Low-Cost Task Scheduling for Distributed-Memory Machines

Andrei Radulescu and Arjan J.C. van Gemund, *Member*, *IEEE Computer Society*

Abstract—In compile-time task scheduling for distributed-memory systems, list scheduling is generally accepted as an attractive approach since it pairs low cost with good results. List scheduling algorithms schedule tasks in order of their priority. This priority can be computed either 1) statically, before the scheduling, or 2) dynamically, during the scheduling. In this paper, we show that list scheduling with statically computed priorities can be performed at a significantly lower cost than existing approaches, without sacrificing performance. Our approach is general, i.e., it can be applied to any list scheduling algorithm with static priorities. The low-complexity is achieved by using low-complexity methods for the most time consuming parts in list scheduling algorithms, i.e., processor selection and task selection, preserving the criteria used in the original algorithms. We exemplify our method by applying it to the MCP algorithm. Using an extension of this method, we can also reduce the time complexity of a particular class of list scheduling with dynamic priorities (including algorithms such as DLS, ETF, or ERT). Our results confirm that the modified versions of the list scheduling algorithms obtain a performance comparable to their original versions, yet at a significantly lower cost. We also show that the modified versions of the list scheduling algorithms consistently outperform multistep algorithms, such as DSC-LLB, which also have higher complexity and clearly outperform algorithms in the same class of complexity, such as CPM.

Index Terms—Compile-time task scheduling, list scheduling, dataflow graphs, distributed-memory multiprocessors.

1 Introduction

A key issue in obtaining performance from a parallel program is to efficiently map it to the target system. The problem is generally addressed in terms of task scheduling [4], [6], [33], [16], [27], where the tasks are the schedulable units of a program. As scheduling parallel applications have been proven to be NP-complete [7] heuristics are used. In order to be of practical use for large applications, scheduling heuristics must have a low-complexity. For shared-memory systems, it has been proven that even a low-cost scheduling heuristic is guaranteed to produce a performance that ensures linear speedup [10]. In the distributed-memory case, however, communication must be taken into account, which significantly complicates the problem. In this case, the scheduling problem remains a challenge, especially for algorithms where low cost is of principal interest.

Distributed-memory scheduling heuristics exist for both bounded and unbounded number of processors. Although attractive from a cost perspective, scheduling for an unbounded number of processors (e.g., DSC [33], EZ [26], LC [13], or TCS [19]) is not always applicable, because the required number of processors is usually not available. Hence, their application is typically found within the multistep scheduling methods for a bounded number of processors [25], [26], [32].

- *A. Radulescu is with Phillips Research Laboratories, Eindhoven, The Netherlands. E-mail: andrei.radulescu@phillips.com.*
- *A.J.C. van Gemund is with Delft University of Technology, The Netherlands. E-mail: a.j.c.vangemund@its.tudelft.nl.*

Manuscript received 23 Nov. 1999; revised 8 Mar. 2001; accepted 7 Dec. 2001.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 110989.

Apart from multistep methods, scheduling for a bounded number of processors can also be performed in a single step. Single-step approaches usually produce better results, yet at a higher cost. Scheduling for a bounded number of processors can be performed either using duplication (e.g., DSH [14], BTDH [3], or CPFD [2]), or without duplication (e.g., MCP [30], ETF [11], DPS [21], DLS [29], ISH [14], TSF [5], or GD [12]. Duplicating tasks results in better scheduling performance but significantly increases scheduling cost. Nonduplicating task heuristics have a lower complexity and still obtain good schedules. However, when compiling very large programs for large systems, the complexity of current approaches is often prohibitive.

An important class of scheduling algorithms for a bounded number of processors is list scheduling (e.g., MCP [30], DPS [21], CPND [17], ETF [11], ERT [18], or DLS [29]). It has been shown that list scheduling algorithms perform well at a relatively low cost compared to other higher-cost scheduling algorithms for bounded number of processors (e.g., ISH [14], or GD [12]) [12], [16], [25]).

In list scheduling, two approaches can be distinguished. The first approach is list scheduling with *static* priorities (LSSP) (e.g., MCP [30], DPS [21], HLFET [1], CPND [17], CPM [28], or WL [31]). In LSSP, the tasks are scheduled in the order of their previously computed priorities on the task's "best" processor. Thus, at each scheduling step, first the task is selected and afterwards its destination processor. Usually, if the performance is the main concern, the "best" processor is considered the processor enabling the earliest start time for the given task (e.g., MCP, DPS, or CPND). However, if the speed is given the emphasis, the selected processor is the processor becoming idle the earliest when the task is scheduled (e.g., CPM, or WL).

TABLE 1
Some Well-Known Scheduling Heuristics

Unlimited number of processors, no duplication	
DSC (Dynamic Sequence Clustering) by Yang & Gerasoulis [33]	$O((E+V)\log(V))$
EZ (Edge Zeroing) by Sarkar [26]	$O(E(E+V))$
TCS (Task Clustering and Scheduling) by Palis *et al.* [19]	$O(V(V\log(V)+E))$
LC (Linear Clustering) by Kim & Browne [13]	$O(V^2\log(V))$
Limited number of processors, duplication	
DHS (Duplication Scheduling Heuristic) by Kruatrachue & Lewis [14]	$O(V^4)$
BTDH (Bottom-up Top-down Duplication Heuristic) by Chung & Ranka [3]	$O(V^4)$
CPFD (Critical Path Fast Duplication) by Ahmad & Kwok [2]	$O(V^4)$
Limited number of processors, no duplication, LSSP	
HLFET (High Levels First with Estimated Times) by Adam *et al.* [1]	$O(V\log(V)+(E+V)P)$
CPND (Critical Path Node Dominate) by Kwok *et al.* [17]	$O(V\log(V)+(E+V)P)$
DPS (Decisive Path Scheduling) by Park *et al.* [21]	$O(V\log(V)+(E+V)P)$
MCP (Modified Critical Path) by Wu & Gajski [30]	$O(V^2(\log(V)+P))$
CPM (Critical Path Method) by Park *et al.* [21]	$O(V(\log(V)+P))$
WL (Weighted Length scheduling) by Yang and Fu [31]	$O(V(\log(V)+P))$
Limited number of processors, no duplication, LSDP	
ETF (Earliest Task First) by Hwang *et al.* [11]	$O(V(E+V)P))$
ERT (Earliest Ready Task) by Lee *et al.* [18]	$O(V(E+V)P))$
DLS (Dynamic Level Scheduling) by Sih & Lee [29]	$O(V(E+V)P))$
DCP (Dynamic Critical Path) by Kwok & Ahmad [15]	$O(V(E+V)P))$
Limited number of processors, no duplication, other	
ISH (Insertion Scheduling Heuristic) by Kruatrachue & Lewis [14]	$O(V\log(V)+(E+V)P)$
GD (Graph Decomposition) by Khan *et al.* [12]	$O(V^3)$
TSF (Task Selection First) by Djordjevic & Tosic [5]	$O(V(E+V)))$

The second approach is list scheduling with *dynamic* priorities (LSDP) (e.g., ETF [11], ERT [18], DLS [29], or DCP [15]). In this case, at each scheduling iteration, a ready task and its destination processor are selected at the same time. The selection is based on the priorities computed for pairs of a ready-task and a processor (e.g., the combination that produces the earliest start or finish time as in ETF and ERT, respectively). Because it has a more complex task and processor selection scheme, LSDP is able to produce better schedules than LSSP, however, at a significantly higher cost. In Table 1, we summarize the characteristics of some of the well-known task scheduling algorithms.

