# 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 and represented by iterative task graphs (ITGs). An ITG may or may not have dependence cycles. In this paper, we consider the symbolic scheduling of ITGs on distributed memory architectures with nonzero communication overhead and propose heuristic algorithms for scheduling both cyclic and acyclic ITGs without searching the entire iteration space. Our approach incorporates techniques of software pipelining, graph unfolding, directed acyclic graph (DAG) scheduling and load balancing. We analyze the asymptotic optimality of the algorithms to show that the derived schedules are competitive to the optimal solutions. We also study the sensitivity of scheduling performance on inaccurate weights. Finally we present the experimental results to demonstrate the effectiveness of the optimization techniques.

**KEY WORDS:** Scheduling, communication optimization, granularity, software pipelining, iterative task graphs, directed acyclic graphs.

## 1 Introduction

Task graphs are important in representing parallel computations in compiler systems [2, 17, 20, 24] and many graphical parallel programming tools use coarse-grain computation graphs to model user applications [1, 13]. There are several representation schemes for computation graphs and in this paper we consider the scheduling of a class of graphs called iterative task graphs (ITGs). An ITG represents a sequence of task computations where each iteration consists of the execution of a set of tasks. Task dependence may exist within an iteration and across iterations.

Iterative task computations involve both task and loop parallelism [28]. There is a lot of research work on scheduling task parallelism based on directed acyclic graphs (DAG), e.g. [6, 15, 17, 20, 24, 25, 27]. 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. In loop scheduling research, software pipelining [9, 12, 21] is an important technique proposed for exploiting instruction-level parallelism on VLIW and superscalar architectures. Loop unrolling or graph unfolding techniques [16] 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 scheduling techniques for task and loop parallelism can be combined together for mapping iterative task computation on message-passing architectures. [6] 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 a near-optimal unfolding factor for general ITGs.

Load balancing and communication minimization are the important aspects of the optimizations. It is known that *data locality* must be explored for reducing unnecessary communication. In the task computation model [20], exploring data locality means to localize data communication between tasks by assigning them to the same processor and then tasks could exchange data through a local memory to avoid high-cost inter-processor communication [10].

The optimal solution for a special class of ITGs has been studied in [3]. Our heuristic algorithm is designed for general ITGs. 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 and analyze the scheduling performance when the static estimation of task weights is not accurate.

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 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. Symbols used in this paper and the proof of the theorems are listed in the appendix.

## 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 $T_i^k$ $(1 \leq k \leq N)$. 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 on.
- 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.

[1] Our results can be extended for inexact dependence values such as '+' and '*' used in the literature [18].

- Each task $T_x$ in an ITG has a computation cost $\tau_x$ and there is a communication cost for sending a message from task $T_x$ on one processor to task $T_y$ on another processor. The cost is denoted as $c_{x,y}$, which is usually estimated as "startup cost + transmission speed $\times$ size of message"[5].

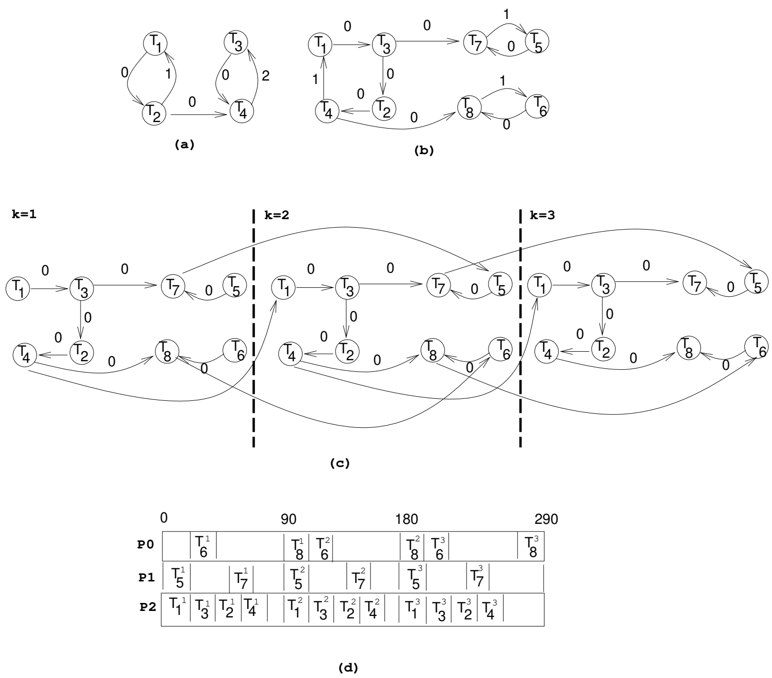



Figure 1: (a) An ITG. (b) Another ITG. (c) The expanded graph of (b) when $N = 3$. (d) The Gantt chart of a schedule when $N = 3$ for (b).

