A Duplicate-Free State-Space Model for Optimal Task Scheduling

Michael Orr and Oliver Sinnen

Department of Electrical and Computer Engineering, University of Auckland, New Zealand

Abstract. The problem of task scheduling with communication delays ($P|prec, c_{ij}|C_{\max}$) is NP-hard, and therefore solutions are often found using a heuristic method. However, an optimal schedule can be very useful in applications such as time critical systems, or as a baseline for the evaluation of heuristics. Branch-and-bound algorithms such as A* have previously been shown to be a promising approach to the optimal solving of this problem, using a state-space model which we refer to as exhaustive list scheduling. However, this model suffers from the possibility of producing high numbers of duplicate states. In this paper we define a new state-space model in which we divide the problem into two distinct subproblems: first we decide the allocations of all tasks to processors, and then we order the tasks on their allocated processors in order to produce a complete schedule. This two-phase state-space model offers no potential for the production of duplicates. An empirical evaluation shows that the use of this new state-space model leads to a marked reduction in the number of states considered by an A* search in many cases, particularly for task graphs with a high communication-to-computation ratio. With additional refinement, and the development of specialised pruning techniques, the performance of this state-space model could be further improved.

1 Introduction

In order to use the full potential of a multiprocessor system in speeding up task execution, efficient schedules are required. In this work, we address the classic problem of task scheduling with communication delays, known as $P|prec, c_{ij}|C_{\max}$ using the $\alpha|\beta|\gamma$ notation [11]. The problem involves a set of tasks, with associated precedence constraints and communication delays, which must be scheduled such that the overall finish time (schedule length) is minimised. The optimal solving of this problem is well known to be NP-hard [7], so that the amount of work required grows exponentially as the number of tasks is increased. For this reason, many heuristic approaches have been developed, trading solution quality for reduced computation time [3, 12, 4, 9]. Unfortunately, the relative quality of these approximate solutions cannot be guaranteed, as no α-approximation scheme for the problem is known [2].

Although the NP-hardness of the problem usually discourages optimal solving, an optimal schedule can give a significant advantage in time critical systems

or applications where a single schedule is reused many times. Optimal solutions are also necessary in order to evaluate the effectiveness of a heuristic scheduling method. Branch-and-bound algorithms have previously shown promise in efficiently finding optimal solutions to this problem [8], but the state-space model used, exhaustive list scheduling (ELS), was prone to the production of duplicate states. This paper presents a new state-space model, in which the task scheduling problem is tackled in two distinct phases: first allocation, and then ordering. The two-phase state-space model (abbreviated AO) does not allow for the possibility of duplicate states.

In Sec. 2, background information is given, including an explanation of the task scheduling model, an overview of branch-and-bound algorithms, and a description of the ELS model. Section 3 describes the new AO model, and how a branch-and-bound search is conducted through it. Section 4 explains how the new model was evaluated by comparison with the old one, and presents the results. Finally, Sec. 5 gives the conclusions of the paper and outlines possible further avenues of study.

2 Background

2.1 Task Scheduling Model

The specific problem that we address here is the scheduling of a task graph $G = \{V, E, w, c\}$ on a set of processors P. G is a directed acyclic graph wherein each node $n \in V$ represents a task, and each edge $e_{ij} \in E$ represents a required communication from task n_i to task n_j. The computation cost of a task $n \in V$ is given by its positive weight $w(n)$, and the communication cost of an edge $e_{ij} \in E$ is given by the non-negative weight $c(e_{ij})$. The target parallel system for our schedule consists of a finite number of homogeneous processors, represented by P. Each processor is dedicated, meaning that no executing task may be preempted. We assume a fully connected communication subsystem, such that each pair of processors $p_i, p_j \in P$ is connected by an identical communication link. Communications are performed concurrently and without contention. Local communication (from p_i to p_i) has zero cost.

