

Scheduling Architecture and Algorithms within the ICENI Grid Middleware

Laurie Young Stephen McGough Steven Newhouse John Darlington

London e-Science Centre, Imperial College London, South Kensington Campus, London SW7 2AZ, UK
Email: lesc-staff@doc.ic.ac.uk

Abstract

The ICENI Grid Middleware system enables the use of component based applications in a distributed heterogeneous environment and is implemented as a layered service oriented architecture. The scheduling of component applications across this environment takes place through the ICENI Scheduling Framework which allows ‘pluggable’ scheduling algorithms and platform specific launchers. The architecture of the Scheduling Framework is described, which decouples the scheduling algorithm from both the application and the architecture. Four scheduling algorithms (game theory, simulated annealing, random and best of n random) are presented and compared under the criteria of time and cost optimisation. We show that the simulated annealing scheduler produces more optimal results in the case of both time and cost optimisation.

1 Introduction

ICENI (Imperial College e-Science Network Infrastructure) is a Grid Middleware system, consisting of a service oriented architecture and an augmented component programming model being developed at the London e-Science Centre [7, 8, 9, 10, 13]. Within ICENI an application is defined to be a collection of interconnected components. Figure 1 shows a simple application in which the first component provides data to two second level components which feed their data into a final component.

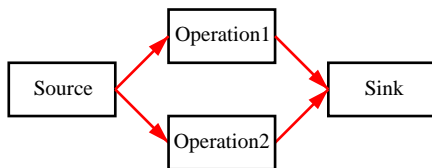


Figure 1: A simple ICENI Application

The role of the Scheduler within ICENI is to determine the ‘best’ place to run each component. Each component can potentially run on any Grid enabled resource limited only by the availability of software components and computational resources. Resource owners may impose a financial *cost* for using their resources which may affect placement decisions. Choosing where to run each component is a complex task. Placing components on the fastest single resource may saturate that resource and cost too much. Placing each component

on different distributed resources is likely to incur large performance penalties in communication between components.

In this paper we model a number of grid applications as a collection of components connected as a DAG (*Directed Acyclic Graph*). A number of DAGs were generated, these had varying depths, representing the sequential dependencies of the component application, and varying widths, representing the number of components that could potentially run in parallel. The application shown in Figure 1 shows an application with four components, a depth of three and a width of two.

In Section 2 we describe the ICENI middleware and its Scheduling Framework. Section 3 describes the four scheduling algorithms, each of which displays different characteristics. The performance of these algorithms, are examined in Section 4. A range of application and Grid sizes are considered, along with the performance of both the network and the resources. We look at related work in Section 5 and our conclusions are presented in Section 6.

2 The ICENI Architecture

One of the main research strategies within ICENI is the exploration of the role meta-data will play in the efficient and correct use of the Grid. This is explored within the modular framework for

scheduling and launching applications onto the Grid, which allows different scheduling algorithms to be examined (see Figure 2).

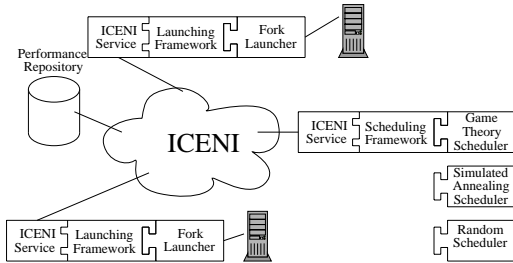


Figure 2: ICENI Scheduling and Launching Framework

2.1 Overview

ICENI has an implementation independent service discovery and interaction layer which can currently be provided by Jini[1] (with services exposed through OGSA), JXTA[2] and OGSA[6]. The Scheduling Framework provides an abstraction between the ICENI environment and the Scheduler, allowing the developer to concentrate on the scheduling algorithm rather than the specifics of the ICENI architecture. The Scheduling Framework provides the Scheduler on demand, with any relevant meta-data that it may require by orchestrating the retrieval of that information from the appropriate sources.

