Theory and Operation of the Warwick Multiprocessor Scheduling (MS) System

S. C. Perry and J. S. Harper and D. J. Kerbyson and G. R. Nudd

High Performance Systems Group
Department of Computer Science
University of Warwick
Coventry, UK

Abstract

This paper is concerned with the application of *performance prediction* techniques to the optimisation of parallel systems, and, in particular, the use of these techniques *on-the-fly* for optimising performance at run-time. In contrast to other performance tools, performance prediction results are made available very rapidly, which allows their use in real-time environments. When applied to program optimisation, this allows consideration of run-time variables such as input data and resource availability that are not, in general, available during the traditional (ahead-of-time) performance tuning stage.

The main contribution of this work is the application of predictive performance data to the scheduling of a number of parallel tasks across a large heterogeneous distributed computing system. This is achieved through use of just-in-time performance prediction coupled with iterative heuristic algorithms for optimisation of the meta-schedule.

The paper describes the main theoretical considerations for development of such a scheduling system, and then describes a prototype implementation, the *MS* scheduling system, together with some results obtained from this system when operated over a medium-sized (campus-wide) distributed computing network.

1 Introduction

The principle motivation for the use of any parallel system is the increased performance that can be achieved by distributing workload among a number of separate nodes. With computational systems, the production of a parallel application requires considerable extra work compared to the equivalent sequential program, and it is clearly very important to maximise the performance reward for this additional effort. Consequently, development of parallel applications has traditionally concentrated on finding the 'best' parallel algorithm for an identified target architecture, and subsequent tuning using performance tools such as application measurement/tracing. These tools are generally used before the application is run in earnest, and the 'optimisation' is necessarily specific to the particular target architecture.

However, the efficiency with which an application runs on a parallel machine also depends on other factors not foreseeable when the code was originally optimised. For example, it is possible to optimise cache re-use on a data-dependent basis[1,2], and the choice of algorithm can be made dependent on the input application parameters at run-time[3,4]. Furthermore, heterogeneous data necessitates load balancing within the application, and, at the system level, heterogeneous computing resources must also be scheduled and dynamically load balanced. These issues become even more challenging when one considers the trend towards large distributed computing systems, often built from a range of commodity components, which provide a heterogeneous and constantly changing set of computing resources.

Optimisation techniques used when a program is about to be executed (hereafter called *dynamic* performance optimisation methods) can be conveniently divided into two categories:

- *Single-program* issues are concerned with the optimisation of a single application running on a parallel system. For example, the size and organisation of local memory and the efficiency of the network and message passing interface are 'hardware factors' influencing performance at this level. The mapping of data and subtasks contained within the parallel application to the underlying network architecture is the main 'software factor' influencing performance; this is itself dependent upon the number of processing nodes which have been allocated, and their relative network topology. The choice of algorithms and data-partitioning strategies is consequently very rich at this stage, and dynamic performance optimisation is concerned with selection of these parameters on a system- and data-dependent basis.
- *Multiprogramming* issues are concerned with optimising the performance of the parallel system as a whole. In particular, the problems of quantitative partitioning (i.e. the number of processing nodes assigned to each task) and qualitative partitioning (i.e. the particular subset of nodes and, for a heterogeneous system, their associated computing power and communication costs) are compromises between minimising the execution time of any particular task and maximising the overall efficiency of the parallel system. The search space for this multiple parameter optimisation problem is extremely large, and is not fully defined until run-time.

In contrast to the dynamic optimisation techniques cited above[1–4], this paper will concentrate on the second category, and, in particular, the problem of dynamically optimising the schedule of a large number of parallel tasks, to be run on a heterogeneous computing network.

Theoretical considerations applicable to this problem will now be outlined.

1.1 Intractability

Consider for the moment the simpler case of a homogeneous computer system, consisting of a number of identical processing nodes for which communication costs are similar between all nodes, then the above problem can be stated formally as follows.

Problem 1 *A schedule for a set of parallel tasks* $\{T_0, T_1, \ldots, T_{n-1}\}$ *which are to be run on a network of identical processing nodes* $\{P_0, P_1, \ldots, P_{m-1}\}$ *is defined by the allocation to each task* T_j *a set of nodes* $\beta_j \in \{P_0, \ldots, P_{m-1}\}$ *and a start time* τ_j *at which the allocated nodes all begin to execute the task in unison. Because the processing nodes are identical, the execution time for each task is simply a function,* $t_x()_j$*, of the number of nodes allocated to it, and it is presumed that this function is known in advance. The system uses Run-To-Completion (RTC) scheduling, hence the tasks are not permitted to overlap. The makespan,* w*, for a particular schedule is defined as:*

$$w = \max_{0 \le j \le n-1} \{\tau_j + t_x(||\beta_j||)_j\}\,, \qquad (1)$$

which is the latest completion time of any task. The goal is to minimise this function with respect to the schedule.

Problem 1 is called the Multiprocessor Scheduling (MS) problem.

The intractability of MS has been studied[5], and it is found that although pseudo-polynomial time algorithms exist for the cases of $m = 2$ and $m = 3$, MS is NP-hard for the general case $m > 4$. Approximate algorithms have been proposed[6,7]; these have a worst case bound of $w \le 2w^\star$, where $w^\star$ is the makespan of the optimal schedule. Although these algorithms are a useful fall-back, for high performance systems a factor of two is unacceptable in most cases, and one would hope to achieve much higher levels of optimisation.

