A Path-Driven Loop Scheduling Mapped onto Generalized Hypercubes

Hai Jiang (Member, IEEE),
A. T. Chronopoulos (Senior Member, IEEE)
Department of Computer Science
Wayne State University,
University of Texas, San Antonio
haj@cs.wayne.edu,
atc@cs.utsa.edu

G. Papakonstantinou,
P. Tsanakas
Dept. of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece

Abstract

One of the important issues in automatic code parallelization is the scheduling and mapping of nested loop iterations to different processors. The optimal scheduling problem is known to be NP-complete. Many heuristic static and dynamic loop scheduling techniques have been studied in the past.

Here we propose a new static loop scheduling heuristic method called path-driven scheduling, under the assumption that the loop dependence graph has been generated. This method clusters tasks according to the directed paths on the dependence graph and assigns them to processors in the target architecture. We make comparisons with the free scheduling and the refined free scheduling algorithms [8]. We schedule three widely used nested loops on a generalized hypercube architecture. Our algorithm exhibits the lowest communication cost compared to the other two algorithms, while the execution cost is the same for all three algorithms.

Key Words: Loop scheduling, path-driven scheduling, path generation, path mapping, generalized hypercube.

1 Introduction

The problem of scheduling parallel program modules onto multiprocessor computers is known to be NP-complete in general cases [5]. Many heuristic scheduling algorithms have been proposed in the literature [3], [5], [6], [8], [9], [10], [11], [12], [13]. Some researchers introduced priority based algorithms, such as list [6] and free scheduling [8]. Most of these are suitable for shared memory machines. When these algorithms are applied to distributed memory machines, performance will degrade quickly because of the communication cost. On distributed memory machines, a multi-stage method, *clustering scheduling*, is more practical [6]. It implements scheduling in two steps: *task allocation* and *task ordering*. First task allocation clusters tasks according to *dominant sequences* on processors and make task priority deterministic. Then task ordering could be achieved easily on the assigned processors.

The motivation behind this work is to find out an efficient algorithm to schedule loop tasks on distributed memory system. Given a loop task graph, how tasks are clustered will affect the whole performance of the scheduled program. Once the clustering is applied, we can not change much in global scheduling. In clustering, just grouping high communication tasks together is not sufficient for reducing parallel execution time. Task execution order and global parallel running time still should be taken into account.

Directed paths on the graph reflect the traces of data flowing streams. And clusters are the sets of nodes on a task graph. If we cluster tasks according to data flow (or directed paths), actually we could combine the considerations of communication reducing and task execution ordering. This makes task allocation and ordering no longer distinct internally. Within such clusters, each node contains exactly one immediate predecessor and one immediate successor. So task ordering is optimized and naturally scheduling is the consequence of data flow.

We introduce a heuristic path-driven scheduling and mapping algorithm (P-D) in this paper to obtain the optimized task clusters according to data flows and map them onto the Processing Elements (PEs) of a target machine, the generalized hypercube [1], [4], [14], heuristically. The rest of the paper is organized as follows. The P-D scheduling algorithm is described in Section 2. In Section 3, we discuss the mapping on the PEs of the generalized hypercube. Simulation results are given in Section 4. Finally conclusions are presented in Section 5.

2 Path-Driven Scheduling

A path-driven algorithm is suitable for problems with invariant task computation cost. For such a nested loop

problem, since the P-D has taken the links into consideration, the communication cost is not considered in the clustering phase but it is dealt with in the mapping phase.

2.1 Directed Acyclic Graph model for nested loops

For a general program, we must detect the nested loops and generate a directed acyclic graph (DAG) from them. Our scheduling will be based on the ensemble DAG of all loops.

