2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies

Optimal Scheduling of Task Graphs on Parallel Systems

Ahmed Zaki Semar Shahul
Hewlett-Packard New Zealand
22 Viaduct Harbour Avenue, Maritime Square
PO Box 3860, Shortland Street
Auckland, New Zealand
ahmedzakis@yahoo.com

Oliver Sinnen
Department of Electrical and Computer Engineering
University of Auckland
Private Bag 92019
Auckland, New Zealand
o.sinnen@auckland.ac.nz

Abstract

Scheduling tasks onto the processors of a parallel system is a crucial part of program parallelisation. Due to the NP-hard nature of the task scheduling problem, scheduling algorithms are based on heuristics that try to produce good rather than optimal schedules. Nevertheless, in certain situations it is desirable to have optimal schedules, for example for time critical systems or to evaluate scheduling heuristics. This paper investigates the task scheduling problem using A search algorithm. The A* scheduling algorithm implemented can produce optimal schedules in reasonable time for small to medium sized task graphs. In comparison to a previous approach, the here presented A* scheduling algorithm has a significantly reduced search space due to a much improved cost function $f(s)$ and additional pruning techniques. Last but not least, the experimental results show that the proposed A* scheduling algorithm significantly outperforms the previous approach.*

KEY WORDS
Parallel computing, scheduling, task graphs, optimal schedules, A*

1. Introduction

To realise the potential of a parallel system, computer applications need to be parallelised. Parallelisation of an application involves the decomposition of the program into several (sub)tasks, analysing the dependencies between the (sub)tasks and scheduling the (sub)tasks. The assignment of tasks to the processing units (spatial assignment) and defining their execution order (temporal assignment) statically (at compile-time) is referred to as task scheduling [6, 7]. Task scheduling is crucial to the performance and efficiency of the application. Unfortunately, task scheduling in its general form is an NP-hard problem, e.g. [4], i.e. finding an optimal solution takes exponential time unless NP = P [6, 8]. Hence, several heuristics such as list scheduling exist to tackle the problem of task scheduling which generally produce "good" results. For these heuristic algorithms, the program is modelled as a Directed Acyclic Graph (DAG), called a task graph. Nevertheless, there are several scenarios where a "good" solution will not suffice and there is a necessity for an optimal solution. For instance, in time critical systems where performance is essential and for evaluation of the quality of schedules produced by heuristics.

An optimal scheduling algorithm based on A* search algorithm has been proposed in [1] for the problem of task scheduling. Following that, a preliminary A* scheduling algorithm which produced optimal schedules for small to medium sized task graphs was presented recently in [7]. This paper employs an even better cost function and introduces new pruning and optimisation techniques. In addition, results from extensive experiments are presented that demonstrate the effectiveness of the proposed algorithm.

The rest of the paper is organised as follows. Section 2 defines the basics of the task scheduling model. In Section 3, the proposed A* scheduling algorithm is presented with consideration to previous work in [1]. The improved cost function is presented in Section 4. Pruning techniques are discussed in Section 5. The experimental evaluation of the new scheduling algorithm is presented in Section 6. Conclusions and future work are presented in Section 7.

2. Task scheduling model

A program $\mathcal{P}$ to be scheduled is represented by a Directed Acyclic Graph (DAG), $G = (\mathbf{V}, \mathbf{E}, w, c)$, called a task graph, where $\mathbf{V}$ is the set of nodes representing the (sub)tasks of $\mathcal{P}$ and $\mathbf{E}$ is the set of edges representing the communications (dependencies) between the tasks. An edge $e_{ij} \in \mathbf{E}$ represents the communication from node n_i

978-0-7695-3443-5/08 $25.00 © 2008 IEEE
DOI 10.1109/PDCAT.2008.54

to node n_j where $n_i, n_j \in \mathbf{V}$. The positive weight $w(n)$ of node $n \in \mathbf{V}$ represents its computation cost and the non-negative weight $c(e_{ij})$ of edge $e_{ij} \in E$ represents its communication cost. Figure 1 (a) illustrates a task graph with four nodes.