In this paper, we prove that any LSSP algorithm can be performed at a significantly lower cost compared to existing approaches. Existing LSSP algorithms, such as MCP or DPS, already have a low time complexity ($O(V\log(V)+(E+V)P)$, where V and E are the number of tasks and dependences in the task graph, respectively, and P is the number of target processors. Using our approach, we significantly reduce LSSP complexity to $O(V\log(P)+E)$, yet maintaining comparable performance. The cost is reduced by 1) considering only *two* processors when selecting the destination processor for a given task (proven to preserve the original processor selection criterion) and 2) maintaining a partially-sorted task priority queue in which only a *fixed* number of tasks is sorted.

We generalize our approach to be used for a particular class of LSDP algorithms, which includes algorithms such as DLS, ETF, and ERT. The LSDP time complexity is reduced even more, from $O(V(E+V)P)$ to $O(V\log(P)+E)$, again, with comparable performance.

This paper is organized as follows: Section 2 defines the scheduling problem and introduces some definitions used in the paper. Section 3 describes our approach to reduce LSSP time complexity; in Section 4, the extention to LSDP is described. Section 5 presents their performance in some particular cases. Finally, Section 6 concludes the paper.

2 Preliminaries

A parallel program can be modeled by a directed acyclic graph (DAG) $\mathcal{G}=(\mathcal{V},\mathcal{E})$, where $\mathcal{V}$ is a set of V vertices and $\mathcal{E}$ is a set of E edges. A vertex in $\mathcal{G}$ represents a task, containing instructions that execute sequentially without preemption. Each task $t\in\mathcal{V}$ is assumed to have a *computation cost* $T_w(t)$. The edges correspond to task dependencies (communication messages or precedence constraints) and have a *communication cost* $T_c(t,t')$. If two tasks t_1 and t_2 are scheduled on the same processor, $T_c(t_1,t_2)$ is assumed to be zero.

The *communication to computation ratio* (CCR) of a task graph is a measure of the task graph granularity and can be defined in various ways [8], [12], [16], [20]. We adopted the definition used in [16] which defines CCR as the ratio between the average communication and computation costs in the task graph.

The task graph *width* (W) is defined as the maximum number of tasks that are not connected through a path. Usually, W is considerably less than V; however, in the

TABLE 2
Some Notation Symbols and Their Definitions

Symbol	Definition
V	Number of vertices (tasks) in the task graph.
E	Number of edges (task dependencies) in the task graph.
W	Task graph width.
CCR	Communication to computation ratio.
P	Number of processors.
$T_r(p)$	Time p becomes idle; $p \in \mathcal{P}$
p_r	Processor becoming idle the earliest.
$T_s(t)$	t's start time; $t \in \mathcal{V}$.
$T_f(t)$	t's finish time; $t \in \mathcal{V}$.
$p_t(t)$	Processor t is scheduled on; $t \in \mathcal{V}$.
$T_b(t)$	t's bottom level; $t \in \mathcal{V}$.
$p_e(t)$	t's enabling processor; $t \in \mathcal{V}$.
$T_m(t)$	t's last message arrival time; $t \in \mathcal{V}$.
$T_e(t,p)$	t's effective message arrival time if scheduled on p; $t \in \mathcal{V}, p \in \mathcal{P}$.
$T_s(t,p)$	t's start time if scheduled on p; $t \in \mathcal{V}, p \in \mathcal{P}$.

worse case, W can still be equal to V. For this reason, many papers do not even mention W, but use V instead.

Tasks with no input or no output edges are called *entry* and *exit* tasks, respectively. The *bottom level* (T_b) of a task is defined as the longest path from that task to any exit task, where the path length is the sum of the computation and communication costs of the tasks and dependences belonging to the path. A task is said to be *ready* if all its parents have been scheduled. Note that the number of ready tasks never exceeds W.

As a distributed system, we assume a set $\mathcal{P}$ of P homogeneous processors, connected in a clique topology in which inter-processor communication is performed without contention.

Once scheduled, a task t is associated with a processor $p_t(t)$, a *start time* $T_s(t)$ and a *finish time* $T_f(t)$. If the task is not scheduled, these three values are not defined.

A *partial schedule* is obtained when only a subset of the tasks has been scheduled. The *processor ready time* of a processor $p \in \mathcal{P}$ on a partial schedule is defined as the finish time of the last task scheduled on that processor:

$$T_r(p) = \max_{t \in \mathcal{V}, p_t(t)=p} T_f(t).$$

Given a partial schedule, we define the *processor becoming idle the earliest* (p_r) to be the processor with the minimum T_r:

$$T_r(p_r) = \min_{p \in \mathcal{P}} T_r(p).$$

If there is more than one processors becoming idle at the same earliest T_r, p_r is randomly selected between them. The *last message arrival time* of a ready task t is defined as

$$T_m(t) = \max_{(t',t) \in \mathcal{E}} \{T_f(t') + T_c(t',t)\}.$$

```
Scheduling ()
BEGIN
  FORALL t ∈ V DO
    ComputePrio (t);
    IF t is an entry task THEN
      EnqueueTask (ready_tasks, t, Prio(t));
    END IF
  END FORALL
  WHILE NOT all tasks scheduled DO
    t ← DequeueTask (ready_tasks);
    p ← SelectProcessor (t);
    ScheduleTask (t, p);
    FORALL t' ∈ new ready task set DO
      EnqueueTask (ready_tasks, t', Prio(t'));
    END FORALL
  END WHILE
END
```

Fig. 1. The LSSP framework.

The *enabling processor* of a ready task t, denoted by $p_e(t)$, is the processor from which the last message arrives. Also, in this case, if there are more processors for which the same $T_m(t)$ occurs, the enabling processor $p_e(t)$ is selected randomly between them. The messages sent within the same processor are assumed to take zero communication time. Therefore, we define *effective message arrival time* as

$$T_e(t,p) = \max_{(t',t) \in \mathcal{E}, p_t(t') \neq p} \{T_f(t') + T_c(t',t)\}.$$

The start time of a ready task t when scheduled to a processor p is defined as

$$T_s(t,p) = \max\{T_e(t,p), T_r(p)\}.$$

Note that for a scheduled task t, we have

$$T_s(t) = T_s(t, p_t(t)).$$

A ready task t is said to be *EP* if $T_m(t) \geq T_r(p_e(t))$ and *non-EP*, otherwise. Thus, an EP task starts the earliest on its enabling processor.

The scheduling problem objective is to schedule the tasks in $\mathcal{V}$ on the processors in $\mathcal{P}$ such that the parallel completion time (schedule length) is minimized. The parallel completion time is defined as

$$T_p = \max_{p \in \mathcal{P}} T_r(p).$$

Table 2 summarizes the notation used in this paper.

3 General Framework for LSSP

Analyzing LSSP algorithms, such as MCP, one can distinguish three parts (these parts can also be identified in Fig. 1):

- *Task's priority computation*, which takes at least $O(E+V)$ time, since the whole task graph must be traversed.
- *Task selection* implies sorting the ready tasks according to their priorities and selecting at each iteration the task with the highest priority. Consequently, task

selection takes $O(V \log W)$ time, since the ready tasks have to be maintained sorted.

