

PARALLEL COMPUTING^{*}

Charles Koelbel

This work is produced by The Connexions Project and licensed under the
Creative Commons Attribution License [†]

Abstract

This module serves as the introductory module for the Open Education Cup contest (<http://OpenEducationCup.org>) collaborative book. This introduction outlines the primary focus of the contest and provides examples to help authors get started with their contest entries.

What is parallel computing, and why should I care?

Parallel computing simply means using more than one computer processor to solve a problem. Classically, computers have been presented to newcomers as “sequential” systems, where the processor does one step at a time. There are still machines and applications where this is true, but today most systems have parallel features. Some examples are shown below.¹

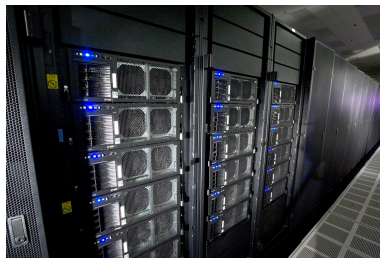


Figure 1

Supercomputers achieve astounding speed on scientific computations by using amazing numbers of processors. The Roadrunner system at Los Alamos National Laboratory (Figure 1) is currently the world's fastest computer, with over 100,000 processor cores combining to reach 1 PetaFLOPS (10^{15} floating point operations per second).

^{*}Version 1.10: Nov 5, 2008 10:48 am US/Central

[†]<http://creativecommons.org/licenses/by/2.0/>

¹We hasten to point out that, although we use certain commercial systems and chips as examples, they are by no means the only ones. Additional examples in all categories are available from other companies. The Open Education Cup would welcome modules on any type of parallel system.



Figure 2

Servers use multiple processors to handle many simultaneous requests in a timely manner. For example, a web server might use dozens of processors to supply hundreds of pages per second. A 64-processor server (Figure 2) could supply an average university department's need for computation, file systems, and other support.

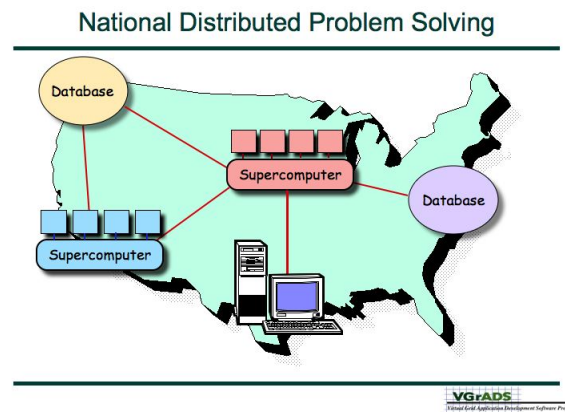


Figure 3

Grid computing combines distributed computers (often from different organizations) into one unified system. One vision (Figure 3) is that a scientist working at her desk in Houston can send a problem to be solved by supercomputers in Illinois and California, accessing data in other locations. Other grid computing projects, such as the World Community Grid, seek to use millions of idle PCs to solve important problems, such as searching for candidate drug designs.

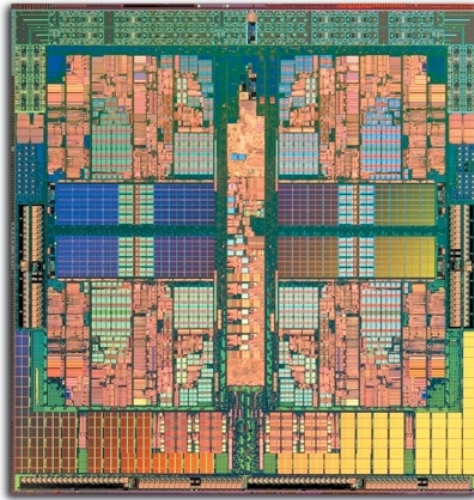


Figure 4

Multicore chips include several processor cores on a single computer chip. Even a laptop (or other small system) that uses a multicore chip is a parallel computer. For example, the Quad-core AMD Opteron processor (Figure 4) is one of a family of chips with 4 cores. As another example, the IBM Cell processor chips used in the Roadrunner supercomputer have 8 “Synergistic Processing Elements” (i.e. fast cores) each, plus 1 Power Processing Element (which controls the others).



