Preprint ANL/MCS-P792-0200, February 2000
Mathematics and Computer Science Division
Argonne National Laboratory

Jean-Pierre Goux · Jeff Linderoth* · Michael Yoder

Metacomputing and the Master-Worker Paradigm**

February 9, 2000

Abstract. The goal of our work is to create a tool that easily allows users to distribute large scientific computations in metacomputing environments. To achieve this goal, a number of difficult implementation issues must be addressed, which may explain the relative lack of *complete* tools addressing this purpose. Our tool relies on the simple master worker paradigm, and we show that this paradigm is nicely suited for performing many of the requisite tasks of our metacomputing tool. We describe an implementation and present a case study showing the paradigm's effectiveness in solving large scientific computing problems.

1. Introduction

As a result of decreasing computer hardware costs and the increasing connectivity between computers, typical users now have access to more computational resources than ever before. When large sets of resources are connected by local or wide area networks or the Internet, they can be assembled into "metacomputers" and used to solve large and complex scientific computing problems.

Before users can take advantage of metacomputers, however, there are numerous issues that must be confronted. To bring together computational resources to attack a single problem, existing algorithms must be made to work in a distributed fashion. The problem of distributing a computation has been studied and is understood in the context of traditional computing environments, but less is known about how to effectively distribute computations in a metacomputing environment. Projects such as Condor [LBRT97], Charlotte [BKKW99], Legion [GFKH99], SNIPE [FMD99], MOL [RBD+97], and Globus [FK97] all provide basic software infrastructure for supporting metacomputing. For application programmers, however, using this infrastructure to build a metacomputing application can be quite difficult. The goal of our work, then, is to

Jean-Pierre Goux: Department of Electrical and Computer Engineering, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208, goux@ece.nwu.edu

Jeff Linderoth: Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439 linderoth@mcs.anl.gov

Michael Yoder: Computer Sciences Department, University of Wisconsin - Madison, 1210 West Dayton Street, Madison, WI 53706, yoderme@cs.wisc.edu

Mathematics Subject Classification (1991): ??, ??, ??

* Research of this author 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.

** This work was supported in part by Grant No. CDA-9726385 from the National Science Foundation.

develop a complete, easy-to-use tool whereby users can distribute large, diverse scientific computations in a metacomputing environment. The goals of this project are quite similar to that of the EveryWare toolkit [WBK+99].

The primary purpose of this paper is to show that the implementation difficulties encountered when distributing computations on a metacomputer can easily be resolved through the use of the master-worker paradigm. Furthermore, we describe our implementation of a master-worker framework that enables users quickly and easily to build master-worker applications that will run on metacomputing platforms.

The paper is organized as follows. In Section 2, we describe the distinguishing characteristics of our target computing environment, we mention features of an ideal parallel metacomputing tool for distributing large scientific computations, and we discuss the degree to which the master-worker paradigm can achieve this ideal. Section 3.2 describes the framework from which metacomputing-based, master-worker applications can be easily built. In Section 4, we demonstrate how to exploit algorithm characteristics in order to build efficient master-worker implementations. Section 5 presents a case study showing the effectiveness of the master-worker approach in solving a complex mathematical optimization problem. We conclude by giving some future directions of research.

2. An Effective Paradigm for Metacomputing

The term *metacomputing* is generally used to denote the idea of bringing together diverse, heterogeneous, possibly geographically distributed computing environments in a seamless fashion in order to attack large-scale computing problems. A metacomputer differs from a traditional parallel computer in a number of ways, but two characteristics are important to this discussion:

- **Dynamic availability**. The quantity of resources available and the amount of computation delivered by a single resource may vary over time. Participating resources may also disappear without notice.
- **Heterogeneity**. Resources may have varying physical characteristics, such as architecture, operating system, amount of memory, and processor speed.