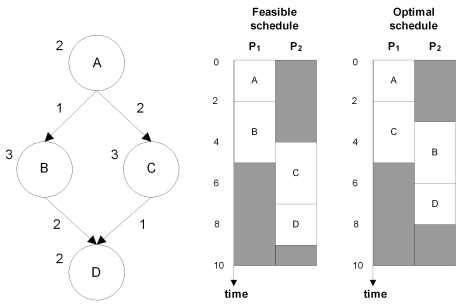

Figure 1. Task scheduling example. (a) A task graph with four nodes. (b) A feasible (left) and an optimal (right) schedule with a schedule length of nine and eight time units respectively

To achieve a *complete schedule* for a task graph G onto a target parallel system with a set of processors $\mathbf{P}$, each node $n \in \mathbf{V}$ must be associated with a start time $t_s(n)$, i.e. temporal assignment, and be assigned to a processor $proc(n) = P$, where $P \in \mathbf{P}$, i.e. spatial assignment. The following classical assumptions [6] are made about the target parallel system: i) the system is dedicated, i.e. no other program is executed while G is executed; ii) tasks are non-preemptive; iii) local communication is cost free; iv) there is a communication subsystem, i.e. processors are not involved in communication; v) communications can be performed concurrently, i.e. there is no contention for communication resources; vi) the processors are fully connected; vii) the processors are identical.

The following graph definitions are based on the definitions in [5, 6]: i) *finish time* t_f, of any node $n \in \mathbf{V}$ can be defined as its start time plus its computation cost. Formally, $t_f(n) = t_s(n) + w(n)$; ii) *Processor constraints* ensure that any processor in the parallel system can execute only one task at a time; iii) *Precedence constraints* impose the execution order of the nodes in light of their communications (dependencies), i.e. for $n_i, n_j \in \mathbf{V}$, $e_{ij} \in \mathbf{E}$ and $i \neq j$:

$$t_s(n_j) \geq t_f(n_i) + \begin{cases} 0 & if\ proc(n_i) = proc(n_j) \\ c(e_{ij}) & otherwise \end{cases}. \quad (1)$$

iv) The *schedule length*, *sl,* of a complete and feasible schedule $\mathcal{S}$, i.e. the execution time of G, is the finish time of the last node assuming that the first node starts at time unit 0, i.e. $sl(\mathcal{S}) = \max_{n \in \mathbf{V}}\{t_f(n)\}$; v) The *computation critical path* is a longest computation path p in a task graph G, i.e. $len_w(cp_w) = \max_{p \in G}\{len_w(p)\}$ where $len_w(p) = \sum_{n \in p, \mathbf{V}} w(n)$; vi) The *computation bottom level* of a node n, $bl_w(n)$, is defined as the computation length of the longest path p leaving n. For instance, $bl_w(B) = 5$ in Fig. 1 (a). The computation critical path and computation bottom level also provide a lower bound on the schedule length, i.e.

$$sl(\mathcal{S}) \geq len_w(cp_w) \quad (2)$$

$$sl(\mathcal{S}) \geq t_s(n) + bl_w(n)\ \forall n \in \mathbf{V}. \quad (3)$$

Figure 1 (b) shows a feasible (left) and an optimal (right) schedule for the task graph represented on two processors. Finding the optimal schedule for a given task graph and for a given number of processors is the NP-hard problem addressed in this paper.

3. A* search algorithm

A state space represents the abstract space of solutions which must be searched to find an optimal solution for a problem. In [1], it was proposed to use the A* algorithm for the problem of task scheduling such that each state represents a schedule. The initial state is an empty schedule and the goal state is a complete schedule. New states are created from a state s by taking the partial schedule represented by s and scheduling all free nodes to every available processor, i.e. each node-processor pairing gives rise to one new state. A node n_j is free to be scheduled if all of its predecessors ($\mathbf{pred}(n_j) = \{n_i \in \mathbf{V} : e_{ij} \in \mathbf{E}\}$) are already part of the partial schedule $\mathcal{S}$ of s, i.e. $\mathbf{free}(s) = \{n_j \in \mathbf{V} : n_j \notin \mathcal{S}(s) \wedge \mathbf{pred}(n_j) \subseteq \mathcal{S}(s)\}$. Thus, the maximum number of new states that can be created from s is:

$$new(s) = |\mathbf{free}(s)| \times |\mathbf{P}|. \quad (4)$$

This indicates that the state space can explode in size very quickly. Hence, it is vital to prune the state space to ensure the efficiency of the scheduling algorithm [5].

A* algorithm maintains two lists: OPEN and CLOSED. It takes the state s with the lowest f value from the OPEN list and expands new states that are larger (partial) solutions. An f value is calculated for each of these new states and they are placed into the OPEN list provided that they are neither present in the CLOSED nor OPEN lists already. At the end of the step, state s is put into the CLOSED list. The algorithm terminates when the state that is taken out of the OPEN list is a goal state, i.e. a complete schedule. The A* scheduling algorithm is detailed in [5].

