An Integrated Approach for Processor Allocation and Scheduling of Mixed-Parallel Applications

NAGAVIJAYALAKSHMI VYDYANATHAN, SRIRAM KRISHNAMOORTHY, GERALD SABIN, UMIT CATALYUREK, TAHSIN KURC, P. SADAYAPPAN AND JOEL SALTZ

Technical Report
OSU-CISRC-2/06-TR20

An Integrated Approach for Processor Allocation and Scheduling of Mixed-Parallel Applications

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

***Abstract*— Large 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, iė; 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. 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. Our approach employs a look ahead technique to escape local minima and uses a priority based backfill scheduling scheme 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 approach consistently performs better than CPR and CPA, two previously proposed scheduling schemes, as well as pure task and data parallelism.**

I. Introduction

Parallel scientific applications can often be decomposed into a set of 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. Task parallelism refers to the concurrent execution of independent tasks of the application on the same or different data elements. Data parallelism, on the other hand, refers to the parallel execution of a single task on data distributed over multiple processors. In a pure task-parallel approach, each task is assigned to a single processor and multiple tasks are executed concurrently such that 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, 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.

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 or 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. Priority based backfill scheduling is used to improve effective processor utilization and minimize idle time slots. We compare the proposed algorithm with two previously proposed scheduling schemes: Critical Path Reduction (CPR) [4] and Critical Path and Allocation (CPA) [5], that have been shown to give good improvement over other existing approaches like TSAS [3] and TwoL [6], as well as pure task-parallel and pure data-parallel schemes. The approach is evaluated using synthetic task graphs and task graphs based on real applications, from the Standard Task Graph Repository [7] as well as task graphs from the domains of Tensor Contraction Engine [8], [9] and Strassen Matrix Multiplication [10]. We find 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 and some definitions. Section 3 describes the allocation and scheduling al-

gorithm and Section 4 describes the benchmarks used for evaluations and the experimental results. Section 5 gives an overview of the related work and in section 6, we summarize the conclusions and outline possible directions for future research.

II. Task Graph 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 data parallel computations and E, the set of edges, represents precedence constraints. Each data-parallel computation can be executed on any number of processors. There are two distinguished vertices in the graph: the *source vertex* precedes all other vertices and the *sink vertex* succeeds all other vertices.

The weight of each vertex corresponds to its execution time on different numbers of processors. This function can be provided by the application developer, or obtained by profiling the execution of the task on different numbers of processors. Downey's model is commonly used to model the speedup of parallel programs [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 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:

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

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 terms, vertices and tasks are used interchangeably in the rest of the paper.

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, $CP(G)$, 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 $st(t)$ defines the *start time* of a task t, and $ft(t)$ define 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 $est(t) = \max_{(t',t)\in E} ft(t')$. Due to resource limitations the start time of a task t can be greater than its earliest start time, i.e., $st(t) \geq est(t)$. Note that with non-preemptive execution of tasks, $ft(t) = st(t) + et(t, np(t))$. Here, $np(t)$ 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 or makespan of G is the finish time of the sink vertex.

III. Processor Allocation and Scheduling Algorithm

In this section, we describe iCASLB (an iterative Coupled processor Allocation and Scheduling algorithm with Lookahead and Backfill), a new algorithm for processor allocation and scheduling of mixed-parallel applications. Unlike schemes that dissociate the allocation and scheduling phases [3], [5], iCASLB is a one-phase algorithm that simultaneously determines both. iCASLB is designed to reduce the makespan of a DAG by:

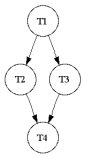
- implementing a one phase approach for allocating resources and scheduling tasks which can exploit detailed knowledge of both the application structure and the resource availability.
- utilizing priority based backfilling to increase utilization
- using look-ahead to avoid local optima
- reducing the makespan by increasing the width of task on the schedule's critical path (including induced resource dependencies)
- only increase the width of tasks which have a low degree of potential task parallelism
- increase the width of tasks which have good scalability

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 iCASLB in detail.

A. Initial Allocation and Schedule-DAG Generation

The initial allocation of processors to tasks is computed as follows. For each task we over-estimate the number of "possibly concurrent tasks" and compute the 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 achieved. If the number of available processors is more than 1, we allocate the minimum of the task's best number of processors and the number of available processors. Otherwise we simply start with allocating 1 processor to that task.

iCASLB iteratively refines this initial allocation by identifying the *best candidate* task and increasing its



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

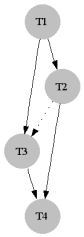
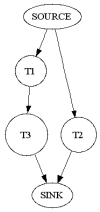


Fig. 1. (a) Task Graph G, (b) Sample allocation of processors, (c) Modified Task Graph, G'.