Our aim is to produce a schedule $S = \{proc, t_s\}$, where $proc(n)$ allocates the task to a processor in P, and $t_s(n)$ assigns it a start time on this processor. For a schedule to be valid, it must fulfill two conditions for all tasks in G. The Processor Constraint requires that only one task is executed by a processor at any one time. The Precedence Constraint requires that a task n may only be executed once all of its predecessors have finished execution, and all required data has been communicated to $proc(n)$. The goal of optimal task scheduling is to find such a schedule S for which the total execution time or schedule length $sl(S)$ is the lowest possible.

It is useful to define the concept of node levels for a task graph [9]. For a task n, the top level $tl(n)$ is the length of the longest path in the task graph that ends with n. This does not include the weight of n, or any communication costs.

Similarly, the bottom level $bl(n)$ is the length of the longest path beginning with n, excluding communication costs. The weight of n is included in $bl(n)$.

2.2 Branch-and-Bound

The term branch-and-bound refers to a family of search algorithms which are widely used for the solving of combinatorial optimisation problems. They do this by implicitly enumerating all solutions to a problem, simultaneously finding an optimal solution and proving its optimality [1]. A search tree is constructed in which each node (usually referred to as a state) represents a partial solution to the problem. From the partial solution represented by a state s, some set of operations is applied to produce new partial solutions which are closer to a complete solution. In this way we define the children of s, and thereby *branch*. Each state must also be *bounded*: we evaluate each state s using a cost function f, such that $f(s)$ is a lower bound on the cost of any solution that can be reached from s. Using these bounds, we can guide our search away from unpromising partial solutions and therefore remove large subsets of the potential solutions from the need to be fully examined.

A* is a particularly popular variant of branch-and-bound which uses a best-first search approach [5]. A* has the interesting property that it is optimally efficient; using the same cost function f, no search algorithm could find an optimal solution while examining fewer states. To achieve this property, it is necessary that the cost function f provides an underestimate. That is, it must be the case that $f(s) \leq f^*(s)$, where $f^*(s)$ is the true lowest cost of a complete solution in the subtree rooted at s. A cost function with this property is said to be *admissable*.

2.3 Exhaustive List Scheduling

Previous branch-and-bound approaches to optimal task scheduling have used a state-space model that is inspired by list scheduling algorithms [8]. States are partial schedules in which some subset of the tasks in the problem instance have been assigned to a processor and given a start time. At each branching step, successors are created by putting every possible ready task (tasks for which all parents are already scheduled) on every possible processor at the earliest possible start time. In this way, the search space demonstrates every possible sequence of decisions that a list scheduling algorithm could make. This branch-and-bound strategy can therefore be described as exhaustive list scheduling.

Branch-and-bound works most efficiently when the subtrees produced when branching are entirely disjoint. Another way of stating this is that there is only one possible path from the root of the tree to any given state, and therefore there is only one way in which a search can create this state. When this is not the case, a large amount of work can be wasted: the same state could be expanded, and its subtree subsequently explored, mulitple times. Avoiding this requires doing work to detect duplicate states, such as keeping a set of already created

states with which all new states must be compared. This process increases the algorithm's need for both time and memory.

Unfortunately, the ELS strategy creates a lot of potential for duplicated states [10]. This stems from two main sources: firstly, since the processors are homogeneous, any permutation of the processors in a schedule represents an entirely equivalent schedule. This means that for each truly unique complete schedule, there will be $|P|!$ equivalent complete schedules in the state space. This makes it very important to use some strategy of processor normalisation when branching, such that these equivalent states cannot be produced. The other source of duplicate states is more difficult to deal with. When tasks are independent of each other, the order in which they are selected for scheduling can be changed without affecting the resulting schedule. This means there is more than one path to the corresponding state, and therefore a potential duplicate. The only way to avoid these duplicates is to enforce a particular sequence onto these scheduling decisions. Under the ELS strategy, however, no method is apparent in which this could be achieved while also allowing all possible legitimate schedules to be produced.

