Further Explorations in State-Space Search for Optimal Task Scheduling

Michael Orr & Oliver Sinnen
Department of Electrical and Computer Engineering
The University of Auckland
New Zealand
Email: morr010@aucklanduni.ac.nz

***Abstract*—The problem of task scheduling with communication delays is NP-hard. State-space search algorithms such as A* have been shown to be a promising approach to solving this problem optimally. A recently proposed state-space model for task scheduling, known as Allocation-Ordering (AO), allows state-space search methods to be applied to the problem of optimal task scheduling without the need for duplicate avoidance mechanisms. This paper examines the performance of two parallel search algorithms when applied to both the AO model and the older ELS state-space model. This suggests that its use may provide an advantage with many different variations on state-space search. This paper explores the application of AO to some of these variants, namely depth-first branch-and-bound (DFBnB) and parallel search. We also present an update to the formulation of AO that prevents invalid states from being considered during a search. An evaluation shows that AO gives a clear advantage to DFBnB and allows greater scalability for parallel search algorithms. The update to AO's formulation has no significant impact on performance either way.**

I. Introduction

Efficient schedules are crucial to allowing parallel systems to reach their maximum potential. This work addresses the classic problem of task scheduling with communication delays, known as $P|prec, c_{ij}|C_{\max}$ using the $\alpha|\beta|\gamma$ notation [1]. This problem models a program as a directed acyclic graph of tasks, giving precedence constraints and communication delays, which must be arranged onto a set of processors in order to produce a schedule of minimum length. Solving this problem optimally is NP-hard [2], and therefore there is no currently known method for which the work required to do so does not grow exponentially with the number of tasks. As such, this problem is usually handled with heuristic solutions, of which a large number have been developed. [3], [4], [5], [6]. The quality of these solutions relative to the optimal cannot be guaranteed, however. [7].

The complexity of the problem usually discourages attempts at optimal solving, but possession of an optimal schedule can make an important difference for time-critical applications. Without optimal solutions, it is also very difficult to evaluate the quality of approximation methods. Branch-and-bound algorithms such as A* have shown some promise in finding optimal schedules, with two notable state-space models having been proposed: exhaustive list scheduling [8], and allocation-ordering[9]. AO, the more recent model, has the advantageous property of being duplicate-free. This property suggests that AO should be more appropriate for the use of parallel state-search algorithms, as it removes the need for certain data structures which would otherwise cause contention between threads. It also suggests that AO may give improved performance for search algorithms that do not traditionally have the capacity for duplicate avoidance, such as depth-first branch-and-bound. In this paper, we explore the potential of the AO model when applied to these variants of branch-and-bound, and evaluate its performance in comparison to the ELS model. We also propose a change to the formulation of the AO state-space which prevents invalid states from ever being considered during a search, and therefore avoids performing unnecessary work regarding these states.

Section II provides important background information on the problem, branch-and-bound methods, and the state-space models used for task scheduling. Section III describes a previous limitation of the AO model, and an update to fix it. Section IV describes the application of DFBnB to the AO model, while Section V describes the parallel search algorithms that were used to evaluate the new state-space model. Finally, Section VI presents the conclusions of the paper.

II. Background

A. Task Scheduling Model

The problem addressed in this work is the scheduling of a task graph $G = \{V, E, w, c\}$ on a set of processors P. G is a directed acyclic graph (or DAG). Each node $n \in V$ is a task belonging to the task graph. Tasks represent an indivisible block of work that must be performed by a program represented by G. Each task n_i has a weight $w(n_i)$ which represents the computation time needed to complete that task. An edge $e_{ij} \in E$ represents that task n_j relies on task n_i; data output from n_i is required as an input for n_j, and therefore n_j cannot begin execution until n_i has finished and communicated the necessary data to n_j. Each edge e_{ij} has a weight $c(e_{ij})$ which represents the communication time needed to transmit the necessary data from n_i to n_j. The target system for our schedule consists of a finite number of homogeneous dedicated processors, P. Each pair of processors p_i,$p_j \in P$ possess an identical communication link. All communication can be performed concurrently and without contention. Local communication, from p_i to p_i, has no cost.

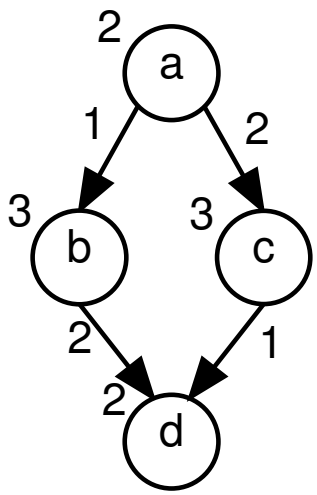


Figure 1: A simple task graph.

