Heterogeneous Computing: Challenges and Opportunities

Ashfaq A. Khokhar, Viktor K. Prasanna, Muhammad E. Shaaban,
and Cho-Li Wang
University of Southern California

Anytime you work with oranges and apples, you'll need a number of schemes to organize total performance. This article surveys the challenges posed by heterogeneous computing and discusses some approaches to opening up its opportunities.

Homogeneous computing, which uses one or more machines of the same type, has provided adequate performance for many applications in the past. Many of these applications had more than one type of embedded parallelism, such as single instruction, multiple data (SIMD) and multiple instruction, multiple data (MIMD). Most of the current parallel machines are suited only for homogeneous computing. However, numerous applications that have more than one type of embedded parallelism are now being considered for parallel implementation. On the other hand, as the amount of homogeneous parallelism in applications decreases, homogeneous systems cannot offer the desired speedups. To exploit the heterogeneity in computations, researchers are investigating a suite of heterogeneous architectures.

Heterogeneous computing (HC) is the well-orchestrated and coordinated effective use of a suite of diverse high-performance machines (including parallel machines) to provide superspeed processing for computationally demanding tasks with diverse computing needs.[1] An HC system includes heterogeneous machines, high-speed networks, interfaces, operating systems, communication protocols, and programming environments, all combining to produce a positive impact on ease of use and performance. Figure 1 shows an example HC environment.

Heterogeneous computing should be distinguished from network computing or high-performance distributed computing, which have generally come to mean either clusters of workstations or ad hoc connectivity among computers using little more than opportunistic load-balancing. HC is a plausible, novel technique for solving computationally intensive problems that have several types of embedded parallelism. HC also helps to reduce design risks by incorporating proven technology and existing designs instead of developing them from scratch. However, several issues and problems arise from employing this technique, which we discuss.

In the past few years, several technical meetings have addressed many of these issues. There is also a growing interest in using this paradigm to solve Grand Challenges problems. Richard Freund has organized the Heterogeneous Processing Workshops held each year at the IEEE International Parallel Processing

0018-9162/93/0600-0018$03.00 © 1993 IEEE

Symposiums.[2] Another related yearly meeting is the IEEE International Symposium on High-Performance Distributed Computing.[2]

Heterogeneous systems

The quest for higher computational power suitable for a wide range of applications at a reasonable cost has exposed several inherent limitations of homogeneous systems. Replacing such systems with yet more powerful homogeneous systems is not feasible. Moreover, this approach does not improve the versatility of the system. HC offers a novel cost-effective approach to these problems; instead of replacing existing multiprocessor systems at high cost, HC proposes using existing systems in an integrated environment.

Limitations of homogeneous systems. Conventional homogeneous systems usually use one mode of parallelism in a given machine (like SIMD, MIMD, or vector processing) and thus cannot adequately meet the requirements of applications that require more than one

Glossary

Analytical benchmarking: A procedure to analyze the relative effectiveness of machines on various computational types.

Code-type profiling: A code-specific function to identify various types of parallelism present in code and to estimate the execution times of each code type.

Cross-machine debuggers: Those available within the heterogeneous computing environment to help debug the application code that executes over multiple machines.

Cross-over overhead: That incurred in transferring data from one machine to another. It also includes data-format-conversion overhead between the two machines.

Cross-parallel compiler: An intelligent compiler that can generate intermediate code executable on different parallel machines.

Heterogeneous computing (HC): A well-orchestrated, coordinated effective use of a suite of diverse high-performance machines (including parallel machines) to provide fast processing for computationally demanding tasks that have diverse computing needs.

Metacomputations: Computations exhibiting coarse-grained heterogeneity in terms of embedded parallelism.

Mixed-mode computations: Computations exhibiting fine-grained heterogeneity in terms of embedded parallelism.

Multiple instruction, multiple data (MIMD): A mode in which code stored in each processor's local memory is executed independently.

Single instruction, multiple data (SIMD): A mode in which all processors execute the same instruction synchronously on data stored in their local memory.

