Adaptive Scheduling for Task Farming with Grid Middleware

Henri Casanova [*] MyungHo Kim [†] James S. Plank [‡]
Jack J. Dongarra[‡ §]

June 16, 1999

Abstract

Scheduling in metacomputing environments is an active field of research as the vision of a Computational Grid becomes more concrete. An important class of Grid applications are long-running parallel computations with large numbers of somewhat independent tasks (Monte-Carlo simulations, parameter-space searches, etc.). A number of Grid middleware projects are available to implement such applications but scheduling strategies are still open research issues. This is mainly due to the diversity of both Grid resource types and of their availability patterns. The purpose of this work is to develop and validate a general adaptive scheduling algorithm for task farming applications along with a user interface that makes the algorithm accessible to domain scientists. Our algorithm is general in that it is not tailored to a particular Grid middleware and that it requires very few assumptions concerning the nature of the resources. Our first testbed is NetSolve as it allows quick and easy development of the algorithm by isolating the developer from issues such as process control, I/O, remote software access, or fault-tolerance.

Keywords
Farming, Master-Slave Parallelism, Scheduling, Metacomputing, Grid Computing.

[*]Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093-0114, USA
[†]School of Computing, SoongSil University, Seoul, 156-743 Korea
[‡]Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, USA
[§]Mathematical Science Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

1 Introduction

The concept of a *Computational Grid* envisioned in [1] has emerged to capture the vision of a network computing system that provides broad access not only to massive information resources, but to massive computational resources as well. Such computational grids will use high-performance network technology to connect hardware, software, instruments, databases, and people into a seamless web that supports a new generation of computation-rich problem solving environments for scientists and engineers. Grid resources will be ubiquitous thereby justifying the analogy to the Power Grid.

Those features have generated interest among many domain scientists and new classes of applications arise as being potentially *griddable*. Grid resources and their access policies are inherently very diverse, ranging from directly accessible single workstations to clusters of workstations managed by Condor [2], or MPP systems with batch queuing management. Furthermore, the availability of these resources changes dynamically in a way that is close to unpredictable. Lastly, predicting networking behavior on the grid is an active but still open research area. Scheduling applications in such a chaotic environment according the end-users' need for fast response-time is not an easy task. The concept of a universal scheduling paradigm for any application at the current time is intractable and the current trend in the scheduling research community is to focus on schedulers for broad *classes* of applications. Given the characteristics of the Grid, it is not surprising that even applications with extremely simple structures raise many challenges in terms of scheduling.

In this paper we address applications that have simple task-parallel structures (master-slave) but require large number of computational resources. We call such applications *task farming applications* according to the terminology introduced in [3]. Examples of such applications include Monte-Carlo simulations and parameter-space searches. Our goal is not only to design a scheduling algorithm but also to provide a convenient user interface that can be used by domain scientists that have no knowledge about the Grid structure.

Section 2 shows how some of the challenges can be addressed by using a class of grid middleware projects as underlying operating environments, while others need to be addressed specifically with adaptive scheduling algorithms. Section 3 gives an overview of related research work and highlights the original elements of this work. Section 4 contains a brief overview of NetSolve, the grid middleware that we used as a testbed. Sections 5 and 6 describe the implementation of the task farming interface and the implementation of the adaptive scheduling algorithm underneath that interface. Section 7 presents experimental results to validate the scheduling strategy. Section 8 concludes with future research and software design directions.

2 Motivation and Challenges for Farming

Our intent is to design and build an easily accessible computational framework for task farming applications. An obvious difficulty then is to isolate the users from details such as I/O, process control, connections to remote hosts, fault-tolerance, etc. Fortunately an emerging class of Grid middleware projects provides the necessary tools and features to transparently handle most of the low-level issues on behalf of the user. We call these middleware projects *functional metacomputing environments*, the user's interface to the Grid is a functional remote procedure call (i.e a call without side effects). The middleware intercepts the procedure call and treats it as a request for service. The procedure call arguments are wrapped up and sent to the Grid resources that are currently best able to service the request, and when the request has been serviced, the results are shipped back to the user, and his or her procedure call returns. The middleware is responsible for the details of managing the service on the Grid – resource selection and allocation, data movement, I/O, and fault-tolerance.

