Reducing the solution space of optimal task scheduling

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

6th September 2013

Abstract

Many scheduling problems are tackled by heuristics due to their NP-hard nature. Task scheduling with communication delays ($P|prec, c_{ij}|C_{max}$) is no exception. Nevertheless, it can be of significant advantage to have optimal schedules, e.g. for time critical systems or as a baseline to evaluate heuristics. A promising approach to optimal task scheduling with communication delays for small problems is the use of exhaustive search techniques such as A*. A* is a best first search algorithm guided by a cost function. While good cost functions reduce the search space, early results have shown that problem specific pruning techniques are paramount. This paper proposes two novel pruning techniques that significantly reduce the search space for $P|prec, c_{ij}|C_{max}$. The pruning techniques Fixed Task Order and Equivalent Schedules are carefully investigated based on observations made with simple graph structures such as fork, join and fork-join, yet they are generally applicable. An extensive experimental evaluation of computing more than two thousand schedules demonstrates the efficiency of the novel pruning techniques in significantly reducing the solution space.

1 Introduction

Scheduling is an essential and crucial part of parallel computing. In the scheduling problem addressed in this paper, a set of tasks with precedence constraints and communication delays, represented by a task graph, is to be scheduled on a set of identical processors is such a way that the schedule length, or makespan, is minimised. This problem is a classical scheduling problem and written as $P|prec, c_{ij}|C_{max}$ in the $\alpha|\beta|\gamma$ notation [7, 28]. It is well known that this problem is NP-hard in the strong sense [19]. The only known guaranteed approximation algorithm, [11], has an approximation factor depending on communication costs of the longest path in the schedule [6]. Many heuristics have been proposed for this classical problem, even quite recently [2, 12, 13, 14, 16, 26].

Despite the NP-hardness of the problem, it can be of significant advantage to have an optimal schedule in certain situations. This can be the case for very time critical systems or for application areas where a schedule is reused many times, e.g. in an embedded system or where the schedule is enclosed by a loop. Moreover, having optimal solutions for scheduling instances allows to better judge the quality of heuristics and thereby to gain insights into their behaviour. The lack of good guaranteed approximation algorithms makes this very relevant.

The work presented in this paper addresses the optimal solution of the $P|prec, c_{ij}|C_{max}$ scheduling problem. A task graph represents the program to be scheduled, where the nodes represent the tasks and the edges the communications among them. Weights represent computation and communication costs, respectively. The scheduling problem is the assignment of a processor and a start time to each task, in such a way that the schedule length is minimised. It is trivial to show that this is equivalent to finding a processor allocation and an ordering of the tasks [23]: the tasks just need to start as early as possible adhering to all constraints. In other words, our scheduling problem is a combinatorial optimisation problem. Such problems have been widely addressed with two different optimisation techniques: *i)* Mixed Integer Linear Programming (MILP) and *ii)* smartly enumerating all feasible solutions, e.g. using a branch-and-bound algorithm.

For our scheduling problem, there have been surprisingly few attempts at solving it optimally. Scheduling problems often have a weak LP relaxation (i.e. the problem without the integer constraint, which is solvable in polynomial time), which leads to long runtimes of MILP solvers. Further, with the communication costs in the scheduling model and the fact that local communication is cost free, it is non-trivial to formulate an efficient MILP for our problem. Examples of MILP formulations are [1] and more recently [3, 4]. In [4], the communication cost leads to bilinear constraints which have to be linearised, significantly reducing the efficiency of such an approach. Sometimes the MILP approach is used to design an efficient heuristic for the scheduling problem [9].

In this paper we will look at the second technique of enumerating all feasible solutions. For a task graph with $|\mathbf{V}|$ tasks scheduled on $|\mathbf{P}|$ processors, the solution space is spawned by all possible orderings of the tasks ($|\mathbf{V}|!$) times all possible processor allocations $|\mathbf{P}|^{|\mathbf{V}|}$. It is clear that the exploration of the solution space must be done very intelligently if we want to go beyond half a dozen of tasks and two processors. A* is a best-first search technique that can be applied to such solution space exploration [5]. In contrast to typical branch-and-bound algorithms [18], it does not follow a depth-first search, but always expands the state in the search space that looks most promising. In other words, A* uses a priority queue for states to expand, while branch-and-bound usually uses a LIFO (Last In First Out) queue. This comes at the cost of much higher memory consumption, but A* has the nice property that it will traverse the least number of states for a given cost estimation heuristic, compared to other exploration techniques [20]. In [22], we proposed an A* based scheduling algorithm, which was inspired by an earlier attempt [15, 17].

From these earlier results, we gained two important insights. First, pruning techniques play a paramount role to reduce the search space, even with good A* cost estimate functions. Second, the performance of the A* based scheduling algorithm is very dependent on the structure and density of the task graph. Paradoxically, very simple graph structures like fork and join were more difficult to schedule. i.e. it took longer to find the optimal solution for the same number of tasks, than more complex and dense structures like pipeline or stencil. The problem is that the simpler structures have more degrees of freedom and hence have a larger solution space.

The contribution of this paper is the proposal of new pruning techniques to significantly reduce the solution space of the $P|prec, c_{ij}|C_{max}$ scheduling problem. The two major techniques are Fixed Task Order pruning and Equivalent Schedule pruning. Both techniques are developed from observations made about the scheduling of simple task graph structures, namely independent tasks, fork graphs, join graphs and fork-join graphs. The techniques are investigated based on our A* scheduling algorithm, but the concepts and methods can be applicable to other exhaustive search techniques. They possibly extent to metaheuristics such as Genetic Algorithms [32, 29], by reducing the search space on which the metaheuristic operators work. A further pruning technique proposed is the extended utilisation of schedules produced by heuristics, which can prune the solution space significantly under certain conditions.

This paper also contributes an extensive evaluation computing more than two thousand schedules. Scheduling task graphs optimally with the novel techniques demonstrates that we achieved our goal of significantly reducing the search space. The novel pruning techniques are especially successful for those graph structures that were most difficult to schedule before, which are fork, join, fork-join and trees.

Section 2 continues this paper with the background and definition of the scheduling problem, presenting the basic A* scheduling algorithm and discussing its cost estimate function f. Existing pruning techniques are reviewed in Section 3, leading to the proposal of the new pruning techniques in Section 3.4, Section 4 and Section 5. They are evaluated in Section 6 and the paper concludes with Section 7.

2 Background

The problem to be addressed in this work is the scheduling of a Directed Acyclic Graph (DAG), called task graph, $G = (\mathbf{V}, \mathbf{E}, w, c)$ on a set of processors $\mathbf{P}$. Each node $n \in \mathbf{V}$ represents a **non-divisible** sequential task of the program modelled by the task graph. An edge $e_{ij} \in \mathbf{E}$ represents the communication from task n_i to task n_j. The positive weight $w(n)$ of task $n \in \mathbf{V}$ represents its computation cost and the non-negative weight $c(e_{ij})$ of edge $e_{ij} \in \mathbf{E}$ represents its communication cost. Figure 1 depicts a sample task graph with 4 tasks; weights are displayed besides the tasks and edges. The set $\{n_x \in \mathbf{V} : e_{xi} \in \mathbf{E}\}$ of all direct predecessors (parents) of n_i is denoted by **parents**(n_i) and the set $\{n_x \in \mathbf{V} : e_{ix} \in \mathbf{E}\}$ of all direct successors (children) of n_i, is denoted by **children**(n_i). A task $n \in \mathbf{V}$ without parents, **parents**$(n) = \emptyset$, is named **source** task and if it is without children, **children**$(n) = \emptyset$, it is named **sink** task.

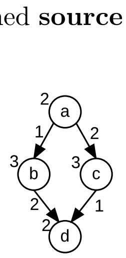


Figure 1: Graph with 4 tasks

The target parallel system consists of a finite set of **dedicated** processors $\mathbf{P}$ (there is **no preemption** of an executing task) connected by a communication network. It has the following properties: i) local communication has zero cost; ii) communication is performed by a communication subsystem; iii) communication can be performed concurrently; iv) the communication network is fully connected. This system model and its idealising assumptions are sometimes referred to as the classic model [23], as opposed to more advanced model that consider communication contention [24, 25].

Scheduling this task graph G on the processors $\mathbf{P}$ is the assignment of a **processor allocation** $proc(n)$ and a **start time** $t_s(n)$ to each task. The node's **finish time** is given by $t_f(n) = t_s(n)+w(n)$, i.e. the task's start time plus its computation costs, as homogeneous processors are assumed in this work. Let $t_f(P) = \max_{n\in\mathbf{V}:proc(n)=P}\{t_f(n)\}$ be the **processor finish time** of P and let $sl(\mathcal{S}) = \max_{n\in\mathbf{V}}\{t_f(n)\}$ be the **schedule length** (or makespan) of schedule $\mathcal{S}$, assuming $\min_{n\in\mathbf{V}}\{t_s(n)\} = 0$.

