An Integrated Approach to Locality Conscious Processor Allocation and Scheduling of Mixed Parallel Applications

N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan and J. Saltz

Technical Report
OSU-CISRC-2/08-TR04

An Integrated Approach to Locality Conscious Processor Allocation and Scheduling of Mixed Parallel Applications[*]

Naga Vydyanathan[†], Sriram Krishnamoorthy[†], Gerald Sabin[†],
Umit Catalyurek[‡], Tahsin Kurc[‡], Ponnuswamy Sadayappan[†], and Joel Saltz[‡]
[†] Dept. of Computer Science and Engineering, [‡] Dept. of Biomedical Informatics
The Ohio State University

Abstract

Complex parallel applications can often be modeled as directed acyclic graphs of coarse-grained application-tasks with dependences. These applications exhibit both task- and data-parallelism, and combining these two (also called mixed-parallelism), has been shown to be an effective model for their execution. In this paper, we present an algorithm to compute the appropriate mix of task- and data-parallelism required to minimize the parallel completion time (makespan) of these applications. In other words, our algorithm determines the set of tasks that should be run concurrently and the number of processors to be allocated to each task. The processor allocation and scheduling decisions are made in an integrated manner and are based on several factors such as the structure of the task-graph, the runtime estimates and scalability characteristics of the tasks and the inter-task data communication volumes. A locality conscious scheduling strategy is used to improve inter-task data reuse. Evaluation through simulations and actual executions of task graphs derived from real applications as well as synthetic graphs shows that our algorithm consistently generates schedules with lower makespan as compared to CPR and CPA, two previously proposed scheduling algorithms. Our algorithm also produces schedules that have lower makespan than pure task- and data-parallel schedules. For task graphs with known optimal schedules or lower bounds on the makespan, our algorithm generates schedules that are closer to the optima than other scheduling approaches.

1 Introduction

Parallel computing has proven to be a powerful tool for meeting the growing computational requirements of existing and emerging scientific and engineering applications [20, 27]. To optimally exploit the computing capabilities of current massively parallel systems, it is crucial to design algorithms for efficient scheduling of parallel applications on these architectures. A large body of research has addressed this scheduling problem by modeling the parallel application as a directed acyclic graph of sequential

[*]This research was supported in part by the National Science Foundation under Grants #CCF-0342615 and #CNS-0403342.

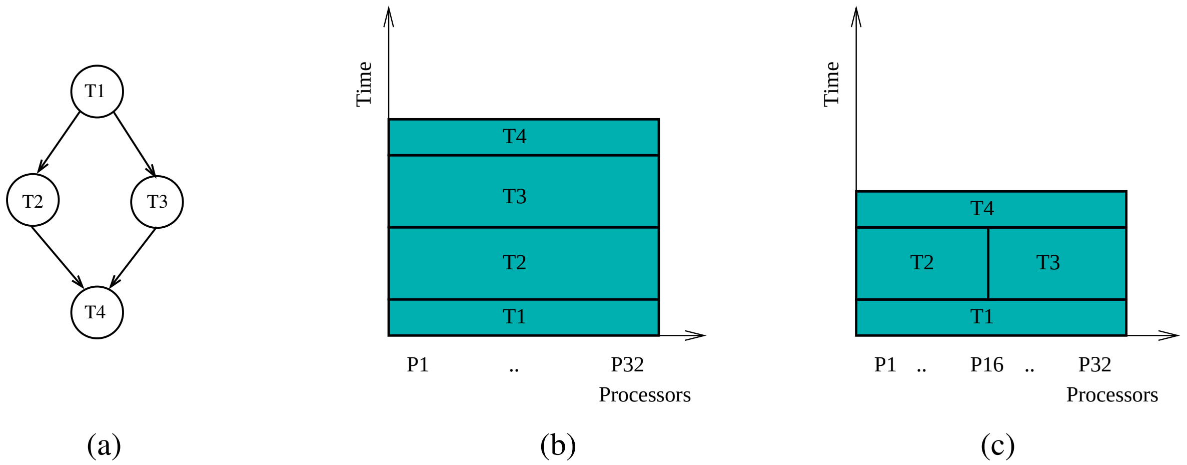


Figure 1. (a) A sample task graph, (b) A pure data-parallel schedule, (c) A mixed-parallel schedule.

tasks with dependences, where a task represents an atomic operation in the application [21]. However, as applications grow in their computational complexity, these task graphs could scale to millions of vertices, making their scheduling intractable. An alternative higher level of abstraction is to view the parallel application as a collection of coarse-grained parallel tasks with data dependences. This model of representation is also known as a macro data-flow graph [30].

Parallel applications represented as macro data-flow graphs can benefit from two forms of parallelism: task- and data-parallelism. Task-parallelism is the concurrent execution of independent tasks of the application on the same or different data elements. Data-parallelism is the parallel execution of a single task on data distributed over multiple processors. For example, in the task graph shown in Figure 1(a), executing tasks $T2$ and $T3$ concurrently denotes task-parallelism, while executing any of the four tasks on more than one processor denotes data-parallelism.

In a pure task-parallel schedule, each task is assigned to a single processor and multiple tasks are executed concurrently as long as precedence constraints are not violated and there are sufficient number of processors in the system. In a pure data-parallel schedule, the tasks are run in a sequence on all available processors. However, a pure task- or a pure data-parallel schedule may not be the optimal execution paradigm for many applications due to the following reasons. Task-parallelism in many applications is often limited by precedence constraints. The sub-linear speedups achieved with increasing number of processors leads to poor performance of pure data-parallel schedules. In fact, several researchers have shown that a combination of both, called mixed parallelism, yields better speedups [8, 30, 15]. In mixed-parallel execution, each parallel task in the application is allocated a subset of the processors and several such tasks are executed concurrently. The number of processors allocated to a parallel task denotes its degree of data-parallelism. The number of parallel tasks that are executed concurrently signifies the degree of task-parallelism.

The benefit of mixing task- and data-parallelism is illustrated in Figure 1. Consider that the tasks $T2$ and $T3$ in Figure 1(a) are identical and that their parallel speedup saturates at 16 processors, i.e allocating more than 16 processors does not yield any significant reduction in the task's execution times. Let their execution time on 16 processors be t time units. A pure data-parallel schedule on 32 processors is illustrated in Figure 1(b) where all four tasks are run on all 32 processors in a sequence. An alternative mixed-parallel schedule is shown in Figure 1(c), where tasks $T2$ and $T3$ are run concurrently on 16 processors each. The concurrent execution of tasks $T2$ and $T3$ exploits the task-parallelism, while running $T2$ and $T3$ on 16 processors and $T1$ and $T4$ on 32 processors exploits the data-parallelism. It is easy to note that this mixed-parallel schedule has a makespan that is lower than the data-parallel schedule by t time units.

In this paper, we propose an approach for computing the appropriate mix of task- and data-parallelism needed to minimize the parallel completion time or makespan of applications modeled as macro data-flow graphs. Given the runtime estimates of the tasks, their speedup functions and the inter-task data communication volumes, our algorithm computes the best processor allocation to tasks and the schedule in an integrated manner. To increase inter-task data reuse, we use a locality conscious backfill scheduling scheme. This scheme tries to schedule tasks to the same or overlapping processor groups as their predecessors, whenever possible, to maximize data reuse. Backfill scheduling tries to pack tasks as tight as possible and hence helps in improving the effective processor utilization by minimizing idle time slots. Our algorithm also uses a bounded look-ahead mechanism to escape local minima.

This work integrates the algorithms presented in our previous work [35, 36] and includes more extensive experimental evaluations. It also explores the feasibility of alleviating the scheduling overheads through parallelization of the proposed scheduling algorithm.

We evaluate the approach through comparison with two previously proposed scheduling schemes, Critical Path Reduction (CPR) [28] and Critical Path and Allocation (CPA) [29]. These schemes have been shown to give good improvement over other existing approaches like TSAS [30] and TwoL [31]. We also compare the algorithm against pure task- and pure data-parallel scheduling approaches. Evaluations are done through simulations and actual executions using synthetic task graphs and task graphs based on applications from the Standard Task Graph Repository [1] as well as task graphs from the domains of Tensor Contraction Engine [4] and Strassen Matrix Multiplication [14]. We show that our scheme consistently generates schedules with lower makespan than other approaches, by intelligently exploiting data locality and task scalability.

This paper is organized as follows. The next section gives an overview of the related work. Section 3 introduces the task graph and system model and Section 4 describes the proposed allocation and scheduling algorithm. Section 5 evaluates our scheduling scheme and Section 6 presents our conclusions.

2 Related Work

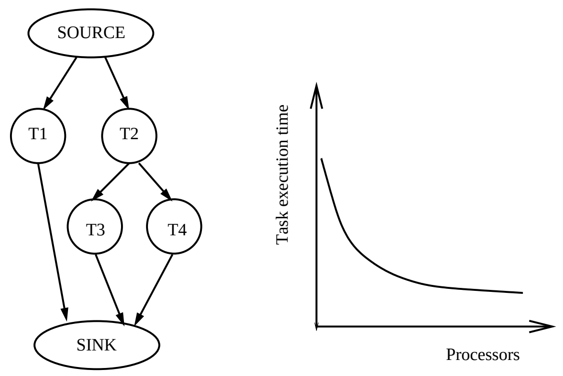
Optimal scheduling, even in the context of task graphs comprising sequential tasks, has been shown to be a hard problem to solve. Papadimitriou and Yannakakis [24] have proved that the problem of scheduling sequential tasks with precedence constraints is NP-complete. Du and Leung [13] studied the complexity of scheduling parallel task systems to minimize the parallel completion time and showed that scheduling independent parallel tasks is strongly NP-hard for 5 processors, and scheduling parallel tasks with precedence constraints consisting of a set of chains is strongly NP-hard even for 2 processors. Hence, several researchers have proposed heuristic solutions and approximation algorithms. This paper proposes a heuristic solution to the scheduling problem. The rest of this section discusses some research works that propose approximation algorithms followed by those that propose heuristic solutions.

Turek et al. [34] proposed an approximation algorithm for scheduling independent parallel tasks with performance within a factor of 2 compared to the optimal, and Jansen and Porkolab [17] proposed a polynomial approximation scheme based on integer linear programming for scheduling independent malleable tasks. Malleable tasks are parallel tasks whose execution time is a function of the number of processors allocated. Blazewicz et al. [6] proposed a linear time algorithm to solve this problem when the speedup functions of the tasks are convex. Lepere et al. [22] presented polynomial time, constant-factor approximation algorithm with approximation ratio of $\approx$5.236 for scheduling malleable tasks with dependences and this was improved to $\approx$4.73 by Jansen and Zhang [18]. However, these works do not model the inter-task data communication costs while formulating the scheduling problem.

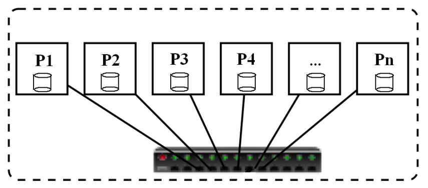
Ramaswamy et al [30] use a representation called the Macro Data-flow Graph (MDG) to denote the structure of mixed-parallel applications i.e applications that exhibit both task- and data-parallelism. The MDG is a directed acyclic graph with vertices representing sequential or data-parallel computations and edges representing precedence constraints. They proposed a two-step allocation and scheduling scheme, TSAS, to schedule mixed parallel applications on a P processor system. In the first step, a convex programming formulation is used to decide the processor allocation. In the second step, the tasks are scheduled using a prioritized list scheduling algorithm. A low-cost two-step approach was also proposed by Radulescu et al. [29] called Critical Path and Allocation (CPA), where a greedy heuristic is employed to iteratively compute the processor allocation (task allocation phase), followed by scheduling of the tasks (task scheduling phase). CPA is shown to generally perform better than TSAS, while having a lower scheduling complexity. Both TSAS and CPA attempt to minimize the maximum of the average processor area and the critical path length. The critical path is the longest path in the task graph. The processor area of a task is defined as the product of the number of processors allocated to it and its corresponding execution time. The average processor area is computed as the sum of the processor areas of all tasks in the task graph divided by the total number of processors on which the task graph is to be scheduled. The task allocation phase of CPA starts with an initial allocation of one processor to each task and iteratively refines this allocation as long as the critical path length exceeds the average processor area. To minimize the critical path length, at each step, the critical path is computed and a task from this path is chosen and its processor allocation is increased by one. In the task scheduling phase, the tasks are prioritized based on their bottom levels (this term is defined in Section 3) and scheduled in priority order to the earliest available set of processors. Due to the decoupling of the processor allocation and scheduling phases, both TSAS and CPA are limited in the quality of the schedules they can produce. In contrast, our proposed algorithm addresses the processor allocation and scheduling phases in an integrated manner.

