Applications and Performance Analysis of A Compile–Time Optimization Approach for List Scheduling Algorithms on Distributed Memory Multiprocessors

Yeh–Ching Chung
Department of Information Engineering
Feng Chia University
Taichung, Taiwan 40724, ROC
e–mail : ychung@pine.iecs.fcu.edu.tw

Sanjay Ranka*
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244–4100
(315) 443–4457
e–mail : ranka@top.cis.syr.edu

Abstract – In [5], we have proposed a compile–time optimization approach, the *bottom–up top–down duplication heuristic* (BTDH), for static scheduling of *directed–acyclic graphs* (DAGs) on *distributed memory multiprocessors* (DMMs). In this paper, we discuss the applications of BTDH for *list scheduling algorithms* (LSAs). There are two ways to use BTDH for LSAs. (1) BTDH can be used with a LSA to form a new scheduling algorithm (LSA/BTDH). (2) It can be used as a pure optimization algorithm for a LSA (LSA–BTDH). We have applied BTDH with two well known LSAs, the *highest level first with estimated time* (HLFET) and the *earlier task first* (ETF) heuristics. We have performed extensive simulation to study the performance of BTDH for LSAs. Three parameters, *graph parallelism* (GP) of a DAG [19], the ratio of average communication cost to average computation cost (CCR) of a DAG [5], and the processor number (PN) of a DMM, are simulated. From the simulation, we have the following conclusions. (I) Given a DAG, the GP of the DAG can accurately predict the number of processors to be used such that a good scheduling length and a good resource utilization (or efficiency) can be achieved simultaneously. (II) In terms of speedups, we have LSA/BTDH ≥ LSA–BTDH ≥ ETF ≥ HLFET. Experimental results of scheduling FFT programs, which are written in *Single Program Multiple Data* (SPMD) programming approach, on NCUBE–2 are also presented. The results confirm our simulation results and show that the speedups of LSA/BTDH and LSA–BTDH are better than the speedups of LSAs.

1. Introduction

The main purpose of using parallel computers is to reduce the execution time of application programs. Optimal execution of application programs on parallel computers depends on the methods of partitioning an application program into tasks (the partitioning problem) and scheduling tasks on a parallel computer (the scheduling problem). The main aspects of the partitioning problem are (1) how to partition an application program into tasks while exploiting as much parallelism as possible; (2) how to determine the size of tasks (grain size) such that a better scheduling length can be produced by a scheduling algorithm. Once an application program has been partitioned, it can be represented, in general, by a *directed–acyclic graph* (DAG). In a DAG, nodes denote tasks and an arc from node u to node v represents the data dependency between the two nodes, i.e. node v can not be executed until the execution of node u has been completed. Weights are associated with nodes and arcs to represent the computation cost (proportional to the time to execute the task) and the communication cost (proportional to the number of message units to be transferred), respectively. The scheduling problem is to assign tasks of a DAG to processors of a parallel computer such that the execution time of a DAG is minimized. This problem is also known as the *multiprocessor scheduling problem* (MSP).

It has been shown that an algorithm for solving MSP falls into the class of NP–complete problems [33]. Therefore, many heuristic approaches are used to find satisfactory sub–optimal solutions [1]–[32]. The most well known approach for MSP is the *list scheduling algorithm* (LSA) [22] [23] [25]. The underlying assumption of LSA is that the interprocessor communication overhead of a computing system, such as processor communication or memory contention, is negligible. Under this assumption, LSA can produce near optimal solutions for most instances. However, this assumption is not valid for *distributed memory multiprocessors* (DMMs) where interprocessor communication overhead is an important factor of system performance and is typically not negligible. In fact, LSA is a load balancing heuristic. It tries to distribute computation load to processors as evenly as possible and does not consider the communication overhead. It has been shown that LSA algorithms produce poor scheduling results when interprocessor communication overhead is not negligible [3] [11]. Therefore, many approaches have been proposed for MSP with interprocessor communication overhead [1]–[21].

In [5], we have proposed a compile–time optimization approach, *bottom–up top–down duplication heuristic* (BTDH), for static scheduling of DAGs on DMMs. The underlying concept of BTDH is properly duplicating some tasks on processors such that the earliest start time of tasks on processors can be reduced. Therefore, a better scheduling length can be achieved. In

[5], we also compared BTDH with another task duplication heuristic DSH [11]. The major drawback of DSH is that, when a task T_n is scheduled on a processor P_x, the duplication process is applied only to those *predecessors* (will be defined in Section 2) which can be inserted in the idle time slot between the finish time of the task in front of T_n and the earliest start time of T_n; and do not increase the earliest start time of T_n. It is possible that the insertion of some of the predecessors of T_n will increase the earliest start time of T_n at a particular stage of the duplication process. But, the insertion of those predecessors will eventually decrease the earliest start time of T_n at a later stage of the duplication process. To overcome the drawback of DSH, BTDH allows the duplication of a predecessor T_i of a task T_n even thought the duplication of T_i in front of T_n will increase the earliest start time of T_n. The simulation results in [5] show that BTDH outperforms DSH, especially when the ratio of the average communication cost to the average computation cost is large.

In this paper, we will discuss the applications of BTDH for LSAs. There are two ways to use BTDH for LSAs. (1) BTDH can be used with a LSA to form a new scheduling algorithm (LSA/BTDH). (2) It can be used as a pure optimization algorithm for a LSA (LSA–BTDH). We have applied BTDH with two well known LSAs, the *highest level first with estimated time* (HLFET) [22] and the *earliest task first* (ETF) heuristics [8]. We have performed extensive simulation to study the performance of BTDH for LSAs. Three parameters, *graph parallelism* (GP) of a DAG [19], the ratio of average communication cost to average computation cost (CCR) of a DAG [5], and the processor number (PN) of a DMM, are simulated. From the performance analysis, we have the following conclusions. (I) Given a DAG, the GP of the DAG can accurately predict the number of processors to be used such that a good scheduling length and a good resource utilization (or efficiency) can be achieved simultaneously. (II) In terms of makespans, we have LSA/BTDH ≥ LSA–BTDH ≥ ETF ≥ HLFET. Experimental results of scheduling FFT programs, which are written in *Single Program Multiple Data* (SPMD) programming approach, on NCUBE–2 are also presented. The results confirm our simulation results and show that the speedups of LSA/BTDH and LSA–BTDH are better than speedups of LSAs.

