An Integrated Approach for Processor Allocation and Scheduling of 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 tasks with precedence constraints. Researchers have shown that combining task and data parallelism (mixed parallelism) can be an effective approach for executing these applications, as compared to pure task or data parallelism. In this paper, we present an approach to determine the appropriate mix of task and data parallelism, i.e., the set of tasks that should be run concurrently and the number of processors to be allocated to each task. An iterative algorithm is proposed that couples processor allocation and scheduling, of mixed-parallel applications on compute clusters so as to minimize the parallel completion time (makespan). Our 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. The approach employs a look-ahead technique to escape local minima and uses priority based backfill scheduling to efficiently schedule the parallel tasks onto processors. Evaluation using benchmark task graphs derived from real applications as well as synthetic graphs shows that our algorithm consistently performs better than CPR and CPA, two previously proposed scheduling schemes, as well as pure task and data parallelism.

1 Introduction

Parallel applications can often be decomposed into coarse-grained data-parallel tasks with precedence constraints that signify data and control dependences. These applications can benefit from two forms of parallelism: task and data parallelism. In a pure 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 pure data-parallel approach, the tasks are run in a sequence on all available processors. However, a pure task- or pure data-parallel approach may not be the optimal execution paradigm. Most applications exhibit limited task parallelism due to precedence constraints. The sub-linear speedups achieved leads to poor performance of pure data-parallel schedules. In fact, researchers have shown that a combination of both, called mixed parallelism, yields better speedups [17, 8]. In mixed-parallel execution, several data-parallel tasks are executed concurrently in a task-parallel manner.

This paper proposes a single-step approach for processor allocation and scheduling of mixed-parallel executions 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 and speedup functions of the constituent tasks. Starting from an initial processor allocation and schedule, the proposed algorithm iteratively reduces the makespan by increasing the degree of data parallelism of selected tasks on the critical path. A look-ahead mechanism is used to escape local minima. backfill scheduling is used to improve processor utilization. We compare the proposed approach with two previously proposed scheduling schemes: Critical Path Reduction (CPR) [15] and Critical Path and Allocation (CPA) [16], which have been shown to give good improvement over other approaches like TSAS [17] and TwoL [18], as well as pure task-parallel and pure data-parallel schemes. The approach is evaluated 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 [2] and Strassen Matrix Multiplication [7]. We show that our algorithm consistently performs better than the other scheduling approaches.

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

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

outlines possible directions for future research.

2 Task Graph Model

A mixed-parallel program can be represented as a macro data-flow graph [17] which is a weighted directed acyclic graph (DAG), $G = (V, E)$, where V, the set of vertices, represents the data-parallel tasks and E, the set of edges, represents precedence constraints. Each data-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.

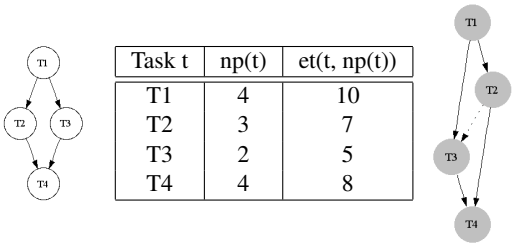
The weight of each vertex corresponds to the execution time of the parallel task it represents. The execution time of a task is a function of the number of processors allocated to it. This function can be provided by the application developer, or obtained by profiling the execution of the task on different numbers of processors. It is assumed that the communication costs within a data-parallel task dominate communication costs between data-parallel tasks. This assumption holds when each vertex of the DAG is a coarse-grained parallel program. Each task 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 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 after all its predecessors are finished, i.e., the *earliest start time* of t is defined as $est(t) = \max_{(t',t)\in E} ft(t')$. 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 t on p processors. The parallel completion time (makespan) of G is the finish time of the sink vertex.

3 Processor Allocation and Scheduling

This section describes iCASLB (an iterative Coupled processor Allocation and Scheduling algorithm with Look-ahead and Backfill), a new algorithm for processor allocation and scheduling of mixed-parallel applications to reduce the makespan of a DAG. Unlike schemes that dissociate the allocation and scheduling phases [16, 17], iCASLB is a one-phase algorithm that simultaneously determines both task allocation and scheduling. To reduce the makespan, it takes an integrated approach that can exploit detailed knowledge of both application structure and resource availability. It assigns more processors to tasks on the schedule's critical path that are scalable and have a low degree of potential task parallelism. It also uses priority based backfilling to increase utilization and look-ahead to avoid local optima.