To overcome some of the drawbacks of a decoupled approach, Radulescu et al. [28] proposed a heuristic called Critical Path Reduction (CPR) that couples the processor allocation and scheduling phases. Similar to CPA, CPR starts with an allocation of one processor to each task and prioritizes the tasks based on the sum of their top and bottom levels (described in Section 3). The tasks are traversed in priority order and their processor allocation is incremented in steps of one, as long as there is an improvement in makespan. The makespan is computed by scheduling the tasks using the same algorithm as used in the task scheduling phase of CPA. Though CPR has been shown to perform better than decoupled approaches like TSAS and CPA, all of these approaches do not optimize data reuse while scheduling the tasks to processors. Our proposed algorithm on the other hand, uses a locality conscious scheduling strategy to maximize inter-task data reuse.

Boudet et al. [7] proposed a single step approach for scheduling task graphs when the execution platform is a set of pre-determined processor grids. Each parallel task is only executed on one of these pre-determined processor grids. Li [23] has proposed a scheme for scheduling constrained rigid parallel tasks on multiprocessors, where the processor requirement of each task is fixed and known. On the other hand, we assume a more general model, where a parallel task may execute on any subset of



(a)



(b)

Figure 2. (a) Task Model, (b) System Model.

processors. Barbosa et al. [3] proposed a scheme for minimizing the makespan of dependent parallel tasks on heterogeneous systems, but they do not optimize inter-task data reuse.

Some researchers have proposed approaches for optimal scheduling for specific task graph topologies. Subhlok and Vondran proposed an approach for the optimal scheduling of pipelined linear chains of parallel tasks [33], and Choudhary et al. [9] addressed the throughput and response-time optimization of pipelined computations of parallel tasks with series-parallel precedence constraints. Prasanna and Musicus [25] described an approach for the optimal scheduling of tree DAGS and series parallel graphs for specific speedup functions. Rauber and Rünger [31] proposed the TwoL scheduling approach for task graphs with a series-parallel topology. In this work, we target task graphs of arbitrary structures.

3 Task Graph and System Model

A mixed-parallel application can be represented as a macro data-flow graph [30] which is a weighted directed acyclic graph (DAG), $G = (V, E)$, where V, the set of vertices, represents the parallel tasks and E, the set of edges, represents precedence constraints and data dependences. Each parallel task can be executed on any number of processors. There are two distinguished vertices in the graph: the *source vertex* which precedes all other vertices and the *sink vertex* which succeeds all other vertices. Figure 2(a) shows a sample MDG with 4 tasks. Please note that the terms, vertices and tasks are used interchangeably.

The weight of a vertex (task), $t_i \in V$, is its execution/computation time, which is a function of the number of processors allocated to it. This function can be provided by the application developer, or obtained by profiling the execution of the task on different numbers of processors or estimated through cost models [30, 31, 16, 10]. Typically the tasks show sub-linear speedups with increasing number of processors as shown in Figure 2(a) where beyond a certain number of processors, adding more processors does not show a significant decrease in the task's execution time. Each edge, $e_{i,j} \in E$, of the task graph is associated with the volume of data, $D_{i,j}$, to be communicated between the two incident vertices (tasks), t_i and t_j. The weight of an edge $e_{i,j}$ is the communication cost, measured as the time taken to transfer data $D_{i,j}$ between t_i and t_j. This is a function of the data volume $D_{i,j}$, network characteristics, the set of processors allocated to tasks t_i and t_j and the data distribution over these processor sets. Our approach for estimating the weight of an edge is described in Section 4.2. A task is assumed to run non-preemptively and start execution only after the completion of all its predecessors.

The length of a path in a DAG G is the sum of the weights of the vertices (tasks) and edges along that path. The *critical path* of G, denoted by $CP(G)$, is defined as the longest path in G. The *top level* of a task t in G, denoted by $topL(t)$, is the length of the longest path from the source vertex (task) to t, excluding the weight of t. The *bottom level* of a task t in G, denoted by $bottomL(t)$, is the length of the longest path from t to the sink, including the weight of t. Any task t with maximum value of the sum of $topL(t)$ and $bottomL(t)$ belongs to a critical path in G.

Let $st(t)$ denote the *start time* of a task t, and $ft(t)$ denote its *finish time*. A task t is eligible to start execution after all its predecessors (given by $Pred(t)$) are finished and their output data has been redistributed to t, i.e., the *earliest start time* of t is defined as $est(t) = \max_{t' \in Pred(t)}(ft(t') + rct(t', p', t, p))$. $rct(t', p', t, p)$ is the time to redistribute data generated by task t' on p' (p' is the processor group on which $t'$is scheduled), to t scheduled on p. Due to resource limitations, the start time of a task t might be later than its earliest start time, i.e., $st(t) \geq est(t)$. Note that with non-preemptive execution of tasks, $ft(t) = st(t) + et(t, np(t))$, where $np(t)$ is the number of processors allocated to task t, and $et(t, p)$ is the execution time of task t on p processors. The parallel completion time or the makespan of G is the finish time of the sink vertex.

The application task graph is assumed to be executed on a homogeneous compute cluster, with each compute node having local disks (Figure 2(b)). The nodes are interconnected by a switch and hence all nodes are only one hop away. Each parallel task distributes its output data among the processors/compute nodes on which it executes. A single-port communication model is assumed, i.e., each compute node can participate in no more than one data transfer in any given time-step. This is a realistic model for many single port systems, where multiple sends and receive operations are serialized by the single hardware port to the network [5]. The system model initially assumes overlap of computation and communication, as most clusters today are equipped with high performance interconnects which provide asynchronous communication calls. However, large communications which require I/O reads of chunks of data from disk to memory followed by their network transfer, may involve the host processor to a significant extent. Also, in practice, computation and communication may not be completely overlapped. Therefore, we have included evaluations of the algorithms assuming no overlap of computation and communication.

4 Locality Conscious Processor Allocation and Scheduling

This section presents the proposed Locality Conscious Mixed Parallel processor allocation and Scheduling algorithm (LoC-MPS). Unlike schemes that dissociate the allocation and scheduling phases [30, 29], LoC-MPS is a one-phase algorithm in the sense that the processor allocation and scheduling phases are coupled and the allocation and scheduling decisions are made in an integrated manner. LoC-MPS is designed to reduce the makespan of a given application task graph by:

- taking an integrated approach for allocating resources and scheduling tasks that can exploit detailed knowledge of both application characteristics and resource availability,
- iteratively minimizing computation and communication costs along the schedule's critical path, which includes dependences induced by resource constraints,
- maximizing inter-task data reuse and system utilization through a priority based locality conscious backfill scheduling scheme,

- and using a bounded look-ahead to escape local optima.

As confirmed by the experimental results, these features allow LoC-MPS to produce schedules with lower makespans than previous schemes. A detailed description of the algorithm is given in the following sections.

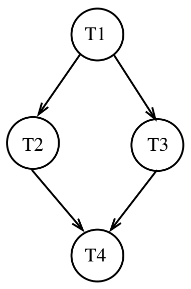
4.1 Initial Allocation and Schedule-DAG Generation

LoC-MPS starts with an initial allocation and schedule and iteratively reduces the makespan. When inter-task communication costs are significant and cannot be ignored, LoC-MPS starts with an initial allocation of one processor to each task. If inter-task communication costs are negligible and can be ignored, the initial allocation of processors to tasks is computed as follows. For each task t, we estimate the maximal set of concurrent tasks i.e tasks that do not have a dependency with t. For example, for the task graph in Figure 2(a), the maximal set of concurrent tasks for task $T1$ is $\{T2, T3, T4\}$, for $T2$ is $\{T1\}$, for $T3$ is $\{T1, T4\}$ and for $T4$ is $\{T1, T3\}$. Next, assuming that we hypothetically allocate the best number of processors to each of these concurrent tasks, we compute the number of remaining processors that could be allocated to task t, say p. The best number of processors for a task t, $P_{best}(t)$ is defined as the least number of processors on which task t's minimum execution time is expected, i.e allocating more than $P_{best}(t)$ processors to t does not cause further reduction in t's execution time or may infact cause an increase in execution time due to greater communication overheads. If the number of remaining processors, p, is more than one, task t is allocated $\min(P_{best}(t), p)$ processors, otherwise one processor is allocated to the task. Assuming that the task graph in Figure 2(a) has to be scheduled on 8 processors and that $P_{best}(t)$ is 4 for all tasks in the task graph, the remaining number of processors that could be allocated to $T1$ is -4 (after hypothetically allocating 4 processors each to its concurrent tasks $T2$, $T3$ and $T4$), to $T2$ is 4 and to $T3$ and $T4$ is 0 each. Hence, for this example, we start with an initial allocation of 1 processor to $T1$, 4 processors to $T2$ and 1 processor each to $T3$ and $T4$.

After computing the initial processor allocation, LoC-MPS schedules the parallel tasks to processors using a locality conscious scheduling algorithm that is described in subsequent sections. This initial allocation and schedule is iteratively refined to minimize the makespan. At each iteration, LoC-MPS selects the *best candidate* computation vertex or communication edge from the critical path of the schedule, whose weight/cost is to be reduced. The critical path of the schedule is given by $CP(G')$, where the schedule-DAG G' is the original DAG G with zero weight edges (pseudo-edges) added to represent *induced dependences* due to resource limitations (see Figure 3). Reducing $CP(G')$, which represents the longest path in the current schedule, potentially reduces the makespan. Consider the scheduling of the task graph displayed in Figure 3(a) on 4 processors. The processor allocation information is given in Figure 3(b). For simplicity, let us assume zero communication costs along the edges. Due to resource limitations tasks $T2$ and $T3$ are serialized in the schedule. Hence, the modified DAG G' (Figure 3(c)) includes an additional pseudo-edge between vertices $T2$ and $T3$. The makespan of the schedule G', which is the critical path length of G', is 30.

4.2 Best Candidate Selection

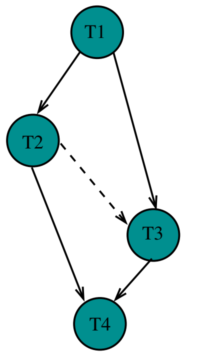
Once the critical path of the schedule-DAG G' is computed, LoC-MPS identifies whether the makespan is dominated by computation costs or communication costs and attempts to minimize the dominating cost in each iteration. The computation cost along the critical path is estimated as the sum of the weights of the vertices in $CP(G')$, and the communication cost is computed as the sum of the edge weights.



(a)

Task t	np(t)	et(t, np(t))
T1	4	10
T2	3	7
T3	2	5
T4	4	8

(b)



(c)

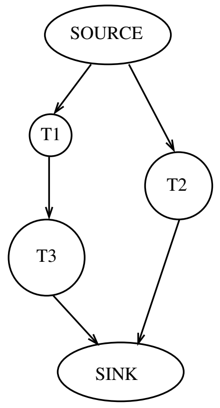
Figure 3. (a) Task Graph G to be scheduled on 4 processors, (b) Sample processor allocation, (c) Modified Task Graph, G'.

The vertex weight, which denotes the execution time of a task on the allocated number of processors is obtained from the task's execution profile. The edge weight denotes the communication cost to redistribute data between the processor groups associated with each task/endpoint of the edge. The weight of edge $e_{i,j}$ (between tasks t_i and t_j), is estimated as $wt(e_{i,j}) = \frac{d_{i,j}}{bw_{i,j}}$, where $d_{i,j}$ is the total data volume to be *redistributed*, and $bw_{i,j}$ is the aggregate communication bandwidth between t_i and t_j. In our experiments, we assume a block-cyclic distribution of data and estimate the volume of data to be redistributed between the producer and consumer tasks using the fast runtime block-cyclic data redistribution algorithm presented by Prylli and Tourancheau [26]. The aggregate communication bandwidth $bw_{i,j}$ is given by $bw_{i,j} = \min(np(t_i), np(t_j)) \times bandwidth$. As a task distributes its output among the disks/memory of the nodes on which it executes and the nodes are interconnected by a switch (as explained in Section 3), the bandwidth term in the above equation would correspond to the minimum of disk or memory bandwidth of the system depending on the location of data, and the network bandwidth.