In order to find (near) optimal solutions to this combinatorial optimisation problem, the approach taken in this work is to find good schedules through use of iterative heuristic methods. Two heuristic techniques have been studied: Simulated Annealing (SA)[8,9] and a Genetic Algorithm (GA)[10,11]. These methods proceed through use of a ‘fitness function’ which assigns a quality value to any particular solution (in this case corresponding to a schedule for the task set). The algorithms are concerned only with the maximisation of this value. Details of these methods' implementation are given in section 2.

1.2 Heterogeneous combinatorial considerations

As mentioned above, the function $t_x()$, which provides a program's execution time for a given allocation of processing nodes, β (hereafter referred to as a *performance scenario*), is assumed to be known in advance. For an otherwise unloaded homogeneous parallel system, where the execution times are not data-dependent, profile data could conceivably be used for this purpose (although for large one-shot applications this would somewhat defeat the object of the optimisation). The nodes are identical, so, for example, with a 60 node system there exists exactly 60 different performance scenarios per task (although if the optimum speedup occurs at a lower number of nodes then the later scenarios will not be used).

However, for the more realistic case of a heterogeneous system, combinatorial explosion generally precludes prior knowledge of the execution time. In the extreme case, the number of performance scenarios for m entirely *different* processing nodes is $\sum_{k=1}^{m} k!$, which for $m = 60$ is of the order 10^{81}. Clearly, for any non-trivial heterogeneous system it will not be possible to obtain the function $t_x()$ in advance of the scheduling stage. To overcome this difficulty, we propose to use data from a *performance prediction* system, calculated in real time as the scheduling program requires it. This just-in-time approach effectively provides $t_x(\beta)$ for the heuristic methods for heterogeneous systems. The implementation is described further in section 5.2.

1.3 Performance prediction

This work will concentrate entirely on performance prediction as the method of providing $t_x(\beta)$, either in advance for homogeneous situations or in real time for heterogeneous systems. Performance prediction is the technique of estimating values for various aspects of how an application program will execute on a computer system, given certain well-defined descriptions of both program and system. The performance prediction toolset used here, PACE, operates by characterising the application in terms of its principle operations (representing computation costs) and its parallelisation strategy (which dictates the communication pattern). These are then combined with models of the system environment to produce a prediction for the execution time. The key to this strategy is that the separation of program and system models allows predictions for heterogeneous computing environments.

Predictions are made in a fraction of a second, and for applications whose internal structures have a low level of data-dependence is generally accurate to within 10 % of the measured time. However, it is important to note that for run-time optimisation purposes the accuracy of the prediction is not an overriding concern; any information from detailed trace data down to a basic measure of program complexity is always useful – it is better than no information at all. A complete description of the PACE system can be found in [12].

2 Solution of MS using Heuristic Methods

A number of standard texts describe the operation of the SA and GA techniques; the following sections detail only issues relating to solution of the MS problem using these methods.

2.1 Coding scheme and fitness function

Both techniques require a coding scheme which can represent all legitimate solutions to the optimisation problem. For both algorithms, any possible solution is uniquely represented by a particular string, S_i, and strings are manipulated in various ways until the algorithm converges upon an optimal solution. In order for this manipulation to proceed in the correct direction, a method of prescribing a quality value (or *fitness*) to each solution string is also required. The algorithm for providing this value is called the fitness function $f_v(S_i)$.

The fitness values of solutions to the MS problem are readily obtained – the solutions which represent the schedule with the least makespan are the most desirable, and vice-versa. The processing node set for each task, and the order in which the tasks are executed, are encoded in each solution string, and the execution times for each task (given the set of nodes allocated) are obtained from the prediction system. It is therefore straightforward to calculate the makespan of the schedule represented by any solution string S_i. This number may be converted from a cost function (f_c) to a value function (f_v) by multiplying all the makespans by -1 and normalising on the interval $0 \leq f_v(S_i) \leq 1$.

The coding scheme we have developed for this problem consists of 2 parts:

- An ordering part, which specifies the order in which the tasks are to be executed. This part of the string is q-ary coded where $q = n$.
- A mapping part, which specifies the allocation of processing nodes to each task. This part of the string is binary coded, consisting of $n \times m$ bits spec-

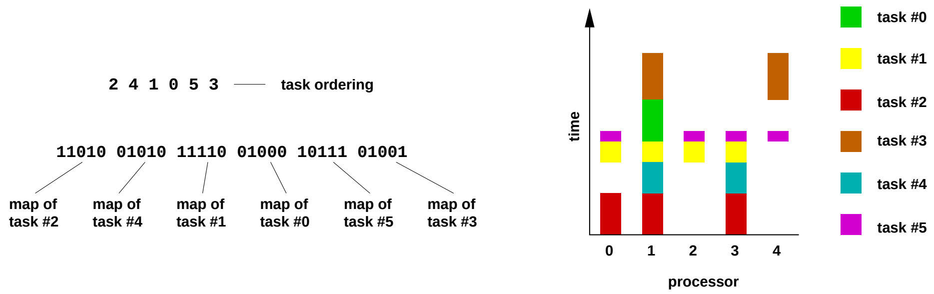

Fig. 1. An example solution string (LHS) and Gantt chart representing its schedule (RHS). The two parts of the solution string are shown separately, with the ordering part above and the mapping part below. Note that the execution times of the various tasks are a function of the processing node set allocated and are provided by the performance prediction system. This data is associated with the task object and is never used directly by the optimisation algorithm, but only through the fitness function f_v.

ifying whether or not a particular node is assigned to a particular task.

