Performance Prediction in Symbolic Scheduling of Partitioned Programs with Weight Variation

Tao Yang and Oscar H. Ibarra
Department of Computer Science
University of California
Santa Barbara, CA 93106
{tyang,ibarra}@cs.ucsb.edu

Abstract

In this paper we consider the symbolic scheduling of partitioned loop programs which are modeled as iterative task graphs (ITGs). Each task in such a graph is coarse grained and contains a large chunk of computations. The weights of computation and communication vary from one iteration to another depending on the index value of the loop. The goal of scheduling such graphs is to incorporate the symbolic variables in weight functions and loop bounds and provide an asymptotically optimal schedule and predict its performance accurately. We provide a lower bound for optimal scheduling when weights of iterative task graphs change monotonically in the course of iterations and there is a sufficient number of processors. We present a technique that devises a valid symbolic schedule without searching all task instances and examine the asymptotic performance of this schedule compared to an optimal solution. Finally, we present case studies and experimental results on nCUBE-2 to verify our solutions.

1 Introduction

Parallel programming on distributed memory machines requires coarse grain partitioning since there is a large startup overhead in message transmission [9, 19, 21]. A partitioned program usually contains a set of tasks with large chunk of computation. Finding a mapping of weighted coarse-grain dependence task graphs onto target architectures with the shortest parallel time is important for studying the impact of partitioning and generating efficient code. The task weights and communication cost play an important role in schedule optimization, as demonstrated in task scheduling based on directed acyclic graph (DAG) [3, 13, 17, 25]. The difficulty in scheduling such coarse grain tasks produced from a loop program is that dependence structures and weights may contain symbolic information [5].

In this paper we consider the scheduling of partitioned loop programs which are modeled as iterative task graphs (ITGs) and many scientific computations can be represented by ITGs. An ITG represents a sequence of parallel task computation. Each iteration contains the execution of a set of tasks and there exists task dependence within an iteration and between iterations. Because coarse-grained tasks may contain symbolic information, their weights could vary from one iteration to another. The symbolic information considered in this paper is the loop bound and the loop index in weight functions.

The previous work on scheduling task computation has used the macro-data flow model to exploit task parallelism[17]. Many algorithms for mapping weighted DAGs in the presence of communication have been proposed and used in compiler tools, e.g. [15, 17, 25]. The difficulty in utilizing this macro-data flow model for scheduling loop programs is that task graph representation in [17, 25] does not precisely capture the symbolic information arising in loop programs. The problem of loop parallelization has been addressed extensively in the literature [1, 6, 14, 19, 21]. One issue is that many program transformation techniques for exploiting loop parallelism are based on heuristics that achieve local optimum. Graph scheduling techniques that examine the relationship between different program segments are attractive if they can guarantee global performance competitive to the optimal solution.

If an innermost loop body contains a sequence of statements, pipelining techniques [1, 2, 12, 20, 16, 24] have been proposed to uncover parallelism across iterations for VLIW and superscalar architectures. They use computational weight information for load balancing, determine a schedule and predict the performance of parallelization. However, they assume that the computation weights do not change during the course of the iterations and also they have not included the communication cost in schedule determination because this is not needed for the architectures at which they are targeted. For message-passing architectures, communication could be asynchronous and coarse grain partitioning is commonly used to produce tasks. Techniques for program partitioning and weighted coarse grain graph generation are described in [5, 19]. Scheduling such graphs is hard since graph representation involves symbolic information on loop bounds and task weights, and task weights may vary during iterations.

In [26, 27], we have considered the ITG scheduling algorithms and applications with *constant* task and communication weights. In this paper, we consider the ITG scheduling with weight variation and communication delay. We take a two-stage scheduling approach: 1) Cluster tasks into a set of threads. The goal of mapping is to derive a clustering such that the parallel time is minimized when there is a sufficient number of processors. This paper is focused on this clustering stage. 2) Map task clusters (threads) to p physical processors for multi-threaded computation. The idea of such a multi-stage approach has been used in SISAL compilation and DAG scheduling [17, 22, 25]. We use the ideas of the macro dataflow task model [9, 17] and pipelining techniques, and examine how these ideas can be combined and extended for the symbolic scheduling of iterative task computation without searching the entire iteration space. We develop a clustering algorithm and predict its performance for mapping iterative task graphs when there is a sufficient number of processors, and task weights change monotonically during the course of iteration. Since mapping iterative task graphs is NP-hard, we will be comparing the performance of this heuristic schedule with the optimal solution obtained by searching the entire iteration space. Using the result of performance analysis, we show how program partitioning and task granularity affect the performance of parallelization.

In Section 2, we give problem definitions and assumptions. In Section 3, we present a scheduling algorithm that takes into account communication overhead and weight variations. We also conduct a theoretical analysis on its correctness and performance, and study the impact of partitioning. In Section 4, we give experimental results that validate the correctness of the analytic results. The appendix lists a procedure used in our algorithm.

2 Problem Definitions and Assumptions

We assume that each iteration of an ITG contains v tasks: $T_1, T_2, \cdots, T_v$. Each task T_i can be a statement or a chunk of computation obtained by partitioning techniques [14, 19, 21]. We assume that the information on task dependence and weights are available using techniques described in [5, 18]. These v tasks will be executed N times with iteration numbers $0, 1, 2, \cdots, N-1$. If each task is a statement, the instruction level scheduling has been studied in the context of software pipelining proposed for VLIW and superscalar architectures [1, 12, 20]. For message-passing architectures such as the Intel Paragon, CM-5, nCUBE-2 and Meiko CS-2, fine grain partitioning is not appropriate due to high communication startup time. Thus we assume that a task usually contains a large chunk of computation and its weight may change in the course of iterations (e.g. a task contains a loop, the loop bound depends on the iteration number. An example is discussed later). We will show how a schedule can be derived by considering weight variations and communication overhead, and how parallel performance can be estimated and the impact of partitioning can be analyzed.

Some definitions on dependence and weights are listed below.

- We use the *macro-dataflow task model* for computation execution [17]. A task receives all input before starting execution, executes to completion without interruption, and immediately sends the output to all successor tasks.

- Let the instance of task T_i at iteration k be T_i^k $(0 \leq k \leq N-1)$. The iteration space of this computation (called $E(G, N)$) contains all $N*v$ task instances T_i^k $(1 \leq i \leq v, 0 \leq k \leq N-1)$. Notice that N is a symbolic variable and v is a constant known at compile-time.

- Let (T_i, T_j) be a dependence edge [1] from T_i to T_j, the iteration delay is a distance function $d_{i,j}$. Namely, task instance $T_j^{k+d_{i,j}}$ depends on T_i^k. We assume that all distances $d_{i,j}$ are non-negative integers [2].

  Let $succ(T_i)$ be the set of tasks immediately depending on T_i in G. The dependence of tasks within and between iterations constitutes a directed dependence graph G.

[1] Here we consider true data dependence only. Output and anti-dependence can be removed by renaming techniques. Control dependence is not considered in this paper.