The A* algorithm guides its search through the state space using the $f(s)$ function. The exact minimum cost of a solution from the initial state to the goal state via a state s is denoted by $f^*(s)$. An $f(s)$ function is *admissible*, i.e. an underestimate of $f^*(s)$, if it satisfies the condition $f(s) \leq f^*(s)$ for any state s. The informativeness of the $f(s)$ function determines the number of states that the A* algorithm examines. A well informed $f(s)$ function is a function that is close to $f^*(s)$, i.e. if $f_1(s) < f_2(s)$ then $f_2(s)$ is strictly more informed (provided it is admissible) and will result in less states being examined [3]. Ideally, $f(s) = f^*(s)$. If $f_1(s) \leq f_2(s)$, then $f_2(s)$ can possibly examine more states than $f_1(s)$. However, note that $f_2(s)$ is still more informed (not strictly) than $f_1(s)$. Another desirable property of the cost function, $f(s)$, is *consistency* (also called *monotonicity*). A consistent cost function ensures non-decreasing f values along any path in the state space [2]. The A* algorithm is guaranteed to find an optimal solution with an admissible and consistent $f(s)$ function [5].

The $f(s)$ function proposed in [1] for the task scheduling problem, here designated as $f_{KA}(s)$, is calculated as follows: Let n_{max} be a node in the partial schedule $\mathcal{S}$ associated with s that has the latest finish time $n_{max} = n \in \mathcal{S}$: $t_f(n) = \max_{n_i \in \mathcal{S}(s)} \{t_f(n_i)\}$:

$$f_{KA}(s) = t_f(n_{max}) + \max_{n \in \mathbf{succ}(n_{max})} \{bl_w(n)\}. \quad (5)$$

As per the definition of computation bottom level, this is identical to the start time of n_{max} plus its computation bottom level, i.e. $f_{KA}(s) = t_s(n_{max}) + bl_w(n_{max})$. The $f_{KA}(s)$ function is clearly admissible but not consistent. In this paper, an improved cost function that is admissible and consistent is presented. In addition, several pruning and optimisation techniques have been implemented to the A* scheduling algorithm to improve the efficiency of the implementation.

4. Proposed cost function

The proposed cost function $f(s)$ is derived from three components: the computation bottom level, idle time and data ready time.

4.1. Computation bottom level

This component of the $f(s)$ function is defined as follows:

$$f_{bl_w}(s) = \max_{n \in \mathcal{S}} \{t_s(n) + bl_w(n)\}. \quad (6)$$

$f_{bl_w}(s)$ is clearly admissible considering the lower bound on the schedule length in (3). It is also more informed than $f_{KA}(s)$ as the latest finishing node in a partial schedule does not necessarily have the highest computation bottom level. The definition of $f_{bl_w}(s)$ above can be redefined as

$$f_{bl_w}(s) = \max\{f_{bl_w}(s_{parent}), t_s(n_{last}) + bl_w(n_{last})\} \quad (7)$$

where n_{last} is the last node that was added to the partial schedule of s and s_{parent} is the parent state of s. This is equivalent to (6) but only the last node n_{last} has to be considered for each new state. Hence, the calculation complexity of $f_{bl_w}(s)$ is $O(1)$ which is the same as for $f_{KA}(s)$ [5].

4.2. Idle time

Due to precedence constraints, any given schedule can have idle times, i.e. times between the execution of two consecutive nodes (or before the execution of the first node) where a processor runs idle. For example, the optimal schedule (right) in Figure 1 (b) has an idle period of three time units on processor P_2 before node B is executed. Let $idle(\mathcal{S})$ be the total idle time, i.e. the sum of all idle periods of all processors of a schedule $\mathcal{S}$. The schedule length is then bounded by the sum of $idle(\mathcal{S})$ and the total weight (execution time) of all nodes divided by the number of processors. Therefore,

$$f_{IT}(s) = \left(idle(\mathcal{S}) + \sum_{n \in \mathbf{V}} w(n) \right) / |\mathbf{P}|. \quad (8)$$

$f_{IT}(s)$ never decreases for a larger partial schedule and it can be calculated incrementally leading to a calculation complexity of $O(1)$ [5].