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 the dependence distances between tasks. Fig. 1(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. 1(b) is in (c). A schedule for this graph when $N = 3$ is depicted in Fig. 1(d). The parallel time is 290.

A graph can be unfolded first before being mapped to multiple processors. 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 $G$, $E(G, N)$.

## 3 Approaches for Mapping ITGs

Finding optimal solutions for scheduling ITGs is hard. We discuss several issues in designing heuristic algorithms:

- *A trade-off between parallelization and data locality.* Since communication overhead is significant in distributed memory machines, a heuristic algorithm needs to eliminate unnecessary communication and overlap communication with computation. We assume that communication between two tasks is zero if these two tasks are mapped to the same processor. Placing two tasks together on one processor to take advantage of data locality could eliminate some communication overhead, but it might reduce parallelism as well.

- *Exploit both task and loop parallelism together.* To maximize the performance, we need to consider both task parallelism within each iteration and loop parallelism across iterations.

- *A trade-off between scheduling performance and complexity.* Since we do not search the entire iteration space, an important issue is to make sure that the derived schedule is near-optimal and is correct in the sense that all dependence constraints are satisfied.

There are several solutions for mapping ITGs:

- Direct application of DAG algorithms to an expanded graph can produce a good solution, but this approach may not be feasible since the number of iterations 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, from Fig. 1(b) we can derive the DAG of each iteration by deleting all edges with nonzero distance, as shown in Fig. 2(a). We schedule this DAG and derive a schedule for it in Fig. 2(b). Then the schedule for the original ITG is shown in Fig. 2(c), which is longer than the schedule in Fig. 1(d). In Fig. 1(d), the executions of task instances from different iterations are overlapped.

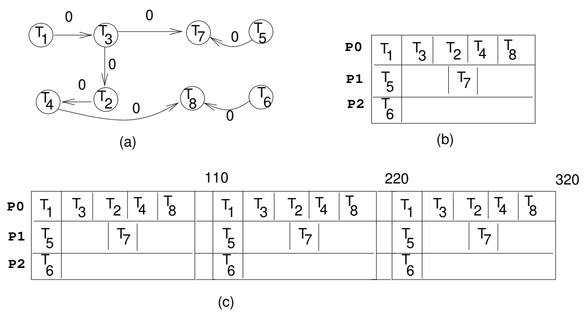



Figure 2: (a) The DAG for one iteration of the ITG in Fig. 1(b). (b) The DAG schedule. (c) The schedule for all iterations.

- The idea of exploring parallelism within and across iterations has been proposed in the context of instruction-level software pipelining [12]. But most of the previous software pipelining algorithms do not give the performance bound. In [9], a near-optimal scheduling algorithm for pipelining with no communication delay is proposed. We need to extend this result to incorporate communication optimization.

- As we mentioned above, graph unfolding techniques [16] have been also developed to increase the number of tasks within each iteration, which gives a scheduler more flexibility to exploit parallelism. For example, in Figure 1(a), the degree of DAG parallelism within one iteration is 2 but after being unrolled by 2, the degree of DAG parallelism becomes 3 in Fig. 1(b) (see Fig. 2(a)).

  Our goal is to examine tasks only from few iterations but exploit parallelism of the entire iteration space. The optimality of our solution is guaranteed by comparing it with the optimum that achieves the minimum time for the expanded graph $E(G, N)$.

We will use the concept of integer-periodic scheduling [19] as in the 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 $t + \beta_p$ where $\beta_p$ 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. 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 $\beta_p$.

It should be emphasized that our analysis measures the asymptotic performance of scheduling and studies 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 denote a term that is negligible when $N$ is very large.

# 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 to 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 $\epsilon$ 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 unfolding factor

Without unfolding the graph, the number of tasks may be too small for a scheduler to optimize load balancing and communication. If we choose a very large unfolding factor, the complexity of this scheduler increases significantly. Our criterion is to select one that can guarantee the scheduler to achieve a performance within a constant factor of the optimum.

The analysis of the achievable bound based on the selected unfolding factor is described in Section 4.6 after the algorithm description. We summarize the information needed in computing the unfolding factor as follows.

- **Smallest iteration interval**. This is the smallest possible value $\beta_p$ for any $p$ when the communication cost is zero [19]. It is defined as:

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

where $d(C)$ is the summation of dependence (edge) distances in this cycle $C$ and $\tau(C)$ is the summation of computation weights of the tasks in this cycle. 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 $\alpha_y$ is a non-negative unknown associated with task $T_y$ and $\beta$ 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 [8] where constant $\epsilon$ is the desired accuracy in finding the minimum value $\beta$ such that $\beta^*(G) \leq \beta \leq \beta^*(G) + \epsilon$. For example, for Fig.1(a), $\epsilon$ is chosen as 0.5 and the value of $\beta$ is computed as 40.