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

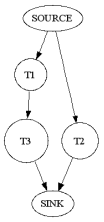
Figure 1. (a) Task Graph G, (b) Processor allocation, (c) Modified Task Graph, G'.

As confirmed by the experimental results, these features allow iCASLB to produce better schedules than previous schemes. The rest of this section presents the salient features of iCASLB in detail.

3.1 Initial Allocation and Schedule-DAG Generation

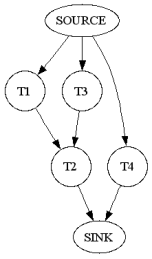
iCASLB starts with an initial allocation and schedule and iteratively reduces the makespan. To compute the initial allocation, for each task, we over-estimate the number of "possibly concurrent tasks" and compute the minimum available number of processors; assuming we allocate the best number of processors to each of those concurrent tasks. The best number of processors for a task is defined as the number of processors on which the task's minimum execution time is expected. If the number of available processors is more than one, the minimum of the task's best number of processors and the number of available processors is allocated. Otherwise one processor is allocated to the task.

iCASLB iteratively refines this initial allocation by identifying the *best candidate* task and increasing its processor allocation. *Candidate* tasks lie on the critical path of the schedule. The critical path of the schedule is given by $CP(G')$, where G', the schedule-DAG, is the original DAG G with pseudo-edges added because of *induced dependences* due to resource limitations. $CP(G')$ represents the longest path in the current schedule, hence reducing this path length will tend to reduce the makespan. The addition of pseudo-edges to form the schedule-DAG is illustrated in Figure 1. Consider scheduling the task graph in Figure 1(a)



Tasks	Number of Processors			
	1	2	3	4
T1	12.0	9.0	7.0	5.6
T2	35.0	20.0	13.0	11.0
T3	100.0	65.0	48.0	35.0

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



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

Figure 3. (a) Task Graph G, (b) Execution time profile.

on 4 processors. Due to resource limitations tasks $T2$ and $T3$ are serialized in the schedule. Hence, the modified DAG G' (Fig 1(c)) which represents the schedule, includes an additional pseudo-edge between vertices $T2$ and $T3$. The critical path length of 30 of G' is the makespan of the application.

3.2 Best Candidate Task Selection

Once the candidate tasks are selected, the *best candidate* task must be chosen for expansion in a given iteration. A poor choice of the best candidate will affect the quality of the resulting schedule. Let the task graph in 2(a) 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 51.6. On the other hand, if $T3$ were chosen, the resulting schedule would have shorter makespan of 48 by allocating 4 processors to $T3$, 1 processor to $T1$ and 3 processors to $T2$. iCASLB 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 ct is computed as $et(ct, np(ct)) - et(ct, np(ct) + 1)$. However, picking the 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 be run concurrently. Consider that the task graph in 3(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, iCASLB 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:

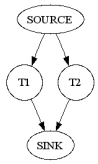
$$cr(t) = \frac{\sum_{t' \in c_G(t)} et(t', 1)}{et(t, 1)}$$

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 means that 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 $CP(G')$ 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, iCASLB widens tasks that scale well and are competing for resources with relatively few other "heavy" tasks.

3.3 Intelligent Look-ahead

Once the best candidate is selected, its processor allocation is incremented by one and a new schedule is computed using Priority-based Backfill Scheduling Algorithm described in the next sub-section. The makespan of the new schedule might be more than that of last schedule computed. If only schedules that decrease the makespan from the previous schedule were allowed, there is a possibility of getting trapped in a local minima. Consider the DAG shown in Figure 4 and the execution profile assuming linear speedup. Assume that this DAG has to be scheduled on 4 processors.



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 4. (a) Task Graph G, (b) Execution time profile (linear speedup).

As $T2$ is more critical, $T2$ would be chosen to be widened to 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 from 40.0 to 46.7. 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 of 30.0.