Throughout this paper, we consider n-way nested loops with l_j and u_j as the lower and upper bounds of the *jth* loop. Without loss of generality, we assume l_j and u_j are integer-valued constants and $l_j \leq u_j$ for all $1 \leq j \leq n$. The iteration space (or index set) is expressed by $J^n = \{(i_1, i_2, \cdots, i_n) | \ l_j \leq i_j \leq u_j$, for $1 \leq j \leq n\}$. In the following sections, we only consider loop-carried dependences [5]. So we treat all statements at each iteration as a single element of J^n. And each iteration can be referenced by its index vector $\bar{i} = (i_1, i_2, \ldots, i_n)$.

For a data dependence, we use the following terminology and notations:

1. *Dependence vector:* If a variable *x* is defined or used at iteration $\bar{i}$, and redefined or used at iteration $\bar{j}$, then there is a dependence vector $\bar{d}$ between these iterations based on the variable *x*, where $\bar{d} = (d_1, d_2, \ldots, d_n)^t$, $d_k = j_k - i_k$, for $1 \leq k \leq n$.

Using dependence vectors, we can denote all data dependences among any iterations in this iteration space. All three types of dependences, *flow dependence*, *antidependence* and *output dependence* will be expressed in the same way.

2. *Dependence matrix:* $D = [\bar{d_1}, \bar{d_2}, \ldots, \bar{d_m}]$, for $m \in Z^+$ is the set of all dependence vectors in the iteration space.

When parallelizing a nested loop, all dependences have to be respected. In a flow dependence, variables are defined at an earlier iteration $\bar{i}$ and used at a later iteration $\bar{j}$. So data must be passed from $\bar{i}$ to $\bar{j}$. In an output dependence case, variables are defined at $\bar{i}$ and are redefined at $\bar{j}$. This just indicates a clear relation between the two iterations. Their execution orders in a sequential program should be respected in the parallel program scheduling. In an antidependence case, variables used at $\bar{i}$ as input are redefined at $\bar{j}$ ($\bar{i} < \bar{j}$). Actually antidependence is some kind of a restriction. Iteration $\bar{i}$ uses some variables defined before $\bar{i}$ in sequential program. And iteration $\bar{j}$ overwrites them. Thus no matter how we schedule iteration $\bar{i}$, it has to be executed before iteration $\bar{j}$.

To detect an antidependence, we just need to check the dependence vector $\bar{d}$. Let d_{i_0} be the first non-zero negative entry, i.e., $d_i = 0$, $1 \leq i < i_0$, and $d_{i_0} < 0$, then the dependence vector $\bar{d}$ indicates an antidependence. Since antidependence deals with a relation between an iteration and a later iteration with redefinitions of some common variables, it has to be changed to indicate proper execution order. One possible way is to convert it to a corresponding flow dependence. Then we can assume there exists a pseudo-data flow between two iterations. To change the direction of dependence vector $\bar{d}$, we replace it with a vector $\bar{d'} = (d'_1, d'_2, \ldots, d'_n)^t$, $d'_i = -d_i$, $1 \leq i \leq n$. From now on, we will not distinguish between flow, pseudo-flow or output dependence. We will just refer to them as dependences.

From the iteration space and the modified dependence matrix, we can generate a DAG $G(V, E)$ with :

Vertices

$$V = \{ \ v(i_1, i_2, \ldots, i_n) \ | \ \bar{i} = (i_1, i_2, \ldots, i_n), \ l_l \leq i_l \leq u_l, \ for \ 1 \leq l \leq n\}$$

and edges

$$E = \{ \ e_{v(i_1,i_2,\ldots,i_n),v(j_1,j_2,\ldots,j_n)} \ | \ \bar{i} = (i_1, i_2, \ldots, i_n), \ \bar{j} = (j_1, j_2, \ldots, j_n), \text{ such that } \exists \ \bar{d} \in D \text{ with } \bar{j} = \bar{i} + \bar{d} \ \}$$

Each iteration is a vertex on the DAG, because we only consider loop-carried dependences. If a dependence exists between two iterations, there is an edge between the two corresponding vertices [8].

