Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing

Andrew J. Page and Thomas J. Naughton
Department of Computer Science,
National University of Ireland, Maynooth,
County Kildare, Ireland.
andrew.j.page@nuim.ie, tom.naughton@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. We have performed simulations with randomly generated task sets, using uniform, normal, and Poisson distributions, whilst varying the communication overheads between the clients and scheduler. We have achieved more efficient results than all other schedulers across a range of different scenarios while scheduling 10,000 tasks on up to 50 heterogeneous processors.

1. Introduction

Distributed computing is a promising approach to meet the increasing computational requirements of scientific research. However, a number of issues arise which are not encountered in sequential processing which, if not properly handled, can nullify the benefits of parallelization. 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 and can offset the gains from parallelization due to excessive communication overhead or under-utilisation of resources. 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 [5].

Many heuristic algorithms exist for specific instances of the task scheduling problem, but are inefficient for a more general case [9]. The use of Holland's genetic algorithms [7] (GAs) 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 [8, 19] and heterogeneous [1, 11, 15, 18] multi-processor systems with notable success.

Unfortunately, assumptions are often made which reduce the generality of these solutions, such that scheduling can be calculated off-line in advance and cannot change [1, 8, 15, 18], all communications times are known in advance [1, 8, 15, 18], networks provide instantaneous message passing [19], that all processors have equal capabilities and are dedicated to processing tasks from the scheduler [1, 8, 9, 14, 15, 17, 18, 19, 20]. 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 continuously, and considers variable system resources, which has not been considered by other dynamic GA schedulers. This paper is an updated version of [13] in which we present a revised algorithm, more comprehensive experiments, and significant testing and verification.

In Sect. 2 we review related work and give an overview of how a GA operates. In Sect. 3 we describe our scheduling algorithm. In Sect. 4 we present the results of our performance experiments. In Sect. 5 we give our conclusions and suggest future directions for our work in Sect. 6.

```
initialise population
do{
   crossover
   random mutation
   selection
}while(stopping conditions not met)

return best individual
```

Figure 1. Pseudo code for genetic algorithm

2. Genetic Algorithms and scheduling

There are many examples in the literature of artificial intelligence techniques being applied to task scheduling [1, 8, 11, 14, 15, 17, 18, 19, 20]. Meta-heuristic search techniques such as GAs [7], tabu [6], and ant colony search [3] are most applicable to the task scheduling problem because we wish to quickly search for a near optimal schedule out of all possible schedules. Good results have resulted from the use of GAs in task scheduling algorithms [1, 8, 11, 14, 15, 17, 19, 20].

A GA is a meta-heuristic search technique which allows for large solution spaces to be partially searched in polynomial time, by applying evolutionary techniques from nature [7]. 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 Fig. 1. Selection according to fitness (efficiency in our case) is a source of exploitation, and crossover and random mutations promote exploration.

A generation of a GA contains a population of individuals, each of which correspond to a possible solution from the search space. Each individual in the population is evaluated with a fitness function to produce a value 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. Random mutations 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 [1, 8, 15, 18], 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 [11, 19, 20] create schedules at runtime, with knowledge about the properties of the system and tasks possibly not known in advance, allowing 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 [19, 20], although assumptions that have been made limit their generality. For example, instantaneous message passing [19], homogeneous processing resources [19, 20], variable communications costs and variable processing resources are not considered [19, 20].

3. Scheduling Algorithm

The algorithm we have developed is based on the state-of-the-art homogeneous GA scheduler developed by Zomaya *et al.* [19, 20]. We have created an algorithm which can adapt to varying resource environments and can produce near-optimal schedules. 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 can may have other tasks that 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. A processor does not contain a queue of tasks; because network resources are limited and processing resources are not dedicated, we wish to avoid repeatedly issuing the same task multiple times, e.g., when a machine is switched off.

