Chapter 5

# PARALLEL ALGORITHM DESIGN FOR BRANCH AND BOUND

David A. Bader
*Department of Electrical & Computer Engineering, University of New Mexico*
dbader@ece.unm.edu

William E. Hart
*Discrete Mathematics and Algorithms Department, Sandia National Laboratories*
wehart@sandia.gov

Cynthia A. Phillips
*Discrete Mathematics and Algorithms Department, Sandia National Laboratories*
caphill@sandia.gov

**Abstract** Large and/or computationally expensive optimization problems sometimes require parallel or high-performance computing systems to achieve reasonable running times. This chapter gives an introduction to parallel computing for those familiar with serial optimization. We present techniques to assist the porting of serial optimization codes to parallel systems and discuss more fundamentally parallel approaches to optimization. We survey the state-of-the-art in distributed- and shared-memory architectures and give an overview of the programming models appropriate for efficient algorithms on these platforms. As concrete examples, we discuss the design of parallel branch-and-bound algorithms for mixed-integer programming on a distributed-memory system, quadratic assignment problem on a grid architecture, and maximum parsimony in evolutionary trees on a shared-memory system.

**Keywords:** parallel algorithms; optimization; branch and bound; distributed memory; shared memory; grid computing

## Introduction

Although parallel computing is often considered synonymous with supercomputing, the increased availability of multi-processor workstations and Beowulf-style clusters has made parallel computing resources available to most academic departments and modest-scale companies. Consequently, researchers have applied parallel computers to problems such as weather and climate modeling, bioinformatics analysis, logistics and transportation, and engineering design. Furthermore, commercial applications are driving development of effective parallel software for large-scale applications such as data mining and computational medicine.

In the simplest sense, parallel computing involves the simultaneous use of multiple compute resources to solve a computational problem. However, the choice of target compute platform often significantly influences the structure and performance of a parallel computation. There are two main properties that classify parallel compute platforms: physical proximity of compute resources and distribution of memory. In *tightly-coupled* parallel computers, the processors are physically co-located and typically have a fast communication network. In *loosely coupled* compute platforms, compute resources are distributed, and consequently inter-process communication is often slow. In *shared memory* systems, all of the RAM is physically shared. In *distributed memory* systems each processor/node controls its own RAM. The owner is the only processor that can access that RAM.

The term *system* usually refers to tightly-coupled architectures. *Shared-memory* systems typically consist of a single computer with multiple processors using many simultaneous asynchronous threads of execution. Most massively-parallel machines are *distributed memory* systems. Parallel software for these machines typically requires explicit problem decomposition across the processors, and the fast communication network enables synchronous inter-processor communication. *Grid* compute platforms exemplify the extreme of loosely-coupled distributed-memory compute platforms. Grid compute platforms may integrate compute resources across extended physical distances, and thus asynchronous, parallel decomposition is best suited for these platforms. Loosely-coupled shared-memory platforms have not proven successful because of the inherent communication delays in these architectures.

This chapter illustrates how to develop scientific parallel software for each of these three types of canonical parallel compute platforms. The *programming model* for developing parallel software is somewhat different for each of these compute platforms. Skillicorn and Talia [103] define a programming model as: “an abstract machine providing certain opera-

tions to the programming level above and requiring implementations on all of the architectures below." Parallel code will perform best if the programming model and the hardware match. However, in general parallel systems can emulate the others (provide the other required abstractions) with some loss of performance.

As a concrete example, we consider the design of parallel branch and bound. We discuss how the compute platform influences the design of parallel branch and bound by affecting factors like task decomposition and inter-processor coordination. We discuss the application of parallel branch and bound to three real-world problems that illustrate the impact of the parallelization: solving mixed-integer programs, solving quadratic assignment problems, and reconstructing evolutionary trees.

Finally, we provide some broad guidance on how to design and debug parallel scientific software. We survey some parallel algorithmic primitives that form the basic steps of many parallel codes. As with serial software, a working knowledge of these types of primitives is essential for effective parallel software development. Because parallel software is notoriously difficult to debug, we also discuss practical strategies for debugging parallel codes.

## 5.1 Parallel Computing Systems

Over the past two decades, high-performance computing systems have evolved from special-purpose prototypes into commercially-available commodity systems for general-purpose computing. We loosely categorize parallel architectures as distributed memory, shared memory, or grid; realizing that modern systems may comprise features from several of these classes (for example, a cluster of symmetric multiprocessors, or a computational grid of multithreaded and distributed memory resources). In this section, we briefly describe some theoretical parallel models, the types of high-performance computing architectures available today, and the programming models that facilitate efficient implementations for each platform. For a more details, we refer the reader to a number of excellent resources on parallel computing [32, 33, 49, 58], parallel programming [13, 24, 93, 104, 115], and parallel algorithms [57, 77, 97].

### 5.1.1 Theoretical Models of Parallel Computers

Theoretical analysis of parallel algorithms requires an abstract machine model. The two primary models roughly abstract shared-memory and distributed-memory systems. For serial computations the RAM model, perhaps augmented with a memory hierarchy, is universally accepted. We are aware of no single model that is both realistic enough

that theoretical comparisons carry over to practice and simple enough to allow clean and powerful analyses. However, some techniques from theoretically good algorithms are useful in practice. If the reader wishes to survey the parallel algorithms literature before beginning an implementation, he will need some familiarity with parallel machine models to understand asymptotic running times.

The *Parallel Random Access Machine* (PRAM) [42] has a set of identical processors and a shared memory. At each synchronized step, the processors perform local computation and simultaneously access the shared memory in a legal pattern. In the EREW (exclusive-read exclusive-write) PRAM the access is legal if all processors access unique memory locations. The CRCW (concurrent-read concurrent write) PRAM allows arbitrary access and the CREW PRAM allows only simultaneous reads. The PRAM roughly abstracts shared-memory systems. In Section 5.1.4 we show ways in which it is still unrealistic, frequently fatally so.

The asynchronous LogP model is a reasonable abstraction of distributed-memory systems. It explicitly models communication bandwidth (how much data a processor can exchange with the network), message latency (time to send a message point to point) and how well a system overlaps communication and computation. It can be difficult to compare different algorithms using this model because running times can be complicated functions of the various parameters. The Bulk Synchronous Parallel (BSP) model[113] is somewhat between LogP and PRAM. Processors have local memory and communicate with messages, but have frequent explicit synchronizations.

### 5.1.2 Distributed Memory Architectures

**5.1.2.1 Overview.** In a distributed-memory system each node is a workstation or PC-level processor, possibly even an SMP (see Section 5.1.4). Each node runs its own operating system, controls its own local memory, and is connected to the other processors via a communication network. In this section, we consider distributed-memory *systems* where the processors work together as a single tightly-coupled machine. Such a machine usually has a scheduler that allocates a subset of the nodes on a machine to each user request. However, most of the discussion and application examples for distributed-memory systems also apply to independent workstations on a local-area network, provided the user has access privileges to all the workstations in the computation.

The number of processors and interconnect network topology varies widely among systems. The ASCI Red Storm supercomputer, that is being built by Cray for Sandia National Laboratories, will have 10,368

AMD Opteron processors connected via a three-dimensional mesh with some toroidal wraps. The Earth Simulator at the Earth Simulator Center in Japan has 640 nodes, each an 8-processor NEC vector machine, connected via a crossbar. The networks in both these supercomputers use custom technology. Commercial (monolithic) systems include IBM SP and Blades, Apple G5, Cray systems, and Intel Xeon clusters. However, any department/company can build a Beowulf cluster by buying as many processors as they can afford and linking them using commercial network technology such as Ethernet, Myrinet, Quadrics, or InfiniBand. These can be quite powerful. For example, the University of New Mexico runs IBM's first Linux supercluster, a 512-processor cluster with Myrinet (LosLobos), allocated to National Science Foundation users, and the Heidelberg Linux Cluster System (HELICS) has 512 AMD Athlon PC processors connected as a Clos network with commercial Myrinet technology. Even at the low end with only a few processors connected by Ethernet, one can benefit from this form of parallelism.

Because distributed-memory systems can have far more processors and total memory than the shared-memory systems discussed in Section 5.1.4, they are well suited for applications with extremely large problem instances.

#### 5.1.2.2 Programming Models for Distributed-Memory.

One programs a distributed-memory machine using a standard high-level language such as C++ or Fortran with explicit message passing. There are two standard Application Programming Interfaces (APIs) for message passing: Message-Passing Interface (MPI [104]) and Parallel Virtual Machine (PVM [92]). MPI is the standard for tightly-coupled large-scale parallel machines because it is more efficient. PVM's extra capabilities to handle heterogeneous and faulty processors are more important for the grid than in this setting.

For either API, the user can assume some message-passing primitives. Each processor has a unique rank or ID, perhaps within a subset of the system, which serves as an address for messages. A processor can send a message directly to another specifically-named processor in a *point-to-point* message. A *broadcast* sends a message from a single source processor to all other processors. In an *all-to-all* message, each of the $P$ processors sends $k$ bytes to all other processors. After the message, each processor has $kP$ bytes with the information from processor $i$ in the $i$th block of size $k$. A *reduction* operation (see Section 5.3.1) takes a value from each processor, computes a function on all the values, and gives the result to all processors. For example, a sum reduction gives every processor the sum of all the input values across all processors.

The performance of a distributed-memory application depends critically upon the message complexity: number of messages, message size, and *contention* (number of messages simultaneously competing for access to an individual processor or channel). The bandwidth and speed of the interconnection network determines the total amount of message traffic an application can tolerate. Thus a single program may perform differently on different systems. When a parallel algorithm such as branch and bound has many small independent jobs, the user can tailor the *granularity* of the computation (the amount of work grouped into a single unit) to set the communication to a level the system can support. Some unavoidable communication is necessary for the correctness of a computation. This the user cannot control as easily. Thus a large complex code may require clever problem-dependent message management.

The amount of contention for resources depends in part upon *data layout*, that is, which processor owns what data. The programmer also explicitly manages data distribution. If shared data is static, meaning it does not change during a computation, and the data set is not too large relative to the size of a single processor's memory, it should be *replicated* on all processors. Data replication also works well for data that is widely shared but rarely changed, especially if the computation can tolerate an out-of-date value. Any processor making a change broadcasts the new value. In general, data can *migrate* to another processor as a computation evolves, either for load balancing or because the need to access that data changes. Automated graph-partition-based tools such as Chaco [55] and (Par)METIS [60] assist in initial data assignment if the communication pattern is predictable.

### 5.1.3 Grid Computing

**5.1.3.1 Overview.** *Grid computing* (or "metacomputing") generally describes parallel computations on a geographically-distributed, heterogeneous platform [45, 44]. Grid computing technologies enable coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations. Specifically, these tools enable direct access to computers, software, data, and other resources (as opposed to sharing via file exchange). A virtual organization is defined by a set of sharing rules between individuals and/or organizations. These sharing rules define how resource providers coordinate with consumers, what is shared, who can share, and the conditions under which sharing occurs.

Grid computing platforms for scientific applications can use shared workstations, nodes of PC clusters, and supercomputers. Although grid computing methodologies can effectively connect supercomputers,

a clear motivation for grid computing is the wide availability of idle compute cycles in personal computers. Large institutions can have hundreds or thousands of computers that are often idle. Consequently, grid computing can enable large-scale scientific computation using existing computational resources without significant additional investment.

Projects like Condor [70], Legion [50] and Globus [43] provide the underlying infrastructure for resource management to support grid computing. These toolkits provide components that define a protocol for interacting with remote resources and an application program interface to invoke that protocol. Higher-level libraries, services, tools and applications can use these components to implement more complex global functionality. For example, various Globus Toolkit components are reviewed by Foster et al. [45]. Many grid services build on the Globus Connectivity and Resource protocols, such as (1) directory services that discover the existence and/or properties of resources, (2) scheduling and brokering services that request the scheduling and allocation of resources, and (3) monitoring and diagnostic services that provide information about the availability and utilization of resources.

Large-scale Grid deployments include: "Data Grid" projects like EU DataGrid (`www.eu-datagrid.org`), the DOE Science Grid (`http://www.doesciencegrid.org`), NASAs Information Power Grid (`www.ipg.nasa.gov`), the Distributed ASCI Supercomputer (DAS-2) system (`www.cs.vu.nl/das2/`), the DISCOM Grid that links DOE laboratories (`www.cs.sandia.gov/discom/`), and the TeraGrid under construction to link major U.S. academic sites (`www.teragrid.org`). These systems integrate resources from multiple institutions using open, general-purpose protocols like the Globus Toolkit.

**5.1.3.2 Programming Model.** Grid computing platforms differ from conventional multiprocessor supercomputers and from Linux clusters in several important respects.