2.1. A Parallel Metacomputing Tool

The dynamic environment of a metacomputing system raises the need for *resource management software* that detects when processors are available, determines when they leave the computation, and matches jobs to available processors. Parallelizing a computation involves breaking the computation into elementary tasks, scheduling these tasks, and making the results of the tasks available. Therefore, to enable parallel applications to run on metacomputers, we require a tool that performs resource management, facilitates the decomposition of the problem into manageable computational subtasks, and enables the exchange of information between processors. Ideally, this parallel metacomputing tool 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 [LR99]. That is, the resources should do useful work over long time periods.

In a heterogeneous and unreliable computing environment, building an efficient high-throughput system is a more realistic goal than aiming for traditional high-performance computing metrics such as FLOP rates. The focus of high-throughput computing is on the amount of useful work done over long time spans. Since the problems that we target to solve will require days or weeks of computing time, it makes sense to build a system that is effective on this time scale.

2.2. The Master-Worker Paradigm

In this section, we address the effectiveness of the master-worker paradigm in light of its ability to meet each of the goals of our parallel metacomputing system.

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 [KRR88], genetic algorithms [CP98], parameter analysis for engineering design [ASGH95], and Monte Carlo simulations [BRL99] 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 parallel metacomputing system.

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 parallel metacomputing tool.

Using the master-worker paradigm, we can easily achieve goal (III) of our metacomputing 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 checkpointed. This is a simple matter, since all state information is located in the master process.

Attaining efficiency of our parallel metacomputing tool (goal (IV)) is not completely straightforward, but the master-worker paradigm *can* be used to build efficient implementations in the high-throughput sense. In this context, there are two main roadblocks to efficiency: scalability and task dependence. The master-worker paradigm is not scalable, since as the number of workers increases, there may be a bottleneck at the master, because it attempts to deal with the many requests from the workers. By task dependence, we mean the degree to which the start of one task computation depends on the completion of other task computations. A high degree of task dependence can greatly decrease efficiency in metacomputing environments, since processing and communication times of tasks and results are highly variable. The combination of this variability and task dependence can result in a large number of idle processors that are waiting on the completion of a small number of tasks. To overcome the potential problems with the master-worker paradigm in a metacomputing environment, it may be necessary to exploit certain characteristics of the parallel algorithm. These characteristics may be inherent in the algorithm, or the algorithm may be modified in order to highlight these characteristics.

One algorithm characteristic that can be exploited is *dynamic grain size*, which refers to the ability to break the computation into portions of work of variable size. We can use a dynamic grain size in order to increase a program's scalability. For example, in a tree search, the master can assign larger portions of the tree to each processor, reducing the rate at which the processors make work requests. However, this contention reduction technique comes at the expense of a loss in algorithm control, since workers now process for longer amounts of time without any feedback on whether their search is useful. Section 4.2 gives a case study showing the importance of this algorithm characteristic.

A second algorithm characteristic that can be exploited is an *incremental data requirement*, referring to the situation in which a large amount of data is required to initialize a worker process, while each individual task can be defined by a relatively small amount of additional data. Scalability problems at the master can stem from the large amount of network traffic to the workers. Only passing the incremental data required for the task reduces the necessary bandwidth and increases the scalability.

A final algorithm characteristic that can be exploited is a *weak synchronization requirement*, or a low task dependence, meaning that the ability to execute a task does not depend on the completion of a large number of other tasks. In a cutting plane algorithm for stochastic programming [KW94], each iteration consists of a number of tasks. However, the correctness of the algorithm does not depend on all of the tasks completing before proceeding to the next iteration. The synchronization requirement can be reduced, and the efficiency increased, by starting the next iteration after only a certain number of the previous iteration's tasks have completed. The significance of exploiting this algorithm characteristic will be demonstrated in Section 4.1.

Note that increasing the efficiency of the algorithm by increasing the grain size or by reducing the synchronization requirement may actually worsen the basic algorithm; more tree nodes might be explored or more iterations required. This algorithm deterioration is made in the hope of obtaining a higher parallel efficiency and lower overall computation time. Users of metacomputers and the master-worker paradigm should be keenly aware of this tradeoff, however.