2.2 Generation of Scheduled Paths

The P-D schedules and maps tasks based on certain parameters. Let $u, v \in V$ and the set of positive integers is denoted by Z^+, we define the following:

1. *Predecessor Set:* $pre(v) = \{ \ u \mid e_{u,v} \in E \ \}$
2. *Successor Set:* $suc(v) = \{ \ u \mid e_{v,u} \in E \ \}$
3. *Earliest Schedule Level:*

$$esl(v) = \begin{cases} 1 & \text{if } pre(v) = \phi \\ \max_{u \in pre(v)}(esl(u)) + 1 & \text{otherwise} \end{cases}$$

4. *Graph Path:* A set of vertices which are connected by a sequence of edges on the DAG.
5. *Critical Path Length:*

$$cpl(G(V, E)) = \max_{v \in V}(esl(v))$$

Critical paths are the longest graph paths on the DAG. They dominate the execution time. The critical path length is at least the same as the loop parallel time.

6. *Latest Schedule Level:*

$$lsl(v) = \begin{cases} cpl(G(V, E)), & \text{if } suc(v) = \phi \\ \min_{u \in suc(v)}(lsl(u)) - 1, & \text{otherwise} \end{cases}$$

7. *Task Priority Value:* Each task is assigned a priority value τ for selection in clustering. Smaller values indicate higher priority. For $v \in V$,

$$\tau(v) = lsl(v) - esl(v).$$

8. *Task Level Set:* Tasks are classified by levels according to their *Earliest Schedule Level*. All tasks which could be executed in parallel at time slot $i \in Z^+$ will be scheduled at tls_i, where
$$tls_i = \{v \mid esl(v) = i, v \in V, i \in Z^+\}.$$
9. *Scheduled Path (sp):* Each sp is a set of vertices selected by the P-D. Each vertex of the DAG belongs to only one sp. Critical Scheduled Paths (csp) are the longest $sp's$. Tasks of each sp will be mapped to a particular PE for execution.
10. *Path Communication Set (pcs):* Each scheduled path communicates with others through this set of edges. For sp_i, $i \in Z^+$,
$$pcs_i = \{e_{u,v} \mid e_{u,v} \in E, (u \in sp_i \text{ and } v \in V - sp_i) \mid\mid (v \in sp_i \text{ and } u \in V - sp_i)\}.$$
11. *Path Exchange Set:* Two scheduled paths might need to exchange data through a set of edges just between them. For $sp_i, sp_j, i, j \in Z^+$,
$$pes_{i,j} = \{e_{u,v} \mid e_{u,v} \in E, (u \in sp_i \text{ and } v \in sp_j) \mid\mid (v \in sp_i \text{ and } u \in sp_j)\}.$$

For a given DAG, the P-D assigns tasks to different PEs and sets up the execution order. Generally, a task scheduling algorithm consists of three steps [6]:

1. Partition the DAG into a set of distinct task clusters.
2. Reorder the tasks inside their clusters.
3. Map the clusters to PEs maintaining communication efficiency.

Scheduled paths in the P-D are similar to task clusters. The difference is that tasks on scheduled paths must have data dependence relations, but this is not necessarily true for the tasks in clusters. Scheduled paths are generated according to data dependences. Initially a scheduled path is a trace of data flow. Data stream passes through it from the start to the end. And at the same time, task orders have been fixed on the sp because of the dependences. During the mapping, the costs of all communication links on the same scheduled paths have been zeroed. So internally (on each PE) the three separate scheduling steps are collapsed into one step.

Given a DAG, the length of critical scheduled paths will be constant regardless how they might be generated. Longer scheduled paths contain more tasks, and possibly more communication links since they are generated by data dependences. In general if fewer PEs are used and if the scheduled paths are longer, the expected efficiency of the execution will be higher.

