# Integrating Task Duplication in Optimal Task Scheduling with Communication Delays

Michael Orr and Oliver Sinnen

**Abstract**—Task scheduling with communication delays is an NP-hard problem. Some previous attempts at finding optimal solutions to this problem have used branch-and-bound state-space search, with promising results. Duplication is an extension to the task scheduling model which allows tasks to be executed multiple times within a schedule, providing benefits to schedule length where this allows a reduction in communication costs. This paper proposes the first approach to state-space search for optimal task scheduling with task duplication.Also presented are new definitions for important standard bounding metrics in the context of duplication. An extensive empirical evaluation shows that the use of duplication significantly increases the difficulty of optimal scheduling, but the proposed approach also gives certainty that a large proportion of task graphs can be scheduled more effectively when duplication is allowed, and permits to quantify the exact advantage.

**Index Terms**—Scheduling; Parallel systems; Graph and tree search strategies; Optimization

## 1 Introduction

Efficient schedules are required in order to maximise the potential of parallel systems to improve the execution time of programs. The classic problem of task scheduling with communication delays, known as $P|prec, c_{ij}|C_{\max}$ using the $\alpha|\beta|\gamma$ notation[25] (see Section 2.1), involves a set of tasks, along with precedence constraints and communication costs, which must be scheduled on a set of processors with the goal of minimising the overall execution time. This problem is NP-hard, which means that no polynomial time algorithm is known which can solve it optimally [15]. For that reason, task scheduling problems are usually solved with approximation algorithms, giving non-optimal but hopefully good solutions [7], [9]. Unfortunately, there is no way to guarantee the quality of these approximate solutions relative to the optimal, as no $\alpha$-approximation scheme is known for the problem[4]. This means that it is necessary to be able to find optimal solutions in order to fully evaluate the performance of approximation algorithms. Branch-and-bound state-space search has been used for optimally solving the problem with homogeneous processors, and shown some promise[16]. In particular, a state-space model known as Allocation-Ordering (AO) has demonstrated to be most effective with this method [12]. The AO model avoids the duplication of *states, i.e. partial schedules*, not to be mistaken with the duplication of *tasks*, addressed in this paper.

Duplication of tasks is an extension to the basic task scheduling problem which allows tasks to be executed multiple times within a schedule, with different copies of a task assigned to different processors. The resulting problem can be referred to as $P|prec, c_{ij}, dup|C_{\max}$. While it might not seem intuitive that a schedule can be improved by performing the same work multiple times, duplication can provide a benefit through the reduction of communication costs. Often, the additional computation time needed to re-execute a task on a different processor is less than the time that would be needed to communicate the task's output data to its children on that processor. Allowing duplication to occur can therefore significantly improve schedule lengths. Duplication is often incorporated into approximation methods, or introduced in a pre- or post-processing step with heuristic methods. In this work we aim to incorporate duplication fully with an optimal solving method, allowing provably optimal schedules with duplication to be found.

In this work, changes to branching procedures for both phases of the AO model are proposed to allow an optimal schedule to be found among all valid schedules with or without duplicated tasks. We propose significantly different definitions for allocated top and bottom levels in the context of task duplication, and use these to propose admissable lower bound heuristics for the modified AO model. An experimental evaluation shows the impact of task duplication on the difficulty of optimal solving, demonstrating that it makes the problem significantly more difficult. It is also shown how the exact benefit of task duplication for schedule lengths can be quantified through optimal solving with this model. With this method, we see that a large number of problem instances receive some benefit, particularly those with certain graph structures or with high communication-to-computation ratios.

In Sections 2 and 3 we discuss relevant background information and related work, including the task scheduling model used and the original formulation of the AO model. Section 4 discusses how the AO model was reformulated to allow duplication, and the additional complexity this introduces to the state-space. Section 5 discusses necessary changes to the way lower bounds are calculated when duplication is introduced. Subsequently, Section 6 presents an empirical evaluation of the performance of the reformulated AO model when duplication is allowed, and which task graphs benefit from duplication when optimally scheduled. Finally, Section 7 presents the conclusions of the paper.

- *M. Orr and O. Sinnen are with the Department of Electrical and Computer Engineering, University of Auckland, New Zealand.*

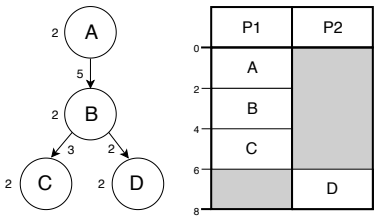


Figure 1. A simple task graph and valid schedule.

## 2 BACKGROUND

### 2.1 Task Scheduling Model

The problem addressed by this work is the classic problem of task scheduling with communication delays, known as $P|prec, c_{ij}|C_{\max}$ using the $\alpha|\beta|\gamma$ notation for scheduling ($\alpha$ denotes the processor environment, $\beta$ denotes properties of the tasks, and $\gamma$ denotes the objective function). $P$ means that the processors are homogeneous. $prec, c_{ij}$ means the tasks have precedence constraints with associated communication costs. $C_{max}$ means the objective is to minimise the maximum completion time among all tasks, i.e. the schedule length. The problem is defined as creating a schedule $S$ for a task graph $G$ on a set of processors $P$.

**Definition 2.1.** Task Graph

A task graph $G = \{V, E, w, c\}$ is a directed acyclic graph (DAG) representing a program to be executed. Nodes $n \in V$ represent specific tasks that need to be completed by the program, with weight $w(n)$ (also called computation cost) indicating a number of time units needed to complete a task. An edge $e_{ij} \in E$ indicates that task $n_j$ depends on task $n_i$; in other words, task $n_i$ must be completed before task $n_j$ can begin execution - data produced as an output of task $n_i$ is necessary as an input of $n_j$. The edge weight $c(e)$ (also called communication cost) represents a number of time units needed to communicate the required data between processors, if necessary.

An example of a task graph can be seen in Figure 1. The set of parents (or predecessors) of task $n$ is denoted by $pred(n)$, while the set of children (or successors) is denoted by $succ(n)$. With $deg^-(n)$ and $deg^+(n)$ we denote the in-degree and out-degree, respectively, of a task $n$, which is the number of incoming and outgoing edges. In this model, we assume that the processors $p \in P$ are fully connected by a homogeneous communication subsystem. Data is transmitted between any pair of processors $p_i, p_j \in P$ uniformly and without contention, and the computational work of the processors is not affected. The specific details of the communication links between processors are not considered beyond the assumption that they result in these properties. Both computation and communication costs are considered to be fixed and unchanging for the purposes of this model. The exception is that local communication, from $p_i$to $p_i$, has zero cost - if a parent and child task are executed by the same processor, no communication is necessary.

**Definition 2.2.** Schedule

A schedule $S = \{proc, t_s\}$ is a plan for execution of a task graph $G$ on a set of processors $P$. $proc(n)$ maps a task $n \in V$ to a processor $p \in P$, signifying that $n$ is to be executed by $p$. $t_s(n)$ gives a start time for task $n$, being the number of time units after the start of the program's execution at which this $n$ should begin to be executed by $proc(n)$. A valid schedule must define $proc(n)$ and $t_s(n)$ for all $n \in V$, such that two constraining conditions are met. The processor constraint mandates that only one task may be executed by a processor at any given time. The precedence constraint requires that a task $n$ may only begin execution once all of its predecessors have completed execution, and all of the required data produced by those predecessors has been communicated to $proc(n)$. A schedule's length (also sometimes called its *makespan*) is the time taken for the full schedule to be executed: this is equal to the largest of $t_s(n) + w(n)$ among all $n \in V$.

