Mapping Iterative Task Graphs on Distributed Memory Machines

Tao Yang Cong Fu
Dept. of Computer Science
University of California
Santa Barbara, CA 93106
{tyang,cfu}@cs.ucsb.edu

Apostolos Gerasoulis
Dept. of Computer Science
Rutgers University
New Brunswick, NJ 08903
gerasoulis@cs.rutgers.edu

Vivek Sarkar
ADTI, 555 Bailey Ave.
IBM Software Solutions Division
San Jose, CA 95141
vivek_sarkar@vnet.ibm.com

Abstract

This paper addresses the problem of scheduling iterative task graphs on distributed memory architectures with nonzero communication overhead. The proposed algorithm incorporates techniques of software pipelining, graph unfolding and directed acyclic graph scheduling. The goal of optimization is to minimize overall parallel time, which is achieved by balancing processor loads, exploring task parallelism within and across iterations, overlapping communication and computation, and eliminating unnecessary communication. This paper gives a method to execute static schedules, studies the sensitivity of run-time performance when weights are not estimated accurately at compile-time, and presents experimental results to demonstrate the effectiveness of this approach.

1 Introduction

Many scientific applications can be viewed as the repeated execution of a set of computational tasks and can be modeled by iterative task graphs (ITGs). Mapping weighted iterative task graphs on message-passing architectures requires the exploration of both task and loop parallelism. Task parallelism has been addressed in the context of DAG (Directed Acyclic Graphs) scheduling[4, 16, 20, 24, 27]. Using graph scheduling algorithms for iterative computation is not feasible since the number of iterations may be too large or may not even be known at compile-time. Loop scheduling has been studied extensively in the previous work e.g. [6, 17, 21]. Software pipelining [2, 11, 15, 19, 22] is an important technique proposed for instruction-level loop scheduling on VLIW and superscalar architectures. Loop unrolling or graph unfolding techniques [2, 18] 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.

In addition to load balancing, the important optimization needed for distributed memory systems is the elimination of unnecessary communication and overlapping of computation with communication. It is known that communication in such systems suffers high startup cost and *data locality* must be explored. In the task model, exploring data locality means to localize data communication between tasks by assigning them in the same processor [20].

The main focus of our work is to guide task mapping based on the theory of graph scheduling and minimize the performance difference between the proposed solution and an optimal solution. The goal of this paper is to briefly present scheduling algorithms for ITGs and then discuss how the scheduling results can be used in executing an ITG. The proposed algorithm contains the following parts: graph characterization, unfolding, graph transformation and DAG scheduling, construction of the ITG schedule, and schedule optimization. The loop unfolding factor is derived so that the scheduling performance can be guaranteed compared to the optimal solution. 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.

This paper is organized as follows. Section 2 gives the problem definition and assumptions. Section 3 gives a brief description of a scheduling algorithm and Section 4 provides the properties of this algorithm.

Section 5 presents the schedule executing scheme and examines the run-time performance of static schedules. Section 6 presents the experimental results.

2 Definitions and Backgrounds

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 $(1 \leq k \leq N)$. 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. 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. Control dependence is not considered in this paper. 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 $c_{x,y}$ for sending a message from task T_x in one processor to task T_y in another processor.

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. 1(a) shows an ITG where edges are labeled with dependence distances. Fig. 1(b) shows another ITG. For these two examples, all task weights are 20 and all edge weights are 10 between any two processors. The expanded graph for Fig. 1(b) is in (c). A schedule for this graph when $N = 3$ is depicted in Fig. 1(d). The parallel time is 290.

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. 1(b) is an ITG obtained by unfolding the ITG of Fig. 1(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, $E(G, N)$.

In general, the scheduling problem for ITGs is NP-hard. There could be several approaches for addressing this problem. 1) Applying DAG algorithms for an expanded graph has a high complexity and it is not feasible if N is unknown. 2) Applying a DAG algorithm for one iteration can only exploit parallelism within one iteration but not across iterations. 3) The idea of exploring parallelism within and across iterations has been proposed for instruction-level software pipelining [2, 15]. In [11], 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. The optimal solution for a special class of ITGs has been studied in [5]. Graph unfolding techniques [2, 18] 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 for general graphs so that the complexity of scheduling is not too high. [9] presented an approach to optimal unfolding for a special case.

