# Localised Workload Management using Performance Prediction and QoS Contracts

D.P. Spooner * J. Cao J.D. Turner
H.N. Lim Choi Keung S.A. Jarvis G.R. Nudd

**Abstract**

In this research, we approach the problem of grid workload management with the development of a multi-tiered architecture that employs performance prediction and a hierarchy of homogeneous brokers to meet quality-of-service contracts and improve resource efficiency. By coupling application performance data with iterative heuristic algorithms, the framework is able to balance the processes of minimising run-to-completion time (makespan) and idle time, whilst adhering to service deadlines on a per-task basis. This paper provides a brief introduction to the architecture, and then focuses on the lowest tier which is responsible for scheduling tasks on grid-enabled resources. Experimental results demonstrate that these components are able to adjust to the conflicting requirements of system metrics and QoS contracts in real-time, using the just-in-time capabilities of the PACE prediction environment and the iterative workload management algorithms.

## 1 Introduction

It is anticipated that grids will develop to deliver high performance computing capabilities with flexible resource sharing to dynamic virtual organisations [8]. Essential to this growth is the development of fundamental infrastructure services that will integrate and manage large heterogeneous pools of resources, and offer them seamlessly to differentiated users. A key service is workload management, which involves distributing tasks and data over a selection of appropriate resources and coordinating communication and transfer systems. While commonality exists between grid workload management and task scheduling on MIMD architectures (and clusters), the dynamic and diverse nature of the grid, coupled with multiple domains of administration, differentiates this type of scheduling from traditional multi-processor work.

The research presented in this paper considers the allocation of independent tasks to groups of distributed, heterogeneous systems where performance models exist and hardware has been characterised using benchmarking tools. The resulting system, known as Titan, adopts a multi-tiered approach to workload management, where standard grid protocols (such as Globus [7]) are used as the top tier, a collection of service-providing brokers are used at the middle tier and a localised scheduling system is used at the lowest tier.

*High Performance Systems Group, Dept. of Computer Science, University of Warwick, Coventry. Email: {dps, junwei, jdt, hlck, saj, grn}@dcs.warwick.ac.uk

This paper focuses on the lowest-level tier where the performance prediction environment PACE [9] is used to estimate execution times for a particular task on a given a set of architecture types prior to run-time. This is combined with iterative heuristic algorithms to select schedules and organise the task queue to a given metric or sets of metrics. The rationale for selecting such an algorithm is that it adapts to minor changes in the problem space (such as a host disconnecting, or a user submitting a task), and can be guided by means of a fitness function to improve task allocation according to multiple metrics.

There are a number of other active grid projects that address workload and resource management issues, most notably AppLeS [2], Nimrod [3] and Ninf [10]. Scheduling in AppLeS and Ninf utilise the NWS [11] resource monitoring service and are based on performance evaluation techniques; a different approach is taken in our work which is based on the performance prediction capabilities provided by PACE. The PACE evaluation engine is parametric which is similar to the kernel module of the Nimrod architecture; Nimrod also utilises hybrid heuristic algorithms [1] for allocating jobs in a grid environment.

The organisation of this paper is as follows: section 2 introduces the workload management problem and how a performance prediction system can assist; in section 3, the algorithm used in the lowest tier is presented along with the coding scheme. Section 4 discusses the implementation and introduces preliminary experimental data; conclusions are presented in section 5.

## 2 Workload Management

While grid computing is not aimed specifically at the high performance community, many organisations regard the grid as a means to deliver commodity computation that exceeds the capabilities of their current systems. While grid software environments will exist at the organisational level, harnessing the computing power of local clusters and dual-mode workstations, it is envisaged that full computational grids will exist across many organisations, geographic regions and domains of administration.

This presents difficulties when attempting to improve single application performance which in the past may have included the tuning of a specific algorithm, the re-mapping of data, or the adjustment of communication behaviour. These factors can all be considered, to some extent, during development where appropriate decisions can be made based on application usage and expected behaviour. In a grid setting, system-wide issues such as contention, load and congestion can incur a major impact on performance and are difficult factors to predict *a priori*, particularly when access to system information is restricted. Additionally, computational systems will invariably provide services to customers with conflicting needs and requirements. Contract issues including deadline and response times must therefore be balanced in order to meet respective service-level agreements.