There are two main functional metacomputing environments available today. These are NetSolve (see Section 4) and Ninf [4]. Building our framework on top of such architectures allows us to focus on meaningful issues like the scheduling algorithm rather than building a whole system from the ground up. Our choice of NetSolve as a testbed is motivated by the authors' experience with that system. Section 5 describes our first attempts at an API.

Of course, the main challenge is scheduling. Indeed, for long running farming applications it is to be expected that the availability and workload of resources within the server pool will change dynamically. We must therefore design and validate an adaptive scheduling algorithm (see Section 6). Furthermore, that algorithm should be general and applicable not only for a large class of applications but also for any operating environment. The algorithm is therefore designed to be portable to other metacomputing environments [4, 5, 6, 2, 7].

3 Related Work

Nimrod [7] is targeted to computational applications based on the "exploration of a range of parameterized scenarios" which is similar to our definition of task farming. The user interfaces in Nimrod are at the moment more evolved than the API described in Section 5. However, we believe that our API will be a building block for high-level interfaces (see Section 8). The current version of Nimrod (or Clustor, the commercial version available from [8]) does not use any metacomputing infrastructure project whereas our task farming framework is built on top of grid middleware. However, a recent effort, Nimrod/G [9], plans to build Nimrod directly on top of Globus [5]. We believe that a project like NetSolve (or Ninf) is a better choice for this research work. First, NetSolve is freely available. Second, NetSolve provides a very simple interface, letting us focus on scheduling algorithms rather than Grid infrastructure details. Third, NetSolve can and probably will be implemented on top of most Globus services and will then leverage the Grid infrastructure without modifications of our scheduling algorithms. Another distinction between this work and Nimrod is that the latter does not contain adaptive algorithms for scheduling like the one described in Section 6. In fact, it is not inconceivable that the algorithms eventually produced by this work could be incorporated seamlessly into Nimrod.

Calypso [10] is a programming environment for a loose collection of distributed resources. It is based on C++ and shared memory, but exploits task-based parallelism of relatively independent jobs. It has an eager scheduling algorithm and like the functional metacomputing environments described in this paper, uses the idempotence of the tasks to enable a replication-based fault-tolerance.

A system implemented on top of the Helios OS that allows users to program master-slave programs using a "Farming" API is described in[3, 11]. Like in Calypso, the idempotence of tasks is used to achieve fault-tolerance. They do not focus on scheduling.

The AppLeS project [12, 13] develops metacomputing scheduling agents for broad classes of computational applications. Part of the effort targets scheduling master-slave applications [14] (task farming applications with our terminology). A collaboration between the NetSolve and the AppLeS team has been initiated and and integration of AppLeS technology, the Network Weather Service (NWS) [15] NetSolve-like systems, and the results in this document is underway.

As mentioned earlier, numerous ongoing projects are trying to establish the foundations of the *Computational Grid* envisioned in [1]. Ninf [4] is similar to NetSolve in that it is targeted to domain scientists. Like NetSolve, Ninf provides simple computational services and and the development teams are collaborating to make the two system interoperate and standardize the basic protocols. At a lower-level are Globus [5] and Legion [6] which aim at providing basic infrastructure for the grid. Condor [2, 16] defines and implements a powerful model for grid components by allowing the idle cycles of networks of workstation to be harvested for the benefit of grid users without penalizing local users.