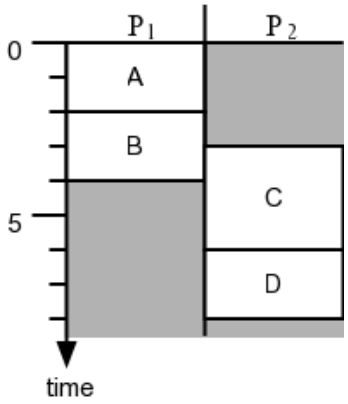


Figure 2: A valid schedule for the simple task graph of Fig. 1.

We aim to produce a schedule $S = \{proc, t_s\}$, where $proc(n)$ gives the processor to which n is assigned, and $t_s(n)$ gives the start time of n. A valid schedule is one for which all tasks in G are assigned a processor and a start time, and which satisfies two conditions for each task. The first condition, known as the processor constraint, requires that each processor is executing at most one task at any given time. The second condition, known as the precedence constraint, requires that a task n may only begin execution once all of its parents have finished execution, and the necessary data has been communicated to $proc(n)$. An optimal schedule is one for which the total execution time is the lowest possible.

B. Branch-and-Bound

Branch-and-bound is a family of search algorithms commonly used to solve combinatorial optimisation problems. Through search, they implicitly enumerate all solutions to a problem, and thereby both find an optimal solution and prove that it is optimal [10]. Each node in the search tree, usually referred to as a state, represents a partial solution to the problem. Given a partial solution represented by a state s, some operation is applied to produce new partial solutions which are closer to a complete solution. Performing this operation to find the children of s is known as branching. Additionally, every state must be *bounded*: a cost function f is used to evaluate each state, where f is defined such that $f(s)$ is a lower bound on the cost of any solution that can be reached from s. These bounds allow the search to be guided away from unpromising solutions, as a single state can be used to judge the entire subtree that proceeds from it.

The most well known variant of branch-and-bound is a best-first search method known as A* [11]. A* has the major advantage that it is optimally efficient; using the same cost function f, no search algorithm could find an optimal solution while examining fewer states (disregarding states which have the same f-value as the optimal). A* relies on the cost function f to always provide an underestimate. This means it is guaranteed that $f(s) \leq f^*(s)$, where $f^*(s)$ is the true lowest cost of any state in the subtree proceeding from s.

Depth-first branch-and-bound (DFBnB) is a variant of branch-and-bound which uses a depth-first search strategy, moving as far into the state-space as possible before backtracking to try other paths. Just as with A*, a cost function f is used to evaluate each state s, producing a lower bound $f(s)$ on the quality of any solution which could be reached from s. When DFBnB first encounters a state s_c representing a complete solution, the cost $f(s_c)$ is recorded as f_{best}. Subsequently, if a state is encountered such that $f(s) \geq f_{best}$, the search will not proceed to that state's children; we have already found a solution at least as good as any that can be reached from this state. If another complete solution s_{c^i} is encountered, and $f(s_{c^i}) < f_{best}$, then $f_{best}$is overwritten with $f(s_{c^i})$. The search ends when no further states can be reached with $f(s) < f_{best}$. At this point, the complete solution with cost f_{best}has been proven to be optimal.

The most obvious advantage of DFBnB when compared with A* is its much lower memory requirements. The best-first nature of A* necessitates the maintenance of a priority queue requiring $O(b^d)$ space (where d refers to the depth of the state-space and b is its average branching factor), while a depth-frst search requires only states on the current path and their children to be in memory at any given time, using $O(bd)$ space. In the case of task scheduling, this means the memory requirement of A* scales exponentially with the number of tasks, while for DFBnB it scales only linearly. This fact suggests another advantage: since the data structures used in depth-first search (usually a stack) tend to be much smaller and simpler, operations performed on them are expected to take less time. This is likely to mean that a depth-first search can process states at a faster rate than a best-frst search.