For such a schedule to be feasible, the following two conditions must be fulfilled for all tasks in G. The **Processor Constraint** enforces that only one task is executed by a processor at any point in time, which means for any two tasks n_i,$n_j \in \mathbf{V}$, $proc(n_i) = proc(n_j) \Rightarrow \left\{ \begin{array}{ll} & t_f(n_i) \leq t_s(n_j) \\ \text{or} & t_f(n_j) \leq t_s(n_i) \end{array} \right.$. The **Precedence Constraint** enforces that for every edge $e_{ij} \in \mathbf{E}$, $n_i, n_j \in \mathbf{V}$, the destination task n_j can only start after the communication associated with e_{ij} has arrived at n_j's processor P_{dst}, $t_s(n_j) \geq t_f(n_i) + \left\{ \begin{array}{ll} 0 & \text{if } proc(n_i) = proc(n_j) \\ c(e_{ij}) & \text{otherwise} \end{array} \right.$. Considering all parents, n_j's start time on processor P is constrained by the so called **Data Ready Time (DRT)** $t_{dr}(n_j, P)$. This is the time, when all communications from n_j's predecessors have arrived at P, given as

$$t_{dr}(n_j, P) = \max_{n_i\in\mathbf{parents}(n_j)} \left\{ t_f(n_i) + \begin{cases} 0 & \text{if } proc(n_i) = P \\ c(e_{ij}) & \text{otherwise} \end{cases} \right\} \tag{1}$$

If n_j is a source task then $t_{dr}(n_j, P) = 0$.

The objective of this scheduling problem is to find a schedule for G on the processors $\mathbf{P}$ with the shortest possible schedule length. This is the general $P|prec, c_{ij}|C_{max}$ problem in $\alpha|\beta|\gamma$ notation [7, 28], which is well known to be NP-hard in the strong sense [19]. The most popular approach in heuristic algorithms for this scheduling problem is **List Scheduling** [10, 31]. The simple algorithm has two components:

1. The tasks $\mathbf{V}$ of graph G are arranged in a list L, whereby the tasks are ordered according to a priority, which also ensures topological order.

2. Iterating over the list L, for every task n its earliest start time is evaluated on every processor $P \in \mathbf{P}$.

$$t_{EST}(n, P) = \max\{t_f(P), t_{dr}(n, P)\} \tag{2}$$

   Task n is positioned after all other tasks on P, which is called **end scheduling** [23]. n is scheduled onto the processor $P_{EST} \in \mathbf{P}$ that allows its **Earliest Start Time**, $t_{EST}(n) = \min_{P\in\mathbf{P}}\{t_{EST}(n, P)\}$.

Many variants of this basic algorithm have been proposed, varying the task priority and the selection of the processor [8, 11, 27, 30]. Despite its popularity, the created schedule can have a length that is greater than the sequential schedule on one processor, when the communication costs are high in comparison to the computation costs [23].

2.1 A* algorithm and scheduling

The new pruning techniques proposed in this paper are based on essentially the same A* implementation as proposed in [22]. It will be briefly described in the following.

A* is a best-first search algorithm that searches through a graph, visiting one node at a time. It does so by looking in each step at the most promising node, according to a cost function f, hence the best-first search. The nodes adjacent to the currently best one are added to the candidate nodes for the next step.

When A* is used for combinatorial optimisation, as in our case, the graph has the form of a tree. Each node (usually called state) of the tree corresponds to a partial solution of the problem to be optimised, which becomes more complete the deeper we get in the tree. The root of the tree is the initial, empty state s_{init}. While a depth-first search or a breadth-first search only looks at a limited number of possible next states in each step, A* considers all currently unexplored states and chooses the state s with the best cost value $f(s)$. To achieve that it uses a priority queue, called `OPEN`, into which all discovered but unexplored states are inserted, ordered according to their cost f. Taking the best state s from `OPEN` in each step, s is expanded creating new states. This is done by extending the partial solution represented by state s, making it more complete. For each of these newly created states the cost is computed and they are inserted into the OPEN queue. When a state has been fully expanded, it is removed from OPEN and placed into the CLOSED list. The purpose of CLOSED is to be able to detect duplicates, i.e. identical states, before states are inserted into OPEN. The algorithm continues with this process until the state with the lowest cost value is also a **goal state**, i.e. a complete solution. This state is the optimal solution, if certain requirements on the cost function f hold for all states (see Section 2.2). Algorithm 1 gives the pseudo-code for the basic A* algorithm.

Algorithm 1 A* algorithm

```
Require: OPEN is priority queue, ordered by ascending $f$ value of elements
 1: OPEN ← $s_{init}$
 2: while OPEN $\neq \emptyset$ do
 3:     $s$ ← headOf(OPEN)
 4:     if $s$ complete state then
 5:         return optimal solution $s$
 6:     Expand $s$ to new states NEW
 7:     for all $s_i \in$ NEW do
 8:         Calculate $f(s_i)$
 9:         Insert $s_i$ into OPEN, unless duplicate in CLOSED or OPEN
10:     CLOSED ←CLOSED+$s$; OPEN ←OPEN−$s$
```

To apply this A* algorithm to solve our scheduling problem, we essentially follow the algorithmic principle of list scheduling as described above. We formulate the following components:

- **State** s: A partial schedule where some tasks have been allocated and scheduled on the processors.
- **Initial state** s_{init}: The initial state represents an empty schedule, where no task has been scheduled yet.
- **Expansion operator**: Based on state s, a new state is created by scheduling one more task, hence growing the partial solution represented by s. A task that can be scheduled must be *free*, which means it must be either independent or all its predecessors have already been scheduled in the partial schedule of s. We denote the set of all those tasks as $\mathbf{free}(s)$. The number of new states expanded from s is then the product of all free tasks times the number of processors, $|\mathbf{NEW}| = |\mathbf{free}(s)| \cdot |\mathbf{P}|$. Each task of $\mathbf{free}(s)$ is scheduled on every processor $P \in \mathbf{P}$ as early as possible after all other tasks on the same processor according to eq. (2).
- **Cost function** f: The cost function $f(s)$ is an *underestimate* of the length of a complete schedule based on the partial schedule represented by s. We discuss the f function in the next section.

One can think of this A* algorithm as a List Scheduling algorithm that is modified so that it schedules the tasks in all possible orderings using all possible processor allocations.

2.2 f function

According to the A* principle [20], the $f(s)$ function is an *underestimate* of the exact minimum cost $f^*(s)$ of any goal state that is based on the state s. Applied to our scheduling problem, $f^*(s)$ is the minimum schedule length of all possible schedules that can be constructed using the partial schedule represented by s. If the function $f(s)$ fulfils $f(s) \leq f^*(s)$ for every state s, it is called *admissible.* With an admissible $f(s)$ A* is guaranteed to find an optimal solution. The number of examined states depends on how accurate $f(s)$ is, i.e. how close it is to $f^*(s)$. The f function we proposed has several components [22].

Idle time The first lower bound component is based on the, in general idealising, assumption that all processors will finish at the same time, $t_f(P) = sl(\mathcal{S}), \forall P \in \mathbf{P}$, in a schedule $\mathcal{S}$. If there are no idle times in $\mathcal{S}$, i.e. times were a processor does not execute any task, then the schedule length is given as the computation costs divided by the number of processors, $sl(\mathcal{S}) = \sum_{n \in \mathbf{V}} w(n)/|\mathbf{P}|$. Now let $idle(\mathcal{S})$ be the sum of all idle times in schedule $\mathcal{S}$, then the previous equation becomes, $sl(\mathcal{S}) = (\sum_{n \in \mathbf{V}} w(n) + idle(\mathcal{S}))/|\mathbf{P}|$. Given a partial schedule $\mathcal{S}_{partial}$ of state s, we can use this consideration to formulate a lower bound on the best achievable schedule length $sl^*(\mathcal{S}_{partial})$ for any complete schedule based on $\mathcal{S}_{partial}$, by assuming that the idle time $idle(\mathcal{S}_{partial})$ (or short $idle(s)$) will not increase:

$$f_{idle-time}(s) = \frac{\sum_{n \in \mathbf{V}} w(n) + idle(s)}{|\mathbf{P}|} \tag{3}$$

It should be easy to see, that $f_{idle-time}(s)$ can only be quite close to the final schedule length if good load balancing can be achieved, which usually implies low communication costs and/or few precedence constraints (i.e. edges). The next component addresses the situation when this is not the case.

Bottom level The longest path in the task graph, in terms of the sum of computation times of its tasks, is called the critical path. Its length is a lower bound for the schedule length. The concept of critical path can be generalised to so called node (here task) levels [23]. The computation bottom level of a task n_i is the length of the longest path starting in n_i, denoted by $bl_w(n_i)$. Recursively it is defined as$bl_w(n_i) = w(n_i) + \max_{n_j \in \mathbf{children}(n_i)}\{bl_w(n_j)\}$Given the start time of any node n, the schedule length is bounded by $t_s(n) + bl_w(n)$. In other words, after the task n has started execution, it still takes (at least) the time to sequentially execute all the tasks of the longest path starting in n. For a partial schedule $\mathcal{S}_{partial}$ of state s, maximising over all scheduled tasks leads to

$$f_{bl}(s) = \max_{n \in \mathcal{S}_{partial}} \{t_s(n) + bl_w(n)\} \tag{4}$$

This f function component is based on the already scheduled tasks. The next component extends the consideration to some unscheduled tasks.