4 Brief Overview of NetSolve

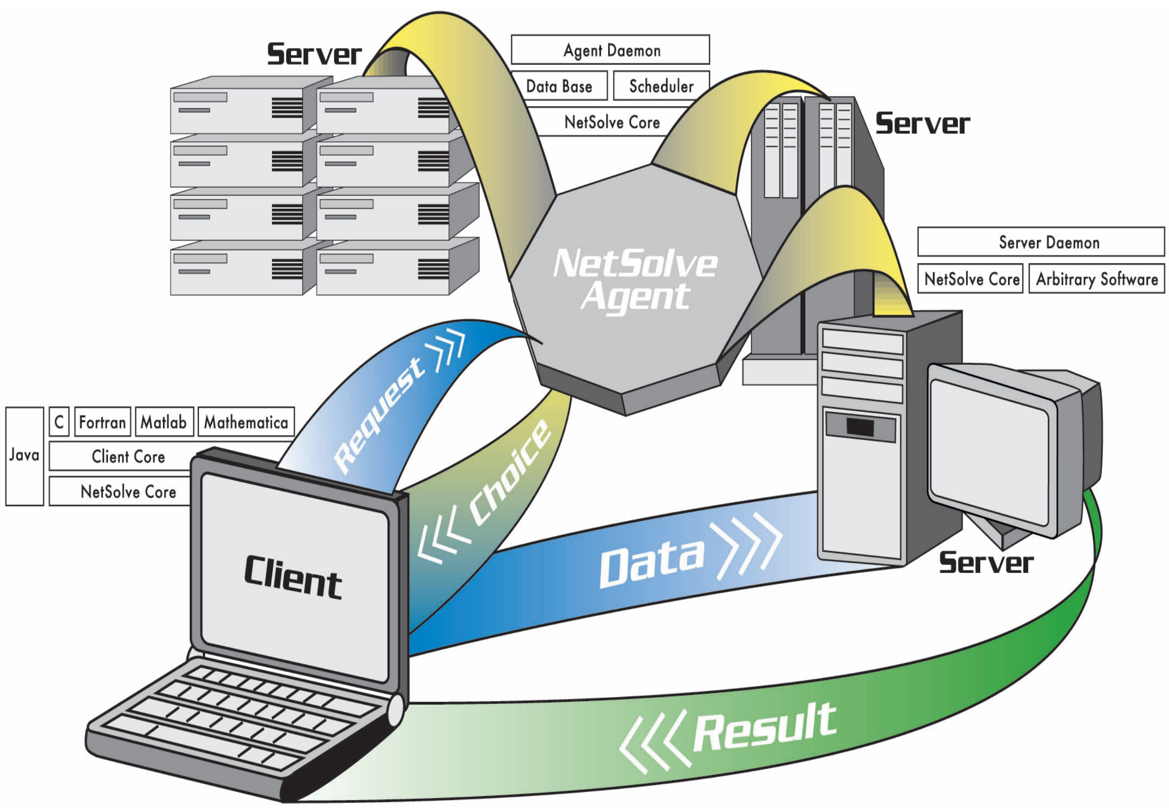

Figure 1: The NetSolve System

The NetSolve project is under development at the University of Tennessee and the Oak Ridge National Laboratory. Its original goal is to alleviate the difficulties that domain scientists usually encounter when trying to locate/install/use numerical software, especially on multiple platforms. With NetSolve, the user does not need to be concerned with the location/type of the hardware resources being used or with the software installation. Furthermore, NetSolve provides transparent fault-tolerance mechanisms and implements scheduling algorithms to minimize overall response time. As seen on Figure 1 NetSolve has a three-tiered design in that a *client* consults an *agent* prior to sending requests to a *server*. Let us give basic concepts about those three components as well as information about the current status of the project.

The NetSolve Server: A NetSolve server can be started on any hardware resource (single workstation, cluster of workstations, MPP). It can then provide access to arbitrary software installed on that resource (NetSolve provides mechanisms to integrate any software component into a server so that it may become available to NetSolve clients [17]).

The NetSolve Agent: The NetSolve agent is the key to the computation-resource mapping decisions as it maintains a database about the statuses and capabilities of servers. It uses that database to make scheduling decisions for incoming user requests. The agent is also the primary participant in the fault-tolerance mechanisms. Note that there can be multiple instances of the NetSolve agent to manage a confederation of servers.