To alleviate this problem, iCASLB uses an intelligent look-ahead mechanism that 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. 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 data parallel execution (using the maximum number of processors).

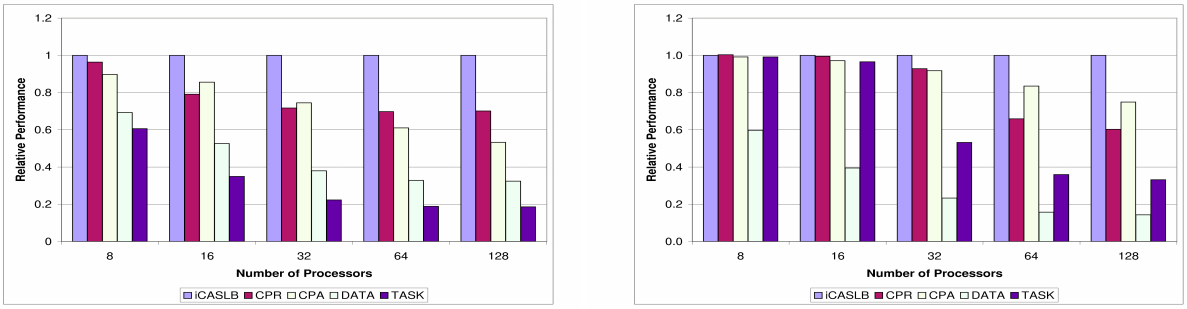
3.4 Priority Based Backfill Scheduling

Priority based list scheduling is a popular approach for scheduling task graphs containing sequential tasks with dependences [11]. The tasks are prioritized and at each scheduling step the ready task with the highest priority is scheduled. This approach keeps track of the latest free time for each processor, and forces all tasks to be executed in strict priority order.This strict priority ordering tends to needlessly waste compute cycles. Parallel job schedulers use *backfilling* [19] to allow lower priority jobs to use unused processor cycles without delaying higher priority jobs, thereby increasing processor utilization. Parallel job scheduling can be viewed as a 2D chart with time along one axis and the number of processors along the other axis, where the purpose is to efficiently pack the 2D chart (schedule) with jobs. Each job is 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 those holes. iCASLB uses a conservative backfilling strategy to backfill tasks of lower priority that fit in the "holes" as long as they do not delay a previously scheduled higher priority task.

Algorithm 1 Coupled Allocation and Scheduling

1: **for all** $t \in V$ **do**
2: $p \leftarrow P - \sum_{t' \in c_G(t)} P_{best}(t')$ ▷ number of available processors if we allocate best number of processors to each of the concurrent tasks
3: **if** $p > 1$ **then**
4: $np(t) \leftarrow \min(P_{best}(t), p)$
5: **else**
6: $np(t) \leftarrow 1$
7: $best\_Alloc \leftarrow \{(t, np(t)) | t \in V\}$ ▷ Best allocation is the initial allocation
8: $(best\_sl, G') \leftarrow PrBS(G, best\_Alloc)$
9: **repeat**
10: $\{(t, np(t)) | t \in V\} \leftarrow best\_Alloc$ ▷ Start with best allocation
11: $old\_sl \leftarrow best\_sl$ ▷ and best schedule
12: $LookAheadDepth \leftarrow 2 \times max_{t \in V}(P - np(t))$
13: $iter\_cnt \leftarrow 0$
14: **while** $iter\_cnt \leq LookAheadDepth$ **do**
15: $CP \leftarrow$ Critical Path in G'
16: $t_{best} \leftarrow BestCandidate$ in CP with $np(t) < \min(P, P_{best}(t))$ and t is not marked if $iter\_cnt = 0$
17: **if** $iter\_cnt = 0$ **then**
18: $t_{entry} \leftarrow t_{best}$ ▷ t_{entry} signifies the point of start of this look-ahead search
19: $np(t_{best}) \leftarrow np(t_{best}) + 1$
20: $A' \leftarrow \{(t, np(t)) | t \in V\}$
21: $(cur\_sl, G') \leftarrow PrBS(G, A')$
22: **if** $cur\_sl < best\_sl$ **then**
23: $best\_Alloc \leftarrow \{(t, np(t)) | t \in V\}$
24: $(best\_sl, G') \leftarrow PrBS(G, best\_Alloc)$
25: $iter\_cnt \leftarrow iter\_cnt + 1$
26: **if** $best\_sl \geq old\_sl$ **then**
27: Mark t_{entry} as a bad starting point for future searches
28: **else**
29: Commit this allocation and unmark all marked tasks
30: **until** for all tasks $t \in CP$, t is either marked or $np(t) = \min(P, P_{best}(t))$

