FLB: Fast Load Balancing for Distributed-Memory Machines *

Andrei Rădulescu   Arjan J.C. van Gemund
Faculty of Information Technology and Systems
Delft University of Technology
P.O.Box 5031, 2600 GA Delft, The Netherlands
{*A.Radulescu,A.J.C.vanGemund*}*@its.tudelft.nl*

Abstract

This paper describes a novel compile-time list-based task scheduling algorithm for distributed-memory systems, called Fast Load Balancing (FLB). Compared to other typical list scheduling heuristics, FLB drastically reduces scheduling time complexity to $O(V(\log(W) + \log(P)) + E)$, where V and E are the number of tasks and edges in the task graph, respectively, W is the task graph width and P is the number of processors. It is proven that FLB is essentially equivalent to the existing ETF scheduling algorithm of $O(W(E+V)P)$ time complexity. Experiments also show that FLB performs equally to other one-step algorithms of much higher cost, such as MCP. Moreover, FLB consistently outperforms multi-step algorithms such as DSC-LLB that also have higher cost.

1 Introduction

Scheduling heuristics exist for both bounded and unbounded number of processors. Although attractive from a cost perspective, scheduling for an unbounded number of processors is rarely of practical use, because the required number of processors is not usually available. Hence, their application is typically found within the multi-step scheduling method for a bounded number of processors [8, 9].

Alternatively, scheduling for a bounded number of processors can be performed in a single step. Using this one-step approach, the results are usually better, but at a higher cost. Scheduling for a bounded number of processors can be done either using duplication (e.g., DSH [4], BTDH [2] or CPFD [1]) or without duplication (e.g., MCP [11], ETF [3], DLS [10] or FCP [7]). Duplicating tasks results in better scheduling performance but significantly increases scheduling cost. Non-duplicating task heuristics have a lower complexity and still obtain good schedules. However, when compiling large programs for large systems, the complexity of current approaches is still prohibitive.

This paper presents a new compile-time task scheduling algorithm, called Fast Load Balancing (FLB). We prove that given a partial schedule, FLB schedules at each iteration the ready task that starts the earliest. The main idea is to keep the processors busy, in this respect being close to a load balancing scheme (hence the acronym). The same criterion is also used by ETF, however at a much higher complexity of $O(W(E+V)P)$ compared to FLB's $O(V(\log(W)+\log(P))+E)$ complexity, where V and E are the number of tasks and edges in the task graph, respectively, W is the task graph width, and P is the number of processors. Experiments also show that FLB performs equally to existing one-step scheduling algorithms of much higher complexity such as MCP. Moreover, FLB consistently outperforms multi-step algorithms like DSC-LLB which also have higher cost.

The paper is organized as follows: The next section specifies the scheduling problem and introduces some definitions used in the paper. Section 3 briefly reviews some of the well-known scheduling algorithms for bounded number of processors on which we base our comparisons. In Section 4, the FLB algorithm is presented. Section 5 illustrates the functionality of the algorithm through an execution trace. Section 6 describes its performance. Section 7 concludes the paper.

2 Preliminaries

A parallel program can be modeled by a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of V nodes and $\mathcal{E}$ is a set of E edges. A node in the DAG represents a task, containing instructions that execute sequentially without preemption. Each task $t \in \mathcal{V}$ has a weight $comp(t)$ associated with it, which represents the *computation cost* of executing the task. The edges correspond to task dependencies (communication messages or precedence constraints). Each edge (t, t') has a weight $comm(t, t')$ associated with it, which represents the *communication cost* to satisfy the dependence. The *communication to computation ratio* (CCR) of a parallel program is defined as the ratio between its average communication cost and its average computation cost. If two tasks are scheduled to the same processor, the communication cost between them is assumed to be zero. The task graph *width* (W) is defined as maximum number of tasks that are not connected through a path.

A task with no input edges is called an *entry* task, while a task with no output edges is called an *exit* task. A task is said to be *ready* if all its parents have finished their execution. Note that at any given time the number of ready tasks never exceeds W. A task can start its execution only after all its messages have been received.

*This research is part of the Automap project granted by the Netherlands Computer Science Foundation (SION) with financial support from the Netherlands Organization for Scientific Research (NWO) under grant number SION-2519/612-33-005.

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

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

The *processor ready time* of a given processor $p \in \mathcal{P}$ on a partial schedule is defined as the finish time of the last task scheduled on the given processor, according to $PRT(p) = \max_{t \in \mathcal{V}, PROC(t)=p} FT(t)$.

The *enabling processor* of a ready task t, $EP(t)$ is the processor from which the last message arrives. The *last message arrival time* of a ready task t is defined as $LMT(t) = \max_{(t',t) \in \mathcal{E}} \{FT(t') + comm(t',t)\}$. When task t is tentatively scheduled to a processor p, the messages sent by t's predecessors that are already scheduled to p are assumed to take 0 communication time. Therefore, the *effective message arrival time* will be $EMT(t,p) = \max_{(t',t) \in \mathcal{E}, PROC(t') \neq p} \{FT(t') + comm(t',t)\}$. The estimated start time when scheduling a ready task t to a processor p is defined as $EST(t,p) = \max\{EMT(t,p), PRT(p)\}$.

