Fast and Effective Task Scheduling in Heterogeneous Systems

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

Recently, we presented two very low-cost approaches to compile-time list scheduling where the tasks' priorities are computed statically or dynamically, respectively. For homogeneous systems, these two algorithms, called FCP and FLB, have shown to yield a performance equivalent to other much more costly algorithms such as MCP and ETF. In this paper we present modified versions of FCP and FLB targeted to heterogeneous systems. We show that the modified versions yield a good overall performance, which is generally comparable to algorithms specifically designed for heterogeneous systems, such as HEFT or ERT. There are a few cases, mainly for irregular problems and large processor speed variance, where FCP and FLB's performance drops down to 32% and 63%, respectively. Considering the good overall performance and their very low cost however, FCP and FLB are interesting options for scheduling very large problems on heterogeneous systems.

Keywords: compile-time task scheduling, list scheduling, low-cost, heterogeneous systems

1 Introduction

Heterogeneous systems have recently become widely used as a cheap way of obtaining a parallel system. Clusters of workstations connected by high-speed networks, or simply the Internet are common examples of heterogeneous systems. However, in order to obtain high-performance from such a system, both compile-time and runtime support is necessary, in which scheduling the application to the parallel system is a crucial factor. The problem, known as task scheduling, has been shown to be NP-complete [3].

The general problem of task scheduling has been extensively studied, mainly for homogeneous systems. Various heuristics have been proposed, including list algorithms [4, 11, 12, 13, 20], multi-step algorithms [14, 15, 22], duplication based algorithms [7, 2, 1], genetic algorithms [18], algorithms using local search [21], bin packing [19], or graph decomposition [6]. Within all these approaches, list scheduling has been shown to have a good cost-performance trade-off, as considering its low cost, the performance is still very good [8, 13, 12]. The low-cost is a key issue for large problems, in which even a $O(V^2)$ algorithm, where V is the number of tasks, may have a prohibitive cost.

Task scheduling has also been studied in the specific context of heterogeneous systems ([5, 9, 10, 16, 17]). It has been shown that minimizing the tasks' completion time throughout the schedule is preferable to minimizing the tasks' start time [10, 17]. With respect to list scheduling algorithms, one can note that most of them can be easily modified to meet the task's completion time minimization criterion, and thus obtain good performance also in the heterogeneous case (e.g., HEFT [17] and ERT [9] are the versions using the tasks' completion time as the task priority of MCP [20] and ETF [4], respectively). However, two very low-cost list scheduling algorithms that we proposed recently, namely FCP (Fast Critical Path) [13] and FLB (Fast Load Balancing) [12], cannot be modified in such an easy way without sacrificing their competitively low cost.

In this paper we present the modifications required to obtain a good performance from FCP and FLB in heterogeneous systems. We show that the modified versions of FCP and FLB yield a good overall performance, which is generally comparable to algorithms specifically designed for heterogeneous systems, such as HEFT (Heterogeneous Earliest-Finish-Time) [17] and ERT (Earliest Task First) [9]. There are a few cases, mainly for irregular problems and wide processor speed ranges, in which FCP and FLB's performance drops down to 32% and 63%, respectively. Considering their very low cost and reasonably good performance, we believe that FCP and FLB are interesting options for task scheduling in heterogeneous systems, especially for large problems where scheduling time would otherwise be prohibitive.

This paper is organized as follows: The next two sections briefly describe the scheduling problem, and the FCP and FLB algorithms, respectively. In Section 4 we study their performance for heterogeneous systems. Section 5 concludes the paper.

2 Preliminaries

The task scheduling algorithm input is a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, that models a parallel program, 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 is assumed to have a *computation cost*. The edges correspond to task dependencies (communication messages or precedence constraints) and have a *communication cost*. The *communication-to-computation ratio* (CCR) of a parallel program is defined as the ratio between its average communication and computation costs. 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 the 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. The task's *bottom level* is defined as the longest path from the current task to any exit task, where the path length is the sum of the computation and communication costs of the tasks and edges belonging to the path. 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.

