Heuristic Algorithms for Scheduling Iterative Task Computations on Distributed Memory Machines

Tao Yang   Cong Fu
Department of Computer Science
University of California
Santa Barbara, CA 93106.

Abstract

Many partitioned scientific programs can be modeled as iterative execution of computational tasks, represented by iterative task graphs (ITGs). In this paper, we consider the symbolic scheduling of ITGs on distributed memory architectures with nonzero communication overhead without searching the entire iteration space. An ITG may or may not have dependence cycles and we propose heuristic algorithms for mapping cyclic and acyclic ITGs, which incorporate techniques of software pipelining, graph unfolding, directed acyclic graph (DAG) scheduling and load balancing. We provide an analysis for computing near-optimal unfolding factors and comparing the performance of the proposed heuristic algorithms with the optimal solutions. We also study the stability of run-time performance when weights are not estimated accurately at compile-time. Our experiments study the scheduling performance of solving several scientific computing problems and analyze the effectiveness of optimization techniques used in our methods. We also present experimental results for mapping SOR-based iterative computation in solving sparse and banded matrix systems, and compare our approach with the multi-coloring method.

KEY WORDS: Scheduling, communication optimization, granularity, load balancing, iterative task graphs, directed acyclic graphs.

1 Introduction

Parallel programming on distributed memory machines requires coarse grain program partitioning since there is a large startup overhead in message transmission. There are many techniques proposed for program partitioning and granularity control [CL95, GY93a, SK92, WF89] and partitioned programs can be represented in a format of coarse-grain task graphs. Finding a mapping of weighted coarse-grain task graphs onto target architectures with the shortest parallel time is important for studying the impact of partitioning and generating efficient code. In this paper we consider the scheduling of a class of partitioned programs which can be modeled as iterative task graphs (ITGs). An iterative task graph (ITG) represents a sequence of task computations where each iteration consists of the execution of a set of tasks. There exists task dependence within an iteration and across iterations.

Iterative task computations involve both task and loop parallelism [YFGS]. There is a lot of research work on scheduling task parallelism based on directed acyclic graphs (DAG), e.g. [CR92, ELA94, PLW94, PAM93, SK89, WF93, WG88, YG92]. The main optimization is to minimize parallel time by balancing load, reducing communication overhead, and overlapping communication with computation. Using DAG scheduling algorithms for ITGs is not feasible since the number of iterations may be too large or may not even be known at compile-time. Loop parallelism can be uncovered by transformation methods [P88, SK92, WF89] and various loop scheduling techniques have been proposed. Self-scheduling (e.g. [P88]) is a dynamic method for DOALL parallelism when task weights are not predictable at static time. Compared to this work, we are interested in computation in which there exist dependencies between tasks; task weights are predictable statically and do not change significantly at run-time. The solution to such a problem is also useful for run-time compilation with the inspector/executor approach [SCMB]. Mapping multiple loops are studied in [RA93] for exploring fine grain parallelism. Software pipelining [AN88, GS92, L88, RE68, VGN92] is an important technique proposed for instruction-level loop scheduling on VLIW and superscalar architectures. Loop unrolling or graph unfolding techniques [PM91, N88] have also been developed to allow a compiler to explore more parallelism. Our work has been motivated by the above research but takes into consideration the characteristics of asynchronous parallelism and the impact of communication. We will examine how task graph and loop scheduling techniques can be combined together for mapping iterative task computation on message-passing architectures. [ELA94] presented an approach which uses a DAG scheduling algorithm for one iteration of the computation and computes the optimal unfolding for a special case. We will show how to derive the near-optimal unfolding factor for general ITGs.

Load balancing and communication minimization are the important aspects of optimization. It is known that *data locality* must be explored for reducing unnecessary communication. In the task computation model [SK89, KB88], exploring data locality means to localize data communication between tasks by assigning them in the same processor and then tasks could exchange data through a local memory to avoid high-cost inter-processor communication [GY93a, SK89]. The performance of scheduling is also greatly affected by the granularity of task graphs. [GY93a] presents a quantitative analysis on the impact of granularity over the choice of optimal DAG scheduling strategies.

What distinguishes our work with that of others is the incorporation of both communication and computation costs in ITG scheduling. The optimal solution for a special class of ITGs has been studied in [Ch93]. Our heuristic algorithm is for general ITGs. The proposed algorithm contains the following parts: graph characterization, unfolding, graph transformation, DAG scheduling and the ITG schedule transformation. The optimal loop unfolding factor is derived so that the scheduling performance can be guaranteed compared to the optimal solution. We will show that the quality of DAG scheduling is critical to the performance of ITG scheduling. To make use of the scheduling result, we also present a method that executes the derived ITG schedule on distributed memory machines. We analyze the performance of run-time execution of static schedules when the estimation of task weights at static-time is not accurate. We conduct several experiments to verify our approach. We also investigate the applications of our techniques for scientific computing such as SOR [DY95, FYG95].

This paper is organized as follows. Section 2 gives the problem definition and assumptions. Section 3 discusses the possible approaches for mapping ITGs. Section 4 presents the scheduling algorithm and the optimality analysis in details for the cyclic ITGs. Section 5 gives the algorithm and the analysis for acyclic ITGs. Section 6 discusses the schedule executing scheme and examines the run-time performance of static schedules by studying the sensitivity of scheduling performance on inaccurate weights. Section 7 presents the experimental results.

2 Definitions and Backgrounds

Task graphs have been traditionally used to model task parallelism in which a set of functional tasks communicate with each other. Their dependence structure is expressed as a directed acyclic graph (DAG). A task is an indivisible unit of computation which reads its input data, it performs operations and then produces new data [SK89, WG88]. The scheduling problem is to find the mapping of tasks to processors and assign the execution starting times to those tasks. Fig. 1 shows an example of task parallelism with five tasks and their schedule on 2 processors.

$$
\begin{aligned}
&A : a = f_1(a);\\
&B : b = f_2(a, d);\\
&C : c = f_3(a, d);\\
&D : d = f_4(b, c);\\
&E : e = f_5(a, c);
\end{aligned}
$$

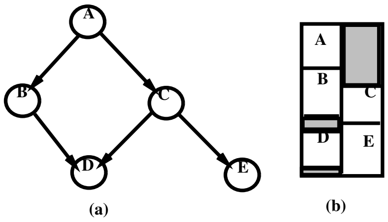

Figure 1: An example of task parallelism. (a) The task graph (DAG). (b) a schedule.

An iterative task graph (ITG) represents a sequence of task computations where each iteration consists of the execution of a set of tasks. There exists task dependence within an iteration and across iterations. Control dependence is not considered in this paper. Fig. 2(a) depicts a task graph arising from the PDE computation and it contains three iterations. Since the number of PDE iterations could be unknown or large, an ITG representation can capture the dependence between iterations, as depicted in Fig. 2(b). A formal definition of the ITG is given below.

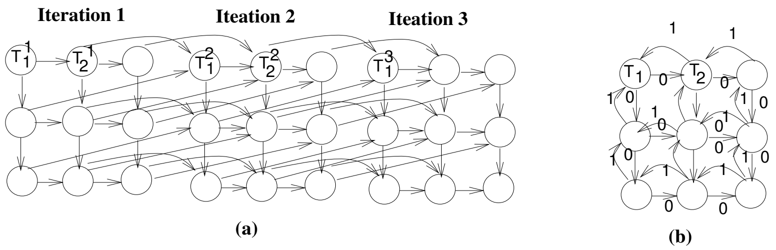

Figure 2: (a) A task graph with three iterations. (b) The ITG representation of task dependence.

Formally, an ITG G contains v tasks $T_1, T_2, \cdots, T_v$, and e data dependence edges. This graph is repeatedly executed for N iterations. Let the instance of task T_i at iteration k be T_i^k $(1 \leq k \leq N)$. We define the following terms.

- Let (T_i, T_j) be a dependence edge from T_i to T_j in G, labeled $d_{i,j}$. $d_{i,j}$ is a non-negative integer called dependence distance [1]. Thus task instance $T_j^{k+d_{i,j}}$ depends on T_i^k and it cannot start to execute until T_i^k has completed its computation and the data produced by T_i^k is available at the local memory.

- Let $Children(T_x, G)$ be a set of tasks in G that depend on task T_x. Let $Parent(T_x, G)$ be a set of tasks in G that T_x depends.

- The expanded graph $E(G, N)$ contains all $N * v$ task instances T_i^k $(1 \leq i \leq v,\ 1 \leq k \leq N)$, and these tasks and their dependencies in $E(G, N)$ constitute a DAG.

- Each task T_x in an ITG has a computation cost τ_x and there is a communication cost for sending a message from task T_x in one processor to task T_y in another processor. The cost is denoted as $c_{x,y}$, which is usually estimated as communication latency + transmission speed $\times$ size of message[Dun91].

The scheduling problem for an ITG is to assign the instances of tasks to the given p processors and determine the execution order of tasks within each processor. The parallel time is the completion time of the last task instance and the goal of scheduling is to minimize the parallel time. Fig. 3(a) shows an ITG where edges are labeled with the dependence distances between tasks. Fig. 3(b) shows another ITG. For these two examples, the computation weights of all tasks are 20 and the communication edge weights are 10 between any two processors. The expanded graph for Fig. 3(b) is in (c). A schedule for this graph when $N = 3$ is depicted in Fig. 3(d). The parallel time is 290.

[1] Our results can be extended for inexact dependence values such as '+' and '*' used in the literature [PKK93, P88, WF89]

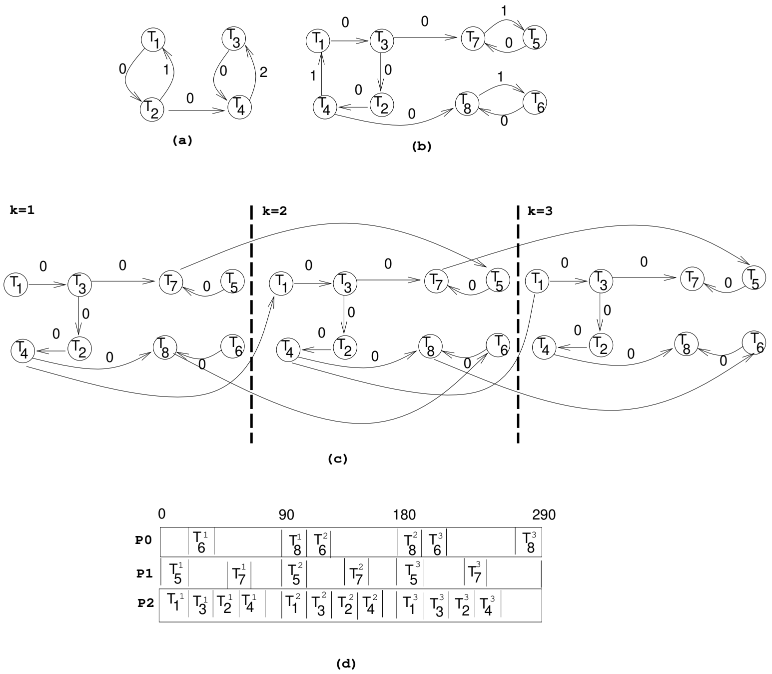

Figure 3: (a) An ITG. (b) Another ITG. (c) The expanded graph of (b) when $N = 3$. (d) The Gantt chart of a schedule when $N = 3$ for (b).

Loop unrolling or graph unfolding techniques [PM91, N88] have been found useful to explore more parallelism. Let f be the unfolding factor. When a graph is unfolded f times, the number of tasks increases by a factor of f and the number of iterations for the new ITG is $\lfloor N/f \rfloor$. For example, Fig. 3(b) is an ITG obtained by unfolding the ITG of Fig. 3(a) by a factor of 2. The number of tasks is doubled from (a) to (b). If an ITG is unfolded N times, then the resulting task graph is the expanded graph of G, $E(G, N)$.