[2] Some dependence graphs may contain imprecise dependence direction information such as '+' instead of precise distance values [21]. Our result can be extended for this case easily. For example, reduce '+' as 1.

- The computation weight function of task T_i^k is $\tau_i(k)$, which is the time that T_i^k takes to execute. The communication cost from T_i^k to $T_j^{k+d_{i,j}}$ is $c_{i,j}(k)$. $c_{i,j}(k)$ is 0 if these two tasks are assigned to the same processor. We assume that the cost of fetching data items from a local memory or communication buffer is negligible compared to the cost of transferring a message from one processor to another. Inter-processor communication costs include message-passing startup time. Usually the communication delay is modeled as communication startup time+ transmission speed $\times$ size of message. The physical distance between processors may affect communication delay; however, in a wormhole routing network the cost is not so sensitive to processor distances [3]. Kumar et.al. [10] discussed the formula for communication delay in a store-forward or wormhole routing network. In this paper, we assume that the worst-case cost is used in estimating task communication delay. Cosnard and Loi [5], Sarkar [18] discussed methods for estimating communication and computation cost. We will discuss the assumptions on weight functions later.

  Assigning communication weights zero indicates that tasks communicate through local memory. This is related to the concept of "exploiting data locality" used in the literature. In task computation model [17], exploiting data locality means to localize data communication between tasks by assigning them in the same processor and then tasks could exchange data through a local memory to avoid high-cost inter-processor communication. When we determine a schedule and estimate its performance, we have included the impact of "data locality".

- $ST(T_i^k)$ is the starting time of task T_i^k in a schedule. $CT(T_i^k)$ is the completion time of task T_i^k, defined as $ST(T_i^k) + \tau_i(k)$. $PA(T_i^k)$ is the processor location of T_i^k.

  A scheduling algorithm will determine $PA(T_i^k)$ and $ST(T_i^k)$. A valid schedule must satisfy the following constraint:

  - **Resource constraint (R1)**: $1 \leq PA(T_i^k) \leq p$ where p is the given number of processors. And if $PA(T_i^k) = PA(T_j^t)$ for two different task instances T_i^k and T_j^t in $E(G, N)$, then $ST(T_i^k) - CT(T_j^t) \geq 0$ or $CT(T_i^k) - ST(T_j^t) \leq 0$.
  - **Dependence constraint (R2)**: If a dependence delay from T_i to T_j is an integer d, then for any two tasks T_j^{k+d} and T_i^k in $E(G, N)$, $ST(T_j^{k+d}) - ST(T_i^k) \geq \tau_i(k) + c_{i,j}(k)$.

For example, a partitioned BLAS-3 program for solving a lower triangular system $Ax = b$ is listed in Fig. 1. A is a lower triangular matrix of size $n \times n$, which is partitioned into $N \times N$ submatrices where $n = N \times h$. Each submatrix $A_{i,j}$ is of dimension $h \times h$. The column vectors x and b of dimension n are partitioned into N subvectors, $X_1, X_2, \cdots X_N$ and $B_1, B_2, \cdots, B_N$. Fig. 2 is an ITG for this program. The dependence from S to T_q broadcasts a partial solution vector X_j. The dependence from T_q to S is to send one subvector B_{j+1} to S. The weight information of this task graph is discussed in Section 4. Noted that the dependence is actually overestimated since only

[3] If network load is high, communication delay is still sensitive to distances in a wormhole-routing scheme due to network contention [4]. Contention is not considered in this paper and we assume that network communication load is low during coarse grained computations.

for $j = 0$ **to** $N-1$

  $\boxed{S^j\text{: Solve } A_{j+1,j+1}X_{j+1} = B_{j+1}.}$

  for $q = 1$ **to** P

  T_q^j

    for $i = \lceil \frac{j+1-q}{P} \rceil * P + q$ to N step P

      $B_i = B_i - A_{i,j+1} * X_{j+1}$.

    endfor

  endfor

endfor

Figure 1: A partitioned program for solving a triangular system.

one edge for each iteration is needed while in Fig. 2 there are P edges from $T_1, T_2, \cdots, T_P$ to S. For this case, it is not easy to summarize inter-iteration dependence accurately. In fact, current dependence analysis algorithms [21] may overestimate dependence since accurate estimation may be intractable; however, overestimation does not affect the correctness of parallel execution.

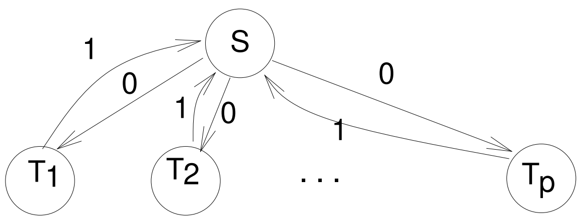

Figure 2: An ITG solving a triangular system.

Fig. 3 is another example of a partitioned loop program whose loop body contains three tasks. Fig. 4(a) is the ITG of this program. (b) is the expanded graph of this ITG representing the iteration space of this program. (c) is a schedule of this program on 3 processors. We assume that the communication delay between tasks assigned in different processors is 1 time unit. The task weight functions are $\tau_x(k) = k+1$, $\tau_y(k) = 1.5k+1$ and $\tau_z(k) = 2k+1$.

for k = 0, N-1

  $\boxed{T_x^k: \quad a[k] = f_1(b[k-2])}$;

  $\boxed{T_y^k : b[k] = f_2(b[k-1], a[k], c[k-2])}$;

  $\boxed{T_z^k : c[k] = f_3(b[k])}$;

endfor

Figure 3: A partitioned program.

If weights change irregularly during iterations, it is difficult to obtain a valid uniform schedule for each task $ST(T_i^k)$ without examining the entire expanded graph of G. In practice, weight functions

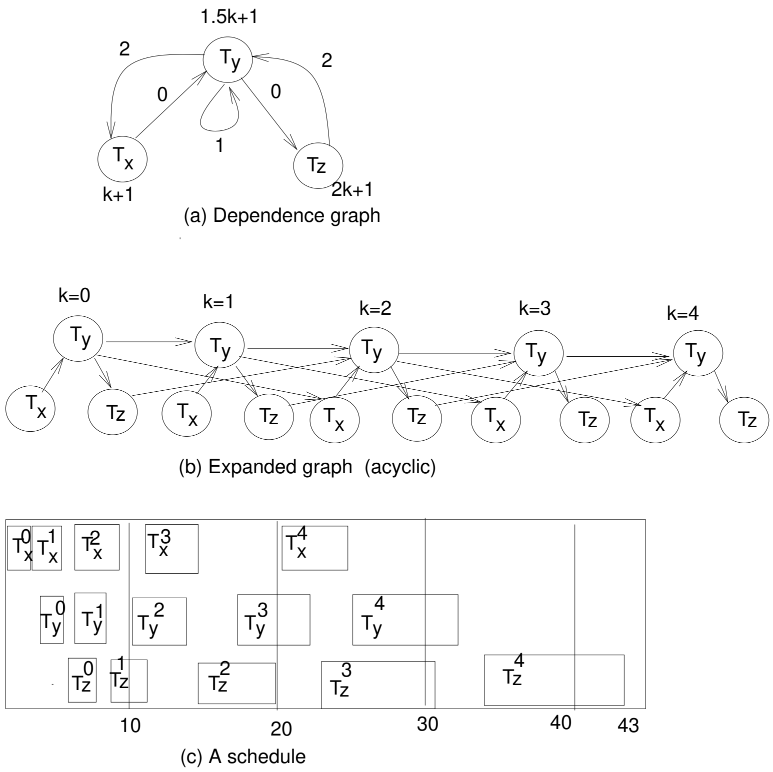

Figure 4: (a) The ITG G for Fig. 4. (b) The expanded graph $E(G, 5)$. (c) A valid schedule.