The NetSolve Client: The user can submit (possibly simultaneous) requests to the system and retrieve results with one of the provided interfaces (C, Fortran, Matlab [18], Mathematica [19], Java APIs or Java GUI).

Current Status of NetSolve: At this time, a pre-version of NetSolve 1.2, containing full-fledge software for all UNIX flavors, Win32 C, and Matlab APIs, can be downloaded from the homepage at:

`http://www.cs.utk.edu/netsolve`

The NetSolve Users' Guide [20] contains general purpose information and examples. Details about the NetSolve agent can be found in [21]. Recent developments and applications of NetSolve are described in [22]. Lastly, technical details about the current NetSolve implementation are to be found in [23].

5 Task Farming API

5.1 Basics

In this work, we assume that a functional metacomputing environment is available (see Section 2). That environment provides an API that contains two functions: (i)`submit()` to send a request asynchronously for computation and (ii)`poll()` to poll asynchronously for the completion of a request. Polling returns immediately with the status of the request. If the computation is complete, the result is returned as well. The NetSolve and Ninf APIs satisfy these requirements. In addition, the environment provides access to pre-installed software and hardware resources. The user just provides input data and a way to identify which software should be used to process that data. Again, both NetSolve and Ninf comply.

A farming job is one composed of a large number of independent requests that may be serviced simultaneously. This is sometimes referred to as the "bag-of-tasks" model [24, 25]. Farming jobs fall into the class of "embarrassingly parallel" programs, for which it is very clear how to partition the jobs for parallel programming environments. Many important classes of problems, such as Monte-Carlo simulations (e.g. [26]) and parameter-space searches (e.g. [7]) fall into this category.

Without a farming API, the user is responsible for managing the requests himself. One possibility would be to submit all the desired requests at once and let the system schedule them. However, we have seen that scheduling on the grid is a challenging issue and as a result, the available grid middleware projects implement only minimal scheduling capabilities that do not optimize even this simple class of parallel programs. A second possibility is for the user to manually manage a ready queue by having at most n requests submitted to the system at any point in time. This solution seems more reasonable; however the optimal value of n depends on Grid resource availability, which is beyond the user's control and is dynamic.

5.2 API

It is difficult to design an API that is both convenient for the end-user and sophisticated enough to handle many real applications. Our farming API contains one function, `farm()`, with which the user specifies all data for all the computation tasks. The main idea is to replace multiple calls to `submit()` by one call to `farm()` whose arguments are *lists* of arguments to `submit()`. In this first implementation, we assume that arguments to `submit()` are either integers or pointers (which is consistent with the NetSolve specification). Extending the call to support other argument types would be trivial. The first argument to `farm()` specifies the number of request by declaring an induction variable and defining its range. The syntax if `"i=%d,%d"` (see example below). The second argument is the identifier for the computational functionality in the metacomputing environment (a string with Ninf and NetSolve). Then follow a (variable) number of argument lists. Our implementation provides three functions that need to be called to generate such lists:(i)`expr()` allows an argument to computation i to be an integer computed as an arithmetic expression containing i; (ii)`int_array()` allows an integer argument to computation i to be an element of an integer array indexed by the value of an arithmetic expression containing i; (iii)`ptr_array()` is similar to `int_array()` but handles pointer arguments. Arithmetic expressions are specified with Bourne Shell syntax (accessing the value of i with '`$i`').

Let us show an example assuming that the underlying metacomputing environment provides a computational function called "foo". The code:

```
double x[10],y[30],z[10];
```

```
submit("foo",2,x,10);
submit("foo",4,y,30);
submit("foo",6,z,10);
```

makes three requests to run the "foo" software with the sets of arguments: (2,x,10), (4,y,30) and (6,z,10). Note that x, y and z will hold the results of the NetSolve call. With farming these calls are replaced by

needs to be replaced by

```
void *ptrs[3];
int  *ints[3];

ptrs[0] = x;  ptrs[1] = y;  ptrs[2] = z;
ints[0] = 10; ints[1] = 30; ints[2] = 10;

farm("i=0,2","foo",expr("2*($i+1)"),ptr_array(ptrs,"$i"),
                        int_array(ints,"$i"));
```

We expect to use this API as a basis for more evolved interfaces (e.g. graphical or Shell-based). So far, we have used the API directly to implement basic example computations (2D block-cyclic matrix-multiply, Mandelbrot set computation) and to build a Shell interface to MCell (see Section 7). Section 8 describes how we plan to generalize this work to automatically generate high-level interfaces.