3 Approaches for Mapping ITGs

In general, finding optimal solutions for mapping ITGs is NP-hard. The main difficulty in developing heuristic algorithms is that we need to consider performance trade-off in several aspects of optimization. First of all, since communication overhead is significant in distributed memory machines, a heuristic algorithm needs to eliminate unnecessary communication and overlap communication with computation. The communication optimization should be consistent with the goal of load balancing. We assume that communication between two tasks is zero if these two tasks are mapped to the same processor. Thus placing two tasks together in one processor to take advantage of data locality could eliminate some communication overhead although it might reduce parallelism. A trade-off between parallelization and communication locality must be addressed. Because of this reason, the problem of scheduling with communication is much harder than that without communication [Ch89, SK89] in terms of complexity. For example, optimal scheduling of a DAG without communication in an unbounded number of processors is straightforward while the problem becomes NP-complete if inter-processor communication is not zero.

Secondly, there exists task parallelism within each iteration and loop parallelism across iterations. Exploiting both task and loop parallelism together is essential to achieve a good performance.

Finally there is a trade-off between the performance and complexity of scheduling. The best solution can be found by examining the entire iteration space. However it is too expensive or not feasible to search the entire iteration space. It is possible to investigate few iterations to identify regularities of available parallelism. But examining as few task iterations as possible is important to balance the optimality of the solution and the complexity of scheduling. Since we do not search the entire iteration space, an important issue is to make sure the derived schedule is correct in the sense that all dependence constraints are satisfied.

There are several solutions for mapping ITGs:

- Direct application of DAG algorithms for an expanded graph to map task instances from all iterations can produce a good solution, but this approach may not be feasible since the iteration number may not be known in advance, or the complexity is too high.
- Applying a DAG algorithm for one iteration is possible; however, parallelism across iterations cannot be exploited in such an approach. For example, for Fig. 3(b) we can derive the DAG of each iteration by keeping all edges with distance zero, as shown in Fig. 4(a). We schedule this DAG and derive a schedule for it in Fig. 4(b). Then the schedule for the original ITG is shown in Fig. 4(c), which is longer than the schedule of Fig. 3(d). In Fig. 3(d), task instances in different iterations are executed at the same time.

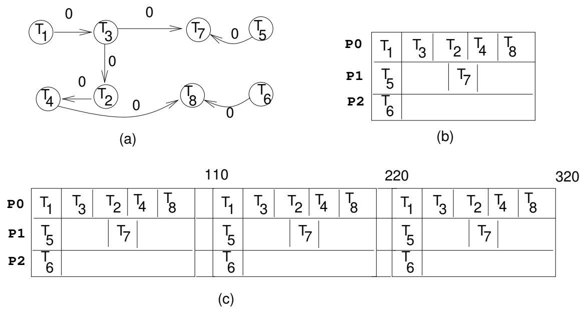

Figure 4: (a) The DAG for one iteration of ITG. (b) The DAG schedule. (c) The schedule for all iterations.

- The idea of exploring parallelism within and across iterations has been proposed in the context of instruction-level software pipelining [AN88, L88]. In [GS92], a near-optimal scheduling algorithm for pipelining with no communication delay is proposed. We need to extend this result to incorporate communication optimization with load balancing since communication is a major overhead in a message-passing machine.

- As we mentioned above, graph unfolding techniques [PM91, N88] have been also developed. Unfolding can increase the number of tasks within each iteration and thus more parallelism could be explored. Our goal is to only examine tasks from few iterations (limited by the unfolding factor f) but explore parallelism of the entire iteration space. We need to derive a small unfolding factor since the complexity of scheduling could be too high when the unfolding factor is large.

We will use the concept of integer-periodic scheduling [RE68] as in instruction-level software pipelining and structure a schedule as follows: all instances of task T_i are assigned to the same processor. If task T_i^{k+1} is executed at time t, then task T_i^{k+1} is executed after T_i^k at time $t+\beta$ where β is the iteration interval. Thus the starting time of this task instance T_i^k is modeled as $ST(T_i^k) = \alpha_{p,i} + \beta_p(k-1)$ where $\alpha_{p,i}$ is a constant to task T_i and is called the startup delay. β_p is called the length of iteration interval. Notice that for a task T_i, T_i^1 is executed at time $\alpha_{p,i}$, T_i^2 is executed at time $\alpha_{p,i} + \beta_p$ and so on. Since $1 \leq k \leq N$, the parallel time for a given ITG G is $PT(G) \approx \beta_p * N$ when N is large. The goal of scheduling is to minimize β_p.

In order to derive a schedule which is near optimal, we need to examine the performance of the optimal schedule for this problem. The optimal solution is one that obtains the minimum time for the expanded graph $E(G, N)$. We conduct a comparison of our schedule with an optimal solution. We should emphasize that our analysis measures the asymptotic performance of scheduling and study the competitive ratio of the derived parallel time over the optimal time when N is very large. We will use symbol $o(N)$ in our analysis to represent a term which is negligible when N is very large.

In order to execute a schedule on a parallel machine, we need to construct a code that follows the scheduling result. If we study the schedule in Fig. 3(d), we can see that the execution of each processor possesses certain local periodic patterns. In general, the local schedule of a processor contains three parts: prelude, iterative execution based on periodic pattern and postlude. For example, Processor 0 has a periodic pattern T_6, T_8 at interval $[20, 110)$. It will repeat at $[110, 200)$ and so on. There is no prelude and postlude. However the periodic pattern is not unique. Another pattern is T_8, T_6 at $[90, 180)$. The prelude is T_6^1 and the postlude is T_8^3. Notice that in instruction-level software pipelining [AN88], a *global* periodic pattern (a pattern which starts at the same time for all processors) is computed and this is not feasible in message-passing machines with asynchronous communication. In our approach, a local periodic pattern is produced. We will also uniformly express the prelude and postlude within a periodic pattern to avoid explicitly enumerating and sorting tasks in the prelude and postlude. The generated code is MPMD (multiple-program multiple-data) in nature, since each processor has a different local schedule. However, in practice, it is convenient to use a single SPMD program for all processors, and to let the program determine which local schedule should be used based on the processor identification.

4 The Scheduling Algorithm for Cyclic ITGs

We first present an algorithm for mapping ITGs with dependence cycles on p processors. The algorithm contains the following steps: First, we compute the unfolding factor based on an analysis in Section 4.6. Then we unfold the given ITG and transform the unfolded ITG as a DAG. Next we use the DAG scheduling algorithm to produce a mapping and construct an ITG schedule based on this mapping. The overall time complexity of this algorithm is $O(\sqrt{v}e\log^2\frac{Seq(G)}{\epsilon}+v\log v)$ where constant ϵ is the desired accuracy to be explained in the algorithm, and $Seq(G)=\sum_{i=1}^{v}\tau_i$. Notice that v is the number of nodes in the task graph and e is the number of dependence edges.

4.1 Computing the performance parameters

In this step, we determine the unfolding factor based on the performance bound analysis presented in Section 4.6.

Smallest iteration interval. We first need to estimate the maximum performance this task graph could achieve. The most important factor is

$$\beta^*(G)=\max_{\text{all cycles C in } G}\frac{\tau(C)}{d(C)}.$$

where $d(C)$ is the summation of edge distances in this cycle C and $\tau(C)$ is the summation of computation weights of tasks in this cycle. This is the well-known optimal rate (the smallest iteration interval) of the pipelining when the communication cost is zero and there are a sufficient number of processors [RE68]. To compute this value, we set up the following inequalities for all dependence edges (T_x,T_y) in G:

$$\alpha_y-\alpha_x+d_{x,y}\beta\geq\tau_x.$$

where α_y is a non-negative unknown associated with task T_y and β is an approximation of $\beta^*(G)$. These inequalities can be solved in a complexity of $O(\sqrt{v}e\log^2\frac{Seq(G)}{\epsilon})$ using the shortest path algorithm [GR89] where constant ϵ is the desired accuracy in finding the minimum value β such that $\beta^*(G)\leq\beta\leq\beta^*(G)+\epsilon$. For example, for Fig.3(a), ϵ is chosen as 0.5 and the value of β is computed as 40.

Granularity. We need to estimate the granularity of the graph, this value will be useful in computing the performance bound of the produced schedule and the unfolding factor. We use the following value to represent the granularity for a given graph:

$$g(G) = \min_{T_x \in G}\{\min_{T_y \in Parents(T_x,G)} \frac{\tau_x}{c_{y,x}}\}.$$

For example, for Fig.3(a), $g(G) = 2$.

Unfolding factor. Then we compute the unfolding factor f. The analysis in Section 4.6 shows that such a factor will guarantee that the asymptotic performance of this scheduling is competitive to an optimal solution. The performance bound compared to an optimal solution is estimated as:

$$B = S1(1 + \frac{\epsilon}{\max(\beta - \epsilon, Seq(G)/p)}) + 1.$$

where $S1 = 1 - 1/p + 1/g(G)$. The unfolding factor f is $\lceil \max(L1, L2) \rceil$ where

$$L1 = \frac{S1(\tau^{max} + \epsilon) + \tau^{max}/g(G)}{(B-1)Seq(G)/p - S1\beta}, \quad L2 = \frac{S1(\tau^{max} + \epsilon) + \tau^{max}/g(G)}{B(\beta - \epsilon) - S1\beta - Seq(G)/p},$$

and τ^{max} is the maximum task weight in G. For example, for Fig.3(a), $B = 2.68$ and $f = 2$.

4.2 Unfolding the ITG

In this step, the graph is unfolded by a factor of f. This graph is called G^f. The new graph needs to be executed only $\lfloor N/f \rfloor$ iterations. We use the unfolding algorithm proposed in [PM91] to construct G^f. The new graph G^f contains $f * v$ nodes such that each node T_i in G has f nodes in G^f: $T_{i,1}, T_{i,2}, \cdots, T_{i,f}$. Let (T_x, T_y) be an edge with distance d in G. If $d = 0$, then add edge $(T_{x,j}, T_{y,j})$ with distance 0 for $1 \le j \le f$. Otherwise if $d < f$, add edge $(T_{x,j-d}, T_{y,j})$ with distance 0 for $d + 1 \le j \le f$, and edge $(T_{x,f-d+j}, T_{y,j})$ with distance 1 for $1 \le j \le d$. If $d \ge f$, for each $1 \le j \le f$, add edge $(T_{x,h*f-d+j}, T_{y,j})$ with distance h where $h = \lceil \frac{d-j+1}{f} \rceil$.

For example, unfolding the ITG in Fig. 3(a) with $f = 2$ results in an ITG shown in Fig. 3(b). Notice the unfolded graph has the following properties [PM91]: $\beta^*(G^f) = f\beta^*(G)$.

4.3 Graph transformation

We transform G^f into a DAG so that the DAG scheduling technique could be applied. Define $DIV(x, y)$ as the largest integer that is less than or equal to x/y, i.e. $\lfloor x/y \rfloor$. Define $MOD(x, y) = x - DIV(x, y) * y$. Let the tasks in G^f be renumbered as $T_1, T_2, \cdots, T_{fv}$. We first compute the minimum integer values for β_f and α_i ($1 \le i \le fv$) based on the following inequalities for each dependence edge (T_i, T_j) in G^f: $\alpha_j - \alpha_i + \beta_f d_{i,j} \ge \tau_i$. where α_j is a non-negative unknown associated with task T_j in G^f and β_f is an approximation of $\beta^*(G^f)$. Since we know that $\beta^*(G^f) = f\beta^*(G)$, it can be shown that $f\beta - f\epsilon \leq \beta^*(G^f) \leq f\beta$. Thus the value β_f can be easily found in this range using the binary searching and the shortest path algorithm. The result satisfies $\beta^*(G) \leq \beta_f \leq \beta^*(G) + \epsilon$.

For example, edge (T_4, T_1) in Fig. 3(b) corresponds to the inequality $\alpha_1 - \alpha_4 + 1\beta_f \geq 20$. Based on inequalities of all edges, we can produce a solution $\beta_f = 80$ with $\epsilon = 0.5$, $\alpha_1 = 0$, $\alpha_2 = 40$, $\alpha_3 = 20$, $\alpha_4 = 60$. $\alpha_5 = 0$, $\alpha_6 = 20$, $\alpha_7 = 40$, $\alpha_8 = 80$.