- **Granularity**. We compute the granularity for a given graph as follows:

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

For example, the granularity for the ITG in Fig.1(a) is $g(G) = 2$.

- **The intended bound.** It is difficult to reach a tight performance bound. In Section 4.6 we show that the following bound is reachable and is almost the tightest.

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

where $S1 = 1 - 1/p + 1/g(G)$. Thus our algorithm tries to achieve this bound.

Based on the above pre-computation, we can determine the unfolding factor $f$ as

$$f = \lceil \max(L1, L2) \rceil$$

where

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

and $\tau^{max}$ is the maximum task weight in $G$. For example, for Fig.1(a), $B = 2.68$ and $f = 2$. $L1$ and $L2$ are intermediate terms for $f$ and the details are in Section 4.6. An intuitive explanation is that $L1$ relates to the result of comparing the expected performance with a perfectly load-balanced schedule ($Seq(G)/p$) and $L2$ relates to the smallest iteration interval value ($\beta$) which reflects the critical path of the expanded graph. Notice that both $Seq(G)/p$ and $\beta$ are lower bounds for any optimal solution.

### 4.2 Unfolding the ITG

In this step, the graph is unfolded by a factor of $f$. This new graph is called $G^f$ and needs to be executed only $\lfloor N/f \rfloor$ iterations. We use the unfolding algorithm proposed in [16] to construct $G^f$. The new graph $G^f$ contains $f * v$ nodes such that each node $T_i$ in $G$ corresponds to $f$ 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 $h = \lceil \frac{d-j+1}{f} \rceil$.

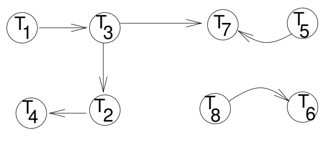
For example, unfolding the ITG in Fig. 1(a) with $f = 2$ results in an ITG shown in Fig. 1(b). Notice the unfolded graph has the following property [16]: $\beta^*(G^f) = f\beta^*(G)$.

### 4.3 Graph transformation

We transform $G^f$ into a DAG so that the DAG scheduling technique could be applied. The main benefit of this approach is that we could estimate the performance of scheduling and compute the relative bound compared to the asymptotically-optimal solution. Our goal is to make sure the algorithm has a solid theoretical foundation.

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 $\beta_f$ and $\alpha_i$ $(1 \le i \le fv)$ 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 $\alpha_j$ is a non-negative unknown associated with task $T_j$ in $G^f$ and $\beta_f$ is an approximation of $\beta^*(G^f)$ which satisfies $\beta^*(G) \le \beta_f \le \beta^*(G) + \epsilon$. In order to find a $\beta_f$ that approximates $\beta^*(G^f)$ using the binary searching and the shortest path algorithm, we need to know the range of this value. Since $\beta^*(G^f) = f\beta^*(G)$, it can be shown that $f\beta - f\epsilon \le \beta^*(G^f) \le f\beta$.

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 $\epsilon = 0.5$, $\alpha_1 = 0$, $\alpha_2 = 40$, $\alpha_3 = 20$, $\alpha_4 = 60$. $\alpha_5 = 0$, $\alpha_6 = 20$, $\alpha_7 = 40$, $\alpha_8 = 80$.




| | | | | |
|---|---|---|---|---|
| P0 | $T_1$ | $T_3$ | $T_2$ | $T_4$ |
| P1 | $T_5$ | | $T_7$ | |
| P2 | $T_8$ | $T_6$ | | |

(b)

Figure 3: (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 [9] 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 dependencies 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. 1(b) is in Fig. 3(a).

### 4.4 Mapping the kernel graph

In this stage, a DAG schedule is derived for $K(G^f)$. The starting time $\gamma_i$ 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 the two solutions produced by these algorithms. The result of the DAG scheduling is used for constructing the ITG schedule in Section 4.5. The PYRROS algorithm involves the following multiple stages in scheduling: 1) It clusters tasks with intensive communication using the DSC algorithm [26]. 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 $O((v+e)\log v)$. 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 PYRROS algorithm 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. In Section 7, we conduct experiments to examine the impact of using this one-stage algorithm.

In the one-stage approach, we use the idea of the DSC algorithm [26] 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 processor). 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 the 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 $\gamma_x$ of this task unless there are no processors available. The minimization procedure is also used in DSC ( [26], 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 unnecessary communication could further be saved.

