Loop Scheduling on Distributed-Memory Parallel Processors

Hesham El-Rewini
Department of Math and Computer Science
University of Nebraska at Omaha
Omaha, NE 68182-0243

Ted Lewis
Computer Science Department
Oregon State University
Corvallis, OR 97331-3902

91-60-1

Loop Scheduling On Distributed-Memory Parallel Processors

Hesham El-Rewini

Department of Math & Computer Science
University of Nebraska at Omaha
Omaha, NE 68182-0243

(402) 554-2852

rewini@unocss.unomaha.edu

Ted Lewis

Computer Science Department
Oregon State University
Corvallis, OR 97331-3902

(503) 737-3273

lewis@mist.cs.orst.edu

Abstract

We provide a new solution to the problem of scheduling parallel program tasks that are enclosed in a set of nested loops on distributed-memory parallel computers. The new solution is extended to a new scheduling heuristic for scheduling unrolled loops onto arbitrary target machines. Our technique allows several iterations of a set of loops as well as tasks within the same iteration to overlap in execution in a way that minimizes the loop completion time. We then compare *local neighborhood search* versus *simulated annealing* optimization methods to find the best way to unroll the nested loops. Finally, the schedule is presented in the form of a Gantt chart that indicates the allocation and the order of the tasks in the unrolled loops.

1 Introduction

Many researchers in the areas of compiler optimization and task scheduling have studied loops [4, 9, 10, 12, 13], because they offer potentially great improvements in parallel program performance. For example, work has been done on the transformation and reduction of recurrences in sequential loops and partitioning such loops into independent sequential loop components which can execute in parallel [6, 7]. Wolfe [10] has studied several high level compiler loop transformations, namely vectorization, concurrentization, loop fusion, and loop interchange in order to speed up the execution of loops using parallel computers. The Doacross technique [2], takes advantage of loop-carried dependences and provides a unified framework to schedule sequential and parallel loops for both SIMD and MIMD parallel machines.

Although relatively efficient schedules can be obtained for sequential loops using the Doacross scheduling technique, fine grained parallelism in such loops is not exploited. To utilize fine-grained parallelism in a sequential loop, some compilers unroll the loop body several times and compact the produced code by treating the unrolled loop as an acyclic dependence graph. Zaky and Sadayappan [13] addressed the problem of optimally scheduling sequential loops on synchronous parallel processors such as VLIW. They showed that when a relatively high degree of hardware parallelism is available in the system, simple loop unrolling is not an effective approach to extracting parallelism and gave an algorithm that produces an optimal synchronous schedule for an innermost sequential loop on an idealized parallel processor system with an unbounded number of processors.

A load balancing technique called loop spreading that evenly distributes parallel tasks on multiple processors without decrease in performance even when the size of the input data is not a multiple of the number of processors, was introduced by Wu and Lewis [11]. One common approach that has been used to schedule tasks contained in a loop on parallel computers is to assign each iteration to one processor. Assigning all tasks in one iteration to the same processor completely ignores any parallelism that might occur within each iteration. Also, when data are passed from one iteration to another, the synchronization delay might slow down the execution. On the other hand when data are passed from one task to another within the same iteration, it might be useful to assign all tasks to the same processor to reduce communication delay. Consequently, a scheduling method that can exploit parallelism that might exist among different iterations as well as within each iteration is needed.

In this paper we introduce a representation of parallel program task dependencies that are enclosed in a set of nested loops. We also introduce a loop unrolling technique that allows several iterations of a set of loops as well as tasks within the same iteration to overlap in execution in a way that minimizes the loop execution time. In addition, this technique can handle the case when the loop upper bounds are not known before execution time.

We use the scheduling heuristic MH developed earlier by the authors and described in [3], to schedule the body of the unrolled loops on a given parallel machine. Because this heuristic accounts for communication time delays, it is especially appropriate for distributed-memory machines. However, the technique is also applicable to shared-memory machines.

For scheduling, we compare local neighborhood search versus simulated annealing techniques to find the best loop unrolling and achieve near-optimal mapping of the program tasks on the given target machine. We have integrated both local neighborhood search and simulated annealing methods in one tool to find: 1) the best unrolling vector for a particular set of tasks when they run on a particular target machine and 2) the Gantt chart that indicates the allocation and the order of the tasks in the unrolled loop on the available processing elements. In general we recommend that both methods be tried since their performance differs from one application to another and from one target machine to another.

The paper is organized as follows. In section 2 we discuss dependence among tasks and give the terminology used in the paper. We discuss loop unrolling in section 3. Scheduling post-unrolling loops is given in section 4. The loop unrolling optimization problem is introduced in section 5. We present two simple examples in section 6. Finally, we give our conclusion in section 7.

2 Dependence Among Tasks

A dependence between two tasks can be a data dependence or a control dependence. A control dependence is a consequence of the flow of control in a program. For example, the execution of a task in one path under an *if* test is contingent on the *if* test taking that path. Thus, the task under control of the *if* is control dependent upon the *if* test. Data dependence is a consequence of the flow of data in a program. A task that uses a variable in an expression is data dependent upon the task which computes the value of the variable. Dependence relations between tasks forming a program can be viewed as precedence relations. If task w is control dependent or data dependent on task v, then execution of task v must precede execution of task w . Data dependence in loops can be further classified as follows: 1) loop-carried dependence; if data are passed between different iterations and 2) loop-independent dependence; if data are passed from one task to another within the same iteration.

2.1 Terminology and Definitions

This section gives definitions which will be used throughout this paper. We use V to represent the set of all tasks that are enclosed in n nested loops and k = |V|.

Iteration Vector. When some tasks are contained in n nested loops, we refer to separate instances of their execution using an *iteration vector*. A vector $I = \langle i_1, i_2, \ldots, i_n \rangle$ is called an iteration vector if the loop body is executed in the period when the j^{th} level loop is in the

i_j^{th} iteration, $1 \leq j \leq n$. Simply, an iteration vector holds the values of the loop control variables of the n nested loops.

Distance Vector. Using iteration vectors, we can define a *distance vector* for each dependence between tasks. Suppose that v and w are two tasks enclosed in n nested loops. If w during iteration I_w is dependent on v during iteration I_v the *distance vector* for this dependence is $I_d = I_w - I_v$. Task w is "loop-carried" data dependent on task v iff $I_w \neq I_v$ (the distance vector elements are not all zeros). Otherwise the dependence is called "loop-independent" (the distance vector elements are all zeros).

Dependency Ordered Pair. We define a *dependency ordered pair* between two tasks w, during iteration I_w, and v, during iteration I_v, as (I_d, D) where I_d is the distance vector $I_d = I_w - I_v$ and D is the size of the message that w receives from v. Task w can have more than one data dependency ordered pairs from task v. This multi-dependence between v and w can be represented using a dependency set.

Dependency Set. The *dependency set* between two tasks is the set of all dependency ordered pairs between those tasks.

Upper Bound Vector. We define the *upper bound vector* for n nested loops to be $<b_1,b_2,...,b_n>$, where b_i is the upper bound of the loop at the i^{th} level (notice that the outermost loop is at the first level and the innermost loop is at the n^{th} level).

Unrolling Vector. We define the *unrolling vector* = $<u_1,u_2,...,u_n>$, that is the i^{th} loop is unrolled u_i times.

Elements of iteration and distance vectors are numbered from outermost to innermost, as are loops. Thus, the outermost and innermost loops are at levels 1 and n, respectively. We assume that loops are normalized to iterate from one to some upper bound in steps of one. Non-normalized loops can be normalized through simple transformations [10]. We also assume perfect (tightly) nested loops which means all the tasks are enclosed in all the nested loops. At least two ways of transforming imperfectly nested loops into perfect ones have been developed; loop distribution [10], and the Non-Basic-to-Basic-Loop transformation [1]. The last assumption we make is that all distance vectors can have only non-negative values to guarantee acyclic loop-carried data dependence.