4.2.1 Minimizing Computation Costs

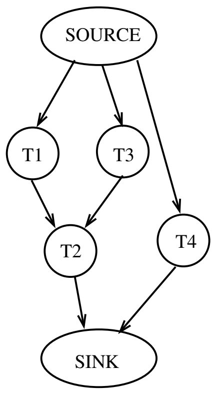
If computation cost dominates, LoC-MPS selects the *best candidate* from the tasks on a critical path (*candidate tasks*) and allocates one additional processor to this task. A poor choice of the best candidate will affect the quality of the resulting schedule as shown in the following example. Let the task graph in Figure 4 be scheduled on 4 processors and each task be initially allocated one processor. Tasks $T1$ and $T3$ lie on the critical path and either of them could be chosen to decrease the critical path length. If $T1$ were chosen and were allocated 4 processors, a data parallel schedule would be generated, with a makespan of 49.6. On the other hand, if $T3$ were chosen, the resulting schedule would have shorter makespan of 47 by allocating 4 processors to $T3$, 1 processor to $T1$ and 3 processors to $T2$.

To derive schedules of lower makespan through judicious choice of the best candidate task, LoC-MPS selects the best candidate task by considering two aspects: 1) scalability of the tasks and 2) global structure of the DAG. The goal of choosing a best candidate task is to choose a task which will reduce the makespan the most. First, the improvement in execution time of each candidate task t_c is computed as $et(t_c, np(t_c)) - et(t_c, np(t_c)+1)$. However, picking the best candidate task just based on the execution time improvement is a greedy choice that does not consider the global structure of the DAG and may result in a poor schedule. An increase in processor allocation to a task limits the number of tasks that can



Tasks	Number of Processors			
	1	2	3	4
T1	12.0	9.0	6.0	5.6
T2	30.0	17.0	11.0	9.0
T3	100.0	65.0	48.0	35.0

Figure 4. Task Graph G and its execution time profile.



Tasks	Number of Processors		
	1	2	3
T1	11.0	7.0	5.0
T2	8.0	6.0	5.0
T3	9.0	6.0	5.0
T4	7.0	5.0	4.0

Figure 5. Task Graph G and its execution time profile.

be run concurrently. Consider that the task graph in Figure 5 is to be scheduled on 3 processors. Each task is initially allocated one processor. Tasks $T1$ and $T2$ lie on the critical path and $T1$ has the maximum decrease in execution time. However, increasing the processor allocation of $T1$ to two processors will serialize the execution of $T4$, resulting in a makespan of 17. Any further iterations, do not improve the makespan. A better choice in this example is to choose $T2$ as the best candidate, and schedule it on 3 processors, while the other tasks are run on 1 processor each. This leads to a shorter makespan of 16.

To address this problem, LoC-MPS chooses a candidate task that not only provides a good execution time improvement, but also has a low *concurrency ratio*. The concurrency ratio of task t, $cr(t)$, is a measure of the amount of work that can potentially be done concurrent to t, relative to its own work. The work done by a parallel task is defined as the product of the number of processors on which it is executed , say 'p', and its execution time on 'p' processors. As we assume that tasks have either linear or sub-linear speedups, the minimum work done by a task is its execution time on 1 processor. Therefore, the concurrency ratio is given by: $cr(t) = \frac{\sum_{t' \in c_G(t)} et(t',1)}{et(t,1)}$ where $c_G(t)$ represents the maximal set of tasks that can run concurrent to t. A task t' is said to be concurrent to a task t in G, if there is no path between t and t' in G. This implies there is no direct or indirect dependence between t' and t, hence t' can potentially run concurrently with t. Depth First Search (DFS) is used to identify the dependent tasks.

DFS from a task t on G returns the set of nodes in G that are reachable from t. This is used to compute a list of tasks that depend on t. Next, a DFS on the transpose of G, G^T (obtained by reversing the direction of the edges on G) computes the list of tasks which t is dependent on. The remaining tasks constitutes the maximal set of concurrent tasks in G for task t: $c_G(t) = V - (DFS(G,t) + DFS(G^T,t))$. The calculation of the maximal set of concurrent tasks can be done in an optimized manner by calling DFS from the source vertex of the task graph and computing the list of dependent tasks for each vertex during its depth-first traversal.

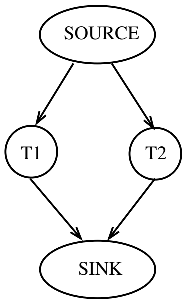
To select the best candidate task, the tasks in the critical path of the schedule are sorted in non-increasing order based on the amount of decrease in execution time. Among tasks within a certain percentage from the top of the list (which is estimated by tuning), the task with the minimum concurrency ratio is chosen as the best candidate. Inspecting the top 10% of the tasks from the list yielded good results for our experiments. To summarize, LoC-MPS widens tasks along the critical path that scale well and compete for resources with relatively few other "heavy" tasks.

4.2.2 Minimizing Communication Costs

In each iteration where communication costs dominate the makespan, LoC-MPS minimizes the cost of the heaviest edge along the critical path. Communication costs are minimized by: 1) reducing the cost of an edge by enhancing the aggregate communication bandwidth between the source and the destination processor groups and 2) maximizing inter-task data reuse through the use of a locality conscious scheduling scheme. The aggregate communication bandwidth is enhanced by improving the degree of parallel data transfer by increasing the processor allocation to the incident tasks. To minimize the cost of edge $e_{i,j}$, the processor allocation of t_i or t_j, whichever has lesser number of processors, is incremented by 1. If both t_i and t_j are allocated equal number of processors, both the processor counts are incremented by 1. The goal is to improve the communication bandwidth by increasing the term $\min(np(t_i), np(t_j))$ in the equation for the aggregate communication bandwith given in Section 4.2. In other words, when data is distributed among more nodes, communication costs can be reduced by exploiting parallel transfer mechanisms. Data redistribution costs are further reduced by intelligent allocation of tasks to processors using the locality conscious backfill scheduling scheme which maximizes inter-task data reuse by mapping producer and consumer tasks to same or overlapping processor groups, whenever possible. This algorithm is described in Section 4.4.

4.3 Bounded Look-ahead

Once the best candidate is selected and the processor allocation is refined, a new schedule is computed using our locality conscious backfill scheduler. The heuristics employed in LoC-MPS may generate a schedule with a worse makespan than in the previous iteration. If only schedule modifications that decrease the makespan are used, it is possible to be trapped in a local minima. Consider the DAG shown in Figure 6 and the execution profile assuming linear speedup. Assume that this DAG has to be scheduled on 4 processors. As $T2$ is more critical, the processor allocation of $T2$ would be incremented by one, until $T2$ is allocated 3 processors. In the next iteration, $T1$ is more critical. However, increasing the processor allocation of $T1$ to 2 causes an increase in the makespan. If the algorithm does not allow temporary increases in makespan, the schedule is stuck in a local minima: allocating 3 processors to $T2$ and 1 processor to $T1$. However, the data parallel schedule, i.e., running $T1$ and $T2$ on all 4 processors, leads to the smallest makespan.



Tasks	Number of Processors			
	1	2	3	4
T1	40.0	20.0	13.3	10.0
T2	80.0	40.0	26.7	20.0

Figure 6. Task Graph G and its execution time profile assuming linear speedup.

To alleviate this problem, LoC-MPS uses a bounded look-ahead mechanism. The look-ahead mechanism allows allocations that cause an increase in makespan for a bounded number of iterations. After these iterations, the allocation with the minimum makespan is chosen and committed. When inter-task communication costs are negligible, the bound for the number of iterations is taken to be $2 \times \max_{t \in V}(P - np(t))$. This is motivated by the observation that an increase in makespan is caused by two previously concurrent tasks being serialized due to resource limitations. Therefore, choosing the number of iterations in this way allows any two tasks to transform from a task parallel to a data parallel execution (using the maximum number of processors). When communication costs are significant, to limit the scheduling overheads, the look-ahead was bounded to a constant number of iterations.

4.4 Locality Conscious Backfill Scheduling (LoCBS)

Backfilling is a useful technique employed by many parallel job schedulers [32] to improve processor utilization of schedules. Backfilling identifies unused processor cycles in the schedule and allows lower priority jobs to use these cycles without delaying higher priority jobs. The parallel job schedule can be viewed as a 2D chart with time along one axis and the processors along the other axis, where the purpose is to efficiently pack the 2D chart (schedule) with jobs. Each job can be modeled as a rectangle whose height is the estimated run time and the width is the number of processors allocated. Backfilling works by identifying ”holes” or idle slots in the 2D chart and scheduling lower priority smaller jobs (which fit those holes) to these slots.

Due to data dependences and heterogeneous allocation of processors to tasks, schedules of task graphs with parallel tasks are liable to have ”holes” and, backfilling, which tries to pack parallel tasks as tightly as possible, could prove useful in improving the processor utilization and possibly decreasing the makespan. Hence, our algorithm uses backfilling. An example of backfilling is given in the step by step illustration of our scheduling algorithm described later (refer to Figure 7(f)). In order to maximize inter-task data reuse, rather than blindly scheduling a task to the earliest available idle slot where it fits, our locality conscious backfilling algorithm schedules a task on the idle slot that minimizes the tasks' completion time. A tasks' completion time is computed as the sum of the time taken to communicate its input data from the predecessor tasks and its execution time on the allocated number of processors. Hence, choosing the idle slot that minimizes the tasks' completion time tries to map tasks to same or overlapping processor groups as their predecessors whenever possible, thereby maximizing inter-task data reuse and minimizing the cost of communicating input data.

4.5 Overall Algorithm

Algorithm 1 outlines LoC-MPS. The algorithm starts from an initial allocation of processors to tasks that is computed in steps 1-9. In the main *repeat-until* loop (steps 12-50), the algorithm performs a look-ahead, starting from the current best solution (steps 20-45) and keeps track of the best solution found so far (step 42-44). If the look-ahead process does not yield a better solution, the task or edge that was the best candidate is marked as a bad starting point for future searches. However, if a better makespan was found, all marked tasks and edges are unmarked, the best allocation is committed and the search continues from this state. The look-ahead, marking, unmarking, and committing steps are repeated until either all tasks and edges in the critical path are marked or are allocated the maximum possible number of processors.

Algorithm 2 presents the pseudo code for the locality conscious backfill scheduling algorithm (LoCBS). The algorithm picks the ready task t_p with the highest priority (step 4) and schedules it on the set of processors that minimizes its finish time (steps 5-16). If t_p is not scheduled to start as soon as it becomes ready to execute, the set of tasks that "touch" t_p in the schedule are computed and pseudo edges are added between tasks in this set and t_p (steps 17-18). These pseudo edges signify induced dependences due to resource limitations.

Figure 7 illustrates the step by step generation of the mixed parallel schedule for the task graph given in Figure 4 on 4 processors. For the sake of simplicity, we assume that the communication costs are negligible. The algorithm starts with an initial allocation of 1 processor to each task and creates the schedule shown in Figure 7(a). In these figures, x-axis denotes the processors, while y-axis is time. A task is represented by a rectangular box whose width is the number of processors allocated and hieght is the corresponding execution time. The shaded boxes denote tasks that lie on the critical path. Initially, tasks $T1$ and $T3$ lie on the critical path and LoC-MPS picks $T3$ as the best candidate task and increases its processor allocation, based on the heuristic described in Section 4.2. $T3$ is widened upto 3 processors (Figures 7(b) and 7(c)) as the makespan decreases at each step. When $T3$ is allocated 4 processors (Figure 7(d)), due to resource limitations, task $T2$ is forced to start after $T3$ resulting in an increase in makespan. However, due to the look-ahead search technique, LoC-MPS allows this move and continues to search for a better schedule. Based on the best candidate selection heuristic, $T2$ is now chosen for widening upto 3 processors (Figures 7(e) and 7(f)). At this point, task $T2$ can backfill and start earlier (Figure 7(f)) leading to a lower makespan of 47. Though the algorithm continues to search for a better schedule by further widening tasks $T2$ and $T1$, as the schedule in Figure 7(f) yields the least makespan, it commits the moves only till this point and outputs this mixed-parallel schedule.