The ordering of the task-allocation sections in the mapping part of the string is commensurate with the task order. A short example of this type of solution string and its associated schedule are shown in Fig. 1. The ordering part of this string is always guaranteed to be legitimate by the various manipulation functions used in the heuristic algorithms. However the same is not true of the mapping part, and if the fitness function encounters a string with a task that has no processing nodes allocated it will randomly assign one. Further manipulation may be required when using the system in a heterogeneous environment, as described in section 4.

From Fig. 1 it is clear that the task of parallel schedule optimisation is a 'packing problem' where the goal is to fit the programs together as tightly as possible. Examples of a simple schedule before and after application of a heuristic algorithm are shown in Fig. 2.

2.2 Genetic algorithm-based optimisation

A GA-based optimisation program, using the coding scheme described above, was developed for testing with simulated (homogeneous) performance data and the MS problem fitness function. The code was based upon the ideas developed in [11], using a population size of 60 and stochastic remainder selection. Specialised crossover and mutation functions were developed for use with the two-part coding scheme. The crossover function first splices the two ordering strings at a random location, and then re-orders the offspring to produce legitimate solutions. The mapping parts are crossed over by first re-ordering them to be consistent with the new task order, and then performing

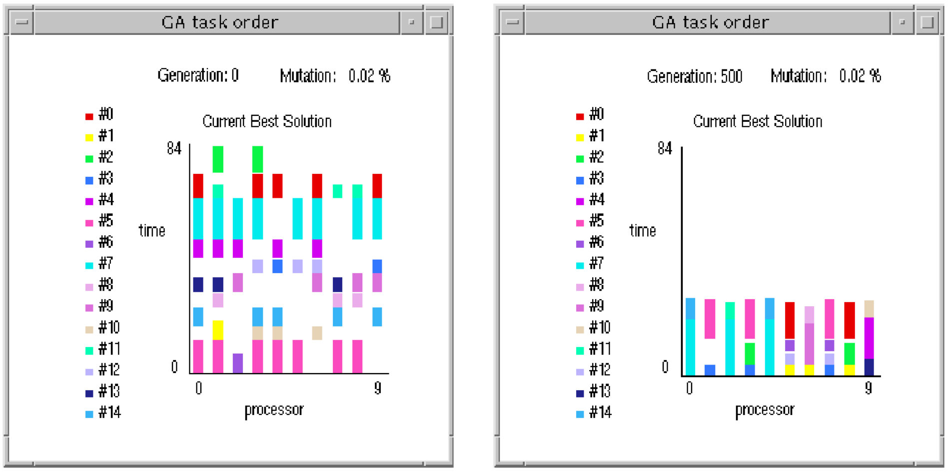

Fig. 2. Example schedules for 15 programs running on 10 processors, both before optimisation (left panel) and after 500 iterations of the optimising (GA) algorithm (right panel). The different shaded rectangles represent the programs indexed on the left.

a single-point (binary) crossover. The motivation for the re-ordering is to preserve the node mapping associated with a particular task from one generation to the next. The mutation stage is also two-part, with a switching operator randomly applied to the ordering parts, and a random bit-flip applied to the mapping parts.

2.3 Simulated annealing-based optimisation

The SA-based optimisation program used the same coding scheme and fitness function described above, and was a modification of the travelling code given in Numerical Recipes[13, chap. 7]. The program again uses a two-part rearrangement function, with path transport or reversal (performed according to an annealing schedule derived from the metropolis algorithm) occurring separately for the ordering and mapping parts, and the mapping part reordered in between.

3 Performance of the heuristics

In order to conveniently test the performance of the algorithms with different problem sizes, tasks were simulated from a simple homogeneous parallel computation model, viz,

$$t_x(i) = \frac{C_{cpu}}{i} + C_{com} \times i, \qquad i = 1, \ldots, m. \tag{2}$$

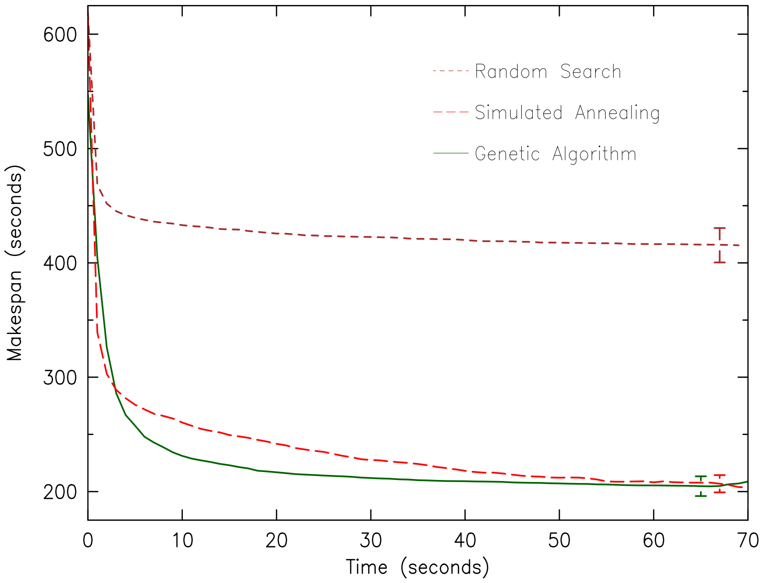

Fig. 3. Comparison of the (cumulative) best schedules found by the different heuristic approaches for a set of 20 simulated tasks running on a 16 node system. The lines are the average of 100 runs of each search method; the error bars show the width of 1 standard deviation from the mean at convergence.

