PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 41, NO. 3, PP. 231–239 (1997)

AN EXPERIMENTAL INVESTIGATION OF A MULTI-PROCESSOR SCHEDULING SYSTEM

Colin REEVES and Helen KARATZA*

School of Mathematical and Information Sciences
Coventry University, UK
Email: CRReeves@uk.ac.cov.cck
* Department of Mathematics
Aristotle University of Thessaloniki
Thessaloniki, Greece
Email: cbdz05@grtheunl1.earn

Received: June 23, 1995

Abstract

The scheduling of jobs through a multi-processor system is important from many aspects. It is often assumed that jobs are scheduled on the basis of some simple rule, such as First-Come First-Served, or Shortest Processing Time First.

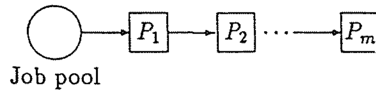
In earlier work we found some evidence to suggest that use of a more sophisticated strategy, based on the use of a Genetic Algorithm (GA) to 'look ahead', could enhance system performance. Here we investigate this idea more thoroughly.

1. Introduction

Recent advanced in heuristic methods for static sequencing problems have included several reports [1, 2, 3] of the use of *genetic algorithms,* which have been found to be robust and efficient ways of solving such problems. In an earlier paper [4], we considered the use of a genetic algorithm (GA) to solve a dynamic flowshop sequencing problem. This problem relates to the sequencing of jobs on machines in a manufacturing environment, but this case has obvious parallels in a computing context, where the jobs are program tasks, and the machines are processors. There is a difference in that the scheduling of computer program tasks needs to be done in real time, which is not so critical a requirement in a manufacturing environment. But first we need to establish whether such an approach can indeed out-perform simple scheduling rules, before considering how it could be implemented in practice.

We consider a somewhat idealised problem, where jobs arrive at a job pool before passing through m processors arranged in series. The time t_{ij} required for processing job i on processor j is known, or can be reliably estimated. There is infinite buffer storage between consecutive processors, and no job pre-emption is allowed. Initially there are n' jobs in the pool,

but further jobs arrive as time passes, in accordance with a known inter-arrival time probability distribution.



The problem is at any stage to determine the sequence in which the jobs in the pool should be processed in order to optimise some measure of performance. This is clearly a dynamic problem, since as more jobs arrive the current 'best' sequence may have to change. Of course this is an approximation to what really happens — in real problems jobs may not call on all processors in the same order, they may need to visit a subset of processors more than once, and so on. However, our purpose in studying this simplified version of the problem, as outlined above, was to test the effectiveness of different ways of scheduling jobs. Simple scheduling rules are usually concerned only with the next job, without trying to consider the current job pool as a whole. Our hypothesis is that using a GA to 'look ahead' would be a more effective means of approaching such a problem.

There are a number of ways of assessing the performance of a system like that described. It was decided that the most natural performance measures would be the mean *response time,*

$$R(n) = \sum_{j=1}^{n} \frac{C_j - A_j}{n},$$

where n jobs have been processed, and job j arrived at time A_j, and was completed at time C_j; and the *throughput rate,*

$$T(n) = \frac{n}{C_n - A_1}.$$

These performance measures are of course correlated to some extent, but while response time refers to the system performance from the viewpoint of the jobs, throughput rate measures performance from the server's perspective.

2. Implementation

A simulation model of the system described above was programmed, as shown in the box below:

```
- Initialise job pool;
- Compute job sequence;
- Schedule 1st job;
- Compute 1st event time T_E;
- Repeat
      If no arrivals before T_E then
          1. schedule next job;
          2. compute next event time T_E;
- else
          1. add additional job(s) to current job pool;
          2. re-compute job sequence for current pool;
          3. schedule next job
          4. compute next event time T_E;
- Until simulation time exceeds a specified limit.
```

Clearly, the GA enters at the points where a re-computation of the 'best' sequence of the current job pool is required. The simple scheduling rules would also be implemented at this stage.

It is important to realize that by re-computing the best sequence from the current job pool, we make the assumption that a good overall solution will be obtained if we try at any stage to sequence the currently available jobs as if no more jobs will arrive. It is this hypothesis that we shall evaluate by comparing with more traditional job scheduling criteria.

2.1. The Genetic Algorithm

The GA used to solve the sequencing problem was adapted from that described in [3], whose characteristics can be summarised as follows:

- an initial population of 30 chromosomes using a sequence representation;
- parent selection using ranking;
- incremental population replacement (also known as a steady-state GA);
- replacement of a randomly chosen string of below-median fitness;
  - a sequence-based crossover (see [3] for details);
  - an adaptive mutation rate;
  - a termination condition of $\mathcal{O}(nm \log[m+n])$ objective function evaluations.