Data Ready Time $t_{dr}(n, P)$, eq. (1), is the data ready time of task n on processor P. The minimum DRT of task n across all processors $\mathbf{P}$ is: $t_{dr}(n) = \min_{P \in \mathbf{P}}\{t_{dr}(n, P)\}$. Given a partial schedule $\mathcal{S}_{partial}$, we can use the DRT of all free tasks to bound the achievable schedule length. Corresponding to eq. (4), we now use the minimum DRT as a lower bound for the start time of any free task

$$f_{DRT}(s) = \max_{n \in \mathbf{free}(s)} \{t_{dr}(n) + bl_w(n)\} \tag{5}$$

Complete f function and initial value With equations (3),(4) and (5), the complete f function is

$$f(s) = \max\{f_{idle-time}(s), f_{bl}(s), f_{DRT}(s)\} \tag{6}$$

Using an incremental calculation of the f value, the complexity of calculating it can be considered constant [22]. From the definition of the complete f function and its components it follows that the f-value of the initial state s_{init} (no task has been scheduled yet) is

$$f(s_{init}) = \max\{\frac{\sum_{n \in \mathbf{V}} w(n)}{|\mathbf{P}|}, \max_{n \in \mathbf{V}}\{bl_w(n)\}\}, \tag{7}$$

which is the maximum of perfect load balancing (total computations divided by number of processors) and the length of the critical path of the graph, which is the highest bottom level among all tasks. This is also the lower bound on the optimal schedule length.

3 Pruning the search space

Despite the guidance of A*'s search through the solution space by the f function, the search space to be explored is still extremely large. This is no surprise as the problem remains NP-hard. Nevertheless, the efficiency of this A* based algorithm (and any other for that matter) can be dramatically improved by using problem specific knowledge to prune the solution space. Before we present the two new innovative pruning techniques in the next sections, we briefly review techniques previously proposed which are also used in our algorithm.

3.1 Duplication detection

Duplication detection is not a problem specific pruning technique per se, but not all formulations of state space exploration create duplicates. In our scheduling problem we do create duplicates. All possible global task orders spawn the solution space, but only the local order, i.e. the order among tasks on the same processor, is relevant. In other words, by using a different global order of scheduling, the same local orders can be created. As an example, consider two states (corresponding to two partial schedules) of the sample task graph (Figure 1). In both states, a, b, c have already been scheduled on two processors: a, c on P_1 and b on P_2. In one of the two states, the tasks were scheduled in the order a, b, c and in the other in the order a, c, b. This leads to the same partial schedule for both states and one of them can be discarded.

In Algorithm 1 we detect and discard duplicates by first checking in the `CLOSED` and `OPEN` lists if an identical state is present before inserting a new state into `OPEN`. Other pruning techniques are based on the ability to detect and discard duplicates.

3.2 Processor normalisation

For the pruning technique discussed in this subsection, we use the fact that in our scheduling problem the processors are homogeneous and connected in a homogeneous and completely symmetrical communication network. This means all processors are identical and exchangeable.

Consider the two partial schedules in Figure 2. While they are different, they become the same by swapping the processor names in the left schedule, i.e. P_1 becomes P_2 and P_2becomes P_1. There is no inherent preference for one or the other partial schedule in our optimisation problem and we can discard one.

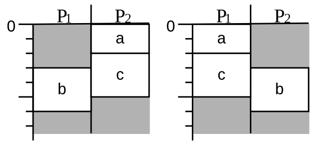


Figure 2: Two different schedules that can be made identical by normalisation

To prune such states, we normalise the processor names of the partial schedules of a state before checking for duplicates and inserting into `OPEN.` Please refer to [22] for the normalisation procedure.

3.3 Identical tasks

Task graphs may contain tasks that are identical in every aspect, especially in regular graphs. A trivial example is a set of independent tasks (there are no edges) with the same computation weight. In general we can say that all tasks in a set **A** are identical if the following requirements hold for any two tasks $n_i, n_j \in \mathbf{A}$ [22]:

- Same task weight: $w(n_i) = w(n_j)$
- Same parents: $\mathbf{parents}(n_i) = \mathbf{parents}(n_j)$
- Same children: $\mathbf{children}(n_i) = \mathbf{children}(n_j)$
- Same incoming edge weights: $c(e_{xi}) = c(e_{xj})\, \forall n_x \in \mathbf{parents}(n_i)$
- Same outgoing edge weights: $c(e_{iy}) = c(e_{jy})\, \forall n_y \in \mathbf{children}(n_i)$

In a less formal way, tasks are identical if one can in general swap any two tasks $n_i, n_j \in \mathbf{A}$ in a complete schedule without influencing the start time and processor allocation of any other task, hence maintaining the schedule length. While the above conditions seem very restrictive, there is an important graph (sub)structure for which they are often fulfilled, namely fork-joins where weights are regular.

Per the definition of identical tasks, all members of an identical task group will become free at the same time to be scheduled in our state space exploration. As they are independent of each other, our A* algorithm will consider all possible permutation orders. With the knowledge that they are identical, this is not necessary, any single fixed order suffices. Fixing such an order will prune the state space significantly. To achieve this we parse the input task graph at the start of the optimisation and insert virtual edges between the identical tasks of one group, enforcing a single ordering according to the indices of the tasks. When states are expanded during the A* scheduling algorithm, only one of the identical tasks becomes free at a time, enforcing the task ordering.

3.4 Heuristic schedules

Using a heuristic H to produce a valid schedule with length sl_H gives an upper bound on the optimal schedule length sl^*: $sl^* \leq sl_H$. All states that have a higher f function value, $f(s) > sl_H$, can be immediately discarded as their final schedule length cannot be optimal. While this does not reduce the search space (those states would never be expanded by A*), in reduces the number of states in `CLOSED` and `OPEN`, which increases the performance of A*.

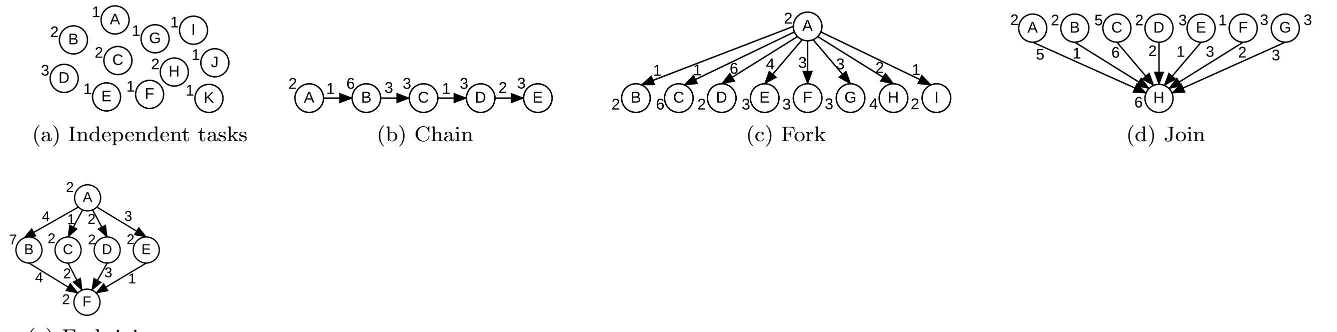
This optimisation was already employed in the original proposal of our A* algorithm in [22] and in [17]. In this paper we propose to go further and use the heuristic solution for pruning. If we put the heuristic solution as a complete state s_H into the `OPEN` queue, with $f(s_H) = sl_H$, then we can discard every state that has an f value greater than or *equal to* the heuristic schedule length, $f(s) \geq sl_H$. While this does not prune the solution space in general, it does so when the heuristic schedule has optimal length, $sl^* = sl_H$. The pruned states are then partial schedules that have an f function value equal to the heuristic solution, hence their final schedule length would be at least that long. One might wonder why A* is used if the heuristic solution is already optimal. The reason is of course that it is in general not known that it is optimal and it has to be proven with A*. This also leads to the observation that this pruning is extremely efficient when the heuristic length is equal to the lower bound on the optimal length, hence provably optimal.

In this work a simple list scheduling as presented in Section 2 is employed and tasks are ordered by their bottom level, here including the communication costs: $bl(n_i) = w(n_i) + \max_{n_j \in \mathbf{children}(n_i)}\{c(e_{ij}) + bl(n_j)\}$.

4 Fixed task order

The experiments of [22] demonstrated the importance of pruning techniques such as duplication detection, processor normalisation and identical tasks. Unfortunately, the latter two techniques are only effective under certain circumstances: for a larger number of processors and (partially) regular graphs, respectively.

In this and the next section, we are investigating more sophisticated pruning techniques that apply to more optimisation instances. A very special concern was with simple graph (sub-)structures, such as independent, fork, join and fork-join (Figure 3). While their structures are very simple, they are often more difficult to optimise with the A* algorithm as described so far, compared to more complex structures, such as irregular series-parallel graphs [22]. This seems like a paradox, as heuristics often handle simple structures quite well. The reason for the worse performance of such graphs lies in the higher degree of task ordering freedom. In all of the above mentioned structures, almost all tasks are independent of each other (except for the source and sink tasks), hence creating many valid task orderings. Formulated in a different way, the constraints imposed by the edges do very little in terms of reducing the search space. In order to develop a general pruning technique, we look at these structures step by step and use the insights to create new pruning techniques.



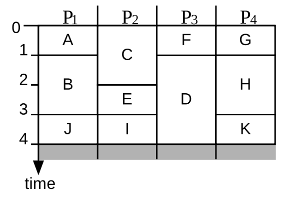
(a) Independent tasks (b) Chain (c) Fork (d) Join

(e) Fork-join

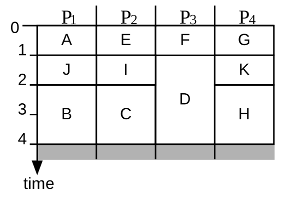
Figure 3: Simple task graph structures

4.1 Independent tasks

Starting with the simplest structure, we look at a graph without edges, i.e. a set of independent tasks. When no edges are involved, the optimisation problem is actually different, as we only need to find the optimal processor allocation. The ordering of the tasks is not relevant any more. This can be quickly illustrated by considering the two different schedules of the independent tasks (Figure 3a) on four processors in Figure 4. Observe that the processor allocations are the same but not the task orderings. For our A* algorithm that means we could ignore the ordering and reduce the solution space by a factor $|\mathbf{V}|!$ for the optimisation of this special case.