In optimal task scheduling, our goal is to find a schedule which has the lowest possible total execution time, denoted as $S^*$. Figure 1 shows a valid optimal schedule for the simple task graph.It is also useful to introduce the concept of node levels. Given a task $n$, the top level $tl(n)$ is usually defined as the length of the longest path in the task graph ending with $n$. Here, length means the sum of the weights of the tasks included in the path. It excludes the weight of $n$ itself, and does not include the weights of any edges. The top level for a task $n$ is intended to define the absolute minimum value that could be given to $t_s(n)$ in a valid schedule. This is why communication costs are not included: a schedule can always be constructed in which all tasks on the path are placed on the same processor, and therefore all relevant communications have zero cost. Similarly, the bottom level $bl(n)$ is the length of the longest path in the task graph beginning with $n$. This value includes the weight of $n$, but still excludes communication costs. Another useful value is the data-ready time $drt(n, p)$. This is the time at which all data from the parents of task $n$ would finish communication to processor $p$. In other words, it is the earliest possible time that task $n$ could begin execution on processor $p$. This value is only defined when all parents of $n$ have been allocated and given start times.

### 2.2 Duplication

Duplication is an extension to the basic task scheduling problem which allows tasks to be executed multiple times within a schedule, with different copies of a task assigned to different processors [1]. The function $proc$ is now defined such that $proc(n)$ maps to some subset of $P$, rather than a single member $p \in P$. Each task can be allocated to any number of processors, but only once per processor. The instance of task $n_i$ allocated to processor $p_j$ can be denoted as task $n_i^j$. A child $n_k$ of $n_i$, where $n_i$ is duplicated, can have its necessary input data provided by any one of the duplicates $n_i^j$. We say that $n_k$ is enabled by $n_i^j$. The child $n_k$ may also be duplicated, in which case each of its instances $n_k^l$ must of course be enabled by some $n_i^j$. We denote the set of all instances of task $n_i$ with $dups(n_i)$.

While it is valid to duplicate any task, there are limited circumstances in which doing so will allow a schedule to be improved. To demonstrate the general case in which it is beneficial, say that we have a task $n_i$ allocated on processor $p_i$, and one of its children $n_j$ allocated on $p_j$. The start time of $n_j$ will be at least $drt(n_i, p_i) + w(n_i) + c(e_{ij})$ - it may be

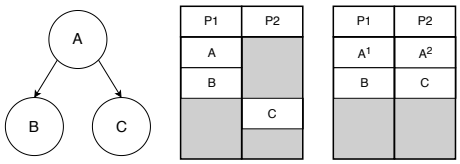


Figure 2. A basic example of beneficial duplication.

later due to communication from another parent of $n_j$, or if $n_j$ is delayed by other tasks on $p_j$. If $n_i$was duplicated, with $n_i^1$ on $p_i$ and $n_i^2$ on $p_j$, it would be possible for $n_i^2$ to enable $n_j$ starting from time $drt(n_i, p_j) + w(n_i)$. The elimination of the cost $c(e_{ij})$ means that $n_j$ may be able to begin execution earlier, which in turn may allow the total schedule length to be decreased. Figure 2 demonstrates how a simple fork-type task graph can be scheduled with duplication in a beneficial way. The source task $A$ can start at time zero on every processor, and therefore duplication allows all communication costs for its children to be avoided. This is a case that applies to any task graph with just one source task.

*2.2.1*

### 2.3 Branch-and-Bound

Branch-and-bound is the name of a family of state-space search algorithms which are widely used to solve combinatorial optimisation problems. These algorithms use search to implicitly enumerate all possible solutions to the problem, thereby both finding an optimal solution and proving that it is optimal [3]. They differ from a brute force, exhaustive search approach in that bounds are used to remove large subsets of similar solutions from consideration. A search tree is constructed in which the nodes, usually referred to as states, represent partial solutions to the problem. A set of operations is defined which transforms a given partial solution $s$ into a number of new partial solutions which are closer to a complete solution. This process of defining the child states of $s$ is known as branching. The rules for branching, along with an initial state, define the state-space to be searched. Upon discovery, each state $s$ will be bounded. This means it will be evaluated using a cost-function $f$, which gives a lower bound on the cost of any complete solution which could be reached from $s$. The bound given by $f(s)$ is usually known as the state's $f$-value. These bounds allow many states to be ignored by the search, by proving that they cannot lead to better solutions than those found elsewhere.

### 2.4 Allocation-Ordering Model

Allocation-Ordering is a state-space model for task scheduling which contains two distinct phases within its search tree: first allocation, and then ordering . In the allocation phase, each task is assigned to a processor[12]. Once a complete allocation has been found, the ordering phase begins, and the tasks allocated to each processor are arranged into a specific sequence. With each task allocated to a processor, and the order of the tasks on each processor decided, a complete schedule can be uniquely derived. This simply involves placing each task onto its assigned processor at the earliest start time allowed by its ordering. It is important to understand that the allocation and ordering phases do not represent separate search processes; they belong seamlessly to a single state-space, and search algorithms may move back and forth between them as needed. The AO model has been shown to allow superior performance when compared with an earlier state-space model, due to its lack of duplicate states.

In the allocation phase, states represent a partial allocation of tasks to processors. A complete allocation is represented by a partition of the set of tasks, $V$. A partition of a set $X$ is a set of mutually exclusive subsets (or "parts"), the union of which is equal to the original set $X$. Each allocation state, therefore, is a "partial partition" of $V$. We start with a list of the tasks $n \in V$, arranged in some topological order. At each step, we take the next task $n_i$ from the head of the list and add it to our partial partition, inserting it into a part. This means that we will either insert it into one of the existing parts, or use it to begin a new part. The full range of options here gives the set of children of each state. The number of parts allowed in a partition is limited to the number of processors in $P$. This process allows all possible groupings of tasks to be considered.

For a state in the allocation phase representing a partial partition $A$, bounding is performed using two different metrics. The first is the maximum total computational load among any of the parts $a \in A$. The second is the critical path through the task graph given the known allocations. This latter metric uses the concept of allocated levels for tasks: the allocated top level, $tl_A(n)$, and the allocated bottom level, $bl_A(n)$. These closely resemble the normal top and bottom level concepts, but incorporate known information about the allocation of tasks to processors. For homogeneous processors, this means adding communication costs that we now know must be incurred to the lengths of the paths. If $proc(n_i) \neq proc(n_j)$, then the communication required by edge $e_{ij}$ must now occur, and so the weight of the edge is included in any relevant allocated node levels. The two cost functions for the allocation phase are as follows, with the final $f$-value being the maximum of the two for a given state.

$$f_{\text{load}}(s) = \max_{a \in A} \left\{ \sum_{n \in a} w(n) \right\} \tag{1}$$

$$f_{\text{acp}}(s) = \max_{n \in V} \{ tl_{\text{A}}(n) + bl_A(n) \} \tag{2}$$

For states which represent a complete allocation, their sole child is the beginning of a new ordering phase. With a complete partition $A$, it is trivial to map each part $a \in A$ to a processor $p \in P$, such that for all $n \in V$ we can now define $proc(n)$. Given that allocation, things proceed in a manner similar to a list scheduling algorithm, but on a per-processor basis. For each processor, a "ready list" is maintained of tasks which have had all their dependencies satisfied. At each step, a single task $n$ is chosen from among those in the ready list of a processor and placed next in sequence on that processor. The task $n$ is then considered to be "ordered". A task $n_i$ allocated to $p_i$ is considered to not be ready if there is an unordered task $n_j$ also allocated to $p_i$ which is an ancestor of $n_i$ in the graph $G$. Otherwise, $n_i$ is ready. The decision of which processor to order a task on next is essentially arbitrary, but must be made according to some

deterministic scheme such that the processor selected can be determined while knowing nothing about the new state but its depth. A simple round-robin method is used in our implementation. This process continues until all tasks have been ordered, at which point a complete schedule has been constructed.