In Section 2, the computational and the architectural models used in this paper are described. The optimization approach BTDH will be described briefly in Section 3. In Section 4, applications of BTDH for LSAs will be described in detail. The performance analysis of BTDH for LSAs will be given in Section 5 by using simulation approach. In Section 6, experimental results by using LSAs with BTDH to schedule FFT programs on NCUBE–2 are presented.

2. The Computational and the Architectural Models

In this paper, we consider scheduling of *static* (the number of tasks of a DAG is fixed during the execution), *non–preempted* (the execution of a task can not be interrupted once it starts), *with communication delay* (interprocessor communication overhead is not negligi-

ble), and *duplicated* (a task may have several copies on processors) MSP on a DMM. An application program is modeled as a directed–acyclic graph (DAG) $G = \{T, A\}$, where $T = \{T_1, T_2, \ldots, T_n\}$ is a set of n tasks and A is a set of arcs between tasks which define a partial order or precedence constrain ($<$) on T such that arc a_{ij} directed from task T_i into task T_j implies that T_i must precede T_j ($T_i < T_j$) in execution. In this paper, we do not address the issue of partitioning an application program into a DAG. However, for the experimental results shown in Section 6, we will briefly describe how to partition and transform an application program into a DAG. Every task T_i in a DAG is associated with a positive number, denoted by $\mu(T_i)$, which represents the computation cost of task T_i. Every arc a_{ij} is also associated with a positive number, denoted by $\eta(T_i, T_j)$, which represents the number of message units sent from task T_i to task T_j. T_i is a *predecessor* of T_j and T_j is a *successor* of T_i if there exists a path from T_i to T_j. T_i is an *immediate predecessor* of T_j and T_j is an *immediate successor* of T_i if there is an arc directed from T_i to T_j. A task without immediate predecessors is called a *source task* and a task without immediate successors is called a *sink task*.

An example of a DAG is shown in Figure 2. In Figure 2, the underlined number represents the computation cost of a task and the italic number beside an arc a_{ij} denotes the number of message units sent from task T_i to task T_j. For example, the computation cost of T_6 is equal to 6, i.e., $\mu(T_6) = 6$ and the number of message units sent from task T_4 to T_7 is equal to 10, i.e., $\eta(T_4, T_7) = 10$. T_1, T_2, and T_3 are predecessors of T_6 and T_3 is an immediate predecessor of T_6. T_1 is a source task and T_6, T_9, T_{10}, and T_{11} are sink tasks.

In a distributed memory multiprocessor, a processor communicates with other processors through message–passing. To characterize a DMM, a parameter $\tau(P_i, P_j)$ is introduced to represent the time to transfer a message unit from processor P_i to P_j. For real cases, especially for small size of problems, the setup time between two processors may have some effect on the makespans of scheduling algorithms. However, in our simulation model, we assume that the setup time is much smaller than the time to transfer a message unit from one processor to another. Therefore, it is negligible. A DMM is then defined as $S = (P, \tau)$, where $P = \{P_1, P_2, \ldots, P_m\}$ is a set of m processors. By varying the values of $\tau(P_i, P_j)$, the architectural model can be used to model several types of networks such as a fully connected network, a local area network, or a hypercube. We make the following assumptions regarding the functions of our architectural model.

1. Every processor in a DMM is identical.
2. The intra–processor communication overhead is negligible, i.e., $\tau(P_i, P_i) = 0$.
3. The communication subsystem is contention free.
4. A processor can send messages to some or all processors in a DMM simultaneously.
5. The system overhead, such as initialization of a send communication primitive, is negligible.

In real machine, such as the NCUBE–2 on which we present some experimental results, some of the above assumptions may not be valid. Assumption 2 is a realistic

```
algorithm BTDH(Tₙ, Pₓ)
1. repeat
2.     { e_time = e(Tₙ, Pₓ), T_top = previous(Tₙ, Pₓ), T_end = Tₙ, weight = 0.
3.       idle_time = e(T_end, Pₓ) - f(T_top, Pₓ).
4.       loop
5.         { if ((∃ Tᵢ ∈ θ(Tₙ, Pₓ)) and (Tᵢ is not scheduled on Pₓ))
6.           then { Duplicate Tᵢ in front of Tₙ on Pₓ.
7.                  Recompute the earliest start time of tasks from Tᵢ to T_end on Pₓ.
8.                  if (e(T_end, Pₓ) ≤ e_time) then { Tₙ = Tᵢ, exit. }
9.                  weight = weight + μ(Tᵢ).
10.                 if (weight < idle_time) then Tₙ = Tᵢ.
                    else { Remove all the tasks between T_top and T_end.
11.                        e(T_end, Pₓ) = e_time.
12.                        Tₙ = next(T_end, Pₓ), recompute e(Tₙ, Pₓ), exit. }
                    }
13.         else if (Tₙ ≠ T_end) then Tₙ = next(Tₙ, Pₓ).
            else { Remove all the tasks between T_top and T_end.
14.                e(T_end, Pₓ) = e_time.
15.                Tₙ = next(T_end, Pₓ), recompute e(Tₙ, Pₓ), exit. }
16.         } forever.
17.     } until (Tₙ = ∅).
18. return(the earliest start time of the last task scheduled on Pₓ).
end_of_algorithm_BTDH
```

Figure 1 : Algorithm BTDH.

---

assumption as the cost of transferring something within a processor is equivalent to data copying (or changing of a few pointers). This cost is negligible as compared to sending messages across the network.

Assumption 3 is useful for estimating the cost of sending a message from one processor to another without taking into account the contention due to other messages. This assumption is a good approximation for most architectures till the maximal bandwidth required by a problem is much less than the total available bandwidth.

Assumptions 4 and 5 are necessary for the reduction of time complexity of our mapping algorithms. This is because if the system overhead (the setup time before a non–blocking send is returned) is considered then the scheduling algorithms need to take into account which message needs to be sent first (in case a task has outdegree greater than 1). Such a decision increases the complexity of the algorithm. By ignoring the system overhead, Assumption 4 is automatically true for architectures which support non–blocking send (such as NCUBE–2). Further, if the grain size of the problem is such that the CCR is large or the grain size of communication is large (i.e., the number of bytes per message is large), then the system overhead does not play a major role. Our experimental results on NCUBE–2 show that our heuristics behave close to (and highly correlated to) the simulation results (which does not include the system overhead) and produce very good mappings.