Scheduled paths are created to cover all the tasks in a DAG. Some tasks are shared by several graph paths, as in a DAG there are some joint nodes, (e.g. fork and join points). These shared tasks could only belong to one scheduled path. Once a scheduled path is extracted, those task vertices will not be isolated and cut off from the DAG as in linear clustering methods [10]. They are sharable and stay on the DAG to enable data flows passing through for the detection of other scheduled paths. For example, if the parent task is an unprocessed task, and its immediate children have been assigned to some scheduled paths by their other parents, these selected children could be used as pseudo-tasks on the new scheduled path to let the parent find its unprocessed grand-children and place them on the same new scheduled path. Both parent and its grand-children have data links with the tasks in between (parent's children). Putting them on the same scheduled path will benefit the future mapping in reducing communication distance. So the P-D could generate longer scheduled paths naturally and reduce the potential difficulty in merging them afterwards.

One could use heuristic methods in getting a suboptimal scheduled path. For the safest solution, we should enumerate all possible graph paths, then pick out the scheduled paths in decreasing order of their lengths. But actually this method is not practical. Both time and space costs are as high as task duplication scheduling. We propose a heuristic strategy of scheduled path generation as follows:

1. *Determine the earliest schedule levels:* Traverse the DAG from top down to determine the earliest scheduling levels (esl) of all tasks. If a task is independent, its $esl = 1$. If a task only depends on tasks with $esl = 1$, its $esl = 2$, and so on. If a task depends on some tasks with $esl \leq i$ (i.e. at least one task has $esl = i$), this task's $esl = i + 1$.

2. *Determine the latest schedule levels:* The tasks with the biggest esl are actually the exit-nodes on critical scheduled paths. Then lsl(task) = esl (task). From bottom up, we traverse DAG again to determine the lsl for all tasks. For a task, if its successors exist with $lsl \geq i$ (at least one task has lsl(task) = i), this task's $lsl = i - 1$.

3. *Group tasks:* Tasks should be grouped and placed into different task level sets (tls). This is done according to esl, *i.e.*, if a task's $esl = i$, it's placed in tls_i. Those tasks which have the same values of esl and lsl are called *Critical Path Tasks* (*cpt*). Each tls_i contains at least one *cpt*.

4. *Generate scheduled paths:* All critical scheduled paths are identified before the non-critical scheduled paths. There could be several critical scheduled paths. First, we scan the *task level set* table from top down (i.e., from tls_1 to tls_{cpl}) to select the start of a scheduled path, then identify other tasks from data dependences. If tls_i is not empty, we choose one critical path task at random. If none is available, we choose one with smallest priority value τ (which implies it belongs to a longer scheduled path), and mark it. Then we check the successors of the selected task and select an unmarked task with the smallest τ. If none is available, we choose one as a pseudo-task, and check its successors until exit-tasks are reached. We mark the whole scheduled path, and then restart to generate another one until all tasks have been marked.

In this strategy, we can see that each scheduled path contains at least one task. There might be some redundant pseudo-tasks. They help to detect more tasks along the same data streams. Once a scheduled path is created, its tasks could be eliminated from the scheduling record, but not from the DAG.

Algorithm (Scheduled Path Generation)

Input: DAG $G(V, E)$ with $tls(i), 1 \leq i \leq cpl$ and $\tau(j),\ 1 \leq j \leq |V|$

Output: List $sp_1, sp_2, \cdots, sp_m$, for some $m \geq 1$

1. $m = 0$
2. **for** $i = 1$ **to** cpl
3. **while** not all tasks in tls_i are marked "selected" **do**
4. $m = m + 1; sp_m = \phi$
5. select v such that $\tau(v) = \min\{\tau(u), \forall u \in tls_i$ and u is unmarked $\}$
6. $sp_m = sp_m \cup \{v\}$, and mark v "selected"
7. $j = i$
8. **while** $j < cpl$ **do**
9. **if** $\exists\ u \in suc(v)$ and u is unmarked
10. **then** select unmarked x, $\tau(x) = \min\{\tau(y),$ $\forall$ unmarked $y \in suc(v)\}$
11. $sp_m = sp_m \cup \{x\}$
12. mark x "selected"
13. **else** select a marked $x \in suc(v)$ at random
14. **endif**
15. $v = x$
16. $j = j + 1$
17. **endwhile**
18. **endwhile**
19. **endfor**