- Interprocessor Connectivity: Communication latency (time between message initiation and message delivery) is generally much higher on grid platforms than on tightly-coupled supercomputers and even on Linux clusters. Grid communication latencies are often highly variable and unpredictable.

- Processor Reliability and Availability: Computational resources may disappear without notice in some grid computing frameworks. For example, jobs running on PCs may terminate or halt indefinitely while the PC is used interactively. Similarly, new compute resources may become available during a parallel computation.

- Processor Heterogeneity: Computational resources may vary in their operational characteristics, such as available memory, swap space, processor speed, and operating system (type and version).

It is challenging to divide and robustly coordinate parallel computation across heterogeneous and/or unreliable resources. When resources reside in different administrative domains, they may be subject to different access control policies, and be connected by networks with widely varying performance characteristics.

For simple applications augmented distributed-memory programming models may suffice. For example, MPICH-G2 generalizes MPI to provide a low-level grid programming model that manages heterogeneity directly [59]. Since MPI has proven so effective for tightly-coupled distributed computing environments, MPICH-G2 provides a straightforward way to adapt existing parallel software to grid computing environments. MPICH-G2 is integrated with the Globus Toolkit, which manages the allocation and coordination of grid resources.

Although MPICH-G2 can adapt MPI codes for grid computing environments, the programmer must generally modify the original code for robustness. For example, the new code must monitor node status and reassign work given to killed nodes. Synchronous codes must be made asynchronous for both correctness and performance. When the number of processors vary, it's difficult to tell when all processors have participated in a synchronous step. Since some processor is likely to participate slowly or not at all, each synchronization can be intolerably slow.

A variety of frameworks can facilitate development of grid computing applications. NetSolve [23] provides an API to access and schedule Grid resources in a seamless way, particularly for embarrassingly parallel applications. The Everyware toolkit [117] draws computational resources transparently from the Grid, though it is not abstracted as a programming tool. The MW and CARMI/Wodi tools provide interfaces for programming master-worker applications [48, 90]. These frameworks provide mechanisms for allocating, scheduling computational resources, and for monitoring the progress of remote jobs.

### 5.1.4 Shared Memory Systems

**5.1.4.1 Overview.** Shared-memory systems, often called symmetric multiprocessors (SMPs), contain from two to hundreds of microprocessors tightly coupled to a shared memory subsystem, running under a single system image of one operating system. For instance, the IBM "Regatta" p690, Sun Fire E15K, and SGI Origin, all scale from dozens to hundreds of processors in shared-memory images with near-

uniform memory access. In addition to a growing number of processors in a single shared memory system, we anticipate the next generation of microprocessors will be "SMPs-on-a-chip". For example, uniprocessors such as the IBM Power4 using simultaneous multithreading (SMT), Sun UltraSparc IV, and Intel Pentium 4 using Hyper-Threading each act like a dual-processor SMP. Future processor generations are likely to have four to eight cores on a single silicon chip. Over the next five to ten years, SMPs will likely become the standard workstation for engineering and scientific applications, while clusters of very large SMPs (with hundreds of multi-core processors) will likely provide the backbone of high-end computing systems.

Since an SMP is a true (hardware-based) shared-memory machine, it allows the programmer to share data structures and information at a fine grain at memory speeds. An SMP processor can access a shared memory location up to two orders of magnitude faster than a processor can access (via a message) a remote location in a distributed memory system. Because processors all access the same data structures (same physical memory), there is no need to explicitly manage data distribution. Computations can naturally synchronize on data structure states, so shared-memory implementations need fewer explicit synchronizations in some contexts. These issues are especially important for irregular applications with unpredictable execution traces and data localities, often characteristics of combinatorial optimization problems, security applications, and emerging computational problems in biology and genomics.

While an SMP is a shared-memory architecture, it is by no means the Parallel Random Access Memory (PRAM) model (see [57, 97]) used in theoretical work. Several important differences are: (i) the number of processors in real SMP systems remains quite low compared to the polynomial number of processors assumed by theoretic models; (ii) real SMP systems have no lockstep synchronization (PRAMs assume perfect synchronization at the fetch-execute-store level); (iii) SMP memory bandwidth is limited; (iv) real SMPs have caches that require a high degree of spatial and temporal locality for good performance; and (v) these SMP caches must be kept *coherent*. That is, when a processor reads a value of a memory location from its local cache, this value must correspond to the value last written to that location by any processor in the system. For example, a 4-way or 8-way SMP cannot support concurrent read to the same location by a thousand threads without significant slowdown.

The memory hierarchy of a large SMP is typically quite deep, since the main memory can be so large. Thus cache-friendly algorithms and implementations are even more important on large-scale SMPs than on workstations. Very small-scale SMPs maintain cache coherence by a

*snoopy protocol* with substantial hardware assistance. In this protocol, whenever a processor writes a memory location, it broadcasts the write on a shared bus. All other processors monitor these broadcasts. If a processor sees an update to a memory location in its cache, it either updates or invalidates that entry. In the latter case, it's not just a single value that's invalidated, but an entire cache line. This can lead to considerable slowdowns and memory congestion for codes with little spatial locality. In the *directory-based* protocol, the operating system records the caches containing each cache line. When a location in the line changes, the operating system invalidates that line for all non-writing processors. This requires less bandwidth than snooping and scales better for larger SMP systems, but it can be slower. *False sharing* occurs when two unrelated data items $a$ and $b$ are grouped in a single cache line. Thus a processor that has the line in its cache to access item $a$ may have the entire line invalidated for an update to item $b$, even though the processor never accesses $b$. Eliminating false sharing to improve performance must currently be handled by the compiler or the programmer.

Finally, synchronization is perhaps the biggest obstacle to the correct and efficient implementation of parallel algorithms on shared-memory machines. A theoretic algorithm may assume lockstep execution across all processors down to the level of the fetch-execute-store cycle; for instance, if processor $i$ is to shift the contents of location $i$ of an array into location $i+1$, each processor reads its array value in lockstep, then stores it in the new location in lockstep. In a real machine, some processors could easily start reading after their neighbor has already completed its task, resulting in errors. The programming solution is to introduce barriers at any point in the code where lack of synchronization could cause indeterminacy in the final answer. However, such a solution is expensive when implemented in software and, if needed on a finely-divided time scale will utterly dominate the running time.

**5.1.4.2 Programming Models.** Various programming models and abstractions are available for taking advantage of shared-memory architectures. At the lowest level, one can use libraries, such as POSIX threads (for example, see [62, 89, 108]) to explicitly handle threading of an application. Other standards such as OpenMP (see [24, 85, 93]) require compiler and runtime support. The programmer provides hints to the compiler on how to parallelize sections of a correct sequential implementation. Finally, new high-level languages such as Unified Parallel C (UPC) [21], a parallel extension of C, Co-Array Fortran (CAF) (see `www.co-array.org`) with global arrays, and Titanium [122], are also emerging as new, efficient programming models for multiprocessors.

POSIX threads (often referred to as "pthreads") are native threads of processing that run within a single process/application and can share access to resources and memory at a fine-scale. The programmer explicitly creates and manages threads, with each thread inheriting its parent's access to resources. The programmer can synchronize threads and protect critical sections, such as shared memory locations in data structures and access to I/O resources, via mutual exclusion (or "mutex") locks. These support three operations: lock, unlock, and try, a non-blocking version of lock where a thread either succeeds at acquiring the lock, or resumes execution without the lock. Condition variables suspend a thread until an event occurs that wakes up the thread. These in conjunction with mutex locks can create higher-level synchronization events such as shared-memory barriers. In a threaded code, the programmer can then rely on coherency protocols to update shared memory locations.

OpenMP is a higher-level abstraction for programming shared memory that makes use of compiler directives, runtime systems, and environment variables. The programmer often begins with a working sequential code in C, C++, or Fortran, and inserts directives into the code to guide the compiler and runtime support for parallelization. One can specify, for example, that a loop has no dependencies among its iterations and can be parallelized in a straightforward manner. One can also specify that a loop is a reduction or scan operation that can then be automatically parallelized (see Section 5.3). OpenMP also allows the programmer to mark critical sections and insert synchronization barriers. The programmer can specify how the work varies from iteration to iteration (for example if the work is constant, random, or dependent upon the loop iteration). These hints can improve the scheduling of loop iterations.

UPC [21] is an extension of C that provides a shared address space and a common syntax and semantics for explicitly parallel programming in C. UPC strikes a balance between ease-of-use and performance. The programming model for UPC assumes a number of threads, each with private or shared pointers that can point to local or global memory locations. UPC provides explicit synchronization including barriers. Unlike POSIX threads, UPC provides a library of collective communication routines commonly needed in scientific and technical codes. UPC is emerging as an alternative for parallel programming that builds upon prior languages such as AC and Split-C.

Shared-memory has enabled the high-performance implementation of parallel algorithms for several combinatorial problems that up to now have not had implementations that performed well on parallel systems for arbitrary inputs. We have released several such high-performance shared-memory codes for important problems such as list ranking and

sorting [53, 54], ear decomposition [6], spanning tree [4], minimum spanning tree [5], and Euler tour [29]. These parallel codes are freely-available under the GNU General Public License (GPL) from Bader's web site. They use a shared-memory framework for POSIX threads [7].

## 5.2 Application Examples

In this section, we describe basic branch and bound (B&B) and discuss issues of special consideration in parallelizing B&B applications. We then give three parallel B&B example applications: B&B for mixed-integer programming on a distributed-memory architecture, B&B for the quadratic assignment problem on a grid architecture, and B&B for phylogeny reconstruction on a shared-memory architecture.

### 5.2.1 Branch and Bound

Branch and bound is an intelligent search heuristic for finding a global optimum to problems of the form $\min_{x \in X} f(x)$. Here $X$ is the *feasible region* and $f(x)$ is the *objective function*. Basic B&B searches the feasible region by iteratively subdividing the feasible region and recursively searching each piece for an optimal feasible solution. B&B is often more efficient than straight enumeration because it can eliminate regions that provably do not contain an optimal solution.

To use B&B for a given problem, one must specify problem-specific implementations of the following procedures. The *bound* procedure gives a lower bound for a problem instance over any feasible region. That is, for instance $\mathcal{I}$ with feasible region $X_{\mathcal{I}}$, the bound procedure returns $b(\mathcal{I})$, such that for all $x \in X_{\mathcal{I}}$ we have $b(\mathcal{I}) \leq f(x)$. The *branch* or *split* procedure breaks the feasible region $X$ into $k$ subregions $X_1, X_2, \ldots, X_k$. In most efficient B&B implementations, subregions are disjoint ($X_i \cap X_j = \emptyset$ for $i \neq j$), though this need not be the case. The only requirement for correctness is that there exists an $x \in X$ with minimum value of $f(x)$ such that $x \in \bigcup_{i=1}^{k} X_i$ (if we require all optima, then this must be the case for all optimal $x$). Finally, a *candidate* procedure takes an instance of the problem (a description of $X$) and returns a feasible solution $x \in X$ if possible. This procedure can fail to return a feasible solution even if $X$ contains feasible solutions. However, if $X$ consists of a single point, then the candidate solution procedure must correctly determine the feasibility of this point. In general, one may have many candidate solution methods. At any point in a B&B search, the best feasible (candidate) solution found so far is called the *incumbent*, denoted $x_I$.

We now describe how to find a globally optimal solution using B&B given instantiations of these procedures. We grow a search tree with

the initial problem $r$ as the root. If the candidate procedure (called on the root) returns a feasible solution, it becomes the first incumbent. Otherwise, we start with no incumbent and define an incumbent value $f(x_I) = +\infty$. We bound the root, which yields a lower bound $b(r)$. If $b(r) = f(x_I)$, then the incumbent is an optimal solution and we are done. Otherwise, we split the root into $k$ subproblems and make each subproblem a child of the root. We process a child subproblem similarly. We bound the subproblem $c$ to obtain a bound $b(c)$. If $b(c) > f(x_I)$, then no feasible solution in subproblem $c$ can be better than the incumbent. Therefore, we can *fathom* subproblem $c$, meaning we eliminate it from further consideration. If candidate($c$) returns a solution $x$ such that $f(x) = b(c)$, then $x$ is an optimal solution for this subproblem. Subproblem $c$ becomes a leaf of the tree (no need to further subdivide it) and solution $x$ replaces the incumbent if $f(x) < f(x_I)$. Otherwise, we split subproblem $c$ and continue. Any subproblem that is not a leaf (still awaiting processing) is called an *open* or *active* subproblem. At any point in the computation, let $A$ be the set of active subproblems. Then $L = \min_{a \in A} b(a)$ is a global lower bound on the original problem. B&B terminates when there are no active subproblems or when the relative or absolute gap between $L$ and $f(x_I)$ is sufficiently small.