Given a partial schedule and a ready task t is said to be of *type EP* if its last message arrival time is greater than the ready time of its enabling processor: $LMT(t) \geq PRT(EP(t))$. and of *type non-EP* otherwise. Thus, for EP type tasks start the earliest on their enabling processor. A processor is called *active* if there are EP type tasks for which it is the enabling processor and *passive* otherwise.

The objective of the scheduling problem is to find a scheduling of 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 $T_{par} = \max_{p \in \mathcal{P}} PRT(p)$.

3 Related Work

In this section, three existing scheduling algorithms and their characteristics are described, namely the one-step scheduling algorithms MCP [11] and ETF [3], and the multi-step method DSC-LLB [8, 12].

3.1 MCP

MCP (Modified Critical Path) is a list scheduling algorithm in which task priorities are based on the *latest possible start time* of the tasks. The latest possible start time is computed as difference between the critical path of the graph and the longest path from the current task to any exit task. A path length is the sum of the execution times and the communication costs of the tasks and edges belonging to the path. A task with the smallest latest possible start time has the highest priority. Ties are broken considering the priorities of the task's descendents. The tasks are selected in the order of their priorities and scheduled on the processor that can execute it the earliest. The time complexity of MCP is $O(V^2(\log(V) + P))$.

MCP is relatively fast compared with other one-step scheduling algorithms. Furthermore, its scheduling performance is shown to be superior compared to most of the other algorithms for bounded number of processors [5, 8]. MCP can be modified to run faster by choosing a random tie breaking scheme at a negligible loss of performance. In this case, the time complexity is reduced to $O(V \log(V) + (E+V)P)$.

3.2 ETF

ETF (Earliest Task First) is a one-step scheduling algorithm in which at each iteration, the ready task that can start the earliest is scheduled. The earliest starting task is determined by tentatively scheduling all the ready tasks to all processors. The task with the minimum starting time is selected and scheduled to the corresponding processor on which it starts the earliest. The time complexity of ETF is $O(W(E+V)P)$.

ETF has a high complexity because at each iteration, it is required to compute the start times of each task for every processor scheduling. The main idea behind ETF is to keep the processors busy, in this respect being close to a load balancing scheme. Its scheduling performance is shown to be comparable with MCP's performance [5].

3.3 DSC-LLB

DSC-LLB is a multi-step scheduling algorithm composed of DSC (Dominant Sequence Clustering) and LLB (List-Based Load Balancing). The first step, using DSC, is intended to minimize communication by grouping the highly communicating tasks together in clusters. The second step, using LLB, maps the clusters to the existing processors and orders the tasks within the clusters.

In the DSC algorithm, task priorities are dynamically computed based on the sum of their *top level* and *bottom level*. The top level and bottom level are the sum of the computation and communication costs along the longest path from the given task to an entry task and an exit task, respectively. Again, the communication costs are assumed to be zero between two tasks mapped to the same processor. While the bottom level is statically computed at the beginning, the top level is computed incrementally during the scheduling process. The tasks are scheduled in the order of their priorities, where the largest sum of top and bottom levels takes priority. The destination processor is either the processor from which the last message arrives, or a new processor depending on which the given task can start earlier. The time complexity of DSC is $O((E+V)\log V)$.

In the LLB algorithm, a task is mapped to a processor if there is at least another task from the same cluster scheduled on that processor and not mapped otherwise. LLB is a load balancing scheme. First, the destination processor is selected as the processor that becomes idle the earliest. Second, the task to be scheduled is selected. There are two candidates: (a) a task already mapped to the selected processor having the least bottom level, or (b) an unmapped task with the least bottom level. The one starting the earliest is scheduled. The time complexity of LLB is $O((C \log(C) + V)$, where C is the number of clusters obtained in the clustering step.

DSC-LLB is a low-cost algorithm. Not surprisingly, compared to a higher cost scheduling algorithm as MCP, it has worse scheduling performance. However, the DSC-LLB output performance is still shown to be within 40% of the MCP output performance, while outperforming other known multi-step scheduling algorithms [8].

4 The FLB Algorithm

As mentioned earlier, list scheduling algorithms generally perform better compared to other scheduling algorithms for a bounded number of processors, such as multi-step methods [8]. Recently, a list scheduling algorithm, called FCP, has been proposed [7] whose time complexity is significantly reduced without sacrificing performance. For FCP, it has been proven that given a partial schedule and an arbitrary ready task, only *two* processors need to be considered to find the minimum start time of the given task. However, given the partial schedule and the set of ready tasks, the selected task need not always be the preferred one (i.e., the ready task that can actually start the earliest). In the FLB heuristic proposed in this paper, we improve this task selection by scheduling the ready task that can start the earliest. We prove that only two cases need to be considered to indeed select the preferred task and its corresponding processor. The scheduling process is therefore improved, while the low time complexity is maintained.

