Noname manuscript No.
(will be inserted by the editor)

Scheduling Task Graphs Optimally with A*

Ahmed Zaki Semar Shahul, Oliver Sinnen

the date of receipt and acceptance should be inserted later

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 the A* search algorithm which is a best-first state space search. The adaptation of the A* search algorithm for the task scheduling problem is referred to as the A* scheduling algorithm. The A* scheduling algorithm can produce optimal schedules in reasonable time for small to medium sized task graphs with several tens of nodes. In comparison to a previous approach, the here presented A* scheduling algorithm has a significantly reduced search space due to a much improved consistent and admissible cost function $f(s)$ and additional pruning techniques. Experimental results show that the cost function and the various pruning techniques are very effective for the workload. Last but not least, the results show that the proposed A* scheduling algorithm significantly outperforms the previous approach.

Department of Electrical and Computer Engineering
University of Auckland
Private Bag 92019
Auckland, New Zealand
Email: ahmedzakis@yahoo.com, o.sinnen@auckland.ac.nz

1 Introduction

Parallelisation of a software application involves three steps, namely task decomposition, dependence analysis and task scheduling [6]. The assignment of tasks to the processing units (spatial assignment) and defining their execution order (temporal assignment) are referred to as task scheduling [23]. 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. [19], i.e. finding an optimal solution takes exponential time, unless NP = P. Hence, many heuristics such as list scheduling exist to tackle the problem of task scheduling [4,7,8,16,18,25,28,29]. For these heuristic algorithms, the program is modeled as a Directed Acyclic Graph (DAG), called a task graph, where the nodes represent the tasks of the program and the edges the communications (dependencies) between the tasks. While the heuristics usually provide "good" results, there is no guarantee that the solutions are close to optimal, especially for task graphs with high communication costs [26]. There are several scenarios where even a "good" solution will not suffice and an optimal solution is needed. For instance, in time critical systems where performance is essential, for loop kernels and for evaluation of the quality of schedules produced by heuristics.

Given the NP-hardness, finding an optimal solution requires an exhaustive search of the entire solution space. For scheduling, this solution space is spanned by all possible processor assignments combined with all possible task orderings. Clearly this search space grows exponentially with the number of tasks, thus finding an optimal solution becomes impractical even for very small task graphs. The only hope for finding an efficient algorithm

for small to medium sized problems with tens of nodes is by guiding and pruning the search space with problem specific knowledge, as recent success in other NP-hard optimisation problems has shown, e.g. Knapsack [17] or Travelling Salesman Problem [1].

Depending on the formulation of the problem and its constraints, different algorithmic techniques to explore the solution space can be employed. Integer Linear Programming or Constraint Programming are such techniques, which have already been applied to the scheduling problem [2,3]. One major drawback of these and similar techniques for the task scheduling problem is that the size of the search space does not only depend on the number of tasks and processors, but also on the values of the node and edge weights [22]. More promising seem to be techniques such as branch-and-bound and specifically A* [5,21,6], where problem specific knowledge can be deeply integrated.

A scheduling algorithm based on the A* search algorithm has been proposed in [10] for the problem of task scheduling that produces optimal solutions. A* is a best-first state space search algorithm [5,21]. Every state s in the search space represents a partial solution and has an underestimated cost value f associated with it. This value represents the minimum schedule length that any complete solution based on the partial solution represented by s will incur. In every step, the A* algorithm expands the state with the lowest f value. With this approach, A* can drastically reduce the number of states it needs to search in order to find the optimal solution. The determining factor in the effectiveness of A* algorithm is the quality of the underestimated f values.

This paper presents a scheduling algorithm based on A*, referred to as the A* scheduling algorithm, that produces optimal schedules for small to medium sized task graphs with some dozens of nodes. In comparison to the only comparable algorithm proposed in [10], the heuristic for the calculation of f values is drastically improved. Moreover, new pruning techniques have been introduced that significantly reduce the number of states in the state space. Experimental results have been presented that demonstrate the effectiveness of the proposed algorithm. The experimental results also give new insights into the relation between the properties of a task graph and its scheduling behaviour.

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. Pruning techniques are discussed in Section 4. The experimental evaluation of the new scheduling algorithm is presented in Section 5. Conclusions and future work are presented in Section 6.

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 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 illustrates a task graph with four nodes.

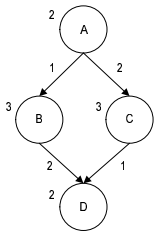


Fig. 1 A directed acyclic graph (task graph) with four nodes (tasks)

To schedule 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. Alternatively, a *complete schedule* for a task graph is the association of the function pair $(t_s, proc)$ to each node $n \in \mathbf{V}$. The following classical assumptions are made about the target parallel system [23]: 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.

Based on these properties of the target system, the finish time, t_f, of any node $n \in \mathbf{V}$ can be defined as its start time plus its computation cost, formally, $t_f(n) = t_s(n) + w(n)$.

Furthermore, two sets of constraints apply to the scheduling of a task graph G onto a target system $\mathbf{P}$. They are referred to as processor and precedence constraints. Processor constraints ensure that any processor in the parallel system can execute only one task at a time. Precedence constraints impose the execution

order of the nodes in light of their communications (dependencies), i.e. a node n_j can only start execution after all its predecessors, $\mathbf{pred}(n_j) = \{n_i \in \mathbf{V} : e_{ij} \in \mathbf{E}\}$, have completed execution plus communication time if the predecessors are executed on different processors, i.e. For $n_i, n_j \in \mathbf{V}$, $e_{ij} \in \mathbf{E}$ and $i \neq j$:

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

Figure 2 shows a feasible (left) and an optimal schedule (right) for the task graph represented in Figure 1 on two processors.

