Locality Conscious Processor Allocation and Scheduling for Mixed Parallel Applications

N. Vydyanathan[†], S. Krishnamoorthy[†], G. Sabin[†], U. Catalyurek[‡],
T. Kurc[‡], P. Sadayappan[†], J. Saltz[‡]
[†] Dept. of Computer Science and Engineering,
[‡] Dept. of Biomedical Informatics
The Ohio State University

Abstract

Computationally complex applications can often be viewed as a collection of coarse-grained data-parallel application components with precedence constraints. These applications can be modeled as directed acyclic task graphs, with vertices representing parallel computations and edges denoting precedence constraints and data dependences. In the past, researchers have shown that combining task and data parallelism (mixed parallelism) can be an effective execution paradigm for these applications. In this paper, we present an algorithm to compute the appropriate mix of task and data parallelism based on the scalability characteristics of the tasks as well as the inter-task data communication costs, such that the parallel completion time (*makespan*) is minimized. The algorithm iteratively reduces the makespan by increasing the degree of data parallelism of tasks on the critical path that have good scalability and a low degree of potential task parallelism. Data communication costs along the critical path are minimized by exploiting parallel transfer mechanisms and use of a locality conscious backfill scheduler. A look-ahead technique is integrated into our approach to escape local minima. Evaluation using benchmark task graphs derived from real applications as well as synthetic graphs shows that our algorithm consistently performs better than previously proposed scheduling schemes.

I. Introduction

Parallel scientific applications and workflows can often be decomposed into a set of coarse-grained data-parallel tasks with precedence constraints that signify data dependences. These applications can benefit from two forms of parallelism: task and data parallelism. In a purely task-parallel approach, 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 purely data-parallel approach, the tasks are run in a sequence on all available processors. However, a pure task-parallel or data-parallel approach may not be the optimal execution paradigm. Many applications exhibit limited task parallelism due to precedence constraints. Sub-linear speedups lead to the poor performance of pure data-parallel schedules. Several researchers have shown that a combination of both, called mixed parallelism, yields better speedups [1], [2], [3]. In mixed-parallel execution, several data-parallel tasks are

executed concurrently in a task-parallel manner.

In recent prior work [4], we developed an integrated algorithm for processor allocation and scheduling under the assumption that inter-task data communication costs are negligible. However, in practice, inter-task data communication and redistribution costs can be significant, especially in the context of communication- and data-intensive applications. In this paper, we extend our previous work, to model these costs and exploit data locality when computing the schedule.

This paper proposes a single-step approach for locality conscious processor allocation and scheduling of mixed-parallel execution of applications consisting of coarse-grained parallel tasks with dependences. The goal is to minimize the parallel completion time (makespan) of an application task graph, given the runtime estimates, speedup functions of the constituent tasks and the inter-task data communication volumes. The proposed algorithm is an iterative one-phase scheme that is designed to minimize the makespan of an application task graph. Starting from a pure task-parallel schedule, the proposed algorithm iteratively reduces the makespan by minimizing the computation and communication costs along the critical path. Computation costs are minimized by increasing the degree of data parallelism of tasks that exhibit good scalability and low degree of potential task parallelism. Communication costs are minimized by exploiting parallel transfer mechanisms and using a locality aware priority based backfill scheduling scheme to schedule the parallel tasks onto processor groups. Backfill scheduling also helps in improving the effective processor utilization by minimizing idle time slots. A look-ahead mechanism is in-built to escape local minima.

We evaluate the approach through comparison with our previous work, as well as two previously proposed scheduling schemes, Critical Path Reduction (CPR) [5] and Critical Path and Allocation (CPA) [6], which have been shown to give good improvement over other existing approaches like TSAS [3] and TwoL [7]. We also compare the algorithm against pure task-parallel and pure data-parallel schemes. Evaluations are done using synthetic task graphs and task graphs derived from real applications in the domains of Tensor Contraction Engine [8], [9] and Strassen Matrix Multiplication [10]. We show that our scheme consistently performs better

than other approaches, by intelligently exploiting data locality and task scalability.

This paper is organized as follows. The next section introduces the task graph and system model. Section 3 describes the proposed locality conscious allocation and scheduling algorithm. Section 4 evaluates the proposed scheduling scheme and section 5 discusses related work. Section 6 presents our conclusions and outlines possible directions for future research.

II. Task Graph and System Model

A mixed-parallel program can be represented as a macro data-flow graph [3] 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. Please note that the terms, vertices and tasks are used interchangeably in the paper.

The weight of a vertex, $v_i \in V$, corresponds to the execution time of the parallel task it represents. The execution time of a task 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. 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, v_i and v_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 v_i and v_j. This is a function of the data volume $D_{i,j}$, network characteristics, the set of processors allocated to tasks represented by v_i and v_j and the data distribution over these processor sets. This function can be computed based on the data layout scheme of the application and the network characteristics. is assumed to run non-preemptively and can start 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 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 vertex v in G, denoted by $topL(v)$, is defined as the length of the longest path from the source vertex to v, excluding the vertex weight of v. The *bottom level* of a vertex v

in G, denoted by $bottomL(v)$, is defined as the length of the longest path from v to the sink, including the vertex weight of v. Any vertex v with maximum value of the sum of $topL(v)$ and $bottomL(v)$ belongs to a critical path in G.

