Proc. of Parallel and Distributed Computing Systems (PDCS'99), Fort Lauderdale, August 1999, Tech Note DHPC-066.

Scheduling Independent Tasks on Metacomputing Systems

H.A. James, K.A. Hawick*and P.D. Coddington
Distributed and High Performance Computing Group
Department of Computer Science
The University of Adelaide
SA 5005, Australia

9 March 1999

Abstract

Metacomputing is a convenient and powerful abstraction for dealing with the complexities that arise when managing and using a large collection of heterogeneous computational resources. One of the most fundamental characteristics of a metacomputing system is the algorithm it uses for the scheduling placement of jobs on processing nodes. We describe five schedule placement algorithms, and report on their success and failure modes when used to schedule job distributions. We investigate five different distributions of job execution time and the effects of predictability on the algorithms' performance. Our objective in this work is to develop a hierarchical scheduling model for large scale job management in a metacomputing system. We investigate the use of a gateway model for controlling job placement on sub-clusters of a larger cluster of resources.

Keywords: metacomputing; scheduling; cluster computing; adaptive scheduling.

1 Introduction

The term metacomputer was coined by Fox [4] to describe a collection of possibly heterogeneous computational nodes which can be treated as a single virtual computer for both resource management and remote execution purposes [13]. General metacomputing environments allow users to submit serial or parallel programs and have tasks or jobs run on the virtual computer. Optimal scheduling of interacting jobs on metacomputer systems is a difficult problem. In this paper we focus on the simpler problem of scheduling independent tasks.

In implementing metacomputing environments it may be infeasible or impossible to have a pre-set up point-of-presence, or daemon, running on each node in the system. The node may not be able to support the environment in which the daemon is written. For example, Java is not available for some platforms such as the Thinking Machines CM-5. The node's administrator or owner may not want the machine directly accessible to the remainder of the metacomputing environment. For example on a Beowulf cluster [8] a front-end node may be acting as a gateway to hidden worker nodes. It is therefore sensible to design a software scheduling system to allow a designated node to act as an active gateway for other passive nodes.

Under this model, the active node must be able to schedule remote processes to provide good job throughput in the larger scale of the whole metacomputing environment, which may comprise a cluster of compute clusters. It must also ensure that fair access is granted to resources which may be shared. Metacomputing resources often comprise individual workstations and other computers accessed by their owners and other users *directly*, outwith the control of any scheduling software.

In this paper we focus on the problems of scheduling on cluster computing. There is already a large body of existing work in this area, including research projects such as: NQS [9]; PBS [2]; DQS [14]; EASY [11]; and the Prospero Resource Manager [12] (PRM), as well as commercial products such as: LoadLeveler [10]; and Codine [5]. These software systems are broadly characterised as queuing software which runs on *all* of the participating nodes.

*Contact Author, Email:khawick@cs.adelaide.edu.au, Fax: +61 8 8303 4366,Tel +61 8 8303 4519.

Although several support partial node availability they are often hard to set up and maintain with policy information spread across distributed nodes. These systems do however use individual node loads in load balance calculations, which is more difficult for a centralised scheduler that does not have access to daemons on each node. In this paper we experiment with a centralised scheduler model to manage a sub-cluster of resources. A full metacomputing environment may integrate many such transient sub-clusters together using the decentralised daemon approach with each sub-cluster managed by a daemon. This is useful if transient sub-clusters comprise resources temporarily assigned from other duties such as individual users' workstations.

We consider a number of scheduling algorithms. We measure their behaviour when all the jobs that are to be run on a particular sub-cluster are registered *before* the scheduling process begins, but the load state of the system is unknown. While a job might take a relatively well defined time to execute when all required input data is available on the local node, this can increase significantly when it must retrieve data either from the local gateway or a remote system.