A B&B computation is logically divided into two phases: (1) find an optimal solution $x^*$ and (2) prove that $x^*$ is optimal. At any given point in the computation, all active subproblems $a$ such that $b(a) < f(x^*)$ must be processed to prove the optimality of $x^*$; all other subproblems can be pruned once we have found $x^*$. Thus for any given bounding and splitting strategy there is a minimum-size tree: that obtained by seeding the computation with the optimal solution or finding the optimal at the root. A candidate procedure that finds near-optimal solutions early in the B&B computation can reduce tree size tree by allowing early pruning.

When the incumbent procedure is weak near the root, the following adaptive branching strategy can quickly identify a feasible solution to enable at least some pruning. Initially apply depth-first search to find a leaf as soon as possible. Given an incumbent, switch to best-first search, which selects the node $n$ with the minimum value of $b(n)$. This hybrid search strategy is particularly applicable to combinatorial problems, since depth-first search will eventually subdivide the region until a feasible solution is found (e.g. after all possible choices have been made for one node). Depth-first search can open up many more subproblems than best-first search with a good early incumbent, but the hybrid strategy can be superior when it is difficult to find an incumbent.

In parallel B&B, one can parallelize the computation of independent subproblems (or subtrees) or parallelize the evaluation of individual sub-

problems. The latter is better when the tree is small at the start or end of the computation, unless the other processors can be kept busy with other independent work, such as generating incumbents (see Section 5.2.2).

It is important to keep the total tree size close to the size of that generated by a good serial solver. Good branch choices are critical early in the computation and good *load balancing* is critical later. That is, each processor must stay sufficiently busy with high-quality work. Otherwise in pathological cases the parallel computation performs so much more work than its serial counterpart that there is actually a slowdown anomaly [35, 64–67]. That is, adding processors increases wall-clock time to finish a computation.

Parallel platforms of any kind have far more memory than a single computer. A serial B&B computation may be forced to throw away seemingly unfavorable active nodes, thus risking losing the optimal solution. A parallel system is much more likely to have sufficient memory to finish the search.

There are many frameworks for parallel branch and bound including PUBB [102], Bob [12], PPBB-lib [112], PICO [36], Zram [73], and ALPS/BiCePS [96]. The user defines the above problem-specific procedures and the framework provides a parallel implementation. Bob++ [16] and Mallba [71] are even higher-level frameworks for solving combinatorial optimization problems.

### 5.2.2 Mixed-Integer Programming

A mixed-integer program (MIP) in standard form is:

$$\text{(MIP)} \quad \text{minimize} \quad c^T x \qquad \text{where} \quad \begin{cases} Ax = b \\ \ell \le x \le u \\ x_j \in \mathcal{Z} & \forall j \in D \subseteq \{1, \ldots, n\} \end{cases}$$

where $x$ and $c$ are $n$-vectors, $A$ is an $m \times n$ matrix, $b$ is an $m$-vector, and $\mathcal{Z}$ is the set of integers. Though in principle all input data are reals, for practical solution on a computer they are rational. Frequently the entries of $A$, $c$, and $b$ are integers. We can convert an inequality constraint in either direction to an equality by adding a variable to take up slack between the value of $ax$ and its bound $b$.

The only nonlinearity in MIP is the integrality constraints. Frequently binary variables represent decisions that must be yes/no (i.e. there can be no partial decision for partial cost and partial benefit). In principle MIPs can express any NP-complete optimization problem. In practice they are used for resource allocation problems such as transportation

logistics, facility location, and manufacturing scheduling, or for the study of natural systems such as protein folding.

In practice, MIPs are commonly solved with B&B and its variants. In this section, we consider the application of B&B to solve MIPs on a distributed-memory system. Specifically, we summarize the implementation of Eckstein, Hart, and Phillips, within the Parallel Integer and Combinatorial Optimizer (PICO) system [36].

**5.2.2.1 Branch-and-Bound Strategy.** If the integrality constraints are relaxed (removed), a MIP problem becomes a linear program (LP). Solving this LP is the classic bounding procedure for B&B. LPs are theoretically solvable in polynomial time [61] and are usually solved efficiently in practice with commercial tools such as CPLEX [31], XPRESS [119], or OSL [86], or free tools such as COIN-LP [26].

Serial B&B for MIP begins with the MIP as the root problem and bounds the root by computing the LP relaxation. If all integer variables ($x_j$ with $j \in D$) have integer values (within tolerance), then this is a feasible integer solution whose value matches a lower bound, and hence it is an optimal solution. If the LP relaxation is not a feasible solution, then there is some $j \in D$ such that the optimal solution to the LP relaxation $x^*$ has value $x_j^* \notin \mathcal{Z}$. We then create two new sub-MIPs as children: one with the restriction $x_j \leq \left\lfloor x_j^* \right\rfloor$ and one with the restriction $x_j \geq \left\lceil x_j^* \right\rceil$. For binary variables, one child has $x_j = 0$ and the other has $x_j = 1$. The feasible regions of the two children are disjoint and any solution with $\left\lfloor x_j^* \right\rfloor < x_j < \left\lceil x_j^* \right\rceil$, including $x^*$, is no longer feasible in either child. Thus the LP relaxation of a child provides a lower bound on the optimal solution within this subregion, and it will be different from the LP relaxation of the parent.

There are a number of common ways to improve the performance of this standard B&B MIP computation. Most MIP systems apply general and/or problem-specific cutting planes before branching to improve the lower bound on a subproblem while delaying branching as long as possible. Given $x^*$, an optimal non-integral solution to the LP relation of a (sub)problem, a cutting plane is a constraint $ax = b$ such that $ax' = b$ for all possible (optimal) integer solutions $x'$ but $ax^* \neq b$. Adding this constraint to the system makes the current LP optimal infeasible.

Careful branch selection in MIP solvers can significantly impact search performance. In strong branching, one tries a branch and partially evaluates the subtree to determine a merit rating for the branch. Most or all the work done for strong branching is thrown away, but it can sometimes reduce tree size sufficiently to merit the effort. A less computationally

demanding strategy is to maintain gradients for each branch choice. For the simple branching described above, the gradient for a single branch is the change in LP objective value divided by the change in the variable (the latter is always less than 1). When a variable is a branching candidate for the first time, one can initialize its gradient by pretending to branch in each direction. This is much better than, for example, setting an uninitialized gradient to the average of the gradients computed so far [69]. However, each gradient initialization requires two LP bounding operations. Finally, one can compute problem-specific constraints to partition the current feasible region, though this may be expensive.

One can also improve performance of B&B for MIP by finding feasible integer solutions using methods other than finding leaf nodes when an LP relaxation is integer feasible. Since MIPs frequently have combinatorial structure, one can use general heuristics such as evolutionary algorithms or tabu search or problem-specific methods that exploit structure. In particular, there are many approximation algorithms for combinatorial problems that find an LP-relaxation for a MIP and "round" this nontrivially to obtain a feasible integer solution whose objective value is provably close to the LP relaxation bound (see [11, 19, 88] for a tiny sample). The provable bound is limited by the integrality gap of the problem (the ratio between the best integer solution and the best LP solution; this is a measure of the strength of the formulation [22]).

There is a variety of parallel MIP solvers. PARINO [69], SYMPHONY [94], and COIN/BCP [63, 72] are designed for small-scale, distributed-memory systems such as clusters. BLIS [96], under development, is designed as a more scalable version of SYMPHONY and BCP. It will be part of COIN once it is available. See the discussion of all three in [95]. FATCOP [25] is designed for grid systems. PICO's parallel B&B search strategy is particularly well-suited for solving MIPs on tightly-coupled massively-parallel distributed-memory architectures, such as those available at the National Laboratories for solution of national-scale problems. For these architectures, one has exclusive use of perhaps thousands of processors for the entire computation. Thus one major concern is effectively using all these processors during the initial *ramp up* phase, when the search tree is small. After ramp up, PICO enters a *parallel-subproblem* phase managing an asynchronous parallel search using load balancing to ensure that all worker processes are solving interesting subproblems. These two aspects of parallel B&B in PICO are discussed in the next two sections. The combination of strategies can use massive parallelism effectively. In preliminary experiments on some problems PICO had near perfect speed up through 512 processors [36].

**5.2.2.2 Managing Ramp Up.** PICO uses an explicit ramp up phase in which all the processors work on a single subproblem, parallelizing the individual subproblem evaluation steps. In particular, PICO supports parallel (1) initialization of gradient estimation (used for branching prioritization), (2) problem-specific preprocessing, (3) root bounding, and (4) incumbent and cutting-plane generation. Also some processors can search for incumbents and cutting planes independently from those growing the tree. For example, since parallel LP solvers do not currently scale well to thousands of processors, excess processors can search for incumbents during root bounding.

PICO parallelizes gradient initialization during ramp up. In general if there are $f$ potential branch variables with uninitialized gradients, each of the $P$ processors initializes the gradients for $\lceil f/P \rceil$ or $\lfloor f/P \rfloor$ variables and sends these values to all processors using all-to-all communication. Though PICO doesn't currently support strong branching, in principle the work for strong branching could be divided among the processors in the same way with the same exchange of branch quality information. One can also parallelize complex custom branching computations.

To parallelize problem-specific preprocessing, processors can cooperate on individual preprocessing steps or they can compute independent separate steps. Good preprocessing is often critical for computing an exact solution to hard combinatorial problems (e.g. those expressed by MIPs). Real-world instances frequently have special structure that is captured by a small upper bound on a parameter $k$ (e.g. the degree of a graph, maximum contention for a resource, etc.). To solve fixed-parameter-tractable problems, one first *kernelizes* the problem, transforming it in polynomial time into a new instance with size bounded by a function of $k$. These preprocessing steps are frequently local and therefore good candidates for parallelization. See Fellows [39] for an excellent summary of the theory and practice of kernelization.

One can also parallelize the LP bounding procedure. This is particularly desirable at the root because bounding the root problem can be more than an order of magnitude more expensive than bounding subproblems. A subproblem bounding calculation can start from the parent's basis and, since there is typically only one new constraint, the LP is usually quickly re-solved with a few pivots of dual simplex. The root problem frequently starts with nothing, not even a feasible point. The pPCx code [27] is a parallel interior-point LP solver. The core computational problem is the solution of a linear system of the form $AD^2A^Tx = b$ where $A$ is the original constraint matrix and $D$ is a diagonal matrix that changes each iteration. Parallel direct Cholesky solvers are robust, but currently do not provide reasonable speed up beyond a

few dozen processors. Sandia National Laboratories is leading a research effort to find more scalable interior-point solvers using iterative linear systems solvers, but this is still an open research problem. We are not aware of any (massively) parallel dual simplex solvers so even during ramp up, subproblem re-solves are serial for the moment.

PICO must use free LP solvers for serial (and eventually parallel) bounding. Faster commercial LP codes do not have licensing schemes for massively-parallel (MP) machines, and individual processor licenses would be prohibitively expensive. PICO has a coping strategy to avoid slow root solves in MP computations. The user can solve the root LP offline (e.g. using a fast commercial solver) and then feed this LP solution to a subsequent parallel computation.

At any tree node, one can parallelize the generation of cutting planes and the search for feasible integer solutions, by cooperating on the generation of one plane/solution or by generating different planes/solutions in parallel. Heuristic methods such as evolutionary algorithms can effectively use many parallel processors independently from the ramp-up tree growth computation. LP-based methods need the LP solutions and are usually best integrated with the tree evaluation. In some cases, such as alpha-point heuristics for scheduling problems [88], an approximation algorithm has a natural parameter (alpha) whose range of values can be partitioned among the processors. PICO's general cut-pivot-dive heuristic can explore multiple strategy choices in parallel [80].

**5.2.2.3 Managing Independent Subproblems.** PICO's ramp-up phase usually terminates when there are enough subproblems to keep most processors busy. After the ramp up phase, PICO switches to a phase where processors work on separate subproblems. During this phase, a number of hubs coordinate the search. Each hub controls a set of worker processors, few enough that the workers are not slowed by contention for attention from the hub.

When ramp up ends, each hub takes control of an equal share of the active subproblems. Each worker has a local pool of subproblems to reduce dependence on its hub. Though PICO has considerable flexibility in subproblem selection criteria, once there is an incumbent both hubs and workers generally use a best-first strategy.

