Master-Worker: An Enabling Framework for Applications on the Computational Grid *

Jean-Pierre Goux [a] Sanjeev Kulkarni [b] Jeff Linderoth [c] Michael Yoder [d]

[a] *Department of Electrical and Computer Engineering*
Northwestern University
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue Argonne, Illinois 60439
E-mail: goux@mcs.anl.gov
[b] *Computer Sciences Department*
University of Wisconsin - Madison
1210 West Dayton Street, Madison, WI 53706
E-mail: sanjeevk@cs.wisc.edu
[c] *Mathematics and Computer Science Division*
Argonne National Laboratory
9700 South Cass Avenue, Argonne, Illinois 60439
E-mail: linderot@mcs.anl.gov
[d] *Computer Sciences Department*
University of Wisconsin - Madison
1210 West Dayton Street, Madison, WI 53706
E-mail: yoderme@cs.wisc.edu

The goal of this work is to create a tool that allows users to easily distribute large scientific computations on computational grids. Our tool MW relies on the simple master-worker paradigm. MW provides both a top level interface to application software and a bottom level interface to existing grid computing toolkits. Both interfaces are briefly described. We conclude with a case study, where the necessary Grid services are provided by the Condor high-throughput computing system, and the MW-enabled application code is used to solve a combinatorial optimization problem of unprecedented complexity.

Keywords: Grid Computing, Metacomputing, Master-Worker Paradigm

1. Introduction

By its very definition, the Grid [14] is a powerful and complex computing environment. In order to help harness its power, a large number of different programming efforts are underway that seek to provide robust middleware services [13] [17] [21] [12] [4] [27]. For users hoping to parallelize a large, single, coordinated application over the Grid, the overhead required to learn and assemble these Grid-enabling software components could (at this stage of their implementation) be discouraging. Thus, to enable a larger community of users to build applications running in parallel on the Grid, higher-level programming frameworks leveraging existing Grid services software are needed. NetSolve [7] provides an API to access and schedule Grid resources in a seamless way but it is not suited for writing non-embarrassingly parallel codes. Everyware [28] is a heroic effort that shows that an application can draw computational power transparently from the Grid, but Everyware is not abstracted as a programming tool at this stage of its implementation. CARMI/Wodi [25] was a useful programming interface for developing master-worker based parallel applications to run on the Grid, but it was strongly tied to the Condor-PVM [24] software tool, limited to applications with fixed work cycles, and finally abandoned.

Our abstract programming framework MW is a complete, easy to use tool whereby users can distribute large, diverse, scientific computations in a Grid computing environment. To parallelize such algorithms on Grid computing platforms, users must address issues such as fault tolerance, task scheduling, and interprocess communication. By handling some of these issues automatically and exposing others, MW provides an API for rapidly implementing Grid-enabled master-worker algorithms. MW also abstracts an Infrastructure Programming Interface (IPI) such that it can be ported to use various Grid software toolkits without any changes from the application developer. MW has been used in the MetaNEOS project [23] to implement efficient parallel numerical optimization algorithms with complex control structures. The marriage of efficient algorithms with Grid computational resources has allowed the solution of problems of record-breaking sizes [3] [19]. Other authors have also focused on providing support for master-worker applications in a dynamic computing environment; Piranha [15] and Bayanihan [26] are notable examples.

The paper is organized as follows. In Section 2 we explain the goals of our Grid computing tool, and we argue that the master-worker paradigm is well

suited to meeting these goals. In Section 3, we introduce MW, and we describe the interfaces to both application software and Grid infrastructure software. Section 4 discusses additional features of MW that help developers build efficient and robust applications. Section 5 presents a case study where the Grid services are provided by Condor[21], and the application code is used to solve a combinatorial optimization problem of unprecedented complexity. Conclusions about this line of research are also given.

2. The Master-Worker Paradigm

The dynamic and heterogenous nature of the Grid makes parallel algorithm implementation a daunting issue for potential users. We aim to ease this burden on the users with a runtime support library. Ideally, the runtime support library should have the following characteristics:

(I) **Programmability**. Users should easily be able to take an existing application code and interface it with the system.

(II) **Adaptability**. The system should transparently (to the user) adapt to the dynamic and heterogeneous execution environment. Thus, new resources of varying types should be seamlessly integrated into the computation at any point.

(III) **Reliability**. The system should perform the correct computations in the presence of processors failing.