Hence from the derivative of Eq. 2 it was straightforward to create task sets with specified distributions of sizes and optimum processing node numbers.

Examples of the convergence times for a simple scheduling problem are presented in Fig. 3. The figure shows the relative performance of the heuristic algorithms and a random search algorithm (using peak speedup node allocations for each task). The intractability of the MS problem precludes comparison of the results with an optimal schedule, but we assume the results at convergence to be near-optimal. As expected, the heuristic methods offer an appreciable improvement over the random search (which demonstrates the vastness of the search space even for a relatively simple problem). Although the solution quality for the two heuristic algorithms is comparable at convergence, in general GA provides higher-quality solutions at shorter times than SA. The reason for this is believed to be that the operation of SA is based upon patient movement towards a minimum following an annealing schedule, and does not lend itself to early termination. Conversely, GA is based upon the evolution of a set (population) of solution strings, which not only finds minima more quickly, but also adapts easily to changes in the problem (e.g. the addition or completion of a task, or a change in the available hardware resources), because any current solution set will serve as good quality starting points in a slightly changed situation. Hence hence for realistic, real-time scheduling environments, where the pool of tasks and computing resources is constantly changing, the ratio of the convergence times for GA vs. SA (shown in Fig. 3) is greatly exaggerated, with SA having to complete its annealing schedule for each new situation, whereas GA can quickly evolve from the previous solution, and hence start much further along the convergence curve.

3.1 Real-time extension of the MS problem

The simulations described above were concerned with the solution of the MS problem as defined in Prob. 1, with all the tasks and their associated performance data known in advance. Consider now a slightly more realistic case, where new tasks may be added to the queue after the first tasks have started. Hence the MS problem becomes dynamic; tasks are added and removed (when they are complete) from the scheduling problem *in real time*. As mentioned earlier, the evolutionary basis of GA is particularly well suited to this situation, and this was the only heuristic used for the following simulations.

The simulations work by first allowing a “warm-up” period for the GA with a set of tasks, and then removing the tasks which have been scheduled for execution (i.e. the ones at the beginning of the Gantt chart). This tends to leave a jagged ‘base’ which forms the basis of the next packing problem. When a task completes, the algorithm is stopped and interrogated for its best solution, and any tasks at the bottom of the schedule begin execution. Tasks are continuously added so as to keep the number of pre-scheduled tasks constant.

For the case where the queue initially contains all the tasks to be executed, minimisation of the makespan provides the most efficient schedule. However, for the real time case now considered, one should also take into account the nature of the idle time in the schedule. Idle time at the bottom of the schedule is particularly undesirable, because this is the processing time which will be wasted first, and is least likely to be recovered by further iteration of the GA or when more tasks are added. For this reason we propose and use a modified cost function:

$$f_c = w \times \left\{ 1 + \sum_{\text{nodes}} \delta\left[t_j^{\text{idle}}(0) \right] \right\} \tag{3}$$

Where $t_j^{\text{idle}}(t)$ is the amount of idle time on node j starting at time t, with time beginning at zero. Within this regime solutions with idle time at the beginning of the schedule on any of the processors are penalised with a higher cost function, in proportion to the overall makespan. Equation 3 will be referred to as the Extended MS (XMS) cost function.

Shown in table 1 are the task execution rates for the standard and extended cost functions used with the GA, compared with the results from a simpler first-come-first-served (FIFO) scheduler. GA-XMS gives a 20 % improvement in the task execution rate over GA-MS (using the normal MS quality func-

$< \tau_j >$	16	32	64
FIFO	242	110	58
GA-MS	821	357	182
GA-XMS	963	428	222

Table 1
Rates of execution (hr^{-1}) for simulated tasks of average size τ_j. The row marked "FIFO" shows results for a 'first-come-first-served' schedule. The rows prefixed "GA" show results for GA optimised schedules with different quality functions, as described in the text.

tion, i.e. makespan alone). However, separate simulations have shown that the makespan is generally 10 % worse for GA-XMS at convergence. This discrepancy demonstrates the penalty incurred for lack of knowledge of the entire task queue before scheduling begins.

In either case, the heuristic algorithms give an increase in the execution rate of 3-4 times over the standard queueing method – this is a direct result of knowledge of the execution time function, $t_x(\beta)$, provided by performance prediction.

4 An Implementation: The MS Scheduling System

The ideas presented in the previous sections have been developed into a working system for scheduling sequential and parallel tasks over a heterogeneous distributed computing network. The goal of this project, called the MS scheduling system, is to investigate and demonstrate the role of performance prediction in the optimisation of large distributed systems. We hope that this work is complementary to the many other distributed computing projects in operation at the moment [14,15]. Our prototype system addresses the issues of intractability, heterogeneous systems, dynamic and evolving computing resources, and performance prediction. We do not attempt to duplicate or re-invent previous or ongoing work, hence our prototype system does not address issues such as security, network operating systems, resource discovery/identification etc. The state-of-the-art in these and other areas are covered in [15].

