Artificial Intelligence Review (2005) 24:415–429
DOI 10.1007/s10462-005-9002-x

Framework for Task Scheduling in Heterogeneous Distributed Computing Using Genetic Algorithms

ANDREW J. PAGE* & THOMAS J. NAUGHTON
*Department of Computer Science, National University of Ireland, Maynooth, County Kildare, Ireland (*author for correspondence, e-mail: andrew.j.page@nuim.ie)*

Abstract. An algorithm has been developed to dynamically schedule heterogeneous tasks on heterogeneous processors in a distributed system. The scheduler operates in an environment with dynamically changing resources and adapts to variable system resources. It operates in a batch fashion and utilises a genetic algorithm to minimise the total execution time. We have compared our scheduler to six other schedulers, three batch-mode and three immediate-mode schedulers. Experiments show that the algorithm outperforms each of the others and can achieve near optimal efficiency, with up to 100,000 tasks being scheduled.

Keywords: distributed computing, genetic algorithms, task scheduling

1. Introduction

Distributed computing is a promising approach to meet the increasing computational requirements of scientific research. However, a number of issues arise that are not encountered in sequential processing that, if not properly handled, can nullify the benefits of parallelisation. We believe that task scheduling is the most important of these issues because inappropriate scheduling of tasks can fail to exploit the true potential of a distributed system due to excessive communication overhead or under-utilisation of resources, and can offset the gains from parallelisation. Thus it falls to one's scheduling strategy to produce schedules that efficiently utilise the resources of the distributed system and minimise the total execution time. The problem of scheduling heterogeneous tasks onto heterogeneous resources, otherwise known as the task allocation problem, is an NP-hard problem for the general case (Garey and Johnson, 1979).

Many heuristic algorithms exist for specific instances of the task scheduling problem, but are inefficient for a more general case (Kasahara and Narita, 1984). The use of Holland's genetic algorithms (GAs) (Holland, 1992) in scheduling, which apply evolutionary strategies to allow for the fast exploration of the search space of schedules,

allows good solutions to be found quickly, and for the scheduler to be applied to more general problems. Many researchers have investigated the use of GAs to schedule tasks in homogeneous (Hou et al., 1994; Zomaya and Teh, 2001) and heterogeneous (Maheswaran et al., 1999; Ahmad et al., 2001; Theys et al., 2001; Zomaya et al., 2001) multiprocessor systems with notable success.

Unfortunately, assumptions are often made which reduce the generality of these solutions, such that scheduling can be calculated offline in advance and cannot change (Hou et al., 1994; Ahmad et al., 2001; Theys et al., 2001; Zomaya et al., 2001), all communications times are known in advance (Hou et al., 1994; Ahmad et al., 2001; Theys et al., 2001; Zomaya et al., 2001), networks provide instantaneous message passing (Zomaya and Teh, 2001), and that all processors have equal capabilities and are dedicated to processing tasks from the scheduler (Kasahara and Narita, 1984; Hou et al., 1994; Siegel et al., 1996; Zomaya et al., 1998, 1999, 2001; Ahmad et al., 2001; Theys et al., 2001; Zomaya and Teh, 2001). These assumptions limit the generality of these scheduling strategies in real-world distributed systems. It would be more preferable to make no assumptions about the homogeneity of the processors, or about the availability of system resources.

In this paper, a scheduling strategy is presented which uses a GA to schedule heterogeneous tasks on to heterogeneous processors to minimise the total execution time. It operates dynamically, allowing for tasks to arrive for processing at arbitrary intervals, and considers variable system resources, which has not been considered by other dynamic GA schedulers.

Section 2 reviews related work and gives an overview of how a GA operates. Section 3 describes our scheduling algorithm. Section 4 presents the results of our performance experiments and we conclude in Section 5.