Algorithm 1 outlines iCASLB. The initial allocation of processors to tasks is described in (steps 1-6). In the main *repeat-until* loop (steps 9-30), starting from the current best solution, the algorithm does a look-ahead (steps 14-25) and keeps the best solution found so far (step 22-24). If the look-ahead process does not yield a better solution, the task that was the first best candidate in that look-ahead process, is marked as a bad starting point for future search. However, if a better makespan was found, all marked tasks 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 in the critical path are marked or are allocated the best possible number of processors. Algorithm 2 presents the



(a) (b)

Figure 5. Relative performance for (a) Robot Control DAG (b) Sparse Matrix Solver DAG

Algorithm 2 PrBS - Priority-Based Backfill Scheduling

1: **function** PRBS($G, \{(t, np(t))|t \in V\}$)
2: $G' \leftarrow G$
3: **while not** all tasks scheduled **do**
4: Let t be the task with highest value of $bottomL(t)$
5: $st(t) \leftarrow earliest$ time ($>= est(t)$) at which $np(t)$ processors are available for duration $et(t, np(t))$
6: **if** $st(t) > est(t)$ **then**
7: Select a set of tasks $t' \in V$, such that $ft(t') = st(t)$ and $\sum np(t') \geq np(t)$
8: Add a *pseudo-edge* between each task in this set and t
9: **return** <Schedule length, G' >

pseudo code for the scheduling algorithm $PrBS$. $PrBS$ picks the task t with the largest bottom level and schedules it at the earliest time ($>= est(t)$) when enough processors are available for duration $et(t, np(t)$ (steps 4-5). If t is not scheduled to start as soon as it becomes ready to execute (step 6), the set of tasks that "touch" t in the schedule are computed and pseudo-edges are added between tasks in this set and t (steps 7-8). These pseudo-edges signify potential induced dependences among tasks due to resource limitations.

$PrBS$ takes (a) $O(|V| + |E|)$ steps for computing the bottom levels of tasks, (b) $O(|V|log|V|)$ to sort them in the decreasing order of their bottom levels, and (c) $O(|V|^2)$ to schedule the tasks. Thus, the complexity of $PrBS$ is $O(|E| + |V|^2)$. iCASLB requires $O(|V| + |E'|)$ steps to compute $CP(G')$ and choosing the best candidate takes constant time. Therefore, the while loop in steps 14-25 is $O(P(|E'| + |V|^2))$. The repeat-until loop in steps 9-30, has at most $|V|P$ iterations, as there are at most $|V|$ tasks in CP and each can be allocated at most P processors. Hence, the worst-case complexity of iCASLB is $O(|V|^3P^2 + |V|P^2|E'|)$. On the other hand, complexity of CPR is $O(|E||V|^2P + |V|^3P(log|V| + PlogP))$. CPA is a low cost algorithm with complexity $O(|V|P(|V| + |E|))$.

4 Performance Analysis

This section compares the quality (makespan) of the schedules generated by iCASLB with those generated by CPR, CPA, pure task-parallel (TASK) and pure data-parallel schemes (DATA). CPR is a single-step approach while CPA is a two-phase scheme. TASK allocates one processor to each task and DATA executes each task in a sequence on all processors. The algorithms are evaluated using task graphs from the Standard Task Graph Set (STG) [1], and task graphs from two applications through simulations.