Naturally, DFBnB also has several disadvantages when compared to A*. The first is that, since it is a depth-first search, it cannot be applied to state-spaces of infinite depth without careful modification. As both AO and ELS have a finite (and relatively shallow) depth, this is not important to our application. Another disadvantage is that, unlike A*, DFBnB is not optimally efficient. Like A*, DFBnB will always examine every state which has $f(s)$ less than the optimal

solution, but it is likely that DFBnB will also examine states with $f(s)$ greater than the optimal solution, which A* will never examine. Indeed, the only case in which DFBnB will not examine extraneous states is if the very first complete solution it encounters is also optimal. This does not mean that DFBnB is guaranteed to examine more states in total than A*; if A* happens to examine a greater proportion of the states where $f(s)$ is equal to the optimal solution, it can still end up doing more work. Such situations, however, are strongly implementation-dependent and unpredictable. It is prudent to assume that, for an arbitrary problem instance, DFBnB is likely to perform more work overall.

C. Exhaustive List Scheduling State-Space

Exhaustive list scheduling is a state-space model for optimal task scheduling which bears a strong resemblance to list scheduling heuristic methods for approximate task scheduling. In this model, each state is a partial schedule. Each task is either scheduled or unscheduled in each state. If a task is scheduled, then it has been assigned to a processor and given a start time. If a task is unscheduled, it may be "free" or not free. A task is free if all of its dependencies have been met; that is, if all of its parents have already been scheduled. At each step, the model branches by placing any free task onto any processor. The full set of children of a state s therefore represent all possible free tasks that could have been chosen, and all possible processors they could have been placed on [8]. In this way, the model simulates all possible decisions that a list scheduling algorithm could make.

Branch-and-bound methods are most effective when the state-space they are searching has the property that all subtrees are disjoint. This means that there is only one path from the root of the tree to any state in the state-space. This is also equivalent to saying that there is only one way of reaching any given state. When this is not the case, we refer to any paths that lead to a state already visited 'duplicates'. Equivalently, any state reached which has already been encountered is termed a 'duplicate' state. If duplicate states are not detected, then the search algorithm can perform a substantial amount of unnecessary work: not only might they examine one duplicate state when an alternate path is found, they may also proceed to re-explore the entire subtree rooted at that state. Detecting duplicate states requires keeping a record of those states which have already been visited, which represents a significant investment of memory.

Unfortunately, the ELS strategy creates a lot of potential for duplicated states [12]. One type which are fundamental to the model are independent decision duplicates. Tasks which are independent of each other can be selected for scheduling in different orders, but be assigned to the same processors in each case. Performing the same scheduling decisions in a different order constitutes taking a different path to reach the same partial schedule, and therefore a duplicate state. These duplicates can only be avoided by enforcing a specific sequence onto these scheduling decisions. There is no known method by which this can be achieved under the ELS model, while still allowing all possible schedules to be reached.

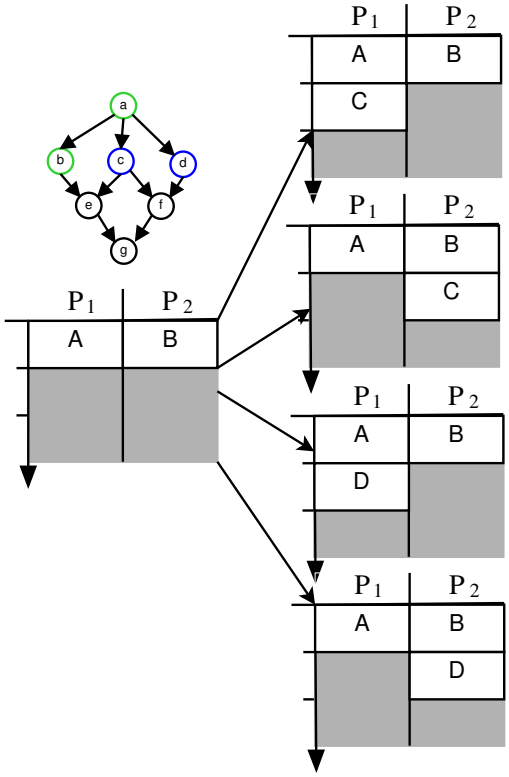


Figure 3: Branching in the ELS state space.

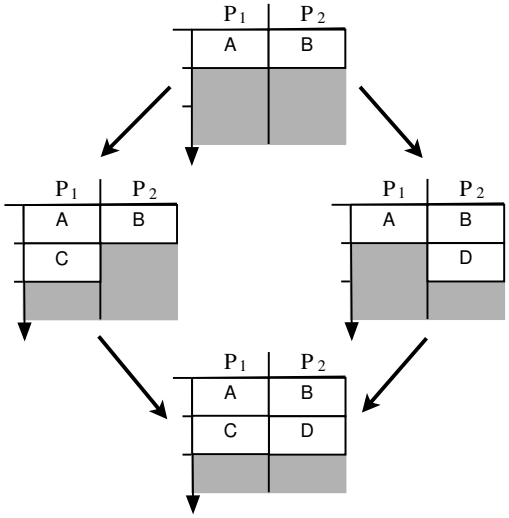


Figure 4: Independent decision duplicates.