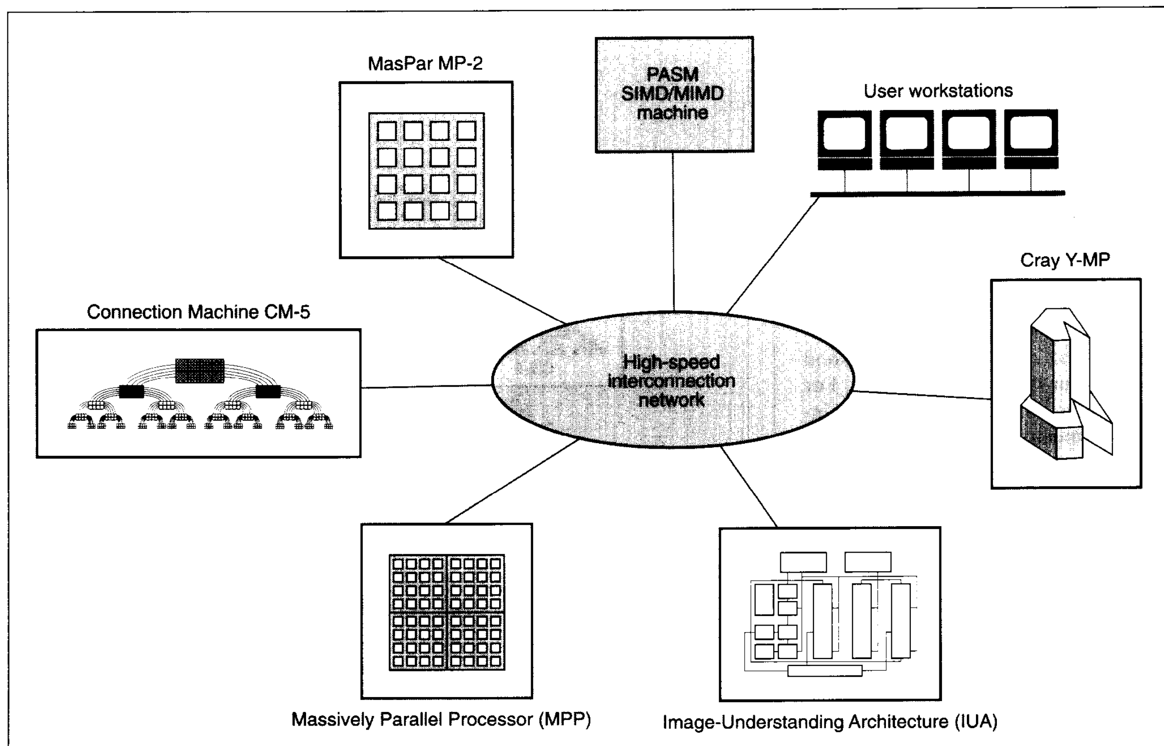


Figure 1. An example heterogeneous computing environment.

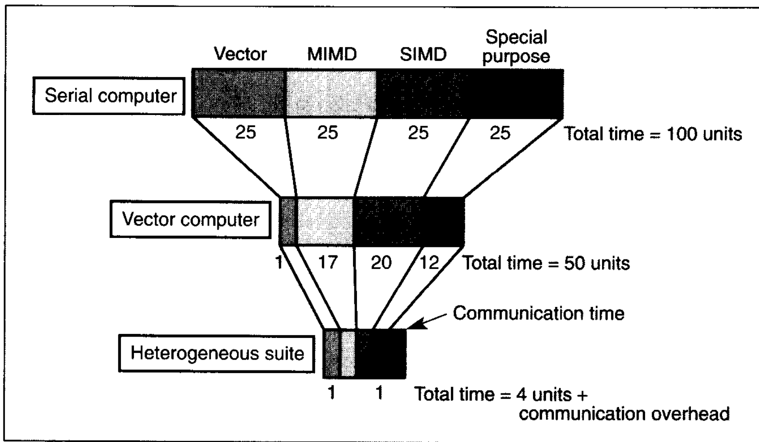


Figure 2. Execution of example code using various systems.

type of parallelism. As a result, any single type of machine often spends its time executing code for which it is poorly suited. Moreover, many applications need to process information at more than one level concurrently, with different types of parallelism at each level. Image understanding, a Grand Challenges problem, is one such application.[3]

At the lowest level of computer vision, image-processing operations are applied to the raw image. These computations have a massive SIMD-type parallelism. In contrast, the participants in the DARPA Image-Understanding Benchmark exercises[4] observed that high-level image-understanding computations exhibit coarse-grained MIMD-type characteristics. For such applications, users of a conventional multiprocessor system must either settle for degraded performance on the existing hardware or acquire more powerful (and expensive) machines.

Each type of homogeneous system suffers from inherent limitations. For example, vector machines employ interleaved memory with a pipelined arithmetic logic unit, leading to performance in high million floating-point operations per second (Mflops). If the data distribution of an application and the resulting computations cannot exploit these features, the performance degrades severely.

Consider an application code having mixed types of embedded parallelism. Assume that the code when executed on a serial machine spends 100 units of time. When this code is executed on a vector machine, the vector portion of the code is executed rapidly, while other portions of the code still have relatively higher execution times. Similarly, the same code when executed on a suite of heterogeneous machines (so that each portion of the code is executed on its matching machine type) is likely to achieve speedups. Figure 2 illustrates a possible scenario (the numbers are execution times in terms of basic units).

Heterogeneous computing. Heterogeneity in computing systems is not an entirely new concept. Several types of special-purpose processors have been used to provide specific services for improving system throughput. One of the most common is I/O handling. Attaching floating-point processors to host computers is yet another heterogeneous approach to enhance system performance. In high-performance computers, the concept of heterogeneity manifests itself at the instruction level in the form of several types of functional units, such as vector arithmetic pipelines and fast scalar processors. However, current multiprocessor systems remain

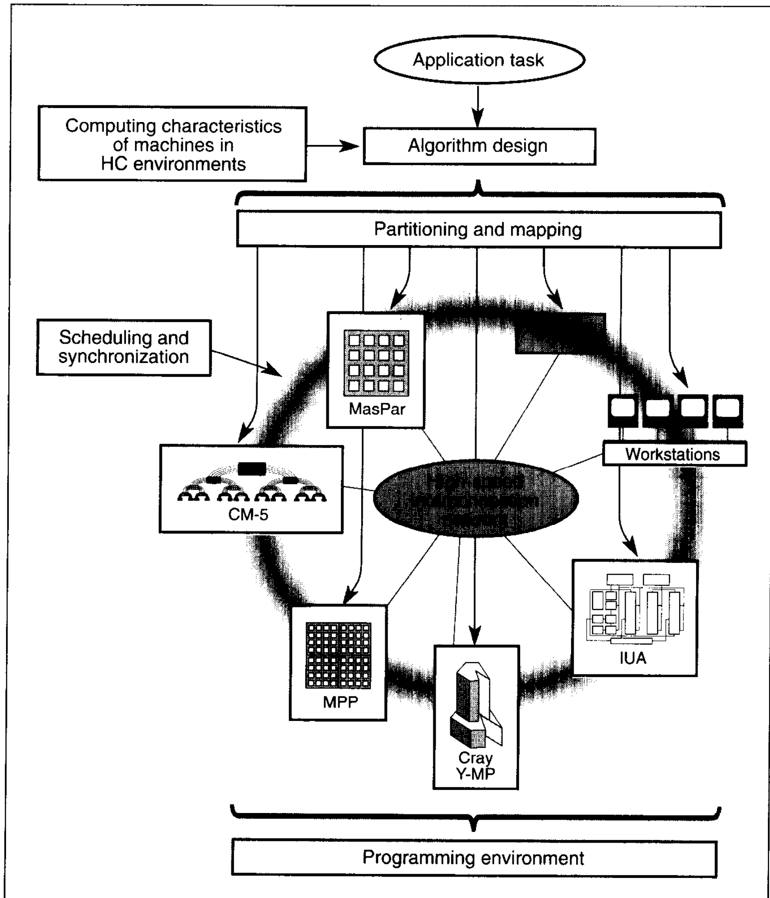


Figure 3. User-directed approach.

mostly homogeneous as far as the type of parallelism supported by them. Such systems have been traditionally classified according to the number of instruction and data streams.

