

MW: A Software Framework for Combinatorial Optimization on Computational Grids

Wasu Glankwamdee* J. T. Linderoth†

October 20, 2005

1 Introduction

Branch and bound is the backbone of many algorithms for solving combinatorial optimization problems, dating at least as far back as the work of Little, Murty, Sweeney, and Karel in solving the traveling salesman problem [18]. Branch and bound and similar tree-search techniques have been implemented on a variety of parallel computing platforms dating back to the advent of multiprocessor machines. See Gendron and Crainic [14] for a survey of parallel branch-and-bound algorithms, including references to early works.

Our goal in this paper is to demonstrate that branch-and-bound algorithms for combinatorial optimization can be effectively implemented on a relatively new type of multiprocessor platform known as a *computational grid* [12]. A computational grid consists of collections of loosely-coupled, non-dedicated, heterogeneous computing resources. Computational grids can be quite powerful, consisting of a large number of processors, but they can be difficult to use effectively. We will argue that to easily and effectively harness the power of computational grids for branch-and-bound algorithms, the master-worker paradigm should be used to control the algorithm. While recognizing that the master-worker paradigm is inherently not scalable, we will also show that the manner in which the tree search is performed can have a significant impact on the resulting parallel branch-and-bound algorithm's scalability and efficiency. Many of these ideas were (implicitly) present in the branch-and-bound implementation of Anstreicher *et al.* [4], used to solve a number of quadratic assignment problems to optimality. In this work we show that these ideas equally well apply to more general branch-and-bound implementations. We will also briefly describe a software framework MW that can ease an application developer's burden when implementing master-worker based parallel algorithms on computational grids. We will focus specifically on features of MW that are of the most utility to users wishing to implement branch-and-bound algorithms. To illustrate the impact of the issues we discuss, the paper ends with a case study implementation of a branch-and-bound solver to solve the 0-1 knapsack problem running on a wide-area computational grid of hundreds of processors.

2 Computational Grids

Networks of computers, such as the Internet, that have been highly successful as communication and information-sharing devices, are beginning to realize their enormous potential as computational grids—collections of loosely-coupled, geographically distributed, heterogeneous computing resources that can

*Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015, wag3@lehigh.edu

†Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015, jt13@lehigh.edu, <http://www.lehigh.edu/~jt13>

provide significant computing power over long time periods. As an example of the vast computing power available on a computational grid, consider the SETI@home project [24], which since its inception in the 1990's has delivered over 18,000 *centuries* of CPU time to a signal processing effort. Computational grids are generally used in this manner—as *high throughput computing devices*. In high-throughput computing, the focus is on using resources over long time periods to solve larger problems than would otherwise be possible. This is in contrast to *high performance computing* in which the performance is usually delivered and measured on a shorter time scale. Another important distinction in our work is that high performance resources are typically scheduled—a user must request a fixed number of processors for a fixed computing time. It is extremely difficult to accurately predict the CPU time required for branch-and-bound algorithms, which makes using resources in such a rigid manner nearly impossible for purveyors of branch-and-bound. Our grid computing approach must be more flexible.

This work will focus on computational grids built using the Condor software system [19], which manages distributively-owned collections of workstations known as *Condor pools*. A unique and powerful feature of Condor is that each machine's owner specifies the conditions under which jobs are allowed to run. In particular, the default policy is to stop a Condor job when a workstation's owner begins using the machine. In this way, Condor jobs only use cycles that would have otherwise been wasted. Because of the minimal intrusion of the Condor system, workstation owners are often quite willing to donate their machines, and large computational grids from Condor pools can be built.

In recent years, Condor has been equipped with a variety of features that allow collections of Condor pools to be linked together. One mechanism, called *flocking* [9], allows for jobs submit to one Condor pool to be run in a different (perhaps geographically distant) pool. A second mechanism, called *glide-in*, allows for scheduled (usually high performance) resources to temporarily join existing Condor pools [13]. With mechanism such as flocking and glide-in, large-scale computing configurations can be built, but this increase in available CPU power comes at a price. Additional CPU decentralization leads to further loss of authority and control of the resources, which implies that the fault tolerance aspects of algorithms running on such computational grids will be extremely important. We will demonstrate the use of flocking and glide-in to solve a large-scale knapsack problem in Section 6.4

2.1 Related Work

Aida, Natsume, and Futukata [1] describe a *hierarchical* master-worker paradigm aimed at reducing application performance degradation that may occur as a result of a single master. Their framework is applied on a branch-and-bound algorithm to minimize the maximum eigenvalue of a matrix function and run on a distributed computational grid testbed of up to 50 processors. Aida and Osumi extend this work in [2], scaling the algorithm up to 384 processors. In [1], the authors conclude that “the conventional master-worker paradigm is not suitable to efficiently solve the optimization problem with fine-grain tasks on the WAN setting, because communication overhead is too high compared to the costs of the tasks.” While this conclusion is certainly true, it is our contention that a significant majority of branch-and-bound algorithms can be made to consist of coarse-grained tasks, and the loss of coordinated control induced by such an algorithmic decision does not result in significant redundant work being done. Fault tolerance is not addressed in the works [1] and [2]. Our grids will draw CPU power from the Condor system of nondedicated processors, so fault tolerance is of extreme importance to our work.

Tanaka *et al.* [25] describe a master-worker based, grid-enabled algorithm for the 0-1 knapsack problem. In Section 6, we also will give such an algorithm. In [25], the focus is to enable communication links between processors on opposite sides of a firewall, and for this, they use software components from the Globus toolkit [11]. The focus is less on the performance or load balancing aspects of the branch-and-bound algorithm itself.

Other notable works for implementing parallel branch-and-bound algorithms include ALPS [27], BOB [5], PICO [8], and PPBB-Lib [26]. However, these works do not explicitly address the significant fault tolerance issues necessary to run on computational grids composed of harnessed idle CPU cycles.

Iamnitchi and Foster [17] propose a fully-decentralized branch-and-bound algorithm that addresses the fault recovery issue by propagating messages about the completed subtrees to all processors through a gossip mechanism [3]. This mechanism may result in significant overhead, both in terms of redundant work and in bandwidth usage. However, simulated results on reasonably-sized configurations show that in many cases the overhead is acceptable.

The works of Drummond *et al.* [7] and Filho *et al.* [10] describe a decentralized branch-and-bound algorithmic framework that is used to solve instances of the Steiner Problem using a branch-and-bound algorithm. Fault tolerance is achieved via sharing checkpoint information among processors in a round-robin fashion. Simultaneous failures of worker processes are difficult from which to recover, so the approach may be suited for “moderate” levels of fault recovery. Good computational results are presented on configurations of up to 48 processors.