As a distributed system we assume a set $\mathcal{P}$ of P processors connected in a clique topology in which interprocessor communication is assumed to perform without contention. The processors' computing speeds differ and are represented as fractions of the slowest processor speed. We assume that the task execution time is proportional with the speed of the processor it is executed on, and consists of the computation cost multiplied by the processor speed.

In our algorithms, an important concept is that of the *enabling processor* of a ready task t, $EP(t)$, which is the processor from which the last message arrives. Given a partial schedule and a ready task t, the task is said to be of *type EP* if its last message arrival time is greater than the ready time of its enabling processor and of *type non-EP* otherwise. Thus, an EP type task starts the earliest on its enabling processor.

3 The Algorithms

List scheduling algorithms use two approaches to schedule tasks. The first category is the *static* list scheduling algorithms (e.g., MCP [20], DPS [11], HEFT [17], FCP [13]) that schedule the tasks in the order of their previously computed priorities. A task is usually scheduled on the processor that gives the earliest start time for the given task. Thus, at each scheduling step, first the task is selected and afterwards its destination processor.

The second approach is *dynamic* list scheduling (e.g. ETF [4], ERT [9], FLB [12]). In this case, the tasks do not have a precomputed priority. At each scheduling step, each ready task is tentatively scheduled to each processor, and the best <task, processor> pair is selected (e.g., the ready task that starts the earliest on the processor where this earliest start time is obtained for ETF, or the ready task that finishes the earliest on the processor where this earliest finish time is obtained for ERT). Thus, at each step both the task and its destination processor are selected at the same time.

Both static and dynamic approaches of list scheduling have their advantages and drawbacks in terms of the schedule quality they produce. Static approaches are more suited for communication-intensive and irregular problems, where selecting important tasks first is more crucial. Dynamic approaches are more suited for computation-intensive applications with a high degree of parallelism, because these algorithms focus on obtaining a good processor utilization.

FCP (Fast Critical Path) [13] and FLB (Fast Load Balancing) [12] significantly reduce the cost of the static and dynamic list scheduling approaches, respectively. In the next two sections, we describe both algorithms and we outline the differences between them and previous list scheduling algorithms.

3.1 FCP

Static list scheduling algorithms have three important steps: (a) *task priorities computation*, that takes at least $O(E + V)$ time, since the whole task graph has to be traversed, (b) *task selection* according to their priorities, that takes $O(V \log W)$ time, and (c) processor selection, that selects the "best" processor for the previously selected task, usually the processor where the current task starts/finishes the earliest. Processor selection takes $O((E + V)P)$ time, since each task is tentatively scheduled to each processor. Thus, the highest complexity steps are the task and processor selection steps, which determine the $O(V \log{(W)} + (E+V)P)$ time complexity of the static list scheduling algorithms

In FCP, the processor selection complexity is significantly reduced by restricting the choice for the destination processor from all processors to only *two* processors: (a) the task's enabling processor, or (b) the processor which becomes idle the earliest. In [13] we prove that the start time of a given task is minimized by selecting one of these two destination processors. The proof is based on the fact that the start time of a task t on a candidate processor p is defined as the maximum between (a) the time the last message to t arrives, and (b) the time p becomes idle. As the above-mentioned processors minimize the two components of the task's start time, respectively, it follows that one of the two processors minimizes the task's start time. Consequently, the algorithm's performance is not affected, while the time complexity is drastically reduced from $O((E+V)P)$ to $O(V \log{(P)} + E)$.

The task selection complexity can be reduced by maintaining only a *constant size* sorted list of ready tasks. Thus, we sort as many tasks as they fit in the fixed size sorted list, while the others are stored in an unsorted FIFO list which has an $O(1)$ access time. The time complexity of sorting tasks using a list of size H decreases to $O(V \log H)$ as all the tasks are enqueued and dequeued in the sorted list only once. We have found that for FCP, which uses bottom level as task priority, a size of P is required to achieve a performance comparable to the original list scheduling algorithm (see Section 4). A sorted list size of P results in a task sorting complexity of $O(V \log P)$.

Using the described techniques for task sorting and processor selection the total time complexity of FCP ($O(V \log{(P)} + E)$) is clearly a significant improvement over the time complexity of typical list scheduling approaches with statically computed priority.