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 to avoid confusion. The execution rate is measured using Dongarra's Linpack benchmark [4]. This is a recognised standard used to benchmark systems for inclusion in the list of Top 500 Supercomputers [16]. 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 sched-

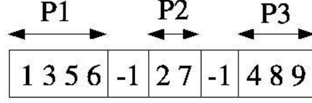

Figure 2. Encoding of a schedule

ule. To speedup the scheduler, and reduce the chance of processors becoming idle, we only consider a subset of the unscheduled tasks, which we call a batch. A larger batch will usually result in a more efficient schedule [19]. 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. Fig. 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 number of characters is $H + M - 1$, where H is the number of tasks in the batch, and M is the number of processors.

3.2. Fitness Function

A fitness function attaches a value to each individual in the population, which indicates the goodness of the schedule. We use relative error to generate the fitness values. We wish to calculate the fitness of each individual in the population. Previously assigned, but unprocessed, load for each processor is considered by calculating the finishing time of a processor j. $\delta_{j,i} = (L_j/P_j)$, where L_j denotes the previously assigned load, measured in MFLOPs, and P_j is the current processing power in Mflop/s of processor j.

The theoretical optimal processing time can now be found as,
$\psi = \left(\sum_{i=1}^{N} t_i / \sum_{j=1}^{M} P_j\right) + \sum_{j=1}^{M} \delta_j$, where t_i is the processing requirement of task i in the batch (in MFLOPs) and N is the total number of tasks in the batch.

The relative error of individual i is given as

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

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$F_i = [0, 1]$. 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 [8, 14, 19]. 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 [12] to promote exploration as used in [19]. 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. Then we use a re-balancing heuristic to mutate and improve the population. The initial population is generated using a list scheduling heuristic. A percentage of 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 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 [19] and demonstrated in Fig. 3). 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. Rebalancing heuristic

We have introduced a re-balancing heuristic to improve the quality of results returned. For each individual in the population, in each generation, we select the most heavily loaded processor. A task is then selected at random from another processor and if it is smaller than a task in the most heavily loaded processor, a swap is performed. We only allow a maximum of 5 random searches for a smaller task. If the resulting schedule is fitter, it is kept.

Fig. 3 shows the average percentage decrease in makespan after each generation of the GA, with points taken after every generation. Each point on the graph is an average of 50 runs of the scheduler. The largest reductions in makespan occur in the first 100 generations. After that the reductions begin to level out, requiring larger numbers of generations, with little improvement.

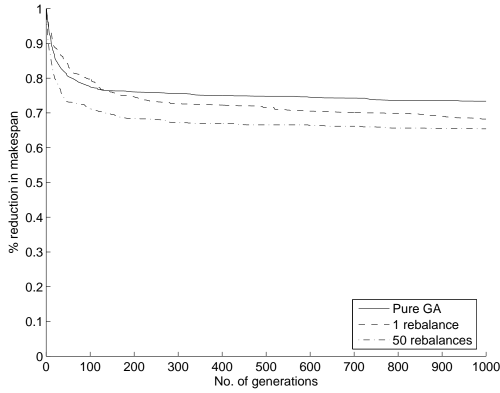

Figure 3. Average reduction in makespan after each generation of the GA

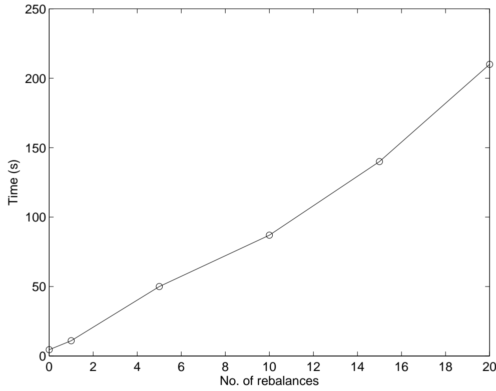

Figure 4. Time taken to schedule 10,000 tasks with varying numbers of re-balances in every generation of the GA

The re-balancing heuristic minimises the makespan further than a pure GA, with 50 re-balances per individual in the population per generation, resulting in the makespan being reduced to only 65% of its original value after 1000 generations (from Fig. 3). A single rebalance reduces the makespan to approximatly 70%, whilst no rebalancing (pure GA) reduces to 75%.