D. Allocation-Ordering State-Space

Allocation-Ordering (AO) is a new state-space model [9] constructed such that a specific order is enforced on all scheduling decisions, and therefore the duplicates found in ELS do not exist. The model is named for the two distinct phases in which it creates a schedule: first Allocation, where each task is assigned to a processor, and then Ordering, where

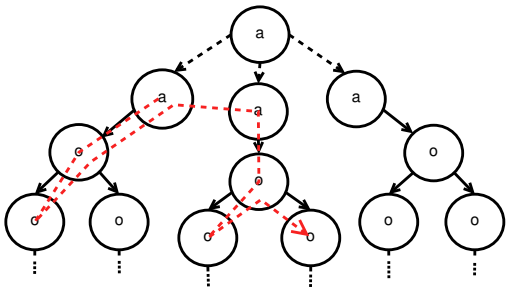


Figure 5: Branching in the AO state-space.

a sequence is decided for the set of tasks allocated to each processor. As it first decides how tasks are grouped together on processors, this model bears a resemblance to the scheduling approximation methods known as clustering, whereas ELS resembles list scheduling.

In the Allocation phase, a partition of the set of tasks is built iteratively, with the maximum number of subsets in the partition being the number of processors available for scheduling. At each step, the current task can either be added to one of the existing subsets, or used to begin a new subset. This process allows all possible groupings of the tasks to be reached, with only one possible path to each grouping.

The Ordering phase begins with a complete allocation. From there, it proceeds in a manner similar to ELS, but on a per-processor basis. For a processor p_i, a task n_i allocated to p_i is considered to be "free" for ordering if there is no task n_j also on p_i which is an ancestor of n_i in the task graph G. At each step, one task is selected from among all those which are currently free on p_i, and placed next in order. This is repeated until all tasks on p_i have been given a place in the sequence. The decision of which processor to order a task on next can be made arbitrarily, but it must be deterministic such that the processor which is selected can be determined entirely by the depth of the current state. Once this process has been completed for all processors, a complete schedule can be derived, simply by giving each task its earliest possible start time given the processor and place in sequence it has been assigned.

By assigning each task to a processor ahead of time, and enforcing a strict order in which the processors are considered, this model eliminates the possibility of independent decision duplicates. In ELS it was possible to place task n_1 on p_1 and then n_2 on p_2, but equally valid to place n_2 on p_2 before placing $n_1$on p_1. AO can force p_1 to be considered before p_2, making only the first sequence of decisions a possibility.

E. Other Optimal Solution Methods

Another combinatorial optimisation technique which has been applied to this task scheduling problem is integer linear programming (ILP). This involves formulating the problem instance as a linear program, a series of simultaneous linear equations, where the variables are constrained to integer values. A number of possible ILP formulations of the $P|prec, c_{ij}|C_{\max}$ problem have been proposed [13], [14], [15], with similarly promising results as branch-and-bound. Neither technique has been shown to have a significant advantage over the other in terms of the size of task scheduling problem that they can solve practically.

III. Avoiding Invalid States in AO

In the AO model, there are some cases in which the combination of valid local orders for all processors produce an overall schedule with an invalid global ordering [9]. To explain why this happens, we model a partial schedule as a graph showing all of the dependencies between tasks. For a task graph G, a partial schedule S'can be represented by augmenting G to produce a partial schedule graph $G_{S'}$. We begin with the graph G. Say that in S', a task n_1 is ordered on processor p_1, and a task n_2 is also ordered on p_1, but later in the sequence. We check for the edge e_{12} in the task graph. If $e_{12} \notin E$, we add $e_{12}$to E. This new edge represents that, according to the ordering defined by our partial schedule, $n_2$must begin after n_1. This can be considered as a new type of dependency, which we call an ordering dependency, as opposed to the original communication dependencies in G. Once edges have been added corresponding to all ordered tasks in S', we have our graph $G_{S'}$. The presence of a cycle in this graph indicates that the ordering is invalid, as a cycle of dependencies is unsatisfiable. Since the graph G is acyclic, and if n_i is an ancestor of n_j in G then the ordering edge e_{ji} cannot be created, it is necessary that any cycle in $G_{S'}$ will contain at least two ordering edges.