Note that FLB uses the same task selection criterion as in ETF. In contrast to ETF however, the preferred task is identified in $O(\log P)$ instead of $O(WP)$. Although essentially similar, there is a small difference in the task selection scheme. Throughout the scheduling process it may happen that several ready tasks can start at the same earliest time. ETF an FLB have different criteria to break this tie, and as a consequence in some cases they may still select a different task-processor pair to be scheduled.

Priority-based scheduling algorithms have three parts: (a) task priorities computation, (b) sorting tasks according to their priorities and (c) task scheduling. Computing task priorities takes at least $O(E+V)$ time, since the whole task graph has to be traversed. Sorting tasks takes $O(V \log W)$ time. The variety of scheduling approaches refers to the last part (c).

For most list scheduling algorithms, the task scheduling part takes $O((E+V)P)$ time complexity caused by the fact that all processors are considered to find the processor on which the current task can start the earliest. ETF [3] uses an even more elaborate approach. At each iteration, it schedules the ready task that can start the earliest to the processor on which this earliest start time is achieved. However, the complexity is $O(W)$ higher ($O(W(E+V)P)$), caused by the fact that at each iteration all the tasks need to be considered in the selection process. In [7] has been proven that processor selection in list scheduling algorithms can be performed at a considerable lower time complexity of $O(V \log{(P)} + E)$. In this paper we go one step further and prove that even the stronger task selection criterion used in the ETF scheduling algorithm can also be achieved at the same cost of $O(V \log{(P)} + E)$.

4.1 Algorithm Description

In FLB, at each iteration of the algorithm, the ready task that can start the earliest is scheduled to the processor on which that start time is achieved. To select the earliest starting task, pairs of a ready task and the processor on which it starts the earliest need to be considered.

In order to obtain the earliest start time of a given ready task on a partial schedule, the given task must be scheduled either (a) to the processor from which the last message is sent from or (b) to the processor becoming idle the earliest (as proven in [7]). An informal explanation is that if the processor the last message is sent from is idle before that message is received, then the start time of the task can be improved by scheduling the task on that processor. If that processor is busy at the time the last message arrives, then the task cannot start earlier than the last message arrival time which ever processor is selected. Consequently, the task will start the earliest on the processor becoming idle the earliest. It follows that an EP type task starts the earliest on its enabling processor and a non-EP type task start the earliest on the processor becoming idle the earliest.

Given a partial schedule, there are only two pairs task-processor that can achieve the minimum start time for a task: (a) the EP type task t with the minimum estimated start time $EST(t, EP(t))$ on its enabling processor, and (b) the non-EP type task t' with the minimum last message arrival time $LMT(t')$ on the processor becoming idle the earliest. The first case minimizes the earliest start time of the EP type tasks, while the second case minimizes the earliest start time of the non-EP type tasks. If in both cases the same earliest start time is obtained, the non-EP type task is preferred, because the communication caused by the messages sent from the task's predecessors are already overlapped with the previous computation. Considering the two cases discussed above guarantees that the ready task with the earliest start time will be identified. A formal proof is given in the appendix.

For each processor, there are two sorted lists storing the EP type tasks for which the given processor is the enabling processor. The tasks in the first list (`EMT_EP_task_l`) are decreasingly sorted by the effective message arrival time on their enabling processor ($EMT(t, EP(t))$). The tasks in the second list (`LMT_EP_task_l`) are decreasingly sorted by their last message arrival time ($LMT(t)$). The non-EP type tasks are also stored in a list (`nonEP_task_l`), decreasingly sorted by $LMT(t)$. For all three task lists, ties are broken by selecting the task with the longest path to any exit tasks.

There is a list of active processors (`active_proc_l`), decreasingly sorted by the minimum $EST(t)$ of the EP type tasks enabled by them. The minimum $EST(t)$ of the EP type tasks enabled by a processor is computed in $O(1)$ as the maximum between EMT of the first task in the processor's `EMT_EP_task_l` and the processor's PRT. A second global processor list (`all_proc_l`) stores all processors, decreasingly sorted by the PRT.

The FLB algorithm is formalized below.

```
FLB ()
BEGIN
  FORALL t ∈ V DO
    IF t is an entry task THEN
      SetEMT (t, 0)
      SetLMT (t, 0)
      Enqueue (nonEP_task_l, t, 0)
    END IF
  END FORALL
  FORALL p ← 0 TO P-1 DO
      Enqueue (all_proc_l, p, 0)
    END IF
  END FORALL
  WHILE NOT all tasks scheduled DO
    (t,p) ← ScheduleTask();
    UpdateTaskLists (t, p)
    UpdateProcLists (t, p)
    UpdateReadyTasks (t, p)
  END WHILE
END
```

At the beginning, the set of ready tasks consists of the entry tasks. None of them satisfies the EP type condition. Consequently, they are stored in the non-EP type task list. Moreover, as there are no EP type tasks, there is no active processor and the active processor list is empty. All the processors have the ready time 0, so all of them have the same priority, 0, in the global processor list.