It is therefore non-trivial for workload management systems to select suitable resources for a particular task given a varied, dynamic resource pool. The search space for this multi-parameter scheduling problem is large and not fully defined until run-time. As a result, a *just-in-time* approach to performance prediction is adopted in this research along with a collection of distribution brokers and localised workload management systems, so that run-time variables and resource load can be used to assist task and resource allocation while maintaining prescribed service contracts.

## 2.1 Titan Architecture

To address the issue of scalability, Titan adopts a loosely hierarchical structure consisting of workload managers, distribution brokers and Globus interoperability providers. These services represent different tiers in the framework and can be provided by any distributed Titan node. While different behaviours are exhibited by individual nodes depending on whether they are configured for a particular tier, essentially each node is identical.

The top tier provides Globus interoperability via a layer that allows tasks to be submitted to the distribution broker on the local node or to other nodes directly. The distribution broker systems are based on the A4 advertising and discovery agent structure [4]. At the lowest tier, the workload managers are responsible for apportioning tasks to resources.

The hierarchical multi-tiered structure is illustrated in figure 1. Tasks are submitted to a service broker by Globus or an A4 portal through the interoperability layer. The distribution broker subsequently integrates the local workload manager (if one exists) to ascertain whether the task can execute on the available resources and meet service quality metrics (QoS contracts). If successful, the task is submitted to the local manager for scheduling. If not, the broker will attempt to locate a remote resource using the discovery and advertisement processes.

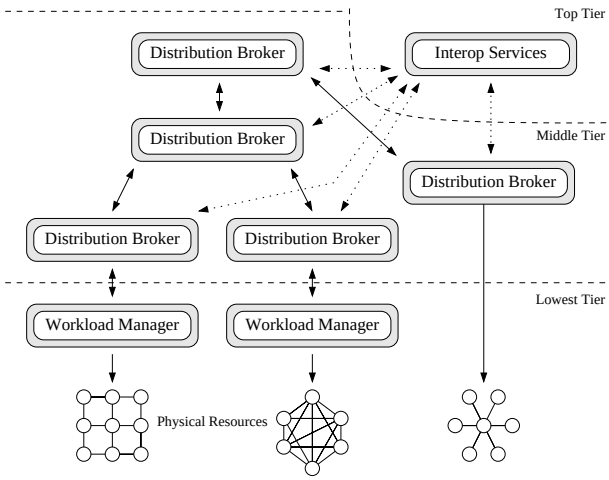


Figure 1: The Titan architecture consisting of distribution brokers that communicate with other brokers and workload managers to execute tasks in accordance with service policy restraints. An interoperability layer provides an interface to Globus permitting job submission to a particular broker.

In a pure 'discovery' environment, tasks that cannot run on a local workload manager force the distribution broker to probe or 'discover' other brokers on demand. While this has a performance impact on the network, it may be applicable to a system with high dynamics. In an 'advertisement' orientated environment, brokers will publish their schedules periodically so that lookups are not required. In practice, a trade-off is found between these extremes and can be configured as the system is running as it is set by a series of broker policies that determine what information is cached and how.

This paper concentrates on the lowest tier whose objective is to organise tasks in a manner that satisfies contract policies and system metrics (such as minimising execution time or idle time). At each tier a different strategy is employed but the overall effect is the same - tasks enter the system with requirements that are used in conjunction with performance prediction to map tasks to resources.

## 2.2 Task Allocation

In this work, the localised scheduling problem is defined by the allocation of a group of independent tasks $T = \{T_0, T_1, \ldots, T_{n-1}\}$ to a network of hosts $H = \{H_0, H_1, \ldots, H_{m-1}\}$. Tasks are run in a designated order $\ell \in P(T)$ where $P$ is the set of permutations. $\ell_j$ defines the $j^{th}$ task of $\ell$ to execute where $\forall \ell_j, \exists \beta_j \subseteq H, \beta_j \neq \emptyset$, that is, each task in $\ell$ has a suitable host architecture on which to run. To provide a compact representation of a schedule, a two dimensional coding scheme is utilised:

$$S_k = \begin{array}{c|cccc} & \ell_0 & \ell_1 & \ldots & \ell_{n-1} \\ \hline H_0 & M_{0,0} & M_{0,1} & \ldots & M_{0,n-1} \\ H_1 & M_{1,0} & M_{1,1} & \ldots & M_{1,n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ H_{m-1} & M_{m-1,0} & M_{m-1,1} & \ldots & M_{m-1,n-1} \end{array} \quad (1)$$

where $S_k$ is the $k^{th}$ schedule in a set of schedules $S = \{S_0, S_1, \ldots, S_{p-1}\}$ and $M_{ij}$ is the mapping of hosts to a particular application:

$$M_{ij} = \begin{cases} 1, & if \quad H_i \in \beta_j \\ 0, & if \quad H_i \notin \beta_j \end{cases} \quad (2)$$