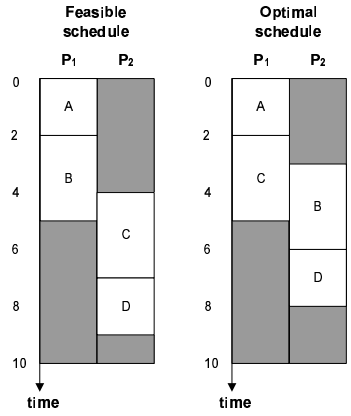


Fig. 2 A feasible (left) and an optimal (right) schedule for the task graph shown in Fig. 1

A *feasible schedule* is a schedule that adheres to processor and precedence constraints. 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. This can be expressed as $sl(\mathcal{S}) = \max_{n \in \mathbf{V}}\{t_f(n)\}$. In Figure 2, the length of the feasible schedule is nine time units and the length of the optimal schedule is eight time units. The finish time of a processor $P \in \mathbf{P}$ is defined as $t_f(P) = \max_{n \in \mathbf{V}: proc(n)=P}\{t_f(n)\}$. In Fig. 2, the finish times of P_1 and P_2 of the feasible schedule (left) are five and nine respectively. Since all schedules considered in this paper are feasible schedules, the term "feasible" is implied and therefore will not be explicitly stated from now on. An *optimal schedule* is a complete schedule that has the shortest possible schedule length. There can be several complete and optimal schedules for a given task graph G on target system $\mathbf{P}$. The set of optimal schedules, $\mathbf{optimal}(G)$, is a subset of the set of complete schedules, $\mathbf{complete}(G)$, i.e. $\mathbf{optimal}(G) \subset \mathbf{complete}(G)$. Finding the optimal schedule for a given task graph and for a given number of processors is the problem addressed in this paper. This problem is known to be NP-hard [19], i.e. finding an optimal schedule takes exponential time unless NP = P.

The task graph (DAG) depicted in Fig. 1 can be used to define some basic graph terminologies/properties [23]. The length of a path p in G is the sum of the weights of its nodes and edges:

$$len(p) = \sum_{n \in p, \mathbf{V}} w(n) + \sum_{e \in p, \mathbf{E}} c(e). \quad (2)$$

The *computation length* of a path p in G is the sum of the weights of its nodes, i.e. $len_w(p) = \sum_{n \in p, \mathbf{V}} w(n)$. A *critical path* is a longest path in the task graph. The *computation critical path* is a longest computation path in a task graph G, i.e. $len_w(cp_w) = \max_{p \in G}\{len_w(p)\}$. The computation critical path, cp_w, provides a lower bound on the schedule length such that any schedule $\mathcal{S}$ of G on target system $\mathbf{P}$ adheres to:

$$sl(\mathcal{S}) \geq len_w(cp_w). \quad (3)$$

The *bottom level* of n, $bl(n)$, is the length of the longest path starting with n:

$$bl(n) = \max_{n_i \in \mathbf{desc}(n) \cap \mathbf{sink}(G)} \{len(p(n \rightarrow n_i))\} \quad (4)$$

where $\mathbf{sink}(G) = \{v \in \mathbf{V} : \mathbf{succ}(v) = \emptyset\}$ ($\mathbf{succ}(n_i) = \{n_j \in \mathbf{V} : e_{ij} \in \mathbf{E}\}$), i.e. set of nodes that have no successors, and $\mathbf{desc}(n) = \{v \in \mathbf{V} : \exists p(n \rightarrow v) \in G\}$, i.e. a node v is a descendant of n if there is a path p from n to v. If $\mathbf{desc}(n) = \emptyset$, then $bl(n) = w(n)$. The *top level* of n, $tl(n)$, is the length of the longest path ending in n, excluding $w(n)$:

$$tl(n) = \max_{n_i \in \mathbf{ance}(n) \cap \mathbf{source}(G)} \{len(p(n_i \rightarrow n))\} - w(n) \quad (5)$$

where $\mathbf{source}(G) = \{v \in \mathbf{V} : \mathbf{pred}(v) = \emptyset\}$, i.e. set of nodes that have no predecessors, and $\mathbf{ance}(n) = \{v \in \mathbf{V} : \exists p(v \rightarrow n) \in G\}$, i.e. a node v is an ancestor of n if there is a path p from v to n. If $\mathbf{ance}(n) = \emptyset$, then $tl(n) = 0$.

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, node B in Fig. 1 has a computation bottom level of five, $bl_w(B) = 5$. The interesting property of the computation bottom level is that it gives a lower bound on the schedule length, i.e. a schedule cannot finish earlier than the start time of any of

its nodes plus the corresponding computation bottom level:

$$sl(\mathcal{S}) \geq t_s(n) + bl_w(n) \; \forall n \in \mathbf{V}. \quad (6)$$

This is due to the precedence constraints in (1) imposed by the edges of the task graph G.

The *sequential length* of a task graph G is the execution time of G on one processor only, i.e. $seq(G) = \sum_{n \in \mathbf{V}} w(n)$.

A common heuristic technique used to tackle the task scheduling problem is list scheduling. List scheduling in its simplest form consists of two parts. Firstly, the technique involves sorting the nodes of the task graph based on a priority scheme while adhering to its precedence constraints. Following that it involves successively scheduling the nodes to a chosen processor. An algorithm applying the list scheduling technique has the freedom to define the two criteria: the priority scheme for the nodes and the choice criterion for the processor. The priority scheme for the nodes is usually based on bottom/top level. Sometimes dynamic priority schemes are also used. And the processor chosen is usually the one that allows the earliest start time for the node. List scheduling generally produces good schedules but there is no guarantee that the solutions are close to optimal, especially for task graphs with high communication costs[23].

3 Task Scheduling with A*

The task scheduling problem is tackled with the A* algorithm which is a best-first state space search algorithm [21]. Hence, the task scheduling problem is first formulated as a state space search problem. Following that the A* algorithm is employed to search through the state space and the cost function used by the A* algorithm is discussed.

3.1 State space search formulation

A state space represents the abstract space of solutions which must be searched to find an optimal solution for a problem. Hence, it is sometimes referred to as a solution space. In the case of task scheduling, each state represents a schedule. The following components contribute to formulating the state space:

- **Initial state**: The state in which no nodes have been scheduled, i.e. an empty schedule. The A* algorithm commences its search from this state.
- **Expansion operator**: An operator that determines a scheme for expanding existing states to create new states in the state space. In this case, 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. A node scheduled on a processor gives rise to a new state, i.e. each node-processor pairing gives rise to one new state.
- **Goal state**: The goal state is a complete schedule. The algorithm terminates its search through the state space when a goal state is reached.

Figure 3 shows a partial state space for the task graph depicted in Fig. 1. The state space shows all the possible ways of assigning the first three nodes to the two target processors. Figure 3 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.