3 Duplicate-Free State-Space Model

Both sources of duplicate states can be eliminated by adopting a new state-space model (AO), in which the two dimensions of task scheduling are dealt with separately. Rather than making all decisions about a task's placement simultaneously, the search proceeds in two stages. In the first stage, we decide for each task the processor to which it will be assigned. We refer to this as the allocation phase. The second stage of the search, beginning after all tasks are allocated, decides the start times of each task. Given that each processor has a known set of tasks allocated to it, this is equivalent to deciding on an ordering for each set. Therefore, we refer to this as the ordering phase. Once the allocation phase has determined the tasks' positions in space, and the ordering phase has determined the tasks' positions in time, a complete schedule is produced. Essentially, we divide the problem of task scheduling into two distinct subproblems, each of which can be solved separately using distinct methods.

3.1 Allocation

In the allocation phase, we wish to allocate each task to a processor. Since the processors in our task scheduling problem are homogeneous, the exact processor on which a task is placed is unimportant. What matters is the way the tasks are grouped on the processors. The problem of task allocation is therefore equivalent to the problem of producing a partition of a set. A partition of a set X is a set of non-overlapping subsets of X, such that the union of the subsets is equal to X. In other words, the set of all partitions of X represents all possible ways of grouping the elements of X. Applying this to our task scheduling problem, we find all possible ways in which tasks could be grouped on processors. In

the allocation phase, we are therefore searching for an optimal partition of the set V, consisting of all tasks in our task graph. The search is conducted by constructing a series of partial partitions of V. A partial partition A of V is defined as a partition of a set V', $V' \subset V$ [6]. At each level of the search we expand the subset V' by adding one additional task, until $V' = V$ and all tasks are allocated. At each stage, the task n selected can be placed into any existing set $a \in A$, or alternatively, a new set can be added to A containing only n.

A search using this method has the potential to produce every possible partition of V, and there is only one possible path to each partition. In this way, we remove the first source of duplicates: there is no possibility of producing allocations that differ from each other only by the permutation of processors. If we are allocating tasks to a finite number of processors, we simply limit the number of sets allowed in a partial partition to the same number. This has no effect other than to reduce the search space by disregarding partitions consisting of a larger number of sets.

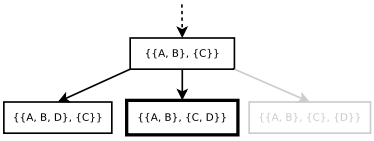


Fig. 1: Branching in the allocation state-space with a maximum of two processors.

Allocation Heuristic. In order to guide our branch-and-bound search towards an optimal partition, we need a heuristic by which to determine f-values for each state s. In the case of allocation, there are two crucial types of information we can obtain from a partial partition A which allow us to determine a lower bound for the length of a resulting schedule. The first, and simplest, is the total computational weight of the tasks in each grouping. Even without waiting for communication, each processor must take at least the sum of the computation times of its assigned tasks to finish. The length of the overall schedule must therefore be at least as long as the time needed for the most heavily loaded processor to perform its computations.

$$f_{load}(s) = \max_{a \in A} \left\{ \sum_{n \in a} w(n) \right\}$$

The second bound derives from our knowledge of which communication costs must be incurred, and is obtained from the length of the allocated critical path of the task graph; that is, the longest path through the task graph given the particular set of allocations. The critical path as calculated only includes the weights of edges for which both tasks have been allocated. If two tasks are allocated to the same processor, an edge between them is considered to have

a length of zero and does not extend the critical path. However, if they are allocated to different processors, the communication cost is incurred and the length of the critical path may increase. The allocated critical path represents the longest sequence of computations and communications that we know must occur given this allocation. Therefore, again, the resulting schedule must be at least as long.

$$f_{\text{acp}}(s) = \max_{n \in V'} \{tl_{\text{a}}(n) + bl(n)\}$$

Since we want the tightest bound possible, the maximum of these two bounds is taken as the final f-value.

$$f_{\text{alloc}}(s) = \max\{f_{\text{load}}(s), f_{\text{acp}}(s)\}$$

By their nature, these two bounds oppose each other; lowering one is likely to increase the other. The shortest possible allocated critical path can be trivially obtained simply by allocating all tasks to the same processor, but this will cause the total computational weight of that processor to be the maximum possible. Likewise, the lowest possible computational weight on a single processor can be achieved simply by allocating each task to a different processor, but this means that all communication costs will be incurred and therefore the allocated critical path will be the longest possible. Combining these two bounds guides the search to find the best possible compromise between computational load-balancing and the elimination of communication costs.

