On Symbolic Scheduling and Parallel Complexity of Loops

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 first consider the symbolic scheduling and performance prediction of a partitioned single loop on message-passing architectures with nonzero communication and a sufficient number of processors. The loop body contains a set of coarse-grain tasks whose computational weights change during the course of the iterations. Using the macro-dataflow task model and software pipelining techniques, we develop an algorithm for computing a heuristic schedule, and provide analytic and experimental results on the correctness and asymptotic performance of this schedule. Using this result, we can show the impact of partitioning on the performance of parallelization. While this result is effective for a class of loops, there are many other methods proposed for loop parallelization. An interesting fundamental question is whether every instance of a nested loop can be efficiently executed. We present some positive and negative results on this issue.

1 Introduction

The problem of loop parallelization has been addressed extensively in the literature [1, 14, 24]. There are many loop transformation and partitioning techniques proposed to uncover loop parallelism. Since many techniques are applicable to each given loop, a scheduling algorithm that predicts achievable performance is helpful in selecting transformation and also for producing efficient code. Scheduling techniques for fine-grain machines have been studied in [1, 12, 16, 25]. Computational weight information has been used for load balancing, but communication cost is usually not included because this is not needed for the architectures at which they are targeted. For message-passing architectures, communication could be asynchronous and message transmission delay affects the efficiency of computation. Coarse grain partitioning has been used for loop parallelization to avoid high communication startup cost. The previous work on task scheduling and performance prediction in [4, 7, 13] has considered communication overhead; however, these results are for directed acyclic task graphs.

Techniques for generating coarse grain dependence graphs for loop programs are discussed in [5]. The difficulty in providing a good loop schedule with accurate performance prediction is that loop bounds may be unknown, and coarse grain tasks may contain large chunks of computation and may involve loop indices and task weights may change during iterations. A scheduling algorithm needs to consider symbolic information, weight variations and communication delay.

In this paper, we demonstrate a solution for symbolic scheduling of a class of partitioned single loop programs by utilizing the ideas of the macro dataflow task model [7, 18] and pipelining techniques. We assume that weights of tasks in the loop body change monotonically during the course of iteration and there is a sufficient number of processors. The problem of scheduling with a sufficient number of processors is important in the sense that we can predict the best achievable performance which can guide program partitioning and mapping. Also the solution to this problem produces a number of task clusters which can be merged to a limited number of clusters for multi-threaded computation. The idea of such multi-stage scheduling approaches has been used in [18, 26]. Finding an optimal scheduling solution is NP-complete [18]; thus 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.

Our paper can be regarded as yet another work on loop scheduling by making certain assumptions and taking advantage of some known information. Considering other related existing or future work dealing with compile and run time loop parallelization, one might be led to believe that a parallelizing compiler can be developed eventually, which leads to efficient parallelization of all loops. We conduct an analysis on such a possibility using the PRAM theory. We formally show that it is unlikely (in some precise sense, to be defined later) that a "general" compiler or a run-time compilation scheme can be constructed to produce an efficient parallel code for every instance of a doubly nested loop. However, we show that it is possible for some single loops. While this result is of theoretical interests, it provides a formal analysis when an efficient parallelization is possible or not possible.

Section 2 gives problem definitions and assumptions. Section 3 presents a scheduling algorithm that takes into account communication overhead and weight variations. Section 4 provides experimental results that validate the correctness of the analytic results. Section 5 gives some observations concerning the complexity of loop parallelization.

2 Definitions

We first study the problem of scheduling the following loop on message-passing architectures:

for k= 0 **to** N-1
  $T_1; T_2; \cdots, T_v$
endfor

The loop body contains v tasks and the information on task dependence is available. We assume that due to coarse grain partitioning, a task within a loop body 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).

We use the *macro-dataflow task model* for computation execution [18]. 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 loop (called $E(G,N)$) is a Directed Acyclic Graph (DAG) containing all $N * v$ task instances T_i^k $(1 \leq i \leq v,\ 0 \leq k \leq N-1)$.

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.

The weight function of task T_i^k is $\tau_i(k)$. 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[3]. 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 are modeled in [3]. We assume that physical processor distances does not significantly affect communication delay in a wormhole routing network. [5, 18] discussed methods for estimating communication and computation cost. We will discuss the assumptions on weight functions later.