3 Branch and Bound

Branch-and-bound algorithms are generally applied to \mathcal{NP} -Hard problems, so harvesting the enormous computing power of computational grids for branch-and-bound algorithms is a natural idea to consider. However, a fundamental drawback of using non-dedicated resources in the case of branch-and-bound is that if a processor leaves the computation, then the nodes on that processor must be re-evaluated. Thus, on a computational grid, we may wish to favor parallelization strategies in which nodes are centralized on a master processor that is under our direct control. Failure of the master processor can be dealt with through a checkpointing mechanism—by periodically writing the nodes to the disk. Having a single master processor responsible for managing the list of nodes that must be evaluated is also appealing from the standpoint that it provides a simple mechanism for dealing with the dynamic, error-prone nature of computational grids. If a new resource becomes available during the course of the computation, it can simply be assigned active nodes from the master processor. Likewise, should a resource be reclaimed (or fail) while evaluating a node, the master processor can simply assign that node to a different processor. Thus, for reasons of simplicity, the master-worker paradigm is very appealing for a grid computing environment.

However, the master-worker paradigm is inherently not scalable. That is, for configurations consisting of a large number of workers, the master processor may be overwhelmed in dealing with requests from the workers and *contention* may occur. Many parallel branch-and-bound methods have a more loosely coupled form of coordinated control that allows for more scalability. It is our goal in this work to show the limits to which branch-and-bound algorithms can be scaled using the master-worker paradigm, with a well-engineered version of the algorithm running on a computational grid.

Contention The lack of scalability of the master-worker paradigm comes from the bottleneck of a single master process serving many worker requests. The contention problem can be quite serious in a grid computing environment, as our goal is to have hundreds or thousands of workers served by a single master. To ease the contention problem, it is useful to think of the master-worker paradigm as a simple G/G/1 queueing model. There are two ways to increase the efficiency of the model:

1. Decrease the arrival rate. This can be accomplished by increasing the *grain size* of the computation. In the context of branch-and-bound, the grain size can be increased by making the base unit of work in the parallel branch-and-bound algorithm a *subtree*, not a single node. The grain size can be limited by giving an upper bound on the CPU time (or number of nodes) spent evaluating the subtree.

2. Increase the service rate. This can be accomplished by searching the subtrees in a depth-first manner. Searching the subtrees depth-first minimizes the number of nodes that will be left unexplored if the evaluation limit on the subtree is reached. This has two positive effects for increasing the service rate of the master processor. First, the size of the messages passed to the master is reduced, and second, the size of the list of unexplored nodes on the master is kept small. We will demonstrate the effect of node selection on contention and parallel efficiency of a master-worker branch-and-bound algorithm in Section 6.3.1.

Clean-up The unit of work in our parallel branch-and-bound algorithm will be a time or node limited subtree in order to ease contention effects at the master. However, a subtle point as regards to this strategy is that even though we may wish a worker to evaluate a subtree for γ seconds, it may take significantly less than γ seconds to *completely* evaluate and fathom the subtree. Somehow, we would like to ensure that if a node enters the master’s task queue, then it is *likely* that it will require the full time γ (or close to the full time γ) to evaluate. This is accomplished with a second (or clean-up) phase in every task. The goal of the clean-up phase is to fathom nodes that are unlikely to lead to full-length tasks. Nodes deeper in the branch-and-bound tree are likely to lead to short tasks, so in the clean-up phase, the focus is on evaluating these nodes. One strategy for implementing clean-up is the following. When the time limit γ is reached on the worker, the worker computes the average depth \bar{d} of its unevaluated nodes. Then, the worker is given an additional $\tau_1\gamma$ seconds to attempt to evaluate every node whose depth is larger than $\psi_1\bar{d}$. Note that if the worker is evaluating nodes in a depth-first fashion, this simply amounts to “popping up” the stack of nodes to depth $\psi_1\bar{d}$. This simple idea can be extended to a multi-phase clean up, wherein if the first phase of clean-up is not successful in removing all nodes of depth larger than $\psi_1\bar{d}$, the worker is given an additional $\tau_2\gamma$ seconds to remove all nodes whose depth is larger than $\psi_2\bar{d}$. Typically, $\psi_2 > \psi_1$, the goal is to make it more likely for subsequent clean-up phases to complete. We will demonstrate the effectiveness of clean-up in removing short tasks in the case study in Section 6.

Ramp-up and Ramp-down Contention is not the only issue that may cause a lack of efficiency of a parallel branch-and-bound algorithm. *Ramp-up* and *ramp-down*, referring to the times at the beginning and the end of the computation when there are more processors available than active nodes of the search tree, can also reduce efficiency. A simple and effective way to deal with these issues is to exploit the fact that the grain size of the branch-and-bound algorithm can be dynamically altered. If the number of tasks in the master’s list is less than α , the maximum task time is set to a small number of seconds β . Note that this strategy works to improve the efficiency in both the ramp-up and ramp-down phases.

4 MW API

MW consists of an Application Programming Interface (API) designed to be *easy* for application programmers to use, but one that also allows users to exploit specific properties of the algorithm in order to build an efficient implementation. The main characteristic of the parallel branch-and-bound algorithm that we exploit in order to increase parallel efficiency will be *dynamic grain size*.

In order to parallelize an application with MW, the application programmer must re-implement three abstract base classes – MWDriver, MWTask, and MWWorker.

4.1 MWDriver

To create the MWDriver, the user need re-implement four pure virtual methods:

- `get_userinfo(int argc, char *argv[])` – Processes arguments and does basic setup.
- `setup_initial_tasks(int *n, MWTask ***tasks)` – Returns the address of an array of pointers to tasks on which the computation is to begin. For branch-and-bound algorithms, $n=1$, and the task is a description of the root node.
- `pack_worker_init_data()`– Packs the initial data to be sent to the worker upon startup. Typically this consists of at least a description of the problem instance to be solved.
- `act_on_completed_task(MWTask *task)` – Is called every time a task finishes. For branch-and-bound algorithms, typically this method involved calling the `MWDriver::addTasks(int n, MWTask ***tasks)` method if the recently completed task has resulted in new nodes (tasks) that must be completed.

The `MWDriver` manages a set of `MWTasks` and a set of `MWWorkers` to execute those tasks. The `MWDriver` base class handles workers joining and leaving the computation, assigns tasks to appropriate workers, and rematches running tasks when workers are lost. All this complexity is hidden from the application programmer.

4.2 MWTask

The `MWTask` is the abstraction of one unit of work. The class holds both the data describing the work to be done by a task and the results computed by the worker. For branch-and-bound algorithms implemented in the manner suggested in Section 3, the input portion of the task consists of a description of one node. For the input node, the goal is to evaluate the entire subtree rooted at this node. The result portion of the task is a list of nodes that were unevaluated when the task CPU limit was reached. The derived task class must implement functions for sending and receiving its data between the master and worker. The names of these functions are self-explanatory: `pack_work()`, `unpack_work()`, `pack_results()`, and `unpack_results()`. These functions will call associated `pack()` and `unpack()` functions in the `MWRMComm` class (described in Section 5.4) to perform the communication.

4.3 MWWorker

The `MWWorker` class is the core of the worker executable. Two pure virtual functions must be implemented:

- `unpack_init_data()`– Unpacks the initialization information passed in the `MWDriver`'s `pack_worker_init_data()`. This method can also perform any computations necessary to initialize a worker before receiving tasks.
- `execute_task(MWTask *task)`– Given a task, computes the results.