(a) Schedule of independent tasks



(b) Other schedule of independent tasks

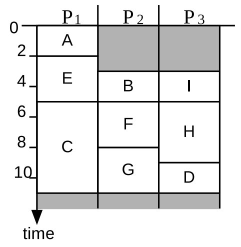
Figure 4: Task order does not matter for independent tasks

4.2 Fork and join

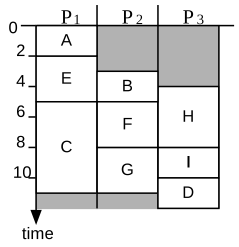
Next we want to carefully and systematically introduce some edges into the graphs we consider. Two extreme cases can be distinguished: *1)* chains (e.g. Figure 3b) or *2)* forks/joins (e.g. Figures 3c and 3d). They are extreme in the sense that with $|\mathbf{V}| - 1$ edges, they expose the minimum and maximum amount of possible concurrency, respectively. Scheduling a chain is trivial and we do not need to further consider it. For an optimal schedule, all tasks must be executed on the same processor (to avoid communication costs) and the order is enforced by the precedence constraints.

A fork graph has one source (or root) task, all other tasks are children of this task and there are no other edges, illustrated in Figure 3c. A join graph has the opposite structure, where there is one sink task, all tasks are parents of this task and there are no other edges, see Figure 3d. Fork and join graphs are simple and common, but have the highest degree of ordering freedom given their $|\mathbf{V}| - 1$ edges. The only precedence constraint is that the source (sink) task must be scheduled first (last).

Yet, due to the communication costs, the local ordering of the tasks now becomes relevant. Figure 5a shows an (optimal) schedule of Figure 3c's fork. With the same processor allocation, but a different local ordering (H and I are swapped on P_3) the schedule length is increased as can be seen in the schedule of Figure 5b. Cause is the longer incoming communication time of H ($c(e_{AH}) = 2$) compared to I's ($c(e_{AI}) = 1$), resulting in a later start of H. The same relevance of ordering can be observed for join graphs due to the outgoing communication to the sink task.



(a) Optimal schedule of fork (Fig. 3c)



(b) Non-optimal schedule of fork (Fig. 3c)

Figure 5: Scheduling fork graph on 3 processors

So, in contrast to completely independent tasks, the ordering of the mutually independent tasks in forks/joins is no longer irrelevant. Note that this is only true, because the scheduling problem includes communication cost. Without its consideration, the order of those tasks would not matter.

Despite these observations, scheduling a fork or join graph is not significantly more complex than scheduling completely independent tasks. The reason lies in the fact that we can straightforwardly order the tasks allocated to one processor.

Definition 1 (Fork-order and join-order)

- ***Fork-order****: It suffices to order the tasks of a fork graph according to non-decreasing in-edge weight. Let us call this fork-order. The task with the lowest incoming communication cost should start first, then the one with the next lowest edge cost and so forth.*
- ***Join-order****: For a join graph, the task with the highest outgoing communication should be scheduled first, followed by the next highest and so forth. We call this join-order.*

So scheduling a fork or join graph is still only a processor allocation problem.

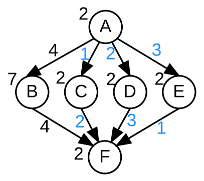
4.3 Fork-join

A natural extension of the fork and join structure is their combination into a fork-join structure, as depicted in Figure 3e. This is a very typical structure encountered in parallel programs and thus also a common sub-structure of larger graphs. A fork-join graph has one source task, one sink task and all other tasks are children of the source task and parents of the sink task. There are no other edges. A question which naturally arises is whether we can use the fork-order or join-order studied in the previous sub-section. Unfortunately, we cannot.

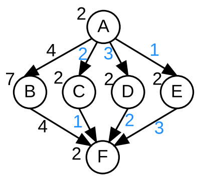
Regard Figure 6, where three similar fork-join graphs are scheduled on two processors. The three graphs only differ in the weights of the edges incident on tasks C, D, E, as highlighted in the figures. In the first graph (Figure 6a), the optimal schedule (Figure 6d) is achieved by applying the fork-order, i.e. C, D, E, B. Remember that the order of A and E is fixed by the precedence constraints and that only the local order, i.e. the task order on each processor, is relevant. Hence B's position is not relevant in this constructed example. Task C must start first on P_2 as it has the lowest incoming communication cost $c(e_{AC}) = 1$. If tasks D or E were at this position, their start delay would be longer resulting in a longer schedule. Further, task E must be last due to its lowest outgoing communication weight $c(e_{EF}) = 1$. Any other task at this position would delay the start of F until after time unit 10.

When we consider the next graph (Figure 6b), we see that a join-order results in the optimal schedule. The tasks are ordered by the non-increasing cost $c(e)$ of their out-edges, resulting in the order B, E, D, C. Important is here that the task C is the last to be executed on P_2 as it has the shortest communication delay $c(e_{CF}) = 1$. Equally important is that task E is executed first on P_2 with the same justification we had for the fork-order.

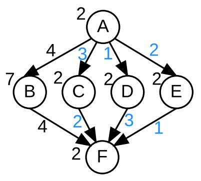
In the last graph, Figure 6c, the weights of the edges are such that neither a fork-order nor a join-order will lead to the optimal solution. Task D needs to be first and task E needs to be last with the same reasons as before. But we cannot calculate this automatically by only looking at the in-edges or only at the out-edges.



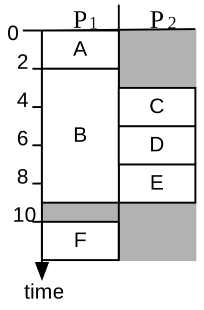
(a) Fork-order graph



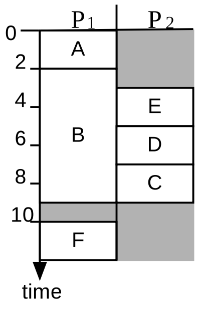
(b) Join-order graph



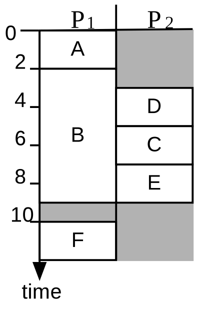
(c) No-order graph



(d) Schedule of Fig. 6a



(e) Schedule of Fig. 6b



(f) Schedule of Fig. 6c

Figure 6: Optimal task order depends on in-edge *and* out-edge

Still, fork-order and join-order can still be useful for many practical fork-join graphs. It is not uncommon that either the incoming edges or the outgoing edges have identical edge weight. This happens, for example, when all tasks receive the same data or when all task produce a simple result of the same size. For such a case we can use the following procedure to obtain an ordering of the graph's tasks.

Definition 2 (Fork-join order)

1. *Order all tasks in fork-order. When doing so break ties, i.e. decide the order of tasks with the same in-edge weight, based on non-increasing out-edge costs.*

2. *Verify that the tasks are also in join-order. If yes, the order is valid, it can be used to order the tasks locally, otherwise not.*

4.4 Pruning: Fixing task order

We could now formulate special versions of our A* algorithm for independent tasks, for forks and for joins. But such algorithms would already fail if we only departed slightly from the specified graph structures, for example mostly independent tasks with only a handful of edges. The local task order does generally matter for such graphs. Our objective is to use the foregoing observations to create a general pruning technique. The intention is to use the observations of the previous sections whenever we come across a corresponding sub-structure in a graph.

This objective is achieved by considering the tasks that are free to be scheduled when expanding state s; $\mathbf{free}(s)$ is the set of all these tasks (Section 2.1). Remember that A* will create $|\mathbf{P}|$ new states for each task $n \in \mathbf{free}(s)$, in each of which task n is scheduled on a different processor. The resulting branching factor is $\mathbf{free}(s) \times |\mathbf{P}|$, i.e. $\mathbf{free}(s) \times |\mathbf{P}|$ new states are created from s. The higher the branching factor, the larger the solution space we have to search. For graph structures like independent tasks, fork and join the set of free tasks is large, containing most (if not all) tasks still to be scheduled. But in the previous sections we have just established that we can easily determine the order in which the tasks should be scheduled, without searching through all possible orderings. If the tasks in $\mathbf{free}(s)$ fulfil certain conditions, we can fix their order. This means we can select a single task from $\mathbf{free}(s)$ for the expansion, reducing the branching factor to $|\mathbf{P}|$.

We now need to establish the conditions under which we can select a single task for expansion and how to select that task. The conditions are established by generalising the observations from the previous section. For every task $n_f \in \mathbf{free}(s)$ the following condition must hold.