4.6 Scheduling Complexity

LoCBS takes $O(|V| + |E|)$ steps for computing the bottom level of tasks and $O(|V|)$ steps to pick the ready task with the highest priority. When communication costs cannot be ignored, scheduling the highest priority task and the necessary communications on the set of processors that minimizes its finish time (steps 5-16) takes $O(|V|^2 P \log P + d^2|V||E|P)$, where d is the average degree of a task, and adding pseudo edges takes $O(|V|P \log P)$ and thus, the complexity of LoCBS is $O(|V|^3 P \log P + d^2|V|^2|E|P)$. If communication costs can be ignored, LoCBS takes $O(|V|^2 + |E|)$ steps. LoC-MPS requires $O(|V| + |E'|)$ steps to compute $CP(G')$ and choose the best candidate task or edge. As there are at most $|V|$ tasks in CP and each can be allocated at most P processors, the repeat-until loop in steps 12-50 has at most $O(|V|^2 P)$ iterations, when the communication costs are significant, and

Algorithm 1 LoC-MPS: Locality Conscious Mixed Parallel Scheduling Algorithm

```
1: for all $t \in V$ do
2:     if inter-task communication costs is ignored then
3:         $p \leftarrow P - \sum_{t' \in c_G(t)} P_{best}(t')$    ▷ number of available processors if we allocate best number of processors to each of the concurrent tasks
4:         if $p > 1$ then
5:             $np(t) \leftarrow \min(P_{best}(t), p)$
6:         else
7:             $np(t) \leftarrow 1$
8:     else
9:         $np(t) \leftarrow 1$
10: $best\_Alloc \leftarrow \{np(t) | t \in V\}$    ▷ Best allocation is the initial allocation
11: $< G', best\_sl > \leftarrow LoCBS(G, best\_Alloc)$    ▷ Compute the schedule and the schedule length
12: repeat
13:     $\{np(t) | t \in V\} \leftarrow best\_Alloc$    ▷ Start with best allocation
14:     $old\_sl \leftarrow best\_sl$    ▷ and best schedule
15:     if inter-task communication costs is ignored then
16:         $LookAheadDepth \leftarrow 2 \times \max_{t \in V}(P - np(t))$
17:     else
18:         $LookAheadDepth \leftarrow Constant, K$
19:     $iter\_cnt \leftarrow 0$
20:     while $iter\_cnt < LookAheadDepth$ do
21:         $CP \leftarrow$ Critical Path in $G'$
22:         $T_{comp} \leftarrow$ Total computation cost along $CP$
23:         $T_{comm} \leftarrow$ Total communication cost along $CP$
24:         if $T_{comp} > T_{comm}$ then
25:             $t_b \leftarrow BestCandidate$ task in $CP$ with $np(t_b) < \min(P, P_{best}(t_b))$ and $t_b$ is not marked if $iter\_cnt = 0$    ▷ $P_{best}(t)$ is the least number of processors on which the execution time of $t$ is minimum and only unmarked tasks are selected at the beginning of the look-ahead search
26:             if $iter\_cnt = 0$ then
27:                 $LAStart \leftarrow t_b$    ▷ $LAStart$ signifies the start point of this look-ahead search
28:             $np(t_b) \leftarrow np(t_b) + 1$
29:         else
30:             $e_{s,d} \leftarrow$ heaviest edge along $CP$ that is not marked if $iter\_cnt = 0$ and $np(t_s)$ or $np(t_d) < P$    ▷ Please note that only unmarked edges are selected at the beginning of the look-ahead search
31:             if $np(t_s) > np(t_d)$ then
32:                 $np(t_d) \leftarrow np(t_d) + 1$
33:             else if $np(t_s) < np(t_d)$ then
34:                 $np(t_s) \leftarrow np(t_s) + 1$
35:             else
36:                 $np(t_d) \leftarrow np(t_d) + 1$
37:                 $np(t_s) \leftarrow np(t_s) + 1$
38:             if $iter\_cnt = 0$ then
39:                 $LAStart \leftarrow e_{s,d}$    ▷ $LAStart$ signifies the point of start of this look-ahead search
40:         $A' \leftarrow \{np(t) | t \in V\}$
41:         $< G', cur\_sl > \leftarrow LoCBS(G, A')$
42:         if $cur\_sl < best\_sl$ then
43:             $best\_Alloc \leftarrow \{np(t) | t \in V\}$
44:             $best\_sl \leftarrow cur\_sl$
45:         $iter\_cnt \leftarrow iter\_cnt + 1$
46:     if $best\_sl \geq old\_sl$ then
47:         Mark $LAStart$ as a bad starting point for future searches
48:     else
49:         Commit $best\_Alloc$ and unmark all marked tasks and edges
50: until all task and edges in CP are marked or for all tasks $t \in CP$, $np(t) = P$
```

$O(|V|^2P^2)$ iterations when there are no communication costs. Thus overall complexity of LoC-MPS is $O(|V|^2P(|V|^3P\log P + d^2|V|^2|E|P))$ when communication costs are significant and $O(|V|^2P^2(|V|^2 + |E'|)$ when communication costs can be ignored. On the other hand, complexity of CPR is $O(|E||V|^2P + |V|^3P(\log|V| + P\log P))$ [28]. CPA is a low cost algorithm with complexity $O(|V|P(|V| + |E|))$ [29]. Though LoC-MPS is more expensive than the other approaches, as the number of vertices is few in most

Algorithm 2 LoCBS: Locality Conscious Backfill Scheduling

```
1: function LoCBS($G, \{np(t) | t \in V\}$)
2:     $G' \leftarrow G$
3:     while not all tasks scheduled do
4:         Let $t_p$ be the ready task with highest value of $bottomL(t_p) + \max_{t_i \in Pred(t_p)} wt(e_{i,p})$
5:         $p_{best}(t_p) \leftarrow \emptyset$    ▷ Best processor set
6:         $ft_{min}(t_p) \leftarrow \infty$    ▷ Minimum finish time
7:         $free\_resource\_list \leftarrow \{< p, eat(p), dur(p) > \| |p| \geq np(t_p) \wedge dur(p) \geq et(t_p, np(t_p)) \wedge (eat(p) + dur(p)) \geq \max_{t' \in Pred(t_p)} ft(t') + et(t_p, np(t_p))\}$    ▷ List of idle-slots or holes in the schedule that can hold task $t_p$; $p$: processor set, $eat(p)$: earliest available time of $p$, $dur(p)$: duration for which the processor set $p$ is available
8:         for $< p, eat(p), dur(p) > \in free\_resource\_list$ do
9:             $p_{sub} \leftarrow$ subset of processors in $p$ that have maximum potential for data reuse for $t_p$
10:            $est(t_p) \leftarrow \max_{t' \in Pred(t_p)}(ft(t') + rct(t', p', t_p, p_{sub}))$    ▷ $rct(t', p', t_p, p_{sub})$ is the time to redistribute data output by predecessors (given by $Pred(t_p)$) of $t_p$ on the processor sets on which they are scheduled to $p_{sub}$
11:            $st(t_p) \leftarrow \max(eat(p), est(t_p))$
12:            $ft(t_p) \leftarrow st(t_p) + et(t_p, np(t_p))$
13:            if $ft(t_p) \leq eat(p) + dur(p)$ then    ▷ Task can complete in this backfill window
14:                if $ft(t_p) < ft_{min}(t_p)$ then
15:                    $p_{best}(t_p) \leftarrow p'$ and $ft_{min}(t_p) \leftarrow ft(t_p)$    ▷ Save the processor set that yields the minimum finish time
16:        Schedule $t_p$ on the processor set $p_{best}(t_p)$
17:        if $st(t_p) > est(t_p)$ then
18:            Add a pseudo edge in $G'$ between each task $t_i$ and $t_p$ where $ft(t_i) = st(t_p)$ and $p(t_i) \cap p(t_p) \neq \emptyset$
19:    return $< G'$,Schedule length$>$
```

mixed-parallel applications, the scheduling times are reasonable as seen in the experimental results.

5 Performance Analysis

This section evaluates the quality (makespan) of the schedules generated by LoC-MPS against those generated by CPR, CPA, pure task-parallel (TASK) and pure data-parallel (DATA) schemes. CPR is a single-step approach, while CPA is a two-phase scheme. These schemes are explained in detail in Section 2. TASK allocates one processor to each task and uses the locality conscious backfill scheduling algorithm to schedule them to processors. DATA executes tasks in a sequence on all processors. We evaluate the schemes using a block-cyclic distribution of data and estimate the volume of data to be redistributed between the producer and consumer tasks using the fast runtime block-cyclic data redistribution algorithm presented by Prylli and Tourancheau [26]. In DATA, as all tasks are executed on all processors, no redistribution cost is incurred. The performance of the scheduling algorithms are evaluated using synthetic task graphs as well as those from applications, through simulation and actual execution.

5.1 Task Graphs from the Standard Task Graph Set

The Standard Task Graph Set (STG) [1] is a benchmark suite for the evaluation of multi-processor scheduling algorithms that contain both randomly generated task graphs and task graphs modeled from applications. The following experiments use random DAGs and two application DAGs - *Robot Control* (Newton-Euler dynamic control calculation [19]), and *Sparse Matrix Solver* (random sparse matrix solver of an electronic circuit simulation). The robot control DAG contains 88 tasks, while the sparse matrix solver DAG contains 96 tasks. The task graphs in STG do not model inter-task communication costs and hence they are assumed to be negligible.

Parallel task speedup was generated using Downey's model [11]. Downey's speedup model is a non-linear function of two parameters: A, the average parallelism of a task, and σ, a measure of the

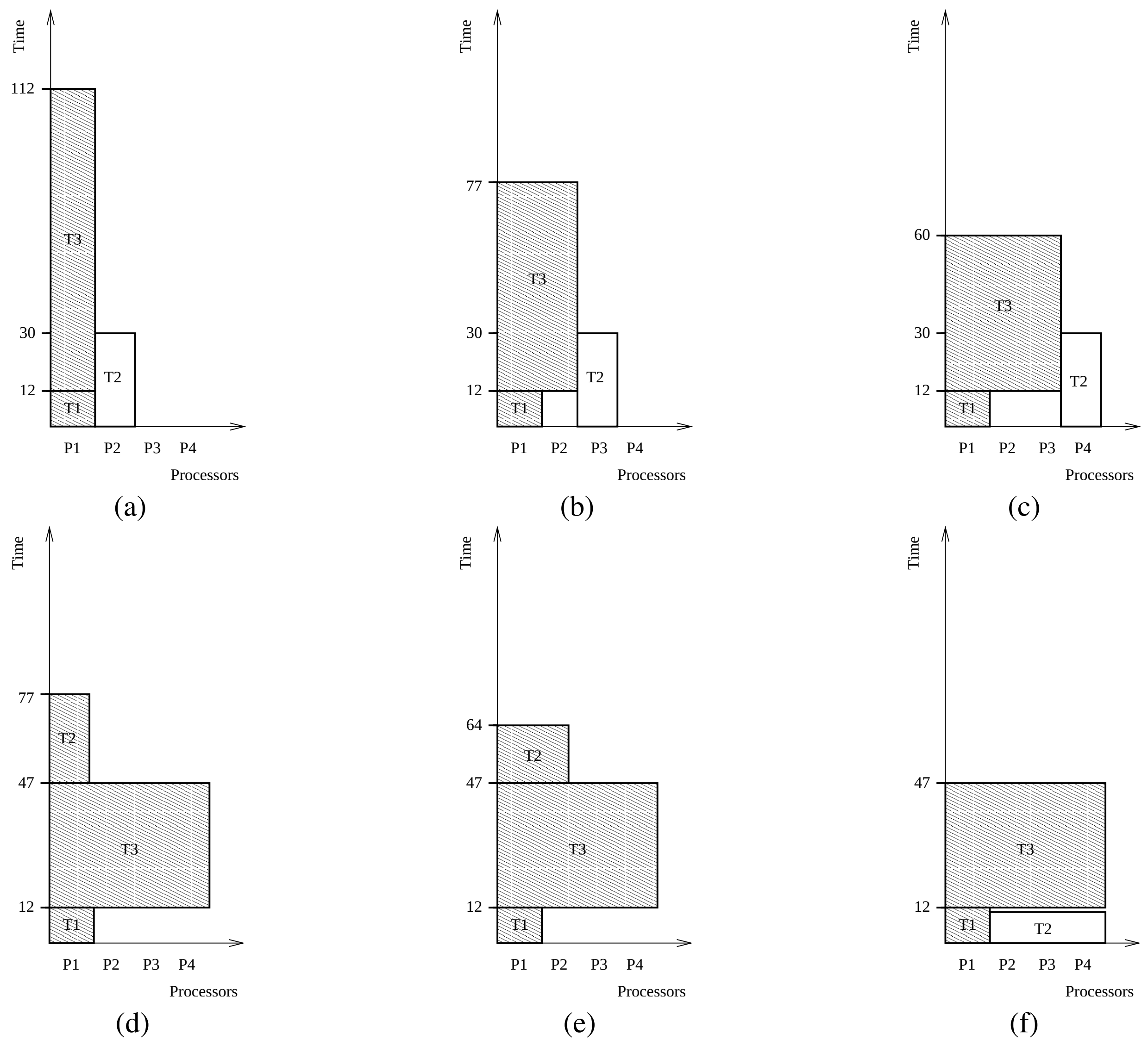
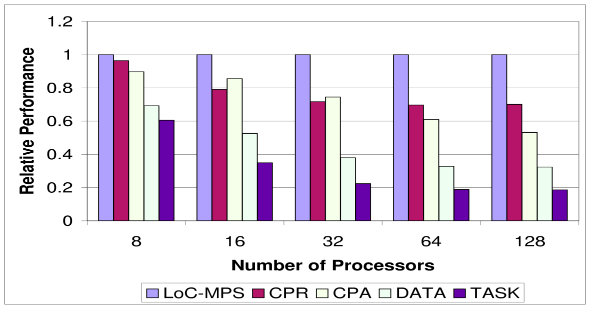