2. Related Work

There are many examples in the literature of artificial intelligence techniques being applied to task scheduling (Hou et al., 1994; Siegel et al., 1996; Zomaya et al., 1998, 1999, 2001; Maheswaran et al., 1999; Ahmad et al., 2001; Theys et al., 2001; Zomaya and Teh, 2001). Meta-heuristic search techniques such as GAs (Holland, 1992), tabu (Glover, 1986), and ant colony search (Colorni et al., 1992) are most

applicable to the task scheduling problem because we wish to quickly search for a near optimal schedule out of all possible schedules. Efficient solutions have resulted from the use of GAs in task scheduling algorithms (Hou et al., 1994; Siegel et al., 1996; Zomaya et al., 1998, 1999, 2001; Maheswaran et al., 1999; Ahmad et al., 2001; Theys et al., 2001).

A GA is a meta-heuristic search technique which allows for large solution spaces to be non-deterministically searched in polynomial time, by applying evolutionary techniques from nature (Holland, 1992). The GAs use historical information to exploit the best solutions from previous searches, known as generations, along with random mutations to explore new regions of the solution space. In general, a GA repeats three steps (selection, crossover, and random mutations) as shown by the pseudo code in Figure 1. Selection according to fitness (efficiency in our case) is a source of exploitation, while crossover and random mutations promote exploration.

A generation of a GA contains a population of individuals, each of which corresponds to a possible solution from the search space. Each individual in the population is evaluated with a fitness function which indicates the goodness of a solution. Selection takes a certain number of individuals in the population and brings them forward to the next generation. Crossover takes pairs of individuals and uses parts of each to produce new individuals. The random mutation step swaps parts of an individual to prevent the GA from getting caught in a local minimum.

Much work has been done on using GAs for static scheduling (Hou et al., 1994; Ahmad et al., 2001; Theys et al., 2001; Zomaya et al., 2001), where schedules are created before runtime. However, the state of all tasks and system resources must be known a priori and cannot change. This limits these schedulers to specific problems and systems.

Dynamic GA schedulers (Maheswaran et al., 1999; Zomaya and Teh, 2001; Zomaya et al., 1999) create schedules at runtime, with

```
initialise population of individuals
do to population{
   crossover
   random mutation
   selection
}while(no individuals have met stopping conditions)
return fittest individual
```

Figure 1. Pseudo code for a genetic algorithm.

knowledge about the properties of the system and tasks possibly not known in advance. This allows for variable system and task properties to be considered. Dynamic GA schedulers are thus more practical than static schedulers for real-world distributed systems. Current dynamic GA schedulers have been shown to produce near optimal schedules in simulations (Zomaya et al., 1999; Zomaya and Teh, 2001), although assumptions that have been made limit their generality. For example, instantaneous message passing (Zomaya and Teh, 2001), homogeneous processing resources (Zomaya et al., 1999; Zomaya and Teh, 2001), variable communications costs and variable processing resources are not considered (Zomaya et al., 1999; Zomaya and Teh, 2001).

3. Scheduling Algorithm

The algorithm presented in this paper is based on the state-of-the-art homogeneous GA scheduler developed by Zomaya et al. (1999) and Zomaya and Teh (2001). We wish to schedule an unknown total number of tasks for processing on a distributed system with a minimal total execution time, otherwise known as makespan.

The processors of the distributed system are heterogeneous. The available network resources between processors in the distributed system can vary over time. The availability of each processor can vary over time (processors are not dedicated and may have other tasks which partially use their resources). Tasks are indivisible, independent of all other tasks, arrive randomly, and can be processed by any processor in the distributed system.

When tasks arrive they are placed in a queue of unscheduled tasks. Batches of tasks from this queue are scheduled on processors during each invocation of the scheduler. Each idle processor in the system requests a task to process from the scheduler, which it processes and returns. The scheduler contains a queue of future tasks for each processor, and when a request for work is received the task at the head of the corresponding queue is sent for processing. We wish to avoid repeatedly issuing the same task multiple times (e.g., a machine could be switched off) because network resources are limited and processing resources are not dedicated. To achieve this the processors do not contain a queue of tasks.

