List-Scheduling vs. Cluster-Scheduling

Huijun Wang, Oliver Sinnen
University of Auckland, New Zealand

***Abstract*—In scheduling theory and parallel computing practice, programs are often represented as directed acyclic graphs. Finding a makespan-minimising schedule for such a graph on a given number of homogenous processors ($P|prec, c_{ij}|C_{max}$) is an NP-hard optimisation problem. Among the many proposed heuristics, the two dominant approaches are list-scheduling and cluster-scheduling (based on clustering), whereby clustering targets an unlimited number of processors at its core. Given their heuristic nature, many experimental comparisons exist. However, their overwhelming majority compares algorithms within but not across categories. Hence it is not clear how cluster-scheduling, for a limited number of processors, performs relative to list scheduling or how list scheduling, for an unlimited number of processors, performs against clustering. This study addresses these open questions by comparing a large set of representative algorithms from the two approaches in an extensive experimental evaluation. The algorithms are discussed and studied in a modular nature, categorizing algorithms into components. Some of the included algorithms are previously unpublished combinations of these techniques. This approach also permits to study the separate merit of techniques like task insertion or lookahead. The results show that simple low-complexity algorithms are surprisingly competitive and that more sophisticated algorithms only exhibit their strengths under certain conditions.**

I. Introduction

In parallel processing, an important effort is to maximise the efficiency of program (or workload) schedules. Tasks in the program that can execute independently are scheduled to processors and time slots in a way that takes advantage of available processing power and minimises the total length of execution. In scheduling theory [34], [8] and in runtime environments [6], [3], programs are represented as a directed acyclic graph (DAG) where nodes represent atomic tasks and edges represent communications between them. A communication happens when a task provides (as its output) the inputs required for another task, creating a precedence relationship. Processors are assumed to be homogeneous and fully connected, and take time to communicate with each other.

Finding the optimal task schedule for a given DAG and number of processors ($P|prec, c_{ij}|C_{max}$ in $\alpha|\beta|\gamma$-notation) is intractable (a strongly NP-hard problem [34]) and many heuristic methods to find a solution exist. There are two very dominant solution approaches. One is list-scheduling which produces a schedule in one pass. Another one is cluster-scheduling, sometimes called clustering, where tasks are first grouped into clusters that are best executed together on the same processor to avoid extensive communication, and subsequent steps develop a schedule from that assignment. A fundamental difference between the two approaches is that list-scheduling normally targets a *limited* number of processors but clustering targets an *unlimited* number. To make the latter useful for a limited number of processors, methods exist to map and schedule the clusters to a limited number of processors [32], [30], [40], [23], [41]. Many algorithms have been proposed for each of the approaches and they have been compared in many studies as will be discussed below. However, the vast majority of the studies only compared algorithms within their categories, e.g. limited-processor schedulers against each other or clustering algorithms against each other. An open question is whether schedulers based on clustering provide an advantage over simpler one pass list-schedulers for a limited number of processors. Also, how well do list-scheduling algorithms perform for an unlimited number of processors (i.e. number of processor set to number of tasks)? These are two of the questions (section II-C) which this study intends to answer.

Many comparisons of scheduling algorithms can be found both on their own and accompanying other work [1], [17], [26], [15], [7]. Few of them have set out to compare list-scheduling and cluster-scheduling together. [15], [7] do not include cluster-schedulers. [1] treats clustering and scheduling to limited processors as different classes of problems which they are on their own. [26] compares both types of schedulers on unlimited processors. [30], [29], [31] briefly uses both types of schedulers to benchmark new algorithms. [17] compares both approaches but adds randomisation to algorithms. [23] only discusses methods within cluster-scheduling.

This work compares the two approaches with an extensive experimental study using a large set of representative algorithms from each class and a large set of graphs, with many different structures, sizes and other parameters. The algorithms are constructed and implemented in a modular way, breaking them down into different techniques that were proposed across the literature. For example, the general list-scheduling approach is separated from task priorities and techniques like lookahead, which are considered as modular features that can be used or not used with a given algorithm. Techniques are recombined where they have not been before, to evaluate unexplored combinations.

Section II formally defines the scheduling model, gives an overview of the considered algorithms and defines our experimental aims. Sections III and IV study the different algorithmic techniques and components of list-scheduling and cluster-scheduling, respectively. The setup of the simulation experiments is presented in section V followed by the evaluation is section VI. We conclude the paper in section VII.

II. Scheduling

A program to be parallelised is represented as a weighted directed acyclic graph (DAG) called a task graph, where nodes are indivisible units of work called tasks and edges are communications between tasks. The node and edge weights respectively give the computation and communication costs. In a complete schedule, every task n is allocated a processor $proc(n)$ and start time $start(n)$, with the condition that all of n's incoming communications arrive on $proc(n)$ before $start(n)$, meaning all of n's input data are available. This requires the tasks to execute in a topological order. The target system modelled is one where processors are homogeneous and fully connected with no network contention or processing cost for communications. Tasks take the same time to complete on all processors, as do communications between all pairs of processors. Communications within processors are instant and their costs are "zeroed".

A. Definitions

Inputs to the problem can be given as the tuple $(\boldsymbol{V}, \boldsymbol{E}, w, c, \boldsymbol{P})$. Its elements are (in this order) the set of tasks, set of edges, vector of task costs, matrix of communication costs, and set of processors, where the first four makes up task graph $\boldsymbol{G}$. The outputs are processor allocations $proc : \boldsymbol{V} \to \boldsymbol{P}$ and start times $start : \boldsymbol{V} \to \mathbb{N}^0$. Table I summarises these notations. The functions $succ$ and $pred$ are derived from $\boldsymbol{G}$. Some useful concepts are introduced next.

Table I
Notations used in scheduling discussion

Notation	Meaning
w_n	Computation cost of task n
$c_{i,j}$	Cost of communication from task i to task j
$succ(n)$	Children (successors) of task n
$pred(n)$	Parents (predecessors) of task n
$proc(n)$	Processor to which task n is assigned
$start(n)$	Start time of task n

Ready task – A task is ready when all of its parents have been scheduled. This is used when tasks are scheduled in topological order.

Data ready time – of a task n on processor p, $t_{dr}(n, p)$, is the time at which all of n's incoming communications arrive at p from scheduled parents. Let $\boldsymbol{S}$ be scheduled tasks in the following definition.

$$t_{dr}(n,p) = \max_{i \in pred(n) \cap \boldsymbol{S}} \begin{cases} p = proc(i) & 0 \\ otherwise & start(i) + w_i + c_{i,n} \end{cases}$$

Earliest start time (EST) – (or ASAP start time) of a ready task n, $est(n)$, is the earliest time at which n may begin executing on any processor. The following is a list-scheduling definition, where $available(p, n)$ is the earliest time at which processor p can fit n, whether behind all other tasks or into a gap (insertion strategy III-B1).

$$est(n) = \min_{p \in \mathbf{P}}(\max(\{available(p,n), t_{dr}(n,p)\}))$$

The EST of n on a specific processor p is defined as $est(n,p) = \max(\{available(p,n), t_{dr}(n,p)\})$.

Top-level – of a task n, $tlevel(n)$, is the length of the longest path through the DAG from a source task (including itself) to n. It can be recursively defined as follows.

$$tlevel(n) = \begin{cases} n \in sources & 0 \\ otherwise & \max\limits_{i \in pred(n)}(tlevel(i) + w_i + c_{i,n}) \end{cases}$$

Bottom-level – of a task n, $blevel(n)$, is the length of the longest path from n to any sink task (including n and the sink task). It is recursively defined as follows.

$$blevel(n) = \begin{cases} n \in sinks & w_n \\ otherwise & \max\limits_{j \in succ(n)}(blevel(j) + c_{n,j}) + w_n \end{cases}$$

Critical path – this is the longest path from a source task to a sink task through the DAG, including all computation and communication costs.

The allocated bottom(top)-level/critical path is the bottom(top)-level/critical path calculated with communications within processors having no cost.

Pseudo-edges – (virtual edges) are zero-weight edges added to the original DAG to reflect additional precedence constraints due to sequential execution within processors. Some algorithms use pseudo-edges in calculations of allocated top and bottom levels.

B. Algorithms Overview

Two main approaches to task scheduling using this model are list-scheduling and cluster-scheduling.

List-scheduling assigns priorities to tasks, and schedules them in a topological order in decreasing priority. Priorities can be static and remain constant for a DAG, or be dynamic and change with the schedule. Tasks are normally scheduled to processors which give them the earliest start times, but there are lookahead techniques which can be applied (section III-B2). Algorithm 1 shows the general structure of list-scheduling.