(IV) **Efficiency**. The system should be effective in the *high-throughput* sense [22]. That is, the resources should do useful work over long time periods.

In this section, we address the effectiveness of the master-worker paradigm in light of its ability to meet each of these design goals.

The master-worker paradigm is very easy to program. All algorithm control is done by one processor—the master. The user need not be burdened with the difficult issue of how to distribute algorithm control information to the various processors. Moreover, the typical parallel programming hurdles of load balancing and termination detection are circumvented. Having a central point of control facilitates the collection of a job's statistics. Furthermore, a surprising number of sequential approaches to large-scale problems can be mapped naturally to the master-worker paradigm. Tree search algorithms [18], genetic algorithms [6],

parameter analysis for engineering design [1], and Monte Carlo simulations [5] are just a few examples of natural master-worker computations. All these features increase a system's ease of use, accomplishing goal (I) of our Grid runtime support library.

Programs with centralized control are easily able to adapt to a dynamic and heterogeneous computing environment. If additional processors become available during the course of the computation, they simply become workers and are given portions of the computation to perform. Having centralized control also eases the burden of adapting to a heterogeneous environment, since only the master need be concerned with the matchmaking process of assigning tasks to resources making the best use of the resource characteristics. Thus, we are able to accomplish goal (II) for our Grid runtime support library.

Using the master-worker paradigm, we can easily achieve goal (III) of our Grid computing tool and ensure that the computation is fault-tolerant. If a worker fails and is executing a portion of the computation, the master simply reschedules that portion of the computation. A small difficulty is that the basic master-worker paradigm is not robust in the presence of failure of the master. To overcome this liability, the state of the computation can be occasionally check-pointed. This is a simple matter, since all state information is located in the master process.

Bulding efficient parallel implementations with the master-worker paradigm is not straightforward, especially considering the number of resources that the Grid can provide. In this context, there are two main roadblocks to efficiency: scalability and task dependence. To overcome these problems, it may be necessary to exploit algorithm characteristics such as dynamic grain size or a weak synchro-nization requirement. These characteristics may be inherent in the algorithm, or the algorithm may be modified in order to highlight these characteristics. The specific details on how to use these advantageous characteristics is problem de-pendent and beyond the scope of this paper, but our Grid runtime support library must give the users the flexibility to exploit them when necessary.

In short, the master-worker paradigm is simple enough so that implemen-tation difficulties associated with the computational grid can be easily resolved, yet we feel that the paradigm is flexible enough to accomidate a wide range of algorithms.

3. MW

MW is a software framework that allows a user to easily parallelize a master-worker application on Grid resources. MW is a set of C++ abstract classes providing interfaces to both application programmer and Grid-infrastructure programmer. To Grid-enable an application with MW, the application programmer must re-implement a small number of virtual functions. Likewise, to port the MW framework to a new Grid software toolkit, the Grid infrastructure programmer need only re-implement a small number of virtual functions.

3.1. Infrastructure Interface

To distribute a master-worker computation on the Grid, we at least require software that can perform

- Communication – Portions of the computation and results must be passed between master and workers,
- Resource Management – The state of the available computational resources on the Grid must be known.

Our usage of the term *resource management* is a bit broader than most. In this context, resource management encompasses

- Resource request and detection – Asking for and identifying available processors,
- Infrastructure querying – Determining information about processors and the interconnections between them,
- Fault-detection – Noticing when processors leave the computation,
- Remote execution – Starting processes on remote machines when they become available.

There are a number of tools being built that provide these basic services, as well as features necessary to other Grid applications (such as security and remote data access). The *Infrastructure Programming Interface* (IPI) abstracts the core communication and resource management requirements for master-worker applications into the MWRMComm class. To allow MW applications to interact with existing Grid-services software, a concrete instance of the abstract MWRMComm class is derived, where the functionality required by MWRMComm is provided by the services in the specific Grid software toolkit.

3.1.1. Communication

The sole communications functionality required by MWRMComm is that point-to-point messages can be sent between the master and the worker processes. As such, MWRMComm has the (pure) virtual functions:

- `pack(<type> array, int size)`
- `unpack(<type> array, int size)`
- `send(int to_whom, int message_tag)`
- `recv(int from_whom, int message_tag)`

All messages must be buffered by the MWRMComm implementation, and the `send()` function should be implemented as a nonblocking call. These design criteria are due to the fact that processors may disappear during the course of the computation. Since the Grid is heterogeneous, the `pack()` and `unpack()` functions must account for different native data types. In MWRMComm, the `recv()` routine should be implemented as a blocking function call, for reasons described in Section 3.1.2.