- *Processor selection* selects the "best" processor for the previously selected task, usually as the processor on which the task starts the earliest (e.g., MCP, DPS). Processor selection takes $O((E+V)P)$ time, when to find the earliest start time of a task, $T_s(t,p)$ must be computed for each processor.

One can note that the highest-complexity parts of the LSSP algorithms are the task and processor selection, which have $O(V \log W)$ and $O((E+V)P)$ time complexity, respectively. In the rest of the section, we describe a general framework to reduce the time complexity of both task and processor selection to $O(V \log P)$ for any LSSP algorithm. Moreover, in Section 4, we extend this framework to a particular class of LSDP algorithms. In Section 5, we show that despite its much lower complexity, this framework produces results comparable to the higher-complexity scheduling algorithms.

3.1 Processor Selection

In this section, we prove that selecting the processor on which a task starts the earliest need not consider all processors but only two: 1) the enabling processor $p_e(t)$ and 2) the processor becoming idle the earliest p_r.

The start time of a task t on a processor p is defined as the maximum between 1) the effective message arrival time and 2) the time p becomes idle: $T_s(t) = \max\{T_e(t,p), T_r(p)\}$. Thus, $T_s(t)$ is minimized on one of the two processors that minimize the two components of the start time. Consequently, there are only two possible candidates: 1) the enabling processor $p_e(t)$, because only mapping t on $p_e(t)$ can zero the last message communication cost and 2) the processor becoming idle the earliest p_r. This is formalized in the following.

First, we prove that for any ready task t and processor $p \neq p_e(t)$, $T_e(t,p) = T_m(t)$.

Lemma 1. *Let $t \in \mathcal{V}$ be a ready task. Then, $\forall p \neq p_e(t)$: $T_e(t,p) = T_m(t)$*

Proof. Let $t' \in \mathcal{V}$ be the task that determines $p_e(t)$: $p_t(t') = p_e(t)$ and $(t',t) \in \mathcal{E}$ such that $T_f(t') + T_c(t',t) = T_m(t)$. From the definition of T_e, it follows that

$$\forall p \neq p_e(t): \quad T_m(t) \geq T_e(t,p) \geq T_f(t') + T_c(t',t) = T_m(t)$$

It follows that $\forall p \neq p_e(t):\ T_m(t) = T_e(t,p)$. □

Next, using the previous lemma, we prove that, indeed, given a task t, one of the two processors: $p_e(t)$ and p_r determines the minimum $T_s(t,p)$.

Theorem 1. *Let t be a ready task. Then, at least one of the processors $p \in \{p_e(t), p_r\}$ satisfies*

$$T_s(t,p) = \min_{p_x \in \mathcal{P}} T_s(t,p_x).$$

Proof. From Lemma 1, it follows that

$$\forall p_x \neq p_e(t):\ T_e(t,p_x) = T_m(t).$$

```
SelectProcessor (t)
BEGIN
  IF T_s(t, p_e(t)) < T_s(t, p_r) THEN
    p ← p_e(t);
  ELSE
    p ← p_r;
  END IF
  RETURN p;
END
```

Fig. 2. Processor selection.

As $T_e(t,p_e(t)) \leq T_m(t)$, it follows that

$$\forall p \in \mathcal{P}: \quad T_e(t,p_e(t)) \leq T_e(t,p). \tag{1}$$

By its definition, $T_s(t,p)$ is minimized on either 1) the processor that minimizes $T_e(t,p)$ or 2) the processor that minimizes $T_r(p)$. Using (1) and p_r's definition, $T_s(t,p)$ is minimized on either 1) $p_e(t)$ or 2) p_r. □

From Theorem 1, it follows that restricting the selection to these two processors indeed does not affect the performance of the algorithm. However, although essentially similar, there is a minor difference in the processor selection scheme between our processor selection and the original processor selection. Throughout the scheduling process it may happen that a ready tasks can start at the same earliest time on different processors. Our scheme and the original scheme have different criteria to break this tie because our scheme considers only two candidate processors, while the original scheme considers all processors. As a consequence, there are few cases in which the two schemes may still select a different processor for a task to be scheduled. See Fig. 2 for a pseudocode implementation of our processor selection scheme.

Even though our processor selection still performs with the same accuracy as the original processor selection, the total processor-selection time complexity is significantly reduced from $O((E+V)P)$ to $O(V \log(P) + E)$ (We need $O(E+V)$ to traverse the task graph $O(V \log P)$ to maintain the processors sorted at each scheduling step).

3.2 Task Selection

The $O(V \log W)$ complexity of the task selection step can be reduced by sorting only a *constant* number of ready tasks. Thus, the task priority queue is composed from 1) a sorted list of a fixed size H and 2) a first-in first-out list (see Fig. 3). We sort as many tasks as fit in the fixed-size sorted list, while the others are stored in an unsorted FIFO list which has an $O(1)$ access time complexity. When a task becomes ready it is added to the sorted list when there is room to accommodate it, otherwise it is added to the FIFO list. For this reason, as long as the sorted list is not full, there cannot be tasks in the FIFO list. The tasks are always dequeued from the sorted list. After a dequeue operation, if the FIFO list is not empty, one task is moved to the sorted list. See Fig. 4 for a pseudocode implementation of the above operations on a priority queue of size H.

The time complexity of sorting tasks when using a priority queue of size H decreases from $O(V \log W)$ to $O(V \log H)$ as all the tasks are enqueued and dequeued in

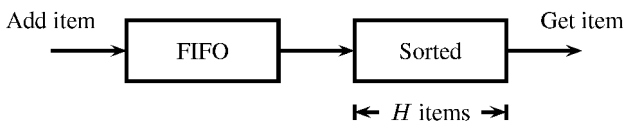

Fig. 3. Fixed-size priority queue.

the sorted list only once, respectively. We still keep H as part of the complexity, i.e., not drop it as a constant, because for achieving a good performance, H needs to be adjusted with P.

A possible drawback of sorting only a limited number of tasks is that the task with the highest priority may not be included in the sorted list, but be temporarily stored in the FIFO list. The size of the sorted list must therefore be large enough not to affect the performance of the algorithm too much. At the same time, it should not be too large in view of the time complexity. In our experiments, we find that a size of P is required to maintain a performance comparable to the original list scheduling algorithm (see Section 5.3), which results in a task-selection complexity of $O(V \log P)$.

By introducing the above techniques for task and processor selection, the total complexity of the LSSP algorithm is decreased to $O(V \log (P) + E)$, which is clearly a significant improvement over the typical time complexity $O(V \log (W) + (E + V)P)$ of the current LSSP approaches.

3.3 Complexity Analysis

The complexity of the generic LSSP algorithm described above is as follows: Computing task priorities (e.g., T_b) takes $O(E + V)$. Each task is once added to and once removed from the ready-task priority queue. For a fully-sorted priority queue, task selection takes $O(V \log W)$. For a partially-sorted priority queue of size H, task selection takes $O(V \log H)$. As a partially-sorted priority queue of size P yields good results (as shown in Section 5.3), task selection takes $O(V \log P)$. Selecting the enabling processor for each task implies scanning all the tasks and edges in the task graph, which takes $O(E + V)$. Finding the processor becoming idle the earliest takes $O(\log P)$ for each task, resulting in $O(V \log P)$ time for all tasks. As a result, the total complexity of LSSP is $O(V(\log (W) + \log(P)) + E)$ if the fully-sorted priority queue is used and $O(V(\log (P)) + E)$ if the partially-sorted priority queue is used.