Condition 1 (Fixing order)

1. *n_f has at most **one parent** and at most **one child**:*

$$|\mathbf{parents}(n_f)| \leq 1 \text{ and } |\mathbf{children}(n_f)| \leq 1 \tag{8}$$

2. *if n_f has a child, $|\mathbf{childs}(n_f)| = 1$, then it is the **same child** as for any other task in $\mathbf{free}(s)$:*

$$|\bigcup_{n \in \mathbf{free}(s)} \mathbf{children}(n)| = 1 \tag{9}$$

3. *if n_f has a parent, $|\mathbf{parents}(n_f)| = 1$, then all other **parents** of tasks in $\mathbf{free}(s)$ are **allocated to the same processor** P_p:*

$$\forall n_p \in \bigcup_{n \in \mathbf{free}(s)} \mathbf{parents}(n), \, proc(n_p) = P_p \tag{10}$$

Observe that these three condition points are fulfilled by the mutually independent tasks of the structures we have discussed in the previous subsections, i.e. independent tasks, fork/join and fork-join. More importantly, they also hold for mixtures of those graph structures, for example a fork or a join together with independent tasks, shown in Figures 7a and 7b. To illustrate this further consider Figure 7c, which shows a fraction of a partially scheduled task graph. Tasks U, V, W have already been scheduled, hence so have their ancestors. In the situation depicted, the tasks A to G are free tasks and form the set $\mathbf{free}(s)$. Assuming that U, V, W are allocated to the same processor P_p, all tasks in $\mathbf{free}(s)$ fulfil the three conditions above.

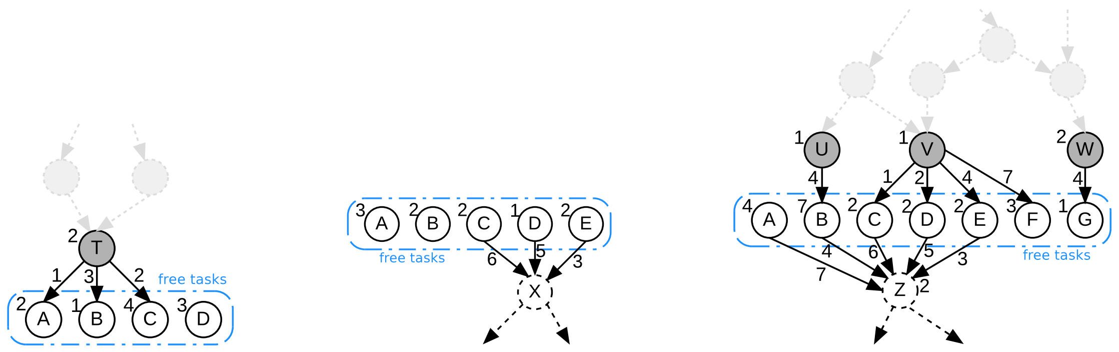
Fulfilling Condition 1 is necessary but not sufficient. Before the order of the free tasks can be fixed, we need to sort them and verify that they are in a feasible order. Akin to the fork-order and join-order (Section 4.2) the sorting condition is as follows.

Condition 2 (Task sorting)

1. *Sort tasks of $\mathbf{free}(s)$ by their non-decreasing data ready time (eq. (1)), assuming communication is remote. Here, this is data ready time of task $n_i \in \mathbf{free}(s)$ is the finish time of the parent n_p plus the weight of the edge e_{pi}*

$$t_{dr}^{P \neq P_p}(n_i) = t_f(n_p) + c(e_{pi}) \tag{11}$$

Without a parent, this time is set to zero.



(a) Fork & independent (b) Join & independent (c) Fork, join & fork-join

Figure 7: Examples of free tasks whose order can be fixed

2. *Break ties by sorting according to non-increasing out-edge costs $c(e_{ic})$. If there is no out-edge, set cost to zero.*

3. *Verify that tasks of* **free**(s) *are in non-increasing out-edge cost order.*

So this corresponds to the order procedure of Section 4.3, with the extension that the finish times of the parent tasks are considered. This is necessary, because more than one parent is allowed, under the condition that they are all scheduled on the same processor P_p. Note that the data ready time of task $n_i \in$ **free**(s) on $P = P_p$ is different to (11), which could lead to a different overall ordering. However, on P_p all incoming communications for any task $n_i \in$ **free**(s) will be local, hence have zero cost. The ordering then does not matter, in terms of the incoming communication.

Let us apply this sorting procedure to the free tasks displayed in Figure 7c, with the assumption that tasks U, V, W are tightly scheduled in this order on P_p (i.e. $t_s(U) = t$, $t_s(V) = t + 1$, $t_s(W) = t + 2$). The resulting order is then A, C, D, B, E, G, F. This order is also a valid non-increasing out-edge cost order, hence the task order can be fixed.

Once the order of the task in **free**(s) is fixed it also remains fixed in the child state, say $s + 1$, the child of the child $s+2$ and so forth until a new task becomes free (or until **free** is empty, which means the schedule is complete). If no task becomes free, there is no need to recheck the conditions or perform the sorting of the remaining tasks.

An example for a case where a new task does become free is given in Figure 7c. After task E has been scheduled in some descendent state, say s_x, the task Z becomes free. Now, depending on whether G, F, Z fulfil all Condition 1 and can be sorted (Condition 2), the order of **free**(s_x) might be fixed or not. One might wonder if the freeing of Z has any bearing on the validity of the foregoing fixing of the free task order. The answer is no, because the fixed order is such as to minimise any created idle time (i.e. gaps between tasks). Scheduling G and/or F earlier could only have created more, not less idle time (due to their late incoming communications). They also have no out-going edges (if they had, their child would be Z), hence scheduling them earlier has no impact on any other task.

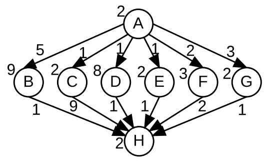
So the new Fixed Task Order pruning technique to be used in our A* based algorithm has been proposed. Before we move on, we need to check if it interferes with the other pruning techniques employed so far, which are duplication detection, identical tasks pruning and processor normalisation (Section 3). Neither the duplication detection nor the processor normalisation have any correlation with Fixed Task Order pruning. Fixing the task order will actually avoid the creation of duplicates during the fixing. The processor normalisation has no impact on the task order, as only processors are renamed and every task is still allocated to every processor in each step.

The identical tasks pruning only puts one of the identical tasks in the **free**(s) set. After that task has been scheduled, the next identical task (if any remaining) will be inserted into **free**$(s + 1)$, as it has been freed due to the virtual edge. But that new task fulfils the same conditions as its equivalent task in state s, hence either the order can be fixed for both **free**(s) and **free**$(s + 1)$ or for none. As a consequence, all identical tasks of one group will be scheduled continuously.

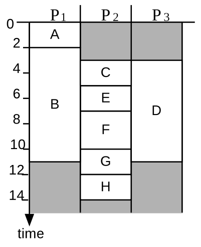
While the taken approach of fixing the task order of free tasks makes this a general technique usable with every task graph, the conditions to be fulfilled are quite strict. They will be most efficient for the simple structures as discussed before, or in graphs were such sub-structures are common, for example series-parallel graphs. Nevertheless, the considerations in this section are also the basis for the pruning technique proposed in the next section that generalises this further.

5 Equivalent schedules

To develop the novel pruning technique let us start with a motivating example. Figure 8 displays a fork graph and an optimal schedule of it on three processors. The fork is such that the order of the mutually independent tasks A to G cannot be fixed: in the fork-order, E needs to be before F (due to $c(e_{AE}) < c(e_{AF})$), but this is the other way around in the join-order (due to $c(e_{FH})l > c(e_{EH})$). However, when we look at the schedule, we see that we can partially change the order of the tasks on processor P_2. Given that the sink task H is allocated to P_2, we can have many different task orders on P_2 that still result in the same schedule length. The only requirement is that either C or E comes first, due to the smallest incoming communication cost ($c(e_{AC}) = c(e_{AE}) = 1$). But even without knowing where H is allocated to, there is still some freedom. For example task E and G can be swapped as they have the same out-edge weight ($c(e_{EH}) = c(e_{GH}) = 1$). In other words, there are many partially different schedules which are equivalent and result in the same schedule length. We want to use that observation for pruning.



(a) Fork graph



(b) Optimal schedule on 3 processors

Figure 8: Motivating example

5.1 Scheduling horizon

A (partial) schedule is defined by the starting times of the tasks and their processor allocations. But as we just have seen, the order of some tasks can be irrelevant for the final schedule length. So, when examining the important properties of a partial schedule of a certain subset of tasks, we realise that it is only relevant what starting times the partial schedule permits for the remaining unscheduled tasks. Only that will determine the final schedule length. As we know, the starting time of a task is influenced by two aspects: *1)* the processor finish times (Section 2); and *2)* the data ready time (eq. (1)).

Figure 9 depicts a partial schedule of some tasks of a graph on four processors. On purpose not all is shown to emphasise the fact that not all details are relevant for the subsequent scheduling of the remaining tasks. Highlighted by a dashed blue line is the contour of the finish times of the processors. This is in general relevant for the start times of tasks to be scheduled; they cannot start earlier. Equally important are the data arrival times of communications to yet unscheduled tasks. In other words, these are the edges that have no scheduled target yet. They are symbolised by dashed arrows in the figure.

These two aspects (for the given set of scheduled tasks) fully define a partial schedule for our optimisation purpose. We call this set of information the **scheduling horizon**. If we have two different partial schedules for the same subset of tasks that have an identical scheduling horizon, they are fully exchangeable without any impact on the best final schedule length that can be achieved. Or seen from the point of view of our A* algorithm, one of the two partial schedules can be discarded.