The complexity of this algorithm is $O((v+e)\log v)$. If we apply this algorithm to the given DAG and also the reversed graph, we can show that this algorithm has the following properties: when there is a sufficient number of processors, the algorithm reaches the optimum for fork, join and 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.*

The ETF algorithm [11] 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 $\gamma_i$ 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 $k$ in the ITG schedule is modeled as $ST(T_i^k) = \alpha_{p,i} + \beta_p(k-1)$, where $\alpha_{p,i} = \gamma_i + \beta_p H_i$, $H_i = DIV(\alpha_i, \beta_f)$, $\beta_p = max(PT(K(G^f)), D)$ and

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

The correctness of the above terms is discussed in Section 4.6. The $D$ represents the delay needed to insure dependence edges 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. 3(a) and $p = 3$, the DAG scheduling produces the following solution as shown in Fig. 3(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 $\gamma_4 = 60$. Also $D = 90$ and $PT(K(G^f)) = 80$, thus $\beta_p = 90$. We need to transform this schedule into an ITG schedule. The result is shown in Fig. 1(d). 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 [9]. 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.*

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 only relies 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 $E(G, N)$).*

This theorem indicates that since $\epsilon$ 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 the data that a task communicates is proportional to the amount of computation work the task does. Therefore 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 the program partitioning that produces ITGs should make $g(G)$ not too small.

## 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 bound. We set $B = \frac{2p}{p+1} + 1$.

Step 2: We construct a weighted DAG which has 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 the load among those processors is balanced. Our load balancing method uses the classical list scheduling algorithm with the highest-weight-task-first principle. The results of this mapping are the starting time $\gamma_i$ for each task and the processor assignment $Proc(i)$. 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. 4(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. All the communication delays 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. 4(b) shows the processor mapping obtained by the load balancing method described in Step 3 with $LPT = 40$. Fig. 4(c) shows the Gantt chart of part of the schedule with $\beta_p = 40$, and the asymptotic speedup of this schedule is 2.5 .

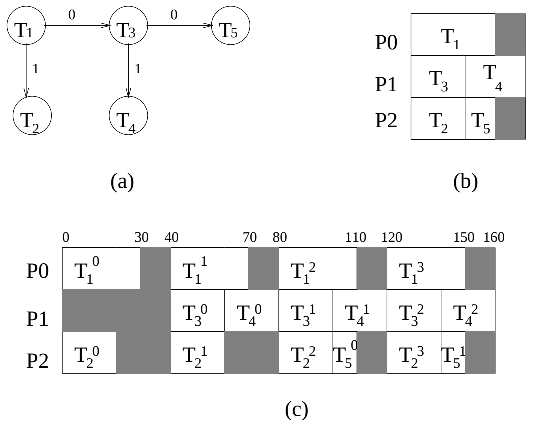



Figure 4: (a) An acyclic ITG. (b) The processor assignment derived in Step 3 using the load balancing algorithm. (c) A portion of the final schedule for this ITG.

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.*

**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*

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

*under condition $B > \frac{2p}{p+1}$.*

We can see that for the 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 the 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.

# 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

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 to the overall performance. Our approach is based on the idea of "periodic execution" in software pipelining. The difference is that in the instruction-level pipelining, a periodic pattern is usually global 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 the starting time function $ST(T_x) = \alpha_{i,p} + \beta_p(k-1)$. Let $Tasks(j)$ be all the 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. 1, processor 0 has two tasks $T_6$ and $T_8$. Since $max(\alpha_{p,6}, \alpha_{p,8}) = max(20, 90) = 90$, a local periodic pattern starts at $[90, 180)$. The iteration sequence numbers for $T_6$ and $T_8$ are 2 and 1 respectively. 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 $o_i = \lceil \frac{L_j - \alpha_{i,p}}{\beta_p} \rceil + 1$.*

Based on the above theorem, we give the code structure for executing the schedule at processor $j$ in Fig. 5. When executing each individual task, the task receives the data items it needs, performs computation, and then sends out data produced for its children (using aggregate multicasting if possible). The style is similar to the one used in the RAPID and PYRROS systems [7, 27]. Next we examine the performance of this code structure and provide an analysis on performance variation between the predicted parallel time and the actual run-time performance.

Let $o^{max} = \max_{T_x \in Tasks(j)} o_x$ and $o^{min} = \min_{T_x \in Tasks(j)} o_x$.
**For** $r = -o^{max}$ **to** $N - o^{min}$
Execute tasks $T_i^k$ ($T_i \in Tasks(j)$) in the 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 5: A generic form of code for processor $P_j$.

### 6.2 An analysis of run-time performance

The code in Fig.5 executes a 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 the computed $\beta_p$. We call the periodic schedule used in Fig.5 to be S, and define $PT(G)$ to be the parallel time estimated by our algorithm using parameter $\beta_p$. Let $PT(S, G)$ be the parallel time produced by the code in Fig.5. 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 predicted. 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, the cost of fetching data from the communication buffer and copying the data from the local memory to the communication buffer. 2) 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. 5 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 $\tau_i$ is changed to $\tau_i^r$. They satisfy: $(1-\delta_1)c_{i,j} \leq c^r_{i,j} \leq (1+\delta_2)c_{i,j}$, $(1-\delta_1)\tau_i \leq \tau_i^r \leq (1+\delta_2)\tau_i$. We have shown in Theorems 3 and 5 that the derived schedule satisfies: $PT(G) \leq B \times PT_{opt}(G)$. The following theorem identifies the relationship between $PT(S, G^r)$ and $PT_{opt}(G^r)$.