2.2 Scheduling

The pluggable Scheduler attached to the Scheduling Framework will not have exclusive control of the resources made available to it and is therefore considered to be an ‘Application Scheduler’ as defined by [3]. The ICENI architecture allows for multiple Schedulers (each within their own Scheduling Framework) to be made available to the user. The user is able to select the most appropriate Scheduler for the work to be undertaken. Resources may also be accessible to users through other (non-ICENI) systems as is the case in most Grid environments.

2.3 Launching

Resources are exposed through an extensible Launching Framework by platform specific Launchers. A Launcher may represent one or more resources provided that all resources use the same launching mechanism. Current Launchers include Fork, Condor[16], Globus[17] and Sun Grid Engine [14]. It is up to the resource administrator to determine which resources should be clustered

together in this manner. The Launcher service provides meta-data such as processor count, processor speed and architecture advertised through the ICENI architecture.

2.4 Performance Repository

Performance data relating to the execution times of the components on different architectures and with different network bandwidths is collected in a performance repository. This repository is also responsible for collecting performance data from running components in order to improve future predictions. This is made available to the Scheduler through the Scheduling Framework API.

2.5 Scheduling Framework Application Programming Interface (API)

The Scheduling Framework provides the Scheduler with a standard interface to access all the meta-data (from the underlying ICENI architecture) relevant for scheduling, along with a number of helper functions. This maintains the modular design approach used within ICENI, thus allowing the Schedulers to be interchangeable.

All requests from the Scheduler to the Scheduling Framework require the user’s credentials, ensuring that only those resources, components and network connections accessible to the user are reported. Helper functions are provided for performance predictions, to discover component implementations and other tasks common to all schedulers.

3 Schedulers

We have implemented four different scheduling algorithms, using different underlying theories. The goal of each scheduler is to optimise some metric or combination of metrics, as requested by the user. In order to allow different optimisation policies to be used, the metrics are abstracted into a *benefit value*[11] which is computed via a *benefit function*. Each policy is specified as a benefit function describing the benefit of running an application using a given schedule as a function of scheduling metrics, such as predicted execution time and/or cost. Each scheduler is therefore concerned with maximising the benefit value.

ICENI applications consist of multiple components each of which must be scheduled for execution. We therefore require that all schedulers be capable of co-allocation [12, 15], the ability to schedule all components within an application at the ap-

propriate time.

Scheduling an application requires an estimate of the execution time and cost of each component. In order to compute the predicted execution time of a component on a resource, equation 1 is used. Define n as the number of components on that resource, m is the number of CPUs on that resource, b is the CPU speed of the resource and l is the number of operations required by that component. This models the slowdown that occurs when a resource executes more components than it has available CPUs. The cost of running a component is defined as the price per second, multiplied by expected duration.

$$\text{time } t = \frac{\lceil \frac{n}{m} \rceil l}{b} \quad (1)$$

3.1 Random and Best of n Random

The task of scheduling an application is a complex one often taking a significant amount of time. When the complexity of scheduling the application becomes sufficiently large it may be the case that the scheduling phase will be responsible for a major proportion of the total application duration. In extreme cases it is possible that the execution time saved by selecting the optimal schedule will be less than the increase in time required to find that schedule.

In order to avoid this problem we consider a random scheduling algorithm which simply chooses a resource at random for each component in the application. Other than checking that each component is capable of executing on its chosen resource, no form of active selection takes place. Producing a schedule this way is very fast and will not produce an excessive delay in selecting a schedule.

An optimisation of this algorithm is also presented. The Best of n Random scheduler randomly generates n schedules and then returns the one with the highest benefit value. This method maintains the speed of the random scheduler while increasing the chance of selecting a high quality schedule.

3.2 Simulated Annealing

Simulated annealing is a generalization of the Monte Carlo method used for optimisation of multi-variable problems. Possible solutions are generated randomly and then accepted or discarded based on the difference in their benefit in comparison to a currently selected solution, as shown in Algorithm 1. Simulated annealing has been used

in order to select how many resources a Grid application should be split over[18], but never to select which resources are used.