The scheduling loop is repeated as long as there exist unscheduled tasks. At each iteration, one task is scheduled using `ScheduleTask`. After each task scheduling, the task lists, the processor lists and the ready task set must be updated, according to the new conditions following the last scheduling step.

The four procedures FLB is based on `ScheduleTask`, `UpdateTaskLists`, `UpdateProcLists` and `UpdateReadyTasks` are described below.

```
ScheduleTask ()
BEGIN
  p1 ← Head (active_proc_l)
  IF p1 ≠ NULL THEN
    t1 ← Head (EMT_EP_task_l[p1])
  END IF
  p2 ← Head (all_proc_l)
  t2 ← Head (nonEP_task_l)

  IF (p1 ≠ NULL) ∧ (EST(t1,p1) < EST(t2,p2)) THEN
    Schedule task t1 on processor p1
    Dequeue (active_proc_l)
    Dequeue (EMT_EP_task_l[p1])
    RemoveItem (LMT_EP_task_l[p1], t1)
    RETURN (t1, p1)
  ELSE
    Schedule task t2 on processor p2
    Dequeue (all_proc_l)
    Dequeue (nonEP_task_l)
    RETURN (t2,p2)
  END IF
END
```

The `ScheduleTask` procedure selects two task-processor pairs for the scheduling step. The first pair consists of the EP type task with the minimum EMT time and its enabling processor. The second pair is the non-EP task with the minimum LMT and processor becoming idle the earliest. The pair which achieves the minimum start time is selected and the selected task is scheduled to the selected processor. Scheduling a task also implies assigning to it its corresponding processor, start time and finish time. The `Dequeue` and `RemoveItem` procedures are used to remove the heads and the elements from the sorted lists, respectively.

```
UpdateTaskLists (t,p)
BEGIN
  WHILE true DO
    t ← Head (LMT_EP_task_l[p])
    IF t = NULL THEN
      BREAK
    END IF
    IF LMT(t) ≥ PRT(p) THEN
      BREAK
    Dequeue (LMT_EP_task_l[p])
    RemoveItem (EMT_EP_task_l[p], t)
    Enqueue (nonEP_task_l, t, LMT(t))
  END WHILE
END
```

After scheduling a task to a processor, the processor's ready time changes. Consequently, some of the EP type tasks enabled by that processor may no longer satisfy the condition to be of EP type. `UpdateTaskLists` moves the tasks that no longer satisfy the EP type condition to the non EP type task list. The tasks are tested in the order of their LMT. Finally, if there are no more EP type tasks enabled by the given processor, the processor is removed from the active processor list.

```
UpdateProcLists (t,p)
BEGIN
  t ← Head (EMT_EP_task_l[p])
  IF t = NULL THEN
    RemoveItem (active_proc_l,p)
  ELSE
    prio ← EST(t,p)
    BalanceList (active_proc_l, p, prio)
  END IF
END
```

`UpdateProcLists` updates the active processor list. If there is no EP type task enabled by the given processor, the processor must be removed from the active processor list. If there are still EP type tasks enabled by the given processor they may not be the same and as a consequence the processor priority in the active processor list could be modified. Therefore, the processor priority and the active processor list must be updated. Also, the global processor list must be updated because the current processor has changed the time it becomes idle.

```
UpdateReadyTasks (t,p)
BEGIN
  FORALL t' ∈ Succ(t) DO
    IF Ready(t') THEN
      p' ← GetEnablingProc (t')
      ComputeLMT (t')
      ComputeEMT (t',p')
      IF LMT(t') < PRT(p') THEN
        Enqueue (nonEP_task_l, t', LMT(t'))
      ELSE
        IF Head(EMT_EP_task_l[p']) = NULL THEN
          prio ← EST(t',p')
          Enqueue (active_proc_l, p', prio)
        ELSE
          t'' ← Head(EMT_EP_task_l[p'])
          IF EMT(t',p') < EMT(t'',p') THEN
            prio ← maxEMT(t'),PRT(p')
            BalanceList (active_proc_l, p, prio)
          END IF
        END IF
        Enqueue (EMT_EP_task_l[p'],t',EMT(t',p'))
        Enqueue (LMT_EP_task_l[p'],t',LMT(t'))
      END IF
    END IF
  END FORALL
END
```

`UpdateReadyTasks` adds the tasks that become ready due to the current task scheduling to one of the ready task lists, depending if it is of type EP or not. Moreover, if the task is of type EP, its enabling processor may become active and it must be added to the active processor list. Also, if the processor is already active and its priority changes, the active processor list must be updated.

4.2 Complexity Analysis

The complexity of the FLB algorithm is as follows. To compute task priorities throughout the algorithm, all the tasks and dependencies must be considered once. Consequently, the total complexity of computing task priorities takes $O(E+V)$ time.