PICO has three load balancing mechanisms. First, if a hub runs out of useful work not delegated to a worker, it can rebalance among its workers by pulling some work back from a worker and giving it to others. Second, when a worker decides not to keep a subproblem locally, it returns it to its hub or probabilistically scatters it to a random hub. The probability of scattering depends upon the load controlled by its

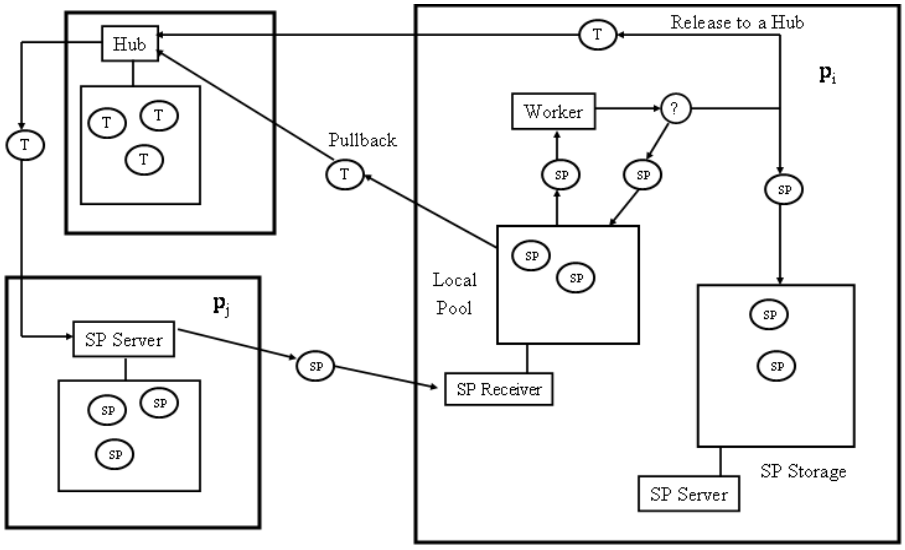


*Figure 5.1.* PICO's subproblem management. Hubs handle only tokens. When a worker generates a subproblem, it either keeps the subproblem in a local pool or gives control to a hub (not necessarily its own), storing the data to send directly to the worker to which the problem is later assigned. The worker on processor $p_j$ is only partially illustrated. [Figure by Jonathan Eckstein, PICO core designer]

hub relative to average system load. The third mechanism is global load balancing done with a rendezvous method as described in Section 5.3.

Because hub communication can become a bottleneck, hubs do not handle all the data associated with the subproblems they control. Instead they keep a small *token* containing only the information needed to control the problem. In particular, the token contains the subproblem bound, and the ID of the worker that created the subproblem. When a worker creates a subproblem and relinquishes control to a hub, it stores all the subproblem data (bounds, basis, etc), and sends this small token to the hub (represented by $T$ in Fig. 5.1). When a hub wants to dispatch a subproblem to processor $p_i$, it sends a message to the processor $p_j$ that created the subproblem telling processor $p_j$ to send the subproblem data to processor $p_i$ (represented by $SP$ in Fig. 5.1). Thus the communication pattern in a MIP computation has many small messages going to and from the hubs and, because of the load balancing, long messages going point to point in a reasonably random pattern.

Whenever a processor finds a new incumbent, it broadcasts the new objective value to all processors using a binary tree rooted at that processor. Processors give priority to handling these messages because new

incumbents can prune active subproblems. Processors only forward the best incumbent they have seen so far, so if there are multiple incumbent message waves propagating through the system, all dominated ones die as they meet better solutions.

### 5.2.3 Quadratic Assignment Problem

In this section, we provide another example of B&B applied to the quadratic assignment problem (QAP). Specifically, we summarize the grid-based QAP solver developed by Anstreicher et al. [2]. This solver uses a master-worker paradigm to parallelize B&B. The master-worker paradigm has been widely used to parallelize B&B algorithms [46], and it is very well-suited for grid-based applications. We discuss this programming model, and we provide an overview of the MW framework which was used by Anstreicher et al. [2].

**5.2.3.1 The Master-Worker Paradigm.** The master-worker paradigm is a canonical programming paradigm for parallel computing that is particularly well-suited for grid-computing applications. In a master-worker application, all algorithm control is done by a *master* processor. *Worker* processors concurrently execute independent tasks. A wide variety of sequential approaches to large-scale problems map naturally to the master-worker paradigm [48] including tree search algorithms (e.g. for integer programming), stochastic programming, population-based search methods like genetic algorithms, parameter analysis for engineering design, and Monte Carlo simulations.

The master-worker paradigm can parallelize many programs with centralized control using grid computing platforms because it works effectively in dynamic and heterogeneous computing environments. If additional processors become available during the course of a computation, they can be integrated as workers that are given independently executable computational tasks. If a worker fails while executing a task, the master can simply reschedule that portion of the computation. In this manner, the master-worker paradigm is a flexible and reliable tool for grid computing applications. Furthermore, this type of centralized control eases the burden of adapting to a heterogeneous computational environment, since only the master process needs to be concerned with how tasks are assigned to resources.

The basic master-worker paradigm has several limitations. It is not robust if the master fails. However, the programmer can overcome this limitation by using checkpoints to save the state of the master (which implicitly knows the global state of the entire calculation). Another limitation is that the total throughput may become limited because (a) workers can

become idle while waiting for work from the master, and (b) precedence constraints between tasks can limit total parallelism of the calculation.

The master-worker paradigm is inherently unscalable because workers can become idle when the master cannot quickly respond to all requests. If the workers finish tasks quickly, there may simply be too many requests for the master to service immediately. The master's response rate is reduced by auxiliary computation (e.g. to prioritize available tasks). In many cases, such bottlenecks can be minimized by adapting the granularity of the tasks. For example, in a tree search the computation required by a task depends on the size/depth of the tasks's subtree. Thus, the master can reduce the rate at which processors make work requests by assigning larger portions of the tree to each processor [48].

A precedence constraint between a pair of tasks $(a, b)$ indicates task $a$ must complete before task $b$ starts. This serialization limits the number of independent tasks at any point in the computation; in the worst case, precedence constraints impose a total linear order on the tasks. On a grid system, with unpredictable processing and communication times, predecessor jobs can delay their successors indefinitely. Some applications permit relaxed precedence. For example, Goux et al. [48] discuss a cutting plane algorithm for stochastic programming for which tasks are weakly synchronized. In this application, the next iteration of the cutting plane algorithm can start after a fraction of the previous iteration's tasks have completed, thereby avoiding some synchronization delays.

Increasing the efficiency of a master-worker algorithm by increasing the grain size or by reducing synchronization can sometimes worsen the basic algorithm. For example, increasing the granularity in a tree search may lead to the exploration of parts of the tree that would have otherwise been ignored. Thus the application developer must balance parallel efficiency (keeping workers busy) with total computational efficiency.

**5.2.3.2 The MW Framework.** The MW framework facilitates implementation of parallel master-worker applications on computational grids [48]. The application programming interface of MW is a set of C++ abstract classes. The programmer provides concrete implementation of this abstract functionality for a particular application. These classes define the basic elements of the master controller, how workers are launched and how tasks are executed. The MW framework provides general mechanisms for distributing algorithm control information to the worker processors. Further, MW collects job statistics, performs dynamic load balancing and detects termination conditions. Thus MW handles many difficult metacomputing issues for an application developer, thereby allowing rapid development of sophisticated applications.

MW is a C++ library, and three MW classes must be extended to define a new master-worker application [48]. The master process controls task distribution via the MWDriver class. The MWDriver base class handles workers joining and leaving the computation, assigns tasks to appropriate workers, and rematches running tasks when workers are lost. The programmer must specify how to process commandline information, determine a set of initial jobs, process a completed task, and what information workers need at start up. The MWWorker class controls the worker. The programmer must specify how to process start up data and how to execute a task. The MWTask class describes task data and results. The derived task class must implement functions for sending and receiving this data.

To implement MW on a particular computational grid, a programmer must also extend the MWRMComm class to derive a grid communication object. The initial implementation of MW uses Condor [28] as its resource management system. Condor manages distributively owned collections ("pools") of processors of different types, including workstations, nodes from PC clusters, and nodes from conventional multiprocessor platforms. When a user submits a job, the Condor system discovers a suitable processor for the job in the pool, transfers the executable, and starts the job on that processor. Condor may checkpoint the state of a job periodically, and it migrates a job to a different processor in the pool if the current host becomes unavailable for any reason. Currently, communication between master and workers uses a Condor-enabled version of PVM [92] or Condor's remote system call functionality. MW has also been extended at Sandia National Laboratories to use an MPI-based communication object.

**5.2.3.3 Solving QAP with Branch-and-Bound.** The quadratic assignment problem (QAP) is a standard problem in location theory. The QAP in *Koopmans-Beckmann* form is

$$\min_{\pi} \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{\pi(i),\pi(j)} + \sum_{i=1}^{n} c_{i,\pi(i)},$$

where $n$ is the number of facilities and locations, $a_{ij}$ is the flow between facilities $i$ and $j$, $b_{kl}$ is the distance between locations $k$ and $l$, $c_{ik}$ is the fixed cost of assigning facility $i$ to location $k$, and $\pi(i) = k$ if facility $i$ is assigned to location $k$. The QAP is NP-hard. Most exact QAP methods are variants of B&B. Anstreicher et al. [1, 2, 17] developed and applied a new convex quadratic programming bound that provides stronger bounds than previous methods. Because QAP is so difficult, previous exact solutions used parallel high-performance computers. Anstreicher

et al. [2] review these results and note that grid computing may be more cost-effective for these problems.

Anstreicher et. al. [2] developed an MW-based branch-and-bound QAP solver (MWQAP). They ran it on a Condor pool communicating with remote system calls. Since Condor provides a particularly dynamic grid computing environment, with processors leaving and entering the pool regularly, MW was critical to ensure tolerance of worker processor failures. To make computations fully reliable, MWQAP uses MW's checkpointing feature to save the state of the master process. This is particularly important for QAP because computations currently require many days on many machines.

The heterogeneous and dynamic nature of a Condor-based computational grid makes application performance difficult to assess. Standard performance measures such as wall clock time and cumulative CPU time do not separate application code performance from computing platform performance. MW supports the calculation of application-specific benchmark tasks to determine the power of each worker so the evaluator can normalize CPU times. For the QAP solver, the benchmark task is evaluating a small, specific portion of the B&B tree.

As we noted earlier, the master-worker paradigm usually requires application-specific tuning for scalability. MWQAP uses coarse granularity. Each worker receives an active node and computes within that subtree independently. If after a fixed number of seconds the worker has not completely evaluated the subtree, it passes the remaining active nodes back to the master. To avoid sending back "easy" subproblems, workers order unsolved nodes based on the relative gap and spend extra time solving deep nodes that have small relative gaps. "Easy" subproblems can lead to bottlenecks at the master because they cannot keep their new workers busy.

The master generally assigns the next worker the deepest active node on its list. However, when the set of active nodes is small, the master dispatches difficult nodes and gives the workers shorter time slices so the workers can return challenging open nodes. This ensures that the master's pool of unsolved subproblems is sufficiently large to keep all available workers busy.

Anstreicher et. al. note that in general B&B, this independent subtree execution could explore many more nodes than its sequential counterpart. However, their QAP calculations are seeded with good known solutions, so many of these nodes are pruned during each worker's search of a subtree. Thus this strategy limits master-worker communication without significantly impairing the overall search performance.

MWQAP has solved instances of the QAP that have remained unsolved for decades, including the nug30 problem defined by Nugent, Vollmand, and Ruml [83]. The Nugent problems are the most-solved set of QAPs, and the solution of these problems has marked advances in processor capability and QAP solution methods. Solution of nug30 required an average of 650 workers for one week when seeded with a previously known solution [2]. The computation halted and restarted five times using MW's checkpointing feature. On average MWQAP solved approximately one million linear assignment problems per second.

### 5.2.4 Phylogenetic Tree Reconstruction

In this section, we provide an example of B&B applied to reconstructing an evolutionary history (phylogenetic tree). Specifically, we focus on the shared-memory parallelization of the maximum parsimony (MP) problem using B&B based on work by Bader and Yan[10, 78, 120, 121].

**5.2.4.1 Biological Significance and Background.** All biological disciplines agree that species share a common history. The genealogical history of life is called phylogeny or an evolutionary tree. Reconstructing phylogenies is a fundamental problem in biological, medical, and pharmaceutical research and one of the key tools in understanding evolution. Problems related to phylogeny reconstruction are widely studied. Most have been proven or are believed to be NP-hard problems that can take years to solve on realistic datasets [20, 87]. Many biologists throughout the world compute phylogenies involving weeks or years of computation without necessarily finding global optima. Certainly more such computational analyses will be needed for larger datasets. The enormous computational demands in terms of time and storage for solving phylogenetic problems can only be met through high-performance computing (in this example, large-scale B&B techniques).