An overview of the design and implementation of the MS system is given below.

4.1 System organisation

The MS scheduling system has at its heart a GA scheduling program to provide good-quality solutions to the scheduling problem in real time, as described above. However, a number of other programs (daemons) and design features are necessary in order to produce a working system.

The main goal of the implementation was to make the 'state' of the system both visible and transcendent. To this end, MS makes extensive use of the filesystem (which should facilitate its extension to larger distributed computing environments such as GLOBUS), and a collection of various directories and database files represent its complete state at any particular instant in time.

The most important type of file is a job-file. After a job [1] has been submitted, it undergoes a series of state changes, with each state representing a particular stage in its lifetime. In reality, this corresponds to a job-file (a database which uniquely represents the job, and contains various job parameters and other information) being moved through a directory hierarchy. At most changes of state, additional information is added to the job-file. This hierarchy is shown schematically on the LHS of Fig. 4.

Other important files are the hosts and host-stats databases, which provide MS with real-time information on the state of the available computing resources. These databases will be described further in section 5.1.5. The various job states (directories) will now be described individually.

- Submitted. This is the entry point to the scheduling system, where the user presents jobs to be executed. At this point the job-file contains the basic information required to run the job, including the location of the executable(s), the input data, and various information pertaining to the performance data or model.
- Queued. When the job-file is moved into this directory it is given a unique job identification number (JID). It then awaits the attention of the schedule optimiser.
- Scheduling. Once the schedule optimiser becomes aware of a job it is moved into this directory for the duration of the optimisation procedure.
- Runnable. At the time that the schedule optimiser decides to run a job it moves the job-file into this directory where another daemon is responsible for running it on the system. The schedule optimiser specifies the computing

[1] A task is called a "job" within the context of the scheduling system (by analogy with batch queue systems), and we will use the two terms interchangeably here. Likewise, processor nodes are referred to as "hosts", as they generally correspond to individual workstations in distributed systems.