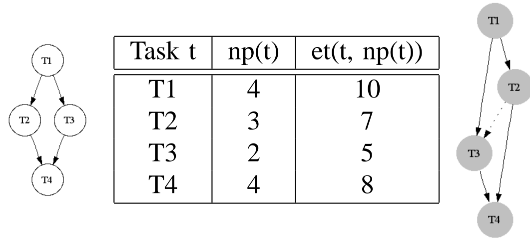
Let $st(t)$ denote the *start time* of a task t, and $ft(t)$ denote its *finish time*. A task t is eligible to start execution after all its predecessors are finished, i.e., the *earliest start time* of t is defined as $est(t) = \max_{(t',t)\in E}(ft(t') + ct(t',t))$, where $ct(t',t)$ is the communication time to transfer data from task t' to t. 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. Each parallel task distributes its output data among the processors 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. The system model initially assumes a significant overlap of computation and communication, as most clusters today are equipped with high performance interconnects which provide asynchronous communication calls. However, as this may not be feasible for systems where communication involves I/O operations at the sender/receiver, we have included evaluations of the algorithms assuming no overlap of computation and communication.

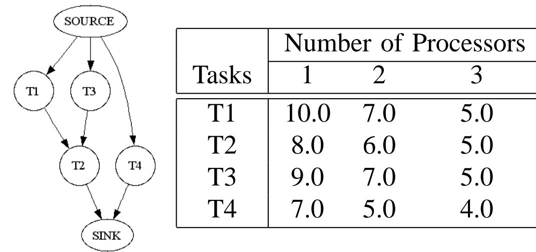
III. LOCALITY CONSCIOUS MIXED PARALLEL SCHEDULING ALGORITHM

This section presents LoC-MPS, a Locality Conscious Mixed Parallel allocation and Scheduling algorithm. LoC-MPS is an iterative algorithm that makes integrated resource allocation and scheduling decisions based on detailed knowledge of both application structure and resource availability. It minimizes the makespan of an application task graph by minimizing the computation and communication costs along the schedule's critical path, which includes dependences induced by resource constraints. Computation costs are reduced by increasing the



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

Fig. 1. (a) Task Graph G, (b) Sample allocation of processors, (c) Modified Task Graph, G'.



Tasks	Number of Processors		
	1	2	3
T1	10.0	7.0	5.0
T2	8.0	6.0	5.0
T3	9.0	7.0	5.0
T4	7.0	5.0	4.0

Fig. 2. (a) Task Graph G, (b) Execution time profile.

width of scalable tasks along the critical path with a low degree of potential task parallelism. Communication costs are minimized by exploiting inter-task data locality and by increasing the degree of parallel data transfers. A bounded look-ahead technique is used to escape local optima. The backfill scheduling algorithm employed increases the overall system utilization. These salient features allow LoC-MPS to compute better schedules than previously proposed schemes. A detailed description of the algorithm is given in the following sub-sections.

A. *Initial Allocation and Schedule-DAG Generation*

LoC-MPS starts with an initial pure task-parallel schedule (i.e., allocation of one processor to each task) and refines this schedule iteratively 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 1). 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 1(a) on 4 processors. The processor allocation information is given in Fig 1(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' (Fig 1(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.

B. Best Candidate Selection

Once the critical path of the schedule-DAG G', $CP(G')$, is computed, LoC-MPS identifies whether the makespan is dominated by computation costs or communication costs. 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. 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. The aggregate communication bandwidth $bw_{i,j}$ is given by:

$$bw_{i,j} = \min(np(t_i), np(t_j)) \times bandwidth$$

where bandwidth would correspond to the minimum of disk or memory bandwidth of the system depending on the location of data and the network bandwidth. In each iteration, LoC-MPS determines whether the makespan is dominated by the computation or communication cost, and attempts to minimize the dominating cost.

C. Minimizing Computation Cost

If computation cost dominates, LoC-MPS selects the *best candidate* from the tasks on a critical path (*candidate tasks*). The best candidate task is allocated one additional processor. We leverage our previous work [4] in choosing the best candidate task.

The choice of the best candidate is made by considering two aspects: 1) the scalability of the tasks and 2) the global structure of the DAG. The goal of choosing a best candidate task is to choose a task that can give the greatest makespan reduction. First, the improvement in execution time of each candidate task ct is computed as $et(ct, np(ct)) - et(ct, np(ct)+1)$. However, picking

the best candidate task based solely 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 be run concurrently. Consider that the task graph in 2(a) is to be scheduled on 3 processors. Each task is initially allocated one processor each. Tasks $T1$ and $T2$ lie on the critical path and $T1$ has the maximum decrease in execution time. However, increasing the processor allocation of $T1$ will serialize the execution of $T3$ or $T4$, resulting finally in a makespan of 17. A better choice in this example is to choose $T2$ as the best candidate, and schedule it on 3 processors, leading to a makespan of 15.