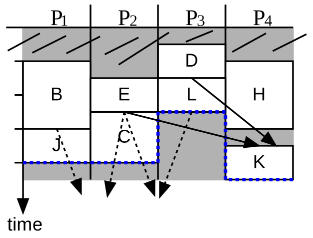
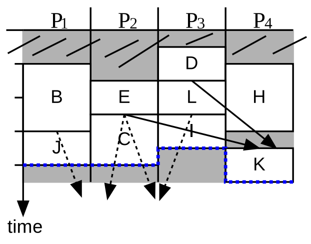


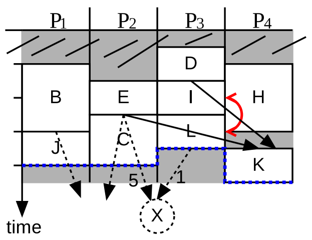
Figure 9: Schedule horizon

5.2 Swapping tasks

So, how can we formulate this into a pruning technique? A* takes a partial schedule and adds one more task to it, thereby creating new states, one for each processor. That is the point were our pruning technique can hook in. To illustrate this consider Figure 10 where one more task, namely I, has been scheduled to the partial schedule of Figure 9. The scheduling horizon is of course altered by the later finish time of P_3, but there are no outgoing communications of I. What we now try to do is to bring the tasks on P_3 into a certain order, here an alphabetical order. Using an iterative approach, we try to swap I with its foregoing task, here L. This results in the partial schedule depicted in Figure 10b. We do not further "bubble up´´ task I as the tasks on P_3 are now in alphabetical order. As can be seen, the finish time of P_3 has not changed, but task L has an outgoing communication as indicated by the dashed edge. This communication will now arrive later at its destination X (unless X is scheduled on P_3 where communication costs are zero). Nevertheless, the later arrival of this communication will not delay the start of X. Task E also sends data to X and the higher weight of edge e_{EX} (together with the finish time of E) dominates the data ready time of X. In conclusion, the schedule horizon of the partial schedule has not been changed by the task swap.



(a) One more task



(b) Tasks swapped

Figure 10: Reorder tasks

Only one of the two orders needs to be considered in the search through the solution space as the best schedule length possible with the partial schedule before the swap will be the same as after the swap. Hence we have **equivalent schedules**. A possible pruning approach at this stage is to normalise the order of the tasks and then use duplication detection in the same way we do with processor normalisation. A subtle problem impedes this approach. When we swap the order of two tasks and schedule each task as early as possible[1] according to eq. (2), it might happen that we *reduce* idle time present in the partial schedule before the new task was scheduled. In other words, we can have less idle time with one more task scheduled, i.e. a better partial schedule. This can result in an improved f-value from state s to state $s+1$, due to $f_{idle-time}(s+1) < f_{idle-time}(s)$. Such inconsistency is already bad for the performance of an A* algorithm [20], but even worse could lead to an inadmissible f-function that would destroy A*´s ability to find the optimal solution.

Instead of normalising the task order, we take a more aggressive approach. After scheduling the next free task, let us call it m, we still try to reorder the tasks into a certain order (more later) by swapping. But as soon as a swap is found that does not worsen the schedule horizon compared to the initial position of m, we immediately discard the state. This approach has even two advantages compared to the above considered normalisation: *1)* it is not necessary to bring the tasks into a final order, the procedure can terminate early; *2)* the computational effort of duplication detection and insertion into `OPEN` is saved (the number of states in `CLOSED` and `OPEN` can be huge). The reason this approach can be taken lies in the fact that all possible task orders are explored by A*, including the one we try to order it into. Hence, whenever a partial schedule for a state s_x can be reordered with the same or better schedule horizon, we know that this reordered partial schedule is created in another state, say s_y, as well, therefore we can delete the state s_x. We will later discuss any possible interference with other pruning techniques.

5.3 Pruning: Equivalent schedules

The Equivalent Schedule Pruning analyses every new state, more precisely, the new partial schedule by trying to bring the newly added task m on processor P to a certain position among the tasks already on P. To define and compute the order of the tasks and the position where task m should be "bubbled up´´ to, each task n is given an index $index(n)$, with $0 \leq index(n) \leq |\mathbf{V}| - 1$. When sorted in ascending index order, the tasks must also be in a topological order. One single defined order is necessary, as we need to know which states to discard (the ones where

[1]Tasks must start as early as possible, otherwise the same processor allocation and task order could lead to different schedules.

the order can be changed) and which states to keep (the ones where the order cannot be changed). Without a fixed order, it could be possible to discard all states of a subtree, which possibly contains the only optimal solution. The topological order saves the need to test for dependences while bubbling up the task, as the analysis stops when we encounter a task with a lower index than $index(m)$.

Algorithm 2 Equivalence check

```
Require: Each task $n \in \mathbf{V}$ has a unique index $0 \leq index(n) \leq |\mathbf{V}| - 1$; in ascending index order tasks are also in
    a topological order
    Input: Partial schedule, including last scheduled task $m$ at end of processor $P$
 1: $t^{max} = t_f(m)$
 2: Name tasks on $P$ $n_1$ to $n_l$ in start time order                    $\triangleright$ $n_l = m$
 3: $i \leftarrow l - 1$
 4: while $i \geq 0 \wedge index(m) < index(n_i)$ do
 5:     Swap position of $m$ and $n_i$
 6:     Schedule $m$ and $n_i \ldots n_{l-1}$ each as early as possible
 7:     if $t_f(n_{l-1}) \leq t^{max} \wedge$ OutgoingCommsOK($n_i \ldots n_{l-1}$) then
 8:         return EQUIVALENT
 9:     $i \leftarrow i - 1$
10: return NOT EQUIVALENT
```

Algorithm 2 shows the pseudo code for the equivalent schedule pruning. The input of the procedure is the partial schedule belonging to a newly created state, where m has been scheduled at the end of processor P. In the first iteration of the while-loop, we try to swap the position of m with the forgoing task n_{l-1}(the tasks on P are named $n_1$to n_l). The two tasks m and n_{l-1} are started as early as possible, according to eq. (2), and it is checked that the finish time of processor P (i.e. the finish time of the now last task n_{l-1}) has not increased. It is also checked that the outgoing communications of n_{l-1} do not delay the start of any descendent task in comparison to the original partial schedule. This is done in the procedure `OutgoingCommsOK()`, Algorithm 3, more details later. If the two checks are positive, the swap of the two tasks did not worsen the schedule horizon, thus the schedule is equivalent and can be dropped. If not, in the next iteration of the while-loop the algorithm tries to move m further "up´´ by swapping with task n_{l-2}, hence the last three tasks on P become m, n_{l-2}, n_{l-1}. All tests are done again and this procedure is repeated until m is the first task on P or the task to swap with has a lower index, i.e. the while-loop terminates. When this happens it means that there was no early return, so the state is not equivalent and therefore kept.

Testing that the outgoing communications of the swapped tasks do not delay any descendants is more involved than might be assumed at first, as illustrated in Algorithm 3. Our testing is limited to the direct descendants, i.e. the children of the moved tasks $n_i \ldots n_{l-1}$. Conceivably, a delayed child might still not result in a worse schedule horizon, e.g. the children of the child can be unaffected due to slack time in the schedule. However, such testing would be much more involved and could require the rescheduling of tasks and is therefore not considered. Remember also that A* will generate many states and any pruning test needs to be repeated many times. Consequently, there is a trade-off between perfect pruning and the average time spent per state.

Now to the details of Algorithm 3. For each task $n_k \in \{n_i \ldots n_{l-1}\}$ that starts after task m it is necessary to test that the start of any child $n_c \in \mathbf{children}(n_k)$ is not delayed. If n_k starts as early as before the swap, there is no delay for the children. If not, we have to distinguish whether n_c has already been scheduled or not. If yes, it suffices to test that the data from n_k arrives before n_c starts. If not, the testing becomes more difficult. n_k's later start is still admissible if data from any other parent of n_c arrives later than or at the same time as n_k´s communication, verified *on all processors*. Considering all processors is necessary here, as the child n_c has not yet been scheduled and can therefore be allocated to any processor. The calculation of the arrival time of data from parents of n_c has also to distinguish between the cases that the parent has or has not been scheduled. When the parent has not yet been scheduled, the best case for the data arrival is assumed (not shown in Algorithm 3 for clarity). When no problem is discovered for any of the tasks $n_i \ldots n_{l-1}$, the algorithm returns true, signalling that the schedule horizon it terms of outgoing communications has not degraded.

Algorithm 3 Subroutine `OutgoingCommsOK`($n_i \dots n_{l-1}$)

```
for all $n_k \in \{n_i \dots n_{l-1}\}$ do
  if $t_s(n_k) > t_s^{orig}(n_k)$ then                                  ▷ check only if $n_k$ starts later
    for all $n_c \in$ children($n_k$) do
      $T \leftarrow t_f(n_k) + c(e_{kc})$                                  ▷ remote data arrival from $n_k$
      if $n_c$ scheduled then
        if $t_s(n_c) > T \wedge proc(n_c) \neq P$ then                     ▷ on same proc always OK
          return false
      else                                                                 ▷ $n_c$ not scheduled yet
        for all $P_i \in \mathbf{P}/P$ do                                   ▷ $n_c$ can be on any proc; $P$ always OK
          $atLeastOneLater \leftarrow$ false
          for all $n_p \in$ parents($n_c$) $- n_k$ do
            if data arrival from $n_p \geq T$ then
              $atLeastOneLater \leftarrow$ true
          if $atLeastOneLater =$ false then
            return false
return true
```

5.4 Discussion

Interference with other pruning techniques

An essential assumption of Equivalent Schedule Pruning is that all possible task orders, i.e. all permutations permitted by the precedence constraints, are included in the search space. The danger is that other pruning techniques might not fulfil this premise. In that case a schedule could be considered as equivalent and therefore be dropped, even though the corresponding other schedule (already in the right task order) is never generated.