Finding lower bounds in the ordering phase is somewhat more complicated. It relies heavily on the concept of an estimated earliest start time for each task, $eest(n)$, this being the minimum value which $t_s(n)$ could eventually take given the allocation and ordering decisions made so far. For any task which is unordered, $eest(n) = tl_A(n)$, its allocated top level. For ordered tasks, we first define $prev(n)$ as the task which is ordered directly before $n$ on the processor $proc(n)$. We also define the estimated data ready time $edrt(n_j) = max_{n_i \in pred(n_j)} \{eest(n_i) + w(n_i) + c(e_{ij})\}$. If $prev(n)$ does not exist, $eest(n) = edrt(n)$. If it does exist, $eest(n) = max(eest(prev(n)) + w(prev(n)), edrt(n))$. Again, we use two metrics for bounding in this phase. The first is known as the partially scheduled critical path, being the maximum among all tasks of their estimated earliest start time plus their allocated bottom level. The second is the latest estimated finish time of any processor plus the total computational weight of the tasks not yet ordered on that processor.

$$f_{\mathrm{scp}}(s) = \max_{n \in V} \{eest(n) + bl_{\mathrm{a}}(n)\} \tag{3}$$

$$f_{\mathrm{ordered-load}}(s) = \max_{p \in P} \left\{ t_{\mathrm{f}}(p) + \sum_{n \in p \cap unordered(s)} w(n) \right\} \tag{4}$$

## 3 Related Work

Duplication is often incorporated in heuristic algorithms for task scheduling [7], [9]. This includes algorithms based on list-scheduling [22], [8] and clustering[6], [14] approaches, as well as methods such as genetic algorithms [24].The general approach to duplication in list scheduling is to attempt duplication of ancestor tasks when a new task is scheduled. Some set of possible ancestor tasks will be enumerated and duplicated on the processor of the newly scheduled task, and the duplicates will then be removed if they do not improve the new task's start time [1]. Some algorithms give no regard to redundancies in duplication, while others attempt to minimise duplicated tasks [17].In clustering algorithms, duplication can be introduced by allowing clusters to overlap; that is, if adding a particular task is determined to be beneficial to a cluster, that task may be included even if it already belongs to a different cluster [6]. In general, the use of duplication allows these algorithms to produce schedules of a lower makespan, at the cost of some increase in computational complexity. Some of these algorithms produce optimal schedules when applied to a subset of task graphs meeting certain conditions [13], [6].

An approach to optimal task scheduling which is substantially different from branch-and-bound is the use of integer linear programming (ILP). This involves formulating the problem as a linear program, and attempting the best possible solution among those where the variable are constrained to integer values. This is also an NP-hard combinatorial optimisation method, and in fact ILP solvers usually use highly optimised branch-and-bound search as part of the solution process. Several ILP formulations of the $P|prec, c_{ij}|C_{\max}$ task scheduling problem have been proposed[5], [10]. While they have shown similarly promising results as the pure branch-and-bound approach, neither method has been shown to have significantly better performance than the other.

Some ILP formulations with duplication have been proposed [18], [23], [19]. In [23], duplication of tasks was used in order to increase the reliability of schedules, but its use in reducing schedule lengths was not considered. In [2], duplication for the purpose of schedule length reduction is performed. All tasks were considered for duplication on all processors, making the complexity very high. This was built on by [19] with a restricted duplication formulation (RESDMILP). This approach restricts the number of copies of each task according to a given parameter, reducing the complexity of the ILP but sacrificing the optimality of the solution. Only small task graphs were able to be solved optimally with duplication.[18].

## 4 AO with Duplication

In this section, we propose changes to the AO state-space model which allow the representation of schedules with duplicated tasks. This allows branch-and-bound search to find the optimal solution to a task scheduling problem in which duplication is allowed. This is achieved primarily through modification of the allocation phase, as this is when we decide which tasks are executed by which processors. Some smaller changes are required in the ordering phase, and the definitions of some properties used for bounding are changed throughout the algorithm.

### 4.1 Allocation

The purpose of the allocation phase of AO is to determine which tasks will be executed by which processor. More precisely, it is to produce a number of subsets of the set of tasks $V$, such that the members of each will all be executed by a specific processor in the final schedule. The possible outputs of the original allocation phase are the set of all partitions of $V$, ensuring that each task ends up on only one processor. When duplication is allowed, a naive formulation of the allocation phase might have an output space in which each processor could be assigned any subset of $V$, so long as the union of all subsets was equal to $V$. This would be an enormous increase in the size of the state space, and most of the possible duplications of tasks under this scheme would have no chance of improving the schedule length. Table 1 gives some examples which demonstrate the difference in the number of possible allocations considered between the AO model without duplication, this naive method of duplication, and the method we will propose. The notation $\left\{ {|V| \atop k} \right\}$ indicates the number of possible partitions of $V$ of size $k$ (Sterling number of the second kind) [12]. The notation $duplicable(V)$ indicates the set of tasks for which duplication is not proven to be non-beneficial (see Section 4.1.1).

| Problem Instance | Number of Procs | No Duplication | Naive Duplication | Proposed (Upper Bound) |
|---|---|---|---|---|
| Figure 2 | 2 | 4 | 27 | 8 |
| Figure 4 | 2 | 8 | 81 | 32 |
| Figure 5 | 3 | 3,281 | 40,353,607 | 209,984 |
| General | \|P\| | $\sum_{k=1}^{\|P\|} \left\{ {\|V\| \atop k} \right\}$ | $(2^{\|P\|}-1)^{\|V\|}$ | $((2^{\|P\|-1})^{\|duplicable(V)\|}) \left( \sum_{k=1}^{\|P\|} \left\{ {\|V\| \atop k} \right\} \right)$ |

Table 1
Examples of number of possible allocations considered.

#### 4.1.1 Potentially Beneficial Duplication

##### Processors

When considering duplicating a task $n$, we can divide the set of processors into two mutually exclusive subsets: $P^C(n)$ is the set of all processors to which at least one child of $n$ is allocated, and $P^{\overline{C}}(n)$ is the set of processors to which no children are allocated. It is only useful for $n$ to be allocated to at most one processor in $P^{\overline{C}}(n)$. Communicated output data from such an instance will become available to all children at the same time, and so if there are two such instances one must be redundant. On the other hand, it is potentially useful for $n$ to be duplicated on any combination of the processors in $P^C(n)$. It is important to note that even if instances of $n$ are placed on every processor in $P^C(n)$, it is still necessary to consider an additional duplication on the processors in $P^{\overline{C}}(n)$ due to the "duplication anomaly".

The term "duplication anomaly" [20] refers to the possibility that a valid schedule with duplication may benefit from having a child of $n_i$ executed before $n_i$ itself, on the same processor. It is not required that a task $n_i$ enables any child tasks which are allocated to the same processor as itself. Instead, an optimal schedule may involve an instance $n_i^j$ enabling some of its children on the same processor, while an instance $n_i^k$ on a different processor enables the rest. Figure 3 demonstrates how this arrangement might occur, with a task graph and corresponding optimal schedule which relies on the duplication anomaly. Here the "anomaly" manifests as task $C$ being executed before its parent task $B$. The anomaly can occur when the difference in communication costs between two child tasks is sufficiently large, as is the case with tasks $C$ and $E$. Here we see task $C$ being enabled by $B^2$ (the instance of its parent not sharing the same processor), allowing it to begin execution earlier than if it was enabled by $B^1$ - this will in turn allow its child $F$ to begin earlier. Subsequently, task $E$ is enabled by $B^1$, allowing its very high communication cost to be partially avoided - it is able to begin two time units earlier than if it was enabled by $B^2$, thus justifying the duplication of task $B$.

##### Tasks

It is not useful to consider every task for duplication, as the duplication of some tasks can never be beneficial to the schedule length. Duplication is only potentially useful when an additional instance of a task allows a child (or in turn a descendant) of that task to begin execution at an earlier time. Since children may also be duplicated, this includes allowing any duplicated instance of a child task to be executed earlier. We start with the following definition.

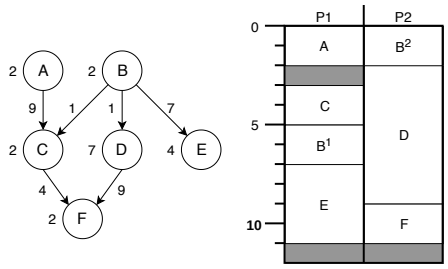


Figure 3. An example of the duplication anomaly.