2.2 Task Graph Representation in Loops

The loop-independent data dependences among tasks can be easily represented using directed edges in task graphs. However, loop-carried data dependences are difficult to

represent in task graphs since they express dependences among tasks in different loop iterations. In this section we introduce the dependency matrix (DM) to represent loop-independent as well as loop-carried data dependences among tasks that are enclosed in a set of nested loops. The amount of computation needed at each task can also be represented using the task size array (TSA).

Dependency Matrix (DM)

Loop-carried and loop-independent data dependence among the tasks in V can be represented using a k×k *Dependency Matrix* (DM). DM[i,j] represents the dependency set from task v_j to v_i, where $v_j, v_i \in V$. A Φ entry in DM[i,j] means that there is no dependence from v_j to v_i. Recall that V is the set of tasks enclosed in the nested loops and k = |V|.

Task Size Array (TSA)

We also define the *task size array* (TSA) as an array of length k. TSA[i] represents the size (the amount of computation) of task v_i.

Figure 1 shows an example of DM and TSA for two tasks **a** and **b** that are enclosed in 2 nested loops. It also shows the array TSA for tasks **a** and **b**. DM[1,1] = {(<1,0>,5),(<1,1>,10)} means that an instance of task **a**, during iteration <i,j>, is data dependent on instances of itself during iterations <i-1,j> and <i-1,j-1> and the data sizes are 5 and 10 bytes, respectively, DM[1,2] = Φ means that there is no data dependence from **b** to **a**, DM[2,1] = {(<0,0>,20)} means that **b** is data dependent on **a** during the same iteration and the data size is 20 bytes (Notice that this is a loop-independent data dependence), and finally DM[2,2] = {(<1,0>,12)} means that an instance of task **b**, during iteration <i,j>, is data dependent on an instance of itself during iteration <i-1,j> and the data size is 12 bytes. The size of tasks **a** and **b** are 7 and 9 respectively as given in TSA.

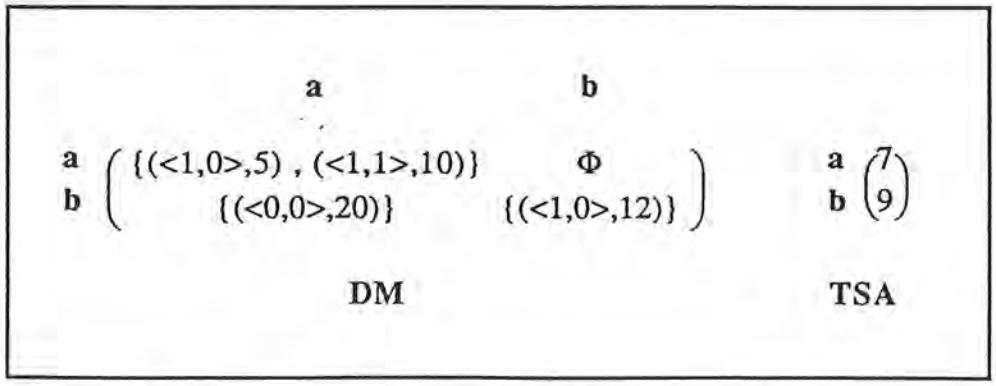


Figure 1 DM and TSA for Two Tasks **a** and **b** Enclosed in Two Nested Loops.

3 Unrolling Loops

When a single loop, with upper bound b, is unrolled u times, u + 1 copies of the body are replicated, the loop control variable is adjusted for each copy, and the step value of the loop is multiplied by u + 1. Similarly, when a set of n nested loops, with upper bound vector = $<b_1,b_2,...,b_n>$ is unrolled using unrolling vector = $<u_1,u_2,...,u_n>$, $\prod_{i=1}^{n}(u_i+1)$ copies of the body are replicated, the loop control variables are adjusted for each copy, and the step value of the i^{th} loop is multiplied by u_i+1.

For each $v_j \in V$, where V is the set of tasks in the pre-unrolling loop, there are $\prod_{i=1}^{n}(u_i+1)$ tasks namely $v_j^{y_1,y_2,...,y_n}$, $(0 \le y_i \le u_i,\ 1 \le i \le n)$ in the post-unrolling loop. Figure 2 shows an example of loop unrolling when 2 nested loops are used. Consider the nested loops given in Figure (a), where the body of the loop has only one task, *v*. Figure (b) shows the loop resulting from unrolling the innermost loop once (u = < 0,1>) and the resulting tasks v^{00} and v^{01}. Similarly Figure (c) shows the loop resulting from unrolling the outermost loop once (u = <1,0>) and the resulting tasks v^{00} and v^{10}. Finally Figure (d) shows the loop resulting from unrolling both loops once each (u = <1,1>) and the resulting tasks v^{00}, v^{01}, v^{10}, and v^{11}. In this example, the loop upper bounds are assumed to be a multiple of two for simplicity.

The tasks that form the body of a loop can be represented by an acyclic directed graph called a *task graph*. A directed edge (i,j) between two tasks i and j exists if there is a data dependency between the two tasks which means that task j cannot start execution until it gets some input from task i after its completion. We define the task graph $G_0 = (V,E)$ that represents the body of a pre-unrolling loop, where V is the original set of tasks as defined above and E is the set of loop-independent data dependency edges given in DM. Thus the loop-carried data dependences are ignored in G_0 and the pre-unrolling graph can be simply scheduled by executing the iterations one after another without any overlap between iterations.

In order to consider the loop-carried dependences in a task graph, we define the task graph $G_u = (V',E')$ that represents the body of an unrolled loop using $u = <u_1,u_2,...,u_n>$ as unrolling vector, where V' is the set of all tasks generated in the post-unrolled loop and E' is the set of edges that represent loop-independent and loop-carried data dependence generated from DM. Scheduling the post-unrolling graph, G_u allows different iterations of the loop to overlap in execution. For example, the task graphs G_0 and G_u ,when u =

<1,1> for the DM and TSA given in Figure 1 are shown in Figures 3-a and 3-b, respectively. The upper portion of each node contains the task title while the lower portion contains the task size. The number next to an edge represents the message size to be passed through that edge.

```
for (i = 1; i ≤ 2*n1; i++)
        for (j = 1; j≤ 2*n2; j++)
            {
v               T(i,j);
            }

            (a)
```

```
for (i = 1; i ≤ 2*n1; i = i++)
        for (j = 1; j ≤ 2*n2; j = j+2)
            {
v00         T(i,j);
v01         T(i,j+1);

            }
            (b)
```

```
for (i = 1; i ≤ 2*n1; i = i+2)
        for (j = 1; j≤ 2*n2; j++)
            {
v00         T(i,j);
v10         T(i+1,j)
            }

            (c)
```

```
for (i = 1; i ≤ 2*n1; i = i+2)
        for (j = 1; j ≤ 2*n2; i = j+2)
            {
v00         T(i,j);
v01         T(i,j+1);
v10         T(i+1,j);
v11         T(i+1,j+1);
            }
            (d)
```

Figure 2 (a) Original Loop, (b) Innermost Loop is Unrolled Once (u = <0,1>), (c) Outermost Loop is Unrolled Once (u = <1,0>), and (d) Both Loops are Unrolled once each (u = <1,1>).