Taking this into account, 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. It is given by:

$$cr(t) = \frac{\sum_{t' \in c_G(t)} et(t', 1)}{et(t, 1)}$$

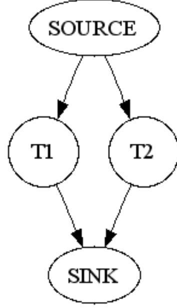
where $c_G(t)$ represents the maximal set of tasks that can run concurrent to t. A task t' is said to be concurrent to a task t in G, if there is no path between t and t' in G. This implies there is no direct or indirect dependence between t' and t, hence t' can potentially run concurrently with t. Depth First Search (DFS) is used to identify the dependent tasks. First, a DFS from task t on G 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 task which t is dependent on. The remaining tasks constitutes the maximal set of concurrent tasks in G for task t: $c_G(t) = V - (DFS(G, t) + DFS(G^T, t))$.

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. From a certain percentage of tasks at the top of the list, 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 all our experiments. To summarize, LoC-MPS widens tasks along the critical path that scale well and are competing for resources with relatively few other "heavy" tasks and thus minimizes

computation costs.

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

1: **for all** $t \in V$ **do**
2: $np(t) \leftarrow 1$
3: $best_Alloc \leftarrow \{np(t)|t \in V\}$ ▷ Best allocation is the initial allocation
4: $< G', best_sl > \leftarrow LoCBS(G, best_Alloc)$ ▷ Compute the schedule and the schedule length
5: **repeat**
6: $\{np(t)|t \in V\} \leftarrow best_Alloc$ ▷ Start with best allocation
7: $old_sl \leftarrow best_sl$ ▷ and best schedule
8: $LookAheadDepth \leftarrow 20$
9: $iter_cnt \leftarrow 0$
10: **while** $iter_cnt < LookAheadDepth$ **do**
11: $CP \leftarrow$ Critical Path in G'
12: $T_{comp} \leftarrow$ Total computation cost along CP
13: $T_{comm} \leftarrow$ Total communication cost along CP
14: **if** $T_{comp} > T_{comm}$ **then**
15: $t_{best} \leftarrow BestCandidate$ task in CP with $np(t) < \min(P, P_{best}(t))$ and t 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
16: **if** $iter_cnt = 0$ **then**
17: $entry_node \leftarrow t_{best}$ ▷ $entry_node$ signifies the start point of this look-ahead search
18: $np(t_{best}) \leftarrow np(t_{best}) + 1$
19: **else**
20: $e_{best}(t_s, t_d) \leftarrow$ heaviest edge along CP that is not marked if $iter_cnt = 0$ and $np(t_s)$ or $np(t_d) < P$
21: **if** $np(t_s) > np(t_d)$ **then**
22: $np(t_d) \leftarrow np(t_d) + 1$
23: **else if** $np(t_s) < np(t_d)$ **then**
24: $np(t_s) \leftarrow np(t_s) + 1$
25: **else**
26: $np(t_d) \leftarrow np(t_d) + 1$
27: $np(t_s) \leftarrow np(t_s) + 1$
28: **if** $iter_cnt = 0$ **then**
29: $entry_node \leftarrow e_{best}(t_s, t_d)$ ▷ $entry_node$ signifies the point of start of this look-ahead search
30: $A' \leftarrow \{np(t)|t \in V\}$
31: $< G', cur_sl > \leftarrow LoCBS(G, A')$
32: **if** $cur_sl < best_sl$ **then**
33: $best_Alloc \leftarrow \{np(t)|t \in V\}$
34: $< G', best_sl > \leftarrow LoCBS(G, best_Alloc)$
35: $iter_cnt \leftarrow iter_cnt + 1$
36: **if** $best_sl \geq old_sl$ **then**
37: Mark $entry_node$ as a bad starting point for future searches
38: **else**
39: Commit this allocation and unmark all marked tasks and edges
40: **until** all task and edges in CP are marked or for all tasks $t \in CP$, $np(t) = P$



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

Fig. 3. (a) Task Graph G, (b) Execution time profile assuming linear speedup.

D. Minimizing Communication Cost

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) exploiting data locality through the use of a locality conscious backfill 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. As data is now distributed among more nodes, communication costs can be reduced by exploiting parallel transfer mechanisms. In each iteration where communication costs dominate computation costs, LoC-MPS minimizes the cost of the heaviest edge along the critical path. Data redistribution costs are further reduced by intelligent allocation of tasks to processors using the locality conscious backfill scheduling scheme which is described in a subsequent sub-section.

E. 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. However, the only schedule modifications that decrease the makespan are used, it is possible to be trapped in local minima. Consider the DAG shown in Figure 3 and the execution profile, which corresponds to linear speedup. Assume that this DAG is to be scheduled on 4 processors. Since $T2$ is on the

critical path, its processor allocation is increased until it is no longer on the critical path. After $T2$ is allocated 3 processors, $T1$ becomes the task on the critical path. 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 both $T1$ and $T2$ on all 4 processors in a sequence, leads to the smallest makespan.

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. A bound of 20 iterations was found to yield good results in our experiments.

