is a collection of mathematical formulations proposed for selecting the most appropriate machine for each code block. Many formulations (3,28,29) define analytical benchmarking as a method of measuring the optimal speedup of a particular machine type executing the best matched code type to a baseline system. The ratio between the actual speedup and the optimal speedup defines how well a code block is matched with each machine type. Generally, the actual speedup is less than the optimal speedup.

The *parallel assessment window system* (PAWS) and the *distributed heterogeneous supercomputing management system* (DHSMS) are briefly examined here. They represent examples of preliminary frameworks for implementing task profiling and analytical benchmarking.

The PAWS prototype consists of four tools: the application characterization tool, the architectural characterization tool, the performance assessment tool, and the interactive graphical display tool (30). First, the *application characterization tool* transforms a given program written in a specific subset of Ada into an acyclic graphical language that illustrates the program's data dependencies. The tool groups sets of nodes and edges into functions and procedures that allow describing the execution behavior of a given program at various levels. However, this tool does not perform task decomposition based on computational requirements and machine capabilities.

To benchmark machines, the *architectural characterization tool* divides the architecture of a specific type of machine into four categories: computation, data movement and communication, I/O, and control. Each category is repeatedly partitioned into subsystems until the lowest level subsystems are described by raw timing information. The *performance assessment tool* uses the information from the architectural characterization tool to generate timing information for operations on a given machine. Two sets of performance parameters for an application, parallelism profiles and execution profiles, are generated by the performance assessment tool. *Parallelism profiles* describe 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 are 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 of the other tools in PAWS.

The DHSMS classifies task profiling and analytical benchmarking results within a systematic framework (27). First, DHSMS generates a *universal set of codes* (USC) for task profiling. The USC is a standardized set of benchmarking programs used in analytical benchmarking. Similar to the hardware organizational information maintained by the architectural characterization tool in PAWS, a USC is constructed by using a hierarchical structure based on the machines in the HC suite. At the highest level of this hierarchical structure, modes of parallelism are selected to specify the machine architectures. At the second level, finer architectural characteristics, such as the organization of the memory system, are chosen. This hierarchical structure is organized so that the architectural characteristics at any level are choices for a given category (e.g., type of interconnection network used). DHSMS assigns a *code type* (i.e., computational characteristic) to each path from the root of the hierarchical structure to a leaf node. Every such path represents a specific set of architectural features, defined by the nodes within the path.

The DHSMS approach is extended in Yang et al. (31) to include generating a *representative set of templates* (RST) that characterize the execution behavior of the programs at various levels of detail. Many HC methodologies include a mathematical formulation for task profiling and analytical benchmarking that is similar in concept to that used in DHSMS (26,28,29,32).

MATCHING AND SCHEDULING

Overview

Matching and scheduling is an important component of the conceptual model of the automatic HC presented earlier. Finding an optimal solution for the matching and scheduling problem is NP-complete (5). For example, consider matching and scheduling 30 subtasks onto five machines. This means that there are 5^{30} possible mappings. Assuming it takes only 1 ns to evaluate the quality of one mapping, an exhaustive comparison of all possible mappings would require 5^{30} ns $>$ 4×10^{10} s $>$ 1000 years! Therefore, it is necessary to have heuristics to find the best mappings rather than to evaluate all possible mapping combinations. Mapping schemes can be either *static,* where the mapping decisions are made off-line before the execution of the subtask (33–38) or d*ynamic,* where the mapping decisions are made on-line during the execution of the subtasks (39–42).

A Mathematical Formulation of Matching and Scheduling in HC

The *optimal selection theory* (OST) (Fre89) provides the first known mathematical formulation for selecting an optimal heterogeneous configuration of machines for a given set of problems under a fixed cost constraint in HC systems. In the OST, it is assumed that the application consists of nonoverlapping *code segments* that are totally ordered in time. The overall execution time of the application equals the sum of the execution times of its code segments.