**Theorem 7**

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

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

The current implementation of our algorithms takes an ITG as input, simulates the scheduling performance on a message-passing machine using its communication and computation parameters, and produces a schedule for each processor. Throughout this section, we will use the following tested cases.

- **A sparse linear system solver**. This solver is based on the iterative SOR method which is still used in many engineering applications with some variation [22], even new iterative methods such as CG are developed. We use submatrix partitioning and a partitioned sparse matrix usually contains many zero blocks. Notice that since the number of iterations for convergence is large, the convergence test can be conducted every few hundreds of iterations instead of every iteration. Sparse SOR ITGs are irregular, cyclic and have mixed granularities. Their structures are demonstrated in Fig. 6(a). Two test sparse matrices are from the Harwell-Boeing Test Suites: matrix BCSSTK14 arising from structural analysis for the roof of Omni Coliseum at Atlanta and matrix BCSSTK15 for Module of an offshore platform.
- **The Laplace PDE solver.** The PDE cyclic ITGs are regular based on a uniform grid partitioning on a 2-D region. Irregular structures are possible if non-uniform grids are used. The previous compiler techniques can solve the regular Laplace PDE problem, but usually cannot explicitly predict the performance. Through the use of this example, we want to demonstrate that the PDE ITGs can be mapped effectively and the speedup performance can be predicted. Also the regularity in this problem makes it easy for us to implement a schedule on nCUBE-2 to verify the speedup. The structure of PDE ITGs is demonstrated in Fig 6(b). Two graphs are tested: one is a coarse grained grid of size $400 \times 400$, another is a fine grained grid of size $200 \times 200$.
- **A banded system solver**. This problem is based on an application in [23]. The structure is cyclic as demonstrated in Fig. 6(c), similar to the sparse SOR solver and but with some regularity. Two testing instances are both coarse grained, for matrices of sizes $600 \times 600$ and $800 \times 800$ with width 60.
- **Trellis.** This is an FFT-based application example used in [14] for processing a sequence of signal information. The ITG for Trellis is acyclic and has mixed granularities. The structure of Trellis ITGs is demonstrated in Fig 6(d) and two instances will be tested.
- **Randomly generated ITGs**. We examine 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 the range [0.1,1]. The number of tasks in each ITG varies from 30 to 60. Each group has 10 ITGs. This experimental approach has been used in the algorithm research area to identify the weakness and strength of an algorithm for all possible situations.

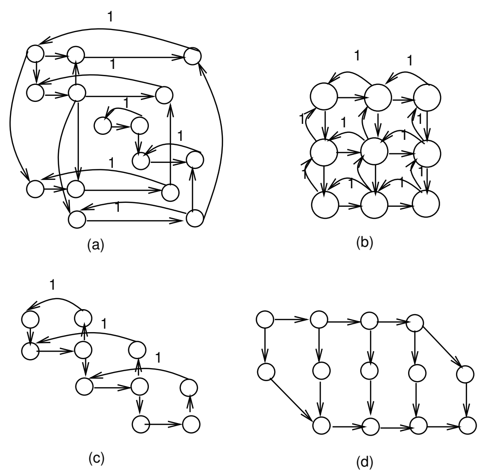


Figure 6: The structures of the four tested application ITGs.

Through the rest of this section, we use SPS, PDE, TRL and BND to denote the first four types of the above graphs respectively. We use RNF, RNM, and RNC to denote the fine, mixed and coarse grained random ITGs. In the following we report the experimental results which address several performance aspects of our algorithms on the above ITGs. Except for the speedups, we report the average performance for each tested group. When we measure the speedup improvement to compare two approaches, we use ratio $Speedup_{new}/Speedup_{old} - 1$.

**Overall speedups obtained by scheduling**. We report the performance of the simulation in which we compute the asymptotic speedup of an ITG schedule as: $\lim_{N\to\infty} \frac{N*Seq(G)}{\max_{T_i\in G} ST(T_i^N)} = Seq(G)/\beta_p$. The results are listed in Table 1. All the speedups are obtained after the ITGs are unfolded by the factor derived from the scheduling algorithms. This result shows that the scheduler can deliver a good performance for each tested case. Even for sparse matrix computation where parallelism is limited, the algorithm is still able to explore a decent amount of parallelism.

| #proc | SPS | | PDE | | TRL | | BND | |
|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 4 | 3.9 | 4 | 3.46 | 3.8 | 4 | 4 | 3.7 | 3.9 |
| 8 | 7.8 | 7.9 | 7.8 | 7.8 | 8 | 8 | 7.4 | 7.7 |
| 16 | 15.6 | 15.8 | 15.3 | 15.4 | 15.9 | 15.9 | 10.7 | 14.8 |
| 32 | 25.9 | 23.4 | 29.8 | 30.4 | 31.9 | 31.9 | 14.7 | 17.5 |

Table 1: The overall speedup for the four tested applications.