Scheduling a task involves a comparison between two tentative task scheduling operations. The decision is taken based

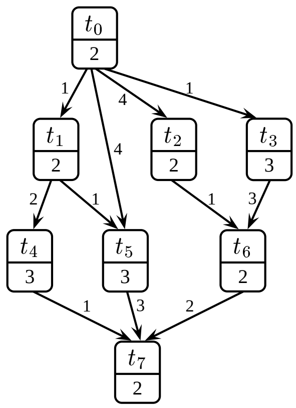

Figure 1. Example task graph

EP tasks		non-EP	Scheduling
on p_0	**on** p_1	**tasks**	$t \rightarrow p$,
$t[EMT(t,p), BL/LMT(t)]$		$t[LMT(t)]$	$[ST(t) - FT(t)]$
$-$	$-$	$t_0[0]$	$t_0 \rightarrow p_0, [0-2]$
$t_3[2, 12/3]$ $t_1[2, 11/3]$ $t_2[2, 9/6]$	$-$ $-$ $-$	$-$ $-$ $-$	$t_3 \rightarrow p_0, [2-5]$
$t_2[2, 9/6]$	$-$	$t_1[3]$	$t_1 \rightarrow p_1, [3-5]$
$t_2[2, 9/6]$ $t_5[6, 8/6]$	$t_4[5, 6/7]$ $-$	$-$ $-$	$t_2 \rightarrow p_0, [5-7]$
$t_6[7, 6/8]$	$t_4[5, 6/7]$	$t_5[6]$	$t_4 \rightarrow p_1, [5-8]$
$t_6[7, 6/8]$	$-$	$t_5[6]$	$t_5 \rightarrow p_0, [7-10]$
$-$	$-$	$t_6[8]$	$t_6 \rightarrow p_1, [8-10]$
$t_7[12, 2/13]$	$-$	$-$	$t_0 \rightarrow p_0, [12-14]$

Table 1. Execution trace of the FLB algorithm.

on the two estimated start times of the tasks tentatively scheduled. Computing the estimated start times takes $O(1)$ time, since the estimated start times are computed as the maximum of the task effective message arrival time on that processor on the one hand and the processor ready time on the other hand. These values are already computed at the time the decision is made. Since there are V tasks, the resulting task scheduling complexity is $O(V)$.

Task lists operations are adding and removing tasks from sorted lists. Every task is added and removed at most once from the two lists of EP type tasks and once from the non-EP type task list. Since there are at most W ready tasks at any moment in time, a list operation takes $O(\log W)$ time. As there are V tasks, the total complexity of maintaining task lists is $O(V \log W)$.

At each scheduling iteration, the global processor list changes, because the processor ready time changes for the destination processor of the scheduling. As a consequence, an operation $O(\log P)$ time complexity must be performed, which implies a total complexity for maintaining the global processor list of $O(V \log P)$.

After becoming idle or being scheduled, a task may change the active or passive status of a processor or its priority in the active processor list. As a consequence, a total complexity of $O(V \log P)$ is required to maintain the active processor list.

Finally, the cost of finding ready tasks takes $O(E+V)$ time, since all the tasks and edges must be scanned throughout the algorithm.

In conclusion, the total time complexity of the FLB algorithm is $O(V(\log(W) + \log(P)) + E)$.

5 Execution Trace

In this section, we illustrate the steps of the FLB algorithm by scheduling the task graph in Fig. 1 on two processors. The execution trace of the FLB algorithm is presented in Table 1. The first two columns in the table list the EP type tasks enabled by p_0 and p_1, respectively, as they are sorted in the corresponding lists. The task's EMT and bottom level are included to illustrate their sorting order, as well as their LMT to motivate their EP type. The third column in the table lists the non-EP type tasks sorted by LMT. 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. It is of non-EP type as it has no enabling processor. Task t_0 is scheduled on one randomly selected processor p_0, as all the processors have zero workload.

After scheduling t_0, three tasks become ready, namely t_1, t_2 and t_3. As there are no non-EP type tasks, the active processor accommodating the earliest starting EP type task is selected, in this case being the only active processor p_0. The EP type task with the higher priority is selected. Since the EMT is the same for all three tasks, the tie breaking mechanism stipulates that the task with the higher bottom level t_3 be selected and scheduled.

After scheduling t_3, t_1 becomes non-EP type, because $PRT(p_0)$ becomes larger than $LMT(t_1)$. The other ready task t_2 still satisfies the EP type condition. The two task-processor pairs to be compared for the next scheduling iteration are now t_2-p_0 and t_1-p_1. Processor p_1 is selected for the latter, because it is the processor becoming idle the earliest. Thus, t_1-p_1 is selected since $EST(t_1) = 3$ is lower than $EST(t_2) = 5$.

Two other tasks become idle: t_4 and t_5, both of type EP, enabled by p_1 and p_0, respectively. There are no non-EP tasks, so the first starting EP type task t_2 is selected and scheduled to its enabling processor p_0.

Task t_5 becomes non-EP type and is moved to the non-EP type task list. Task t_6 becomes ready and of EP type, so it is stored to the EP task list of its enabling processor p_0. The first starting EP type task is t_4 which starts on p_1 at 5. The non-EP task with the lowest LMT, t_5, starts on the processor becoming idle the earliest, p_0, at 6. Consequently, t_4 is scheduled on p_1 at 5.