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 $bottomL(t_p) + \max_{t_i \in parent(t_p)} wt(e_{i,p})$
5:         $p_{best}(t_p) \leftarrow \emptyset$ ▷ Best processor set
6:         $ft_{min}(t_p) \leftarrow \infty$ ▷ Minimum finish time
7:         $free_resource_list \leftarrow \{(p, eat(p), dur(p)) | eat(p) \geq est(t_p) \wedge |p| \geq np(t_p) \wedge dur(p) \geq et(t_p, np(t_p))\}$ ▷ 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:         **while** $free_resource_list$ **not** empty **do** ▷ Traverse through this list
9:             $p' \leftarrow$ subset of processors in p that have maximum locality for t_p
10:             $rct(t_p, p') \leftarrow$ data redistribution completion time for t_p on p'
11:             $st(t_p) \leftarrow \max(eat(p), rct(t_p, p'))$
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** $st(t) > est(t)$ **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, $>$

F. Locality Conscious Backfill Scheduling (LoCBS)

Typically task graphs composed of sequential tasks with dependences are scheduled using a priority based list scheduling approach [11]. List scheduling keeps track of the latest free time for each processor, and forces all tasks to be executed in strict priority order. The strict priority

order of list scheduling tends to needlessly waste compute cycles. Parallel job schedulers use *backfilling* [12] to allow lower priority jobs to use unused processor cycles without delaying higher priority jobs, thereby increasing processor utilization and decreasing makespan. Parallel job scheduling can be viewed as a 2D chart with time along one axis and the processors along the other axis, where the objective is to efficiently pack the 2D chart (schedule) with jobs/tasks. 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" in the 2D chart and moving forward smaller jobs that fit these holes. In order to exploit data locality, rather than choosing the earliest available idle slot where a task fits, our locality conscious backfilling algorithm schedules a task on the idle slot that minimizes the tasks' completion time, taking into consideration data reuse.

Algorithm 1 outlines LoC-MPS. The algorithm starts from a pure task-parallel allocation (steps 1-2). In the main *repeat-until* loop (steps 5-40), the algorithm performs a look-ahead, starting from the current best solution (steps 10-35) and keeps track of the best solution found so far (step 32-34). 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 current 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.

$LoCBS$ 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. Scheduling the highest priority task on the set

of processors that minimizes its finish time (steps 5-16) takes $O(|V|^2 PlogP + |V|^3|E|P)$ in the worst case and adding psuedo edges takes $O(|V|PlogP)$ Thus, the worst case complexity of $loCBS$ is $O(|V|^3 PlogP + |V|^4|E|P)$. 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 10-35 has at most $|V|P$ iterations. Thus overall worst case complexity of LoC-MPS is $O(|V|P(|V|^3 PlogP + |V|^4|E|P))$. On the other hand, complexity of CPR is $O(|E||V|^2 P + |V|^3 P(log|V| + PlogP))$. CPA is a low cost algorithm with complexity $O(|V|P(|V| + |E|))$. Though LoC-MPS is more expensive than the other approaches, as the number of vertices is few in most mixed-parallel applications, the scheduling times are reasonable as seen in the experimental results.

IV. Performance Analysis

This section evaluates the quality (makespan) of the schedules generated by the locality conscious mixed parallel scheduling algorithm (LoC-MPS) against those generated by iCASLB, CPR, CPA, pure task-parallel and pure data-parallel schemes. CPR is a single-step approach, while CPA is a two-phase scheme; both have been shown in [5], [6] to perform better than other allocation and scheduling approaches such as TSAS [3]. iCASLB [4] assumes that inter-task data communication and redistribution costs are negligible in comparison to intra-task communication costs. A pure task-parallel schedule (TASK) allocates one processor to each task and the locality conscious backfill scheduling algorithm to schedule them to processors. A pure data-parallel schedule (DATA) executes tasks in a sequence, with each task using 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 in [13]. In DATA, as all tasks are executed on all processors, no redistribution cost is incurred.

The performance of the scheduling algorithms are evaluated via simulation, using both synthetic task graphs as well as task graphs from two applications.

A. Synthetic Task Graphs

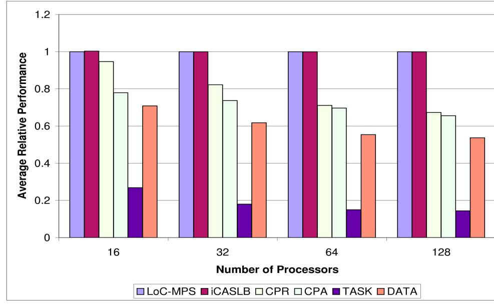
To study the impact of scalability characteristics of tasks and communication to computation ratios on the performance of the scheduling algorithms, a set of 30 synthetic graphs was generated using a DAG generation tool [14]. The number of tasks was varied from 10 to 50 and the average out-degree and in-degree per task was 4. The uniprocessor 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 uniprocessor 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 were considered to model applications that are compute-intensive as well as those that have comparable communication costs.

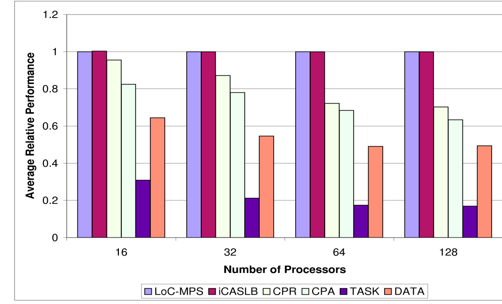
Parallel task speedup was generated using Downey's model [15]. Downey's speedup model is a non-linear function of two parameters: A, the average parallelism of a task, and σ, a measure of the variations of parallelism. According to this model, the speedup S of a task as a function of the number of processors n is given by:

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

A σ value of 0 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 $(48, 2)$.