Figure 5

Graphics processors (often called GPUs) are a special type of chip originally designed for image processing. They feature massive numbers of scalar processors to allow rendering many pixels in parallel. For example, The NVIDIA Tesla series (Figure 5) boasts 240 processors. Such special-purpose parallel chips are the basis for accelerated computing.

In effect, all modern computing is parallel computing.

Getting the most out of a parallel computer requires finding and exploiting opportunities to do several things at once. Some examples of such opportunities in realistic computer applications might include:

- Data parallelism – updating many elements of a data structure simultaneously, like a image processing program that smoothes out pixels

- Pipelining – using separate processors on different stages of a computation, allowing several problems to be “in the pipeline” at once
- Task parallelism – assigning different processors to different conceptual tasks, such as a climate simulation that separates the atmospheric and oceanic models
- Task farms – running many similar copies of a task for later combining, as in Monte Carlo simulation or testing design alternatives
- Speculative parallelism – trying alternatives that may not be necessary to pre-compute possible answers, as in searching for the best move in a game tree

However, doing this may conflict with traditional sequential thinking. For example, a speculative program may do more total computation than its sequential counterpart, albeit more quickly. As another example, task-parallel programs may repeat calculations to avoid synchronizing and moving data. Moreover, some algorithms (such as depth-first search) are theoretically impossible to execute in parallel, because each step requires information from the one before it. These must be replaced with others (such as breadth-first search) that can run in parallel. Other algorithms require unintuitive changes to execute in parallel. For example, consider the following program:

```
for i = 2 to N
  a[i] = a[i] + a[i-1]
end
```

One might think that this can only be computed sequentially. After all, every element $a[i]$ depends on the one computed before it. But this is not so. We leave it to other module-writers to explain how the elements $a[i]$ can all be computed in $\log(N)$ data-parallel steps; for now, we simply note that the parallel program is not a simple loop. In summary, parallel computing requires parallel thinking, and this is very different from the sequential thinking taught to our computer and computational scientists.

This brings us to the need for new, different, and hopefully better training in parallel computing. We must train new students to embrace parallel computing if the field is going to advance. Equally, we must retrain programmers working today if we want new programs to be better than today. To that end, the Open Education Cup is soliciting educational modules in five areas at the core of parallel computing:

- **Parallel Architectures:** Descriptions of parallel computers and how they operate, including particular systems (e.g. multicore chips) and general design principles (e.g. building interconnection networks). More information about this area is in the Parallel Architectures section (Section 1: Parallel Architectures).
- **Parallel Programming Models and Languages:** Abstractions of parallel computers useful for programming them efficiently, including both machine models (e.g. PRAM or LogP) and languages or extensions (e.g. OpenMP or MPI). More information about this area is in the Parallel Programming Models and Languages section (Section 2: Parallel Programming Models and Languages).
- **Parallel Algorithms and Applications:** Methods of solving problems on parallel computers, from basic computations (e.g. parallel FFT or sorting algorithms) to full systems (e.g. parallel databases or seismic processing). More information about this area is in the Parallel Algorithms and Applications section (Section 3: Parallel Algorithms and Applications).
- **Parallel Software Tools:** Tools for understanding and improving parallel programs, including parallel debuggers (e.g. TotalView) and performance analyzers (e.g. HPCtoolkit and TAU Performance System). More information about this area is in the Parallel Software Tools section (Section 4: Parallel Software Tools).
- **Accelerated Computing:** Hardware and associated software can add parallel performance to more conventional systems, from Graphics Processing Units (GPUs) to Field Programmable Gate Arrays (FPGAs) to Application-Specific Integrated Circuits (ASICs) to innovative special-purpose hardware.

More information about this area is in the Accelerated Computing section (Section 5: Accelerated Computing).