At the first stage, the initial chromosomes were chosen at random, and this could also be done at each subsequent application of the GA. However, by basing, at each stage, the initial population on the population of solutions obtained at the previous stage, we found that good solutions to the current problem were determined more rapidly, which may be an advantage when a decision on the next job to be sequenced is needed in real-time.

2.2. Other Selection Criteria

There were 3 obvious candidates for simple selection criteria instead of the GA: we could use

- job arrival order (FCFS);
- shortest (first machine) processing-time order (SPT(1)).
- shortest (total) processing-time order (SPT(all));

The first of these corresponds to doing nothing, simply scheduling on a First-Come First-Served basis; the other two attempt to take into account the likely delay to other jobs that could be incurred by scheduling a specified job now. Clearly, by scheduling a job with a large processing-time requirement when other (shorter) jobs are available, the response-time for those other jobs is likely to be increased. The first machine is of course the most important in our model, since once the current job completes processing on the first machine, we are free to schedule another. The rationale for SPT (all) is that, like the GA, it also tries to 'look ahead' beyond the immediate decision.

These type of criteria have been studied for some special cases of single-processor scheduling problems using a queuing-theoretic framework, and CONWAY et al. [5] have an interesting discussion which shows that,

under certain conditions, the Shortest-Processing Time criterion is optimal for single-machine problems. However, this cannot be shown to hold for multi-processor problems.

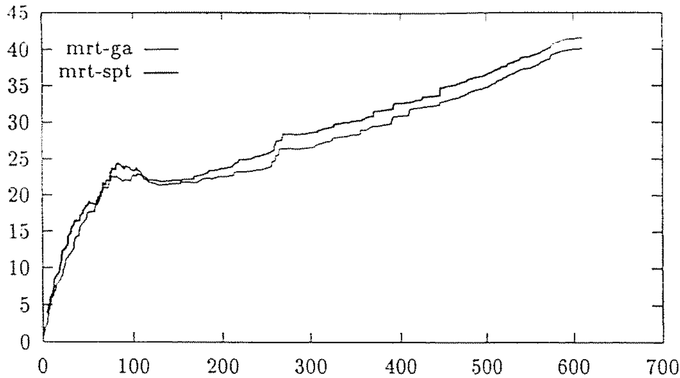


Fig. 1. Mean response times

3. Test Problems

Several sets of test problems were generated. In each case, the arrival rate and service (processing) rate of jobs were assumed to be the same: clearly, if the arrival rate is greater than the service rate, the size of the job pool will increase without bound, which would not be tolerated in a real system. Job arrivals were assumed to occur according to a Poisson process, but job-processing times were generated from 5 different distributions with different coefficients of variation (CVs) of processing-times. We used Erlang-k (hypo-exponential) distributions with $k = 4$ and $k = 16$, an exponential distribution (corresponding to a Poisson process), and two branching-Erlang distributions to simulate distributions with high CVs. The complete range of CV values was $\{0.25, 0.5, 1, 2, 4\}$. General details of the distributions used and their characteristics can be found in, for example, SAUER – CHANDY [6].

In each case, 30 jobs were assumed to be in the pool initially, and the simulation was continued until a further 530 jobs had entered the system. In the first group of problems, the number of processors was set at 4. The values of $T(n)$ and $R(n)$ were measured when each job completed all its tasks. They could then be plotted on a graph as shown in the example below.

To do this for every run is clearly impracticable; in order to summarise these graphs, we calculated the average difference between the $T(n)$ and $R(n)$ values for GA and FCFS over the whole run length. Thus we could obtain a measure of the success of the GA against the 'donothing' option. We then repeated this for SPT(1) against FCFS, and SPT(all) against FCFS. Each case was replicated 4 times, so the results reported in *Table 1* below are the means of 4 runs in each case. In this table, the first value in each cell is the average difference for $R(n)$, the second for $T(n)$.

Table 1
Average differences in MRT & TPR: 4 processors

CV	GA	SPT(1)	SPT(all)
0.25	−10.33	−9.15	−12.76
	0.024	0.014	0.034
0.50	−10.48	−8.73	−16.03
	0.023	0.012	0.058
1.00	−17.35	−13.26	−20.37
	0.056	0.032	0.112
2.00	−32.32	−21.05	−39.90
	0.119	0.052	0.201
4.00	−30.76	−6.97	−37.61
	0.141	0.026	0.174

The whole procedure was then repeated for the case of 8 processors, with the results shown in *Table 2*.

Table 2
Average differences in MRT & TPR: 8 processors

CV	GA	SPT(1)	SPT(all)
0.25	−12.52	−8.64	−12.54
	0.038	0.016	0.036
0.50	−8.72	−6.94	−13.03
	0.022	0.007	0.041
1.00	−17.54	−10.06	−21.05
	0.056	0.006	0.097
2.00	−34.58	−12.50	−37.74
	0.090	0.024	0.140
4.00	−62.32	−21.16	−59.71
	0.133	0.028	0.170

On the whole, all 3 rules were able to improve system performance, in terms of both performance metrics: that is, it is possible to improve performance both from the point of view of the system throughput and at the same time to provide a better service from the point of view of the customer.