Tasks	Number of Processors			
	1	2	3	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

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

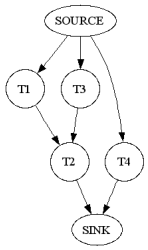
processor allocation. *Candidate* tasks lie on the critical path of the schedule. The critical path of the schedule is given by $CP(G')$, where G', the schedule-DAG, is the original DAG G with edges added because of *induced dependences* due to resource limitations. $CP(G')$ represents the longest path 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 below. 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). 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.

B. Best Candidate Task Selection

Once candidate tasks are selected, a "best" 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 as shown in the following example. 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, we would obtain a data parallel schedule, with a makespan of 51.6. On the other hand, if $T3$ were chosen, we could get a shorter makespan of 48 by allocating 4 processors to $T3$, 1 processor to $T1$ and 3 processors to $T2$.

In order to reduce the probability of the previous example, iCASLB selects the best candidate task by



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

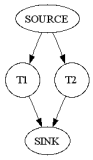
Fig. 3. (a) Task Graph G, (b) Execution time profile.

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, just picking the candidate task with the maximum decrease in execution time 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 if we were to increase the processor allocation of $T1$, it will serialize the execution of $T3$ or $T4$, resulting finally in a makespan of 17. A better choice here, is to choose $T2$ as the best candidate, and schedule it on 3 processors, leading to a makespan of 15.

To avoid this, iCASLB chooses a candidate task that not only has good execution time improvement, but also has a low *concurrency ratio*. The concurrency ratio of task t, $cr(t)$ is a measure of the minimum amount of work that can be done concurrent to t, relative to its own work, that is,

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

Here, $c_G(t)$ 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 dependent tasks. First, a DFS from task t on G is used to compute a list of tasks that depends 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, $c_G(t)$, for task t.



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

To select the best candidate task, the tasks in the critical path of G' are sorted in non-increasing order based on the amount of decrease in execution time.From the top X% of the list, the task with the minimum concurrency ratio is chosen as the best candidate. Choosing an X of 10 has yields good results for all of our experiments. Therefore, iCASLB widens tasks such that the chosen candidate task scales well and is competing for resources with relatively few other tasks.

C. Intelligent Look-ahead

Once the best candidate is selected, its processor allocation is incremented by one, a new schedule is computed using PrBS (described in the next sub-section), and the makespan of the new schedule is computed. If only schedules which decrease the makespan from the previous schedule were allowed, it would be easy to be trapped in local minima. Consider the simple DAG shown in Figure 4 and the execution profile assuming 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. 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.

To alleviate this problem, iCASLB uses an intelligent 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. 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).

D. Priority Based Backfill Scheduling (PrBS)

Priority based list scheduling is a popular and effective approach for scheduling task graphs composed of sequential tasks with dependences [12]. The tasks are prioritized and at each scheduling step the ready task with the highest priority is scheduled. List scheduling keeps track the latest free time for each processor, and forces all tasks to be executed in priority order.

The strict priority order of list scheduling tends to needlessly waste compute cycles. Parallel job schedulers use *backfilling* [13] 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 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 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 those holes.

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: $best\_Alloc \leftarrow \{np(t) | t \in V\}$ ▷ Best allocation is the initial allocation
8: $best\_sl \leftarrow PrBS(G, best\_Alloc)$
9: **repeat**
10: $\{np(t) | t \in V\} \leftarrow best\_Alloc$ ▷ 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: $A' \leftarrow \{np(t) | t \in V\}$
21: $cur\_sl \leftarrow PrBS(G, A')$
22: **if** $cur\_sl < best\_sl$ **then**
23: $best\_Alloc \leftarrow \{np(t) | t \in V\}$
24: $best\_sl \leftarrow PrBS(G, best\_Alloc)$
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 2 PrBS - Priority-Based Backfill Scheduling

1: **function** PRBS($G, \{np(t)|t \in V\}$)
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 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 *psuedo edge* between each task in this set and t
9: **return** <Schedule length, G' >

iCASLB uses 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 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 we continue the search from this state. The look-ahead, marking, unmarking, and committing steps are repeated until either all tasks in the critical path are marked or all of them are allocated the best possible number of processors. The psuedo code for the scheduling algorithm $PrBS$ is given in algorithm 2.