Task time-stamping: Once we get all scheduled paths, we know which $sp's$ tasks belongs to and their sequential orders. But we still need to assign each task a time-stamp to indicate at which step it could be run on a particular PE. From the length of $csp's$, we know the number of parallel execution steps. Then along each sp, there exists the same number of *execution time slots*. Each *time slot* can contain only one task. Task time-stamping is needed to place each task into a reasonable time slot on its sp according to the dependences in the DAG. The $esl's$ generated during the scheduled path generation are used for the time-stamping to fill out the *time slots*.

Scheduled path optimization: Up to now, a sp is a trace of data flow. If it is not contiguous, it must be sharing some tasks with others and the shared tasks are not on the current sp. It's costly to let all $sp's$ always find some available tasks beyond those shared ones during the scheduled path generation period. Then several broken shorter scheduled paths could be created.

If each task on a sp_i is scheduled at an earlier esl than all those on a sp_j, these two scheduled paths will be good candidates for merging. If the last task on sp_i is connected to the first task on sp_j through some other vertices in the DAG, we call them *related sp's*. The benefits of merging them are:

- To reduce the number of PEs for higher efficiency.
- To make mapping easier for a non-fully connected topology which might not be rich in communication channels. Merging scheduled paths can reduce such requests because the shared tasks need to exchange data with other tasks on both original shorter $sp's$.

For general cases, any two scheduled paths whose tasks are on different lsl, i.e. with different esl, can be merged together. But this doesn't guarantee that they are *related sp's*. If they are *non-related sp's*, they can not always reduce the communication cost because it depends on the underlying non-fully connected topology. This problem needs further investigation.

3 Mapping on Target Machine

In theory, all $sp's$ can be mapped onto any PEs on a target physical machine. In a shared memory machine model, the underlying PE network topology could be thought to be fully connected. Each PE is similar to all others regarding the communication links and locations. In this case, mapping is easy. Any scheduled path could be mapped onto any PE.

For a distributed memory machine, communication latency plays a big role. Different mappings might result in different communication costs, and make longer the total parallel execution time. The heuristic mapping tries to map scheduled paths to nearby PEs if they share more communication links.

3.1 Detection of Scheduled Path Relationship

The communication frequency of sp_i with all other $sp's$ will be denoted by $|pcs_i|$. This indicates the number of links shared between sp_i and all others. The sp with bigger $|pcs_i|$ should be mapped earlier than others. This could make easier the mapping of subsequent $sp's$.

For sp_i and sp_j, the communication frequency between them will be denoted by $|pes_{i,j}|$. Then $sp's$ with bigger $|pes_{i,j}|$ should be mapped to nearby PEs in order to reduce the communication cost between them.

During mapping, these communication frequencies are used to determine the mapping order and location.

3.2 Generalized Hypercube

The Generalized Hypercube topology architecture has been studied in [1], [14]. Due to its high degree of node

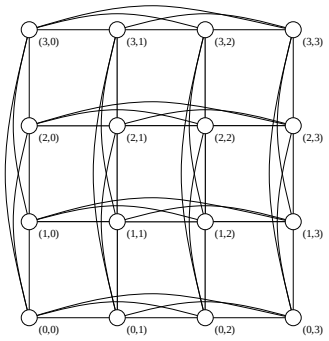

Figure 1: The 2-D generalized hypercube $GH_{(2,4)}$

connectivity, it offers a viable alternative to the shared memory and other distributed memory architectures. Many past and current massively parallel computers are based on meshes or k-ary n-cubes (e.g. Cray T3E, Intel Paragon, Tera and the design in [2]). Unlike the mesh or k-ary n-cubes, the generalized hypercubes ($GH_{(n,k)}$, where: n= number of dimensions and k= number of nodes in each dimension) have k fully interconnected nodes in each dimension. As a result they have a very low diameter and a very high bisection width. However, the number of interconnection links in each dimension increases linearly with the number of nodes k [1].