Figure 7. Step by step illustration of schedule generation for the task graph in Figure 4 on 4 processors.

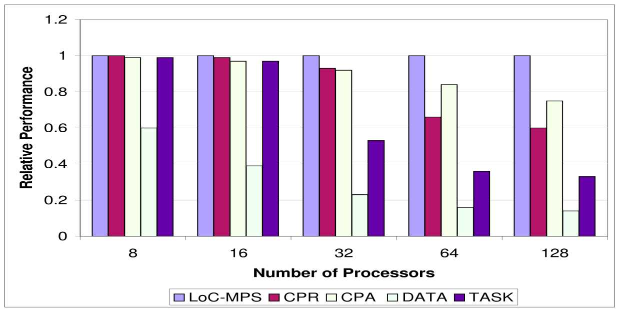
variations in parallelism. According to this model, the speedup S of a task, as a function of the number of processors n, A and σ is given by:

$$S(n,A,\sigma) = \begin{cases} \frac{An}{A+\sigma(n-1)/2} & (\sigma \leq 1) \wedge (1 \leq n \leq A) \\ \frac{An}{\sigma(A-1/2)+n(1-\sigma/2)} & (\sigma \leq 1) \wedge (A \leq n \leq 2A-1) \\ A & (\sigma \leq 1) \wedge (n \geq 2A-1) \\ \frac{nA(\sigma+1)}{\sigma(n+A-1)+A} & (\sigma \geq 1) \wedge (1 \leq n \leq A + A\sigma - \sigma) \\ A & (\sigma \geq 1) \wedge (n \geq A + A\sigma - \sigma) \end{cases}$$

Parallel task speedup is calculated by generating A and σ as uniform random variables in the intervals [1-32] and [0-2.0] respectively, to represent the common scalability characteristics of many parallel jobs [12].

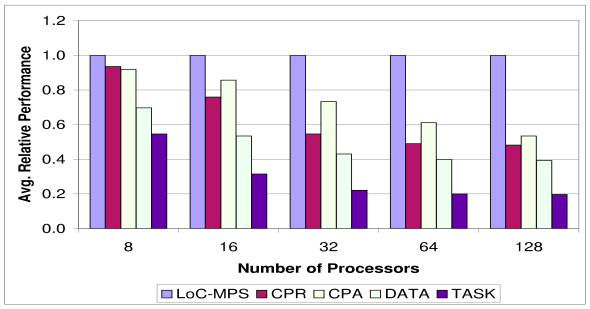


(a)

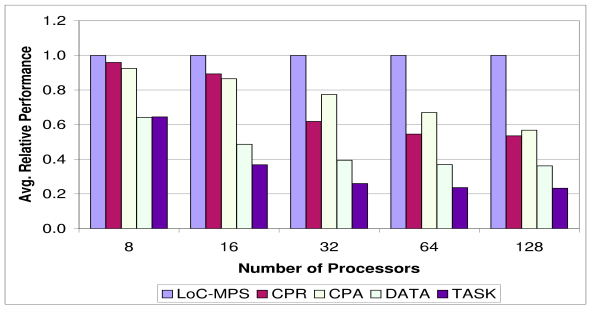


(b)

Figure 8. Performance of the scheduling schemes for (a) Robot Control DAG (b) Sparse Matrix Solver DAG.



(a)



(b)

Figure 9. Performance of the scheduling schemes for Synthetic DAGs having (a) 50 tasks (b) 100 tasks.

Figure 8 plots the relative performance of the different schemes for the two application DAGs from STG as the number of processors in the system is increased. The relative performance of an algorithm is computed as the ratio of the makespan produced by LoC-MPS to that of the given algorithm, when both are applied on the same number of processors. Therefore, a ratio less than one implies lower performance than that achieved by LoC-MPS. For the robot control application, LoC-MPS achieves up to 30% improvement over CPR and up to 47% over CPA. LoC-MPS also achieves up to 81% and 68% improvement over TASK and DATA. The performance improvement of our scheme over the other approaches increases as we increase the number of processors in the system. For the sparse matrix solver application, LoC-MPS, CPR and CPA perform similar to TASK up to 16 processors as the DAG is very wide. Beyond 16 processors the performance of the various schemes beings to differentiate. When the number of processors is increased to 128, LoC-MPS shows an improvement of up to 40% over CPR, 25% over CPA, and 67% and 86% over TASK and DATA, respectively. DATA performs poorly as the tasks have sub-linear speedup and the sparse matrix DAG is wide.

Figure 9 shows the average relative performance of the schemes for 20 random graphs in the Standard

Task Graph Set, having 50 and 100 tasks respectively. Again, we see similar trends as for the application DAGs. LoC-MPS performs the best and shows an improvement up to 52%, 47%, 80%, and 61% over CPR, CPA, TASK, and DATA, respectively.

5.2 Task Graphs with known Optimal Schedules or Lower Bounds on Makespan

This group comprises of two sets of task graphs that have an optimal schedule or well known lower bound on makespan. The goal is to compare how well the algorithms perform against the optimal schedule or lower bound. In these task graphs, the inter-task communication costs are assumed to be negligible.

The first set consists of randomly generated DAGs from STG, where all the tasks have perfectly linear speedup. For this type of graph, a pure data-parallel schedule gives the minimum makespan.

Proposition 1 *Given a task graph G that is an arbitrary DAG where each task is perfectly scalable and inter-task communication costs are negligible, the optimal schedule that yields the minimum makespan of G is a pure data-parallel schedule.*

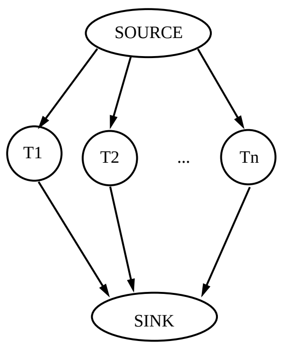
Proof The work done by a parallel task is defined by the product of the number of processors on which it is executed, say 'p' and its execution time on 'p' processors. Since every task is perfectly scalable, the work done by each task in G is independent of the number of processors allocated to it and is equal to its execution time on one processor. The lower bound on the makespan of G, M_{LB} is $M_{LB} = \frac{\sum_{t \in G} et(t,1)}{P}$, where $et(t,1)$ represents the execution time of task t on 1 processor. As a pure data parallel schedule is packed with no idle slots or "holes", it achieves this lower bound and hence is the optimal schedule.

We present a theoretical analysis of the performance of our algorithm for some special topologies of task graphs with perfectly scalable tasks, followed by experimental evaluations for arbitrary DAGs.

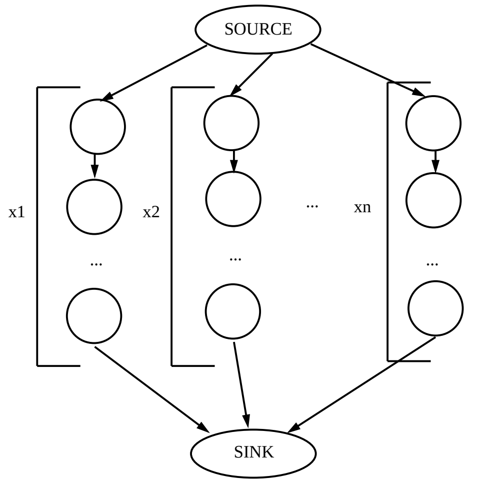
Linear Chains: Consider DAGs consisting of a linear chain of tasks, i.e the set of tasks $T1$ to Tn , $n \geq 1$ are connected in a sequence. $T1$ has no predecessor task, Tn has no successor task, and every other task has exactly one predecessor and one successor. Tasks have perfectly linear speedup and inter-task communication costs are negligible. For these task graphs, it is trivial to see that LoC-MPS generates the optimal data-parallel schedule. The initial processor allocation, as described in Section 4.1, will be $\forall t np(t) = P_{best}(t)$, which is a data-parallel schedule where each task is executed on all available processors. Algorithms like CPR and CPA will also produce data-parallel schedules in this case. Infact, even if we relax our task model to include tasks with sub-linear speedups, these algorithms will produce the optimal schedule, where each task is allocated the number of processors on which their minimum execution time is expected.

Diamond DAGs: A diamond DAG consists of 4 tasks arranged as given in Figure 1(a). Consider a diamond DAG with perfectly scalable tasks and negligible inter-task communication costs. LoC-MPS produces the optimal data-parallel schedule for this DAG.

Proof Assuming that we have P processors to schedule the diamond DAG, LoC-MPS starts with an initial allocation of P processors for tasks $T1$ and $T4$ and 1 processor each to $T2$ and $T3$ (Section 4.1). At each iteration, LoC-MPS picks $T2$ or $T3$, whichever is on the critical path and increases its processor allocation by one additional processor. As look-ahead depth is $2 \times \max_{t \in V}(P - np(t))$ (Section 4.3), which is enough for both tasks $T2$ and $T3$ to transform into a data-parallel schedule, LoC-MPS produces a data-parallel schedule which is optimal.



(a)



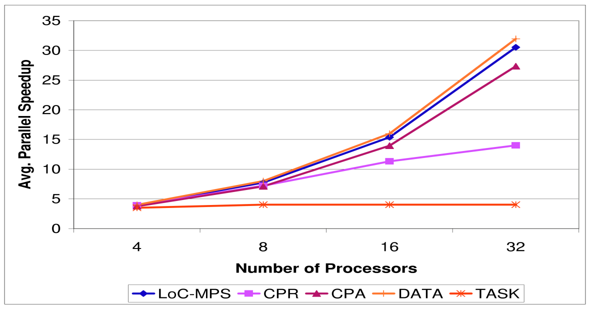
(b)

Figure 10. A task graph of (a) independent identical parallel tasks, (b) parallel chains of identical tasks.