Previously, such states were removed from consideration during the search as their cyclic nature made their f-values increase infinitely during calculation, until they passed an upper bound for the schedule length and it was clear they could be ignored. However, it is possible for a state to exist in which no cycle yet exists, but for which it is inevitable that a cycle will be created as the ordering process continues. Say, for example, that the introduction of edge e_{ij} would create a cycle in $G_{S'}$, but in S' the task n_i has already been ordered while n_j has not. In order for the schedule to be completed, n_j must eventually be ordered, at which point a cycle will be formed. Here, S'represents an entire subtree of states from which no valid schedule can be reached. None of these states can be selected as the optimal solution, so this does not present a threat to the accuracy of the search process. However, it does represent a potentially substantial amount of wasted work performed by the search algorithm. Ideally the formulation of the AO model would be such that it allows the creation of any valid solution, and *only* valid solutions.

The key to avoiding this unnecessary work is the observation that, given that n_i has been ordered and n_j has not, it is inevitable that $n_j$must eventually be ordered later than n_i. Therefore, for all descendents of this partial schedule in which n_j is ordered, the ordering edge e_{ij} must be in $G_{S'}$. We can therefore define a more useful augmented task graph, $G^*_{S'}$, which 'looks ahead' to determine cycles that must inevitably occur. In this graph, the ordering edge e_{ij} exists if

$proc(n_i) = proc(n_j)$ and either n_j is ordered later than n_i, or n_i has been ordered and n_j has not.

To avoid these cycles, we propose a modification to the condition we use to determine if a task is free to be ordered. The new condition is this: a task n_i on processor p_i is free to be ordered if it has no ancestors in graph $G^*_{S'}$ which are also on p_i and have not already been ordered in S'. In the original formulation of AO, this condition used only the graph G. However, the ordering edges specific to the partial solution S' must be considered equally with the communication edges that are common to all partial solutions. The creation of a cycle in $G^*_{S'}$ requires that an ordering edge is introduced from a task n_i to task n_j, where n_i was already reachable from n_j using at least one ordering edge. By definition, this means that n_j is the ancestor of n_i in $G^*_{S'}$. Therefore, according to the new condition, n_i cannot be considered free until n_j has been ordered, meaning that the edge e_{ij} can never be introduced and the cycle can never be formed. By treating the ordering dependencies created during the ordering process in the same way as the original communication dependencies, we ensure that states with an invalid global ordering cannot be reached.

We implemented this change by maintaining a record of $G^*_{S'}$with each state in the form of a transitive closure matrix. Whenever a new task is ordered, the transitive closure is updated to reflect the new ordering dependencies. We can then use this matrix to determine which tasks are free when creating the children of a state.

A. Evaluation

To determine empirically the impact of this change to the formulation of the AO model, we performed A* searches on a set of task graphs using versions of the model both with and without this change. Task graphs were chosen corresponding to a wide variety of program structures. Approximately 270 graphs with 21 tasks were selected. These graphs were a mix of the following DAG structure types: Independent, Fork, Join, Fork-Join, Out-Tree, In-Tree, Pipeline, Random, Series-Parallel, and Stencil. We attempted to find an optimal schedule using both 2 and 4 processors, once each for both versions of AO, giving a total of over 1000 trials. The algorithms were implemented in the Java programming language. All tests were run on a Linux machine with 4 Intel Xeon E7-4830 v3 @2.1GHz processors. The tests were single-threaded, so they would only have gained marginal benefit from the multi-core system. The tests were allowed a time limit of 10 minutes to complete. Each search was started in a new JVM instance, to remove the possibility of previous searches impacting them through garbage collection and JIT compilation.

Figure 6 shows the results of these tests. Both versions of AO were able to solve approximately 80% of the problem instances within 10 minutes, without a significant difference in performance. This suggests that the presence of invalid states in the original formulation did not have much of a negative impact on average. It also suggests, however, that the addition of the transitive closure and associated operations did not significantly slow down the implementation of the new AO formulation.

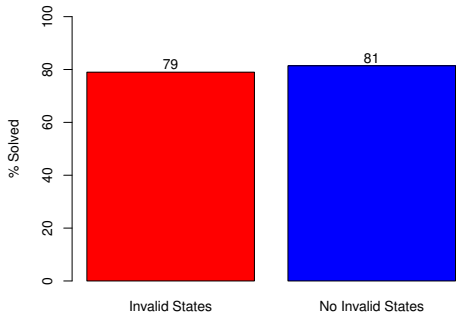


Figure 6: Comparing the performance of AO with and without invalid states.