3.2 A* search algorithm

The A* algorithm maintains two lists, OPEN and CLOSED, to search the state space. The OPEN list holds all the states that are awaiting expansion. The CLOSED list holds the states that have been fully expanded. The following is the A* algorithm in its general form:

1. Put the initial state s_{init} in the OPEN list. The states in the OPEN list are sorted according to cost function $f(s)$.
2. Remove from OPEN the state s with the best f value.
3. Test s for the goal condition. If true, then a solution has been found and the algorithm terminates.
4. i) Expand s to form new states.
   ii) For each new state check whether it is present in either the CLOSED or OPEN list. If yes discard the state, otherwise, insert it into the OPEN list.
   iii) Place s in the CLOSED list.
5. Go to Step 2.

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 $f^*(s)$. $f(s)$ function is an underestimate of the function $f^*(s)$. In other words, if the function $f(s)$ satisfies the condition $f(s) \leq f^*(s)$ for any state s, it is called an *admissible* function. The $f(s)$ function is made up of two components: the cost

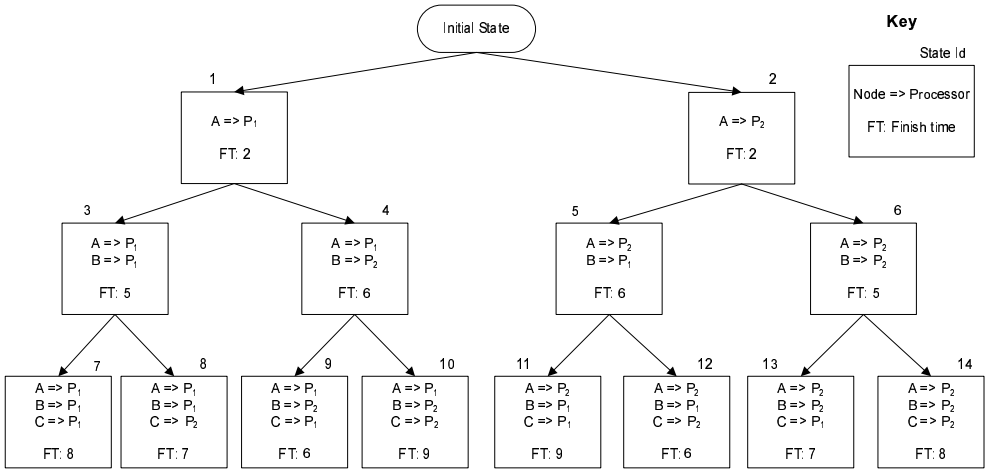


Fig. 3 A partial state space for the task graph depicted in Fig. 1

from the initial state to state s, $g(s)$, and the estimated cost function from state s to the goal state, $h(s)$. Formally, $f(s) = g(s) + h(s)$. Since $g(s)$ can be determined precisely for any state s, $h(s)$ function is the heuristic component and directly influences $f(s)$. The exact cost from a state s to the goal state is represented by $h^*(s)$ and $h(s)$ is an underestimate of $h^*(s)$, i.e. $h(s) \leq h^*(s)$. This condition implicitly ensures that the $f(s)$ function remains admissible.

The informativeness of the $h(s)$ function determines the number of states that the A* algorithm examines. A well informed $h(s)$ function is a function that is close to $h^*(s)$, i.e. if $h_1(s) < h_2(s)$ then $h_2(s)$ is strictly more informed provided it is admissible and will result in fewer states being examined. The more informed the $h(s)$ function, i.e. the closer the $h(s)$ to $h^*(s)$, the less the number of states that have to be examined [15]. In the ideal case, $h(s) = h^*(s)$. Conversely, if the $h(s)$ function were to be underestimated to zero in all cases, then the A* search algorithm effectively degenerates to uniform cost search [21]. Another case is when $h_1(s) \leq h_2(s)$, which means that $h_2(s)$ is still a more informed function than $h_1(s)$, but not strictly. In general, an A* algorithm using $h_2(s)$ will examine fewer states than an algorithm using $h_1(s)$, however this is not guaranteed and under certain conditions using $h_2(s)$ can occasionally lead to more states being examined (Section 5). Nevertheless, it is the aim to have an as well informed function as possible.

Another desirable property of the heuristic function, $h(s)$, is *consistency* (also called *monotonicity*). A consistent heuristic function is a function that holds the following property for all states s_i and s_j such that s_j is a descendant of s_i [12]:

$$h(s_i) - h(s_j) \leq g(s_j) - g(s_i) \quad (7)$$

In other words, the f value along any path in the state space should be monotonically non-decreasing, i.e. the f value can either remain the same or increase but never decrease along a path in the state space. The A* algorithm is guaranteed to find an optimal solution with an admissible and consistent $f(s)$ function.

3.3 Adapting A* for task scheduling

In [10], it was proposed to use the A* algorithm for the problem of task scheduling. The initial state s_{init} is the empty schedule where no node has been scheduled yet. New states are expanded from a state s by taking the partial schedule represented by s and scheduling all *free* nodes to every available processor. The number of new states generated from a state s depends on the number of nodes that are free to be scheduled in state s and the number of target processors available. A node n_j is free to be scheduled if all of its predecessors are already part of the partial schedule $\mathcal{S}$ of s, i.e.

$$\mathbf{free}(s) = \{n_j \in \mathbf{V} : n_j \notin \mathcal{S}(s) \wedge \mathbf{pred}(n_j) \subseteq \mathcal{S}(s)\} \quad (8)$$

Each free node can be scheduled on each target processor following the nodes that have already been scheduled. Thus, the maximum number of new states that can be created from s is the product of the number of free nodes and the number of target processors, i.e.

$$new(s) = |\mathbf{free}(s)| \times |\mathbf{P}| \quad (9)$$

Newly created states are added to the OPEN list. The goal state is a complete schedule. A complete schedule will inevitably be an optimal schedule due to the fact that the A* algorithm will return an optimal solution when a consistent and admissible $f(s)$ function is used.

