LLB: A Fast and Effective Scheduling Algorithm for Distributed-Memory Systems

Andrei Rădulescu Arjan J.C. van Gemund Hai-Xiang Lin
Faculty of Information Technology and Systems
Delft University of Technology
P.O.Box 5031, 2600 GA Delft, The Netherlands

Abstract

This paper presents a new algorithm called List-based Load Balancing (LLB) for compile-time task scheduling on distributed-memory machines. LLB is intended as a cluster-mapping and task-ordering step in the multi-step class of scheduling algorithms. Unlike current multi-step approaches, LLB integrates cluster-mapping and task-ordering in a single step. The benefits of this integration are twofold. First, it allows dynamic load balancing in time, because only the ready tasks are considered in the mapping process. Second, communication is also considered, as opposed to algorithms like WCM and GLB. The algorithm has a low time complexity of $O(E+V(\log V+\log P))$, where E is the number of dependences, V is the number of tasks and P is the number of processors. Experimental results show that LLB outperforms known cluster-mapping algorithms of comparable complexity, improving the schedule lengths up to 42%. Furthermore, compared with LCA, a much higher-complexity algorithm, LLB obtains comparable results for fine-grain graphs and yields improvements up to 16% for coarse-grain graphs.

1 Introduction

The problem of efficiently scheduling programs is one of the most important and difficult issues in a parallel processing environment. The goal of the scheduling problem is to minimize the parallel execution time of a program. Except for very restricted cases, the scheduling problem has been shown to be NP-complete [3]. For shared-memory systems, it has been proven that even a low-cost scheduling heuristic is guaranteed to produce acceptable performance [4]. However, for distributed-memory architectures, there is no such guarantee and the task scheduling problem remains a challenge, especially for algorithms where low cost is of key interest.

The heuristic algorithms used for compile-time task scheduling on distributed-memory systems can be divided into (a) scheduling algorithms for an *unbounded* number of processors (e.g., DSC [12], EZ [9], STDS [2] and DFRN [6]) and (b) scheduling algorithms for a *bounded* number of processors (e.g., MCP [10], ETF [5] and CPFD [1]). The problem in the unbounded case is easier, because the constraint on the number of processors need not be considered. Therefore, scheduling in the unbounded case can be performed with good results at a lower cost compared to the bounded case. However, in practical situations, the necessary number of processors requested by an algorithm for the unbounded case is rarely available.

Recently, a multi-step approach has been proposed, which allows task scheduling at the low complexity of the scheduling algorithms for the unbounded case [9, 11]. In the first step, called *clustering*, scheduling without duplication is performed for an unbounded number of processors, tasks being grouped in clusters which are placed on virtual processors. In the second step, called *cluster-mapping*, the clusters are mapped on the available processors. Finally, in the third step, called *task-ordering*, the mapped tasks are ordered on processors. Although the multi-step approach has a very low complexity, previous results have shown that their schedule lengths can be up to twice the length of a list scheduling algorithm such as MCP [7].

This paper presents a new algorithm, called List-based Load Balancing (LLB) for cluster-mapping and task-ordering. It significantly improves the schedule lengths of the multi-step approach to a level almost comparable to the list scheduling algorithms, yet retaining the low complexity of the scheduling algorithms for an unbounded number of processors. The LLB algorithm combines the cluster-mapping and task-ordering steps into a single one, which allows a much more precise tuning of the cluster-mapping process.

This paper is organized as follows: Next section describes the scheduling problem and introduces some definitions used in the paper. Section 3 briefly reviews existing

cluster-mapping algorithms. Section 4 presents the LLB algorithm, while Section 5 shows its performance. Section 6 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. The *computation cost* of a task t, $comp(t)$, is the cost of executing the task on any processor. The edges correspond to task dependencies (communication messages or precedence constraints). The *communication cost* of an edge (t, t'), $comm(t, t')$, is the 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.

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. A ready task can start its execution only after all its dependencies have been satisfied. If two tasks are mapped to the same processor or cluster, the communication cost between them is assumed to be zero. The number of clusters obtained in the clustering step is C.

As a distributed system, we assume a P processor homogeneous clique topology with no contention on communication operations and non-preemptive tasks execution.