3.1.2. Resource management

The application programmer may make a resource requests by calling the function `MWRMComm::set_target_num_workers( int num_workers )`. It is up to the MWRMComm implementation to make appropriate resource requests in an attempt to garner this number of workers for the master-worker application, and also to make new requests if participating workers leave the computation.

An important design decision for MW is that both communication and resource management functionality is included in a common class. MW relies on an upcall mechanism from the resource management software to signal changes in the state of the computational resources. The changes are signalled as messages received by the master with specific tags such as `HOSTADD` and `HOSTDELETE`. Thus, an implementation of the (blocking) `MWRMComm::recv()` function on the master process should not only test for incoming messages from workers, but also check for changes to the state of the existing computational resources and report these changes as messages.

For example, when a `HOSTADD` message is received, the MWRMComm specification requires that the function call `MWRMComm::start_worker(MWWorkerID *w)` will (attempt to) start a remote process on the machine that has been added, and will assign a unique process identifier in the MWWorkerID. When

a `HOSTDELETE` message is received, MWRMComm requires that the unique process identifier be packed in the message buffer.

A final important function in the MWRMComm class is `MWRMComm::get_worker_info( MWWorkerID *w )`. This function uses underlying Grid services to populate the MWWorkerID class with "useful" information about the remote processor. Data members of the MWWorkerID class include the architecture, operating system, amount of memory, disk space, and speed of the remote machine.

Clearly, this is not the entire specification of the MWRMComm class. Indeed, we consider the IPI that we have laid out in MW to be a work in progress. The interface will likely change, and additional functionality will be added as warranted. Due the layered design of MW, application programs will be shielded from the interface changes.

3.1.3. Example MWRMComm Implementations

There are currently two implementations of the MWRMComm class. Both rely on the resource management facilities provided by the Condor high-throughput computing system [21]. As such, MW must deal with many processor faults, since the default Condor behavior is to vacate a running process when the owner of the machine returns. In one implementation, communication is done with PVM, and in the other, communication is done by using Condor's remote I/O mechanism [20] to write a series of shared files. Preliminary plans are being made for a port to the Globus software toolkit [13]. Table 1 highlights how the Grid service software provides (or could provide) the functionality required by MWRMComm.

3.2. MW Application Programming Interface

In Section 2, we argue that many scientific applications can be parallelized quite effectively for a Grid environment by using the master-worker paradigm. Algorithms that are best suited for a master-worker implementation share the following characteristics:

- Incremental Data Requirement,
- Weak Synchronization,
- Dynamic Grain Size.

Services	Condor-PVM	Condor-Files	Globus
Communi-cation	Messages buffered and passed through default PVM mechanisms	Messages passed through shared worker files via Condor Remote I/O.	Messages passed and handled via Nexus `nexus_ send_rsr()`.
Resource Request and Detection	Requests formulated with Condor Class Ads, served by Condor matchmaking, and detection is notified by `pvm_ notify()`.	Requests formulated with Condor Class Ads, served by Condor matchmaking and detected, by checking Condor logs.	Requests in Globus RSL handled and queued by GRAM via `gram_ client_job_ request()`.
Info Querying	Information collected via condor_status command	Information collected via condor_status command	Information queried from MDS via LDAP protocol.
Fault Detection	Faults detected by Condor-PVM and passed through `pvm_ notify()`.	Faults detected by checking Condor logs.	Faults detected by HBM local monitors are collected by HBM data collector agent running on master.
Remote Execution	Job started by `pvm_ spawn()`.	Job started by condor_startd daemon on remote resource.	Job started by GRAM when requests are served.

Table 1

Summary of How Grid Services are Provided

The MW API was designed to provide an interface that would be *easy* for application programmers to use, but also would allow these algorithmic characteristics to be exploited to build efficient master-worker applications.

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

3.2.1. MWDriver

To create the MWDriver, the user need only implement four pure virtual functions:

- `get_userinfo(int argc, char *argv[])` – Processes arguments and does basic setup.

- `setup_initial_tasks(int *n, MWTask ***tasks)` – Returns a set of tasks on which the computation is to begin.