The `MWWorker` asks the master for a task and sits in a simple loop. Given a task, it computes the results, reports the results back, and waits for another task. The loop finishes when the master asks the worker to end.

5 Additional MW Features

5.1 Task List Management

In MW the master class manages a list of uncompleted tasks and a list of workers. The default scheduling mechanism in MW is to simply assign the task at the head of the task list to the first idle worker in the worker list. However, MW gives flexibility to the user in the manner in which each of the lists are ordered.

For example, MW allows the user to easily implement both a Last-In-First-Out policy (LIFO) and a First-In-First-Out policy (FIFO) by simply specifying the location at which new tasks are added to the task list (the `MWTaskAdditionMode`) to be one of `ADD_AT_END` or `ADD_AT_BEGIN` in the method

```
MWDriver::set_task_add_mode(MWTaskAdditionMode t).
```

A more dynamic way in which to manage the task list is through *task keys*. Each (derived) `MWTask` may be assigned a key value through the method `MWDriver::set_task_key_function(MWKey(*) (MWTask *) key_func)`, where `key_func` is the address of a function that takes a pointer to a `MWTask` and returns the `MWKey` of the task, which is typed to be a double. The `task_key` may be changed dynamically during the course of the computation by using this method. The task list can be sorted through the method `MWDriver::sort_task_list()`, and once sorted, tasks can be added and retrieved from the sorted list by task key value. In MW, the task list is sorted from smallest key value to largest key value.

The ability to dynamically alter the task key during the course of the search is important for some branch-and-bound computations. For example, many branch-and-bound algorithms search the tree in a best-first manner. For large branch-and-bound trees, this can lead to the number of active nodes becoming very large, exhausting the available memory on the master processor. Instead, by dynamically altering the task list ordering, the user can adopt an approach where the nodes are searched best-first until the number of tasks at the master exceeds a “high-water” level h , and then the task list is reordered so that the tasks with the worst bound are searched. The task list can be kept in this order until its size of becomes smaller than a “low-water” level ℓ , at which time the list can be reordered in a best-first fashion.

One final method that can be of particular importance to branch-and-bound applications is a call that can delete all tasks in the task list whose key values are larger than a specified value:

```
MWDriver::delete_tasks_worse_than( MWKey ).
```

5.2 User-level checkpointing

Long running computations will invariably at some point fail. A program bug, a power loss, an operating system upgrade, or a network failure will cause the program to stop execution. Being able to easily deal with failures of the worker executables is precisely the purpose of MW, so these failures cause little problem—MW can detect each failure and resend the task the failed worker was executing to another worker. A crash on the master is more serious. To mitigate this risk, MW provides a mechanism for user-level checkpointing of the master state. To implement checkpointing in MW the users need implement the following four methods that read and write the state of the master and a task to a file:

- `MWDriver::write_master_state(FILE *fp),`
- `MWDriver::read_master_state(FILE *fp),`
- `MWTask::write_ckpt_info(FILE *fp),`
- `MWTask::read_ckpt_info(FILE *fp)`

5.3 Statistics and Benchmarking

At the end of a computation MW reports various performance statistics. MW relies on a (dynamically changing) set of workers W , and for each worker $j \in W$, a variety of statistics are collected, including

- u_j , the wall clock time that worker j was available
- c_j , the CPU time that worker j used to execute tasks, and
- s_j , the wall clock time that worker j was suspended by the resource manager

MW will report to the user the overall parallel performance η :

$$\eta \stackrel{\text{def}}{=} \frac{\sum_{j \in W} c_j}{\sum_{j \in W} (u_j - s_j)}.$$

Benchmarking computations that run on dynamically available, heterogeneous processors is a very difficult issue. In order to compute an algorithmic performance statistic that is comparable between runs, users can register a *benchmark task* with MW. When a worker is available to begin computation, the MWDriver will send the worker the benchmark task to compute, record the CPU time required to perform this task, and use this number to compute a normalized total CPU time. If $b_j = 1/t_j$, $\forall j \in W$ is the reciprocal of the CPU time worker j required to complete the benchmark task, and $b_M = 1/t_M$ is the reciprocal of the time time for the master processor to complete the task, then MW will report a normalized CPU time of

$$\mathcal{T} = \frac{\sum_{j \in W} c_j b_j}{b_M}.$$

Branch-and-bound algorithm find the normalized benchmark feature quite useful for tuning the myriad of parameters in the algorithm.

5.4 The RMComm Layer

MW contains an abstract interface to different resource management and communication mechanisms. Thus, by simply linking with the appropriate libraries, users can rely on different software to find appropriate worker processors and communicate between master and worker processors. Currently, there are four RMComm implementations in the standard MW distribution: Condor-Sockets, Condor-PVM, Condor-Files, and an Independent layer, useful for debugging, in which both master and worker exist in a single process. In our case study of Section 6, we use both the Condor-PVM and Condor-Sockets RMComm layers.

6 Case Study: The Knapsack Problem

6.1 Background

In this section, we describe a branch-and-bound implementation of MW to solve the 0–1 knapsack problem. In the 0 – 1 knapsack problem, there is a set $N = \{1, \dots, n\}$ of items each with profit c_i and weight a_i , a knapsack capacity b , and the objective is to fill the knapsack as profitably as possible, i.e. solve

$$z^* = \max\{c^T x \mid a^T x \leq b, x \in \mathbb{B}^n\}. \quad (1)$$

Our MW implementation MWKnap has three goals. First, we would like to demonstrate that building a parallel branch and bound solver is easily accomplished with MW. Second, we would like the solver

to be flexible enough to show the improvement in efficiency that can be obtained by tuning the search appropriately, as discussed in Section 3. Finally, we wish to demonstrate that very large problem instances can be solved with branch-and-bound by harnessing the CPU cycles of a computational grid.

Without loss of generality, we assume that the items are sorted in decreasing order of profit to weight ratio c_i/a_i . The lower bound in the branch-and-bound algorithm is computed by greedily inserting items while the knapsack capacity is not exceeded. The upper bound in the algorithm is obtained by additionally inserting the fractional part of the last item that exceeds the knapsack capacity in order to fill the knapsack exactly. This can be seen as solving the linear programming relaxation of (1) [6]. Note that in the solution there is at most one fractional item, which we denote as f . Therefore, the solution to the LP relaxation is $x_i = 1$ for $i = 1, \dots, f - 1$, $x_f = \{b - \sum_{i=1}^{f-1} a_i\}/a_f$, and $x_i = 0$ for $i = f + 1, \dots, n$. The lower bound on the optimal solution value z_L is given by the formula

$$z_L = \sum_{i=1}^{f-1} c_i,$$

and the upper bound on the optimal solution value z_U is given by

$$z_U = \sum_{i=1}^{f-1} c_i + (b - \sum_{i=1}^{f-1} a_i) \frac{c_f}{a_f}.$$

If all items fit into the knapsack ($f = n$), then the lower and upper bounds are

$$z_L = z_U = \sum_{i=1}^f c_i.$$

Let us make a few definition to precisely define the algorithm. We use the term *node* (\mathcal{N}) to denote the problem associated with a certain portion of the feasible region of the problem. A node is characterized by the sets of variables fixed to 0 and 1. Namely, $\mathcal{N} = (N_0, N_1, N_F)$, where

$$\begin{aligned} N_0 &\stackrel{\text{def}}{=} \{i \mid x_i = 0\}, \\ N_1 &\stackrel{\text{def}}{=} \{i \mid x_i = 1\}, \text{ and} \\ N_F &\stackrel{\text{def}}{=} \{i \mid i \notin N_0 \cup N_1\} = N \setminus N_0 \setminus N_1. \end{aligned}$$