The $f(s)$ function proposed in [10], here designated as $f_{KA}(s)$, is calculated as follows: Let n_{max} be the node in the partial schedule S associated with s that has the latest finish time $n_{max} = n \in \mathcal{S} : t_f(n) = \max_{n_i \in \mathcal{S}(s)} \{t_f(n_i)\}$:

$$f_{KA}(s) = t_f(n_{max}) + \max_{n \in \mathbf{succ}(n_{max})} \{bl_w(n)\} \quad (10)$$

So, $f_{KA}(s)$ is defined as the sum of the schedule length of the partial schedule $\mathcal{S}$ of s, since $sl(\mathcal{S}) = t_f(n_{max})$, and the maximum computation bottom level of all successors of n_{max}. In terms of $g(s)$ and $h(s)$ we have: $g(s) = t_f(n_{max})$ and $h_{KA}(s) = \max_{n \in \mathbf{succ}(n_{max})} \{bl_w(n)\}$. Per definition of the computation bottom level, this is identical to the start time of n_{max} plus its computation bottom level, i.e. $f_{KA}(s) = t_s(n_{max}) + bl_w(n_{max})$. With the lower bound on the schedule length established in (6), this function is clearly admissible, since the value of $f^*_{KA}(s)$ is the length of a complete schedule. However, the $f_{KA}(s)$ function is not consistent as desired, because the last node to finish, n_{max}, can change from state to state and $h_{KA}(s)$ changes with the node in a non-monotonic way. But, in the case of the task scheduling problem only an admissible cost function suffices as states representing the same schedule have the same f value because the path taken to achieve a particular schedule is irrelevant for the f value.

Thus, the A* algorithm has been adapted to the task scheduling problem in [10]. The A* algorithm adapted for the task scheduling problem is referred to as the A* scheduling algorithm from here on. In this paper, several optimisation and pruning techniques have been implemented to the A* scheduling algorithm to improve the efficiency of the implementation. In addition, the $f(s)$ function has been improved significantly which is detailed in the following section.

3.4 Proposed $f(s)$ function

The $f(s)$ function proposed in this paper is based not only on the last node that finishes in the partial schedule but on all the nodes in the partial schedule. This makes the $f(s)$ more informed and at the same time consistent. To further improve the informedness of our $f(s)$ two additional components are proposed, thus our $f(s)$ function is derived from three components: the computation bottom level, idle time and data ready time.

Computation bottom level

The computation bottom level of all nodes in the partial schedule are considered in deriving this component of the $f(s)$ function as follows:

$$f_{bl_w}(s) = \max_{n \in \mathcal{S}} \{t_s(n) + bl_w(n)\} \quad (11)$$

Clearly, $f_{bl_w}(s)$ is admissible considering the lower bound on the schedule length established in (6). It is intuitive that $f_{bl_w}(s)$ is more accurate than $f_{KA}(s)$ as the latest finishing node in a partial schedule does not necessarily have the highest computation bottom level. In fact the following holds for any state s:

$$f_{KA}(s) \leq f_{bl_w}(s) \leq f^*(s) \quad (12)$$

i.e. $f_{bl_w}(s)$ is tighter than $f_{KA}(s)$ and therefore an implementation of the A* scheduling algorithm using the $f_{bl_w}(s)$ function needs to expand fewer states in general as opposed to the same implementation using the $f_{KA}(s)$ function. This is shown in experimental results in Section 5.3.

In order to have a fast A* scheduling algorithm, it is also important that the calculation of the f value of a state is fast. The definition of $f_{bl_w}(s)$ in (11) needs to consider all nodes of the partial schedule which is $O(|\mathbf{V}|)$ for each calculation of $f_{bl_w}(s)$. However, $f_{bl_w}(s)$ can be calculated incrementally. Let n_{last} be the last node that was added to the partial schedule of s and s_{parent} the parent state of s. Then, f_{bl_w} can be redefined as

$$f_{bl_w}(s) = \max \{f_{bl_w}(s_{parent}), t_s(n_{last}) + bl_w(n_{last})\}. \quad (13)$$

This is equivalent to (11) but only the last node n_{last} has to be considered for each new state. Hence, the calculation of $f_{bl_w}(s)$ is $O(1)$ which is the same as for $f_{KA}(s)$.

Idle time

While the computation bottom level is very useful to establish a quite accurate lower bound on the schedule length in structured task graphs with a medium to high number of edges, it is not very useful for graphs with few edges or graphs with several sink (exit) nodes. The reason is that the computation bottom level is a path metric which is less useful when there are relatively few edges and therefore relatively few and short paths in

the task graph. Therefore, an additional metric, Idle time, for the calculation of the $f(s)$ function is proposed which is not derived from a path metric.

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 depicted in Figure 2 (right) 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, i.e.

$$sl(\mathcal{S}) \geq \left(idle(\mathcal{S}) + \sum_{n \in \mathbf{V}} w(n) \right) / |\mathbf{P}|$$

$$f_{IT}(s) = \left(idle(\mathcal{S}) + \sum_{n \in \mathbf{V}} w(n) \right) / |\mathbf{P}|. \quad (14)$$

The bound is tight if and only if all processors finish at the same time, i.e. their last nodes finish at the same time. The $f_{IT}(s)$ function never decreases for a larger partial schedule as new nodes are scheduled after already scheduled nodes during node expansion and not inserted into idle periods. $f_{IT}(s)$ can be calculated incrementally and therefore the calculation complexity of $f_{IT}(s)$ is $O(1)$.

