A Low-Cost Approach Towards Mixed Task and Data Parallel Scheduling *

Andrei Rădulescu, Arjan J.C. van Gemund
Faculty of Information Technology and Systems
Delft University of Technology
{A.Radulescu,A.vGemund}@its.tudelft.nl

Abstract

A relatively new trend in parallel programming scheduling is the so-called mixed task and data scheduling. It has been shown that mixing task and data parallelism to solve large computational applications often yields better speedups compared to either applying pure task parallelism or pure data parallelism. In this paper we present a new compile-time heuristic, named Critical Path and Allocation (CPA), for scheduling data-parallel task graphs. Designed to have a very low cost, its complexity is much lower compared to existing approaches, such as TSAS, TwoL or CPR, by one order of magnitude or even more. Experimental results based on graphs derived from real problems as well as synthetic graphs, show that the performance loss of CPA relative to the above algorithms does not exceed 50%. *These results are also confirmed by performance measurements of two real applications (i.e., complex matrix multiplication and Strassen matrix multiplication) running on a cluster of workstations.*

1 Introduction

Recent studies has shown that, for a large class of large computational applications, exploiting both task and data parallelism yields better speedups compared to either pure task parallelism or pure data parallelism [2, 7, 20, 17, 26]. In large applications, pure data parallelism is relatively limited, especially for irregular applications. Pure task parallelism, on the other hand, can indeed expose a degree of parallelism that is comparable to the combined task and data parallelism. However, this can only be achieved using a huge time and memory compared to a mixed task and data parallelism approach. Thus, exploiting the hierarchy of task and data parallelism has emerged as a natural solution.

*This research is partially 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.

Efficiently exploiting mixed task and data parallelism within an application involves (a) good support at the language level for both task and data parallelism, and (b) a good scheduler. At the language level, there has been a considerable effort in (a) adding task support to data-parallel languages, as in Fx [24, 26], Fortran M [7] or Paradigm HPF [17], or (b) adding data-parallel support to task-parallel languages such as in Orca [2].

Efficient support for mixed task and data parallel scheduling is a critical issue for fully exploiting the potential advantage of the mixed task and data parallelism. Mixed task and data parallel scheduling algorithms use a directed acyclic graph in which data parallel tasks are the nodes and precedence relationships are the edges [16, 17, 23]. A task can run on any number of processors, as opposed to pure task scheduling algorithms where a task runs on a single processor [12, 16, 21, 22, 23, 29, 30]. To distinguish between the two scheduling techniques, we shall use the terms *M-task* and *S-task* to denote a task that can run on multiple processors and a single processor, respectively.

An M-task can be either a purely data parallel task, or a mixed task/data parallel routine. While pure data-parallel scheduling techniques [1, 5, 13, 14, 15, 18, 28] could still be applied within data-parallel M-tasks, pure task scheduling techniques [12, 16, 21, 22, 23, 29, 30] are no longer applicable to schedule M-tasks. As a result, new approaches have to be found that fully exploit the available parallelism.

Scheduling is known to be NP-complete even for the more simpler problem of scheduling S-tasks [9]. As a result, M-task scheduling is also NP-complete and heuristics are used. Ramaswamy *et al.* have proposed a two-step approach, called Two Step Allocation and Scheduling (TSAS) [17]. First, they use convex programming to find the number of processors each M-task will be executed on, such that a compromise is obtained between processor utilization and the overall critical path. The M-tasks are then scheduled to processors using a list scheduling algorithm. Rădulescu *et al.* have proposed a single step scheduling algorithm, called Critical Path Reduction (CPR) [20]. CPR starts from an one-processor allocation for each task,

0190-3918/01 $10.00 © 2001 IEEE

and iteratively improves this allocation until no further gain is obtained. Although compared to TSAS, CPR has a higher time complexity, the schedules are better. Rauber and Rünger consider a restricted case of task graphs with a series-parallel (SP) topology [19]. SP graphs are constructed merely by series and/or parallel node compositions [6]. A parallel composition consists of a set of independent M-tasks that are scheduled by partitioning the processors to disjoint sets and assigning the M-tasks to these processors sets. A series composition consists of a sequence of M-tasks, which are allocated the entire processor set mapped to the composition. Other approaches that focus on specific task graph topologies include the approach of Subhlok and Vondran that schedules pipelined M-tasks computations [25], and the approach of Prasanna *et al.* that schedules M-tasks organized in a tree topology [16]. Both latter approaches obtain optimal results for these particular cases, respectively. Turek present an approximate algorithm for scheduling independent M-tasks, which are shown to yield a performance within a factor of 2 compared to the optimal *et al.* [27]. Chakrabarti *et al.* also tackle the mixed task and data parallel scheduling problem by switching between pure task and pure data parallelism while running the program [4]. This, somewhat different, scheme is designed to have very low cost as it is used at runtime, as opposed to the other approaches that focus on compile-time scheduling.