3.2 FLB

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. Note that FLB uses the same task selection criterion as in ETF. In contrast to ETF however, the preferred task and its destination processor are identified in $O(\log{(W)} + \log{(P)})$ time instead of $O(WP)$.

To select the earliest starting task, pairs of a ready task and the processor on which the task starts the earliest need to be considered. As shown earlier, in order to obtain the earliest start time of a ready task on a partial schedule, the given task must be scheduled either (a) to the task's enabling processor, or (b) to 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

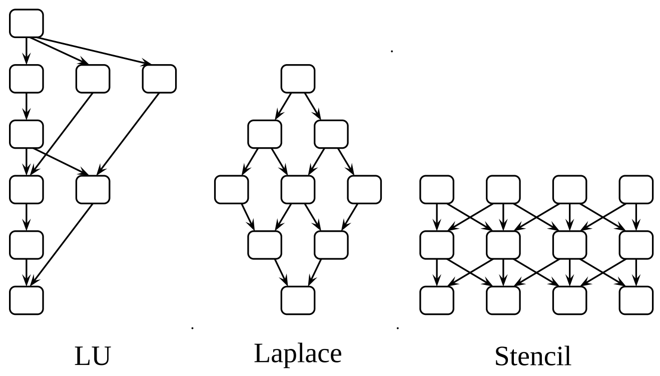

Figure 1: Miniature task graphs

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 [12].

To reduce the complexity even further, the same scheme as in FCP can be used. Instead of maintaining all EP and non-EP tasks sorted, only a fixed number of tasks are stored sorted, while the other are stored in FIFO order. The FLB's complexity is reduced to $O(V \log(P) + E)$, while the performance is maintained at a level comparable to using the fully sorted task lists (see Section 4).

3.3 The Modifications

As mentioned earlier, task scheduling algorithms for heterogeneous systems perform better when they sort tasks by their finish time rather than start time. The reason is that sorting by finish time implicitly takes into consideration processor speeds. However, in order to maintain their very low complexity, FCP and FLB must sort the tasks according to their start time. As a consequence, the processor speed is not considered when scheduling a non-EP task, but only the time the processors becomes idle.

To overcome this deficiency, we change the priority criterion for processors for both FCP and FLB. Instead of using the time the processor becomes idle the earliest as a priority, we now use the *sum* of the processor idle time and the mean task execution time. Using this priority scheme, we are now able to incorporate the processor speed when selecting the processor for a non-EP task. This is a raw approximation of finding the processor where a non-EP type task finishes the earliest.

In FLB, we also modify the task priority for the EP-type tasks. The EP-type tasks are sorted by their finish time on their enabling processor instead of their start time.

Finally, for both FCP and FLB, we change the final choice between the two candidate tasks, by selecting the task finishing the earliest instead of the task starting the earliest.

Note, that all these modifications of FCP and FLB do not involve any extra cost compared to the original versions. As a consequence, the cost of both FCP and FLB is maintained at the same very low level.

4 Performance Results

The FCP and FLB algorithms are compared with ERT (Earliest Task First) [9] and HEFT (Heterogeneous Earliest-Finish-Time) [17]. ERT ($O(W(E + V)P)$) and HEFT ($O(V \log W + (E + V)P)$) are well-known and have been shown to obtain competitive results in heterogeneous systems [9, 17].

For both FCP and FLB we used two versions. The first version uses *fully* sorted task lists. For this first version, FCP and FLB have exactly the same scheduling criteria as MCP and ETF, respectively. The second version uses

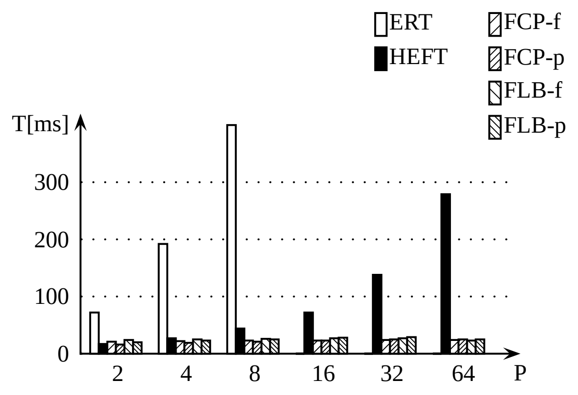

Figure 2: Cost comparison

partially sorted priority lists of size P. We call the first version of the algorithms FCP-f and FLB-f, and the second FCP-p and FLB-p, respectively.