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

The objective of the scheduling problem is to find a scheduling of the tasks in $\mathcal{V}$ on the target system such that the parallel completion time of the problem (schedule length) is minimized. The parallel completion time is defined as $T_{par} = \max_{t \in \mathcal{V}} FT(t)$.

For performance comparison, the *normalized schedule length* is used. In order to define the normalized scheduled length, first the *ideal completion time* is defined as the ratio between the sequential time (aggregate computation cost) and the number of processors: $T_{ideal} = T_{seq}/P$. The normalized schedule length is defined as the ratio between the actual parallel time (schedule length) and the ideal completion time: $NSL = T_{par}/T_{ideal}$. It should be noted that the ideal time may not always be achieved, and that the optimal schedule length may be greater than the ideal time.

3 Related Work

In this section, three existing cluster-mapping algorithms and their characteristics are described: *List Cluster Assignment* (LCA) [9], *Wrap Cluster Merging* (WCM) [11] and *Guided Load Balancing* (GLB) [7].

3.1 List Cluster Assignment (LCA)

LCA is an incremental algorithm, that performs both cluster-mapping and task-ordering in a single step. At each step, LCA considers the unmapped task with the highest priority. The task, along with the cluster it belongs to, is mapped to the processor that yields *the minimum increase of parallel completion time*. The parallel completion time is calculated as if each unmapped cluster would be executed on a separate processor. The time complexity of LCA is $O(PC(C + E))$.

One can note, that if each task would have been mapped to a separate cluster, LCA degenerates to a traditional list scheduling algorithm. If the clustering step produces large clusters, the time spent to map the clusters to processors is decreased, since all tasks in a cluster are mapped in a single step. However, the complexity of LCA is still high, because at each cluster-mapping step, it is necessary to compute the total parallel completion time P times.

3.2 Wrap Cluster Merging (WCM)

In WCM, cluster-mapping is based only on the cluster workloads. First, each of the clusters with a workload higher than the average is mapped to a separate processor. Second, the remaining clusters are sorted in increasing order by workload. Assuming the remaining processors are renumbered as $0, 1, \ldots Q-1$ and the remaining clusters as $0, 1, \ldots R-1$, the processor for cluster c_k is defined as $PE(c_k) = k \bmod Q, \quad k = 0, 1, \ldots R-1$. The time complexity of WCM is $O(C \log C + V)$.

Assuming that in the previous clustering step the effect of the largest communication delays are eliminated, the communication delays are not considered in the cluster-mapping step. The time complexity of WCM is very low. However, it can lead to load imbalances among processors at different stages during the execution. That is, the cluster-mapping is performed considering only the sum of the execution times of the tasks in clusters, irrespective of the tasks' starting times. As a result, processors may be temporarily overloaded throughout the time.

3.3 Guided Load Balancing (GLB)

GLB improves on WCM by exploiting knowledge about the task start times computed in the clustering step. A cluster start time is defined as the earliest start time of the tasks in the given cluster. The clusters are mapped in the order of their start times to the least loaded processor at that time. The time complexity of GLB is $O(C \log C + V)$.

The start time of the first task determines the cluster priority, thus providing information from the dependence analysis in the clustering step. Topologically ordering tasks yields a natural cluster ordering. Scheduling clusters in the order they become available for execution generally yields a better schedule compared to scheduling when only the workload in a cluster is considered, because the work in the clusters is spread over time. Using this approach, the concurrent tasks (without dependencies between them) are more likely to be placed on different processors and therefore run in parallel.

However, there are cases in which GLB still does not produce the right load balance. For example, if a larger cluster is mapped on a processor, it will significantly increase the aggregate load of that processor and cause that processor not to be considered in the next mappings, even if it is already idle.

In [8] a graphical illustration is given of how the above algorithms perform on an example task graph.

4 The LLB Algorithm

4.1 Motivation

Comparing the existing low-cost multi-step scheduling algorithms with higher-cost scheduling algorithms, it can be noticed that low-cost multi-step algorithms produce up to 100% longer schedules compared to higher-cost scheduling algorithms like MCP [7]. The existing low-cost cluster-mapping algorithms (e.g., WCM, GLB) aim to balance the workload of the clusters on processors. Neither the communication costs, nor the order imposed to tasks by their dependencies are considered when the clusters are mapped. On the other hand, the higher cost cluster-mapping algorithms (e.g. LCA), despite their better performance, are less attractive in many practical cases, because of their high cost.