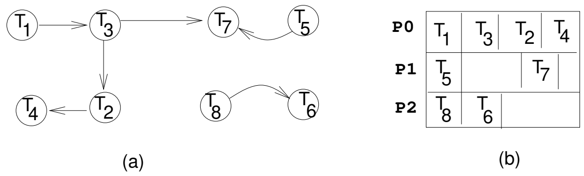

Figure 5: (a) The kernel DAG of Fig. 1(b). (b) The schedule for this DAG.

Then we transform the dependence graph G by deleting all edges (T_i, T_j) in the graph G which satisfy: $MOD(\alpha_j, \beta) - MOD(\alpha_i, \beta) < \tau_i$. For example, edge (T_4, T_1) of Fig. 3(b) is deleted because its edge weight satisfies $MOD(\alpha_4, \beta) - MOD(\alpha_1, \beta) < \tau_1$. This transformation was first proposed in [GS92] for mapping graphs when communication is zero. In Section 4.6, we will show such a transformation is still valid for our case by carefully designing the ITG schedule in Section 4.5. The dependence represented by these deleted edges will be satisfied in the final schedule constructed in Section 4.5. It can be verified that the resulting graph after edge deletions is a DAG. We call the transformed graph the kernel DAG $K(G^f)$. For example, the kernel DAG of Fig. 3(b) is in Fig. 5(a).

4.4 Mapping the kernel graph

In this stage, a DAG schedule is derived for $K(G^f)$. The starting time γ_i and the processor assignment $Proc(i)$ of each task T_i are provided. In mapping the kernel DAG, we use two algorithms: one is a multi-stage algorithm designed for the PYRROS system [YG92], another is a one-stage algorithm. We will choose the smaller one between two solutions produced by these algorithms. The result of the DAG scheduling is used for constructing the ITG schedule in Section 4.5. The first algorithm involves the following multiple stages in scheduling: 1) It clusters tasks with intensive communication using the DSC algorithm [YG94]. 2) It merges clusters to p processors using a load balancing algorithm. 3) It reorders tasks within each processor to overlap computation with communication. The algorithm has a complexity $O((v + e) \log v)$ [2] and works well in practice but we are not able to derive a performance bound. The second algorithm has the same complexity but only involves one-stage. We will derive the performance bound of the second algorithm, which is used in deriving the near-optimal unfolding factor.

[2]The size of the graph increases by f after unfolding. However in practice, we find that f is small, thus f is not included in the complexity term.

In the one-stage approach, we use the idea of the DSC algorithm [YG94] but limit the number of clusters to p. The algorithm first computes the *blevel* value for each task where $blevel(T_x)$ is the length of the longest path from this task T_x to an exit node. The algorithm updates the value of *tlevel* for each unscheduled task where $tlevel(T_x)$ is the length of the longest path from an entry task to this task T_x (the partial order includes the execution order between tasks within the same processors). The algorithm maintains a list of free tasks and at each step it picks one free task T_1 with the smallest *tlevel* value and another free task T_2 with the biggest $tlevel(T_2) + blevel(T_2)$ value. Then we will try to schedule these two tasks separately through the minimization procedure described in the following. And we will select one of these two tasks that has earlier starting time. If a tie occurs, we will pick the latter one. The *tlevel* value of the selected task will be the latest starting time γ_x of this task unless there are no processors available. The minimization procedure is also used in DSC ([YG94], page 958). The idea is to further reduce the starting time of the selected task by merging one or more parents of this examined task to one processor. In this way the unnecessary communication could be saved.

Figure 6(a) shows the selected task is T_x, (b) depicts that some parents of T_x are assigned to the same processor which leads to the minimum starting time of this task.

Let $T_y \in Parents(T_x)$, define $arrival(T_y, T_x) = \gamma_y + \tau_y + c_{y,x}$. Then

$$tlevel(T_x) = \max_{T_y \in Parents(T_x)} arrival(T_y, T_x).$$

Assume that the parents of T_x as depicted in Fig. 6 are sorted such that:

$$arrival(T_j, T_x) \geq arrival(T_{j+1}, T_x) \text{ for } 1 \leq j \leq m-1.$$

Then T_1 is called the *most dominating* parent for T_x and $tlevel(T_x) = arrival(T_x, T_1)$. $tlevel(T_x)$ is the worst starting time of T_x if there is an idle processor available at that time. However the starting time of T_x could be further reduced if T_x is assigned to the processor of T_1 and some other parents of T_x are assigned to the same processors. Assuming k parents are assigned to $Proc(T_1)$, then the starting time of T_x is

$$\gamma_x = max(avail(Proc(T_1)) + \sum_{i=2}^{k} \tau_i, arrival(T_{k+1}, T_x))$$

where $avail(i)$ is the available time of processor i. Optimal k that leads to the minimum γ_x can be found by a binary searching algorithm. When a parent of task T_x (say T_2) is moved from one processor to $Proc(T_1)$, we need to impose a constraint that such relocation should not affect the *tlevel* or starting times of children of T_2 other than T_x. Currently we use a simple but restrictive strategy: a parent T_2 is movable to $Proc(T_1)$ if T_2 does not have a child other than T_x. When T_2 is moved to the new processor location, the *avail* time of this processor may decrease.

The complexity of this algorithm is $O((v+e)\log v)$. We apply this algorithm to the given DAG and also the reversed graph, and we can show that this algorithm has the following properties: when

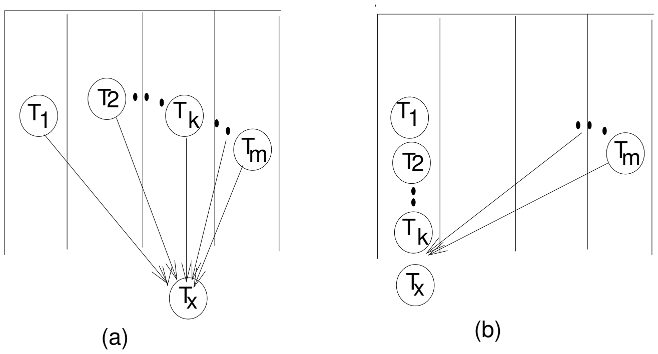

Figure 6: Minimization of the starting time for a select task T_x.

there is a sufficient number of processors, the algorithm *reaches the optimum for fork, join, coarse grained tree DAGs.* Also the performance bound is studied in the following theorem.

Theorem 1 *Given a DAG R and p processors, the parallel time produced by this algorithm is bounded by*

$$PT(R) \leq (1 - 1/p + 1/g(R))CP(R) + Seq(R)/p$$

where $CP(R)$ is the length of the critical path (the longest path, excluding edge communication weights) in this graph R.

Proof: We call a task is "relocated" if it is rescheduled from one processor to another by the minimization procedure. Otherwise this task is called "position-fixed". In the proof, we will restructure the derived schedule, move relocated tasks back to its original processor positions if there are empty slots (partial filling is possible if the original positions are filled by other tasks, but this does not affect the proof).

Let task T_x be the task examined at each step of the scheduling. Let T_y be the most dominating parent for T_x, $tlevel(T_x) = arrival(T_y, T_x)$. Then if T_x is a position-fixed task, we can see two properties hold:

- T_y is also a position-fixed task.
- If $arrival(T_y, T_x) < \gamma_x$, then there is no processor idle in time interval $[arrival(T_y, T_x), \gamma_x]$. Define $CT(T_x)$ to be the completion time of task T_x, $CT(T_x) = \gamma_x + \tau_x$. The total processor idle area in time interval $[CT(T_y), CT(T_x)]$ is at most

$$p * c_{y,x} + (p-1)\tau_x \leq p * \tau_x/g(R) + (p-1)\tau_x = (p - 1 + p/g(R))\tau_x.$$

Notice that the relocated parents of a task T_x could be moved back to its original positions in the above schedule reconstruction (for the purpose of proof only) , there could be some idle space left in $Proc(T_x)$ before time γ_x. But it is easy to verify that these idle slots are in time interval $[CT(T_y), arrival(T_y, T_x)]$.

Let task T_x be the last task in the DAG schedule. Then it must be a position-fixed task. Then we can search its most dominating parent. Repeat this reasoning, we can get a dependence chain of tasks, such that $T_1, T_2, \cdots, T_x$, such all these tasks are position-fixed. And it is also easy to verify that there is no idle time from time 0 to the starting time of task T_1 because all tasks that are scheduled before T_1 have to be entry tasks which are independent to each other. So the total idle area from time 0 to the completion of task T_x is at most

$$\sum_i \tau_i(p - 1 + p/g(R)) \leq p * CP(R)(1 - 1/p + 1/g(R)).$$

Since $p * PT(R)$ is equal to the summation of working and idle areas, thus we have

$$PT(R) \leq (1 - 1/p + 1/g(R))CP(R) + Seq(R)/p. \blacksquare$$

The ETF algorithm [H89] has a similar performance bound with a complexity $O(pv^2)$ while our algorithm has a lower complexity $O((v + e) \log v)$.

4.5 Schedule Transformation

The DAG scheduling produces processor assignment $Proc(i)$ and the starting time γ_i for each task T_i in $K(G^f)$. Next we construct a schedule for the original ITG based on the DAG scheduling result. Since the starting time of a task T_i at iteration k is modeled as $ST(T_i^k) = \alpha_{p,i} + \beta_p(k - 1)$, In our ITG schedule is $\alpha_{p,i} = \gamma_i + \beta_p DIV(\alpha_i, \beta_f)$ and $\beta_p = max(PT(K(G^f)), D)$ where

$$D = \max_{(T_x,T_y)\in G^f,\text{ but } \notin K(G^f)} \{\frac{\gamma_x + \tau_x - \gamma_y + c_{x,y}}{DIV(\alpha_y, \beta_f) - DIV(\alpha_x, \beta_f) + d_{x,y}}\}.$$

D represents the delay needed to insure edge dependence which are not considered in the DAG scheduling. The processor assignment of instances of T_i in this ITG schedule is the same as $Proc(i)$ derived by the DAG scheduling algorithm. Notice that $c_{x,y}$ is zero if tasks T_x an T_y are assigned to the same processor.

For the kernel DAG of Fig. 5(a) and $p = 3$, the DAG scheduling produces the following solution as shown in Fig. 5(b): $Proc(T_6) = Proc(T_8) = 0$. $Proc(T_5) = Proc(T_7) = 1$, $Proc(T_1) = Proc(T_2) = Proc(T_3) = Proc(T_4) = 2$, And $\gamma_1 = \gamma_5 = \gamma_8 = 0$, $\gamma_3 = \gamma_6 = 20$, $\gamma_2 = 40$, $\gamma_7 = 50$, and $\gamma_4 = 60$. $D = 90$ and $PT(K(G^f)) = 80$, thus $\beta_p = 90$. The startup delays for $T_1, T_2, \cdots, T_8$ are 0, 40, 20, 60, 0, 20, 50, 90 respectively.

4.6 Analysis for scheduling cyclic ITGs

We can show below that the schedule derived by the above algorithm satisfies the resource and dependence constraints. The proof is based on the analysis in [GS92]. The main difference is that we consider communication delay and also we add a proof on resource constraint satisfaction.

Theorem 2 *The algorithm correctly produces a schedule for an ITG.*

Proof: Let $H_i = DIV(\alpha_i, \beta_f)$. The starting time of a task instance T_i^k is $ST(T_i^k) = \gamma_i + \beta_p H_i + \beta_p(k-1)$.

(1) **Resource constraint.** We first show that no tasks in the same processor are executed in the same time. For two instances of T_i assigned at processor q, Since $\beta_p \geq \tau_i$,

$$|ST(T_i^k) - ST(T_i^{k'})| = |\beta_p(k-k')| \geq \tau_i \text{ if } k \neq k'.$$

Thus no instances of task T_i would have the execution conflict. For instances of different tasks T_i^k and $T_j^{k'}$, we have: $ST(T_j^{k'}) - ST(T_i^k) = \gamma_j - \gamma_i + \beta_p(H_j - H_i + k' - k)$. Without loss of generality, assume that $\gamma_j > \gamma_i$, then according the DAG schedule, $\gamma_j \geq \gamma_i + \tau_i$. If $H_j - H_i + k' - k \geq 0$, $ST(T_j^{k'}) - ST(T_i^k) = \gamma_j - \gamma_i + \beta_p(H_j - H_i + k' - k) \geq \tau_i$. i.e. $T_j^{k'}$ is executed after T_i^k. If $H_j - H_i + k' - k < 0$, $ST(T_j^{k'}) - ST(T_i^k) \leq \gamma_j - \beta_p \leq -\tau_j$. i.e. $T_j^{k'}$ is executed before T_i^k.