Entries that bridge the gap between two areas are welcome. For example, a module about a particular accelerated computing system might describe its accompanying language, and thus fall into both the Parallel Programming Models and Languages and Accelerated Computing categories. However, such modules can only be entered in one category each. To be eligible for awards in multiple categories, the module would need to be divided into appropriate parts, each entered in one category.

1 Parallel Architectures

Today, nearly all computers have some parallel aspects. However, there are a variety of ways that processors can be organized into effective parallel systems. The classic classification of **parallel architectures** is *Flynn's taxonomy*[6] based on the number of distinct instruction and data streams supplied to the parallel processors.

- **Single Instruction, Single Data (SISD)** – These are sequential computers. This is the only class that is not a parallel computer. We include it only for completeness.
- **Multiple Instruction, Single Data (MISD)** – Several independent processors work on the same stream. Few computers of this type exist. Arguably the clearest examples are fault tolerant systems that replicate a computation, comparing answers to detect errors; the flight controller on the US Space Shuttle is based on such a design. Pipelined computations are also sometimes considered MISD. However, the data passed between stages of the pipeline has been changed, so the “single” data aspect is murky at best.
- **Single Instruction, Multiple Data (SIMD)** – A central controller sends the same stream of instructions to a set of identical processors, each of which operates on its own data. Additional control instructions move data or exclude unneeded processors. At a low level of modern architectures, this is often used to update arrays or large data structures. For example, GPUs typically get most of their speed from SIMD operation. Perhaps the most famous large-scale SIMD computer was the Thinking Machines CM-2 in the 1980s, which boasted up to 65,536 bit-serial processors.
- **Multiple Instruction, Multiple Data (MIMD)** – All processors execute their own instruction stream, each operating on its own data. Additional instructions are needed to synchronize and communicate between processors. Most computers sold as “parallel computers” today fall into this class. Examples include the supercomputers, servers, grid computers, and multicore chips described above.

Hierarchies are also possible. For example, a MIMD supercomputer might include SIMD chips as accelerators on each of its boards.

Beyond general class, many architectural decisions are critical in designing a parallel computer architecture. Two of the most important include:

- **Memory hierarchy and organization.** To reduce data access time, most modern computers use a hierarchy of caches to keep frequently-used data accessible. This becomes particularly important in parallel computers, where many processors mean even more accesses. Moreover, parallel computers must arrange for data to be shared between processors. **Shared memory architectures** do this by allowing multiple processors to access the same memory. **Nonshared memory architectures** allot each processor its own memory, and require explicit communication to move data to another processor. A hybrid approach – non-uniform shared memory – places memory with each processor for fast access, but allows slower access to other processors’ memory.
- **Interconnection topology.** To communicate and synchronize, processors in a parallel computer need a connection with each other. However, as a practical matter not all of these can be direct connections. Thus is born the need for interconnection networks. For example, interconnects in use today include simple buses, crossbar switches, token rings, fat trees, and 2- and 3-D torus topologies. At the same

time, the underlying technology or system environment may affect the networks that are feasible. For example, grid computing systems typically have to accept the wide-area network that they have available.

All of these considerations (and more) can be formalized, quantified, and studied.

The Open Education Cup will accept entries relevant to any of the above architectures (except SISD). This might include case studies of parallel computers, comparisons of architectural approaches, design studies for future systems, or parallel computing hardware issues that we do not touch on here.

2 Parallel Programming Models and Languages

Parallel computers require software in order to produce useful results. Writing that software requires a means of expressing it (that is, a language), which must in turn be based on an idea of how it will work (that is, a model). While it is possible to write programs specific to a given parallel architecture, many would prefer a more general model and language, just as most sequential programmers use a general language rather than working directly with assembly code.

Unfortunately, there is no single widely-accepted model of parallel computation comparable to the sequential Random Access Machine (RAM) model. In part, this reflects the diversity of parallel computer architectures and their resulting range of operations and costs. The following list gives just a few examples of the models that have been suggested.