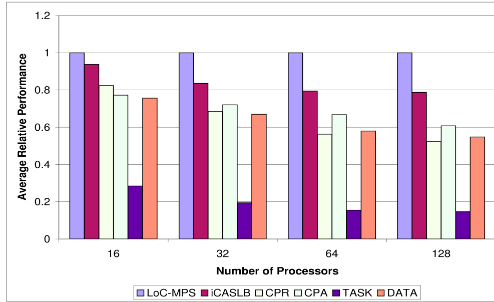
Figure 4 shows the relative performance of the scheduling algorithms as the number of processors in the system is increased, when the communication costs are insignificant, i.e.,

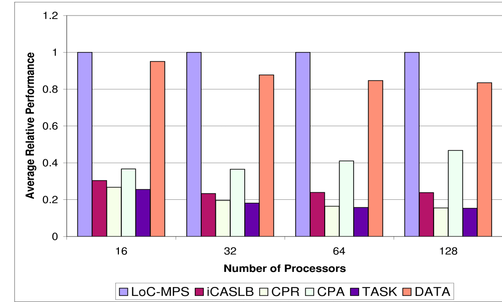




(a) (b)

Fig. 4. Performance of the scheduling schemes on synthetic graphs with CCR=0 (a) $A_{max} = 64, \sigma = 1$ (b) $A_{max} = 48, \sigma = 2$



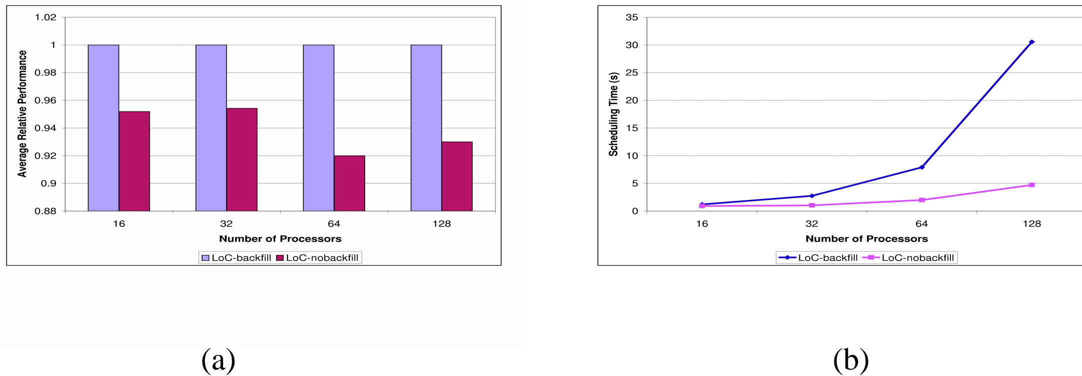


(a) (b)

Fig. 5. Performance of the scheduling schemes on synthetic graphs with $A_{max} = 64, \sigma = 1$ and (a) CCR = 0.1 (b) CCR = 1

CCR=0. 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. LoC-MPS and iCASLB show similar performance, as CCR is 0. 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 upto 30%, 37%, 86%, and 51% improvement over CPR, CPA, TASK, and DATA respectively.

Figure 5 presents the performance of the scheduling schemes as data communication costs become significant. As CCR is increased, performance of iCASLB deteriorates as it does not model inter-task communication costs. We see that CPR and CPA also perform poorly for CCR = 1. Though CPR and CPA model inter-task communication costs, they do not use a locality aware scheduling algorithm and hence performance deteriorates as CCR increases. On the other hand,



(a) (b)

Fig. 6. Comparison of performance and scheduling times of LoC-MPS on synthetic graphs with CCR=0.1, $A_{max} = 48$ and $\sigma = 2$, (a) with backfilling and (b) without backfilling

relative performance of DATA improves as CCR is increased. This is because DATA incurs no communication and re-distribution costs - we assume a block-cyclic distribution of data across the processors. Hence, as communication costs become significant, the relative performance of DATA improves, since it has no communication costs but the other schemes do. However, as the number of processors in the system is increased, performance of DATA suffers due to imperfect scalability of the tasks.

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 upto 30 seconds on 128 processors. Therefore, to study the tradeoff between performance and scheduling time, our algorithm coupled with backfilling was compared 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. Figure 6 shows the comparison of performance and scheduling times of the two schemes. We see that the no-backfill approach has lower scheduling overheads but is upto 8% worse in performance as compared to scheduling with backfill. However, in evaluations with real applications (presented in the next sub-section), the scheduling overheads of LoC-MPS with backfilling was found to be atleast two orders of magnitude smaller than the makespans.