Again, we assume WLOG that N_F is ordered in decreasing order of c_i/a_i . The lower and upper bounds of a node \mathcal{N} are denoted by $z_L^{\mathcal{N}}$ and $z_U^{\mathcal{N}}$ respectively. \mathcal{L} is a set of nodes that must still be evaluated, and z^* holds the current best solution value. With these definitions, the general branch-and-bound algorithm for 0 – 1 knapsack problem is summarized in Algorithm 1.

At each node, a branching operation may be performed on the sole fractional variable x_f . The node is fathomed if the lower bound is equal to the upper bound, if the upper bound is lower than the current best solution value, or if all items are able to be placed into the knapsack. The next node in the list \mathcal{L} to be evaluated might be chosen in a depth-first fashion, as suggested by Greenberg and Hegerich [15], or in a best-first fashion, in which the node \mathcal{N} with the largest value of $z_U^{\mathcal{N}}$ is chosen. For more sophisticated, improved variants of Algorithm 1, see the work of Horowitz and Sahni [16] and a survey by Pisinger and Toth [23].

6.2 MW Implementation

To create a parallel solver for the knapsack problem with MW, we must re-implement `MWTask`, the `MWWorker` that executes these tasks, and the `MWDriver` that guides the computation by acting on completed tasks.

Algorithm 1 The Branch-and-Bound Algorithm for 0 – 1 Knapsack Problem

Require: $c_i > 0, a_i > 0$. c_i/a_i are sorted in decreasing order.

$z^* = 0$. Put the root node in \mathcal{L} .

while $\mathcal{L} \neq \emptyset$ **do**

 Choose and delete node $\mathcal{N} = (N_0, N_1, N_F)$ from \mathcal{L} .

 Let f be the smallest index such that $\sum_{i \in N_F, i=1}^f a_i > b - \sum_{i \in N_1} a_i$.

if $\sum_{i \in N_F, i=1}^f a_i \leq b - \sum_{i \in N_1} a_i$ **then**

$$z_L^{\mathcal{N}} = z_U^{\mathcal{N}} = \sum_{i \in N_1} c_i + \sum_{i \in N_F, i=1}^f c_i$$

else

$$z_L^{\mathcal{N}} = \sum_{i \in N_1} c_i + \sum_{i \in N_F, i=1}^{f-1} c_i$$

$$z_U^{\mathcal{N}} = z_L^{\mathcal{N}} + (b - \sum_{i \in N_F, i=1}^{f-1} c_i) \frac{c_f}{a_f}.$$

end if

if $z_L^{\mathcal{N}} > z^*$ **then**

$$z^* = z_L^{\mathcal{N}}.$$

 Remove nodes $\mathcal{N}' \in \mathcal{L}$ such that $z_U^{\mathcal{N}'} < z^*$.

end if

if $z_L^{\mathcal{N}} = z_U^{\mathcal{N}}$ or $z_U^{\mathcal{N}} < z^*$ **then**

 Fathom node \mathcal{N} .

else

 Add a new node $\hat{\mathcal{N}} = (N_0 \cup \{f\}, N_1, N_F \setminus \{f\})$ to \mathcal{L} , with $z_U^{\hat{\mathcal{N}}} = z_U^{\mathcal{N}}$.

if $\sum_{i \in N_1} a_i + a_f < b$ **then**

 Add a new node $\hat{\mathcal{N}} = (N_0, N_1 \cup \{f\}, N_F \setminus \{f\})$ to \mathcal{L} , with $z_U^{\hat{\mathcal{N}}} = z_U^{\mathcal{N}}$.

end if

end if

end while

MWTask Algorithm 1 will solve the 0 – 1 knapsack instance to optimality. As discussed in Section 3, we wish to parallelize Algorithm 1 within the master-worker framework by making the base unit of work a limited subtree. Thus, in our parallel implementation Algorithm 1 becomes a *task*, with the exception that the grain size is controlled by specifying the maximum CPU time or maximum number of nodes that a worker is allowed to evaluate before reporting back to the master. In MW, there are two portions of a task, the work portion and the result portion. For our solver MWKnap, a *KnapTask* class is derived from the base *MWTask*, and the work portion of the *KnapTask* consists of a single input node. The result portion consists of an improved solution (if one is found), and a list containing the nodes of the input subtree that are not able to be evaluated before reaching the task’s node or time limit. Figure 1 shows a portion of the *KnapTask* C++ header file.

```

1 class KnapTask : public MWTask
2 {
3     // Work portion
4     KnapNode inputNode_;
5
6     // Result portion
7     bool foundImprovedSolution_;
8     double solutionValue_;
9     std::vector<KnapNode *> outputNode_;
10 };

```

Figure 1: The work and result portions of *KnapTask*

MWWorker In MW, the (pure virtual) *MWWorker::execute_task(MWTask *task)* method is entirely in the user’s control. Therefore, when implementing branch-and-bound algorithms for which the task is to evaluate a subtree, the user is responsible for writing code to manage the heap of unevaluated subtree nodes. For MWKnap, we implement a heap structure using C++ Standard Template Library to maintain the set of active nodes. The heap can be ordered by either node depth or node upper bound, so we can quantify the effect of different worker node selection techniques on overall parallel efficiency. Figure 2 shows portions of the derived worker’s *execute_task* method.

Note on line 3 of Figure 2, we need to downcast the abstract *MWTask* to an instance of the *KnapTask* that can be executed. On line 4, the node heap is created and instantiated to be either a best-first or depth-first heap by setting the variable *currentNodeOrder_*. The while-loop from lines 6 to 46 contains the implementation of Algorithm 1. The procedure *finished(heap)* on line 6 is implemented separately and ensures that the worker will evaluate the given subtree as long as there are active nodes left in that subtree or the node or time limit of the task is not violated. The purpose of the for-loop from lines 12 to 20 is to identify the fractional item *f*. The if-statement beginning at line 23 is used to check the feasibility of the solution and compute lower and upper bounds of the node. The if-statement from lines 34 to 37 is exercised when a better lower bound is found and infeasible nodes are fathomed. The child nodes are generated from the result of the if-statement from lines 39 to 43. On line 47, when the grain size limit is reached, the nodes left on the heap are copied to the result portion of the task and returned back to the master task pool.