The typical use case scenario we are interested in for our prototype DISCWorld [7] metacomputing environment is for high level user queries to be decomposed into well defined independent jobs. These jobs may be parameterised simulations, or simple data filtering jobs. The execution model is that of a gateway node which hosts a scheduling daemon which can remotely execute jobs independently on the sub-cluster it is responsible for.

2 Scheduling Independent Jobs

In a set of independent jobs, one of the defining characteristics is the execution time of each job. In the situation where each job has similar resource requirements, it is possible to test the performance of a scheduling algorithm on different distributions of job execution times. In this study, we approximate the time taken to copy the data necessary for the job to be executed, and also the effects of different sized data by considering different distributions of execution time. We consider five different distributions of task execution time: homogeneous; bimodal; uniform random; normal; and Poisson. All results we report in this paper were carried out on a cluster of 8 DEC Alpha Workstation running Digital Unix. We are experimenting with the effects of herogeneous cluster nodes but do not report on that work here. The job execution times that we used for the different distributions are shown in table 1.

Distribution	Characteristics
Homogeneous	job duration = 0.1sec
Bimodal	job duration randomly chosen from 0.1 or 10sec
Uniform Random	job duration interval [0.1,10)sec
Normal	mean = 0.1sec, σ = 1.0sec
Poisson	mean = 0.1sec, σ = 1.0sec

Table 1: Simulation job duration characteristics

As the name suggests, jobs with *homogeneous* execution times all run for the same duration. There are many examples of jobs with a homogeneous execution time distribution, most commonly found when requesting that the same application be run with either no inputs, slightly different, but equivalent inputs which are already present on the local node. *Bimodal* distributions occur when the jobs they represent take one of two durations to execute. In the context of a metacomputing environment, this distribution occurs most often when a single job is being executed which sometimes has to retrieve input data from a remote node. For the purposes of this study, when jobs have a bimodal execution time distribution, they have an equal (50%) chance of being randomly assigned to be a long or short job. *Uniform random* distributions must be used to model jobs when nothing is known about their execution time. An example of this is when a job has roughly equal probability of being able to either run without any parameters, or it may need parameters that can vary in size, and may be available from the local node or a remote node. A *normal* distribution occurs most frequently when a job is used with input data that is only loosely clustered around a mean size, which is most often available from the local node. An example of a Poisson job distribution is a job which is usually run with input data that is very tightly clustered around a mean size and which is nearly always resident on the local node.

3 Scheduling Algorithms

Efficient scheduling software should minimise idle processing time. No single algorithm can achieve optimal performance on all possible job execution time distributions. However, given extra information such

as which distribution dominates the job spectrum a smart scheduling system can choose which policy will be near optimal. To investigate this, we tested five scheduling policy algorithms against some simple job time distributions. We implemented these algorithms in a prototype centralised scheduler program known as *dploader*. This multi-threaded program was implemented in C and uses a single controlling thread to assign jobs to remote processors and an additional single thread per processor executing remote jobs. It therefore acts as the central sub-cluster manager discussed above and provides a test framework. At run time, the *dploader* program is set up with a host list of those remote nodes or hosts to which it is able to assign jobs.

A number of scheduling algorithms were tested, including: round-robin (perfect); round-robin with clustering (perfect-clustering); minimal adaptive (adaptive); continual adaptive (continual-adaptive); and first-come first-serve (fcfs). The names in parentheses are used as shorthand on the graphs and tables of results.

Scheduling is said to be *static* when the processors on which the jobs will run are assigned at compile-time or before execution. *Dynamic scheduling* [1] or *load balancing* is performed at run-time. This involves all jobs being assigned to processors, and the use of transfer policies and placement policies in order to decide when a job will be moved between processors and to where it should be moved, respectively.

Our problem is that we are not able to run daemons that report accurate and up-to-date sub-cluster node status information to the centralised scheduling software. The only information that we receive from a remote node is the execution time of the last job, or if multiple jobs were run, the average execution time. In our metacomputing context, the classic terminology of static and dynamic scheduling is no longer entirely appropriate due to its inability to describe run-time job assignment algorithms which use minimal information about remote nodes. These algorithms are not strictly static as job placement is performed at run-time. They are also not strictly load balancing algorithms in the sense that a centralised algorithm decides job placement and assignment until jobs are actually ready to be executed.