GH is a symmetric topology. All PEs have the same numbers and structures of links with others. Figure 1 illustrates the 2-D $GH_{(n,k)}$, with $n = 2$ and $k = 4$. The diameter of a 2-D $GH_{(n,k)}$ is only 2 and the bisection width is $k^3/4$.

3.3 Mapping on GH

We choose the 2-D generalized hypercube ($GH_{(2,k)}$) as a network topology model to show how to map scheduled paths to a physical machine. Between two PEs on the same dimension of $GH_{(2,k)}$, each communication takes a single time unit. Otherwise, it takes twice this time.

We distinguish two cases: (a) n_p (number of paths) $\leq$ p (number of PEs) (b) $n_p > p$. In (b) several scheduled paths are mapped onto each PE, which results in zeroing all communication links between them. But the parallel time will be longer. In such cases, the mapping is split into global and local mappings. Each PE has several scheduled path slots. Initially, the global mapping is applied to map a scheduled path on a PE by selecting one with the biggest $|pcs_i|$. Then for the *local mapping*, $|pes_{i,j}|$ should be used to find an unselected scheduled path which has the most communication links with the already mapped ones on the current PE, and map the new one to a spare slot. This local mapping process like this should continue until the local slots are full or locally mapped scheduled paths have no communication need with any others.

Because of the peculiarity of the $GH_{(2,k)}$, the global mapping strategy could be formulated as the following algorithm, whereas the local mapping remains the same. To illustrate the global mapping algorithm, some terminology for $GH_{(2,k)}$ is introduced.

- *Link Counters for Rows (or Columns):* For each newly selected sp_i, each GH row (or column) maintains a parameter $lcr_i[j]$ (or $lcc_i[j]$) for the total numbers of links between sp_i and all scheduled paths which have been assigned on these rows (or columns). For each GH row (or column) j,

$$lcr_i[j] = \sum_{\forall\ sp_l \text{ mapped on row } j} |pes_{i,l}|$$

$$lcc_i[j] = \sum_{\forall\ sp_l \text{ mapped on column } j} |pes_{i,l}|$$

We next present the *global mapping* strategy on the 2-D GH.

1. Sort $sp's$ decreasingly according to $|pcs_i|$.
2. Select an unselected sp_i with the biggest $|pcs_i|$.
3. Sort GH rows and columns according to lcr_i and lcc_i.
4. Find an available GH node (m_0, n_0) such that $lcr_i[m_0] + lcc_i[n_0] = \max\{lcr_i[m] + lcc_i[n]$, for any GH node $(m, n)\}$
5. Map the selected sp_i to the GH node (m_0, n_0).
6. Activate the local mapping process to map more $sp's$ to available local slots.
7. Repeat step 2, 3, 4, and 6 until all $sp's$ have been mapped.

4 Simulated Experiments

In this paper, the parallel computation time is determined by the critical path length unless the target machine runs out of PEs. The performance of the algorithms will be evaluated by the *number of inter-communication links* ($nicl$) among tasks on PEs. For a fully connected topology, $nicl$ is just the number of links among all scheduled paths. For a real target machine, $nicl$ is the summation of communication costs among $sp's$ on PEs expressed as the sum of the total numbers of hops.

For the 2-D GH, on each dimension, PE nodes are fully connected. Any communication along the same dimension will have weight = 1. Other communication between two scheduled paths with different rows and columns will have weight = 2. The summation of them will indicate the total communication cost.

Next, free scheduling, refined free scheduling and path-driven scheduling are applied to a simple nested loop example, the matrix multiplication and the Jacobi loops. The comparison of $nicl's$ will be reported on the fully connected topology and 2-D GH.