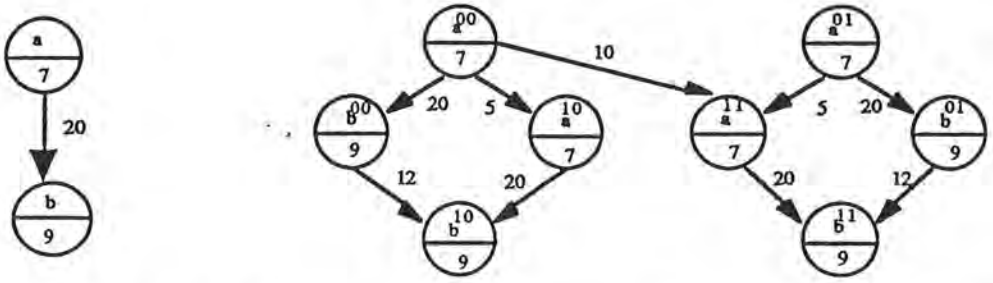


Figure 3 (a) G_0 (b) G_u (u = <1,1>), for the Loop Represented by DM and TSA Given in Figure 1.

4 Scheduling Post-Unrolling Loops

Because loop-carried dependences may allow several iterations of a set of loops to overlap in execution, loop unrolling can help to exploit the parallelism that might exist among different iterations. The basic idea is to unroll the loop using some iteration vector to allow dependencies among different iterations to appear in the task graph G_u as given above. That task graph is then scheduled on a given target machine using one of the techniques given in [3]. The iterations of the unrolled loop are executed one after another. This allows the set of tasks in the unrolled version to overlap in execution.

Three different cases should be considered in the program: 1) if the number of loop iterations is greater than and a multiple of the (unrolling value + 1), then no more iterations need to be executed after the unrolled loop, 2) if the number of loop iterations is greater than and not a multiple of the (unrolling value + 1), then the tasks in the remaining iterations can still be assigned to the same processing elements they were assigned to in the generated schedule, and 3) if the number of loop iterations is less than the unrolling value, then the tasks in all iterations are assigned to the same processing elements they were assigned to in the generated schedule.

Example 1

Consider the DM and the TSA given in Figure 4. Tasks **a** and **b** with task size = 15 as shown in the TSA are enclosed in a single loop (n = 1). The pre-unrolling task graph, G_0 is given in Figure 5a which shows that there is no loop-independent data dependence between the two tasks which means that within one iteration tasks **a** and **b** can run totally in parallel. Let's assume (just for the sake of discussion) that the upper bound of the loop is known to be 4. In the case when we have only one processor, tasks **a** and **b** run sequentially in any order (we assume that task **a** executes first) and this process is repeated 4 times with total execution time = 120 units of time as shown in Figure 5b.

Now suppose that we have two processors connected together and we are going to exploit only the parallelism within each iteration and neglect the parallelism that might take place if we unroll the loop. We run tasks **a** and **b** concurrently on processors p_1 and p_2, respectively so each iteration takes only 15 units of time. Since task **a** during iteration <i> receives data from task **b** during iterations <i-1> and task **b** during iteration <i> receives data from task **a** during iteration <i-1> and since tasks **a** and **b** are scheduled on different processors, a new iteration has to wait for 10 units of time before starting execution. Figure 5c shows that each iteration takes 15 units of time, however due to the

communication delay a new iteration can be initialized every 25 units of time and the 4 iterations take 90 units of time.

Figure 6a shows the task graph G_u, when the loop is unrolled once (u <1>). It can be easily noticed that scheduling tasks $\mathbf{a}^0$ and $\mathbf{b}^1$ together on one processor and tasks $\mathbf{b}^0$ and $\mathbf{a}^1$ on the other is the best way to schedule task graph G_u on two processors. Now we are able to initialize a new iteration every 30 units of time without having to wait for any communication and the total execution time is only 60 units of time, as shown in Figure 6b. By exploiting the parallelism without unrolling we get speedup = $1\frac{1}{3}$, however after unrolling the loop once, speedup = 2 (which is the maximum theoretical speedup we can get using two processors). It is not always the case that we have zero communication delay between iterations because it depends on the way we assign the tasks to the available processing elements.

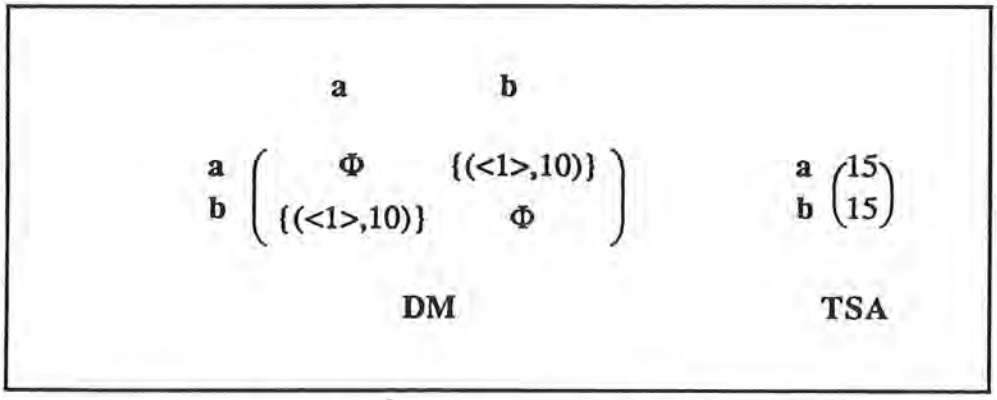


Figure 4 DM and TSA for Two Tasks **a** and **b** Enclosed in a Single Loop.

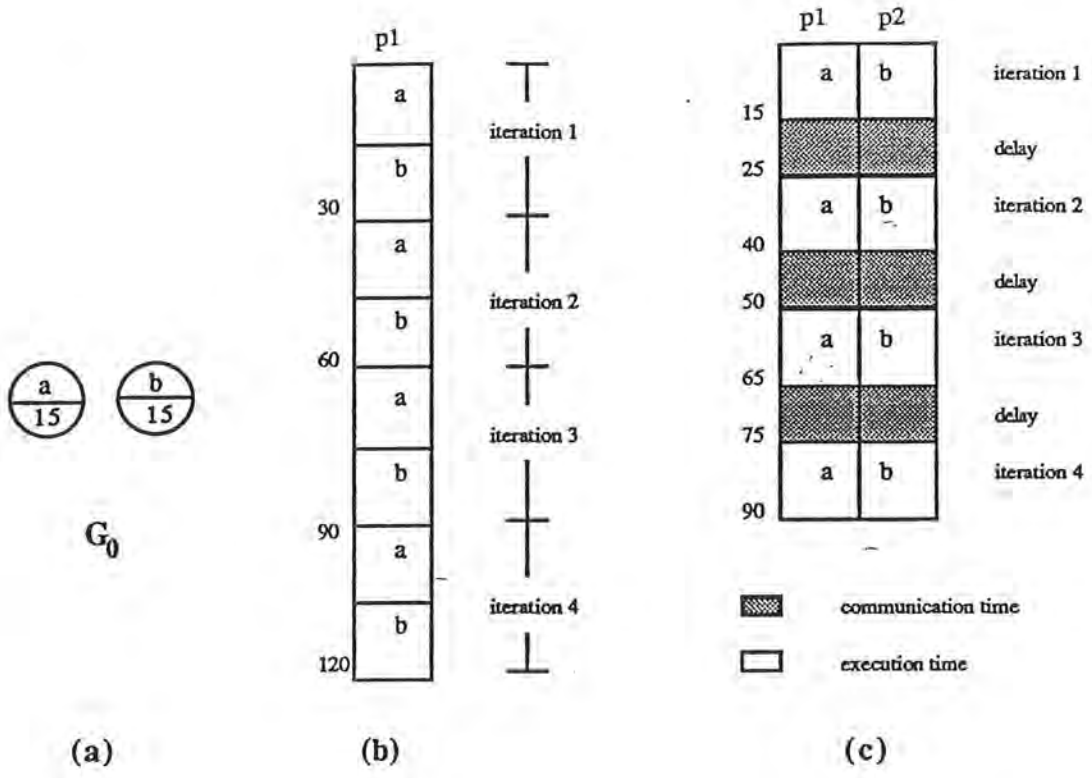


Figure 5 (a) Task Graph G_0, and (b)&(c) Gantt Charts Result From Scheduling Four Iterations of the Loop on One and Two Processing Elements, respectively.