usually possess certain regularity depending on iteration numbers [5]. In this paper we first consider weights monotonically increase or remain the same from one iteration to another. More specifically,

- $\tau_i(k) = a_i k + b_i$ where $a_i \geq 0$ and $b_i \geq 0$.
- $c_{i,j}(k) = u_{i,j} k + v_{i,j}$ where $u_{i,j} \geq 0$ and $v_{i,j} \geq 0$.

We consider the case when the weights decrease monotonically in Section 3.5.

The following terms will be used in the rest of this paper. Let

$$A = \sum_{i=1}^{v} a_i, \quad B = \sum_{i=1}^{v} b_i, \; a^{max} = \max_{i=1}^{v} a_i, \; b^{max} = \max_{i=1}^{v} b_i, \; a^{avg} = \frac{A}{v}.$$

Let C be a cycle in an ITG G, without loss of generality, assume that $C = T_1, T_2 \cdots T_j, T_1$. Let

$$A1_C = \sum_{i=1}^{j} a_i, \quad B1_C = \sum_{i=1}^{j} b_i$$

$$A2_C = a_j + u_{j,1} + \sum_{i=1}^{j-1} (a_i + u_{i,i+1}), \quad B2_C = b_j + v_{j,1} + \sum_{i=1}^{j-1} (b_i + v_{i,i+1}),$$

$$A2 = \max_{\text{all cycles C}} A2_C, \quad B2 = \max_{\text{all cycles C}} B2_C.$$

3 Scheduling with nonzero communication and weight variation

The problem of task graph scheduling with nonzero communication and a sufficient number of processors is NP-complete [17]. We first present a bound on the performance of any optimal schedule derived by searching the entire iteration space. This result is useful when we compare it with our heuristic schedule to determine the performance difference. We will give a heuristic schedule for an ITG G with v tasks when there are v processors. Then we increase the number of processors to obtain a shorter schedule.

3.1 Performance Bounds

Given p processors and a DAG, a well-known lower bound of parallel time for executing this graph is $max(\frac{Seq}{p}, L_{crit})$ where Seq is the summation of the total computation weights and L_{crit} is the length of the critical path (the longest path) in this DAG. We examine performance bounds for scheduling below.

Theorem 1 *Given an ITG G and p processors, let C be a cycle in G, D_C is the summation of dependence distances in this cycle. Let $M_C = \lfloor \frac{N-1}{D_C} \rfloor$.*

$$PT_p \geq \frac{0.5 * A * N(N-1) + BN}{p} \quad \textit{and} \quad PT_p \geq \max_{\textit{all cycles } C} (M_C * B1_C + 0.5 M_C(M_C - 1) * A1_C * D_C).$$

Proof: The total computation weight of $E(G, N)$ is:

$$Seq = \sum_{k=0}^{N-1} \sum_{i=1}^{v} (a_i * k + b_i) = A * N(N-1)/2 + B * N.$$

Thus a performance bound with respect to the computation load is

$$PT_p \geq \frac{A * N(N-1)/2 + B * N}{p}.$$

Next we examine a dependence path in $E(G, N)$. Assume that there is a dependence cycle C in G, $C = T_1, T_2, \cdots T_j, T_1$. Let $d_i = d_{i,i+1}$ for $i = 1, \cdots j-1$ and $d_j = d_{j,1}$. The summation of dependence distance is

$$D_C = d_{j,1} + \sum_{i=1}^{j-1} d_{i,i+1} = \sum_{i=1}^{j} d_i.$$

Then a path in $E(G, N)$ is

$$T_1^0, T_2^{d_1}, T_3^{d_1+d_2}, \cdots, T_j^{d_1+d_2+\cdots+d_{j-1}}, T_1^{D_C}.$$

Then another path is

$$T_1^{D_C}, T_2^{D_C+d_1}, T_3^{D_C+d_1+d_2}, \cdots, T_j^{D_C+d_1+d_2+\cdots+d_{j-1}}, T_1^{2D_C}.$$

Then we can start more paths from $T_1^{2D_C}$, $T_1^{3D_C}$ and so on. The length of the q-th path ($q = 0, 1, 2, \cdots$) is

$$L(q) = \sum_{i=1}^{j}(a_i(q * D_C + \sum_{t=1}^{i} d_t)) + b_i \geq q * A1_C * D_C + B1_C.$$

The total number of such paths are $M_C = \lfloor \frac{N-1}{D_C} \rfloor$, i.e. $0 \leq q < M_C$. Notice that these M_C paths constitute one path starting from T_1^0 to $T_1^{M_C * D_C}$. The length of the critical path in $E(G, N)$:

$$L_{crit} \geq \sum_{q=0}^{M_C-1} L(q) \geq M_C * B1_C + M_C(M_C - 1)/2 * A1_C * D_C.$$

■

Example: In Fig. 4(a) there is a dependence cycle T_y, T_z, T_y with $D_C = 2$. A dependence path in $E(G, N)$ expanded from this cycle is $T_y^0, T_z^0, T_y^2, T_z^2, T_y^2, T_z^2, T_y^4$ when $N = 5$ and $M_C = 4/2 = 2$.

When $a^{avg} = 0$, the bound is reduced to $L_{crit} \geq M_C * B1_C = \lfloor \frac{N-1}{D_C} \rfloor B1_C$ which is a lower bound for any cyclic scheduling [12].

3.2 Scheduling on v processors

We first present a schedule when the number of processors is v (equal to the number of tasks in an ITG). We will provide an analysis of its asymptotic performance. Then we add more processors if necessary to improve the scheduling performance.

Periodic scheduling proposed in software pipelining [1] assumes that task weights are fixed and if task T_i^k is executed at time t then task T_i^{k+1} is executed at time $t + c$ where c is a constant. This model does not apply to our case because task weights change during iterations. Thus we will use a non-linear function to model the starting time of task T_x^k as:

$$ST(T_x^k) = \alpha_x + (\beta_x + r)k + \frac{H}{2}k^2. \quad (1)$$

We assign the instances of task T_i to processor i and we will show the correctness and performance of this schedule model in Theorems 2 and 3. For such a model, we need to determine following coefficients: two nonnegative values α_x and β_x corresponding to each task, and two nonnegative coefficients H and r. These coefficients may not be integral and are computed in two steps as described below.