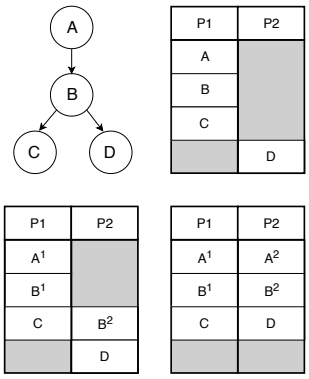


Figure 4. An example of a chain of duplicated tasks.

**Definition 4.1.** Duplicable
For a task $n \in V$, we define $duplicable(n)$ such that if $duplicable(n)$ is false, there exists no valid schedule in which $n$ is allowed to be duplicated with a smaller makespan than an optimal schedule in which $n$ is not allowed to be duplicated.

Based on this definition, we can formulate the following lemma, which states which tasks of a given task graph cannot be worth duplicating - and inversely, which tasks may be beneficial to duplicate . As such it provides a necessary, but not a sufficient condition for a task to be worthwhile duplicating.

**Lemma 4.2.** *Duplication-Worthy Tasks*
*Given is a task graph $G = \{V, E, w, c\}$. For any task $n \in V$, $duplicable(n) = false$ if $n$ has out-degree $deg^+(n) \leq 1$, and there is no task $d \in descendants(n)$ which has out-degree $deg^+(d) > 1$.*

*Proof:* We demonstrate the correctness by induction. As the base case, we take a "sink" task $n_\Omega$, with an out-degree of zero. Since $n_\Omega$ has no children, there is no way in which duplicating it could be beneficial. Therefore, $deg^+(n_\Omega) = 0 \implies duplicable(n_\Omega) = false$. Now consider

a parent $n_i$ of $n_\Omega$ for which $deg^+(n_i) = 1$. The single instance of $n_\Omega$ is all that $n_i$ will be required to enable. Assume that $n_i$ were duplicated, so that we have instances $n_i^1$ and $n_i^2$. One of these must allow $n_\Omega$ to begin execution at the earliest time, and so the other is not required to enable any other task. This means it can be removed from the schedule without harming the schedule length. Therefore, duplication cannot be beneficial in this case, and $duplicable(n_i) = false$. To complete the proof, this step is repeated until we reach a source task or a task which has $deg^+(n_i) > 1$ (each of such a task's ancestors by definition have a a descendant $d$ with $deg^+(d) > 1$). □

This lemma gives a static analysis as to which tasks are useful to consider for duplication. For example in Figure 5 we can observe that only three tasks, 1, 2 and 4, may be worth duplicating in principle. The lemma, however, does not make a statement about when it is useful to consider duplication for the other tasks, given a particular partial schedule. To help with this, we introduce the concept of allocated out-degree.

**Definition 4.3.** Allocated Out-Degree
Given a task $n$ and a partial partition $A$, the allocated out-degree $deg_\alpha^+(n, A)$ is the total number of child task instances which instances of $n$ will be required to enable in a valid schedule. For any task $n_i \in V$ we define $|n_i|^A$ as the number of instances of task $n_i$ that exist in partial allocation $A$, with $|n_i|^A = 1$ if $n_i$ has not yet been allocated in $A$. The allocated out-degree is then defined as $deg_\alpha^+(n, A) = \sum_{n_c \in succ(n)}(|n_c|^A)$. This value is greater or equal to the static out-degree $deg^+(n)$, which is simply the number of children of $n$.

It can only be useful to duplicate a task $n$ if there are multiple child tasks (or instances) dependent on it. This obviously includes all tasks with a static out-degree $deg^+(n) > 1$. However, since children of $n_i$ may also be duplicated, the allocated out-degree $deg_\alpha^+(n, A)$ may become higher than the static out-degree. Duplication should be considered if $deg_\alpha^+(n, A) > 1$. As shown in Figure 5, this means that a task $n$ will only ever be considered for duplication if it has $deg^+(n) > 1$, or is the ancestor of such a task.

Using all of this, we define the complete set of allocations which we intend to allow to exist in our state-space.

**Definition 4.4.** Valid Allocation Allowing Potentially Beneficial Task Duplication

A valid allocation allowing potentially beneficial task duplication is an allocation $A_D$ which meets the following constraints for each task $n \in |V|$:

1) Task $n$ is assigned to at least one processor.
2) If $deg_\alpha^+(n, A_D) = 1$, task $n$ is assigned to exactly one processor.
3) If $deg_\alpha^+(n, A_D) > 1$, task $n$ is assigned to zero or more processors in $P^C(n)$, in any combination. Additionally, $n$ is assigned to zero or one processors in $P^{\overline{C}}(n)$.

#### 4.1.2 Allocation Algorithm with Duplication

To define a state space which allows the creation of all valid allocations allowing potentially beneficial task duplication, we propose a branching scheme which closely resembles the one used in the original allocation phase, but with some additional possibilities at each step. Without duplication, branching in the allocation phase proceeds by selecting tasks for allocation one at a time in some arbitrary fixed order. With duplication, an arbitrary order will no longer suffice: the tasks must be considered in a reverse topological order, such that no task will be selected until all its children have already been allocated. When task $n$ is selected to be allocated, we check its allocated out-degree. If $deg_\alpha^+(n, A) \leq 1$, we do not consider duplication, and allocation is performed as normal. Otherwise, we proceed through two steps:

1) Task $n$ may be allocated to zero or more parts to which children of $n$ have already been assigned. This means that task $n$ could be placed in any combination of groupings which already contain one of its children. This set of possibilities is the power set $\mathbb{P}(P^C(n))$, that is the set of all subsets of $P^C(n)$.
2) Task $n$ may be allocated to at most one part to which none of its children were assigned. If $n$ was not allocated to any parts in step 1, it must be allocated to one part now. The task is placed in a grouping just as it would be in the original allocation phase, either in an existing grouping or as the first task of a new grouping, but with the restriction that it cannot be placed in the same grouping as any of its children, as case 1 already covers this possibility.

These rules define all possible child states in the new allocation phase with duplication. Algorithm 1 gives the corresponding pseudocode demonstrating the process required to create the *child states* of a given allocation state. The outer loop in line 7 ensures that all possibilities from the power set of $P^C(n)$ in step one are allowed. Lines 13 and 17 deal with the possibilities from $P^{\overline{C}}(n)$ in step two, paired with the possibilities in step one. Line 11 ensures that it is possible for child states to be created with task $n$ allocated to either no processors in $P^C(n)$, or no processors in $P^{\overline{C}}(n)$, but never to no processors at all.

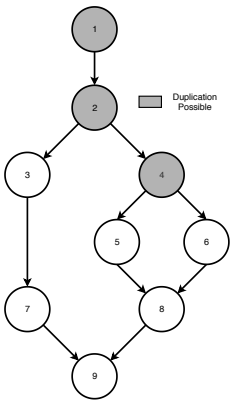


Figure 5. Example graph showing tasks eligible for duplication.