One final point can be made in the case where the task dependencies of a computation can be grouped into "work cycles", where the whole algorithm is blocked until a certain set of tasks is completed. As noted before, ideally the parallel algorithm would not require this much task dependence. If work cycles are unavoidable, and tasks in a work cycle are not of equal size, or the processors performing the tasks are not of equal speed, then many processors may be left idle. An advantage of using the master-worker paradigm in this case is that the master can attempt to balance the work cycles by varying the size of the tasks sent to various processors. Pruyne and Livny [PL96] have made a similar observation.

3. A Software Framework

We have developed a software framework that meets the goals of programmability, adaptability, reliability, and efficiency outlined in Section 2.1. Based on the master-worker paradigm, it is called MW. In this section, we briefly discuss the resource management system used by MW (Section 3.1) and then describe the abstract base classes of MW (Section 3.2).

3.1. Resource Scheduling and Message Passing

MW uses Condor [LBRT97] as its resource management system. Condor differs from many other resource management systems in that it harnesses nondedicated computing resources that would otherwise go wasted. The Condor system matches user-submitted jobs with idle machines in its "pool". Specifically, MW uses Condor's support for the popular PVM software package [PL96]. PVM provides the necessary message-passing interface between the master and workers. Condor's resource management decisions are passed to the master process through special PVM messages. The master is held on a dedicated processor, and the workers are nondedicated resources from the Condor pool. These relationship are depicted in Figure 1.

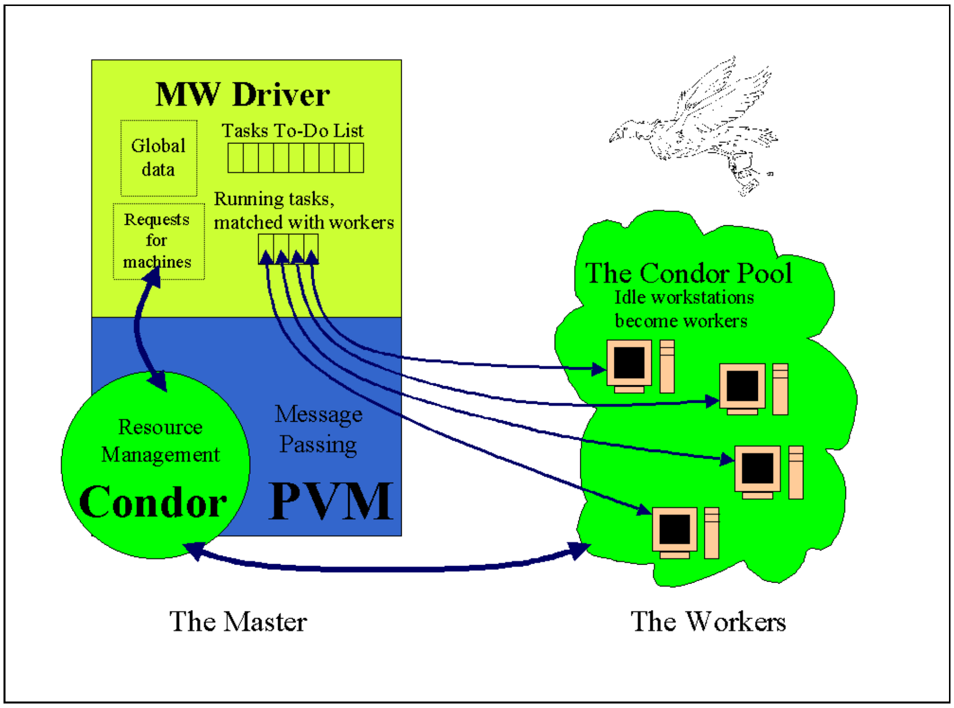

Fig. 1. Relationship between the MWDriver, Condor, and PVM.

Although we build our framework from these specific tools, the ideas presented in the preceding section are general enough to work with other scheduling and message-passing software. We are investigating porting our framework to other systems.

3.2. MW

MW is a set of C++ abstract base classes. These classes hide the difficult metacomputing issues, allowing for rapid development of sophisticated scientific computing applications. In the rest of this section, we describe the design of MW and how to build an application using MW.

