Performance Evaluation of Heuristics for Scheduling Pipelined Multiprocessor Tasks

M. Fikret Ercan[1], Ceyda Oguz[2], and Yu-Fai Fung[3]

[1] School of Electrical and Electronic Engineering, Singapore Polytechnic, Singapore
mfercan@sp.edu.sg

[2] Department of Management, The Hong Kong Polytechnic University, Hong Kong S.A.R.
msceyda@polyu.edu.hk

[3] Department of Electrical Eng., The Hong Kong Polytechnic University, Hong Kong S.A.R.
eeyffung@polyu.edu.hk

Abstract. This paper presents the evaluation of the solution quality of heuristic algorithms developed for scheduling multiprocessor tasks in a class of multiprocessor architecture designed for real-time operations. MIMD parallelism and multiprogramming support are the two main characteristics of multiprocessor architecture considered. The solution methodology includes different techniques including simulated annealing, tabu search, as well as well-known simple priority rule based heuristics. The results obtained by these different techniques are analyzed for different number of jobs and machine configurations.

1 Introduction

In order to cope with the computing requirements in many real-time applications, such as machine vision, robotics, and power system simulation, parallelism in two directions, space (data or control) and time (temporal), are exploited simultaneously [4, 10]. Multitasking computing platforms are particularly developed to exploit this computing structure. These architectures provide either a pool of processors that can be partitioned into processor clusters or processor arrays prearranged in multiple layers. PASM [11], NETRA[2] or IUA [12] are the examples to such architectures. In both approaches a communication mechanism is provided among processor clusters to support pipelining of tasks. The computing platform achieves multi-tasking (or multiprogramming) by allowing simultaneous execution of independent parallel algorithms in independent processor groups. This class of computers is specially developed for applications where operations are repetitive. A good example to this computing structure is real-time computer vision, where overall structure is made of a stream of related tasks. Operations performed on each image frame can be categorized as low, intermediate and high level. The result of an algorithm in low level initiates another algorithm in intermediate level and so on. By exploiting available spatial parallelism, algorithms at each level can be split into smaller grains to reduce their computation

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2073, pp. 61-70, 2001.

time. In addition, when continuous image frames are processed, temporal parallelism can be exploited to improve computing performance even further. That is, algorithms at each level can be mapped to a processing layer (or cluster) of a multi-tasking architecture and executed simultaneously to create a pipelining effect. In the remainder of this paper, as well as in our problem definition, we will name a single pipeline, made of multiprocessing tasks (*MPT*), as a job.

In general, high performance parallel computing requires two techniques: program partitioning and task scheduling. Program partitioning deals with finding best grain size for the parallel algorithm considering the trade-off between parallelism and overhead. There are many techniques introduced in literature including simple heuristics, graph partitioning techniques, as well as meta-heuristics [1,4]. The main approach in these studies is to partition a task into subtasks considering network topology, processor, link, memory parameters and processor load balance to optimize the performance of computation. On the other hand, task scheduling deals with optimally scheduling *MPT*s so that overall makespan of the parallel application can be minimized. Various aspects of task scheduling have been studied in literature including deterministic and dynamic tasks, periodic tasks, preemptive, and non-preemptive tasks [1,2,4].

In different to these studies, we focus on job scheduling problem. As mentioned a job consists of multiple interdependent *MPT*s. The job scheduling problem is basically finding a sequence of jobs that can be processed on the system in minimum time. This problem as it stands is very complex; therefore we study a more restricted case in terms of computing platform and job parameters. In this paper, we consider jobs with deterministic parameters processed on a multi-tasking architecture with only two layers (or clusters). In our earlier study, we have developed list based heuristic algorithms especially for dynamic scheduling of jobs [6]. In the dynamic case, once a schedule is obtained it is implemented by control processors of the physical system. Most of the multi-tasking architectures employ a master-slave organization at each independent layer where master processor is responsible for initiating the processes. These heuristics provided fast solutions though their minimization of makespan were limited. On the other hand, for the deterministic cases scheduling can be done off-line during program compilation stage. This allows to employ more complex local search algorithms such as simulated annealing, tabu search, and genetic algorithms. These algorithms typically search for improved solutions until a stopping criterion is reached. It is most likely to find a better solution with these algorithms though their execution time is long due to their iterative nature. In this paper, we study simulated annealing and tabu search algorithms and evaluate their performances. In the following, a formal definition of the problem, simulated annealing and tabu search algorithms, and computational studies will be presented.