Data Ready Time

The *Data Ready Time* (DRT) of a node $n_j \in \mathbf{V}$ on processor P is

$$t_{dr}(n_j, P) = \max_{n_i \in \mathbf{pred}(n_j)} \{ t_f(n_i) + \begin{cases} 0 & \text{if } proc(n_i) = proc(n_j) \\ c(e_{ij}) & \text{otherwise} \end{cases} \quad (15)$$

If $\mathbf{pred}(n_j) = \emptyset$, i.e. n_j is a source node, $t_{dr}(n_j) = t_{dr}(n_j, P) = 0 \; \forall P \in \mathbf{P}$ [23]. Note that the DRT can be determined for any processor $P \in \mathbf{P}$ independently of the processor to which n_j is allocated. The minimum DRT of a node $n \in \mathbf{V}$ on all processors in $\mathbf{P}$ is:

$$t_{dr}(n) = \min_{P \in \mathbf{P}} \{ t_{dr}(n, P) \}. \quad (16)$$

This introduces a DRT lower bound on the start time of a node n:

$$t_s(n) \geq t_{dr}(n). \quad (17)$$

Alternatively, a node cannot start execution before all data is ready. Condition (17) can be further extended to define the earliest possible start time $t_{s_{min}}$ of a node $n \in \mathbf{V}$ on processor $P \in \mathbf{P}$: $t_{s_{min}}(n, P) = \max(t_{dr}(n, P), t_f(P))$ where $t_f(P)$ is the finish time of processor $P \in \mathbf{P}$. Then, the earliest start time of n on all target processors is $t_{s_{min}}(n) = \min_{P \in \mathbf{P}} \{ t_{s_{min}}(n, P) \}$. This provides yet another tighter lower bound on the start time of a node:

$$t_s(n) \geq t_{s_{min}}(n). \quad (18)$$

The DRT component of the $f(s)$ function, $f_{DRT}(s)$, can be expressed as:

$$f_{DRT}(s) = \max_{n \in \mathbf{free}(s)} \{ t_{dr}(n) + bl_w(n) \}. \quad (19)$$

Equations (11) and (19) can be combined to produce a general case as follows:

$$\max_{n \in \mathcal{S}(s) \cup \mathbf{free}(s)} \begin{cases} t_s(n) & if \; n \in \mathcal{S}(s) \\ t_{dr}(n) & otherwise \end{cases} + bl_w(n). \quad (20)$$

Figure 4 illustrates the concept of using the data ready time of a node to calculate the f value of a state. In this case, the minimum data ready time of free node D on all processors, $t_{s_{min}}(D)$ is five. Hence, the f value of a state representing the partial schedule shown in Figure 4 can be increased to seven using $f_{DRT}(s)$.

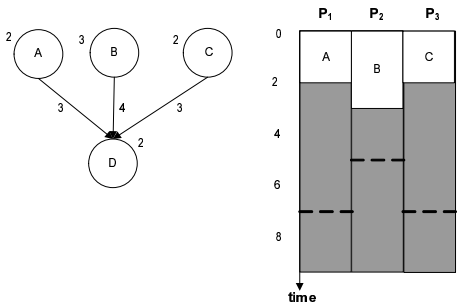


Fig. 4 Using the data ready time method to calculate the f value of a state. Dotted lines indicate the data ready time of free node D on each processor

The preceding equation is admissible considering Conditions (6) and (18). The complexity of calculating $f_{DRT}(s)$ is $O(|\mathbf{free}(s)| \cdot |\mathbf{P}|)$ per state as the DRT is calculated for all free nodes in s with respect to all target processors. However, when traversing a state space, i.e. going from the initial state to a goal state, the number of free nodes cumulatively is exactly the number of nodes in the task graph, i.e. $|\mathbf{V}|$. Since there are $|\mathbf{V}|$ states from an initial state to a goal state along

any path in the state space it averages to one node per state. To calculate the DRT of one node all the incoming edges to the node have to be checked which on average is the density of the task graph. The density of a task graph representing any practical program is normally a small constant and does not grow with the size of the task graph. Hence, the complexity amortises to the number of processors, i.e. $O(|\mathbf{P}|)$. For a fixed target system, this is constant. Therefore, the complexity of calculating $f_{DRT}(s)$ can be considered $O(1)$ just like $f_{KA}(s)$. The data ready time metric like the computation bottom level metric is useful for task graphs that are structured with medium to high number of edges.

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

$$f(s) = \max\{f_{bl_w}(s), f_{IT}(s), f_{DRT}(s)\} \tag{21}$$

The preceding $f(s)$ function takes into account the parent state's f value denoted by $f(s_{parent})$ as it incorporates the value of $f_{bl_w}(s_{parent})$. This maximising approach also ensures that the $f(s)$ function remains consistent. Note that the $f_{KA}(s)$ function is an inconsistent function. The f value of the initial state, $f(s_{init})$, representing an empty schedule can also be increased from zero (as suggested in [10]) to the following:

$$f(s_{init}) = \max\left(\left(\sum_{n\in\mathbf{V}} w(n)\right)/|\mathbf{P}|, len_w(cp_w)\right) \tag{22}$$

The preceding function is admissible considering Conditions in (3) and (14). The following holds for any state s:

$$f_{KA}(s) \leq f(s) \leq f^*(s) \tag{23}$$

This shows that the $f(s)$ function is more informed (not strictly) than the $f_{KA}(s)$ function.

4 Pruning Techniques

Pruning is a process that eliminates unpromising subtrees in the search space. The rationale is that if it is possible to determine that a subtree will not lead to an optimal solution then it is not worth maintaining it in in the search space. This implicitly means that 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.

4.1 Processor normalisation

Two processors are isomorphic if they are both empty, i.e. no nodes are already scheduled to them [10]. 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 two schedules $\mathcal{S}_1$ (left) and $\mathcal{S}_2$ (right) shown in Figure 5. Observe that the start times of the nodes are the same across the schedules, e.g. B starts at time unit 2 in both schedules. Schedule $\mathcal{S}_2$ becomes identical to $\mathcal{S}_1$ when the processors P_1 and P_2 are renamed to P_2 and P_1 respectively in $\mathcal{S}_2$. This is valid for the scheduling problem considered here as all processors are identical and the target system is fully connected (Section 2).

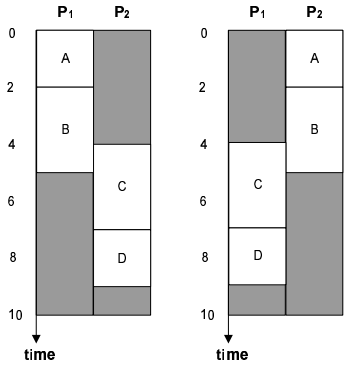


Fig. 5 Processor normalisation: the two schedules are identical after swapping the processor names for one of the schedules

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 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 detected using the OPEN and CLOSED lists and eliminated. Note that processor name normalisation can also be used in scheduling algorithms based on stochastic search techniques, e.g. Genetic Algorithms (GA) [27,29], in order to reduce the size of the search space.