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 2000 nodes. For each problem, we varied the task graph granularities, by varying the communication-to-computation ratio (CCR). The values used for CCR are 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), the results being the average over the 5 graphs (in view of the low overall variance, 5 samples are sufficient). Miniature task graphs samples of each type are shown in Figure 1.

We schedule the task graphs on 2, 4, 8, 16 and 32 processors. For each P, we use 10 heterogeneous configurations in which the processors' speed are uniformly distributed over the following intervals: $[8, 12]$, $[6, 14]$ and $[4, 16]$. Thus, the total number of test configurations is 3 (problems) $\times$ 2 (CCR) $\times$ 5 (sample graphs) $\times$ 5 (processor ranges) $\times$ 10 (processor configurations) $\times$ 3 (processor intervals) $= 5500$.

4.1 Running Times

In Fig. 2 the average running time of the algorithms is shown in CPU seconds as measured on a Pentium Pro/300MHz PC with 64Mb RAM running Linux 2.0.32. ERT is the most costly among the compared algorithms. Its cost increases from 72 ms for 2 processors up to 11 s for 64 processors (we do not include ERT's running times for $P \geq 16$ in Figure 2 due to their too much higher values). HEFT's cost also increases with the number of processors, but it is significantly lower. For $P = 2$, it runs for 17 ms, while for $P = 64$, the running time is 279 ms.

Both versions of the FCP and FLB have considerably lower running times. FCP-p's running time is the lowest, varying from 16 ms for $P = 2$ to 25 ms for $P = 64$. FCP-f varies from 21 ms for $P = 2$ to 24 ms for $P = 64$. One can note that for larger number of processors both versions of FCP have the same running times. The reason is that the ready tasks fit in the sorted part of the FCP-f's priority list.

FLB has a slightly higher cost compared to FCP, because of the more complicated task and processor selection schemes. The running times vary around 26 ms and 24 ms for FLB-f and FLB-p, respectively. Their running times do not vary significantly with the number of processors. One can note that for larger number of processors, FCP and FLB's running times tend to become similar.

4.2 Scheduling Performance

In this section we study how the FCP and FLB algorithms perform. We first compare FCP and FLB's performance to ERT and HEFT's performance, with respect to granularity, problem type and processor heterogeneity. Next, we show the speedups achieved by FCP and FLB.

