Solving Optimization Problems on Computational Grids

Stephen J. Wright*

November 21, 2000

1 Multiprocessors and Computational Grids

Multiprocessor computing platforms, which have become available more and more widely since the mid-1980s, are now heavily used by organizations that need to solve very demanding computational problems. There has also been a great deal of research on computational techniques that are suited to these platforms, and on the software infrastructure needed to compile and run programs on them. Parallel computing is now central to the culture of many research communities, in such areas as meteorology, computational physics and chemistry, and cryptography. Nevertheless, fundamental research in numerical techniques for these platforms remains a major topic of investigation in numerical PDE and numerical linear algebra.

The nature of parallel platforms has evolved rapidly during the past 15 years. The eighties saw a profusion of manufacturers (Intel, Denelcor, Alliant, Thinking Machines, Convex, Encore, Sequent, among others) with a corresponding proliferation of architectural features: hypercube, mesh, and ring topologies; shared and distributed memories; memory hierarchies of different types; butterfly switches; and global buses. Compilers and runtime support tools were machine specific and idiosyncratic. Argonne's Advanced Computing Research Facility kept a zoo of these machines during the late 1980s, allowing free access to many researchers in the United States and giving many of us our first taste of this brave new world.

By the early 1990s, the situation had started to change and stabilize. Most of the vendors went out of business, and their machines gradually were turned off—one processor at a time, in some cases. Architectures gravitated toward two easily understood paradigms that prevail today. One paradigm is the shared-memory model typified by the SGI Origin series and by computers manufactured by Sun and Hewlett-Packard. The other paradigm, typified by the IBM SP series, can be viewed roughly as a uniform collection of processors, each with its own memory and all able to pass messages to one another at a rate roughly independent of the locations of the two processors involved. On the software side, the advent of software tools such as p4, MPI, and PVM allowed users to write code that could be compiled and executed without alteration on the machines of different manufacturers, as well as on networks of workstations.

The optimization community was quick to take advantage of parallel computers. In designing optimization algorithms for these machines, it was best in some cases to exploit parallelism at a lower level (that is, at the level of the linear algebra or the function/derivative evaluations) and leave the control flow of the optimization algorithm essentially unchanged. Other cases required a complete rethinking

*Mathematics and Computer Science Division, Argonne National Laboratory, and Computer Science Department, University of Chicago.

of the algorithms, to allow simultaneous exploration of different regions of the solution space, different parts of the branch-and-bound tree, or different candidates for the next iterate. Novel parallel approaches were developed for global optimization, network optimization, and direct-search methods for nonlinear optimization. Activity was particularly widespread in parallel branch-and-bound approaches for various problems in combinatorial and network optimization, as a brief Web search can attest.

As the cost of personal computers and low-end workstations has continued to fall, while the speed and capacity of processors and networks have increased dramatically, "cluster" platforms have become popular in many settings. Though the software infrastructure has yet to mature, clusters have made supercomputing inexpensive and accessible to an even wider audience.

A somewhat different type of parallel computing platform known as a *computational grid* (alternatively, *metacomputer*) has arisen in comparatively recent times [1]. Broadly speaking, this term refers not to a multiprocessor with identical processing nodes but rather to a heterogeneous collection of devices that are widely distributed, possibly around the globe. The advantage of such platforms is obvious: they have the potential to deliver enormous computing power. (A particular type of grid, one made up of unused compute cycles of workstations on a number of campuses, has the additional advantage of costing essentially nothing.) Just as obviously, however, the complexity of grids makes them very difficult to use. The software infrastructure and the applications programs that run on them must be able to handle the following features:

- heterogeneity of the various processors in the grid;
- the dynamic nature of the platform (the pool of processors available to the user may grow and shrink during the computation);
- the possibility that a processor performing part of the computation may disappear without warning;
- latency (that is, time for communications between the processors) that is highly variable but often slow.

In many applications, however, the potential power and/or low cost of computational grids make the effort of meeting these challenges worthwhile. The Condor team, headed by Miron Livny at the University of Wisconsin, were among the pioneers in providing infrastructure for grid computations. The Condor system can be used to assemble a parallel platform from workstations, PC clusters, and multiprocessors and can be configured to use only "free" cycles on these machines, sharing them with their respective owners and other users. More recently, the Globus project has developed technologies to support computations on geographically distributed platforms consisting of high-end computers, storage and visualization devices, and other scientific instruments.