LLB is a low-cost algorithm intended to improve load balance throughout the time by performing cluster-mapping and task-ordering in a single step. Integrating the two steps in a single one allows a better tracking of the running tasks throughout the time when a new mapping decision is made. This approach helps the scheduling process in two ways. First it allows a dynamic load balancing throughout the time, because only the ready tasks are considered in the mapping process. Second, communication costs are also considered when selecting tasks, as opposed to algorithms such as WCM and GLB.

Similar to the other algorithms presented earlier, LLB maps all tasks from a cluster in a single step. However, instead of selecting the best processor for a given cluster, the algorithm selects the best cluster for a given processor. The selected processor is the first one becoming idle. This new approach is taken because it significantly decreases the complexity of the algorithm. Instead of scanning all processors to find the best mapping for the cluster ($O(PV)$), the selected processor is simply determined by identifying the minimal processor ready time ($O(\log P)$). The tasks are scheduled one at a time, which implies task-ordering in the same step as cluster-mapping. If the selected task has not been mapped before, all other tasks in its cluster are mapped along with the selected task. Scheduling tasks one at a time allows better control of the scheduling throughout the time, comparable to list scheduling algorithms, therefore leading to better schedules.

4.2 The LLB Algorithm

Before describing the LLB algorithm, we introduce few concepts on which the algorithm is based. In this paper there is a clear distinction between the concept of a mapped and a scheduled task. A task is *mapped*, if its destination processor has been assigned to it. A task is *scheduled* if it is mapped *and* its start time has been computed. A *ready* task is an unscheduled task that has all its direct predecessors scheduled. The *urgency* of a task is defined as a static priority, in the same way as in list scheduling algorithms. We use as the task urgency the sum of the computation and the inter-cluster communication on the longest path from the given task to any exit task. This urgency definition provides a good measure of the task importance, because the larger the *urgency* is, the more work is still to be completed until the program finishes.

The LLB algorithm maps all tasks in the same cluster on the same processor. When the first task in a cluster is scheduled on a processor, all the other tasks in the same cluster are mapped on the same processor. The reason is to keep the communication at the same low level as obtained in the clustering step.

At each step one task is scheduled. First, the destination processor is selected as the processor becoming idle the earliest. Each processor has a list of ready tasks already mapped on it. The ready unmapped tasks are kept in a global list. Initially, the ready mapped task lists are empty and the ready unmapped task list contains the entry tasks. All ready task lists are sorted in descending order using the task urgencies.

After selecting the destination processor a task to be scheduled on that processor is selected. The candidates are the most urgent ready task mapped on the selected processor and the most urgent ready unmapped task. Between these two tasks, the one starting the earliest is selected and scheduled on the selected processor. If the two tasks start at the same time the mapped task is selected. If none of the two candidates exists, the most urgent task mapped on another processor is selected and scheduled on its own processor.

Finally, the ready task lists are updated. Scheduling a

task may lead to other tasks becoming ready. These ready tasks can be mapped or unmapped. The mapped tasks are added to the ready tasks lists corresponding to their processors. The unmapped tasks are added to the global ready unmapped task list.

```
LLB ()
BEGIN
  For each task compute urgencies.
  WHILE NOT all tasks scheduled DO
    p ← the processor becoming idle the earliest
    t_m ← the most urgent ready task mapped on p
    t_u ← the most urgent ready unmapped task
    SWITCH
      CASE t_m and t_u exist:
        IF ST(t_m,p) ≤ ST(t_u,p) THEN
          task ← t_m
        ELSE
          task ← t_u
        END IF
      CASE only t_m exists:
        task ← t_m
      CASE only t_u exists:
        task ← t_u
      CASE neither t_m nor t_u exist:
        t ← the most urgent ready task.
    END SWITCH
    Schedule t on p.
    IF t has not been mapped THEN
      Map all tasks in t's cluster on p
    END IF
    Add the new ready tasks to
    the ready task lists.
  END WHILE
END
```