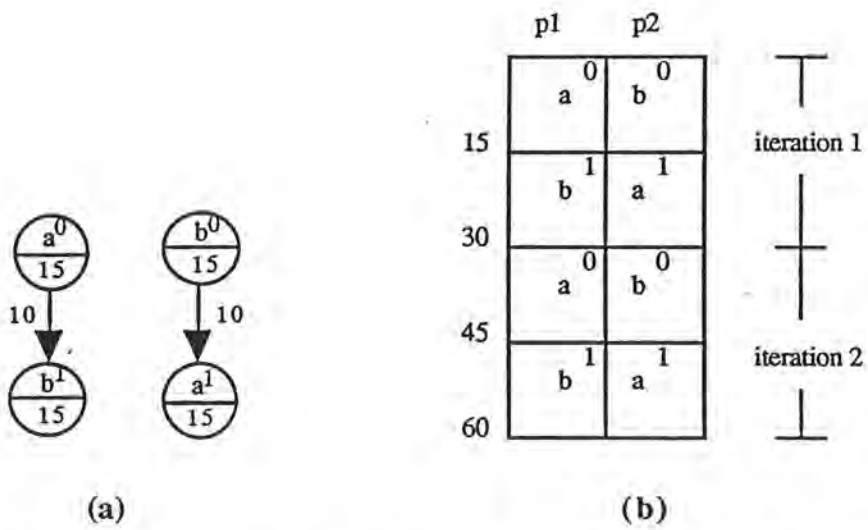


Figure 6 (a) Task Graph G_u, u = <1>, and (b) Gantt Chart Results From Scheduling Two Iterations of the Unrolled Loop on Two Processing Elements

Example 2

In the case of nested loops, the performance might differ depending on which loop is unrolled. Figure 7 shows the DM and the TSA for tasks **a** and **b** enclosed in 2 nested loops. In this example, we assume that the upper bound vector b = <2,4>. Figures 8a, b, c show the original loop, the loop after unrolling the outermost loop once, and the loop after unrolling the innermost loop once.

Figure 9a shows the task graph G_u, when the outermost loop is unrolled once (u = <1,0>). The Gantt chart given in Figure 9b shows that a new iteration has to wait 10 units of time because task **a** (**b**) during iteration (i,j) receives data from task **b** (**a**) during iteration (i,j-1) and they are located on different processors. That leads to total execution time equal to 110 units of time. On the other hand, when the innermost loop is unrolled once, no delay is needed and the total execution time is 80 units of time. Figure 10a shows the task graph G_u, when the innermost loop is unrolled once (u = <0,1>) and the Gantt chart is given in Figure 10b.

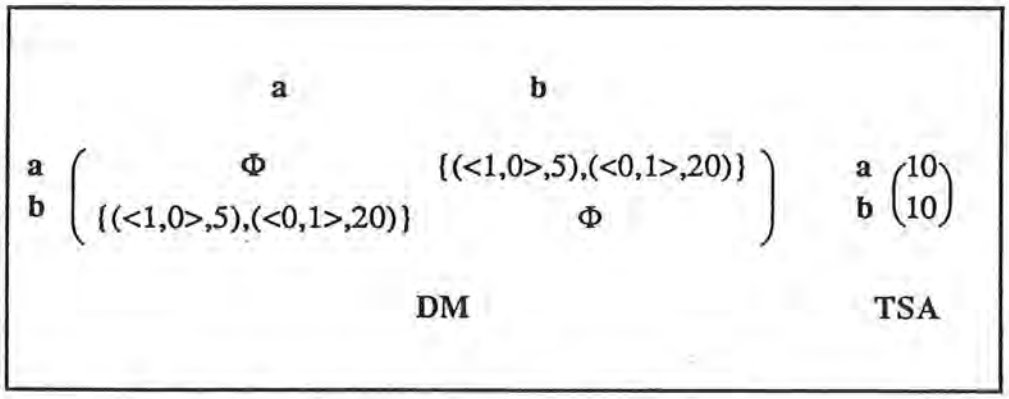
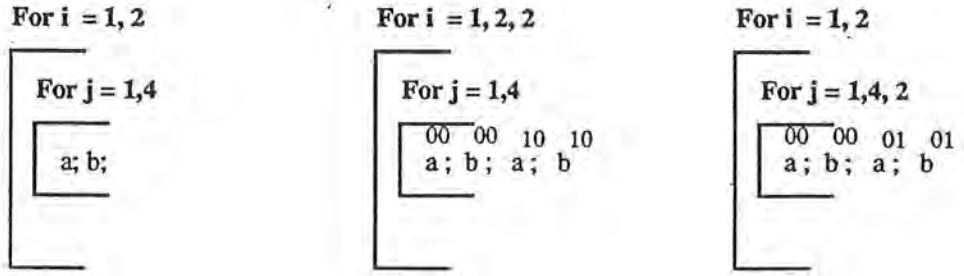
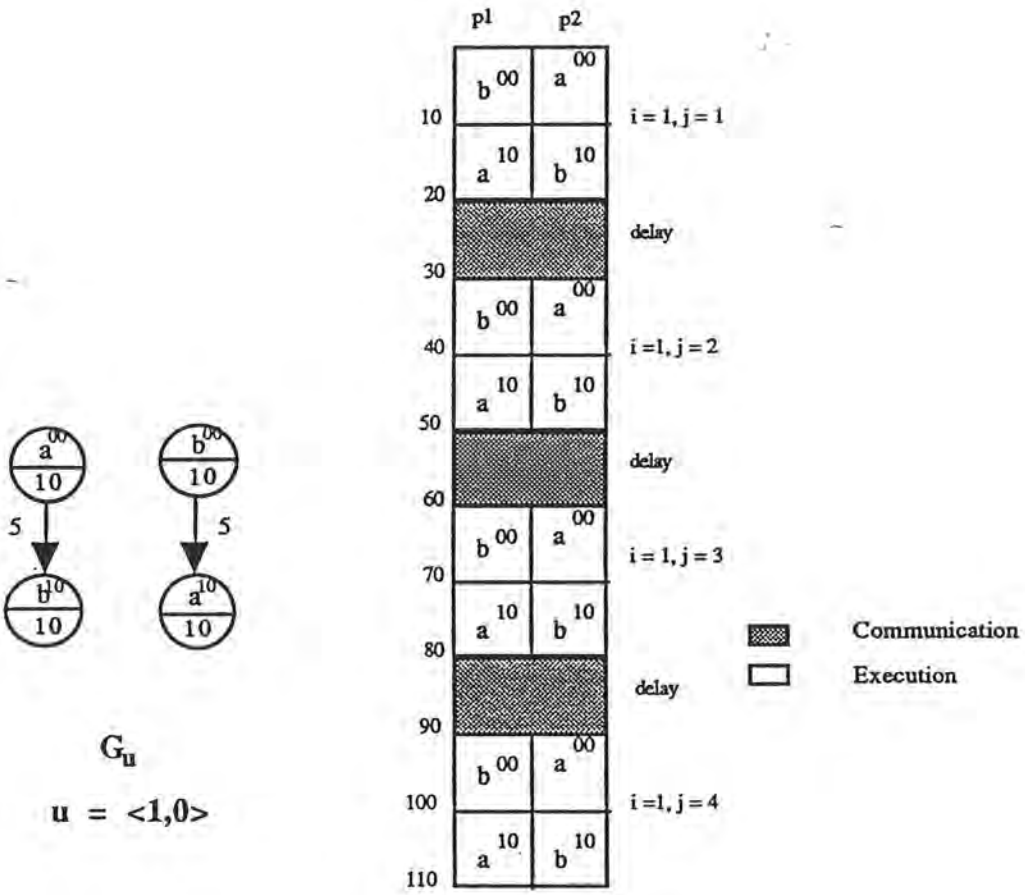


Figure 7 Two Tasks **a** and **b** Enclosed in Two Nested Loops Represented Using DM and TSA.