These re-balances do have an associated additional cost in terms of time. Fig. 4 shows the time taken to run a GA for 1000 generations with varying numbers of re-balances. It increases the time taken linearly. We have decided to only perform a single re-balancing at each generation to enable the algorithm to run quickly.

3.6. Smoothing Function

A smoothing function is defined that 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.

3.7. Dynamic Batch Size

We wish to define batch sizes that are large enough so that the processor hosting the scheduler is utilized 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 Sect. 3 has been implemented and and applied to simulated data. A number of different experiments have been performed to demonstrate the effectiveness of the scheduling algorithm with varying communicating costs. 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.

A representative set of heterogeneous computing task benchmarks does not exist as yet, as noted by Theys *et al.* [15]. Our task sizes are randomly generated using, uniform, normal, and Poisson distributions. By using different random distributions, we can demonstrate the flexibility of our scheduling algorithm. This is often overlooked in the

literature [1, 8, 9, 11, 14, 15, 17, 19, 20]. For these experiments we will vary the communication costs and the task sizes.

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 FCFS (first come, first served) 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 [11, 15].

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. 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 [19] 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.

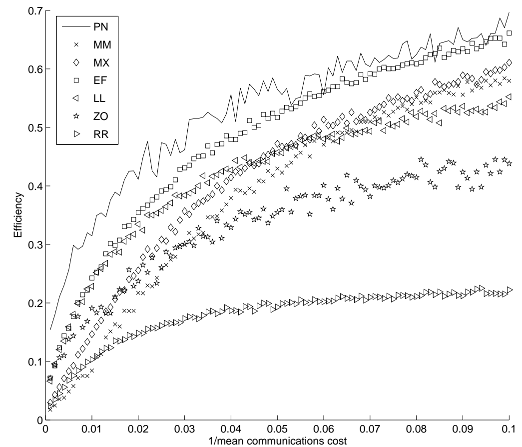

Figure 5. Efficiency of schedulers with a normal distribution of task sizes and varying communication costs.

We have validated our implementation of this scheduler by reproducing some of the performance results in [19] (not included here).

4.2. Setup

We simulated the performance of our scheduler against the performance of six other schedulers, described in Sect. 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 10,000 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 [2] and used in [19, 20], which speeds up computation time without impacting greatly on the final result.

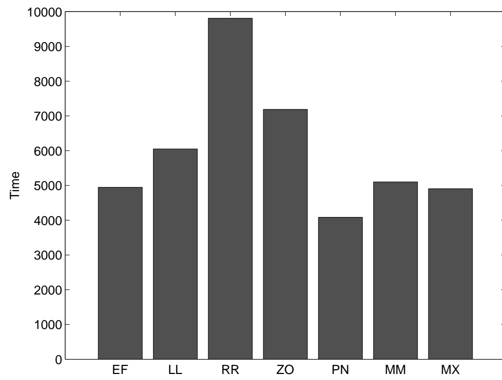

Figure 6. Makespan when task sizes have a normal distributed with a mean of 1000 MFLOPs and a variance of 9×10^5

4.3. Normal distribution

Fig. 5 shows the efficiency of the seven different scheduling algorithms when the task sizes are normally distributed. We used a batch size of 200 with 1000 tasks to be scheduled which were randomly generated at the beginning of each scheduling simulation with each point on the graph consisting of an average of 20 complete schedules. Fig. 5 consists of the efficiency of 2000 complete schedules with varying communications costs, and all other variables kept fixed. The task sizes were generated with a mean of 1000 MFLOPs and a variance of 9×10^5. The horizontal axis in Figs. 5 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. Fig. 5 shows that our scheduler (PN) gives the best processor efficiency. Fig. 6 is the makespan for the algorithm, with a varying batch size and shows that PN out performs all the other schedulers in terms of total execution time.

