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

Jean-Pierre Goux†
Sanjeev Kulkarni‡
Jeff Linderoth§
Michael Yoder‡

March 10, 2000

Abstract

We describe MW – a software framework that allows users to quickly and easily parallelize scientific computations using the master-worker paradigm on the computational grid. 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.

*This work was supported in part by Grants No. CDA-9726385 and CDA-9623632 from the National Science Foundation.

†Department of Electrical and Computer Engineering, Northwestern University, and Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439, goux@mcs.anl.gov

‡Computer Sciences Department, University of Wisconsin - Madison, 1210 West Dayton Street, Madison, WI 53706, {sanjeevk,yoderme}@cs.wisc.edu

§Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439, linderot@mcs.anl.gov

1 Introduction

By its very definition, the Grid [11] 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 [10] [14] [17] [9] [3] [21]. 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 [4] 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 [22] 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 [20] 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 [19] 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. The focus is on parallel applications with weak synchronization and reasonably large grain size that can be fit into a master-worker paradigm without significant loss of efficiency. 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 [18] 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 [2] [15].

The paper is organized as follows. In Section 2, we introduce MW, and we describe the interfaces to both application software and Grid infrastructure software. Section 3 discusses additional features of MW that help developers build efficient and robust applications. Section 4 presents a case study where the Grid services are provided by Condor[17], 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 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.

2.1 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.

2.1.1 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.

2.1.2 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. The reason behind this decision is that MW requires that all information about the state of the computational resources be passed to it in the form of messages 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.

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.

2.1.3 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 [17]. As such, the MWDriver 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 [16] to write a series of shared files. Preliminary plans are being made for a port to the Globus software toolkit [10]. Table 1 highlights how the Grid service software provides (or could provide) the functionality required by MWRMComm.

There are advantages and disadvantages to having MW act as an "upper middleware" layer between application code and Grid service software code. The additional software layer 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 [8], where geographically distributed Condor pools are conceptually linked as one, and glide-in [7], 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 4.

2.2 MW Application Programming Interface

In a companion work [13], we argue that many scientific applications can be parallelized quite effectively for a Grid environment by using the master-worker paradigm. Our specific experience is with algorithms for solving numerical optimization problems and many of these algorithms share the following characteristics:

- Incremental Data Requirement – A potentially large amount of data must be passed to worker processes at initialization, but thereafter, messages are small "incremental" changes to the initial data.

- Weak Synchronization – The ability to execute a task does not depend on the completion of a large number of other tasks.

Services	Condor-PVM	Condor-Files	Globus
Communi-cation	Messages buffered and passed through PVM `pvm_pk()` in XDR format.	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

- Dynamic Grain Size – The computation can naturally be broken into portions of work of variable size.

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.

2.2.1 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 for the computation to begin work on.

- `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.

By carefully deciding on actions to take in the `act_on_completed_task()` method, the user can take advantage of a *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 3.

2.2.2 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 MWRMComm class.

2.2.3 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 Additional Functionality

In addition to the necessary services provided by the MWDriver and the MWRMComm implementation, users of the MW-framework benefit from a number of other useful features that are available through methods in the base MWDriver class.

3.1 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 4.

3.2 Normalized Performance Measurement

The heterogeneous and dynamic nature of the Grid makes application performance difficult to assess. Standard performance measures such as wall clock time and cumulative CPU time do not separate application code and computing platform performance. By normalizing the CPU time spent on a given task with the performance of the corresponding worker, the MWDriver 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.

If we make the following definitions:

- $\alpha(i)$ – Worker i performance normalization factor,
- $U(i)$ – Worker i uptime,
- $w(j)$ – Index of worker who solved task j,
- $t(j)$ – User time spent by worker $w(j)$ at solving j,
- $\mathcal{W}$ – Wall clock time,
- T – Cumulative workers CPU time.

We can then define the following statistics:

- $\mathcal{T}$ – Normalized cumulative time :

$$\mathcal{T} = \sum_{j \in \mathcal{J}} \alpha(w(j)) * t(j)$$

- $\mathcal{P}$ – Equivalent Pool Performance :

$$\mathcal{P} = \frac{\sum_{i \in \mathcal{I}} \alpha(i) * U(i)}{\sum_{i \in \mathcal{I}} U(i)}$$

- $\mathcal{N}$ – Average number of workers :

$$\mathcal{N} = \frac{\sum_{i \in \mathcal{I}} U(i)}{\mathcal{W}}$$

- η – Parallel efficiency :

$$\eta = \frac{\sum_{j \in \mathcal{J}} t(j)}{\sum_{i \in \mathcal{I}} U(i)}$$

Table 3.2 shows the variations of performance statistics between runs of a Grid-enabled application (presented in Section 4). 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.

	Mean	Std. Dev.	Min	Max
$\mathcal{W}$	915	1019	489	1780
T	22182	27900	8844	37671
$\mathcal{T}$	5864	341	5739	6054
$\mathcal{P}$	7.27	7.16	3.2	12.4
$\mathcal{N}$	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.

As expected, the statistics show the large variance of $\mathcal{W}$ and T. However, there is little variance of $\mathcal{T}$, which can therefore be used to do comparisons between runs and assess the application performance.

3.3 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 are not data-intensive, so use of advanced services such as the Network Weather Service **??** to improve scheduling has not been warranted.