Each task has a resource requirement which is measured in millions of floating point operations (MFLOPs). The available processing

resources, or execution rate, of each processor is measured in MFLOPs per second, which we write as Mflop/s. The execution rate is measured using Dongarra's Linpack benchmark (Dongarra et al., 1979). This is a recognised standard used to benchmark systems for inclusion in the list of Top 500 Supercomputers (2005). Available processing and network resources vary over time, so a smoothing function is used to minimise localised fluctuations, thus allowing for a more realistic processing environment. A single processor is dedicated to scheduling.

The queue of unscheduled tasks could contain a large number of tasks and if all where to be scheduled at once, the scheduler could take a long time to find an efficient schedule. To speed up the scheduler, and reduce the chance of processors becoming idle, we only consider a subsequence of the unscheduled tasks, which we call a batch. A larger batch will usually result in a more efficient schedule (Zomaya and Teh, 2001). We must thus trade the batch size with running time. To do this we dynamically set the batch size according to the estimated amount of time until the first processor becomes idle.

3.1. *Encoding*

Each individual in the population represents a possible schedule for a batch of tasks. Figure 2 shows the encoding used. Each character is a mapping between a task and processor. Each character contains the unique identification number of a task, with -1 being used to delimit different processor queues, where P_i is processor i. Thus the length (number of characters) of each schedule is $N + M - 1$, where N is the number of tasks in the batch, and M is the total number of processors.

3.2. *Fitness function*

A fitness function is used to measure the closeness of the makespan of each individual in the population to a lower bound on the makespan.

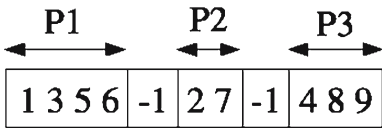

Figure 2. Encoding of an individual representing mappings of tasks to processors.

The finishing time of each processor j is calculated as $\delta_j = L_j / P_j$, where L_j is its previously assigned load in MFLOPs, and P_j is its current processing power in Mflop/s.

A lower bound on the makespan is defined as

$$\psi = \sum_{i=1}^{N} t_i \left(\sum_{j=1}^{M} P_j \right)^{-1} + \sum_{j=1}^{M} \delta_j,$$

where t_i is the processing requirement of task i in the batch (in MFLOPs) and where M and N are defined in Section 3.1.

The difference between the makespan of individual i and the lower bound on makespan is given by

$$E_i = \left(\sum_{j=1}^{M} \left| \psi - \left(L_{j,i} + \sum_{y=1}^{N} ((t_y / P_j) + \Gamma^c_{(y,j)}) \right) \right|^2 \right)^{1/2},$$

where $\Gamma^c_{(y,j)}$ is the communication cost of scheduling task y on processor j. The fitness value of individual i is $F_i = 1/E_i$, and $F_i = [0, \infty]$. A larger value indicates a better or fitter schedule.

3.3. *Selection, crossover and mutation*

We choose to use the standard weighted roulette wheel method of selection which is widely used by previous researchers who have applied GAs to task scheduling (Hou et al., 1994; Siegel et al., 1996; Zomaya and Teh, 2001). Each individual i in the population is assigned a slot between 0 and 1. The size of slot i is

$$\varsigma_i = F_i \times \left(\sum_{j=1}^{\rho} F_j \right)^{-1},$$

where $\sum_{i=1}^{\rho} \varsigma_i = 1$ and p is the number of individuals in the population. After the selection process is complete, we use the cycle crossover method (Oliver et al., 1987) to promote exploration as used in Zomaya and Teh (2001).

We have chosen to use two types of mutation to promote exploration of the search space. First, we randomly swap elements of a randomly chosen individual in the population. We allow the delimiters

between processor queues to be swapped, which allows for the lengths of queues within an individual to be randomly mutated. Then we use a re-balancing heuristic to mutate and improve the population.

The initial population is generated using a list scheduling heuristic, as follows. A percentage of the tasks are randomly assigned to processors, with the remaining tasks being assigned to the processors that will finish processing them the earliest. This leads to a well balanced and randomised initial population.