The complexity of the $PrBS$ algorithm can be analyzed as follows: (a) $O(|V| + |E|)$ for computing the bottom levels of the tasks, (b) $O(|V|log|V|)$ to sort the vertices in the decreasing order of their bottom levels, (c) $O(|V|^2)$ to schedule the tasks on the processors and adding pseudo-edges. Thus, the overall complexity of $PrBS$ is $O(|E| + |V|^2)$.

iCASLB requires $O(|V| + |E'|)$ time to compute the critical path (CP) in 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 overall worst-case complexity of iCASLB is $O(|V|^3P^2 + |V|P^2|E'|)$.

IV. Performance Analysis

We have compared the quality of the schedules generated by our approach with those generated by CPR, CPA and by pure task-parallel and data-parallel schemes. CPR is a single-step approach while CPA is a two-phase scheme and both have been shown in [4], [5] to perform better than other allocation and scheduling approaches like TSAS [3]. Pure task-parallel schedule (TASK) allocates one processor to each task, whereas pure data parallel schedule (DATA) executes each task on all processors one after the other.

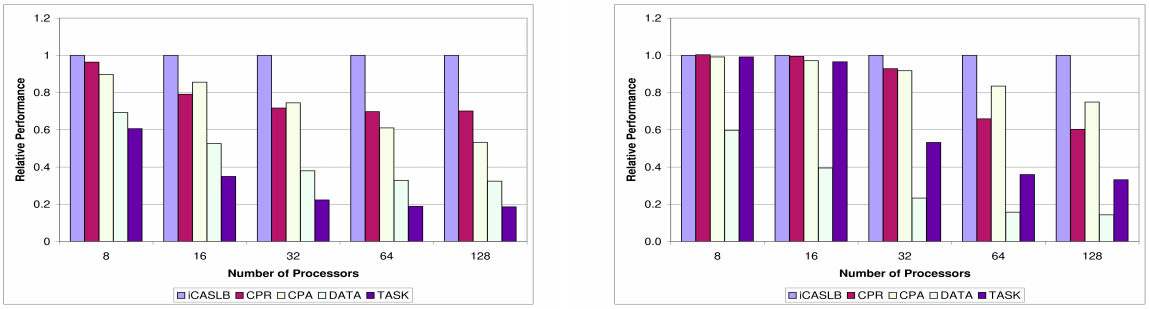
We have evaluated the various scheduling approaches using task graphs from the Standard Task Graph Set [7], and task graphs derived from two applications.

A. Task Graphs from the Standard Task Graph Set

The Standard Task Graph Set [7] is a benchmark suite for evaluation of multiprocessor scheduling algorithms. It contains both randomly generated task graphs and those modeled from actual applications. In this set, the shapes (precedence constraints) of the random graphs are determined based on four different reported methods [14], [15], [16]. In our experiments, we have used both random DAGs as well as two application DAGs - *Robot Control* which is the task graph for Newton-Euler dynamic control calculation [17], and *Sparse Matrix Solver*, which is a task graph for a random sparse matrix solver of an electronic circuit simulation. The robot control DAG contains 88 tasks, while the sparse matrix solver DAG has 96 tasks. Due to limited space, we have not included these DAGs in the paper. We derived the parallel speedups for the tasks in these DAGs using the Downey model [11] by generating σ, the variance in average parallelism as a uniform random variable in the interval [0-2.0] to represent the common scalability characteristics of most parallel jobs [18].

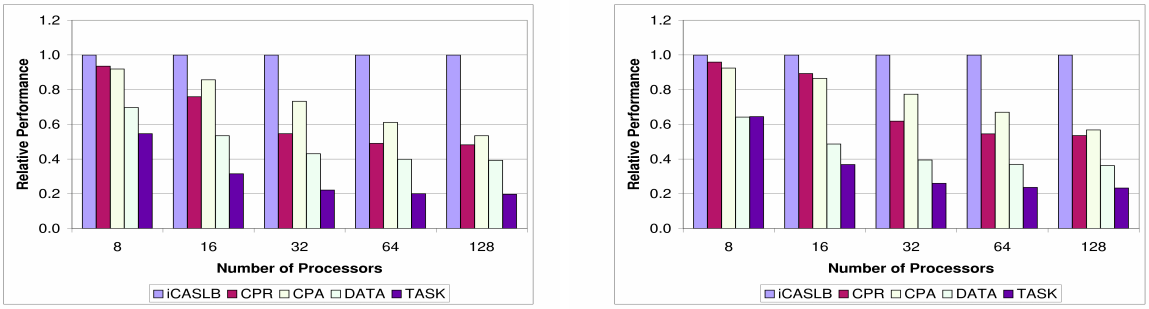
Figure 5 shows the relative performance of the different schemes for these two applications as we increase the number of processors in the system. The relative performance of an algorithm is computed as the ratio between the makespan produced by iCASLB and the makespan of the given algorithm when both are applied on the same number of processors. Therefore, a ratio less than 1 implies lower performance than iCASLB. For the robot control application, iCASLB achieves upto 30% improvement over CPR and upto 47% over CPA. We also achieve upto 81% and 68% improvement over TASK and DATA. 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, iCASLB shows an improvement upto 40% over CPR, 25% over CPA and 67% and 86% over TASK and DATA respectively, for 128 processors. DATA performs poorly as the tasks have sub-linear speedup and the sparse matrix DAG is wide.