3.4 Case Study

In this section, we illustrate our task and processor selection techniques by applying them to a slightly simplified version of MCP (Modified Critical Path) [30]. In MCP, the ready task with the highest bottom level has the highest priority. We break the ties randomly (in the original version of MCP, the ties are broken by also considering bottom levels of the task's descendents). At each iteration, the task with the highest priority is scheduled on the processor that can execute it the earliest.

We modify MCP to use our task and processor selection techniques and we name the resulted algorithm Fast Critical Path (FCP) [24]. In Table 3, we present the execution trace of the FCP algorithm using a partially-sorted priority queue of size 2 by scheduling the task graph in Fig. 5 on three processors. The first two columns in the table list the sorted and FIFO lists within the priority queue. The task's T_b is included to illustrate their sorting order. The third column in the table lists the task to be scheduled at the current step. In the next two columns, the enabling processor and the processor becoming idle the earliest are listed along with the current task's start times on each of them. Finally, the last column shows the scheduling at the current iteration, including the task's start time and finish time.

At the beginning, there is only one ready task, namely, t_0, which is accommodated by the sorted list. There is no enabling processor for t_0 since it has no parents. Task t_0 is thus scheduled on processor p_0 as one of the processors becoming idle the earliest at time 0.

After scheduling t_0, three tasks become ready, namely, t_1, t_2, and t_3. Tasks t_1 and t_2 are successively added to the task priority queue and are accommodated in the sorted part. Task t_3, however, cannot be added to the sorted list because the sorted list is already filled. Therefore, t_3 is stored in the FIFO list, despite the fact it has the highest priority.

The next task to be scheduled is the first task from the sorted list, namely, t_1. It starts at time 2 on its enabling processor p_0 and at time 3 on the processor becoming idle the earliest p_1. Consequently, t_1 is scheduled on p_0 at time 2. As one task has been dequeued from the sorted list, t_3 can be moved from the top of the FIFO list to the sorted list. Also t_4 and t_5 that become ready are added to the FIFO list.

The next task t_3 is scheduled on processor p_1 which becomes idle the earliest, as it provides the earliest start time: 3. In the same manner, t_2 is then scheduled on p_0. Next, task t_5 achieves the same starting time 6 on both it's enabling processor and the processor becoming idle the

```
EnqueueTask (l, t, ρ)
BEGIN
  IF size (l.sorted) ≤ H THEN
    Enqueue_sorted (l.sorted, t, ρ);
  ELSE
    Enqueue_fifo (l.fifo, t);
  END IF
END
```

```
DequeueTask (l)
BEGIN
  task ← Dequeue_sorted (l.sorted);
  IF l.FIFO is not empty THEN
    t ← Dequeue_fifo (l.fifo);
    Enqueue_sorted (l.sorted, t, ρ(t));
  END IF
  RETURN task;
END
```

Fig. 4. Task selection.

TABLE 3
Execution Trace of the FCP Algorithm

Ready tasks $[T_b]$ sorted	Ready tasks $[T_b]$ FIFO	t	$p_e(t)/T_s$	p_r/T_s	Scheduling $t \rightarrow p, [T_s - T_f]$
$t_0[15]$	–	t_0	–	$p_0/0$	$t_0 \rightarrow p_0, [0-2]$
$t_1[11]$ $t_2[9]$	$t_3[12]$	t_1	$p_0/2$	$p_1/3$	$t_1 \rightarrow p_0, [2-4]$
$t_3[12]$ $t_2[9]$	$t_4[6]$ $t_5[8]$	t_3	$p_0/4$	$p_1/3$	$t_3 \rightarrow p_1, [3-6]$
$t_2[9]$ $t_4[6]$	$t_5[8]$	t_2	$p_0/4$	$p_2/6$	$t_2 \rightarrow p_0, [4-6]$
$t_5[8]$ $t_4[6]$	$t_6[6]$	t_5	$p_0/6$	$p_2/6$	$t_5 \rightarrow p_2, [6-9]$
$t_4[6]$ $t_6[6]$	–	t_4	$p_0/6$	$p_0/6$	$t_4 \rightarrow p_0, [6-9]$
$t_6[6]$	–	t_6	$p_1/7$	$p_1/7$	$t_6 \rightarrow p_1, [7-9]$
$t_7[2]$	–	t_7	$p_2/11$	$p_0/12$	$t_7 \rightarrow p_2, [11-13]$

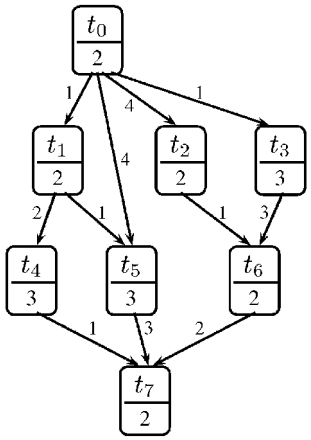

Fig. 5. Example of DAG.

earliest. The tie is broken by selecting the processor becoming idle the earliest p_2. The same applies to t_4 which is scheduled next on p_0 at 6. The last two tasks t_6 and t_7 are similarly scheduled on p_1 and p_2 at times 7 and 11, respectively.

4 Extensions for LSDP

In this section, we explore the possibility of extending the results for LSSP algorithms described in Sections 3.1 and 3.2 to LSDP algorithms.

In LSDP algorithms, priorities are associated to pairs of a task and a processor. At each iteration, the pair of task and processor yielding the highest priority is selected. As priorities change throughout the scheduling process, they must be recomputed at each iteration. For example, ETF schedules at each step the ready task that starts the earliest on the processor where this time is obtained: $\rho_{ETF}(t,p) = \max\{T_e(t,p), T_r(p)\}$ [11], ERT schedules the ready task finishing the earliest on the processor where this time is obtained: $\rho_{ERT}(t,p) = \max\{T_e(t,p), T_r(p)\} + T_w(t)$ [18] and DLS defines its priority (called dynamic level) as $\rho_{DLS}(t,p) = T_b(t) - \max\{T_e(t,p), T_r(p)\}$, the task and processor with the highest dynamic level being scheduled [29]. Using a general notation, we have the following dynamic priority:

$$\rho(t,p) = \alpha(t) + \max\{T_e(t,p), T_r(p)\},$$

where $\alpha(t)$ is a value that is independent of the scheduling process which can therefore can be computed before the scheduling is started (e.g., $\alpha_{ETF}(t) = 0$, $\alpha_{ERT}(t) = T_w(t)$, and $\alpha_{DLS}(t) = -T_b(t)$). The pair of task and processor with the lowest value for ρ has the highest priority.

Using Lemma 1, the priority of task t on processor p is:

$$\rho(t,p) = \begin{cases} \alpha(t) + \max\{T_e(t,p), T_r(p)\}, & p = p_e(t) \\ \alpha(t) + \max\{T_m(t), T_r(p)\}, & p \neq p_e(t). \end{cases} \quad (2)$$

As for LSSP, we treat separately the two cases of tasks starting on their enabling processors and tasks starting on a nonenabling processor. In the *EP case*, selecting the task with the highest priority on its enabling processor is performed in two steps. First, on each processor, the tasks enabled by that processor are sorted according to their priority. Second, the processors are sorted by the highest priority of the tasks enabled by them.