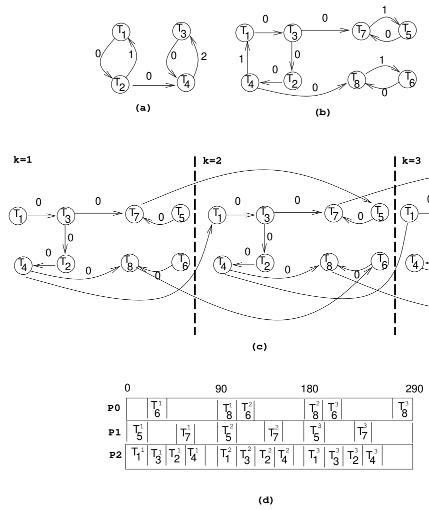

Figure 1: (a) An ITG. (b) Another ITG. (c) The expanded graph of (b) when $N = 3$. (d) A schedule for (b).

The problem of scheduling with communication is much harder than that without communication [20]. Our algorithm needs to eliminate unnecessary communication and overlap communication with computation. The above 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.

We will use the concept of integer-periodic scheduling [19] 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 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. In order to execute a schedule on a parallel machine, we need to construct a code that follows the scheduling result. It can be seen from Fig. 1(d) 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 [2], 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.

In order to derive a schedule which is near optimal, we conduct a comparison of our schedule with an optimal solution with the minimum time for $E(G, N)$. We assume that N is very large and our analysis measures the asymptotic performance of scheduling. We also assume that the given graph always has a cycle. The acyclic case is discussed in [28].

3 The Scheduling Algorithm

The algorithm for mapping an ITG on p processors contains the six steps.

3.1 Computing performance parameters

Smallest iteration interval. We first 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 task weights in this cycle. This is the well-known optimal rate (the smallest iteration interval) of pipelining when communication cost is zero and there is a sufficient number of processors [19]. To compute this value, we set up the following inequalities for all 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 [10] where constant ϵ is the desired accuracy in finding the minimum β such that $\beta^*(G) \leq \beta \leq \beta^*(G) + \epsilon$. $Seq(G) = \sum_{i=1}^{v} \tau_i$. For Fig.1(a), ϵ is chosen as 0.5 and β is computed as 40.

Granularity. Next we estimate the granularity of the graph as follows. This value will be useful in computing the performance bound of the produced schedule and the unfolding factor.

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

For example, for Fig.1(a), $g(G) = 2$.

Unfolding factor. Then we compute the unfolding factor f. The analysis in Section 4 shows that such a factor will guarantee that the asymptotic performance of this scheduling is competitive to an optimal solution. The competitive ratio is: $B = S1(1 + \frac{\epsilon}{\max(\beta-\epsilon, Seq(G)/p)}) + 1$. where $S1 = 1 - 1/p + 1/g(G)$. The unfolding factor f is $\lceil \min(L1, L2) \rceil$ where

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

and τ^{max} is the maximum task weight in G. For example, for Fig.1(a), $B = 2.68$ and $f = 2$.

3.2 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 [18] to construct G^f.

For example, unfolding the ITG in Fig. 1(a) with $f = 2$ results in an ITG shown in Fig. 1(b). Notice the unfolded graph has the following properties [18]: $\beta^*(G^f) = f\beta^*(G)$.

3.3 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 $f\beta - f\epsilon \le \beta^*(G^f) \le f\beta$. Thus the value β_f can be easily found in this range using the binary searching and the shortest path algorithm. The result satisfies $\beta^*(G) \le \beta_f \le \beta^*(G) + \epsilon$.

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

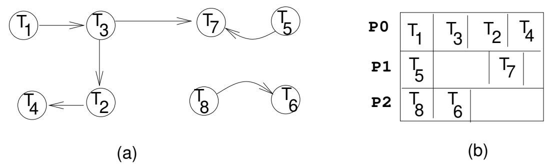

Figure 2: (a) The kernel DAG of Fig. 1(b). (b) The schedule for this DAG.