In this paper, we aim at exploiting the cost advantage of the two-step approach. We present a new heuristic, named Critical Path and Area-based Scheduling (CPA), which solves the M-task scheduling problem, like in TSAS, in two steps. In the first step, the tasks are allocated a number of processors on which the tasks will run. In the second step, the tasks are scheduled on the available number of processors using a list scheduling. The difference between CPA and TSAS lies in the first step, where TSAS uses convex programming to find the task allocations, while CPA uses a greedy heuristic. Our approach has a considerably lower time complexity, while the results are still accurate. The performance of CPA is measured using both simulations and real executions based on synthetic, as well as real application task graphs. Our experiments show that CPA has a good cost/performance ratio compared to the other task/data parallel scheduling algorithms.

The paper is organized as follows. The next section specifies the scheduling problem and introduces some definitions used in the paper. In Section 3, our CPA algorithm is presented. Section 4 describes its performance, while Section 5 concludes the paper.

2 Preliminaries

A parallel program can be modelled 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 task graph represents a task that runs non-preemptively on any number of processors. Such a task is called an M-task, as opposed to a S-task which can run only on a single processor. Each M-task $t \in \mathcal{V}$ is assumed to have a *computation time*, denoted $T_w(t, N_p)$, which is a function of the number of processors (N_p). The computation time function can be obtained either by estimation [17, 19] or by profiling. When the cost is obtained through estimation, it is usually a simple function according to Amdahl's law. As a consequence, the execution time of an M-task t is predicted by: $T_w(t, N_p) = (\alpha + (1-\alpha)/N_p)\tau$, where τ is the M-task's execution time on a single processor, N_p is the number of processors, and α is the fraction of the M-task that executes serially. If profiling is used, the M-tasks' computation time is either fitted to a function such as described above (e.g., in case profiling data is incomplete over the whole number of processors), or the profiled values are used directly through a table.

The task graph's edges correspond to precedence relationships (communication messages or synchronization constraints) and are assigned a *communication cost*, denoted $T_c(t_1, t_2)$, which depends on the network characteristics, the amount of data to be transferred, the number of processors allocated to the predecessor and successor M-tasks, and the data distribution over the processors for both the predecessor task t_1 and the successor task t_2. As for computation costs, communication costs can be obtained by estimation [17, 19] or by profiling. In contrast to S-task scheduling, where scheduling a series of two tasks on the same processor typically implies that the communication cost becomes negligible, for M-task scheduling, mapping a series of two tasks on the same set of processors does not necessarily mean that the communication cost becomes negligible, since a data redistribution may occur.

An M-task with no input edges is called an *entry* task, while an M-task with no output edges is called an *exit* task. The length of a path is the sum of the computation and communication costs of the M-tasks and edges belonging to the path. The *critical path* is the longest path in the graph. An M-task's *top level* (T_t) is defined as the longest path from any entry task to the given task, excluding the given task. Similarly, the M-task's *bottom level* (T_b) is defined as the longest path from and including the given task to any exit task. One can note that the tasks in the critical path have the highest $T_t + T_b$ values. An M-task is said to be *ready* if all its parents have finished their execution. 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 homogeneous processors, connected in a clique topology in which inter-processor communication is assumed to perform without contention.

The information on an M-task t which is generated in the scheduling process is: (a) the number of allocated pro-

```
CPA ()
BEGIN
  Obtain the task allocation using MA().
  Schedule the tasks using MLS().
END
```

Figure 1. The CPA Algorithm

cessors $N_p(t)$, (b) the allocated processor set $\mathcal{P}_t(t)$, (c) the *start time* $T_s(t)$, and (d) the *finish time* $T_f(t)$. The *last message arrival time* of a ready task t is defined as $T_m(t) = \max_{(t',t)\in\mathcal{E}} \{T_f(t') + T_c(t',t)\}$. The *processor ready time* of a processor $p \in \mathcal{P}$ on a partial schedule is defined as the finish time of the last task scheduled on that processor: $T_r(p) = \max_{t\in\mathcal{V},p\in\mathcal{P}_t(t)} T_f(t)$.