(2) **Dependence constraint**. Next we show that for any edge (T_i, T_j) in G with dependence distance d and two valid indices k, $k+d$. $ST_j^{k+d} - ST_i^k \geq \tau_i + c_{i,j}$.

If T_i has a self dependence with a distance of d, d must satisfy $d \geq 1$. Then $ST(T_i^{k+d}) - ST(T_i^k) = \beta_p * d \geq \tau_i$. For two different task instances T_j^{k+d} and T_i^k $(i \neq j)$ such that $T_j^{k+d_{i,j}}$ depends on T_i^k. $ST(T_j^{k+d}) - ST(T_i^k) = \gamma_j - \gamma_i + \beta_p(H_j - H_i + d_{i,j})$.

Notice that

$$H_j - H_i + d_{i,j} = DIV(\alpha_j, \beta_f) - DIV(\alpha_i, \beta_f) + d = \frac{\alpha_j}{\beta_f} - \frac{\alpha_i}{\beta_f} + d - \frac{\tau_i}{\beta_f} + \frac{\tau_i - MOD(\alpha_j, \beta_f) + MOD(\alpha_i, \beta_f)}{\beta_f}.$$

By the inequalities in Section4.3, $\frac{\alpha_j}{\beta_f} - \frac{\alpha_i}{\beta_f} + d \geq \frac{\tau_i}{\beta_f}$. Thus $H_j - H_i + d_{i,j} \geq \frac{\tau_i - MOD(\alpha_j, \beta_f) + MOD(\alpha_i, \beta_f)}{\beta_f}$

Let $F = \frac{\tau_i - MOD(\alpha_j, \beta_f) + MOD(\alpha_i, \beta_f)}{\beta_f}$. We discuss two cases:

- **Case 1:** If the edge $(T_i, T_j) \in K(G^f)$, then $F \geq \frac{\tau_i - MOD(\alpha_j, \beta_f)}{\beta_f} > \frac{\tau_i - \beta_f}{\beta_f} > -1$. Thus $H_j - H_i + d_{i,j} > -1$, since $H_j - H_i + d_{i,j}$ is an integer, $H_j - H_i + d_{i,j} \geq 0$. Then $ST(T_j^{k+d_{i,j}}) - ST(T_i^k) \geq \gamma_j - \gamma_i \geq \tau_i + c_{i,j}$.

- **Case 2:** If the edge $(T_i, T_j) \notin K(G^f)$, then $F > 0$, then $H_j - H_i + d_{i,j} > 0$. Since $H_j - H_i + d_{i,j}$ is an integer, $H_j - H_i + d_{i,j} \geq 1$. Since by the ITG schedule constructing method in Section 4.5, $\beta_p \geq D \geq \frac{\gamma_i + \tau_i - \gamma_j + c_{i,j}}{H_j - H_i + d_{i,j}}$. $ST(T_j^{k+d_{i,j}}) - ST(T_i^k) \geq \gamma_j - \gamma_i + (\gamma_i + \tau_i - \gamma_j + c_{i,j}) = \tau_i + c_{i,j}$. ■

We also conduct an analysis to compute the unfolding factor that gives a guarantee on the asymptotic performance bound of this heuristic algorithm. Because the result of this analysis is used in the first step of our algorithm, the expression for estimating the unfolding factor can only rely on information from the given graph G, not G^f. Thus a connection between the characteristics of G and G^f must be constructed.

Theorem 3 *Let the unfolding factor f be chosen as: $f = \lceil \max(L1, L2) \rceil$ where*

$$L1 = \frac{S1(\tau^{max} + \epsilon) + \tau^{max}/g(G)}{(B-1)Seq(G)/p - S1\beta} \text{ and } L2 = \frac{S1(\tau^{max} + \epsilon) + \tau^{max}/g(G)}{B(\beta - \epsilon) - S1\beta - Seq(G)/p}.$$

Let $\epsilon \neq \beta - Seq/p$. Then

$$\lim_{N\to\infty} \frac{PT(G)}{PT_{opt}(G)} \leq B$$

where

$$B = S1(1 + \frac{\epsilon}{max(\beta - \epsilon, Seq(G)/p)}) + 1$$

and $S1 = 1 + 1/g(G) - 1/p$. $PT(G)$ is the solution produced by our algorithm, and $PT_{opt}(G)$ is the optimal solution (by examining the entire iteration space $E(G, N)$).

Proof: We notice the following relationship between G, G^f, and $K(G^f)$:

$$Seq(G^f) = f * Seq(G^f) = f * Seq(K(G^f)), \quad g(G) = g(G^f) \leq g(K(G^f)), \quad \text{and } \beta^*(G^f) = f\beta^*(G).$$

By Theorem 1,

$$\max\{r_x + \tau_x - r_y\} \leq PT(K(G^f)) \leq CP(K(G^f))(1 - 1/p + 1/g(K(G^f))) + Seq(K(G^f))/p.$$

By the case 2 analysis in Theorem 2, if the edge $(T_x, T_y) \notin K(G^f)$, $DIV(\alpha_y, \beta_f) - DIV(\alpha_x, \beta_f) + d_{x,y} \geq 1$.

$$D = \max_{(T_x,T_y)\in G^f, \text{ but } \notin K(G^f)} \{\frac{\gamma_x + \tau_x - \gamma_y + c_{x,y}}{DIV(\alpha_y, \beta_f) - DIV(\alpha_x, \beta_f) + d_{x,y}}\} \leq \max_{(T_x,T_y)\in G^f, \text{but} \notin K(G^f)} \{\gamma_x + \tau_x - \gamma_y + c_{x,y}\}$$
$$\leq CP(K(G^f))(1 - 1/p + 1/g(K(G^f))) + f * Seq(G)/p + \tau^{max}/g(K(G^f)).$$

Assume that a computational critical path in $K(G^f)$ is $T_1, T_2, \cdots T_m$. By the transformation method in Section 4.3,

$$CP(K(G^f)) = \sum_{i=1}^{m} \tau_i \leq \tau_m + \sum_{i=1}^{m-1} (MOD(\alpha_{i+1}, \beta_f) - MOD(\alpha_i, \beta_f))$$
$$= \tau_m + MOD(\alpha_m, \beta_f) - MOD(\alpha_1, \beta_f) < \tau^{max} + \beta_f.$$

$$\begin{aligned}\beta_p = max(PT(K(G^f)), D) &\le (\beta_f + \tau^{max})(1 - 1/p + 1/g(K(G^f))) + f * Seq(G)/p + \tau^{max}/g(K(G^f)) \\ &\le S1(\beta_f + \tau^{max}) + f * Seq(G)/p + \tau^{max}/g(G) \\ &\le S1(\beta_f^* + \epsilon + \tau^{max}) + f * Seq(G)/p + \tau^{max}/g(G) \\ &\le S1(f\beta + \epsilon + \tau^{max}) + f * Seq(G)/p + \tau^{max}/g(G)\end{aligned}$$

Since $PT(G) = \beta_p * N/f + o(N)$,

$$\begin{aligned}\lim_{N->\infty} \frac{PT(G)}{PT_{opt}(G)} = \lim_{N->\infty} \frac{PT(G)/N}{PT_{opt}(G)/N} &\le \frac{\beta_p/f}{max(\beta^*, Seq(G)/p)} \le \frac{\beta_p/f}{max(\beta - \epsilon, Seq(G)/p)} \\ &\le \frac{S1(f\beta + \epsilon + \tau^{max}) + f * Seq(G)/p + \tau^{max}/g(G)}{max(f(\beta - \epsilon), f * Seq(G)/p)}\end{aligned}$$

To make the bound $\le B$, we have

$$\frac{S1(f\beta_f + \epsilon + \tau^{max}) + f * Seq(G)/p + \tau^{max}/g(G)}{max(f(\beta - \epsilon), f * Seq(G)/p)} \le B.$$

When $\beta - \epsilon > Seq(G)/p$, then we have

$$f \ge L2 = \frac{S1(\tau^{max} + \epsilon) + \tau^{max}/g}{B(\beta - \epsilon) - S1\beta - Seq(G)/p}$$

and the denominator of $L2$ must be positive. Thus the condition is:

$$B > S1\frac{\beta}{\beta - \epsilon} + \frac{Seq/p}{\beta - \epsilon}$$

Thus, choose $f = L2$ and we can set B:

$$B = S1(1 + \frac{\epsilon}{\beta - \epsilon}) + 1.$$

If $\beta - \epsilon < Seq/p$, then

$$f \ge L1 = \frac{S1(\tau^{max} + \epsilon) + \tau^{max}/g(G)}{(B - 1)Seq(G)/p - S1\beta}$$

and the denominator of $L2$ must be positive. Thus the condition is:

$$B > S1\frac{\beta}{Seq(G)/p} + 1$$

Thus choose $f = L1$ and we can set B:

$$B = S1(1 + \frac{\epsilon}{Seq(G)/p}) + 1.$$

Notice that if $\beta - \epsilon > Seq(G)/p$ then $L1 < 0$ otherwise $L2 < 0$, then we can combine the above two conditions together. We have:

$$B = S1(1 + \frac{\epsilon}{max(\beta - \epsilon, Seq(G)/p)}) + 1, \text{ and } f = \lceil \max(L1, L2) \rceil.$$

■

This theorem indicates that since ϵ could be chosen small, the bound is about $S1+1=(1-1/p+1/g(G))+1$. When the granularity of the G is not too small, this bound is tight. Notice that our granularity estimation may not be effective when some tasks perform very small computation but receive large messages from others. In practice, the size of data a task communicates is proportional to the amount of computation work it does, we expect our algorithm performs well and the bound is relatively tight. We will verify this in our experiments. On the other hand, the theorem indicates that program partitioning that produces ITGs should make $g(G)$ not too small. This is consistent to the previous results [GY93a, CR92].

5 Scheduling for acyclic ITGs

If an ITG does not have a dependence cycle, then the ITG itself is a DAG. The following algorithm is used to handle such a graph.

Step 1: We unfold the graph G in a factor $f=\lceil \max(L3,L4)\rceil$ where

$$L3=\frac{\tau^{max}}{Seq(G)*g(G)*(\frac{B}{p}-\frac{2}{p+1})},\quad L4=\frac{(1+\frac{1}{g(G)})\tau^{max}}{B*Seq(G)/p},$$

and B is the desired performance competitive bound. We set $B=\frac{2p}{p+1}+1$.

Step 2: We construct a weighted DAG which as the same structure as the given ITG G^f except the weight of an edge (T_i,T_j) is assigned as $1-d_{i,j}$ and the weight of each node is zero. This graph is called $R(G^f)$. Then we use the topological sorting algorithm to compute value H_i for each task T_i, which is the length of the longest path from this task T_i to an exit task in $R(G^f)$.

Step 3: We assign tasks in the ITG G^f to p processors such that load among those processors is balanced. Our load balancing method uses the classical list scheduling algorithm [CD73] with the highest-weight-task-first principle. The result of this mapping is the starting time γ_i for each task and the processor assignment PA. Let the length of this schedule be LPT.

Step 4: The starting time of a task T_i at iteration k in the ITG schedule is modeled as $ST(T_i^k)=\alpha_{p,i}+\beta_p(k-1)$ where startup delay $\alpha_{p,i}$ is $\alpha_{p,i}=\gamma_i+\beta_p*H_i$, the iteration interval $\beta_p=max(LPT,D)$ and

$$D=\max_{(T_x,T_y)\in G^f}\{\frac{\gamma_x+\tau_x-\gamma_y+c_{x,y}}{H_y-H_x+d_{x,y}}\}.$$

LPT represents the minimum gap between two consecutive DAG schedules in the ITG execution and D represents the delay needed to insure the dependence between tasks. The processor assignment of T_i is the same as $Proc(i)$ derived by the load balancing algorithm at Step 3.