3. BTDH

In the following, we briefly describe BTDH (for detail, see [5]). Let $e(T_n, P_x)$ be the earliest start time of task T_n on processor P_x, $f(T_n, P_x)$ be the finish time of task T_n on processor P_x, $previous(T_n, P_x)$ be the task which will be executed right before the execution of task T_n on processor P_x, $next(T_n, P_x)$ be the task which will be executed

right after the execution of task T_n on processor P_x, and $\theta(T_n, P_x)$ be the set of immediate predecessors of task T_n (T_n is scheduled on processor P_x) such that for every task T_p in $\theta(T_n, P_x)$ and T_p is scheduled on processors P_k, the earliest start time of T_n on P_x is equal to the sum of the finish time of T_p and the time of sending messages of T_p from P_k to P_x, i.e., $e(T_n, P_x) = f(T_p, P_k) + \tau(P_k, P_x) \times \eta(T_p, T_n)$.

Assume that a task T_n is scheduled on a processor P_x (T_n is the last task scheduled on P_x at this moment). BTDH tries to minimize the earliest start time of T_n on P_x by duplicating predecessors of T_n on the critical path (or paths) from the source. A high level description of the algorithm is given in Figure 1.

T_{end} (on P_x) represents the task for which the earliest start time is being considered for possible reduction (loop between lines 4 – 16). T_{top} represents the task scheduled on P_x before T_{end}. This gives the window of size of e_time ($e(T_{end}, P_x) - f(T_{top}, P_x)$) in which tasks can be potentially replicated. BTDH tries to replicate tasks on the critical path even though the earliest start time of the T_{end} may increase (temporary) till the tasks keep fitting this window. There are three ways to exit this loop.

Case 1 : The duplication of a predecessor of T_n may lead to the reduction of the earliest start time of T_{end} (line 8).

Case 2 : The duplication of a predecessor of T_n overflows the window (line 10 else part).

Case 3 : The duplication has not shown any reduction of $e(T_{end}, P_x)$ (line 13 else part).

In Case 2 and Case 3, the duplicated tasks between T_{top} and T_{end} are removed and algorithm proceeds to minimize the earliest start time of task immediately after T_{end}. In the first case, the algorithm proceeds to reduce the earliest start time of the predecessors. This may eventually lead to further reduction of the initial task T_n on which the duplication heuristic was applied. The com-

plexity of algorithm *BTDH* is $O(r^3)$, where r is the maximum number of predecessors of any task in a DAG (see [5]).

4. Applications of BTDH for LSAs

Many list scheduling algorithms have been proposed in the literature [2] [6] [8] [11] [13] [15] [19] [21] [22] [23] [28] [29] [31] [32]. In general, a LSA can be described as follow:

Algorithm LSA :
Phase 1 : Find the best (task, processor) pair from the *ready to schedule task list* (RSTL) and the *available processors list* (APL) according to some cost functions.
Phase 2 : Assign the task to the processor.
Phase 3 : Update the RSTL and the APL.
Phase 4 : Repeat Phase 1 to Phase 3 until all tasks are scheduled.

Since BTDH is a task duplication heuristic, it can be applied to a LSA in two ways:

(1) BTDH is used as a pure optimization algorithm for a LSA : When BTDH is used as a pure optimization algorithm, it tries to reduce the earliest start time of each task on each processor (Note that, in this case, tasks are already assigned to processors by a LSA before BTDH is used). In algorithm *LSA*, only one (task, processor) pair is selected whenever Phase 1 to Phase 3 are executed. When all tasks are scheduled on processors, we have a sequence of (task, processor) pairs. Since a (task, processor) pair is selected according to some cost function at a given time, the sequence of (task, processor) pairs provides us an order to select tasks for optimization. The algorithm is given as follows:

Algorithm LSA–BTDH :
Phase 1 : Use algorithm *LSA* to schedule a DAG on a DMM and keep the sequence of (task, processor) pairs in a queue Q.
Phase 2 : Let (T_i, P_j) = the first (task, processor) pair in Q.
Phase 3 : Assign T_i to P_j and apply BTDH to minimize the earliest start time of T_i on P_j.
Phase 4 : Delete the pair (T_i, P_j) from Q.
Phase 5 : Repeat Phases 2 – 4 until Q is empty.

Algorithm *LSA–BTDH* is a generic term. The term *LSA* can be substituted by any scheduling algorithm. For example, if BTDH is used as an optimization algorithm for HLFET. Then, the scheduling algorithm is HLFET–BTDH, i.e., BTDH is used for HLFET as an optimization heuristic.

(2) BTDH is used with a LSA to form a new scheduling algorithm : Let (T_n, P_k) be the best (task, processor) pair found from the RSTL and the APL according to some cost functions in each execution of Phase 1 to Phase 3 of algorithm *LSA*. BTDH can be applied to the pair (T_n, P_k) to reduce the start time of T_n on P_k. Since some of the immediate predecessors of T_n may be duplicated on some

other processors in the APL after some tasks are scheduled, BTDH is also applied to the pair (T_n, P_m), where P_m is in the APL and some of the immediate predecessors of T_n are duplicated on P_m. For those $(T_n$, processor) pairs which BTDH is applied, we choose the pair (T_n, P_x) as the best pair, where $e(T_n, P_x)$ is the minimum for all processors BTDH applied. The algorithm is given as follows:

Algorithm LSA/BTDH :
Phase 1 : Find the best (task, processor) pair from the RSTL and the APL according to some cost functions. Let the task and processor be T_n and P_k, respectively.
Phase 2 : Find the set of processors A_p, where for each $P_m \in A_p$, P_m is in the APL and some of the immediate predecessors of T_n are duplicated on P_m.
Phase 3 : For each $P_m \in A_p \cup \{P_k\}$, apply BTDH to each pair of (T_n, P_m) and choose the pair (T_n, P_x) as the best pair, where $e(T_n, P_x)$ is the minimum for all processors BTDH applied.
Phase 4 : Assign T_n to P_x.
Phase 5 : Update the RSTL and the APL.
Phase 6 : Repeat Phase 1 to Phase 5 until all tasks are scheduled.

Algorithm *LSA/BTDH* is a generic term. The term *LSA* can be substituted by any scheduling algorithm. For example, if BTDH is used with the ETF to form a scheduling algorithm. Then, the scheduling algorithm is ETF/BTDH.

5. Simulation Results

In this section, we use simulation approach to evaluate the performance of scheduling algorithms (Experimental results for some real programs on NCUBE–2 will be provided in Section 6). There are many factors such as the graph size, the processor number (PN), the ratio of the average communication cost to the average computational cost (CCR), the graph parallelism (GP) [19] of a DAG, and the topology of a given parallel machine can affect the makespan of a scheduling algorithm. In this paper, we will focus on the study of the relationship of CCR, GP, and PN to the scheduling length or speedup.