With this in mind, an analysis of the other four presented pruning techniques is necessary. Combining this pruning technique with duplication detection or processor normalisation is not an issue, as they do not alter the premise that all permutations are explored. However, Fixed Task Order Pruning and identical tasks pruning invalidate this assumption. Let us consider the latter first.

With identical tasks pruning (Section 3.3), only one task of an identical task group is in the free set at any point in time. Effectively, this enforces a fixed order among these identical tasks, hence not all orderings are part of the search space. Fortunately, it is easy to make sure that no state is dropped by Equivalent Task Pruning. The simple solution is to make sure that the index order of the identical tasks corresponds to the order enforced by the virtual edges. The first task of the identical task group has the lowest index, the (virtual) child of this task the next lowest and so forth. By enforcing this, it is guaranteed that no reordering can occur among those tasks (the test for equivalence) and therefore no equivalent schedule pruning will be performed. A lower indexed task will always be scheduled before a higher indexed task of the same group, hence the latter can never bubble up beyond the former. Note that Equivalent Schedule Pruning still works with all other tasks, also in combination with the identical tasks, but this is OK, as all corresponding permutations of the tasks are part of the search space.

The situation is different with the last pruning technique to consider, Fixed Task Order Pruning. As in the previous case, not all orderings of the tasks are part of the search space. The order is fixed based on the rules defined in Section 4.4. In general, the resulting order of the free tasks of a state is not compatible with the index order of the tasks. Hence, bubbling up of a task could occur[2], which would lead to a state being dropped by the pruning without having the equivalent schedule in the search space. For that reason, Equivalent Schedule Pruning is disabled when the order of tasks is fixed. It is disabled not only while the task order is fixed, but for the entire sub-tree rooted in the state where the task order was fixed first. It is important to note that this pruning would not be of any advantage while the tasks are fixed anyway, because only one single permutation of the tasks is considered, in other words there is nothing to prune.

Avoiding number of states increase

In general, Equivalent Schedule Pruning will decrease the number of states that have to be explored until an optimal solution is found – quite significantly so as will be seen in evaluation in Section 6. However, it is possible that this

[2]Note that although the fixed order is guaranteed to lead to an optimal solution, there could be another order with the *same* (not better) schedule horizon.

pruning technique leads to more states being explored. Such a situation can occur under the following scenario: the lower bound on the schedule length, which is the initial f-value (eq. (7)), is tight, i.e. the optimal schedule length equals this bound. In such a situation no state exists that has an f-value lower than the optimal schedule length. It follows that as soon as we have a complete solution (schedule) with that f-value the algorithm can terminate. A* can be very fast on such scheduling problem instances, going deep into the search tree very quickly and finding the optimal solution very fast.

With Equivalent Schedule Pruning it is now possible that the pruning drops promising partial schedules as they are recognised to be equivalent. In the worst case, all equivalent schedules are checked first, before A* progresses deeper into the search tree. In other words, the partial schedule where no task can be "bubbled up´´ is generated last. Fortunately, it is relatively easy to address this problem. The solution is to make the algorithm create and examine states which are already in the right (index) order first. This can be achieved by adjusting the OPEN priority queue such that these states are at the head of the queue compared to other tasks with the same f-value and number of scheduled tasks.

In our implementation we achieve this by using partial expansion [22]. In this A* optimisation, not all states are expanded from a state s in one go (i.e. $\mathbf{free}(s) \times |\mathbf{P}|$ new states), but only the states belonging to one of the free tasks (i.e. $|\mathbf{P}|$ new states). If the f-value of one of those new states, say s_x, is the same as $f(s)$, we expand s_x next and go deeper in the search very quickly. The order in which the tasks are considered for the partial expansion is the index order of the tasks, hence the order where there is no "bubbling up´´.

6 Evaluation

In this evaluation section we study how good the proposed pruning techniques are at reducing the search space. To do that, a large set of graphs was scheduled onto different numbers of processors.

We used two versions of our A* algorithm for the scheduling: 1) the best algorithm as used in [22] (called OLD) and 2) the same algorithm plus the new pruning techniques proposed in this paper (called NEW). As a reminder, these are Fixed Task Order (Section 4), Equivalent Schedules (Section 5) and Heuristic Schedules (Section 3.4).

Our interest is not directly in the runtime of the algorithms, but in the number of states they need to explore to find the optimal solution. Our hypothesis is that the new pruning techniques will very significantly reduce the search space, i.e. the number of states to explore. We further wanted to investigate the influence of graph characteristics on the size of the solution space. To cope with the resulting large number of graphs, the execution time of the algorithms was limited to *one minute*, after which the algorithm was terminated and the schedule considered as not completed. A desirable consequence of the large number of graphs we employ is also that unintended but unavoidable differences between the two algorithms, such as a different ordering of the tasks due to pruning technique requirements, which might lead to variations in performance, have little impact on the overall evaluation and conclusions.

The algorithms were implemented in Java using a single thread and a heap space of 2.5GByte. The initial OLD algorithm was executed on an Intel Xeon E5320 @ 1.86GHz, while the new results using the NEW algorithm were obtained on an Intel Core 2 Duo P9400 @ 2.4 GHz. The initial system was not any more available when the improvements were tested, so we carefully chose a system with very comparable performance. The frequencies are different, but the additional difference in architecture results in quite comparable processors. Remember, that we are not directly measuring the runtime, but the number of states needed, which is independent of the used processor.

6.1 Workload

The graphs of the workload cover a wide spectrum of graph types, sizes and characteristics, summarised in Table 1 In contrast to scheduling heuristics, the runtime of the A* algorithm is strongly influenced by the structure of the graph and its density and very little by the computation or communication weight and their relation [22]. To acknowledge that, we have used ten different graph structures, ranging from basic, but common structures, such as fork and join, to more complex graphs such as series-parallel (sp) graphs and stencil. All non-fixed parts of the graph structures were created randomly, such as branching factor of the trees or the structure of series-parallel graphs. Also the weights of the nodes and edges were determined at random from a uniform distribution. To achieve different communication to computation ratios, $CCR = \frac{\sum_{e\in\mathbf{E}} c(e)}{\sum_{n\in\mathbf{V}} w(n)}$, the weights were scaled correspondingly.

The parameters size and CCR were varied in all their combinations for every graph structure. In total the workload consisted of 751 graphs which were scheduled on 2,4 and 8 processors, resulting in 2253 schedules.

structures	independent, fork, join, fork-join, in-trees, out-trees, random (densities 0.5,1,2,5), series-parallel (sp), pipeline, stencil
numbers of tasks	10,12,13,14,15,16,18,21,24,27,30
weights (task & edge)	random, uniform distribution
CCR	0.1,1,2,10

Table 1: Characteristics of graph workload

6.2 Results

Over all schedules, the number of processed states by the NEW A* algorithm within the time limit varied between 2 and almost 76 million states[3], whereby the average was 5.7 million. Let us first regard Table 2, where we analyse the number of schedules that were completed within the time limit. This is done for all schedules (row 2) and also differentiated by the number of processors on which the graphs were scheduled (rows 3-5). Shown are the total completion rate, how many more and how many less schedules were finished in comparison to the OLD algorithm that did not use the new pruning techniques. As one can see, NEW is able to complete many more schedules within the time limit. There are only two outliers (less than 0.1%), were OLD performed better (more to this later). Overall, the completion rate improved by 32.2%. As one can expect, this improvement was more pronounced for 2 processors (38.4%), than for 4 processors (32.5%) and 8 processors (25.3%). Schedules with more processors simply need more time in general and could not be completed within the time limit even with the new pruning techniques.

	schedules	completed	more completed	less completed	improvement
all	2253	(1467) 65.1%	(359) 15.9%	(2) 0.1%	32.2%
$p = 2$	751	(533) 71.0%	(148) 19.7%	(0) 0.0%	38.4%
$p = 4$	751	(469) 62.5%	(117) 15.6%	(2) 0.1%	32.5%
$p = 8$	751	(465) 61.9%	(94) 12.5%	(0) 0.0%	25.3%

Table 2: Completed schedules within time limit (1min)

The results of Table 2 demonstrate an impressive improvement, but we cannot quantify by how many states the search was reduced, when the schedules were not completed by OLD. To analyse that, we now concentrate on the schedules that did complete with both algorithms. Table 3 shows for the schedules that were completed with both algorithm versions in how many cases the number of states are reduced, equal or increased. As one can see, for the overwhelming number of schedules, the new pruning techniques resulted in a state reduction. This was of course expected, but as the few cases where the number of states increases show, it is not guaranteed. Nevertheless, the new pruning techniques are effective for almost all problem instances, in other words for all types of graphs.

	analysed schedules	less states	equal states	more states
all	1098	(1055) 95.2%	(33) 3.0%	(20) 1.8%
$p = 2$	385	(357) 92.7%	(11) 2.9%	(17) 4.4%
$p = 4$	352	(338) 96.0%	(11) 3.1%	(3) 0.9%
$p = 8$	371	(360) 97.0%	(11) 3.0%	(0) 0.0%

Table 3: State reduction of completed schedules

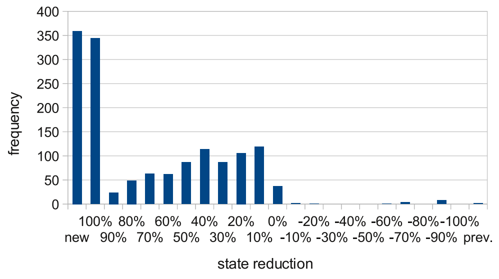
Now we would like to understand better when and by how much the state reductions occur and why sometimes more states are needed despite the new pruning techniques. Figure 11a depicts a frequency plot over the percentage of state reduction for all schedules. Each bar shows the number of schedules for which the state reduction is between the percentage label and the next lower label. The leftmost bar shows the schedules that are newly completed and the rightmost bar the ones that are only completed with OLD. The 1055 schedules (Table 3) for which the number of states was reduced are spread over all positive percentages, but there is a strong clustering for 100%-90% state reduction (2nd bar from left). For about one third of the completed schedules the required states have been reduced dramatically, thus the new pruning techniques are extremely successful. We will see below that this reduction is more often found with certain graph structures than with others. Reductions due to list scheduling pruning are

[3]Only two states (initial state and heuristic state) are needed, when the list scheduling heuristic provides a provable optimal result, hence its makespan equals the lower bound.