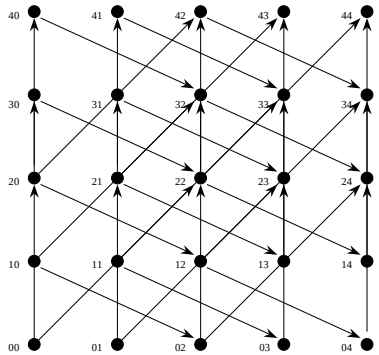

Figure 2: The DAG for the simple nested loop example

4.1 Simple nested loop example

We consider the following nested loop example:

For $i = 0$ to 4 do
  For $j = 0$ to 4 do
    $a[i, j] = a[i, j-2] + a[i-2, j+1]$
      $+a[i-2, j-2]$
  end
end

The dependence matrix is

$$D = [\bar{d_1}, \bar{d_2}, \bar{d_3}] = \begin{bmatrix} 0 & 2 & 2 \\ 2 & -1 & 2 \end{bmatrix}$$

and the dependence graph is shown in Figure 2.

The *task level sets* are shown in Table 4.1.

esl	**Tasks**					
1	00	01	10	11		
2	02	03	12	13	20	30
3	04	14	21	22	31	32
4	23	24	33	34	40	41
5	42	43				
6	44					

Table 4.1. The $tls's$ for the simple nested loop example

If we apply the P-D on this example, the $sp's$ are generated as in Table 4.2. The $(\ )^*$ entries are shared tasks.

Path No.	**Tasks** at each time step					
	t_0	t_1	t_2	t_3	t_4	t_5
1	00	02	04	23	42	44
2			21	40	$(42)^*$	$(44)^*$
3	01	03	22	24	43	
4	10	12	14	33		
5	11	13	32	34		
6		20			$(42)^*$	$(44)^*$
7		30	$(32)^*$	$(34)^*$		
8			31	$(33)^*$		
9				41	$(43)^*$	

Table 4.2. The $sp's$ for the simple nested loop example

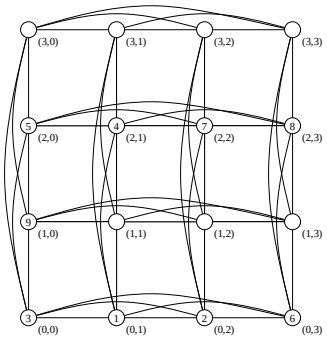

Figure 3: Mapping the simple nested loop example on $GH_{(2,4)}$

The total communication costs of the three scheduling algorithms are listed in Table 4.3 and the mapping result on $GH_{(2,4)}$ is illustrated on Figure 3.

Scheduling Algorithm	$nicl$	
	Fully connected	$GH_{(2,4)}$
Free	26	36
Refined Free	22	28
Path-Driven	20	20

Table 4.3. Comparison of communication costs for the simple nested loop example

4.2 Matrix multiplication

We consider the matrix multiplication problem:

For $i = 0$ to 3 do
  For $j = 0$ to 3 do
    For $k = 0$ to 3 do
    $c[i, j] = c[i, j] + a[i, k] * b[k, j]$;
    end
  end
end

The total communication costs of the three scheduling algorithms are listed in Table 4.4.

Scheduling Algorithm	$nicl$		
	Fully connected	$GH_{(2,8)}$	$GH_{(2,3)}$
Free	115	135	158
Refined Free	115	135	158
Path-Driven	92	96	63

Table 4.4. Comparison of communication costs for the Matrix Multiplication

4.3 Jacobi loops

We consider the Jacobi loops:

```
For i = 0 to 2 do
 For j = 0 to 2 do
  For k = 0 to 2 do
   For l = 0 to 2 do
    a[i,j,k,l]= 1/6 × ( a[i-1,j+1,k,l]
                +a[i,j-1,k,l] + a[i-1,j,k+1,j]
                +a[i,j,k-1,l] + a[i-1,j,k,l+1]
                +a[i,j,k,l-1] );
   end
  end
 end
end
```

The total communication costs of the three scheduling algorithms are listed Table 4.5.