Step 1 For each dependence edge (T_x, T_y) $(T_x \neq T_y)$ we set up the following inequalities called $F1$:

$$\beta_y - \beta_x + dH_{x,y} \geq a_x + u_{x,y}.$$

We call a procedure $LPATH(F1, v, H, a^{max})$(see the appendix for the details) to determine values of β_x $(1 \leq x \leq v)$ and H in a complexity of $O(ve \log \frac{A2}{\epsilon})$ satisfying the above inequalities and also $H \geq a^{max}$. ϵ is the desired accuracy in finding the minimum value H.

Step 2 For each dependence edge (T_x, T_y) $(T_x \neq T_y)$ in G, we setup the following inequalities called $F2$:

$$\alpha_y - \alpha_x + r * d_{x,y} \geq b_x + v_{x,y} - \beta_y d_{x,y} - \frac{Hd^2}{2}.$$

Let $r_1 = max(0, b_x - \beta_x)$. We call $LPATH(F2, v, r, r_1)$ in a complexity of $O(ve \log \frac{B2}{\epsilon})$ to determine r and α_x $(1 \leq x \leq v)$ that satisfy the above inequalities and $r \geq r_1$.

The total complexity of this algorithm is $O(ve \log \frac{A2*B2}{\epsilon^2})$. Notice that if there is no inter-task dependence and there exists only self-dependence, we let the coefficients of the starting time function $ST(T_x^k)$ be $\alpha_x = \beta_x = 0$, $r = b^{max}, H = a^{max}$.

Theorem 2 *The schedule determined by the above algorithm is legal.*

Proof:

Condition R1. Since all and only instances of the task T_x are executed in processor x, the following condition is sufficient to make R1 true:

$$ST(T_x^{k+1}) - ST(T_x^k) \geq \tau_x(k) = a_x k + b_x. \tag{2}$$

That is

$$ST(T_x^{k+1}) - ST(T_x^k) = \beta_x + r + Hk \geq a_x k + b_x.$$

Since in Step 1, $H \geq a_x$ and in Step 2, $r \geq r_1 \geq max(0, b_x - \beta_x)$, Inequality (2) holds true for every processor.

Condition R2. We first show that all self-dependence is satisfied automatically. Notice that instances T_x^k of task T_x are executed in the same processor in an order of $k = 0, 1 \cdots$. When task T_x in G has a self dependence, for any integer d ($d \geq 1$), $ST(T_x^{k+d}) - CT(T_x^k) \geq \tau_x(k)$.

Next we examine non-self dependence. For each edge (T_x, T_y) in G $(T_x \neq T_y)$, let $d = d_{x,y}$. We need to show:

$$ST(T_y^{k+d}) - ST(T_x^k) \geq \tau_x(k) + c_{x,y}(k) = a_x k + b_x + u_{x,y} k + v_{x,y}. \tag{3}$$

Using expression (1), $ST(T_y^{k+d}) - ST(T_x^k)$ can be expanded as:

$$(\beta_y - \beta_x + dH)k + \alpha_y - \alpha_x + (\beta_y + r)d + \frac{Hd^2}{2}.$$

Notice that in Step 1, β_x $(1 \le x \le v)$ and H are solved such that the following condition is true:

$$\beta_y - \beta_x + dH \ge a_x + u_{x,y}$$

and in Step 2:

$$\alpha_y - \alpha_x + (\beta_y + r)d + \frac{Hd^2}{2} \ge b_x + v_{x,y}.$$

Thus inequality (3) is true for every dependence edge.

$\blacksquare$

Example: For Fig. 4(a), we have the following inequalities $F1$ in Step 1:

$$\begin{aligned} \beta_y - \beta_x \ge 1, &\quad \beta_x - \beta_y + 2H \ge 1.5, \\ \beta_z - \beta_y \ge 1.5, &\quad \beta_y - \beta_z + 2H \ge 2. \end{aligned}$$

Notice that $v = 3$, all communication weights are 1 and $a^{max} = 2$. The results of procedure $LPATH(F1, 3, H, 2)$ are $H = 2$, $\beta_x = 0$, $\beta_y = 1$ and $\beta_z = 2.5$.

In Step 2, notice that all $b_i = 1$ and $v_{i,j} = 1$. We have the following inequalities $F2$:

$$\begin{aligned} \alpha_y - \alpha_x \ge 1 + 1, &\quad \alpha_x - \alpha_y + 2r \ge 1 + 1 - 0 * 2 - 2^2, \\ \alpha_z - \alpha_y \ge 1 + 1, &\quad \alpha_y - \alpha_z + 2r \ge 1 + 1 - 1 * 2 - 2^2. \end{aligned}$$

$r_1 = max(0, b_x - \beta_x) = 1$ and procedure $LPATH(F2, 3, r, 1)$ produces a solution $r = 1$, $\alpha_x = 0$, $\alpha_y = 2$ and $\alpha_z = 4$. We have the following schedule:

$$\begin{aligned} ST(T_x^k) &= k + k^2, \\ ST(T_y^k) &= 2 + 2k + k^2, \\ ST(T_z^k) &= 4 + 3.5k + k^2. \end{aligned}$$

Thus the parallel time is $PT = ST(T_z^{N-1}) + \tau_z(N-1) = 4 + 3.5(N-1) + (N-1)^2 + 2(N-1) + 1$. The sequential time

$$Seq = \sum_{k=0}^{N-1} (k + 1 + 1.5k + 1 + 2k + 1) = 4.5(N-1)N/2 + 3(N-1).$$

When N is large, the speedup is approximately equal to $4.5/2 = 2.25$.

3.3 Performance Analysis

We estimate the asymptotic speedup of the above schedule and compare its performance with that of an optimal schedule using a sufficient number of processors. Notice that a program partitioning results in a task graph which possesses certain characteristics: a^{avg}, a^{max}, dependence cycles, and task granularity. We will show how these characteristics affect the scheduling performance and examine what kinds of partitioning are preferred.