4.1 Task Graphs from STG

The Standard Task Graph Set (STG) [1] is a benchmark suite for the evaluation of multi-processor scheduling algorithms that contain both random 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 [10]), and *Sparse Matrix Solver* (sparse matrix solver of an electronic circuit simulation). The robot control DAG contains 88 tasks, while the sparse matrix solver DAG contains 96 tasks. Due to limited space, they are not shown here. Parallel task speedup is calculated using Downey's model [4], which is a non-linear function of two parameters: A, the average parallelism of a task, and σ, a measure of the variations of parallelism. Based on this model, the task speedup S as a function of the number of processors n is given by:

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

For our experiments, we generated 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 [5].

Figure 5 shows the relative performance of the different schemes for these two applications 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 iCASLB 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 iCASLB. For the robot control application, iCASLB achieves upto 30% improvement over CPR and upto 47% over CPA. iCALSB also achieves upto 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, iCASLB, CPR and CPA perform similar to TASK upto 16 processors as the DAG is very wide. Beyond 16 processors the performance of the various schemes begins to differentiate. When the number of processors is increased to 128, iCASLB shows an improvement of upto 40% over CPR, 25% over CPA, and 67% and 86% over TASK and DATA, respectively. DATA performs poorly as the tasks have sublinear speedup and the sparse matrix DAG is wide.

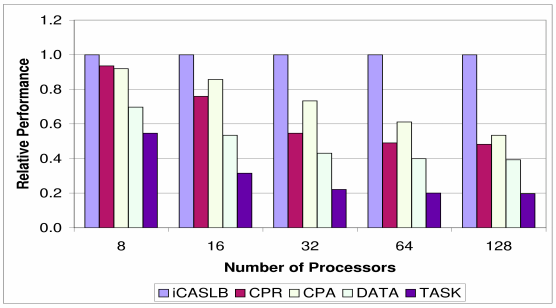


Figure 6. Relative performance for Synthetic DAGs

Figure 6 shows the average relative performance of the schemes for 20 random graphs in the Standard Task Graph Set, having 50 tasks each. Again, we see similar trends as for the application DAGs and iCASLB performs the best.

4.2 Task Graphs from Applications

The first task graph in this group comes from an application called Tensor Contraction Engine (TCE). The Tensor Contraction Engine [2] is a domain-specific compiler for expressing ab initio quantum chemistry models. The TCE takes as input, a high-level specification of a computation expressed as a set of tensor contraction expressions 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) are used to evaluate the scheduling schemes. Figure 7(a) displays the DAG for the CCSD-T1 computation, where each vertex represents a tensor contraction which is a binary operation between two input tensors to generate a result. 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 the Strassen Matrix Multiplication [7] shown in Figure 7(b). 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. The relative performance and scheduling times of the schemes for the CCSD T1 equation and Strassen Multiplication are shown in Figure 8 and Figure 9 respectively. 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 performs poorly. iCASLB shows upto 48% improvement over DATA. CPR also performs well and is only upto 8% worse than iCASLB. CPA is upto 25% worse than iCASLB. For Strassen, we find that iCASLB shows 32% and 23% improvement over CPR and 48% and 34% over CPA for 8 and 16 processors for matrix size of 1024×1024. iCASLB also achieves upto 48% and 42% improvement over TASK and DATA, respectively. When the matrix size is increased by 4 times, the performance of DATA improves as the tasks become more scalable (Figure 10).

With respect to scheduling times, CPA is a low cost algorithm and is quick in computing the allocation and schedule. iCASLB scales better than CPR as the number of processors is increased. Similar trends were observed in the case of synthetic DAGS. In all cases, the scheduling time is orders of magnitude smaller than the makespan, suggesting that scheduling is not a time critical operation for these applications.

5 Related Work

Optimal scheduling even in the context of sequential task graphs has been shown to be a hard problem to solve [13, 6]. Hence, several researchers have proposed heuristic solutions and approximation algorithms [21, 12, 9].