Let $ST(T_i^k)$ be 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 task dependence.

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: Solving a triangular system.

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 $n \times n$, 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 a dependence graph 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[4]. The weight information of this task graph is discussed in Section 4.

Fig. 3 is another example of a partitioned loop program whose loop body contains three tasks. Fig. 4(a) is the dependence graph of this program. (b) is the expanded task graph of G 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

[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 [24] instead of precise distance values. Our result can be extended for this case and but will not be discussed here due to space constraint.

[3]This is related to the concept of "exploiting data locality" used in the literature. In task computation model, exploring data locality means to localize data communication between tasks by assigning them in the same processor.

[4]The dependence is actually overestimated since only 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 [24] may overestimate dependence since accurate estimation may be intractable; however, overestimation does not affect the correctness of parallel execution.

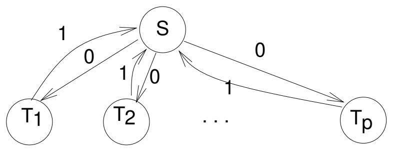

Figure 2: A task dependence graph for solving a triangular system.

$\tau_x(k) = k+1$, $\tau_y(k) = 1.5k+1$ and $\tau_z(k) = 2k+1$.

for k = 0, N-1

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

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

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

endfor

Figure 3: A partitioned single loop 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 usually possess certain regularity depending on iteration numbers. 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 $b_i \geq 0$. And $c_{i,j}(k) = u_{i,j}k + v_{i,j}$ where $u_{i,j} \geq 0$ and $v_{i,j} \geq 0$. We consider the case when the weights decrease monotonically in Section 3.4.

The following terms will be used in the rest of this paper. Let $A = \sum_{i=1}^{v} a_i$, $B = \sum_{i=1}^{v} b_i$, $a^{max} = \max_{i=1}^{v} a_i$, $a^{avg} = \frac{A}{v}$. Let C be a cycle in a task dependence graph G, without the loss of generality, assume that $C = T_1, T_2 \cdots T_j, T_1$. Let $A1_C = \sum_{i=1}^{j} a_i$, $B1_C = \sum_{i=1}^{j} b_i$, $A2_C = a_j + u_{j,1} + \sum_{i=1}^{j-1}(a_i + u_{i,i+1})$, $B2_C = b_j + v_{j,1} + \sum_{i=1}^{j-1}(b_i + v_{i,i+1})$, $A2 = \max_{\text{all cycles C}} A2_C$, $B2 = \max_{\text{all cycles C}} B2_C$.

3 Scheduling with nonzero communication and weight variation

3.1 Performance Bounds

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.

Given p processors and a DAG, a 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.

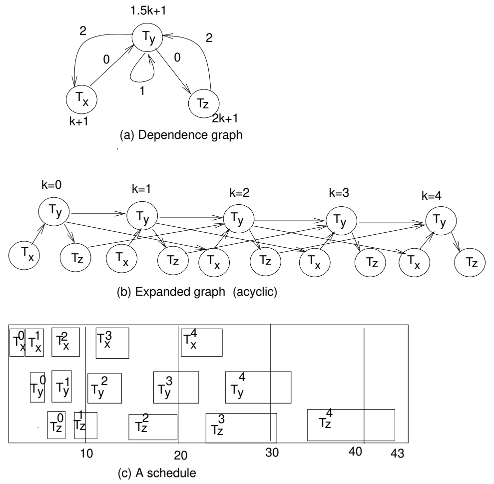

Figure 4: (a) The task dependence graph G for Fig. 4. (b) The expanded graph $E(G,5)$. (c) A valid schedule.

Theorem 3.1 *Given a task dependence graph G and p processors, let C be a cycle in G, D_C is the summation of distances in this cycle. Let $M_C = \lfloor \frac{N-1}{D_C} \rfloor$. $PT_p \geq \frac{0.5*A*N(N-1)+BN}{p}$, and $PT_p \geq \max_{\text{all cycles } C}(M_C * B1_C + 0.5M_C(M_C-1)*A1_C * D_C)$.*

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 $M_C = 4/2 = 2$.