Algorithm 1: List-scheduling

```
while there are tasks to be scheduled do
    Identify a highest priority task n (e.g. from a list);
    Choose a processor p for n;
    Schedule n on p at est(n, p);
end
```

Cluster-scheduling involves several steps where list-scheduling has one. These are clustering, cluster-merging, and task-ordering within clusters. The aim of clustering is to group the tasks into any number of clusters where tasks in the same cluster are set to be scheduled on the same processor, and to do this in a way that best balances the reduction of inter-processor communications with the parallelism of task executions. This is often approached as a problem of scheduling the DAG onto an unlimited number of processors, where tasks scheduled onto the same

processor are grouped into a cluster. In the cluster-merging phase, clusters are merged so that their number matches the number of available processors. Since all processors are identical, the merged clusters are arbitrarily mapped onto processors. In the final task-ordering phase, task orderings are determined to obtain a schedule. Algorithm 2 shows the cluster-scheduling procedure. Since cluster-schedulers make it their focus to reduce communications, they should be more effective on communication-intensive graphs.

Algorithm 2: Cluster-scheduling

CLUSTERING:
Group the tasks into a set of clusters $\boldsymbol{C}$;
CLUSTER-MERGING:
while $|\boldsymbol{C}| > numberofprocessors$ **do**
  Merge a number of clusters in $\boldsymbol{C}$;
end
Map each cluster to a different processor;
TASK-ORDERING:
Order tasks within clusters to produce a schedule. Alternately, schedule tasks in some order and use the processor allocations;

The following is an outline of the algorithms that are used in this study. In addition to those schedulers used in their original forms as presented in literature, there are other ones that are mixtures of techniques taken from those algorithms and others, which have not been explored.

Both cluster-scheduling and list-scheduling can be modularised into independent components that can be combined in different ways. Cluster-scheduling is naturally split into three stages, normally with one algorithm for each stage. List-scheduling can also be split into two parts, a task priority scheme and a task placement scheme, both of which are often independent of each other. The task priority scheme is the basis on which the list-scheduler decides the order that tasks are scheduled in. The task placement scheme is the method by which the scheduler places an examined task onto a processor and start time allocation. Tables II and III show a list of static priority schemes and a list of placement schemes used respectively. They are described in more detail in section III. Dynamic priority list-schedulers are shown separately in table IV, because using lookahead with them (section III-B2a) is not as straightforward. Tables V, VI, and VII show some clustering algorithms, cluster-merging algorithms, and intra-cluster task-ordering algorithms respectively. All the complexities given are amortised time complexities. V is the number of nodes in the graph, E is the number of edges, W is the width of the graph (the highest possible number of ready tasks at once), P is the number of processors, and C is the number of clusters.

The tables only include the representative algorithms used in this study. Some algorithms were not used because they work with slightly less information [25], [21], or more information in a way that does not translate [2], [10], [13], [16], [24], [27], [37]. Figure 1 shows a summary of all the schedulers used in the experiment and how they have been composed from different algorithms and techniques. List-scheduling algorithms that exist in literature exactly as they have been put together here also have their names and

Table II
TASK PRIORITY SCHEMES (STATIC) IN LIST-SCHEDULING

Description	Code	Reference	Complexity
highest $blevel$ first	bl	[39], [36]	$O(E + V \log W)$
highest $blevel + tlevel$ first	tlbl	[36]	$O(E + V \log W)$
Critical-Path-Node-Directed	cpn	[19]	$O(E \log E + VlogV)$
Decisive Path Scheduling (variant of cpn)	dps	[28]	$O(E \log E + VlogV)$

Table III
TASK PLACEMENT SCHEMES IN LIST-SCHEDULING

Description	Code	Reference	Complexity (no insertion)	Complexity (insertion)
Scheduling to EST-processor	est	[36]	$O(E + V \log P)$	$O(V^2)$
Critical Child lookahead	cc	[20]	$O(E + VP)$	$O(V^2)$
Children Weighted EST lookahead	cwe	[5]	$O(V(E + VP))$	$O(V^3)$
Children Latest EST lookahead	cle	[5]	$O(V(E + VP))$	$O(V^3)$

Table IV
DYNAMIC PRIORITY LIST-SCHEDULERS

Description	Code	Reference	Complexity (no insertion)	Complexity (insertion)
Earliest Times First	etf	[14], [29]	$O(V(\log W + \log P) + E)$	–
Dynamic Level Scheduling	dls	[33]	$O(V^2)$	$O(V^3)$
etf with critical child lookahead	etfcc	–	$O(VWP(V + E))$	Not used
dls with critical child lookahead	dlscc	–	$O(VWP(V + E))$	Not used

Table V
CLUSTERING ALGORITHMS

Description	Code	Reference	Complexity
Dominant Sequence Clustering	dsc	[40]	$O(V^2)$ (version in study)
Dynamic Critical Path	dcp	[20]	$O(V^3)$
Linear Clustering	lc	[18]	$O(V(E + V))$

Table VI
CLUSTER-MERGING ALGORITHMS

Description	Code	Reference	Complexity
Adapted List-scheduling	ls	[32]	$O(PC(V + E))$
Guided Load Balancing	glb	[30]	$O((V + E)C))$ (version in study)

Table VII
TASK-ORDERING METHODS

Description	Code	Reference	Complexity
Highest bottom-level first	bl	[32]	$O(E + V \log W)$
Earliest EST first	etf	[41]	$O(V(\log W + \log P) + E)$

sources from literature identified. The name codes for the algorithms are concatenations of the codes for their components (in the order the components have been described). The string "ins" added to the end of a list-scheduler means that the algorithm uses insertion in its placement scheme. The cluster-merger with code "glbro" is the version of "glb" that ignores task orders produced by the clustering phase and re-orders tasks using bottom-level. It is included for comparison with "glb" which makes use of task orders from the clustering phase (section IV-B1). Some techniques are featured less because of bad performance, but instances of them are left in for comparison.

C. Experimental Aim

The major questions we want answers to in our experimental evaluation can be summarised as follows:

- How does list-scheduling compare with cluster-scheduling?
- How does list-scheduling compare with clustering algorithms when scheduling to an unlimited number of processors?
- Some list-scheduling components have not been evaluated on their own merits ("cpn", "tlbl", "cc"), are they good by themselves?
- Are there better task priorities for list-scheduling than the simple and effective bottom-level?
- How much better is list-scheduling with lookahead?
- How much better is list-scheduling with insertion?
- Are there particular combinations of list-scheduling techniques that are better than usual?

III. List-Scheduling

List-scheduling examines tasks in some order and gives a processor and start time to each examined task. This can be divided into two components, the task priority scheme, which determines the order in which tasks are listed and scheduled, and the task placement scheme which takes any given task to be scheduled and finds a processor and start time for it [34]. There are a range of options for both components, these will be described next.

A. Task Priority

Task priorities in list-scheduling can be static or dynamic. Static priorities do not change with the schedule and are computed once for the entire scheduling process, dynamic priorities are updated with the schedule. A static priority list-scheduler can have a priority list that is traversed to give tasks to schedule in decreasing priority. The list has to follow topological order. If the priority does not and a task's descendant can have equal or higher priority than itself, then ready tasks can be inserted into a priority queue which is dequeued to give tasks to schedule. For dynamic priorities, the scheduler has to re-compute the priority for free tasks when selecting one of them to schedule, since the priorities have potentially changed. This makes use of more information but increases complexity.

1) Static Priorities in Algorithms:

a) HEFT and MCP: Heterogeneous Earliest Finish Times (HEFT) [36] and Modified Critical Path (MCP) [39] use the bottom-level as their priority metrics. Their priority lists are ordered in descending bottom-level, and where there are ties, HEFT breaks them randomly and MCP breaks them with the bottom-levels of descendants. This study uses HEFT's version where ties are broken randomly, this should be effectively the same as MCP's alternative because ties in bottom-level will be rare in the graph sets used. This priority list takes $O(E + V \log W)$ time to build.

b) CPOP: The Critical Path on a Processor (CPOP) algorithm [36] uses the sum of the top-level and bottom-level as the priority metric, ordering tasks in decreasing $tlevel + blevel$. This one uses the priority queue like described earlier. This priority list also takes $O(E+V \log V)$ time to construct.

c) CPND: A task ordering scheme named Critical-Path-Node-Dominate (CPND) are used in [19]. The scheme recognizes three categories of nodes: the Critical Path Nodes (CPNs) which are nodes belonging to a critical path, the In-Branch Nodes which are nodes that lead to a CPN, and the Out-Branch Nodes (OBN) which belong to neither of the previous two. It empties a stack (into the priority list) which is initialized with all CPNs (with source CPN on top), popping the top node when it has no unlisted parents and pushing the unlisted parents otherwise. After all the CPNs have been added to the list and therefore all the IBNs, what remains are the OBNs which can be added in some order. An order is also needed for the unlisted parents of an examined node when they are pushed into the stack. An original version in [19] orders unlisted parents by decreasing bottom-level, breaking ties by ordering in increasing top-level, and orders OBNs by decreasing bottom-level. A priority list constructed in this way takes $O(E \log E + VlogV)$ time.

Some have used the idea of this technique and changed IBN and OBN orderings. One of these variants is the algorithm Decisive Path Scheduling (DPS) [28]. It first examines unlisted parents which are most constraining to the task's top-level – parents which have the highest top-level plus the cost of their communication to the task. In giving priority to the constraining parents and reducing their start times, the start time of the task itself, which is related to its top-level, should be reduced as well. This priority list also takes $O(E \log E + VlogV)$ time to build.