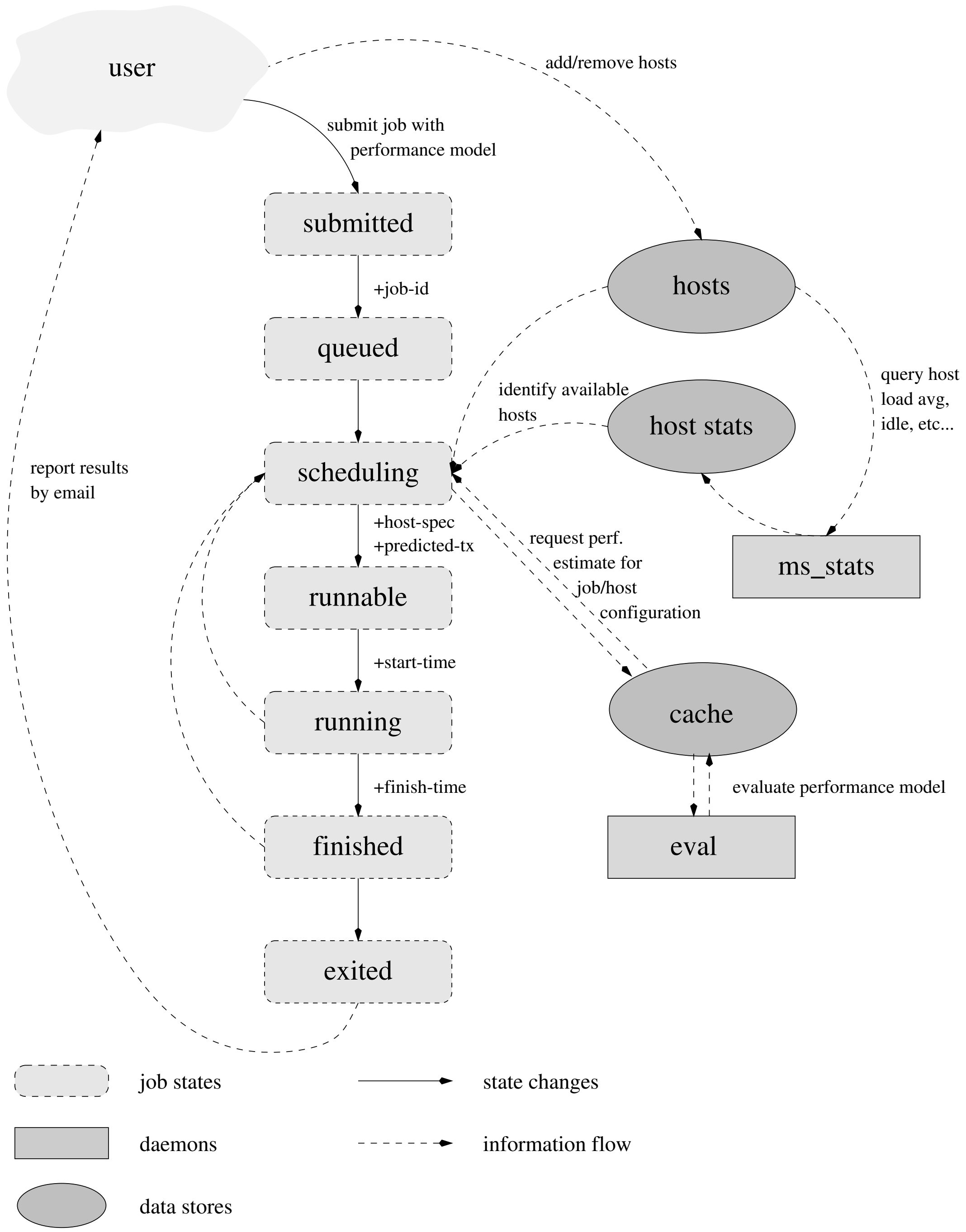

Fig. 4. Flowchart diagram of the MS scheduling system. A full explanation is given in the text. Note that not all MS daemons are shown in this diagram.

resources on which the job is to be run, and adds information about the predicted execution time to the job-file.

- Running. Once the job has begun to be executed on the system its job-file is moved to this directory, where the start-time is added (the start-time and predicted execution times of the currently running jobs are required by the schedule optimiser, to provide the 'base' for the packing problem).

- Finished. Once the job process has completed it is moved to this directory, where it waits for the schedule optimiser to move it to the next directory. In this way, the optimiser program is able to keep track of when jobs finish relative to their predicted finish time.
- Exited. When the job-file reaches this directory it signifies that the job is completed as far as the system is concerned. The JID number is recycled, and the user is sent notification that the job has been completed.

The maintenance of these directories and databases is the responsibility of various daemons, as described in the section 5.1.

5 The MS toolset

The various programs and libraries which make up the MS system will now be described.

5.1 Daemons

A number of background processes, or *daemons*, are used to implement the state machine described above. All interprocess communication is performed via the file-system, with a single daemon being designated the owner of each particular queue directory. Ownership gives the right to edit the job files in a directory. Daemons without ownership of a directory are only allowed to read or add jobs in the queue.

5.1.1 ms-init

The ms-init daemon monitors the submission queue. As job files are added to the directory by a user, the init daemon allocates an unused job id, and moves the job into the queueing directory, ready for the scheduler.

5.1.2 ms-sched

The scheduler daemon encompasses the ideas of sections 1-3 of this paper. It uses the GA heuristic to search for optimal solutions to the schedule problem at hand, and interrogates the GA when there are free resources available, in order to submit jobs for execution.

The daemon operates by periodically scanning the “queued” queue for new entries; when these are found the job is passed to the GA for addition to its optimisation pool, and the job is moved to the "schedule" queue. The scheduler also scans the "running" queue, in order to find the base for the optimisation (packing) problem, as described earlier. It also monitors the "finished" queue, in order to ascertain if the predicted execution times were correct, and modify its schedule base if this is not the case. When the scheduler finds a job in the "finished" queue it is moved to the "exited" queue, in order to let the system know that the scheduler is aware of the job's completion, and that it is no longer required.

When the scheduler submits a job to the "runnable" directory it adds the specification for the hosts that the job is to be run on. It also adds the predicted execution time for the job, which it requires once the job has started running, as described above.

5.1.3 ms-run

After the scheduler, the ms-run daemon is perhaps the most complex. It is responsible for executing the program associated with a job on a specified list of hosts. Currently, only MPI conforming programs are supported, but it is envisaged that this will be extended as the system matures (for example to support PVM-based applications). Various fields of the job data structure are used to control the execution of the program, these include options to control the arguments to the process, and how the input and output streams of the process are treated.

Currently, the executable programs must be pre-compiled and available in all local filesystems. Heterogeneity is handled by allowing the filenames to be constructed on a host-by-host basis, and including local parameter substitutions, such as the name of the architecture. A framework is in place to allow executables to be compiled on demand from "packets" of source code, but this is yet to be completed.

The ms-run daemon scans the "runnable" queue, executing any jobs before moving them to the "running" queue. After they complete their execution, the exit status and time of completion are added to the job structure, before being moved to the "finished" queue.

If for some reason a job is unable to be run, it is returned to the "queueing" directory, ready to be rescheduled at a later date, unless this it has already been queued too many times, in which case it is rejected.

5.1.4 ms-reaper

After the scheduler has noted job completions, and moved them to the "exited" queue, the ms-reaper daemon reports the results back to the user (in the current implementation this is done via email). If no destination was specified for the program's standard output stream, the output of the program is included in the report. The report also includes information concerning the execution time of the program (and how accurate the performance model was), and which hosts the job was executed on. Finally the job file is deleted, and its job id is recycled.

The reaper daemon also monitors the "reject" queue. This is where the other daemons place jobs that are in some way incorrectly specified, or otherwise unable to be executed.

5.1.5 ms-stats

The ms-stats daemon is responsible for gathering statistics concerning the hosts on which tasks may be scheduled. The three statistics required are the uptime, load average, and idle time of each host. The uptime is the time since the system was booted, the idle time is the time since the user of that machine last gave any input (or infinity if there is no current user), and the load average is the number of processes in the system scheduler's run queue, amortised over a recent fixed period.

Every five minutes the ms-stats daemon queries each system in the database of known hosts for these three parameters. It uses standard UNIX utilities to gather the information: `finger` and `rup`. As the statistics are gathered, they are added to the database of host statistics.