The scheduling problem objective 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 $T_p = \max_{p\in\mathcal{P}} T_r(p)$.

3 The Algorithm

In this section, we present our multi-step scheduling algorithm, called Critical Path and Area-based scheduling (CPA), that produces good schedules, despite the fact that the time complexity is reduced with one order of magnitude compared to other well-known multi-step methods, such as TSAS and TwoL. Unlike some of the related work, the CPA algorithm is generally applicable, i.e., it admits task graphs of any topology. CPA uses a two-step approach similar to TSAS (see Figure 1). First, CPA allocates the number of processors on which each task will run. Second, CPA schedules the allocated tasks using a list scheduling algorithm, such as the one described in Section 3.2. However, instead of using the costly convex programming, as in TSAS, CPA uses a greedy algorithm to obtain the task allocation in the first step.

3.1 Task Allocation

Like in TSAS, we define the critical path of the task graph: $T_{CP} = \max_{t\in\mathcal{V}} \{T_b(t)\}$ and computing area: $T_A = \frac{1}{P} \sum_{t\in\mathcal{V}} (T_w(t, N_p(t)) \times N_p(t))$.

Both metrics represent theoretical lower bounds for the completion time T_p. However, T_{CP} and T_A characterize two different aspects of T_p. T_{CP} is a measure of the dependence paths, that can be shortened by allocating more processors to tasks. T_A is a measure of processor utilization, that is increased by allocating more processors to tasks.

The parallel completion time, T_p, can be approximated by the following formula: $T_p^e = \max\{T_{CP}, T_A\}$. Our goal in the allocation phase is to find a task allocation that minimizes T_p^e, which in TSAS is achieved through convex programming. We present a different approach using a simple yet effective greedy heuristic as described in Figure 2.

```
MA ()
BEGIN
  FORALL t ∈ V DO
    N_p(t) ← 1;
  END FORALL
  WHILE T_CP > T_A DO
    t ← CP task such that (N_p(t) < P) and
      ( T_w(t,N_p(t))/N_p(t) − T_w(t,N_p(t)+1)/(N_p(t)+1) ) is maximized;
    N_p(t) ← N_p(t) + 1;
    Recompute T_t and T_b values;
  END WHILE
END
```

Figure 2. The Task Allocation Procedure

We start with the most unfavourable case for T_{CP}, namely with one processor allocated for each task. This task allocation yields the minimum value for T_A. Therefore, if $T_A \geq T_{CP}$ then this allocation is also the one that minimizes T_p^e. If $T_A < T_{CP}$, we must decrease T_{CP} by allocating more processors to tasks.

We proceed by allocating at each step one more processor to a critical path task. We consider only critical path tasks, because increasing the number of processors for any other task does not reduce T_{CP}. From the critical path tasks, we select the task that determines the highest decrease of T_{CP} (i.e., the maximum gain for T_p^e).

The algorithm terminates when T_{CP} exceeds T_A, from which point onwards any task allocation increase will only increase T_p^e, as the largest term T_A is increased.

At each iteration, T_t and T_b need to be recomputed due to the new task allocation. This computation is not necessary for all the tasks, but only for the tasks affected by the reallocation. Consequently, only the T_t values of the current task's ancestors and the T_b values of the current task's descendents must be recomputed.

3.2 Task Scheduling

After the M-tasks have been allocated a number of processors, the tasks are actually scheduled on the processors using a list scheduling procedure (MLS, see Figure 3) similar to that used in TSAS or CPR.