Figure 6 shows the average relative performance of the different schemes for 20 random graphs in STG having 50 and 100 tasks respectively. Again, we see



(a) (b)

Fig. 5. Performance of the scheduling schemes for (a) Robot Control DAG (b) Sparse Matrix Solver DAG



(a) (b)

Fig. 6. Performance of the scheduling schemes for Synthetic DAGs having (a) 50 tasks (b) 100 tasks

similar trends as for the application DAGs. iCASLB performs the best and shows an improvement upto 52%, 47%, 80%, 61% over CPR, CPA, TASK and DATA respectively.

B. Task Graphs from 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 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 which are generalized matrix multiplications in a computation, form a directed acyclic graph, and are processed over multiple iterations, until convergence. We have evaluated the scheduling schemes on equations from the coupled-cluster theory with single and double excitations (CCSD). This computation is an iterative method involving the computation of T1 and T2 amplitudes, which are two and four dimensional arrays respectively. 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 [10]. The task graph for this application is shown in Figure 7(b), where the vertices represent matrix operations and the edges represent inter-task dependences. We have used matrix sizes of 1024X1024 in our 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. The relative performance of the schemes for the CCSD T1 equation is shown in Figure 8(a). 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 while CPA is upto 25% worse than iCASLB. With respect to scheduling times, CPA is a low cost algorithm and is quick in computing the processor allocation and schedule. iCASLB scales better than CPR as the number of processors is increased. In all cases, the

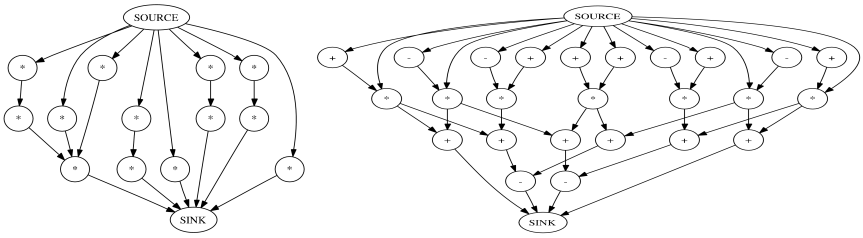
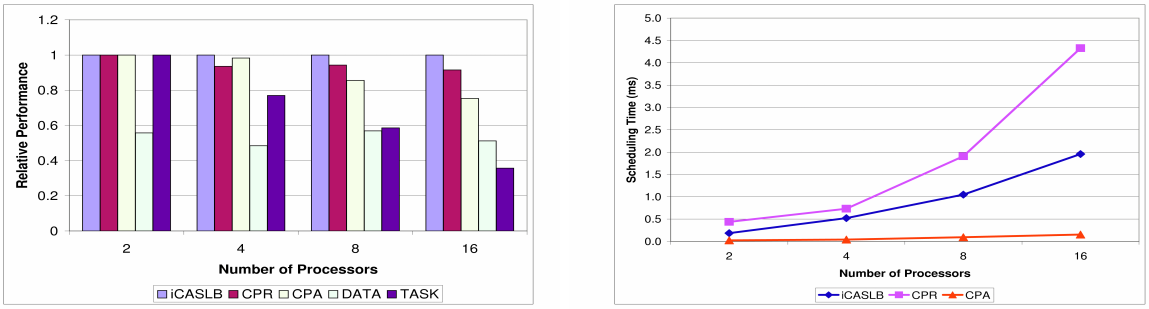
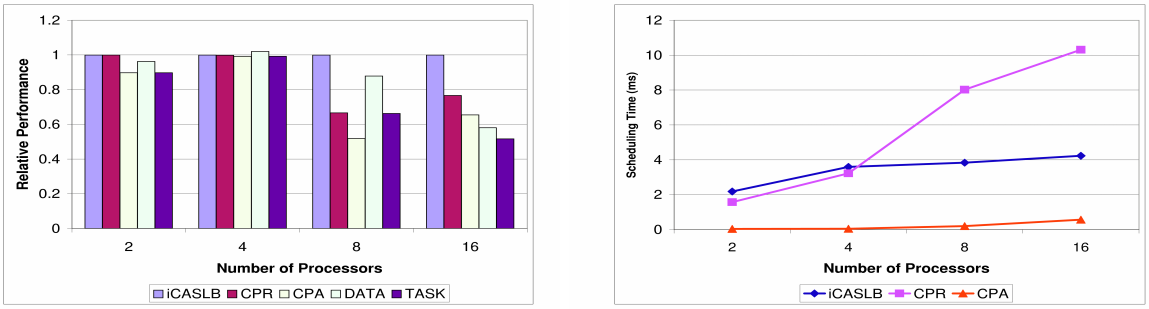