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. 1(b) is deleted because its edge weight satisfies $MOD(\alpha_4, \beta) - MOD(\alpha_1, \beta) < \tau_1$. This transformation was first proposed in [11] for mapping graphs when communication is zero. We can show such a transformation is still valid for our case by carefully designing the ITG schedule in Section 3.5. 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. 1(b) is in Fig. 2(a).

3.4 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 [27], 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 3.5. Both algorithms have a complexity $O((v+e)\log v)$[1].

Our multi-stage approach contains the following mapping operations: 1) Assign tasks to a set of clusters using the DSC Algorithm [25]. The goal of clustering is to identify communication-intensive tasks and Tasks in the same cluster will be executed in the same processor. 2) If the number of clusters is larger than p, we use a simple load balancing heuristic to balance the processor load. 3) We further order the execution of tasks within each processor using the RCP algorithm [26] with a goal of overlapping communication with computation to hide communication latency. [12] conducted a comparison of this approach with a higher complexity ETF method[14] and found that this approach has a much lower complexity but a competitive performance.

The one-stage approach uses the idea of the DSC algorithm [25] but limits the number of clusters to p. The detail is in [28]. We apply this algorithm to the given DAG and also the reversed graph. This algorithm can obtain the optimum for fork, join, coarse grained tree DAGs if there is a sufficient number of processors and has the following bound.

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

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

where $CP(R)$ is the length of the critical path in this graph R.

The ETF algorithm [14] has a similar performance bound with a complexity $O(pv^2)$ while our algorithm has a lower complexity $O((v+e)\log v)$.

3.5 Constructing an ITG schedule

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. The starting time of a task T_i at iteration k is modeled as $ST(T_i^k) = \alpha_{p,i} + \beta_p(k-1)$ where $\alpha_{p,i}$ is called startup delay and β_p is the length of the 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. The startup delay of task T_i in the 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 is

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

[1] 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.

The processor assignment of instances of T_i in this ITG schedule is the same as $Proc(i)$ derived by the DAG scheduling algorithm.

For the kernel DAG of Fig. 2(a) and $p = 3$, the DAG scheduling produces the following solution as shown in Fig. 2(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.

3.6 Optimization for communication contention

So far we have not considered the processor distance. It has been shown (e.g. [7]) 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. In this step, we incorporate the processor distance factor in estimating communication overhead and perform a series of pairwise interchanges to adjust the ITG schedule. The complexity is $O(p^2(v \log v + e))$. A similar strategy is used in PYRROS [27]. It should be noted that communication contention exists in other routing schemes. Modeling contention is hard and more investigation is needed.

4 Analysis

The overall time complexity of this algorithm is $O(\sqrt{v}e \log^2 \frac{Seq(G)}{\epsilon} + p^2(v \log v + e))$. We can show that the derived schedule satisfies all dependence and resource constraints. We also conduct an analysis to compute the unfolding factor and show that this factor gives a guarantee on the asymptotic performance bound of this heuristic algorithm.

Theorem 2 *Let the unfolding factor f be chosen as: $f = \lceil \min(L1, L2) \rceil$ where*

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

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.

This theorem indicates that since ϵ could be chosen small, the bound is about $S1+1 = (1-1/p+1/g(G))+1$. When $g(G)$ 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. The theorem also indicates that program partitioning that produces ITGs should make $g(G)$ not too small. This is consistent to the previous results [13].

5 Run-time execution and sensitivity analysis

5.1 A schedule executing method

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 3, 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 [2] 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 $\alpha_{p,i} + \beta_p(o_i - 1)$. In Fig. 1, 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 is T_8, T_6. We show the correctness of this method as follows.

Theorem 3 *$[L_j, L_j + \beta)$ 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 $o_i = \lceil \frac{L_j - \alpha_{i,p}}{\beta_p} \rceil + 1$.*

Based on the above theorem, we give the code structure for executing the schedule at processor j in Fig. 3. 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 [27].

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

Figure 3: A generic form of code for Processor P_j.

5.2 An analysis of run-time performance

Code in Fig.3 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.3 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.3. 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: 1) Variation of task computation weight, due to inaccurate estimation of static weight and the cost of data copying. 2) Variation of communication delay, due to inaccurate estimation of static communication weight and unexpected run-time network contention.