In our simulation, we assume that the system overhead is negligible. In real situations, especially for small size of problems, the system overhead such as the initiation of a communication primitive may have some effect on makespans of scheduling algorithms. In this case, as will be seen in Section 6, the system overhead sometimes will greatly offset the speedups of scheduling algorithms. However, the results of relative performance comparisons of scheduling algorithms for both simulation and experimental cases, in general, are identical. For the simulation, we also assume that the target machine has a complete interconnection between processors.

We set the values of CCR = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 50}, the values of GP = {4, 8, 16, 32, 64}, and the values of PN = {4, 8, 16, 32}. For each tuple (CCR, GP, PN), we randomly generate 10 DAGs as the test samples.

515

Each DAG has 300 tasks and the difference between the GP we set and the value of GP of the DAG is ± 0.5. The total computation time of a DAG on a single processor is around 1670 time units.

Let GPP be the ratio of GP to PN, i.e. GPP = GP / PN. Since the GP is the ratio of the total computation time of a DAG to the total computation time of tasks on the critical path of a DAG, it represents the maximal speedup that can be achieved by a scheduling algorithm for a given DAG on a DMM. In the following, we will analyze the performance of scheduling algorithms based on the values of GPP.

5.1 GPP < 1

GPP < 1 implies that the PN used in a scheduling algorithm is greater than the GP of a DAG. Makespans for scheduling algorithm with GPP = 0.5 and GPP = 0.25 are shown in Figure 3(a) and Figure 3(b), respectively. From Figure 3, we can see that LSA/BTDH and LSA–BTDH outperformed LSA for CCR ≥ 1. When the value of CCR increased, the difference of makespans between LSA and LSA/BTDH (or LSA–BTDH) is increased as well. The makespans of ETF, ETF/BTDH, ETF–BTDH, HLFET, HLFET/BTDH, and HLFET–BTDH in Figure 3, have the following order:

$$(HLFET/BTDH)_{makespan} \leq (ETF/BTDH)_{makespan} \leq (HLFET\text{-}BTDH)_{makespan} \leq (ETF\text{-}BTDH)_{makespan} \leq (ETF)_{makespan} \leq (HLFET)_{makespan},$$

where $(scheduling\ algorithm)_{makespan}$ is the makespan for a scheduling algorithm.

Since GPP < 1 implies that the PN used in a scheduling algorithm is greater than the GP of a DAG, it is of interest to see if the makespans for the case where GPP < 1 are less than the makespans for the case where GPP = 1. In Figure 4, the speedups for scheduling algorithms with CCR = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, GP = 8, and GPP = {0.25, 0.5, 1} are shown. From Figure 4, we can see that, in general, the more processors we used, the better speedup we can expect from LSA/BTDH and LSA–BTDH. However, the gain is not proportional to the number of processors used. For example, in Figure 4(d), the speedups of HLFET–BTDH for DAGs with CCR = 1 and GPP = 0.25, 0.5, and 1 are 4.71, 5.16, and 5.69, respectively. The ratio of processors used is 32 : 16 : 8 = 4 : 2 : 1 and the ratio of speedups is 5.69 : 5.17 : 4.71 = 1.21 : 1.1 : 1. Therefore, GP can accurately predict the number of processors to be used for a DAG such that a good scheduling length and a good resource utilization (or efficiency) can be achieved simultaneously.

5.2 GPP = 1

GPP = 1 implies that the PN used in a scheduling algorithm is equal to the GP of a DAG. Since the value of GP of a DAG is the maximal speedup can be achieved for a scheduling algorithm, the case where GPP = 1 seems likely to be the most economical way to use processors. In Figure 5, makespans for scheduling algorithm with GPP = 1 are shown. From Figure 5, we can see that LSA/BTDH and LSA–BTDH outperforms LSA for CCR ≥ 2. When the value of CCR is increased, the difference of

makespans between LSA and LSA/BTDH (or LSA–BTDH) is increased as well. The makespans of HLFET, ETF, LSA/BTDH, and LSA–BTDH in Figure 5, in general, have the following order:

$$(LSA/BTDH)_{makespan} \leq (HLFET\text{-}BTDH)_{makespan} \leq (ETF\text{-}BTDH)_{makespan} \leq (ETF)_{makespan} \leq (HLFET)_{makespan},$$

where $(scheduling\ algorithm)_{makespan}$ is the makespan for a scheduling algorithm. The difference between HLFET/BTDH and ETF/BTDH is negligible.

5.3 GPP > 1

GPP > 1 implies that the PN used in a scheduling algorithm is less than the GP of a DAG. In Figure 6, makespans for scheduling algorithm with GPP = {2, 4, 8, 16} are shown. From Figure 6, we can see that the makespans of ETF are almost the same as those of LSA/BTDH and LSA–BTDH when CCR ≤ 10 and GPP ≥ 4. This implies that when GPP is large enough and CCR is less than a threshold, the makespans of ETF are almost the same as LSA/BTDH and LSA–BTDH.

5.4 Comparisons of Execution Time of Scheduling Algorithms

The execution time for each scheduling algorithm to schedule the test samples on a DMM with complete connections between processors, on a SUN SPARC-STATION 2, is given in Table 1. From Table 1, we can see that the execution time of LSA/BTDH is much higher than those of LSA–BTDH and LSAs. When the number of tasks of a DAG is large, the time for LSA/BTDH may be relatively large. Therefore, LSA/BTDH is suitable for DAGs with a few tens to a few hundreds of tasks. Since the execution time of LSA–BTDH is a little higher than those of LSAs, for DAGs with large number of tasks, LSA–BTDH, in general, can produce better makespans than LSAs in a reasonable time.

5.5 Discussion

From the above comparisons, we have the following conclusions:

1) Given a DAG, the GP of the DAG can accurately predict the number of processors to be used such that a good scheduling length and a better resource utilization can be achieved simultaneously.

2) BTDH can significantly improve the makespan of DAGs over LSA; however, if GPP is very large, BTDH may improve speedup or scheduling length only slightly.

3) In general, $(LSA/BTDH)_{makespan} \leq (LSA\text{-}BTDH)_{makespan} \leq (ETF)_{makespan} \leq (HLFET)_{makespan}$, where $(scheduling\ algorithm)_{makespan}$ is the makespan for a scheduling algorithm.

4) In terms of time complexity, in general, we have $(LSA/BTDH)_{time} \geq (LSA\text{-}BTDH)_{time} \geq (ETF)_{time} \geq (HLFET)_{time}$, where $(scheduling\ algorithm)_{time}$ is the time complexity of a scheduling algorithm.