### 4.2 Ordering

The introduction of duplication requires a change to the definition of the local ready lists in the ordering phase. If

**Algorithm 1** Defining child states of an allocation state with task duplication.

**Input:** $A$, a partial partition of $V$
**Output:** $childStates_A$, set of partial partitions more complete than $A$

$unallocated \leftarrow$ list of all tasks $v \in V$ not in $A$;
Sort $unallocated$ into reverse topological order;
$nextTask \leftarrow$ first task in $unallocated$;
$childStates_A \leftarrow \emptyset$;
$A^C(n) \leftarrow$ all parts $a \in A$ containing a child task of $nextTask$;
$A^{\overline{C}}(n) \leftarrow A \setminus A^C(n)$;
**for** *all* $x \in \mathbb{P}(A^C(n))$ **do**
  $childState^\alpha \leftarrow A$;
  **for** *all* $a \in x$ **do**
    Insert $nextTask$ into part in $childState^\alpha$ equal to $a$;
  **if** $x \neq \emptyset$ **then**
    $childStates_A \leftarrow childStates_A \cup childState^\alpha$;
  **for** *all* $y \in A^{\overline{C}}(n)$ **do**
    $childState^\beta \leftarrow childState^\alpha$;
    Insert $nextTask$ into part in $childState^\beta$ equal to $y$;
    $childStates_A \leftarrow childStates_A \cup childState^\beta$;
  **if** $|A| < |P|$ **then**
    $childState^\beta \leftarrow childState^\alpha \cup \{nextTask\}$;
    $childStates_A \leftarrow childStates_A \cup childState^\beta$;
**return** $childStates_A$;

a task has a parent which is duplicated, we cannot know *a priori* which copy of the parent will be able to provide the necessary input data first, even if one of the copies is on the same processor. Therefore, if a task is allocated to more than one processor, it cannot prevent any descendants also allocated to one of those processors from being considered ready. With our new definition, a task $n_i$ allocated to $p_i$ is considered to not be ready if there is an unordered *and unduplicated* task $n_j$ also allocated to $p_i$ which is an ancestor of $n_i$ in the graph $G$. Otherwise, $n_i$ is ready.

## 5 Lower Bounds with Duplication

The guarantee of optimality in branch-and-bound search depends on the calculation of lower bounds on the quality of complete solutions that can be reached from a given partial solution. A heuristic function is said to be *admissable* if it will never provide an overestimate, i.e. it is truly a lower bound. Given a state $s$, we require an admissable heuristic function $f$ such that $f(s) \leq f^*(s)$, where $f^*(s)$ is the actual lowest cost of any complete solution in the sub-tree rooted at $s$. The introduction of duplication means that the lower bounds previously used with AO are no longer admissable, as they do not take into account the possible reductions in schedule length which duplication allows. However, the necessary changes to these bounds can be isolated to the definition of some values that are common between them: the allocated top and bottom levels of tasks, $tl_A(n)$ and $bl_A(n)$. When statically analyzing a task graph, the top level of a task $tl(n)$ gives a lower bound on the time between the beginning of the schedule and the beginning of $n$'s execution. The bottom level $bl(n)$ gives a lower bound on the time between the beginning of $n$'s execution and the end of the schedule. When combined, these values can give us a lower bound for the total length of the schedule. The allocated levels are dynamic properties dependent on a specific partial or complete allocation, $A$. As tasks are allocated, we gain information about which communication costs will be incurred, and our bounds can be made tighter. Since duplication complicates the rules as to which communications are necessary, these values must be redefined. Importantly, when a task $n$ is duplicated, each instance of $n$ in the allocation can now be considered to have a distinct allocated top and bottom level. The $j$th instance of task $n_i$ will be denoted by $n_i^j$. We will define and use both **specific levels** for each duplicated instance, $tl_A(n_i^j)$, and **collective levels** for the set of duplicates of a task, $tl_A(n_i)$.

### 5.1 Collective and Specific Levels

Allocated **top levels** are usually found with a recursive procedure. To find $tl_A(n_i)$, we simply iterate over the parents of $n_i$ and find the maximum value for the sum of their allocated top level, their computational weight, and the necessary communication cost. This is expressed by the following formulas:

$$tl_A(n_i) = \max_{n_p \in pred(n_i)} \{tl_A(n_p) + w(n_p) + c_A(n_p, n_i)\} \quad (5)$$

$$c_A(n_p, n_i) = \begin{cases} c(e_{pi}), & n_i, n_p \in A \wedge proc(n_i) \neq proc(n_p) \\ 0, & otherwise \end{cases} \quad (6)$$

With duplication, however, the parents of $n_i$ may be duplicated. Since only one of those duplicate parents needs to supply input data to $n_i$, duplication introduces options. Previously, we could define the top level of $n_i$ as the longest path in the task graph ending at $n_i$. Duplication introduces new paths through the task graph, but not all of them are required to be taken. We need only consider the best possible option of parent instance to enable our task.

**Definition 5.1.** Specific Allocated Top Level

The specific allocated top level of $n_i^j$ is the longest path through the task graph ending with $n_i^j$ which we are forced to take. This is expressed by the formula:

$$tl_A(n_i^j) = \max_{n_p \in pred(n_i)} \left\{ \min_{n_p^k \in dups(n_p)} \left\{ tl_A(n_p^k) + w(n_p) + c_A(n_p^k, n_i^j) \right\} \right\} \quad (7)$$

Say that $n_i^j$ has a parent $n_p$, which may be duplicated. Among the set of instances of $n_p$, there is one which can provide data for $n_i^j$ at the earliest time. For the allocated top level $tl_A(n_i^j)$, we need only consider the path through the task graph which includes this instance of $n_p$.

We also wish to define a collective allocated top level for $n_i$.

**Definition 5.2.** Collective Allocated Top Level

The collective allocated top level for $n_i$ is the earliest time that any instance of $n_i \in dups(n_i)$ could start. This gives us the following formula:

$$tl_A(n_i) = min_{n_i^j \in dups(n_i)} \left\{ tl_A(n_i^j) \right\} \tag{8}$$

Duplication has similar, but not symmetric, implications for the allocated **bottom level**. Previously, $bl_A(n_i)$ could be determined recursively in the same way as $tl_A(n_i)$, but considering the bottom levels of $n_i$'s children rather than the top levels of its parents. The duplication of children does not have an effect on how allocated bottom level is calculated, as all instances of $n_i$'s children must always be supplied with input data. However, if $n_i$ itself is duplicated, we need to find an allocated bottom level for $n_i^j$. It is not clear that any given instance of $n_i$'s children will need to be enabled by $n_i^j$, as only a single instance of $n_i$ needs to perform that role. It is possible that a duplicate instance $n_i^j$ may not be required to enable any of the instances of its children. In this case, $n_i^j$ could be executed arbitrarily late in the schedule. In practice, if this scenario occurred, it would mean that this duplication is providing no benefit to the schedule. However, this cannot be decided before ordering is performed: we can only say that it is a possibility.

**Definition 5.3.** Specific Allocated Bottom Level

The specific allocated bottom level for an instance of a duplicated task $n_i^j$ gives a lower bound on the time between the start of execution of $n_i^j$ and the end of the schedule. This is equal to its computational weight, $bl_A(n_i^j) = w(n_i)$, as any instance of a duplicated task could be placed at the end of a valid schedule.

It is this fact which motivates the definition of both specific and collective levels, as the specific allocated bottom level in this instance is not a very useful bound.

**Definition 5.4.** Collective Allocated Bottom Level

The collective allocated bottom level $bl_A(n_i)$ gives a lower bound on the time between when the earliest instance of $n_i$ starts execution and the end of the schedule. We know that some instance of $n_i$ must enable each of the children of $n_i$. To find $bl_A(n_i)$, we iterate over all children $n_c \in succ(n_i)$. For each child $n_c$, we iterate over the instances $n_c^k \in dups(n_c)$. For each instance $n_c^k$, we iterate over the instances of $n_i$ and find the minimum necessary communication cost, then add the collective allocated bottom level of the child. This represents the outgoing path from the best possible enabling instance for that child. The allocated bottom level $bl_A(n_i)$ is then the maximum such outgoing path found, plus the computation cost $w(n_i)$. This is described by the following formula:

$$bl_A(n_i) = w(n_i) + \max_{n_c \in succ(n_i)} \left\{ bl_A(n_c) + \min_{n_c^k \in dups(n_c)} \left\{ \min_{n_i^j \in dups(n_i)} \left\{ c_A(n_i^j, n_c^k) \right\} \right\} \right\} \tag{9}$$

When producing a bound for the length of the entire schedule, it is important to pair the specific and collective levels carefully. For a given task instance $n_i^j$, the sum of the specific levels $tl_A(n_i^j) + bl_A(n_i^j)$ gives an admissable lower bound. For a given task $n_i$, the sum of the collective levels $tl_A(n_i) + bl_A(n_i)$ gives an admissable lower bound.