In the *non-EP case*, selecting the task with the highest priority on a nonenabling processor is based on the observation that a task's priority on a nonenabling processor is minimized by the processor becoming idle the earliest. To make the selection processor even, we do not separate the enabling and nonenabling processors in this case. The selection result is not affected because, if a task and its enabling processor are selected, the task will have a priority higher than any task starting on a nonenabling processor. As a consequence, the EP-case selection will give the task and processor with the highest priority.

Let

$$\begin{aligned} \rho'(t) &= \alpha(t) + \max\{T_m(t), T_r(p_r)\} \\ &= \begin{cases} \alpha(t) + T_m(t), & T_m() \geq T_r(p_r) \\ \alpha(t) + T_r(p_r), & T_m() < T_r(p_r). \end{cases} \end{aligned} \quad (3)$$

The task with the lowest $\rho'(t)$ is one of the two tasks that minimize the two cases of $\rho'(t)$: 1) the task with the minimum $\alpha(t) + T_m(t)$ or 2) the task with the minimum $\alpha(t) + T_r(p_r)$.

Using the task and processor selection schemes described above, we are able to find the task and processor pair having the highest priority in only three tries, one for the EP case two for the non-EP case. There are P task priority queues maintained for the EP case two for the non-EP case. However, each task is added to three task priority queues, one for the EP case and two for the non-EP case. Two other processor queues are maintained, one for the EP case and one for the non-EP case. As a consequence, the time complexity of the LSDP algorithms, when using the approach described in this section, becomes $O(V(\log(W) + \log(P)) + E)$. This is already a significant improvement compared to the original $O(W(E+V)P)$ time complexity.

We can further reduce this time complexity using the partially-sorted priority queue described in Section 3.2 for the task priority queues, in a similar manner as for LSSP algorithms. Also for LSDP, our experiments show that a size of P is required to maintain a performance comparable to the

original list scheduling algorithm [22]. In this case, the time complexity is further reduced to $O(V(\log(P)+E))$. Details about implementing these techniques to modify DLS and ETF can be found in [22] and [23], respectively.

5 Performance Results

To evaluate our scheduling approach, we compare two algorithms that use our framework with five other well-known scheduling algorithms. The two algorithms using our framework are the fast versions of MCP and DLS, which we call FCP (Fast Critical Path) and FDLS (Fast DLS), respectively. FCP is selected as a representative for the general framework for LSSP algorithms, while FDLS as a representative for the extension applicable to the LSDP algorithms.

The five well-known scheduling algorithms included in our comparison are MCP (Modified Critical Path) [30], DLS (Dynamic Level Scheduling) [29], DSC-LLB (Dynamic Sequence Clustering—List Load Balancing) [25], [33], DSC1 (a version of DSC for a bounded number of processors) [31], and CPM (Critical Path Method) [21]. We use a slightly simplified version of MCP in which ties between task priorities are broken randomly instead of considering all descendants of those tasks. MCP and DLS have been shown to obtain competitive results for list scheduling with static and dynamic priorities, respectively [16], [25]. They are also the original algorithms from which FCP and FDLS have been derived, therefore, we can see what are the differences induced by using our techniques. DSC-LLB, at a relatively low cost, is one of the best performing multistep scheduling algorithms [25]. DSC1 is a version of DSC which performs the scheduling for a bounded number of processors in one step. As in its original description in [31] the processor selection implementation is not described, we use the processor selection scheme described in Section 3.1 to implement DSC1. CPM has an even lower cost, as its processor selection has the same complexity as in our framework.

We consider task graphs representing various types of parallel algorithms. The selected problems are *LU decomposition* ("LU"), a *Laplace equation solver* ("Laplace") and a *stencil algorithm* ("Stencil") [9]. For each of these problems, we adjusted the problem size to obtain task graphs of about 2,000 tasks. For each problem, we varied the task graph granularities, by varying the communication to computation ratio (CCR). The values used for CCR were 0.2 and 5.0. For each problem and each CCR value, we generated five graphs with random execution times and communication delays (independent identical distributed uniform distribution with unit coefficient of variation), the results being the average over the five graphs (in view of the low overall variance, five samples are sufficient). For these problems, we use up to 32 processors, which, given the problem sizes, is the processor range where speedup is still obtained.

5.1 Running Times

Our main objective is to reduce task scheduling cost (i.e., running time), while maintaining performance. In Fig. 6, the average running time of the algorithms is shown in CPU seconds as measured on a Pentium Pro/233MHz PC with 64Mb RAM running Linux 2.0.32. DLS is the most costly among the compared algorithms. Its cost increases from 86 ms for two processors up to 1 s for 32 processors. MCP also has a runtime proportional with the number of processors, but its cost is significantly lower. For $P = 2$, it runs for 32 ms, while for $P = 32$, the running time is 206 ms. DSC-LLB does not vary with P, as its most costly step, clustering, is independent of number of processors. The DSC-LLB running times vary around 98 ms. DSC1 has a running cost comparable to DSC-LLB, because it is a derivation of DSC and because we use the processor selection scheme described in Section 3.1 which introduces marginal extra cost compared to the processor selection in the unbounded case. The slightly extra cost in DSC-LLB compared to DSC1 is caused by the extra steps performed in DSC-LLB for cluster merging and task reordering. The DSC1 running times vary around 82 ms.

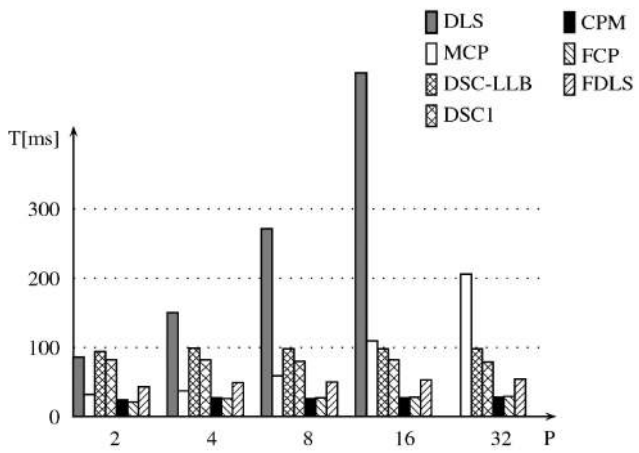

Fig. 6. Cost comparison of the algorithms using our framework versus existing algorithms.

The other three algorithms, CPM, FCP, and FDLS have considerably lower running times. They do not vary significantly with the number of processors, varying around 27 ms, 27 ms, and 50 ms, respectively.

5.2 Scheduling Performance

In this section, we first show how the resulting FCP schedules scale to the number of processors, in terms of speedup. Then, we show that both FCP and FDLS achieve performance comparable with the existing list scheduling algorithms.

In Fig. 7, we illustrate the speedup achieved by FCP for the considered problems. For LU and Laplace, there is a large number of join operations. As a consequence, there is not much parallelism available and the speedup is limited, especially for high values of CCR. Stencil is more regular and, therefore, more parallelism can be exploited. As a consequence, the resulting speedup is almost linear for the coarse-grain (i.e., CCR = 0.2).