5.2 Evaluation Library

As noted earlier, the search space is generally so large such that it is not possible to pre-calculate all required performance estimates ahead of time. Instead a demand-driven evaluation scheme is used, coupled with a cache of past evaluations. The motivation for the cache is that there is a large degree of repetition in the list of performance scenarios that the GA will require.

Although it would be straightforward to use other prediction methods, this implementation uses models created by the PACE environment. For each application that is submitted to the MS system there must be an associated performance model. PACE allows the textual performance descriptions to be compiled into a binary executable; invoking this binary with a list of parameter

```
$ ./AppParticles -i
pace? list
Nproc 1
N 500
pace? set Nproc 2
pace? set N 20000
pace? eval
2.19181e+08
pace? hrduse SunUltra1 SunUltra1
pace? eval
4.89571e+08
pace? set N 10000
pace? eval
1.22434e+08
```

Fig. 5. Example PACE session

definitions will evaluate the performance model for the specified configuration.

To avoid the startup overhead associated with evaluating a model, PACE provides an interactive interface, where multiple evaluations can be performed, specifying the parameters and hardware configurations for each separate evaluation. Figure 5 shows an example interactive session. The MS evaluation library creates a process running each performance model, then uses the session interface to invoke evaluations and read the results, communicating across a pair of sockets. This method allows the possibility of distributing evaluations across multiple hosts, although this hasn't been necessary as yet.

When requesting evaluations the scheduler translates from the vector of host names specifying which hosts the job would run on, to a vector of architecture names and cpu load averages. For example if a specified host is a Sun Ultra-1 workstation, the associated PACE hardware model is called `SunUltra1`, and its current load average is 0.3 (this data is collected by the statistics daemon, described in Section 5.1.5), the string `SunUltra1:hardware/CPU_LOAD=0` would be specified as the hardware model of that particular host. Repeating this process for all hosts that the job would run on gives the vector of hardware models that the evaluation requires.

Although evaluations complete relatively quickly (usually in the order of a few tenths of a second), this is still a less than ideal. For example, if the GA has a population size of 50, and there are 20 jobs being scheduled, then 1000 evaluations are required each generation. If each evaluation takes 0.01 seconds, then this is 10 seconds per generation. However, many of the evaluations requested by the genetic algorithm are likely to be exactly the same as those required by previous generations (due to the nature of the crossover and mutation operators).

To capitalise on this redundancy a cache of all previous evaluations has been added between the scheduler and the performance model. When a particular evaluation result is requested, the cache is searched (the cache uses a hash-table, so lookups are fast). If the result already exists, it is returned to the scheduler. Otherwise the performance model is called, and the result is added to the cache before being returned to the scheduler. The library also supports a secondary level of caching, using a database stored in the file system. The results of all previous evaluations of a particular model are recorded, along with the model parameters for each result. This has several benefits to the scheduler: firstly it is possible to stop and then restart scheduler processes without losing the evaluation history, and secondly similar jobs may be scheduled more than once, but the model is only evaluated the first time.

5.3 User Interface

Naturally, the end user of the scheduling system can not be expected to manipulate the job files themselves. To this end a graphical user interface has been developed, allowing all of the information contained within the system to be displayed and modified.

The first part of the interface is the browser. This allows all of the databases in the system to be displayed. These databases include the job queue directories, and the control databases, such as those containing the host data and statistics. A screenshot of the browser is shown in Figure 6.

The other part of the user interface is the scheduler front end. This displays the Gantt chart of the current schedule, and allows the various daemons to be controlled. Figure 7 shows a typical screenshot of the scheduler interface.

An alternative interface to the system is via the World Wide Web, through a CGI script. This interface allows much the same actions as the desktop interface, with the exception of the schedule Gantt chart (we are currently developing a Java applet to allow this).

Acknowledgments

This work is funded in part by U.K. government E.P.S.R.C. grant GR/L13025 and by DARPA contract N66001-97-C-8530 awarded under the Performance Technology Initiative administered by NOSC.