3.4. *Stopping conditions*

The GA will evolve the population until one or more stopping conditions are met. The individual with the lowest makespan is selected after each generation and if it is less than a specified minimum, the GA stops evolving. The maximum number of generations is set at 1000, because the quality of the schedules returned with more than that number does not justify the increased computation cost (as in Zomaya and Teh, 2001). The GA will also stop evolving if one of the processors becomes idle, in which case it will return the best schedule found so far.

3.5. *Exponential smoothing function*

A smoothing function, finds a single representative value for a sequence of values. As each new value is added to the sequence, this representative value is updated. For the first i values of a sequence of values $a_1, a_2, \ldots$, this representative value would be denoted Γ_i^a, and defined recursively as $\Gamma_i^a = \Gamma_{i-1}^a + \nu(a_i - \Gamma_{i-1}^a)$, where the smoothness of the sequence of representative values is controlled by $\nu \in [0, 1]$, and where we let $\Gamma_0^a = a_1$. The function allows one to vary the influence of more recent sequence values on the representative value, from no influence ($\nu = 0$) to complete dominance ($\nu = 1$). The smoothing function is employed in several instances in our scheduler. In this paper, we describe the application of the smoothing function to the first i values of an arbitrary sequence $x_1, x_2, \ldots$ with the notation Γ_i^x.

3.6. *Most-Into-Least*

An initial population is generated using the Most-Into-Least (MIL) list scheduling heuristic, which has been successfully used in other GA task schedulers (Correa et al., 1999; Greene, 2001). A random number

of tasks, are assigned to processors in a round robin fashion. The remaining tasks are then sorted, using Quicksort (Hoare, 1962), and allocated in a round robin fashion to the processors which will finish processing them the earliest, taking into account existing and assigned tasks for each processor. This leads to a well balanced randomised initial population.

3.7. *Dynamic batch size*

We wish to define batch sizes that are large enough so that the processor hosting the scheduler is utilised fully (and to achieve low makespans), but not too large that any processors become idle before the schedule has been fully computed. The GA takes $\Theta(H^2)$ time to create a schedule, where H is the number of tasks in a batch (batch size). After the pth batch has been scheduled, the first processor will become idle after $s_p = \min_{j=1}^{M}(\delta_j / P_j)$, where δ_j is the total processing time in MFLOPs of the tasks waiting to be processed by processor j, and M is the number of processors. We choose $H_{p+1} = \left\lfloor \left(\Gamma_p^s + 1\right)^{1/2} \right\rfloor$ as a simple approximation of the optimal size for batch $p+1$. Once a schedule has been assigned the batch size is recalculated.

4. Experiments

The scheduling algorithm described in Section 3 has been implemented and applied to simulated data, with up to 50 heterogeneous processors, and up to 100,000 randomly generated heterogeneous tasks. Each experiment was repeated a number of times and an average result was calculated for each point. We also implemented the original algorithm that our algorithm is based on, developed by Zomaya and Teh (2001), which is the current state of the art dynamic GA task scheduler for homogeneous distributed computing. It was easily adapted to work with heterogeneous processors by using Mflop/s as the measure of the rate of execution rather than time. We compare our scheduler to six other schedulers, and evaluate the results using two different but related metrics, makespan and efficiency. Makespan is the total execution time of a schedule. Efficiency is the percentage of the time that processors actually spend processing rather than communicating or idling.

Tasks are scheduled across 50 heterogeneous processors with a processing resource range of 10 to 100 Mflop/s. We assume that all of the

tasks arrive for processing at the beginning of the simulation, for these experiments. A representative set of heterogeneous computing task benchmarks does not exist as yet, as noted by Theys et al. (2001). We have decided to generate random sets of tasks for scheduling using the Poisson distribution. We use randomly generated task sets because: we wish to demonstrate the algorithms effectiveness over a broad range of conditions, a set of heterogeneous computing benchmark tasks do not exist, and it is not clear what characteristics a 'typical' task would exhibit (Theys et al., 2001).

We have decided to use a population size of 20, which is known as a micro GA (Chipperfield and Flemming, 1996) and used in (Zomaya et al., 1999; Greene, 2001; Zomaya and Teh, 2001), which speeds up computation time without impacting greatly on the final result.