To compare scheduling performance, we use the *normalized schedule lengths* (NSL), defined as the ratio between the schedule length of the algorithm under study and the schedule length of a reference algorithm. In Fig. 8, we compare the algorithms using our framework with MCP, DLS, DSC-LLB, and DSC1 using average schedule lengths normalized to the reference algorithm MCP. In Fig. 9, we

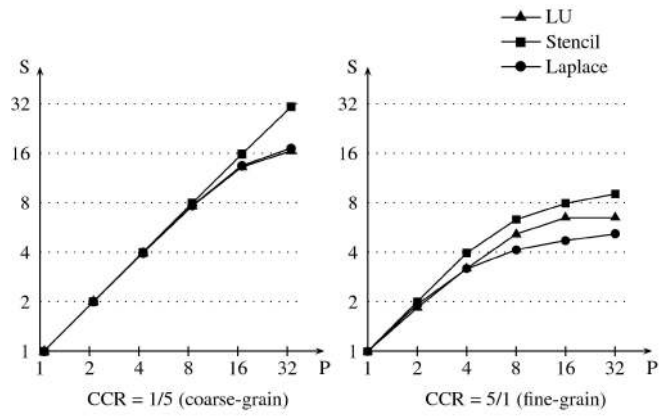

Fig. 7. FCP speedup.

compare against CPM using average schedule lengths normalized to FCP.

FCP has both low cost and good performance. Compared to the more expensive two algorithms, MCP and DLS, one can note that FCP usually performs comparable. The only case in which FCP performs with 10 percent worse is for fine-grained Stencil, which is caused by the large number of tasks at each Stencil iteration, all with different priorities, for which a fully sorted ready task list is requested for best performance. Compared with DSC-LLB, FCP consistently produces better schedules (up to 44 percent), while at the same time running five times faster. DSC1 performs comparable to FCP (better up to 8 percent, or worse down to 5 percent depending on the problem). Compared to the low-cost CPM, FCP has a consistently better performance, although the running times are comparable. Thus, FCP outperforms CPM up to 140 percent.

FDLS achieves an even better performance, generally outperforming all of MCP, DLS and DSC1 up to 22 percent, 16 percent, and 11 percent, respectively. FDLS also consistently outperforms DSC-LLB and CPM up to 34 percent and 152 percent, respectively.

5.3 Priority Queue Size Sensitivity

As mentioned earlier, in our framework, only a fixed number of tasks is maintained sorted at each scheduling step. In Fig. 10, we study the influence of the partially-sorted priority queue size H to the scheduling performance of FCP. When computing NSL, we use as a reference algorithm the FCP version using a P-length priority queue. For the influence of the priority queue size on FDLS, see [22].

For problems where there is more parallelism to be exploited (e.g., Stencil), FCP yields good results, even for very small sizes of the partially-sorted priority queue. Moreover, in the fine-grain case, for a zero-length priority queue, FCP performs even better than for a fully sorted priority queue. The reason is that Stencil has a regular task graph which yields a very good schedule simply by

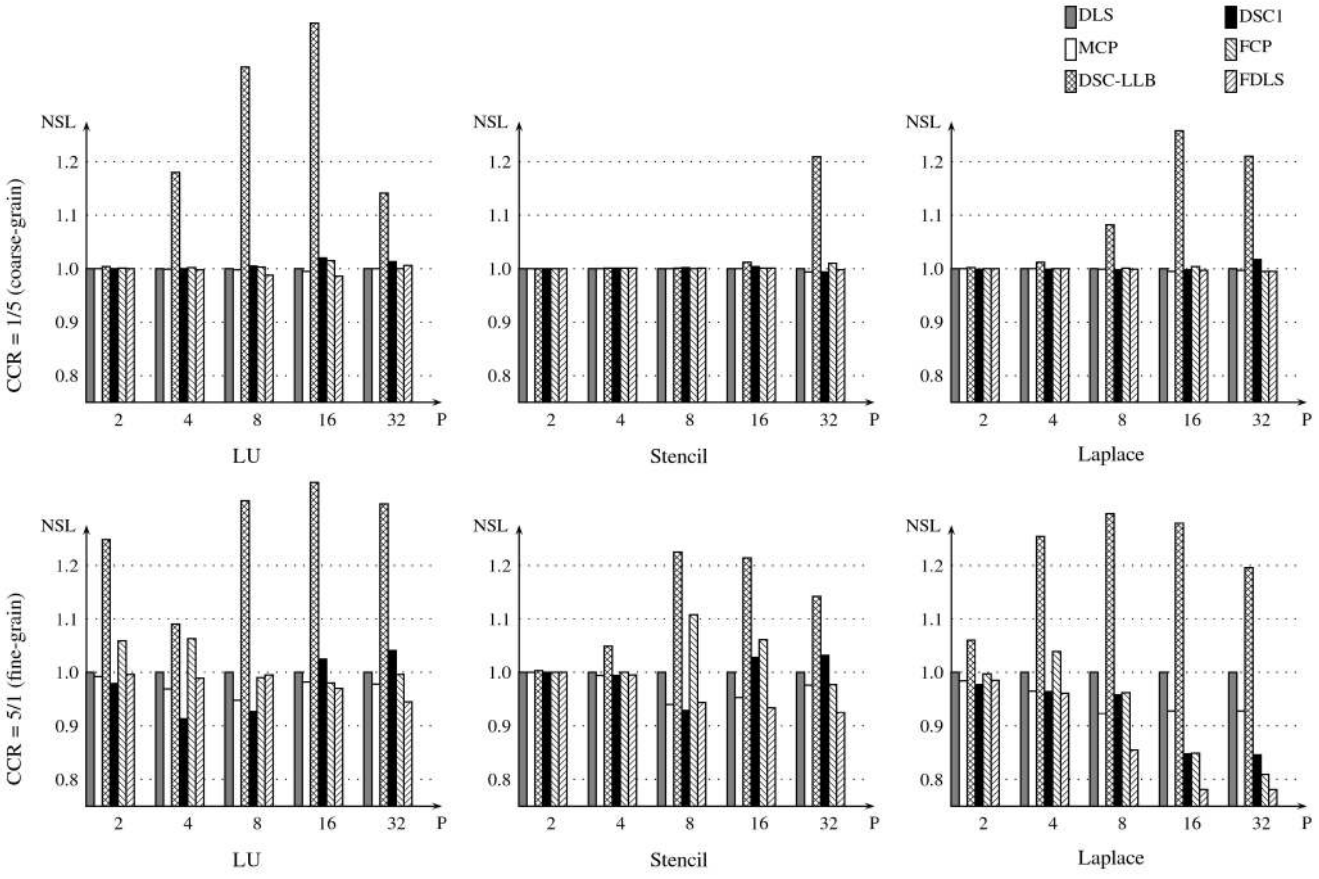

Fig. 8. Performance comparison of the algorithms using our framework versus existing algorithms. Reference is MCP.