(a) (b) (c)

Figure 8 (a) Original Loop, (b) the Loop After Unrolling the Outermost Loop Once, and (c) the Loop After Unrolling the Innermost Loop Once.



(a) (b)

Figure 9 (a) Task Graph G_u, u = <1,0>, and (b)The Schedule After Unrolling the Outermost Loop Once.

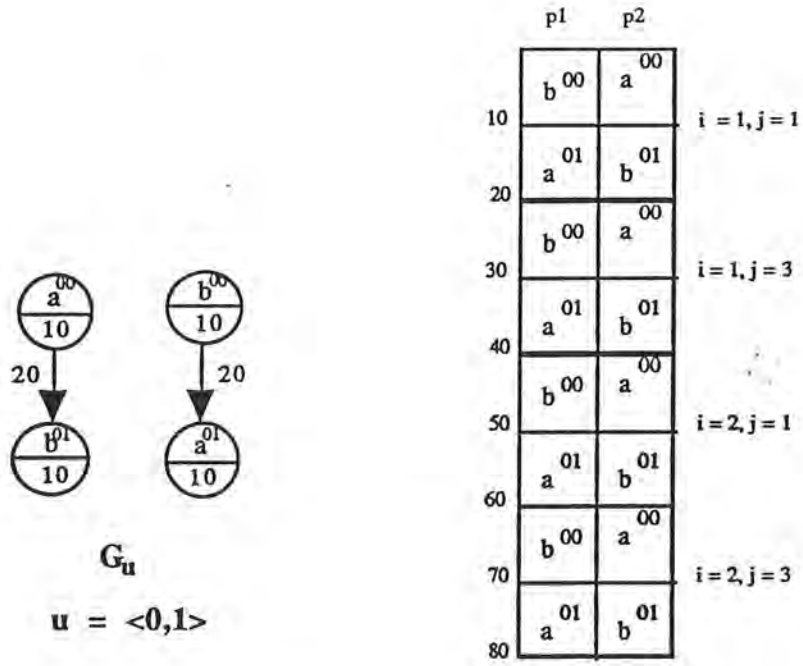


Figure 10 (a) Task Graph G_u, u = <0,1>, and (b) the Schedule After Unrolling the Innermost Loop Once.

These two examples show how loop unrolling reduces loop execution time and therefore improves performance. The examples also show the change in the loop execution time that results from using different loop unrolling vectors. In both examples we assumed that the communication time between any two tasks running on the same processor is negligible. We also assumed that the processing speed and the transfer rate in the target machine is always equal to one (this allows us to deal with the message size and the task size as units of time). These may not be realistic assumptions, however.

5 Loop Unrolling Optimization Problem

How many times should a loop be unrolled to speed up its execution? How should the tasks forming the body of the loop be scheduled onto a given arbitrary target machine? Our goal is to find the vector $u = \langle u_1, u_2, \ldots, u_n \rangle$ and schedule the tasks in the task graph G_u on a given parallel system such that the total loop execution time is minimized. This problem is an example of combinatorial minimization. The space over which the function is defined is n-dimensional discrete but very large, so it cannot be explored exhaustively. Therefore, we need an efficient optimization algorithm.

The most obvious optimization technique uses a greedy algorithm. Given some starting point, move downhill as far as possible. This leads to a local, but not necessarily a global, minimum. An example of this technique is Local Neighborhood Search [5].

Another optimization technique that avoids the pitfalls of finding a local minimum is simulated annealing. Simulated annealing is a probabilistic modification of traditional neighborhood search techniques [8].

Both approaches find a solution to a combinatorial optimization problem by starting with some solution and making a series of modifications to the solution. In neighborhood search algorithms, modifications which improve the solution by some given cost criterion are accepted and others are rejected. The acceptance criterion in simulated annealing is more complex. All modifications which lead to a better solution are accepted. All modifications which result in a poorer solution (higher cost) are accepted with probability $\exp(-\frac{\Delta E}{T})$ where ΔE is the difference between the costs of the solutions before and after the update, and T is a parameter known as temperature. Over time, the parameter T is slowly reduced, causing a reduction in the probability that a modification which results in a poorer solution will be accepted.

To make use of the local neighborhood or the simulated annealing methods, we must provide a description of possible system configuration, a generator of changes in the configuration, and an objective function whose minimization is the goal of the procedure. The unrolling vector can be used as our system configuration which can be updated to generate new system configurations. If the upper bound vector is known when the schedule is generated, a complete unrolling might be done to generate the task graph G_u for scheduling. If the nested loops are too large to unroll completely, then an exact formula for the time to execute the outermost loop is used as an objective function. If the upper bound vector is not known when the schedule is generated but is known before the loop begins execution, another formula that does not contain the loop upper bounds is needed to reflect the effect of using different values of the unrolling vector.

In section 5.1 we give the formulation of the problem for both local neighborhood search and simulated annealing methods. The objective function and loop completion time parameters are given in section 5.2.

5.1 Problem Formulation

The loop unrolling optimization problem can be formalized as follows:

1. *Configuration.* A configuration is a vector $u = <u_1,u_2,...,u_n>$, where $u_i \geq 0$; $1 \leq i \leq n$.

2. *Update.* The update consists of two types: a) An i is chosen randomly in the range [1,n] then u_i is changed according to some policy; or b) some u_i's (chosen randomly) are selected for change, $1 \leq i \leq n$.

3. *Objective Function.* The total loop execution time. (This will be explained in the next section).

4. *Schedule.* The local neighborhood search method keeps trying until some terminating condition happens. In the simulated annealing solution we choose a starting value for the temperature parameter T greater than the largest ΔE. We proceed downward in multiplicative steps each amounting to some decrease in T. We hold each new value of T constant for some number of changes in the unrolling vector, or for some number of successful moves, whichever comes first. When effort to further reduce E becomes sufficiently discouraging, we stop.

The problem now is how to choose the initial value of u. We define the *maximum distance vector* for a matrix DM to be $<m_1, m_2,..., m_n>$, where $m_i \geq x_i$, x_i is the i^{th} component in a distance vector I_d, I_d is the first component in an ordered pair p, $p \in DM[i,j]$, $1 \leq i \leq k$, and $1 \leq j \leq k$. It can be noticed that the maximum distance vector is the minimum unrolling vector that can uncover all the hidden loop-carried dependence edges in a loop. Considering those edges in the scheduling procedure can help in exploiting the parallelism that might exist among different iterations. Consequently, we use the maximum distance vector as the initial value of the unrolling vector u.

5.2 The Objective Function

Our goal is to minimize the total execution time of the loop. The completion time of the tasks forming the body of the loop as well as the communication delay among different iterations in the post-unrolling loop play an important role in computing the total execution time of a loop. In this section we show the algorithm we use to obtain the completion time of the body of a loop and the communication delay between different iterations. We then show the objective function used in the local neighborhood search and the simulated annealing methods.

Given a DM that represents data dependence among tasks enclosed in a set of n nested loops with an upper bound vector $b = <b_1,b_2,...,b_n>$. We assume that the loop is unrolled using $u = <u_1,u_2,...,u_n>$. Recall that G_u is a task graph that represents the body of an unrolled loop. We define the following:

$\tau_{u,i}$: is the time to execute the loop at level i. Notice that $\tau_{u,n+1}$ is the time to execute G_u only once, and that $\tau_{u,1}$ is the time to execute the outermost loop (the whole thing).