4.2 Partial expansion

The number of new states created from a given state s is affected by the number of target processors and free

nodes as shown in (9). However, it is not necessary that all the free nodes in s are scheduled exhaustively with all the available target processors. Instead, each free node is scheduled 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 also has the lowest f value. Since the newly created state will have one more node scheduled than s it is within the reasoning of A* to immediately continue with that node. 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 reduces the explosion of states that can otherwise occur. This pruning technique is most effective when the $f(s)$ value is equal to C^*, where C^* is the cost of the optimal solution path to the goal, i.e. $f(s) = C^*$. Note that this pruning technique can occasionally increase the number of states examined. This is because a full expansion could have resulted in the algorithm potentially picking a different child state than that picked if there are multiple states with the same $f(s)$ value.

4.3 Node equivalence

The number of free nodes in a state s has an effect on the number of new states created when expanding s as shown in (9). However, this factor can be reduced when equivalent nodes are detected. Two nodes n_i and n_j are said to be equivalent when the following conditions are true:

- $w(n_i) = w(n_j)$ - they have the same computation costs.
- $\mathbf{pred}(n_i) = \mathbf{pred}(n_j)$ - they have the same set of predecessors.
- $\mathbf{succ}(n_i) = \mathbf{succ}(n_j)$ - they have the same set of successors.
- $c(e_{pi}) = c(e_{pj})\ \forall n_p \in \mathbf{pred}(n_i)$ - the corresponding edge weights to the predecessors are equal.
- $c(e_{is}) = c(e_{js})\ \forall n_s \in \mathbf{succ}(n_i)$ - 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. This pruning technique is very effective for specific task graph structures as shown in the experimental results in Section 5.5. Note that the preceding five conditions to determine node equivalence differs from the set of conditions defined in [10].

5 Experimental Results

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

5.1 Workload

To comprehensively evaluate the performance of the task scheduling algorithm and the pruning techniques presented, a large set of task graphs differing in properties such as the task graph structure, number of nodes (4 - 40), density, Communication to Computation Ratio (CCR is the sum of all edge weights divided by sum of all node weights), weight type and maximum branching factor were generated [22].

Ten different task graph structures, namely independent, out tree, in tree, fork, join, fork-join, pipeline (Diamond) [9], stencil [11], series-parallel (SP-graphs) and random were generated [22]. These task graph structures are explained in [23].

It is to be noted that certain properties do not apply to certain task graph structures. For example, it is not possible to vary the density of a fork, join or a fork-join graph as the density of these structures are determined by the number of nodes present in the task graph.

In total, the workload comprises 2834 task graphs. All the task graphs in the workload are scheduled on a two, four and eight processor target machine. Hence, 8502 runs of each implementation of the A* scheduling algorithm were carried out. 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”. The runs that did not timeout are referred to as complete runs. Note that the data analysed in this section only includes complete

runs, i.e. runs that completed in the implementations that are compared.

5.2 Measure for performance evaluation

Classically, evaluations of scheduling algorithms compare the lengths of the produced schedules [13,14,20]. Yet, all schedules are guaranteed to be optimal so we are interested in the scheduling time. As this time is hardware and implementation dependent, the measure used to evaluate the implemented A* scheduling algorithm and the pruning techniques is the total number of created states, 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 [22].

For the purposes of evaluation, the percentage difference of the number of created states is used as the measure to determine the effectiveness of the proposed cost function, $f(s)$, and the pruning techniques. The percentage difference in the number of created states is calculated using

$$\left(\frac{N_B - N_A}{N_A}\right) \times 100 \tag{24}$$

where N_A and N_B are the numbers of created states when using algorithm/technique A and B, respectively. In the following, the baseline algorithm/technique A is usually the full featured proposed algorithm with all pruning techniques.

5.3 Cost function evaluation

The cost function $f(s)$ plays an important role in the performance of the A* scheduling algorithm as discussed in Section 3.2. The $f(s)$ function proposed in (21) is compared with the $f_{KA}(s)$ function presented in (10) that was initially proposed in [10]. All pruning techniques presented in Section 4 are employed in both cases. Figure 6 shows the average percentage difference of the number of created states when using $f_{KA}(s)$ instead of the proposed $f(s)$ function ($f(s)$ is baseline). As can be observed, $f_{KA}(s)$ creates dramatically more states, in other words the new $f(s)$ function reduces the number of created states dramatically for the task

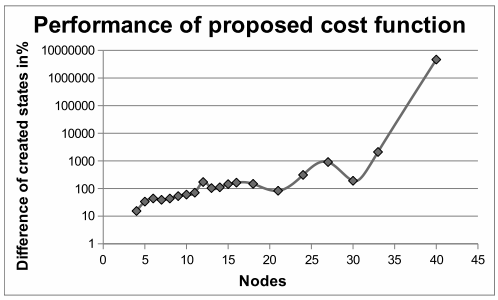


Fig. 6 Difference of number of created states between using $f_{KA}(s)$ and the proposed $f(s)$ (baseline)

graphs in the workload. The new $f(s)$ function is apparently a much closer underestimate of $f^*(s)$, i.e. more informed.

In addition, the effect of using the Data Ready Time (DRT) component in the $f(s)$ function is analysed, i.e. the number of states created when using the $f(s)$ function in (21) is compared with the number of states created when using the same function *without* the DRT component, $f_{DRT}(s)$ (with DRT component is baseline). The average percentage difference of the number of created states due to the DRT component is shown in Fig. 7.

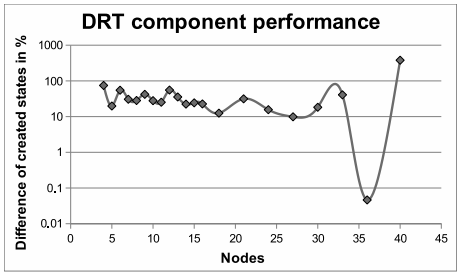


Fig. 7 Difference of number of created states without/with DRT component in the proposed $f(s)$ function (baseline is with DRT component)

5.4 Task graph scheduling difficulty

The results presented in [10] suggest that the number of tasks present in the task graph is the determining factor that affects the difficulty of finding an optimal schedule for a task graph. Figure 8 shows the average number of created states for ten node task graphs across various task graph structures in the workload. Intuitively, one might expect that the average number of states created will be more or less consistent across various task graph structures as only a uniform task graph size of ten nodes are considered. However, the graph shows

that the average number of created states vary dramatically across various task graph structures. Note that the chart has an exponential scale. It shows that the task graph structure plays an important role in determining the difficulty of scheduling a task graph.