The objective of the framework is to produce schedules that fulfil system and user requirements. This is achieved using an algorithm that selects suitable candidate schedules from $S$ and produces new schedules in $S'$. A fitness function is used to guide the selection of successful schedules by ejecting poor schedules from the system and maintaining good schedules in the representation space.

In order to obtain the fitness values, application start and end times must be calculated and composited into a schedule. In the first instance, we consider the simpler case where all the hosts are architecturally identical and that the communication costs between hosts are similar. In this homogeneous situation, the predicted execution time for each task is simply a function, $p_{ext}()_j$, of $\beta_j$ which is the number of allocated resources for task $\ell_j$. Assuming that this function is known in advance, and that a run-to-completion (non-overlap) strategy is adopted, the end time for a particular task is equal to the earliest possible start time plus the expected execution time, and can be readily obtained from the schedule representation $S_k$.

Each arrangement of $\beta_j$ is a unique *performance scenario* that will result in a different run pattern for each task. While distributed applications exist that benefit from as many hosts as are available, most will experience degradation in performance as the communication costs become dominant. It is therefore important that $\beta_j$ is selected appropriately depending on the task involved.

In the case where all the architectures are identical, the number of different performance scenarios is simply equal to the number of hosts. It is therefore straightforward to formulate the function $p_{ext}()_j$, either by retaining historical data or by one-shot executions. However, in a heterogeneous grid environment, the architectural and system diversity is enormous and the potential number of performance scenarios increases exponentially. For $m = 64$, the number of possible scenarios is of the order of $10^{89}$ in the extreme case where each host is entirely different. For any non-trivial resource network, it would be impossible to obtain $p_{ext}()_j$ in real time prior to scheduling.

### 2.3 Performance Prediction

The PACE system provides a method to obtain $p_{ext}()$ dynamically, given an application model and suitable hardware descriptions. The models are accurate for predicting the behavioural properties of scientific computing applications on static, pre-modelled architectures. For applications whose internal structures have a low level of data-dependency, the performance models can be within 5% of the actual run-time.

PACE models are modular and are constructed by a set of hierarchical layers. At the uppermost level, applications are captured as a sequence of parallel subtasks; where each subtask is subsequently described using control flow and resource usage information. The parallel characteristics of the subtasks are described using parallelisation templates which allow the computation-communication interactions between processors to be described. At the lowest level in the model the target hardware is characterised.

When combined, the modelled system components form a complex representation which can be used to establish prediction metrics for the execution of a particular application on a target architectural platform. The level of detail in the model can be chosen so as to improve accuracy and models are accumulated so that they can be reused. The use of separate parallelisation strategies also provides a convenient means of analysing the effects of different parallelisation strategies on resource usage and execution time.

As the models are parameterised, it is possible to evaluate $p_{ext}()_j$ with differing host allocations and task models. Additionally, the models evaluate rapidly permitting many enquiries per second. However, while PACE can provide the timing data, the combinatorial problem still exists as there are many possible arrangements of task orders and host mappings, even for a limited number of tasks and hosts.

## 3 Localised Workload Management

Workload managers utilise PACE and iterative heuristics to explore areas of the solution space to find good-quality schedules. Such algorithms are well-suited to search problems of this nature [6] and have been applied to a number of similar areas such as dynamic load balancing [12]. The approach used in this work generates a set of initial schedules, evaluates the schedules to obtain a measure of fitness, selects the most appropriate and combines them together using operators (crossover and mutation) to formulate a new set of solutions.

This is repeated using the current schedule as a basis, rather than restarting the process, allowing the system to capitalise on the organisation that has occurred previously.

As with other iterative searching algorithms, this process is based upon the notion of a fitness function that provides a quantitative representation of how 'good' a solution is with respect to other solutions. In this manner, a set of solutions can be created and evaluated with respect to multiple metrics to isolate superior solutions. This 'survival of the fittest' approach maintains good solutions and penalises poor solutions in order to move toward a good-quality result.