A code segment is defined to be *decomposable* if it can be partitioned further into *code blocks* that are executed in multiple copies of the best matched machine type. A sufficient number of machines of the best-matched machine type are assumed to be available. For simplicity, linear speedup is assumed for a decomposable code segment. Let the application have $S \geq 1$ code segments and $M \geq 1$ different types of machines to execute the code segments. Let v_j be the number of machines of type j and the cost of using a machine of type j is c_j. The estimated execution time of code segment i on machine type j is given by $t_{i,j}$ for all $1 \leq i \leq S$, $1 \leq j \leq M$. The optimization problem involves minimizing the total execution time T of the application, defined below, subject to a given constraint on the total cost C of the machines used. The cost incurred by using type j machines is given by v_jc_j. Assume that code segment i is best suited to machine type j. Because there are v_j number of type j machines, the execution time of code segment i on this type of machine is given by $t_{i,j}/v_j$. Thus, the goal is to minimize the total execution time of the application:

$$\text{minimize} \qquad T = \sum_{i=1}^{S} \left\{ \frac{t_{i,j}}{v_j} \right\}$$

given the total cost constraint

$$\sum_{j=1}^{M} v_j c_j \leq C$$

The *augmented optimal selection theory* (AOST) (29) is an extension of the OST. The AOST considers the performance of the code segments for all available machine type choices (not just the best matched machine type) and a fixed number of machines of each type. In practice, this extension is useful because the best matched machine may not be available, and only a limited number of machines of each type may be available. Another extension of the OST is provided by the *heterogeneous optimal selection theory* (HOST) (28). The HOST extends AOST by allowing concurrent execution of mutually independent code segments on different types of machine and incorporating the effects of different possible local mappings. Consider an example of a code block for multiplying two matrices onto a distributed memory parallel machine. Many implementations with varying execution characteristics can be derived for this code block. The HOST assumes that the best mapping choice (minimum execution time) is known for each code block.

The *generalized optimal selection theory* (GOST) further refines the OST to handle communication delays (32). In the GOST, the basic code element is called a *process,* which is *nondecomposable.* The application is represented by a directed acyclic graph (DAG), where a node denotes a process and an arc denotes a dependency between two processes. A node has a number of weights attached to it, corresponding to the execution times of the process on each machine type for each known mapping onto that machine. An edge has a number of weights, one for each communication path between each possible pair of host machines. In Narahari et al. (32), a matching and scheduling problem is formulated to assign each node to a machine type and to find a start time for each node so that the overall completion time of the application is minimized. Polynomial-time algorithms are provided in Ref. 32 for certain types of DAGs.

Static Matching and Scheduling Heuristics

The heuristics summarized later are based on the following assumptions, unless noted otherwise: Each application task is represented by a DAG, whose nodes are the subtasks that need to be executed to perform the application and whose arcs are the data dependencies between subtasks. Each edge is labeled by the global data item that is transferred between the subtasks connected by the edge. The matching and scheduling algorithm controls the HC machine suite (hardware platform). Subtask execution is nonpreemptive. The estimated expected execution time of each subtask on each machine is known. For each pair of machines in the HC suite, an equation for estimating the time to send data between those machines as a function of data set size is known.

Cluster-M Mapping Heuristic. The HC matching and scheduling process can be thought of as mapping a graph that represents a set of subtasks (*task graph*) onto a graph that represents the set of machines in the HC suite (*system graph*) (23). In Cluster-M, the mapping is performed in two stages. In the first stage, the task graph and system graph are clustered. The task-graph clustering combines the communication intensive subtasks into the same cluster. Similarly, the system-graph clustering combines the machines that are tightly coupled (i.e., small intermachine communication times) into the same cluster. The clustering of the task graph does not depend on the clustering of the system graph and vice versa. Therefore, a task or system graph needs to be clustered only once. In the second phase, the clustered task graph is mapped onto a clustered system graph. The clustering reduces the complexity of the mapping problem and improves the quality of the resulting mapping.

The Levelized Min-Time Heuristic. The *levelized min time* (LMT) heuristic is a static matching and scheduling algorithm for subtasks in an HC system (43). It is based on a list-scheduling class of algorithms. The LMT algorithm uses a two-phase approach. The first phase uses a technique called level sorting to order the subtasks based on the precedence constraints. The level sorting is defined as follows: The level 0 contains subtasks with no incident arcs. All predecessors with arcs to a level k subtask are in levels $(k - 1)$ to 0. For each subtask in level k, at least one incident arc (data dependency) exists such that the source subtask is in level $(k - 1)$. The level-sorting technique clusters subtasks that execute in parallel.