Three abstract base classes must be reimplemented. The MWDriver class corresponds to the master process and contains the control center for distributing tasks to workers. The MWTask class describes the inputs and outputs—the data and results—that are associated with a single unit of work. The MWWorker class contains code to initialize a worker process and to execute any tasks that are sent to it by the master.

3.2.1. MWDriver To create the MWDriver—the master process—the user need only implement four pure virtual functions:

- `get_userinfo()`– Processes arguments and does basic setup. This is called once when the master is started up.
- `setup_initial_tasks()`– Returns a set of tasks for the computation to begin work on. This is executed once at the beginning.
- `pack_worker_init_data()`– Sends some initial data to a worker (if necessary) upon startup.
- `act_on_completed_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.

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 3.2.4.

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. The derived task class must also 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()`.

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()`– Given a task, computes the results.

After doing some basic initialization, the MWWorker sits in a simple loop. It asks the master for a task, 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.

3.2.4. Other MWDriver Features To make computations fully reliable, MWDriver offers features to checkpoint the state of the computation on a user-defined frequency. MWDriver can restart from that state after a crash of the master. To enable checkpointing, the user must implement functions for writing and reading the state contained in its application's master and tasks. This feature can be used to perform rudimentary "computational steering" if the user stops the computation by hand, modifies the checkpoint file or executable, and then restarts from that checkpoint.

To help the user make the best use of available resources, MWDriver has abstract mechanisms to sort the task pool according to user-supplied priorities. MWDriver also maintains information about each participating worker. This information can be used to develop advanced scheduling policies that match tasks with the best-suited workers.

By using the `set_target_num_workers()` function, the user can set the number of workers desired for a computation. This function can be called as needed to change the target size of the pool of workers. To deal with heterogeneous resources, MWDriver currently uses the multi-architecture features of Condor-PVM. The user compiles the workers for the targeted architectures, and the MWDriver selects the correct executable as new workers enter the computation.

4. Efficiency Considerations

This section contains two brief experiments showing the impact that varying algorithmic characteristics can have on the effectiveness of master-worker algorithms running in a metacomputing environment. MW has been used to build master-worker implementations of parallel algorithms for solving mathematical optimization problems, and we use these implementations for the purposes of our study. The experiments are not meant to be comprehensive, only to give the reader a feel for the issues that application programmers must consider when building efficient master-worker applications for metacomputing environments. The computing environment used for these studies was the University of Wisconsin Condor pool, consisting of over 350 heterogeneous, nondedicated processors.

4.1. Task Dependence

The first experiment shows the potential improvements resulting from reducing task dependence in the parallel algorithm. As mentioned in Section 2.2, each iteration of the cutting plane method for stochastic programming consists of a number of tasks that must be completed; however, all tasks need not complete before beginning the next iteration. The degree of task dependence can be varied in this algorithm by changing the percentage of tasks that must complete successfully before starting a new iteration. A parallel algorithm that exploits this weak synchronization requirement was built using MW. Figure 2 shows the efficiency of the parallel code, and Figure 3 shows the number of iterations required for convergence for various values of the task dependence percentage. Efficiency is defined as the ratio of total time that workers process tasks to the total time allocated to the workers by the resource management system. Each test has been repeated multiple times in an attempt to smooth the randomness inherent to metacomputing computations, and the average and extreme values are reported.