The previous work [9] has shown that the performance of DAG scheduling depends on the granularity of DAGs. Our result also indicates this correlation but works for task graphs with cycles. A definition and formal analysis on granularity of a DAG is given in [9] and we use a slightly different definition to quantify the granularity of an ITG G as:

$$g = \min_{T_x \in G}\{\min_{T_y \in succ(T_x)} \frac{a_x}{u_{x,y}}\}.$$

Theorem 3 *If a^{max} and b^{max} are bounded constants, let*

$$Q(G) = \max_{all\ cycles\ C} \frac{A1_C}{D_C}.$$

Then the asymptotic speedup lower bound is:

$$\lim_{N\to\infty} Inf\{\frac{Seq}{PT}\} = \frac{a^{avg} * v}{max(a^{max}, \epsilon + (1+1/g)Q(G))}.$$

The asymptotic performance upper bound is:

$$\lim_{N\to\infty} Sup\{\frac{PT}{PT_{opt}}\} = \max(\frac{a^{max}}{Q(G)}, 1 + 1/g + \frac{\epsilon}{Q(G)}).$$

Proof: Define

$$Q2(G) = \max_{\text{all cycles C}} \frac{A2_C}{D_C}.$$

From step 1,

$$H = \max(a^{max}, \epsilon + Q2(G)).$$

Since $A2_C \leq A1_C + 1/gA1_C$,

$$H \leq max(a^{max}, \epsilon + (1+1/g)Q(G)).$$

Since

$$PT = \max_{T_x \in G}\{\alpha_x + (\beta_x + r)(N-1) + \frac{H}{2}(N-1)^2 + \tau_x(N-1)\},$$

when N is very large,

$$PT = \frac{H}{2}N^2 + t(N)$$

where $t(N)$ is a function in an order of N. That is, $\lim_{N\to\infty}\frac{t(N)}{N^2}=0$.

Since $Seq = A/2 * N^2 + t1(N)$ where $t1$ is a function in an order of N,

$$\lim_{N\to\infty}\frac{Seq}{PT}=\frac{A}{H}=\frac{a^{avg}*v}{max(a^{max},\epsilon+(1+1/g)Q(G))}.$$

Next we examine the performance bound.

$$\frac{PT}{PT_{opt}}\leq 0.5N^2\max(\frac{a^{max}}{PT_{opt}},\frac{Q(G)}{PT_{opt}}(1+1/g)+\frac{\epsilon}{T_{opt}})+\frac{t(N)}{T_{opt}}.$$

Recall that $M_C=\lfloor\frac{N-1}{D_C}\rfloor$. By Theorem 1,

$$PT_{opt}\geq\max_C M_C(M_C-1)/2A_C*D_C=0.5N^2Q(G)+t3(N)$$

where $t2$ and $t3$ are functions in an order of N.

$$\lim_{N\to\infty}\frac{PT_{opt}}{N^2}\geq 0.5Q(G).$$

Thus

$$\lim_{N\to\infty}\frac{PT}{PT_{opt}}\leq\max(\frac{a^{max}}{Q(G)},1+1/g+\frac{\epsilon}{Q(G)}).$$

■

Example: For Fig.4, we have

$$Q(G)=3.5/2=1.75,\quad a^{avg}=4.5/3=1.5,\quad a^{max}=2,\quad 1/g=0.$$

Notice that the communication cost is 1, thus $g=\infty$. Then the scheduling performance is bounded as:

$$\frac{PT}{PT_{opt}}\leq max(\frac{2}{1.75},1+\frac{\epsilon}{1.75}).$$

For a sufficiently small ϵ, we have: $\frac{PT}{PT_{opt}}\leq 1.143$. The asymptotic speedup is 4.5/2=2.25.

We also have the analysis result when $a^{avg}=0$. We will not present it in this paper.

From the above analysis, we can see that if $a^{max}\leq(1+1/g)Q(G)$, the heuristic solution is competitive to the optimal solution in a ratio of about $1+1/g$. For coarse grain parallelism with $g\geq 1$ and large N, this heuristic can reach 50% of the optimum. The achievable speedup is above $\frac{a^{avg}*v}{(1+1/g)Q(G)}$. Using this formula, we can evaluate if a partitioning for a program can result in a task graph with a good parallel performance.

If there is no cycle in G, $Q(G)=0$, then the above performance bound goes to infinity, indicating that the optimal solution may use an infinite number of processors as N goes to infinity. And the speedup for using v processors is bounded as $a^{avg} * v / a^{max}$. If a partitioning is chosen such that a^{avg} is close to a^{max}, a good efficiency can be obtained.

If $a^{max} > (1+1/g)Q(G)$, then the competitive ratio is about $a^{max}/Q(G)$ and the speedup is bounded by $a^{avg} * v / a^{max}$. Thus if a^{max} is very large, the scheduling performance deteriorates. An explanation for this is that processors that execute the tasks T_x with $a_x = a^{max}$ are overloaded. Next we examine how to use more processors to improve the performance.

3.4 Adding more processors

We need to compute how many processors are needed if $a^{max} > (1 + 1/g)Q(G)$. We use graph unfolding techniques proposed in software pipelining [1] to increase the number of tasks within the loop body and use more processors to execute these tasks.

For example, we unfold Fig 4(a) by 2 (i.e. unroll the loop of Fig 3 by 2). The unrolled program is shown in Fig. 5. For the new ITG, we can obtain a schedule using 6 processors with $PT = \frac{7}{8}N^2 + O(N)$ and $\lim_{N\to\infty} \frac{PT}{PT_{opt}} = 1$.

for k $= 0, \lfloor \frac{N-1}{2} \rfloor$
  $T^k_{x,1} : a[2k] = f_1(b[2k-2]);$
  $T^k_{y,1} : b[2k] = f_2(b[2k-1], a[2k], c[2k-2]);$
  $T^k_{z,1} : c[2k] = f_3(b[2k]);$
  $T^k_{x,2} : a[2k+1] = f_1(b[2k-1]);$
  $T^k_{y,2} : b[2k+1] = f_2(b[2k], a[2k+1], c[2k-1]);$
  $T^k_{z,2} : c[2k+1] = f_3(b[2k+1]);$
endfor
for k $= 2 * \lfloor \frac{N-1}{2} \rfloor$ to N
  $T^k_x : a[k] = f_1(b[k-2]);$
  $T^k_y : b[k] = f_2(b[k-1], a[k], c[k-2]);$
  $T^k_z : c[k] = f_3(b[k]);$
endfor

Figure 5: An example of unrolled loop.

In general if we unfold an ITG by f times, the number of iterations is reduced from N to $\lfloor N/f \rfloor$ and the number of tasks in the ITG is increased by f times. The unfolded graph is called G^f.

For G^f, we apply the two-step method in Section 3.2 using fv processors. We can show that a^{max} of G^f is $f * a^{max}$, $Q(G^f) = f^2 Q(G)$,

$$\lim_{N\to\infty} Inf\{\frac{Seq}{PT}\} = \frac{a^{avg} * v}{max(a^{max}/f, \epsilon + (1+1/g)Q(G))}, \text{ and}$$

$$\lim_{N\to\infty} Sup\{\frac{PT}{PT_{opt}}\} = \max(\frac{a^{max}/f}{Q(G)}, 1 + 1/g + \frac{\epsilon}{Q(G)}).$$

Assume that ϵ is very small. Thus set

$$f = \lceil \frac{a^{max}}{(1+1/g) * Q(G)} \rceil.$$

For such an unfolding factor f, the heuristic solution is competitive to the optimal solution in a ratio of about $1 + 1/g$. The achievable speedup bound is about $\frac{a^{avg} * v}{(1+1/g)Q(G)} \approx a^{avg} * vf/a^{max}$.