The second phase of the LMT algorithm uses a min-time algorithm to assign the subtasks level by level. The min-time algorithm is a greedy method that attempts to assign each subtask to the best machine. If the number of subtasks is more than the number of machines, then the smallest subtasks are merged until the number of subtasks is equal to the number of machines. Then the subtasks are ordered in descending order by their average computational time. Each subtask is assigned to the machine with the minimum completion time. Sorting the subtasks by the average computational time increases the likelihood that larger subtasks get faster machines.

One optimization to the LMT algorithm discussed in Iverson et al. (43) involves using information on the amount of communication between subtasks in different levels. This enables the scheduler to map subtasks that share large amounts of data to the same machine.

Genetic Matching and Scheduling Heuristic. In *genetic algorithms* (GAs), some of the possible solutions are encoded as *chromosomes,* the set of which is called a *population.* This population is iteratively operated on by the following steps until a stopping criterion is met. The first step is the selection step, where some chromosomes are removed and others are duplicated based on their *fitness value* (a measure of the quality of the solution represented by a chromosome). This is followed by the crossover step, where some chromosomes are paired and the corresponding components of the paired chromosomes are exchanged. Then, the chromosomes are randomly mutated, with the constraint that the resulting chromosomes still represent valid solutions for the physical problem.

To apply GAs to the subtask matching and scheduling problem in HC systems by using the approach presented in Wang et al. (38), the chromosomes are encoded with two parts: the matching string (mat) and the scheduling string (ss). If $\mathrm{mat}(i) = j$, then subtask s_i is assigned to machine m_j. The scheduling string is a topological sort of the DAG representing the task (i.e., a valid total ordering of the partially ordered DAG). If $\mathrm{ss}(k) = i$, then subtask s_i is the kth subtask in the total ordering. Each chromosome is associated with a fitness value, which is the completion time of the solution represented by this chromosome (i.e., the expected execution time of the application task if the mapping specified by this chromosome were used).

On small-scale tests with up to ten subtasks, three machines, and population size of 50, the GA approach found a solution (mapping) that had the same expected completion time as the optimal solution found by exhaustive search. On large-scale tests with up to 100 subtasks, 20 machines, and a population size of 200, the GA approach produced solutions (mappings) that were on the average 150% to almost 300% better than those produced by the (faster) nonevolutionary basic levelized min-time (LMT) heuristic proposed in Iverson et al. (43).

Dynamic Matching and Scheduling Heuristics

Static mapping heuristics assume that accurate estimates are available for parameters, such as subtask completion times and data transfer times. However, generally, such estimates have a degree of uncertainty in them because subtask computational times and data transfer times may depend on input data. Therefore, dynamic mapping heuristics that handle the uncertainty are needed. Researchers have proposed different dynamic heuristics for varying HC models (39–41,44). Furthermore, in dynamic mapping heuristics, machines come online and go off-line at run time.

Hybrid Remapper. The hybrid remapper heuristic described here is a dynamic algorithm for matching and scheduling subtask DAGs onto HC systems (42). An initial, statically obtained matching and scheduling is provided as input to the hybrid remapper. The hybrid remapper executes in two phases. In the first phase of the hybrid remapper, performed before application execution, the subtasks are partitioned into L levels as in the LMT heuristic. Each subtask is assigned a rank by examining the subtasks from level $(L - 1)$ to level 0. The *rank* of each subtask in the $(L - 1)$th level is set to its expected computational time on the machine to which it was assigned by the initial matching. The rank of a subtask s_i in level k is determined by computing the length of the critical path from s_i to the subtask where the execution terminates.

The second phase of the hybrid remapper occurs during the application execution. The hybrid remapper changes the matching and scheduling of the subtasks in level k while the subtasks in level $(k - 1)$ or before are running. The subtasks in level k are examined in descending order of static rank and each subtask is assigned to a machine with the earliest completion time for that particular subtask. The hybrid remapper starts scheduling level k subtasks when the first level $(k - 1)$ subtask begins its execution, and must finish the level k remapping before any level k subtask has the input data and machine available it needs to execute. When level k is being scheduled, it is highly likely that actual execution time information is used for many subtasks from levels 0 to $(k - 2)$. There may be some subtasks from levels 0 to $(k - 2)$ that are still running or waiting execution when subtasks from level k are being considered for remapping. Expected execution times are used for such subtasks.