At each step the ready M-task with the highest priority ($\rho(t)$) is scheduled. Tasks can have different priorities, such as the earliest starting time, as in TSAS, or bottom level, as in CPR. In our algorithms we use bottom level as the M-task priority used in MLS. As the number of processors (N_p) to be allocated on each M-task has already been determined, MLS only assigns the physical processors to the tasks, and also computes the task start and finish times.

```
MLS ()
BEGIN
  WHILE NOT all tasks scheduled DO
    t ← ready task with the maximum ρ(t).
    Schedule t on the first N_p(t) processors becoming idle.
  END WHILE
END
```

Figure 3. The List Scheduling Procedure

The processors are maintained sorted by their processor ready times (T_r). The set $\mathcal{P}_t(t)$ of the first $N_p(t)$ processors having the lowest T_r are assigned to the task t that is to be scheduled. The task start and finish times are then computed as $T_s(t) = \max_{p \in \mathcal{P}_t(t)}\{T_m(t), T_r(N_p(t))\}$, and $T_f(t) = T_s(t) + T_w(t, N_p(t))$, respectively.

Once the task has been scheduled, MLS checks all the successors of the given task to see if they become ready for execution. The algorithm computes the priorities for the new ready tasks and adds them to the ready task set. Then the procedure continues until all the tasks have been scheduled.

3.3 Complexity Analysis

The time complexity of the task allocation procedure is analyzed as follows. Computing T_t, T_b, T_{CP} and T_A takes $O(V + E)$ time. Computing the critical path and selecting the task with the maximum T_p^e gain also takes $O(V + E)$. Consequently, the loop body of the task allocation procedure (MA) takes $O(V + E)$ time.

The scheduling loop is executed at most P times for each task, because at each step one more processor can be allocated and there can be at most P processors allocated to a task. As there are V tasks in the task graph, the scheduling loop can be executed at most VP times. Although this is the worst case, in practice, however, the scheduling loop is executed considerably less frequent. Nevertheless, the worst-case time complexity for the task allocation procedure is $O(V(V + E)P)$.

The complexity of MLS breaks down in three components: (a) $O(E + V)$ to compute task priorities, (b) $O(V \log V)$ to sort the tasks, and (c) $O(VP)$ to schedule tasks to processors, resulting in a total time complexity of $O(E + V \log(V) + VP)$. Thus, it follows that the total worst time complexity of CPA is $O(V(V + E)P)$.

The worst time complexity of CPA is considerably lower compared to the worst time complexities of TSAS ($O(V^{2.5}P \log P)$), TwoL ($O(V^2P^2 \log P)$), and CPR ($O(EV^2P + V^3P \log(V) + V^3P^2)$). Considering its low time complexity, and its relatively good performance (see Section 4), we consider it as being a good choice when reasonably good schedules must be produced at very low cost.

4 Performance Analysis

The performance of our CPA algorithm is compared with three well-known existing M-task scheduling algorithms, namely TSAS [17], TwoL [19], and CPR [20], as well as with pure task and data parallel scheduling algorithms (called TASK and DATA, respectively). TASK and DATA use an allocation of 1 processor, and all P processors for each M-task using a list scheduling algorithm, respectively. We selected TSAS and CPR because, like CPA, they are targeted towards general task graphs and it has been shown to yield good performance compared to pure task and data scheduling algorithms [17, 20]. As the convex programming algorithm used by TSAS was not available, we have used the GENOCOP nonlinear solver [11] instead. We selected TwoL, because it is targeted to a large class of task graphs (i.e., SP graphs) and has been shown to yield good performance for a number of real problems [19, 20].

We consider both real problems and synthetic task graphs. The selected real problems are complex matrix multiplication ("Cmatmul") and Strassen's matrix multiplication ("Strassen") [10]. We use matrix sizes of 64 × 64 and 128 × 128 for Cmatmul, and 128 × 128 and 256 × 256 for Strassen, in order to observe how the scheduling algorithms perform on different levels of problem granularity. The execution times for the M-tasks are given by the cost estimation functions as defined in Section 2. These estimation functions are an approximation of the task execution times on our network of workstations cluster.

For the two real problems, we present both simulation and execution measurements. For the execution measurements, we use a cluster of Pentium Pro/200MHz PCs with 64MB RAM connected through Myrinet. We use Panda as the thread and communication library [3].

We generated 10 different synthetic task graphs for our comparison. The task graphs have an SP topology, such that they can be scheduled by TwoL without any modification, thus avoiding possible unfairness. The task graphs contain between 11 and 22 M-tasks, with an average sequential fraction of $\alpha = 0.2$.

For all the task graphs, we use as a relative performance measure the normalized schedule lengths, defined as the ratio between the T_p produced by the measured algorithm and that produced by TSAS: $T_n^{TSAS} = T_p^{alg}/T_p^{TSAS}$.