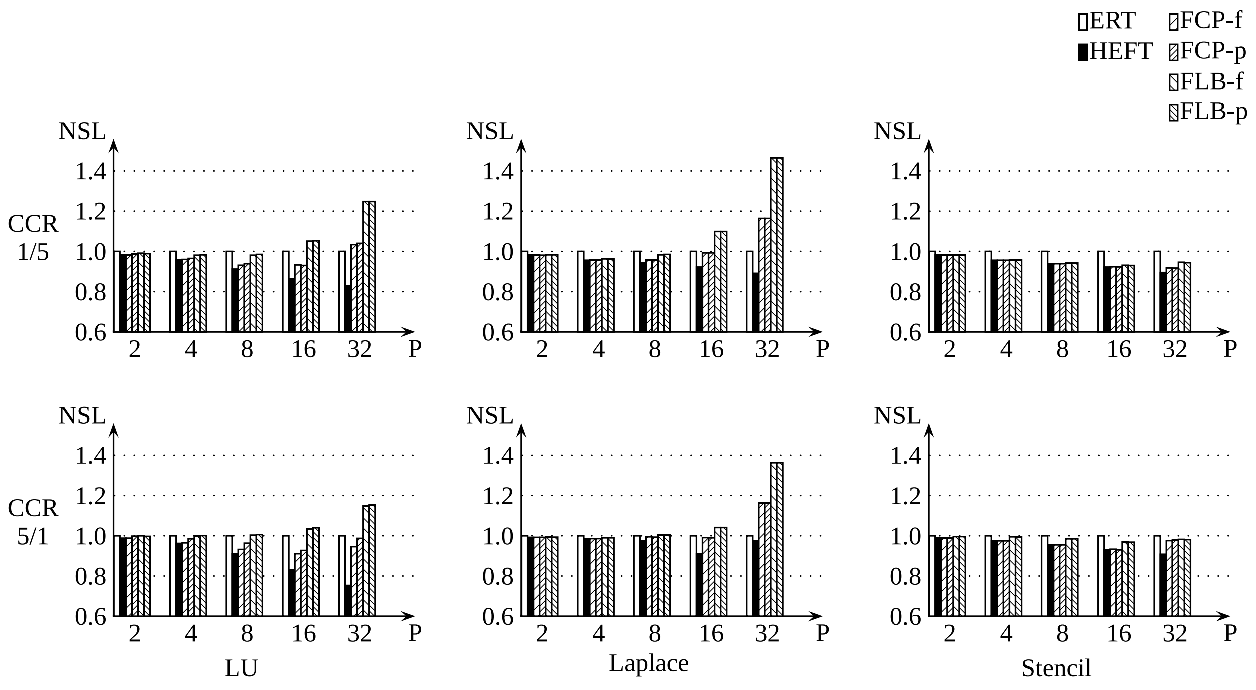

Figure 3: Performance comparison with respect to the problem

For performance comparison, we use the *normalized schedule length* (NSL), defined as the ratio between the schedule length of the given algorithm and the schedule length of ERT.

In Figure 3 we study the algorithms' performance with respect to the problem type by comparing the schedule lengths averaged over the three processor speed intervals. One can note that for both FCP and FLB, the partial versions obtain performance similar to the full versions. Therefore we will further refer only to the partial versions of FCP and FLB.

One can note that the overall performance of FCP is comparable to ERT's performance, although at a much lower cost. For problems involving a large number of fork and join tasks, such as LU and Laplace, for a large number of processors ERT performs better, up to 16% for both coarse and fine-grain cases (Laplace, $P = 32$). For all the other cases (i.e., for regular problems, such as Stencil, or for small number of processors) FCP performs equal or better compared to ERT, up to 8% (Stencil, $P = 32$) and 7% (LU, $P = 16$) for coarse and fine-grain problems, respectively.

Compared to HEFT, FCP is outperformed for problems involving a large number of fork and join tasks, such as LU and Laplace, for a large number of processors, with up to 27% (Laplace, $P = 32$) and 23% (LU, $P = 32$) for coarse and fine-grain cases, respectively. However, in all the other cases (i.e., for regular problems, such as Stencil, or for small number of processors) FCP performs comparable to HEFT.

FLB's performance is generally worse, being outperformed by ERT, HEFT and FCP by up to 46%, 57%, and 30% (all for coarse-grain Laplace, $P = 32\%$), respectively. However, even for FLB, the performance becomes comparable to the other three algorithms for regular problems, such as Stencil, or small number of processors.

In Figure 4 we study the influence of the heterogeneity to the performance. The results are averaged over the LU, Laplace and Stencil problems. Again, both FCP and FLB obtain similar performance for the full and partial versions.