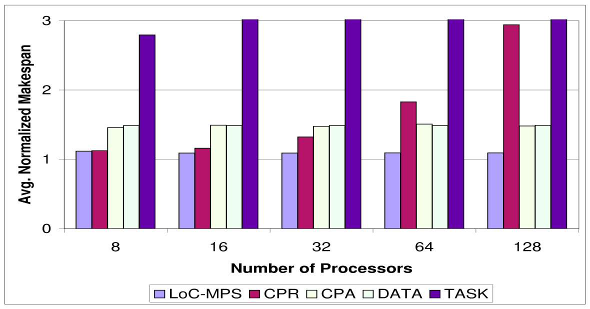
Infact, LoC-MPS generates optimal schedules even for diamond DAGs where tasks have arbitrary speedup functions. This is because our initial processor allocation always begins with allocating to $T1$ and $T4$ the optimal number of processors on which their execution time is minimum. For $T2$ and $T3$, since the look-ahead depth is large enough to explore all allocations from pure task- to pure data-parallel, LoC-MPS can identify the optimal allocation. On the other hand, CPR will get caught in a local minima if there exists two critical paths. For example, for diamond DAGs where $T2$ and $T3$ are identical, CPR will output a pure task-parallel schedule, even when the tasks have linear speedup and a pure data-parallel schedule is optimal. CPA decouples the allocation and scheduling phases, and hence, can also output sub-optimal schedules. For a diamond DAG with identical tasks having linear speedup, CPA will allocate $\lceil \frac{P}{2} \rceil$ processors to tasks $T2$ and $T3$ each. If P is not even, for example, if $P = 7$, this will allocate 4 processors each to $T2$ and $T3$, resulting in a makespan of $\frac{11}{14} \times et(t, 1)$, when all tasks are identical and have perfectly linear speedup. This is 1.375 times the optimal makespan.

Task Graphs with Independent Identical Parallel Tasks: A task graph representing a collection of independent identical parallel tasks is shown in Figure 10(a). All tasks have perfectly linear speedup and inter-task communication costs are negligible. Let there be n tasks in the task graph and the task graph is to be scheduled on P processors. The makespan generated by LoC-MPS is atmost twice the optimal makespan.

Proof Case 1: $P \geq n$**:** Let $P = a \times n + b$, where $a \geq 1$ and $0 \leq b < n$. LoC-MPS will start with an initial allocation of one processor to each task and then keep incrementing the processor allocation of each task by one additional processor in a round robin fashion until each task is allocated a processors. At this point, the makespan of the task graph is $\frac{et(t,1)}{a}$, where $et(t, 1)$ is the execution time of any task t in the task graph on one processor (please note that all tasks are identical). If $b = 0$, this is the optimal makespan. If b is non-zero, b tasks are allocated one more processor, after which an increase in processor allocation of a task will cause an increase in makespan due to resource limitations. Since, the look-ahead depth may not be sufficient to transfrom all tasks in the task graph to a pure data-parallel schedule, the



(a)



(b)

Figure 11. Performance of the scheduling algorithms for tasks graphs with known optimal schedules or lower bounds (a) random DAGs with perfectly scalable tasks (b) random tree DAGs with speedup function p^{α}.

worst case makespan of any schedule produced by LoC-MPS will be $\frac{et(t,1)}{a}$. The optimal makespan, which is generated by a pure data-parallel schedule is $n \times \frac{et(t,1)}{a \times n + b}$. Hence, the worst case makespan of LoC-MPS is $1 + \frac{b}{a \times n}$ times the optimal. Since $a \geq 1$ and $0 \leq b < n$, $\frac{b}{a \times n} < 1$. Hence, the makespan generated by LoC-MPS is atmost twice the optimal.

Case 2: $P < n$**:** If $n \bmod P$ is zero, a pure task-parallel schedule, i.e allocating one processor to each task, will pack the schedule tightly with no idle slots and will be optimal. If $n \bmod P$ is non-zero, the worst case makespan of LoC-MPS, where each task is allocated one processor, is $(\lfloor \frac{n}{P} \rfloor + 1) \times et(t,1)$, where $et(t,1)$ is the execution time of any task t in the task graph on one processors (please note that all tasks are identical). The optimal makespan is $n \times \frac{et(t,1)}{P}$. Hence, the worst case makespan of LoC-MPS is $(\lfloor \frac{n}{P} \rfloor + 1) \times \frac{P}{n}$ times the optimal. Since $P < n$, $(\lfloor \frac{n}{P} \rfloor + 1) \times \frac{P}{n}$ is less than 2. Hence, the makespan generated by LoC-MPS is atmost twice the optimal.

On the other hand, CPR will be caught in a local minima and always output a task-parallel schedule with one processor assigned to each task. Hence its worst case makespan is $\frac{P}{n}$ times the optimal when $P > n$, which is unbounded. When $P < n$, the worst case makespan of CPR is atmost twice the optimal.

If we relax the task graph topology to be a set of n parallel chains of identical tasks as shown in Figure 10(b), we can prove that the worst case makespan of LoC-MPS is atmost twice the optimal along similar lines as the proof above, under the constraint that $P = a \times (x_1 + x_2 + ...x_n) + b$ and $a \geq 1$ and $0 \leq b < x_1 + x_2 + ...x_n$, where n is the number of parallel chains.

Having analyzed the performance of our algorithm for some specific task graph topologies of perfectly scalable tasks, we now present experimental evaluations using arbitrary DAGs. Figure 11(a) shows the average parallel speedup achieved by the different schemes on 20 random task graphs from STG with all tasks having perfectly linear speedup. DATA provides a provably optimal schedule for these DAGs. LoC-MPS provides speedups within 5% of DATA. CPA generates speedups within 11% of DATA, while CPR performs worse by up to 52% at 32 processors. CPR produces larger makespans because it avoids any processor allocation that shows an increase in makespan and hence is prone to be caught in a local minima. Therefore, it fails to realize a data parallel schedule. In contrast, the look-ahead technique integrated into LoC-MPS alleviates this problem and enables it to perform well.

The second set of task graphs contains randomly generated tree DAGs where all tasks have the same speedup function of the form p^α, $0 < \alpha < 1$, i.e., the execution time of a task t on p processors is given by $et(t,p) = et(t,1)/p^\alpha$. Prasanna et al. [25] have derived the optimal schedule for this class of DAGs assuming fractional processing power. Since our system does not allow for fractional processing power, the optimal makespan may not be achievable and hence would in fact define the lower bound. For this class of graphs the different algorithms are compared with respect to this lower bound.

In Figure 11(b), we study the performance on 20 randomly generated tree DAGs with all tasks having same speedup function of the form p^α. We compare the makespan produced by each of the algorithms normalized to the lower bound computed using the approach in [25]. We see that LoC-MPS is within 12% of the lower bound, CPR is within 32% up to 32 processors but performs worse for larger number of processors. CPA and DATA are within 48% and 49% respectively, while TASK performs the worst. As tree DAGS may exhibit significant amount of concurrency, intelligent choice of the best candidate considering both task scalability and the task graph structure enables LoC-MPS to perform well. In addition, the many branches in tree DAGs can easily trap schemes like CPR in a local minima, while the look-ahead technique in LoC-MPS may prove useful in handling this problem. Thus, due to these features, LoC-MPS achieves schedules closest to the optimal or lower bound as compared to other schemes.

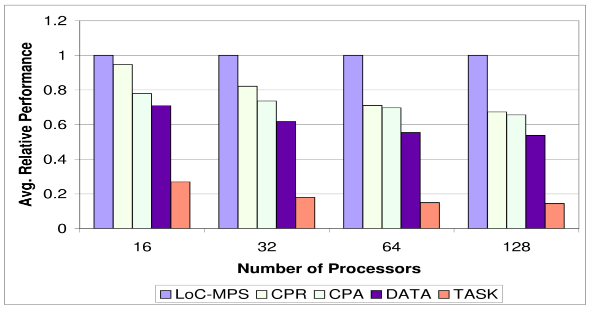
5.3 Synthetic Task Graphs

To study the impact of scalability characteristics of tasks and communication to computation ratios on the performance of the scheduling algorithms, a set of 30 synthetic graphs was generated using a DAG generation tool [2]. The number of tasks was varied from 10 to 50 and the average out-degree and in-degree per task was 4. The uni-processor computation time of each task in a task graph was generated as a uniform random variable with mean equal to 30. As our task model comprises of parallel tasks whose execution times vary as a function of the number of processors allocated, the communication-to-computation ratio (CCR) is defined for the instance of the task graph where each task is allocated one processor. Therefore, the communication cost of an edge was randomly selected from a uniform distribution with mean equal to 30 (the average uni-processor computation time) times the specified value of CCR. The data volume associated with an edge was determined as the product of the communication cost of the edge and the bandwidth of the underlying network, which was assumed to be a 100 Mbps fast ethernet network. In this paper, CCR values of 0, 0.1 and 1.0 were considered to model applications that are compute-intensive as well as those that have comparable communication costs.

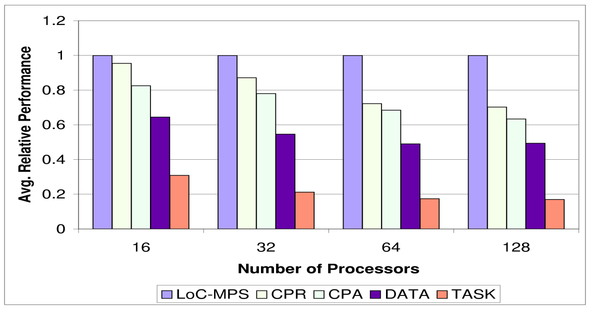
Parallel task speedup was generated using Downey's model [11] described in the section 5.1. A σ value of 0 in this model, indicates perfect scalability while higher values denote poor scalability. Workloads with varying scalability were created by generating A as a uniform random value in $[1, A_{max}]$. We evaluated the various schemes with (A_{max}, σ) as $(64, 1)$ and $(48, 2)$.

Figure 12 shows the relative performance of the algorithms for varying scalability, when the communication costs are insignificant, i.e., CCR = 0. As expected, we find that the performance of DATA decreases as we increase σ and decrease A_{max} as tasks become less scalable. LoC-MPS shows better performance benefits with increasing number of processors and achieves up to 30%, 37%, 86%, and 51% improvement over CPR, CPA, TASK, and DATA respectively.

Figure 13 presents the performance of the scheduling schemes as CCR and task scalability is varied. As CCR is increased, we see that CPR and CPA perform poorly. Though CPR and CPA model inter-task communication costs, they do not maximize inter-task data reuse through the use of a locality aware



(a)



(b)

Figure 12. Performance on synthetic graphs with CCR=0 (a) $A_{max} = 64, \sigma = 1$ **(b)** $A_{max} = 48, \sigma = 2$**.**

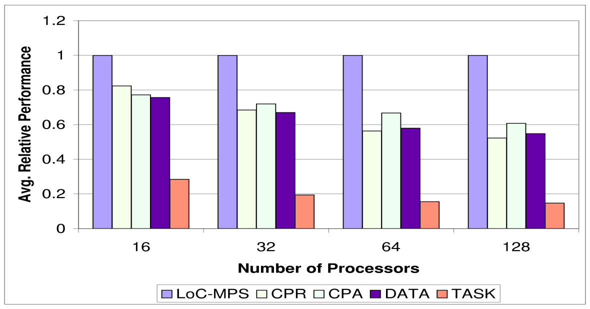
scheduling algorithm and hence performance deteriorates as CCR increases. On the other hand, relative performance of DATA improves as CCR is increased. This is because DATA incurs no communication and re-distribution costs as we assume a block-cyclic distribution of data across the processors. Therefore, as communication costs become significant, the relative performance of DATA improves. However, as the number of processors in the system is increased, performance of DATA suffers due to imperfect scalability of the tasks. LoC-MPS is able to perform well even with increase in CCR as it schedules tasks in a way that optimizes data reuse.

Due to the complexity of the locality conscious backfill scheduling algorithm, it was noticed that the scheduling overheads associated with LoC-MPS with backfill was close to 25 seconds on 128 processors. Therefore, to study the trade off between performance and scheduling time, we compared the performance of our algorithm coupled with backfilling against our algorithm without backfilling. The latter scheme schedules a task on the subset of processors that gives its minimum completion time while taking into account the data locality, but keeps track of only the latest free time of each processor rather than the idle slots in the schedule. In addition, we parallelized the locality conscious backfill scheduling phase of our algorithm, to alleviate some of the scheduling overheads, while maintaining the same quality as obtained with backfilling.