2) Dynamic Priorities in Algorithms: ETF and DLS are two list scheduling algorithms with dynamic task priorities. The dynamic attribute in their priorities is the task's EST in the partial schedule, which is defined in section II.

a) ETF: ETF (Earliest Time First) [14] prioritizes tasks with the smallest EST, ties are broken by choosing tasks with higher computation bottom-levels. A quality of ETF is that it tends to minimize gaps in the schedule. At any time, the earliest starting task is guaranteed to leave the smallest gap in execution on the processor providing its EST if scheduled to it, since no other task can start earlier there or anywhere else, including those which will be freed after scheduling any current free tasks.

The principle of EST scheduling has already been explored in task graph scheduling without communication

Algorithm Type	Task Priority Type (list-scheduling)	Insertion (list-scheduling)	Task Priority (list-scheduling)	Task Placement (list-scheduling)	Algorithm (code)	Literature
List Scheduling	Static	No Insertion	blevel	EST-processor	bl-est	–
				children latest EST lookahead	bl-cle	–
				children weighted EST lookahead	bl-cwe	–
				critical child lookahead	bl-cc	–
			tlevel + blevel	EST-processor	tlbl-est	–
				Weighted EST lookahead	tlbl-cwe	–
			cpn-directed	EST-processor	cpn-est	–
				Weighted EST lookahead	cpn-cwe	–
				Critical Child lookahead	cpn-cc	–
			cpn-directed variant (dps)	EST-processor	dps-est	DPS [19]
				Weighted EST lookahead	dps-cwe	–
				Critical Child lookahead	dps-cc	–
		Insertion	blevel	EST-processor	bl-est-ins	HEFT [27], MCP [29]
				Latest EST lookahead	bl-cle-ins	–
				Weighted EST lookahead	bl-cwe-ins	–
				Critical Child lookahead	bl-cc-ins	–
				critical path on a processor	bl-cpop-ins	–
			tlevel + blevel	EST-processor	tlbl-est-ins	–
				critical path on a processor	tlbl-cpop-ins	CPOP [27]
				Weighted EST lookahead	tlbl-cwe-ins	–
			cpn-directed	EST-processor	cpn-est-ins	–
				Weighted EST lookahead	cpn-cwe-ins	–
				Critical Child lookahead	cpn-cc-ins	–
			cpn-directed variant (dps)	EST-processor	dps-est-ins	–
				Weighted EST lookahead	dps-cwe-ins	–
				Critical Child lookahead	dps-cc-ins	–
	Dynamic		EST - blevel	EST-processor	dls-ins	–
		No Insertion		EST-processor	dls	DLS [24]
			dls with cc lookahead	Critical Child lookahead	dlscc	–
			EST	EST-processor	etf	ETF [11]
			etf with cc lookahead	Critical Child lookahead	etfcc	–

Cluster Scheduling	Dynamic Critical Path	Guided Load Balancing	Decreasing b-level	dcp-glb-bl	
			Earliest Time First	dcp-glb-etf	
		Adapted List Scheduling		dcp-ls-etf	
	Dynamic Critical Path (adjusted)	Guided Load Balancing		dcpx-glb-etf	
		Adapted List Scheduling		dcpx-ls-etf	
	Dominant Sequence Clustering	Guided Load Balancing		dsc-glb-etf	
		Adapted List Scheduling		dsc-ls-etf	
		Guided Load Balancing (reorder)		dsc-glbro-etf	
	Linear Clustering	Guided Load Balancing		lc-glb-etf	
		Adapted List Scheduling		lc-ls-etf	
Algorithm Type	Clusterer (cluster scheduling)	Cluster Merger (cluster scheduling)	Intra-cluster Ordering (clustering scheduling)	Algorithm (code)	NA

Figure 1. Algorithm composition table of algorithms tested in the experiment

costs [12], where algorithms can be formulated as event based, where an event is the completion of at least one task.

b) DLS: DLS (Dynamic Level Scheduling) [33] defines an attribute called the dynamic level which is the task's bottom-level minus its EST, and chooses tasks which maximize it. It could be seen as a compromise between minimizing the earliest start time and maximizing the bottom-level. Its complexity including placement is $O(V^2)$.

B. Task Placement

1) Standard Task Placement: In list-scheduling, a standard processor choice for an examined task is one that allows the EST for the task. An EST can be found for either inserting or appending the task to the schedule. Appending the task means to schedule it behind all other tasks previously scheduled to its processor, at the earliest time when its incoming data are ready and the processor has become indefinitely idle. With this method, the only processor choice is between the one which sends the latest finishing communication to the task and the one which finishes the earliest. Its complexity is $O(V \log P + E)$ [31].

Using insertion to find a start time for a task is to insert it in an idle gap between the execution times of previously scheduled tasks in the schedule. This has a higher complexity of $O(V^2)$ but allows earlier start times to be found. Finding an EST with insertion for each task requires examining the time slots of all tasks scheduled before it to find gaps, which takes $O(V^2)$ time, and finding the data ready times takes the same time as scheduling without insertion. Many list scheduling algorithms use the insertion strategy to find ESTs, in particular ones with static priorities. However, insertion is like revising the order of examination of the tasks and it proves the existence of better ones that could have produced the same result but just by appending the tasks. It could be argued that the appropriate task selection would create less opportunities for insertion, making the technique less useful. The ETF algorithm is an example where insertion cannot be applied at all, because no task should be able to start earlier than any previously scheduled ones, otherwise it would have been scheduled before them.

The Critical Path on a Processor algorithm mentioned in section III-A1 also uses an approach that assigns all tasks on the critical path to the same processor, and otherwise assigns tasks to their EST processors in the normal way. This does not increase complexity.

2) Lookahead Placement Schemes : Apart from the standard placement scheme of assigning each task to its earliest-start-time processor, there exist others which could perform better but with higher time complexities. They are generally lookahead techniques, which try to gauge the impact of processor choices for an examined task beyond its own start time and on the tasks scheduled after it.

a) Critical Child Lookahead Technique in DCP: DCP (Dynamic Critical Path) [20] is a clustering algorithm that is similar in format to list-scheduling and has sometimes been called one, but it does not schedule to a limited number of processors and a conversion for it is not straightforward. It uses a lookahead technique in processor selection which is easily adapted to be used in list scheduling in the following form. To evaluate a processor choice, it considers the task's own start time on that processor as well as the earliest start time of its critical child if it were to be scheduled after the task is scheduled to that processor. The critical child is one that is on the longest path through the task and determines the task's bottom-level, meaning that it has the highest bottom-level plus the cost of the communication from the task to itself, out of all the children. It is possible that the critical child has other unscheduled parents, so only an estimate of its earliest start time is used which is taken by ignoring the missing communications from unscheduled parents. In selecting a processor, the algorithm minimizes the sum of the start time of the examined task on the selected processor and the EST of its critical child if the task is scheduled to that processor. Its time complexity is $O(E+VP)$ when used without insertion, and $O(V^2)$ when used with insertion [38].

This lookahead placement scheme is tried with priority schemes from the algorithms ETF and DLS, with appropriate changes to them. Both algorithms' priority schemes are dependent on their placement schemes, so when their placement scheme is changed to critical child lookahead, their priorities have to be redefined around the critical child lookahead scheme. For ETF's priority scheme, this means to apply the critical child lookahead to all candidate tasks to be scheduled, and to select one that gives the minimum EST sum. In short, the EST sum from critical child lookahead replaces the normal EST. The same applies to DLS's priority scheme, the EST sum replaces the normal EST in its metric $blevel - EST$. These are only done without insertion, and both have been implemented with $O(VWP(V+E))$ complexity [38].

b) Children-Latest-EST and Children-Weighted-EST Lookahead Techniques: Several lookahead schemes for list scheduling were suggested in [5]. They are for heterogeneous systems but are useful in homogeneous systems too. One of them is the Children-latest-EST lookahead which places the examined task to a processor that minimizes the latest of all ESTs of the task's children. Another one is the Children-weighted-EST lookahead which minimizes the sum of the ESTs of the children, each weighted with the child's bottom-level. In the same way as in the critical child lookahead, unscheduled parents are ignored when calculating the children's ESTs. These lookahead schemes take $O(V(E+VP))$ time when used without insertion, and $O(V^3)$ time when used with insertion. Additional levels of lookahead applied recursively were found not to have much potential [5].

IV. Cluster-Scheduling

Clustering algorithms are largely algorithms that produce schedules for an unlimited number of processors, each one executing a cluster of tasks. Their results can be adapted to a limited number of processors by merging the clusters. Cluster-scheduling involves three steps. The first one is a clustering algorithm which clusters tasks together that

communicate extensively, for them to execute on the same processor. The cluster-merging step merges clusters until there is one processor for each cluster. The task-ordering step provides an ordering for tasks within clusters, which finishes the schedule. The following describes some methods that exist for each step, including all the ones used in the experiments.