For example, Fig. 7(a) is an acyclic ITG in which task T_1 has weight 30, T_5 has weight 10 and the rest of tasks have weight 20 and all the communication delay are 10. The H_i's are computed as $H_1 = H_2 = 0$, $H_3 = H_4 = 1$, and $H_5 = 2$. The calculated unfolding factor is 1 for this case. Fig. 7(b) shows the processor mapping obtained by the load balancing method described in Step 3 with $LPT = 40$. Fig. 7(c) shows the Gantt chart of part of the schedule with $\beta_p = 40$, and the asymptotic speedup of this schedule is 2.5 .

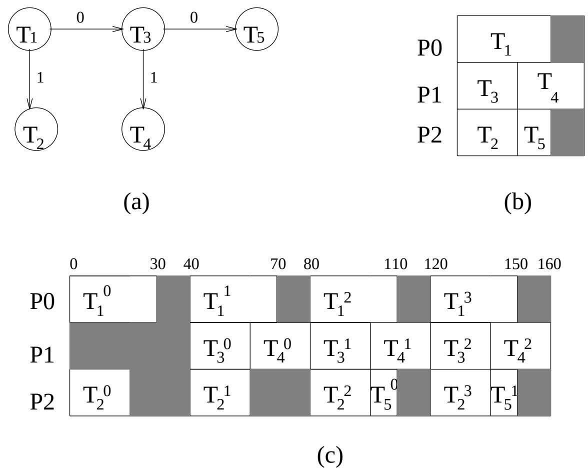

Figure 7: (a) An acyclic ITG. (b) The processor assignment derived in Step 3 using the load balancing algorithm. (c) A portion of the final schedule for this ITG.

This algorithm has a time complexity of $O(v \log p + e)$ and has the following properties.

Theorem 4 *The above 4-step algorithm produces a correct schedule for an ITG with no dependence cycle.*

Proof: (1) The proof for the resource constraint satisfaction is the same as the one for Theorem 2.

(2) Now we show that all dependence is satisfied. For two different task instances T_j^{k+d} and T_i^k $(i \neq j)$ such that $T_j^{k+d_{i,j}}$ depends on T_i^k. Since H_j is the length of the longest path from task T_j to an exit task in $R(G^f)$. For each dependence edge (T_i, T_j), $H_j - H_i \geq 1 - d_{i,j}$. Then $H_j - H_i + d_{i,j} \geq 1$.

Since $\beta_p \geq D \geq \frac{\gamma_i + \tau_i - \gamma_j + c_{i,j}}{H_j - H_i + d_{i,j}}$, $\beta_p * (H_j - H_i + d_{i,j}) \geq \gamma_i + \tau_i - \gamma_j + c_{i,j}$. Thus

$$ST(T_j^{k+d}) - ST(T_i^k) = \gamma_j - \gamma_i + \beta_p(H_j - H_i + d_{i,j}) \geq \gamma_j - \gamma_i + (\gamma_i + \tau_i - \gamma_j + c_{i,j}) = \tau_i + c_{i,j}. \blacksquare$$

Theorem 5 *Let $PT(G)$ be the solution produced by our algorithm for an acyclic ITG G and $PT_{opt}(G)$ be the optimal solution by examining the entire iteration space $E(G, N)$. Then*

$$\lim_{N \to \infty} \frac{PT(G)}{PT_{opt}(G)} \leq B.$$

where B has to be set as $B > \frac{2p}{p+1}$.

Proof: The load balancing algorithm used at Step 3 can achieve the following performance: $LPT \leq \max\{\frac{2*Seq(G^f)}{p+1}, \tau^{max}\}$. We explain this briefly as follows. Let $[s,t]$ be the longest time interval in the load balanced schedule produced at Step 3 such that there exists some idle processor but also some busy processor at any time of this interval. Then $t = LPT$ and there is no processor idle before time s. Also there is no task T_x with assignment r_x such that $r_x > s$ otherwise this task would have been scheduled at time s. Thus there exists one task T_y executed in that interval such that $\tau_y \geq t - s$. Then we have

$$LPT * p \leq Seq(G^f) + (p-1) * \tau_y.$$

Case 1: If there is a task T_z completing before task T_y starts. Then $\tau_z \geq \tau_y$ and $LPT \geq \tau_z + \tau_y \geq 2\tau_y$. Thus

$$LPT * p \leq Seq(G^f) + (p-1) * \frac{LPT}{2}.$$

Then $LPT \leq \frac{2*Seq(G^f)}{p+1}$.

Case 2: If there is no task competing before T_y, then T_y starts at time 0, thus $LPT = \tau_y = \tau^{max}$. In summary, $LPT \leq max(\frac{2*Seq(G^f)}{p+1}, \tau^{max})$. Thus for the ITG schedule:

$$\max\{r_x + \tau_x - r_y\} \leq LPT \leq \max\{\frac{2*f*Seq(G)}{p+1}, \tau^{max}\}.$$

Since $H_j - H_i + d_{x,y} \geq 1$,

$$D = \max_{(T_x,T_y)\in G^f}\{\frac{\gamma_x + \tau_x - \gamma_y + c_{x,y}}{H_j - H_i + d_{x,y}}\} \leq \max_{(T_x,T_y)\in G^f}\{\gamma_x + \tau_x - \gamma_y + c_{x,y}\}$$
$$\leq \max\{\frac{2*f*Seq(G)}{p+1}, \tau^{max}\} + \frac{\tau^{max}}{g(G^f)}.$$

$$\beta_p = max(LPT, D) \leq \max\{\frac{2*f*Seq(G)}{p+1} + \frac{\tau^{max}}{g(G)}, (1 + \frac{1}{g(G)})\tau^{max}\}.$$

Since $PT(G) = \beta_p * N/f + o(N)$,

$$\lim_{N->\infty} \frac{PT(G)}{PT_{opt}(G)} = \lim_{N->\infty} \frac{PT(G)/N}{PT_{opt}(G)/N} \leq \frac{\beta_p/f}{Seq(G)/p}$$
$$\leq \frac{\max\{2*f*Seq(G)/(p+1) + \tau^{max}/g(G), (1+1/g(G))\tau^{max}\}}{f*Seq(G)/p}.$$

To make the bound $\leq B$. We have

$$\frac{\max\{2*f*Seq(G)/(p+1) + \tau^{max}/g(G), (1+1/g(G))\tau^{max}\}}{f*Seq(G)/p} \leq B$$

i.e.

$$f * Seq(G) * g(G)(\frac{B}{p} - \frac{2}{p+1}) \geq \frac{\tau^{max}}{g(G)}, \text{ and, } \frac{B * f * Seq(G)}{p} \geq (1 + \frac{1}{g(G)})\tau^{max}.$$

When $B > \frac{2p}{p+1}$, we have

$$f \geq L3 = \frac{\tau^{max}}{Seq(G) * g(G) * (\frac{B}{p} - \frac{2}{p+1})}, \quad f \geq L4 = \frac{(1 + \frac{1}{g(G)})\tau^{max}}{B * Seq(G)/p}.$$

Thus we choose $f = \lceil \max(L3, L4) \rceil$.

■

We can see that when for coarse grain graphs with $g(G) \geq 1$, $L3$ and $L4$ are very small if $\tau^{max} * p/Seq(G)$ is not large. Thus without unfolding, the scheduling algorithm can produce a reasonable performance. Another note is that if a given ITG is cyclic, and it only has self dependence, then this above algorithm without unfolding still works since the self-dependence in G^f is actually satisfied automatically. Thus for a coarse-grain ITG which only has self-dependence cycles, we can still apply the above algorithm without unfolding and the performance should be good according to the above analysis. In Section 7, we will examine such an ITG arising from matrix-matrix multiplication.

6 Run-time Execution and Sensitivity Analysis

The scheduling algorithm can determine the processor assignment of tasks and the starting time of each task. In order to utilize this result, we need to study the scheme that executes this schedule. We will first discuss the run-time method for executing parallel tasks following the static schedule and then discuss the run-time performance variation due to inaccurate estimation of task and communication weights.

6.1 The schedule execution

We assume that the target is a message-passing parallel machine, each processor has its own private memory and a communication buffer to hold the outgoing and incoming messages. In Section 4, we have derived the starting function value for each task instance. If we execute tasks by sorting the starting time of these tasks at run-time, then the cost of sorting contributes a significant amount of overhead in the overall performance. Our approach is based on the idea of "periodic execution" in software pipelining. The difference is that in instruction-level pipelining, periodic pattern is global usually [AN88] while in a distributed memory machine, we use local period patterns to utilize asynchronous parallelism. Our code does not have separate prelude and postlude parts, and in this way the code length can be shortened.

The starting time of task instances depends on the iteration number k. We use the following method to find the periodic pattern for processor j from the coefficients in starting time function $ST(T_x) = \alpha_{i,p} + \beta_p(k-1)$. Let $Tasks(j)$ be all tasks assigned to processor j. Let $L_j = \max_{T_i \in Tasks(j)} \alpha_{p,i}$. We can show that $[L_j, L_j + \beta_p)$ contains the periodic pattern of Processor j. Let $o_i = \lceil \frac{L_j - \alpha_{p,i}}{\beta_p} \rceil + 1$. We can show that o_i is the iteration sequence number of task T_i at $[L_j, L_j + \beta_p)$. The tasks in this interval are executed in the increasing order of value $\alpha_{p,i} + \beta_p(o_i - 1)$.

For the example of Fig. 3, processor 0 has two tasks T_6 and T_8. $max(\alpha_{p,6}, \alpha_{p,8}) = max(20, 90) = 90$. A local periodic pattern starts at $[90, 180)$. The iteration sequence numbers for T_6 and T_8 are 2 and 1. Thus the ordering for this processor is T_8, T_6. We show the correctness of this method as follows.

Theorem 6 $[L_j, L_j + \beta_p)$ *is the initial interval of a local periodic pattern for processor* j *where* $L_j = \max_{T_i \in Tasks(j)} \ \alpha_{i,p}$. *Within interval* $[L_j, Lj + \beta_p)$, *each processor* j *executes one and only one instance of a task* T_i *in* $Tasks(j)$ *and the iteration number is* $o_i = \lceil \frac{L_j - \alpha_{i,p}}{\beta_p} \rceil + 1$.

Proof: Notice that $L_j \geq \alpha_{i,p}$, thus $o_i = \lceil \frac{L_j - \alpha_{i,p}}{\beta_p} \rceil + 1 \geq 1$. We first show that there is one instance of any task T_i with iteration number o_i executed in interval $[L_j, L_j + \beta_p)$. Since

$$\frac{L_j - \alpha_{i,p}}{\beta_p} \leq \lceil \frac{L_j - \alpha_{i,p}}{\beta_p} \rceil < \frac{L_j - \alpha_{i,p}}{\beta_p} + 1$$

Thus

$$L_j \leq \alpha_{i,p} + \beta_p(\lceil \frac{L_j - \alpha_{i,p}}{\beta_p} \rceil + 1 - 1) < L_j + \beta_p.$$

i.e. $T_i^{o_i}$ is executed within this interval $[L_j, L_j + \beta_p)$. Since $|ST_i^k - ST_i^{k'}| \geq \beta_p$ for $k \neq k'$, only one instance of T_i is executed in this interval. The execution order of tasks in $[L_j, L_j + \beta_p)$ is the same as the one in $[L_j + k\beta, L_j + (k+1)\beta_p)$ because of the constant offset of the execution starting time for each task between these two intervals. ∎

Based on the above theorem, we give the code structure for executing the schedule at processor j in Fig. 8. When executing each individual task, the task receives the data items it needs, performs computation, and then sends out data produced to its children (using aggregate multicasting if possible). The style is similar to the one used in the PYRROS system [YG92].

Next we examine the performance of this code structure and provide an analysis on performance variation between predicted time and actual run-time performance.

```
Let o^max = max_{T_x ∈ Tasks(j)} o_x. o^min = min_{T_x ∈ Tasks(j)} o_x.
For r = -o^max to N - o^min
    Execute tasks T_i^k in an order of α_{i,p} + β_p(o_i - 1) where k = r + o_i;
If k > N or k < 1, skip this task.
endfor
```

Figure 8: A generic form of code for Processor P_j.