It is clear that SPT(1) provides the last improvement in performance over FCFS. It can also be seen that SPT(all) nearly always does slightly better than using the GA. This contradicts our earlier findings, reported in [4]. We have not yet fully resolved this contradiction, which may be simply an artefact of the random number streams used in the simulation. However, this latest work is based on more extensive and comprehensive testing, and is probably more reliable. The differences are in any case fairly small.

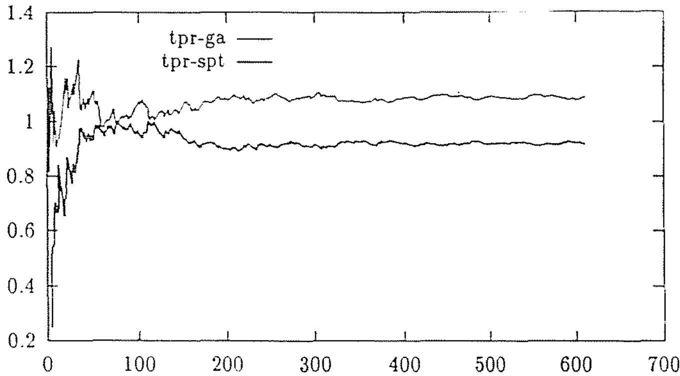


Fig. 2. Throughput rates

We must also bear in mind the amount of computing has to be done in each case. In terms of time actually spent in selecting the next job, SPT(1) needs the least, while SPT(all) needs slightly more, since it requires the summation of processing times for m machines. The amount needed for the GA can be user-controlled, depending on what degree of convergence to the (unknown) optimal sequence is desired. In practice, and on average, it took an order of magnitude more computation than the SPT rules. (There was considerable variability, too: in the case $CV < 1$, for most of the simulation period the queue lengths tended to be much greater, which meant the GA had far bigger problems to solve, and thus took much longer.) In view of this, it would seem desirable to use the simpler SPT(all) rule.

4. Conclusions

The results obtained confirm that, on average, the performance of a multi-processor system is improved by using a 'look-ahead' rule for scheduling rather than FCFS. However, contrary to our original expectations, the extra sophistication of a GA-based scheduler was not in this case worth using. The SPT(all) rule also uses a 'look-ahead' principle, and in this case produced superior results to the GA, and produced them much faster. In any particular problem the actual time needed for scheduling must also be considered, so whether there is any gain over FCFS in practice will of course be problem-dependent.

The problem investigated is rather straightforward, so in a sense it is not surprizing that a simple rule like SPT(all) performs well. We intend to investigate more complex problems where more sophisticated approaches such as GAs might be needed.

We should also emphasize that in real problems processing times are not always known in advance, although we may be able to predict them with a fair degree of accuracy. Genetic algorithms have been found effective for stochastic flowshop sequencing [7], and future work will also investigate the potential for using GAs in multi-processor systems which are both dynamic and stochastic.

References

1. CLEVELAND, G. A. – SMITH, S. F. (1989): Using Genetic Algorithms to Schedule Flow Shop Releases. In J. D. Schaffer (Ed.) (1989) *Proceedings of the 3rd International Conference on Genetic Algorithms.* Morgan Kaufmann, Los Altos, CA.
2. CARTWRIGHT, H. M. – MOTT, G. F. (1991): Looking Around: Using Clues from the Data Space to Guide Genetic Algorithm Searches. In R. K. Belew and L. B. Booker (Eds.) (1991) *Proceedings of the 4th International Conference on Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA.
3. REEVES, C. R. (1993): A Genetic Algorithm for Flowshop Sequencing. *Computers & Ops. Res.*, (to appear).
4. REEVES, C. R. – KARATZA, H. (1993): Dynamic Sequencing of a Multi-processor System: a Genetic Algorithm Approach. In R. F. Albrecht, C. R. Reeves and N. C. Steele (Eds.) (1993) *Proceedings of International Conference on Artificial Neural Networks and Genetic Algorithms,* Springer-Verlag, Vienna.
5. CONWAY, R. W. – MAXWELL, W. L. – MILLER, L. W. (1967): Theory of Scheduling. Addison-Wesley, Reading, Mass.
6. SAUER, C. H. – CHANDY, K. M. (1981): Computer Systems Performance Modelling. Prentice-Hall, New Jersey.
7. REEVES, C. R. (1992): A Genetic Algorithm Approach to Stochastic Flowshop Sequencing. *Proc. IEE Colloquium on Genetic Algorithms for Control and Systems Engineering.* Digest No. 1992/106, IEE, London.