3.2 Ordering

In the ordering phase, we begin with a complete allocation, and our aim is to produce a complete schedule S. After giving an arbitrary ordering to both the sets in A and the processors in P, we can define the processor allocation in S such that $n \in a_i \implies proc(n) = p_i$. Our remaining task is to determine the optimal start time for each task. Given a particular ordering of the tasks $n \in p_i$, the best start time for each task is trivial to obtain, as it is simply the earliest it is possible for that task to start. To complete our schedule we therefore only need to determine an ordering for each set of tasks $p_i \in P$. Our search could proceed by enumerating all possible permutations of the tasks within their processors.However, it is likely that many of the possible permutations do not describe a valid schedule. This will occur if any task is placed in order after one of its descendants (or before one of its ancestors).

In order to produce only valid orderings, an approach inspired by list scheduling is taken. In this variant, however, each processor p_i is considered separately, with a local ready list $R(p_i)$. Initially, a task $n \in p_i$ is said to be locally ready if it has no predecessors also on p_i. At each step we can select a task $n \in R(p_i)$ and place it next in order on p_i. Those tasks which have been selected and placed in order are called *ordered,* while those which have not are called *unordered.* In general, a task $n \in p_i$ belongs to $R(p_i)$ if it has no unordered predecessors also on p_i. After a task n has been ordered, each of its descendants on p_i must be

checked to see if this condition has now been met, in which case they will be added to $R(p_i)$. Following this process to the end, we can produce any possible valid ordering of the tasks on p_i.

Producing a full schedule requires that this process be completed for all processors in P. At each level of the search, we can select a processor $p_i \in P$ and order one of its tasks. The order in which processors are selected can be decided arbitrarily; however, in order to avoid duplication, it must be fixed by some scheme such that the processor selected can be determined solely by the depth of the current state. The simplest method to achieve this is to proceed through the processors in order: first order all the tasks on p_1, then all the tasks on p_2, and so on to p_n. Another method is to alternate between the processors in a round-robin fashion. Unlike in exhaustive list scheduling, tasks are not guaranteed to be placed into the schedule in topological order. When a task is ordered, its predecessors on other processors may still be unordered, and therefore their start times may not be known. During the ordering process, therefore, a task n may only be given an *estimated earliest start time* $eest(n)$. For all unordered tasks, $eest(n) = tl_{\mathrm{a}}(n)$. For ordered tasks, we first define $prev(n)$ as the task ordered immediately before n. We also define the estimated data ready time $edrt(n_j) = max_{n_i \in parents(n_j)} \{eest(n_i) + w(n_i) + c(e_{ij})\}$. Where $prev(n)$ does not exist, $eest(n) = edrt(n)$. Otherwise, $eest(n) = max(eest(prev(n)) + w(prev(n)), edrt(n))$.

In this way, we have solved the problem of duplicates arising from making the same decisions in a different order. By allocating each task to a processor ahead of time, and enforcing a strict order on the processors, it is no longer possible for these situations to arise. Where before we might have placed task B on p_2 and then task A on p_1, we now must always place task A on p_1 and then task B on p_2. Unfortunately, in a small number of cases, the combination of valid local orders for all processors produces an overall schedule with an invalid global ordering. Consider a partial schedule as a graph in which there is a directed edge from each task to the one placed in order after it, and also where interprocessor communications occur. For invalid states, this graph will display a cycle. In our implementation, such states cause the f-value of a state to be increased indefinitely; they are therefore detected and removed from consideration once the f-value reaches an upper bound (e.g., the sequential schedule length).