IV. Depth-First Branch and Bound

When applying DFBnB to a state-space containing duplicate states, there are two possible approaches: ignore the duplicates, or implement a duplicate-detection mechanism. If we ignore duplicate states, we are likely to greatly increase the amount of work necessary to find the optimal solution. A depth-first search will examine every possible path in the state-space: this could mean that entire sub-trees are explored many times over. On the other hand, the addition of a duplicate-detection mechanism will largely negate the main advantage of DFBnB over A*, that being its much lower memory requirement. In order to avoid repeating work, the search algorithm must keep a record of states it has already examined. Although many strategies could exist for deciding exactly which states should be remembered, any strategy that is maximally effective at detecting duplicates will require $O(b^d)$ memory, just as A* does. With such an implementation of DFBnB requiring an exponentially growing amount of memory, and not being optimally efficient, it is hard to imagine a situation in which it would be preferable to A*.

For those reasons, it seems likely that DFBnB would perform significantly better on AO, a duplicate-free state-space, than it would on ELS, a state-space with duplicates. If this is true, then the use of AO could make DFBnB a more practical option for optimal task scheduling, making its benefits available for situations where memory is the more constraining factor.

A. Evaluation

To empirically evaluate the hypothesis that AO would allow better performance from depth-first branch-and-bound, we performed DFBnB searches on a set of task graphs using each state-space model. Task graphs were chosen corresponding to a wide variety of program structures. A total of 71 graphs with 21 tasks were selected. We attempted to find an optimal schedule using both 2 and 4 processors, once each for both

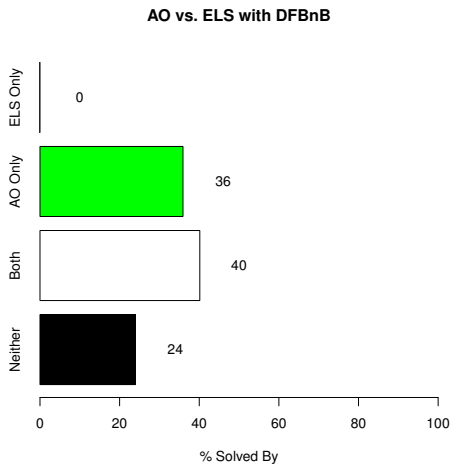


Figure 7: Comparing the performance of DFBnB using AO and ELS.

state-space models, giving a total of 142 trials per model. The algorithms were implemented in the Java programming language. Existing implementations of both ELS and AO were utilised. All tests were run on a Linux machine with 4 Intel Xeon E7-4830 v3 @2.1GHz processors. The tests were single-threaded, so they would only have gained marginal benefit from the multi-core system. The tests were allowed a time limit of 10 minutes to complete. Each search was started in a new JVM instance, to remove the possibility of previous searches impacting them through garbage collection and JIT compilation.

Figure 7 shows the results of these tests. DFBnB was able to solve 40% of the problem instances within 10 minutes for both models, and 24% of problem instances could not be solved within ten minutes using either model. The remaining 36% of problem instances were able to be solved only when using the AO model. There were no instances which were only solved using the ELS model. This clearly demonstrates that DFBnB gained an advantage when using the AO model - in fact, almost twice as many optimal schedules were able to be found when using AO than when using ELS.

V. Parallel Search

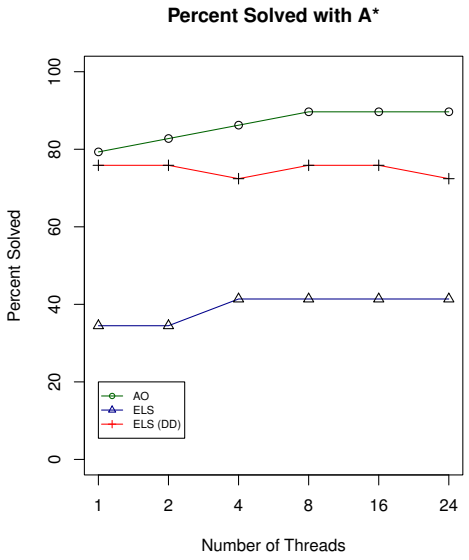
As the AO model is duplicate-free, it does not require the use of a duplicate-detection mechanism, or any of the data structures associated with one. In a parallelised implementation of branch-and-bound, these data structures require synchronisation between workers, adding greatly to the potential for contention and likely limiting overall speedup. Therefore, without the need for duplicate detection, it seems probable that parallel branch-and-bound could be more effective when used with the AO model than with the ELS model.