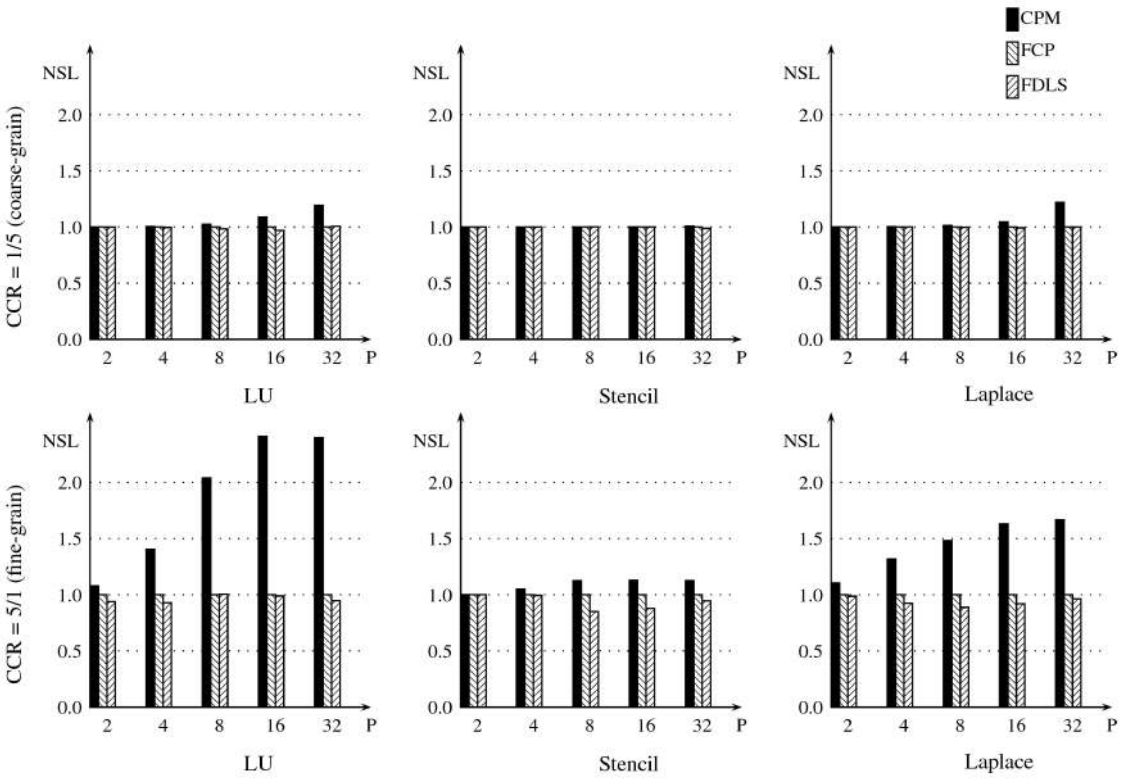

Fig. 9. Performance comparison of the algorithms using our framework versus existing algorithms. Reference is FCP.

scheduling the tasks in order they become ready (i.e., FIFO order).

For problems where there is not so much parallelism, such as LU, the performance degrades for small sizes of the priority queue size. However, one can note that for coarse-grain task graphs, a priority queue size greater than P does not improve the schedule significantly. For fine-grain LU, sorting more tasks still improves the schedule, however, these cases occur when the speedup does not increase significantly (i.e., $S = 4.1$ for $P = 8$, $S = 4.7$ for $P = 16$, etc.) the schedule improvement do not exceed 15 percent.

An informal explanation of the fact that the schedule improvements are bounded by a priority queue of size P in the coarse-grain case is the following. If the number of ready tasks is greater than P, the scheduling process tends to become a load-balancing scheme. The reason is that after mapping the first P ready tasks, the communication costs for the remaining ready tasks tends to be overlapped with

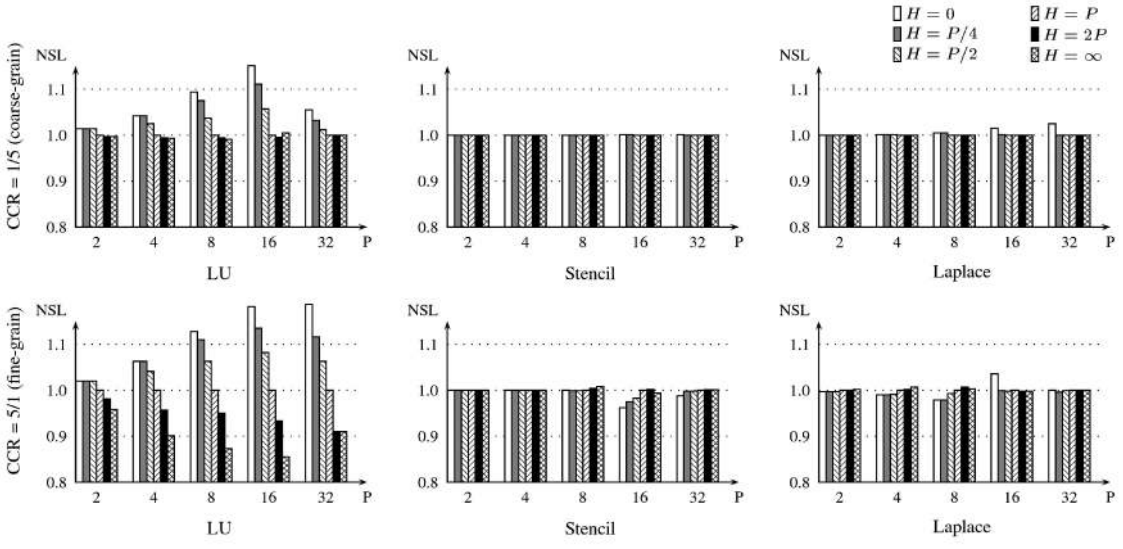

Fig. 10. The influence of priority queue size to the FCP performance.

the execution of the previous P mapped tasks. The task priorities, used to select the task with the least probable delay caused by communication costs, therefore become less important. As a result, a priority queue with a smaller size can be used. For a small number of ready tasks at each scheduling step, the task priorities become more important. In this case, the tasks must be all maintained sorted to obtain good performance. From the above experiments, it can be concluded that a priority queue size of P is a good choice for the FCP algorithm. A smaller size penalizes problems with limited parallelism, while a greater size does not yield further improvements.

6 Conclusion

In this paper, we show that list scheduling with statically computed priorities can be performed at a significantly lower cost than existing approaches, without sacrificing performance. Our approach is general, i.e., it can be applied to any list scheduling algorithm with static priorities.

In our framework, the list scheduling algorithms have a low time complexity because low-complexity methods are used for the most time consuming parts of them, i.e., processor selection and task selection, preserving the criteria used in the original algorithms. Processor selection is performed by selecting between only two processors: either the task's enabling processor or the processor which becomes idle the earliest. For task selection, instead of sorting all the tasks, only a limited number of tasks are sorted at any given time. By using these methods, the time complexity is reduced to $O(V \log(P) + E)$, which represents a significant cost improvement over the $O(V \log V + (E+V)P)$ complexity of the original list scheduling algorithms, while performance is maintained.

Using an extention of this method, we can also significantly reduce the time complexity of a particular class of list scheduling with dynamic priorities (including algorithms such as DLS, ETF, or ERT) from $O(W(E+V)P)$ to $O(V \log(P) + E)$.

We exemplify our method by applying it to MCP and DLS. Our results confirm that the modified versions of the list scheduling algorithms perform comparable to their original versions, yet at a significantly lower cost. We also show that the modified versions of the list scheduling algorithms consistently outperform multistep algorithms, such as DSC-LLB, which also have higher complexity clearly outperform algorithms in the same class of complexity, such as CPM.