6 Scheduling Strategy

6.1 The Scheduling Algorithm

The main idea behind the scheduling algorithm has already been presented in Section 5.1: managing a ready queue. We mentioned that the user had no elements on which to base the choice for n, the size of the ready queue. Our farming algorithm manages a ready queue and adapts to the underlying metacomputing environment by modifying the value of n dynamically according to constant computation throughput measurement. The algorithm really sees the environment as an opaque entity that gives varying responses (request response times) to repeated occurrence of the same event (the sending of a request).

Let us go through the algorithm shown in Figure 2. First, the algorithm chooses the initial value of n. That choice can be arbitrary but it may benefit from additional information provided by the underlying metacomputing environment. NetSolve provides a way to query the agent about the number of available servers for a given computation and that number is the initial guess for n in this first implementation. Second, the algorithm sets the *scheduling factor* α which takes values in $[0, 1)$ and determines the behavior of the algorithm. Indeed the value of n may be changed only when more than n tasks completed during one iteration of the outermost while loop. A value of $\alpha = 1$ causes the algorithm to be extremely conservative (only when all n requests are completed instantly may the value of n be changed). The smaller α the more often will the algorithm try to modify n. The algorithm keeps a running

```
n = initial guess on the queue size;
α = scheduling factor;
δ = 1;
while (tasks remaining) {
    while (number of pending tasks < n) {
        submit();
    }
    foreach (pending task) {
        poll();
    }
    if (n− number of pending tasks ≥ n × α) {
        if (average task response time has improved) {
            n = n + δ;
            δ = δ + 1;
        }
        else {
            n = n − δ;
            δ = 1;
        }
    }
}
```

Figure 2: Adaptive scheduling algorithm

history of the average request response times for all requests in the queue. That history is used to detect improvements or deterioration in performance and modify the value of n accordingly.

This algorithm is rather straightforward at the moment but it will undoubtedly be improved after more experiments have been conducted. However, early experimental results shown in Section 7 are encouraging.

6.2 Current Implementation

In our testbed implementation of farming for NetSolve, we implement `farm()` as an additional layer on top of the traditional NetSolve API, exactly as detailed in Section 5.2. A similar implementation would be valid for a system like Ninf. In other metacomputing environments, placing the scheduling algorithm within the client library might not be feasible, in which case the algorithm needs to be implemented in other parts of the system (central scheduler, client proxy, etc..). However, the algorithm is designed to rest on top of the metacomputing system, rather than merged with the internals of system.

6.3 Possible Extensions

The NetSolve farming interface is very general and we believe that it can serve as a low-level building-block for deploying various classes of applications. However, this generality leads to shortcomings. The embedded scheduler cannot take advantage of application-specific features, such as exploitable data patterns. Real applications are likely to manipulate very large amounts of data and it may be possible for the scheduler to make decisions based on I/O requirements. For instance, one can imagine that a subset of the tasks to farm make uses of one or more constant input data. This is a frequent situation in MCell (see Section 7.1) for example. Such input data could then be *shared* (via files for instance) by multiple resources as opposed to being replicated across all the resources. Another possibility would be for the farming application to contain simple data dependences between tasks. In that case, our framework could detect those dependences and schedule the computations accordingly. Another shortcomings of the farming interface that is a direct cause of its generality is that the call to `farm()` is completely atomic. This is an advantage from the point of view of ease-of-use, but it prevents such things as visualization of results as they become available for instance. Once again, such a feature would be desirable for MCell. Section 8 lays ground for research in these directions and work is under way in the context of MCell.