We have implemented simple shared-memory parallel versions of both A* and DFBnB. In both cases, each worker thread possesses its own data structure from which states are retrieved to be examined, and to which child states are added. In the case of A*, this is a heap-based priority queue, while for DFBnB it is a linked-list-based deque. The A* implementation also has the option of using a closed list, shared between all the workers for the purposes of duplicate detection. The data structure used is Java's ConcurrentHashMap, a thread-safe hash map designed for high concurrency. In each algorithm, workers will only examine states with a lower f-value than their currently best known solution. After every hundred thousand states, they will check to see if any other worker has discovered a better solution. With depth-first search, solutions are expected to be discovered very often early on, and less so as the optimal is approached. With A*'s best-first approach, the opposite is expected, with no solutions at all found until near the end of the search. For both algorithms, the search is finished once all workers have no states remaining with f-values lower than the current best-known solution. This solution has then been proven to be optimal.

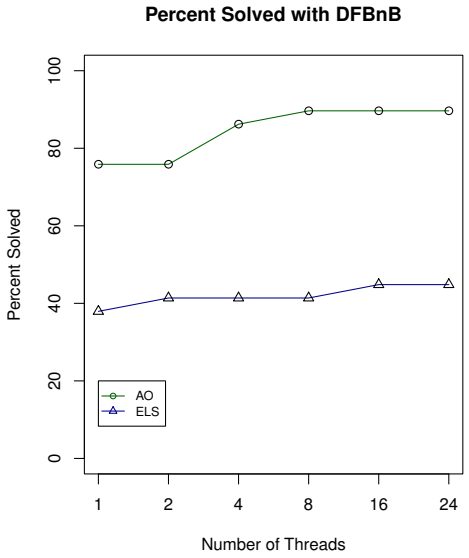
Both algorithms use work-stealing to aid with load-balancing: when a worker has exhausted all potentially useful work to be done from its own data structure, it visits another worker's data structure and takes a state from it. For DFBnB, a state is stolen from the back end of the deque. This both minimises the chance of contention between the stealer and the victim, and maximises the total amount of work stolen - since states at the tail of the deque are highest up in the state-space, they lead to the largest subtrees. This will hopefully ensure that the stealer does not have to steal again soon after. For A*, it is the current best state in the victim's priority queue that is taken - meaning that it is the most likely state present to lead to the optimal solution, and therefore most likely to represent useful work. By contrast, stealing from the back of the queue could yield a very low quality state which, if it is useful to examine it at all, is relatively unlikely to lead to a significantly sized worthwhile subtree. For both approaches, the victims of stealing are selected randomly. Compared to some deterministic methods, this lowers the chance that a worker will be serially victimised and slowed down by others, and requires little communication between threads for a decision to be made.

A. Evaluation

To empirically evaluate the hypothesis that AO would allow better performance for parallel search algorithms, we performed parallel searches on a set of task graphs using the parallel implementations of A* and DFBnB. Task graphs were chosen corresponding to a wide variety of program structures. A total of 30 graphs with 21 tasks were selected, with each of these graphs being known to take at least 5 minutes to solve by a serial A* algorithm. This decision was made so that there would be significant potential for reduction in time taken by the parallel versions. These graphs included Fork, Out-Tree, Pipeline, Random, Series-Parallel and Stencil structures. We attempted to find an optimal schedule using 4 processors, with both state-space models, with 1, 2, 4, 8, 16 and 24 worker threads, once each for each parallel algorithm. This gave a total of 696 trials. The algorithms were implemented



(a) A*



(b) DFBnB

Figure 8: Comparing the performance of parallel algorithms using AO and ELS.

in the Java programming language. Existing implementations of both ELS and AO were utilised. All tests were run on a Linux machine with 4 Intel Xeon E7-4830 v3 @2.1GHz processors. This system has a total of 48 cores. The processes were restricted to a subset of cores, depending on the number of worker threads, such that the threads were concentrated on as few of the 4 processors as possible. Initial tests showed this to be more effective than spreading threads evenly among the processors, presumably as the locality allowed them to share a cache. The tests were allowed a time limit of 10 minutes to complete. Each search was started in a new JVM instance, to remove the possibility of previous searches impacting them through garbage collection and JIT compilation.