Ordering Heuristic. The heuristic for determining f-values in the ordering stage follows a similar pattern to that for allocation. As we assign estimated start times to tasks, it is possible that we introduce idle time to a processor in which it will not be executing any task. The difference between the heuristics for allocation and ordering lies in the incorporation of these idle times. For each state s, the current estimated finish time of a processor p_i is the latest estimated finish time of any task $n \in p_i$ which has so far been ordered. This estimated finish time must include both the full computation time of each task already ordered on p_i, as well as any idle time incurred between tasks.

With this in mind, we define our two bounds like so: first, the latest estimated start time of any task already ordered, plus the allocated bottom level of that task. We refer to this as the partially scheduled critical path, as it corresponds to the allocated critical path through our task graph, but with the addition of the now known idle times.

$$f_{\text{scp}}(s) = \max_{n \in ordered(s)} \{eest(n) + bl_{\text{a}}(n)\}$$

Second, the latest finish time of any processor in the partial schedule, plus the total computational weight of all tasks allocated to that processor which are not yet scheduled.

$$f_{\text{ordered-load}}(s) = \max_{p \in P} \left\{ t_{\text{f}}(p) + \sum_{n \in p \cap unordered(s)} w(n) \right\}$$

Again, this corresponds to the total computational load on a processor with the addition of now known idle times. To obtain the tightest possible bound, the maximum of these bounds is taken as the final f-value.

$$f_{\text{order}}(s) = \max\{f_{\text{scp}}(s), f_{\text{ordered-load}}(s)\}$$

3.3 Combined State-Space

Solving a task scheduling problem instance requires both the allocation and ordering subproblems to be solved in conjunction. To produce a combined state-space, we begin with the allocation search tree, S_{A}. The leaves of this tree represent every possible distinct allocation of tasks in G to processors in P. Say that leaf l_i represents allocation a_i. We produce the ordering search tree S_{O_i} using a_i. The leaves of S_{O_i} represent every distinct complete schedule of G which is consistent with a_i. For each leaf l_{a_i}, we set the root of tree S_{O_i} as its child. The result is a tree S_{AO}, the leaves of which represent every distinct complete schedule of G on the processors in P.

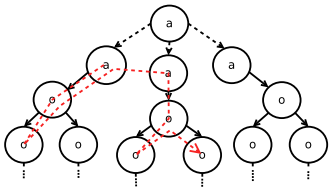


Fig. 2: A possible search path through the combined state space.

A branch-and-bound search conducted on this state-space will begin by searching the allocation state-space. Each allocation state representing a complete allocation has one child state, which is an initial ordering state with this

allocation. When considering the allocation subproblem in isolation, we define the optimal allocation as that which has the smallest possible lower bound on the length of a schedule resulting from it. Unfortunately, these lower bounds cannot be exact and therefore it is not guaranteed that the allocation with the smallest lower bound will actually produce the shortest possible schedule. This means that in the combined state-space, a number of complete allocations may be investigated by the search, having their possible orderings evaluated. The tighter the bound which can be calculated, the more quickly the search is likely to be guided toward a truly optimal allocation.

A search of this state-space model is theoretically able to benefit from several pruning techniques already developed for ELS. Namely, these are identical task pruning, equivalent schedule pruning, fixed order pruning and heuristic list scheduling [10].