Again, the overall performance of FCP is comparable

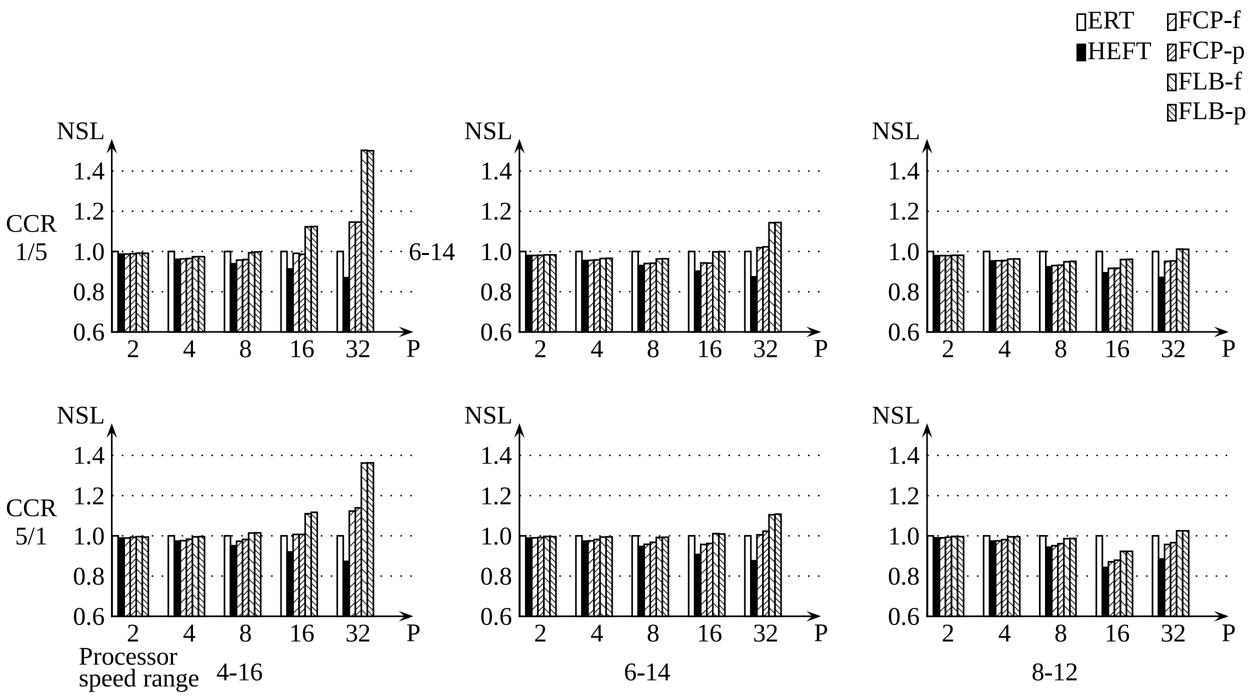

Figure 4: Performance comparison with respect to heterogeneity

to ERT's performance. For a large processor speed variance (i.e., $4-16$) and for a large number of processors ERT performs better, up to 15% and 12% for coarse and fine-grain cases ($4-16$ processor speed range, $P=32$), respectively. For all the other cases (i.e., small processor speed variance, or for small number of processors) FCP performs equal or even better compared to ERT, up to 8% and 12% (both for Stencil, $P=16$) for coarse and fine-grain problems, respectively.

Compared to HEFT, FCP is also outperformed for a large processor speed variance and for a large number of processors, with up to 28% and 26% (both for $4-16$ processor speed range, $P=32$) for coarse and fine-grain cases, respectively. However, for small processor speed variance, or for small number of processors, FCP's performance tends to become comparable to HEFT.

FLB's performance is generally worse, being outperformed by ERT, HEFT and FCP with up to 50%, 63%, and 35% (all for $4-16$ processor speed range, coarse-grain problems, $P=32\%$), respectively. However, even for FLB, the performance becomes comparable to the other three algorithms for regular problems, such as Stencil, or small number of processors.

One can note that for heterogeneous systems, the versions using fully and partially sorted priority lists perform comparable for both FCP and FLB. Similar to homogeneous systems, a partially sorted list of size P yields competitive results, while the scheduling complexity becomes extremely low: $O(V\log(P)+E)$.

Figures 5 and 6 show the speedups achieved for the FCP and FLB algorithms respectively. Although FCP performs better, the two algorithms perform similar with respect to problem type, granularity and processor speed range. For Stencil the speedup is almost linear. However, for LU and Laplace the speedup starts leveling off for more than 32 processors. The reason is that LU and Laplace have a large number of fork and join nodes, and as a consequence a limited parallelism, while Stencil is a regular problem with a large and constant parallelism. Also, one can note that for a large processor speed