4.1. *Other schedulers*

We have also compared our scheduling algorithm against a number of well known batch and immediate mode heuristic schedulers. An immediate mode scheduler only considers a single task for scheduling on a first come, first served (FCFS) basis while a batch mode scheduler considers a number of tasks at once for scheduling. We will compare our algorithm to three immediate mode and three batch mode schedulers (Maheswaran et al., 1999; Theys et al., 2001).

The earliest first (EF) algorithm is an immediate mode scheduler. When a task is presented for processing, the scheduler considers the existing load on each processor and allocates the task to the processor which will finish processing it the earliest. The EF algorithm uses the available information about the task and the processors when making a scheduling decision. It has a worst case complexity of $\Theta(M)$, where M is the number of processors, when scheduling a single task.

The lightest loaded (LL) scheduler is an immediate mode scheduler which allocates tasks to the processor with the lowest current load, measured in our case as MFLOPs. It does not consider the size of a task when scheduling. It has a worst case complexity of $\Theta(M)$.

The round robin (RR) scheduler is the most basic of the immediate mode schedulers used in these experiments, where tasks are assigned to processors in a round robin fashion. No load or task information is used when making a scheduling decision. It has a worst case complexity of $\Theta(1)$.

The max–min (MX) scheduler is a batch mode heuristic scheduler. It takes batches of tasks on a FCFS basis. These tasks are then

sorted according to task size in a descending order. The largest task is then allocated to the processor that will finish processing it first (same as EF). This is repeated until the batch is empty, after which another batch is considered. The main aim of this scheduler is to have the largest tasks scheduled as early as possible, with smaller tasks at the end filling in the gaps. It has a complexity of $\Theta(\max(M, n \log n))$, where n is the size of the batch.

The min–min (MM) scheduler is similar to the MX scheduler, except tasks are sorted in ascending order according to size.

The scheduler proposed by Zomaya et al. (ZO) in Zomaya and Teh (2001) has been implemented for this paper. It is the current state-of-the-art homogeneous GA scheduler and the basis for our scheduler. The ZO scheduler was easily converted from a homogeneous scheduler to a heterogeneous scheduler by using the Mflop/s benchmark for task sizes rather than time. It is a batch scheduler which uses GAs to create schedules. We have validated our implementation of this scheduler by reproducing some of the performance results in Zomaya and Teh (2001) (not included here).

4.2. *Setup*

We simulated the performance of our scheduler against the performance of six other schedulers, described in Section 4.1, for these experiments. All of the tasks arrived for scheduling at the beginning of the simulation. Each experiment was repeated 50 times and an average result was calculated for each point on the resulting graphs.

We scheduled up to 100,000 heterogeneous tasks onto 50 heterogeneous processors. For these experiments each processor was assumed to have a fixed execution rate, measured in Mflop/s. The aim of these experiments is to show that predicting the communication costs in advance will improve the efficiency, compared to heuristics which adapt to communication costs after they have been incurred. All schedulers were presented with the same set of tasks for scheduling and all schedulers have the same information available to them.

We have decided to use a population size of 20, which is known as a micro GA (Chipperfield and Flemming, 1996) and used in Zomaya and Teh (2001) and Zomaya et al. (1999), which speeds up computation time without impacting greatly on the final result.

4.3. *Communication*

We wish to show that our algorithm provides greater efficiency in a system with variable communication costs. To demonstrate its effectiveness we vary the ratio of the task processing requirement to communications costs, and measure the efficiency achieved. We fix the available processing resources and the size of the batch, to allow for the effect of communication costs to be demonstrated. We wish to schedule 100,000 tasks with a view to maximising the efficiency of the processing resources in the distributed system.