- `pack_worker_init_data()`– Packs the initial data to be sent to the worker upon startup. Use of this function allows the application to exploit an *incremental data requirement.*

- `act_on_completed_task(MWTask *task)` – Is called every time a task finishes. Some actions that the user could take include adding more tasks or making calculations based on the result of the task. Tasks are added by calling the `MWDriver::addTasks(MWTask **tasks)` base method.

By carefully deciding on actions to take in the `act_on_completed_task()` method, the user can take advantage of *weak synchronization* inherent in the parallel application.

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. Further, the MWDriver offers more advanced functionality, as explained in Section 4.

3.2.2. MWTask

The MWTask is the abstraction of one unit of work. The class holds both the data describing that task and the results computed by the worker. By deciding on the size of the task, the application can use *dynamic grain size* to its advantage, easing contention at the master process, and increasing parallel efficiency. 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 MWRM-Comm class.

3.2.3. 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()`.

- `execute_task( MWTask *task )`– Given a task, computes the results.

After doing some basic initialization, the MWWorker 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. It is an easy matter to bring in other libraries, such as highly optimized FORTRAN routines to the worker. They can be linked with the C++ code, and called by the `execute_task()` function.

3.3. The MW Software Layer

Figure 1 depicts relationships between the software components enabling an application to utilize the computation grid. MW adds an addition software layer and acts as a filter, hiding complexity of Grid service software, but also potentially hiding underlying functionality and knowledge of how the communication and resource management services are performed. A significant challenge is how to impart this functionality and knowledge to the application programmer, while still presenting a simple interface. MW errs on the side of simplicity, with the thought that additional Grid service functionality will be made available to the application programmer as needed.

An advantage of the layered approach is that some advances in Grid services software can be leveraged by the application programmers to increase application performance. For our Condor-based MWRMComm implementations, two examples include flocking [11], where geographically distributed Condor pools are conceptually linked as one, and glide-in [10], where processors from an existing Globus resource can be added to a Condor pool on a temporary basis. These advanced Condor features are used by the application presented in Section 5.

4. Additional Functionality

In this section, we outline a number of other useful features that are available through methods in the base MWDriver class.

Figure 1. Components used to Grid-enable Application Software
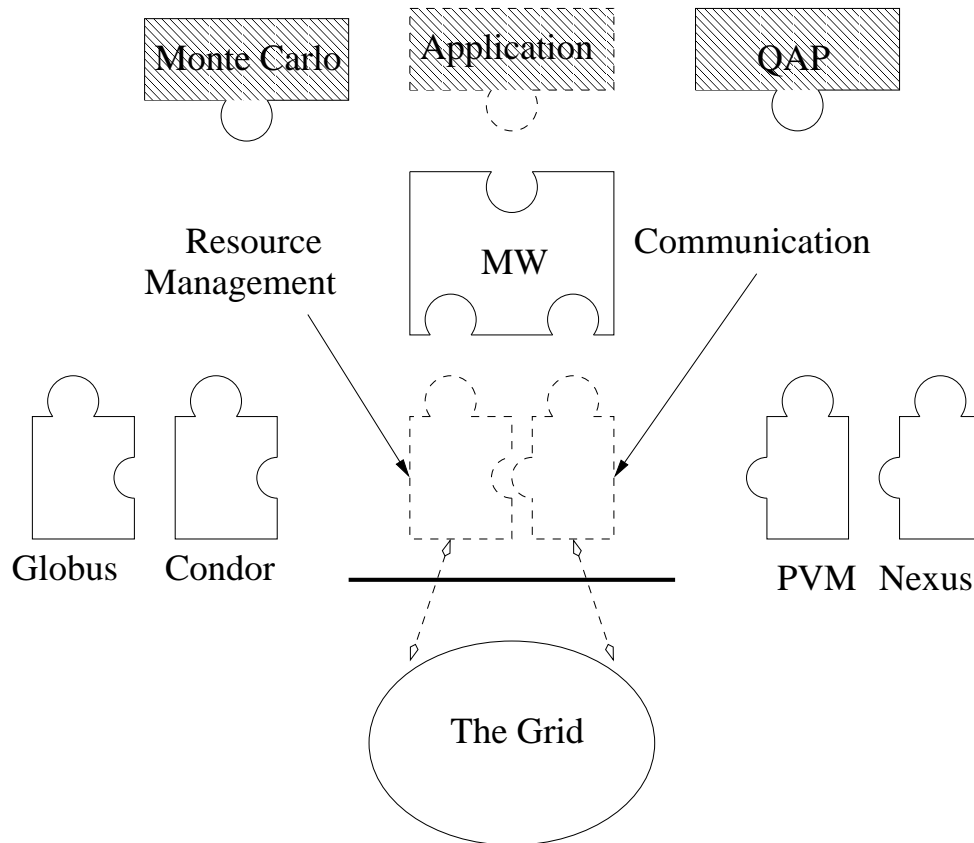
4.1. Checkpointing