At the next iteration both EP type and non-EP type tasks, namely t_6 and t_5, respectively, start at the same time: 7. The non-EP type task is preferred and scheduled on p_0.

In each of the next two iterations, no choice need be made as there is only one task to be scheduled. First t_6 is scheduled on p_1 at 8, then t_7 is scheduled on $p0$ at 12.

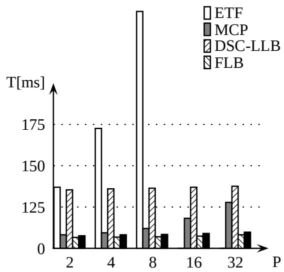

Figure 2. Scheduling algorithm costs

6 Performance Results

The FLB algorithm is compared with three algorithms, namely MCP, ETF, and DSC-LLB. The three algorithms are well-known, use different scheduling schemes, and were shown to obtain competitive results [5, 8, 11]. We selected the lower-cost version of MCP, in which if there are more tasks with the same priority, the task to be scheduled is chosen randomly.

We consider task graphs representing various types of parallel algorithms. The selected problems are *LU decomposition* ("LU"), *Laplace equation solver* ("Laplace") and a *stencil algorithm* ("Stencil"). For each of these problems, we adjusted the problem size to obtain task graphs of about $V = 2000$ nodes. 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 5 graphs with random execution times and communication delays (i.i.d., uniform distribution with unit coefficient of variation). A larger set of problems and granularities has been used to measure the FLB's performance in [6].

6.1 Running Times

Our main objective is to reduce task scheduling cost (i.e., running time), while maintaining performance. In Fig. 2 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. ETF is the most costly among the compared algorithms. Its running time increases from 185 ms for $P = 2$ up to 2.6 s for $P = 32$. MCP also increases its running times with the number of processors, but its cost is significantly lower. For $P = 2$, it runs for 41 ms, while for $P = 32$, the running time is 139 ms. DSC-LLB does not vary significantly with P, as its most costly step, clustering, is independent of the number of processors. The DSC-LLB running times vary around 180 ms.

FCP achieves the lowest running time, varying from 33 ms for $P = 2$ to 41 ms for $P = 32$. FLB running time is at the same level compared to FCP. It increases from 38 ms for $P = 2$ up to 49 ms for $P = 32$. The FLB's actual running times are comparable to FCP's running times, despite the fact that the FLB's complexity of $O(V(\log{(W)} + \log{(P)}) + E)$ is higher compared to FCP's complexity of $O(V \log{(P)} + E)$. This is due to the fact that usually, the width of the task graph has reasonable low values.

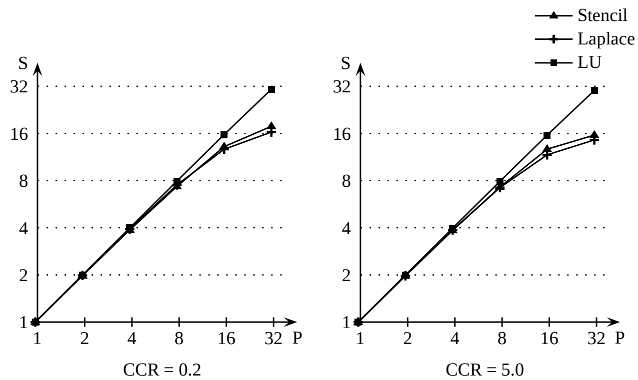

Figure 3. FLB speedup

6.2 Scheduling Performance

In Fig. 3 we show the FLB speedup for the considered problems. For all the problems, FLB obtains significant speedup. There are two classes of problems: (a) Stencil and FFT which achieve linear speedup and (b) LU and Laplace for which there less speedup is obtained for large number of processors. Stencil and FFT are more regular. Therefore more parallelism can be exploited and better speedup is obtained. For LU and Laplace, there are a large number of join operations. As a consequence, there is not much parallelism available and the speedup is lower.

To compare the FLB algorithm with the other algorithms, we use the *normalized schedule lengths* (NSL), which is defined as the ratio between the schedule length of the given algorithm and the schedule length of MCP.

In Fig. 4 we show the average normalized schedule lengths for the selected algorithms and problems. For each of the considered CCR values a set of NSL values is presented.

MCP and ETF consistently yield relatively good schedules. Depending on the problem and granularity, either one or the other performs better. The differences become larger for fine-grained problems, which are more sensitive to scheduling decisions. For LU, MCP schedules are up to 23% better, while for Laplace, ETF schedules are up to 5% better. For coarse-grain problems, the results are comparable.

DSC-LLB is a multi-step method intended to obtain acceptable results while aiming for minimum complexity. Its scheduling performance is not much worse compared to MCP and ETF. Typically, the schedule lengths are no more than 20% longer than the MCP and ETF's schedule lengths. Although in some cases the difference can be higher (up to 42%) there are also cases in which DSC-LLB performs better (up to 10%) compared to a list scheduling algorithm.