The complexity of the LLB algorithm is as follows. Computing task priorities takes $O(E+V)$. At each task scheduling step, processor selection takes $O(\log P)$ time. Task selection requires at most two dequeue operations from the ready task lists, which takes $O(\log V)$ time. The cumulative complexity of computing the task start times throughout the execution of the while loop is $O(E+V)$, since all the edges of the task graph must be considered. Also, each new ready task in the task graph must be added to one of the ready task lists. Adding one task to a list takes $O(\log V)$ time and, as there are V tasks, $O(V \log V)$ time is required to maintain the ready task lists throughout the execution of the while loop. The resulting complexity of the LLB algorithm is therefore $O(V(\log V + \log P))$.

In [8] an elaborate description of the algorithm is presented including a sample execution trace and a comparison to related work.

5 Performance Results

The LLB algorithm is compared with the three algorithms described in Section 3, LCA, WCM and GLB. The algorithms are compared within a full multi-step scheduling method to obtain a realistic test environment. The algorithm used for the clustering step is DSC [12], because of both its low cost and good performance. In case of using WCM and GLB for cluster-mapping, we use RCP [11] for task-ordering. In our comparisons we also included the well-known list scheduling algorithm MCP [10] to obtain a reference to other type of scheduling algorithms.

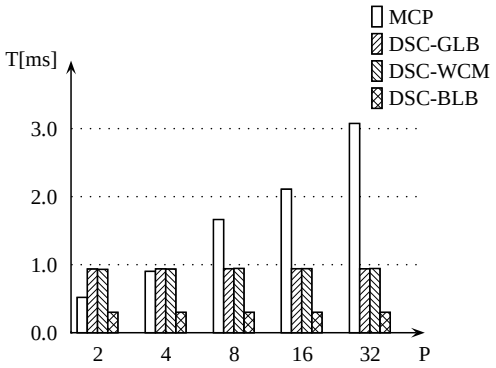

Figure 1. Algorithms running times

We consider task graphs representing various types of parallel algorithms. The selected problems are *LU decomposition*, *Laplace equation solver*, a *stencil algorithm* and *fast Fourier transform* (sample examples are shown in [8]). For each of these problems, we adjusted the problem size to obtain task graphs of about 2000 nodes. We varied the task graph granularities, by varying the communication to computation ratio (CCR). The values used for CCR were 0.2, 0.5, 1.0, 2.0 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 coefficient of variation 1)

One of our objectives is to observe the trade-offs between the performance (i.e., the schedule length), and the cost to obtain these results (i.e., the running time) required to generate the schedule. In Fig. 1 the average running time of the algorithms is shown. For the multi-step scheduling methods, the total scheduling time is displayed (i.e., the sum of clustering, cluster-mapping and task-ordering times). The running time of MCP grows linearly with the number of processors. For a small number of processors, the running time of MCP is comparable with the running times of the three low-cost multi-step scheduling methods (DSC-WCM, DSC-GLB and DSC-LLB). However, for a larger number of processors, the running time of MCP is much higher. All three low-cost multi-step scheduling methods (DSC-WCM, DSC-GLB and DSC-LLB) have comparable small running times, which do not vary significantly with the number of processors. DSC-LCA is not displayed, because its running times are much higher compared to the other running times, varying from 82 seconds for $P = 2$ to 13 minutes for $P = 32$.