A. Clustering

a) LC: LC (Linear Clustering) [18] was an early technique. It assigns the critical path to an individual cluster and removes the nodes and incident edges of the path from consideration, and this is repeated using the longest path through the unexamined nodes of the task graph at each iteration, continuing until all nodes have been assigned to a cluster. LC has complexity $O(V(E+V))$.

b) DSC: DSC (Dominant Sequence Clustering) [40] is a more complex algorithm. It uses the critical path like LC, but also more heuristics and more complex methods than LC. Dominant sequence is another name for the allocated critical path with pseudo-edges included.

Described as scheduling to an unlimited number of clusters, it schedules tasks in a topological order and minimizes under certain constraints the start time of each one at the point it is scheduled, through the choice of the cluster to be scheduled to and by sometimes relocating its parents. Task priorities are defined as the sum of their top and bottom levels, which is the length of the longest path through them. At every iteration, the free task with the highest priority is selected and scheduled to one of two cluster choices: a unit cluster by itself, or (appended to) the cluster of its parent sending the latest finishing communication to it (the constraining parent), depending on where it can start earlier. If placed to the cluster with its constraining parent, its other parents are then examined in decreasing order of their communication arrival times and moved to the same cluster if the resulting data ready time for the task is able to be reduced by the zeroed communication (despite the parent's own delayed start time) and they do not have other children (as to not cause cascading changes). The top-level here is updated for relocated tasks which increases complexity from the original to $O(V^2)$. DSRW in the algorithm is not used in our study because it did not improve performance [38].

c) DCP: DCP (Dynamic Critical Path) [20] is an algorithm with complexity $O(V^3)$. It also uses the sum of top and bottom levels as the task priority, both are allocated-levels with pseudo-edges and updated in every iteration. At each iteration, the algorithm schedules a task with the highest priority (this would not be in topological order). There would be multiple such tasks with the same priority as they lie on the same path which is longest through all of them, so the highest one (i.e. with the lowest top-level) is chosen. To keep candidate clusters to only those which would allow communications to be zeroed, with the exception of a new cluster which guarantees a start time at the data ready time, they are set to only the ones containing the scheduled parents or children of the chosen task, in addition to a new cluster. In choosing the best cluster to schedule to, the algorithm uses the sum of the ESTs of the task and its critical child when tentatively scheduled to the same candidate clusters. The critical child is one that is on the longest path through the task, out of all the unscheduled children. The algorithm uses squeeze when finding an EST for the examined task (not its critical child), and only if it is on the critical path. Squeezing is to insert the task anywhere on that cluster, pushing forward the execution times of other tasks if needed, on the condition that no tasks are delayed so much that the schedule length is increased (i.e. the EST of the delayed task plus its bottom-level is less than the current schedule length). The original algorithm also tries to save clusters by fitting non-critical tasks into existing clusters if possible, avoiding new clusters even if those tasks have to start later on clusters that do not help them. This would save processors if DCP's schedule to unlimited processors is to be used directly, but it does not benefit cluster-scheduling, where tasks should only be clustered if they have to, so this feature is not used here.

B. Cluster-Merging

1) Load balancing approaches: Define the load of a processor as the total computation cost of tasks in the clusters assigned to it. An easy approach to cluster-merging is to balance the processor load, examining clusters in decreasing order of their total computation costs and assigning each one to the least loaded processor, merging it with clusters assigned to the same processor. This can be called load-balancing. Its complexity is $O(C \log C + V)$.

Load balancing is not reliable when the loads in consideration start at different times in parallel (because of data ready time). For example, if processor A has load 2 units occupying times 0-2 units, processor B has load 1 unit occupying times 2-3 units, a cluster Z with a series of tasks that start at time 3 units would start earlier on processor A even though A has higher load, because A's load does not coincide with Z's load (in terms of earliest start times), whereas on B, B and Z's loads would delay each other. Guided Load balancing (GLB) [30] maps clusters to processors in increasing order of their earliest start times so that processor loads at any time are more representative of how they would affect the start time of the mapped cluster, since they execute at similar times. A schedule has to be created from the clusters to determine their start times. Normally the tasks would be ordered in decreasing bottom-level to obtain the schedule, but they could also use some of their original ordering from the clustering algorithm, so both versions are used (The one which completely reorders tasks is under the code "glbro"). In the version implemented in this paper, the schedule is updated at each iteration so it has complexity $O((V+E)C))$.

2) Merging by list-scheduling adaptation: [32] uses a technique that directly extends list-scheduling. It examines tasks in a priority order as in list scheduling, but assigns each one with its entire cluster to the processor which minimizes the resulting schedule length when it is merged with other clusters assigned to that processor. If a task's cluster has already been assigned to a processor, then it

follows that assignment. A schedule has to be constructed to evaluate each potential processor choice. The algorithm has complexity$O(PC(V+E))$.

C. Task-Ordering with Processor Allocations

The strategies used to order tasks in list-scheduling are applied here also. One of them is to use the bottom-level, and its allocated version since the processor assignments are known. This has the same complexity as when used in list-scheduling, which is $O(E+V\log V)$.

A method called Ready Critical Path [41] chooses to schedule tasks which can start the earliest, so that gaps are avoided, and breaks ties with the allocated bottom-level. It is the same strategy as in ETF list scheduling. As with ETF its complexity can be $O(V(\log W+\log P)+E)$ [29].

V. Experimental Setup

In this section we describe the workload and setup of the simulation experiments, whose results are discussed in the next section.

The input graphs are generated with a variety of structures and methods of assigning weights. The majority ranges from 50 to 500 nodes, a few structures have up to 1151. The graphs have Communication to Computation Ratios (CCR) of 0.1, 1, and 10. The CCR is defined as the ratio of the sum of all communication costs to the sum of all computation costs, and measures the importance of communication delays in the workload. The graphs are tested with processor counts of 2, 8, 32, 128, 256, and 512, or as high as that goes without exceeding the node count of the graph. There are in total 1600 graphs tested.

A. Graph Structures

Table VIII lists the employed graph structures.

Table VIII
Graph structures

Category	Graph (with number of)	Reference
regular structures	in-trees (84), out-trees (84), series-parallel (216), pipeline (15), fork (60), join (60), fork-join (84)	–
random graphs	Erdos-Rényi (216), layer by layer (216), fan-in fan-out (216), intersecting orders (216)	[9]
application graphs	FFT (15), Gauss (9), Cholesky (15), Stencil (15)	–
application graphs	CyberShake (15), Epigenomics (15), Montage (15), Inspiral (15), Sipht (15)	[4]
application graphs	matrix solver (3), robot control (3), SPEC fpppp (3)	[35]

The graph structures used include random structures and regular structures, as well as those from real workloads of parallelised applications. Regular structures include fork-join graphs , in-trees and out-trees, series-parallel (SP) graphs, and pipeline graphs. Graphs of real applications include ones derived from stencil computing, the fast Fourier transform, Gaussian elimination, and Cholesky factorisation. In addition, specialised parallel computing applications were taken from the graph set described in [35] and scientific workflows described in [4].

Random graphs are generated using four different methods (aggregated in [9] and discussed in more detail): generation by matrix (Erdos-Rényi), fan-in fan-out, layer by layer [35], and intersecting total orders. Generating random graphs "by matrix" involves committing the tasks to a topological order and then going through the triangular matrix of possible edges between them, creating each one with an identical and set probability. Generating graphs "layer-by-layer" involves grouping the tasks into ordered layers, and then adding edges from each task to tasks in the layers below it, with a set probability for each edge. The "fan-in fan-out" method starts with a source task and then goes through a probabilistic sequence of operations of either adding several children to an existing task or adding a task with several existing parents, all within the limits of maximum and in and out degrees. Generation by "intersecting total orders" intersects randomly generated total orders to produce a partial order, and then applies a transitive reduction on that partial order before creating a graph from it.

Using these different methods of generating random graphs serves to balance out certain graph characteristics that each method tends to produce. For example, the matrix method will add more edges to and from tasks in the middle of its topological order because they have more possible edges, creating a denser middle. Another case is that no transitivity exists in graphs produced by intersecting total orders, because of applying the transitive reduction to them (otherwise there are too many), while many transitive relations are produced by the matrix method.

B. Graph Weight Generation

For node and edge weights, the generation of the graphs involves both random weights, and regular weights with respect to the graph structure, where they are applicable. Three distributions were used in the generation of random weights: a uniform distribution, a Gamma distribution, and a bimodal distribution made from two Gamma distributions (Table IX). Gamma distributions give non-negative values and a right skew, this resembles approximations of real workload weight distributions [22].

Table IX
Weight generation parameters

Distribution	Range	Modes
Uniform	10 - 100	–
Gamma	100 - 300	~200
Bimodal (2 Gammas)	50 - 250	~100, ~200

Regular weight type	Graph structures
Uniform node weights	stencil, FFT, pipeline
Uniform edge weights	stencil, FFT, pipeline, Cholesky, Gauss
Weights from nature of program	Cholesky, Gauss, trees

The uniform distribution spans one order of magnitude, so its weights are diverse. In the Gamma distribution, weights are more concentrated. The bimodal distribution represents a situation where there are two task types with different weight ranges.