In 1997, we started the metaneos project as a collaborative effort between optimization specialists and the Condor and Globus groups. Our aim was to address complex, difficult optimization problems in several areas, designing and implementing the algorithms and the software infrastructure needed to solve these problems on computational grids. A coordinated effort on both the optimization and the computer science sides was essential. The existing Condor and Globus tools were inadequate for direct use as a base for programming optimization algorithms, whose control structures are inevitably more complex than those required for task-farming applications. Many existing parallel algorithms for optimization were "not parallel enough" to exploit the full power of typical grid platforms. Moreover, they were often "not asynchronous enough," in that they required too much communication between tasks to execute efficiently on platforms with the heterogeneity and communications latency properties of our target platforms. A further challenge was that, in contrast to other grid applications, the computational resources required to solve an optimization problem often cannot be predicted with much confidence, making it difficult to assemble and utilize these resources effectively.

This article describes some of the results we have obtained during the first three years of the metaneos project. Our efforts have led to development of the runtime support library MW, for implementing algorithms with master-worker control structure on Condor platforms. This work is discussed below, along with our work on algorithms and codes for integer linear programming, the quadratic assignment problem, and stochastic linear programming. Other metaneos work, not discussed below, includes work in global optimization, integer nonlinear programming, and verification of solution quality for stochastic programming.

2 Condor, Globus, and the MW Framework

The Condor system [2, 3] had its origins at the University of Wisconsin in the 1980s. It focuses on collections of computing resources, known as *Condor pools*, that are distributively owned. To understand the implications of "distributed ownership," consider a typical machine in a pool: a workstation on the desk of a researcher. The Condor system provides a means by which other users (not known to the machine's owner) can exploit some of the unused cycles on the machine, which otherwise would sit idle most of the time. The owner maintains control over the access rights of Condor to his machine, specifying the hours in which Condor is allowed to schedule processes on the machine and the conditions under which Condor must terminate any process it is running when the owner starts a process of his own. Whenever Condor needs to terminate a process under these conditions, it migrates the process to another machine in the pool, guaranteeing eventual completion.

When a user submits a process, the Condor system finds a machine in the pool that matches the software and hardware requirements of the user. Condor executes the user's process on this machine, trapping any system calls made by the process (such as input/output operations) and referring them back to the submitting machine. In this way, Condor preserves much of the submitting machine's environment on the execution machine. Users can submit a large number of processes to the pool at once. Since each such process maintains contact with the submitting machine, this feature of Condor opens up the possibility of parallel processing. Condor provides an *opportunistic* environment, one that can make use of whatever resources currently are available in its pool. This set of resources grows and shrinks dynamically during execution of the user's job, and his algorithm should be able to exploit this situation.

The Globus Toolkit [4] is a set of components that can be used to develop applications or programming tools for computational grids. Currently, the Toolkit contains tools for resource allocation management and resource discovery across a grid, security and authentication, data movement, message-passing communication, and monitoring of grid components. The main use of Globus within the metaneos project has been at a level below Condor. By a Globus mechanism known as *glide-in*, a user can add machines at a remote location into the Condor pool on a temporary basis, making them accessible only to his own processes. In this way, a user can marshal a large and powerful set of resources over multiple sites, some or all of them dedicated exclusively to his job.

MW is a software framework that facilitates implementation of algorithms of master-worker type on computational grids. It was developed as part of the metaneos project by Condor team members Mike Yoder and Sanjeev Kulkarni in collaboration with optimization specialists Jeff Linderoth and Jean-Pierre Goux [5, 6]. MW takes the form of a set of C++ abstract classes, which the user implements to perform the particular operations associated with his algorithm and problem class. There are just ten virtual functions, grouped into the following three fundamental base classes:

- **MWDriver** contains four functions that obtain initial user information, set up the initial set of tasks, pack the data required to initialize each worker processor as it becomes available, and act on the results that are returned to the master when a task is completed.
- **MWWorker** contains two functions, to unpack the initialization data for the worker and to execute a task sent by the master.
- **MWTask** contains four functions to pack and unpack the data defining a single task and to pack and unpack the results associated with that task.

MW also contains functions that monitor performance of the grid and gather various statistics about the run.

Internally, MW works by managing a list of workers and a list of tasks. The resource management mechanisms of the underlying grid are used to obtain new workers for the list and provide information about each worker. The information can be used to order the worker list so that the most suitable workers (e.g., the fastest machines) are at the head of the list and hence are the first to receive tasks. Similarly, the task list can be ordered by a user-defined key to ensure that the most important tasks are performed first. Scheduling of tasks to workers then becomes simple: The first task on the list is assigned to the first available worker. New tasks are added to the list by the master process in response to results received from completion of an earlier task.