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 the loop). We will provide an analysis of its asymptotic performance. Then we add more processors if necessary to improve the scheduling performance.

Periodic scheduling used in software pipelining [1] executes tasks using a constant interval, i.e. 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 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:

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

The following coefficients are to be determined: two nonnegative values α_x and β_x corresponding to each task, and two nonnegative coefficients H and r. Notice that these coefficients may not be integral. We assign the instances of task T_i to processor i. We compute the coefficients in two steps as described below.

Step 1. For each dependence edge (T_x, T_y) ($T_x \neq$

T_y) we setup the following inequalities called $F1$:

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

We use the shortest path algorithm 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 use the shortest path algorithm 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 $O(ve \log \frac{A2*B2}{\epsilon^2})$. We can show that the above algorithm produces a legal solution.

Theorem 3.2 *The schedule determined by the above algorithm is legal.*

Example: For Fig. 4(a), we have the following inequalities $F1$ in Step 1: $\beta_y - \beta_x \geq 1$, $\beta_x - \beta_y + H2 \geq 1.5$, $\beta_z - \beta_y \geq 1.5$, $\beta_y - \beta_z + H2 \geq 2$. Notice that $v = 3$, all communication weights are 1 and $a^{max} = 2$. The following values are determined: $H = 2$, $\beta_x = 0$, $\beta_y = 1$ and $\beta_z = 2.5$.

In Step 2, notice that all $b_i = 1$ and $v_{i,j} = 1$. We have the following inequalities $F2$: $\alpha_y - \alpha_x \geq 1 + 1$, $\alpha_x - \alpha_y + r2 \geq 1 + 1 - 0*2 - 2^2$, $\alpha_z - \alpha_y \geq 1 + 1$, $\alpha_y - \alpha_z + r2 \geq 1 + 1 - 1*2 - 2^2$. Thus, $r_1 = max(0, b_x - \beta_x) = 1$ and then we have $r = 1$, $\alpha_x = 0$, $\alpha_y = 2$ and $\alpha_z = 4$. We have the following schedule: $ST(T_x^k) = k + k^2$, $ST(T_y^k) = 2 + 2k + k^2$, $ST(T_z^k) = 4 + 3.5k + k^2$.

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 $Seq = 4.5(N-1)N/2 + 3(N-1)$. When N is large, the speedup is approximately equal to $4.5/2 = 2.25$.

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 performance of parallelization and examine what kinds of partitioning are preferred.

A definition and formal analysis on granularity of a DAG is given in [7] and we use a slightly different definition to quantify the granularity of a task dependence graph G as:

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

Theorem 3.3 *If a^{max} and b^{max} are bounded constants, let*

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

The asymptotic performance upper bound is:

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

Example: For Fig.4, we have $Q(G) = 3.5/2 = 1.75$, $a^{avg} = 4.5/3 = 1.5$, $a^{max} = 2$, $1/g = 0$. 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.

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.

3.3 Adding more processors

We use loop unrolling technique [1, 15] to increase the number of tasks within the loop body and use more processors to execute these tasks.

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

In general if we enroll a loop by f times, the number of iterations is reduced from N to $\lfloor N/f \rfloor$ and the number of tasks in the loop body is increased by f times. The unrolled 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 $f * a^{max}$, $Q(G^f) = f^2 Q(G)$,

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

Assume that ϵ is very small. Thus set $f = \lceil \frac{a^{max}}{(1+1/g)*Q(G)} \rceil$. For such an unrolling factor f, the

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

Figure 5: An example of unrolled loop.

heuristic solution is competitive to the optimal solution in a ratio of about $1 + 1/g$. The achievable speedup bound is about $\frac{a^{avg} * v}{(1+1/g)Q(G)} \approx a^{avg} * vf/a^{max}$. If a partitioning is chosen such that a^{avg} is close to a^{max}, a good efficiency can be obtained.

3.4 Monotonically decreasing weight functions

We discuss a schedule when weights monotonically decrease. We assume that $\tau_i(k) = -a_i k + b_i$ 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 to derive the starting time for each task T^k_x denoted as $ST(G^r, T^k_x)$. 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 3.4 *The reversed schedule is legal for the original graph G.*

4 Experiments

In this section, we examine the performance of this scheduling solution on two numerical problems and compare the actual performance on nCUBE-2 with the predicted performance.