We define performance as the total time taken to schedule and execute the jobs assigned. This metric incorporates not only the time taken to execute jobs on the remote machine, but also overheads associated with the establishment of remote shells, and any other time spent waiting for a job to be assigned to a waiting processor. The time taken to establish a job (or group of jobs) on a remote machine is termed the job assignment latency.

In **perfect** or **round-robin** scheduling, each node receives an equal proportion of the jobs. Thus, if there are n jobs and m nodes in the workstation cluster, then each machine will receive $\frac{n}{m}$ jobs to process. Nodes are cyclically assigned a single job in the order of their appearance in the host list after the preceding host has been assigned; thus node $i+1$ will not receive a new job until it is possible to assign a job to node i. Processor 0 receives job 0, processor 1 receives job 1, ..., processor 0 receives job m, processor 1 receives job $m+1$ and so forth. It can be seen that this implementation of perfect scheduling may lead to node $i+1$ being idle if node i is relatively slower. This algorithm is best suited to the case in which the processors that comprise the sub-cluster are homogeneous, both with respect to node performance, and actual load on the processors. Perfect scheduling is designed to average out the number of jobs executed on each host. Unfortunately, this algorithm is expected to be the worst performing of those studied, for several reasons, including those of job and node non-homogeneity in practice, as well as high individual job assignment latency.

In the **clustered round-robin** algorithm, the total number of jobs are divided amongst the host list. Therefore, a processor receives all of the jobs it must execute at startup. Thus, processor 0 receives jobs $0 \ldots \frac{n}{m} - 1$, processor 1 receives jobs $\frac{n}{m} \ldots \frac{2n}{m} - 1$, and so forth. This algorithm has the advantages of being the easiest out of those studied to implement, and also that it reduces the job assignment latency. However, it suffers from the deficiency of not being able to control the number of jobs to execute if a processor is found to be relatively slower than the others. In terms of job assignment latency, this scheduling algorithm is the least expensive of those studied, as it only performs a single job assignment on each remote processor. Thus the execution time of the whole program is that of the slowest processor, or if the processors are homogeneous, the processors that gets assigned the greatest number of longer jobs in a distribution.

The algorithm we term **minimal adaptive** scheduling is that of testing the relative performance of nodes *before* placing the remaining jobs onto them. The object of this is to avoid the effect incurred by the clustered round-robin algorithm whereby the same number of jobs may be assigned to execute on machines with vastly different loads. Provided $n > m$, where n is the number of jobs and m is the number of

processors, we select and place the first m jobs onto the nodes, one job per node. After timing how long it takes for each node to process these single jobs, a relative ranking of nodes is made, and the remaining $n-m$ jobs are proportionally assigned to the nodes. This algorithm does not incur as much job assignment latency as the perfect scheduling algorithm, but as it performs two sets of job assignments (one to perform benchmarking on the remote hosts, and the other to assign the remainder of the jobs), it suffers from double the latency of clustered round-robin scheduling. Minimal adaptive scheduling makes the underlying assumption that the state of the target processor for the duration of the first job is indicative of the steady state of the machine. Furthermore, the algorithm ignores job heterogeneity. It is quite possible that if the first m jobs that are in the queue waiting to be processed are not the same, then the resulting estimate of the relative processing capacity of the target processors will be incorrect. Fluctuations in the machines' load are ignored by this algorithm.

Continual adaptive scheduling is an algorithm designed to achieve the best turnaround time for a group of jobs. Jobs are assigned to each processor in groups of t, a tunable parameter. After each processor has finished processing its group, the group execution time is recorded, and the total number of jobs that the processor will be able to execute such that all processors should finish at approximately the same time, is estimated. The continual adaptive scheduling algorithm performs better than the perfect algorithm, but not as well as clustered round-robin nor as well as the minimal adaptive algorithms in terms of job assignment latency. This is due to the need for the algorithm to assign blocks of jobs to the remote processors, and analyse the performance of each, before assigning new jobs.