MW is currently implemented on two slightly different grid platforms. The first uses Condor's version of the PVM (parallel virtual machine) protocol, while the second uses the remote I/O features of Condor to allow master and workers to communicate via series of shared files. In addition, MW provides a "bottom-level" interface that allows it to be implemented in other grid computing toolkits.

3 Integer Programming

Consider the linear mixed integer programming problem

$$\min c^T x \text{ subject to } Ax \le b,\ l \le x \le u,$$
$$x_i \in Z, \text{ for all } i \in I,$$

where x is a vector of length n, Z represents the integers, and $I \subset \{1, 2, \ldots, n\}$. Parallel algorithms and frameworks for this problem have been investigated by a number of authors in recent times. The approaches described in [7, 8, 9, 10] implement enhancements of the branch-and-bound procedure, in which the work of exploring the branch-and-bound tree is distributed among a fixed number of processors. These approaches are differentiated by their use of virtual-shared-memory vs. message-passing models, their load balancing procedures, their choice of branching rules, and their use of cuts.

By contrast, the FATCOP code of Chen, Ferris, and Linderoth [11] uses the MW framework to greedily use whatever computational resources become available from the Condor pool. The algorithm implemented in FATCOP (the name is a loose acronym for "fault-tolerant Condor PVM") is an enhanced branch-and-bound procedure that utilizes (globally valid) cuts, pseudocosts for branching, preprocessing at nodes within the branch-and-bound tree, and heuristics to identify integer feasible solutions rapidly.

In FATCOP's master-worker algorithm, each task consists of exploration of a subtree of the branch-and-bound tree, not just evaluation of a single node of the tree. Given a root node for the subtree, and other information such as the global cut and pseudocost pools, the task executes for a given amount of time, making its own branching decisions and accumulating its own collection of cuts and pseudocosts. It may also perform a "diving" heuristic from its root node to seek a new integer feasible solution. When the task is complete, it returns to the master a stack representing the unexplored portions of its subtree. (Depth-first search is used to limit the size of this stack.) The task also sends back any new cut and pseudocost information it generated, which is added to the master's global cut and pseudocost pools.

By processing a subtree rather than a single node, FATCOP increases the granularity of the task and improves utilization of the computational power of each worker. The time for which a processor is sitting idle while waiting for the task information to be sent to and from the master, and for the master to process its results and assign it a new task, is generally small relative to the computation time.

The master process is responsible for maintaining the task pool, as well as the pools of cuts and pseudocosts. It recognizes new workers as they join the computation pool, and send them copies of the problem data together with the current cut and pseudocost pools. Moreover, it sends tasks to these workers and processes the results of these tasks by updating its pools and possibly its incumbent solution. Another important function of the master is to detect when a machine has disappeared from the worker pool. In this case, the task that was occupying that machine is lost, and the master must assign it to another worker. (This is the "fault tolerant" feature that makes FATCOP fat!) On long computations, the master process "checkpoints" by writing out the current state of computation to disk. By doing so, it can restart the computation at the latest checkpoint after a crash of the master processor.

Table 1: Performance of FATCOP on gesa2_o

	$\bar{P}$	Nodes	Time
minumum	43.2	6207993	6951
average	62.5	8214031	10074
maximum	86.3	9693518	13198

To illustrate FATCOP's performance, we consider the solution of the problem gesa2_o from the MIPLIB test set. This problem arises in an electricity generation application in the Balearic Islands. FATCOP was run ten times on the Condor pool at the University of Wisconsin. Because of the dynamic computational environment—the size and composition of the pool of available workers and communication times on the network varied between runs and during each run—the search pattern followed by FATCOP was quite different in each instance, and different from what one would obtain from a serial implementation of the same approach. However, using the statistical features of MW, we can find the average number of workers used during each run, defined as

$$\bar{P} = \sum_{k=1}^{\infty} k\tau_k / T,$$

where τ_k is the total time during which the run had control of k processors, and T is the wall clock time for the run. The minimum, maximum, and mean values of $\bar{P}$ over the ten runs are shown in Table 1. This table also shows statistics for the number of nodes evaluated by FATCOP, and the wall clock times.