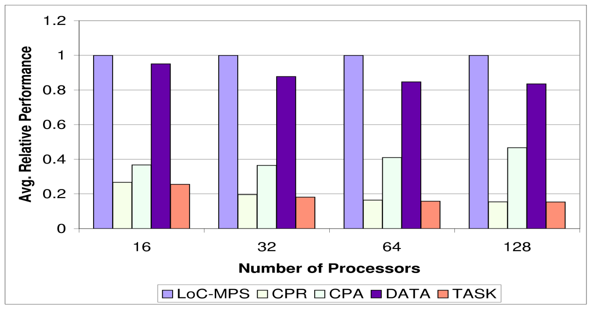
Figure 14 compares the performance and scheduling times of the above three schemes. The no-backfill approach has lower scheduling overheads but is up to 8% worse in performance as compared to backfill scheduling. The parallel version is able to generate schedules with similar quality as backfilling but with much lower scheduling overheads. Furthermore, in evaluations with real applications (presented in Section 5.4), the scheduling overheads of LoC-MPS with backfilling was found to be at least two orders of magnitude smaller than the makespans, denoting that the scheduling overheads do not impact overall performance.

5.4 Application Task Graphs

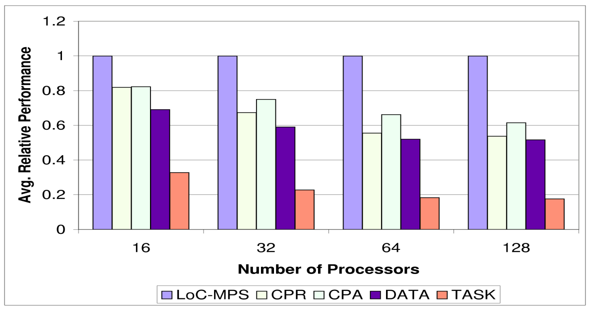
The first task graph in this group comes from an application called Tensor Contraction Engine (TCE). The Tensor Contraction Engine [4] is a domain-specific compiler to facilitate implementation of ab initio quantum chemistry models. The TCE takes a high-level specification of a computation expressed as a set of tensor contraction expressions as its input, and transforms it into efficient parallel code. The tensor contractions are generalized matrix multiplications in a computation that form a directed acyclic



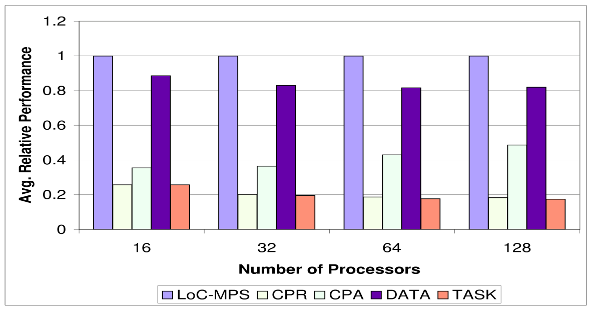
(a)



(b)



(c)

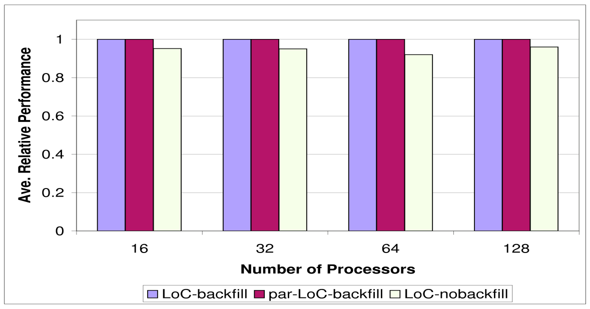


(d)

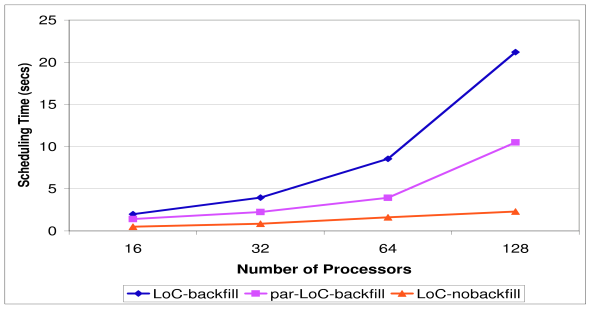
Figure 13. Performance on synthetic graphs for varying CCR and scalability (a) CCR=0.1, $A_{max} = 64, \sigma = 1$ **(b) CCR=1,** $A_{max} = 64, \sigma = 1$**, (c) CCR=0.1,** $A_{max} = 48, \sigma = 2$ **and (d) CCR=1,** $A_{max} = 48, \sigma = 2$**.**

graph, and are processed over multiple iterations until convergence is achieved. Equations from the coupled-cluster theory with single and double excitations (CCSD and CCD) were used to evaluate the scheduling schemes. Figures 15(a) and (b) display the DAGs for the CCSD T1 computation and CCD T2 computation. Each vertex in the DAG represents a tensor contraction of two input tensors to generate a result tensor. The edges in the figure denote inter-task dependences and hence many of the vertices have a single incident edge. Some of the results are accumulated to form a partial product. Contractions that take a partial product and another tensor as input have multiple incident edges. The second application is Strassen Matrix Multiplication [14], shown in Figure 15(c). The vertices represent matrix operations and the edges represent inter-task dependences. The speedup curves of the tasks in these applications were obtained by profiling them on a cluster of Itanium-2 machines with 4GB memory per node and connected by a 2Gbps Myrinet interconnect.

Figure 16 presents the performance of the scheduling algorithms for the CCSD T1 equation, evaluated through simulations. As overlap of computation and communication may not always be feasible, evaluations under two system models assuming: 1) complete overlap of computation and communication and 2) no overlap of computation and communication, are included. Currently, the TCE task graphs are executed using a pure data-parallel schedule. As the CCSD T1 DAG is characterized by a few large tasks and many small tasks which are not scalable, DATA performs poorly. On systems with complete overlap of computation and communication, LoC-MPS with backfill scheduling shows up to 13%, and

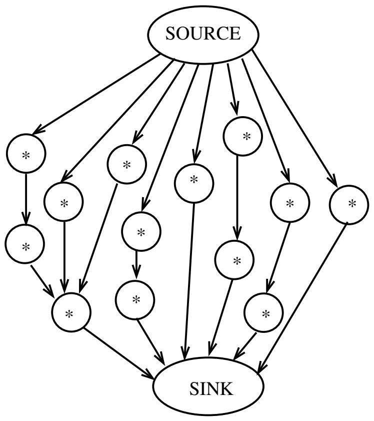


(a)

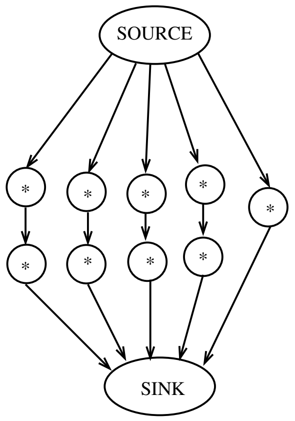


(b)

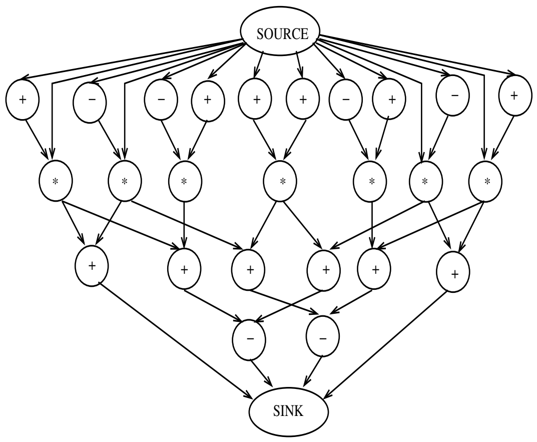
Figure 14. Comparison of (a) performance and (b) scheduling times of variants of LoC-MPS on synthetic graphs with CCR=0.1, $A_{max} = 48$ and $\sigma = 2$.



(a)



(b)



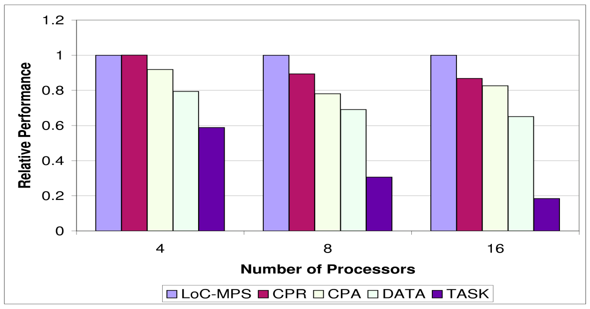
(c)

Figure 15. Task graphs for (a) CCSD T1 computation (b) CCD T2 computation (c) Strassen Matrix Multiplication.

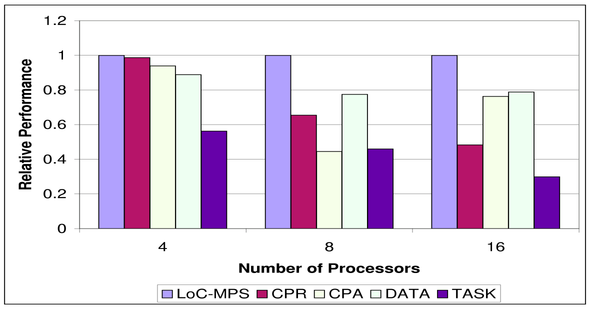
17% improvement over CPR, and CPA respectively and up to 82% and 35% improvement over TASK and DATA respectively. On systems with no overlap of computation and communication, the benefit of LoC-MPS over CPR and CPA is larger, as LoC-MPS minimizes communication overheads by maximizing inter-task data reuse, and communication proves more expensive when it cannot be overlapped with computation. In contrast, the relative performance of DATA improves as it does not incur any redistribution cost.

Figure 17 shows the performance results and scheduling times for an actual execution of the CCSD T1 computation on a Itanium-2 cluster of 900MHz dual processor nodes with 4GB memory per node and inter-connected through a 2Gbps Myrinet interconnect. We see trends similar to the simulation runs modeling systems with no overlap. At 16 processors, LoC-MPS achieves up to 48%, 25%, 22% and 68% better performance than CPR, CPA, DATA and TASK repsectively. The scheduling times are only a fraction of a second, suggesting that scheduling is not a time critical operation for this application.

The performance results for CCD T2 computation is presented in Figure 18. We notice that the actual

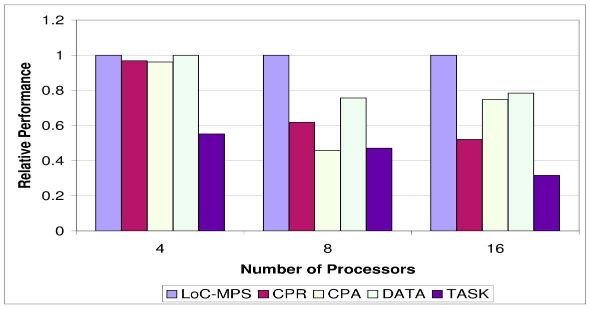


(a)

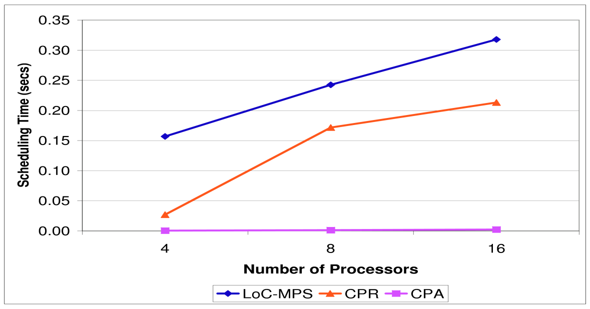


(b)

Figure 16. Performance for CCSD T1 computation through simulations (a) System with overlap of computation and communication (b) System with no overlap of computation and communication.