7 Preliminary Experimental Results

7.1 MCell

MCell [26, 27] is a general Monte Carlo simulator of cellular microphysiology. MCell uses Monte Carlo diffusion and chemical reaction algorithms in 3D to simulate the complex biochemical interactions of molecules inside and outside of living cells. MCell is a collaborative effort between the Terry Sejnowski lab at the Salk Institute, and the Miriam Salpeter lab

Scheduling	Time	Resource Availability	Relative performance
adaptive	3982 s	64 %	100 %
$n = 25$	4518 s	62 %	85 %
$n = 5$	10214 s	63 %	38 %

Table 1: Preliminary experimental results

at Cornell University. Like any Monte Carlo simulation, MCell must run large numbers of identical, independent simulations for different values of its random number generator seed. It therefore qualifies as a task farming application and was our first motivation to develop a farming API along with a scheduling algorithm.

As mentioned earlier, we developed for MCell a Shell-based interface on top of the C farming API. This interface takes as input a user-written *script* and automatically generates the call to `farm()`. The script is very intuitive as it follows the MCell command-line syntax by just adding the possibility for *ranges* of values as opposed to fixed values. For instance, instead of calling MCell as:

```
mcell foo1 1
mcell foo1 2
....
mcell foo1 100
```

it is possible to call MCell as

```
mcell foo1 [1-100]
```

which is simpler, uses Grid computational resources from NetSolve and ensures good scheduling with the use of the algorithm described in Section 6.

7.2 Results

The results presented in this section were obtained by using a NetSolve system spanning 5 to 25 servers on a network of Sun workstations (Sparc ULTRA 1) interconnected via 100Mb Ethernet. The farming application uses MCell to compute the shape of the parameter space which describes the possible modes of operation for the process of synaptic transmission at the vertebrate neuromuscular junction. Since MCell's results include the true stochastic noise in the system the signal must be averaged at each parameter space point. This is done by running each point 10 times with 10 different values of the random number generator seed. In this example, three separate 3-D parameter spaces are sampled, each parameter-space is of dimension $3 \times 3 \times 3$. The number of tasks to farm is therefore $3 \times 3 \times 3 \times 3 \times 10 = 810$ and each task generates 10 output files.

These preliminary experiments were run on a dedicated network. However, we simulated a dynamically changing resource pool by linearly increasing and decreasing the number of available NetSolve computational servers. Results are shown in Table 1 for our adaptive scheduling, a fixed queue size of $n = 25$ and a fixed queue size of $n = 5$.

The resource availability measures the fraction of servers available during one run of the experiment. As this number changes throughout time, the availability is defined as the sum of the number of servers available servers over all time steps (10 seconds). We compare scheduling strategies by measuring *relative performance* which we define as a ratio of *adjusted elapsed times*, taking the adaptive scheduling as a reference. Adjusted elapsed times are computed by assuming a 100% availability and scaling the real elapsed times accordingly. One can see that the adaptive strategy performs 15% better than the strategy with $n = 25$. Of course, the strategy with $n = 5$ performs very poorly since it does not take advantage of all the available resources.

These first results are encouraging but not as satisfactory as expected. This is due to the implementation of NetSolve and the way the experiment was set-up. Indeed, NetSolve computational tasks are not interrupted when a NetSolve server is terminated. Terminating a server only means that no further requests will be answered but that pending requests are allowed to terminate. Thus, this experiment does not reflect the worst case scenario of machines being shutdown causing all processes to terminate. We expect our adaptive strategy to perform even better in an environment where tasks are terminated prematurely and need to be restarted from scratch on remaining available resources. Due to time constraints, this article does not contain results to corroborate this assumption, but experiments are underway.