FCP has both low-cost and good performance. FCP obtains comparable performance with MCP, because it uses the same scheduling criteria as MCP, but in a more efficient manner. Compared to ETF, the overall performance is also comparable. However the differences between FCP and ETF (up to 18%) are higher compared to the differences between FCP and MCP (5%). Finally, compared to DSC-LLB, FCP obtains consistently better performance.

FLB has both low-cost and good performance. FLB obtains

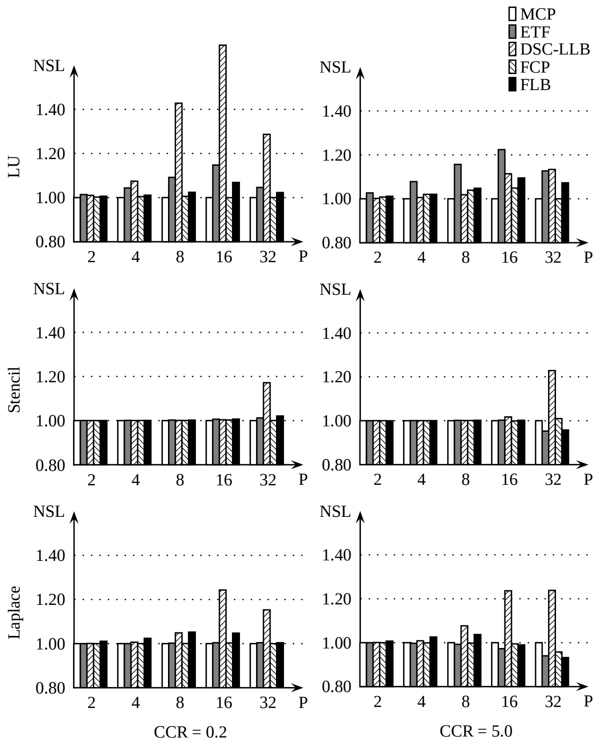

Figure 4. Scheduling algorithm performance

an overall comparable performance with ETF, MCP and FCP, and consistently outperform DSC-LLB. Compared to ETF, FLB performs better in most of the cases, up to 12%, despite they have the same task selection criterion. As explained earlier, this is due to the fact that ETF an FLB have different criteria to break ties. ETF uses statically computed task priorities. FLB uses message arrival times as task priorities and if the tie persists, the task bottom level. If there are more tasks starting at the same earliest time, the task with the highest priority is chosen. As the priorities are different, FLB and ETF may select a different task-processor pair to be scheduled. FLB performs better because its priority scheme is dynamic, therefore being more accurate.

Compared with MCP, FLB performs better or worse if ETF performs better or worse, respectively. FLB performs better than MCP for communication-intensive problems that have a regular structure (e.g., Stencil or Laplace). For LU, which involves many successive forks and joins, the relative performance of FLB compared to MCP is lower. The reason is that FLB, like ETF, does not consider future communication and computation when taking a scheduling decision, which in this case yields worse schedules. Finally, compared to DSC-LLB, FLB performs consistently better.

7 Conclusion

In this paper, a new scheduling algorithm, called Fast Load Balancing (FLB), is presented which is intended as a compile-time scheduling algorithm for distributed-memory systems. We prove that FLB uses the same criterion for task selection and scheduling as ETF, which is to schedule at each iteration the task starting the earliest. However, FLB's time complexity of $O(V(\log(W)+\log(P))+E)$ is drastically lower compared to ETF's $O(W(E+V)P)$ complexity.

Experimental results indeed show that FLB obtains a comparable performance with ETF as well as MCP. Compared to ETF, FLB performs equal or better in most of the cases, up to 12% due to differences in the particular heuristic used to break ties with respect to ready tasks that can start at the same earliest time. FLB's tie breaking heuristic performs better because its priority scheme is dynamic, therefore being more accurate. Compared with MCP, FLB performs better or worse if ETF performs better or worse, respectively. Again, note that MCP's cost is much higher compared to FLB's cost. Furthermore, FLB has consistently better performance compared to DSC-LLB, which has a higher cost compared to FLB's cost.

References

[1] I. Ahmad and Y.-K. Kwok. A new approach to scheduling parallel programs using task duplication. In *Proc. ICPP*, pages 47–51, Aug. 1994.

[2] Y. C. Chung and S. Ranka. Application and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed-memory multiprocessors. In *Proc. Supercomputing*, pages 512–521, Nov. 1992.

[3] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. on Computing*, 18:244–257, Apr. 1989.

[4] B. Kruatrachue and T. G. Lewis. Grain size determination for parallel processing. *IEEE Software*, pages 23–32, Jan. 1988.

[5] Y.-K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *Proc. IPPS/SPDP*, pages 531–537, Mar. 1998.

[6] A. Rădulescu and A. J. C. van Gemund. FLB: Fast load balancing for distributed-memory machines. TR 1-68340-44(1999)03, Delft Univ. of Technology, Feb. 1999.

[7] A. Rădulescu and A. J. C. van Gemund. On the complexity of list scheduling algorithms for distributed-memory systems. In *Proc. ICS*, pages 68–75, June 1999.