4.1 Solving a triangular system

In Fig.2, the weight of task S^j is $h^2 w$ where w is the time for addition or multiplication. The weight of task T^j_q is about $\frac{(N-j-1)2h^2 w}{P}$. 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 $PT = h^2 w + \log(P+1)(s+ht) + (\frac{h^2 w}{P} + h^2 w + (1+\log(P+1))(s+ht))(N-1) + \frac{h^2 w}{P}(N-1)^2$. We imple-

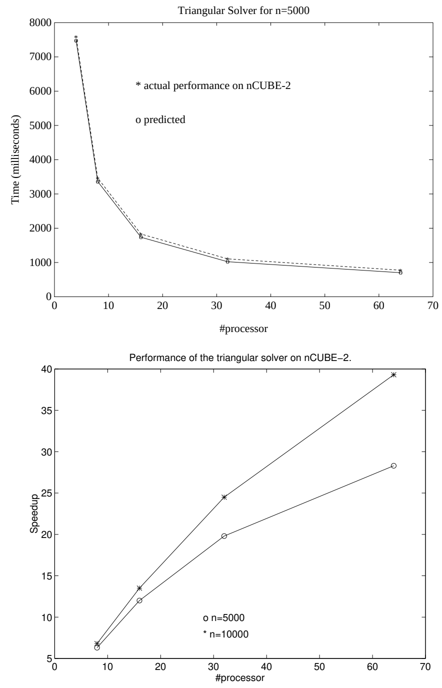

Figure 6: The performance of the triangular solver on nCUBE-2.

ment 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. 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 top part of Fig.6. The predicted performance is close to the actual parallel time in nCUBE-2. The bottom part of Fig.6 lists the speedups for $n = 5000$ and $n = 10000$.

4.2 Matrix multiplication

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.

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 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 task T_q^k. This is useful when processors of a parallel machine do not have sufficient space to store the entire matrix B[5]. 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.

[5]Matrix multiplication could be simply modeled as matrix vector multiplication, which results in a fully parallel algorithm; however it requires 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 [11]) and task communication is needed to transfer matrix elements in these algorithms.

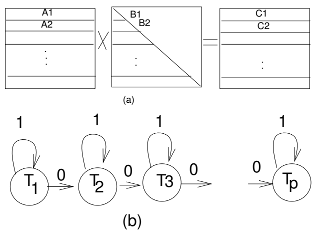

Figure 8: (a) A matrix multiplication problem. (b) The task dependence graph for (a).

The weight functions are monotonically increasing. We execute this graph on P processors. Using the solution in Section 3, we can determine the parallel time as $PT = (P-1)(s+t+2wh) + w/2 + (2wn + (P-1)t + w)(n-1)^2 + (hw + w/4)(n-1)^2 + 2whn$. $Seq = \frac{n(n+1)}{4}w + n^2(n+1)w$.

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 $n = 1000$. Again the predicted and actual performance are very close and this result together with speedups is listed in Fig. 9.

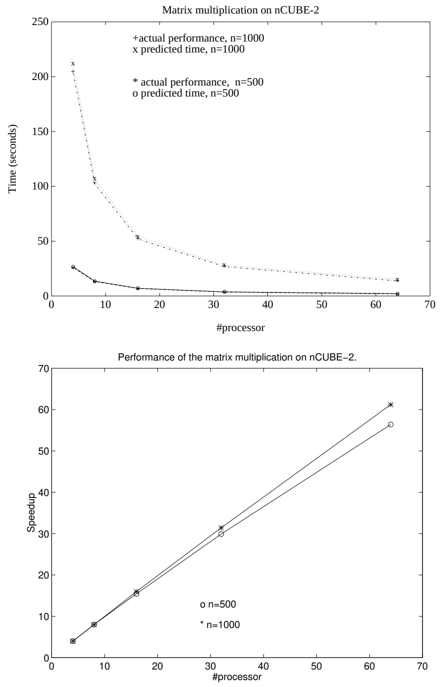

Figure 9: The performance of the matrix multiplication on nCUBE-2 for $n = 500$ and $n = 1000$.