Figure 3 shows that PN consistently provides schedules with greater efficiency over all of the other scheduling algorithms. The horizontal axis in Figure 3 is the mean communication cost for all communication links between all clients and the scheduler. Each communications link has its own randomly generated mean cost, which is normally distributed. The consideration of communication costs allows the improved scheduler to estimate a communications cost when creating

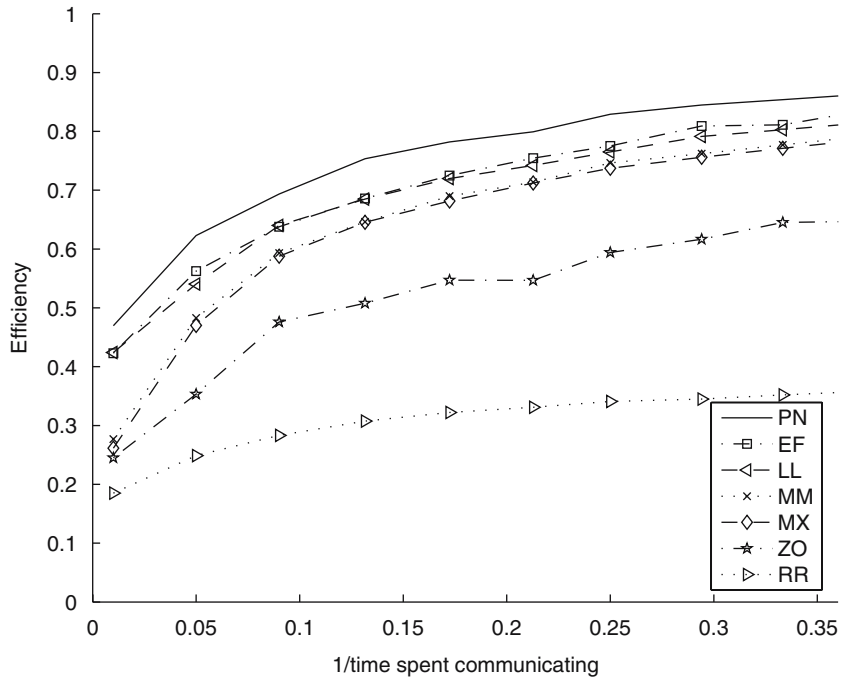

Figure 3. Efficiency of schedulers varying communication to task size ratio.

a schedule, resulting in an overall improvement in efficiency of the scheduler.

4.4. *Task size distribution*

We have randomly generated sets of tasks using a Poisson distribution and varied the mean of the set. Each set contained 10,000 tasks. In Figure 4 we can see that PN performs the best followed by MM, whilst MX performs quite badly, when the mean is small. When the mean is increased to 100 MFLOPs (see Figure 5) the batch schedulers all perform well, whilst the immediate mode schedulers do not perform as well.

5. Conclusion

A scheduling algorithm has been developed to schedule heterogeneous tasks onto heterogeneous processors in a distributed computing system. It provides efficient schedules and adapts to varying resource