Algorithm 1 Simulated Annealing

```

1: cSol ← generateNewSolution()
2: cBenefit ← getBenefit(cSol)
3: while noAcceptedSolutions > 0 do
4:   noAcceptedSolutions ← 0
5:   for  $i = 0$  to maxNoOfTrialSolutions do
6:     tSol ← generateTrialSolution()
7:     tBenefit ← getBenefit(tSol)
8:     if acceptTrialSolution() then
9:       cSol ← tSol
10:      cBenefit ← tBenefit
11:      noAcceptedSolutions++
12:      if noAcceptedSolutions ≥ maxAcceptedSolutions then
13:        break out of for loop
14:      end if
15:    end if
16:  end for
17:  reduce T
18: end while

```

An initial schedule is generated at random and its benefit calculated. A new schedule, a permutation of the previous solution, generated by moving one component onto a different resource is then created. The new schedule is either accepted or discarded as the new solution through the Metropolis algorithm, see Algorithm 2. If the new solution has greater benefit value than the current solution it is accepted as the new selection. However, if the new solution has a lower benefit then it is accepted with a probability $e^{-d\beta/T}$, where $d\beta$ is the difference in benefit value between the two solutions, and T is a control parameter.

Algorithm 2 Metropolis Algorithm

```

1: if  $d\beta < 0$  then
2:   return true
3: else if  $R < e^{-\frac{d\beta}{T}}$  then
4:   return true
5: else
6:   return false
7: end if

```

This process is repeated with each iteration consisting of either a maximum number of new solutions being accepted N , or a maximum number of solutions being considered M . At the end of each iteration T is decreased. Once an iteration is completed with no new solutions being accepted, the current solution is returned as the best available solution.

Low values of T decrease the probability that a solution with a lower benefit value will be selected, as do large values of $d\beta$. At high values of T worse solutions are often accepted, reducing the chance of the algorithm getting caught in a local maxima. As T is decreased, so is the regularity with which worse schedules are accepted, allowing the algorithm to settle with the best algorithm found.

3.3 Game Theory

Grid scheduling can also be modelled using *game theory*, a technique commonly used to solve economic problems. In game theory a number of players each attempt to optimise their own payoff by selecting one of many strategies[4]. To apply this to scheduling each component is modelled as a player whose available strategies are the resources on which the component could run. Each player is unable to influence the strategy selected by other players leading to a branch of game theory classed as *non-cooperative*. The payoff for a player is defined to be the sum of the benefit value for running the component and all communication arriving at that component. Hence, for the player P_i , the payoff is dependent on the strategy chosen S_i , and the strategies chosen by P'_i , the set of players sending data to player P_i .

Algorithm 3 Elimination of strictly dominated strategies

```

1: for all candidateStrategies in StrategyList do
2:   strictlyDominated  $\leftarrow$  true
3:   for all alternateStrategy in StrategyList do
4:     if candidateStrategy is not dominated by
       alternateStrategy then
5:       strictlyDominated  $\leftarrow$  false
6:     end if
7:   end for
8:   if strictlyDominated then
9:     remove candidateStrategy from StrategyList
10:  end if
11: end for

```

Game theory examines the existence of a stable solution in which players are unable to improve their payoff by changing only their strategy. The game used here is a *static game of complete information*. This is a class of game in which the payoff for all players is known for all combinations of strategies and each player has to make one single choice of strategy. In this class of game, the stable state is known as the *Nash Equilibrium* see chapter 2 of [4] and can be shown to always exist. While

this solution is not always optimal for the collective payoff, in many cases it is close to optimal.

Algorithm 4 Is candidate strategy dominated by alternate strategy

```

1: for all combinations of remaining strategies
   from preceeding players do
2:   if payoff(candidateStrategy) > payoff(alternativeStrategy) then
3:     return false
4:   end if
5: end for
6: return true

```

The game is solved using *elimination of strictly dominated strategies*, in which the least optimal solutions are continually discarded. This is shown in Algorithm 3. All players simultaneously look at their list of potential strategies and for each strategy attempts to determine if it is *strictly dominated*. A strictly dominated strategy is one that can be said to be dominated by all other available strategies. Strategy S_i is dominated by S_k if, for all combinations of strategies of all other players, S_i has a lower payoff than S_k . Once a strategy has been identified as strictly dominated it is guaranteed not to be an optimal solution, so it can be removed from all further consideration.