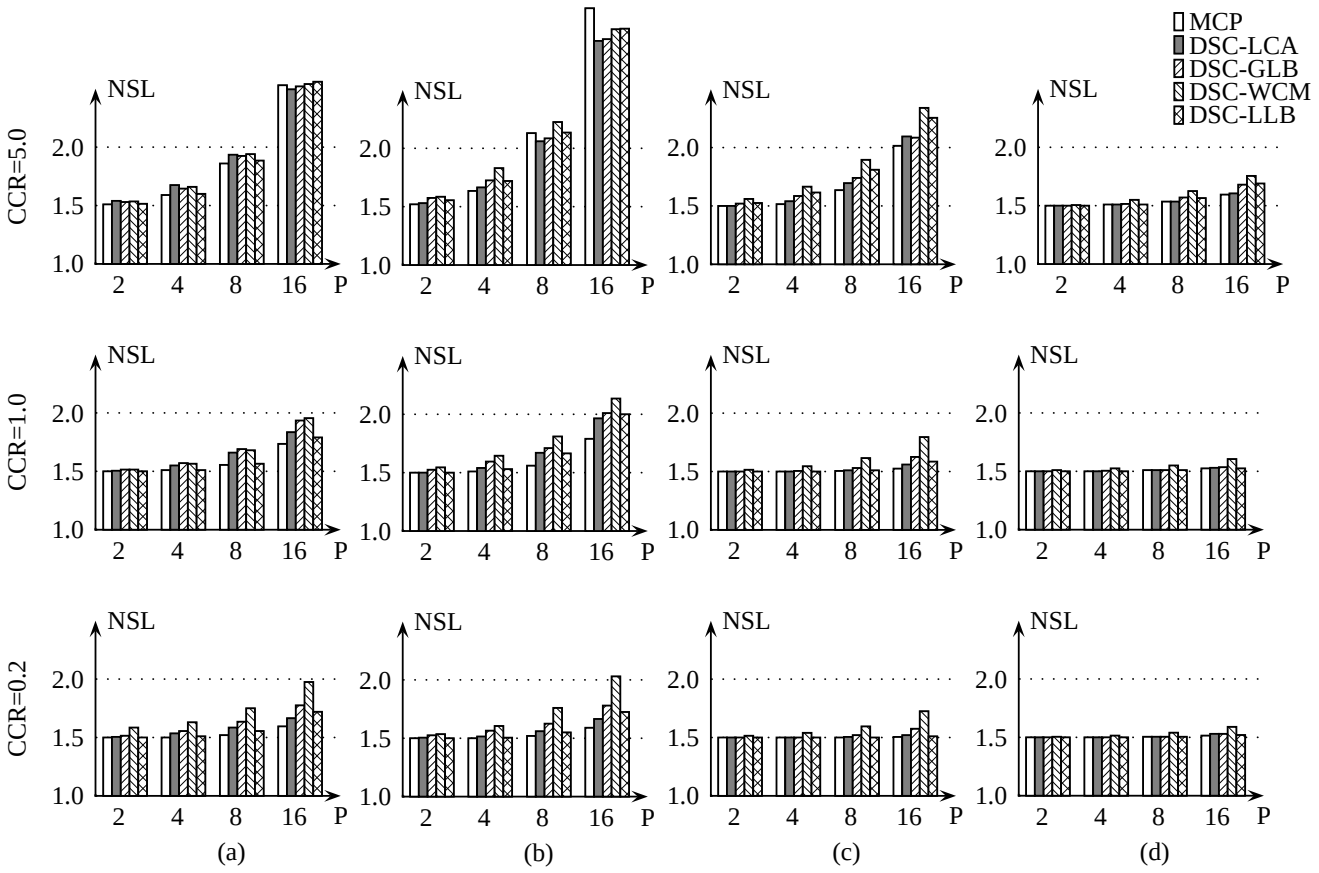

Figure 2. Normalized scheduling lengths for: (a) LU, (b) Laplace, (c) stencil (d) FFT

The average normalized schedule lengths (defined in Section 2) for the selected problems are shown in Figure 2 for CCR values 0.2, 1.0 and 5.0 only (more measurements are presented in [8]). For each of the considered CCR values a set of NSL values is presented. Note that the NSL generally increases with P as a result of the limited parallelism in the task graphs.

Within the class of high-cost algorithms, the schedule lengths of DSC-LCA are still longer compared to MCP. In a multi-step approach the degree of freedom in mapping tasks to processors is decreased by clustering tasks together. Mapping a single task from a cluster to a processor forces all the other tasks within the same cluster to be mapped to the same processor. In this case, using a higher-cost algorithm (DSC-LCA vs. MCP) does not imply an increase in scheduling quality.

While for small number of processors the performance of the DSC-WCM is comparable with the high-cost algorithms, for a large number of processors the performance drops. If the computation time dominates, the difference in the quality of the schedules is increasing in favor of high-cost algorithms, going up to a factor of 2 in some cases (row LU decomposition, $CCR = 0.2$, $P = 16$). In the case of fine-grain task graphs, DSC-WCM obtains results comparable with high-cost algorithms. Minimizing the communication delays in the clustering step is the key factor in obtaining these results.