2 Basic Parameters and Problem Definition

We consider a set J of n independent and simultaneously available jobs to be processed in a computing platform with two multiprocessor layers where layer j has m_j

identical parallel processors, $j = 1,2$. The level of pipeline in each job is the same and compatible with the number of processing layers available in the computing platform. Each job J_i J has two multiprocessor tasks (*MPT*s), namely $(i,1)$ and $(i,2)$. $MPT(i, j)$ should be processed on $size_{ij}$ number of processors simultaneously at layer j for a period of p_{ij} without interruption ($i = 1,2,...,n$ and $j = 1,2$). Hence, each $MPT(i, j)$ is characterized by its processing time, p_{ij}, and its processor requirement, $size_{ij}$ ($i = 1,2,...,n$ and $j = 1,2$). All the processors are continuously available from time 0 onwards and each processor can handle no more than one *MPT* at a time. Jobs flow through from layer 1 to layer 2 by utilizing any of the processors and by satisfying the *MPT* constraints. The objective is to find an optimal schedule for the jobs so as to minimize the maximum completion time of all jobs, i.e. the makespan, $C_{\max}$.

As in most allocation methods, we assume that processors are capable of simultaneously executing a task and performing a communication. This assumption is also based on the practical fact that majority of novel parallel architectures possess such feature. In our computations, communication cost between the subtasks is considered, though, for the sake of simplicity, this cost is included in p_{ij} as part of the total time that processors are occupied while performing a task.

3 Task Mapping Heuristic

Task mapping heuristic allocates tasks from a given job list by simply evaluating processor availability of the underlying hardware and requirements of *MPT*s. The algorithm performs following steps:

Step 1. Given a sequence S of the jobs, construct a schedule in layer 1 by assigning the first unassigned $MPT(i,1)$ of job J_i in S to the earliest time slot where at least $size_{i1}$ processors are available.

Step 2. As the *MPT*s are processed and finished in layer 1 in order, their counterpart became available to be processed in layer 2. Hence, schedule available *MPT*s to the earliest time slot in layer 2 by also taking into account their sequence in S.

4 Simulated Annealing

The stochastic methodologies can be used to improve the quality of allocations. Simulated annealing [9], *SA*, is an example to such methods. It performs heuristic hill climbing to transverse a search space in a manner, which is resistant to stopping prematurely at local critical points that are less optimum than the global one. As it is

known, in order to achieve this, *SA* scheme moves from one solution to another with the probability defined by the following equation:

$p(n) = \exp \quad \frac{E}{T(n)}$ where E is the difference in cost between the current solution and the new solution and $T(n)$ is a control parameter, which is also called 'temperature' at step n. A new state is accepted whenever its cost, or energy function, is better than the one associated with the previously accepted state. T is analogous to temperature associated with physical processes of the annealing. In general, T, is initialized with the value T_{init} and is then decreased in the manner dictated by the associated cooling schedule until it reaches the freezing temperature.

In order to apply *SA* to a practical problem several decisions have to be made. Next, we present our approach for each of these decisions.

Initial Solution: The initial solution is generated by setting all jobs in ascending order of job indices.

Neighborhood generation mechanism: A neighbor of the current solution is obtained in various ways. One method is to exchange two randomly chosen jobs from the priority list. This method is called *interchange neighborhood*. A special case of interchange neighborhood is simple *switch neighborhood*. It is defined by exchanging a randomly chosen job with its predecessor. Third method is called *shift neighborhood*, which involves removing a randomly selected job from one position in the priority list and putting it into another randomly chosen position. We have employed a preliminary computational experiment to examine the performance of these three methods. The results showed that the best performing neighborhood generation mechanism is *interchange* method. It is followed by *shift* and *simple switch* methods. Hence, *interchange* method is employed in our further experiments.