An HC environment must contain the following components:

- a set of heterogeneous machines,
- an intelligent high-speed network connecting all machines, and
- a (user-friendly) programming environment.

HC lets a given system be adapted to a wide range of applications by augmenting it with specific functional or performance capabilities without requiring a complete redesign. Since HC comprises several autonomous computers, overall system fault tolerance and longevity are likely to improve.

Issues

We consider two approaches to using the HC paradigm. The first one analyzes an application to explore embedded heterogeneous parallelism. Researchers must devise new algorithms or modify existing ones to exploit the heterogeneity present in the application. Based on these algorithms, users develop the code to be executed by the machines.

In the second approach, an existing parallel code of the application is taken as input. To run this code in an HC environment, users must profile the types of heterogeneous parallelism embedded in the code. For this purpose, code-type profilers need to be designed. Figures 3 and 4 illustrate these approaches. However, both approaches need strategies for partitioning, mapping, scheduling, and synchronization. New tools and metrics for performance evaluation are also required. Parallel programming environments are needed to orchestrate the effective use of the computing resources.

Algorithm design. Heterogeneous computing opens new opportunities for developing parallel algorithms. In this section, we identify the efforts needed to devise suitable algorithms. The following issues must be considered by the designer:

(1) the types of machines available and their inherent computing characteristics,
(2) alternate solutions to various subproblems of the application, and
(3) the costs of performing the communication over the network.

Computations in HC can be classified into two types:[5,6]

- *Metacomputing.* Computations in this class fall into the category of coarse-grained heterogeneity. Instructions belonging to a particular class of parallelism are grouped to form a module; each module is then executed on a suitable parallel machine. Metacomputing refers to heterogeneity at the module level.
- *Mixed-mode computing.* In this fine-grained heterogeneity, almost every alternate parallel instruction belongs to a different class of parallel computation. Programs exhibiting this type of heterogeneity are not suitable for execution on a suite of heterogeneous machines because the communication overhead due to frequent exchange of information between machines can become a bottleneck. However, these programs can be executed efficiently on a single machine such as PASM (Partitionable SIMD/MIMD) which incorporates heterogeneous modes of computation. Mixed-mode computing refers to heterogeneity at the instruction level.

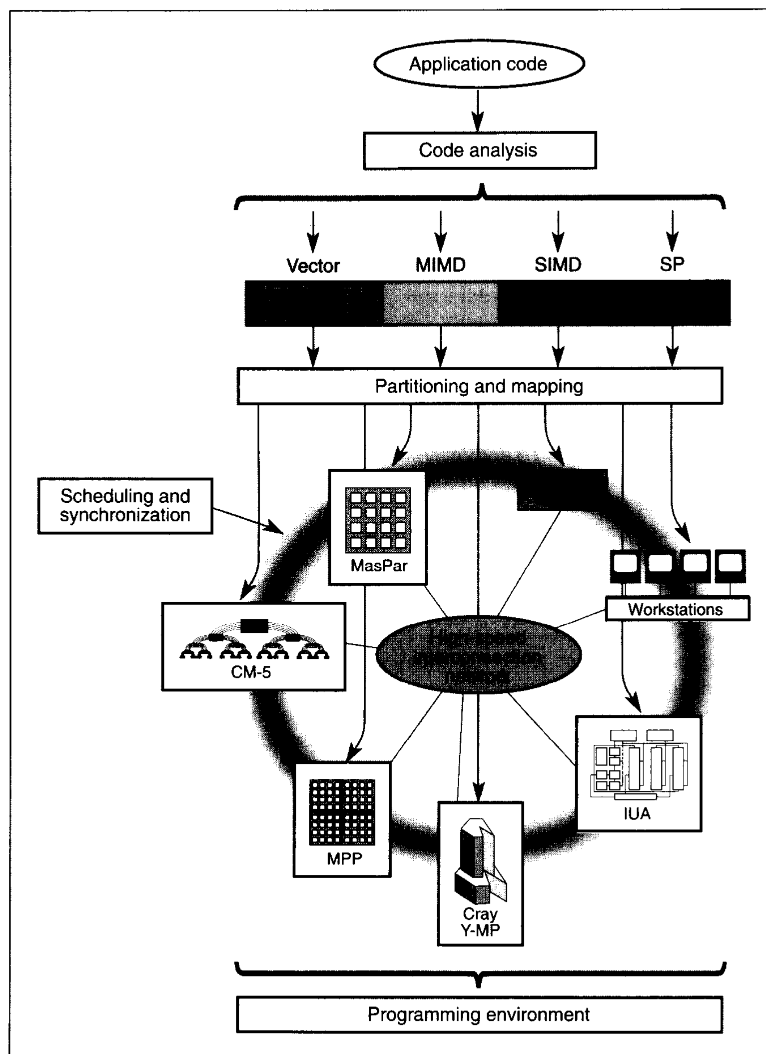


Figure 4. Compiler-directed approach.

Mixed-mode machines can achieve large speedups for fine-grained heterogeneity by using the mixed-mode processing available in a single machine. A mixed-mode machine, for example, can use its mode-switching capability to support SIMD/MIMD parallelism and hardware-barrier synchronization, thus improving its performance over a machine operating in SIMD or MIMD mode only.