[8] A. Rădulescu, A. J. C. van Gemund, and H.-X. Lin. LLB: A fast and effective scheduling algorithm for distributed-memory systems. In *Proc. IPPS/SPDP*, pages 525–530, Apr. 1999.

[9] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. PhD thesis, MIT, 1989.

[10] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconection-constrained heterogeneous processor architectures. *IEEE Trans. on Parallel and Distributed Systems*, 4(2):175–187, Feb. 1993.

[11] M.-Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. on Parallel and Distributed Systems*, 1(7):330–343, July 1990.

[12] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. on Parallel and Distributed Systems*, 5(9):951–967, Dec. 1994.

A Task Selection

In this appendix we prove that given a partial schedule and a set of ready tasks, the selection of a task starting the earliest is restricted to comparing only two cases. The two choices are either (a) the EP type task with the minimum estimated start time on its enabling processor, or (b) the non-EP task with the minimum last message arrival time on the processor becoming idle the earliest. The first choice clearly minimizes the EP type tasks' EST. The second choice is proven to minimize the non-EP type tasks' EST. Consequently, selecting from these two tasks the one starting the earliest guarantees that the earliest starting task is chosen.

The next lemma states that a non-EP type task cannot start earlier than its LMT. To prove this, two cases have to be considered. First, the start time on the task's enabling processor, which is greater than the processor ready time, cannot be greater than LMT due to the definition of the non-EP type task. Second, if the task is scheduled on any other processor than the enabling processor, the task's EMT will be equal to its LMT. Consequently, the task will start at LMT the earliest.

Lemma 1 *Let $t \in V$ be a non-EP type task. Then*

$$\forall p \in \mathcal{P}, \quad LMT(t) \leq EST(t,p)$$

Proof
Let (t', t) be the dependence corresponding to $EP(t)$. By definition of LMT it holds

$$LMT(t) = FT(t') + comm(t', t) \tag{1}$$

By definition of EMT it follows that

$$\forall p \neq EP(t), \quad FT(t') + comm(t', t) \leq EMT(t,p) \leq LMT(t) \tag{2}$$

From (1) and (2) it follows that

$$\forall p \neq EP(t), \quad EMT(t,p) = LMT(t) \leq EST(t,p) \tag{3}$$

By definition of a non-EP type task it follows that for $p = EP(t)$

$$LMT(t) < PRT(EP(t)) \leq EST(t, EP(t)) \tag{4}$$

From (3) and (4) it follows that

$$\forall p \in \mathcal{P}, \quad LMT(t) \leq EST(t,p) \tag{5}$$

□

We use the lemma to show that the start time of a non-EP type task t on a processor p is the maximum of $LMT(t)$ and $PRT(p)$. $LMT(t)$ can be used instead of $EMT(t,p)$, because LMT is always the greater of the two and as the previous lemma states, $LMT(t)$ is always less than the task start time for a non-EP type task.

Corollary 2 *Let $t \in V$ be a non-EP type task. Then*

$$\forall p \in \mathcal{P}, \quad EST(t,p) = \max\{LMT(t), PRT(p)\}$$

Proof
Combining Lemma 1 and TST definition we can state that

$$\forall p \in \mathcal{P}, \quad EST(t,p) = \max\{LMT(t), EMT(t,p), PRT(p)\} \tag{6}$$

From the definitions of LMT and EMT it follows that the EMT term can be removed as stated by

$$\forall p \in \mathcal{P}, \quad EST(t,p) = \max\{LMT(t), PRT(p)\} \tag{7}$$

□

The following theorem proves that the two task-processor pairs considered by the FLB algorithm are sufficient to find the earliest starting ready task. The first pair, consisting of the EP type task with the minimum $EST(t, EP(t))$ on its enabling processor, minimizes the start time of the EP type tasks. The second pair consists of the non-EP type task with the minimum $LMT(t)$ on the processor becoming idle the earliest. Corollary 2 proves that the start time for the non-EP tasks has two independent components, namely $LMT(t)$ and $PRT(p)$ which are both minimized by the way the task and processor are selected. As a consequence, the task-processor pair that determines the earliest start time can be found using only the two cases mentioned above.

Theorem 3 *Let $t \in V$ be a ready task and $p \in P$ a processor such that*

$$EST(t,p) = \min_{t_x \in \mathcal{V}, p_x \in \mathcal{P}} EST(t_x, p_x)$$

Then t and p are either (a) the EP type task t_1 with the minimum estimated start time ($EST(t_1, EP(t_1))$) on its enabling processor p_1, or (b) the non-EP type task t_2 with the minimum last message arrival time ($LMT(t_2)$) on the processor becoming idle the earliest.

Proof
(a) t_1 is the earliest starting EP type task, simply by its selection criterion.
(b) t_2 is the non-EP task with the lowest LMT and p_2 is the processor with lowest PRT. From Corollary 2 it follows that t_2 is the earliest starting non-EP type task if scheduled on p_2.

From (a) and (b) it follows that the task-processor pair that determines the earliest start time can be found using only the two cases mentioned above.
□