6. Experimental Results of Scheduling Algorithms on NCUBE–2

To demonstrate the performance of BTDH for real programs on an NCUBE–2 parallel machine, the FFT algorithm is implemented. The program is written in C language by using the *Single Program Multiple Data* (SPMD) programming approach. Since the grain size of a DAG determines the value of CCR, we generate DAGs with CCR > 1, 0.5 ≤ CCR < 1, and CCR < 0.5 and compare the performance of scheduling algorithms for each case.

6.1 The performance of Scheduling Algorithms for FFT

A FFT program, in general, can be described as follows:

Algorithm FFT(A)
1. $n = length(A)$; /* n is a power of 2 */
2. if ($n = 1$) then return(A);
3. $Y^{(0)} = FFT(A[0 : n–2 : 2])$;
4. $Y^{(1)} = FFT(A[1 : n–1 : 2])$;
5. $\omega_n = e^{2\pi i/n}$, $\omega = 1$;
6. for $k = 0$ to $n/2–1$ do
7. { $Y[k] = Y^{(0)}[k] + \omega * Y^{(1)}[k]$;
8. $Y[k+n/2] = Y^{(0)}[k] – \omega * Y^{(1)}[k]$;
9. $\omega = \omega * \omega_n$; }
10. return(Y); /* Y is assumed to be column vector */
End_of_FFT

---

where A and Y are arrays, $A[0 : n–2 : 2] = \{A[0], A[2], ..., A[n–2]\}$, and $A[1 : n–1 : 2] = \{A[1], A[3], ..., A[n–1]\}$. The behavior of FFT with input vector size = 4 is shown in Figure 7. In Figure 7, the computation of FFT consists of two operations, the *input vector operation* (IVO) (lines 3 and 4 in algorithm *FFT*) and the *butterfly operation* (BO) (lines 5 to 9 in algorithm *FFT*). Since the grain size of a DAG determines the value of CCR, to produce the desire value of CCR we want, array A can be split into a appropriate number of subarrays.

The DAG of FFT we used for scheduling algorithms is shown in Figure 8. In Figure 8, we have $2^l – 1$ IVO–task, 2^l FFT–task, and $(l+1) \times 2^l$ BO–task, where $l > 0$. For each IVO–task with input vector size = k, it needs to send a vector with size = $k/2$ to its immediate successors. For each FFT–task or BO–task with input vector size = k, it needs to send a vector with size = k to its immediate successors. We found out if the number of FFT–task in Figure 8 is less than or equal to 16 and array A has 1024 elements, we have CCR < 0.5. If the number of FFT–task in Figure 8 is equal to 32 and array A has 1024 elements, we have $0.5 \leq CCR \leq 1$. If the number of FFT–task in Figure 8 is greater than or equal to 64 and array A has 1024 elements, we have CCR ≥ 1. According to the values of CCR we want, we generate DAGs with different number of tasks. In the following, the input vector size of FFT is 1024.

Figure 9 shows the speedups of scheduling algorithms for a DAG G_4 with 511 tasks. The values of CCR and GP of G_4 are 1.04 and 14.56, respectively. From Figure 9, we can see that the system overhead can greatly offset the speedups we predict. This is due to the fine grain nature of the *FFT*–task and the BO–task (which represent majority of the tasks). Further, the amount of communication send to the *FFT*–task and the BO–task is relatively small (i.e., grain size of the communication is small and thus the system overhead plays a major role).

Figure 10 shows the speedups of scheduling algorithms for a DAG G_5 with 223 tasks. The values of CCR and GP of G_5 are 0.59 and 11.26, respectively. In Figure 10, the system overhead can offset some of speedups we predict (But, it is not so severe than that of G_4).

Figure 11 shows the speedups of scheduling algorithms for a DAG G_6 with 95 tasks. The values of CCR and GP of G_6 are 0.33 and 8.91, respectively. From Figure 11, we can see that the real speedups are very close to the predicted speedups. Thus, as the grain size of tasks and the grain size of communication (i.e., the number of bytes transferred per message) increases, the effect of system overhead becomes negligible and the experimental speedups are close to the speedups provided by the simulation method (which performs optimization assuming no system overhead).

From Figure 9 to Figure 11, in general, the speedups for scheduling algorithms has the following order:

$$(LSA/BTDH)_{speedup} \geq (LSA–BTDH)_{speedup} \geq (ETF)_{speedup} \geq (HLFET)_{speedup},$$

where (scheduling algorithm)$_{speedup}$ is the speedup for a scheduling algorithm. These results show a similar behavior pattern as the simulation results (of other graphs) given in Section 5.

6.2 Discussion

Based on the above experimental results, we have the following conclusions.

1) The grain size determination has a great impact for speedups of scheduling algorithms. If a DAG is too fine, the system overhead will sometimes greatly offset the speedups of scheduling algorithms. If a DAG is too coarse, the value of GP may be too small, i.e., too much parallelism is lost. In our experimental results, the best speedups of scheduling algorithms on NCUBE–2 are obtained when DAGs with $0.25 \leq CCR \leq 0.75$ are used.

2) The relative speedups comparisons between different algorithms based on experimental results are similar to those of simulation results. Neglection of system overhead affects measurement of absolute performance, but not relative performance.

7. Conclusions

In this paper, we have discussed the applications of BTDH for LSAs. There are two ways to use BTDH for LSAs. (1) BTDH can be used with a LSA to form a new scheduling algorithm (LSA/BTDH). (2) It can be used as a pure optimization algorithm for a LSA (LSA–BTDH). We have applied BTDH with two LSAs, the HLFET and the ETF, for both applications. We have studied the performance of BTDH for LSAs using simulation as well as on an actual machine. Three parameters, GP, CCR, and

PN, are simulated. From the simulation, we have the following conclusions. (I) Given a DAG, the GP of the DAG can accurately predict the number of processors to be used such that a good scheduling length and a good resource utilization (or efficiency) can be achieved simultaneously. (II) In terms of makespans of scheduling algorithms, in general, we have $(LSA/BTDH)_{makespan} \leq (LSA-BTDH)_{makespan} \leq (ETF)_{makespan} \leq (HLFET)_{makespan}$. In terms of the execution time of scheduling algorithms, in general, we have $(LSA/BTDH)_{time} \geq (LSA-BTDH)_{time} \geq (ETF)_{time} \geq (HLFET)_{time}$. Experimental results of scheduling FFT programs, which are written in a SPMD programming approach, on NCUBE–2 are presented. The results confirm our simulation results and show that the speedups of LSA/BTDH and LSA–BTDH are better than those of LSAs.