8 Conclusion and Future Work

In this paper we have motivated the need for schedulers tailored to broad classes of applications running on the Computational Grid. The extreme diversity of Grid resource types, availabilities and access policies makes the design of schedulers a difficult task. Our approach is to build on existing and available metacomputing environments to access the Grid as easily as possible and to implement a interface and scheduling algorithm for task farming applications. An adaptive scheduling algorithm was described in Section 6. That algorithm is independent from the internal details of the Grid and of the metacomputing environment of choice. We chose NetSolve as a testbed for early experiments with the MCell application. The effectiveness of our scheduler is validated by preliminary experimental results in Section 7. Thanks to our framework for farming, a domain scientist can easily submit large computations to the Grid in a convenient manner and have an efficient adaptive scheduler manage execution on their behalf.

There are many ways in which this work can be further extended. We already mentioned in Section 6.3 that it is possible to use the farming API to detect data dependencies or shared input data between requests. The adaptive scheduling algorithm could be augmented to take into account such patterns. A possibility is for the farming interface to take additional arguments that describe domain-specific features and that may activate more sophisticated scheduling strategies if any. A first approach would be to consider only input or output coming from files (which is applicable to MCell and other applications) and partition the request space such as to minimize the number of file transfers and copies. This will require that the underlying metacomputing environment provide feature to describe such dependences. Work is being done in synergy with the NetSolve project to take into account data locality and the farming interface will undoubtedly take advantage of these developments [28]. This will be fertile ground for scheduling and data logistic research. The scheduling algorithm can also be modified to incorporate more sophisticated techniques. For instance, if the metacomputing environment provides and API to access more details about the status of available resources, it might be the case that n, the size of the ready queue, can be tuned effectively. The danger however is to lose portability as the requirements for the metacomputing environment (see Section 5.1) would be more stringent. Experiments will be conducted in order to investigate whether such requirements can be used to significantly improve scheduling.

The farming API can be enhanced so that certain tasks may be performed upon submitting each request and receiving each result. For instance, the user may want to visualize the data as it is coming back as opposed to have to wait for completion of all the requests. This is not possible at the moment as the call to `farm()` is atomic and does not provide control over each individual request. A possibility would be to pass pointers to user defined functions for `farm()` and execute them for events of interest (e.g. visualization for each reception of a result). Such functions could take arbitrary arguments for the sake of versatility. Some of the available metacomputing environments provide attractive interactive interface to which a farming call could be contributed. Examples include Matlab (NetSolve) and Mathematica (NetSolve,Ninf). In order to make our task farming framework easily accessible to a growing number of domain scientists, we need to develop ways to use the C farming API as a basis for more usable high-level interfaces. Steps in that direction have already been taken with the Shell-interface for MCell (see Section 7.1). It would be rather straightforward to design or use an existing specification language to describe specific farming applications and automatically generate custom Shell-based of graphical interfaces like the ones in [7].

9 Acknowledgments

This material is based upon work supported by the National Science Foundation under grant CCR-9703390, by NASA/NCSA Project Number 790NAS-1085A, under Subaward Agreement #790 and by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615.

References

[1] Ian Foster and Carl Kesselman, editors. *The Grid, Blueprint for a New computing Infrastructure.* Morgan Kaufmann Publishers, Inc., 1998.

[2] M. Litzkow, M. Livny, and M.W. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. of the 8th International Conference of Distributed Computing Systems*, pages 104–111. Department of Computer Science, University of Winsconsin, Madison, June 1988.

[3] L. Silva, B. Veer, and J. Silva. How to Get a Fault-Tolerant Farm. In *World Transputer Congress*, pages 923–938, Sep. 2993.

[4] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. Ninf : Network based Information Library for Globally High Performance Computing. In *Proc. of Parallel Object-Oriented Methods and Applications (POOMA), Santa Fe*, 1996.

[5] I. Foster and K Kesselman. Globus: A Metacomputing Infrastructure Toolkit. In *Proc. Workshop on Environments and Tools*. SIAM, to appear.

[6] A. Grimshaw, W. Wulf, J. French, A. Weaver, and P. Jr. Reynolds. A Synopsis of the Legion Project. Technical Report CS-94-20, Department of Computer Science, University of Virginia, 1994.