$\lambda_{u,i}$: is the communication delay between any two consecutive iterations of a loop at level i. Notice that $\lambda_{u,i} = 0$ when $u_i = b_i - 1$ because when we unroll a loop at the i^{th} level $b_i - 1$ times we generate all the instances of that loop.

Given a number of nested loops (n), unrolling vector (u), dependence matrix (DM), task size array (TSA), and a target machine description (M), we show how we obtain the parameters ($\tau_{u,n+1}$) and ($\lambda_{u,i}$, $1<i \leq n$).

We use MH to schedule the task graph G_u on the given target machine[3]. MH takes two inputs: 1) a description of the parallel program modules and their interactions in the form of a task graph, and 2) description of the target machine in the form of an undirected graph. MH produces as output a Gantt chart that shows the allocation of the program modules onto the target machine's processing elements and the execution order of tasks allocated to each processing element.

From the Gantt chart produced by MH, we can get $\tau_{u,n+1}$ (the time to run the tasks represented by the task graph G_u). The communication delay between any two consecutive iterations of a loop at level k ($\lambda_{u,i}$, $1 \leq i < n$) can also be figured out from the Gantt chart and the task graph G_u as the maximum time that a task has to wait until it receives a message from a task scheduled in some previous iteration. When the upper bound vector is unknown before execution time it becomes impossible to find $\lambda_{u,i}$, $1 \leq i < n$. However, we can always obtain the worst case communication delay by considering only one iteration for all levels greater than i (enclosed in a loop at level i).

5.2.1 Execution Time Formulas

Given a DM that represents data dependence among tasks enclosed in a set of n nested loops with an upper bound vector $b = <b_1,b_2,...,b_n>$ and an unrolling vector $u = <u_1,u_2,...,u_n>$. The time to execute the nested loop can be obtained as follows.

$$\tau_{u,1} = \left\lceil \frac{b_1}{u_1+1} \right\rceil (\tau_{u,2} + \lambda_{u,1}) - \lambda_{u,1} - \left(\left\lceil \frac{b_1}{u_1+1} \right\rceil - \frac{b_1}{u_1+1} \right) \tau_{u,2} \qquad (1)$$

Changing the unrolling vector u might change loop execution time. The best value of the unrolling vector would be the one that gives the shortest loop execution time. When the upper bound vector b is known before execution time, formula 1 can be used to compute the execution time. Different values of the unrolling vector u can be tried in order to find the shortest execution time. When the upper bound vector b is not known, another formula

that does not contain b is needed to compare the effect of using different values of the unrolling vector.

Formula (1) can be bounded as: $\tau_{u,1} \leq \lceil \frac{b_1}{u_1+1} \rceil (\tau_{u,2} + \lambda_{u,1})$ (2)

$\tau_{u,1}$ can be approximated as: $\tau_{u,1} \approx \frac{b_1}{u_1+1} (\tau_{u,2} + \lambda_{u,1})$ (3)

The time to execute a loop at level i can be approximated as follows.

$\tau_{u,i} \approx \frac{b_i}{u_i+1} (\tau_{u,i+1} + \lambda_{u,i}),\ 1 \leq i \leq n$ (4)

Using formula (4), the time to execute the outermost loop can be obtained as follows:

$$\tau_{u,1} \approx \frac{b_1}{u_1+1} \left(\frac{b_2}{u_2+1} \left(\frac{b_3}{u_3+1} \left(\ldots\ldots\left(\frac{b_n}{u_n+1} (\tau_{u,n+1} + \lambda_{u,n}) + \ldots\ldots\right) + \lambda_{u,3}\right) + \lambda_{u,2}\right) + \lambda_{u,1}\right) \quad (5)$$

$$\approx \prod_{i=1}^{n} \frac{b_i}{u_i+1} (\tau_{u,n+1} + \lambda_{u,n}) + \prod_{i=1}^{n-1} \frac{b_i}{u_i+1} \lambda_{u,n-1} + \ldots\ldots\ldots + \prod_{i=1}^{2} \frac{b_i}{u_i+1} \lambda_{u,2} + \frac{b_1}{u_1+1} \lambda_{u,1} \quad (6)$$

(When $\langle u_1, u_2, \ldots, u_n \rangle = \langle b_1-1, b_2-1, \ldots, b_n-1 \rangle$, $\lambda_{u,i} = 0$, $1 \leq i \leq n$ and $\tau_{u,1} \approx \tau_{u,n+1}$).

Since $1 \leq u_i < b_i$ then $\frac{b_i}{u_i+1} \geq 1$, $1 \leq i \leq n$, $\tau_{u,1}$ can be bounded by:

$$\left(\prod_{i=1}^{n} \frac{b_i}{u_i+1}\right) (\tau_{u,n+1} + \sum_{i=1}^{n} \lambda_{u,i}) \quad (7)$$

Notice that when n = 1 (single loop) the two formulas (6) and (7) are identical which means:

$\tau_{u,1} = \frac{b_1}{u_1+1} (\tau_{u,2} + \lambda_{u,1})$ (8)

5.2.2 Objective Function Evaluation

The execution time as given in formula (7) is considered our objective function. However since the upper bound vector b is usually not given before execution time and since $\prod_{i=1}^{n} b_i$ does not change during the optimization search then the objective function can be given as:

$$E = \left(\prod_{i=1}^{n} \frac{1}{u_i+1}\right)\left(\tau_{u,n+1} + \sum_{i=1}^{n} \lambda_{u,i}\right) \quad (9)$$

The objective function can be evaluated as follows:

given:

number of nested loops (n), unrolling vector (u),

dependence matrix (DM), task size array (TSA), machine (M)

procedure:

- build_graph G_u
- call MH (G_u,M) and get $\tau_{u,n+1}$ and ($\lambda_{u,i}$, $1 \le i \le n$)
- compute objective function $E = \left(\prod_{i=1}^{n} \frac{1}{u_i+1}\right)\left(\tau_{u,n+1} + \sum_{i=1}^{n} \lambda_{u,i}\right)$

In comparing the two methods, we found the performance of simulated annealing was either much better or much worse than the performance of local neighborhood search. The simulated annealing algorithm can be tuned by varying some parameters in the algorithm such as starting temperature, temperature reduction policy, and exit condition. Also the local neighborhood search method can be tuned by varying the exit condition. Since the two algorithms can be tuned through a set of parameters and since the performance of the two methods differ from one task graph (represented using DM and TSA) to another and from one target machine to another, we implemented both techniques in one tool so that both methods can be tried. By tuning the set of parameters associated with each algorithm, a near optimal solution can be reached. We believe that finding the best unrolling vector and the best schedule should be achieved through iterative interaction between parallel program designers and the loop unrolling tool.

6 Examples

In this section we illustrate the local neighborhood search and the simulated annealing methods in the loop unrolling techniques through two examples. The examples show two cases; a single loop and three nested loops. In both examples, we use a hypercube of four processing elements as the target machine. In each case information about the unrolling vector, the value of the objective function, number of nodes and edges in the post-unrolling loops, and the average degree ($\frac{\text{number of edges}}{\text{number of nodes}}$) in the unrolled loops are given in tabular form. We also give some curves that show the change in the objective function with each move in the unrolling space.

In both methods, we start with the maximum distance vector as initial unrolling vector. The search space is restricted using some maximum values for the elements of the unrolling vector. The local neighborhood search method keeps trying until some termination condition is reached (in these examples we restricted the number of tries to some number depending on the size of the search space). In the simulated annealing method, we choose a starting value for the temperature parameter T greater than the largest ΔE (some random runs were made to find the range of ΔE). We proceed downward in multiplicative steps each amounting to some decrease in T. We hold each new value of T constant for some number of changes in the unrolling vector, or for some number of successful moves, whichever comes first. When effort to further reduce E becomes sufficiently discouraging, we stop.