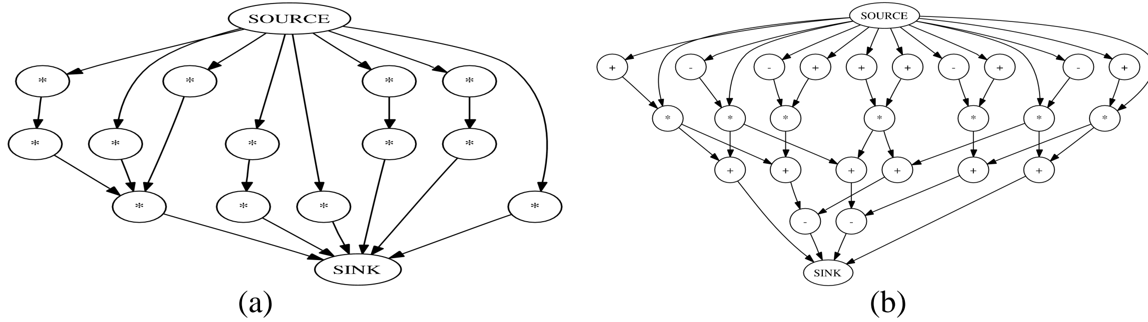


Fig. 7. Task graphs for applications (a) CCSD T1 computation and (b) Strassen Matrix Multiplication.

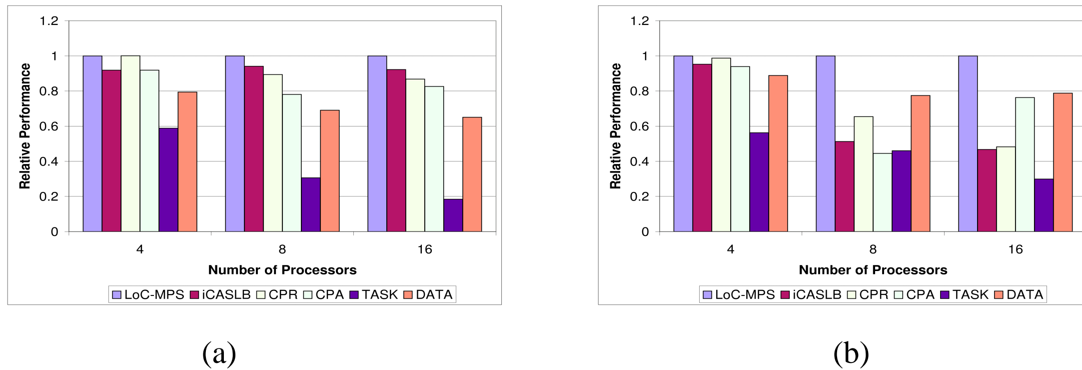


Fig. 8. Performance of the scheduling schemes for CCSD T1 computation (a) System with overlap of computation and communication (b) Systems with no overlap of computation and communication

B. Task Graphs for Applications

The first task graph in this group comes from an application called Tensor Contraction Engine (TCE). The Tensor Contraction Engine [8], [9] 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 graph, and are processed over multiple iterations until convergence is achieved. Equations from the coupled-cluster theory with single and double excitations (CCSD) were used to evaluate the scheduling schemes. Figure 7(a) displays the DAG for the CCSD-T1 computation, where each vertex 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

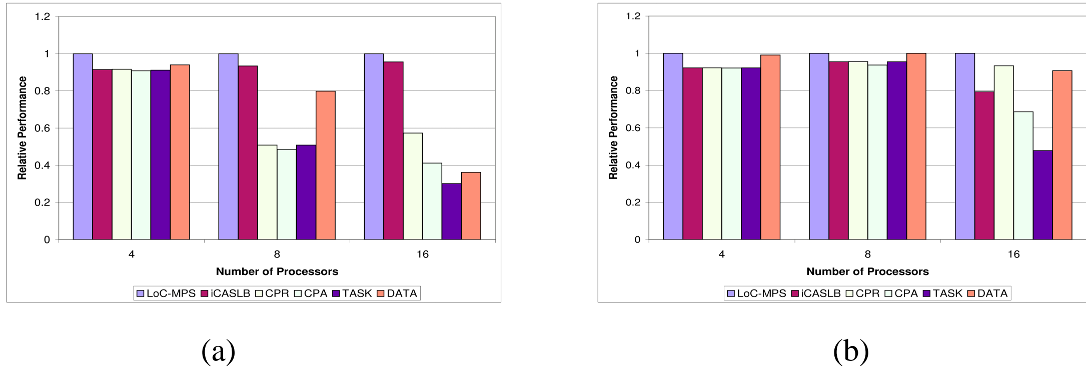


Fig. 9. Performance of the scheduling schemes for Strassen Matrix Multiplication for matrix sizes (a) 1024X1024 (b) 4096X4096

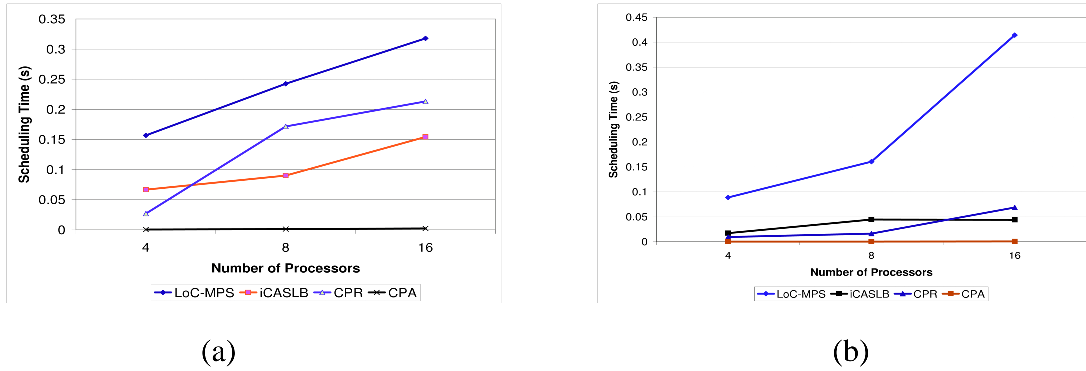


Fig. 10. Scheduling times (a) CCSD T1 computation (b) Strassen Matrix Multiplication

multiple incident edges.