[7] D. Abramson, I. Foster, J. Giddy, A. Lewis, R. Sosic, and R. Sutherst. The Nimrod Computational Workbench: A Case Study in Desktop Metacomputing. In *Proceedings of the 20th Autralasian Computer Science Conference*, Feb. 1997.

[8] `http://www.activetools.com`.

[9] D. Abramson and J. Giddy. Scheduling Large Parametric Modelling Experiments on a Distributed Meta-computer. In *PCW'97*, Sep. 1997.

[10] A. Baratloo, P. Dasgupta, and Z. Kedem. Calypso: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms. In *4th IEEE International Symposium on High Performance Distributed Computing*, Aug. 1995.

[11] L. M. Silva, J. G. Silva, S. Chapple, and L. Clarke. Portable checkpointing and recovery. In *Proceedings of the HPDC-4, High-Performance Distributed Computing*, pages 188–195, Washington, DC, August 1995.

[12] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. In *Proc. of Supercomputing'96, Pittsburgh*, 1996.

[13] F. Berman and R. Wolski. The AppLeS Project: A Status Report. In *Proc. of the 8th NEC Research Symposium, Berlin, Germany*, 1997.

[14] F. Berman, R. Wolski, and G. Shao. Performance Effects of Scheduling Strategies for Master/Slave Distributed Applications. Technical Report TR-CS98-598, U. C., San Diego, 1998.

[15] R. Wolski. Dynamically forecasting network performance using the network weather service. Technical Report TR-CS96-494, U.C. San Diego, October 1996.

[16] M. Litzkow and M. Livny. Experience with the Condor Distributed Batch System. In *Proc. of IEEE Workshop on Experimental Distributed Systems*. Department of Computer Science, University of Winsconsin, Madison, 1990.

[17] H. Casanova and J. Dongarra. Providing Uniform Dynamic Access to Numerical Software. In M. Heath, A. Ranade, and R. Schrieber, editors, *IMA Volumes in Mathematics and its Applications, Algorithms for Parallel Processing*, volume 105, pages 345–355. Springer-Verlag, 1998.

[18] The Math Works Inc. *MATLAB Reference Guide*. The Math Works Inc., 1992.

[19] S. Wolfram. *The Mathematica Book, Third Edition*. Wolfram Median, Inc. and Cambridge University Press, 1996.

[20] H. Casanova, J. Dongarra, and K. Seymour. Client User's Guide to Netsolve. Technical Report CS-96-343, Department of Computer Science, University of Tennessee, 1996.

[21] H Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 1997.

[22] H. Casanova and J. Dongarra. NetSolve's Network Enabled Server: Examples and Applications. *IEEE Computational Science & Engineering*, 5(3):57–67, September 1998.

[23] H. Casanova and J. Dongarra. NetSolve version 1.2: Design and Implementation. Technical Report to appear, Department of Computer Science, University of Tennessee, 1998.

[24] D. E. Bakken and R. D. Schilchting. Supporting fault-tolerant parallel programming in linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302, March 1995.

[25] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with piranha. In *International Conference on Supercomputing*, pages 417–427, Washington, D.C., June 1992. ACM.

[26] J.R. Stiles, T.M. Bartol, E.E. Salpeter, , and M.M. Salpeter. Monte Carlo simulation of neuromuscular transmitter release using MCell, a general simulator of cellular physiological processes. *Computational Neuroscience*, pages 279–284, 1998.

[27] J.R. Stiles, D. Van Helden, T.M. Bartol, E.E. Salpeter, , and M.M. Salpeter. Miniature endplate current rise times $<$100 microseconds from improved dual recordings can be modeled with passive acetylcholine diffusion form a synaptic vesicle. In *Proc. Natl. Acad. Sci. U.S.A.*, volume 93, pages 5745–5752, 1996.

[28] M. Beck, J. Plank, T. Moore, and W. Elwasif. Why IBP Now. *The International Journal of Supercomputer Applications and High Performance Computing*, to appear.