Single Loop (n = 1)

Figure 11 shows the DM and the TSA matrices for four tasks. It can be noticed that there is no loop-independent data dependence edges which means that the four tasks are completely independent within the same iteration. However, loop-carried data dependence edges are shown in DM. Since the four tasks are enclosed in a single loop, the search space is only one dimension and exhaustive search can be applied to show the behavior of the objective function (E) over the search space. Table 1 shows the move number, the unrolling vector (the unrolling vector is just one element; n = 1), the value of the objective function (E), number of nodes, number of edges, and the degree in the post-unrolling loop.

Figure 12 shows the change in the objective function (E), when unrolling increases in steps of one. It can be noticed that the value of E changes up and down in a way that increases the chance of being stuck in a local minimum. We thought that simulated annealing would be a solution to this problem, however we tried both methods. Table 2

and Figure 13 show the results of applying the local neighborhood search. We restricted the number of tries to 20. This method made only three successful moves out of 20 tries and the answer was best u = <12> with E = 6.7610.

Table 3 shows the simulated annealing solution. We proceeded downward in multiplicative steps each amounting to a 20 percent decrease in T. We held each new value of T constant for 8 changes in the unrolling vector or for 4 successful moves. We also restricted the number of tries to 16. Simulated annealing made 9 successful moves and the answer was best u = <12> with E = 6.7610. Figure 14 shows the change in E with each successful move. Of course exhaustive search can be used to search the whole space in the same number of tries. However we used this example with the same restricted search space just to show both methods. Tables 2 and 3 show that the two methods have found the same answer which happened to be the minimum in the restricted search space.

Three Nested Loops (n = 3)

Figure 15 shows DM and TSA for three tasks enclosed in three nested loops. The sizes of the three tasks are 9, 3, and 6 respectively. In this example we restricted the number of tries to 40 and restricted the search space to 1000 points by using maximum value = 11 for the three elements of the unrolling vector. Table 4 shows the local neighborhood search solution and the answer was best u = < 8,7,8> with E = 3.5216 and number of nodes and edges = 1944 and 6246 respectively. Figure 16 shows that this method made only 6 successful moves out of 40. The simulated annealing solution is given in Table 5 and the best u = < 7,7,8> with E = 3.6770 and number of nodes and edges = 1728 and 5492 respectively. Figure 17 shows that simulated annealing made 9 successful moves in the search.

	1	2	3	4
1	{(<4>,8)}	{(<1>,3),(<4>,20)}	{(<4>,9),(<3>,7),(<2>,8)}	{(<3>,1)}
2	{(<4>,14),(<1>,2)}	{(<3>,4)}	{(<3>,17),(<4>,3)}	Φ
3	{(<3>,11)}	{(<2>,20)}	{(<3>,6)}	Φ
4	{(<1>,4)}	{(<3>,16),(<2>,13)}	{(<2>,11)}	{(<2>,10)}

DM

	TSA
1	9
2	2
3	13
4	4

Figure 11 DM and TSA for Four Tasks (1, 2, 3,and 4) Enclosed in a Single Loop.

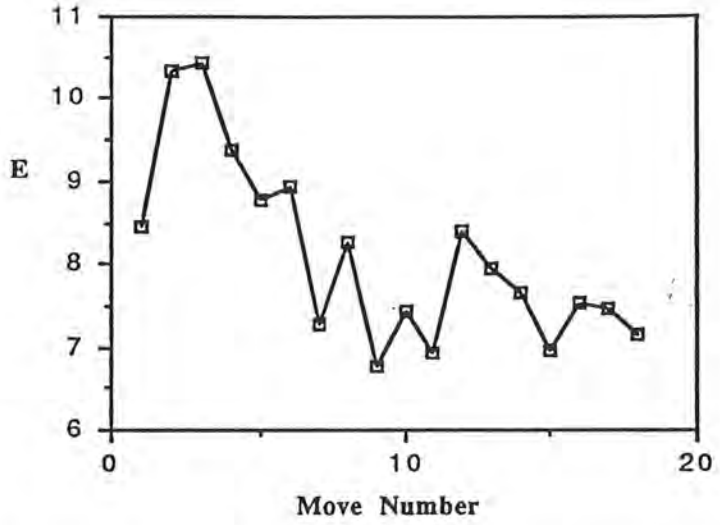


Figure 12 The Change in E when the Loop is Unrolled From 4 to 21 Exhaustively (n=1)

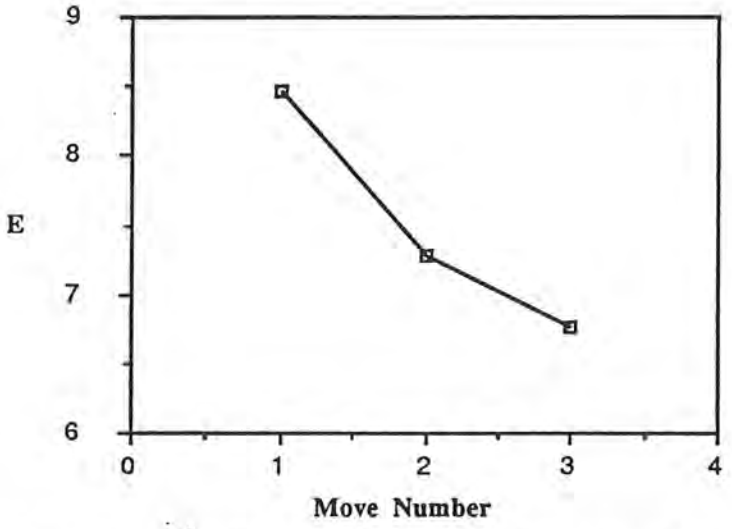


Figure 13 Three Successful Moves in Local Neighborhood Search (n=1)

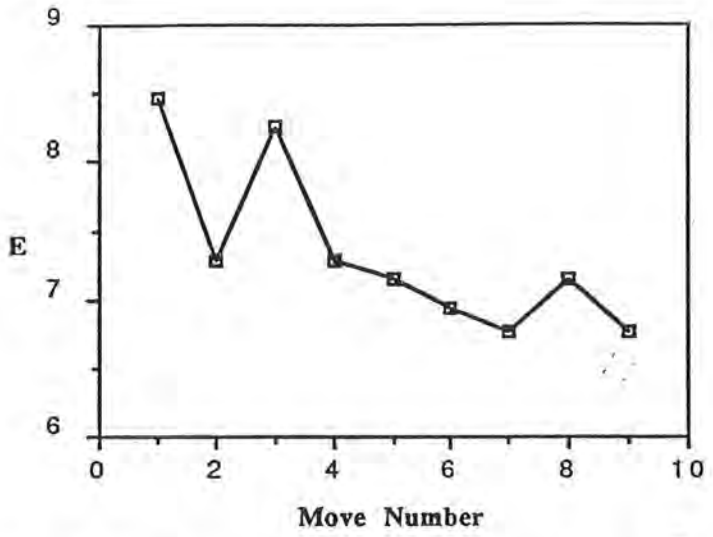


Figure 14 Nine Successful Moves in Simulated Annealing (n = 1)