A phylogeny (phylogenetic tree) is usually a rooted or unrooted bifurcating tree with leaves labeled with species, or more precisely with taxonomic units (called *taxa*) that distinguish species [110]. Locating the root of the evolutionary tree is scientifically difficult so a reconstruction method only recovers the topology of the unrooted tree. Reconstruction of a phylogenetic tree is a statistical inference of a true phylogenetic tree, which is unknown. There are many methods to reconstruct phylogenetic trees from molecular data [81]. Common methods are classified into two major groups: criteria-based and direct methods. Criteria-based approaches assign a score to each phylogenetic tree according to some criteria (e.g., parsimony, likelihood). Sometimes computing the score requires auxiliary computation (e. g. computing hypothetical ancestors

for a leaf-labeled tree topology). These methods then search the space of trees (by enumeration or adaptation) using the evaluation method to select the best one. Direct methods build the search for the tree into the algorithm, thus returning a unique final topology automatically.

We represent species with binary sequences corresponding to morphological (e. g. observable) data. Each bit corresponds to a feature, called a *character*. If a species has a given feature, the corresponding bit is one; otherwise, it is zero. Species can also be described by molecular sequence (nucleotide, DNA, amino acid, protein). Regardless of the type of sequence data, one can use the same parsimony phylogeny reconstruction methods. The evolution of sequences is studied under a simplifying assumption that each site evolves independently.

The Maximum Parsimony (MP) objective selects the tree with the smallest total evolutionary change. The *edit distance* between two species as the minimum number of evolutionary events through which one species evolves into the other. Given a tree in which each node is labeled by a species, the *cost* of this tree (tree length) is the sum of the costs of its edges. The cost of an edge is the edit distance between the species at the edge endpoints. The *length* of a tree $T$ with all leaves labeled by taxa is the minimum cost over all possible labelings of the internal nodes.

Distance-based direct methods ([37, 38, 68]) require a distance matrix $D$ where element $d_{ij}$ is an estimated evolutionary distance between species $i$ and species $j$. The distance-based Neighbor-Joining (NJ) method quickly computes an approximation to the shortest tree. This can generate a good early incumbent for B&B. The neighbor-joining (NJ) algorithm by Saitou and Nei [99], adjusted by Studier and Keppler [106], runs in $O(n^3)$ time, where $n$ is the number of species (leaves). Experimental work shows that the trees it constructs are reasonably close to "true" evolution of synthetic examples, as long as the rate of evolution is neither too low nor too high. The NJ algorithm begins with each species in its own subtree. Using the distance matrix, NJ repeatedly picks two subtrees and merges them. Implicitly the two trees become children of a new node that contains an artificial taxon that mimics the distances to the subtrees. The algorithm uses this new taxon as a representative for the new tree. Thus in each iteration, the number of subtrees decrements by one till there are only two left. This creates a binary topology. A distance matrix is *additive* if there exists a tree for which the inter-species tree distances match the matrix distances exactly. NJ can recover the tree for additive matrices, but in practice distance matrices are rarely additive. Experimental results show that on reasonable-length sequences parsimony-based methods are almost always more accurate (on synthetic data with known evolution) than neighbor-joining and some other com-

petitors, even under adverse conditions [98]. In practice MP works well, and its results are often hard to beat.

In this section we focus on reconstructing phylogeny using maximum parsimony (minimum evolution). A brute-force approach for maximum parsimony examines all possible tree topologies to return one that shows the smallest amount of total evolutionary change. The number of unrooted binary trees on $n$ leaves (representing the species or taxa) is $(2n-5)!! = (2n-5) \cdot (2n-7) \cdots 3$. For instance, this means that there are about 13 billion different trees for an input of $n = 13$ species. Hence it is very time-consuming to examine all trees to obtain the optimal tree. Most researchers focus on heuristic algorithms that examine a much smaller set of most promising topologies and choose the best one examined. One advantage of B&B is that it provides instance-specific lower bounds, showing how close a solution is to optimal [56].

The phylogeny reconstruction problem with maximum parsimony (MP) is defined as follows. The input is a set of $c$ characters and a set of taxa represented as length-$c$ sequences of values (one for each character). For example, the input could come from an aligned set of DNA sequences (corresponding elements matched in order, with gaps). The output is an unrooted binary tree with the given taxa at leaves and assignments to the length-$c$ internal sequences such the resulting tree has minimum total cost (evolutionary change). The characters need not be binary, but each usually has a bounded number of states. Parsimony criteria (restrictions on the changes between adjacent nodes) are often classified into Fitch, Wagner, Dollo, and Generalized (Sankoff) Parsimony [110]. In this example, we use the simplest criteria, Fitch parsimony [41], which imposes no constraints on permissible character state changes. The optimization techniques we discuss are similar across all of these types of parsimony.

Given a topology with leaf labels, we can compute the optimal internal labels for that topology in linear time per character. Consider a single character. In a leaf-to-root sweep, we compute for each internal node $v$ a set of labels optimal for the subtree rooted at $v$ (called the Farris Interval). Specifically, this is the intersection of its children's sets (connect children though $v$) or, if this intersection is empty, the union of its children's sets (agree with one child). At the root, we choose an optimal label and pass it down. Children agree with their parent if possible. Because we assume each site evolves independently, we can set all characters simultaneously. Thus for $m$ character and $n$ sequences, this takes $O(nm)$ time. Since most computers can perform efficient bitwise logical operations, we use the binary encoding of a state in order to implement intersection and union efficiently using bitwise AND and bitwise OR. Even so, this operation dominates the parsimony B&B computation.

The following sections outline the parallel B&B strategy for MP that is used in the GRAPPA (Genome Rearrangement Analysis through Parsimony and other Phylogenetic Algorithms) toolkit [78]. Note that the maximum parsimony problem is actually a minimization problem.

**5.2.4.2 Strategy.** We now define the *branch*, *bound*, and *candidate* functions for phylogeny reconstruction B&B. Each node in the B&B tree is associated with either a partial tree or a complete tree. A tree containing all $n$ taxa is a *complete tree*. A tree on the first $k$ ($k < n$) taxa is a *partial tree*. A complete tree is a candidate solution. Tree $T$ is *consistent* with tree $T'$ iff $T$ can be reduced into $T'$; i.e., $T'$ can be obtained from $T$ by removing all the taxa in $T$ that are not in $T'$. The subproblem for a node with partial tree $T$ is to find the most parsimonious complete tree consistent with $T$.

We partition the active nodes into *levels* such that level $k$, for $3 \leq k \leq n$, contains all active nodes whose partial trees contain the first $k$ taxa from the input. The root node contains the first three taxa (hence, indexed by level 3) since there is only one possible unrooted tree topology with three leaves. The branch function finds the immediate successors of a node associated with a partial tree $T_k$ at level $k$ by inserting the $(k+1)$st taxon at any of the $(2k-3)$ possible places. A new node (with this taxon attached by an edge) can join in the middle of any of the $(2k-3)$ edges in the unrooted tree. For example, in Figure 5.2, the root on three taxa is labeled (A), its three children at level four are labeled (B), (C), and (D), and a few trees at level five (labeled (1) through (5)) are shown. We use depth-first search (DFS) as our primary B&B search strategy, and a heuristic best-first search (BeFS) to break ties between nodes at the same depth. The search space explored by this approach depends on the addition order of taxa, which also influences the efficiency of the B&B algorithm. This issue is important, but not further addressed in this chapter.

Next we discuss the bound function for maximum parsimony. A node $v$ associated with tree $T_k$ represents the subproblem to find the most parsimonious tree in the search space that is consistent with $T_k$. Assume $T_k$ is a tree with leaves labeled by $S_1, \ldots, S_k$. Our goal is to find a tight lower bound of the subproblem. However, one must balance the quality of the lower bound against the time required to compute it in order to gain the best performance of the overall B&B algorithm.

Hendy and Penny [56] describe two practical B&B algorithms for phylogeny reconstruction from sequence data that use the cost of the associated partial tree as the lower bound of this subproblem. This traditional approach is straightforward, and obviously, it satisfies the

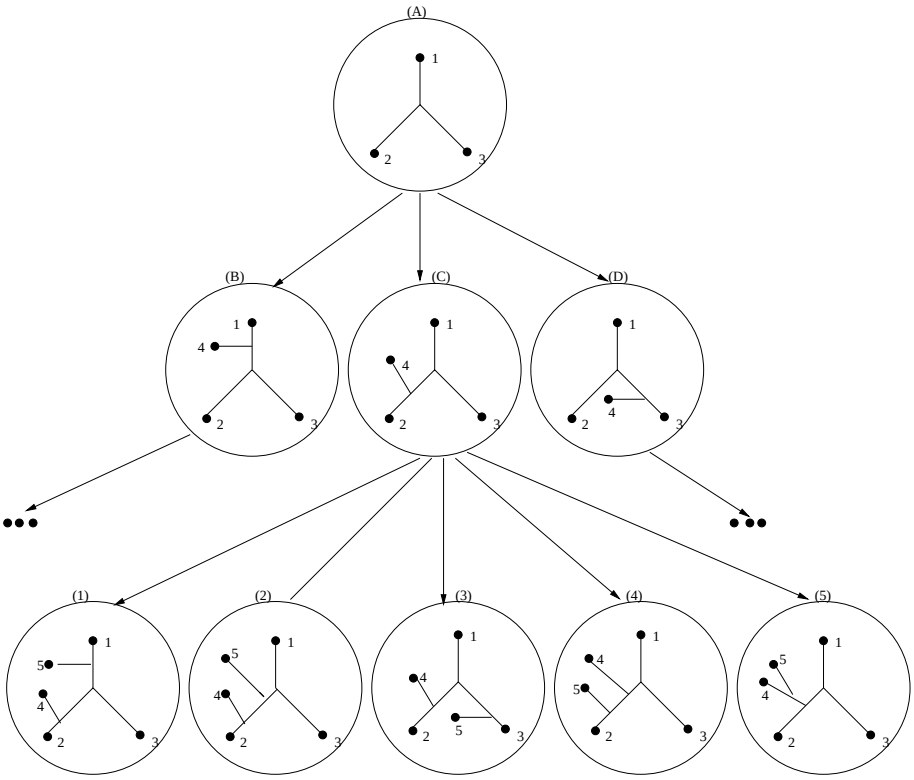


*Figure 5.2.* Maximum Parsimony B&B search space.

necessary properties of the bound function. However, it is not tight and does not prune the search space efficiently. Purdom et al. [91] use single-character discrepancies of the partial tree as the bound function. For each character one computes a difference set, the set of character states that do not occur among the taxa in the partial tree and hence only occur among the remaining taxa. The single-character discrepancy is the sum over all characters of the number of the elements in these difference sets. The lower bound is therefore the sum of the single-character discrepancy plus the cost of the partial tree. This method usually produces much better bounds than Hendy and Penny's method, and experiments show that it usually fathoms more of the search space [91]. Another advantage of Purdom's approach is that given an addition order of taxa, there is only one single-character discrepancy calculation per level. The time needed to compute the bound function is negligible.

Next we discuss the candidate function and incumbent $x_I$. In phylogeny reconstruction, it is expensive to compute a meaningful feasible

solution for each partial tree, so instead we compute an upper bound on the input using a direct method such as neighbor-joining [99, 106] before starting the B&B search. We call this value the global upper bound, $f(x_I)$, the incumbent's objective function. In our implementation, the first incumbent is the best returned by any of several heuristic methods.

The greedy algorithm [34], an alternative incumbent heuristic, proceeds as follows. Begin with a three-taxa core tree and iteratively add one taxon at a time. For an iteration with a $k$-leaf tree, try each of the $n - k$ remaining taxon in each of the $2k - 3$ possible places. Select the lowest-cost $(k + 1)$-leaf tree so formed.

Any program, regardless of the algorithms, requires implementation on a suitable data structure. As mentioned previously, we use DFS as the primary search strategy and BeFS as the secondary search strategy. For phylogeny reconstruction with $n$ taxa, the depth of the subproblems ranges from 3 to $n$. So we use an array to keep the open subproblems sorted by DFS depth. The array element at location $i$ contains a priority queue (PQ) of the subproblems with depth $i$, and each item of the PQ contains an external pointer to stored subproblem information.

The priority queues (PQs) support best-first-search tie breaking and allow efficient deletion of all dominated subproblems whenever we find a new incumbent. There are many ways to organize a PQ (see [12] for an overview). In the phylogeny reconstruction problem, most of the time is spent evaluating the tree length of a partial tree. The choice of PQ data structures does not make a significant difference. So for simplicity, we use a D-heap for our priority queues. A heap is a tree where each node has higher priority than any of its children. In a D-heap, the tree is embedded in an array. The first location holds the root of the tree, and locations $2i$ and $2i + 1$ are the children of location $i$.

**5.2.4.3 Parallel framework.** Our parallel maximum parsimony B&B algorithm uses shared-memory. The processors can concurrently evaluate open nodes, frequently with linear speedup. As described in Section 5.2.4.2, for each level of the search tree (illustrated in Figure 5.2), we use a priority queue represented by binary heaps to maintain the active nodes in a heuristic order. The processors concurrently access these heaps. To ensure each subproblem is processed by exactly one processor and to ensure that the heaps are always in a consistent state, at most one processor can access any part of a heap at once. Each heap $H_i$ (at level $i$) is protected by a lock $Lock_i$. Each processor locks the entire heap $H_i$ whenever it makes an operation on $H_i$.