Two ways were used to generate edge weights. One generates them as the node weights have been generated. Another one uses the weights of incident nodes to influence the weight of the edge, with the assumption that the volume of computation has an effect on the volume of inputs and outputs involved. Here this is done by first generating a value random value R (generated with same method as node weight), and then multiplying it by the sum of the weights w_h and w_t of the head and tail node, respectively, raised to a fractional power ($\frac{1}{2}$) to make its effect weaker, hence $c_{th} = R(w_h + w_t)^{\frac{1}{2}}$. Edge weights are scaled at the end to give the required CCR.

All combinations of these methods were included in generating the graphs. In addition, the following regular weight patterns were applied (Table IX). Stencil, FFT, and pipeline graphs all have uniform computation costs and uniform communication costs, based on their actual algorithms. Cholesky factorization and Gaussian elimination graphs have non-uniform node weight patterns based on their algorithms, and uniform communication costs. Trees, in addition to being generated with random weights, also have versions where their weights increase towards the root (the source for out-trees and sink for in-trees), and each node has a cost equal to the computation cost of the rest of its subtree. This represents a natural pattern in tree workloads where data is continually divided or combined then reprocessed, like the combine phase in a divide and conquer algorithm, e.g. mergesort.

VI. Experimental Evaluation

The performances of the algorithms on each problem (graph and processor count for limited processors) are measured with the lengths of their schedules normalised to the length of the best schedule found by all investigated algorithms for that problem (the normalised schedule length - NSL), which keeps the results on a consistent scale and highlights differences between the performances relative to each other. The boxplots used are Tukey boxplots which means the whiskers are the lowest and highest data points within 1.5 times the interquartile range from the boxes. The plots are colour coded to help with readability. Plots of list-schedulers are coloured while those of cluster-schedulers are greyscale. List-schedulers with the same task priority share the same colour and cluster-schedulers with the same clusterer have the same greyscale shade.

Due to space limitations, we present representative and interesting results in following, please refer to ([38]) for extensive results. While the comparison focuses on the quality of the produced schedules (i.e. their schedule length) we also measured and present the runtimes of the algorithms in section VI-A4 to allow some evaluation of cost-benefit trade-off, complementing the given runtime complexities of the algorithms.

A. Scheduling to Limited Processors

Figures 2, 3, and 4 show boxplots for results on all random, real-application, and series-parallel (SP) graphs with CCRs 0.1, 1, and 10 respectively. The data points are normalised schedule lengths for the algorithms on problems in those graph sets. Random, real-application, and SP graphs can be viewed as the general set that provides diversity and realism. By themselves, their results also have similar trends. Problem instances for random graphs outnumber the other types by roughly 2 to 1. The algorithms are close in performance at low CCR, but become more varied as CCR increases.

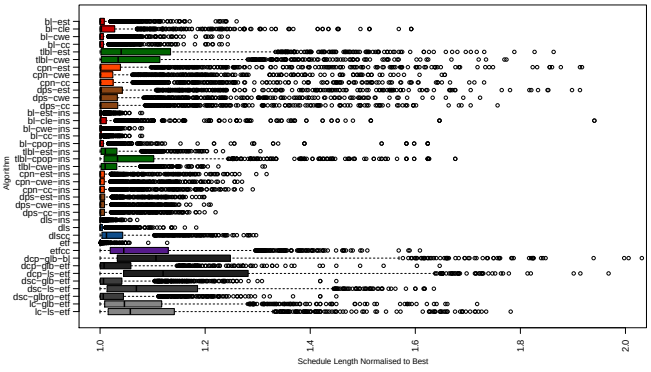


Figure 2. Boxplots for results on all random, real-application, and SP graphs with CCR 0.1

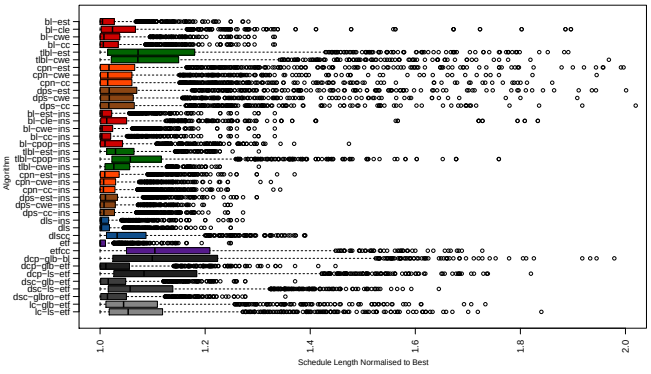


Figure 3. Boxplots for results on all random, real-application, and SP graphs with CCR 1.0

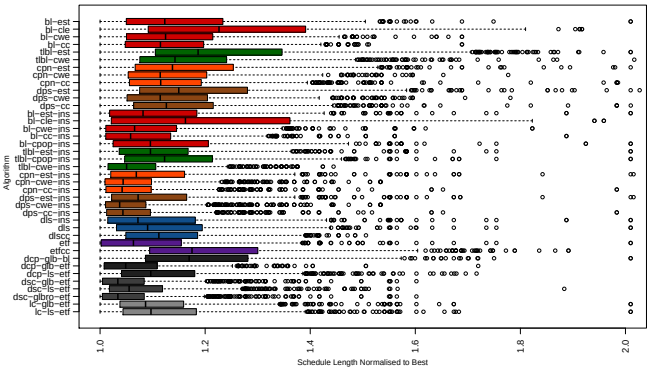


Figure 4. Boxplots for results on all random, real-application, and SP graphs with CCR 10.0

Figures 5 and 6 show performance profiles for the algorithms on these graph types with CCRs 0.1 and 10 respectively. Performance profiles were proposed as a means to

compare optimisation algorithms [11]. They plot the fraction of all results (y-axis) equal to or better than an NSL against that NSL (x-axis). Performance profiles compactly show for which fraction of the results an algorithm achieves a certain quality or overhead compared with the best algorithm for each input instance. A similar colour scheme to the boxplots are used. The point at 1 NSL shows the proportion of results by an algorithm that match the best (so they are best or tied-best). The profiles sometimes overtake each other, which shows that the overtaking algorithm has less results that are better but also less that are worse, at the point of overtake, for example, a few cluster-schedulers overtake non-insertion list-schedulers at around the 90th percentile, having better results for their last 10 percentiles. Most algorithms have almost all of their results better than a 2 NSL. The plots are truncated at 1.86 NSL for readability, a few profiles do not finish at that point but the difference is small and they belong to overall bad performers.. The algorithms with extreme outliers are the worst performers in general.

Let us now discuss the results, addressing the experimental aims stated in section II-C.

1) List-Scheduling vs Cluster-Scheduling: List-scheduling is better than cluster-scheduling at CCRs of 0.1 and 1 as can be observed in the boxplots and performance profiles of figures 2 to 6. The relative performance of cluster-scheduling improves with CCR. At a CCR of 10, certain cluster-schedulers are the best out of all schedulers but are still only marginally better than list-schedulers using lookahead (cpn/dps with cwe/cc with insertion), which are the best list-schedulers at that CCR. In the performance profiles for CCR 10 results, shown in figure 6, a cluster-scheduler (dsc-glb-etf) occupies the top of the band of profiles for a large range that is between its 25th and 95th percentiles, having the lowest NSL for almost all percentiles in that range. However, there are lookahead list-schedulers (cpn/dps with cwe/cc with insertion) which practically perform the same, following its profile very closely.

At high CCR, some cluster-schedulers (dsc with glb/ls with etf) are particularly effective at fork and join structures, which make up trees and series-parallel graphs. List-schedulers have difficulty with those unless they use lookahead techniques which are very helpful for join structures. Figures 7 and 8 show results for in-trees at 1 CCR and 10 CCR. The advantage for cluster-schedulers only comes at 10 CCR. Although the under-performing list-schedulers can improve by scheduling reversed in-trees as out-trees and reversing the schedule back, they would still be worse. This cluster-scheduler advantage is much smaller on out-trees (shown in figure 9), series-parallel graphs (figure 10), and fork-joins. In all cases cluster-scheduling is worse for lower CCRs.

2) List-Scheduler Comparisons: Refer again to figures 2, 3, and 4, for results on general graphs. At lower CCRs, the best schedulers are dynamic priority list-schedulers, in particular "dls-ins" (DLS with insertion) and ETF. ETF has lower complexity than "dls-ins" and is also slightly better. For CCRs 1 and 10 this is visible from the boxplots. For CCR 0.1 it can be seen from the performance profile in figure 5 that ETF has the highest number of best results by a wide margin.

ETF becomes inconsistent at CCR 10, it still has the highest number of best results which can be seen in figure 6, but its results are spread out with its higher percentiles being worse than those for list-schedulers using lookahead. Static priority lookahead list-schedulers are the best list-schedulers at high CCR considering their average and worst case performances.