## 3.1 Task Organisation

### 3.1.1 Timings

In order to evaluate the fitness functions that drive the scheduling process, the expected end-time $te_j$ for each task in the schedule queue must be determined. This is derived from the task start-time plus the task's execution time, given by the following expression:

$$te_j = ts_j + p_{ext}(\beta_j)_j \tag{3}$$

The start time, $ts_j$, can be considered as the latest free time of all hosts mapped to the application. If all of the mapped hosts are idle, then $ts_j$ is set to the current system time $t$, hence:

$$ts_j = \max_{\forall i, H_i \in \beta_j} \{tr_{ji}\} \tag{4}$$

where $tr_{ji}$ is the *release time* for task $j$ on host $i$. For tasks $\ell_0 \ldots \ell_{j-1}$ that execute on $H_i$, this is set to the end time of the application. Where there are no previous applications on this host, then this is set to the host release time, $hr_i$, which is the time when $H_i$ is expected to become available. It is based upon tasks in the running queue and is set from $te_j$ when tasks move out of the scheduling state. It is defined as:

$$tr_{ji} = \max_{\forall r, H_i \in \beta_r} \{TR_{jir}\} \tag{5}$$

$$TR_{jir} = \begin{cases} te_r, & if \quad r < j \\ hr_i, & if \quad r \geq j \end{cases} \tag{6}$$

### 3.1.2 Fitness Function

The most significant factor in our fitness function is the makespan, $\omega$, which is defined as the time from the start of the first task to the end of the last task. It is calculated by evaluating the PACE function $p_{ext}()_j$ for each host allocation and combining this with the end time of the task:

$$\omega = \max_{0 \leq j \leq n-1} \{te_j\} \tag{7}$$

Equation 7 represents the latest completion time of all the tasks. The aim of the process is to minimise this function with respect to the schedule - which

bear similarities to orthogonal rectangular packing. It is in fact an extension of the classical multi-processing (MS) scheduler problem, with the difference that the host pool is diverse and dynamic. The MS problem is known be intractable in the general case [5] and NP-hard for the case $m \geq 5$.

While schedules can be evaluated on makespan alone, in a real-time system, idle time must also be considered to ensure that unnecessary space is removed from the schedule queue. Titan employs an additional function (8) that penalises early idle time more harshly than later idle time, using the following expression:

$$\Gamma_k = \omega_k \sum_i \left( \frac{\Delta T_i^{\text{idle}}}{\omega_k^2} (2\omega_k - 2T_i^{\text{idle}} - \Delta T_i^{\text{idle}}) \right) \tag{8}$$

where $T_i^{\text{idle}}$ is the start of the idle time, and $\Delta T_i^{\text{idle}}$ is the size of idle time space which can be calculated as the difference between the end time of task and the start time of the next task on the same host.

This idle weighting function is therefore a decrease from 100% weighting at $(T_i = 0, \Delta T_i = \omega)$ to 0% weighting at $(T_i = w, \Delta T_i = 0)$. The reason for penalising early idle time is that late idle time has less impact on the final solution as:

1. the algorithm has more time to eliminate the pocket.
2. new tasks may appear that fill the gap.

A linear contract penalty is also evaluated which can be used to ensure that the deadline time for a task, $d_j$, is greater than the expected end-time of the task.

$$\theta_k = \sum_{j=0}^{n-1} \max \left\{ (te_j - d_j), 0 \right\} \tag{9}$$

The three fitness functions are combined with preset weights to formulate an overall fitness function (10). Using these weights, it is possible to change the behaviour of the scheduler *on-the-fly*, allowing the scheduler to prioritise on deadline or compress on makespan for example.

$$f_k = \frac{(W^{\text{i}} * \Gamma_k) + (W^{\text{m}} * \omega_k) + (W^{\text{c}} * \theta_k)}{W^{\text{i}} + W^{\text{m}} + W^{\text{c}}} \tag{10}$$

The fitness value is normalised to a cost function using a dynamic scaling technique, and then used as the basis for the selection function. Schedules with a higher fitness value are more likely to be selected than a schedule with a fitness approaching 0.