In the sequential B&B algorithm, we use DFS strictly so $H_i$ is used only if the heaps at higher level (higher on the tree, lower level number)

are all empty. In the parallel version, to allow multiple processors shared access to the search space, a processor uses $H_i$ if all the heaps at higher levels are empty or locked by other processors.

The shared-memory B&B framework has a simple termination process. A processor can terminate its execution when it detects that all the heaps are unlocked and empty: there are no more active nodes except for those being decomposed by other processors. This is correct, but it could be inefficient, since still-active processors could produce more parallel work for the prematurely-halted processors. If the machine supports it, instead of terminating, a processor can declare itself idle (e. g. by setting a unique bit) and go to sleep. An active processor can then wake it up if there's sufficient new work in the system. The last active processor terminates all sleeping processors and then terminates itself.

**5.2.4.4 Impact of Parallelization.** There are a variety of software packages to reconstruct sequence-based phylogeny. The most popular phylogeny software suites that contain parsimony methods are PAUP* by Swofford [109], PHYLIP by Felsenstein [40], and TNT and NONA by Goloboff [47, 82]. We have developed a freely-available shared-memory code for computing MP, that is part of our software suite, GRAPPA (Genome Rearrangement Analysis through Parsimony and other Phylogenetic Algorithms) [78]. GRAPPA was designed to re-implement, extend, and especially speed up the breakpoint analysis (BP-Analysis) method of Sankoff and Blanchette [100]. Breakpoint analysis is another form of parsimony-based phylogeny where species are represented by ordered sets of genes and distance is measured relative to differences in orderings. It is also solved by branch and bound. Our MP software does not constrain the character states of the input. It can use real molecular data and characters reduced from gene-order data such as Maximum Parsimony on Binary Encodings (MPBE) [30].

The University of New Mexico operates *Los Lobos*, the NSF / Alliance 512-processor Linux supercluster. This platform is a cluster of 256 IBM Netfinity 4500R nodes, each with dual 733 MHz Intel Xeon Pentium processors and 1 GB RAM, interconnected by Myrinet switches. We ran *GRAPPA* on *Los Lobos* and obtained a 512-fold speed-up (linear speedup with respect to the number of processors): a complete breakpoint analysis (with the more demanding inversion distance used in lieu of breakpoint distance) for the 13 genomes in the Campanulaceae data set ran in less than 1.5 hours in an October 2000 run, for a *million-fold* speedup over the original implementation [8, 10]. Our latest version features significantly improved bounds and new distance correction methods and, on the same dataset, exhibits a speedup factor of *over one*

*billion*. In each of these cases a factor of 512 speed up came from parallelization. The remaining speed up came from algorithmic improvements and improved implementation.

## 5.3 Parallel Algorithmic Primitives

This section describes parallel algorithmic primitives that are representative of the techniques commonly used to coordinate parallel processes. Our three illustrations of parallel B&B use some of these. For a more detailed discussion on algorithm engineering for parallel computation, see the survey by Bader, Moret, and Sanders [9].

### 5.3.1 Tree-based Reductions

Tree-based reductions are an efficient way to compute the result of applying an associative operator to $k$ values. For example, this is an efficient way to compute the sum or the max of $k$ values. This uses a *balanced binary tree* of processors with $k$ leaves. That is, the heights of the two subtrees rooted at each internal node are approximately equal. Initially each leaf processor holds one of the $k$ input values. Each leaf passes its value to its parent. Each internal node waits to receive the value from each child, applies the operator (e.g. sum or max) and sends the result to its parent. Eventually the root computes the final value. It then sends the results to its children, which in turn propagate the final value, essentially implementing a broadcast from the root. The internal nodes can also have values of their own they add in as the computation proceeds up the tree. In this case, $k$ is the number of nodes rather than the number of leaves. This communication pattern can implement a *synchronization* for example when all processors must wait to write values until all processors have finished reading. Each processor signals its parent when it has finished reading. When the root receives signals from both its children, all processors in the system have finished. When the signal arrives from the root, each processor can continue safely.

### 5.3.2 Parallel Prefix

The *prefix-sum* operation (also known as the scan operation [14, 15, 57]) takes an array $A$ of length $n$, and a binary, associative operator $*$, and computes the prefix-sum values $b_i = a_0 * a_1 * \ldots * a_i$, for all $0 \leq i < n$. In parallel, processor $i$ stores value $a_i$ at the beginning and value $b_i$ at the end. There are fast implementations based on balanced binary trees. Suppose an arbitrary subset of processors have a given property (e.g. receiving a 1 from a random bit generator). Each processor knows only whether it has the property or not. We would like to *rank*

the processors with the property, giving each a unique number in order starting from 1. This is a prefix operation with $a_i = 1$ if the processor has the property and $a_i = 0$ otherwise. The operator is sum. The $i$th processor with the property has $b_i = i$. See section 5.3.4 for an example application of ranking. Prefix sums also build other primitives such as array compaction, sorting, broadcasting, and segmented prefix-sums. The latter is a prefix sum where the input array is partitioned into consecutive segments of arbitrary size and the values $b_i$ are computed only within the segments.

### 5.3.3 Pointer-jumping

Consider a data structure using pointers where each element has at most one outgoing pointer, for example an in-tree or a linked list. In pointer jumping, each element of the data structure begins with its pointer (or a copy of it) and iteratively replaces it with the pointer in the element it's pointing to until it reaches a node with no outgoing pointer. In a linked list, for example, the first element initially points to the second. After one iteration, it points to the fourth, then the eighth, etc. In $\log n$ iterations it points to the $n$th and final element. This technique is also called path doubling or shortcutting. It can convert trees into strict binary trees (0 or 2 children) by collapsing chains of degree-two nodes. Each node in a rooted, directed forest can find its root quickly. This is a crucial step in handling equivalence classes-such as detecting whether or not two nodes belong to the same component. When the input is a linked list, this algorithm solves the parallel prefix problem.

### 5.3.4 Rendezvous

In distributed-memory systems, the rendezvous algorithm allows processors to "find" other processors that have similar or complementary properties. The properties evolve during the computation and cannot be predicted a priori. If the properties have unique ranks between 1 and $P$ (the number of processors), then all processors with property type $i$ can "meet" at processor $i$. Each processor with property $i$ sends a message to processor $i$ with its identifier. Processor $i$ collects all the identifiers sent to it and sends the list to each processor on the list. This effectively "introduces" the processors. As a particular example, consider load balancing in PICO. Each hub computes its local load, a function of the number and quality of subproblems. The hubs all learn the total load via a tree-based reduction (summing the loads). Each hub determines if it is a *donor*, a processor with load sufficiently above average or a *receiver*, a processor with load sufficiently below average. Using a

parallel prefix operation on the tree, each donor is assigned a unique rank and each receiver is assigned a unique rank. Then the $i$th donor and $i$th receiver rendezvous at processor $i$. Once they know each other, donor $i$ sends work to receiver $i$ in a point-to-point message.

### 5.3.5 Advanced Parallel Techniques

An entire family of techniques of major importance in parallel algorithms is loosely termed *divide-and-conquer*—such techniques decompose the instance into smaller pieces, solve these pieces independently (typically through recursion), and then merge the resulting solutions into a solution to the original instance. Such techniques are used in sorting, in almost any tree-based problem, in a number of computational geometry problems (finding the closest pair, computing the convex hull, etc.), and are also at the heart of fast transform methods such as the fast Fourier transform (FFT). A variation on this theme is a *partitioning strategy*, in which one seeks to decompose the problem into independent subproblems—and thus avoid any significant work when recombining solutions; quicksort is a celebrated example, but numerous problems in computational geometry and discrete optimization can be solved efficiently with this strategy (particularly problems involving the detection of a particular configuration in 3- or higher-dimensional space).

Another general technique for designing parallel algorithms is called *pipelining*. A pipeline acts like an assembly line for a computation. Data enters the first stage and proceeds though each stage in order. All stages can compute values for different data simultaneously. Data pumps through the pipeline synchronously, entering the next stage as the previous data exits. Suppose there are $k$ stages, each requiring $t$ time. Then processing each piece of data requires $kt$ time, but a new value arrives every $t$ time units.

*Symmetry breaking* provides independent work between processors in self-similar problems and allows processors to agree (e. g. on pairings). For example, suppose a set of processors forms a cycle (each processor has two neighbors). Each processor must communicate once with each of its neighbors at the same time that neighbor is communicating with it. In a segment of the ring where processor ranks are monotonic, it's not clear how to do this pairing. One can use previous methods for coloring or maximum independent set or find problem-specific methods.

*Tree contraction* repeatedly collapses two neighboring nodes in a graph into a supernode until there is only one node left. This algorithm (and its inverse, re-expanding the graph) is part of parallel expression evalution algorithms and other parallel graph algorithms. General graph

contraction requires symmetry breaking. Any matching in the graph (a set of edges where no two edges share a vertex) can contract in parallel.

### 5.3.6 Asynchronous Termination

Proper termination can be tricky in asynchronous distributed-memory computations. Each process $p_i$ has a state $s_i$ that reflects whether the process is active or idle. Active processes can activate idle processes (e.g. by sending the idle process a message). The goal is to terminate when all processes are idle and there are no messages or other pending activations in the system. A process may start a *control wave* which visits all processes and determines whether all processors are idle. A control wave can be implemented by circulating a token sequentially through all processors, or with a tree-based reduction. The control wave also collects information about pending activations. For example, if only messages activate processors, the control wave can collect the number of message sends and receipts at each processor and verify that the totals match globally. However, having all processes idle and no pending activations as inspected by the control wave does not imply all these conditions held *simultaneously*. Processes can be reactivated "behind the back" of the wave, which makes correct termination difficult. For example, *aliasing* may occur when a message receipt substitutes for another. If a sender participates in the wave before it sends a message and the receiver participates after receipt, then only the receipt is recorded. Other messages (e.g. sent but truly undelivered) contribute only to the send count.

Mattern [76] discusses methods to ensure proper termination. PICO uses a variant of the four-counter method [75], which is well-suited to asynchronous contexts in which acknowledgments of "activation" are indirect. PICO uses multiple passes of idleness checks and message balance checks to confirm there is no remaining work in the system. The shared-memory B&B application uses condition variables in its termination procedure. Threads finding no active nodes, and thus wishing to terminate, go to sleep, but can either be awoken with additional work (new active nodes) or terminated by the last working thread.

## 5.4 Debugging Parallel Software

Debugging parallel software is notoriously difficult. Parallel software coordinates threads of execution across multiple physical processors. Thus parallel software often exhibits programming errors related to timing and synchronization that are not seen in serial codes. Perhaps the most common symptom of a software error is that a code "hangs" and fails to terminate. This occurs when a process is waiting for some event.

For example, a process may be waiting for a message that is never sent. This could happen if one process encounters an error condition that other processes do not encounter, thereby leading that process to interrupt its typical flow of communication with the other processes.

Unfortunately it is difficult to robustly reproduce these failures. A parallel bug may not be exposed in repeated executions of a code because of inherent nondeterminism (e.g. nondeterministic message delivery order). Some *race conditions* (order-dependent errors) are exposed only with a rare order of events. Similarly, parallel bugs sometimes disappear when code is inserted to track the state of each process. If such code disrupts the relative rate of computation, the synchronization condition that led to the failure may be difficult to reproduce.

One can develop and debug parallel MPI-based code on a single workstation by running multiple MPI processes on the same machine. These processes share the CPU as independent (communicating) threads. As mentioned later, this can help with debugging. However, because only one thread controls the CPU at once, a code that's fully debugged in this setting can still have bugs (race conditions, synchronization errors) associated with true concurrency.

The remainder of this section considers common approaches to parallel debugging. Specifically, we consider debugging modern programming languages (e.g. Fortran, C and C++) which have compilers and interactive debuggers.

### 5.4.1 Print Statements

Using print statements to trace interesting events is perhaps the simplest strategy for debugging software. However, there are several caveats for using them for parallel debugging. First, this technique can significantly impact the relative computation rates of processes in a parallel computation. Printing and especially file I/O are often very slow when compared to other computation tasks. Adding printing changes the behavior of asynchronous parallel programs so the precise error condition the debugger is tracking can disappear.

A second caveat for print-based debugging is that the order in which information is presented to a screen may not reflect the true sequence of events. For example, printing I/O for one process may be delayed while a buffer fills, allowing other processes' I/O to be displayed out of order. Explicitly flushing buffers (e.g. using `flush()`), can help, but even then communication delays can affect output ordering.