Figure 1 profiles the size of the worker pool during a particular run. Note the sharp dip,

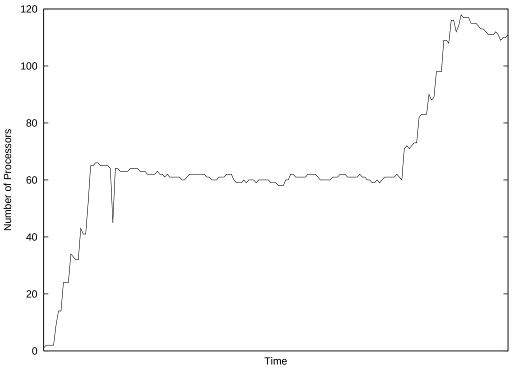

Figure 1: Number of Workers during FATCOP on gesa2_o

which occurred when a set of machines participated in a daily backup procedure, and the gradual buildup during the end of the run, which occurred in the late afternoon when more machines became available as their owners went home.

For a detailed performance analysis of FATCOP, see [11].

A separate but related activity involved solving to optimality the well-known *seymour* problem from the MIPLIB library of integer programs. This well known integer programming problem, posed by Paul Seymour, arises in a new proof [12] of the famous Four-Color Theorem, which states that any map can be colored using four colors in such a way that regions sharing a boundary segment receive different colors. The *seymour* problem is to find the smallest set of configurations such the Four-Color Theorem is true if none of these configurations can exist in a minimal counterexample. Although Seymour claimed to have found a solution with objective value 423, nobody (including Seymour himself) had been able either to reproduce this solution, or prove a strong lower bound on the optimal value. There was some scepticism in the integer programming community as to whether this was indeed the optimal value.

In July 2000, a team of metaneos researchers found solutions with the value 423, and proved its optimality. Gabor Pataki, Stefan Schmieta, and Sebastian Ceria at Columbia used preprocessing, disjunctive cuts and branch-and-bound to break down the problem into a list of 256 integer programs. Michael Ferris at Wisconsin and Jeff Linderoth at Argonne joined the Columbia group in working through this list. The problems were solved separately with the help of the Condor system, using the integer programming packages CPLEX and XPRESS-MP. About 9000 hours of CPU time was needed, the vast majority of it spent in proving optimality.

4 Quadratic Assignment Problem

The quadratic assignment problem (QAP) is a problem in location theory that has proved to be among the most difficult combinatorial optimization problems to solve in practice. Given $n \times n$ matrices A, B, and C, where $A_{i,j}$ represents the flow between facilities i and j, $B_{i,j}$ is the distance between locations i and j, and $C_{i,j}$ is the fixed cost of assigning facility i to location j, the problem is to find the permutation $\{\pi(1), \pi(2), \ldots, \pi(n)\}$ of the index set $\{1, 2, \ldots, n\}$ that minimizes the following objective:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} A_{i,j} B_{\pi(i),\pi(j)} + \sum_{i=1}^{n} C_{i,\pi(i)}.$$

An alternative matrix-form representation is as follows:

$$\begin{aligned} \mathrm{QAP}(A,B,C)\text{:}\ \ & \min \mathrm{tr}(AXB + C)X^T, \\ & \text{s.t. } X \in \Pi, \end{aligned}$$

where $\mathrm{tr}(\cdot)$ represents the trace and Π is the set of $n \times n$ permutation matrices.

The practical difficulty of solving instances of the QAP to optimality grows rapidly with n. As recently as 1998, only the second-largest problem ($n = 25$) from the standard "Nugent" benchmark set [13] had been solved, and this effort required a powerful parallel platform [14]. In June 2000, a team consisting of Kurt Anstreicher and Nate Brixius (University of Iowa) and metaneos investigators Jean-Pierre Goux and Jeff Linderoth solved the largest of the Nugent problems—the $n = 30$ instance known as nug30—verifying that a solution obtained earlier from a heuristic was optimal [15]. They devised a branch-and-bound algorithm based on a convex quadratic programming relaxation of QAP, implemented it using MW, and ran it on a Condor-based computational grid spanning eight institutions. The computation used over 1000 worker processors at its peak, and ran for a week. It was solving linear assignment problems (the core computational operation in the algorithm) at the rate of nearly a million per second during this period.

In the remainder of this section, we outline the various theoretical, heuristic, and computational ingredients that combined to make this achievement possible.