### 3.1.3 Crossover / Mutation

The selected schedules are subject to two algorithmic operators to create a new solution set, namely 'crossover' and 'mutation'. The purpose of crossover is to maintain the qualities of a solution, whilst exploring a new region of the problem space. Mutation adds diversity to the solution set.

As the schedules are represented as a two dimensional string that consist of a task ordering part (which is q-ary coded) and a binary constituent (which is

binary coded), the genetic operators have to be different for the task order and host mappings. Each operator acts on a pair of schedules, and is applied to the entire solution set.

$$S \underset{crossover}{\longmapsto} S' \underset{mutation}{\longmapsto} S'' \tag{11}$$

The task crossover process consists of merging a portion of the task order with the remaining portion of a second task order, and then re-ordering to eliminate illegal orders. The mutation process involves swapping random task orders. For example, two task orders can be represented as follows:

$$\begin{aligned} \ell_A &= [T_0, T_3, T_5, T_2, T_6, T_4, T_1] \\ \ell_B &= [T_5, T_4, T_3, T_2, T_1, T_3, T_6] \end{aligned}$$

$\ell_A$ and $\ell_B$ can be combined at a random point to produce a new schedule task list $\ell'$, where $\ell_{A.0}$ denotes the $0^{th}$ element of $\ell_A$. Task duplications are then removed by undoing the crossover operation for the repeated task, as illustrated in the following expression where $(T_3)$ denotes a task duplication which is resolved by setting the first $(T_3)$ to the value of $\ell_{B.1}$

$$\begin{aligned} \ell' &= [\ell_{A.0}, \ell_{A.1}, \ell_{A.2}, \ell_{B.3}, \ell_{B.4}, \ell_{B.5}, \ell_{B.6}] \\ &= [T_0, (T_3), T_5, T_2, T_1, (T_3), T_6] \\ &= [T_0, T_4, T_5, T_2, T_1, T_3, T_6] \end{aligned}$$

Random changes are then applied to $\ell'$ to give the mutated representation $\ell''$, illustrated with the bracketed tasks in the following expression;

$$\begin{aligned} \ell'' &= [T_0, T_4, T_5, T_2, (T_1), T_3, (T_6)] \\ &= [T_0, T_4, T_5, T_2, T_6, T_3, T_1] \end{aligned}$$

The host map crossover process also merges two schedules together, using a simpler binary copy. The mutation process randomly toggles bits in the hostmap, while ensuring that topology rules have not been broken (such as trying to run the task on zero hosts for example). Expression 12 represented a full schedule of seven tasks and five hosts.

$$\begin{aligned} \ell'' &= [T_0, T_4, T_5, T_2, T_6, T_3, T_1] \\ M'' &= [[1,1,0,1,1], [1,1,0,0,0], [0,0,1,0,0], \\ &\quad [0,0,0,1,0], [0,0,0,0,1], [1,1,0,0,1], \\ &\quad [1,1,1,1,1]] \end{aligned} \tag{12}$$

Figure 2 depicts a visualisation of this representation. The tasks are moved as close to the bottom of the schedule as possible, which can result in a task that has no host dependencies being run prior to a task earlier in the task order. This is illustrated with $T_5$ being scheduled before $T_4$ despite being further on in the task order.

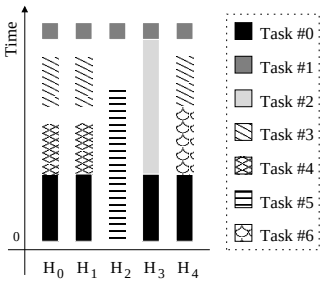


Figure 2: Visualisation of the schedule given in expression 12. Task $T_5$ is run earlier than $T_4$ as it requires $H_2$ which is not used by either $T_0$ or $T_4$.

# 4 Implementation

The research presented in the previous section has been developed into a working system for scheduling sequential and parallel tasks over a heterogeneous network of resources. The algorithm introduced in section 3 forms the centre point of the workload managers and are responsible for selecting, creating and evaluating new schedules. Connection to the PACE system is achieved by a remote evaluation library, with a performance cache to store timings for subsequent re-use.

## 4.1 Workload Management Process

The core data type is the *Task*. Tasks are submitted to the workload manager from the distribution brokers, and enter a pool of Task objects. The scheduler maintains continuous visibility of this pool and applies the strategies described earlier to form solution sets. The scheduler also accepts host information from a second pool which is updated by an on-line host gathering and statistics module. Currently, this information is gained from UNIX '`uptime`' and '`sar`' commands, although a small agent is being developed to remove the operating system dependence that this approach involves.