No particular combination of techniques stands out very much. The following can be noted about specific techniques.

a) Task Priority: Bottom-level ("bl") is the best task priority at 0.1 and 1 CCR. However, at a CCR of 10, the best list-schedulers use variants of CPND ("cpn", "dps"), which are noticeably better than bottom-level priority. This is only with list-schedulers that use insertion, CPND and bottom-level perform similarly at 10 CCR if insertion is not used.

Generally, $tlevel + blevel$ is the least effective list-scheduling priority. It adds the top-level to its priority metric, which is the opposite of what ETF and DLS do, since top-level roughly corresponds to the task's start time. However, it sometimes does well when not used with the critical-path-on-processor placement technique like in its original algorithm, such as on series-parallel graphs with high CCR, shown in figure 10.

b) Insertion: At 0.1 and 1 CCR (figures 2 and 3), the performance increase from using insertion compared to not using insertion is small for their best and average cases. Their medians are practically the same. For a large fraction of results at these CCRs, list-scheduling without insertion gives good performance that cannot be improved upon. However, list-scheduling with insertion always gives much better worst case performance. Insertion's benefit increases with CCR. At CCR 10 (figure 4), it has noticeably better average case performance. It also seems to benefit schedulers with CPND task priority more than it does for ones with bottom-level priority.

c) Lookahead: As with insertion, simpler schedulers already perform well at low CCRs so there is not much room for improvement from using lookahead at 0.1 and 1 CCR (figures 2 and 3). At these CCRs, lookahead does not improve worst case performance either. However, its benefit increases with CCR and it becomes useful at 10 CCR, where it gives the best list-schedulers.

Out of the lookahead techniques used, critical-child lookahead ("cc") and children-weighted-est lookahead ("cwe") are the ones which are effective. Performance is similar between them, with children-weighted-est being slightly better. The children-latest-est lookahead ("cle") is much less effective, although it was shown to give good performance on heterogeneous processors for which it was created [5]. Lookahead does not work well when used with dynamic priority list-schedulers ("dlscc", "etfcc"). The critical-path-on-processor placement scheme ("cpop") is also shown to be ineffective.

Lookahead techniques are especially good on join structures, as they were designed for. This can be seen in results

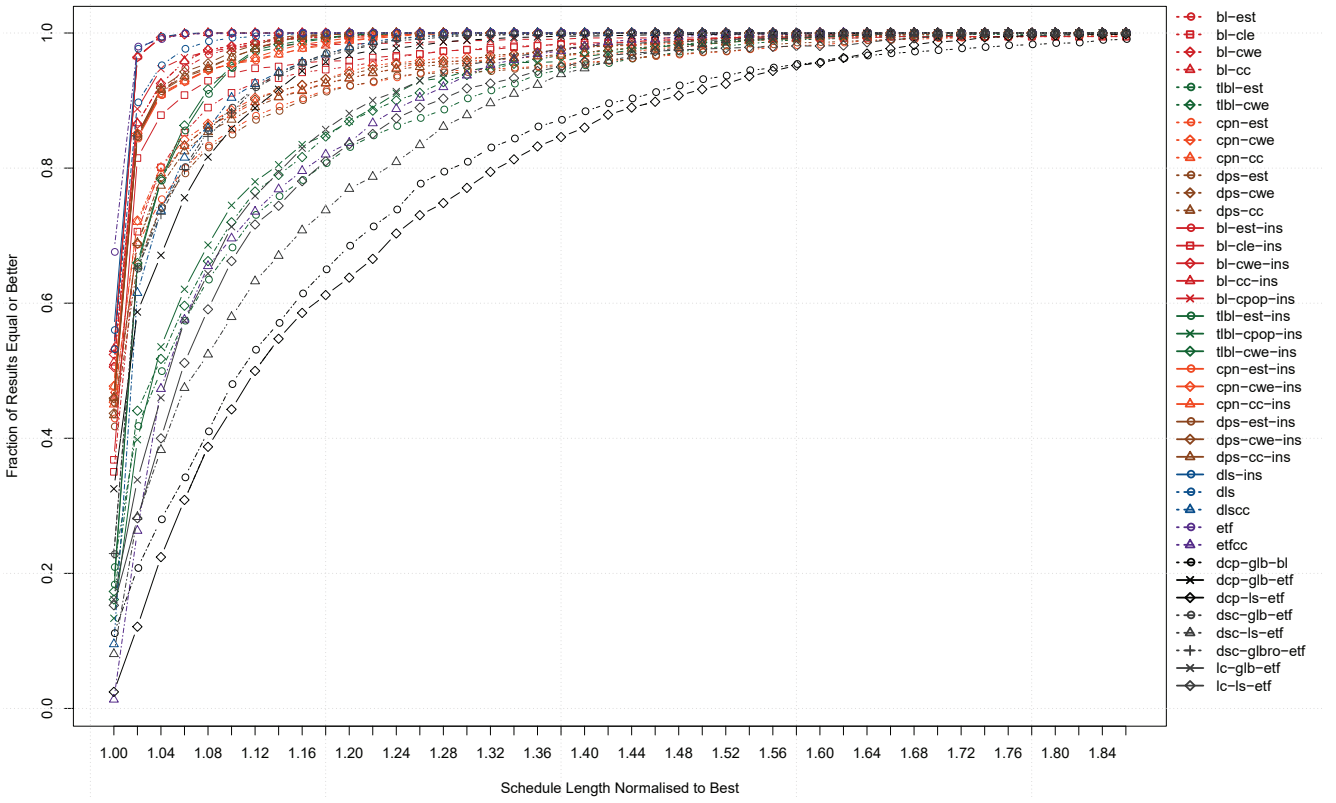


Figure 5. Performance profiles for results on random, real-application, and SP graphs with CCR 0.1

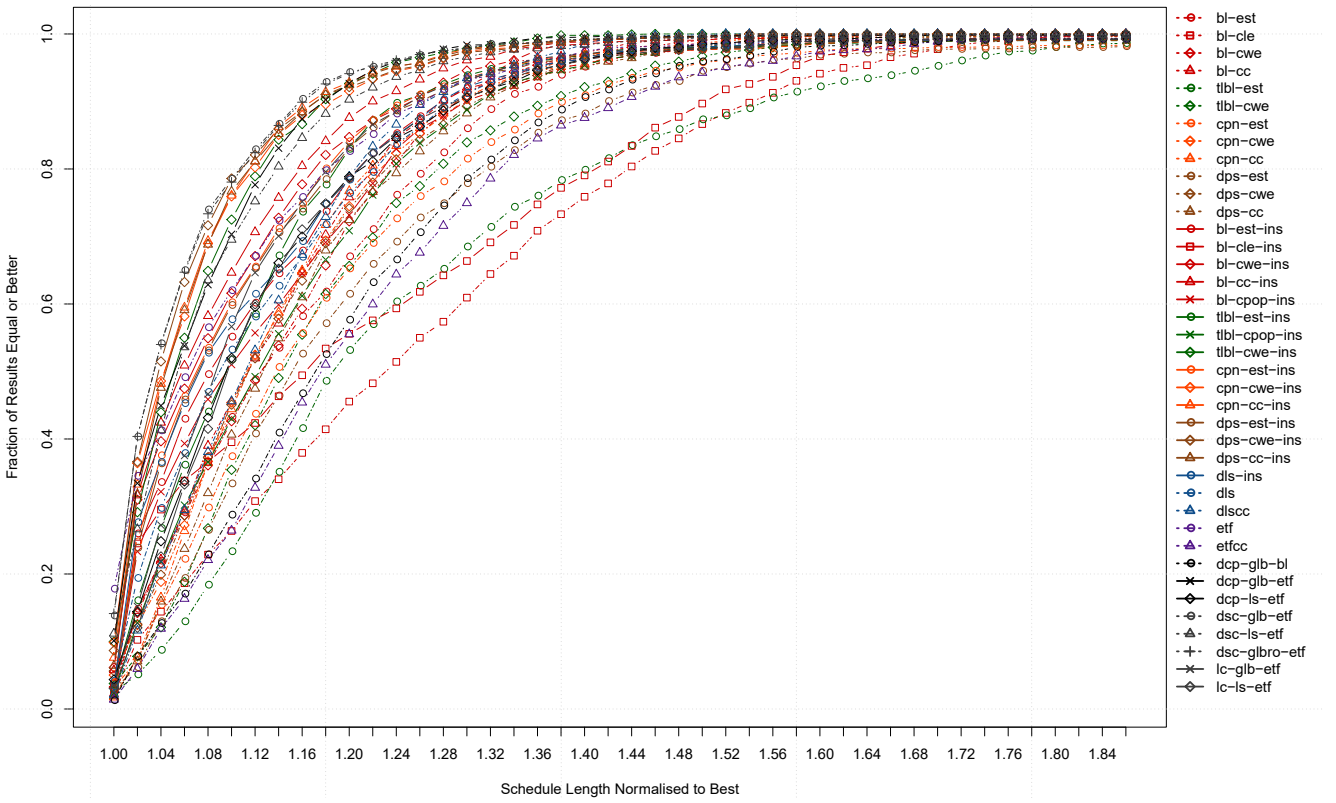


Figure 6. Performance profiles for results on random, real-application, and SP graphs with CCR 10.0