The convex quadratic programming (QP) relaxation of QAP proposed by Anstreicher and Brixius [16] yields a lower bound on the optimal objective that is tighter than alternative bounds based on projected eigenvalues or linear assignment problems. Just as important, an approximate solution to the QP relaxation can be found at reasonable cost by applying the Frank-Wolfe method for quadratic programming. Each iteration of this method requires only the solution of a dense linear assignment problem—an inexpensive operation. Hence, the Frank-Wolfe method is preferable in this context to more sophisticated quadratic programming algorithms; its slow asymptotic convergence properties are not important because only an approximate solution is required

The QAP is solved by embedding the QP relaxation scheme in a branch-and-bound strategy. At each node of the branch-and-bound tree, some subset of the facilities is assigned to certain locations—in the nodes at level k of the tree, exactly k such assignments have been made. At a level-k node, a reduced QAP can be formulated in which the unassigned part of the permutation (which has $n - k$ components) is the unknown. The QP relaxation can then be used on this reduced QAP to find an approximate lower bound on its solution, and therefore on the cost of all possible permutations that include the k assignments already made at this node. If the bound is greater than the cost of the best permutation found to date (the incumbent), the subtree rooted at this node can be discarded. Otherwise, we need to decide whether and how to branch from this node.

Branching is performed by choosing a facility and assigning it to each location in turn (row branching) or by choosing a location and assigning each of the remaining facilities to it in turn (column branching). However, it is not always necessary to examine all possible $n - k$ children of a level-k node; some of them can be eliminated immediately by using information from the dual of the QP relaxation. In fact, one rule for deciding the next node from which to branch at level k of the tree is to choose the node that yields the fewest children. A more expensive branching rule, using a *strong branching* technique, is employed near the root of the tree (k smaller). Here, the consequence of fixing each one of a collection of promising facilities (or locations) is evaluated by provisionally making the assignment in question and solving the corresponding QP relaxation. Estimates of lower bounds are then obtained for the grandchildren of the current node, and these are summed. The node for which this summation is largest is chosen as the branching facility (location).

The branching rule and the parameters that govern the execution of the branching rule are chosen according to the level in the tree and also the *gap*, which measures the closeness of the lower bound at the current node to the incumbent objective. When the gap is large at a particular node, it is likely that exploration of the subtree rooted at this node will be a costly process. Use of a more elaborate (and expensive) branching rule tends to ensure that exploration of unprofitable parts of the subtree is avoided, thereby reducing overall run time.

Parallel implementation of the branch-and-bound technique uses an approach not unlike the FATCOP code for integer programming. Each worker is assigned the root node of a subtree to explore, in a depth-first fashion, for a given amount of time. When its time expires, it returns unexplored nodes from its subtree to the master, together with any new incumbent information. The pool of tasks on the master is ordered by the gap, so that nodes with smaller gaps (corresponding to subtrees that should be less difficult to explore) are assigned first. To reduce the number of easy tasks returned to the master, a "finish-up" heuristic permits a worker extra time to explore its subtree if its gap becomes small.

Exploitation of the symmetries that are present in many large QAPs is another important factor in making solution of nug30 and other large problems a practical proposition. Such symmetries arise when the distance matrix is derived from a rectangular grid. Symmetries can be used, for instance, to decrease the number of child nodes that need to be formed (to considerably fewer than $n-k$ children at a level-k node).

Prediction of the performance profile of a run is also important in tuning algorithmic parameters and in estimating the amount of computational resources needed to tackle the problem. An estimation procedure due to Knuth was enhanced to allow prediction of the number of nodes that need to be evaluated at each level of the branch-and-bound tree, for a specific problem and specific choices of the algorithmic parameters.

Figure 2 shows the number of workers used during the course of the nug30 run in early June 2000. As can be seen from this graph, the run was halted five times—twice because of failures in the resource management software and three times for system maintenance—and restarted each time from the latest master checkpoint.

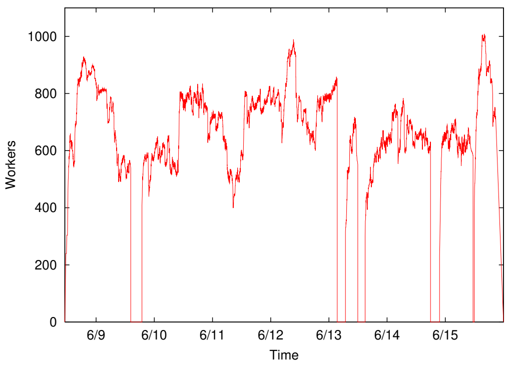

Figure 2: Number of Workers during nug30 Computation

In the weeks since the nug30 computation, the team has solved three more benchmark problems of size $n = 30$ and $n = 32$, using even larger computational grids. Several outstanding problems of size $n = 36$ derived from a backboard wiring application continue to stand as a challenge to this group and to the wider combinatorial optimization community.