First-come first-serve (FCFS) scheduling refers to the technique of placing all available nodes into an ordered list, and assigning each job individually to the node at the front of the list. Once a node is assigned a job, the node is removed from the head of the list until it has finished processing its assigned job, when it is added to the end of the host list. This algorithm has the benefit of favouring the faster nodes, as the faster a node processes a job, the quicker it will be added on to the end of the list, and the sooner it will be assigned a new job. This scheduling algorithm provides the best throughput with respect to job execution, but will cause the machines to be used in an unbalanced fashion. The major deficiency of the algorithm is its need for a low job assignment latency, since jobs are assigned individually.

4 Algorithms Test Results

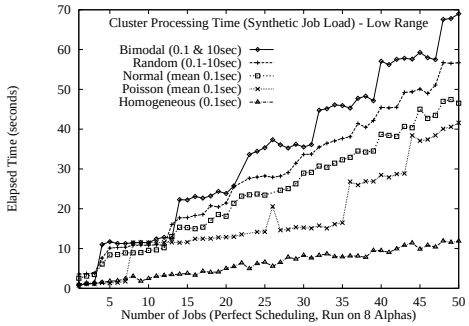
The five scheduling policy algorithms were tested on a variety of homogeneous and heterogeneous clusters of processing nodes, using many independent job distributions. Each distribution of between 1 and 1000 jobs was drawn from the stated statistical distribution. In the limit of $n >> m$, where each machine had a large number of jobs, the scalable behaviour is summarised in table 2. This shows the job throughput and job assignment latency overhead figures for the whole cluster, which scale linearly as expected, but which differentiate the efficiency of each algorithm for different job time distributions. Because job assignment latency is calculated from the execution-time graphs, if the majority of the jobs in the low end of a distribution are very short, this has an effect of skewing the calculation. Figures 1 and 2 show the $n \approx m$ behaviour, which illustrates some of the characteristics of each algorithm.

Generally, the homogeneous distribution evokes the best performance from each of the considered scheduling algorithms. This is because of the uniformity in job execution time, and allows us to view the impact of job assignment latency.

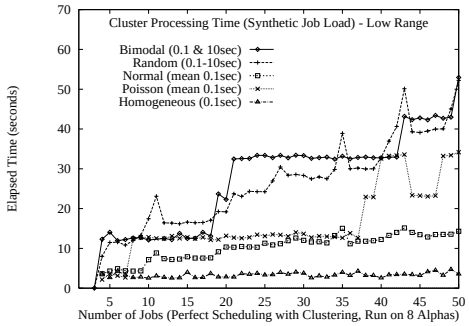
The round-robin scheduling algorithm, shown in figure 1 i), performs no prediction at all. Hence the performance of this algorithm is reflected in a distribution's clustering around the mean job execution time in a synthetic job load. Thus, the jobs exhibiting a homogeneous execution time distribution perform best, followed by Poisson, normal, uniform random and then bimodal. The high variability in the Poisson, normal, uniform random and bimodal distributions causes delays in the assignment of jobs to processors. Because jobs are individually assigned to processors, a high component of the overall performance is the job assignment latency.

On average the slopes representing job execution time distributions in clustered round-robin scheduling, figure 1 ii), are flatter, reflecting a single job assignment latency. The homogeneous and normal distributions perform the best, due to their symmetric clustering of values around the mean. The Poisson distribution, although tightly clustered around the mean, lacks the symmetry of the normal distribution, which is reflected by decreased relative performance. The algorithm performs poorly with random and bimodal job execution distributions due to their high variability.