Objective function: The value of the objective function is defined as minimal value obtained for the completion time of all jobs, i.e. the makespan, $C_{\max}$.

Cooling Strategy: A simple cooling strategy is employed in our implementation. Temperature is decreased in an exponential manner with $T_i = \quad T_{i\ 1}$ where < 1. In our implementation, value is selected as 0.998 after repetitive experiments.

Initial Temperature: It is important to select an initial temperature high enough to allow a large number of probabilistic acceptances. The initial value of temperature is selected using the formula: $T_o = \frac{\overline{E}_{avg}}{\ln(x_0)}$. Here $\overline{E}_{avg}$ is the average increase in the cost for a number of random transitions. Initial acceptance ratio, x_0, is defined as the number of accepted transitions divided by the number of proposed transitions. These parameters estimated after 50 randomly permuted neighborhood solution of the initial solution.

Stopping criterion: We have employed two stopping rules simultaneously. The first rule is the fixed number of iterations. The second rule compares average performance

deviation of the solution from the lower bounds and if it is less then 1% procedure is ended.

5 Tabu Search

Tabu search, *TS*, is another local search method, which is guided by the use of adaptive memory structures [7]. This method has been successfully applied to obtain optimal or sub-optimal solutions to optimization problems. The basic idea of the method is to explore the solution space by a sequence of moves made from one solution to another solution. However, to escape from locally optimal solutions and to prevent cycling, some moves are classified as forbidden or tabu. In the basic short term strategy of *TS*, if there is no better solution found than the current one, s_n, a move to the best possible solution, s, in the neighborhood $N(s_n)$ (or a sub-neighborhood $N'(s_n)$ $N(s_n)$ in the case $N(s_n)$ is too large to be explored efficiently) is performed. A certain number of the last visited solutions are stored in tabu list such that if a solution s is already in the list, the move from current solution (s_n s) is prohibited.

One of the main decision areas of *TS* is specification of a neighborhood structure and possibly of a sub-neighborhood structure. The three neighborhood generation strategies, discussed earlier in *SA* section, are also experimented with *TS* and interchange strategy is found to be the most effective one. For the sub-neighborhood $N'(s_n)$, we pick at random a fixed number of solutions in $N(s_n)$.

In the tabu list, we keep a fixed number of last visited solutions. We have also experimented keeping track of moves made instead of the solution sets. In this case, the computation time was shorter though we did not observe any significant advantage over the solution provided. We have experimented two methods for updating tabu list. These are the elimination of the farthest solution stored in the list, and removing the worst performing solution from the list. For the second method, an additional list to keep makespan values of the solutions in tabu list is required since the performance of a solution is measured with the makespan. This method resulted in slightly better performance than the first one. However, the tactical choices to improve the efficiency of the *TS* algorithm are somewhat longer than the *SA* and for this problem case performance of *TS* algorithm with the standard choices were slightly behind the *SA*.

6 Computational Experiments

Our computational study aims to analyze performance of the *SA* and *TS* methods on the minimization of makespan, as well as to investigate the effect of task characteris-

tics, and processor configurations on the performance. We consider different processing time ratios and different processor configurations for the randomly generated problems as explained below. In order to make sure a comparable computational effort committed by each heuristic, the stopping criterion for the following experiments defined as a fix number of solutions visited. This number has been set at 5000. We also compared these results with our earlier study where we have analyzed performance of several list-based heuristics for the job-scheduling problem.

The number of jobs was selected as $n = 10,30,50$. We have selected following two processing time ratios as defined in [10]. These are a) $p_{i1} \sim U[1,40]$ and $p_{i2} \sim U[1,40]$ b) $p_{i1} \sim U[1,40]$ and $p_{i2} \sim U[1,20]$ $(i = 1,2,...,n)$. The number of processors of multi-layer system was chosen according to following two configurations: a) More processors at layer 1, $m_1 = 2m_2 = 2^k$; b) Identical number of processors at each layer, $m_1 = m_2$; where $k = 1,2,3.$