Simulation results indicate that the hybrid remapper improves the performance of a statically obtained initial matching and scheduling by as much as 15% in some cases. Initial mappings for the simulation were generated by using the baseline heuristic (38). The timings also indicate that the remapping time needed per level of subtasks is on the order of hundreds of milliseconds for up to 50 machines and 500 subtasks. In the worst case situation, the computational time for the shortest running subtask must be greater than the per level scheduling time to obtain complete overlap between the execution of the subtasks and the operation of the hybrid remapper. Ongoing research will examine ways to increase the performance gain obtained from the use of the hybrid remapper.

Generational Scheduling. *Generational scheduling* (GS) heuristic is a dynamic mapping heuristic for subtasks in HC systems (39). It is a cyclic heuristic with four stages. First, the GS forms a partial scheduling problem by pruning all of the subtasks with unsatisfied precedence constraints from the initial DAG that represents the application, that is, the initial partial scheduling problem consists of subtasks that are independent or have no incident edges in the DAG. Then the subtasks in the initial partial scheduling problem are mapped onto the machine by using an auxiliary scheduler. The auxiliary scheduler considers the subtasks for assignment in a first come, first serve order. A subtask is assigned to a machine that minimizes the completion time (but not necessarily the execution time) of that particular subtask.

When a subtask from the initial partial scheduling problem completes its execution, the GS heuristic performs a remapping. During the remapping, the GS revises the partial scheduling problem by adding and removing subtasks from it. The completion of the subtask that triggered the remapping event may have satisfied the precedence constraints of some additional subtasks. These subtasks are added to the initial partial scheduling problem. The subtasks that have already started execution are removed from the initial partial scheduling problem. Once the revised partial scheduling problem is obtained, the subtasks in it are mapped onto the HC machine suite by using the auxiliary scheduler. This procedure is cyclically performed until the completion of all subtasks.

Self-Adjusting Scheduling for Heterogeneous Systems. The *self-adjusting scheduling for heterogeneous systems* (SASH) heuristic is a dynamic scheduling algorithm for mapping a set of independent tasks (metatask) onto an HC suite of machines (40). One processor is dedicated to computing the schedule, and this scheduling is overlapped with the execution of the tasks. At the end of each scheduling phase, the scheduling processor loads the tasks in that phase onto the working processors' local queues. Then the dedicated processor schedules the next subset of tasks while the previously scheduled tasks are being executed by the working processors.

The duration of the scheduling phase is determined by a lower bound estimate of the load on the working processors. The first working processor to complete executing all of the tasks in its local queue signals the scheduling processor, and then the scheduling processor assigns more tasks to all processors based on the partial schedule just computed. The SASH heuristic computes the schedules by using a variation of the branch-and-bound algorithm. In this variation, a tree is used to represent the space of possible schedules. A node in the tree represents a partial schedule consisting of a set of tasks assigned to a corresponding set of processors. An edge from a node represents an augmentation of the schedule by one more task-to-processor assignment.

A scheduling phase consists of one or more SASH iterations. In an iteration, the node with the lowest cost is expanded by augmenting the partial schedule with another task-to-processor assignment. The node expansions terminate when all the tasks are scheduled or when the time for scheduling phase i expires.

MATCHING AND SCHEDULING METATASKS

As defined earlier in this article, a metatask is a collection of independent tasks that need to be mapped onto an HC suite. Some tasks may have subtasks with data dependencies among them. Most of the heuristics and environments considered in the previous sections of this article are suitable for mapping tasks that can be decomposed into subtasks with data dependencies. Exceptions include the environments SmartNet and NetSolve (which manages metatasks and decomposed tasks) and the mapping heuristic SASH (which is for metatasks).

Typically, when independent tasks are involved, the tasks arrive randomly for service at the HC suite. Some machines in the suite may also go off-line, or new machines may come on-line. Therefore, dynamic mapping heuristics are usually employed to assign the tasks to machines. Furthermore, the tasks can have deadlines and priorities associated with them. Two types of dynamic approaches are on-line and interval. The on-line approach assigns each task to a machine when it is submitted. The interval approach waits for a set of new tasks to arrive and then maps those tasks and remaps any earlier tasks that have not yet started execution. Developing heuristics for matching and scheduling metatasks in HC systems is an active research area.