MWDriver In MWKnap, a *KnapMaster* class is derived from the base *MWDriver* class. The *MWDriver::act_on_completed_task(MWTask *t)* method is implemented to handle the results passing

```

1 void KnapWorker::execute_task(MWTask *t)
2 {
3     KnapTask *kt = dynamic_cast<KnapTask *> (t);
4     NodeHeap *heap = new NodeHeap(currentNodeOrder_);
5     heap->push(new KnapNode(kt->getInputNode()));
6     while (!finished(heap)) {
7         KnapNode *node = heap->top(); heap->pop();
8         double remainingSize = instance_.getCap() - node->getUsedCap();
9         double usedValue = node->getUsedValue();
10        int f = 0;
11
12        for (KnapInstance::itemIterator it = instance_.itemsBegin();
13             it != instance_.itemsEnd(); ++it) {
14            if (node->varStatus(f) == Free) {
15                fSize = it->getSize(); fProfit = it->getProfit();
16                remainingSize -= fSize; usedValue += fprofit;
17            }
18            if (remainingSize < 0.0) break;
19            f++;
20        }
21
22        bool branch = false;
23        if (remainingSize >= 0) {
24            nodeLb = usedValue; nodeUb = usedValue;
25        }
26        else {
27            usedValue -= fProfit; remainingSize += fSize;
28            nodeLb = usedValue;
29            nodeUb = usedValue + fProfit/fSize * remainingSize;
30            node->setUpperBound(nodeUb);
31            if (nodeUb > kt->getSolutionValue()) branch = true;
32        }
33
34        if (nodeLb > kt->getSolutionValue()) {
35            kt->setBetterSolution(nodeLb);
36            heap->fathom(nodeLb);
37        }
38
39        if (branch) {
40            heap->push(new KnapNode(*node, f, FixedZero, fSize, fProfit));
41            if (node->getUsedCap() + fSize < instance_.getCap())
42                heap->push(new KnapNode(*node, f, FixedOne, fSize, fProfit));
43        }
44
45        delete node;
46    }
47    kt->addNodesInHeap(*heap);
48    return;
49 }

```

Figure 2: Algorithm 1 in MWWorker

back from the workers. Figure 3 shows a portion of this method. The if-statement from lines 6 to 12 is used to update the improved solution value, and remove nodes in the master pool that have their upper bounds less than the current best solution value. New tasks, which are unevaluated nodes left from the completed task, are added to the master task list by the for-loop beginning at line 15. Here we assume that the master is in best-first mode.

```

1  MWReturn KnapMaster::act_on_completed_task(MWTask *t)
2  {
3      KnapTask *kt = dynamic_cast<KnapTask *> (t);
4
5      // Remove infeasible nodes portion
6      if (kt->foundImprovedSolution()) {
7          double blb = kt->getSolutionValue();
8          if (blb > bestLB_) {
9              bestLB_ = blb;
10             delete_tasks_worse_than(-bestLB_);
11         }
12     }
13
14     // Add new tasks portion
15     for (vector<KnapNode *>::const_iterator it = kt->newNodeBegin();
16         it != kt->newNodeEnd(); ++it) {
17         if ((*it)->getUpperBound() > bestLB_) addTask(new KnapTask(**it));
18         delete *it;
19     }
20 }

```

Figure 3: The portions of `act_on_completed_task(MWTask *t)` in `KnapMaster`

6.3 Computational Experience

This section contains experiments showing the impact of varying algorithmic parameters on the effectiveness of `MWKn timer`. The goals of this section are to answer the following questions:

1. In what order should the master send nodes to the workers?
2. How should the workers search the subtrees given to them by the master? Namely,
 - In what order should the subtree nodes be evaluated?
 - For how long should the subtree be evaluated before reporting back to the master?

We test `MWKn timer` on a family of instances known as *circle(2/3)* [22]. In these instances, the weights are randomly generated from a uniform distribution, $a_i \sim U[1, 1000]$, and the profit of item i is a circular function of its weight: $c_i = (2/3)\sqrt{4000^2 - (a_i - 2000)^2}$. These instances are contrived to be challenging for branch-and-bound algorithms, and various algorithms in the literature can solve the instances with up to 200 items in less than one hour on a single machine [20, 21].

In the first phase of our computational experiments, we use solely a Condor pool at Lehigh University consisting of 246 heterogeneous processors. In the pool, there are 146 Intel Pentium III 1.3GHz processors and 100 AMD Opteron 1.9GHz processors. All machines run the Linux operating system.

6.3.1 Contention

The first experiment is aimed at demonstrating the effect of task grain size and node selection strategy on contention effects at the master processor. For this experiment, the grain size is controlled by limiting the maximum number of nodes the worker evaluates before reporting back to the master (MNW). Different combinations of master node order, worker node order, and grain sizes varying between MNW = 1 and MNW = 100,000 nodes are tested on `cir200`, a `circle(2/3)` instance with 200 items. The maximum number of workers is limited to 64. Tables 1 and 2, show the wall clock time (\mathcal{W}), the total number of nodes evaluated (\mathcal{N}), the parallel performance (η), and the normalized CPU time (\mathcal{T}) for each trial in the experiment.

Table 1: Performance of MWKnap with Best-First Master Task Pool on `cir200`

MNW	Worker Node Order							
	Best-First				Depth-First			
	\mathcal{W} (s)	\mathcal{N}	η	\mathcal{T} (s)	\mathcal{W} (s)	\mathcal{N}	η	\mathcal{T} (s)
1	304.0	1.0E6	0.59	84.95	758.9	1.3E6	0.19	204.63
10	155.4	1.6E6	2.44	103.3	1111	4.5E6	0.34	574.15
100	119.0	5.4E6	8.66	302.5	2214	2.5E7	1.41	2736.0
1000	151.3	2.7E7	22.5	1340	362.6	1.5E8	28.9	14839
10000	140.0	5.1E7	50.0	2436	111.6	3.5E7	46.5	3327.4
100000	122.4	5.1E7	58.2	2417	186.1	1.3E8	60.1	12121

Table 2: Performance of MWKnap with Depth-First Master Task Pool on `cir200`

MNW	Worker Node Order							
	Best-First				Depth-First			
	\mathcal{W} (s)	\mathcal{N}	η	\mathcal{T} (s)	\mathcal{W} (s)	\mathcal{N}	η	\mathcal{T} (s)
1	56122	3.1E6	0.01	292.45	492.3	1.3E6	0.63	205.77
10	13257	6.0E6	0.04	379.28	1171	4.6E6	0.32	569.97
100	5004.8	1.8E7	0.25	999.60	2521	2.9E7	0.80	3159.5
1000	4859.7	1.1E8	1.52	5832.6	520.4	1.5E8	21.5	14659
10000	4718.4	5.5E8	7.27	25464	484.1	4.2E7	12.7	4004.4
100000	4216.5	8.6E8	12.8	40153	187.9	1.1E8	57.4	10297

Even though each combination of master node-order, worker node-order, and grain size is attempted only once, we can still draw some meaningful conclusions from the trends observed in Tables 1 and 2.