To check for domination only the subset of players P'_i and P_i needs to be considered. In addition, many of the players in this subset will have a reduced list of potential strategies as some of their strategies will have been found to be strictly dominated. This significantly reduces the time taken to check for domination, this is shown in Algorithm 4.

Elimination of strictly dominated strategies is applied iteratively until each player has one strategy remaining, or a number of strategies, none of which are strictly dominated. In this case each player chooses a strategy at random.

4 Experimental Comparison

We wish to examine the effectiveness of the scheduling algorithms presented in order to be able to answer the question of which scheduler should be used. In order to do this we need to run each scheduler on a range of different applications and Grid configurations. As each algorithm has been implemented as a pluggable scheduler for the scheduling framework presented in section 2, we created a *simulated scheduling framework* which reads application and Grid configuration data from files and then answers queries from the scheduler

with this information. This allows repeatable experiments to be performed.

4.1 Experimental Setup

To test the schedulers, each scheduler was run on all possible permutation of application, grid and scheduling policy described below. Each scheduler was run five times on each permutation.

4.1.1 Applications

Twenty one applications each consisting of multiple components connected in a DAG configuration were tested. Each application had a depth of between 2 and 7. The number of components in the applications varied between two and thirty eight. The computational complexity of each component was such that its execution time on a 2GHz processor would be between zero and ten minutes, with an average of five minutes, while each communication stage between components would take between zero and two minutes, with an average of one minute, assuming a 100Mbit/s network connection, both with a Gaussian distribution.

4.1.2 Grids

The test Grid comprised of four clusters, containing resources as shown in Table 1. We present equation 2 to calculate the price of resources, defined as how much the user must pay per second of resource usage.

$$P = A\left(\frac{a}{\alpha}\right)^{1/3} B\left(\frac{b}{\beta}\right)^{2/3} C\left(\frac{c}{\gamma}\right) \quad (2)$$

Where a , b and c are the number of CPUs, CPU speed and Interconnect speed respectively, and α , β and γ represent a default node configuration. The powers on the first two terms represent the scalability of these attributes. A , B and C are weighting factors that must satisfy $ABC = 1$ in order to set the price of a default host to unity. For this experiment a default host was taken to be a host from the Viking T cluster, ie $\alpha = 1$, $\beta = 2GHz$ and $\gamma = 100Mbit/s$.

Each cluster is connected to all other clusters with 100Mbit/s network connections with the exception of Saturn and Rhea which are connected to each other via a 1Gbit/s link. Links are priced in a similar manner to resources, using equation 3

$$P = C\left(\frac{c}{\gamma}\right) \quad (3)$$

Each cluster can either be present or not present in a Grid creating a total of sixteen possible Grids.

The trivial cases where no resources exist, or only Saturn or Rhea exist were not tested. All other combinations of clusters were tested.

4.1.3 Policies

Two policies were considered, cost optimisation and time optimisation. Cost optimisation uses the benefit function shown in equation 4 while time optimisation, the benefit function shown in equation 5.

$$\beta = \begin{cases} 10 - \left(\frac{9}{20000}\right)p & p < 20000 \\ \frac{20000}{p} & p > 20000 \end{cases} \quad (4)$$

$$\beta = \begin{cases} 10 - \left(\frac{9}{600}\right)t & t < 600 \\ \frac{600}{t} & t > 600 \end{cases} \quad (5)$$

Where p is the total expected cost and t is the expected execution time.