SUMMARY AND FUTURE DIRECTIONS

This article illustrates the concepts involved in heterogeneous distributed computing by sampling various research and development activities in this area. It is by no means an exhaustive survey of the HC literature. The practical importance of HC is revealed by the application studies summarized in this article. The conceptual model provided in Fig. 2 envisions an automatic HC programming environment. Most components of the model require further research to devise practical and theoretically sound methodologies (2,3,24). A flavor of the work performed in matching and scheduling is also provided in this article.

An important question that is particularly relevant to stages 1 and 2 of the conceptual model is, What information can be obtained automatically and what information should be provided by the programmer? The following areas should be further researched to realize the automatic HC environment envisioned in Fig. 2: (1) developing machine-independent programming languages, (2) designing high-speed networking systems, (3) studying communication protocols for reliable, low-overhead data transmission with a given quality of service requirements, (4) devising debugging tools, (5) formulating algorithms for task migration, fault tolerance, and load balancing, (6) designing user interfaces and user friendly programming environments, and (7) developing algorithms for applications with heterogeneous computing requirements. Most of these issues pertain to metatasks and applications decomposed into subtasks.

Machine-independent programming languages (45) that allow the user to augment the code with compiler directives are necessary to program the HC system. The following aspects should be considered in designing the language and directives: (1) the compilation of the program into efficient code for the machines in the suite, (2) the decomposition of tasks into subtasks, (3) the determination of the computational requirements of each subtask, and (4) the use of machine-dependent subroutine libraries.

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 techniques.

Another area of research is dynamic task migration between different parallel machines at execution time. Current research in this area involves determining how to move an executing task between different machines (46,47) and how to use dynamic task migration for load rebalancing or fault tolerance.

Ideally, information about the current loading and status of the machines in the HC suite and the network should be incorporated into the mapping decisions. Methods must be developed for measuring the current loading, determining the status (e.g., faulty or not faulty), and estimating the subtask completion times. The uncertainty present in the estimated parametric values, such as subtask completion times, should also be considered in determining the machine assignment and execution schedule.

In summary, although the use of currently available HC systems demonstrates their significant benefits, most of them require that the programmer have an intimate knowledge of what is involved in mapping the application task(s) onto the suite of machines. Hence, widespread use of the HC system is hindered. Further research on the areas briefly explained in this article should improve this situation and allow HC to realize its full potential.

ACKNOWLEDGMENT

The development of this article was supported in part by the DARPA/ITO Quorum Program under NPS subcontract numbers N62271-97-M-0900, N62271-98-M-0217, and N62271-98-M-0448. The authors thank N. Beck, J. Dongarra, M. Eshaghian, I. Foster, D. Gannon, R. Kwok, M. Theys, and D. Watson for their comments. Additionally, we acknowledge our past coauthors who became part of the anonymous “et al.’s” in the list of references: S. Ambrosius, J. Antonio, M. Atallah, M. Campbell, W. Cohen, H. Dietz, J. Fortes, R. Freund, M. Gherrity, M. Haldermann, D. Hensgen, E. Keith, T. Kidd, S.-D. Kim, M. Kussow, Y. Li, J. Lima, A. Maciejewski, R. Metzger, F. Mirabile, L. Moore, W. Nation, M. Nichols, V. Roychowdhury, B. Rust, M. Tan, and D. Watson.