References :

[1] M. A. Al–Mouhamed, "Lower Bound on the Number of Processors and Time for Scheduling Precedence Graphs with Communication Costs," *IEEE Transactions on Software Engineering*, Vol. 16, No. 12, pp. 1390–1401, 1990.

[2] F.D. Anger, J.J. Hwang, and Y.C. Chow, "Scheduling with Sufficient Loosely Coupled Processors," *Journal of Parallel and Distributed Computing*, Vol. 9, pp. 87–92, 1990.

[3] J. Baxter and J.H. Patel, "The LAST Algorithm: A Heuristic–Based Static Task Allocation Algorithm," *ICPP*, Vol. 2, pp. 217–222, 1989.

[4] V. Chaudhary and J.K. Aggarwal, "Generalized Mapping of Parallel Algorithms onto Parallel Architectures," *ICPP*, Vol. 2, pp. 137–141, 1990.

[5] Y.C. Chung and S. Ranka, "An Optimization Approach for Static Scheduling of Directed–Acyclic Graphs on Distributed Memory Multiprocessors," International Conference on Parallel Processing, 1992 and CIS Technical Report, Syracuse University, 1992.

[6] H. El–Rewini and T.G. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Target Machines," *Journal of Parallel and Distributed Computing*, Vol. 9, pp. 138–153, 1990.

[7] R. Gupta and M.L. Soffa, "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, pp. 421–431, 1990.

[8] J.J. Hwang et al, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM Journal of Computing*, Vol. 18, pp. 244–257, 1989.

[9] H. Jung, L. Kirousis, and p. Spirakis, "Lower Bounds abd Efficient Algorithms for Multiprocessor Scheduling of DAGs with Communication Delay," *Proceedings of the ACM Symposiums of Parallel Algorithms and Architectures*, pp. 254–264, 1989.

[10] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computations upon Multiprocessor Architectures," *ICPP*, pp 1–8, 1988.

[11] B. Kruatrachue, *Static Task Scheduling and Grain Packing in Parallel Processing Systems*, PhD dissertation, Electrical and Computer Engineering Department, Oregon State University, Corvallis, 1987.

[12] B. Lee, A.R. Hurson, and T.Y. Feng, "A Vertically Layered Allocation Scheme for Data Flow Systems," *Journal of Parallel and Distributed Computing*, Vol. 11, pp. 175–187, 1991.

[13] C.Y. Lee et al, "Multiprocessor Scheduling with Interprocessor Communication Delays," *Operations Research Letters*, Vol. 7, pp. 141–147, 1988.

[14] K.J. Lin, J.Y. Chung, and J. Liu, "Scheduling Real–Time Computations on Hypercubes with Load Balancing," *The Fifth Conference of Distributed Memory Multiprocessors*, pp. 975–983, 1990.

[15] S. Manohara and P. Thanisch, "Assigning Dependency Graphs onto Processor Networks," *Parallel Computing*, Vol. 17, pp. 63–73, 1991.

[16] C.H. Papadimitriou and J.D. Ullman, "A Communication–Time Tradeoff," *SIAM Journal of Computing*, Vol. 14, No. 4, pp. 639–646, 1987.

[17] C.H. Papadimitriou and M. Yannakakis, "Towards an Architecture–Independent Analysis of Parallel Algorithms," *SIAM Journal of Computing*, Vol. 19, pp. 322–328, 1990.

[18] K. Ramamritham, J.A. Stankovic, and P.F. Shiah, "Efficient Scheduling Algorithms for Real–Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, pp. 184–194, 1990.

[19] G.C. Sih and E.A. Lee, "Scheduling to Account for Interprocessor Communication within Interconnection–Constrained Processor Networks," *ICPP*, Vol. 1, pp. 9–16, 1990.

[20] H.S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering*, Vol. 3, No. 1, pp. 85–93, 1977.

[21] M.Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message–Passing Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol.1, No. 3, pp. 330–343, 1990.

[22] T.L. Adam, K.M. Chandy, and J.R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Communication of ACM*, Vol. 17, No. 12, pp. 685–690, 1974.

[23] E.G. Coffman Jr., *Computer and Job–Shop Scheduling Theory*. New York: Wiley, 1976.

[24] D.K. Friesen, "Tighter Bound for LPT Scheduling on Uniform Processors," *SIAM Journal of Computing*, Vol. 16, No. 3, pp. 554–560, 1987.

[25] R.L. Graham, "Bounds on Multiprocessor Timing Anomalies," *SIAM Journal of Applied Mathematics*, Vol. 17, No.2, pp. 416–429, 1969.

[26] M. Granski, I. Koren, and G.M. Silberman, "The Effect of Operation Scheduling on the Performance of a Data Flow Computer," *IEEE Transactions on Computers*, Vol. 36, No. 9, pp. 1019–1029, 1987.

[27] D.S. Hochbaum and D.B. Shmoys, "A Polynomial Approximation Scheme for Scheduling on Uniform Processors: Using the Dual Approximation Approach," *SIAM Journal of Computing*, Vol. 17, No. 3, pp. 539–551, 1988.

[28] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, pp. 841–848, 1961.

[29] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Transactions on Computers*, Vol. 33, No. 11, pp. 1023–1029, 1984.

[30] J.Y.T. Leung and G.H. Young, "Minimizing Schedule Length Subject to Minimum Flow Time," *SIAM*

Journal of Computing, Vol. 18, No. 2, pp. 314–326, 1989.

[31] B. Shirazi and M. Wang, "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling," *Journal of Parallel and Distributed Computing*, Vol. 10, pp. 222–232, 1990.

[32] B.B. Simons and M.K. Warmuth, "A Fast Algorithm for Multiprocessor Scheduling of Unit–Time Jobs," *SIAM Journal of Computing*, Vol. 18, No. 4, pp. 690–710, 1989.