For every $MPT(i, j)$, an integer processor requirement at layer j was generated from a uniform distribution over $[1, m_j]$ ($i = 1,2,...,n$ and $j = 1,2$). For each combination of processing time ratio and processor configuration of the architecture 25 problems were generated which are used to test the performance of *SA* and *TS* algorithms. In this section, we present the results of our computational study. For comparison, we have also included the performance of four best performing priority based heuristic algorithms from our earlier study where we have experimented 48 different heuristics that are a combination of 24 sequencing rules and two task mapping heuristics. The first heuristic algorithm, *H1*, obtains a sequence of jobs by applying Johnson's algorithm, *JA*, [8] assuming that $size_{ij} = m_j = 1$ ($i = 1,2,...,n$ and $j = 1,2$). Whereas, in the second heuristic algorithm, *H2*, a sequence of jobs obtained by first sorting tasks in non-increasing order of layer 2 processor requirements and then by sorting each group of tasks requiring same number of processors in non-increasing order of their layer 2 processing times. The sequencing rule in the third algorithm, *H3*, obtains a job list by simply sorting tasks in non-increasing order of layer 2 processing times. In heuristic *H4*, a set of job sequence is obtained by sorting the tasks in non-increasing order of $p_{i1}size_{i1} + p_{i2}size_{i2}$. In addition, we have also included the result of a heuristic based on random selection of jobs. All the algorithms are implemented using C++ and run on a PC with 350 Mhz Pentium II processor. Results are presented in terms of Average Percentage Deviation (APD) of the solution from the lower bound. The percentage deviation is defined as $((C_{\max}(HE) \quad LB)/LB) \quad 100$ where $C_{\max}(HE)$ denotes the $C_{\max}$ obtained by heuristic algorithms, that is SA, TS or list based heuristics. LB indicates the minimum of five lower bounds used [6]. The APD of each solution are presented in Figures 1 and 2.