- *Communicating Sequential Processes (CSP)*[10]. This is a theoretical model of concurrent (that is, parallel) processes interacting solely through explicit messages. In that way, it is a close model of **nonshared memory architectures**. As the *CSP model has developed*[11], however, its use has emphasized validation of system correctness rather than development of algorithms and programs.
- *Parallel Random Access Machine (PRAM)*[7]. In this model, all processors operate asynchronously and have constant-time access to shared memory. This is similar to a **shared memory architecture**, and is therefore useful in describing *parallel algorithms*[12] for such machines. However, PRAM abstracts actual shared memory architectures by assuming that all memory can be accessed at equal cost, which is generally not true in practice.
- *Bulk Synchronous Processes (BSP)*[25]. In this model, all processors operate asynchronously and have direct access to only their own memory. Algorithms proceed in a series of “supersteps” consisting of local computation, communication exchanges, and barrier synchronizations (where processors wait at the barrier for all others to arrive). This can model either shared or non-shared memory **MIMD architectures**, but simplifies many issues in organizing the communication and synchronization.
- *LogP (for Latency, overhead, gap, Processors)*[5]. This model can be thought of as a refinement of BSP. LogP allows a more detailed treatment of architectural constraints such as interconnect network capacity. It also allows more detailed scheduling of communication, including overlap of computation with communication. LogP can model shared- and non-shared memory MIMD machines, but abstracts the specific topology of the network. (The capitalization is a pun on $O(\log P)$ theoretical bounds).

Programming languages and systems for parallel computers are equally diverse, as the following list shows.

- Libraries can encapsulate parallel operations. Libraries of synchronization and communication primitives are especially useful. For example, *the MPI library*[16] provides the send and receive operations needed for a CSP-like message passing model. When such libraries are called from sequential languages, there must be a convention for starting parallel operations. The most common way to do this is the Single Program Multiple Data (SPMD) paradigm, in which all processors execute the same program text.
- Extensions to sequential languages can follow a particular model of parallel computation. For example, *OpenMP*[19] reflects a PRAM-like shared memory model. *High Performance Fortran (HPF)*[8] (see also[13]) was based on a data-parallel model popularized by *CM Fortran*[21]. Common extensions include parallel loops (often called “**forall**” to evoke “**for**” loops) and synchronization operations.

- Entire new languages can incorporate parallel execution at a deep level, in forms not directly related to the programming models mentioned above. For example, *Cilk*[2] is a functional language that expresses parallel operations by independent function calls. *Sisal*[15] was based on the concept of streams, in the elements in a series (stream) of data could be processed independently. Both languages have been successfully implemented on a variety of platforms, showing the value of a non-hardware-specific abstraction.
- Other languages more directly reflect a parallel architecture, or a class of such architectures. For example, Partitioned Global Address Space (PGAS) languages like *Co-Array Fortran*[17], *Chapel*[3], *Fortress*[1], and *X10*[22] consider memory to be shared, but each processor can access its own memory much faster than other processors' memory. This is similar to many non-uniform shared memory architectures in use today. *CUDA*[18] is a rather different parallel language designed for programming on GPUs. It features explicit partitioning of the computation between the host (i.e. the controller) and the "device" (i.e. GPU processors). Although these languages are to some extent hardware-based, they are general enough that implementations on other platforms are possible.

Neither of the lists above is in any way exhaustive. The Open Education Cup welcomes entries about any aspect of expressing parallel computation. This includes descriptions of parallel models, translations between models, descriptions of parallel languages or programming systems, implementations of the languages, and evaluations of models or systems.

3 Parallel Algorithms and Applications

Unlike performance improvements due to increased clock speed or better compilers, running faster on parallel architectures doesn't just happen. Instead, **parallel algorithms** have to be devised to take advantage of multiple processors, and applications have to be updated to use those algorithms. The methods (and difficulty) of doing this vary widely. A few examples show the range of possibilities.