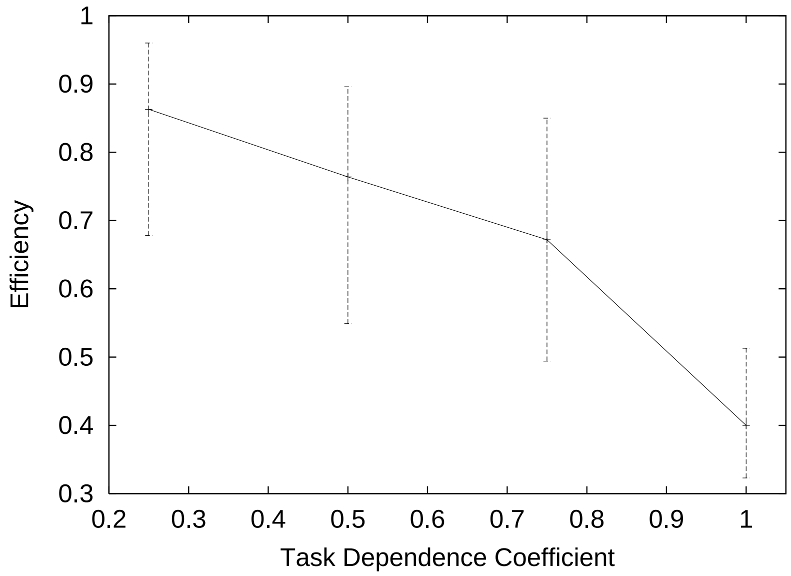

Fig. 2. Effect of Task Dependence on Efficiency

As expected, lowering the task dependence increases the algorithm's efficiency. The improved efficiency comes at the expense of having to perform more iterations.

4.2. Grain Size

The second experiment shows the effect of varying the grain size of the computation. The branch-and-bound algorithm is a tree search method. The grain size of the worker tasks is controlled by specifying the maximum number of nodes that a worker is allowed to visit before reporting to the master. Using MW, a parallel branch-and-bound algorithm for the quadratic assignment problem (QAP) was implemented. The QAP is a difficult combinatorial optimization problem, the aim of which is to minimize the (quadratic) assignment cost of n given facilities to n given locations. Different grain sizes have been tried on a size $n = 16$ problem. Average and extreme results for the efficiency and the number of nodes explored in the search are reported as a function of the grain size in Figures 4 and 5.

The effective use of the workers increases with the grain size but at the price of a larger total number of nodes explored. In order to minimize the overall run time for this application, a trade-off must be found between a high efficiency and a small number of nodes explored.