4.3. Data ready time

The *data ready time* (DRT) of a node $n_j \in \mathbf{V}$ on processor P is

$$t_{dr}(n_j, P) = \max_{n_i \in \mathbf{pred}(n_j)} \left\{ t_f(n_i) + \begin{cases} 0 & if\ proc(n_i) = proc(n_j) \\ c(e_{ij}) & otherwise \end{cases} \right\}. \quad (9)$$

The minimum DRT of a node $n \in \mathbf{V}$ in $\mathbf{P}$ is $t_{dr}(n) = \min_{P \in \mathbf{P}} \{t_{dr}(n, P)\}$. If $\mathbf{pred}(n_j) = \emptyset$, i.e. n_j is a source node, then $t_{dr}(n_j) = t_{dr}(n_j, P) = 0\ \forall P \in \mathbf{P}$ [6]. DRT introduces a lower bound on the start time of a node n, i.e. $t_s(n) \geq t_{dr}(n)$. The DRT component of the $f(s)$ function, $f_{DRT}(s)$, can be expressed as:

$$f_{DRT}(s) = \max_{n \in \mathbf{free}(s)} \{t_{dr}(n) + bl_w(n)\}. \quad (10)$$

In Fig. 2, $t_{dr}(D)$ is five. Hence, the f value of a state representing the partial schedule shown can be increased to

seven $(t_{dr}(D) + bl_w(D))$ using $f_{DRT}(s)$. The complexity of calculating $f_{DRT}(s)$ is $O(1)$ just like $f_{KA}(s)$, with the reasonable assumption for task graphs that $O(\mathbf{E}) = O(\mathbf{V})$ [5].

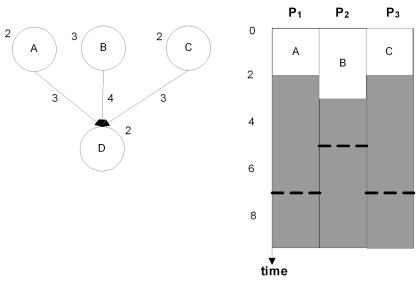

Figure 2. Using the data ready time method to calculate the f value of a state. Dotted lines indicate the DRT of free node D on each processor

The preceding three components can be combined to give the following:

$$f(s) = \max\{f(s_{parent}), f_{bl_w}(s), f_{IT}(s), f_{DRT}(s)\}. \quad (11)$$

This maximising approach ensures that the $f(s)$ function remains consistent. The f value of the initial state, $f(s_{init})$, representing an empty schedule can also be increased from zero (as suggested in [1]) to the following:

$$f(s_{init}) = \max\left(\left(\sum_{n \in \mathbf{V}} w(n)\right)/|\mathbf{P}|, len_w(cp_w)\right). \quad (12)$$

Hence, the proposed $f(s)$ function is more informed than $f_{KA}(s)$, i.e. $f_{KA}(s) \leq f(s) \leq f^*(s)$.

5. Pruning techniques

Pruning is a process that eliminates unpromising subtrees in the search space. A requirement on any pruning technique is that it does not jeopardise the ability of the algorithm to find an optimal solution. The advantages of pruning are reduced memory consumption of the A* scheduling algorithm and the reduction in the number of states that need to be expanded thereby improving the efficiency of the scheduling algorithm.

5.1. Processor normalisation

Two processors are isomorphic if they are both empty, i.e. no nodes are already scheduled to them [1]. Each free node needs only to be scheduled to one of all isomorphic processors. For fully connected and homogeneous target systems, the concept of processor isomorphism can be generalised. To illustrate this, consider the scheduling of two independent nodes A and B. Let $\mathcal{S}_1$ be a schedule where A is on processor P_1 and B on P_2, and let $\mathcal{S}_2$ be a schedule where B is on processor P_1 and A on P_2. The start times of A and B are the same in both schedules and there are no other nodes. Schedule $\mathcal{S}_2$ becomes identical to $\mathcal{S}_1$, when the processors are renamed from P_1 to P_2 and from P_2 to P_1 in $\mathcal{S}_2$.

To benefit from this observation, it is proposed here to normalise the processor names according to the nodes which are assigned to them. Having the nodes in a fixed order, the processor to which the first node is assigned to is named P_1. The processor of the next node, that is not on P_1, is named P_2 and so on. For partial schedules, unscheduled nodes are simply skipped. This process normalises any permutation of the processor names to a single one. After the normalisation, duplicates can be easily eliminated. Note that processor name normalisation can also be used in scheduling algorithms based on stochastic search techniques, e.g. Genetic Algorithms (GA) [9], in order to reduce the size of the search space.