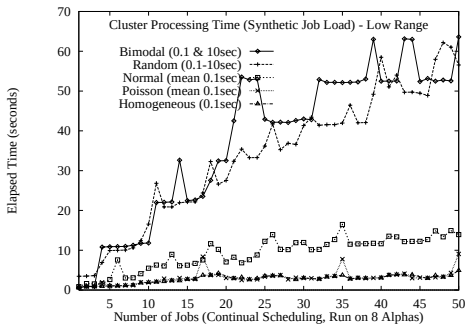
i)



ii)



iii)



iv)

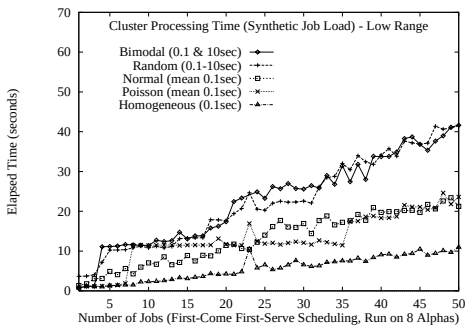
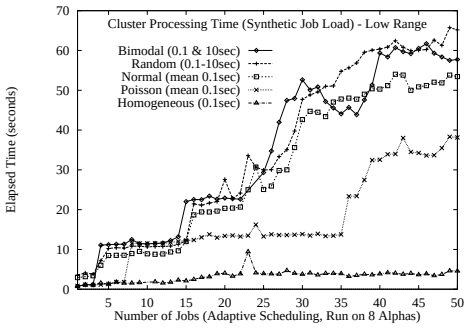


Figure 1: i)-iv) Total execution time for 1-50 processes across 8 machines

i)



ii)

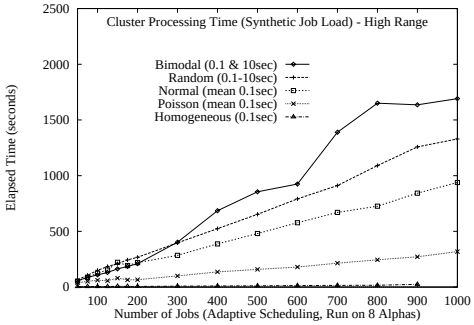


Figure 2: i) Total execution time for 1-50 processes across 8 machines, ii) Execution time for 50-1000 processes across 8 machines

The continual adaptive algorithm's performance, shown in figure 1 iii), is best with those distributions that are easily predictable — homogeneous, Poisson and normal, even though the algorithm incurs a job assignment latency for each processor after synchronization. This is due to its ability to predict, with a greater degree of accuracy, the time that a certain number of jobs will take to run before synchronization is necessary. The main source of performance degradation in this algorithm is the wasted time when processors are idle waiting for another to finish before synchronizing. Bimodal and random, because of their high variability, are hardest to predict successfully.

The slopes representing job execution time distributions in the test of first-come first-serve algorithm, shown in figure 1 iv), are more tightly clustered than those of round-robin, clustered round-robin and continual adaptive. This is due to the opportunistic nature of the scheduling algorithm, where those distributions exhibiting a mixture of long and short jobs average out over many repetitions. Making no predictions on the execution time of future jobs, this algorithm exhibits the best performance out of all algorithms studied for uniform random and bimodal distributions