3.5 Monotonically decreasing weight functions

In this section, we discuss the schedule determination when the weight functions monotonically decrease. We assume that the weight functions of G are:

- $\tau_i(k) = -a_i k + b_i$ where $a_i \geq 0$ and $b_i \geq a_i * (N-1)$ so that there are no negative weights.
- $c_{i,j}(k) = -u_{i,j}k + v_{i,j}$ where $u_{i,j} \geq 0$ and $v_{i,j} \geq u_{i,j}(N-1)$.

We can design a solution for this case as follows:

Step 1. We schedule this graph in a backward manner, in the sense that we first reverse the dependence edge of this graph, the new graph is G^r. We construct the nonnegative weight functions of this new graph as:

- $\tau^r_x(k) = \tau_x(N-k-1) = -a_x(N-k-1) + b_x = a_x * k + b_x - a_x * (N-1)$.
- $c^r_{y,x}(k) = c_{x,y}(N-k-1-d) = -u_{x,y}(N-k-1-d) + v_{x,y} = u_{x,y}k + u_{x,y}d + v_{x,y} - u_{x,y}(N-1)$.

Step 2. Now the new graph has monotonically increasing weight functions. We use the result of section 3.2. We derive the starting time for each task T^k_x denoted as $ST(G^r, T^k_x)$. It satisfies:

$$ST(G^r, T^{k+1}_x) - ST(G^r, T^k_x) \geq \tau^r_x(k).$$

For every edge (T_y, T_x) in G^r (corresponding to $(T_x, T_y) \in G$),

$$ST(G^r, T^{k+d}_x) - ST(G^r, T^k_y) \geq \tau^r_y(k) + c^r_{y,x}(k).$$

Let the length of this schedule be L.

Step 3. For the original task graph G, we define a schedule

$$ST(G, T^k_x) = L - ST(G^r, T^{N-k-1}_x) - \tau^r_x(N-k-1).$$

Notice that the reversed schedule has the same length as the original schedule. Thus we can apply the same performance analysis as we have done in Section 3.2 and we can show that:

Theorem 4 *The reversed schedule is legal for the original graph G.*

Proof:

For **condition R1**

$ST(G, T_x^{k+1}) - ST(G, T_x^k) = ST(G^r, T_x^{N-k-1}) - ST(G^r, T_x^{N-k-2}) + \tau_x^r(N-k-1) - \tau_x^r(N-k-2)$.

Thus

$ST(G, T_x^{k+1}) - ST(G, T_x^k) \geq \tau_x^r(N-k-2) + \tau_x^r(N-k-1) - \tau_x^r(N-k-2) = \tau_x^r(N-k-1) = \tau_x(k)$.

For **condition R2**, for each edge (T_x, T_y) in G:

$ST(G, T_y^{k+d}) - ST(G, T_x^k) = ST(G^r, T_x^{N-k-1}) - ST(G^r, T_y^{N-k-d-1}) + \tau_x^r(N-k-1) - \tau_y^r(N-k-d-1)$.

Thus

$ST(G, T_y^{k+d}) - ST(G, T_x^k) \geq \tau_x^y(N-k-d-1) + c_{y,x}^r(N-k-d-1) + \tau_x^r(N-k-1) - \tau_y^r(N-k-d-1)$.

Then we have $ST(G, T_y^{k+d}) - ST(G, T_x^k) = \tau_x(k) + c_{x,y}(k)$.

■

4 Experiments

In this section, we examine the performance of this scheduling solution on two cases and also present the experiments on nCUBE-2. For these two numerical computing problems, we use the partitioned loop programs and evaluate the performance of scheduling. In the implementation of parallel programs, we follow the task computation model [17, 25]. As soon as data needed by a task are available in the local memory, this task starts the computation and then it sends the produced results to its successors assigned in different processors. We will compare the actual performance with the predicted result.

4.1 Solving a triangular system

In Fig.2, the weight of task S^j is h^2w where w is the time for addition or multiplication. The weight of task T_q^j is about $\frac{(N-j-1)2h^2w}{P}$. There are total $P+1$ tasks in the loop body and $P+1$ processors are used.

The communication from T_q to S is to send a subvector of B and the cost is approximated as $s+t*h$ where s is the startup time and t is the transmission speed. The communication from S to T_q is to broadcast X_{j+1}. The broadcasting communication cost is estimated as $\log(P+1)(s+t*h)$ on nCUBE-2.

The weight function is monotonically decreasing. Using our method in Section 3, we estimate the parallel time as

$$PT = h^2w + \log(P+1)(s+ht) + (\frac{h^2w}{P} + h^2w + (1+\log(P+1))(s+ht))(N-1) + \frac{h^2w}{P}(N-1)^2.$$

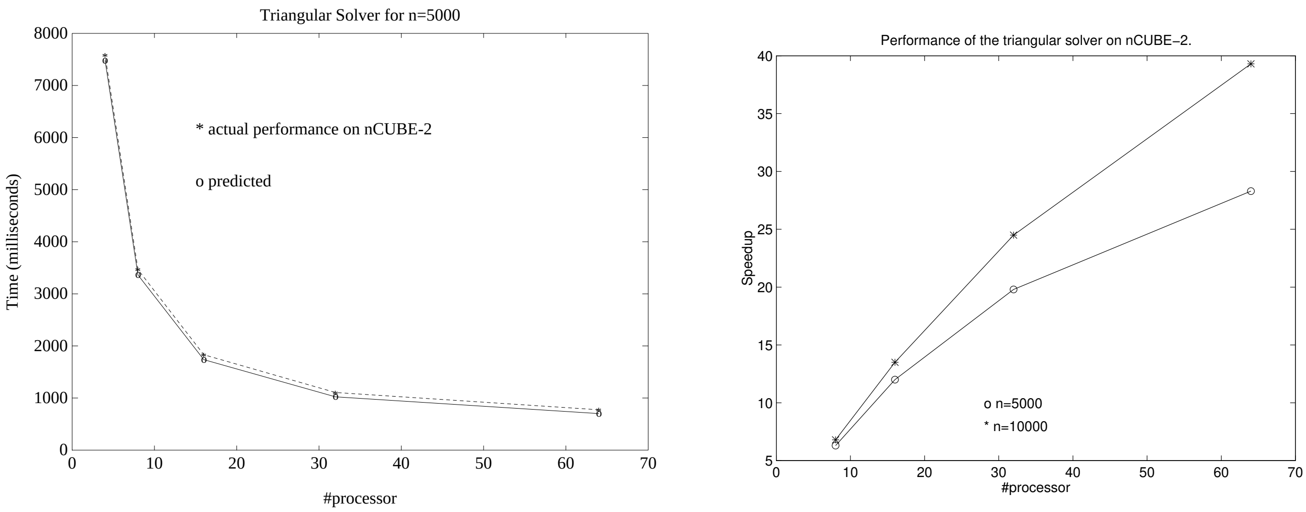

Figure 6: The performance of the triangular solver on nCUBE-2.