[33] M.R. Garey and D.S. Johnson, *Computers and Intractability – A Guide to the Theory of NP–Completeness*, W.H. Freeman, New York, 1979.
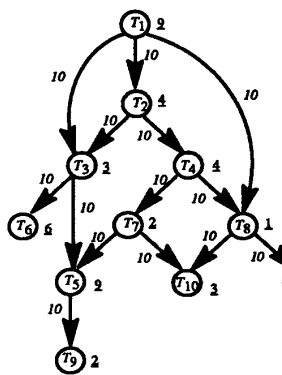
Figure 2 : An example of a DAG.
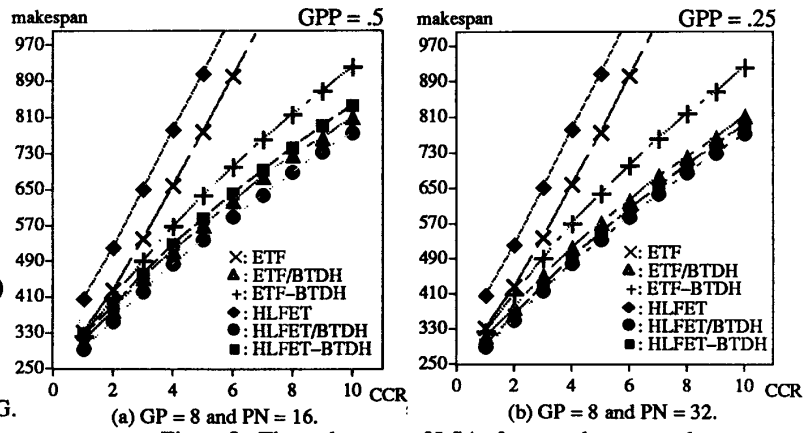
(a) GP = 8 and PN = 16.

(b) GP = 8 and PN = 32.

Figure 3 : The makespans of LSAs for complete networks.
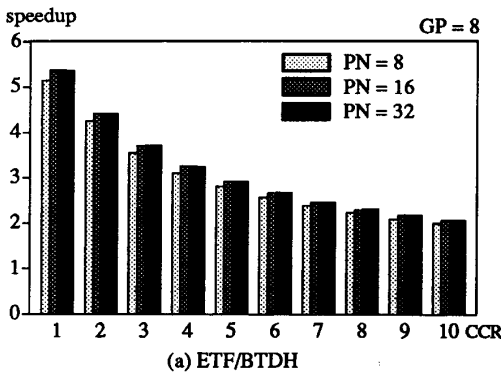
(a) ETF/BTDH
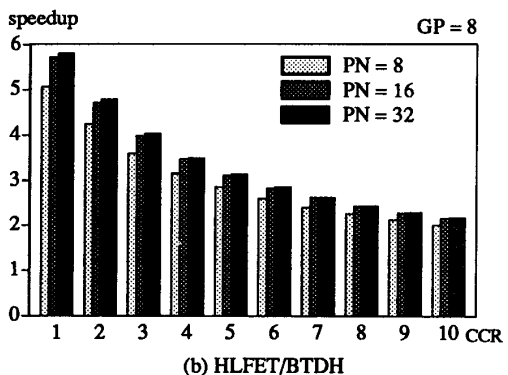
(b) HLFET/BTDH
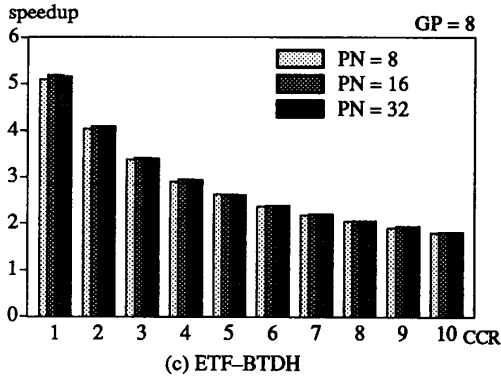
(c) ETF–BTDH
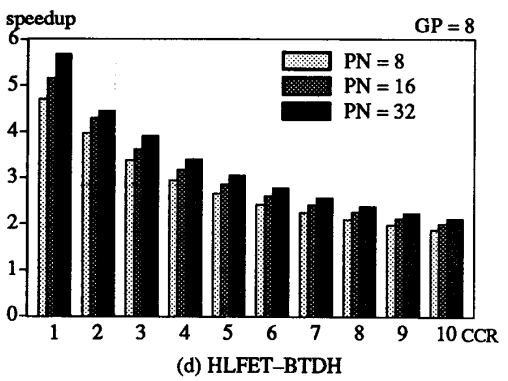
(d) HLFET–BTDH

Figure 4 : The speedups of LSAs for complete networks, where CCR = 1, ..., 10 and GPP ≤ 1.

(a) GP = 16 and PN = 16.

(b) GP = 32 and PN = 32.
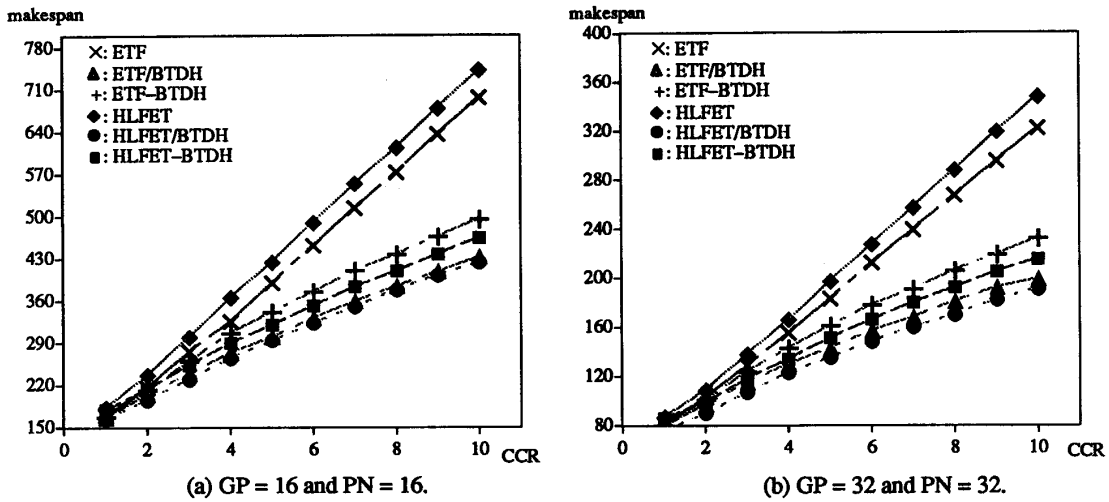
Figure 5 : The makespans of LSAs for complete networks, where CCR = 1, ..., 10 and GPP = 1.