Because the MWDriver reschedules tasks when the processors running these tasks fail, applications running on top of MW are fault tolerant in the presence of all processor failures—except for the master processor. In order to make computations fully reliable, MWDriver offers features to logically checkpoint the state of the computation on the master process on a user-defined frequency. To enable checkpointing, the user must implement functions for writing and reading the state contained in its application's master and task classes. Use of the master checkpoint facility is demonstrated in Section 5.

4.2. Normalized Performance Measurement

The heterogeneous and dynamic nature of a computational grid makes application performance difficult to assess. Standard performance measures such as wall clock time and cumulative CPU time do not separate application code performance from computing platform performance. By normalizing the CPU time spent on a given task with the performance of the corresponding worker, MW aggregates time statistics that are comparable between runs. The normalization factor can be based on vendor information such as MIPS or KFLOPS, if this information is available from the underlying Grid service software. Alternatively, MW allows the user to register an application specific benchmark task that is sent to all workers that join the computational pool. The speed at which the benchmark task is completed is used as the normalization factor.

Besides the normalized CPU time statistic $\mathcal{T}$, MW collects a number of other useful statistics such as the unnormalized CPU time T, the wall clock time ($\mathcal{W}$), the amount of time worker i was available (U_i), and the amount of CPU time (t_j) spent completing task j. At the end of the run, MW reports useful statistics such as the average number of available workers during the course of the run ($\mathcal{N}$) :

$$\mathcal{N} \equiv \frac{\sum_i U(i)}{\mathcal{W}},$$

and the parallel efficiency (η) :

$$\eta \equiv \frac{\sum_j t(j)}{\sum_i U(i)}.$$

Table 4.2 shows the variations of performance statistics between runs of a Grid-enabled application (presented in Section 5). The same problem instance was solved eight times, each time on a different set of processors. A user-defined benchmark task was used to define the normalization factor.

As expected, the statistics exhibit large variance of $\mathcal{W}$ and T due to the dynamic and heterogenous nature of the computing platform. However, there is little variance of $\mathcal{T}$, which can therefore be used to do comparisons between runs and assess the application performance. Use of the normalized performance measurement has proved invaluable for tuning parameters of various Grid-enabled applications, like the one presented in Section 5.

	Mean	Std. Dev.	Min	Max
$\mathcal{W}$	915	1019	489	1780
T	22182	27900	8844	37671
$\mathcal{T}$	5864	341	5739	6054
$\mathcal{N}$	27.5	21.7	16	39
η	0.87	0.07	0.84	0.92

Table 2

Mean, Variance and Extreme Value on 8 different runs.

4.3. Task Scheduling

Internally, the MWDriver manages a list of workers and a list of tasks. Task scheduling is accomplished by assigning the first task in the task list to the first idle worker in the worker list. In MWDriver, there is an interface to specify that the task list be ordered by a user-defined key, ensuring that "important" tasks are performed first. The worker list may be similarly ordered, so that "good" machines are the first to receive tasks. By default, the worker list is ordered using the machine KFLOPS information (if provided by the Grid software implementing MWRMComm), or by the benchmark factor if the user has registered an application specific benchmark task.

While this is a rudimentary scheduling algorithm, it has proven sufficient for all applications implemented to date with MW. The applications have had no need to match specific tasks with specific workers. Also, the applications to date have not been data-intensive, so use of advanced services such as the Network Weather Service [29] to improve scheduling has not been warranted.