Finally, it is difficult (and at best inconvenient) to simultaneously display I/O from multiple processes especially for an execution using hun-

dreds to thousands of processors. Operating systems typically interleave the output of multiple processes, which can quickly lead to unintelligible output. One solution to this problem is to stream each process' I/O to a different file. In C, this can be done using `fprintf` statements with a different file descriptor for each process. More sophisticated solutions can be developed in C++ by exploiting the extensibility of stream operators. For example, we have developed a `CommonIO` class in the UTILIB C++ utility library [51] that provides new streams `ucout` and `ucerr`. These streams replace `cout` and `cerr` to control I/O in a flexible manner. The `CommonIO` class ensures that I/O streamed to `ucout` and `ucerr` is printed as a single block, and thus it is unlikely to be fragmented on a screen. Additionally, each line of the output can be tagged with a processor id and line number:

```
[2]-00002 Printing line 2 from process 2
[3]-00007 Printing line 7 from process 3
[3]-00008 Printing line 8 from process 3
[1]-00003 Printing line 3 from process 1
[1]-00004 Printing line 4 from process 1
[0]-00003 Printing line 3 from process 0
```

This facility makes it easy to extract and order output for each process.

### 5.4.2 Performance Analysis Tools

A natural generalization of print statements is the use of logging, trace analysis, and profiling tools that record process information throughout a parallel run. For example, the MPE library complements the MPI library, adding mechanisms for generating log files. MPE provides simple hooks for opening, closing, and writing to log files. Viewing these logs graphically via the Java-based Jumpshot program provides insight into the dynamic state of a parallel code [79, 123]. The commercial VAMPIR [114] tool provides more general facilities for trace analysis even for multi-threaded applications.

Profiling tools gather statistics about the execution of software. Standard profiling tools like `prof` and `gprof` run on workstations and clusters to identify what parts of the code are responsible for the bulk of the computation. Many commercial workstation vendors provide compilers that support configurable hardware counters which can gather fine-grain statistics about each physical processor's computation. For example, statistics of instruction and data cache misses at various levels of the memory hierarchy and of page faults measure cache performance. These profiling tools are particularly valuable for developing shared-memory

parallel software since they help identify synchronization bottlenecks that are otherwise difficult to detect.

### 5.4.3 Managing and Debugging Errors

A pervasive challenge for software development is the effective management of runtime error conditions (e.g. when an attempt to open a file fails). In parallel codes these include inter-process I/O failures. For example, message passing libraries like MPI include error handler mechanisms. Calls to the MPI library may return errors that indicate simple failure or fatal failure. The process should terminate for fatal failures.

In shared-memory and distributed-computing systems, inter-process communication failures often indicate more critical problems with the operating system or interconnection network. Consequently, it is reasonable to treat these as fatal errors. MPI-based software packages developed for tightly-coupled supercomputers support synchronous communication primitives. They typically assume the network is reliable so that messages sent by one process are received by another. Grid-computing systems usually require asynchronous communication. Inter-process communication failures may reflect problems with the grid computing infrastructure. Tools like Network Weather System (NWS) [84, 116, 118] help identify network-related problems.

One effective strategy for debugging unexpected failures is to generate an `abort()` when an error is detected. This generates a core file with debugging information. When the code runs on a single workstation (preferably with multiple processors), this debugging strategy generates a local core file that contains the call-stack at the point of failure. This debugging strategy can also work for exceptions, which step the computation out of the current context. For example, the UTILIB library includes an `EXCEPTION_MNGR` macro that throws exceptions in a controlled manner. In particular, one configuration of `EXCEPTION_MNGR` calls `abort()` for unexpected exceptions.

### 5.4.4 Interactive Debuggers

Most commercial workstation vendors provide compilers that naturally support interactive debugging of threaded software. Consequently, mature interactive debuggers are available for shared memory parallel software. There are specialized interactive debuggers for distributed or grid applications running on workstations or clusters. Commercial parallel debugging software tools like Etnus TotalView enable a user to interact with processes distributed across different physical processors [111]. Sun Microsystems offers advanced development, debugging, and profil-

ing tools within their commercial Sun Studio compiler suite and with their native MPI implementation in Sun HPC ClusterTools [107].

On workstations, standard interactive debuggers can attach to a running process given the process identifier. Thus one can debug parallel distributed-memory software by attaching a debugger to each active process. For example, each parallel process prints its process id using the `getpid()` system call and then (a) a master process waits on user I/O (e.g. waiting for a newline) and (b) other processes perform a blocking synchronization; the master performs blocking synchronization after performing I/O. After launching the parallel code, a user can attach debuggers to the given process ids, and then continue the computation. At that point, each debugged process is controlled by an interactive debugger. Setting up debuggers in this manner can be time consuming for more than a few parallel processes, but tools like LAM/MPI support the automatic launching of interactive `gdb` debuggers [18, 105].

## 5.5 Final Thoughts

Developing parallel scientific software is typically a challenging endeavor. The choice of target compute platform can significantly influence the manner in which parallelism is used, the algorithmic primitives used for parallelization, and the techniques used to debug the parallel software. Therefore, developing parallel software often requires a greater commitment of time and energy than developing serial counterparts. One must carefully consider the specific benefits of parallelization before developing parallel software. There are a number of issues that might influence the choice of target architecture for parallel software development:

**How large are problem instances? How much parallelism is needed?** Grid compute platforms deployed within large institutions will likely have a peak compute capacity greater than all but the largest super-computers. However, this scale of parallelization may not be needed in practice, or the granularity of an application might require tighter coupling of resources.

**What type of parallelization is needed?** If an application has a natural parallelization strategy, this influences the target compute platform. Distributed-memory architectures currently have the greatest capability to model large-scale physical systems (e.g. shock physics). HPC research within DOE has concentrated on tightly-coupled parallel codes on distributed-memory platforms. However, shared-memory platforms are emerging as the system of choice for moderate-sized problems that

demand unpredictable access to shared, irregular data structures (e.g. databases, and combinatorial problems).

**Who will use the tool, and what type of user support will the software need?** The majority of commercial vendors support only parallel software for shared-memory systems. The compute resources of such a system are well defined, and users need minimum training or support to use such parallel tools. For example, a threaded software tool is often transparent.

**What are the security requirements of user applications?** Security concerns are integral to virtually all business, medical, and government applications, where data has commercial, privacy, or national security sensitivities. The user has control/ownership of the resources and security policy in a distributed-memory system, unlike in a grid system. Even encryption may not provide enough security to run such applications on a grid.

In addition to the issues of decomposition, work granularity and load balancing discussed earlier, the following are algorithmic considerations when developing parallel software:

**Managing I/O:** I/O is often a bottleneck in parallel computations. Naïve parallelizations of serial software frequently contain hidden I/O (e.g. every processor writes to a log file). Although parallel I/O services are not always available in HPC resources, standards like MPI-IO were developed to meet this need.

**Parallel random number generators:** When using pseudo-random number generators in parallel, one must ensure the random streams across different processors are statistically independent and not correlated. This is particularly important for parallel computations, such as parallel Monte Carlo simulations, whose performance or correctness depends upon independent random values. Parallel Monte Carlo simulations are estimated to consume over half of all supercomputer cycles. Other applications that use randomized modeling or sampling techniques include sorting and selection [101, 52, 3]. Numerical libraries such as SPRNG [74] help to ensure this independence.

## Acknowledgments

This work was performed in part at Sandia National Laboratories. Sandia is a multipurpose laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000. Bader's research is supported in

part by NSF Grants CAREER ACI-00-93039, ITR ACI-00-81404, DEB-99-10123, ITR EIA-01-21377, Biocomplexity DEB-01-20709, and ITR EF/BIO 03-31654. We acknowledge the current and former University of New Mexico students who have contributed to the research described in this chapter, including Guojing Cong, Ajith Illendula (Intel), Sukanya Sreshta (OpNet), Nina R. Weisse-Bernstein, Vinila Yarlagadda (Intel), and Mi Yan. We also thank Jonathan Eckstein for his collaboration in the development of PICO.

## References

[1] K. M. Anstreicher and N. W. Brixius. A new bound for the quadratic assignment problem based on convex quadratic programming. *Mathematical Programming*, 89:341–357, 2001.

[2] K.M. Anstreicher, N.W. Brixius, J.-P. Goux, and J. Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming, Series B*, 91:563–588, 2002.

[3] D. A. Bader. An improved randomized selection algorithm with an experimental study. In *Proceedings of the 2nd Workshop on Algorithm Engineering and Experiments (ALENEX00)*, pages 115–129, San Francisco, CA, January 2000. `www.cs.unm.edu/Conferences/ALENEX00/`.

[4] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, April 2004.

[5] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, April 2004.

[6] D. A. Bader, A. K. Illendula, B. M. E. Moret, and N. Weisse-Bernstein. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In G.S. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, editors, *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE 2001)*, volume 2141 of *Lecture Notes in Computer Science*, pages 129–144, Århus, Denmark, 2001. Springer-Verlag.

[7] D. A. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999.

[8] D. A. Bader and B. M. E. Moret. GRAPPA runs in record time. *HPCwire*, 9(47), November 23, 2000.

[9] D. A. Bader, B. M. E. Moret, and P. Sanders. Algorithm engineering for parallel computation. In R. Fleischer, E. Meineche-Schmidt, and B. M. E. Moret, editors, *Experimental Algorithmics*, volume 2547 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 2002.

[10] D. A. Bader, B. M. E. Moret, and L. Vawter. Industrial applications of high-performance computing for phylogeny reconstruction. In H.J. Siegel, editor,

*Proceedings of SPIE Commercial Applications for High-Performance Computing*, volume 4528, pages 159–168, Denver, CO, 2001. SPIE.

[11] A. Bar-Noy, S. Guha, J. Naor, and B. Schieber. Approximating the throughput of multiple machines in real-time scheduling. *SIAM Journal on Computing*, 31(2):331–352, 2001.

[12] M. Benchouche, V.-D Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol. Building a parallel branch and bound library. In A. Ferreira and P. Pardalos, editors, *Solving Combinatorial Optimization Problems in Parallel:Methods and Techniques*, volume 1054 of *Lecture Notes in Computer Science*, pages 201–231. Springer-Verlag, 1996.

[13] R. H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, 2004.

[14] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, 1989.

[15] G. E. Blelloch. Prefix sums and their applications. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 35–60. Morgan Kaufman, San Mateo, CA, 1993.

[16] BOB++. `http://www.prism.uvsq.fr/~blec/Research/BOBO/`.

[17] N. W. Brixius and K. M. Anstreicher. Solving quadratic assignment problems using convex quadratic programming relaxations. *Optimization Methods and Software*, 16:49–68, 2001.

[18] G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[19] G. Calinescu, H. Karloff, and Y. Rabani. An improved approximation algorithm for multiway cut. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 48–52, 1998.

[20] A. Caprara. Formulations and hardness of multiple sorting by reversals. In *3rd Annual International Conference on Computational Molecular Biology (RECOMB99)*, pages 84–93. ACM, April 1999.

[21] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, Bowie, MD, May 1999.

[22] R. D. Carr and G. Konjevod. Polyhedral combinatorics. In H. J. Greenberg, editor, *Tutorials on Emerging Methodologies and Applications in Operations Research*. Kluwer Academic Press, 2004.

[23] H. Casanova and J. Dongarra. NetSolve: Network enabled solvers. *IEEE Computational Science and Engineering*, 5(3):57–67, 1998.

[24] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Academic Press, 2001.

[25] Q. Chen and M. C. Ferris. FATCOP: A fault tolerant Condor-PVM mixed integer programming solver. *SIAM Journal on Optimization*, 11(4):1019–1036, 2001.

[26] Computational Infrastructure for Operations Research, home page, 2004. `http://www-124.ibm.com/developerworks/opensource/coin/`.

[27] T. Coleman, J Czyzyk, C. Sun, M. Wager, and S. Wright. pPCx: Parallel software for linear programming. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[28] The Condor Project Homepage, 2004. `http://www.cs.wisc.edu/condor/`.

[29] G. Cong and D. A. Bader. The Euler tour technique and parallel rooted spanning tree. Technical report, Electrical and Computer Engineering Department, The University of New Mexico, Albuquerque, NM, February 2004.

[30] M. E. Cosner, R. K. Jansen, B. M. E. Moret, L.A. Raubeson, L.-S. Wang, T. Warnow, and S. Wyman. An empirical comparison of phylogenetic methods on chloroplast gene order data in Campanulaceae. In D. Sankoff and J. Nadeau, editors, *Comparative Genomics: Empirical and Analytical Approaches to Gene Order Dynamics, Map Alignment, and the Evolution of Gene Families*, pages 99–121. Kluwer Academic Publishers, Dordrecht, Netherlands, 2000.