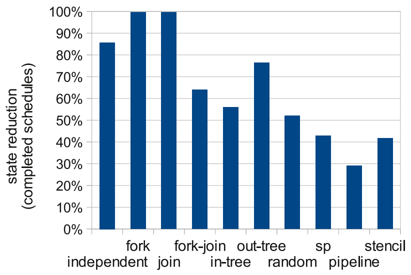
most effective when the schedule created by list scheduling is provably optimal, which is the case if the schedule length equals the lower bound. In those cases the number of states is reduced to only 2, hence also leading to a very strong reduction.

To analyse further why the state reduction varies, regard Figure 11b, where the average state reduction is depicted for the graph types. As we can see, the reduction is the strongest for independent, fork and join. And that is exactly what one would have expected given the strength of the proposed pruning techniques. For those graph types, the fixed task order pruning reduces the search space very significantly by fixing the order of all tasks. In essence, these problems become processor allocation problems only, as the ordering is completely fixed. But also fork-join, in-tree, out-tree and random have benefited with more than 50% reduction on average. As fixing the task order is only partially possible in those graphs, the equivalent schedules pruning is also quite efficient for those graph types.

Let us confirm these observations by studying how the completion rate has improved per graph type. Figure 12a shows the completion percentage for each graph type. The bars have two parts, the schedules completed with OLD (blue) and additionally completed with NEW (red). We observe two things: 1) The completion rate has improved dramatically for fork and join, and very significantly for fork-join, in-trees, out-trees; 2) the improvements are for those graph types that had the worst performance with OLD. The first point confirms our observation about the effectiveness of the new pruning techniques regarding graph types. At the same time the second point shows that the new pruning techniques have really improved the weak parts of the OLD algorithm as intended.



(a) Frequency of state state reduction



(b) Average state reduction per graph type

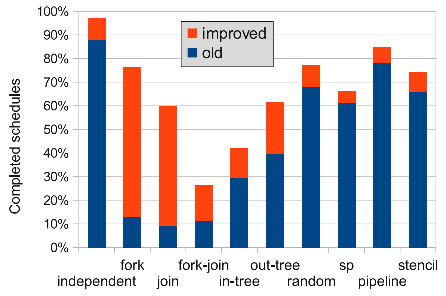
Figure 11: State reduction

It remains to analyse the few cases for which NEW needs more states than OLD. The two schedules that were completed with OLD but not with NEW (Table 2) are both for random graphs and four processors. Figure 12b depicts the change of number of states per graph type, considering the completed schedules as in Table 3. 100% represents all completed schedules for the corresponding graph types and the different regions (less, equal, more) indicate for which percentage of the schedules less, equal or more states are necessary in comparison to OLD. Again, fork, join, fork-join, in-tree and out-tree achieve a perfect improvement with less states for every schedule. Only for random and stencil graphs there are a few cases where more states are needed, 19 schedules and 1 schedule respectively.

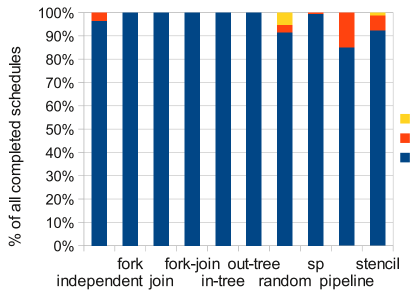
It lies in the nature of A* that under certain conditions pruning or an improved cost function $f(s)$ can lead to more states. For example, pruning can detect that a state is equivalent and then discard it, but this state could have let directly to an optimal solution. A* still finds the optimal solution, but it it might take more states. Apparently this kind of behaviour is more often triggered by the random graph structure. To judge the significance of this behaviour on the value of the new pruning techniques, we have analysed the actual number of states needed in these cases. The average number of states for those 20 cases using the NEW algorithm is 87 thousand, and the maximum is about 1 million, which is both low compared to the global values (average 5.7 million, maximum 76 million states). In other words, the number of states increased for schedules where it was not critical, except for the two cases that could not be completed.

7 Conclusions

This paper addressed the optimal solution of the task scheduling problem with communication delays ($P|prec, c_{ij}|C_{max}$). It carefully investigated two novel pruning techniques that significantly reduce the solution



(a) Completion improvement per graph type



(b) Change of number of states per graph type

Figure 12: Completion improvement and change of states

space of this problem: Fixed Task Order and Equivalent Schedules. Using them with our previously published A* scheduling algorithm dramatically reduced the size of the solution space. The extensive experimental evaluation has shown that the new pruning techniques are specially successful for relatively simple graph structures which were difficult to schedule before: fork, join, fork-join and trees. The contributions made by this paper make optimal scheduling of small task graphs more feasible.

A simple graph structure that still remains difficult to schedule optimally is fork-join. Our current research investigates new pruning and cost estimation techniques to address this. Also, we intend to improve the general A* algorithm to move to memory limited versions such as SMA* or as IDA* [21], which will allow to tackle larger problems given sufficient time.

References

[1] A. Bender. MILP based task mapping for heterogeneous multiprocessor systems. In *Proc. of European Design Automation Conference*, pages 190–197. IEEE, 1996.

[2] M. I. Daoud and N. Kharma. A high performance algorithm for static task matching and scheduling. *Journal of Parallel and Distributed Computing*, 68(4):399–409, 2008.

[3] A. Davare, J. Chong, Q. Zhu, D. Densmore, and A.Sangiovanni-Vincentelli. Classification, customization, and characterization: Using milp for task allocation and scheduling. Technical Report UCB/EECS-2006-166, University of California, Berkeley, December 2006.

[4] T. Davidović, L. Liberti, N. Maculan, and N. Mladenović. Towards the optimal solution of the multiprocessor scheduling problem with communication delays. In *In Proc. 3rd Multidisciplinary Int. Conf. on Scheduling: Theory and Application (MISTA)*, 2007.

[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] M. Drozdowski. *Scheduling for Parallel Processing.* Springer, 2009.

[7] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[8] 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.

[9] C. Hanen and A. Munier. An approximation algorithm for scheduling dependent tasks on m processsors with small communication delays. In *ETFA 95:INRIA/IEEE Symposium on Emerging Technology and Factory Animation*, pages 167–189. IEEE Press, 1995.

[10] T. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[11] 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.

[12] R. Hwang, M. Gen, and H. Katayama. A comparison of multiprocessor task scheduling algorithms with communication costs. *Computer & Operations Research*, 35:976–993, 2008.

[13] S. Jin, G. Schiavone, and D. Turgut. A performance study of multiprocessor task scheduling algorithms. *The Journal of Supercomputing*, 43:77–97, 2008.

[14] D. Kadamuddi and J. J. P. Tsai. Clustering algorithm for parallelizing software systems in multiprocessor environment. *IEEE Transactions on Parallel and Distributed Systems*, 26(4), April 2000.

[15] M. Kafil and I. Ahmad. Optimal task assignment in heterogeneous distributed computing systems. *IEEE Concurrency*, 6:42–51, 1998.

[16] V. Kianzad and S. S. Bhattacharyya. Efficient techniques for clustering and scheduling onto embedded multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):667–680, 2006.

[17] 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.

[18] D. Poole and A. Mackworth. *Artificial Intelligence: Foundations of Computational Agents.* Cambridge University Press.

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

[20] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, second edition, 2003.

[21] Stuart S. Russell. Efficient memory-bounded search methods. In *Proc. of the 10th European conference on Artificial intelligence*, ECAI '92, 1992.

[22] A. Z. Semar Shahul and O. Sinnen. Scheduling task graphs optimally with A*. *The Journal of Supercomputing*, 51(3):310–332, 2010.

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

[24] O. Sinnen and L. Sousa. Scheduling task graphs on arbitrary processor architectures considering contention. In *High Performance Computing and Networking (HPCN'01)*, volume 2110 of *Lecture Notes in Computer Science*, pages 373–382. Springer-Verlag, 2001.

[25] O. Sinnen and L. Sousa. Experimental evaluation of task scheduling accuracy: Implications for the scheduling model. *IEICE Transactions on Information and Systems*, E86-D(9):1620–1627, September 2003.

[26] T. Thanalapati and S. Dandamudi. An efficient adaptive scheduling scheme for distributed memory multicomputer. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):758–768, 2001.

[27] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Perfromance-effective and low complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[28] B. Veltman, B. J. Lageweg, and J. K. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16(2-3):173–182, 1990.

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

[30] M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, July 1990.

[31] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.

[32] 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.