Scheduling Algorithm	$nicl$		
	Fully connected	$GH_{(2,8)}$	$GH_{(2,3)}$
Free	246	311	367
Refined Free	246	311	367
Path-Driven	202	220	171

Table 4.5. Comparison of communication costs for the Jacobi loops

5 Conclusion

In this paper, we study a path-driven scheduling and mapping, from the DAG generation to the target machine mapping. This is particularly useful for distributed memory systems where communication cost is higher than computation cost. We consider a 2-D Generalized Hypercube as a target machine. This method is general enough, and without any extensions, it could be applied to non-uniform nested loops and other distributed memory target machines.

We have implemented the proposed algorithm and tested it. We made comparisons of our algorithm with two other existing algorithms by simulation on three widely used loops. Our algorithm outperforms the other algorithms in terms of estimated communication time while all three algorithms have the same estimated execution time.

Future research could be taking variable computation and communication costs into consideration.

Acknowledgement: This work was supported in part by NSF Grant ASC-9634775, by a CRAY/SGI and in part by the PENED-95 research project of the Hellenic Secretariat of Research and Technology.

References

[1] L.N. Bhuyan and D.P. Agrawal, Generalized Hypercube and Hyperbus Structures for a Computer Network, *IEEE Trans. Comput.*, Vol. 33, No. 4, 1984, pp. 323-333.

[2] W.J. Dally, et. al., The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms, *IEEE Micro*, Vol. 12, Apr. 1992, pp. 23-39.

[3] S. Darbha and D. P. Agrawal, Optimal Scheduling Algorithm for Distributed-Memory Machines, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 9, No.1, 1998, pp. 87-95.

[4] P. Fragopoulou, S. G. Akl and H. Meijer, Optimal communication Primitives on the Generalized Hypercube Network *Journal of Parallel and Distributed Computing*, 32, 173-187, 1996.

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

[6] A. Gerasoulis and T. Yang, A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors, *Journal of Parallel and Distributed Computing*, 16, 1992, pp. 276-291.

[7] C. H. Huang and P. Sadayapan, Communication-Free Hyperplane Partitioning of Nested Loops, *Journal of Parallel and Distributed Computing*, 19, 1993, pp. 90-102.

[8] N. Koziris, G. Papakonstantinou and P. Tsanakas, Optimal Time and Efficient Space Free Scheduling for Nested Loops, *The Computer Journal*, 39(5), 1996, pp. 439-449.

[9] Y-K. Kwok and I. Ahmad, Dynamic Critical-Path Scheduling : An Effective Technique for Allocating Task Graphs onto Multiprocessors, *IEEE Trans. on parallel and Distributed Systems*, 7(5), 1996, pp. 506-521.

[10] G. C. Sih and E. A. Lee, A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures, *IEEE Trans. on Parallel and Distributed Systems*, 4(2), 1993, pp. 75-87.

[11] W. Shang and J. Fortes, Time Optimal Linear Schedules for Algorithms with Uniform Dependencies, *IEEE Trans. Comput.*, 40(6), 1991, pp. 723-742.

[12] J. Sheu and T. Tai, Partitioning and Mapping Nested Loops on Multiprocessor Systems, *IEEE Trans. Parallel Distr. Systems*, 24, 1991, pp. 430-439.

[13] B. Shirazi, M. Wang and G. Pathak, Analysis and Evaluation of Heuristic Methods for Static Scheduling, *Journal of Parallel and Distributed Computing*, 10, 1990, pp. 222-232.

[14] S.G. Ziavras, H. Grebel, and A.T. Chronopoulos, A Low-Complexity Parallel System for Gracious, Scalable Performance. Case Study for Near PetaFLOPS Computing, *6th Symp. Frontiers Massively Parallel Computation*, Special Session on NSF/DARPA New Millennium Computing Point Designs, 1996, pp. 363-370.