At each iteration, the best solution is compared with an overall fittest solution. This allows the scheduler to maintain a copy of the best schedule found to date, whilst allowing the process to explore other regions of the problem space. If a significant change occurs in either the host or task pools, then this schedule is deemed 'stale' and replaced by the fittest schedule in the current generation.

Periodically, the scheduler examines the bottom of the schedule and if there are any tasks to run, will move them into the run queue and remove them from the scheduling queue. Figure 3 provides a summary view of this system.

## 4.2 Evaluation

Figure 4 illustrates the best schedules of three scheduling approaches for 9500 generations. As expected, the heuristic methods offer a significant improvement over a random search. The first approach acts on host mappings only, resulting in rapid reduction in makespan before settling on a solution (not necessarily global). The second approach acts on both the host mappings and the task orders to reduce the makespan further and in less generations. This process is

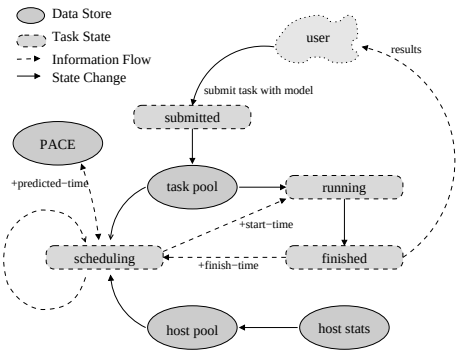


Figure 3: Flowchart diagram of the system implementation.

computationally expensive however, as the crossover and mutation stages for task re-ordering are significantly more complicated than the operators for the binary host mappings.

In figure 5 the weighting function of (10) is demonstrated with a static set of 32 tasks, with 16 tasks having a synthetic deadline imposed after approximately 10000 generations. For the purposes of this experiment, *generation count* is used as 'time'. The initial rise of fitness is indicative of the algorithm rapidly moving away from the random data set. At this stage, all the tasks are within their QoS restrictions and so makespan is the dominant variable for organisation. At generation 10000, the deadline metrics start to effect the schedule queue and the manager attempts to compensate. This has an immediate effect on the global fitness. In this experiment tasks are not removed from the queue and therefore the deadline metric becomes progressively dominant. After 12500 generations, the schedule is deadline-driven as opposed to makespan-driven, demonstrating how multiple parameters can be considered dynamically.

# 5 Conclusions and Further Work

The work presented in this paper is concerned with improving the task allocation process in grid environments using a multi-tiered framework based on Globus providers, distribution brokers and workload managers. This paper has focused primarily on the iterative heuristic algorithms and performance prediction techniques that have been developed for the workload managers.

A test suite has been implemented to tune the algorithm parameters and the PACE models. The results demonstrate that the scheduler converges quickly to a suitable schedule pattern, and that it balances the three functions of idle time, make span and the quality-of-service metric deadline time.

Further work will examine the relationship between low-level scheduling and the middle tier of workload management and top tier of wide area management. It is envisaged that this framework will develop to provide complimentary services to existing grid infrastructures (i.e. Globus) using performance prediction and service brokers to meet QoS demands for grid-aware applications.

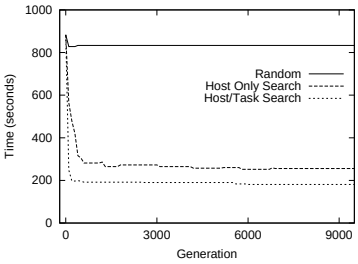


Figure 4: The fitness function applied to a schedule of 8 hosts and 32 tasks. The makespan has been reduced by a factor of four for the initial random allocations in under 1000 generations. Fast convergence is indicative of a genetic algorithm, although the solution may not be optimal.

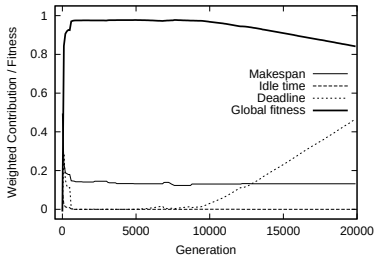


Figure 5: After an initial burst of activity, the contributing weight of makespan remains dominant over idle time and deadline over-run, until deadlines fail at generation 10000. The deadline component steadily increases, with an accompanying reduction in global fitness.