Code-type profiling. Fast parallel execution of the code in a heterogeneous computing environment requires identifying and profiling the embedded parallelism. Traditional *program profiling* involves testing a program assumed to consist of several modules by executing it on suitable test data. The *profiler* monitors the execution of the program and gathers statistics, including the execution time of each program module. This information is then used to modify the modules to improve the overall execution time.

In HC, profiling is done not only to estimate the code's execution time on a particular machine but also to analyze the code's type. This is achieved by *code-type profiling*. As introduced by Freund,[7] this code-specific function is an off-line procedure; the statistics to be gathered include the types of parallelism of various modules in the code and the estimated execution time of each module on the machines available in the environment. Code types that can be identified include vectorizable, SIMD/MIMD parallel, scalar, and special purpose (such as fast Fourier transform).

Analytical benchmarking. This test measures how well the available machines perform on a given code type.[7] While code-type profiling identifies the type of code, analytical benchmarking ranks the available machines in terms of their efficiency in executing a given code type. Thus, analytical benchmarking techniques permit researchers to determine the relative effectiveness of a given parallel machine on various types of computation.

This benchmarking is also an off-line process and is more rigorous than previous benchmarking techniques, which simply looked at the overall result of running an entire benchmark code on a processor. Some experimental results obtained by analytical benchmarking show that SIMD machines are well suited for operations such as matrix computations and low-level image processing. MIMD machines, on the other hand, are most efficient when an application can be partitioned into a number of tasks that have limited intercommunication. Note that analytical benchmark results are used in partitioning and mapping.

Partitioning and mapping. Problems that occur in these areas of a homogeneous parallel environment have been widely studied. The partitioning problem can be divided into two subproblems. *Parallelism detection* determines the parallelism present in a given program. *Clustering* combines several operations into a program module and thus partitions the application into several modules. These two subproblems can be handled by the user, the compiler, or the machine at runtime.

In HC, parallelism detection is not the only objective; code classification based on the type of parallelism is also required. This is accomplished by code-type profiling, which also poses additional constraints on clustering.

Mapping (allocating) program modules to processors has been addressed by many researchers. Informally, in homogeneous environments, the mapping problem can be defined as assigning program modules to processors so that the total execution time (including the communication costs) is minimized. Several other costs, such as the interference cost, have also been considered. In HC, however, other objectives, such as matching the code type to the machine type, result in additional constraints. If such a mapping has to be performed at runtime for load-balancing purposes (or due to machine failure), the mapping problem becomes more complex due to the overhead associated with the code and data-format conversions. Various approaches to optimal and approximate partitioning and mapping in HC have been studied.[8-10]

Mapping in HC can be performed conceptually at two levels: system (or macro) and machine (or micro). At the system-level mapping, each module is assigned to one or more machines in the system so that the parallelism embedded in the module matches the machine type. Machine-level mapping assigns portions of the module to individual processors in the machine. The most common goal of the mapping process is to accomplish these assignments such that the overall runtime of the task is minimized.

Chen et al.[9] proposed a heuristic mapping methodology based on the Cluster-M mdoel, which facilitates the design of portable software. Only one algorithm is required for a given application, regardless of the underlying architecture. Various types of parallelism present in the application are identified. In addition, all communication and computation requirements of the application are preserved in an intermediate specification of the code. The architecture of each machine in the environment is modeled in the system representation, which captures the interconnections of the architecture. The four components of this approach are

- an intermediate model to provide an architecture-independent algorithm specification of the application,
- languages to support the specification in the intermediate model (such languages should be machine-independent and allow a certain amount of abstraction of the computations),
- a tool that lets users specify topologies of the machines employed in the HC environment, and
- a mapping module to match the problem specification and the system representation.

Figure 5 illustrates this methodology.

Machine selection. An interesting problem appears in the design of HC environments: How can one find the most appropriate suite of heterogeneous machines for a given collection of application tasks subject to a given constraint, such as cost and execution time? Freund[1] has proposed the Optimal Selection Theory (OST) to choose an optimal configuration of machines for executing an application task on a heterogeneous suite of computers with the assumption that the number of machines available is unlimited. It is also assumed that machines matching the given set of code types are available and that the application code is decomposed into equal-sized modules.

Wang et al.'s Augmented Optimal Selection Theory (AOST)[10] incorporates the performance of code segments on nonoptimal machine choices, assuming that the number of available machines

for each code type is limited. In this approach, the program module most suitable for one type of machine is assigned to another type of machine. In the formulation of OST and AOST, it has been assumed that the execution of all program modules of a given application code is totally ordered in time. In reality, however, different execution interdependencies can exist among program modules. Also, parallelism can be present inside a module, resulting in further decomposition of program modules. Furthermore, the effect of different mappings on different machines available for a program module has not been considered in the formulation of these selection theories.

The Heterogeneous Optimal Selection Theory (HOST)[9] extends AOST in two ways. It incorporates the effect of various mapping techniques available on different machines for executing a program module. Also, the dependencies between the program modules are specified as a directed graph. Note that OST and AOST assume linear ordering of program modules. In the formulation of HOST, an application code is assumed to consist of subtasks to be executed serially. Each subtask contains a collection of program modules. Each program module is further decomposed into blocks of parallel instructions, called code blocks.

To find an optimal set of machines, we have to assign the program modules to the machines so that

$$\Sigma T^i$$

is minimal, while

$$\Sigma C^i \leq C_{max}$$

where T^i is the time to execute program module i, C^i is the cost of the machine on which program module i is to be executed, and C_{max} is an overall constraint on the cost of the machines. The cost C^i and execution time T^i corresponding to the assignment under consideration can be obtained by using code-type profiling and/or by analyzing the algorithms.

Iqbal[11] presented a selection scheme that finds an assignment of program modules to machines in HC so that the total processing time is minimized, while the total cost of machines employed in the solution does not exceed an upper bound. The scheme can also find a solution to the dual of the above problem, that is, finding a least expensive set of machines to solve a given application subject to a maximal execution time constraint. This scheme is applicable to all of the above selection theories. The accuracy of the scheme, however, depends upon the method used to assign the program modules to the machines. Iqbal also shows that for applications in which the program modules communicate in a restrictive manner, one can find exact algorithms for selecting an optimal set of machines. If, however, the program modules communicate in an arbitrary fashion, the selection problem is NP-complete.