6.2 An analysis of run-time performance

Code in Fig.8 executes the given ITG G N times using the derived periodic pattern. When it executes a task, it starts task computation as soon as this task is ready. Thus if run-time weights are the same as the static time weights, it is possible that the code may execute tasks in an iteration interval smaller than computed β_p. We call the periodic schedule used in Fig.8 to be S, and define $PT(G)$ to be the parallel time estimated by our algorithm using parameter β_p. Let $PT(S, G)$ be the parallel time produced by the code in Fig.8. Then $PT(S, G) \leq PT(G)$.

At run-time, the computation time of each task and communication delay between tasks may not be the same as the one predicted at compile-time. The following parameters affect the performance of a static schedule in a run-time environment:

- Variation of task computation weight, due to inaccurate estimation of static weight, the cost of fetching data from the communication buffer and copying the data from the local memory to the communication buffer.
- Variation of communication delay, due to inaccurate estimation of static communication weight, unexpected run-time network contention.

We address the sensitivity of the scheduling performance on the variation of task communication and computation weights. Let G be the graph for which the scheduling algorithm produces a schedule S. At the run-time, the weights of this graph change, we call the new graph as G^r. The run-time method in Fig. 8 uses the mapping and task ordering of S to execute graph G^r, resulting in a schedule S^r in a length of $PT(S, G^r)$. Assume that $PT_{opt}(G^r)$ is the optimal schedule length for G^r. The interesting question is: what is the competitive ratio of $PT(S, G^r)$ over $PT_{opt}(G^r)$.

We assume that at run-time, each communication weight $c_{i,j}$ is changed to $c^r_{i,j}$ and the computation weight τ_i is changed to τ^r_i. They satisfy:

$$(1 - \delta_1)c_{i,j} \leq c^r_{i,j} \leq (1 + \delta_2)c_{i,j}, \quad (1 - \delta_1)\tau_i \leq \tau^r_i \leq (1 + \delta_2)\tau_i.$$

We have shown in Theorem 3 and 5 that the derived schedule satisfies: $PT(G) \leq B \times PT_{opt}(G)$ where $PT_{opt}(G)$ is the optimal schedule length for G, $B = (1 - 1/p + 1/g(G))(1 + \frac{\epsilon}{max(\beta - \epsilon, Seq(G)/p)}) + 1$ for the cyclic case and $B = \frac{2p}{p+1} + 1$ for the acyclic case. The following theorem identifies the relationship between $PT(S, G^r)$ and $PT_{opt}(G^r)$.

Theorem 7

$$\lim_{N \to \infty} \frac{PT(S, G^r)}{PT_{opt}(G^r)} \leq \frac{1 + \delta_2}{1 - \delta_1} \times B$$

Proof: We first estimate $PT(S, G)$. The length of the critical path in $E(G, N)$ is usually smaller than $PT(S, G)$ because in this schedule independent tasks could be mapped to the same processor, which may not be reflected in a critical path of $E(G, N)$. However, after adding an execution edge from T_x^i to T_y^j if T_x^i is executed immediately before T_y^j in the same processor at schedule S and these two tasks are independent, the length of the critical path in the new graph is the execution time, i.e. $PT(S, G)$. We call this new graph $EE(S, G, N)$. At run-time, the weights of $EE(S, G, N)$ may change. We call the run-time graph $EE(S, G^r, N)$. Note that $EE(S, G^r, N)$ has the same dependence structure as $EE(S, G, N)$.

In the rest of the proof, we use $CP(S, G)$ and $CP(S, G^r)$ to denote a critical path in $EE(S, G, N)$ and $EE(S, G^r, N)$ respectively based on schedule S. We also define $W(P)$ as the summation of task weights and communication weights between adjacent tasks along the path P (the communication weight will be zero if two tasks are assigned to the same processor in the schedule considered).

First it is easy to see that for any path P in $EE(S, G, N)$, we have $PT(S, G) \geq W(P)$. But at run-time, $PT(S, G^r) = W(CP(S, G^r)) \leq (1 + \delta_2)W(P_x)$, where P_x is the path in $EE(S, G, N)$ contains the same tasks and edges as the critical path $CP(S, G^r)$ of $EE(S, G^r, N)$. Note that P_x may not be a critical path of $EE(S, G, N)$.

Since $PT(S, G) \geq W(P_x)$, $\frac{PT(S, G^r)}{PT(S, G)} \leq 1 + \delta_2$. Since $PT(S, G) \leq PT(G)$ and $\lim_{N->\infty} \frac{PT(G)}{PT_{opt}(G)} \leq B$, we have

$$\lim_{N->\infty} \frac{PT(S, G^r)}{PT_{opt}(G)} \leq (1 + \delta_2)B. \tag{1}$$

On the other hand if we consider the optimal schedule for G^r and we call it O^r, then we have $PT_{opt}(G) \leq PT(O^r, G) = W(CP(O, G))$, where $PT(O^r, G)$ is the parallel time obtained when applying the optimal schedule O^r to G. Now let P_y denote the path in $EE(O^r, G^r, N)$ containing the same tasks and edges as $CP(G, O^r)$. Then $PT_{opt}(G^r) = PT(O^r, G^r) = W(CP(O^r, G^r)) \geq W(P_y) \geq (1 - \delta_1)W(CP(O^r, G))$. Thus, we have

$$\frac{PT_{opt}(G^r)}{PT_{opt}(G)} \geq 1 - \delta_1. \tag{2}$$

Combining Equations 1 and 2, we conclude that:

$$\lim_{N->\infty} \frac{PT(S, G^r)}{PT_{opt}(G^r)} \leq \frac{1+\delta_2}{1-\delta_1} \times B.$$

■

This analysis shows that if the run-time variation of weights is not significant, a static schedule still produces a competitive performance. We will verify this in our experiments.

7 Experiments

We have implemented our algorithms in a scheduling system which takes an ITG as input and produces a schedule for each processor. Currently the system is able to simulate the scheduling performance on message-passing machines using their communication and computation parameters. In our experiments, we have examined the performance of scheduling 50 randomly generated ITGs, and several application programs including matrix multiplication, solving a Laplace equation, solving a banded matrix system, and the SOR method for sparse matrix systems. We have also hand-coded some of schedules using our executing scheme in nCUBE-2 distributed memory machines to verify the correctness of the simulation in a real machine. In this section we report the experimental results which address several aspects of the performance of this algorithm.

Overall performance of scheduling. We report the performance of the simulation and also the actual performance on nCUBE-2. In the simulation, we compute the asymptotic speedup of an ITG schedule as: $\lim_{N\to\infty} \frac{N*Seq(G)}{\max_{T_i\in G} ST(T_i^N)} = Seq(G)/\beta_p$. In the nCUBE2 implementation, we use the sequential time divided by the actual parallel time.

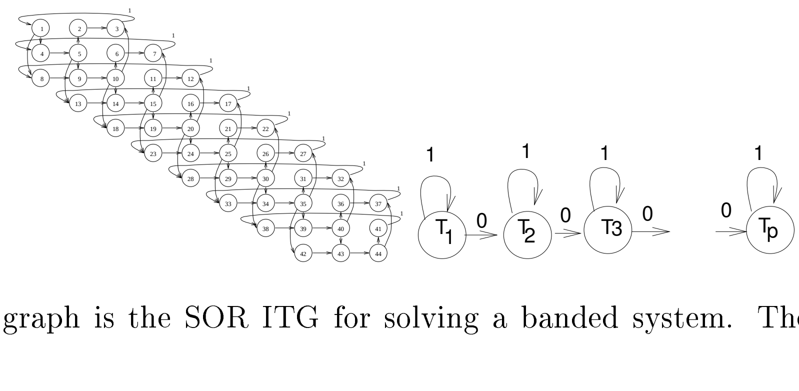

Figure 9: The left graph is the SOR ITG for solving a banded system. The right is for matrix multiplication.

In Fig. 10, we list the results of scheduling three application programs. We first show the simulated and actual performance on nCUBE-2 for the matrix multiplication[3] when the matrix dimension is

[3]The matrix multiplication could be fully parallelized if each processor has a memory to store a matrix. However, we assume there is not enough memory to hold a matrix, the communication is needed between tasks to transfer matrix elements.

$n = 200$ and $n = 400$. We also show the performance of solving Laplace PDEs (200x200 and 400x400 grids), and banded matrices (n=2000 and 4000 with band width=35) using the SOR method. We can see that the simulated performance is close to the actual performance for the studied programs. There is still some discrepancy especially when p is getting large. It indicates the communication overhead prediction still needs some future investigation. The ITGs for those test cases are depicted in Fig. 2(b) and Fig. 9.

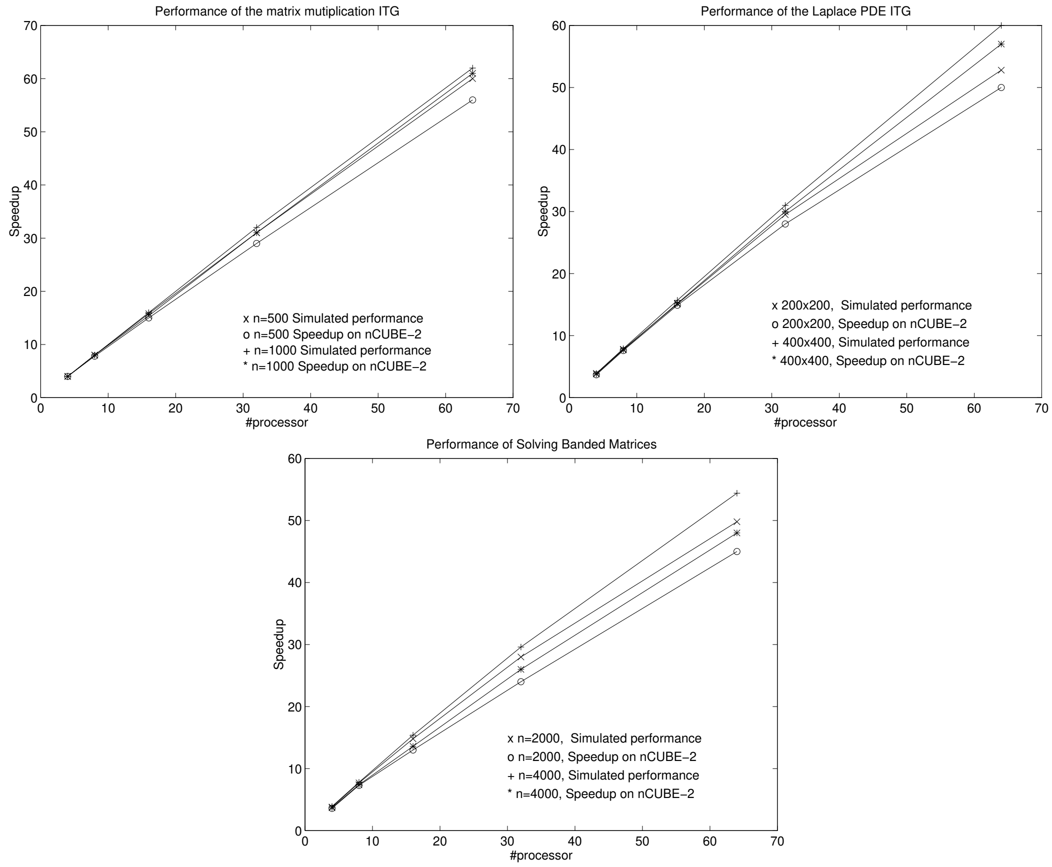

Figure 10: Speedups of three ITGs: matrix multiplication, Laplace equations, and banded matrices.

Performance sensitivity on inaccurate weights. We change the weights of PDE and banded matrix ITGs in a range of $[-10\%, +10\%]$. We compare the variation ratio $\frac{Speedup_{new}}{Speedup_{old}} - 1$. The result is in Fig. 11. We can see the performance is sensitive but variation is small, within 5%.