**Lemma 5.5.** *$tl_A(n_i^j) + bl_A(n_i^j)$ is a lower bound for the length of a schedule.*

*Proof:* If $tl_A(n_i^j)$ is a lower bound for the start time of instance $n_i^j$, and $bl_A(n_i^j)$ is a lower bound for the time from the start of $n_i^j$ till the end of the schedule, the overall schedule must be at least as long as their sum.

- For a task instance $n_i^j$ with no parents, $tl_A(n_i^j) = 0$. A task instance $n_i^j$ with parents must be enabled by some instance $n_p^k$ of each $n_p \in parents(n_i)$. For each $n_p$, there is some $n_p^k$ which gives the minimum value for $tl_A(n_p^k) + w(n_p) + c_A(n_p^k, n_i^j)$ if it enables $n_i^j$. By the precedence constraint, the start time of $n_i^j$ must be at least the maximum of these values among the enabling instances. By induction, $tl_A(n_i^j)$ is a lower bound for the start time of instance $n_i^j$.
- $bl_A(n_i^j)$ is equal to $w(n_i)$, and is therefore trivially a lower bound for the time from the start of $n_i^j$ till the end of the schedule.

□

For a given task $n_i$, the sum of the collective levels $tl_A(n_i) + bl_A(n_i)$ gives an admissable lower bound.

**Lemma 5.6.** *$tl_A(n_i) + bl_A(n_i)$ is a lower bound for the length of a schedule.*

*Proof:* If $tl_A(n_i)$ is a lower bound for the start time of any instance of $n_i$, and $bl_A(n_i)$ is a lower bound for the time from the earliest start of any instance of $n_i$ till the end of the schedule, the overall schedule must be at least as long as their sum.

- $tl_A(n_i)$ is trivially a lower bound for the start time of any instance of $n_i$ because it is the minimum of the specific allocated top levels $tl_A(n_i^j)$ among all $n_i^j$.
- For a task $n_i$ with no children, $bl_A(n_i) = w(n_i)$. For a task with children, each child instance $n_c^k$ must be enabled by some instance $n_i^j$. For each $n_c$, there is some instance $n_i^j$ which gives the minimum value for $c_A(n_i^j, n_c^k)$ if it enables some instance $n_c^k$. By the precedence constraint, some $n_i^j$ must start at least this much time, plus its own weight $w(n_i)$, before the earliest starting instance of $n_c$, bounded by $bl_A(n_c)$. By induction, $bl_A(n_i)$ is a lower bound for the time from the earliest start of any instance of $n_i$ till the end of the schedule.

□

In general, it is preferable to use specific allocated top levels and collective allocated bottom levels, as these tend to provide tighter bounds than their alternatives. However, combining a specific allocated top level with a collective allocated bottom level may produce an overestimate for the length of the optimal schedule. When applying these bounds to the AO model, both types of level are used in each

phase. In the allocation phase, the allocated critical path heuristic uses the collective allocated top and bottom levels. In the ordering phase, the specific allocated top levels are used to determine estimated earliest start times for specific task instances. The specific allocated bottom levels are then added to these to obtain a bound. However, if a task instance $n_i^j$, has the lowest estimated earliest start time among all instances of $n_i$, the collective allocated bottom level is added instead. This is permissible because the estimated earliest start time for $n_i^j$ in this case is a lower bound for the start time of any instance of $n_i$, just like $tl_A(n_i)$.

## 6 Evaluation

### 6.1 Setup and workload

To determine empirically the effect of duplication on the difficulty of solving task scheduling problems with the AO model, we performed searches on a large set of task graphs using a variety of different target systems. Task graphs of sizes 16 and 21 tasks were used, with 270 of each being selected, hence 540 different graphs in total. An additional set of larger graphs with 30 tasks were also selected - these will be discussed in Section 6.3.The graphs used are the same as those used to evaluate the AO model in previous work - a large and diverse data set of generated task graphs [12] differing by the following attributes: graph structure, the number of tasks, and the communication-to-computation ratio (CCR). This set of task graphs is available for use in GXL and DOT formats[1]. The graphs were a mix of the following DAG structure types: Fork, Fork-Join, Out-Tree, Pipeline, Random, Series-Parallel, and Stencil. DAG structures can be divided into three categories: structures which never benefit from duplication, structures where duplicating certain tasks can never be harmful, and the rest for which the impact of duplication is uncertain. The first category includes Join and In-Tree graphs, and for this reason they were excluded from our data set. The second category includes Fork and Fork-Join graphs, where duplicating the entry task can never be harmful (this is also the only task for which duplication is meaningful). The experiments with these fork-based graph structures will therefore be analysed separately from the general graph structures in the remainder of the data set. The graphs in the data set also vary by communication-to-computation ratio (CCR), evenly divided into three categories: low (close to 0.1), medium (close to 1) and high (close to 10).

A mature implementation of parallel depth-first branch-and-bound and the AO model in Java [11] was extended and enhanced to allow duplication. This implementation was used for the evaluation. For each task graph, we attempted to find an optimal schedule for each target system with and without duplication, with 2, 4, and 6 processors, and a time limit of one minute allowed for each search to complete. There were a total of 3240 trials. Each trial was run on a Linux machine with 4 Intel Xeon E7-4830 v3 @2.1GHz processors. To remove the possibility of previous trials affecting subsequent ones due to garbage collection or JIT compilation, a new JVM instance was started each time.

1. http://parallel.auckland.ac.nz/OptimalTaskScheduling/BenchmarkSet.zip

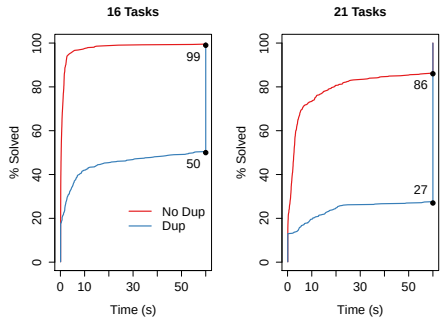


Figure 6. Performance with and without duplication for general graph structures.

### 6.2 Results

#### 6.2.1 Solving difficulty

A summary of results from these trials for general graph structures (i.e. excluding fork-like structures) is shown in Figure 6 as performance profiles: the $x$-axis shows time elapsed, while the $y$-axis shows the cumulative percentage of problem instances which were successfully solved by this time. Almost all problem instances were able to be solved optimally within the one minute time limit when duplication was not used, with 99% solved in the 16 task group and 86% solved in the 21 task group. However, allowing duplication led to a significant decrease in the proportion of task graphs able to be solved. Not only were less problem instances solved with duplication, but the decrease in performance between the 16 and 21 task groups is much more significant, dropping from 50% to only 27%. This is expected, given the additional complexity and therefore much larger state-space associated with duplication.

In Table 2, we have compiled a comparison of ILP formulations in the literature, and their reported success in optimal solving of task scheduling problems with some form of duplication. None of these formulations is directly comparable to the problem addressed in this work, having significant differences in objective function, environment, etc. The most comparable is Bender, as this is the only one which attempts to find a fully optimal duplication (while not limiting duplication to potentially beneficial tasks/processors). While bearing in mind these complications in comparison, it is clear that the size of problem instances solved by these formulations is very similar to that used in this work - even the largest graph solved is of a similar order of magnitude. This shows that despite the general advantages of ILP solvers in speed, the branch-and-bound approach is still very competitive.

#### 6.2.2 Duplication benefit

This naturally leads to the question of whether it is generally worthwhile to attempt to find a solution which includes duplication, given the added difficulty. One factor in making this decision would be the likelihood that allowing duplication will cause a reduction in the length of the optimal schedule for a given task graph.