5 Stochastic Programming

The two-stage stochastic linear programming problem with recourse can be formulated as follows:

$$\begin{array}{c}\min_x \; \mathcal{Q}(x) \stackrel{\text{def}}{=} c^T x + \sum_{i=1}^N p_i \mathcal{Q}_i(x)\\ \text{subject to } \; Ax = b, \;\; x \geq 0,\end{array}$$

where

$$\mathcal{Q}_i(x) \stackrel{\text{def}}{=} \min_{y_i} q_i^T y_i \text{ s.t. } W y_i = h_i - T_i x, \;\; y_i \geq 0.$$

The uncertainty in this formulation is modeled by the data triplets (h_i, T_i, q_i), $i = 1, 2, \ldots, N$, each of which represents a possible *scenario* for the uncertain data (h, T, q). Each p_i represents the probability that scenario i is the one that actually happens; these quantities are nonnegative and sum to 1. The quantities p_i, $i = 1, 2, \ldots, N$ are nonnegative and sum to 1; p_i is the probability that scenario i is the true one.

The two-stage problem with recourse represents a situation in which one set of decisions (represented by the first-stage variables x) must be made at the present time, while a second set of decisions (represented by y_i, $i = 1, 2, \ldots, N$) can be deferred to a later time, when the uncertainty has been resolved and the true second-stage scenario is known. The objective function represents the *expected* cost of x, integrated over the probability distribution for the uncertain part of the model.

In many practical problems, the number of possible scenarios N either is infinite (that is, the probability distribution is continuous) or is finite but much too large to allow practical solution of the full problem. In these instances, sampling is often used to obtain an approximate problem with fewer scenarios.

Decomposition algorithms are well suited to grid platforms, because they allow the computations associated with the N second-stage scenarios to be performed independently and require only modest amounts of data movement between processors. These algorithms view $\mathcal{Q}$ as a piecewise linear, convex function of the variables x, whose subgradient is given by the formula

$$\partial \mathcal{Q}(x) = c + \sum_{i=1}^{N} p_i \partial \mathcal{Q}_i(x).$$

Evaluation of each function $\mathcal{Q}_i$, $i = 1, 2, \ldots, N$ requires solution of the linear program in y_i given above. One element of the subgradient $\partial \mathcal{Q}_i(x)$ of this function can be calculated from the dual solution of this linear program.

In the metaneos project, Linderoth and Wright [17] have implemented a decomposition algorithm based on techniques from nonsmooth optimization and including various enhancements to lessen the need for the master and workers to synchronize their efforts. In the remainder of this section, we outline in turn the trust-region algorithm ATR and its convergence properties, implementation of this algorithm on the Condor grid platform using MW, and our "asynchronous" variant.

The ATR algorithm progressively builds up a piecewise-linear model function $M(x)$ satisfying $M(x) \leq \mathcal{Q}(x)$ for all x. At the kth iteration, a candidate iterate z is obtained by solving the following subproblem:

$$\min_z \ M(z) \ \text{subject to} \ Az = b, \ z \geq 0, \\ \|z - x^k\|_\infty \leq \Delta,$$

where the last constraint represents a trust region with radius $\Delta > 0$. The candidate z becomes the new iterate x^{k+1} if the decrease in objective $\mathcal{Q}(x^k) - \mathcal{Q}(z)$ is a significant fraction of the decrease $\mathcal{Q}(x^k) - M(z)$ promised by the model function. Otherwise, no step is taken. In either case, the trust-region radius Δ may be adjusted, function and subgradient information about $\mathcal{Q}$ at z is used to enhance the model M, and the subproblem is solved again. The algorithm uses a "multicut" variant in which subgradients for partial sums of $\sum_{I=1}^{N} \mathcal{Q}_i(z)$ are included in the model separately, allowing a more accurate model to be constructed in fewer iterations.

In the MW implementation of the ATR algorithm, the function and subgradient information defining M is accumulated at the master processor, and the subproblem is solved on this processor. (Since M is always piecewise linear and the trust region is defined by an ∞-norm, the subproblem can be formulated as a linear program.) Most of the computational work in the algorithm involves solution of the N second-stage linear programs in y_i, from which we obtain $\mathcal{Q}_i(z)$ and $\partial \mathcal{Q}_i(z)$. This work is distributed among T tasks, to be executed in parallel, where each task requires solution of a "chunk" of N/T second-stage linear programs.