5. Application to Combinatorial Optimization

MW has been used in the MetaNEOS project [23] to implement several grid-enabled parallel optimization solvers [9] [16] [19]. One solver has been specialized to solve the quadratic assignment problem (QAP) [8]. Despite its simple statement—to minimize the assignment cost of n facilities to n locations—it is extremely difficult to solve even modest sized instances of the QAP. Problems with $n > 20$ are difficult; problems with $n > 30$ were thought unsolvable. By embedding a new relaxation technique [2] into a branch-and-bound framework,

and implementing the resulting solver within MW, we managed to solve what is regarded by experts in the field as the most difficult QAP instance (size $n=27$) to provable optimality [3].

In order to use the computational resources with maximum efficiency, the parallelization strategy of the branch-and-bound tree search has been carefully designed. Issues such as the proper ordering of the task list and the selection of the grain size were carefully considered in order to minimize communication overhead and contention at the master process without introducing large parallel search anomalies. By using the intuitive MW API, implementing the parallel version of the sequential branch-and-bound code was extremely simple and fast. The MW-ized QAP application code was compiled to use the Condor/File-Based `MWRMComm` implementation.

The computational pool was composed of machines from the Condor pool and a Linux cluster at the University of Wisconsin, a flocked Condor pool at the University of New Mexico, a flocked Condor pool at the National Institute for Nuclear Physics (Bologna, Italy), and the SGI/Origin2000 at Argonne National Laboratory acquired via Globus through the glide-in mechanism. Further information about the computational pool is summarized in Table 3.

Number	Arch-OS	Where	How	(Peak) GFLOPS
179	INTEL/LINUX	Wisc	Main Pool	13.88
34	INTEL/LINUX	UNM	Flocked	1.12
64	INTEL/LINUX	INFN	Flocked	2.76
150	INTEL/SOLARIS	Wisc	Main Pool	7.64
35	SUN/SOLARIS	Wisc	Main Pool	1.44
8	SUN/SOLARIS	INFN	Flocked	0.38
32	SGI/IRIX	Argonne	Glide-in	3.84
502	-	-	-	31.06

Table 3
The Computational Pool.

Figure 2 depicts the cumulative evolution of the number of machines of each

type during our run. A few events are of note. At 11:30AM a glide-in request was made for 32 SGI processors on Argonne's Origin for a period of 12 hours. (The reader can note these machines appear in Figure 2 around this time). At 6:30 PM, the Condor scheduling daemon was reconfigured to allow flocking with the INFN Condor pool in Bologna, Italy. The job was stopped manually at 11PM, and we restarted it at 8AM from the master's checkpoint file, as explained in Section 4.1. When restarted, we did not place a new glide-in request.

In all, 87,036 tasks, each consisting of a number of nodes of the branch and bound tree, were sent from the master to workers. It is impossible to predict the number of nodes in a task, resulting in a wide variance in task grain sizes. The task grain sizes varied from 0.01 CPU seconds to over 1200 CPU seconds, with a mean value of 190.6 seconds. 567,793,866 nodes were explored in solving the problem. Figure 3 shows a moving average of the number of nodes evaluated per second. Over the course of the computation, we used an average of 211.3 machines and with a peak of 285. The parallel efficiency obtained during the run was $\eta = 0.83$. The average performance of the computational pool was 195 times the performance of one of the dedicated Linux nodes. Neglecting parallel search anomalies, the solution of this problem in sequential would have required around over 177 days of computation with the sequential algorithm on a dedicated Linux node. The marriage of Grid resources with the advanced algorithm allowed the solution of a heretofore unsolved problem.

6.   Conclusions and Future Work

MW has allowed algorithm developers to bring together a large number of heterogeneous, geographically dispersed resources to solve extremely large problems. The simple API of MW provided a convenient programming model enabling the user to focus on algorithmic features without worrying on the details of setting up computations, and the IPI has allowed a better portability of the resulting code to different grid computing environments.