4.4. Uniform distribution

Fig. 7 shows the efficiency of the seven different scheduler with varying communication costs. The task sizes were uniformly distributed between 10 and 1000 MFLOPs. The two meta-heuristic schedulers (PN and ZO) clearly provide more efficient schedules compared to the more simple heuristic schedulers. This occurs because the meta-heuristic schedulers have the ability to explore a wider search space. We have also varied the range of task sizes noting the makespan in each case. In Fig. 8 many of the schedulers provide similarly efficient schedules. This is because the ratio of the smallest to the largest task is only 1:10. As the set of tasks becomes more equal, the efficiency of most of the schedulers should improve. We see that when the range is increased in Fig. 9 (1:1000) the differences between the various schedulers become more accentuated.

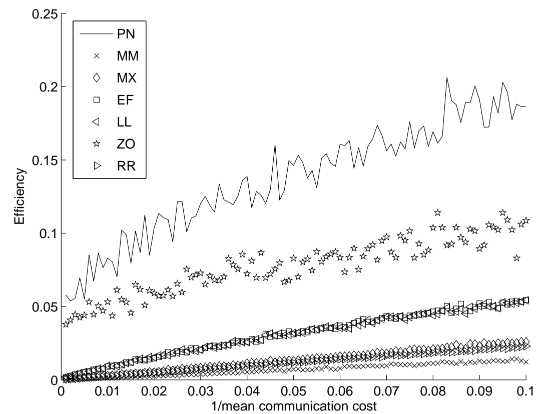

Figure 7. Efficiency of schedulers with a uniform distribution of task sizes and varying communication costs.

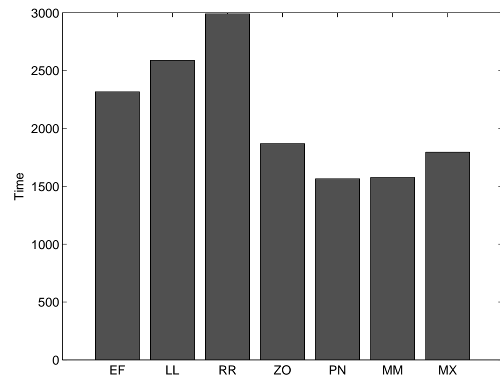

Figure 8. Makespan when task sizes are uniformly distributed between 10 and 100 MFLOPs.

4.5. Poisson distribution

We have randomly generated sets of tasks using a Poisson distribution and varied the mean. In Fig. 10 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 Fig. 11) the batch sched-

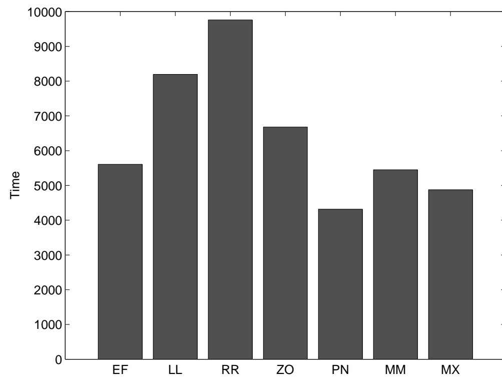

Figure 9. Makespan when task sizes are uniformly distributed between 10 and 10000 MFLOPs.

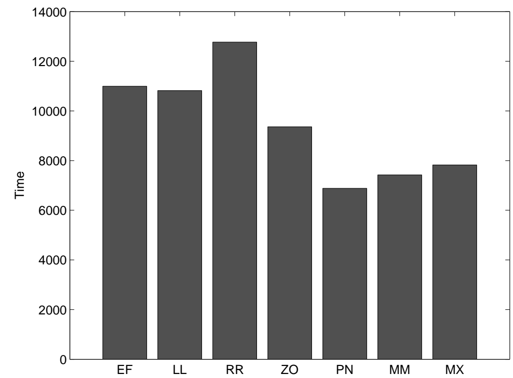

Figure 11. Makespan when task sizes have a Poisson distributed with a mean of 100 MFLOPs

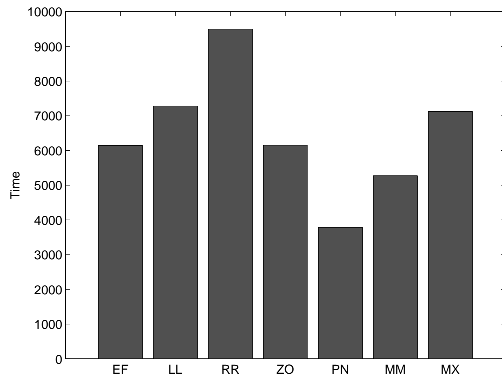

Figure 10. Makespan when task sizes have a Poisson distributed with a mean of 10 MFLOPs

ulers 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 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.