**The effectiveness of unfolding**. We have examined if our algorithms identify the range of necessary unfolding to improve performance. We found that for the tested PDE and SPS graphs, the computed unfolding factors are very close to 1, which implies that further unfolding does not improve the performance any more. For two PDE cases, without unfolding, near full speedups have been achieved. For the BND cases, the performance without unfolding already reaches the limit imposed by the minimum iteration interval. But for TRL and BND graphs, unfolding can still expose more task/loop parallelism. Table 2 lists the average speedup improvements by unfolding on these two applications with 32 processors. We also list the average improvements for randomly generated graphs and the result shows that our algorithms identify the need of unfolding when it is useful.

| BND | TRL | RNF | RNM | RNC |
|---|---|---|---|---|
| 19.5% | 601.9% | 130% | 75% | 68% |

Table 2: The average improvement achieved by unfolding.

**Effectiveness of communication consideration in unfolding**. If we do not consider the communication in computing unfolding factors, unfolding factors will be much smaller, especially for fine-grained and mixed grained cases. Then the algorithm performance degrades if not enough unrolling is conducted. For the cases of TRL, the performance degradation is about 654% while for BND, it is about 19%. This result demonstrates the importantance of considering the communication delay.

**The performance bound**. The average estimated upper bound after unfolding and actual bound are listed in Table 3. The average actual upper bound is approximately computed as $\frac{p}{\text{the obtained speedup}}$ where $p = 32$ for this experiment. 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. For acyclic cases, the bound is usually very tight. The result indicates that our unfolding factor effectively guides the algorithms to obtain a near optimal schedule.

| | SPS | PDE | BND | TRL | RNF | RNM | RNC |
|---|---|---|---|---|---|---|---|
| Estimated | 3.09 | 3.69 | 2.53 | 2.94 | 3.44 | 6.67 | 2.86 |
| Actual | 1.32 | 1.06 | 2.0 | 1.01 | 1.48 | 1.78 | 1.55 |

Table 3: The estimated and actual average bounds achieved by scheduling.

Since the value of unfolding factor affects the complexity and also the size of code, we study how large the unfolding factor is. The unfolding factors computed are small (4-8) in general except for three cases (30-50). We studied those cases and found that bound $B$ 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. 2 that the parallelism is not fully utilized if the DAG scheduling for one iteration without pipelining iterations (called simple DAG scheduling approach) is used. 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*. Table 4 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 when $p = 32$. It can be seen that before unfolding, our approach can achieve as high as 1000% 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. If we decrease the number of processors, the speedup improvement ratio decreases accordingly in general. However there is still a decent amount of improvement.

| | SPS | PDE | BND | TRL | RNF | RNM | RNC |
|---|---|---|---|---|---|---|---|
| No unfolding | 485.9% | 225.1% | 1076.0% | 154.9% | 21.3% | 35.1% | 44.9% |
| Unfolding by 2 | 231.3% | 139.7% | 654.8% | 155.4% | 31% | 45.3% | 41.4% |

Table 4: The average improvement achieved by software pipelining over the simple DAG scheduling.

**The role of DAG scheduling and load balancing**. We investigate how critical the DAG scheduling is for constructing a final ITG schedule. Notice that $\beta_p = max(D, PT(K(G^f))$. We compute the contribution ratio $PT(K(G^f))/\beta_p$ for these cyclic graphs. For acyclic ITGs, we estimate the role of the load balancing algorithm using ratio $LPT/\beta_p$. The larger this contribution ratio is, the more crucial the DAG scheduling or load balancing is. Table 5 lists the average ratios and 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 significant amount to the final performance. For the cases of TRL, the contribution ratio is low without unfolding on p=32 or 8. But after unrolling, the contribution ratio increases to 100%.

| | SPS | PDE | BND | TRL | RNF | RNM | RNC |
|---|---|---|---|---|---|---|---|
| p=2 | 100% | 100% | 100% | 100% | 97% | 99% | 99% |
| p=8 | 100% | 99% | 100% | 57.5% | 88% | 98% | 96% |
| p=32 | 100% | 99% | 99% | 14% | 67% | 86% | 84% |

Table 5: The average contribution ratio of the DAG scheduling for cyclic ITG performance and the load balancing for acyclic cases.

In Section 4.4, we introduce a one-stage DAG scheduling method and use the better result among this method and the PYRROS algorithm. We found that the new method produces better results for most of cases compared to the PYRROS algorithm. Table 6 summarizes the cyclic ITG performance improvement using the new method over the PYRROS.

| | SPS | PDE | BND |
|---|---|---|---|
| p=2 | 1.8% | 0% | 5.0% |
| p=8 | 11.5% | 1.4% | 26.1% |
| p=32 | 49.0% | 2.5% | 30.1% |

Table 6: The average performance improvement of the one-stage method discussed in Section 4.4 over PYRROS DAG scheduling for cyclic ITGs.