4.2 Results

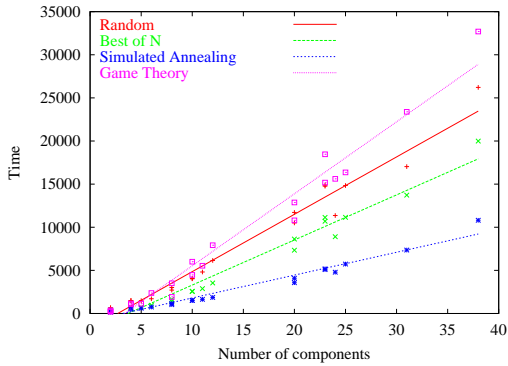
4.2.1 Time Optimisation

In time optimisation the aim is to minimise the time between the submission of the application and its return. Two distinct stages occur in this time period. First the application must be scheduled, and secondly it must be executed. The scheduling algorithms attempt to minimise the duration of the second stage but at the cost of increasing the duration of the first stage. In order to evaluate the effectiveness of this tradeoff we consider the *total time* which is the sum of the scheduling time and execution time.

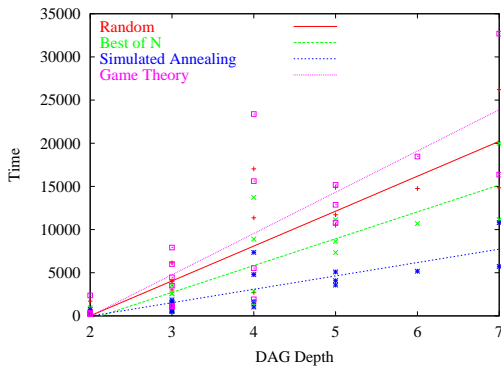
Figures 3A and 3B show how the different schedulers compare with varying number of components and DAG depth. These results are from the case where all four clusters were available for scheduling. The simulated annealing scheduler produces the shortest total time. The game theory scheduler however produced the longest times. This is due to two different factors. First the time taken to produce a schedule using game theory is long, and secondly as the game theory scheduler schedules each component individually, it is difficult for it to know when a resource is already loaded and so tends to overload the few best resources. The random and best of n random schedulers perform as expected with the best of n random scheduler outperforming the random scheduler.

Cluster	Number of Nodes	CPUs Per Node	CPU Speed	Interconnect Speed	Price
Viking T	16	1	2GHz	100Mbit/s	1
Viking C	16	1	2GHz	1Gbit/s	4
Saturn	1	16	750Mhz	5Gbit/s	12
Rhea	1	8	900MHZ	5Gbit/s	12

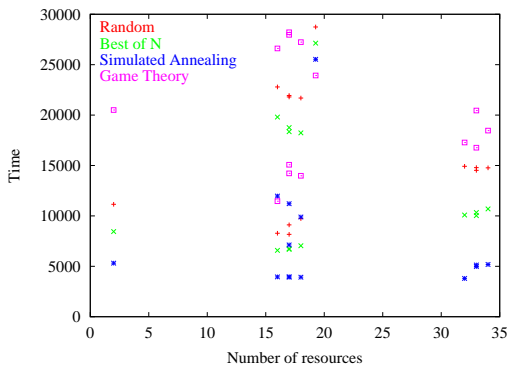
Table 1: Grid Clusters



A - Total time vs Number of Components



B - Total time vs DAG Depth



C - Total time vs Number of Resources

Figure 3: Time optimisation results

Figure 3C shows how the time varies as the number of resources vary. For these results the application was held at six stages long, with twenty three components. Again it shows the same spread of effectiveness of different schedulers but the interesting point here is that at sixteen, seventeen and eighteen resources there are two ranges of possible

execution times. This arises from either Viking C or Viking T being selected. However the case when there are more resources produces a range of times in between these values. This shows that increasing the number of resources available does not necessarily improve the time taken. In fact when more resources are added such that the optimal schedule does not change (ie the new resources are inferior to existing ones) the execution time does not change, but the number of schedules to be considered increases and so the time taken to discover the optimal schedule is longer.

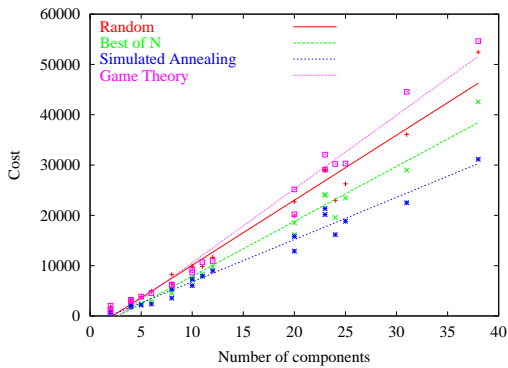
4.2.2 Cost Optimisation

Cost optimisation aims to minimise the expected cost of the application. The expected cost is defined as the sum of the price per unit time (as calculated by equation 2) multiplied by the expected execution time.