Especially, in view of the large problem and processor dimensions involved with real-world high-performance computing, our results indicate that our approach offer a superior cost-performance tradeoff compared to current list scheduling algorithms. Our approach outperforms other low-cost algorithms and even matches the better performing, higher-cost list scheduling algorithms. The significant cost reduction that has been achieved makes, therefore, our approach a viable option for both compile-time and runtime scheduling of practical high-performance applications.

References

[1] T.L. Adam, K.M. Chandy, and J.R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Comm. ACM,* vol. 17, no. 12, pp. 685-690, Dec. 1974.

[2] I. Ahmad and Y.-K. Kwok, "A New Approach to Scheduling Parallel Programs Using Task Duplication," *Proc. Int'l Conf. Parallel Processing (ICPP),* pp. 47-51, Aug. 1994.

[3] Y.C. Chung and S. Ranka, "Application and Performance Analysis of a Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed-Memory Multiprocessors," *Proc. Int'l Conf. Supercomputing (SC),* pp. 512-521, Nov. 1992.

[4] M. Cosnard, E. Jeannot, and L. Rougeot, "Low Memory Cost Dynamic Scheduling of Large Coarse Grain Task Graphs," *Proc. Int'l Parallel Processing Symp./Symp. Parallel and Distributed Processing (IPPS/SPDP),* Mar. 1998.

[5] G.L. Djordjevic and M.B. Tosic, "A Heuristic for Scheduling Task Graphs with Communication Delays onto Multiprocessors," *Parallel Computing,* vol. 22, pp. 1197-1214, Nov. 1996.

[6] H. El-Rewini, H.H. Ali, and T.G. Lewis, "Task Scheduling in Multiprocessing Systems," *Computer,* pp. 27-37, Dec. 1995.

[7] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., 1979.

[8] A. Gerasoulis and T. Yang, "On the Granularity and Clustering of Directed Acyclic Task Graphs," *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 6, pp. 686-701, 1993.

[9] G. Golub and C.F. van Loan, *Matrix Computations.* Baltimore, Md.: Johns Hopkins Univ. Press, 1996.

[10] R.L. Graham, "Bounds on Multiprocessing Timing Anomalies," *SIAM J. Applied Math.,* vol. 17, no. 2, pp. 416-429, Mar. 1969.

[11] J.-J. Hwang, Y.-C. Chow, F.D. Anger, and C.-Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM J. Computing,* vol. 18, pp. 244-257, Apr. 1989.

[12] A.A. Khan, C.L. McCreary, and M.S. Jones, "A Comparison of Multiprocessor Scheduling Heuristics," *Proc. Int'l Conf. Parallel Processing (ICPP),* pp. 243-250, Aug. 1994.

[13] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computation Upon Multiprocessor Architectures," *Proc. Int'l Conf. Parallel Processing (ICPP),* vol. 3, pp. 1-8, Aug. 1988.

[14] B. Kruatrachue and T.G. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software,* pp. 23-32, Jan. 1988.

[15] Y.-K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 5, pp. 506-521, May 1996.

[16] Y.-K. Kwok and I. Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," *J. Parallel and Distributed Computing,* vol. 59, pp. 381-422, 1999.

[17] Y.-K. Kwok, I. Ahmad, and J. Gu, "FAST: A Low-Complexity Algorithm for Efficient Scheduling of DAGs on Parallel Processors," *Proc. Int'l Conf. Parallel Processing (ICPP),* vol. II, pp. 150-157, Aug. 1996.

[18] C.-Y. Lee, J.-J. Hwang, Y.-C. Chow, and F.D. Anger, "Multiprocessor Scheduling with Interprocessor Communication Delays," *Operations Research Letters,* vol. 7, pp. 141-147, June 1988.

[19] M.A. Palis, J.-C. Liou, and D.S.L. Wei, "Task Clustering and Scheduling for Distributed Memory Parallel Architectures," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 1, pp. 46-55, Jan. 1996.

[20] G.-L. Park, B. Shirazi, and J. Marquis, "DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor System," *Proc. Int'l Parallel Processing Symp. (IPPS),* pp. 157-166, Apr. 1997.

[21] G.-L. Park, B. Shirazi, J. Marquis, and H. Choo, "Decisive Path Scheduling: A New List Scheduling Method," *Proc. Int'l Conf. Parallel Processing (ICPP),* pp. 472-480, Aug. 1997.

[22] A. Radulescu, "Compile-Time Scheduling for Distributed-Memory Systems," PhD thesis, Delft Univ. of Technology, June 2001.

[23] A. Radulescu and A.J.C. van Gemund, "FLB: Fast Load Balancing for Distributed-Memory Machines," *Proc. Int'l Conf. Parallel Processing (ICPP),* pp. 534-541, Sept. 1999.

[24] A. Radulescu and A.J.C. van Gemund, "On the Complexity of List Scheduling Algorithms for Distributed-Memory Systems," *Proc. ACM Int'l Conf. Supercomputing (ICS),* pp. 68-75, June 1999.

[25] A. Radulescu and A.J.C. van Gemund, "LLB: A Fast and Effective Scheduling Algorithm for Distributed-Memory Systems," *Proc. Int'l Parallel Processing Symp./Symp. Parallel and Distributed Processing (IPPS/SPDP),* pp. 525-530, Apr. 1999.

[26] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* MIT Press, 1989.
[27] B. Shirazi, H.-B. Chen, and J. Marquis, "Comparative Study of Task Duplication Static Scheduling versus Clustering and Non-Clustering Techniques," *Concurrency: Practice and Experience,* vol. 7, pp. 371-389, Aug. 1995.
[28] B. Shirazi, M. Wang, and G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling," *J. Parallel and Distributed Computing,* vol. 10, no. 3, pp. 222-232, Nov. 1990.
[29] G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 2, pp. 175-187, Feb. 1993.
[30] M.-Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 1, no. 7, pp. 330-343, July 1990.
[31] T. Yang and C. Fu, "Heuristic Algorithms for Scheduling Iterative Task Computations on Distributed Memory Machines," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 6, pp. 608-622, June 1997.
[32] T. Yang and A. Gerasoulis, "Pyrros: Static Task Scheduling and Code Generation for Message Passing Multiprocessors," *Proc. ACM Int'l Conf. Supercomputing (ICS),* pp. 428-437, June 1992.
[33] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, no. 9, pp. 951-967, Dec. 1994.

Andrei Radulescu received the MSc degree in computer science from Politehnica University of Bucharest, Romania, and the PhD degree in computer science from Delft University of Technology, The Netherlands, in 1995 and 2001, respectively. Since 2001, he has been with Philips Research Laboratories in Eindhoven, The Netherlands. His research interests include resource mapping and scheduling, parallel and distributed computing, and interconnection networks.

Arjan J.C. van Gemund received the BSc degree in physics, the MSc degree (cum laude) in computer science, and the PhD degree (cum laude) in computer science, all from Delft University of Technology, the Netherlands. Between 1980 and 1992, he joined various Dutch companies and institutes, where he worked in embedded hardware and software development, acoustics research, fault diagnosis, and high-performance computing. In 1992, he joined the Information Technology and Systems Faculty at Delft University of Technology, where he is currently serving as an associate professor. He is also the cofounder of Science and Technology, a Dutch software and consultancy startup. His research interests are in the areas of parallel systems performance modeling, fault diagnosis, and parallel programming and scheduling. He is a member of the IEEE Computer Society.