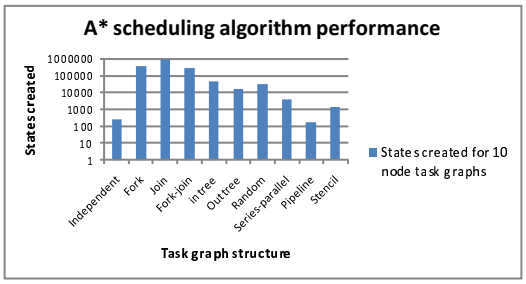


Fig. 8 A* scheduling algorithm performance based on task graph structure for ten node task graphs

5.5 Pruning technique evaluation

The pruning techniques discussed in Section 4 are evaluated to analyse the performance gain due to each of them.

Partial expansion

The partial expansion technique is discussed in Section 4.2. The reduction in the average number of created states when using the partial expansion technique as opposed to full expansion is shown in Table 1. The "Ratio" column indicates the number of states created using full expansion divided by the number of states created using partial expansion. The "Complete runs" column indicates the number of complete runs from which the data was obtained. Note that the number of complete runs are significantly lower for larger task graphs due to the increased difficulty in scheduling them. In addition, it is to be noted that there are not an equal number of task graphs for each task graph size. For example, a five node pipeline task graph was not generated as a pipeline structure cannot be maintained with five nodes. The "Ratio" column indicates that partial expansion decreases the number of states created for a vast majority of cases. Thus, partial expansion is an effective pruning technique for the workload used. Note that only a few runs completed (due to the timeout used) in the last three rows of Table 1 which is possibly the cause for the corresponding high ratio values.

Node equivalence

Node equivalence is discussed in detail in Section 4.3. The average percentage difference of the number of created states when using the node equivalence pruning technique as opposed to not using it is shown in Figure 9 (a) (baseline is with node equivalence pruning). The graph indicates that the number of states created is reduced significantly for a large set of task graphs when the node equivalence pruning technique is used.

Figure 9 (b) shows that the node equivalence pruning technique is most effective for independent task graphs. This can be quite intuitively explained as the only factor determining node equivalence is the computation costs of the nodes for independent task graphs. Thus, the likelihood of identifying equivalent nodes in independent task graphs increases which in turn significantly reduces the number of created states.

The significant difference in the number of created states as shown in Fig. 9 (a) and the wide range of task graphs for which this pruning technique is effective as shown in Fig. 9 (b) indicate that the node equivalence pruning technique is very effective for the workload used.

Processor normalisation

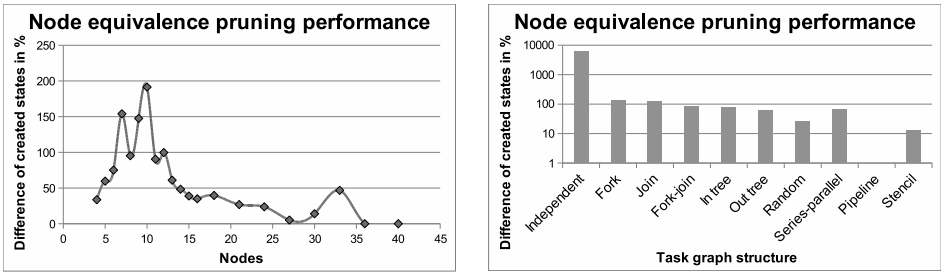
The processor normalisation pruning technique is discussed in Section 4.1. An implementation of the A* scheduling algorithm that uses the processor normalisation pruning technique is compared to the same implementation without the pruning technique. The processor isomorphism pruning technique was used in both the implementations compared. Figure 10 shows the average percentage difference of the number of created states without/with the processor normalisation pruning technique (baseline with processor normalisation). The graph shows three curves, one for the workload scheduled on two, four and eight processors respectively. As expected, the difference of the number of states created is most dramatic for eight processors and least significant for two processors. This is because more states are created with a large set of target processors as defined in (9). Therefore, when the processor normalisation pruning technique reduces the factor $|\mathbf{P}|$ in (9) the reduction in the number of states is dramatic.

5.6 CLOSED list evaluation

The CLOSED list contains all states that have been fully expanded and is used for duplicate detection. In

Nodes	Complete runs	Avg. states - Partial	Avg. states - Full	Ratio
4	414	13	14	1.03909
5	366	34	36	1.05226
6	414	112	116	1.03386
7	366	526	532	1.01056
8	414	3773	3791	1.00495
9	414	27997	28123	1.00449
10	411	98231	98362	1.00134
11	353	274802	274972	1.00062
12	377	229518	231499	1.00863
13	299	653911	653276	0.99903
14	312	782655	827532	1.05734
15	263	828528	764515	0.92274
16	219	714680	775942	1.08572
18	183	706968	773825	1.09457
21	116	689447	751499	1.09000
24	88	386768	630398	1.62991
27	53	435208	761059	1.74872
30	38	362559	713847	1.96892
33	27	110435	946660	8.57211
36	32	30637	389056	12.69892
40	26	114606	839364	7.32394

Table 1 Comparing partial expansion technique to full expansion



(a) (b)

Fig. 9 Difference of number of created states without/with node equivalence pruning (baseline is with node equivalence pruning)

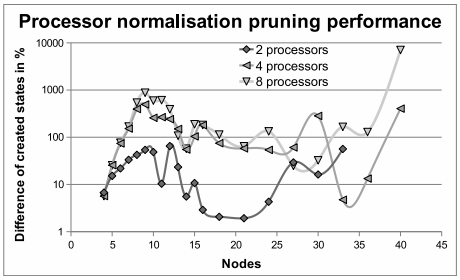


Fig. 10 Difference of number of created states without/with processor normalisation pruning technique (baseline is with processor normalisation)

[24], an implementation of the A* scheduling algorithm without the CLOSED list was implemented. Here, an implementation of the A* scheduling algorithm with the CLOSED list is compared to the same implementation without the CLOSED list, i.e. only OPEN list. The intention is to ascertain the effectiveness of the CLOSED list. Figure 11 shows the average percentage difference of the number of created states without/with the CLOSED list. As can be observed, not using the CLOSED list significantly increases the number of created states.