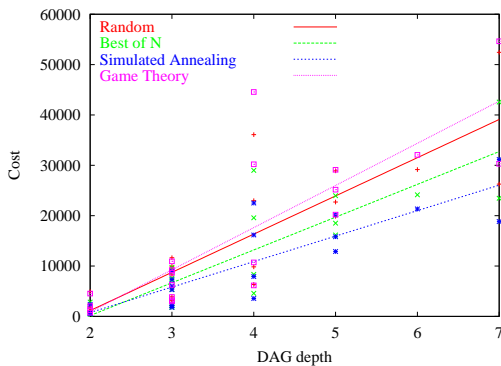
Figures 4A and 4B show how the different schedulers compare with varying number of components, and DAG depth. The simulated annealing scheduler produces the cheapest schedules, with the game theory scheduler again producing the worst (most expensive) schedules. This is explained as above by the game theory algorithm finding it difficult to avoid overloading a few resources.

Figure 4C shows how the cost of the result schedule varies with the number of resources available in the Grid. It shows that as the number of available resources increases then the cost decreases. This is to be expected, as there is a larger choice of resources and the scheduler is less likely to be forced to use an expensive resource. This is similar to the economic situation where a market is saturated by excess supplies.

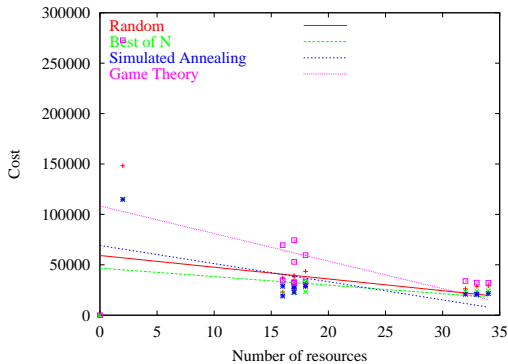
Again simulated annealing is generally the best scheduler, however at small number of resources it is outperformed by both the random and the best of n random scheduler. We believe that this is caused by the small problem space causing the simulated annealing algorithm to repeatedly jump out of the current maxima. This is especially likely in the case when only Saturn and Rhea are considered, as both resources have the same price.



A - Total cost vs Number of components



B - Total Cost vs DAG Depth



C - Total Cost vs Number of Resources

Figure 4: Cost optimisation results

the work within the components can be arbitrarily decomposed.

Maheswaran and Siegel[12] consider the co-allocation problem for scheduling DAGs onto a Grid system. Each DAG represents a single application with its nodes representing the tasks and the edges the communication between the tasks. Their algorithms sort the DAG into multiple blocks such that each block executes entirely before the succeeding block. At the beginning of execution of each block the remainder of the application is rescheduled using list scheduling, ordered on one of three different metrics if beneficial. We do not limit our algorithms to only working on DAG style applications, although these are the only ones considered in this paper.

Dail *et al.*[5] present an algorithm which decouples the search for resources from the selection of those resources. In order to reduce the search space considered by a scheduler resources are sorted into *Candidate Machine Groups* (CMGs) based on desirable machine characteristics (such as CPU speed) and aggregate characteristics (such as high network connectivity). Since each CMG will be significantly smaller than the set of all available resources, a scheduler can find the best mapping of tasks to resources within each CMG in a faster time. Each schedule produced this way can then be placed in a list and sorted according to any desired metric. This could be added into the ICENI system in order to decrease scheduling time.

YarKhan and Dongarra[18] use simulated annealing to address the issue of how many resources should be used when a problem can be split over a varying number of resources. As the number of resources increase the computation time decreases but the network overhead increases. We have shown that simulated annealing can also be used for choosing locations for component execution.