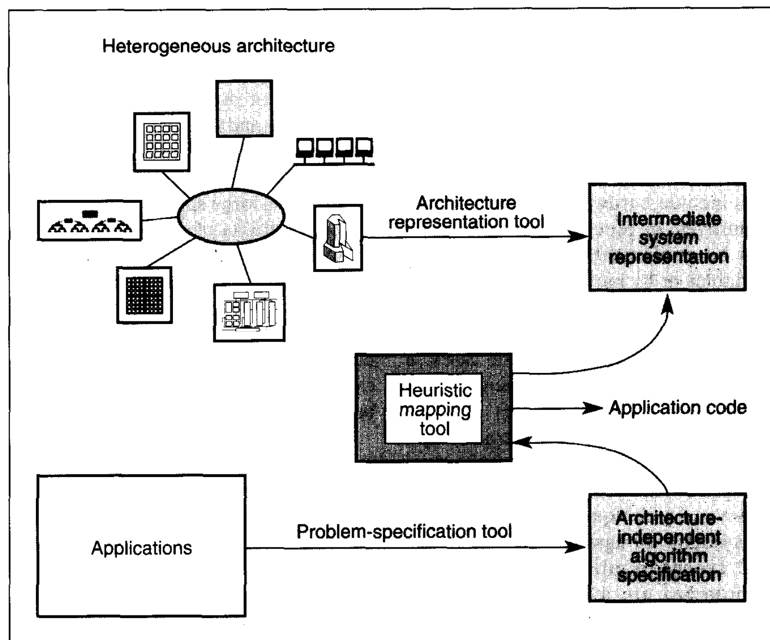


Figure 5. Cluster-M-based heuristic mapping methodology.

Scheduling. In homogeneous environments, a scheduler assigns each program module to a processor to achieve desired performance in terms of processor utilization and throughput. Designers usually employ three scheduling levels. High-level scheduling, also called job scheduling, selects a subset of all submitted jobs competing for the available resources. Intermediate-level scheduling responds to short-term fluctuations in the system load by temporarily suspending and activating processes to achieve smooth system operation. Low-level scheduling determines the next ready process to be assigned to a processor for a certain duration. Different scheduling policies, such as FIFO, round-robin, shortest-job-first, and shortest-remaining-time, can be employed at each level of scheduling.

While all three levels of scheduling can reside in each machine in an HC environment, a fourth level is needed to perform with scheduling at the system level. This scheduler maintains a balanced system-wide workload by monitoring the progress of all program modules. In addition, the scheduler needs to know the different module types and available machine types in the environment, since modules may have to be reassigned when the system configuration changes or overload situations occur. Communication bottlenecks and queueing delays incurred due to the heterogeneity of the hardware add constraints on the scheduler.

Synchronization. This process provides mechanisms to control execution sequencing and to supervise interprocess cooperation. It refers to three distinct but related problems:

- synchronization between the sender and receiver of a message,
- specification and control of the shared activities of cooperating processes, and
- serialization of concurrent accesses to shared objects by multiple processes.

A variety of synchronization methods have been proposed in the past: semaphores, conditional critical regions, monitors, and pass expressions, among others. In addition, some multiprocessors include hardware synchronization primitives. In general, synchronization can be implemented by using shared variables or by message-passing.

In heterogeneous computing, the synchronization problem resembles that of distributed systems. In both cases, a global clock and shared memory are absent, and (unpredictable) network delays and a variety of operating systems and programming environments complicate the process.

Several techniques used in distributed systems are again useful for solving HC synchronization problems. Two approaches are available: centralized (one machine is designated as a control node) and distributed (decision-making is distributed across the entire system). The correct choice depends on the topology, reliability, speed, and bandwidth of the network, in addition to the types and number of machines in the environment. However, reducing synchronization overhead is important to achieving large speedups in HC. Due to the possibility of several concurrently operating autonomous machines in the environment, application-code performance in HC is more sensitive to synchronization overheads. Frequent hand-shaking for synchronization may expend most of the available network bandwidth.

Interconnection requirements. Current local area networks (LANs) are not suitable for HC because higher bandwidth and lower latency networks are needed. The bandwidth of commercially available LANs is limited to about 10 megabits per second. On the other hand, in HC, assuming machines operating at 40 megahertz and 20 million instructions per second with a 32-bit word length, a bandwidth on the order of 1 gigabit/second is required to match the computation and communication speeds.

Even if higher bandwidth networks were available, three main sources of inefficiency would persist in current networks. First, application interfaces incur excessive overhead due to context switching and data copying between the user process and the machine's operating system. Second, each machine must incur the overhead of executing the high-level protocols that ensure reliable communication between program modules. Also, the network interface burdens the machine with interrupt handling and header processing for each packet. This suggests incorporating additional network-interface hardware in each machine.

Nectar[12] is an example of a network backplane for heterogeneous multicomputers. It consists of a high-speed fiber-optic network, large crossbar switches, and powerful network-interface processors. Protocol processing is off-loaded to these interface processors. A networking standard called Hippi (ANSI X3T9.3 High-Performance Parallel Interface)[13] is being implemented for realizing heterogeneous computing environments at various research sites. Hippi is an open standard that defines the physical and logical link layers of a 100-Mbyte/second network.

In HC, hardware modules from various vendors share physical interconnections. Differing communication protocols may make network-management problems complex. The following general approaches for dealing with network heterogeneity have been discussed in the literature:

(1) treat the heterogeneous network as a partitioned network, with each partition employing a uniform set of protocols;
(2) have a single "visible" network management console; and
(3) integrate the heterogeneous management functions at a single management console.