4.1 Complex Matrix Multiplication

In order to assess the cost improvement of CPA, in Figure 4 we show the running times for TwoL, CPR and CPA when applied to Cmatmul when run on a Pentium Pro/200MHz PC. We do not include running times for TSAS, because the original convex algorithm implementation for processor allocation was not available. One can

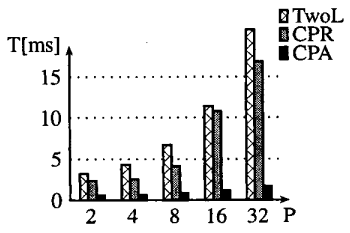

Figure 4. Running Times for Cmatmul

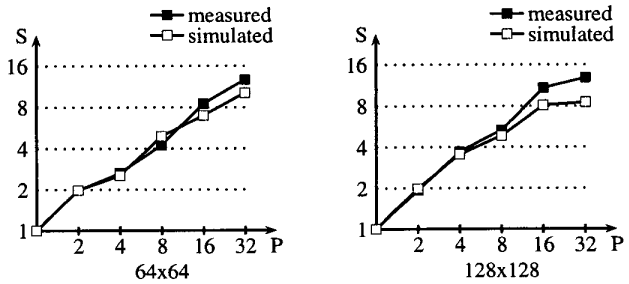

Figure 5. CPA Speedups for Cmatmul

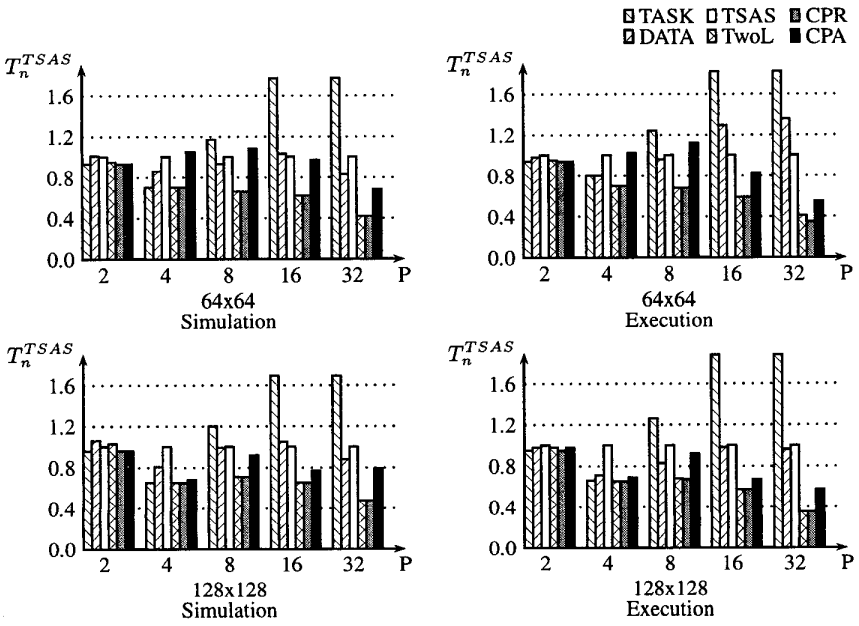

Figure 6. Relative Performance for Cmatmul

note that CPA proves to have a very short running time, running up to 12 and 10 times faster than TwoL and CPR, respectively.

In Figure 5 we present the simulation and execution speedups for Cmatmul using 64×64 and 128×128 matrix sizes. The execution speedups obtained using CPA are 12 and 13 for 64×64 and 128×128 matrix sizes at $P = 32$. One can note some differences with respect to the execution times between the problem simulations and executions. This is caused by the limited quality of the estimation function we used for the task execution times, instead of using profiled execution times.

In Figure 6, we present the normalized schedule lengths of the six M-task scheduling algorithms. Compared to TwoL and CPR, CPA obtains slightly lower performance, down to 44%. CPA has the same drawbacks as TSAS, namely it has a lower performance because in the first step it does not always allocate the ideal number of processors for each task. However, the overall performance of CPA is better when compared to TSAS, up to 32%, even though CPA has a lower time complexity. The main reason is that TSAS obtains its allocation in the real number domain, which must later be transformed to the integer number domain. In contrast, in CPA we directly compute an integer solution, which leads to a more accurate solution.