- The parallel performance increases with the grain size but at the price of a larger total number of nodes evaluated;
- Small grain sizes have very low parallel efficiency; and
- A master search order of best-first is to be preferred.

The best-first search strategy in the worker performs well on the relatively small instance `cir200`, but when this strategy is employed on larger instances, it leads to extremely high memory usage on the master.

For example, Figure 4 shows the memory usage of the master processor when the workers employ a best-first search strategy on `cir250`, a `circle(2/3)` instance with 250 items. After only 3 minutes, the master processor memory usage goes to over 1GB. At this point, the master process crashes, as it is unable to allocate any more memory to add the active nodes to its task list. Therefore, in subsequent experiments, we will employ a best-first node ordering strategy on the master and a depth-first node selection strategy on the workers. Further, we will use a relatively large CPU limited grain size for the tasks. For example, with a grain size of $\gamma = 100$ seconds and increasing the maximum number of workers to 128, `MWKn timer` can solve `cir250` in $\mathcal{W} = 4674.9$ seconds of wall clock time with an average parallel efficiency of $\eta = 65.5\%$.

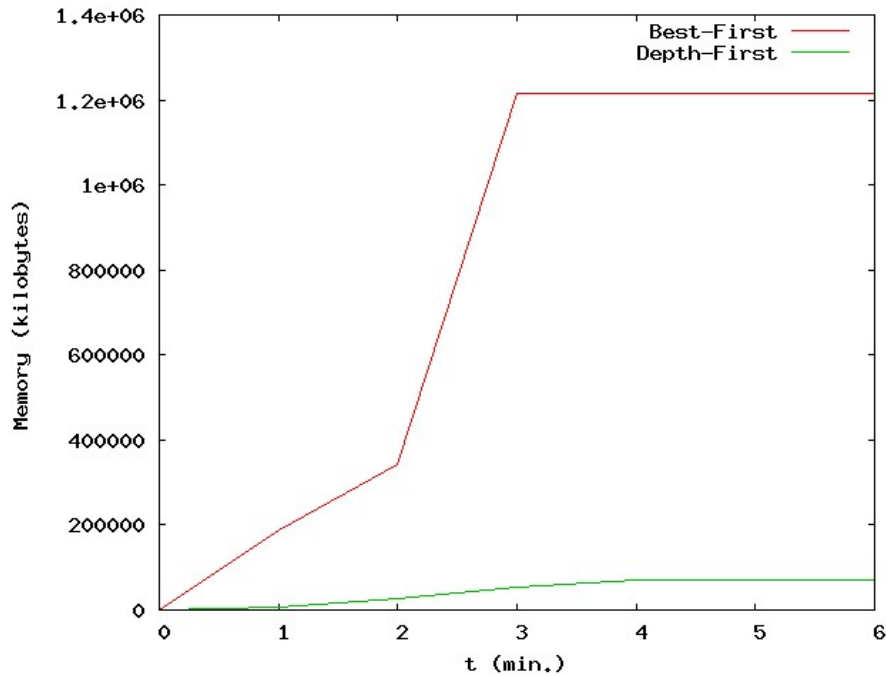


Figure 4: Memory Usage of Different Worker Node Orders on `cir250`

6.3.2 Ramp-up and Ramp-down

Even with careful tuning of the grain size and the master and worker search strategies, the parallel efficiency (65.5%) of `MWKn timer` on the test instance `cir250` is relatively low. Some loss of efficiency is caused because there is a dependence between tasks in the branch-and-bound algorithm. This task dependence leads to situations where workers are sitting idle waiting for other workers to report back their results to the master. In the case of `MWKn timer`, this occurs at the beginning and at the end of the computation when the master pool has less tasks than participating workers.

As mentioned in Section 3, the master pool can be kept populated by dynamically changing the grain size. The efficiency improvement during ramp-up and ramp-down is achieved by reducing γ to 10 seconds when there are less than 1000 nodes in the master pool. Using this worker idle time reduction strategy, the efficiency of `MWKn timer` on `cir250` is increased to $\eta = 77.7\%$ with a wall clock time of $\mathcal{W} = 4230.3$ seconds.

6.3.3 Clean-up

The efficiency of MWKnap can be improved further with a worker clean-up phase designed to evaluate nodes that would subsequently lead to short-length tasks. The MWKnap implementation of the clean-up phase, discussed in Section 3, allows for an additional 100 seconds ($\tau_1\gamma = 100$) to process all nodes deeper than the average node depth of the remaining nodes on the worker ($\psi_1\bar{d} = \bar{d}$). Using clean-up together with the ramp-up and ramp-down strategy, the wall clock time of MWKnap to solve cir250 is reduced to $\mathcal{W} = 4001.4$ seconds, and the parallel efficiency is increased to $\eta = 81.4\%$. The workers are able to eliminate 92.06% of all nodes deeper than the average depth during the clean-up phase.

The clean-up phase can be further refined by allowing an additional 50 seconds ($\tau_2\gamma = 50$) to process the remaining nodes below the average node depth plus five ($\psi_2\bar{d} = \bar{d} + 5$). With this two-phase clean-up, MWKnap is able to eliminate 99.87% of the nodes deeper than the target clean-up depth when solving cir250. The wall clock time decreases to $\mathcal{W} = 3816.8$ seconds and the parallel efficiency increases to $\eta = 90.1\%$.

Figure 5 shows the distribution of task execution times before the optimal solution is found for the initial MWKnap implementation, MWKnap with a single-phase clean-up, and MWKnap with two-phase clean-up. Figure 6 compares the distribution of task execution times for the same three implementations after the optimal solution is found. Since $\gamma = 100$, task times greater than 100 seconds correspond to tasks in which clean-up was necessary. Even though the distributions of task times look nearly the same for all three implementations, the parallel efficiency increases by over 12%. Thus, an interesting conclusion that can be drawn from our work is that a small improvement in the distribution of task times can lead to a significant increase in parallel efficiency for master-worker applications.