Some theoretical work has taken a data-parallel-like approach to designing algorithms, allowing the number of processors to increase with the size of the data. For example, consider summing the elements of an array. Sequentially, we would write this as

```
x = 0
for i = 1 to n
  x = x + a[i]
end
```

Sequentially, this would run in $O(n)$ time. To run this in parallel, consider n processes, numbered 1 to n , each containing original element $a[i]$. We might then perform the computation as a *data parallel tree sum*[9], with each node at the same level of the tree operating in parallel.²

```
step = 1
forall i = 1 to n do
  xtmp[i] = a[i]
end
while step < n do
  forall i = 1 to n-step by 2*step do
    xtmp[i] = xtmp[i] + xtmp[i+step]
  end
  step = step * 2
```

²We use "forall" to denote parallel execution in all examples, although there are many other constructs.

```

end
x = xtmp[1]

```

This takes $O(\log n)$ time, which is optimal in parallel. Many such algorithms and bounds are known, and classes (analogous to P and NP) can be constructed describing “solvable” parallel problems.

Other algorithmic work has concentrated on mapping algorithms to more limited set of parallel processors. For example, the above algorithm might be *mapped onto p processors*[12] in the following way.

```

forall j = 1 to p do
  lo[j] = 1+(j-1)*n/p
  hi[j] = j*n/p
  xtmp[j] = 0
  for i = lo[j] to hi[j] do
    xtmp[j] = xtmp[j] + a[i]
  end
  do
    if j=1
    then step = 1
    barrier_wait()
    while step[j] < n do
      if j+step[j]<p and j mod (step[j]*2) = 1
      then xtmp[j] = xtmp[j] + xtmp[j+step[j]]
    end
    step[j] = step[j] * 2
    barrier_wait()
  end
end
x = xtmp[1]

```

Note that the barrier synchronizations are necessary to ensure that no processor j runs ahead of the others, thus causing some updates of $xtmp[j]$ to use the wrong data values. The time for this algorithm is $O(n/p + \log p)$. This is optimal for operation count, but not necessarily for number of synchronizations.

Many other aspects of parallel algorithms deserve study. For example, the design of algorithms for other important problems is a vast subject, as is describing general families of parallel algorithms. Analyzing algorithms with respect to time, memory, parallel overhead, locality, and many other measures is important in practice. Practical implementation of algorithms, and measuring those implementations, is particularly useful in the real world. Describing the structure of a full parallel application provides an excellent guidepost for future developers. The Open Education Cup invites entries on all these aspects of parallel algorithms and applications, as well as other relevant topics.

4 Parallel Software Tools

Creating a **parallel application** is complex, but developing a correct and efficient parallel program is even harder. We mention just a few of the difficulties in the following list.

- All the bugs that can occur in sequential programming – such as logic errors, dangling pointers, and memory leaks – can also occur in parallel programming. Their effects may be magnified, however. As one architect of a grid computing system put it, “Now, when we talk about global memory, we really mean **global**.”
- Missing or misplaced synchronization operations are possible on all **MIMD** architectures. Often, such errors lead to deadlock, the condition where a set of processors are forever waiting for each other.

- Improper use of synchronization can allow one processor to read data before another has initialized it. Similarly, poor synchronization can allow two processors to update the same data in the wrong order. Such situations are called "race conditions", since the processors are racing to access the data.
- Even with proper synchronization, poor scheduling can prevent a given processor from ever accessing a resource that it needs. Such a situation is called starvation.
- Correct programs may still perform poorly because one processor has much more work than others. The time for the overall application then depends on the slowest processor. This condition is called load imbalance.
- Processors may demand more out of a shared resource, such as a memory bank or the interconnection network, than it can provide. This forces some requests, and thus the processors that issued them, to be delayed. This situation is called contention.
- Perhaps worst of all, the timing between MIMD processors may vary from execution to execution due to outside events. For example, on a shared machine there might be additional load on the interconnection network from other users. When this happens, any of the effects above may change from run to run of the program. Some executions may complete with no errors, while others produce incorrect results or never complete. Such "Heisenbugs" (named for Werner Heisenberg, the discoverer of the uncertainty principle in quantum mechanics) are notoriously difficult to find and fix.