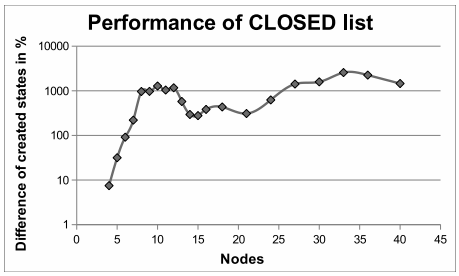


Fig. 11 Difference of number of created states without/with CLOSED list (baseline is with CLOSED list)

It is to be noted that the reduction in the number of created states comes at a cost of storing extra states in memory. Therefore, the maximum number of states

that are stored in memory is compared between the two implementations to ascertain that the extra memory used is not so great that it offsets the gain from reducing the number of created states. Figure 12 shows that the implementation using the CLOSED list stores on average less than two times the number of states stored when using the implementation with only the OPEN list. Thus, with only twice the memory consumption the implementation with the CLOSED list is able to reduce the number of created states dramatically as shown in Fig. 11. Thus, using the CLOSED list is vital for an efficient implementation of the A* scheduling algorithm.

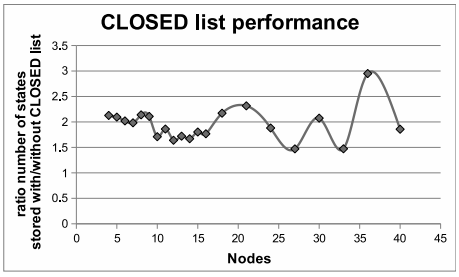


Fig. 12 Additional states in memory when using the CLOSED list as opposed to no using it

6 Conclusions

This paper presented a new task scheduling algorithm based on the best-first search algorithm A*. 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 (less than two minutes in most cases) for small to medium sized task graphs with up to 40 nodes. In comparison to a previous approach, the proposed algorithm offers significant improvements in two crucial areas. First, the much improved cost function $f(s)$ dramatically reduces the search space of the algorithm. Second, several pruning techniques further reduce the search space significantly. Experimental results demonstrated these improvements. They also showed for the first time the relation between the runtime of the A* scheduling algorithm and the structure of scheduled task graphs.

In the future, we believe that further significant improvements can be made to the proposed A* scheduling algorithm. First, we still see significant potential for improving the cost function $f(s)$. Second, more pruning techniques should exploit the freedom in local node ordering. And last but not least, it will also be beneficial to parallelise the A* scheduling algorithm for a reduced runtime.

References

1. D. L. Applegate, R.E. Bixby, V. Chvátal, and W.J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.
2. P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer Academic, 2001.
3. H. Casanova, A. Legrand, and Y. Robert. *Parallel Algorithms*. CRC Press, 2009.
4. E. G. Coffman and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213, 1972.
5. R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, July 1985.
6. A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Pearson, Addison Wesley, UK, second edition, 2003.
7. T. Hagras and J. Janeček. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. *Parallel Computing*, 31(7):653–670, 2005.
8. J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing*, 18(2):244–257, April 1989.
9. S. Y. Kung. *VLSI array processors*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.
10. Y.-K. Kwok and I. Ahmad. On multiprocessor task scheduling using efficient state space approaches. *Journal of Parallel and Distributed Computing*, 65:1515–1532, 2005.
11. Welf Löwe, Wolf Zimmermann, Sven Dickert, and Jörn Eisenbiegler. Source code and task graphs in program optimization. In *HPCN Europe 2001: Proceedings of the 9th International Conference on High-Performance Computing and Networking*, pages 273–282, London, UK, 2001. Springer-Verlag.
12. George Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving (5th Edition)*. Pearson Addison Wesley, 2004.
13. B. S. Macey and A. Y. Zomaya. A performance evaluation of CP list scheduling heuristics for communication intensive task graphs. In *Parallel Processing Symposium, 1998. Proc. of IPPS/SPDP 1998*, pages 538 –541, 1998.
14. C. L. McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle. A comparison of heuristics for scheduling DAGs on multiprocessors. Technical Report ncstrl.auburn_eng/CSE93-07, Auburn University, June 1993.
15. Nils J. Nilsson. *Artificial intelligence: a new synthesis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.
16. A. Palmer and O. Sinnen. Scheduling algorithm based on force directed clustering. In *Proc. of 9th Int. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT'08)*, Dunedin, New Zealand, December 2008. IEEE Press.
17. D. Pisinger. Where are the hard knapsack problems? Technical Report 2003/08, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, 2003.

18. A. Radulescu and A. J. C. van Gemund. Low-cost task scheduling for distributed-memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):648–658, 2002.
19. V. J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.
20. P. Rebreyend, F. E. Sandnes, and G. M. Megson. Static multiprocessor task graph scheduling in the genetic paradigm: A comparison of genotype representations. Research Report 98-25, Ecole Normale Superieure de Lyon, Laboratoire de Informatique du Parallelisme, Lyon, France, 1998.
21. S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2003.
22. Ahmed Zaki Semar Shahul. Optimal Scheduling of Task Graphs on Parallel Systems. *Master's thesis*, University of Auckland, Auckland, New Zealand, 2008.
23. O. Sinnen. *Task Scheduling for Parallel Systems*. Wiley, May 2007.
24. O. Sinnen, A. V. Kozlov, and A. Z. Semar Shahul. Optimal scheduling of task graphs on parallel systems. In *Proc. Int. Conference on Parallel and Distributed Computing and Networks*, Innsbruck, Austria, February 2007.
25. O. Sinnen and L. Sousa. List scheduling: Extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. *Parallel Computing*, 30(1):81–101, January 2004.
26. O. Sinnen and L. Sousa. Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):503–515, June 2005.
27. Annie S. Wu, Han Yu, Shiyuan Jin, Kuo-Chi Lin, and Guy Schiavone. An incremental genetic algorithm approach to multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):824 – 834, 2004.
28. T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.
29. A. Y. Zomaya, C. Ward, and B. S. 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.