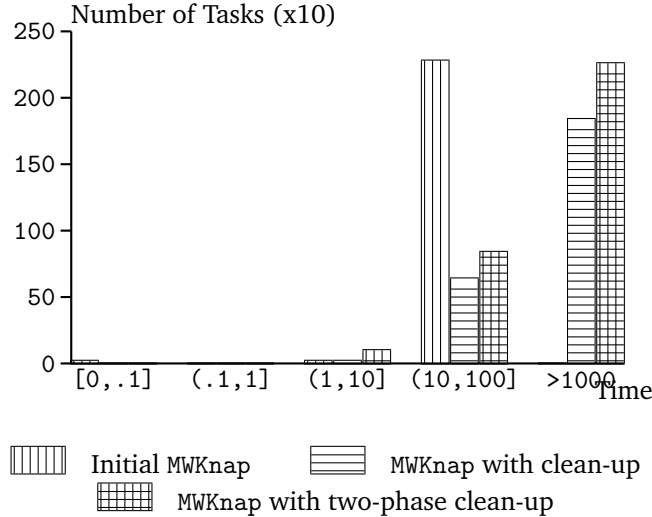


Figure 5: Distribution of Task Time on cir250 Before Optimal Solution is Found

6.4 Large-Scale Computation

In this section, we demonstrate the true power of a computational grid—the ability to harness diverse, geographically distributed resources in an effective manner to solve larger problem instances than can be solved using traditional computing paradigms. Our demonstration is made on an instance cir300, a circle(2/3) knapsack instance of size 300. To solve this instance, we will use a subset of over 4000 available processors whose characteristics are given in Table 6.4. There are three different processor types,

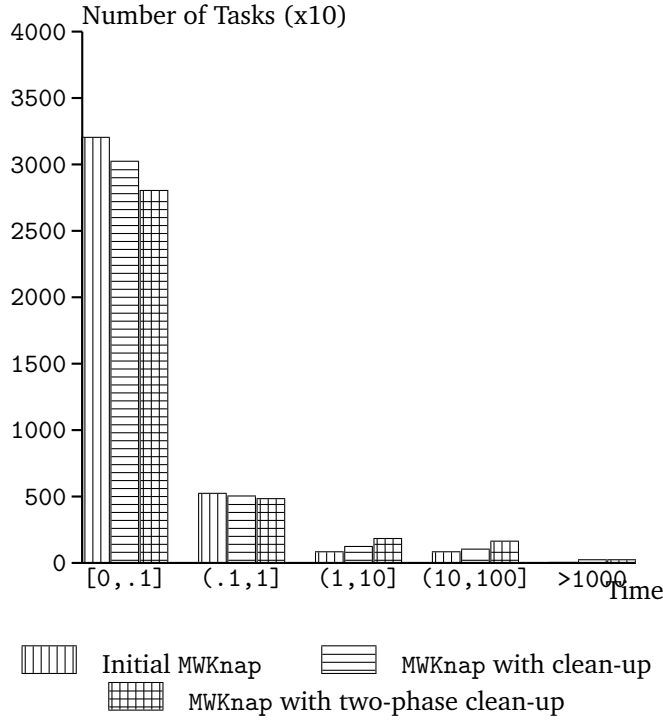


Figure 6: Distribution of Task Time on cir250 After Optimal Solution is Found

running two different operating systems, and located at four different locations in the United States. The processors at the University of Wisconsin compose of the main Condor pool to which our worker jobs are submit. Processors at NCSA and the University of Chicago are part of the Teragrid (<http://www.teragrid.org>). These processors are scheduled using the Portable Batch Scheduler (PBS), and join our knapsack computation through the Condor glide-in mechanism. Other processors join the computational via Condor flocking.

Table 3: Available Processors for Solution of cir300 Instance

Number	Type	Operating System	Location	Access Method
1756	Itanium-2	Linux	NCSA	Glide-in
302	Itanium-2	Linux	UC-ANL	Glide-in
252	Pentium	Linux	NCSA	Flock
508	SGI	IRIX	NCSA	Flock
1182	Pentium (Various)	Linux	Wisconsin	Main
52	Pentium (Various)	Linux	Lehigh	Flock

For this instance, we used a grain size of $\gamma = 250$ CPU seconds. Initial testing on the cir300 instance showed that we could not continually send nodes with the best upper bounds to the worker processors and keep the size of the master task list within the memory bounds on the master processor. Thus, the master node list is kept in best-bound order while there are less than $h = 50,000$ tasks, then it is switched to worst-bound order until the number of tasks is reduced to $\ell = 25,000$ tasks, at which point the order is again reversed.

The instance is solved in a wall clock time of less than three hours, using on average 321.2 workers,

and at a parallel efficiency of $\eta = 84.8\%$. The total CPU time used on all workers is 2,795,897 seconds, or over one CPU month. Figure 7 shows the number of workers available to use while the run is proceeding. We see that the maximum number of workers is achieved at the end of the run, showing that the grid could likely have delivered computing power to solve a larger instance. In Figure 8, the number of tasks in the master task list is plotted during the course of the run. The effect of switching the master node ordering from best-first to worst-first at 50,000 tasks is clearly seen.

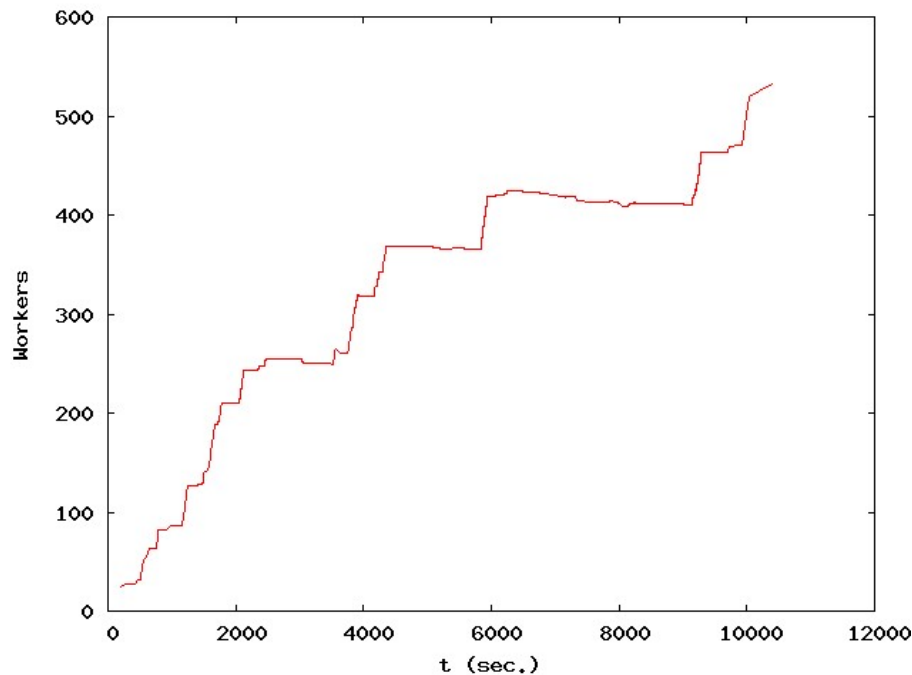


Figure 7: Workers Used During Solution of `cir300` Instance

7 Conclusions

We have introduced MW, a framework for implementing master-worker style computations in a dynamic computational grid computing environment. While the master-worker paradigm is not scalable, we have shown that by carefully tuning search parameters, the algorithm can be made to scale reasonably efficiently, even when there are hundreds of worker processors being served by a single master. Future work will focus on making it even easier for users to build branch-and-bound algorithms with MW. First, MW will be augmented with a general branch-and-bound interface. In our envisioned implementation, the users will need only provide mechanisms for computing bounds and for branching. The efficiency improvement features detailed in this work will be implemented in the base class, relieving the users from this burden. We also have begun working on a “black-box” implementation of MW in which the `MWWorker::execute_task(MWTask *)` method is implemented with a user-supplied executable. MW source code, including the `MWKnap` solver described here, is available from the MW homepage: <http://www.cs.wisc.edu/condor/mw>.