## Acknowledgements

This work is sponsored in part by grants from the NASA AMES Research Center (administrated by USARDSG, contract no. N68171-01-C-9012) and the EPSRC (contract no. GR/R47424/01). The authors would also like to express their gratitude to Darren Kerbyson, John Harper and Stuart Perry for their contribution to this work.

## References

[1] A. Abraham, R. Buyya, and B. Nath. Nature's heuristics for scheduling jobs on computational grids. *Proc. of 8th IEEE Int. Conf. on Advanced Computing and Communications*, 2000.

[2] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. *Proc. of Supercomputing*, 1996.

[3] R. Buyya, D. Abramson, and J. Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. *Proc. of 4th Int. Conf. on High Performance Computing*, 2000.

[4] J. Cao, D. Kerbyson, and G. Nudd. High performance service discovery in large-scale multi-agent and mobile-agent systems. *Int. J. of Software Engineering and Knowledge Engineering, Special Issue on Multi-Agent Systems and Mobile Agents*, 11(5):621–641, 2001.

[5] J. Du and J. Leung. Complexity of scheduling parallel task systems. *SIAM J. on Discrete Mathematics*, November 1989.

[6] D. Dumitrescu, B. Lazzerini, L. Jain, and A. Dumitrescu. *Evolutionary Computation*. CRC Press, 2000.

[7] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. of Supercomputer Applications*, 11(2):115–128, 1997.

[8] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organisations. *Intl J. of Supercomputing Applications*, 2001.

[9] G. Nudd, D. Kerbyson, E. Papaefstathiou, S. Perry, J. Harper, and D. Wilcox. Pace : A toolset for the performance prediction of parallel and distributed systems. *Int. J. of High Performance Computing Applications, Special Issues on Performance Modelling*, 14(3):228–251, 2000.

[10] A. Takefusa, S. Matsuoka, H. Nakada, K. Aida, and U. Nagashima. Overview of a performance evaluation system for global computing scheduling algorithms. *Proc. of 8th IEEE Int. Symp. on High Performance Distributed Computing*, pages 97–104, 1999.

[11] R. Wolski, N. Spring, and J. Haye. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computing Systems*, 1999.

[12] A. Zomaya and Y. Teh. Observations on using genetic algorithms for dynamic load-balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):899–911, 2001.

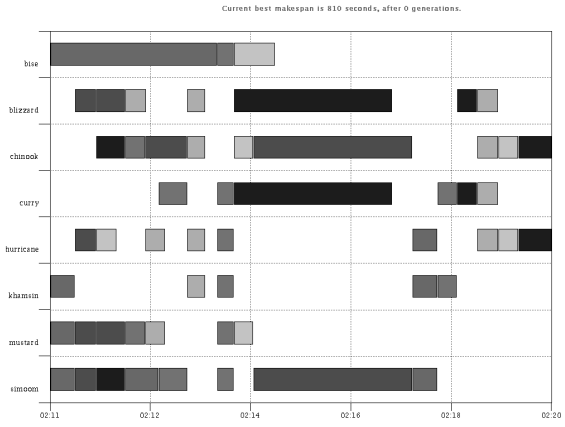


Figure 6: Typical screen shot of the scheduler before the workload manager has reorganised the tasks. The makespan at this time is between eight and nine minutes.

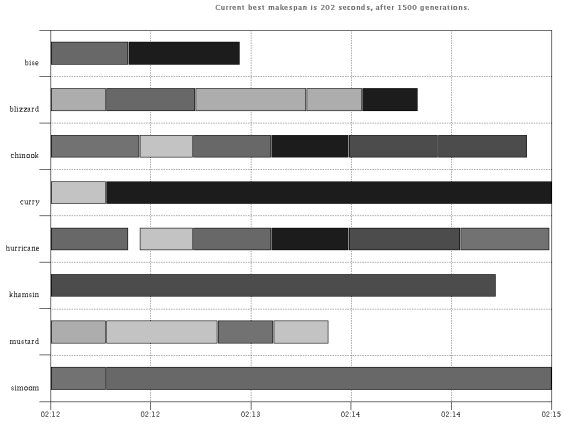


Figure 7: After 1500 iterations, the schedule has been condensed to just over three minutes.