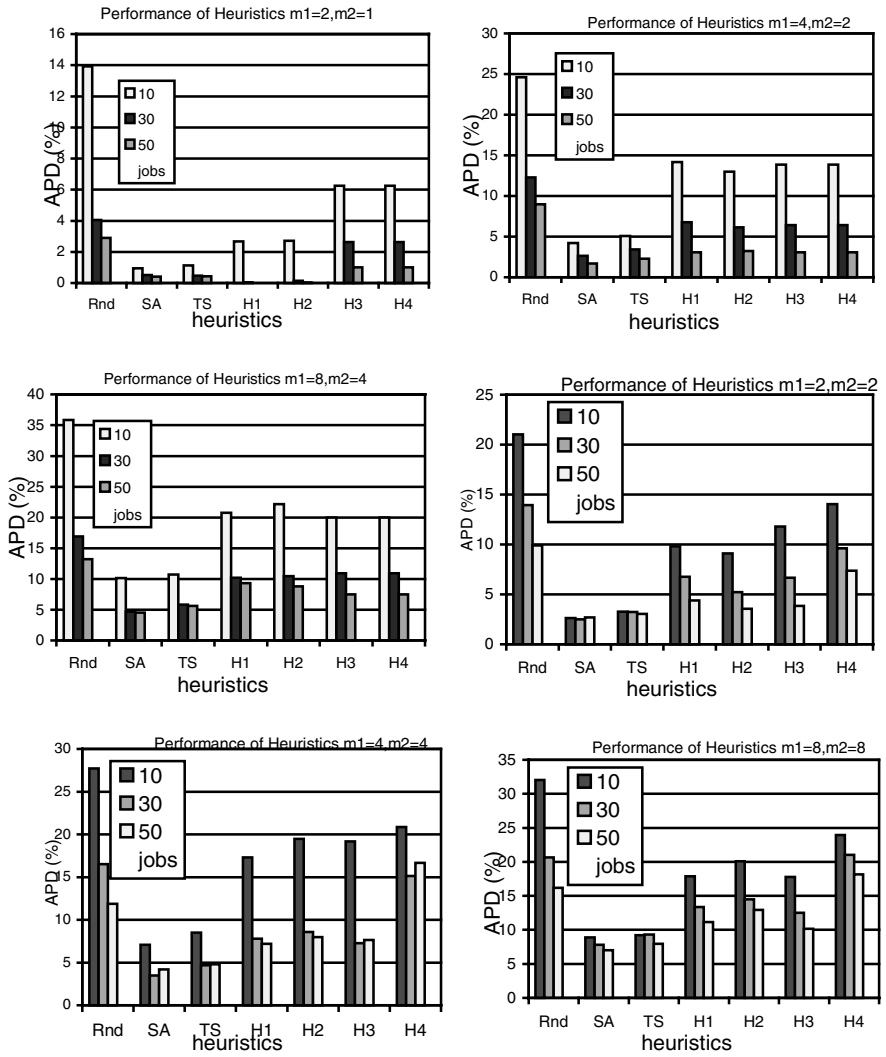

Fig. 1. Average percentage deviation of each algorithm for *P1*:*P2*=40:40.

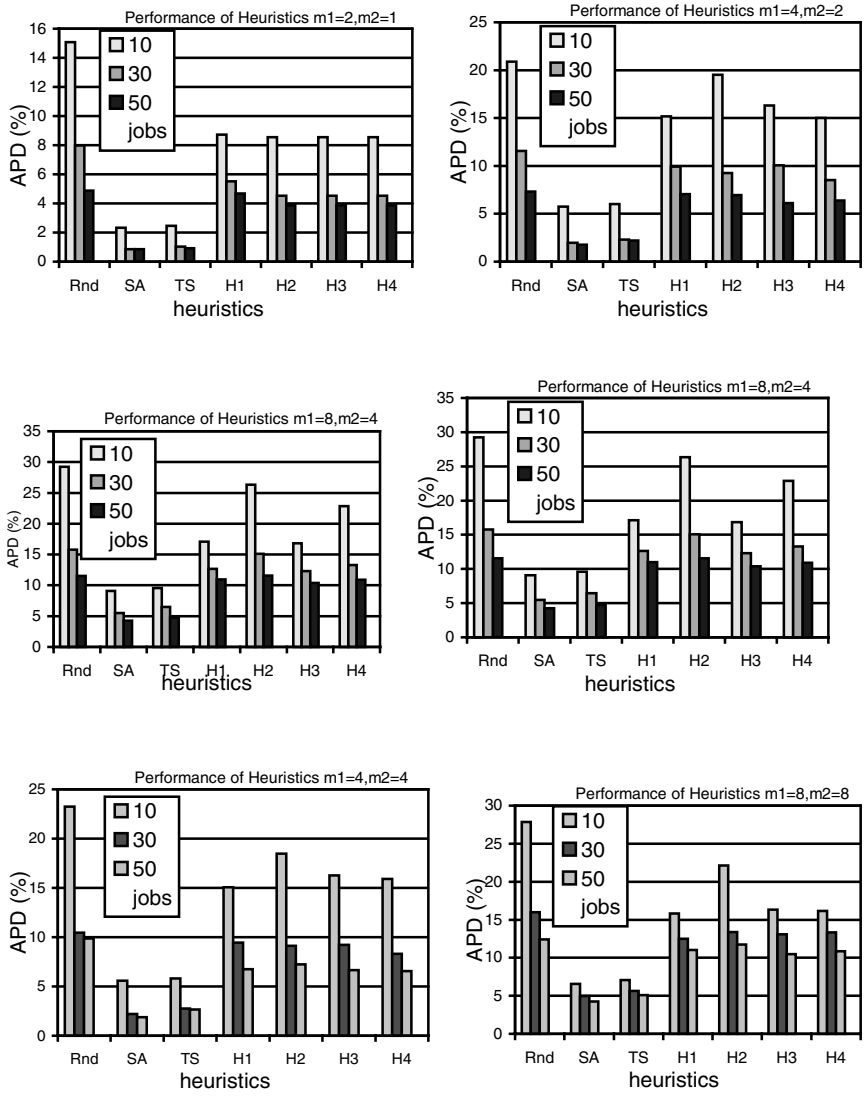

Fig. 2. Average percentage deviation of each algorithm for *P1*:*P2*=40:20.