4 Evaluation

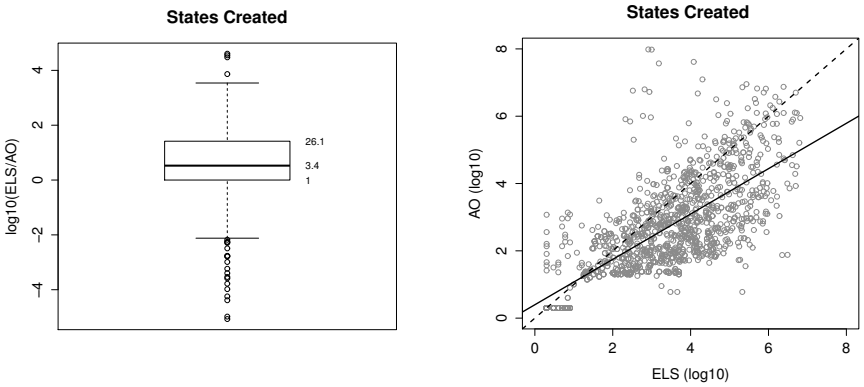
4.1 Experimental Methodology

The AO model was evaluated empirically by comparison with ELS. The evaluation was performed by running branch-and-bound searches on a diverse set of task graphs using each state-space model. Task graphs were chosen that differed by the following attributes: graph structure, the number of tasks, and the communication-to-computation ratio (CCR). Almost 500 task graphs with unique combinations of these attributes were selected. An optimal schedule was found for each task graph using 2, 4, and 8 processors, once each for each state-space model. Searches were performed using the A* search algorithm. Pruning techniques were applied to each state-space model that could take advantage of them. Common to both state space models were identical task pruning, equivalent schedule pruning, and heuristic list scheduling. With ELS, additional pruning techniques were applied: processor normalisation, fixed order pruning, and the use of a closed list.

The implementations were built with the Java programming language. An existing implementation of ELS was used as the basis for an AO implementation, with code for common procedures shared wherever possible. Notably, the basic implementation of the A* search algorithm is shared, with the implementations differing only by how the children of a search node are created. The implementations of commonly applicable pruning techniques are also shared. All tests were run on a Linux (Ubuntu 12.04) machine with 64 processing cores and 256 GB of RAM. The Java environment was version 1.7, running on the OpenJDK 64-Bit Server JVM. Tests were single-threaded, so only one core was utilised by our code, but the JVM's concurrent garbage collector could potentially have benefited from all 64 cores. For all tests, the JVM was given a maximum heap size of 192 GB. A new JVM instance was started for every search, to minimise the possibility of previous searches influencing the performance of later searches due to garbage collection and JIT compilation.

The dependent variable measured was the number of states created in the course of a search. States created includes states which are removed from consideration immediately after their creation by pruning techniques. The number of states created by a search is considered as a more reliable metric for evaluating the performance of a search algorithm, as it can be made deterministic and relies only on the details of the algorithm itself. The time taken for a search to complete is extremely dependent on the environment in which the program is run, as it can vary greatly based on the processing speed of the hardware.

4.2 Results



(a) (b)

Fig. 3: Overall comparison between state-space models.

The overall results, presented in Fig. 3, show that AO performs better than ELS in the majority of cases. Data is displayed on a $\log_{10}$ scale both because of its extreme range, and so that positive (> 1) and negative (< 1) IFs have equal weighting on the axis. The improvement factor (IF) for a particular problem instance is determined by dividing a particular metric for a search with ELS by the same metric for a search with the AO model. Figure 3a shows that the lower quartile IF for the overall dataset was zero; AO performed better than ELS in roughly 75% of cases. As indicated by the median, in roughly 50% of cases AO performed at least 3.4 times better than ELS. At its best, the AO model led to a reduction in the number of states created by a factor of more than 10^4. On the other hand, at its worst, almost 10^5 times more states were created. In Fig. 3b we see a direct case-by-case comparison of the performance of the two models. The central dotted line represents equal performance. Cases above this line are those where AO performed worse, while those below are those where it performed better. The solid regression line shows the average

improvement of AO increasing with the number of states created by ELS. This plot therefore shows us that cases where AO's performance was worse were more often smaller, less difficult problem instances. Of the variables surveyed, the one with the most dramatic impact on the relative performance of the state-space models was the communication-to-computation ratio. Figure 4 shows a clear trend towards better performance for the AO model as the CCR increases.

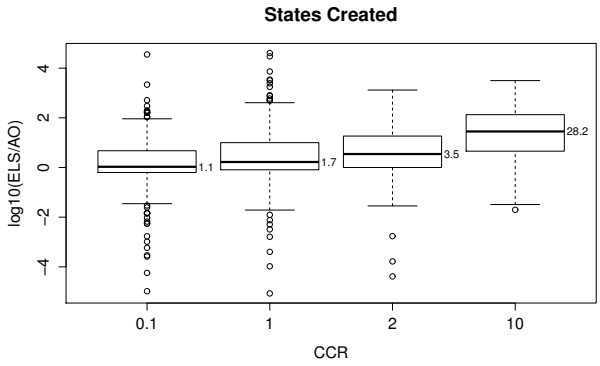


Fig. 4: Comparison by CCR.