The use of chunking allows problems with very large N to make efficient use of a fairly large number of processors. However, the approach still requires evaluation of all the chunks for $\mathcal{Q}(z)$ to be completed before deciding whether to accept or reject z as the next iterate. It is possible that one chunk will be processed much more slowly than the others—its computation may have been interrupted by the workstation's owner reclaiming the machine, for instance. All the other workers in the pool will be left idle while waiting for evaluation of this chunk to complete.

The ATR method maintains not just a single candidate for the next iterate but rather a basket $\mathcal{B}$ containing 5 to 20 possible candidates. At any given time, the workers are evaluating chunks of second-stage problems associated with one or other of these basket points. ATR also maintains an "incumbent" x^I, which is the current best estimate of the solution and is a point for which $\mathcal{Q}(x^I)$ is known. When all the chunks for one of the basket points z have been evaluated, $\mathcal{Q}(z)$ is compared with the incumbent objective $\mathcal{Q}(x^I)$ and with the decrease predicted by the model function M at the time z was generated. As a result, either z becomes the new incumbent and x^I is discarded, or x^I remains the incumbent and z is discarded. In either case, a vacancy is created in the basket $\mathcal{B}$. To fill the vacancy, a new candidate iterate z' is generated by solving a subproblem with the trust-region constraint centered on the incumbent, that is,

$$\|z' - x^I\|_\infty \le \Delta.$$

We show results obtained for sampled instances of problems from the stochastic programming literature, using the MW implementation of ATR running on a Condor pool. The SSN problem described in [18] arises in design of a network for private-line telecommunications services. In this model, each of 86 parameters representing demand can independently take on 3 to 7 values, giving a total of approximately $N = 10^{70}$ scenarios. Sampling is used to obtain problems with more modest values of N, which are then solved with ATR.

Table 2: SSN, with $N = 10,000$

Run	Iter.	Procs.	Eff.	Time (min.)
L	255	19	.46	398
ATR-1	47	19	.35	130
ATR-10	164	71	.57	43

Results for an instance of SSN with $N = 10000$ are shown in Table 2. When written out as a linear program in the unknowns $(x, y_1, y_2, \ldots, y_N)$, this problem has approximately $1,750,000$ rows and $7,060,000$ columns. Table 2 compares three algorithms. The first is an L-shaped method (see [19]), which obtains its iterates from a model function M but does not use a trust region or check sufficient decrease conditions. (The implementation described here is modified to improve parallelism, in that it does not wait for all the chunks for the current point to be evaluated before calculating a new iterate.) The second entry in Table 2 is for the synchronous trust-region approach (which is equivalent to ATR with a basket size of 1), and the third entry is for ATR with a basket size of 10. In all cases, the second-stage evaluations were divided into 10 chunks, and 50 partial subgradients were added to M at each iteration. The table shows the average number of processors used during the run, the proportion of time for which these processors were kept busy, and the wall clock time required to find the solution.

The trust-region approaches were considerably faster than the L-shaped approach, indicating that the need for sound algorithms remains as keen as ever in a parallel environment; we cannot rely on raw computing power to do all the work. The benefits of asynchronicity can also be seen. When ATR has a basket size of 10, it is able to use a larger number of processors and takes less time to complete, even though the number of iterates

Table 3: storm, with $N = 250,000$

Run	Iter.	Procs.	Eff.	Time (min.)
ATR-1	25	67	.57	211
ATR-5	57	86	.96	229

increases significantly over the synchronous trust-region approach.

The real interest lies, however, not in solving single sampled instances of SSN, but in obtaining high-quality solutions to the underlying problem (the one with 10^{70} scenarios). ATR gives a valuable tool that can be used in conjunction with variance reduction and verification techniques to yield such solutions.

Finally, we show results from the "storm" problem, which arises from a cargo flight scheduling application [20]. The ATR implementation was used to solve a sampled instance with $N = 250,000$, for which the full linear program has approximately $132,000,000$ rows and $315,000,000$ columns. The results in Table 3 show that this huge linear program with nontrivial structure can be solved in less than 4 hours on a computational platform that costs essentially nothing. Because the second-stage work can be divided into a much larger number of chunks than for SSN—125 chunks, rather than 10—the synchronous trust-region algorithm is able to make fairly effective use of an average of 67 processors and requires less wall clock time than ATR with a basket size of 5.