(a)

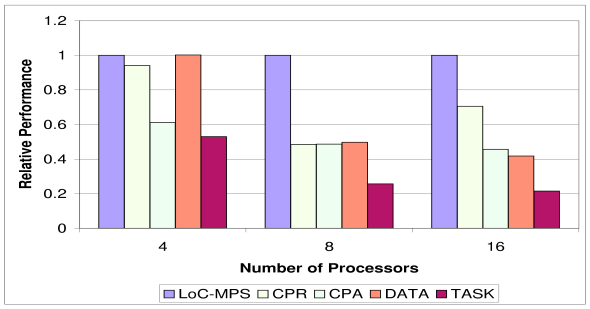


(b)

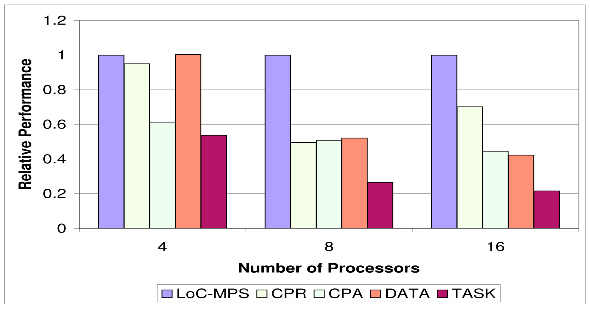
Figure 17. CCSD T1 computation (a) Performance of actual execution (b) Scheduling Times.

execution results show similar trends as the simulation results. Similar to CCSD T1 computation, CCD T2 task graph is also characterized by a few large tasks while the rest are small tasks that do not scale well. Hence, as the number of processors used is increased, performance of DATA deteriorates. LoC-MPS shows good performance benefit over the other schemes, especially as the number of processors is increased. Similar to CCSD T1, all schemes show scheduling times of only a fraction of a second (Figure 19(a)). Figure 19(b) plots the end to end runtime, which is computed as the sum of the applications execution time and scheduling time. The scheduling times are two orders of magnitude smaller than the application execution time, hence we see that the scheduling overheads do not impact the performance trends.

Figure 20 shows the performance for Strassen through simulation and actual execution for 1024×1024 matrix size. The results of actual execution closely follow the simulation results and LoC-MPS shows good improvement over CPR and CPA up to 43% and 62% respectively and up to 41% and 60% improvement over TASK and DATA respectively. Scheduling times and end to end runtimes show similar trends as for the TCE applications (Figure 21). In the case of strassen, CPR generally outperforms CPA, which is in conformance with trends in previous works [29]. However, this trend is not true of all synthetic and application task graphs. As CPR is liable to be trapped in a local minima, it may perform worse than CPA in some instances. Also, as both CPR and CPA do not model data locality while

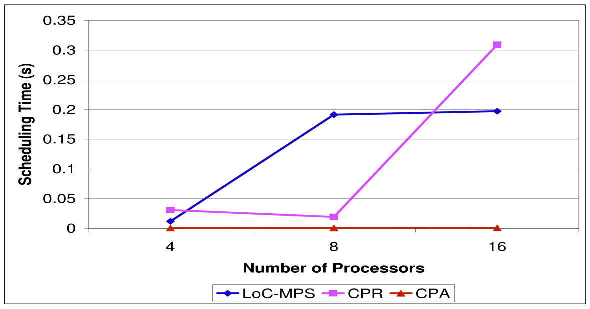


(a)

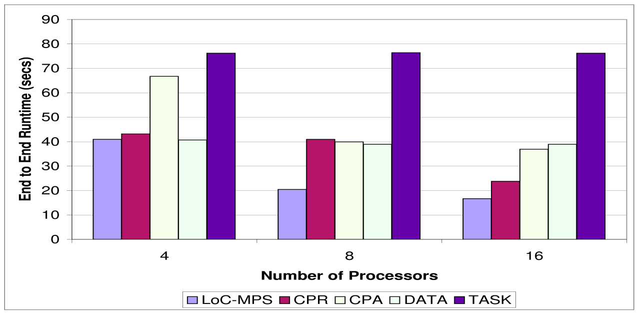


(b)

Figure 18. Performance for CCD T2 computation through (a) Simulation and (b) Actual execution.



(a)



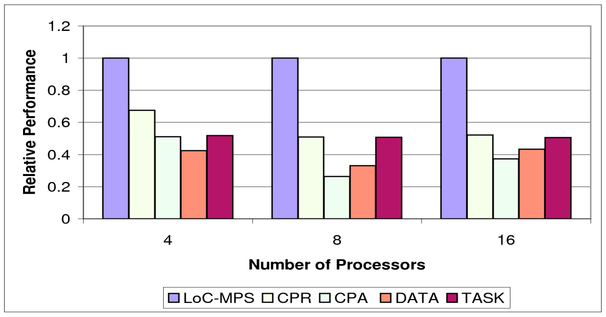
(b)

Figure 19. Performance of CCD T2 computation (a) Scheduling Times and (b) End to End Runtimes.

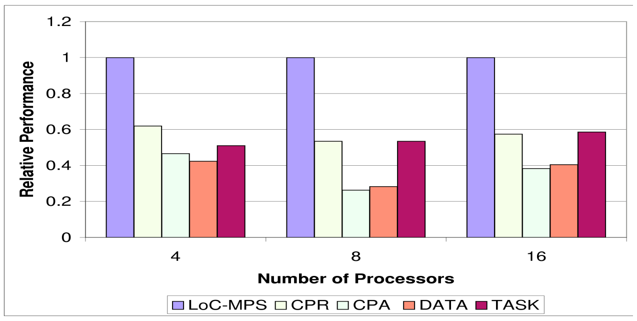
scheduling, depending on the choice of processor sets for task execution, they can incur largely different communication costs, especially for larger CCR values. This can affect their relative performance trends.

6 Conclusions

This paper presents a locality conscious scheduling strategy for minimizing the makespan of mixed parallel applications. Given an application task graph with the scalability characteristics of the constituent tasks and inter-task data communication volumes, the proposed algorithm computes the appropriate mix of task- and data-parallelism by deciding the best processor allocation to tasks and schedule. The processor allocation and scheduling decisions are made in an integrated manner. Inter-task data reuse and effective processor utilization is optimized by employing a locality conscious backfill scheduling strategy and a bounded look-ahead mechanism is used to escape local minima. Evaluation using both synthetic and application task graphs through simulations and actual executions show that the proposed algorithm achieves significant performance improvement over previously proposed schemes and is robust to changing communication costs and/or task scalability.

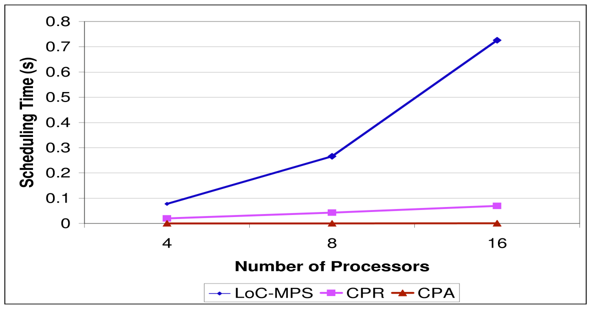


(a)

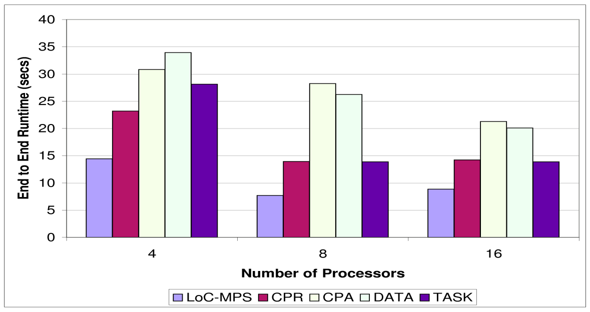


(b)

Figure 20. Performance for Strassen Matrix Multiplication through (a) Simulation and (b) Actual execution.



(a)



(b)

Figure 21. Performance of Strassen Matrix Multiplication (a) Scheduling Times and (b) End to End Runtimes.

References

[1] Standard task graph set. Kasahara Laboratory, Waseda University. http://www.kasahara.elec.waseda.ac.jp/schedule.

[2] Task graphs for free. http://ziyang.ece.northwestern.edu/tgff/index.html.

[3] J. Barbosa, C. Morais, R. Nobrega, and A. Monteiro. Static scheduling of dependent parallel tasks on heterogeneous clusters. In *4th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 1–8, Boston, MA, USA, 2005. IEEE CS Press.

[4] G. Baumgartner, D. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry. In *Proceedings of Supercomputing 2002*, November 2002.

[5] P. B. Bhat, C. S. Raghavendra, and V. K. Prasanna. Efficient collective communication in distributed heterogeneous systems. In *ICDCS 99: 19th International Conference on Distributed Computing Systems*, pages 15–24, 1999.

[6] J. Blazewicz, M. Machowiak, J. Weglarz, M. Kovalyov, and D. Trystram. Scheduling malleable tasks on parallel processors to minimize the makespan. *Annals of Operations Research*, 129(1-4):65–80, 2004.

[7] V. Boudet, F. Desprez, and F. Suter. One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, Nice - France, Apr. 2003. IEEE Computer Society.

[8] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the benefits of mixed data and task parallelism. In *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 74–83, 1995.

[9] A. N. Choudhary, B. Narahari, D. M. Nicol, and R. Simha. Optimal processor assignment for a class of pipelined computations. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):439–445, 1994.

[10] M. Cosnard and M. Loi. Automatic task graph generation techniques. *Parallel Processing Letters*, 5(4):527–538, December 1995.

[11] A. B. Downey. A model for speedup of parallel programs. Technical Report CSD-97-933, 1997.

[12] A. B. Downey. A parallel workload model and its implications for processor allocation. In *HPDC '97: Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC '97)*, page 112, 1997.

[13] J. Du and J. Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal of Discrete Mathematics*, 2(4):473–487, 1989.

[14] G. H. Golub and C. F. V. Loan. *Matrix computations (3rd ed.).* Johns Hopkins University Press, 1996.

[15] S. B. Hassen, H. E. Bal, and C. J. H. Jacobs. A task and data-parallel programming language based on shared objects. *ACM Transactions on Programming Languages and Systems*, 20(6):1131–1170, 1998.

[16] M. Iverson, F. Özgüner, and L. Potter. Statistical prediction of task execution times through analytical benchmarking for scheduling in a heterogeneous environment. *IEEE Transactions on Computers*, 48(12):1374–1379, December 1999.

[17] K. Jansen and L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 490–498, 1999.

[18] K. Jansen and H. Zhang. An approximation algorithm for scheduling malleable tasks under general precedence constraints. *ACM Transactions on Algorithms*, 2(3):416–434, 2006.

[19] H. Kasahara and S. Narita. Parallel processing of robot-arm control computation on a multiprocessor system. *IEEE Journal of Robotics and Automation*, A-1(2):104–113, 1985.

[20] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[21] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Survey*, 31(4):406–471, 1999.

[22] R. Lepere, D. Trystram, and G. J. Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. *International Journal of Foundations of Computer Science*, 13(4):613–627, 2002.

[23] K. Li. Scheduling precedence constrained parallel tasks on multiprocessors using the harmonic system partitioning scheme. *Journal of Information Sciences and Engineering*, 21(2):309–326, 2005.

[24] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal of Computing*, 19(2):322–328, 1990.

[25] G. N. S. Prasanna and B. R. Musicus. Generalised multiprocessor scheduling using optimal control. In *SPAA '91: Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 216–228, 1991.

[26] L. Prylli and B. Tourancheau. Fast runtime block cyclic data redistribution on multiprocessors. *Journal of Parallel and Distributed Computing*, 45(1):63–72, 1997.

[27] M. J. Quinn. *Parallel Computing (2nd ed.): Theory and Practice*. McGraw-Hill, Inc., New York, NY, USA, 1994.

[28] A. Radulescu, C. Nicolescu, A. J. C. van Gemund, and P. Jonker. Cpr: Mixed task and data parallel scheduling for distributed systems. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 39, 2001.

[29] A. Radulescu and A. van Gemund. A low-cost approach towards mixed task and data parallel scheduling. In *Proceedings of International Conference on Parallel Processing*, pages 69 –76, September 2001.

[30] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A framework for exploiting task and data parallelism on distributed memory multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1098–1116, 1997.

[31] T. Rauber and G. Rünger. Compiler support for task scheduling in hierarchical execution models. *Journal of System Architecture*, 45(6-7):483–503, 1999.

[32] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. In *Proceedings of the International Conference on Parallel Processing Workshops*, pages 514–519, 2002.

[33] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 62–71, 1996.

[34] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms scheduling parallelizable tasks. In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 323–332, 1992.

[35] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz. An integrated approach for processor allocation and scheduling of mixed-parallel applications. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 443–450, Washington, DC, USA, 2006. IEEE Computer Society.

[36] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz. Locality conscious processor allocation and scheduling for mixed-parallel applications. In *Cluster '06: Proceedings of the IEEE International Conference on Cluster Computing*, pages 1–10, 2006.