5.2. Partial expansion

The partial expansion pruning technique schedules each free node to all the target processors and once the f value of any of the new states that are created is found to be equal to that of s then the rest of the free nodes are not scheduled for the time being. The rationale is that s has been chosen for expansion by the A* scheduling algorithm as it has the lowest f value in the state space. Therefore, any new state that is created with the same f value as s is also bound to have the lowest f value in the state space. Hence, there is no need to exhaustively schedule all the remaining free nodes in this iteration. It is to be noted that s is left in the OPEN list until it is completely expanded, i.e. all of its free nodes have been scheduled. Partial expansion minimises the explosion of states that can otherwise occur. This pruning technique is most effective when $f(s) = C^*$, where C^* is the cost of the optimal solution path to the goal.

5.3. Node equivalence

The number of states created by the A* algorithm can be reduced when equivalent nodes are detected [1]. Two nodes n_i and n_j are said to be equivalent when the following conditions are true: i) $w(n_i) = w(n_j)$, they have the same computation costs; ii) $\mathbf{pred}(n_i) = \mathbf{pred}(n_j)$, they have the same set of predecessors; iii) $\mathbf{succ}(n_i) = \mathbf{succ}(n_j)$, they have the same set of successors; iv) $c(e_{pi}) = c(e_{pj})\ \forall n_p \in \mathbf{pred}(n_i)$, the corresponding edge weights to the predecessors are equal; v) $c(e_{is}) = c(e_{js})\ \forall n_s \in \mathbf{succ}(n_i)$, the

corresponding edge weights to the successors are equal.

When equivalent nodes are present in the list of free nodes then only one order of the equivalent nodes needs to be considered as the new states created from it would be representative of the states obtained from scheduling the equivalent nodes in any order. Hence, an explosion of states is avoided. This pruning technique is implemented by first pre-analysing the task graph to detect all the equivalent nodes. Note that the five conditions listed in [1] for node equivalence do not guarantee that an optimal solution is found [5].

In general, pruning techniques can potentially change the order in which the nodes are expanded and this in turn can increase the number of states examined. However, the benefits obtained due to the pruning techniques outweigh the adverse effects.

6. Experimental results

The evaluation of the proposed task scheduling algorithm was carried out on a PC running at 1.86 GHz. The scheduling algorithm was implemented using Java and the Java Virtual Machine was allocated 2.5 GB of heap space.

To comprehensively evaluate the implemented task scheduling algorithm, a large set of task graphs differing in various properties such as the task graph structure, number of nodes and density were generated. All the task graphs in the workload were scheduled on a two, four and eight processor target machine. A timeout parameter with a practical value of one minute was used to terminate runs. A task graph run that took more than a minute is referred to as a "timeout" and the runs that did not timeout are referred to as "complete" runs. Note that the data analysed in this section only includes runs that completed in both the implementations that are compared. The results presented in this paper are representative and a comprehensive set of results are presented in [5].

The measure used to evaluate the implemented A* scheduling algorithm and the pruning techniques is the total number of created states, i.e. the total number of states that are created during a run of the algorithm, which is hardware and implementation independent. Another beneficial property of this measure is that the number of states created is roughly proportional to the runtime of the algorithm for the workload used [5].

6.1. Cost function evaluation

The $f(s)$ function proposed in (11) is compared with the $f_{KA}(s)$ function in (5). Figure 3 shows the average percentage reduction in the number of created states when using the proposed $f(s)$ function instead of the $f_{KA}(s)$ function. As can be observed, the new $f(s)$ function reduces the number of created states dramatically for the task graphs in the workload. The new $f(s)$ function is apparently a much closer underestimate of $f^*(s)$, i.e. more informed.

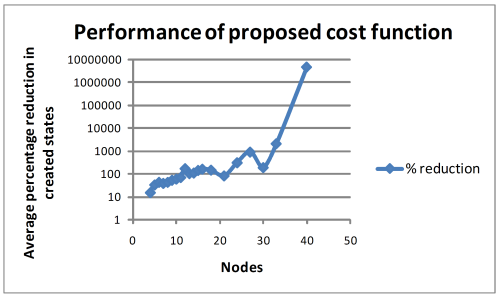

Figure 3. Comparing performance of proposed cost function $f(s)$ to $f_{KA}(s)$ function

6.2. Processor normalisation

An implementation of the A* scheduling algorithm that uses the processor normalisation pruning technique is compared to the same implementation bar the pruning technique. Figure 4 shows three curves, one for the workload scheduled on two, four and eight processors respectively. As expected, the reduction in the number of states created is most dramatic for eight processors and least significant for two processors. The significant reduction in the number of states created indicate that processor normalisation pruning technique is effective for the workload used.