6.2.2.1 Communication to Computation Ratio: In Figure 7 we see cumulative distribution plots for the improvement in schedules gained by duplication for general

| Source | Difference in Scheduling Model | Duplication Details | Max Problem Size Solved | Solving Times |
|---|---|---|---|---|
| Bender 1996 [2] | Unrelated heterogeneous processors | Any task on any processor | 8-12 tasks | Up to 9374 secs |
| Tosun 2012 [23] | Energy and reliability, no communication costs | For reliability only | 2-16 tasks, 2-8 procs | Up to 400 secs |
| Singh 2012 [19] | Unrelated heterogeneous processors | Restricted | Up to 20 tasks | 1000 sec limit |
| Tang 2016 [21] | Allocation predetermined | Predetermined | 14-48 tasks | 10 min limit |

Table 2
Comparison of reported successful optimal solving between ILP formulations with duplication.

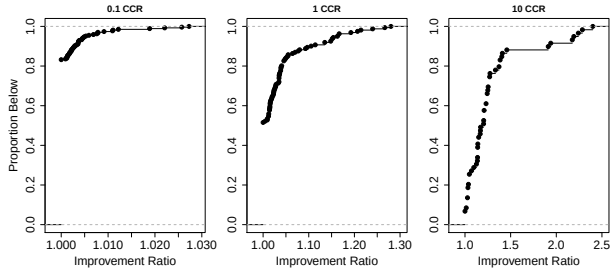


Figure 7. Improvement from duplication by CCR.

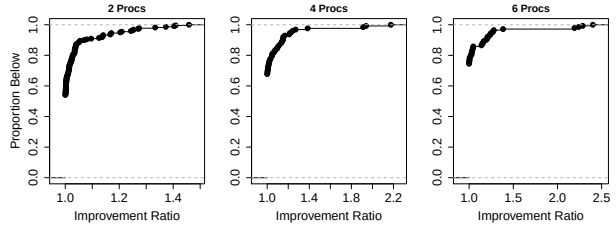


Figure 8. Improvement from duplication by number of processors.

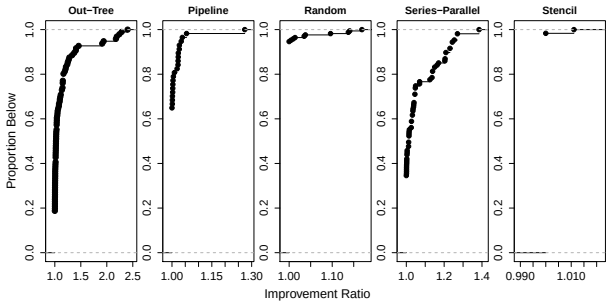


Figure 9. Improvement from duplication by graph structure.

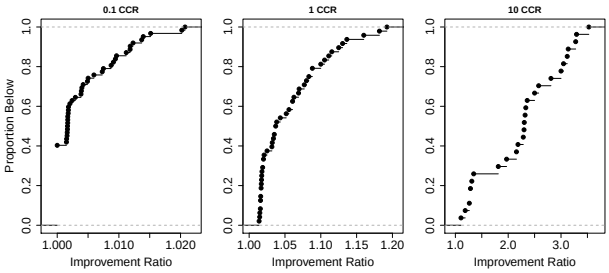


Figure 10. Improvement from duplication for fork-based graph structures.

graph structures. We calculate an improvement ratio by dividing the makespan of each schedule found without duplication by the makespan of the corresponding schedule with duplication. Problem instances are therefore only included in this analysis if they were successfully solved in both cases. The results are broken down by CCR. It is clear that CCR has a very large effect on the usefulness of duplication. With low CCR, only 20% of schedules see any improvement, and at most they are improved by a few percent. With medium CCR we see half of the schedules improved, with the improvement ratio reaching 1.3. For high CCR, almost all schedules are improved by duplication, and the maximum improvement ratio is 2.5, almost twice that of the medium CCR group. Since the advantage of duplication is achieved by trading communication costs for additional computation, it is natural that graphs with a higher CCR would have a larger potential benefit, and these results confirm this. Remember that the best possible improvement is limited by the best possible speedup, and the schedules created were on 2, 4 and 6 processors. A task graph having high CCR indicates some combination of two factors in a real-world scheduling situation it may serve as an abstraction of: either large amounts of data need to be transmitted while relatively simple computations are performed with it, or the communication links of the parallel system are relatively slow compared to the speed of the processing units (as may be true in a distributed system), or both.

6.2.2.2 Number of Processors: Figure 8 shows plots for the improvement in schedules, grouped by the number of processors allowed for scheduling. Compared to CCR, there are much smaller differences between these groups. We do see, however, that as the number of processors increases there is a small decrease in the proportion of schedules which are improved. At the same time, the range of improvement expands significantly, although the majority of instances in all groups have an improvement ratio of less than 1.5. This range expansion is expected as the maximal possible improvement grows with the number of processors, i.e. with the possible speedup.

6.2.2.3 Graph Structures: We plot the improvement in schedules grouped by graph structure in Figure 9. We can see that duplication has the largest effect with Out-Tree and Series-Parallel graphs, with a large majority of these schedules being significantly improved. These structures are most likely to resemble fork-like structures, where scheduling cannot be harmed by duplication. In particular, we can say that Out-Tree schedules cannot be harmed by duplicating the source task, in just the same manner as Fork graphs. The duplication of later tasks will have uncertain results, though.

Let us now look at graphs where duplicating certain tasks can never be harmful, namely fork and fork-join graphs. How much do they benefit from duplication? Figure

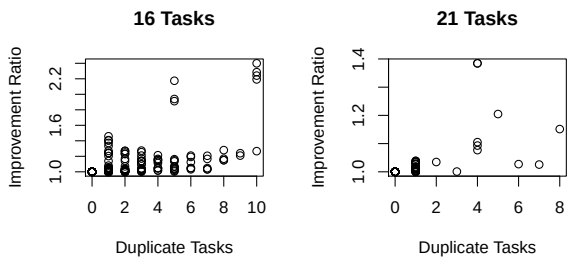


Figure 11. Improvement from duplication vs. number of extra tasks.

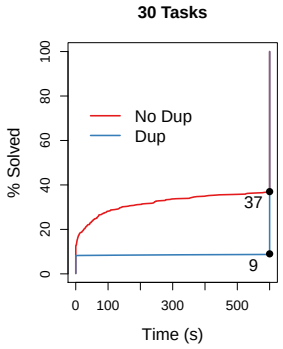


Figure 12. Performance with and without duplication for 30-task graphs.

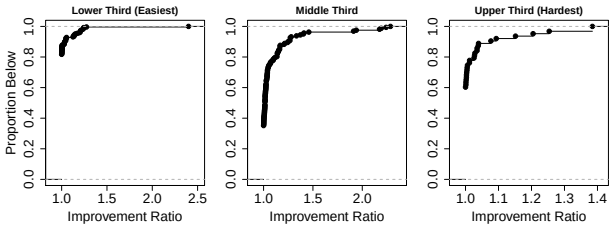


Figure 13. Improvement from duplication by time taken to solve.

10 shows cumulative distribution plots for the improvement in schedules gained by duplication for fork-based graph structures. In contrast to the general graph structures, many more problem instances benefit from duplication and their maximal benefit is also higher, which is of course an expected result.

6.2.2.4 Amount of Duplication: It was considered that some insight may be gained by analysing the extent of the duplication occuring in schedules which benefit from it. Figure 11 shows scatter plots comparing the number of extra tasks found in optimal schedules against the improvement ratio from duplication. The number of extra tasks is calculated as the difference between the original number of tasks in the task graph and the number of tasks planned for execution in the discovered optimal schedule. A limitation of this analysis is that the search method used does not find optimal schedules with minimal duplication - in many cases, extra tasks are redundant, and neither benefit nor harm the length of the schedule. This can be easily observed in the scatter plots from the existence of data points with a range of extra tasks along the baseline of the improvement axis. Bearing this limitation in mind, no particular trend is evident in the data here. However, the possibility of conducting such an analysis may still be interesting for other data sets, particularly if the method is modified to produce schedules with minimal duplication.