BIBLIOGRAPHY

1. R. F. Freund and H. J. Siegel, Heterogeneous processing, *Computer,* **26** (6): 13–17, 1993.
2. H. J. Siegel et al., Heterogeneous computing, in A. Y. Zomaya (ed.), *Parallel and Distributed Computing Handbook,* New York: McGraw-Hill, 1996, pp. 725–761.
3. A. Khokhar et al., Heterogeneous computing: Challenges and opportunities, *Computer,* **26** (6): 18–27, 1993.
4. H. J. Siegel et al., Mixed-mode system heterogeneous computing, in M. M. Eshaghian (ed.), *Heterogeneous Computing,* Norwood, MA: Artech House, 1996, pp. 19–65.
5. D. Fernandez-Baca, Allocating modules to processors in a distributed system, *IEEE Trans. Softw. Eng.,* **SE-15**: 1427–1436, 1989.
6. A. E. Klietz, A. V. Malevsky, and K. Chin-Purcell, A case study in metacomputing: Distributed simulations of mixing in turbulent convection, *2nd Workshop Heterogeneous Process. (WHP ’93),* 1993, pp. 101–106.
7. M. L. Norman et al., Galaxies collide on the I-way: An example of heterogeneous wide-area collaborative supercomputing, *Int. J. Supercomput. Appl. High Perform. Comput.,* **10** (2/3): 132–144, 1996.
8. G. Fox and W. Furmanski, Web based high performance computing and communications, *5th IEEE Symp. High Perform. Distrib. Comput.,* 1996.
9. A. S. Grimshaw et al., Campus-wide computing: Early results using Legion at the University of Virginia, *Int. J. Supercomput. Appl. High Perform. Comput.,* **11** (2): 129–143, 1997.
10. N. Carriero, C. Gelernter, and T. G. Mattson, Linda in heterogeneous computing environments, *1st Workshop Heterogeneous Process. (WHP ’92),* 1992, pp. 43–46.
11. A. S. Grimshaw et al., Meta systems: An approach combining parallel processing and heterogeneous distributed computing systems, *J. Parallel Distrib. Comput.,* **21** (3): 257–270, 1994.
12. S. Sekiguchi et al., Ninf: Network based information library for globally high performance computing, *Parallel Object-Oriented Methods Appl. (POOMA),* 1996.
13. R. M. Butler and E. L. Lusk, Monitors, messages, and clusters: The p4 parallel programming system, *Parallel Comput.,* **20**: 547–564, 1994.
14. R. F. Freund et al., SmartNet: A scheduling framework for metacomputing, *2nd Int. Symp. Parallel Archit., Algorithms, Networks (ISPAN ’96),* 1996, pp. 514–521.
15. R. F. Freund et al., Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet, *7th Heterogeneous Comput. Workshop (HCW ’98),* pp. 184–199, 1998.
16. H. Casanova and J. Dongarra, NetSolve: A network-enabled server for solving computational science problems, *Int. J. Supercomput. Appl. High Perform. Comput.,* **11** (3): 212–223, 1997.
17. A. Beguelin et al., Visualization and debugging in a heterogeneous environment, *Computer,* **26** (6): 88–95, 1993.
18. V. S. Sunderam, PVM: A framework for parallel distributed computing, *Concurrency: Pract. Exper.,* **2** (4): 315–339, 1990.
19. I. Foster and C. Kesselman, Globus: A metacomputing infrastructure toolkit, *Int. J. Supercomput. Appl. High Perform. Comput.,* **11** (2): 115–128, 1997.
20. I. Foster and C. Kesselman, The Globus project: A status report, *7th Heterogeneous Comput. Workshop (HCW ’98),* 1998, pp. 4–18.
21. D. W. Watson et al., A framework for compile-time selection of parallel modes in a SIMD/SPMD heterogeneous environment, *2nd Workshop Heterogeneous Process. (WHP ’93),* 1993, pp. 57–64.
22. I. Ekemecic, T. Tartalja, and V. Milutinovic, A survey of heterogeneous computing: Concepts and systems, *Proc. IEEE,* **84**: 1127–1144, 1996.
23. M. M. Eshaghian (ed.), *Heterogeneous Computing,* Norwood, MA: Artech House, 1996.
24. H. J. Siegel, H. G. Dietz, and J. K. Antonio, Software support for heterogeneous comupting, in A. B. Tucker, Jr. (ed.), *The Computer Science and Engineering Handbook,* Boca Raton, FL: CRC Press, 1997, pp. 1886–1909.
25. M. Maheswaran, T. D. Braun, and H. J. Siegel, High-performance mixed-machine heterogeneous computing, *6th Euromicro Workshop Parallel Distrib. Process.,* 1998, pp. 3–9.
26. R. F. Freund, Optimal selection theory for superconcurrency, *Supercomput. ’89,* 1989, pp. 699–703.
27. A. Ghafoor and J. Yang, Distributed heterogeneous supercomputing management system, *Computer,* **26** (6): 78–86, 1993.
28. S. Chen et al., A selection theory and methodology for heterogeneous supercomputing, *2nd Workshop Heterogeneous Process. (WHP ’93),* 1993, pp. 15–22.
29. M. Wang et al., Augmenting the optimal selection theory for superconcurrency, *1st Workshop Heterogeneous Process. (WHP ’92),* 1992, pp. 13–22.
30. D. Pease et al., PAWS: A performance evaluation tool for parallel computing systems, *Computer,* **24** (1): 18–29, 1991.
31. J. Yang, I. Ahmad, and A. Ghafoor, Estimation of execution times on heterogeneous supercomputer architecture, *Int. Conf. Parallel Process. (ICPP ’93),* vol. 1, pp. 219–225, 1993.
32. B. Narahari, A. Youssef, and H. A. Choi, Matching and scheduling in a generalized optimal selection theory, *3rd Heterogeneous Comput. Workshop (HCW ’94),* 1994, pp. 3–8.
33. M. M. Eshaghian and Y.-C. Wu, A portable programming model for network heterogeneous computing, in M. M. Eshaghian (ed.), *Heterogeneous Computing,* Norwood, MA: Artech House, 1996, pp. 155–195.
34. M. Kafil and I. Ahmad, Optimal task assignment in heterogeneous computing systems, *6th Heterogeneous Comput. Workshop (HCW ’97),* 1997, pp. 135–146.
35. P. Shroff et al., Genetic stimulated annealing for scheduling data-dependent tasks in heterogeneous environments, *5th Heterogeneous Comput. Workshop (HCW ’96),* 1996, pp. 98–117.