5 Related Work

Spencer *et al.*[15] have developed two algorithms for scheduling an application consisting of multiple tasks. Applicable only to problems with a divisible workload, their algorithms based on list scheduling order the tasks according to their execution position in the application. The number of instances of each task is then modified such that the output data rate of the preceding task(s) is balanced by the input data rate of the current task(s). Each task instance is then mapped to the resource that will execute it fastest. Our approach differs from this as we do not make the assumption that

6 Conclusion

We have presented the ICENI scheduling and launching architecture and shown how this enables multiple implementations of schedulers and launchers. This allows us to use different scheduling algorithms, and launch applications onto different Grid fabric solutions. We have presented four co-allocation scheduling algorithms and compared these under two different optimisation policies, showing how their effectiveness varies with three different parameters describing the problem complexity.

We have demonstrated that increasing the com-

plexity of application leads to both longer and more expensive solutions. However increasing size of the Grid reduces the cost of computations, due to increased choice. The effect of increasing the Grid size on the duration of an application size is undetermined, due to the unknown heterogeneity of the added resources.

Of the four schedulers shown, the simulated annealing produces the most optimal results and is therefore the ‘best’ scheduler. The game theory scheduler has proved disappointing, being outperformed by the random scheduler. This is due to the limitations of the uncooperative game theory implementation and might be improved by a cooperative game theory approach.

References

- [1] Jini Network Technology. <http://www.sun.com/software/jini/>.
- [2] Project JXTA. <http://www.jxta.org/>.
- [3] F. Berman. High-performance schedulers. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a new Computing Infrastructure*, chapter 12. Morgan Kaufmann, 1999.
- [4] Ken Binmore. *Essays on the Foundations of Game Theory*. Basil Blackwell, 1990.
- [5] Holly Dail, Henri Casanova, and Fran Berman. A decoupled scheduling approach for the grads program development environment. In *Supercomputing*, 2002.
- [6] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Available at <http://www.globus.org/research/papers/ogsa.pdf>.
- [7] N. Furmento, W. Lee, A. Mayer, S. Newhouse, and J. Darlington. ICENI: An Open Grid Service Architecture Implemented with Jini. In *SuperComputing 2002, Baltimore*, Baltimore, USA, November 2002.
- [8] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. An Integrated Grid Environment for Component Applications. In *2nd International Workshop on Grid Computing, Grid 2001*, volume 2242 of *Lecture Notes in Computer Science*, Denver, USA, November 2001.
- [9] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. Optimisation of Component-based Applications within a Grid Environment. In *SuperComputing 2001*, Denver, USA, November 2001.
- [10] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Journal of Parallel Computing*, 28(12):1753–1772, 2002.
- [11] Muthucumar Maheswaran. Quality of service driven resource management algorithms for network computing. In *International Conference on Parallel and Distributed Processing Technologies and Applications*, 1999.
- [12] Muthucumar Maheswaran and Howard Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Heterogeneous Computing*, 1998.
- [13] A. Mayer, S. McGough, M. Gulamali, L. Young, J. Stanton, S. Newhouse, and J. Darlington. Meaning and Behaviour in Grid Oriented Components. In *3rd International Workshop on Grid Computing, Grid 2002*, volume 2536 of *Lecture Notes in Computer Science*, Baltimore, USA, November 2002.
- [14] Sun Microsystems. How sun grid engine, enterprise edition 5.3 works. <http://www.sun.com/software/gridware/sgeee53/wp-sgeee/index.html>.
- [15] Matthew Spencer, Renato Ferreira, Michael Beynon, Tahsin Kurc, Umit Catalyurek, Alan Sussman, and Joel Saltz. Executing multiple pipelined data analysis operations in the grid. In *Supercomputing*, 2002.
- [16] Condor Team. Condor Project Homepage. <http://www.cs.wisc.edu/condor>.
- [17] The Globus Project. <http://www.globus.org/>.
- [18] A. YarKhan and J. J. Dongarra. Experiments with Scheduling Using Simulated Annealing in a Grid Environment. In *Grid*, November 2002.