6 Conclusions

Our experiences in the metaneos project have shown that cheap, powerful computational grids can be used to tackle large optimization problems of various types. These results have several interesting implications. In an industrial or commercial setting, the results demonstrate that one may not have to buy powerful computational servers to solve many of the large problems arising in areas such as scheduling, portfolio optimization, or logistics; the idle time on employee workstations (or, at worst, an investment in a modest cluster of PCs) may do the job. For the optimization research community, our results motivate further work on parallel, grid-enabled algorithms for solving very large problems of other types. The fact that very large problems can be solved cheaply allows researchers to better understand issues of "practical" complexity and of the role of heuristics. In stochastic optimization, higher-quality solutions can be found, and improvements to sampling methodology can be investigated.

Work remains to be done in making the grid infrastructure robust enough for general use. The logistics of assembling a grid—issues of security and shared ownership—remain challenging. The vision of a computational grid that is as easy to tap into as the electric power grid remains far off, though metaneos gives a glimpse of the way in which optimizers could exploit such a system.

We have investigated just a few of the problem classes that could benefit from solution on computational grids. Global optimization problems of different types should be examined further. Data-intensive applications (from tomography and data mining) represent a potentially huge field of work, but these require a somewhat different approach from the compute-intensive applications we have considered to date.

We hope that optimizers of all flavors, along with grid computing experts and applications specialists, will join the quest. There's plenty of work for all!

Acknowledgments

The metaneos project was funded by National Science Foundation grant CDA-9726385 and was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

For more information on the metaneos project, see `www.mcs.anl.gov/metaneos`. Information on Condor can be found at `www.cs.wisc.edu/condor` and on Globus at `www.globus.org`.

References

[1] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan-Kaufmann, 1998.

[2] J. Basney, M. Livny, and T. Tannenbaum. High throughput computing with Condor. *HPCU News*, 1(2), 1997.

[3] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP*, 11(1), June 1997.

[4] I. Foster and C. Kesselman. The Globus project: A status report. In *Proceedings of the Heterogeneous Computing Workshop*, pages 4–18. IEEE Press, 1998.

[5] J.-P. Goux, J. T. Linderoth, and M. Yoder. Metacomputing and the master-worker paradigm. Preprint MCS/ANL-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., February 2000.

[6] J.-P. Goux, S. Kulkarni, J. T. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, pages 43–50, Pittsburgh, Pennsylvania, August 2000.

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

[8] R. E. Bixby, W. Cook, A. Cox, and Lee. E. K. Computational experience with parallel mixed integer programming in a distributed environment. *Annals of Operations Research*, 90:19–43, 1999.

[9] J. T. Linderoth. *Topics in Parallel Integer Optimization.* Ph.D. thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, 1998.

[10] J. Eckstein, C. A. Phillips, and W. E. Hart. PICO: An object-oriented framework for parallel branch and bound. Research Report RRR-40-2000, RUTCOR, August 2000.

[11] Q. Chen, M. C. Ferris, and J. T. Linderoth. FATCOP 2.0: Advanced features in an opportunistic mixed integer programming solver. Technical Report Data Mining Institute Technical Report 99-11, Computer Sciences Department, University of Wisconsin at Madison, 1999. To appear in *Annals of Operations Research.*

[12] N. Robertson, D. P. Sanders, P. D. Seymour, and R. Thomas. A new proof of the four color theorem. *Electronic Research Announcements of the American Mathematical Society*, 1996.

[13] C. E. Nugent, T. E. Vollman, and J. Ruml. An experimental comparison of techniques for the assignment of facilities to locations. *Operations Research*, 16:150–173, 1968.

[14] A. Marzetta and A. Brünger. A dynamic-programming bound for the quadratic assignment problem. In *Computing and Combinatorics: 5th Annual International Conference, COCOON '99*, volume 1627 of *Lecture Notes in Computer Science*, pages 339–348. Springer, 1999.

[15] K. Anstreicher, N. Brixius, J.-P. Goux, and J. T. Linderoth. Solving large

quadratic assignment problems on computational grids. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., October 2000.

[16] K. M. Anstreicher and N. Brixius. A new bound for the quadratic assignment problem based on convex quadratic programming. Technical report, Department of Management Sciences, The University of Iowa, 1999.

[17] J. T. Linderoth and S. J. Wright. Implementing decomposition algorithms for stochastic programming on a computational grid. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 2000. In preparation.

[18] S. Sen, R. D. Doverspike, and S. Cosares. Network planning with random demand. *Telecommunications Systems*, 3:11–30, 1994.

[19] J. R. Birge and F. Louveaux. *Introduction to Stochastic Programming.* Springer Series in Operations Research. Springer, 1997.

[20] J. M. Mulvey and A. Ruszczyński. A new scenario decomposition method for large scale stochastic optimization. *Operations Research*, 43:477–490, 1995.