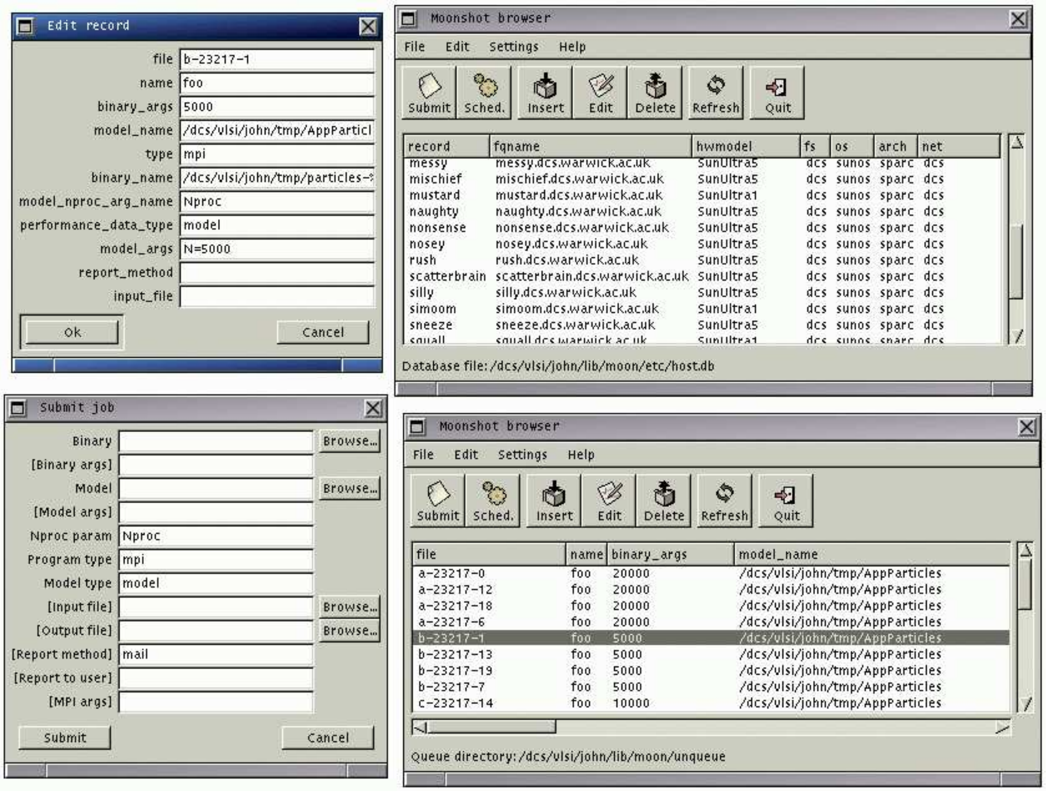

Fig. 6. MS browser

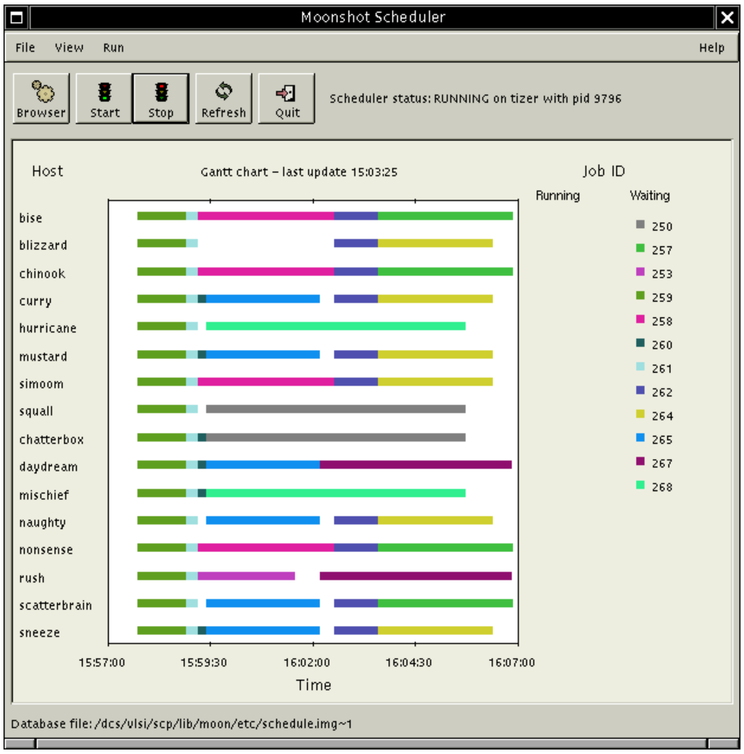

Fig. 7. MS schedule viewer

References

[1] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Efficient analytical modelling of multi-level set-associative caches. In *Proceedings of the International Conference HPCN Europe '99*, volume 1593 of *LNCS*, pages 473–482. Springer, 1999.

[2] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Analytical modeling of set-associative cache behavior. To appear in *IEEE Transactions on Computers*.

[3] D. J. Kerbyson, E. Papaefstathiou, and G. R. Nudd. Application execution steering using on-the-fly performance predition. In *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes in Computer Science*, pages 718–727, Amsterdam, April 1988. Springer.

[4] S. C. Perry, D. J. Kerbyson, E. Papaefstathiou, G. R. Nudd, and R. H. Grimwood. Performance optimization of financial option calculations. *Journal of Parallel Computing, special issue on Economics, Finance and Decision Making*, 26, 2000.

[5] J. Du and J. Leung. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, 2:473, November 1989.

[6] K. Belkhale and P. Banerjee. Approximate scheduling algorithms for the partitionable independent task scheduling problem. In *Proceedings of the 1990 International Conference of Parallel Processing*, volume I, page 72, August 1990.

[7] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms for scheduling parallelizable tasks. *ACM Performance Evaluation Review*, page 225, June 1992.

[8] S. Kirkpatrick, D. C. Gelatt, and M. P. Vecchi. Statistical mechanics algorithm for monte carlo optimization. *Physics Today*, pages 17–19, 1982.

[9] S Kirkpatrick, C D Gelatt, and M P Vecchi. Optimisation by simulated annealing. *Science*, 220:671–680, 1983.

[10] J. H. Holland. *Adaption in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1975.

[11] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Co., Inc., Reading, MA., 1989.

[12] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. Pace - a toolset for the performance prediction of parallel and distributed systems. *Accepted for publication in International Journal of High Performance and Scientific Applications*, 1999.

[13] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipies in C: The art of Scientific Programming.* Cambridge University Press, Cambridge, England, 1988.

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

[15] Ian Foster and Carl Kesselman, editors. *The GRID - Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, San Francisco, USA, 1999.