Ramaswamy et al. [17] introduce the Macro Dataflow Graph (MDG) which is a directed acyclic graph, to represent the structure of mixed-parallel programs. They propose a two-step allocation and scheduling scheme for MDGs called TSAS, that uses a convex programming formula-

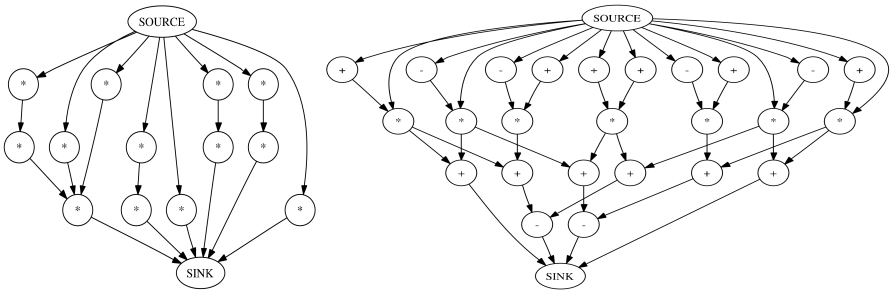
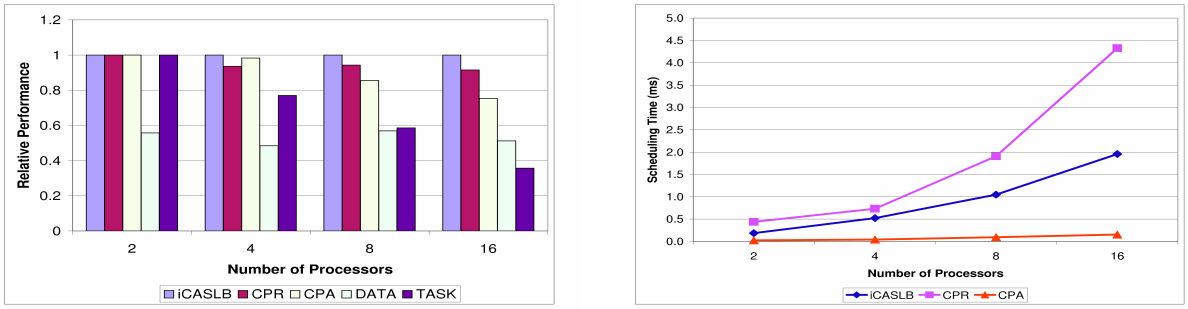


Figure 7. The CCSD task graph T1 computation (left) Strassen Matrix Multiplication (right).



(a) (b)

Figure 8. CCSD T1 computation (a) Relative performance (b) Scheduling time

tion to decide the processor allocation followed by scheduling through a prioritized list scheduling algorithm. A low cost two-step approach has also been proposed by Radulescu et al. [16], where a greedy heuristic is used to iteratively compute the processor allocation. Both these approaches attempt to minimize the maximum of average processor area and critical path length, but are limited in the quality of schedules they can produce due to the decoupling of the allocation and scheduling phases. Another work by Radulescu et al. [15] proposes a single-step heuristic, CPR (Critical Path Reduction) that starts from a one-processor allocation for each task, and iteratively increases the allocation until there is no improvement in makespan. Though iCASLB is also a one-step iterative approach, it employs effective heuristics for choosing the correct critical task for widening that will decrease the makespan if the degree of data parallelism is increased , utilizes an intelligent look-ahead mechanism to avoid local minima, and uses priority-based backfilling to increase processor utilization. Boudet et al. [3] propose an approach for scheduling task graphs which assumes the execution platform to be a set of pre-determined processor grids. In this paper, we target a generic system, where a parallel task can execute on any number of processors.

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 [20], and Prasanna's scheme [14] for scheduling of tree DAGS and series parallel graphs for specific speedup functions.

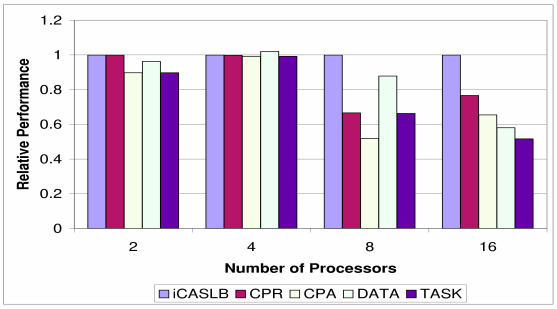
6 Conclusions and Future Work

This paper presents iCASLB, an iterative coupled processor allocation and scheduling strategy for mixed parallel applications. iCASLB makes intelligent allocation and scheduling decisions based on the global structure of the application task graph and the scalability curves of its constituent tasks. The look-ahead mechanism avoids local minima and backfill scheduling improves processor utilization. Experimental results using synthetic task graphs and those from applications show that iCASLB achieves good performance improvement over schemes like CPR, CPA, TASK and DATA.