4 Application to Combinatorial Optimization

MW has been used in the MetaNEOS project [18] to implement several grid enabled parallel optimization solvers [6] [12] [15]. One solver has been specialized to solve the quadratic assignment problem (QAP) [5]. 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 $n > 30$ have not even been attempted yet. By embedding a new relaxation technique [1] 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 to provable optimality [2].

In order to use the computational resources with maximal 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. Our computational pool was composed of

- University of Wisconsin Condor pool workstations,
- a University of Wisconsin dedicated Linux cluster,
- a University of New Mexico "flocked" Condor pool,
- the National Institute for Nuclear Physics, Bologna, Italy, "flocked" Condor pool, and
- the Argonne National Laboratory SGI/Origin2000 acquired via Globus through the glide-in mechanism.

The pool is depicted in Figure 1.

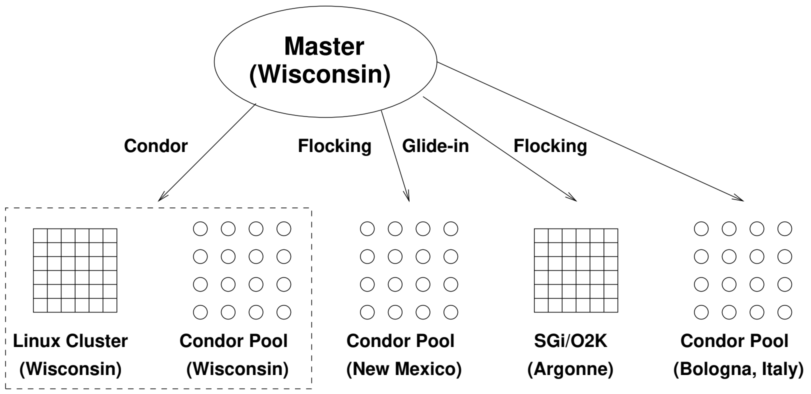

Figure 1: The Computational Pool.

Figure 2 depicts the evolution of the number of machines of each type during our run. At 11:30AM a glide-in request was made for 32 SGI processors on Argonne's O2K for a period of 12 hours. 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 3.1.

Over the course of the computation, 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.

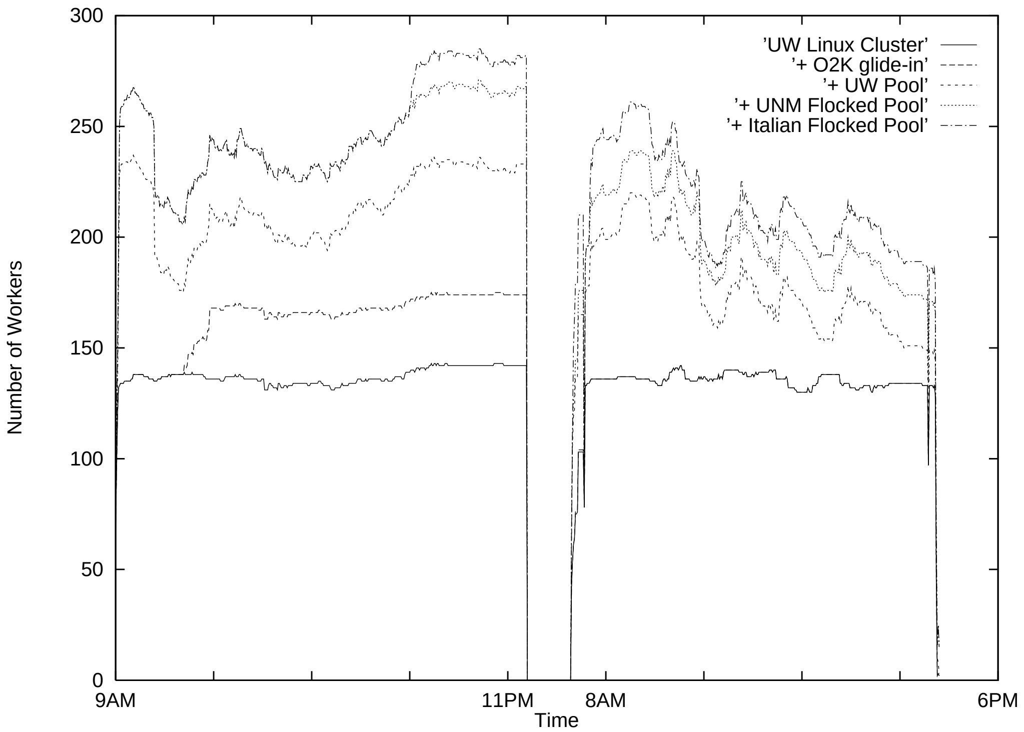

Figure 2: Number of Workers

5 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.

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.

References

[1] 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`.

[2] 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`.

[3] 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.

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

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

[6] Q. Chen, M. Ferris, and J. Linderoth. Fatcop 2.0: Advanced features in an opportunistic mixed integer programming solver. Data Mining Institute 99-11, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1999. Available from `http://www.mcs.anl.gov/metaneos/fatcop2.ps`.

[7] 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`.

[8] 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.

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

[10] 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`.

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

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

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

[14] 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.

[15] J. Linderoth and S. Wright. A cutting plane code for stochastic programming on metacomputers. Invited Presentation at the INFORMS National Meeting, November 1999.

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

[17] 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`.

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

[19] 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`.

[20] 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.

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

[22] 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`.