We address the sensitivity of the scheduling performance on weight variation. Let G be the graph for which the scheduling algorithm produces a schedule S. The weights of this graph may change at run-time and we call the new graph as G^r. The run-time method in Fig. 3 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 $c_{i,j}^r$ and the computation weight τ_i is changed to τ_i^r. They satisfy: $(1-\delta_1)c_{i,j} \leq c_{i,j}^r \leq (1+\delta_2)c_{i,j}, (1-\delta_1)\tau_i \leq \tau_i^r \leq (1+\delta_2)\tau_i$. We have shown in Theorem 2 that the derived schedule satisfies: $PT(G) \leq B \times PT_{opt}(G)$ where $B = (1 - 1/p + 1/g(G))(1 + \frac{\epsilon}{max(\beta - \epsilon, Seq(G)/p)}) + 1$. The following theorem identifies the relationship between $PT(S, G^r)$ and $PT_{opt}(G^r)$. It shows that if the run-time variation of weights is not significant, a static schedule still produces a competitive performance.

Theorem 4

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

6 Experiments

We have implemented our scheduling scheme and have conducted several experiments to examine the scheduling performance of randomly generated ITGs, and application programs including matrix multiplication, solving a Laplace equation and 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 to verify the correctness of the simulation. In this section we report some experimental results.

Overall performance and its sensitivity. The left part of Fig. 4 lists the simulated and actual speedups on nCUBE-2 for solving an Laplace PDE (400x400 grid) and a banded matrix system (matrix dimension 4000, width 35). In the simulation, we compute the asymptotic speedup of an ITG schedule as: $\lim_{N \to \infty} \frac{N * Seq(G)}{\max_{T_i \in G} ST(T_i^N)} = Seq(G)/\beta_p$. We can see that the simulated performance is close to the actual performance for the studied programs. The right part of Fig. 4 shows the performance variation after changing the weights of PDE and banded matrix ITGs in a range of $[-10\%, +10\%]$. The variation ratio is $\frac{Speedup_{new}}{Speedup_{old}} - 1$. We can see the performance is sensitive but variation is small, within 5%.

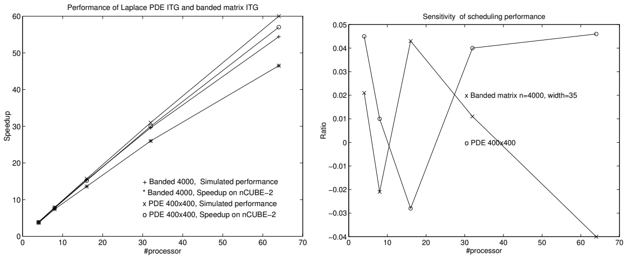

Figure 4: The left is speedups of solving a Laplace equation and a banded matrix system. The right is the performance variation ratio using the inaccurate and accurate weights.

Unfolding and performance bounds. We examine if the computed factor identifies the range of unfolding in improving performance for three groups of randomly generated 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. The results for $p = 32$ are presented in Fig. 5. The X axis is the unfolding 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: $\frac{Speedup_{new}}{Speedup_{old}} - 1$. We can see that unfolding greatly improves the performance. This is because unfolding produces more tasks and gives more flexibility for the DAG scheduling.

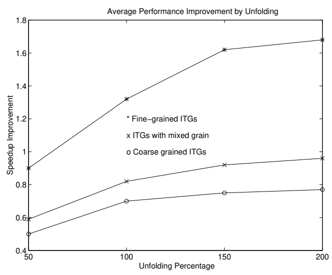

Figure 5: Performance improvement by unfolding.

We have also examined the actual bounds and the estimated upper bounds after unfolding for above tested cases. The actual bounds are no more than 1.8. We find that 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, which indicates that our unfolding factor effectively guides the algorithm to obtain a near optimal schedule. 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.

Software pipelining and DAG scheduling. We have discussed that 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 have examined the 50 random ITGs and compared the performance of this simple DAG scheduling approach with our algorithm. 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 left part of Fig 6 shows the average performance improvements of our approach over this simple DAG scheduling approach after unfolding by a factor of 2.