Alg	Dist	Time/Job(s)	O/head (s)
R-R	homo	0.232 ± 0.005	0 ± 1
	Poisson	0.732 ± 0.006	0 ± 2
	normal	0.887 ± 0.006	1.5 ± 1.5
	random	1.051 ± 0.003	1.6 ± 0.7
	bimodal	1.322 ± 0.002	0 ± 0.5
R-R with Clust	homo	0.0154 ± 0.0004	2.9 ± 0.1
	Poisson	0.170 ± 0.005	13.2 ± 1.3
	normal	0.199 ± 0.001	5.1 ± 0.3
	random	0.663 ± 0.003	9.2 ± 0.8
	bimodal	0.686 ± 0.004	9.7 ± 0.9
Min Adapt	homo	0.0199 ± 0.0008	2.7 ± 0.2
	Poisson	0.299 ± 0.005	10.8 ± 1.2
	normal	0.935 ± 0.008	9.1 ± 1.9
	random	1.344 ± 0.006	0.9 ± 1.4
	bimodal	1.81 ± 0.03	0 ± 7
FCFS	homo	0.185 ± 0.002	0.8 ± 0.5
	Poisson	0.299 ± 0.003	5.1 ± 0.7
	normal	0.413 ± 0.007	1.7 ± 1.8
	random	0.754 ± 0.002	3.3 ± 0.6
	bimodal	0.763 ± 0.002	3.3 ± 0.6
Cont Adapt	homo	0.0242 ± 0.0007	2.1 ± 0.2
	Poisson	0.0238 ± 0.0008	2.4 ± 0.2
	normal	0.151 ± 0.001	5.5 ± 0.3
	random	0.834 ± 0.003	11.8 ± 0.9
	bimodal	0.962 ± 0.006	12.4 ± 1.6

Table 2: Execution time per job (seconds) and job assignment latency overhead (seconds) across all machines in cluster, as measured from experiments varying the number of jobs for each algorithm and job execution time distribution

of job execution time. Since there are no predictions, there are no penalties for incorrect predictions, unlike in the case of the continual adaptive algorithm.

The minimal adaptive algorithm, shown in figure 2 i), represents the minimal amount of effort that can be expended in order to make an estimate of the relative performance of remote processors, based on actual performance of a single job. Thus, in the absence of any load information, it provides a rough basis for processor comparison. This algorithm also takes far less time to make predictions than the continual adaptive algorithm. The performance of the algorithm is similar to that of the continual adaptive algorithm with the exception that the predictions of Poisson and normal distributions, based on a single point, are far more inaccurate, increasing the total execution time. Thus the more predictable the execution time of a job is, the greater performance this algorithm will achieve.

In the limit of large job numbers, the performance of all the algorithms, across all the distributions scaled linearly with job numbers, assuming fixed number of available processing nodes. Most algorithms settled into a steady-state performance scaling regime, and algorithm performance can be ranked in the order of best to worst for the distributions: homogeneous, Poisson, normal, random and bimodal.

In the large job number regime, the minimal adaptive algorithm, shown in figure 2 ii), exhibits irregular behaviour for the bimodal job execution time distribution, even for relatively large numbers of jobs. This is due to the algorithm failing to predict the job execution times properly, which is its major deficiency, but an inherent problem with a bimodal job time distribution.

The difference between the performance of the minimal and continual adaptive algorithms is insignificant in the case of a homogeneous distribution of job execution times. However, as the predictability of job execution times becomes more difficult, the benefit of collecting more execution-time data becomes evident.

In summary, if the distribution of job execution times is known in advance, as in the homogeneous case, then the algorithm with the least amount of job assignment latency and prediction overhead, clustered round-robin, is most suitable. For those distributions exhibiting symmetry, or a high degree of clustering around the mean, such as the Poisson and normal distributions, a continual adaptive algorithm is most suitable. The first-come first-serve algorithm performed the best with those distributions that were hardest to predict, uniform random and bimodal.

5 Discussion and Conclusion

Scheduling jobs on a general metacomputing system can be tackled by modelling the system as a hierarchy of sub-clusters. Each cluster of nodes can be managed by a gateway node with the expected performance and ease of implementation as we have described. Clusters of these sub-clusters can potentially be managed using one or more of the existing scheduling software systems, where each managed node in the cluster is a gateway to a sub-cluster.

In general, we believe the terminology for static and dynamic scheduling is not entirely appropriate in the metacomputing context. For example, those algorithms that perform process placement at run-time but only use a centralized scheduler with minimal knowledge of the remote processors do not fall neatly into either the categories of static or dynamic scheduling.