We have tested our scheduler under various different scenarios. To show the generality of our scheduler we used three different types of random distributions, each with thousands of different randomly generated sets of tasks and varying communication overheads.

The Figs. 5 through 11 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.

6. Future work

We intend to compare all of the schedulers in Sect. 4 on a general-purpose distributed system [10]. The system is currently deployed on over 250 heterogeneous PCs and runs problems from cryptography, bioinformatics, and biomedical engineering. This will allow us to test our scheduler under real-world conditions.

7. Acknowledgement

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

References

[1] I. Ahmad, Y.-K. Kwok, I. Ahmad, and M. Dhodhi. Scheduling parallel programs using genetic algorithms. In A. Y. Zomaya, F. Ercal, and S. Olariu, editors, *Solutions to Parallel and Distributed Computing Problems*, chapter 9, pages 231–254. John Wiley and Sons, New York, USA, 2001.

[2] A. Chipperfield and P. Flemming. Parallel genetic algorithms. In A. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, pages 1118–1143. McGraw-Hill, New York, USA, first edition, 1996.

[3] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In *Proceedings of the First European Conference on Artificial Life*, pages 134–142, Paris, France, 1992. Elsevier.

[4] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users Guide*. SIAM, Philadelphia, USA, 1979.

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

[6] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13:533–549, May 1986.

[7] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, MA, USA, 1992.

[8] E. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113–120, February 1994.

[9] H. Kasahara and S.Narita. Practical multiprocessing scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, 33(11):1023–1029, November 1984.

[10] T. M. Keane. A general-purpose heterogeneous distributed computing system. M.Sc. thesis, Department of Computer Science National University of Ireland, Maynooth, Ireland, 2005.

[11] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, November 1999.

[12] I. M. Oliver, D. J. Smith, and J. Holland. 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*, pages 224–230. Lawrence Erlbaum Associates, Inc., 1987.

[13] A. J. Page and T. J. Naughton. Framework for task scheduling in heterogeneous distributed computing using genetic algorithms. In *15th Artificial Intelligence and Cognitive Science Conference*, pages 137–146, Castlebar, Ireland, September 2004.

[14] H. J. Siegel, L. Wang, V. Roychowdhury, and M. Tan. Computing with heterogeneous parallel machines: advantages and challenges. In *Proceedings on Second International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 368–374, Beijing, China, June 1996.

[15] M. D. Theys, T. D. Braun, H. J. Siegal, A. A. Maciejewski, and Y.-K. Kwok. *Mapping Tasks onto Distributed Heterogeneous Computing Systems Using a Genetic Algorithm Approach*, chapter 6, pages 135–178. John Wiley and Sons, New York, USA, 2001.

[16] Top 500 Super Computers. http://www.top500.org.

[17] A. Y. Zomaya, M. Clements, and S. Olariu. A framework for reinforcement-based scheduling in parallel processor systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):249–260, March 1998.

[18] A. Y. Zomaya, R. C. Lee, and S. Olariu. An introduction to genetic-based scheduling in parallel processor systems. In A. Y. Zomaya, F. Ercal, and S. Olariu, editors, *Solutions to Parallel and Distributed Computing Problems*, chapter 5, pages 111–133. John Wiley and Sons, New York, USA, 2001.

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

[20] A. Y. Zomaya, C. Ward, and B. Macey. Genetic scheduling for parallel processor systems: comparative studies and performance issues. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):795–812, August 1999.