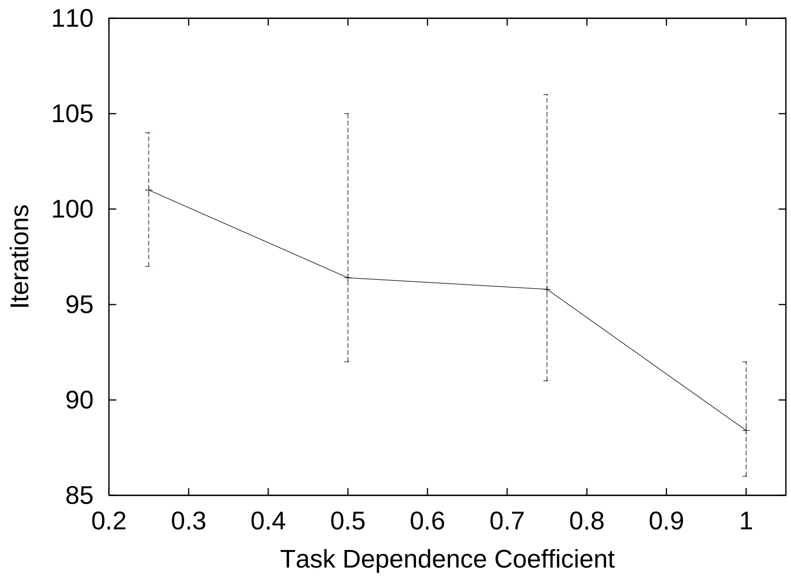

Fig. 3. Effect of Task Dependence on Iteration Count

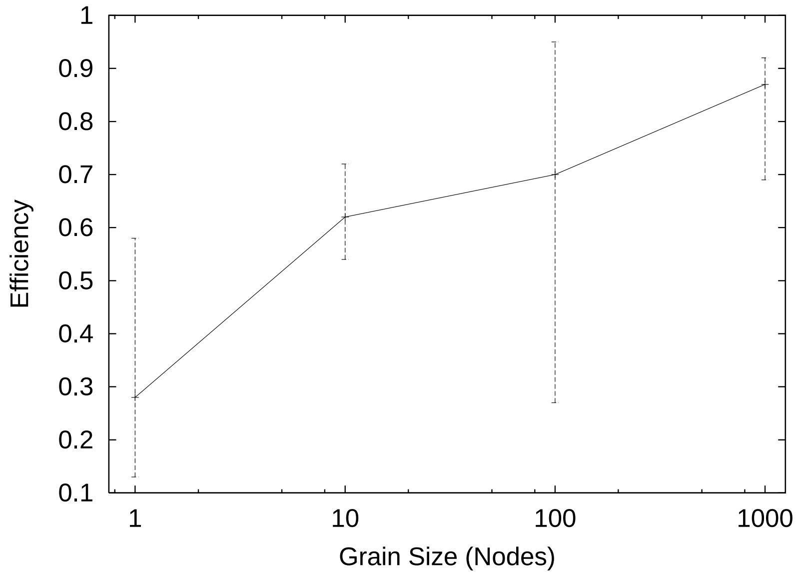

Fig. 4. Effect of Grain Size on Efficiency

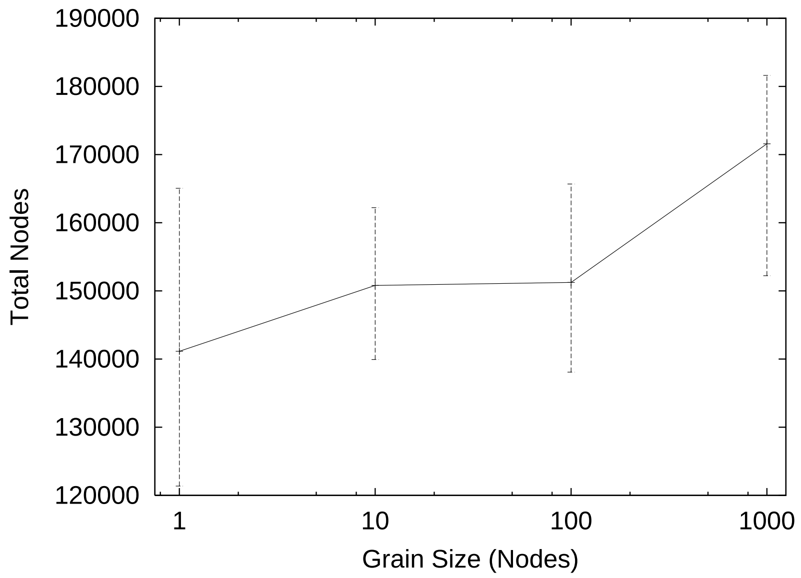

Fig. 5. Effect of Grain Size on Search Size

5. Case Study

In this section, a large case study is presented that demonstrates the power and usefulness of the MW software tool for solving computational problems requiring considerable resources over long time spans. Despite the simplicity with which the QAP is stated, it is still a computationally hard task to solve modest-size problems, say of size $n \geq 20$ [BKR97]. For instance, the solution of a size $n = 22$ problem (NUG22) on a single fast workstation requires 11 days of computation with a sequential version of the algorithm [AB99].

With MW, a parallel version of the sequential code was rapidly prototyped. Based on insight gained from the results of the preceding experiment, we chose a grain size of 1000 nodes maximum per task. The code solved NUG22 in around 11 hours on idle machines from the University of Wisconsin Condor pool. Figure 6 shows the evolution of the number of workers during the computations. At 4:50 AM the master machine was shut down for a system update, but the job was automatically restarted from the checkpoint file as explained in Section 3.2.4.

Participating in this run were 116 machines of type Intel/Solaris and 39 of type Intel/Solaris. The RAM installed on these machines ranged from 32 MB to 512 MB and the CPU speed ranged from 50 to 446 MIPS. MW-based solvers can truly harness the power of a heterogeneous collection of computers. The overall efficiency obtained by this run was over 80%.

The case study demonstrates the power of metacomputing to solve highly complex problems, requiring tremendous computational power that is otherwise far beyond what practitioners can afford. This has been made possible and easy by using the MW framework. These results show that our goals of programmability, adaptability, reliability, and efficiency can be achieved by using MW.