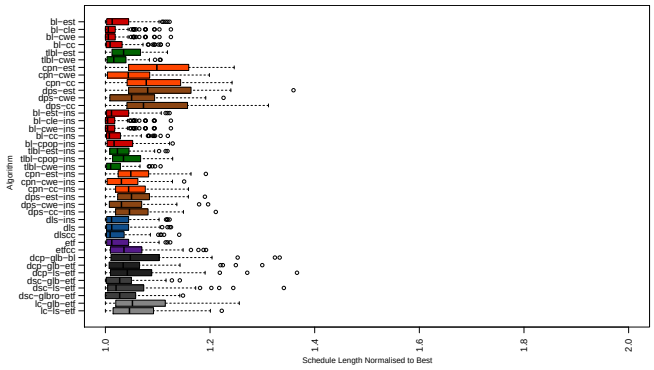


Figure 7. Boxplot for results on in-trees with CCR 1.0

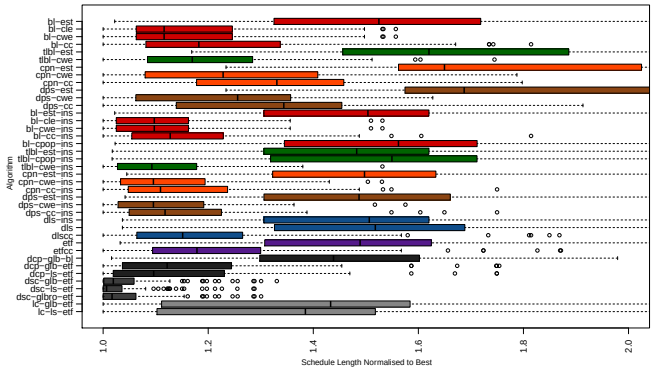


Figure 8. Boxplot for results on in-trees with CCR 10.0

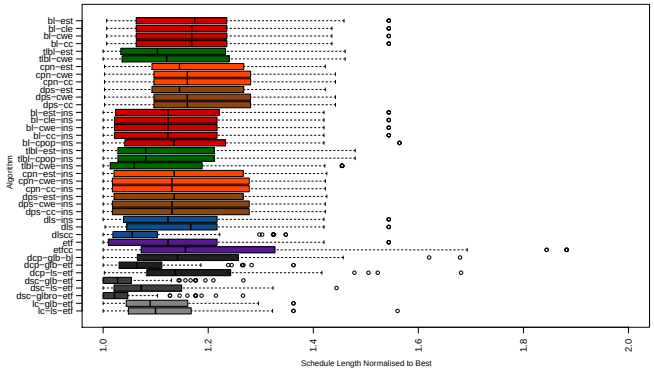


Figure 9. Boxplot for results on out-trees with CCR 10.0

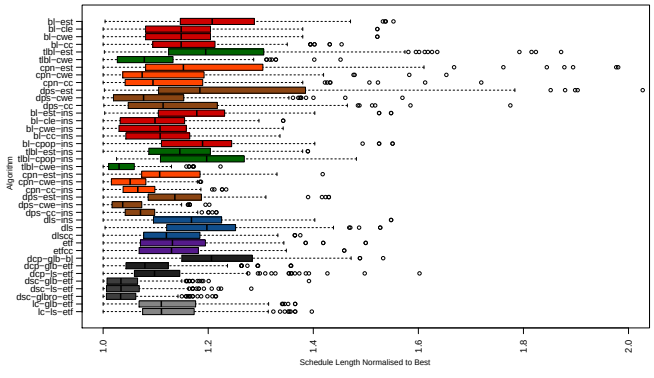


Figure 10. Boxplot for results on series-parallel graphs with CCR 10.0

for in-trees (8) and series-parallel graphs (10).

3) Cluster-Scheduler Comparisons: The DSC clustering algorithm gives the best cluster-schedulers, although it is seen in the next section that other algorithms can give better clusterings as schedules to unlimited processors. DSC is also particularly good relative to other algorithms on in-trees (8), because of its technique of relocating parents for join structures when the parents have only one child. Generally, DCP is close second in performance and LC is farther behind.

Merging clusters with GLB is better than with adapted list-scheduling, when both are used with EST ("etf") task ordering. This difference is actually more significant than between the two best clusterers, DSC and DCP, but reduces with increasing communications. Merging by list-scheduling seems to work better with LC. For GLB, there is minimal difference between using task orders from the clustering step ("glb") and reordering the tasks ("glbro"). EST task ordering is a significant improvement over bottom-level ordering and this difference is the most significant of all three stages.

4) Runtimes: Figure 11 shows typical runtimes for the algorithms on graphs of node size 500. Runtimes more than 1000 milliseconds are rounded to 3 significant figures. The runtimes of clustering algorithms have also been included, to separate them from those of later stages (merging and ordering) which can be much longer. Our implementations of the algorithms may deviate from the complexities listed (to reduce implementation effort), and significant ones are marked with asterisks. As can be seen, the runtimes differ very significantly across the algorithms. Cluster-merging can be particularly slow because of the high complexity methods (cluster-merging) involved, and the large numbers of clusters generated by clustering algorithms (DCP has been adjusted so that it does not try to save clusters at all). However, the best cluster-schedulers are not the slowest ones and have comparable speed to lookahead list-schedulers. Dynamic list-schedulers are the most effective in terms of the performance achieved with their runtimes.

Algorithm	Time (ms)	Algorithm	Time (ms)	Algorithm	Time (ms)
bl-est	2	bl-cc-ins*	70	etfcc	639
bl-cle*	53	bl-cpop-ins	5	dcp-glb-bl*	618
bl-cwe*	59	tlbl-est-ins	6	dcp-glb-etf	589
bl-cc	18	tlbl-cpop-ins	8	dcp-ls-etf	14200
tlbl-est	3	tlbl-cwe-ins*	338	dcpx-glb-etf	624
tlbl-cwe*	54	cpn-est-ins	5	dcpx-ls-etf	13800
cpn-est	2	cpn-cwe-ins*	339	dsc-glb-etf	145
cpn-cwe*	53	cpn-cc-ins*	70	dsc-ls-etf	6800
cpn-cc	20	dps-est-ins	4	dsc-glbro-etf	144
dps-est	2	dps-cwe-ins*	333	lc-glb-etf	2090
dps-cwe*	50	dps-cc-ins*	70	lc-ls-etf	2530
dps-cc	19	dls-ins	296	dcp	344
bl-est-ins	8	dls	6	dcpx	318
bl-cle-ins*	332	dlscc	1160	dsc	2
bl-cwe-ins*	329	etf*	6	lc	121

* Non-optimal implementation with impact on runtime

Figure 11. Algorithm runtimes in milliseconds

B. Scheduling to Unlimited Processors

We now compare the algorithms' performance when targeting an unlimited number of processors, which is the natural element of clustering algorithms. Scheduling to unlimited processors is also addressed here as clustering. The list-schedulers are made to schedule to an unlimited

number of processors (by setting the number of available processors to equal the number of tasks), and are compared in this regard with the clustering algorithms by themselves (so there are no merging or ordering steps). All the clusterers used in this study also work by producing schedules to unlimited processors. Figures 12 and 13 show boxplots for the clustering results on random, real-application, and SP graphs with CCRs 1 and 10 respectively. The clusterers are better here, although the lookahead list-schedulers are also very good and list-schedulers in general are not far behind. LC is the simplest method and does not do well because it cannot reduce multiple incoming or outgoing communications on one node, which is especially bad at high CCR. The results for CCR 0.1 (not shown here) are all extremely close to each other and to the lower bound. Scheduling without communication cost to an unlimited number of processors has easy optimum solutions and low CCR resembles it. For clustering, the only benefit that insertion brings to list-scheduling is fitting more tasks onto each processor (cluster) which avoids communications when communication costs are high, otherwise it gives no advantages.