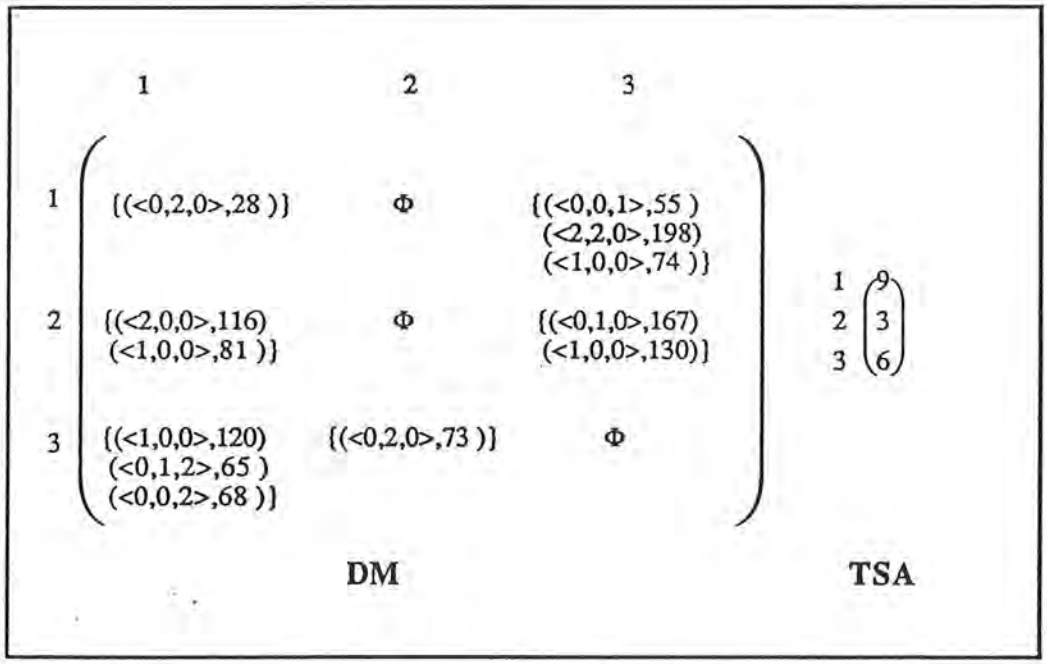


Figure 15 DM and TSA for Three Tasks (1, 2, and 3) Enclosed in Three Nested Loops.

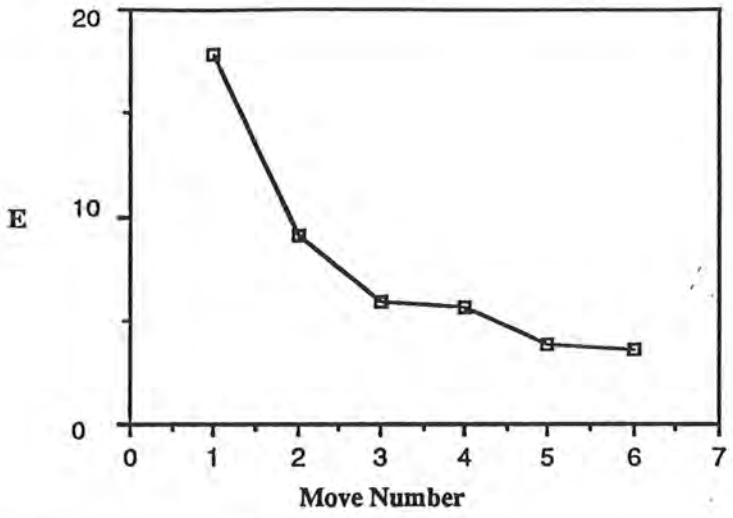


Figure 16 Six Successful Moves in Local Neighborhood Search (n = 3)

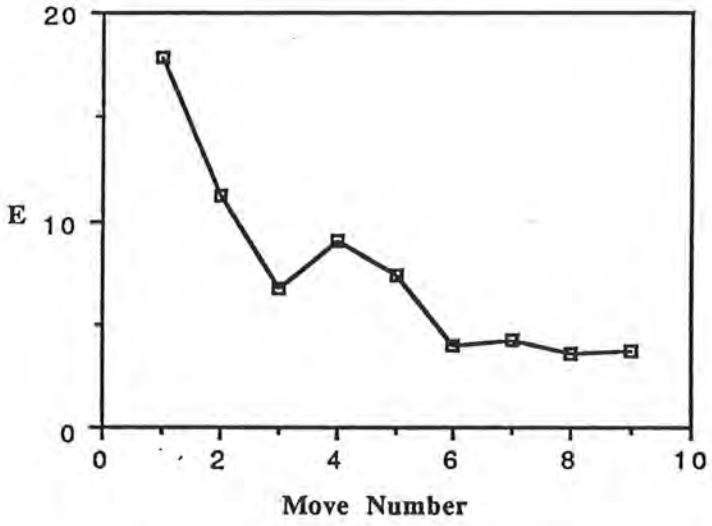


Figure 17 Nine Successful Moves in Simulated Annealing (n = 3).

move number	u_1	E	number of nodes	number of edges	degree
1	4	8.4565	20	46	2.300
2	5	10.3300	24	66	2.750
3	6	10.4288	28	86	3.071
4	7	9.3737	32	106	3.312
5	8	8.7700	36	126	3.500
6	9	8.9364	40	146	3.650
7	10	7.2772	44	166	3.773
8	11	8.2577	48	186	3.875
9	12	6.7610	52	206	3.962
10	13	7.4212	56	226	4.036
11	14	6.9300	60	246	4.100
12	15	8.3901	64	266	4.156
13	16	7.9426	68	286	4.206
14	17	7.6643	72	306	4.250
15	18	6.9424	76	326	4.289
16	19	7.5321	80	346	4.325
17	20	7.4706	84	366	4.357
18	21	7.1389	88	386	4.386

Table 1 Exhaustive Search (n = 1)

move number	u_1	E	number of nodes	number of edges	degree
1	4	8.4565	20	46	2.300
2	10	7.2772	44	166	3.773
3	**12**	**6.7610**	**52**	**206**	**3.962**

Table 2 Local Neighborhood Search (n = 1)

move number	u_1	E	number of nodes	number of edges	degree
1	4	8.4565	20	46	2.300
2	10	7.2772	44	166	3.773
3	11	8.2577	48	186	3.875
4	10	7.2772	44	166	3.773
5	21	7.1389	88	386	4.386
6	18	6.9424	76	326	4.289
7	12	6.7610	52	206	3.962
8	21	7.1389	88	386	4.386
9	**12**	**6.7610**	**52**	**206**	**3.962**

Table 3 Simulated Annealing Solution (n = 1)

move number	u_1	u_2	u_3	E	number of nodes	number of edges	degree
1	2	2	2	17.8148	81	153	1.889
2	2	4	5	9.0000	270	651	2.411
3	2	7	8	5.9120	648	1722	2.657
4	2	6	8	5.6031	567	1485	2.619
5	8	6	8	3.8342	1701	5391	3.169
6	**8**	**7**	**8**	**3.5216**	**1944**	**6246**	**3.213**

Table 4 Local Neighborhood Search (n = 3)

move number	u_1	u_2	u_3	E	number of nodes	number of edges	degree
1	2	2	2	17.8148	81	153	1.889
2	2	8	2	11.2592	243	567	2.333
3	7	2	8	6.7500	648	1732	2.673
4	2	4	5	9.0000	270	651	2.411
5	2	8	5	7.3271	486	1263	2.599
6	7	8	7	3.9184	1728	5512	3.190
7	7	8	6	4.1726	1512	4780	3.161
8	7	8	8	3.6049	1944	6244	3.212
9	**7**	**7**	**8**	**3.6770**	**1728**	**5492**	**3.178**

Table 5 Simulated Annealing Solution (n = 3)

7 Conclusion

In this paper we have introduced a new way to represent parallel program tasks and their precedence relations. The new representation allows us to express loop-carried data dependences among tasks that cannot be represented using ordinary task graphs. We also introduced a new technique for scheduling unrolled loops onto arbitrary target machines in a way that minimizes the completion time. The scheduler considers communication delays among tasks on different processors which makes this method particularly useful for loop unrolling on distributed-memory target machines.

We have integrated both local neighborhood search and simulated annealing methods in one tool to find: 1) the best unrolling vector for a particular set of tasks when they run on a particular target machine and 2) the Gantt chart that indicates the allocation and the order of the tasks in the unrolled loop on the available processing elements. We showed two simple examples for a single loop and three nested loops to show the performance of the two methods in each case. It is recommended that both methods be tried since their