It is the continued goal of this work to draw further application developers by providing a simple interface, access to Grid resources, and useful functionality at no expense to the application code. We also wish to entice Grid infrastructure developers to support MW by providing a simple, well-defined interface, and interesting and useful applications. There is still work to be done to turn these goals into realities.
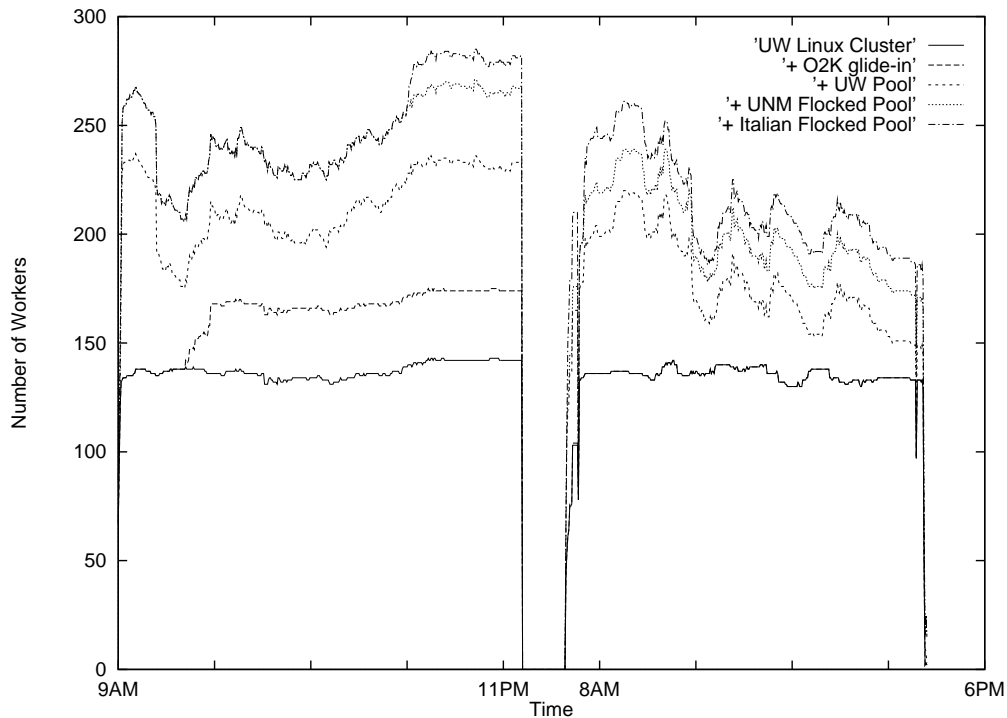
Figure 2. Number of Workers

Further information about MW is available from

`http://www.cs.wisc.edu/condor/mw`

Acknowledgments

The authors would like to sincerely thank the whole Condor team for their tireless efforts and their timely responses to our never-ending queries. In particular, we thank Jaime Frey for his help with the glide-in feature, and Miron Livny for his enthusiastic support. We also thank Nate Brixius and Kurt Anstreicher for allowing us to use their application code in the case study, and for keeping us excited about building a parallel Grid interface. The comments of anonymous referees helped improve the presentation.

References

[1] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations. In *Symposium on High Performance*
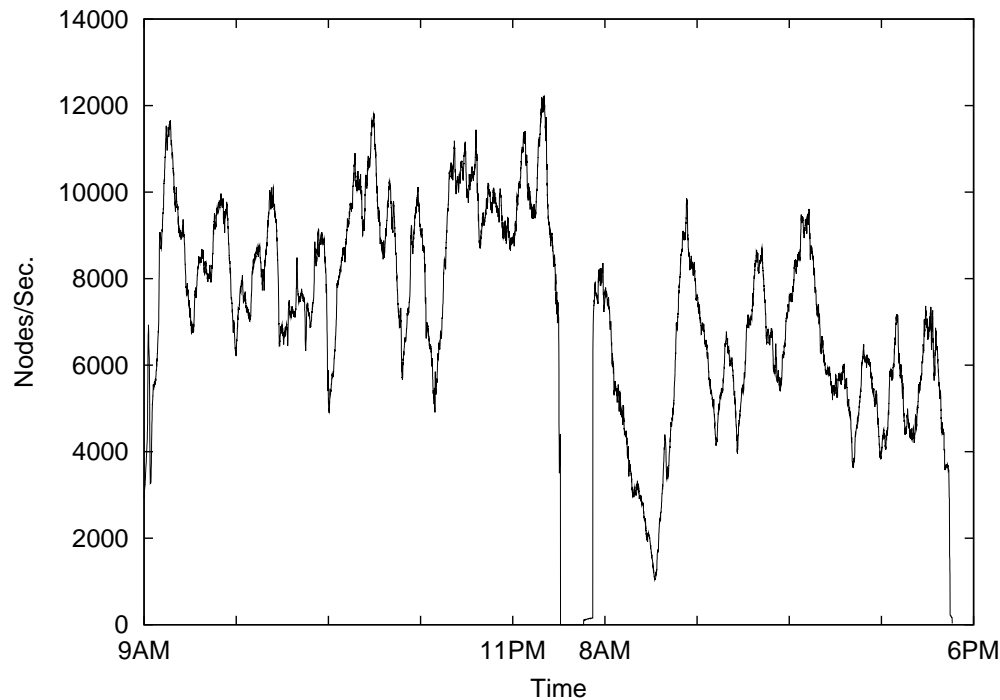
Figure 3. Nodes Per Second

Distributed Computing, Virginia, August 1995. Available from `http://www.dgs.monash.edu.au/~davida/papers/nimrod.ps.Z`.