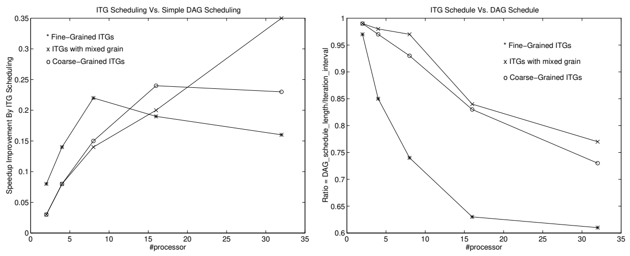

Figure 6: The left is the average improvement ratio by software pipelining after the graphs are unfolded by a factor of 2. The right shows the role of DAG scheduling in our algorithm.

Our pipelining algorithm still uses the DAG scheduling for kernel graphs and the performance of DAG scheduling is critical for constructing the final ITG schedule. Notice that $\beta_p = max(D, PT(K(G^f))$. The right part of Fig. 6 shows the average ratios $PT(K(G^f))/\beta_p$ for three groups of these randomly generated graphs. 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 and the role of communication comes to play. But still the DAG scheduling contributes more than 60% to the final performance.

Sparse matrix computation. In solving a sparse matrix system, the SOR iterative methods could be used [1]. The left part of Fig. 7, shows an 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. The right part of Fig.7 is the simulated scheduling performance for sparse matrices in the Harwell-Boeing Test Suites [8]. 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. While parallelism in a sparse matrix computation is limited, this algorithm is able to explore a decent amount of parallelism.

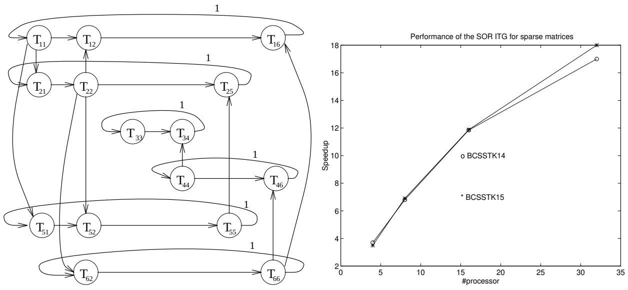

Figure 7: The left is a SOR ITG for a sparse matrix. The right is performance of the SOR ITG for BCSSTK14 and BCSSTK15.

7 Conclusions

Our experiments show that the automatic scheduling algorithm for ITGs delivers good performance on message-passing architectures. Our work is useful for assisting performance prediction in compilation optimization such as program partitioning [3, 17, 21]. Currently we are implementing code generation and runtime support for executing ITG schedules and investigating applications in sparse matrix computations.

Acknowledgement

This was supported in part by NSF RIA CCR-9409695 and by ARPA contract DABT-63-93-C-0064. We thank Pedro Diniz for his help in algorithm implementation, and the referees for their useful comments.

References

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

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

[3] L. Carter, J. Ferrante and S. F. Hummel, Efficient parallelism via hierarchical tiling, *Proc. of 7th SIAM Conf. on Parallel Processing for Scientific Computing,* Feb. 1995, 680-685.

[4] 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.

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

[6] R. Cytron, Doacross: Beyond vectorization for multiprocessors, *Proc. of the 1986 Inter. Conf. on Parallel Processing,* St. Charles, Ill,1986, pp. 836-844.

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

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

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

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

[11] 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.

[12] A. Gerasoulis, J. Jiao, and T. Yang, A multistage approach to scheduling task graphs. To appear in DIMACS Book Series on Parallel Processing of Discrete Optimization Problems. AMS publisher. Edited by P.M. Pardalos, K.G. Ramakrishnan, and M.G.C. Resende.

[13] 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.

[14] 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.

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

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

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

[18] 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.

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

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

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

[22] 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.

[23] R. Wolski and J. Feo, Program Partitioning for NUMA Multiprocessor Computer Systems, *J. of Parallel and Distributed Computing*, 1993.

[24] 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.

[25] 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.

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

[27] 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.

[28] T. Yang, C. Fu, A. Gerasoulis and V. Sarkar, Scheduling Iterative Task Graphs on Distributed Memory Machines, Technical Report TRCS95-07, UCSB, 1995.