### 6.3 Larger Graphs

We wished to evaluate the performance of the AO model with duplication on larger task graphs, to see how it would scale. A set of task graphs with 30 tasks was selected, having been used to evaluate the base AO model in previous work. From this set the graphs with Out-Tree, Pipeline, Random, Series-Parallel, and Stencil were chosen, giving a total of 210 graphs for this evaluation. The experimental protocol was identical to the previous evaluation, with the exception that the time limit was raised to ten minutes. Figure 12 shows performance profiles for AO with and without duplication on this larger dataset. Without duplication, about 37% of problem instances were able to be solved. With duplication, only 9% of instances were solved within the time limit. Within these instances solved successfully with duplication, however, there were no optimal schedules produced in which duplication was beneficial.

A possible explanation for this is that problem instances where duplication can be beneficial tend to be inherently more difficult to solve than those where it cannot be beneficial. To investigate this possibility, additional analysis of the 16 and 21 task data sets was performed. Figure 13 shows the successfully solved instances from those sets divided into three groups based on the time taken to solve them. They are divided approximately evenly into the third of results which took the least time (< 275 ms), the third of results which took the most time (> 2230 ms), and the third in between. These can be considered as a grouping into the easiest solvable problem instances, the hardest, and those of median difficulty. It is clear that far fewer of the optimal schedules from the easiest third use duplication beneficially. This provides some support for this explanation.

## 7 Conclusions and Future Work

In this work, we have proposed a state-space model for optimal task scheduling with duplication, based on the AO model. Allowing tasks to be duplicated and executed multiple times within a schedule can lead to reductions in overall schedule length, and the modified AO model can determine the best such scheme for duplication in a given instance. We have also redefined the concepts of allocated top and bottom levels for the context of scheduling with duplication.

The complexity added when allowing duplication makes finding optimal solutions significantly more difficult, with an empirical evaluation showing a large drop in the number of task graphs solved within one minute. This suggests that allowing duplication represents a significant additional resource investment when deciding how to optimally schedule a task graph. Our evaluation also showed that a large proportion of task graphs can have their optimal schedules improved by the use of duplication, and many significantly.

This is particularly true for graphs with a fork-like structure, and for those with a medium to high CCR.

Modifying this method to produce schedules with minimal duplication would be useful in further analysing the benefit of duplication. Another natural next step from this work would be to further extend the AO model such that optimal schedules including duplication could be found for parallel systems with heterogeneous processors. It could also be adapted to work with more complex and realistic task scheduling models, such as one that considers contention in communication. In addition, the concepts proposed in this work could be used to develop an ILP formulation which attempts optimal task scheduling with duplication in the same manner.

## REFERENCES


[1] Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel & Distributed Systems*, (9):872–892, 1998.

[2] Armin Bender. Milp based task mapping for heterogeneous multiprocessor systems. In *Proceedings EURO-DAC'96. European Design Automation Conference with EURO-VHDL'96 and Exhibition*, pages 190–197. IEEE, 1996.

[3] Alan Bundy and Lincoln Wallen. Branch-and-bound algorithms. In Alan Bundy and Lincoln Wallen, editors, *Catalogue of Artificial Intelligence Tools*, Symbolic Computation, pages 12–12. Springer Berlin Heidelberg, 1984.

[4] Maciej Drozdowski. *Scheduling for Parallel Processing*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[5] Abdessamad Ait El Cadi, Rabie Ben Atitallah, Saï d Hanafi, Nenad Mladenović, and Abdelhakim Artiba. New mip model for multiprocessor scheduling problem with communication delays. *Optimization Letters*, pages 1–17, 2014.

[6] Li Guodong, Chen Daoxu, Wang Daming, and Zhang Defu. Task clustering and scheduling to multiprocessors with duplication. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8–pp. IEEE, 2003.

[7] Shobhit Gupta, Ranjit Rajak, Girish Kumar Singh, and Sanjay Jain. Review of task duplication based (tdb) scheduling algorithms. *SmartCR*, 5(1):67–75, 2015.

[8] Tarek Hagras and Jan Janecek. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 107. IEEE, 2004.

[9] Ravneet Kaur and Ramneek Kaur. Multiprocessor scheduling using task duplication based scheduling algorithms: A review paper. *International Journal of Application or Innovation in Engineering and Management*, 2(4):311–317, 2013.

[10] Sven Mallach. Improved mixed-integer programming models for multiprocessor scheduling with communication delays. 2016.

[11] Michael Orr and Oliver Sinnen. Further explorations in state-space search for optimal task scheduling. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 134–141. IEEE, 2017.

[12] Michael Orr and Oliver Sinnen. Optimal Task Scheduling Benefits From a Duplicate-Free State-Space. *arXiv e-prints*, page arXiv:1901.06899, January 2019.

[13] Chan-Ik Park and Tae-Young Choe. An optimal scheduling algorithm based on task duplication. In *Parallel and Distributed Systems, 2001. ICPADS 2001. Proceedings. Eighth International Conference on*, pages 9–14. IEEE, 2001.

[14] Samantha Ranaweera and Dharma P Agrawal. A task duplication based scheduling algorithm for heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 445–450. IEEE, 2000.

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

[16] Ahmed Zaki Semar Shahul and Oliver Sinnen. Scheduling task graphs optimally with A*. *Journal of Supercomputing*, 51(3):310–332, March 2010.

[17] KwangSik Shin, MyongJin Cha, MunSuck Jang, JinHa Jung, WanOh Yoon, and SangBang Choi. Task scheduling algorithm using minimized duplications in homogeneous systems. *Journal of Parallel and Distributed Computing*, 68(8):1146–1156, 2008.

[18] Jagpreet Singh, Sandeep Betha, Bhargav Mangipudi, and Nitin Auluck. Contention aware energy efficient scheduling on heterogeneous multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1251–1264, 2015.

[19] Jagpreet Singh, Bhargav Mangipudi, Sandeep Betha, and Nitin Auluck. Restricted duplication based milp formulation for scheduling task graphs on unrelated parallel machines. In *2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming*, pages 202–209. IEEE, 2012.

[20] Oliver Sinnen. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007.

[21] Qi Tang, Shang-Feng Wu, Jun-Wu Shi, and Ji-Bo Wei. Optimization of duplication-based schedules on network-on-chip based multiprocessor system-on-chips. *IEEE transactions on parallel and distributed systems*, 28(3):826–837, 2016.

[22] Xiaoyong Tang, Kenli Li, Guiping Liao, and Renfa Li. List scheduling with duplication for heterogeneous computing systems. *Journal of parallel and distributed computing*, 70(4):323–329, 2010.

[23] Suleyman Tosun. Energy-and reliability-aware task scheduling onto heterogeneous mpsoc architectures. *The Journal of Supercomputing*, 62(1):265–289, 2012.

[24] Tatsuhiro Tsuchiya, Tetsuya Osada, and Tohru Kikuno. Genetics-based multiprocessor scheduling using task duplication. *Microprocessors and Microsystems*, 22(3-4):197–207, 1998.

[25] B. Veltman, B. J. Lageweg, and J. K. Lenstra. Multiprocessor Scheduling with Communication Delays. *Parallel Computing*, 16(2-3):173–182, 1990.


**Michael Orr** is a PhD student in the Department of Electrical and Computer Engineering at the University of Auckland, New Zealand, where he is a member of the Parallel and Reconfigurable Computing Lab (PARC). Michael's research focuses on optimal task scheduling for parallel systems. He graduated in Software Engineering from the University of Auckland.

**Oliver Sinnen** is Associate Professor in the Department of Electrical, Computer and Software Engineering at the University of Auckland, where he founded and leads the Parallel and Reconfigurable Computing Lab. He graduated in Electrical and Computer Engineering at RWTH Aachen University, Germany and then received his PhD from Instituto Superior Técnico (IST), University of Lisbon, Portugal. Oliver's research is situated in the area of parallel computing and programming, with focus on scheduling algorithms for parallel systems; reconfigurable computing and acceleration with FPGAs; software engineering for parallel programming. He leads a team at the UoA contributing to the development of the Square Kilometre Array (SKA) radio telescope. Oliver authored the book "Task Scheduling for Parallel Systems", Wiley 2007.