The effectiveness of unfolding and the performance bound. We examine if the computed factor identifies the range of unfolding in improving performance for three groups of randomly generated acyclic and cyclic ITGs: coarse grained graphs with $g(G) \geq 1$; fine grained graphs where $0.4 < g(G) < 1$ and the local granularity (local computation weights vs. communication weights) is also in this range; graphs with mixed local grain values in a range [0.1,1]. The number of tasks in each ITG varies from 30 to 60. Each group has 10 ITGs. The results for $p = 32$ are presented in Fig. 12(the left is for cyclic ITGs and the right is for acyclic ITGs). The X axis is the unfolding

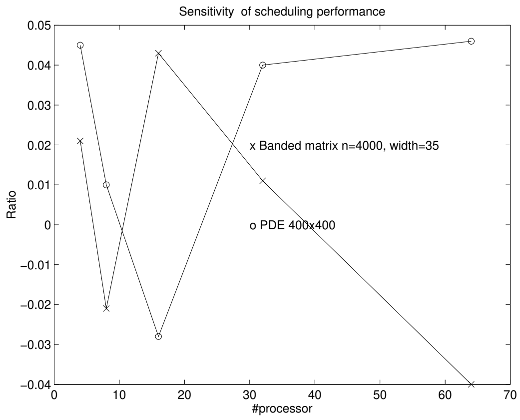

Figure 11: Performance variation ratio of the PDE ITG and banded matrix ITG using the inaccurate and accurate weights.

percentage which is calculated with respect to the unfolding factor obtained by our scheduling algorithm. Assume that f is the calculated factor. 50% unfolding percentage means that $0.5f$ is used. The Y axis is the speedup improvement ratio after unfolding: $\frac{Speedup_{new}}{Speedup_{old}} - 1$. For the acyclic coarse-grain ITGs, the calculated unfolding factors are all 1 and thus the unfolding performance is not listed. We examine the unfolding lower bounds $L3$ and $L4$ derived in Theorem 5 and we can see that when $g(G) \geq 1$, $L3$ and $L4$ are very small if $\tau^{max} * p/Seq(G)$ is not large. In general, we can see that unfolding greatly improves the performance. This is because unfolding produces more tasks for the DAG scheduling, which gives the algorithm more flexibility.

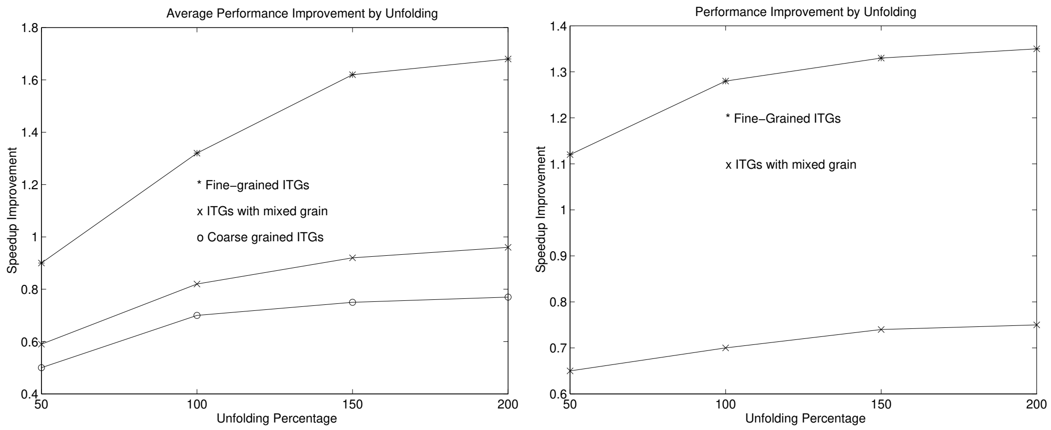

Figure 12: Performance improvement by unfolding. The left is for cyclic ITGs. The right is for acyclic ITGs.

The average estimated upper bound after unfolding and actual bound are listed in Table 1. The average actual upper bound is approximately computed as $\frac{p}{\text{the obtained speedup}}$ where $p = 32$ for this experiment. The bounds for acyclic are small. For cyclic cases, if the granularity value is large, our bound formula is very tight. When granularity value is small (say 0.1), the estimated bound is loose, but the actual upper bound is small. The result indicates that our unfolding factor effectively guides the algorithms to obtain a near optimal schedule.

Average bound	Coarse grained	Fine grained	Mixed
Estimated, cyclic	2.8	3.9	12.2
Actual, cyclic	1.51	1.8	1.45
Estimated, acyclic	2.94	2.94	2.94
Actual, acyclic	1.63	1.15	1.11

Table 1: Estimated bounds and actual bounds for cyclic and acyclic ITGs.

Since the value of unfolding factor affects the complexity and also the size of code, we study how large the unfolding factor is for these random ITGs. The unfolding factors computed are small (4-8) in general except for three cases (30-50). We studied those cases and found that B bound is set to be too small. If we increase B by 1, then the average unfolding factor is reduced to 6.

With and without software pipelining. We have demonstrated in Fig. 4 that the parallelism is not fully utilized if using the DAG scheduling for one iteration without pipelining iterations (called simple DAG scheduling approach). It should be noted that after unfolding an ITG, the simple DAG scheduling approach could also explore parallelism between iterations *for the original ITG* but *not for the new ITG*. We examine the 100 random ITGs and compare the performance of this simple DAG scheduling approach with our algorithm. Fig 13 shows the average performance improvements of our approach over this simple DAG scheduling approach before graph unfolding and after unfolding by a factor of 2. It can be seen that before unfolding, our approach can achieve as high as 50% performance improvement due to the exploration of the across-iteration parallelism especially when the number of processors is large. After unfolding, both approaches have gained more flexibility. But our approach is still a lot better.

The role of DAG scheduling and load balancing. We investigate how critical the DAG scheduling in our algorithm for constructing the final ITG schedule. Notice that $\beta_p = max(D, PT(K(G^f))$. We compute the ratio $PT(K(G^f))/\beta_p$ for these randomly generated graphs. The larger this value is, the more crucial the DAG scheduling is. The left part of Fig. 14 shows the average ratios for three groups of cyclic ITGs. It shows that when p is small, each processor has a heavy load and the DAG schedule dominates the ITG performance. When p is large ($p = 32$), the load per processor becomes smaller (notice that the graph sizes are not too big), the role of communication comes to play. But still the DAG scheduling contributes more than 60% to the final performance. Similar results for acyclic ITG scheduling can be seen in the right part of Fig. 14 on the role of the load balancing algorithm at Step 3.

Iterative sparse matrix computation. In solving a sparse matrix system, the SOR iterative methods could be used [AJ86]. We use submatrix partitioning and a partitioned sparse matrix

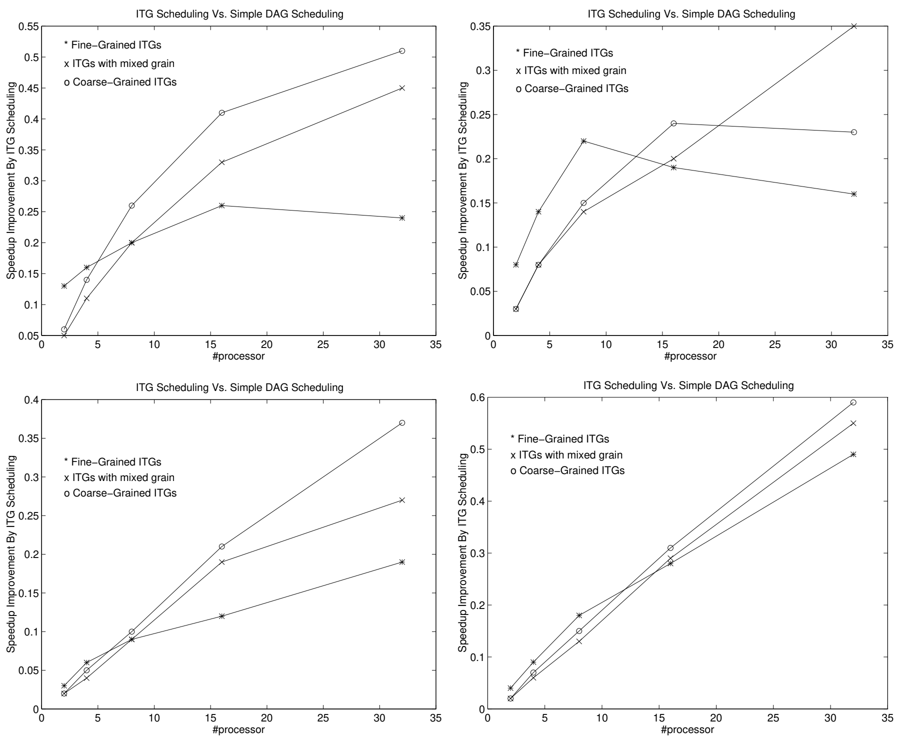

Figure 13: The top left is the average improvement ratio using our cyclic ITG algorithm instead of applying the DAG scheduling for one iteration. The top right is the average improvement ratio after the graphs are unfolded by a factor of 2. The bottom left is the improvement ratio for acyclic ITGs and the bottom right is the ratio after unfolding those ITGs by 2.

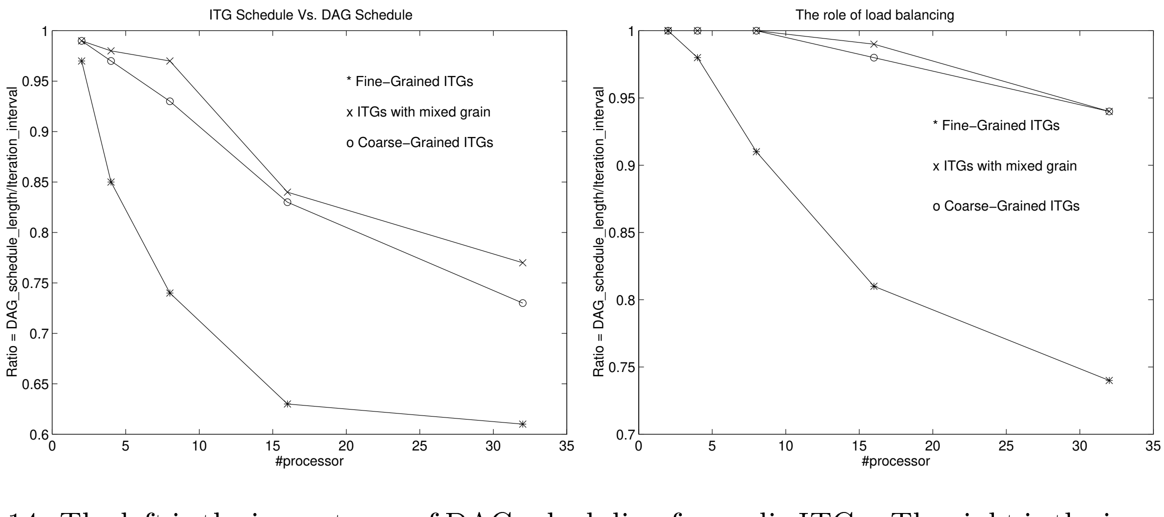

Figure 14: The left is the importance of DAG scheduling for cyclic ITGs. The right is the importance of load balancing for acyclic ITGs.

usually still contains many zeros. The left side of Fig. 15 shows a typical ITG based on the SOR method for a sparse matrix, assuming that the convergence test is not conducted in every iteration but every few hundreds of iterations instead. Some of the edges in this graph are marked distance 1. The unmarked edges have distance 0. We have tested the performance of sparse SOR ITGs. The test matrices are the part of the Harwell-Boeing Test Suites [D92]. We use matrix BCSSTK14 arising from structural analysis for the roof of Omni Coliseum at Atlanta and matrix BCSSTK15 for Module of an offshore platform. The simulated performance is shown in the right part of Fig.15. While the parallelism in a sparse matrix computation is limited, this algorithm is able to explore a decent amount of parallelism.