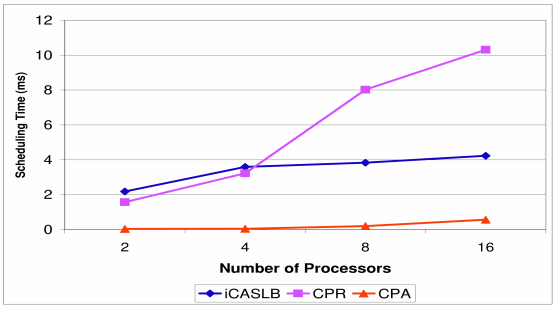
Our future work will be focused on: 1) scheduling of mixed-parallel applications with significant inter-task communication costs and 2) developing a run-time framework for on-line scheduling of these applications.

References

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



(a)



(b)

Figure 9. Strassen (1024×1024 matrix size) (a) Relative performance (b) Scheduling time

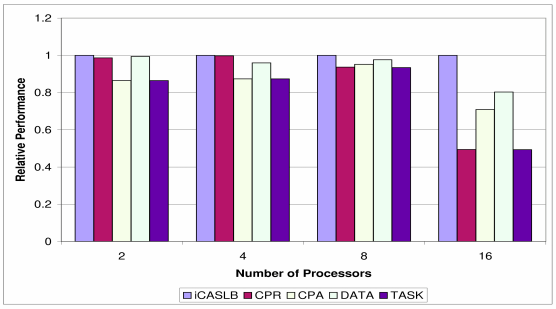


Figure 10. Relative performance for Strassen (2048×2048 matrix size)

[2] 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.

[3] V. Boudet, F. Desprez, and F. Suter. One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication. In *Proc. of the 17th Intl. Parallel and Distrib. Processing Symp.*, France, Apr. 2003.

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

[5] A. B. Downey. A parallel workload model and its implications for processor allocation. In *Proc. of the 6th Intl. Symp. on High Perf. Distrib. Comput.*, pages 112–123, 1997.

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

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

[8] 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.*, 20(6):1131–1170, 1998.

[9] K. Jansen and H. Zhang. Scheduling malleable tasks with precedence constraints. In *Proc. of the 17th ACM Symp. on Parallelism in Algorithms and Archit.*, pages 86–95, 2005.

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

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

[12] R. Lepere, D. Trystram, and G. J. Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. In *Proc. of the 9th European Symp. on Algorithms*, pages 146–157, 2001.

[13] C. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proc. of the 20th ACM Symp. on Theory of Computing*, pages 510–513, 1988.

[14] G. N. S. Prasanna and B. R. Musicus. Generalised multiprocessor scheduling using optimal control. In *Proc. of the 3rd ACM Symp. on Parallel Algorithms and Archit.*, pages 216–228, 1991.

[15] A. Radulescu, C. Nicolescu, A. J. C. van Gemund, and P. Jonker. Cpr: Mixed task and data parallel scheduling for distrib. systems. In *Proc. of the 15th Intl. Parallel & Distrib. Processing Symp.*, 2001.

[16] 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*, pages 69 –76, September 2001.

[17] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A framework for exploiting task and data parallelism on distrib. memory multicomputers. *IEEE Trans. Parallel Distrib. Syst.*, 8(11):1098–1116, 1997.

[18] T. Rauber and G. Rünger. Compiler support for task scheduling in hierarchical execution models. *J. Syst. Archit.*, 45(6-7):483–503, 1999.

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

[20] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proc. of the 8th ACM Symp. on Parallel Algorithms and Archit.*, pages 62–71, 1996.

[21] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms scheduling parallelizable tasks. In *Proc. of the 4th ACM Symp. on Parallel Algorithms and Archit.*, pages 323–332, 1992.