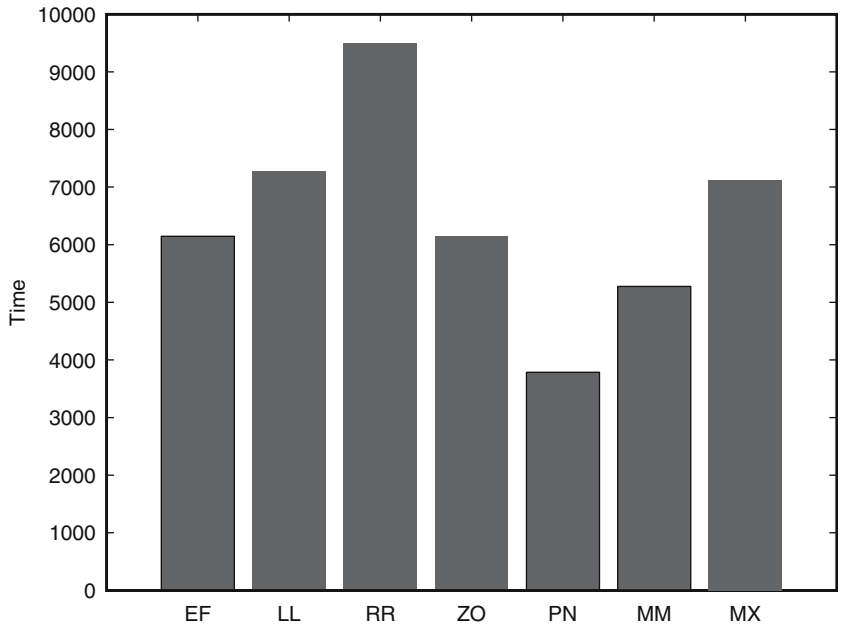

Figure 4. Makespan when task sizes have a Poisson distributed with a mean of 10 MFLOPs.

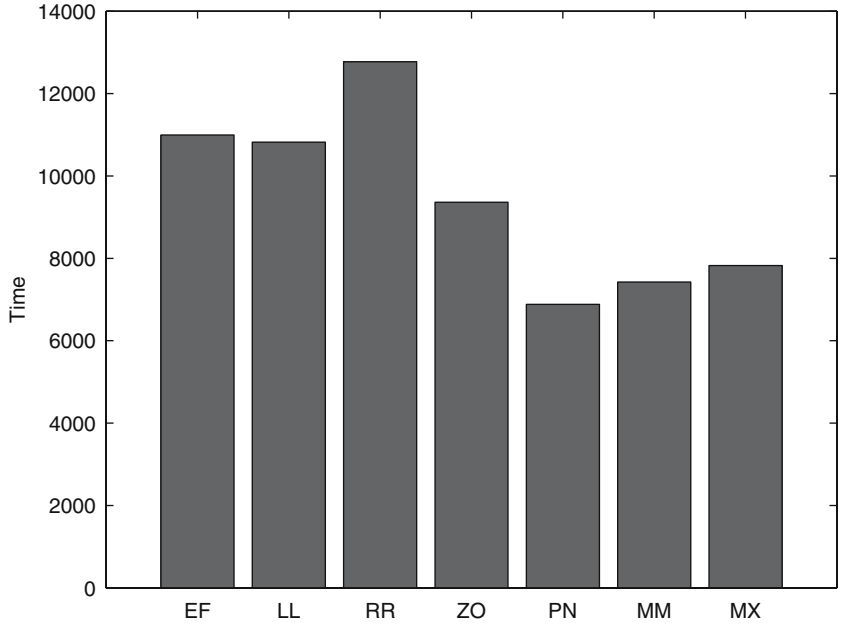

Figure 5. Makespan when task sizes have a Poisson distributed with a mean of 100 MFLOPs.

availability (processing resources and communication costs). The algorithm also fully utilises the dedicated processor running the scheduler. The GA employed a list scheduling heuristic to create a well-balanced randomised initial population. The fitness function utilises the relative error metric internally to find schedules with a low makespan. Roulette wheel selection is used to exploit past results to direct the search for efficient schedules. Cycle crossover promotes exploration of the search space, with random swaps and random re-balancing of processor queues within individuals perturbing this exploration.

The Figures 3–5 show that our scheduler performs better than the other schedulers. We can conclude that our scheduler gives better performance over multiple different scenarios and would give consistently better efficiency in unknown conditions compared to the other techniques tested in this study. Our scheduler estimates the communication costs between each client and server using historical information, so it can create better schedules and reduce the makespan. For the other schedulers, the effect of communication is only considered after tasks or batches of tasks have been scheduled, leading to less efficient solutions.

The algorithm proposed in this paper consistently uses processors more efficiently than the current state-of-the-art GA algorithms for the same problem. It is more suitable for real-world use because it considers properties of distributed systems, such as variable communication costs and variable availability heterogeneous processors, which other algorithms for the task scheduling problem do not consider.

Acknowledgement

Support is acknowledged from the Irish Research Council for Science, Engineering, and Technology, funded by the National Development Plan.