The above two experiments indicate that our schedule provides a relatively accurate estimation of parallel time. 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 The parallel complexity of loops

As mentioned in Section 1, there are various efforts in developing techniques to parallelize loops for different cases, e.g. [14, 24]. These techniques including our result presented above preserve data dependence. There are more aggressive methods which break data dependence but still preserve the input-output semantics of a loop [17]. Run-time compilation techniques [17, 20] provide more avenues to parallelize loops which contain unknown information at compile-time. We expect that compiling of loops will be still a major research focus since loops contain most parallelism in application programs and more sophisticated techniques will be developed. An interesting question is whether a general compiler can be developed eventually that incorporates sophisticated techniques to produce an efficient parallel algorithm for every instance of a nested loop. Notice that we are not concerned about the complexity of the compiler, i.e., a compiler could spend whatever time it needs. We are interested only in the complexity of the parallelized code.

We use the PRAM and complexity theory to conduct a formal analysis. To be precise, define $EREW^i$ to be the class of problems solvable by Exclusive Read Exclusive Write PRAMs in $O(\log^i n)$ time using a polynomial number of processors, and $NC = \bigcup_{i\geq 1} EREW^i$. There is an alternative definition of NC using Boolean circuits. If NC^i is the class of problems solvable by uniform Boolean circuits of polynomial size (= number of gates) and depth $O(\log^i n)$, then $NC = \bigcup_{i\geq 1} NC^i$ [2, 23]. We state the results using the Boolean circuit model of parallel computation.

The situation is analogous to that for sequential computation. The class of problems solvable by by Turing machines in polynomial time is called P. Whereas P can be thought of as the class of problems that are feasible (tractable) under the sequential model of computation, we can think of NC as the class of problems that can be solved efficiently under the parallel model of computation. Although the models for the study of NC may not be too practical because they are assumed to have a polynomial number of processors, the results have practical implications when dealing with the more practical models, where the number of processors is fixed, independent of the size of the problem; i.e., even when the number of processors is fixed, it is reasonable to expect that, generally, the problems in NC will admit more efficient parallel algorithms than those that are not in NC.

It is easy to show that NC is contained in P. The converse is not known, although it is widely conjectured that P is not contained in NC.

A problem is P-complete if it is in P, and it has the property that if it is in NC, then P is contained in NC, i.e., $P = NC$. A P-complete problem can be regarded as a representative of the hardest problems in P to parallelize. Since it is conjectured that P is not equal to NC, a P-complete problem is unlikely to admit an efficient parallel algorithm. Here we observe that doubly-nested loops can solve P-complete problems and, hence, there computations are not likely to be efficiently parallelizable.

Notation: In what follows, A is an alphabet (i.e., a finite set of symbols), S is a subset of A, 2^A is the set of all subsets of A, $\cup$ is set union, and $\cap$ is set intersection; $g, g_1, g_2, ...$ are functions from $A \times 2^A$ to 2^A. For a symbol a and a set of symbols Q, we use $g(a, Q)$ to denote the set $\{g(a, x) | x \in Q\}$. For notational convenience, let $a_0 = b_0 = \epsilon$, and the boundary conditions $R(i, -1) = R(-1, i) = D$, where ϵ and D are dummy values.

Theorem 5.1 *There is an alphabet A, a subset S, and functions g_1 and g_2 from $A \times 2^A$ to 2^A such that determining the value of $R(n, n)$ computed by the following code is unlikely to be in NC.*

$R(0,0) := S;$
for $i = 1$ **to** n
  for $j = 1$ **to** n
    $R(i,j) := g_1(a_i, R(i-1,j)) \cap g_2(b_j, R(i,j-1))$
  endfor
endfor

However, for the case of single loops, we can show the following:

Theorem 5.2 *Determining the value of $R(n,n)$ computed by the following code is in NC^1 for any alphabet A, subset S, and function g from $A \times 2^A$ to 2^A.*

$R(0) := S;$
for $i = 1$ **to** n
  $R(i) := g(a_i, R(i-1))$
endfor

In contrast to Theorem 5.1, when $\cap$ is replaced by

$\cup$, we have

Theorem 5.3 *Determining the value of $R(n,n)$ computed by the following code is in NC^2.*