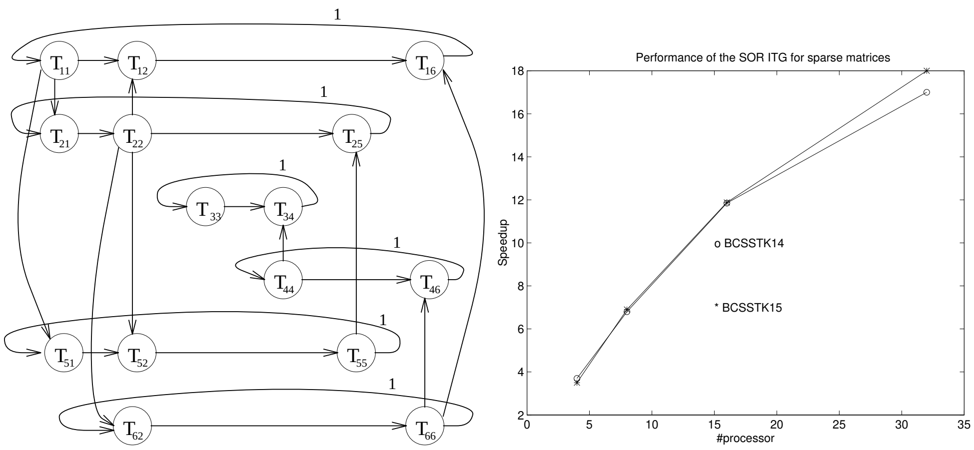

Figure 15: The left part is an ITG for a block-partitioned sparse matrix. The right side is the scheduling performance for BCSSTK14 and BCSSTK15.

The main difficulty in efficiently parallelizing the SOR methods stems from the intra-iteration data dependencies (cf. Jacobi method), as iterates values depend on computed values of the same iteration. The multi-coloring method has been proposed [AJ86] for executing SOR-based iterative computation. This scheme performs variable relabeling to increase the amount of exploitable parallelism since nodes assigned the same color can be computed concurrently. Adams and Jordan [AJ86] have shown that the multi-coloring method is numerically equivalent to performing several SOR iterations computed simultaneously in the absence of convergence tests. However, in order to preserve inter-color data dependencies, processors must communicate between each computation phase associated with each color. This constitutes a drawback in the implementation of the multi-coloring scheme for message-passing machines as communication and computation cannot be overlapped. Our approach based on software pipelining is essentially to overlap computation and communication of several SOR iterations. We conduct the experiments in using the SOR method for solving banded matrices to compare the multi-coloring approach with our approach [DY95]. Table 2 shows the megaflops obtained in nCUBE-2 and Intel Paragon machines (vector units are not used). The left part shows the megaflop performance for fixed matrix size n and $p = 64$. The right part is for a fixed matrix band width. The experiments indicate that our approach that overlaps several SOR iterations outperforms the coloring method when many colors are needed.

width	64 procs n:12,800			
	nCUBE-2		Paragon	
	color	overlap	color	overlap
21	24.5	37.3	149	309
41	26.0	41.4	165	430
81	26.1	43.9	167	537
161	26.5	45.3	171	608

p (procs)	Width 161, Paragon			
	n=12800		n/p=200	
	color	overlap	color	overlap
4	22.4	41	11.1	38
8	41.3	81	22.1	76
16	73.9	162	44.3	152
32	121.4	314	88.9	304

Table 2: The left part is the megaflops obtained with the fixed matrix size and $p = 64$ and the right part is with the fixed matrix bandwidth.

8 Concluding Remarks

Our experiments show that the automatic scheduling algorithms for ITGs delivers good performance on message-passing architectures. We have also presented a method for executing ITG schedules and provided analytic and experimental results to show that run-time performance of static schedules is stable if variations between predicted static weights and actual run-time weights are not significant. Currently we are implementing a run-time support system for efficiently executing graph schedules. [CFH95] proposed a hierarchical tiling framework and our work is useful and can be extended for assisting performance prediction. Our algorithms have not considered the overhead of memory accessing and effect of caching and [WF93] discussed an approach for addressing this issue.

The scheduling algorithms we present have not considered the processor distance. It has been shown (e.g. [Dun91]) in the modern processor architectures, the distance factor does not affect the cost of communication significantly because of wormhole routing technology. However, this routing scheme requires the exclusive use of channels, and there exists communication contention if network traffic is heavy. Assigning communication-intensive tasks close to each other in terms of processor distances would reduce the chance of contention. The communication contention is hard to model. In [YG92], the processor distance information is incorporated in computing communication volume after virtual processor assignments of tasks in a DAG are determined. Then physical mapping can be adjusted [B90]. We are investigating a similar method for mapping ITGs.

Acknowledgment

This was supported in part by NSF RIA CCR-9409695, a startup fund from UCSB and ARPA contract DABT-63-93-C-0064. We thank Apostolos and Vivek Sarkar for participating the initial stage of this research and providing valuable inputs, Val Donaldson and Jeanne Ferrante for their insightful comments and correcting an error in the draft version of this paper, and Pedro Diniz for his help in algorithm implementation.

References

[AJ86] L. Adams, and H. Jordan, Is SOR color-blind?, *SIAM J. Sci. Stat. Comp*, 7 (1986), pp 490-506.

[AN88] A. Aiken and A. Nicolau, Optimal Loop Parallelization, *SIGPLAN 88 Conf. on Programming Language Design and Implementation.* pp.308-317.

[B90] S. H. Bokhari, *Assignment Problems in Parallel and Distributed Computing.*, Kluwer Academic Publisher, 1990.

[CFH95] L. Carter, J. Ferrante and S. F. Hummel, Efficient parallelism via hierarchical tiling, *Proc. of Seventh SIAM Conf. on Parallel Processing for Scientific Computing,* 1995, pp. 680-685.

[CD73] E. G. Coffman and P. J. Denning, *Operating Systems Theory*, Prentice Hall, 1973.

[CL95] M. Cosnard and M. Loi, Automatic Task Graph Generation Techniques, *Proc. of the Hawaii International Conference on System Sciences*, IEEE, Vol II. 1995. pp. 113-122. A revised version will appear in *Parallel Processing Letters*, a special isssue on partitioning and scheduling.

[CR92] Y-C Chung and S. Ranka, Applications and performance analysis of a compile-time optimization application for list scheduling algorithms on distributed memory multiprocessors, Supercomputing 92, pp. 512-521.

[Ch89] P. Chretienne, Task Scheduling over Distributed Memory Machines, *Proc. of Inter. Workshop on Parallel and Distributed Algorithms*, (North Holland, Ed.), 1989.

[Ch93] P. Chretienne, Cyclic scheduling with communication delays: a polynomial special case. Dec 1993. Tech Report, LITP.

[Dun91] T. H. Dunigan, Performance of the INTEL iPSC/860 and nCUBE 6400 Hypercube, ORNL/TM-11790, Oak Ridge National Lab., TN, 1991.

[DY95] P. Diniz and T. Yang. *Efficient Parallelization of Relaxation Iterative Methods for Solving Banded Linear Systems on Multiprocessors*, TRCS94-15, Dept. of CS, UCSB.

[D92] I. S. Duff, R. G. Grimes and J. G. Lewis, Users' Guide for the Harwell-Boeing Sparse Matrix Collection, TR-PA-92-86.

[ELA94] H. El-Rewini, T. G. Lewis and H. H. Ali, Task Scheduling in Parallel and Distributed Systems, Prentice Hall, 1994.

[FYG95] C. Fu, T. Yang, and A. Gerasoulis, Integrating software pipelining and graph scheduling for iterative scientific computing, To appear in *Lecture Notes in Computer Science, Proc. of Irregular '95,* Lyon, France, Sept 1995.

[GR89] H. Gabow and R. Tarjan, Faster scaling algorithms for network problems, *SIAM J. Computing*, Oct 1989.

[GS92] F. Gasperoni and U. Schweigelshohn Scheduling Loops on Parallel Processors: A simple algorithm with close to optimum performance. *Proc. of CONPAR 92* , pp. 613-624.

[GY93a] A. Gerasoulis and T. Yang, On the Granularity and Clustering of Directed Acyclic Task Graphs, *IEEE Trans. on Parallel and Distributed Systems.*, Vol. 4, no. 6, June 1993, pp 686-701.

[H89] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, Scheduling precedence graphs in systems with interprocessor communication times, *SIAM J. Comput.*, pp. 244-257, 1989.

[L88] M. Lam, Software pipelining: an effective scheduling technique for VLIW machines, *ACM Conf. on Programming Language Design and Implementation*, 1988, 318-328.

[KB88] S.J. Kim and J.C. Browne, A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures, *Proc. of Inter. Conf. on Parallel Processing*, 1988, V3, 1-8.

[MG89] C. McCreary and H. Gill, Automatic Determination of Grain Size for Efficient Parallel Processing, *Communications of ACM,* vol. 32, pp. 1073-1078, Sept., 1989.

[N88] A. Nicolau, Loop quantization: A generalized loop unwinding techniques, *J. of Parallel and Distributed Computing,* 5(1988), 568-586.

[PLW94] M. A. Palis, J.-C. Liou; Wei, D.S.L. A greedy task clustering heuristic that is provably good. *Proceedings of the 1994 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN).* IEEE 1994. p. 398-405.

[PM91] K. K. Parhi and D. G. Messerschmitt, Static rate-optimal scheduling of iterative dataflow programs via optimum unfolding, *IEEE Trans. on Computers*, 40:2, 1991, pp. 178-195.

[PAM93] S. S. Pande, D. P. Agrawal and J. Mauney, A scalable scheduling scheme for functional parallelism on distributed memory multiprocessor systems. *IEEE Trans. on Parallel and Distributed Systems,* April 1995, vol.6, (no.4):388-99.

[P88] C. D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic, 1988.

[PKK93] K. Psarris, X. Kong, and D. Klappholz. The Direction Vector I Test. *IEEE Trans. on Parallel and Disributed Systems*, Vol. 4, No. 11, November 1993, pp. 1280-1290.

[RA93] J. Ramanujam, Multiple dimension software pipelining. *Proc. of IPPS 94.*

[RE68] R. Reiter, Scheduling parallel computations, *Journal of ACM*, Oct 1968, pp. 590-599.

[SCMB] Saltz, J., Crowley, K., Mirchandaney, R. and Barryman,H., Run-time scheduling and execution of loops on message passing machines, *J. of Parallel and Distributed Computing,* Vol. 8, 1990, pp. 303-312.

[SK89] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, MIT Press, 1989.

[SK92] V. Sarkar and R. Thekkath, A general framework for iteration-reordering loop transformations, *ACM Conf. on Programming Language Design and Implementation*, 1992, 175-187.

[VGN92] V. H. Van Dongen, G. R. Gao and Q. Ning A polynomial time method for optimal software pipelining. *Proc. of CONPAR 92,* pp. 613-624.

[WF89] M. Wolfe, *Optimizing Supercompilers for Supercomputers,* MIT Press, Boston, 1989.

[WF93] R. Wolski and J. Feo, Program Partitioning for NUMA Multiprocessor Computer Systems, Tech. Report, Lawrence Livermore Nat. Lab., 1992. *J. of Parallel and Distributed Computing*, 1993.

[WG88] M. Y. Wu and D. Gajski, Hypertool: A programming aid for message-passing systems, *IEEE Trans. on Parallel and Distributed Systems,* vol. 1, no. 3, pp.330-343, 1990.

[YG94] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors, *IEEE Transactions on Parallel and Distributed Systems,* Vol. 5, No. 9, 951-967, 1994.

[YG93] T. Yang and A. Gerasoulis. List scheduling with and without communication. *Parallel Computing*, V. 19 (1993) pp. 1321-1344.

[YG92] T. Yang and A. Gerasoulis, PYRROS: Static Task Scheduling and Code Generation for Message-Passing Multiprocessors, *Proc. of 6th ACM Inter. Confer. on Supercomputing,* Washington D.C., 1992, pp. 428-437.

[YFGS] T. Yang, C. Fu, A. Gerasoulis and V. Sarkar, Mapping iterative task graphs on distributed-memory machines. To appear in *Proc. of 24th Inter. Conference on Parallel Processing*, Aug. 1995.