Fig. 7. The CCSD task graph T1 computation (left) Strassen Matrix Multiplication (right).



(a) (b)

Fig. 8. Performance of the scheduling schemes for CCSD T1 computation (a) Relative performance (b) Scheduling time



(a) (b)

Fig. 9. Performance of the scheduling schemes for Strassen Matrix Multiplication (a) Relative performance (b) Scheduling time

scheduling time was orders of magnitude smaller than the makespan of these applications.

The performance for the strassen multiplication is shown in Figure 9(b). We find that iCASLB shows 32% and 23% improvement over CPR and 48% and 34% over CPA for 8 and 16 processors. iCASLB also achieves upto 48% and 42% improvement over TASK and DATA respectively for 16 processors.

V. Related Work

Optimal scheduling of malleable tasks with precedence constraints have been shown to be a hard problem to solve [19], [20]. Papadimitriou and Yannakakis [19] have proved that the problem of scheduling sequential tasks with precedence constraints is NP-complete. Du and Leung [20] have shown 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 [21], [22], [23], [24]. Turek et al. [21] propose an approximation algorithm for scheduling independent parallel tasks with performance within a factor of 2 compared to the optimal, and Jansen and Porkolab [22] propose a polynomial approximation scheme based on integer linear programming. Jansen et al. [23] and Lepere et al. [24] describe approximation algorithms for scheduling malleable tasks with precedence constraints.

In [3], Ramaswamy et al. introduce the Macro Dataflow Graph (MDG) which is a directed acyclic graph, to represent the structure of mixed-parallel programs. The MDG is a directed acyclic graph with

vertices representing sequential or data-parallel computations and edges representing the precedence constraints, and two special nodes, one preceeding and one succeeding all other nodes. Ramaswamy et al. [3] propose 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 has also been proposed by Radulescu et al. [5], 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 work by Radulescu et al. [4] proposes a single-step heuristic, 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 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 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. [25] propose another single step approach for scheduling task graphs which assumes the execution platform to be a set of pre-determined processor grids. Each parallel task can only execute on any of these processor grids. In this paper, we assume a more 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 [26], and Prasanna's scheme [27] for optimal scheduling of tree DAGS and series parallel graphs for specific speedup functions.

VI. 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 scalability curves of the tasks and the global structure of the application task graph. The look-ahead mechanism avoids local minima and priority based backfill scheduling enables effective processor utilization. Experimental results using both synthetic task graphs and those derived from real applications show that iCASLB achieves significant performance improvement over other schemes like CPR, CPA, TASK and DATA.

Our future work in this area will be focused on two key aspects: 1) development of a run-time framework for the on-line scheduling of mixed parallel applications which can adapt to the run-time dynamics of both the system and the application and 2) scheduling out-of-core mixed parallel applications, where each data parallel task is characterized by both computation and I/O.

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] 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.

[5] 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*, September 2001, pp. 69 –76.

[6] 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.

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

[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.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

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

[12] 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.

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

[14] V. A. F. Almeida, I. M. M. Vasconcelos, J. N. C. ŕabe, and D. A. Menascè, "Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems," in *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 683–691.

[15] T. Yang and A. Gerasoulis, “Dsc: Scheduling parallel tasks on an unbounded number of processors,” Santa Barbara, CA, USA, Tech. Rep., 1994.

[16] T. L. Adam, K. M. Chandy, and J. R. Dickson, “A comparison of list schedules for parallel processing systems,” *Commun. ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[17] H. Kasahara and S. Narita, “Parallel processing of robot-arm control computation on a multiprocessor system,” *IEEE J. Robotics and Automation*, vol. A-1, no. 2, pp. 104–113, 1985.

[18] 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)*. Washington, DC, USA: IEEE Computer Society, 1997, p. 112.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[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] 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.

[25] 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.

[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.