$R(0,0) := S;$
for $i = 1$ **to** n
  for $j = 1$ **to** n
    $R(i,j) := g_1(a_i, R(i-1,j)) \cup g_2(b_j, R(i,j-1))$
  endfor
endfor

Example: Consider the string shuffling problem [22], which is defined as follows. Given an alphabet Σ and three strings $x, y, z \in \Sigma^*$, where $|x| = |y|$ and $|z| = |x| + |y|$, determine whether z is a shuffle of x and y. Let $x = a_1 a_2 \ldots a_n$, $y = b_1 b_2 \ldots b_n$ and $z = c_1 c_2 \ldots c_{2n}$. The string shuffling problem can be solved using a doubly-nested loop with $S = t$ and g_1 and g_2 defined below, where t and $\emptyset$ represent *true* and *false*, respectively. $g_1(a_i, c_{i+j}, R(i-1,j)) = \{t\}$ if $R(i-1,j) = \{t\}$ and $a_i = c_{i+j}$ else $\emptyset$. $g_2(b_j, c_{i+j}, R(i,j-1)) = \{t\}$ if $R(i,j-1) = \{t\}$ and $b_j = c_{i+j}$ else $\emptyset$.

The string z is a shuffle of strings x and y if and only if $R(n,m) = \{t\}$. Hence, the string shuffling problem is in NC^2.

The code in Theorem 5.3 or in the preceding example can be generalized to have k nested loops, where $R(i_1, ..., i_t)$ now depends on t unions of functions $g_1, \ldots, g_t$, and each g_s depends essentially on only one coordinate of R. So, for example when $t = 3$, we have $R(i,j) = g_1(a_i, R(i-1,j,k)) \cup g_2(b_j, R(i,j-1,k)) \cup g_3(c_k, R(i,j,k-1))$. This type of nested loops is also in NC^2.

6 Concluding Remarks

We have presented a schedule for mapping weighted task dependence graphs on message-passing machines when there is a sufficient number of processors and task weights vary between iterations. Our future work is to investigate the techniques to execute tasks with more general types of weight functions on a limited number of processors. Some result to derive a schedule when the number of processors is limited, weights are constant but communication is nonzero, is in [27].

We have examined the general question of whether efficient loop parallelization is always possible by using *any* compilation technique. It indicates that it is unlikely that we can find a general technique that will work for all loops. What we can hope for is the development of techniques that will handle large classes of loops.

Acknowledgments

This work was supported in part by NSF Grant CCR-9409695 (Yang) and NSF Grant CCR89-18409 (Ibarra). We thank the anonymous referees for their useful comments.

References

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

[2] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[3] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken LogP: Towards a Realistic Model of Parallel Computation, Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, May 1993

[4] 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.

[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.

[6] A. Darte , Y. Robert, Constructive methods for scheduling uniform loop nests. IEEE Transactions on Parallel and Distributed Systems, Aug. 1994, vol.5, (no.8):814-22.

[7] 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.

[8] O. H. Ibarra and N. Q. Tran. On the parallel complexity of solving recurrence equations. *Proceedings of the 5th International Symposium on Algorithms and Computation*, 834: 469-477, 1994.

[9] J. Jaja. An Introduction to Parallel Algorithms. Addison-Wesley, Reading, Mass., 1992.

[10] N. Jones and W. Lasser. Complete problems for deterministic polynomial time. *Theoretical Computer Science*, 3(1):105–117, 1976.

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

[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, A scalable scheduling scheme for functional parallelism on distributed memory multiprocessor systems, Tech. Report, North Carolina State Univ. 1993.

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

[15] 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.

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

[17] L. Rauchwerger and D. Padua, The privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization, *Proc. of 1994 ACM Int. Conf. on Supercomputing*, 33-43.

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

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

[20] S. D. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das and J. Saltz, Run-time and Compile-time Support for Adaptive Irregular Problems, SC '94, pp. 97-106. IEEE. Nov, 1994.

[21] J. Ullman and A. Van Gelder. Parallel complexity of logical query programs. *Algorthmica*, 3:5–42, 1988.

[22] J. Van Leeuwen and M. Nivat. Efficient recognition of rational relations. *Information Processing Letter*, 14:34–38, 1982.

[23] Walter L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22:365–383, 1981.

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

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

[26] 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.

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