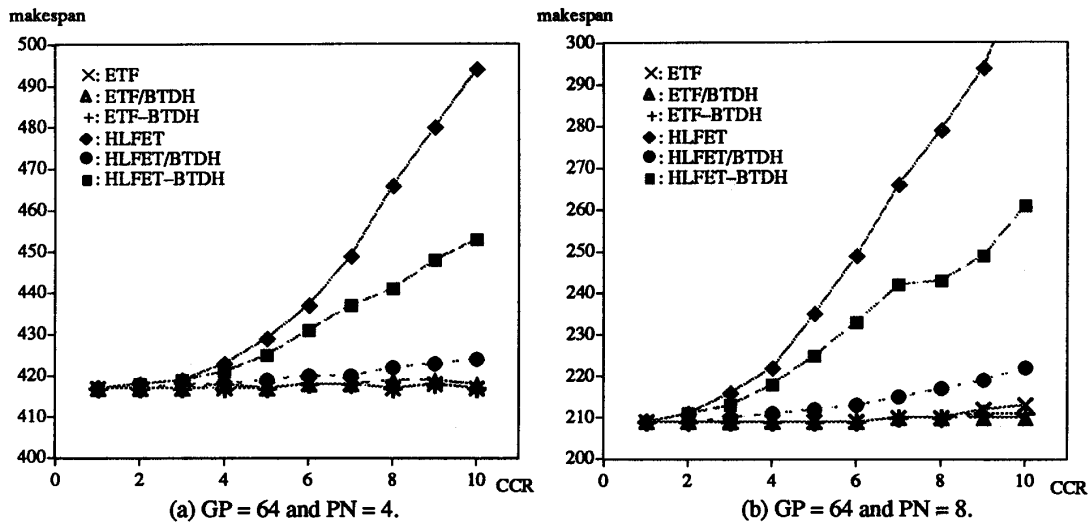(a) GP = 64 and PN = 4.

(b) GP = 64 and PN = 8.

Figure 6 : The makespans of LSAs for complete networks, where CCR = 1, ..., 10 and GPP > 1.

Table 1 : The execution time (in second) of scheduling algorithms for the test samples on 16 processors.

	GPP < 1	GPP = 1	GPP > 1
ETF	0.42 – 0.59	0.44 – 0.65	0.51 – 1.00
ETF/BTDH	14.50 – 59.15	30.7 – 93.03	50.36 – 100.38
ETF–BTDH	1.03 – 3.29	0.82 – 2.53	0.74 – 5.47
HLFET	0.11 – 0.12	0.11 – 0.11	0.10 – 0.11
HLFET/BTDH	10.36 – 64.89	6.22 – 55.04	4.19 – 37.06
HLFET–BTDH	1.10 – 10.88	0.51 – 9.04	0.27 – 5.32
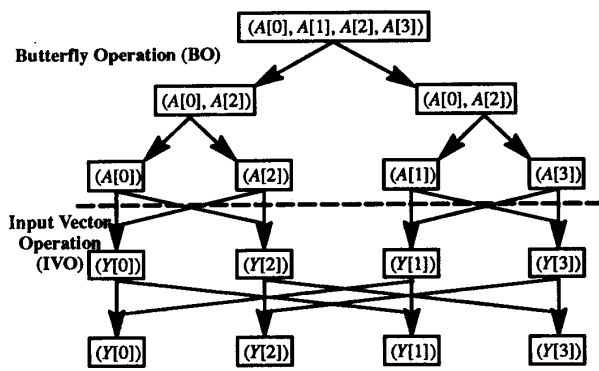
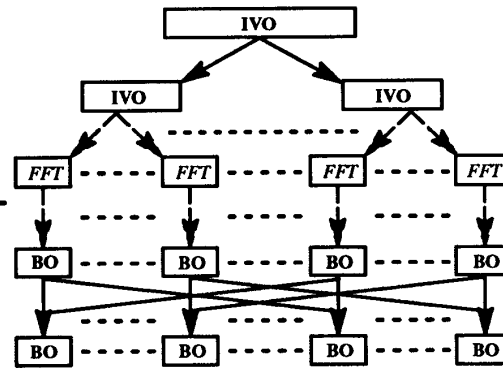Figure 7 : The behavior of FFT with 4 points.
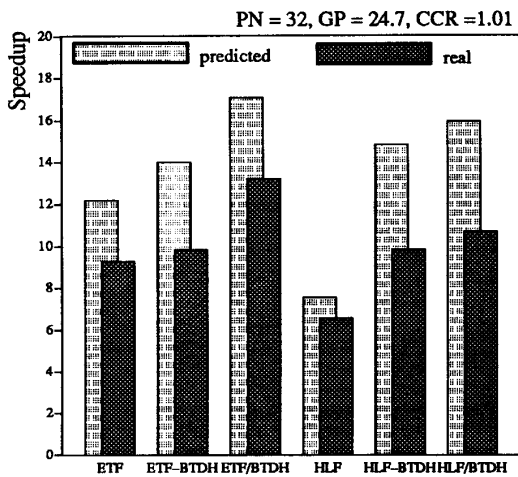
Figure 8 : The DAG generated for scheduling.

Figure 9 : Speedups of scheduling algorithms
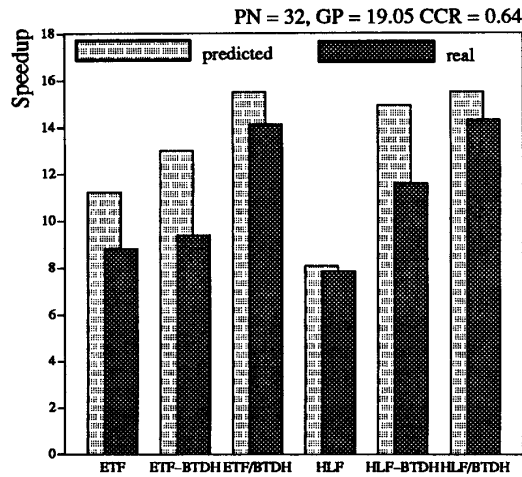for FFT with input vector size = 1024.

Figure 10 : Speedups of scheduling algorithms
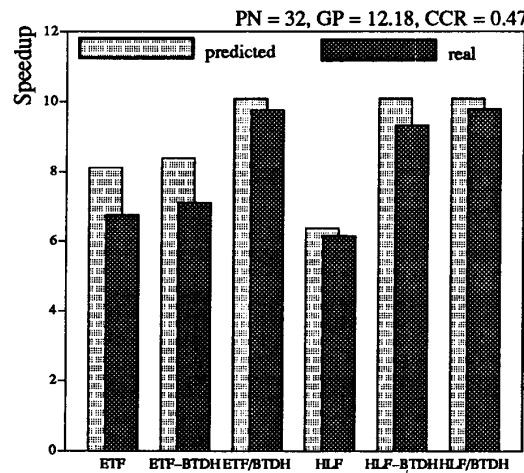for FFT with input vector size = 1024.

Figure 11 : Speedups of scheduling algorithms
for FFT with input vector size = 1024.