We implement this schedule on nCUBE-2. Processor 0 executes task S and processor q executes task T_q $(1 \leq q \leq P)$. Each processor executes its tasks in an increasing order of iteration number j. For $n = 5000$ with single precision, we use $N = 250$ and $h = 20$. The sequential time is 21890 milliseconds. Since the number of operations for the sequential algorithm is n^2w, we estimate $w = 0.8756\mu s$. For nCUBE-2, $s = 160\mu s$ and $t = 0.6\mu s$ per byte [8]. In Fig. 6, we plot the predicted performance by using these parameters in the above formula and compare it to the actual performance in the left part of Fig.6. The predicted performance is close to the actual parallel time in nCUBE-2. The right part of Fig.6 lists the speedups for $n = 5000$ and $n = 10000$.

4.2 Matrix multiplication

A matrix multiplication ($C = A \times B$) program is shown in Fig. 7. A is a $n \times n$ matrix. B is a $n \times n$ lower triangular matrix. We divide the matrix B into n rows, the sizes of rows $B_1, B_2, \cdots, B_n$ are $1, 2, \cdots n$ respectively. We also divide the matrices A and C in a row-wise manner, see Fig. 8a. Let $h = \lceil \frac{n}{P} \rceil$.

The dependence task graph is shown in Fig. 8(b). Each task T_q^k uses row B_{k+1} and row A_i to update row C_i where $h(q-1)+1 \leq i \leq hq$. This program allows the matrix B generated on the fly, so that the program can release the space of row B_{k+1} immediately after it is used by every

for $k = 0$ **to** $n-1$
    for q$=1$ **to** P
    T_q^k: if $q = 1$ Generate row B_{k+1}; endif
        for $i = h*(q-1)+1$ to $\min(h*q,\, n)$
            for $j = 1$ to $k+1$
                $c_{i,j} = c_{i,j} + a_{i,k+1} * b_{k+1,j}$;
            endfor
        endfor
    endfor
endfor

Figure 7: A program for matrix multiplication.

task T_q^k. This is useful when processors of a parallel machine do not have sufficient space to store the entire matrix B[4]. Task T_1^k is responsible to generate row B_{k+1} to be used by tasks T_q^k where $1 \leq q \leq P$. Since row B_{k+1} has $(k+1)$ nonzero elements, the weight of task T_q^k is $(k+1)2wh$ $(2 \leq q \leq P)$, where w is the time for a multiplication or subtraction. We assume the time for generating a row B_{k+1} costs $(k+1)w/2$. The weight of T_1^k is $(k+1)2wh + (k+1)w/2$. We also assume that broadcasting primitive is not used, task T_1^k will send row B_{k+1} to T_2^k, T_2^k sends to T_3^k and so on. The communication cost for sending B_{k+1} is $s + (k+1)t$. In nCUBE-2, we use the gray-code method to embed a linear processor array in a hypercube.

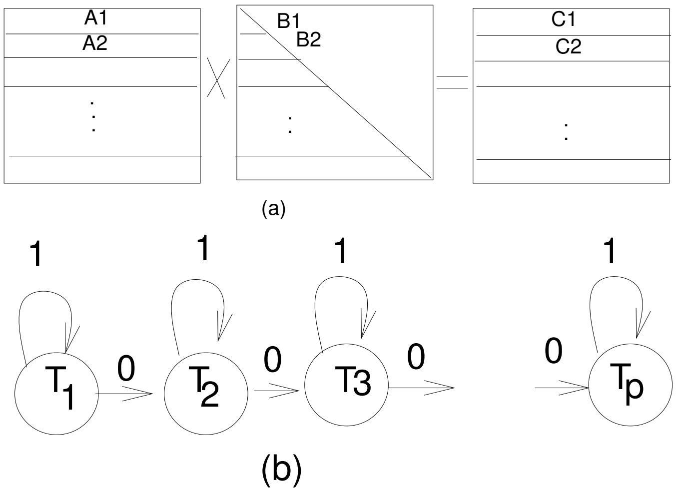

Figure 8: (a) The matrix multiplication problem: $C = A \times B$. B is a lower triangular matrix. (b) An ITG for matrix multiplication.

The weight functions are monotonically increasing. We execute this graph on P processors. Using

[4]Matrix multiplication could be simply modeled as matrix vector multiplication, which results in a fully parallel algorithm; however, it requires that each processor stores the entire matrix. For the partitioning we use, the scheduling results in a good speedup with small memory consumption. The memory-efficient algorithms for matrix multiplication have been studied in the previous work (see [10]) and task communication is needed to transfer matrix elements in these algorithms.

the solution in Section 3, we can determine the parallel time as

$$PT = (P-1)(s+t+2wh) + w/2 + (2wn + (P-1)t + w)(n-1)^2 + (hw + w/4)(n-1)^2 + 2whn.$$

$$Seq = \sum_{k=0}^{n-1}((k+1)w/2 + \sum_{i=1}^{q}(k+1)2hw) = \frac{n(n+1)}{4}w + n^2(n+1)w.$$

When n is large, the speedup approaches to P. We examine the actual performance of this schedule on nCUBE-2 machine for $n = 500$ and $n = 1000$ with single precision. Again the predicted and actual performance are very close and this result together with speedups is listed in Fig. 9.

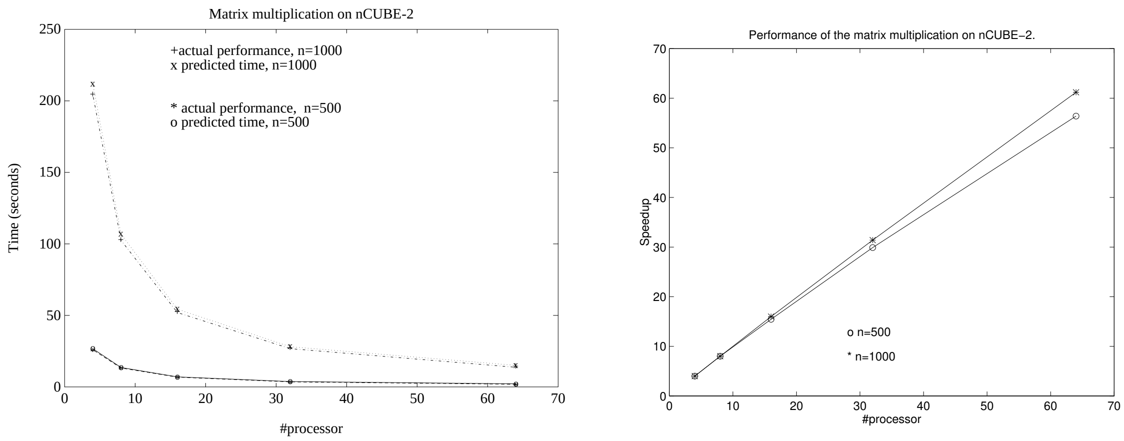

Figure 9: The performance of the matrix multiplication on nCUBE-2 for $n = 500$ and $n = 1000$.