DSC-GLB obtains better schedules compared with DSC-WCM, since more information is used from the clustering step. However, the quality increase varies both with CCR and with the type of problem. For high values of CCR (coarse-grain task graphs), the improvements in schedule lengths over DSC-WCM are $0 - 20\%$ for stencil problems and $0 - 6\%$ for LU decomposition, Laplace equation solver and FFT. For small values of CCR, the improvements over DSC-WCM are higher ($2 - 12\%$ for Laplace equation solver and FFT, and $5 - 25\%$ for LU decomposition and stencil problems).

DSC-LLB outperforms both DSC-WCM and DSC-GLB, while maintaining the cost at a low level. It consistently outperforms DSC-WCM for all type of problems with $0 - 10\%$ for fine-grain graphs and $1 - 42\%$ for coarse-grain graphs ($1 - 13\%$ for FFT, $3 - 42\%$ for Laplace equation solver and stencil problems, and $17 - 35\%$

for LU decomposition)

Compared with DSC-GLB for fine-grain graphs, DSC-LLB generally performs comparable or better if there is still speedup to be obtained (small number of processors). However, in some cases (Laplace equation solver), the performance is somewhat lower. For coarse-grain graphs, DSC-LLB consistently outperforms DSC-GLB with $0-14\%$. Moreover, in many cases DSC-LLB even outperforms DSC-LCA, with up to 16% (LU decomposition, $CCR = 1.0$, $P = 8$).

Compared to the more expensive MCP algorithm, DSC-LLB generally obtains longer schedules. As LLB is similar to list scheduling algorithms, its behavior is similar to MCP. The increase in schedule length depends on the CCR values and the type of problem, but does not exceed 36% (stencil problem, $CCR = 2.0$, $P = 16$).

6 Conclusion

In this paper, a new algorithm, called List-based Load Balancing (LLB), is presented. LLB is intended as a cluster-mapping and task-ordering step in the multi-step class of scheduling algorithms. Unlike the current approaches, LLB integrates cluster-mapping and task-ordering in a single step, which improves the scheduling process in two ways. First it allows a dynamic load balancing in time, because only the ready tasks are considered in the mapping process. Second, the communication is also considered when selecting tasks, as opposed to algorithms like *WCM* and *GLB*.

The $O(E + V(\log V + \log P))$ complexity of LLB does not exceed the complexity of a low-cost clustering algorithm, like DSC. Thus, the low complexity of the multi-step approach remains unaffected, despite our improvements in performance.

Experimental results show that compared with known cluster-mapping algorithms of low-complexity, LLB algorithm improves the schedule lengths up to 42%. Compared with LCA, a much higher-complexity algorithm, LLB obtains comparable results for fine-grain task graphs and even better results for coarse-grain task graphs, yielding improvements up to 16%. In conclusion, LLB outperforms other algorithms in the same low-cost class and even matches the better performing, higher-cost algorithms in the list scheduling class.

Acknowledgements

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.

References

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

[2] S. Darbha and D. P. Agrawal. Scalable scheduling algorithm for distributed memory machines. In *Proc. of the SPDP*, Oct. 1996.

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

[4] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. on Applied Mathematics*, 17(2):416–429, Mar. 1969.

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

[6] G.-L. Park, B. Shirazi, and J. Marquis. DFRN: A new approach for duplication based scheduling for distributed memory multiprocessor system. In *Proc. of the IPPS*, pages 157–166, Apr. 1997.

[7] A. Rădulescu and A. J. C. van Gemund. GLB: A low-cost scheduling algorithm for distributed-memory architectures. In *Proc. of the HiPC*, Dec. 1998. 294–301.

[8] A. Rădulescu, A. J. C. van Gemund, and H.-X. Lin. LLB: A fast and effective scheduling algorithm for distributed-memory systems. Technical Report 1-68340-44(1998)11, Delft Univ. of Technology, Dec. 1998.

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

[10] 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.

[11] T. Yang. *Scheduling and Code Generation for Parallel Architectures*. PhD thesis, Dept. of CS, Rutgers Univ., May 1993.

[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.