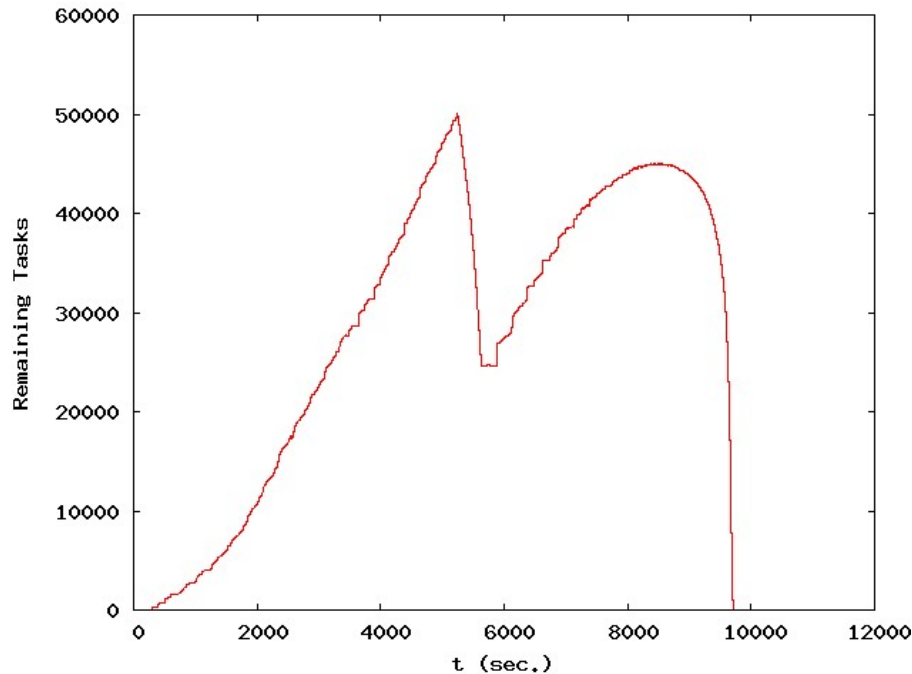


Figure 8: Number of Tasks in Master List During Solution of cir300 Instance

Acknowledgments

The authors would like to sincerely thank the whole Condor team for their tireless efforts in providing a robust and useful software tool. In particular, Greg Thain is instrumental in helping us run the computational experiments and for providing comments that helped to improve the exposition. This work is supported in part by the US National Science Foundation (NSF) under grants CNS-0330607 and DMI-0522796. Computational resources are provided in part by equipment purchased by the NSF through the IGERT Grant DGE-9972780, and through Teragrid resources at the National Center for Supercomputing Applications (NCSA) and the University of Chicago under grant DDM-050005.

References

- [1] K. Aida, W. Natsume, and Y. Futakata. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pages 156–163, 2003.
- [2] K. Aida and T. Osumi. A case study in running a parallel branch and bound application on the grid. In *Proceedings of the IEEE/IPSJ: The 2005 Symposium on Applications & the Internet (SAINT2005)*, pages 164–173, 2005.
- [3] N. Alon, A. Barak, and U. Mander. On disseminating information reliably and without broadcasting. In *7th International Conference on Distributed Computing Systems ICDCS97*. IEEE Press, 1987.
- [4] K. Anstreicher, N. Brixius, J.-P. Goux, and J. T. Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming, Series B*, 91:563–588, 2002.

- [5] M. Benechouche, V.-D. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol. Building a parallel branch and bound library. In *Solving Combinatorial Optimization Problems in Parallel*, Berlin, 1996. Springer.
- [6] G. B. Dantzig. Discrete variable extremum problems. *Operations Research*, 5:266–277, 1957.
- [7] L. M. A. Drummond, E. Uchoa, A. Goncalves, J. M. N. Silva, M. C. P. Santos, and M. C. S. de Castro. A grid-enabled branch-and-bound algorithm with application on the steiner problem in graphs. Unpublished Manuscript.
- [8] J. Eckstein, C. A. Phillips, and W. E. Hart. PICO: An object-oriented framework for parallel branch-and-bound. In *Proc. Inherently Parallel Algorithms in Feasibility and Optimization and Their Applications*, pages 219–265, 2001.
- [9] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors: Load sharing among workstation clusters. *Journal on Future Generation Computer Systems*, 12, 1996.
- [10] J. V. Filho, L. M. A. Drummond, E. Uchoa, and M. C. S. de Castro. Towards a grid enabled branch-and-bound algorithm. available at http://www.optimization-online.org/DB_HTML/2003/10/756.html.
- [11] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. J. Supercomputer Applications*, 11:115–128, 1997.
- [12] I. Foster and C. Kesselman. Computational grids. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999. Chapter 2.
- [13] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, 2002.
- [14] B. Gendron and T. G. Crainic. Parallel branch and bound algorithms: Survey and synthesis. *Operations Research*, 42:1042–1066, 1994.
- [15] H. Greenberg and R. L. Hegerich. A branch search algorithm for the knapsack problem. *Management Science*, 16, 1970.
- [16] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of ACM*, 21:277–292, 1974.
- [17] Adriana Iamnitchi and Ian Foster. A problem specific fault tolerance mechanism for asynchronous, distributed systems. In *Proceedings of 2000 International Conference on Parallel Processing (29th ICPP'00)*. IEEE, 2000.
- [18] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 21:972–989, 1963.
- [19] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP*, 11, 1997.
- [20] S. Martello and P. Toth. A new algorithm for the 0-1 knapsack problem. *Management Science*, 34:633–644, 1988.
- [21] D. Pisinger. An expanding-core algorithm for the exact 0-1 knapsack problem. *European Journal of Operational Research*, 87:175–187, 1995.

- [22] D. Pisinger. Where are the hard knapsack problems? *Computers and Operations Research*, 32:2271–2284, 2005.
- [23] D. Pisinger and P. Toth. Knapsack problems. In D. Z. Du and P. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 1–89. Kluwer, 1998.
- [24] Seti@home: Search for extraterrestrial intelligence at home. <http://setiathome.ssl.berkeley.edu>.
- [25] Y. Tanaka, M. Hirano, M. Sato, H. Nakada, and S. Sekiguchi. Performance evaluation of a firewall-compliant globus-based wide-area cluster system. In *9th IEEE International Symposium on High Performance Distributed Computing (HPDC 2000)*, pages 121–128, 2000.
- [26] S. Tschöke and T. Polzer. Portable parallel branch and bound library user manual, library version 2.0. Technical report, Department of Computer Science, University of Paderborn, Paderborn, Germany, 1998.
- [27] Y. Xu, T.K. Ralphs, L. Ladányi, and M.J. Saltzman. ALPS: A framework for implementing parallel search algorithms. In *The Proceedings of the Ninth Conference of the INFORMS Computing Society*, 2005. To appear, Available from <http://www.lehigh.edu/~tkr2/research/pubs.html>.