[2] K. Anstreicher and N. Brixius. A new bound for the quadratic assignment problem based on convex quadratic programming. Technical report, Department of Management Sciences, University of Iowa, 1999. Available from `http://www.biz.uiowa.edu/faculty/anstreicher/qapqp.ps`.

[3] K. Anstreicher, N. Brixius, J.-P. Goux, G. Hudek-Davis, and J. Linderoth. Location theory gives rise to QAP problem. *data link*, March 2000. The Alliance Online Technical News Letter, available from `http://www.ncsa.uiuc.edu/SCD/Alliance/datalink/0003/QA.Condor.html`.

[4] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. *International Journal on Future Generation Computer Systems*, 15:559–570, 1999.

[5] J. Basney, R. Raman, and M. Livny. High throughput Monte Carlo. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, 1999.

[6] E. Cantu-Paz. Designing efficient master-slave parallel genetic algorithms. In J. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. Fogel, M. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming: Proceedings of the Third Annual Conference*, San Francisco, 1998. Morgan Kaufmann.

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

[8] E. Cela. *The Quadratic Assignment Problem – Theory and Algorithms.* Kluwer, 1998.

[9] Q. Chen, M. Ferris, and J. Linderoth. Fatcop 2.0: Advanced features in an opportunistic mixed integer programming solver. *Annals of Operations Research*, 2000. To appear, Available from `http://www.mcs.anl.gov/metaneos/fatcop2.ps`.

[10] The Condor Team. *Extending your Condor pool by Gliding into Globus-controlled machines*, 2000. `http://www.cs.wisc.edu/condor/manual/v6.1/2_12Extending_your.html`.

[11] 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*, 12:67–85, 1996.

[12] G. Fagg, K. Moore, and J. Dongarra. Scalable networked information processing environment (SNIPE). *International Journal on Future Generation Computer Systems*, 15:595–605, 1999.

[13] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. J. Supercomputer Applications*, 1997. Available as `ftp://ftp.globus.org/pub/globus/papers/globus.ps.gz`.

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

[15] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage : preliminary experience with Piranha. In *ACM, International Conference on Supercomputing,*, July 1992.

[16] J.-P. Goux and Sven Leyffer. Mixed-integer nonlinear programming on metacomputing platform. Working Paper, 1999.

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

[18] V. Kumar, K. Ramesh, and V. N. Rao. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the 1988 National Conference on Artificial Intelligence*, pages 122–127, August 1988.

[19] J. Linderoth and S. Wright. Solving large stochastic programs in a metacomputing environment. Invited Presentation at *APMOD – Applied Modelling for Optimization*, April 2000.

[20] M. Litzkow. Remote Unix – Turning idle workstations into cycle servers. In *Proceedings of Usenix Summer Conference*, 1987.

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

[22] M. Livny and R. Raman. High-throughput resource management. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kauffmann, 1999.

[23] The MetaNEOS Project. *Metacomputing Environments for Optimization*, 2000. `http://www.mcs.anl.gov/metaneos`.

[24] J. Pruyne and M. Livny. Providing resource management services to parallel applications. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, 1994. Available as `http://www.cs.wisc.edu/condor/doc/condor_pvm_framework.ps.Z`.

[25] J. Pruyne and M. Livny. Interfacing Condor and PVM to harness the cycles of workstation clusters. *Journal on Future Generations of Computer Systems*, 12:53–65, 1996.

[26] L. Sarmenta and S. Hirano. Bayanihan: Building and studying volunteer computing systems using java. *Future Generation Computer Systems*, 15(5/6):675–686, 1999.

[27] M. van Steen, P. Homburg, and A. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7:70–78, 1999.

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

[29] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generations of Computer Systems*, 15:757–768, 1999.