The results indicate that the new AO model is generally superior to the more mature ELS model, with an increased advantage in certain classes of scheduling problem. Most obviously, these are problems in which communication costs dominate and have a large influence on the optimal solution. By deciding the allocation of tasks first, a search using the AO model very quickly determines the entire set of communication costs which will be incurred. Allocations which incur very large communication costs are likely to be quickly ruled out, and knowledge of all the communication costs can be used in the calculation of f-values throughout the ordering stage. It is likely that ELS performed better on 25% of cases, despite the duplicates, because the pruning techniques used were specifically developed for it and therefore benefited it much more. It may also be the case that the f-value calculations used by ELS provide tighter bounds than those so far developed for AO, in the case of low communication costs.

5 Conclusions

Previous attempts at optimal task scheduling through branch-and-bound methods have used a state-space model which we refer to as exhaustive list scheduling. This state-space model is limited by its high potential for producing duplicate states. In this paper, we have described a new state-space model which approaches the problem of task scheduling with communication delays in two distinct phases: allocation, and ordering. In the allocation phase, we assign each

task to a processor by searching through all possible groupings of tasks. In the ordering phase, with an allocation already decided, we assign a start time to each task by investigating each possible ordering of the tasks on their processors. Using a priority ordering on processors, and thereby fixing the sequence in which independent tasks must be scheduled, we are able to avoid the production of any duplicate states.

An experimental evaluation shows that in roughly 75% of problem instances, the new AO state-space model significantly outperforms the ELS model. This is most evident when scheduling task graphs with high CCRs, most likely because information about communication costs is able to be used earlier and more extensively. The AO state-space model is therefore a promising avenue for optimal task scheduling. Further research is likely to yield improvements to the heuristics used in each phase, as well as specialised pruning techniques which could greatly improve the performance of the model. In addition, a parallel search implementation using this model could have much potential, since the lack of duplicates eliminates the possibility of collisions between parallel searches.

References

1. Bundy, A., Wallen, L.: Branch-and-bound algorithms. In: Bundy, A., Wallen, L. (eds.) Catalogue of Artificial Intelligence Tools, pp. 12–12. Symbolic Computation, Springer Berlin Heidelberg (1984)
2. Drozdowski, M.: Scheduling for Parallel Processing. Springer Publishing Company, Incorporated, 1st edn. (2009)
3. Hagras, T., Janecek, J.: A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. Parallel Computing 31(7), 653–670 (2005)
4. Hwang, J.J., Chow, Y.C., Anger, F.D., Lee, C.Y.: Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. SIAM J. Comput. 18(2), 244–257 (1989)
5. P. E. Hart, N.J.N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems, Science, and Cybernetics SSC-4(2), 100–107 (1968)
6. Ronse, C.: Closures on partial partitions from closures on sets. Mathematica Slovaca 63(5), 959–978 (2013), http://dx.doi.org/10.2478/s12175-013-0147-9
7. Sarkar, V.: Partitioning and scheduling parallel programs for multiprocessors. MIT press (1989)
8. Semar Shahul, A.Z., Sinnen, O.: Scheduling task graphs optimally with A*. Journal of Supercomputing 51(3), 310–332 (Mar 2010)
9. Sinnen, O.: Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing). Wiley-Interscience (2007)
10. Sinnen, O.: Reducing the solution space of optimal task scheduling. Computers & Operations Research 43(0), 201 – 214 (2014), http://www.sciencedirect.com/science/article/pii/S0305054813002542
11. Veltman, B., Lageweg, B.J., Lenstra, J.K.: Multiprocessor Scheduling with Communication Delays 16(2-3), 173–182 (1990)
12. Yang, T., Gerasoulis, A.: List scheduling with and without communication delays. Parallel Computing 19(12), 1321–1344 (1993), http://www.sciencedirect.com/science/article/pii/016781919390079Z