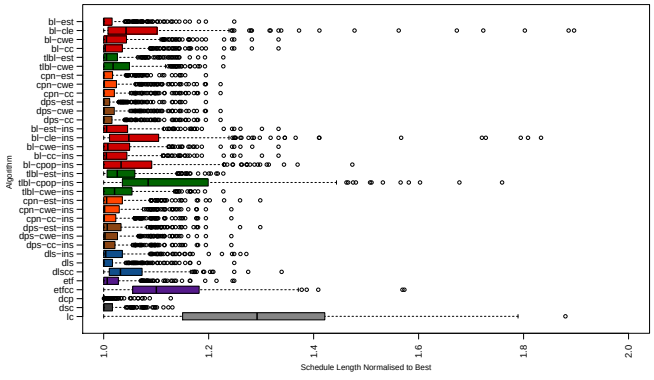


Figure 12. Boxplot for clustering results on random, real-application, and SP graphs with CCR 1.0

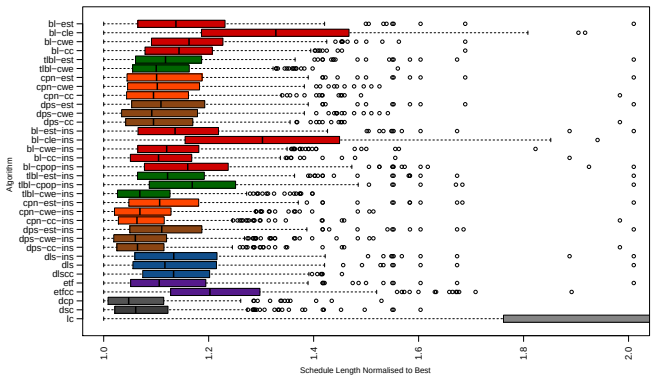


Figure 13. Boxplot for clustering results on random, real-application, and SP with CCR 10.0

DSC is optimal for fork and join graphs [40], DCP performs well on them too. In list-scheduling, lookahead techniques perform very well for join graphs as they have been designed for, and on fork graphs the $tlevel + blevel$ priority gives optimal list-schedulers because it uses edge weights while others do not, and it is similar to DSC in that situation. DSC is close to optimal on fork-join graphs. Just as in scheduling to limited processors, clustering algorithms and lookahead list-schedulers have an advantage on in-trees. Only clustering algorithms can give the best performance on out-trees. The high CCR sets the algorithms apart since clustering is about managing communications and reducing them, and clustering algorithms do not have an advantage at high CCR only because of their approach, like it is for cluster-scheduling to limited processors.

VII. Conclusion

This study set out to compare list-scheduling and cluster-scheduling. To do this, representative algorithms were implemented and compared in a large experimental study, and a systematic categorisation of algorithms was made based on their modular nature, with unpublished algorithms included in the experiment as combinations of modules, and the modules assessed individually. Cluster-schedulers start with reducing communications, and it is demonstrated that their performance increases with communication volume. However, the best cluster-schedulers are only marginally better than list-scheduling at a communication to computation ratio (CCR) of 10, and at lower CCRs list-scheduling is better. Cluster-schedulers have a more obvious advantage over list-scheduling at high CCR on fork and join structures which can be made extreme on trees. ETF is the best scheduler at low CCR while cluster-schedulers along with lookahead list-schedulers perform better at high CCR. When list-schedulers are compared with clustering algorithms by themselves for their ability to schedule to an unlimited number of processors, clustering algorithms are shown to be better in that case for their specific purpose.

References

[1] Ishfaq Ahmad, Yu-Kwong Kwok, and Min-You Wu. Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors. In *Parallel Architectures, Algorithms, and Networks, 1996. Proceedings., Second International Symposium on*, pages 207–213. IEEE, 1996.

[2] H. Arabnejad and J. G. Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):682–694, March 2014.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. H. Su, and K. Vahi. Characterization of scientific workflows. In *2008 Third Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, Nov 2008.

[5] Luiz F Bittencourt, Rizos Sakellariou, and Edmundo RM Madeira. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 27–34. IEEE, 2010.

[6] OpenMP Architecture Review Board. OpenMP Application Program Interface. Technical report, 07 2013.

[7] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou, and Wei Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. In *Grid Computing*, pages 73–84. Springer, 2008.

[8] Edward Grady Coffman and John L Bruno. *Computer and job-shop scheduling theory*. John Wiley & Sons, 1976.

[9] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. Random graph generation for scheduling simulations. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[10] Mohammad I Daoud and Nawwaf Kharma. A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. *Journal of Parallel and distributed computing*, 68(4):399–409, 2008.

[11] D. Elizabeth Dolan and J. Jorge Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[12] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[13] T. Hagras and J. 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.*, pages 107–, April 2004.

[14] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *siam journal on computing*, 18(2):244–257, 1989.

[15] Shiyuan Jin, Guy Schiavone, and Damla Turgut. A performance study of multiprocessor task scheduling algorithms. *The Journal of Supercomputing*, 43(1):77–97, 2008.

[16] H. Kanemitsu, M. Hanada, and H. Nakazato. Clustering-based task scheduling in a large number of heterogeneous processors. *IEEE Transactions on Parallel and Distributed Systems*, 27(11):3144–3157, Nov 2016.

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

[18] Sung J Kim. A general approach to multiprocessor scheduling. Technical report, University of Texas at Austin, Austin, TX, USA, 1988.

[19] Yu-Kwong Kwok and Ishfaq Ahmad. Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures. In *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on*, pages 36–43. IEEE, 1995.

[20] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE transactions on parallel and distributed systems*, 7(5):506–521, 1996.

[21] R. Lepere and D. Trystram. A new clustering algorithm for large communication delays. In *Proceedings 16th International Parallel and Distributed Processing Symposium*, pages 6 pp–, April 2002.

[22] Yan Alexander Li, John K. Antonio, Howard Jay Siegel, Min Tan, and Daniel W. Watson. Determining the execution time distribution for a data parallel program in a heterogeneous computing environment. *J. Parallel Distrib. Comput.*, 44(1):35–52, July 1997.

[23] Jing-Chiou Liou and Michael A Palis. A comparison of general approaches to multiprocessor scheduling. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 152–156. IEEE, 1997.

[24] W. Liu, H. Li, W. Du, and F. Shi. Energy-aware task clustering scheduling algorithm for heterogeneous clusters. In *2011 IEEE/ACM International Conference on Green Computing and Communications*, pages 34–37, Aug 2011.

[25] Carolyn McCreary and Helen Gill. Automatic determination of grain size for efficient parallel processing. *Commun. ACM*, 32(9):1073–1078, September 1989.

[26] Carolyn L McCreary, AA Khan, JJ Thompson, and ME McArdle. A comparison of heuristics for scheduling dags on multiprocessors. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 446–451. IEEE, 1994.

[27] E. U. Munir, S. Mohsin, A. Hussain, M. W. Nisar, and S. Ali. Sdbats: A novel algorithm for task scheduling in heterogeneous computing systems. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 43–53, May 2013.

[28] Gyung-Leen Park, Behrooz Shirazi, Jeff Marquis, and Hyunseung Choo. Decisive path scheduling: a new list scheduling method. In *Parallel Processing, 1997., Proceedings of the 1997 International Conference on*, pages 472–480. IEEE, 1997.

[29] Andrei Radulescu and Arjan J. C. Van Gemund. Flb: Fast load balancing for distributed-memory machines. In *Proceedings of the 1999 International Conference on Parallel Processing*, ICPP '99, pages 534–, Washington, DC, USA, 1999. IEEE Computer Society.

[30] Andrei Radulescu and Arjan JC Van Gemund. Glb: A low-cost scheduling algorithm for distributed-memory architectures. In *High Performance Computing, 1998. HIPC'98. 5th International Conference On*, pages 294–301. IEEE, 1998.

[31] Andrei Rădulescu and Arjan J. C. Van Gemund. On the complexity of list scheduling algorithms for distributed-memory systems. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 68–75, New York, NY, USA, 1999. ACM.

[32] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

[33] Gilbert C Sih and Edward A Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE transactions on Parallel and Distributed systems*, 4(2):175–187, 1993.

[34] Oliver Sinnen. *Task scheduling for parallel systems*, volume 60. John Wiley & Sons, 2007.

[35] Takao Tobita and Hironori Kasahara. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5):379–394, 2002.

[36] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

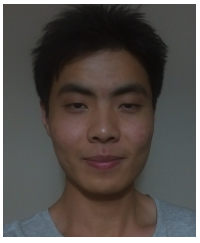
[37] G. Wang, H. Guo, and Y. Wang. A novel heterogeneous scheduling algorithm with improved task priority. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 1826–1831, Aug 2015.

[38] Huijun Wang. List-scheduling vs. cluster-scheduling in experimental comparison. Master's thesis, University of Auckland, Auckland, 1010, 2003.

[39] Min-You Wu and Daniel D Gajski. Hypertool: A programming aid for message-passing systems. *IEEE transactions on parallel and distributed systems*, 1(3):330–343, 1990.

[40] Tao Yang. *Scheduling and code generation for parallel architectures*. PhD thesis, Citeseer, 1993.

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



Huijun Wang received a Bachelor of Engineering (Honours) from the University of Auckland, New Zealand. He is studying a Master of Engineering at the University of Auckland, where his research interest is in task scheduling.

Oliver Sinnen graduated in Electrical and Computer Engineering at RWTH Aachen University, Germany and received his PhD from Instituto Superior Técnico (IST), University of Lisbon, Portugal. He is a Senior Lecturer in the Department of Electrical and Computer Engineering at the University of Auckland, New Zealand, where he leads the Parallel and Reconfigurable Computing Lab. Oliver authored the book "Task Scheduling for Parallel Systems", Wiley.