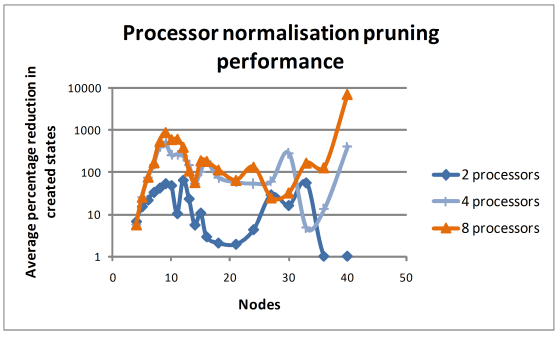

Figure 4. Reduction in the number of created states when using processor normalisation pruning technique

6.3. Partial expansion

Experimental results show that partial expansion decreased the number of states created for a vast majority of

runs and increased the number of states created for a mere 1.6% of the runs. Overall, partial expansion decreased the number of states created by 46%. Also, more runs completed using partial expansion as opposed to using full expansion. Thus, partial expansion is an effective pruning technique for the workload used.

6.4. Node equivalence

Figure 5 indicates that the number of states created is significantly reduced for a large set of task graphs when the node equivalence pruning technique is used. Thus, node equivalence pruning technique is very effective for the workload used.

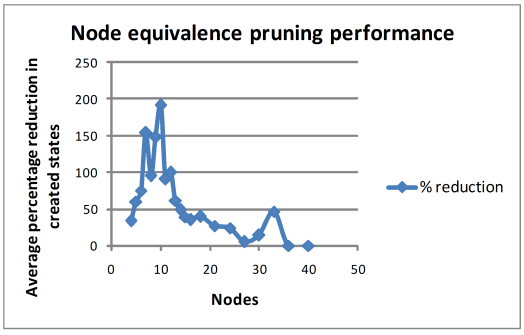

Figure 5. Reduction in the number of created states when using node equivalence pruning

7. Conclusions

This paper presented a new task scheduling algorithm based on A* search algorithm. Following from the properties of A* and the consistency and admissibility of the proposed cost function, $f(s)$, the produced schedules have optimal length with a reasonable runtime for small to medium sized task graphs. In comparison to a previous approach, the proposed algorithm offers a much improved cost function $f(s)$ and several pruning techniques that dramatically reduce the search space of the algorithm. Experimental results demonstrated these improvements.

For future work, the cost function $f(s)$ proposed can be improved further as this is one of the most effective ways of improving the efficiency and performance of the A* scheduling algorithm. The partial expansion technique could be improved further such that a free node is scheduled to one available processor at a time thereby creating only one new state in each iteration of the A* scheduling algorithm. It will also be beneficial to parallelise the A* scheduling algorithm as it has the potential to improve the runtime.

References

[1] Y.-K. Kwok and I. Ahmad. On multiprocessor task scheduling using efficient state space search approaches. *Journal of Parallel and Distributed Computing*, 65(12):1515 – 1532, 2005.

[2] G. Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving (5th Edition)*. Pearson Addison Wesley, 2004.

[3] N. J. Nilsson. *Artificial intelligence: a new synthesis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[4] V. J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18(1):55 – 71, 1987.

[5] A. Z. S. Shahul. Optimal Scheduling of Task Graphs on Parallel Systems. *Master's thesis*, University of Auckland, Auckland, New Zealand, 2008.

[6] O. Sinnen. *Task Scheduling for Parallel Systems*. Wiley, 2007.

[7] O. Sinnen, A. V. Kozlov, and A. Z. S. Shahul. Optimal Scheduling of Task Graphs on Parallel Systems. In *PDCN'07: Proceedings of the 25th conference on Proceedings of the 25th IASTED International Multi-Conference*, pages 170–175, Innsbruck, Austria, 2007. ACTA Press.

[8] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.

[9] A. S. Wu, H. Yu, S. Jin, K.-C. Lin, and G. Schiavone. An incremental genetic algorithm approach to multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):824 – 834, 2004.