**Actual performance and sensitivity analysis on nCUBE-2**. For PDE, we implemented a simplified version of the schedule on nCUBE-2 to verify the simulation result and examine the sensitivities on inaccurate weights. The left part of Figure 7 lists the performance of PDEs. The actual performance is quite close to the prediction of simulation results. There is some discrepancy due to the run-time variation of weights and the system control overhead. We investigate the sensitivity of scheduling by changing the weights of PDE ITGs in a range of $[-10\%, +10\%]$. We compute the variation ratio $\frac{Speedup_{new}}{Speedup_{old}} - 1$ to compare the performance derived using accurate and inaccurate weights. This variation ratio is in the right part of Fig. 7. We can see the performance is sensitive but the variation is small (within 5%).

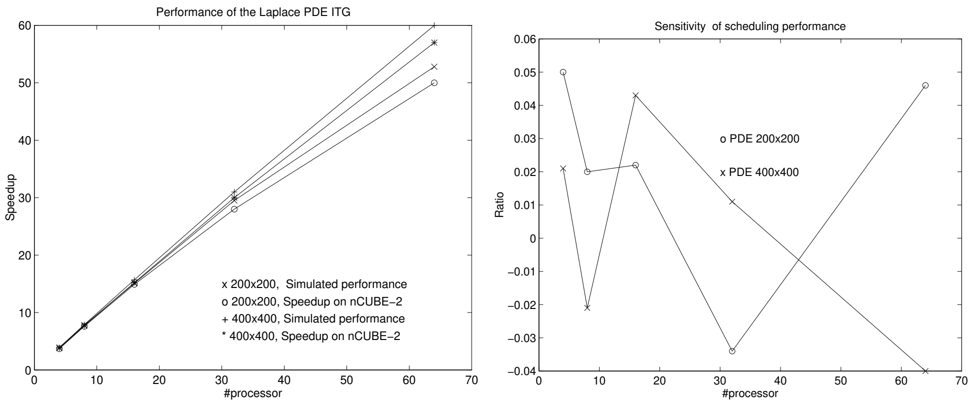



Figure 7: The left is the PDE performance on nCUBE-2 and the right is performance variation ratio of the PDE ITG using the inaccurate and accurate weights.

# 8 Concluding Remarks

The main results of this paper are the development of heuristic ITG scheduling algorithms on distributed memory machines and the theoretical analysis on performance guarantees. An analysis is also provided on run-time performance of static schedules. The theoretical results help us to understand why the algorithms can deliver good performance. These algorithmic and theoretical results are verified through experiments for solving scientific problems which can be represented as ITGs.

The algorithms could be useful for mapping applications in graphical parallel programming tools [1, 7, 13] and run-time systems for irregular problems [7]. Graph scheduling algorithms (mainly for DAGs) have been successful in compiling functional languages such as SISAL [17, 20, 24] and we expect that ITG algorithms will also be useful. For compiling Fortran and C, in order to make graph scheduling applicable, there is work to be done in generating coarse-grain task graphs [2].

The current algorithms still have restrictions. For example, we have not considered processor distances. It has been shown (e.g. [5]) in the modern processor architectures, the distance factor does not affect the cost of communication significantly because of the wormhole routing technology. However, this routing scheme may lead to communication contention if network traffic is heavy. We have not considered the overhead of memory accessing and caching effect [24]. Another interesting future work is to consider non-uniform iteration intervals as proposed in [4].

**Acknowledgment**

This work was supported by NSF RIA CCR-9409695 and INT-9533361. We thank Apostolos Gerasoulis 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, Pedro Diniz for his help in algorithm implementation, and the anonymous referees for their valuable comments.

## References

[1] A. Beguelin, J.J. Dongarra, G.A. Geist, R. Manchek, V.S. Sunderam, Graphical development tools for network-based concurrent supercomputing, *Proc. of Supercomputing '91*, IEEE, 1991, pp.435-444.

[2] M. Cosnard and M. Loi, Automatic Task Graph Generation Techniques, in *Parallel Processing Letters*, Vol. 5, No. 4, pp. 527-538, Dec, 1995.

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

[4] V. Donaldson and J. Ferrante, Determining Asynchronous Pipeline Execution Times, Technical Report, Department of Computer Science, UCSD, 1995, in Proceedings of IEEE IPPS'96 (10th Interational Parallel Processing Symposium).