Software tools like debuggers and profilers have proved their worth in helping write sequential programs. In parallel computing, tools are at least as important. Unfortunately, parallel software tools are newer and therefore less polished than their sequential versions. They also must solve some additional problems, beyond simply dealing with the new issues noted above.

- Parallel tools must handle multiple processors. For example, where a sequential debugger sets a checkpoint, a parallel debugger may need to set that checkpoint on many processors.
- Parallel tools must handle large data. For example, where a sequential trace facility may produce megabytes of data, the parallel trace may produce megabytes for each of hundreds of processors, adding up to gigabytes.
- Parallel tools must be scalable. Just as some bugs do not appear in sequential programs until they are run with massive data sets, some problems in parallel programs do not appear until they execute on thousands of processors.
- Parallel tools must avoid information overload. For example, where a sequential debugger may only need to display a single line number, its parallel counterpart may need to show line numbers on dozens of processors.
- Parallel tools must deal with timing and uncertainty. While it is rare for a sequential program's behavior to depend on time, this is the common case for parallel programs.

Current parallel tools - for example *Cray Apprentice2*[4], *HPCToolkit*[20], *TAU*[23], *TotalView*[24], and *VTune*[14] - have solved some of these problems. For example, data visualization has been successful in avoiding information overload and understanding large data sets. However, other issues remain difficult research problems.

The Open Education Cup welcomes entries related to the theory and practice of parallel software tools. This includes descriptions of existing debuggers, profilers, and other tools; analysis and visualization techniques for parallel programs and data; experiences with parallel tools; and any other topic of interest to tool designers or users.

5 Accelerated Computing

Accelerated computing is a form of **parallel computing** that is rapidly gaining popularity. For many years, some general-purpose computer systems have included accelerator chips to speed up specialized tasks. For example, early microprocessors did not implement floating-point operations directly, so machines in the technical market often included floating-point accelerator chips. Today, microprocessors are much more capable, but some accelerators are still useful. The list below suggests a few of them.

- Graphics Processing Units (GPUs) provide primitives commonly used in graphics rendering. Among the operations sped up by these chips are texture mapping, geometric transformations, and shading. The key to accelerating all of these operations is parallel computing, often realized by computing all pixels of a display or all objects in a list independently.
- Field Programmable Gate Arrays (FPGAs) provide many logic blocks linked by an interconnection fabric. The interconnections can be reconfigured dynamically, thus allowing the hardware datapaths to be optimized for a given application or algorithm. When the logic blocks are full processors, the FPGA can be used as a parallel computer.
- Application Specific Integrated Circuits (ASICs) are chip designs optimized for a special use. ASIC designs can now incorporate several full processors, memory, and other large components. This allows a single ASIC to be a parallel system for running a particular algorithm (or family of related algorithms).
- Digital Signal Processor (DSP) chips are microprocessors specifically designed for signal processing applications such as sensor systems. Many of these applications are very sensitive to latency, so performance of computation and data transfer is heavily optimized. DSPs are able to do this by incorporating **SIMD parallelism** at the instruction level and pipelining of arithmetic units.
- Cell Broadband Engine Architecture (CBEA, or simply Cell) is a relatively new architecture containing a general-purpose computer and several streamlined coprocessors on a single chip. By exploiting the **MIMD parallelism** of the coprocessors and overlapping memory operations with computations, the Cell can achieve impressive performance on many codes. This makes it an attractive adjunct to even very capable systems.

As the list above shows, accelerators are now themselves parallel systems. They can also be seen as a new level in hierarchical machines, where they operate in parallel with the host processors. A few examples illustrate the possibilities.

- General Purpose computing on GPUs (usually abbreviated as GPGPU) harnesses the computational power of GPUs to perform non-graphics calculations. Because many GPU operations are based on matrix and vector operations, this is a particularly good match with linear algebra-based algorithms.
- Reconfigurable Computing uses FPGAs adapt high-speed hardware operations for the needs of an application. As the application goes through phases, the FPGA can be reconfigured to accelerate each phase in turn.
- The Roadrunner supercomputer gets most of its record-setting performance from Cell processors used as accelerators on its processing boards.