The second application is Strassen' Matrix Multiplication [10], shown in Figure 7(b). The vertices represent matrix operations and the edges represent inter-task dependences. Matrix sizes of 1024×1024 and 4096×4096 were used in the experiments. 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 8 presents the performance of the various scheduling algorithms for the CCSD T1 equation. As overlap of computation and communication may not always be feasible, evaluations under two system models: 1) assuming complete overlap of computation and communication and 2) assuming no overlap of computation and communication, are included. Currently, the TCE task graphs are executed assuming 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

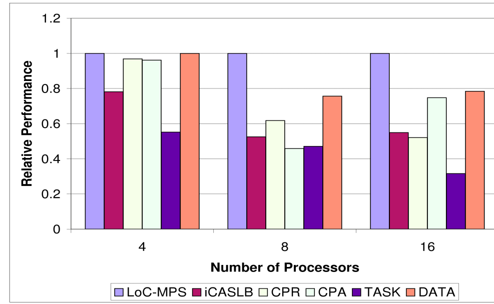


Fig. 11. Performance of actual execution of CCSD T1 computation

performs poorly. On systems with complete overlap of computation and communication, LoC-MPS with backfill scheduling shows upto 8%, 13%, and 17% improvement over iCASLB, CPR, and CPA respectively and upto 82% and 35% improvement over TASK and DATA respectively. On systems with no overlap of computation and communication, the benefits of LoC-MPS over iCASLB, CPR and CPA is enhanced, as these schemes do not minimize communication costs by exploiting data locality and communication proves more expensive when it cannot be overlapped with computation. On the other hand it is seen that the relative performance of DATA improves. This is because in the case of no overlap between communication and computation, LoC-MPS generates larger makespans than while communication can be overlapped with computation, while performance of DATA remains the same as it does not involve any communication costs.

Figure 9 shows the performance for Strassen for 1024×1024 and 4096×4096 matrix sizes. For smaller problem size (1024×1024), DATA performs poorly as the tasks do not scale very well. LoC-MPS shows good improvement over CPR and CPA upto 43% and 59% respectively and upto 70% and 64% improvement over TASK and DATA respectively. As the problem size is increased by a factor of 16, relative performance of DATA improves as the scalability of tasks increases.

The scheduling times are presented in Figure 10. Though LoC-MPS is more expensive than the other approaches, the scheduling times is at least two orders of magnitude smaller than the makespan of these applications, suggesting that scheduling is not a time critical operation for these applications.

Figure 11 shows the performance of the various schemes for the actual execution of the CCSD

T1 computation on an Itanium 2 cluster of 900 MHz dual processor nodes with 4 GB memory per node and interconnected through a 2Gbps Myrinet switch. We see trends similar to the simulation runs, with LoC-MPS showing significant performance improvement over the other schemes especially for larger number of processors.

V. Related Work

Optimal scheduling has been shown to be a hard problem to solve even in the context of sequential task graphs. Papadimitriou and Yannakakis [16] have proved that the problem of scheduling sequential tasks with precedence constraints is NP-complete. Du and Leung [17] showed that scheduling independent malleable tasks is strongly NP-hard for 5 processors, and scheduling malleable tasks with arbitrary precedence constraints is strongly NP-hard for 2 processors. Hence, several researchers have proposed heuristic solutions and approximation algorithms [18], [19], [20], [21], [22], [23].

Ramaswamy et al. [3] 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. [6], where a greedy heuristic is employed to iteratively compute the processor allocation, followed by scheduling of the tasks. Both these approaches attempt to minimize the maximum of average processor area and critical path length. However, they are limited in the quality of schedules they can produce due to the decoupling of the processor allocation and scheduling phases. Another approach proposed by Radulescu et al. [5] uses a single-step heuristic called CPR (Critical Path Reduction), for scheduling data parallel task graphs. Starting from a one-processor allocation for each task, CPR iteratively increases the processor allocation of tasks until there is no improvement in makespan. However, all of the above approaches do not consider data locality while scheduling tasks onto processors.

In recent work [4], we addressed the problem of processor allocation and scheduling for mixed-parallel applications, under the assumption that the inter-task communication costs are insignificant. In this work, we relax this assumption, and present an algorithm that minimizes the

computation and communication costs along the critical path and exploits data locality through a locality aware backfill scheduling scheme. To the best of our knowledge, this is the first work to consider data locality in such a context.

Boudet et al. [24] propose a single step approach for scheduling task graphs, where 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. In this paper, we assume a more general system model, where a parallel task may execute on any subset of processors. Li [25] has proposed a scheme for scheduling constrained rigid parallel tasks on multiprocessors, where the processor requirement of each task is fixed and known.