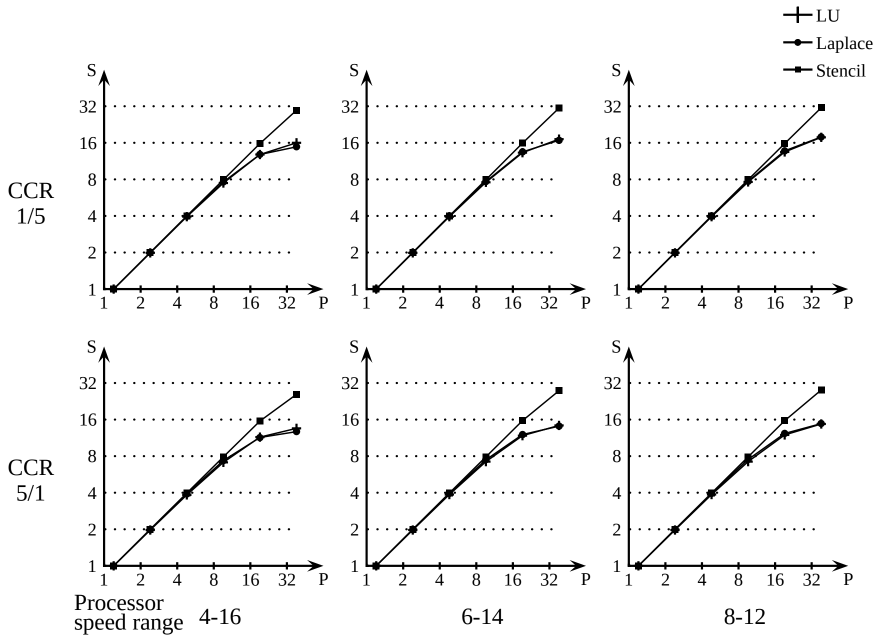

Figure 5: FCP-p Speedup

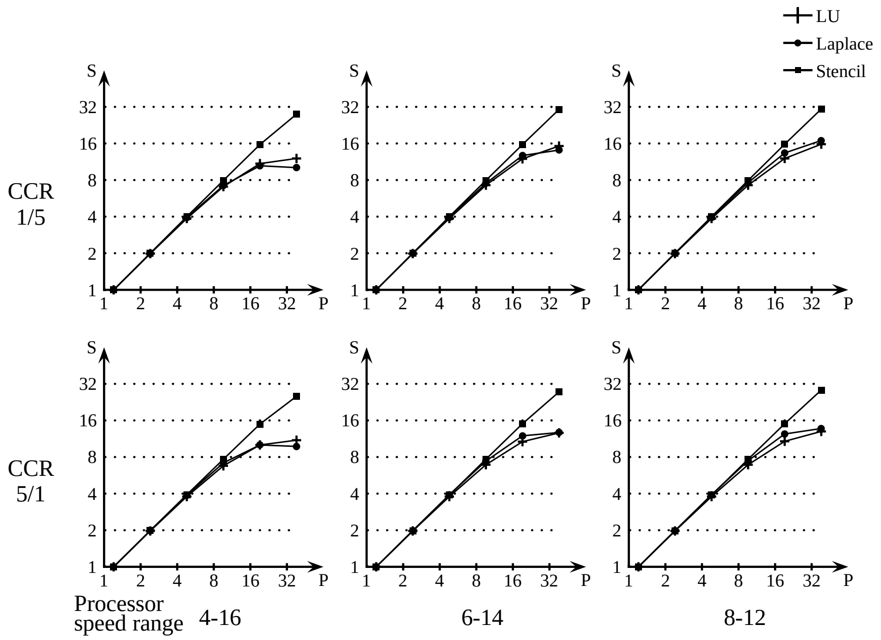

Figure 6: FLB-p Speedup

variance (i.e., $4 - 16$) and a large number of processors ($P = 32$) the speedup is lower compared to a small processor speed variance. Also, for fine-grain problems, the speedup is lower for a large number of processors. In both cases the reason is that there are not enough tasks to fully utilize the existing processors, and, as FCP and FLB are not specifically designed for heterogeneous processors, they do not always select the faster processors first.

5 Conclusion

In this paper we investigate the performance of the low-cost static list scheduling algorithm FCP and dynamic list scheduling algorithm FLB, modified to schedule applications for heterogeneous systems. We show that making minimal modifications that do not affect their very low cost, FCP and FLB still obtain good performance in heterogeneous systems, at a cost that is considerably below typical scheduling algorithms for heterogeneous systems.