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

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

[7] C. Fu and T. Yang, Run-time Compilation for Parallel Sparse Matrix Computations, in *Proc. of the 10th ACM International Conference on Supercomputing,* Philadelphia, pp. 237-244, May, 1996.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] P. Newton and J. Browne, The CODE 2.0 graphical parallel programming language, *Proc. of 6th ACM Inter. Conf. on Supercomputing,* Washington D.C., July, 1992, pp. 167-177.

[14] D. O'Hallaron, The ASSIGN parallel program generator, Tech. Report, CS Dept, CMU, CMU-CS-91-141, 1991.

[15] 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.

[16] 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.

[17] 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.

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

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

[22] K. P. Wang, J. C. Bruch, Jr., An efficient iterative parallel finite element computational method, in *the Mathematics of finite Elements and Applications.*, J. R.Whiteman (Eds.), 1994.

[23] P. Wang, Y.T. Yao, M.P. Tulin, An Efficient Numerical Tank for Nonlinear Water Waves based on the Multi-Subdomain Approach with BEM, in *Int. J. Numerical Methods in Fluids*, Vol. 20, 383-392, 1995.

[24] 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.

[25] 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.

[26] 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.

[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, Mapping iterative task graphs on distributed-memory machines. in *Proc. of 24th Inter. Conference on Parallel Processing*, Aug. 1995, Vol II, pp. 151-158.

# A List of Symbols

$T_i^k$ – The instance of task $T_i$ at iteration $k$.
$N$ – The number of iterations.
$d_{i,j}$ – Dependence distance from task $T_i$ to $T_j$.
$c_{x,y}$ – Communication cost from task $T_i$ to $T_j$.
$\tau_x$ – The computation cost of task $T_x$.
$\tau^{max}$ – The maximum task weight in $G$.
$p$ – The number of available processors.
$\beta_p$ – The iteration interval.
$\alpha_{p,i}$ – The startup delay for task $T_i$ with $p$ processors.
$ST(T_i^k)$ – The starting time of task instance $T_i^k$, modeled as $\alpha_{p,i} + \beta_p(k-1)$.
$\beta^*(G)$ – The smallest iteration interval for $\beta_p$ with any $p$ when the communication cost is zero.
$\beta$ – An approximation of $\beta^*(G)$.
$\epsilon$ – The desired accuracy used in the binary searching algorithm.
$\alpha_y$ – A non-negative constant associated with task $T_y$, determined when $\beta$ is computed.
$Seq(G)$ – It is $\sum_{i=1}^{v} \tau_i$.
$g(G)$ – The granularity of the ITG $G$.
$B$ – The asymptotic bound compared to the optimum.
$f$ – The unfolding factor.
$S1$ – A constant equal to $1 - 1/p + 1/g(G)$, used for computing the bound.
$L1, L2, L3, L4$ – Constants used for computing the unfolding factor.
$DIV(x, y)$ – The largest integer that is less than or equal to $x/y$, i.e. $\lfloor x/y \rfloor$.
$MOD(x, y) = x - DIV(x, y) * y$.
$G^f$ – The unfolded graph.
$\beta_f$ – An approximation of $\beta^*(G^f)$.
$K(G^f)$ – The kernel DAG of ITG $G^f$.
$CP(R)$ – The critical path length in DAG $R$.

$\gamma_i$ – The starting time of $T_i$ derived by the DAG scheduling or the load balancing algorithm.
$H_i$ – It is $DIV(\alpha_i, \beta_f)$.
$PT(G)$ – The length of the schedule by an ITG or DAG mapping algorithm for $G$.
$PT_{opt}(G)$ – The length of the optimum schedule.
$PT(S, G)$ – The parallel time produced by the code in Fig.5 following the schedule $S$ produced by our mapping algorithm for ITG $G$.
$G^r$ – It is the same as the given ITG $G$ except it is weighted using run-time weights.

# B Proofs of the Theorems

**Proof for Theorem 1.**

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 $T_z \in Parents(T_x)$, define $arrival(T_z, T_x) = \gamma_z + \tau_z + c_{z,x}$. Then

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

Let $T_y$ be the most dominating parent for $T_x$ such that $tlevel(T_x) = arrival(T_y, T_x)$. 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

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

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 $\gamma_x$. 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

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

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

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

**Proof for Theorem 2.**

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

**Proof for Theorem 3**

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,

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

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

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

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,

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

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

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

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

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

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

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:

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

■

**Proof for Theorem 4.**

(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

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

**Proof for Theorem 5.**

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, $LPT \le max(\frac{2*Seq(G^f)}{p+1}, \tau^{max})$. Thus for the ITG schedule:

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

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

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

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

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

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

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

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

i.e.

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

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

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

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

■

**Proof for Theorem 6.**

Notice that $L_j \ge \alpha_{i,p}$, thus $o_i = \lceil \frac{L_j - \alpha_{i,p}}{\beta_p} \rceil + 1 \ge 1$. 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

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

Thus

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

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. ■

**Proof for Theorem 7.**

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:

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

■

## List of Figures



## List of Tables