Some researchers have proposed approaches for optimal scheduling for specific task graph topologies. These include Subhlok and Vandron's approach for scheduling pipelined linear chains of parallel tasks [26], and Prasanna's scheme [27] for optimal scheduling of tree DAGS and series parallel graphs, where a specific form is assumed of the task speedup functions.

VI. Conclusions and Future Work

This paper presents LoC-MPS, a locality conscious scheduling strategy for mixed parallel applications. LoC-MPS minimizes the makespan of the application task graph by minimizing the computation and communication costs along the critical path and exploiting data locality through a locality conscious backfill scheduling approach. Processor allocation and scheduling decisions are made by considering the global structure of the application task graph, scalability curves of its constituent tasks, and data redistribution costs. A bounded look-ahead mechanism is employed to escape local minima and a locality conscious backfill scheduling algorithm enables effective processor utilization and reduces data transfer times. Evaluation using both synthetic task graphs and those derived from applications show that LoC-MPS achieves significant performance improvement over other schemes such as iCASLB, CPR, CPA, TASK and DATA.

Future work is planned on: 1) developing strategies to parallelize the scheduling algorithm, in order to reduce scheduling overhead, and 2) incorporation of the scheduling strategy into a run-time framework for the on-line scheduling of mixed parallel applications.

References

[1] S. B. Hassen, H. E. Bal, and C. J. H. Jacobs, "A task and data-parallel programming language based on shared objects," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 6, pp. 1131–1170, 1998.

[2] I. T. Foster and K. M. Chandy, "Fortran m: a language for modular parallel programming," *J. Parallel Distrib. Comput.*, vol. 26, no. 1, pp. 24–35, 1995.

[3] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A framework for exploiting task and data parallelism on distributed memory multicomputers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 11, pp. 1098–1116, 1997.

[4] 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 35th International Conference on Parallel Processing*, 2006, accepted for publication.

[5] A. Radulescu, C. Nicolescu, A. J. C. van Gemund, and P. Jonker, "Cpr: Mixed task and data parallel scheduling for distributed systems," in *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2001, p. 39.

[6] A. Radulescu and A. van Gemund, "A low-cost approach towards mixed task and data parallel scheduling," in *Proc. of Intl. Conf. on Parallel Processing*, September 2001, pp. 69 –76.

[7] T. Rauber and G. Rünger, "Compiler support for task scheduling in hierarchical execution models," *J. Syst. Archit.*, vol. 45, no. 6-7, pp. 483–503, 1999.

[8] 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 *Proc. of Supercomputing 2002*, November 2002.

[9] D. Cociorva, J. Wilkins, G. Baumgartner, P. S. and J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison, "Towards Automatic Synthesis of High-Performance Codes for Electronic Structure Calculations: Data Locality Optimization," in *Proc. Intl. Conf. on High Performance Computing*, vol. 2228. Springer-Verlag, 2001, pp. 237–248.

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

[11] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, 1999.

[12] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Characterization of backfilling strategies for parallel job scheduling," in *ICPPW '02: Proceedings of the International Conference on Parallel Processing Workshops*, 2002, pp. 514–519.

[13] L. Prylli and B. Tourancheau, "Fast runtime block cyclic data redistribution on multiprocessors," *J. Parallel Distrib. Comput.*, vol. 45, no. 1, pp. 63–72, 1997.

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

[15] A. B. Downey, "A model for speedup of parallel programs," http://allendowney.com/research/model/, Tech. Rep. Technical Report CSD-97-933, 1997.

[16] C. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," in *STOC '88:*

Proceedings of the twentieth annual ACM symposium on Theory of computing. New York, NY, USA: ACM Press, 1988, pp. 510–513.

[17] J. Du and J. Y.-T. Leung, "Complexity of scheduling parallel task systems," *SIAM J. Discret. Math.*, vol. 2, no. 4, pp. 473–487, 1989.

[18] 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*. New York, NY, USA: ACM Press, 1992, pp. 323–332.

[19] 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*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999, pp. 490–498.

[20] R. Lepere, D. Trystram, and G. J. Woeginger, "Approximation algorithms for scheduling malleable tasks under precedence constraints," in *ESA '01: Proceedings of the 9th Annual European Symposium on Algorithms*. London, UK: Springer-Verlag, 2001, pp. 146–157.

[21] D. Trystram, "Scheduling parallel applications using malleable tasks on clusters," in *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2001, p. 199.

[22] 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*, vol. 129, no. 1-4, pp. 65–80, 2004.

[23] K. Jansen and H. Zhang, "Scheduling malleable tasks with precedence constraints," in *SPAA'05: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures*. New York, NY, USA: ACM Press, 2005, pp. 86–95.

[24] 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: IEEE Computer Society, Apr. 2003.

[25] K. Li, "Scheduling precedence constrained parallel tasks on multiprocessors using the harmonic system partitioning scheme," *J. of Information Sciences and Engineering*, vol. 21, no. 2, pp. 309–326, 2005.

[26] 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*. New York, NY, USA: ACM Press, 1996, pp. 62–71.

[27] 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*. New York, NY, USA: ACM Press, 1991, pp. 216–228.