The execution performance of CPA has the same characteristics as for simulations, namely it has a lower performance when compared to CPR and TwoL, down to 44%, and outperforms TSAS, up to 45%.

4.2 Strassen Matrix Multiplication

As mentioned earlier, TwoL can only schedule SP task graphs. However, the task graph associated with Strassen has a non-SP topology. As a consequence, in order to be able to schedule this problem with TwoL, we need to trans-

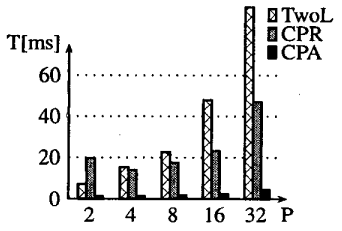

Figure 7. Running Times for Strassen

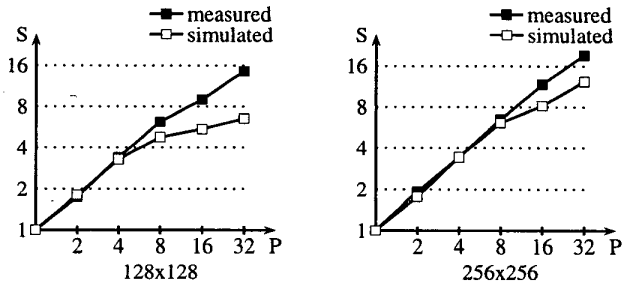

Figure 8. CPA Speedups for Strassen

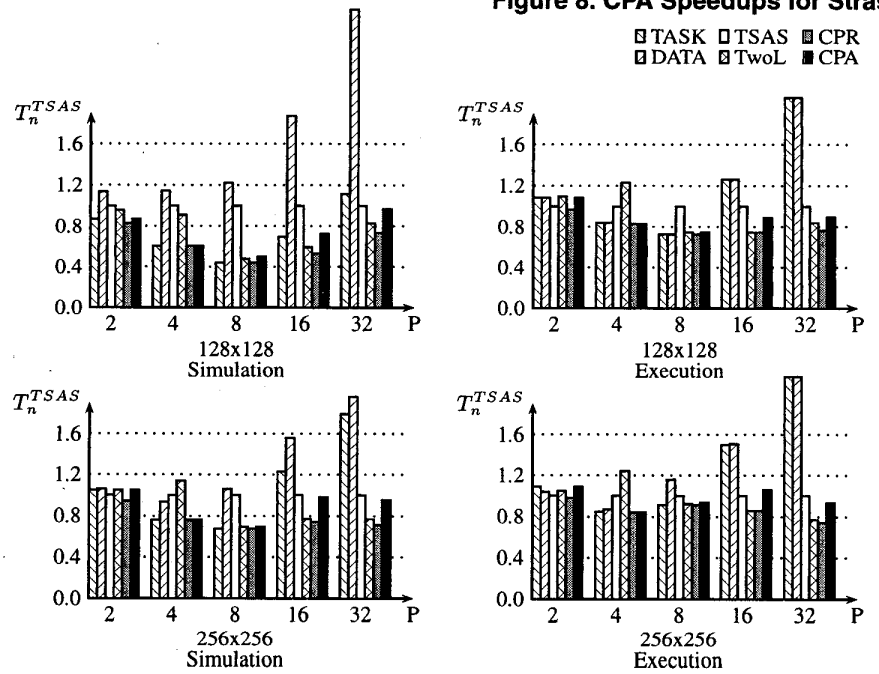

Figure 9. Relative Performance for Strassen

form the task graph of Strassen into an SP graph. This transformation is done by introducing two dummy synchronization tasks, before and after the multiplication tasks. For a fair comparison with the other algorithms, we do not add either execution or communication costs to these dummy tasks. We also do not include the synchronizations corresponding to these dummy tasks in the application implementation, but only use the processor allocation and task ordering obtained using TwoL.

In Figure 7 we show the running times for TwoL, CPR and CPA when applied to Strassen. Again, CPA has the shortest running time, running up to 20 and 10 times faster than TwoL and CPR, respectively.

In Figure 8 we present the simulation and execution speedups for Strassen. As for Cmatmul, slightly different speedups are achieved for simulations and executions due to the limited quality of the estimation function we used for the task execution times, instead of using profiled execution times. CPA yields a speedup of 15 and 19 on 32 processors, for 64×64 and 128×128 matrices, respectively.