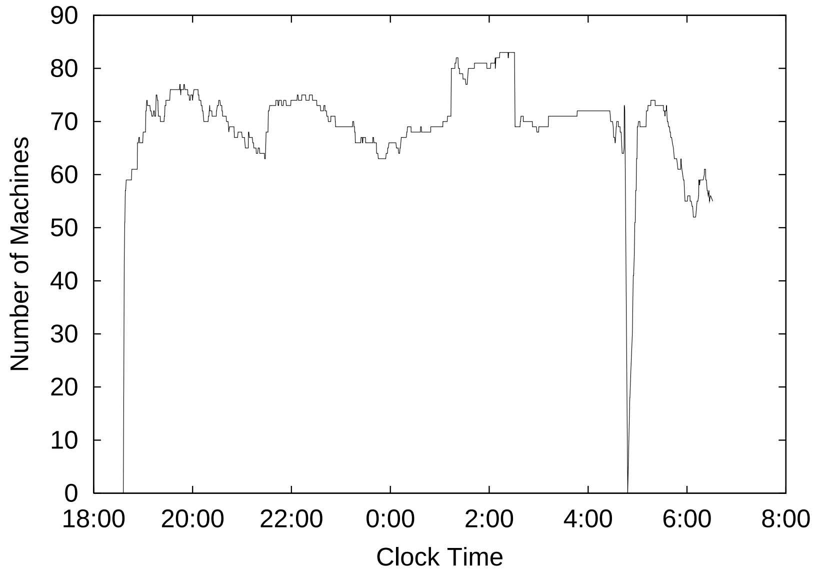

Fig. 6. Number of Workers Involved in the Computation

6. Conclusions and Future Directions

The concepts in this paper can help bridge the gap between existing metacomputing infrastructure and the solution of difficult real-world problems. Master-worker programs are simple for the application programmer, they can deftly handle an environment where processors are unreliable, and they can easily incorporate heterogeneous resources into one computation.

Future work includes addressing the limitations of the master-worker paradigm. Issues such as scalability can be addressed in a number of ways. Different paradigms, such as having a hierarchy of masters that control the computation, or the gossip-based distributed protocol described by Iamnitchi [Iam99] should be explored. MW will be tested on geographically diverse resource pools, where network latency issues enter the picture. In addition, MW will be extended to use different resource management software.

The MW code and documentation can be found at

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

Acknowledgments

The authors thank Miron Livny, Ian Foster, Steve Wright, Jorge Nocedal, and Gail Pieper for discussions and comments that led to an improved presentation.

References

[AB99] 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, University of Iowa, 1999. Available from `http://www.biz.uiowa.edu/faculty/anstreicher/qapqp.ps`.

[ASGH95] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations. In *Symposium on High Performance Distributed Computing, Virginia*, August 1995. Available from `http://www.dgs.monash.edu.au/~davida/papers/nimrod.ps.Z`.

[BKKW99] 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.

[BKR97] R. E. Burkard, S. E. Karisch, and F. Rendl. QAPLIB – A Quadratic Assignment Problem library. *Journal of Global Optimization*, 10:391–403, 1997.

[BRL99] 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.

[CP98] 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.

[FK97] 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`.

[FMD99] G. Fagg, K. Moore, and J. Dongarra. Scalable networked information processing environment (SNIPE). *International Journal on Future Generation Computer Systems*, 15:595–605, 1999.

[GFKH99] 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.

[Iam99] A. Iamnitchi. Branch and bound in grid environments. Master's thesis, Computer Science Department, University of Chicago, Chicago, IL, 1999.

[KRR88] 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.

[KW94] P. Kall and S. Wallace. *Stochastic Programming*. John Wiley and Sons, New York, 1994.

[LBRT97] 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`.

[LR99] 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.

[PL96] J. Pruyne and M. Livny. Interfacing Condor and PVM to harness the cycles of workstation clusters. *Journal on Future Generations of Computer Systems*, 12, 1996.

[RBD+97] A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, J. Simon, T. Römke, and F. Ramme. The MOL project: An open extensible metacomputer. In *Heterogenous Computing Workshop at IPPS*, 1997.

[WBK+99] 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`.