The IEEE Computer Society Technical Committee on Parallel Processing, the Technical Committee on Mass Storage, and several research sites are working together to define interface standards.

Programming environments. A parallel programming environment includes

Some academic sites

A number of academic sites are developing HC environments and applications (this list is not exhaustive).

Systems and architectures

Distributed High-Speed Computing (DHSC) project at Pittsburgh Supercomputing Center, University of Pittsburgh

Image-Understanding Architecture, University of Massachusetts at Amherst

Mentat, University of Virginia

Nectar-Based Heterogeneous System, Carnegie Mellon University

Northeast Parallel Architecture Center (NPAC), Syracuse University

Partitionable SIMD/MIMD (PASM), Purdue University

Institutes and departments

Beckman Institute, University of Illinois at Urbana-Champaign

Department of Biological Sciences, University of California at Los Angeles

Department of Computer Science, Kent State University

Department of Computer Science, University of California at San Diego

Department of Computer and Information Sciences, New Jersey Institute of Technology

Department of Electrical Engineering-Systems, University of Southern California

Department of Math and Computer Science, Emory University

Minnesota Supercomputer Center (MSC), University of Minnesota at Minneapolis

Supercomputer Computations Institute (SCI), Florida State University

parallel languages, intelligent compilers, parallel debuggers, syntax-directed editors, configuration-management tools, and other programming aids.

In homogeneous computing, intelligent compilers detect parallelism in sequential code and translate it into parallel machine code. Parallel programming languages have been developed to support parallel programming, such as MPL for MasPar machines, and Lisp and C for the Connection Machine. In addition, several parallel programming environments and models have been designed, such as Code, Faust, Schedule, and Linda.

HC requires machine-independent and portable parallel programming languages and tools. This requirement creates the need for designing cross-parallel compilers for all machines in the environment, and parallel debuggers for debugging cross-machine code. Several programming models and environments have been developed in the past for heterogeneous computing.[8,14-16]

The Parallel Virtual Machine (PVM) system,[16] evolved over the past three years, consists of software that provides a virtual concurrent computing environment on general-purpose networks of heterogeneous machines. It is composed of a set of user-interface primitives and supporting software that enable concurrent computing on a loosely coupled network of high-performance machines. It can be implemented on a hardware base consisting of different architectures, including single-CPU systems, vector machines, and multiprocessors (see Figure 6).

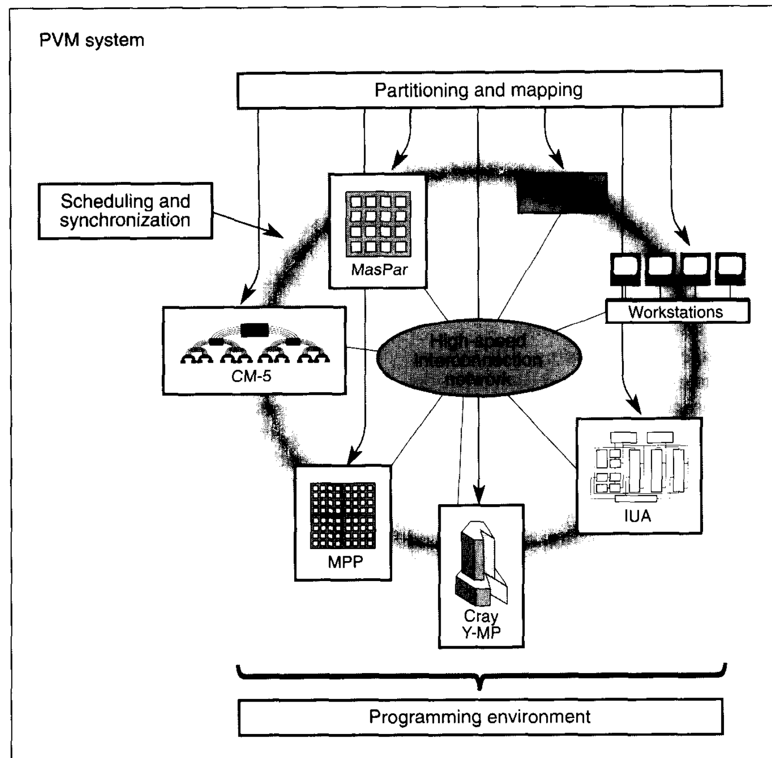


Figure 6. An overview of the Parallel Virtual Machine system.

Application programs view the PVM system as a general and flexible parallel computing resource that supports shared memory, message-passing, and hybrid models of computation. A heterogeneous application can be decomposed into several subtasks based on the embedded types of computation and then executed by using PVM subroutines on different matching machines available on the network. The PVM primitives are provided in the form of libraries linked to application programs written in imperative languages. They support process initiation and management, message-passing, synchronization, and other housekeeping facilities.

Support software provided by the PVM system executes on a set of user-specified computing elements on a network, presenting a virtual concurrent computing environment to users.

Performance evaluation. Performance tools are used to summarize the run-time behavior of an application, including analyzing resource use and the cause of any performance bottleneck. Depending on its design, a performance tool can describe program behaviors at many levels of detail. The two most common are the intraprocess and interprocess levels. Intraprocess performance tools, such as the *gprof* facility on BSD Unix, the HP sampler/3000, and the Mesa Spy, provide information about individual processes.

Performance tools for distributed computing systems concentrate on the interactions between the processes. Integrated performance models that observe the status and the performance events at all levels can be found in the PIE (Programming and Instrumentation Environment) project.[17]

Designing performance-evaluation tools for distributed computing systems involves collecting, interpreting, and evaluating performance information from application programs, the operating system, the communication network, and other hardware modules employed in the environment. The inherent concurrency in a distributed computing environment, the lack of total ordering of events on different machines, and the nondeterministic nature of the communication delays between the processes make the problem of evaluating performance more complex.