References

Ahmad, I., Kwok, Y.-K., Ahmad, I. & Dhodhi, M. (2001). Scheduling Parallel Programs Using Genetic Algorithms. In Zomaya, A. Y., Ercal, F. & Olariu, S. (eds.) *Solutions to Parallel and Distributed Computing Problems*. New York, USA: John Wiley and Sons, Chapt. 9, pp. 231–254.

Chipperfield, A. & Flemming, P. (1996). Parallel Genetic Algorithms. In Zomaya, A. Y. (ed.) *Parallel and Distributed Computing Handbook*. New York, USA: McGraw-Hill, first edition, pp. 1118–1143.

Colorni, A., Dorigo, M. & Maniezzo, V. (1992). Distributed Optimization by Ant Colonies. In *Proceedings of the First European Conference on Artificial Life*. Paris, France, Elsevier, 134–142.

Correa, R., Ferreira, A. & Rebreyend, P. (1999). Scheduling Multiprocessor Tasks with Genetic Algorithms. *IEEE Transactions on Parallel and Distributed Systems* **10**(8): 825–837.

Dongarra, J., Bunch, J., Moler, C. & Stewart, G. (1979). *LINPACK Users Guide*. Philadelphia, USA: SIAM.

Garey, M. R. & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY: W. H. Freeman & Co.

Glover, F. (1986). Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers and Operations Research* **13**: 533–549.

Greene, W. A. (2001). Dynamic Load-Balancing via a Genetic Algorithm. In *13th IEEE International Conference on Tools with Artificial Intelligence*. Dallas, Texas, USA, 121–129.

Hoare, C. A. R. (1962). Quicksort. *Computer Journal* **5**(1): 10–15.

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems*. Cambridge, MA, USA: MIT Press.

Hou, E., Ansari, N. & Ren, H. (1994). A Genetic Algorithm for Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems* **5**(2): 113–120.

Kasahara, H. & Narita, S. (1984). Practical Multiprocessing Scheduling Algorithms for Efficient Parallel Processing. *IEEE Transactions on Computers* **33**(11):1023–1029.

Maheswaran, M., Ali, S., Siegel, H. J., Hensgen, D. & Freund, R. F. (1999). Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing* **59**(2): 107–131.

Oliver, I. M., Smith, D. J. & Holland, J. (1987). A Study of Permutation Crossover Operators on the Traveling Salesman Problem. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*. Lawrence Erlbaum Associates, Inc, 224–230.

Siegel, H. J., Wang, L., Roychowdhury, V. & Tan, M. (1996). Computing with Heterogeneous Parallel Machines: Advantages and Challenges. In *Proceedings on Second International Symposium on Parallel Architectures, Algorithms, and Networks*. Beijing, China, 368–374.

Theys, M. D., Braun, T. D., Siegal, H. J., Maciejewski, A. A. & Kwok, Y.-K. (2001). *Mapping Tasks onto Distributed Heterogeneous Computing Systems Using a Genetic Algorithm Approach*, New York, USA: John Wiley and Sons, Chapt. 6, pp. 135–178.

Top 500 Super Computers (2005). http://www.top500.org.

Zomaya, A. Y., Clements, M. & Olariu, S. (1998). A Framework for Reinforcement-based Scheduling in Parallel Processor Systems. *IEEE Transactions on Parallel and Distributed Systems* **9**(3): 249–260.

Zomaya, A. Y., Lee, R. C. & Olariu, S. (2001). An Introduction to Genetic-Based Scheduling in Parallel Processor Systems. In Zomaya, A. Y., Ercal, F. & Olariu S. (eds.) *Solutions to Parallel and Distributed Computing Problems*. New York, USA: John Wiley and Sons, Chapt. 5, pp. 111–133.

Zomaya, A. Y. & Teh, Y.-H. (2001). Observations on using Genetic Algorithms for Dynamic Load-balancing. *IEEE Transactions on Parallel and Distributed Systems* **12**(9): 899–911.

Zomaya, A. Y., Ward, C. & Macey, B. (1999). Genetic Scheduling for Parallel Processor Systems: Comparative Studies and Performance Issues. *IEEE Transactions on Parallel and Distributed Systems* **10**(8): 795–812.