The above two experiments indicate that our formula for computing the schedule provides a relatively accurate estimation of parallel time. Such a result considers the impact of communication and weight variations. The performance of parallelization is reasonable considering the obtained speedup in nCUBE-2. We explain the reason based on our analysis in section 3.2 as follows. In the first case, the partitioning is coarse-grain and the weights of tasks are larger than those of communication edges linked to them. Also a^{max} is close to $Q(G)$ and a^{avg}. Thus according to our formula in section 3.2, the performance difference between the heuristic schedule and the optimal solution is small. For the second case, there is no cycle and the speedup bound is estimated as $\frac{a^{avg}P}{a^{max}}$. Since $a^{avg} \approx a^{max}$ for this case, the speedup is close to P.

These two examples also demonstrate how performance prediction can guide the program partitioning. We can see that if a loop program is partitioned such that granularity is greater than 1, tasks have similar variation factors from one iteration to another (i.e. a^{max} is as small as possible, close to a^{avg}), and $Q(G)$ is small, the scheduling performance will be reasonable.

5 Concluding Remarks

We have presented a schedule for mapping weighted ITGs and predicting performance on message-passing machines when there is a sufficient number of processors, and have provided an analysis on the asymptotic performance of this schedule when task weights vary between iterations. Since the partitioning affects the performance, the performance prediction using our scheme could be useful for selecting a partitioning that leads to a good performance. The results studied in this paper are used for the two-stage scheduling approach and one of our future work is to investigate the techniques to execute multiple clusters on a limited number of processors. Directly scheduling ITGs on a fixed number of processors is studied in [26, 27] where weights are constant but communication is nonzero.

Our scheme uses a nonlinear function to model the starting time of task instances, but function coefficients for all tasks are uniform. The previous approaches usually use linear functions with uniform coefficient for constant task weights. Donaldson and Ferrante [7] show that using non-uniform coefficients can further improve the mapping performance. It will be interesting to see how such a scheme can be used for ITGs with varying task weights.

Acknowledgments

This work was supported by NSF CCR-9409695 and a startup fund from University of California at Santa Barbara (Yang) and NSF Grant CCR89-18409 (Ibarra). We are grateful to Apostolos Gerasoulis and Rutgers University for providing us access to their nCUBE-2 machine. We also thank Cong Fu, Vivek Sarkar and the anonymous referees for their valuable comments.

References

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

[2] L. F. Chao and E. H. Sha, Unified static scheduling on various models. *Proc. ICPP 1993*, Vol II, pp 231-235.

[3] Y. C. Chung and S. Ranka, Application and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed-memory multiprocessors, Supercomputing'92, 512-521.

[4] S. Chittor, S. and R. Enbody, Performance degradation in large wormhole-routed interprocessor communication networks, *Proc. of ICPP '90*, pp. Vol. I 424-428.

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

[6] A. Darte, A. and Y. Robert, Affine-by-statement scheduling of uniform and affine loop nests over parametric domains. Journal of Parallel and Distributed Computing, 15 Aug. 1995, vol.29, (no.1):43-59.

[7] V. Donaldson and J. Ferrante, Determing Asynchronous Acyclic Pipeline Execution Times, submitted for publication, 1995.

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

[9] 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.

[10] V. Kumar, A. Grama, A. Gupta, G. Karypis, *Introduction to Parallel Computing*, Benjamin/Cummings Pub, 1994.

[11] C. E. Leiserson and J. B. Saxes, Optimizing synchronous systems, IEEE 22nd Annual Symp. Foundations of Computer Science, Oct 1981. pp. 23-35.

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

[13] S. S. Pande, D. P. Agrawal and J. Mauney, Compiling functional parallelism on distributed-memory systems. IEEE Parallel & Distributed Technology: Systems & Applications, Spring 1994, vol.2, (no.1):64-76.

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

[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] J. Ramanujam, Optimal software pipelining of nested loops, Proc. of 1994 IPPS, 335-342.

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

[18] V. Sarkar, Determining average program execution times and their variance, *Proc. of 1989 SIGPLAN*, ACM, pp. 298-312.

[19] V. Sarkar and R. Thekkath, A general framework for iteration-reordering loop transformations, ACM SIGPLAN 92 PLDI. pp 175-187.

[20] 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.

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

[22] 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.

[23] 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.

[24] A. Zaky and P. Sadayappan, Optimal static scheduling of sequential loops on multiprocessors. *Proc. of ICPP 1989*, Vol 3, 130-137

[25] 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.

[26] T. Yang, C. Fu, A. Gerasoulis and V. Sarkar, Mapping iterative task graphs on distributed-memory machines. Proc. of 24th Inter. Conference on Parallel Processing, Wisconsin, Aug. 1995. Vol II. 151-158.

[27] T. Yang, C. Fu, Heuristic Algorithms for Scheduling Iterative Task Computations on Distributed Memory Machines, Tech Report TRCS95-16, UCSB, 1995. HTTP://www.cs.ucsb.edu/Research/rapid_sweb/RAPID.html.

Appendix

A Procedure $LPATH$

The basic procedure $LPATH$ used in the algorithm is described below.

- $LPATH(F, v, r, b)$. The input is a set of inequalities F with forms

  $$f_j - f_i + d_{i,j} * r \geq w_{i,j}$$

  where j and i are integers from 1 to v. $w_{i,j}$ and b are nonnegative constants. $d_{i,j}$ are nonnegative integer constants. The output is the nonnegative values for f_i $i = 1, \cdots v$, and a value for r such that $r \geq b$.

  The solution of above inequalities can be solved by any single-source shortest path algorithm, an approach used in systolic network retiming [11]. Notice that in our case, the solution may not be integer, and also we impose a lower bound for r. The above inequalities are transformed as:

  $$f_j - f_i \geq w_{i,j} - d_{i,j} * r.$$

We construct a directed graph L of v nodes. Each inequality in F corresponds an edge in L. The weight of this edge is $w_{i,j} - d_{i,j} * r$. Notice that r is to be solved. If L is a DAG, then then we set $r = b$. We use the topological searching to derive the solution.

In a cyclic graph, for each cycle C we add weights of edges in this cycle together.

$$0 \geq \sum_{(i,j)\in C} w_{i,j} - d_{i,j} * r, \qquad \text{thus } r \geq \frac{\sum_{(i,j)\in C} w_{i,j}}{\sum_{(i,j)\in C} d_{i,j}}.$$

The minimum value of r that satisfies this condition is

$$R = \max_{\text{all cycles C } C} \frac{\sum_{(i,j)\in C} w_{i,j}}{\sum_{(i,j)\in C} d_{i,j}}.$$

We are not able to enumerate all cycles. But given a value for r, we can use the shortest path algorithm to determine if there is a solution. Note that the input inequalities we use guarantee that $\sum_{(i,j)\in C} d_{i,j} \geq 1$. Thus $R \leq \sum w_{i,j}$. Let $Q = \sum w_{i,j}$. We can use a binary searching to find the value R for r with an accuracy of ϵ in a time complexity of $v|F| \log \frac{Q}{\epsilon}$. r satisfies that $R \leq r \leq R + \epsilon$.