Figure 8 shows the results of these tests. In figure 8a, we see that parallel A* achieved greater scalability when using the AO model than with ELS. Parallel A* allows AO to solve an additional 10% of problem instances, scaling up to 8 threads. ELS achieves similar improvement when duplicate detection is not used; however, its absolute performance is much worse. When duplicate detection is used, the absolute performance of ELS is more comparable to that of AO, but no benefit can be observed from the parallelisation. Figure 8b shows that ELS benefits only marginally from parallel DFBnB, while AO demonstrates similar improvements, and absolute performance, as observed with parallel A*. The absolute performance of ELS is clearly inferior to that of AO when using this duplicate-ignoring solution method.

VI. Conclusions

For optimal task scheduling using branch-and-bound, there are two notable state-space models which can be used. The first, exhaustive list scheduling, is limited by its high potential for producing duplicate states. This means this state-space requires either significantly more memory, if we choose to avoid duplicates, or significantly more time, if we choose not to. The second model, allocation-ordering, does not contain duplicates. We considered that this might be advantageous to a wide variety of branch-and-bound methods. We also presented an update to the formulation of AO which removed a potentially large number of invalid states from the state-space.

An empirical evaluation showed that removing invalid states from the AO state-space did not make a significant impact on its average performance. This suggests that searches using the older version of AO were probably not encountering a significant number of invalid states. On the other hand, this evaluation showed that the implementation tweaks necessary to avoid invalid states did not add sufficient complexity to the algorithm to outweigh the time saved by avoiding them. Overall, the new formulation of the AO model is more theoretically complete, without negative effects on the performance of the model.

Our evaluation showed that AO was greatly superior to ELS when used with DFBnB, solving nearly twice as many problem instances within the time limit. We also saw that AO allowed greater improvements in the number of problems solved with parallel versions of A* and DFBnB.

References

[1] B. Veltman, B. J. Lageweg, and J. K. Lenstra, "Multiprocessor Scheduling with Communication Delays," *Parallel Computing*, vol. 16, no. 2-3, pp. 173–182, 1990.

[2] V. Sarkar, *Partitioning and scheduling parallel programs for multiprocessors*. MIT press, 1989.

[3] T. Hagras and J. Janecek, "A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems," *Parallel Computing*, vol. 31, no. 7, pp. 653–670, 2005.

[4] T. Yang and A. Gerasoulis, "List scheduling with and without communication delays," *Parallel Computing*, vol. 19, no. 12, pp. 1321–1344, 1993.

[5] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM J. Comput.*, vol. 18, no. 2, pp. 244–257, 1989.
[6] O. Sinnen, *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007.
[7] M. Drozdowski, *Scheduling for Parallel Processing*. Springer Publishing Company, Incorporated, 1st ed., 2009.
[8] A. Z. Semar Shahul and O. Sinnen, "Scheduling task graphs optimally with A*," *Journal of Supercomputing*, vol. 51, pp. 310–332, Mar. 2010.
[9] M. Orr and O. Sinnen, "A duplicate-free state-space model for optimal task scheduling," in *Proc. of 21st Int. European Conference on Parallel and Distributed Computing (Euro-Par 2015)*, vol. 9233 of *Lecture Notes in Computer Science*, (Vienna, Austria), Springer, 2015.
[10] A. Bundy and L. Wallen, "Branch-and-bound algorithms," in *Catalogue of Artificial Intelligence Tools* (A. Bundy and L. Wallen, eds.), Symbolic Computation, pp. 12–12, Springer Berlin Heidelberg, 1984.
[11] N. J. N. P. E. Hart and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems, Science, and Cybernetics*, vol. SSC-4, no. 2, pp. 100–107, 1968.
[12] O. Sinnen, "Reducing the solution space of optimal task scheduling," *Computers & Operations Research*, vol. 43, no. 0, pp. 201 – 214, 2014.
[13] A. A. El Cadi, R. B. Atitallah, S. d. Hanafi, N. Mladenović, and A. Artiba, "New mip model for multiprocessor scheduling problem with communication delays," *Optimization Letters*, pp. 1–17, 2014.
[14] S. Venugopalan and O. Sinnen, "Optimal linear programming solutions for multiprocessor scheduling with communication delays," in *Proc. of 12th Int. Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2012)*, vol. 7439 of *Lecture Notes in Computer Science*, (Fukuoka, Japan), pp. 129–138, Springer, Sept. 2012.
[15] S. Mallach, "Improved mixed-integer programming models for multiprocessor scheduling with communication delays," 2016.