In Figure 9, we present the schedule lengths normalized to TSAS for Strassen. Similar to Cmatmul, the simulation results for CPA are similar to those obtained for Cmatmul. CPA performance is lower compared to CPR and TwoL, down to 24%. However, for Strassen there are a few cases, especially for low number of processors, where CPA outperforms TwoL up to 42%. CPA also has an overall better performance compared to the performance of TSAS, up to 50%. One may note that CPA obtains this performance at a significantly lower time complexity compared to TSAS, TwoL, and CPR.

The execution results for Strassen also show that CPA is outperformed by CPR, up to 21%, and outperforms TSAS, up to 74%. CPA outperforms TwoL for low number of pro-

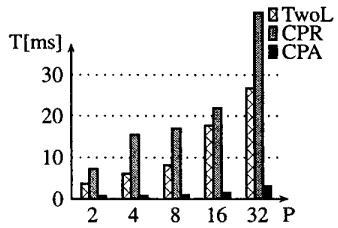

Figure 10. Running Times for Synthetic DAGs

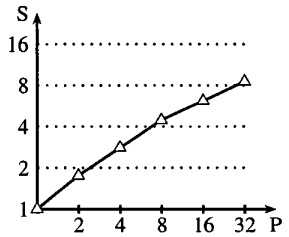

Figure 11. CPA Speed-ups for Synthetic DAGs

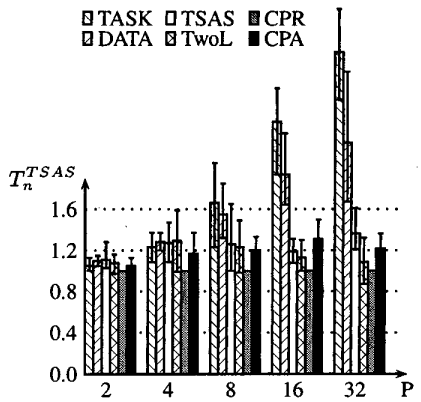

Figure 12. Relative Performance for Synthetic DAGs

cessors, up to 40%, and is outperformed by TwoL for larger number of processors, up to 20%.

4.3 Synthetic Task Graphs

In Figure 10 we show the average running times for TwoL, CPR and CPA when applied to the 10 synthetic task graphs. Also in this case CPA clearly has the shortest running time, up to 9 and 15 times faster compared to TwoL and CPR, respectively.

The speedup averaged over the 10 synthetic task graphs is shown in Figure 11, while in Figure 12 the performance of the algorithms relative to CPR is presented. CPA is consistently outperformed only by CPR, up to a maximum of 50%. On the other hand, CPA generally outperforms TwoL for small number of processors, up to 22%, and is outperformed for larger number of processors, up to 20%. Also in this case, CPA outperforms TSAS, up to 25%, even though the time complexity of CPA is much lower.

5 Conclusion

In this paper we present a new compile-time heuristic, called Critical Path and Area-based scheduling (CPA), for high-performance scheduling of arbitrary M-task graphs. The algorithm is intended for scheduling mixed task and data parallel applications at a very low time complexity, and yet producing reasonable performance.

Experiments are conducted using task graphs derived from real problems as well as synthetic graphs. Experimental results show that CPA yields reasonable good performance, generally outperforming TSAS, up to 74%, and being outperformed by CPR up to 50%. Compared to TwoL, CPA generally performs better for a low number of processors, up to 42%, and is outperformed by TwoL for a larger number of processors, up to 44%.

The CPA algorithm produces these results at a time complexity of $(V(V + E)P)$ which is considerably lower compared to the time complexities of TSAS $(O(V^{2.5}P \log P))$, TwoL $(O(V^2P^2 \log P))$, and CPR $(O(EV^2P + V^3P \log(V) + V^3P^2))$. This is also confirmed by the running times that were measured, showing that CPA runs up to 20 and 15 times faster compared to TwoL and CPR, respectively.

Future work includes moving the M-task scheduling at run-time in a manner similar to the RAPID system for S-task scheduling [8]. The reason for moving the scheduling at runtime is that the scheduling is dependent on the problem size, and therefore needs to be recomputed for each problem size. Moving the scheduling at runtime implies only one compilation at the expense of extra runtime cost. CPA is a good candidate for runtime scheduling, because it produces good schedules at a very low complexity which implicitly means a very small scheduling runtime offset.