The computational study shows that in all the cases *SA* and *TS* significantly outperform random sampling heuristic. In all the experiments, these metaheuristics delivered a better solution than random sampling ranging as high as 81 percent and as minimum as 14.5 percent. In none of the experiments, random selection encounters a solution at lower bounds or closer. The makespan minimization achieved by both SA and TS are quite similar, though, in most of the cases SA delivers a better result. In most of the

cases, SA converges to a reasonable solution within 500 iterations while TS converges within 1000 iterations.

It is also observed that in general the APD results of algorithms are better for the processing time ratio of 40:20 than the ratio 40:40. This can be explained as having a larger range for the main characteristic of the problem makes it difficult to schedule tasks, as it is more likely to have unbalanced processor loads. From Figures 1 and 2, APD seems to decrease as the number of jobs increases for each heuristic algorithm. This is explained as the number of job increases lower bound becomes more effective and close to the optimal solution. APD also deteriorates with the increasing number of processors. In addition, with increasing layer 2 to layer 1 processor ratio APD deteriorates. As layer 2 processors dictates the completion time of jobs, the increase of number of processors at this layer also increases the possibility of having idle processors, which consequently reduces the efficiency.

7 Summary

In this paper, a job-scheduling problem on a multi-tasking multiprocessor environment is considered. A job is made of interrelated multiprocessor tasks, which are modeled with their processing requirements and processing times. Two metaheuristic algorithms have been applied for the solution and their performance have been evaluated based on their capacity to minimize makespan. We compared these results with our earlier study where we have developed heuristic algorithms using simple sequencing rules. The results showed that metaheuristics significantly outperformed the list based heuristics. However, due to their large computation times they can be used in deterministic cases. So far, we have considered restricted case of the problem; a more general case will be dealt in our further study.

References

1. Blażewicz J., Ecker K. H., Pesch E., Schmidt G. and Weglarz J., Scheduling Computer and Manufacturing Processes, Springer-Verlag, Berlin, 1996
2. Bokhari S. H., Assignment Problems in Parallel and Distributed Computing, Kluwer Academic, Boston (1987)
3. Choudhary A. N., Patel J. H and Ahuja N., NETRA: A Hierarchical and Partitionable Architecture for Computer Vision Systems, IEEE Trans. Parallel and Distributed Systems, Vol. 4, (1996) 1092-1104
4. El-Revini H., partitioning and and scheduling, in:A.Y.H. Zomaya (ed.), Parallel and Distrib-uted Computing Handbook, McGraw-Hill, New York, (1996) 239-273

5. Ercan M. F. and Y. F. Fung, Real-time Image Interpretation on a Multi-layer Architecture, IEEE TENCON'99, Vol. 2, (1999) 1303-1306.
6. Ercan M.F., C. Oguz and Y. F. Fung, Scheduling Image Processing Tasks in A Multi-layer System, in print Computers and Electrical Engineering.
7. Glover F. and Laguna, Tabu search, Kluwer Academic Publishers, Boston, (1997)
8. Johnson S. M., Optimal Two and Three-stage Production Schedules with Setup Times Included, Naval Research Logistic Quarterly, Vol. 1, (1954) 61-68
9. Kirkpatrick S., Gelatt, C. D., and Vecchi M. P., Optimization by Simulated Annealing, Science, Vol. 220, (1983) 671-680
10. Lee C.Y and Vairaktarakis G.L., Minimizing Make Span in Hybrid Flow-Shops, Operations Research Letters, Vol. 16, (1994) 149-158
11. Scala M. L., Bose A., Tylavsky J., and Chai J.S., A Highly Parallel Method for Transient Stability Analysis, IEEE Transactions on Power Systems, Vol. 5, (1990) 1439-1446
12. Siegel H.J., Siegel L.J., Kemmerer F.C., Mueller P.T., Smalley H.E., and Smith S.D., PASM- A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition, IEEE Trans. Computers, Vol. C-30, (1981) 934-947
13. Weems C.C., Riseman E.M. and Hanson A.R., Image Understanding Architecture: Exploiting Potential Parallelism in Machine Vision, IEEE Computer, Vol. 25 (1992) 65-68