We show that the performance of the modified versions of FCP and FLB is generally comparable to algorithms specifically designed for heterogeneous systems, such as HEFT and ERT. There are only a few cases, mainly for irregular problems and large processor speed variance, where FCP and FLB's performance drops down to 32% and 63%, respectively.

Considering the overall performance and their very low cost compared to the other algorithms, we believe FCP and FLB to be interesting compile-time candidates for heterogeneous systems, especially considering the large problem sizes that are used in practice.

References

[1] I. Ahmad and Y.-K. Kwok. A new approach to scheduling parallel programs using task duplication. In *Proc. Int'l Conf. on Parallel Processing*, 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*, 1992.

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

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

[5] M. Kafil and I. Ahmad. Optimal task assignment in heterogeneous computing systems. In *Proc. Heterogeneous Computing Workshop*, 1997.

[6] A. A. Khan, C. L. McCreary, and M. S. Jones. A comparison of multiprocessor scheduling heuristics. In *Proc. Int'l Conf. on Parallel Processing*, 1994.

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

[8] Y.-K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *Proc. Int'l Parallel Processing Symp. / Symp. on Parallel and Distributed Processing*, 1998.

[9] C.-Y. Lee, J.-J. Hwang, Y.-C. Chow, and F. D. Anger. Multiprocessor scheduling with interprocessor communication delays. *Operations Research Letters*, 7:141–147, June 1988.

[10] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Proc. Heterogeneous Computing Workshop*, 1998.

[11] G.-L. Park, B. Shirazi, J. Marquis, and H. Choo. Decisive path scheduling: A new list scheduling method. In *Proc. Int'l Conf. on Parallel Processing*, 1997.

[12] A. Rădulescu and A. J. C. van Gemund. FLB: Fast load balancing for distributed-memory machines. In *Proc. Int'l Conf. on Parallel Processing*, 1999.

[13] A. Rădulescu and A. J. C. van Gemund. On the complexity of list scheduling algorithms for distributed-memory systems. In *Proc. ACM Int'l Conf. on Supercomputing*, 1999.

[14] A. Rădulescu, A. J. C. van Gemund, and H.-X. Lin. LLB: A fast and effective scheduling algorithm for distributed-memory systems. In *Proc. Int'l Parallel Processing Symp. / Symp. on Parallel and Distributed Processing*, pages 525–530, 1999.

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

[16] M. Tan, H. J. Siegel, J. K. Antonio, and Y. A. Li. Minimizing the application execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system. *IEEE Trans. on Parallel and Distributed Systems*, 8(8):857–871, Aug. 1997.

[17] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Proc. Heterogeneous Computing Workshop*, 1999.

[18] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47:8–22, 1997.

[19] C. M. Woodside and G. G. Monforton. Fast allocation of processes in distributed and parallel systems. *IEEE Trans. on Parallel and Distributed Systems*, 4(2):164–174, Feb. 1993.

[20] 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.

[21] M.-Y. Wu, W. Shu, and J. Gu. Local search for dag scheduling and task assignment. In *Proc. Int'l Conf. on Parallel Processing*, 1997.

[22] T. Yang and A. Gerasoulis. Pyrros: Static task scheduling and code generation for message passing multiprocessors. In *Proc. ACM Int'l Conf. on Supercomputing*, 1992.

Biographies

Andrei Rădulescu received a MSc degree in Computer Science in 1995 from "Politehnica" University of Bucharest. Between 1995 and 1997 he was a teaching assistant at the "Politehnica" University of Bucharest. Since 1997, he is a PhD student at the Department of Information Technology and Systems of Delft University of Technology. His research interests are in multiprocessor scheduling, software support for parallel computing and parallel and distributed systems programming,

Arjan J.C. van Gemund received a BSc in Physics in 1981, a MSc degree (cum laude) in Computer Science in 1989, and a PhD (cum laude) in 1996, all from Delft University of Technology. In 1981 he joined the R & D organization of a Dutch multinational company as an Electrical Engineer and Systems Programmer. Between 1989 and 1992 he joined the Dutch TNO research organization as a Research Scientist specialized in the field of high-performance computing. Since 1992, he works at the Department of Information Technology and Systems of Delft University of Technology, currently as Associate Professor. His research interests are in the area of parallel and distributed systems programming, scheduling, and performance modeling.