In summary, CPA still produces good schedules at a time complexity which is much lower compared to existing approaches. Therefore, considering its cost/performance ratio, CPA is an attractive option for mixed task/data parallel scheduling of task graphs with arbitrary topology, especially when scheduling is performed at runtime.

References

[1] A. Aiken and A. Nicolau. Optimal loop parallelization. In *PLDI*, 1998.

[2] S. Ben Hassen, H. E. Bal, and C. J. Jacobs. A task and data parallel programming language based on shared

objects. *ACM Trans. on Programming Languages and Systems*, 20(6):1131–1170, Nov. 1998.

[3] R. A. Bhoedjang, T. Ruhl, R. Hofman, K. G. Langendoen, H. E. Bal, and M. F. Kaashoek. Panda: A portable platform to support parallel programming languages. In *SEDMS IV*, 1993.

[4] S. Chakrabarti, J. Demmel, and K. Yelick. Models and scheduling algorithms for mixed data and task parallel programs. *J. of Parallel and Distributed Computing*, 9:168–184, 1997.

[5] A. Darte and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. on Parallel and Distributed Systems*, 5(8):814–822, 1994.

[6] L. Finta, Z. Liu, I. Milis, and E. Bampis. Scheduling UET–UCT series–parallel graphs on two processors. *Theoretical Computer Science*, 162:323–340, Aug. 1996.

[7] I. T. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *J. of Parallel and Distributed Computing*, 26:24–35, 1995.

[8] C. Fu and T. Yang. Run-time techniques for exploiting irregular task parallelism on distributed memory architectures. *J. of Parallel and Distributed Computing*, 42(2):143–156, May 1997.

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

[10] G. Golub and C. F. van Loan. *Matrix computations*. Baltimore: Johns Hopkins University Press, 1996.

[11] S. Kozieland Z. Michalewicz. Evolutionary algorithms, homomorphous mappings, and constrained parameter optimization. *Evolutionary Computation*, 7(1):19–44, 1991.

[12] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. of Parallel and Distributed Computing*, 59:381–422, 1999.

[13] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *PLDI*, 1988.

[14] K. K. Parhi and D. G. Messerschmitt. Static rate-optimal scheduling of iterative dataflow programs via optimum unfolding. *IEEE Trans. on Computers*, 40(2):178–195, 1991.

[15] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic, 1988.

[16] G. Prasanna, A. Agarwal, and B. R. Musicus. Hierarchical compilation of macro dataflow graphs for multiprocessors with local memory. *IEEE Trans. on Parallel and Distributed Systems*, 5(7):720–736, July 1994.

[17] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A framework for exploiting task and data parallelism on distributed memory multicomputers. *IEEE Trans. on Parallel and Distributed Systems*, 8(11):1098–1115, Nov. 1997.

[18] B. R. Rau. Iterative modulo scheduling. *Int'l J. of Parallel Programming*, 24(1):3–64, 1996.

[19] T. Rauber and G. Rünger. Compiler support for task scheduling in hierarchical execution models. *J. of Systems Architecture*, 45:483–503, 1998.

[20] A. Rădulescu, C. Nicolescu, A. J. C. van Gemund, and P. P. Jonker. CPR: Mixed task and data parallel scheduling for distributed systems. In *IPDPS*, 2001. Best Paper Award.

[21] A. Rădulescu and A. J. C. van Gemund. FLB: Fast load balancing for distributed-memory machines. In *ICPP*, 1999.

[22] A. Rădulescu and A. J. C. van Gemund. On the complexity of list scheduling algorithms for distributed-memory systems. In *ICS*, 1999.

[23] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. MIT Press, 1989.

[24] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *PPoPP*, 1993.

[25] J. Subhlok and G. Vondran. Optimal use of mixed task and data parallelism for pipelined computations. *J. of Parallel and Distributed Computing*, 60:297–319, 2000.

[26] J. Subhlok and B. Yang. A new model for integrated nested task and data parallel programming. In *PPoPP*, 1997.

[27] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms for scheduling parallelizable tasks. In *SPAA*, 1992.

[28] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.

[29] 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.

[30] 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.