The impact of the code type must be considered. Thus, performance metrics such as processor utilization, speedup, and efficiency are difficult to compute. Indeed, these metrics must be carefully defined to make a reasonable performance evaluation.

Image understanding

Intrinsic parallelism in image processing and the variety of heuristics available for problems in image understanding make computer vision an ideal vehicle for studying heterogeneous computing. From a computational perspective, vision processing is usually organized as follows:

• *Early processing of the raw image* (often called low-level processing). At this level, the input is an image. The output image is approximately the same

size. Convolutions are performed on each pixel in parallel. The data communication among the pixels is local to each pixel.

• *Interfacing between low-level and image-understanding problems* (often termed intermediate-level processing). The operations performed on each data item can be nonlocal. The communication is also irregular as compared with that of low-level processing.

• *Image understanding.* By this we mean using the acquired data from the above processing (for example, geometric features such as shape, orientation, and moments) to infer semantic attributes of an image. Processing at this level can be classified as knowledge and/or symbolic processing. Search-based techniques are widely used at this level.

As evident in the preliminary results from the 1988 DARPA Image-Understanding Benchmark,[18] each level in computer vision exhibits a different type of parallelism. Therefore, at each level a suitable type of parallel machine must be employed. Corresponding to each of the above classes of problems, a suitable class of architecture was proposed:[3]

• *SIMD machines.* Machines in this class are well suited for computations in low-level and in some intermediate-level computer vision problems because of the regular dataflow and iconic operations in these two levels. For example, two-dimensional cellular arrays and mesh-connected computers have been proposed for a large class of geometric and graph-based problems in image processing. Parallel machines such as the MasPar MP-series and the Connection Machine CM-2000 fall in this category. Pipelined parallel machines (like the Carnegie Mellon University Warp machine) are also well suited for low- and intermediate-level vision computations.

• *Medium-grained MIMD machines.* Various intermediate- and high-level vision tasks are computationally intensive with irregular dataflow. Moreover, the size of the input is smaller than the input image size. Parallel systems having a set of powerful processors are suitable for performing computations in intermediate- and high-level vision tasks. The Connection Machine CM-5, Vista/2, Alliant FX-80, and Sequent Symmetry 81 are some examples.

• *Coarse-grained MIMD machines.* High-level vision tasks such as image understanding/recognition and symbolic processing employ complex data structures. Many of the proposed algorithms for such problems are nondeterministic, and architectural requirements for these problems demand coarse-grained MIMD machines. Parallel machines such as the Aspex ASP and Vista/3 are well suited for this class of problems.

Another approach is to build machines having multiple computational capabilities embedded in a single system. These architectures consist of several levels. Typically, the lower levels operate in SIMD mode and the higher levels operate in MIMD mode. In the Image-Understanding Architecture,[19] the lowest level has bit-serial processors, and the intermediate level consists of digital signal processors. The highest level consists of general-purpose microprocessors operating in MIMD mode.

An example vision task. We present an example vision task and identify the different types of parallelism. We have chosen the DARPA Integrated Image-Understanding Benchmark[4] as an example task. The overall task performed by this benchmark is the recognition of an approximately specified two-and-a-half-dimensional "mobile" sculpture in a cluttered environment, given images from intensity and range sensors.

Steps in the benchmark can be identified by the vision-task classifications. First, low-level operations such as connected component labeling and corner extraction are performed. Then, grouping the corners (an intermediate-level vision operation) results in the extraction of candidate rectangles. Finally, partial matching of the candidate rectangles is followed by confirmed matching (a high-level vision task). The results obtained on several different parallel machines were reported at the 1988 Image-Understanding Workshop. Details of the benchmark results can be found in Weems et al.[18]

As they describe, directly interpreting these results would be unfair, since there were many undefined factors in the benchmark description. However, the benchmark does give pointers to how different machines can be classified with respect to their suitability for performing operations at different levels of vision. Overall, the simulation results show that the (heterogeneous) Image-Understanding Architecture performs better than any single machine considered. These results support the suitability of a heterogeneous environment for computer vision applications.

Heterogeneous computing offers new challenges and opportunities to several research communities. To support this paradigm, the following areas of research must be investigated:

• Designing tools to identify heterogeneous parallelism embedded in applications.
• Studying issues in high-speed networking, including available technologies and specialized hardware for networking.
• Designing communication protocols to reduce the cross-over overheads that occur when different machines communicate in the same environment.
• Developing standards for parallel interfaces between various machines.
• Designing efficient partitioning and mapping strategies to exploit heterogeneous parallelism embedded in applications.
• Designing user interfaces and user-friendly programming environments to program diverse machines in the same environment.
• Developing algorithms for applications with heterogeneous computing requirements.

Indeed, HC provides an opportunity to bring together research from various disciplines of computer science and engineering to develop a feasible approach for applications in the Grand Challenges problem set. ■

Acknowledgments

We thank Richard Freund and Ashraf Iqbal for many helpful discussions. This research was partly supported by the National Science Foundation under Grant No. IRI-9145810.

References

1. R. Freund and D. Conwell, "Superconcurrency: A Form of Distributed Heterogeneous Supercomputing," *Supercomputing Review*, Oct. 1990, pp. 47-50.

2. Newsletter of the IEEE Computer Soci-

ety Technical Committee on Parallel Processing (TCPP), Vol. 1, No. 1, Oct. 1992.

3. V.K. Prasanna Kumar, *Parallel Algorithms and Architectures for Image Understanding*, Academic Press, Boston, 1991.

4. C. Weems et al., "An Integrated Image-Understanding Benchmark: Recognition of a 2-1/2D Mobile," *Proc. DARPA Image-Understanding Workshop*, Morgan Kaufmann Publishers, San Mateo, Calif., 1988, pp. 111-126.

5. T. Berg and H.J. Siegel, "Instruction Execution Trade-Offs for SIMD vs. MIMD vs. Mixed-Mode Parallelism," *Proc. Int'l Parallel Processing Symp. (IPPS)*, IEEE CS Press, Los Alamitos, Calif., Order No. 2167, 1991, pp. 301-308.