The Globus toolkit [3] and Legion [6] are two metacomputing projects which exhibit diametrically opposite approaches to resource management and process execution. Globus uses third-party schedulers, and users are able to submit arbitrary binaries for execution, while Legion is based on an object-oriented runtime environment and all user processes comprise of objects which extend a Legion base class. Thus Globus is only able to use the information reported by the third-party schedulers, whereas the Legion run-time system is able to maintain more detailed information on jobs. Using the hierarchical scheduling approach described in this paper, we intend that our DISCWorld metacomputing system will be able to reap the benefits of both approaches.

The results have shown that for job execution times with a high degree of symmetry around a mean, a continual adaptive scheduling algorithm is most suitable, while for those distributions which are not easily predictable, a first-come first-serve algorithm performs most efficiently.

We have shown that no one scheduling algorithm consistently produces the best performance across the job execution time distributions we have studied. Therefore it is important to know more about the actual distribution so that an appropriate algorithm can be chosen. Through the collection and exploitation of historical job execution times, we anticipate it may be possible for a *smart* scheduler to select and adapt to the most appropriate scheduling policy.

Acknowledgments

This work was carried out under the Distributed High Performance Computing Infrastructure Project (DHPC-I) of the On-Line Data Archives Program (OLDA) of the Advanced Computational Systems (ACSys) Cooperative Research Centre (CRC) and funded by the Research Data Networks (RDN) CRC. ACSys and RDN are funded by the Australian Commonwealth Government CRC Program.

References

[1] Ishfaq Ahmad. Editorial: Resource Management in Parallel and Distributed Systems with Dynamic Scheduling: Dynamic Scheduling. *Concurrency: Practice and Experience*, 7(7):587–590, October 1995.

[2] A. Bayucan, R. L. Henderson, T. Proett, D. Tweten, and B. Kelly. Portable Batch System External Reference Specification. NAS Scientific Computing Branch, NASA Ames Research Center, California, June 1996.

[3] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infra-structure Toolkit. *Int. J. Supercomputer Applications*, 1996.

[4] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works!* Morgan Kaufmann Publishers, Inc., 1994. ISBN 1-55860-253-4.

[5] Genias Software GmbH. CODINE: Computing in Distributed Networked Environments. URL http://www.genias.de/-genias/english/codine/Welcome.html, September 1995.

[6] Andrew S. Grimshaw and Wm. A. Wulf and the Legion team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), January 1997.

[7] K. A. Hawick, H. A. James, A. J. Silis, D. A. Grove, K. E. Kerry, J. A. Mathew, P. D. Coddington, C. J. Patten, J. F. Hercus, and F. A. Vaughan. DISCWorld: An Environment for Service-Based Metacomputing. *Future Generation Computing Systems (FGCS)*, 1998.

[8] K.A. Hawick, D.A. Grove, and F.A. Vaughan. Beowulf - A New Hope for Parallel Computing? Technical Report DHPC-061, Advanced Computational Systems Cooperative Research Center, Department of Computer Science, University of Adelaide, January 1999.

[9] S. Herbert. Official Administrator's Guide to Generic NQS. Sterling Software, September 1994.

[10] International Business Machines Corporation. LoadLeveler. Network job scheduling and job management. 16 May 1998. http://www.austin.ibm.com/-software/sp_products/loadlev.html.

[11] David A. Lifka, Mark W. Henderson, and Karen Rayl. Users Guide to the Argonne SP Scheduling System. Technical Report ANL/MCS-TM-201, Mathematics and Computer Science Division, Argonne National Laboratory, May 1995.

[12] B. Clifford Neuman and Santosh Rao. The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems. *Concurrency: Practice and Experience*, 6(4):339–355, June 1994.

[13] Larry Smarr and Charles E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, June 1992.

[14] Supercomputer Computations Research Institute. Distributed Queueing System 3.1.3 Reference Manual. Florida State University, Tallahassee, Florida, March 1996.