36. G. C. Sih and E. A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, *IEEE Trans. Parallel Distrib. Syst.,* **4**: 175–187, 1993.

37. H. Singh and A. Youssef, Mapping and scheduling heterogeneous task graphs using genetic algorithms, *5th Heterogeneous Comput. Workshop (HCW '96),* 1996, pp. 86–97.

38. L. Wang et al., Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach, *J. Parallel Distrib. Comput.,* **47** (1): 8–22, 1997.

39. B. R. Carter et al., Generational scheduling for dynamic task management heterogeneous computing systems, *Inf. Sci.,* Special Issue on Parallel and Distributed Processing, **106** (3–4): 219–236, 1998.

40. B. Hamidzadeh, D. J. Lilja, and Y. Atif, Dynamic scheduling techniques for heterogeneous computing systems, *Concurrency: Prac. Exper.,* **7** (7): 633–652, 1995.

41. C. Leangsuksun, J. Potter, and S. Scott, Dynamic task mapping algorithms for a distributed heterogeneous computing environment, *4th Heterogeneous Comput. Workshop (HCW '95),* 1995, pp. 30–34.

42. M. Maheswaran and H. J. Siegel, A dynamic matching and scheduling algorithm for heterogeneous computing systems, *7th Heterogeneous Comput. Workshop (HCW '98),* 1998, pp. 57–69.

43. M. A. Iverson, F. Ozguner, and G. J. Follen, Parallelizing existing applications in a distributed heterogeneous environment, *4th Heterogeneous Comput. Workshop (HCW '95),* 1995, pp. 93–100.

44. J. R. Budenske, R. S. Ramanujan, and H. J. Siegel, A method for the on-line use of off-line derived remappings of iterative automatic target recognition tasks onto a particular class of heterogeneous parallel platforms, *Supercomput. J.,* **12** (4): 1998.

45. C. C. Weems, G. E. Weaver, and S. G. Dropsho, Linguistic support for heterogeneous parallel processing: A survey and an approach, *3rd Heterogeneous Comput. Workshop (HCW '94),* 1994, pp. 81–88.

46. J. B. Armstrong et al., Dynamic task migration from SPMD to SIMD virtual machines, *Int. Conf. Parallel Process. (ICPP '94),* vol. 2, pp. 160–169, 1994.

47. J. B. Armstrong and H. J. Siegel, Dynamic task migration from SIMD to SPMD virtual machines, *1st IEEE Int. Conf. Eng. Complex Comput. Syst.,* 1995, pp. 326–333.

MUTHUCUMARU MAHESWARAN
TRACY D. BRAUN
HOWARD JAY SIEGEL
Purdue University