6. A. Khokhar et al., "Heterogeneous Supercomputing: Problems and Issues," *Proc. Workshop on Heterogeneous Processing*, IEEE CS Press, Los Alamitos, Calif., Order No. 2702, 1992, pp. 3-12.

7. R. Freund, "Optimal Selection Theory for Superconcurrency," *Proc. 89 Supercomputing*, IEEE CS Press, Los Alamitos, Calif., Order No. M2021 (microfiche), 1989, pp. 13-17.

8. G. Agha and R. Panwar, "An Actor-Based Framework for Heterogeneous Computing Systems," *Proc. Workshop on Heterogeneous Processing*, IEEE CS Press, Los Alamitos, Calif., Order No. 2702, 1992, pp. 35-42.

9. S. Chen et al., "A Selection Theory and Methodology for Heterogeneous Supercomputing," *Proc. Workshop on Heterogeneous Processing*, IEEE CS Press, Los Alamitos, Calif., Order No. 3532-02, 1993.

10. M. Wang et al., "Augmenting the Optimal Selection Theory for Superconcurrency," *Proc. Workshop on Heterogeneous Processing*, IEEE CS Press, Los Alamitos, Calif., Order No. 2702, 1992, pp. 13-22.

11. M. Iqbal, "Partitioning Problems for Heterogeneous Computer Systems," tech. report, Dept. of Electrical Engineering-Systems, Univ. of Southern California, Los Angeles, 1993.

12. E. Arnould et al., "The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, IEEE CS Press, Los Alamitos, Calif., Order No. M1936 (microfiche), 1989, pp. 205-216.

13. ANSI X3T9.3, "High-Performance Parallel Interface: Hippi-PH, Hippi-SC, Hippi-FP, Hippi-LE, and Hippi-MI," Working Draft Proposed American National Standard for Information Systems, American Nat'l Standards Inst., New York, Jan.-Apr., 1991.

14. C. de Castro and S. Yalamanchili, "Partitioning Signal Flow Graphs for Execution on Heterogeneous Signal Processing Architectures," *Proc. Workshop on Heterogeneous Processing*, IEEE CS Press, Los Alamitos, Calif., Order No. 2702, 1992, pp. 81-86.

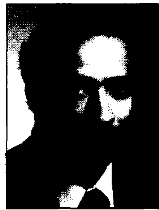
15. J. Potter, "Heterogeneous Associative Computing," *Proc. Workshop on Heterogeneous Processing*, IEEE CS Press, Los Alamitos, Calif., Order No. 3532-02, 1993.

16. V. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, Vol. 2, No. 4, Dec. 1990, pp. 315-339.

17. Z. Segall and L. Rudolph, "PIE: A Programming and Instrumentation Environment for Parallel Processing," *IEEE Software*, Vol. 2, No. 6, Nov. 1985, pp. 22-27.

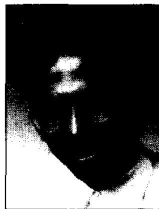
18. C. Weems et al., "Preliminary Results from the DARPA Integrated Image-Understanding Benchmark," *Parallel Architectures and Algorithms for Image Understanding*, V.K. Prasanna, ed., Academic Press, Boston, 1991, pp. 399-499.

19. D. Shu, J. Nash, and C. Weems, "A Multiple-Level Heterogeneous Architecture for Image Understanding," *Proc. Int'l Conf. Pattern Recognition*, IEEE CS Press, Los Alamitos, Calif., Vol 2, Order No. 2063, 1990.



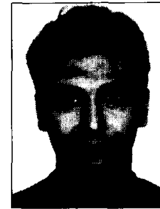
Ashfaq A. Khokhar is a PhD candidate in the Department of Electrical Engineering-Systems at the University of Southern California, Los Angeles. His areas of research include parallel architectures and scalable algorithms, image understanding and parallel processing, VLSI computations, interconnection networks, and heterogeneous computing.

Khokhar received the BSc degree in electrical engineering from the University of Engineering and Technology, Lahore, Pakistan in 1985 and the MS degree in computer engineering from Syracuse University in 1988. He is a student member of the Computer Society.



Viktor K. Prasanna (V.K. Prasanna Kumar) is an associate professor in the Department of Electrical Engineering-Systems, University of Southern California, Los Angeles. His research interests include parallel computation, computer architecture, VLSI computations, and computational aspects of image processing, vision, robotics, and neural networks.

Prasanna received the BS degree in electronics engineering from Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the PhD in computer science from Pennsylvania State University in 1983. He serves as the symposium chair of the 1994 IEEE International Parallel Processing Symposium and is a subject area editor of the *Journal of Parallel and Distributed Computing*, *IEEE Transactions on Computers*, and *IEEE Transactions on Signal Processing*. He is the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing and is a senior member of the Computer Society.



Muhammad E. Shaaban is a PhD candidate in the Department of Electrical Engineering-Systems, University of Southern California. His areas of research include parallel optical interconnection networks, parallel algorithms for image processing, and heterogeneous computing.

Shaaban received the BS and MS degrees in electrical engineering from the University of Petroleum and Minerals, Dhahran, Saudi Arabia, in 1984 and 1986, respectively. He recently served as a session chair at the International Parallel Processing Symposium. He is a student member of the Computer Society.

Cho-Li Wang is a PhD candidate in the Department of Electrical Engineering-Systems, University of Southern California, Los Angeles. His areas of research include computer architectures and algorithms, image understanding and parallel processing, image compression, and heterogeneous computing.

Wang received the BS degree in computer science and information engineering from National Taiwan University, Taiwan, in 1985 and the MS degree in computer engineering from the University of Southern California in 1990.

Readers can contact Viktor K. Prasanna at the School of Engineering, Department of Electrical Engineering-Systems, University of Southern California, University Park, Los Angeles, CA 90089-2562.