The Open Education Cup welcomes entries describing any aspect of accelerated computing in parallel systems. We are particularly interested in descriptions of systems with parallel accelerator components, experience with programming and running these systems, and software designs that automatically exploit parallel accelerators.

Glossary

Definition 1: MIMD

Multiple Instruction Multiple Data; a type of parallel computer in which the processors run independently, possibly performing completely different operations, each on its own data.

Definition 2: nonshared memory

a type of parallel computer in which each processor can directly access only its own section of memory; data to be shared with other processors must be explicitly copied to other memory areas.

Definition 3: parallel

Performing more than one operation at a time; (of computers) having more than one processing unit.

Example

"The parallel computer had 1024 CPU chips, each with 2 processor cores, allowing 2048 simultaneous additions."

Definition 4: shared memory

a type of parallel computer in which all processors can access a common area of memory.

Definition 5: SIMD

Single Instruction Multiple Data; a type of parallel computer in which all processors execute the same instruction simultaneously, each on its own data.

References

- [1] Eric Allen et al. *The Fortress Language Specification, Version 1.0*. Sun Microsystems, Inc., 2008. Available at <http://research.sun.com/projects/plrg/fortress.pdf>.
- [2] Robert D. Blumofe et al. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 2078211;216, 1995.
- [3] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
- [4] Cray, Inc. *Cray Apprentice2 for Cray XT3 systems 3.1 Man Pages*, 2006. Available at http://docs.cray.com/cgi-bin/craydoc.cgi?mode=Search;newest=0;sw_releases=sw_releases-0q550xx9-1149598079;this_sort=title;sq=doc_type%3dman;browse=1.
- [5] David Culler et al. Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–12, 1993.
- [6] Michael Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948, 1972.
- [7] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the ACM Symposium on Theory of Computing (SToC)*, pages 114–118, 1978.
- [8] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*, 1997. Available at <http://hpff.rice.edu/versions/hpf2/hpf-v20.pdf>.
- [9] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170 – 1183, 1986.
- [10] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):6668211;677, 1978.
- [11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. Available at <http://www.usingcsp.com/>.
- [12] J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
- [13] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran: An historical object lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL-III)*, pages 7–1 – 7–22, 2007.

- [14] David Levinthal. *Introduction to Performance Analysis on Intel Core8482; 2 Duo Processors*. Intel Software and Solutions Group, 2006. Available at http://softwarecommunity.intel.com/isn/downloads/softwareproducts/pdfs/performance_analysis.pdf.
- [15] J. R. McGraw et al. *Sisal: Streams and iterations in a single-assignment language, Language Reference Manual, Version 1.1*. Lawrence Livermore National Laboratory, Livermore, CA, 1983.
- [16] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.1*, 2008. Available at <http://www.mpi-forum.org/docs/mpi21-report.pdf>.
- [17] Robert Numrich and Brian Reid. Co-arrays in the next fortran standard. *ACM Fortran Forum*, 24(2):4–17, 2005.
- [18] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 2.0*, 2008. Available at http://developer.download.nvidia.com/compute/cuda/2_0.
- [19] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*, 2008. Available at <http://www.openmp.org/mp-documents/spec30.pdf>.
- [20] Apan Qasem, Ken Kennedy, and John Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *The Journal of Supercomputing*, 36(9):183–196, 2006.
- [21] Gary W. Sabot. Optimizing cm fortran compiler for connection machine computers. *Journal of Parallel and Distributed Computing*, 23(2):224 – 238, 1994.
- [22] Vijay Saraswat and Nathaniel Nystrom. *Report on the Experimental Language X10, Version 1.7*. Sun Microsystems, Inc., 2008. Available at <http://dist.codehaus.org/x10/documentation/languagespec/x10-170.pdf>.
- [23] S. Shende and A. D. Malony. Tau: The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.
- [24] TotalView Technologies. *TotalView Documentation*, 2008. Available at <http://www.totalviewtech.com/support/documentation/totalview/>.
- [25] Leslie Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.