[31] ILOG, CPLEX home page, 2004. `http://www.ilog.com/products/cplex/`.

[32] D. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[33] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and Andy White, editors. *The Sourcebook of Parallel Computing*. Morgan Kaufmann, 2002.

[34] R. V. Eck and M. O. Dayhoff. *Atlas of Protein Sequence and Structure*. National Biomedical Research Foundation, Silver Spring, MD, 1966.

[35] J. Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. *SIAM Journal on Optimization*, 4(4):794–814, 1994.

[36] J. Eckstein, W. E. Hart, and C. A. Phillips. PICO: An object-oriented framework for parallel branch-and-bound. In *Inherently Parallel Algorithms in Feasibility and Optimization and Their Applications*, Elsevier Scientific Series on Studies in Computational Mathematics, pages 219–265, 2001.

[37] D. P. Faith. Distance method and the approximation of most-parsimonious trees. *Systematic Zoology*, 34:312–325, 1985.

[38] J. S. Farris. Estimating phylogenetic trees from distance matrices. *The American Naturalist*, 106:645–668, 1972.

[39] M. Fellows. Parameterized complexity. In R. Fleischer, E. Meineche-Schmidt, and B. M. E. Moret, editors, *Experimental Algorithmics*, volume 2547 of *Lecture Notes in Computer Science*, pages 51–74. Springer-Verlag, 2002.

[40] J. Felsenstein. PHYLIP – phylogeny inference package (version 3.2). *Cladistics*, 5:164–166, 1989.

[41] W. M. Fitch. Toward defining the course of evolution: Minimal change for a specific tree topology. *Systematic Zoology*, 20:406–416, 1971.

[42] S. Fortune and J. Willie. Parallelism in random access machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[43] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of SuperComputer Applications*, 11(2):115–128, 1997.

[44] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[45] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organization. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[46] B. Gendron and T. G. Crainic. Parallel branch and bound algorithms: Survey and synthesis. *Operations Research*, 42:1042–1066, 1994.

[47] P. A. Goloboff. Analyzing large data sets in reasonable times: Solutions for composite optima. *Cladistics*, 15:415–428, 1999.

[48] J.-P. Goux, S. Kulkarni, J. T. Linderoth, and M. E. Yoder. Master-Worker: An enabling framework for applications on the computational grid. *Cluster Computing*, 4:63–70, 2001.

[49] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *An Introduction to Parallel Computing, Design and Analysis of Algorithms*. Addison-Wesley, second edition, 2003.

[50] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Legion: An operating system for wide-area computing. Technical report, 1999. Available at `http://legion.virginia.edu/papers/CS-99-12.ps.Z`.

[51] W. E. Hart. UTILIB user manual version 1.0. Technical Report SAND2001-3788, Sandia National Laboratories, 2001. Available for download at `http://software.sandia.gov/Acro/UTILIB/`.

[52] D. R. Helman, D. A. Bader, and J. JáJá. A randomized parallel sorting algorithm with an experimental study. *Journal of Parallel and Distributed Computing*, 52(1):1–23, 1998.

[53] D. R. Helman and J. JáJá. Sorting on clusters of SMP's. In *Proceedings of the 12th International Parallel Processing Symposium*, pages 1–7, Orlando, FL, March/April 1998.

[54] D. R. Helman and J. JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *Algorithm Engineering and Experimentation*, volume 1619 of *Lecture Notes in Computer Science*, pages 37–56, Baltimore, MD, January 1999. Springer-Verlag.

[55] B. Hendrickson and R. Leland. The Chaco user's guide: Version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, 1994.

[56] M. D. Hendy and D. Penny. Branch and bound algorithms to determine minimal evolutionary trees. *Mathematical Biosciences*, 59:277–290, 1982.

[57] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, New York, 1992.

[58] H. F. Jordan and G. Alaghband. *Fundamentals of Parallel Processing*. Prentice Hall, 2003.

[59] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.

[60] G. Karypis and V. Kumar. *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. Department of Computer Science, University of Minnesota, version 4.0 edition, September 1998.

[61] L. Khachian. A polynomial time algorithm for linear programming. *Soviet Mathematics, Doklady*, 20:191–194, 1979.

[62] S. Kleiman, D. Shah, and B. Smaalders. *Programming with Threads*. Prentice Hall, Englewood Cliffs, NJ, 1996.

[63] L. Ladányi. BCP (Branch, Cut, and Price). Available from `http://www-124.ibm.com/developerworks/opensource/coin/`.

[64] T.-H. Lai and S. Sahni. Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM*, 27:594–602, 1984.

[65] T.-H. Lai and A. Sprague. Performance of parallel branch-and-bound algorithms. *IEEE Transactions on Computing*, C-34:962–964, 1985.

[66] G.-J. Li and B. Wah. Coping with anomalies in parallel branch-and-bound algorithms. *IEEE Transactions on Computing*, C-35:568–573, 1986.

[67] G.-J. Li and B. Wah. Computational efficiency of parallel combinatorial OR-tree searches. *IEEE Transactions on Software Engineering*, 18:13–31, 1990.

[68] W.-H. Li. Simple method for constructing phylogenetic trees from distance matrices. *Proceedings of the National Academy of Sciences USA*, 78:1085–1089, 1981.

[69] J. T. Linderoth. *Topics in Parallel Integer Optimization*. PhD thesis, Georgia Institute of Technology, Department of Industrial and Systems Engineering, 1998.

[70] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP*, 11, 1997. Available from `http://www.cs.wisc.edu/condor/doc/htc_mech.ps`.

[71] Mallba library v2.0. `http://neo.lcc.uma.es/mallba/easy-mallba/`.

[72] F. Margot. BAC: A BCP based branch-and-cut example. Technical Report RC22799, IBM, 2003.

[73] A. Marzetta. *ZRAM*. PhD thesis, ETH Zurich, Institute of Theoretical Computer Science, 1998.

[74] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*, 26:436–461, 2000. Available for download from `http://sprng.cs.fsu.edu/`.

[75] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[76] F. Mattern. Distributed termination detection with sticky state indicators. Technical report, 200/90, Department of Computer Science, University of Kaiserslautern, Germany, 1990.

[77] R. Miller and L. Boxer. *Algorithms Sequential and Parallel: A Unified Approach*. Prentice Hall, 2000.

[78] B. M. E. Moret, S. Wyman, D. A. Bader, T. Warnow, and M. Yan. A new implementation and detailed study of breakpoint analysis. In *Proceedings of the 6th Pacific Symposium on Biocomputing*, pages 583–594, Hawaii, 2001.

[79] Performance visualization for parallel programs, 2004. `http://www-unix.mcs.anl.gov/perfvis/publications/index.htm`.

[80] M. Nediak and J. Eckstein. Pivot, cut, and dive: A heurstic for mixed 0-1 integer programming. Technical Report RRR 53-2001, RUTCOR, October 2001.

[81] M. Nei and S. Kumar. *Molecular Evolution and Phylogenetics*. Oxford University Press, Oxford, UK, 2000.

[82] K.C. Nixon. The parsimony ratchet, a new method for rapid parsimony analysis. *Cladistics*, 15:407–414, 1999.

[83] C. E. Nugent, T. E. Vollman, and J. Ruml. An experimental comparison of techniques for the assignment of facilities to locations. *Operations Research*, pages 150–173, 1968.

[84] Network Weather Service WWW Page, 2004. `http://nws.cs.ucsb.edu/`.

[85] OpenMP Architecture Review Board. OpenMP: A proposed industry standard API for shared memory programming. `www.openmp.org`, October 1997.

[86] IBM Optimization Solutions and Library, home page, 2004. `http://www-306.ibm.com/software/data/bi/osl/`.

[87] I. Pe'er and R. Shamir. The median problems for breakpoints are NP-complete. Technical Report 71, Electronic Colloquium on Computational Complexity, November 1998.

[88] C. Phillips, C. Stein, and J. Wein. Minimizing average completion time in the presence of release dates. *Mathematical Programming B*, 82:199–223, 1998.

[89] POSIX. *Information technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*. Portable Applications Standards Committee of the IEEE, 1996-07-12 edition, 1996. ISO/IEC 9945-1, ANSI/IEEE Std. 1003.1.

[90] J. Pruyne and M. Livny. Interfacing Condor and PVM to harness the cycles of workstation clusters. *Journal on Future Generation of Computer Systems*, 12(53-65), 1996.

[91] P. W. Purdom, Jr., P. G. Bradford, K. Tamura, and S. Kumar. Single column discrepancy and dynamic max-mini optimization for quickly finding the most parsimonious evolutionary trees. *Bioinfomatics*, 2(16):140–151, 2000.

[92] PVM: Parallel Virtual Machine, 2004. `http://www.csm.ornl.gov/pvm/pvm_home.html`.

[93] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

[94] T. K. Ralphs. Symphony 4.0 users manual, 2004. Available from `www.branchandcut.org`.

[95] T. K. Ralphs, L. Ladányi, and M. J. Saltzman. Parallel branch, cut, and price for large-scale discrete optimization. *Mathematical Programming*, 98(1-3), 2003.

[96] T. K. Ralphs, L. Ladányi, and M. J. Saltzman. A library for implementing scalable parallel search algorithms. *The Journal of SuperComputing*, 28(2):215–234, 2004.

[97] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993.

[98] K. Rice and T. Warnow. Parsimony is hard to beat. In *Computing and Combinatorics*, pages 124–133, August 1997.

[99] N. Saitou and M. Nei. The neighbor-joining method: A new method for reconstruction of phylogenetic trees. *Molecular Biological and Evolution*, 4:406–425, 1987.

[100] D. Sankoff and M. Blanchette. Multiple genome rearrangement and breakpoint phylogeny. *Journal of Computational Biology*, 5:555–570, 1998.

[101] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.

[102] Y. Shinano, M. Higaki, and R. Hirabayashi. Generalized utility for parallel branch and bound algorithms. In *Proceedings of the Seventh Symposium on Parallel and Distributed Processing*, 1995.

[103] D. Skillicorn and D. Talia. Models and languages for parallel computation. *Computing Surveys*, 30(2):123–169, 1998.

[104] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Inc., second edition, 1998.

[105] J. M. Squyres and A. Lumsdaine. A component architecture for LAM/MPI. In *Proceedings of the 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, Venice, Italy, September / October 2003. Springer-Verlag.

[106] J. A. Studier and K. J. Keppler. A note on the neighbor-joining method of Saitou and Nei. *Molecular Biological and Evolution*, 5:729–731, 1988.

[107] Sun Microsystems, Inc., WWW Page, 2004. `www.sun.com`.

[108] Sun Microsystems, Inc. POSIX threads. WWW page, 1995. `www.sun.com/developer-products/sig/threads/posix.html`.

[109] D. L. Swofford and D. P. Begle. *PAUP: Phylogenetic analysis using parsimony.* Sinauer Associates, Sunderland, MA, 1993.

[110] D. L. Swofford, G. J. Olsen, P. J. Waddell, and D. M. Hillis. Phylogenetic inference. In D. M. Hillis, C. Moritz, and B. K. Mable, editors, *Molecular Systematics*, pages 407–514. Sinauer, Sunderland, MA, 1996.

[111] Etnus, L.L.C. TotalView WWW Page, 2004. `http://www.etnus.com/Products/TotalView/index.html`.

[112] S. Tschöke and T. Polzer. Portable parallel branch-and-bound library, PPBB-Lib, user manual, library version 2.0. Technical report, University of Paderborn Department of Computer Science, 1996.

[113] L. G. Valiant. A bridging model for parallel computation. *Commununications of the ACM*, 33(8):103–111, 1990.

[114] Intel GmbH VAMPIR WWW Page, 2004. `http://www.pallas.com/e/products/vampir/`.

[115] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, second edition, 2004.

[116] R. Wolski. Dynamically forecasting network performance using the Network Weather Service. *Journal of Cluster Computing*, 1:119–132, January 1998.

[117] R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring, and A. Su. Running Everyware on the computational grid. In *SC99 Conference on High Performance Computing*, 1999. Available from `http://www.cs.utk.edu/~rich/papers/ev-sc99.ps.gz`.

[118] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5–6):757–768, 1999.

[119] Dash Optimization, XPRESS-MP, 2004. `http://www.dashoptimization.com/`.

[120] M. Yan. *High Performance Algorithms for Phylogeny Reconstruction with Maximum Parsimony*. PhD thesis, Electrical and Computer Engineering Department, University of New Mexico, Albuquerque, NM, January 2004.

[121] M. Yan and D. A. Bader. Fast character optimization in parsimony phylogeny reconstruction. Technical report, Electrical and Computer Engineering Department, The University of New Mexico, Albuquerque, NM, August 2003.

[122] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.

[123] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, 1999.