JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 11, 155-181 (1995)

DOI: 10.6688/JISE.1995.11.2.1

Applications and Performance Analysis of An Optimization Approach for List Scheduling Algorithms on Distributed Memory Multiprocessors

YEH-CHING CHUNG, CHIA-CHENG LIU, AND J.-S. LIU
Department of Information Engineering
Feng Chia University
Taichung, Taiwan 407, R.O.C.
E-mail: ychung, ccliu, liuj@pine. iecs. fcu. edu.tw
Tel: 886-4-2522250 ext. 3714

We have proposed an optimization approach, the *bottom-up top-down duplication* heuristic (BTDH), for static scheduling of *directed-acyclic graphs* (DAGs) on *distributed memory multiprocessors* [5]. In this paper, we discuss the applications of BTDH for *list scheduling algorithms* (LSAs). There are two ways to use BTDH with LSAs. (1) BTDH can be used with an LSA to form a new scheduling algorithm (LSA/BTDH). (2) It can be used as a pure optimization algorithm for an LSA (LSA-BTDH). We have applied BTDH to two well-known LSAs, the *highest level first with estimated time* (HLFET) and the *earlier task first* (ETF) heuristics. Extensive simulation has been conducted to study the performance of BTDH for LSAs. Three parameters, *graph parallelism* (GP) of a DAG [19], the ratio of the average communication cost to the average computation cost (CCR) of a DAG [5], and the processor number (PN) of a distributed memory multiprocessor, have been simulated. From the simulation, we have the following conclusions. (I) Given a DAG, the GP of a DAG can accurately predict the number of processors to be used such that a good scheduling length and good resource utilization (or efficiency) can be achieved simultaneously. (II) In terms of speedups, in general, we have LSA/BTDH ≥ LSA-BTDH ≥ ETF ≥ HLFET. Experimental results of scheduling FFT programs on an 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.

Keywords: List scheduling algorithm, heuristic, graph parallelism, directed-acyclic graph, distributed memory multiprocessor.

1. INTRODUCTION

The main purpose of using parallel computers is to reduce the execution time of application programs. Optimal execution of an application programs on a parallel computer depends on the method of partitioning an application program into tasks (the partitioning problem) and the method of scheduling tasks on a parallel computer (the scheduling problem). The main aspects of the partitioning prob-

Received July 6, 1993; revised September 30, 1994.
Communicated by Wen-Tsuen Chen.

lem 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; that is, 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 needed 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.*

It has been shown that an algorithm for solving the multiprocessor scheduling problem 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 the multiprocessor scheduling problem is the *list scheduling algorithm* (LSA) [22, 23, 25]. The underlying assumption of an LSA is that the interprocessor communication overhead of a computing system, such as processor communication or memory contention, is negligible. Under this assumption, LSAs can produce near optimal solutions for most instances. However, this assumption is not valid for *distributed memory multiprocessors* where interprocessor communication overhead is an important factor of system performance and is typically not negligible. In fact, an LSA is a load balancing heuristic. It tries to distribute the computation load among processors as evenly as possible and does not consider the communication overhead. It has been shown that an LSA produces poor scheduling results when interprocessor communication overhead is not negligible [3, 11]. Therefore, many approaches have been proposed for the multiprocessor scheduling problem with interprocessor communication overhead [1-21].

In [5], we have proposed an optimization approach, the *bottom-up top-down duplication heuristic* (BTDH), for static scheduling of DAGs on distributed memory multiprocessors. The underlying concept of BTDH is proper duplication of some tasks on processors such that the earliest start time of tasks on processors can be achieved. 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* (defined in Section 2) which can be inserted into the idle time slot between the finish time of the task before 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. However, 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 this drawback of DSH, BTDH allows the duplication of a predecessor T_i of a task T_n even though the duplication of T_i before 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 increased.

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 an LSA to form a new scheduling algorithm (LSA/BTDH). (2) It can be used as a pure optimization algorithm for an LSA (LSA-BTDH). We have applied BTDH to two well-known LSAs, the *highest level first with estimated time* (HLFET) [22] and the *earliest task first* (ETF) heuristics [8]. Extensive simulation has been conducted to study the performance of BTDH for LSAs. Three parameters, *graph parallelism* (GP) of a DAG [19], the ratio of the average communication cost to the average computation cost (CCR) of a DAG [5], and the processor number (PN) of a distributed memory multiprocessor, have been simulated. From the perfor-mance analysis, we have the following conclusions. (I) Given a DAG, the GP of a DAG can accurately predict the number of processors to be used such that a good scheduling length and good resource utilization (or efficiency) can be achieved simultaneously. (II) In terms of speedups, in general, we have LSA/BTDH $\geq$ LSA-BTDH $\geq$ ETF $\geq$ HLFET. Experimental results of scheduling FFT programs on an 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.

In section 2, the computational and the architectural models used in this paper will be 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 LASs using a simulation approach will be given in Section 5. In Section 6, experimental results of scheduling FFT programs on an NCUBE-2 will be presented.

2. THE COMPUTATIONAL AND THE ARCHITECTURAL MODELS

In this paper, we will consider scheduling of *static* (the number of tasks of a DAG is fixed during execution), *non-preempted* (the execution of a task can not be interrupted once it has started), *with communication delay* (interprocessor communication overhead is not negligible), and *duplicated* (a task may have several copies on processors) multiprocessor scheduling problem on distributed memory multiprocessors. An application program is modeled as a directed-acyclic graph (DAG) $G = \{T, A\}$, where $T = \{T_1, T_2, ..., T_n\}$ 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. Ti 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 Fig. 1 (a). In Fig. 1 (a), 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, that is, $\mu(T_6) = 6$, and the number of message units sent from task T_4 to T_7 is equal to 2, that is, $\eta(T_4, T_7) = 2$. 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. 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 distributed memory multiprocessor, a parameter $\tau(P_i, P_j)$ is introduced to represent the time needed to transfer a message unit from processor P_i to P_j. For real cases, especially for small size problems, the setup time between two processors may have some effect on the scheduling length of scheduling algorithms. However, in our simulation model, we assume that the setup time is much smaller than the time needed to transfer a message unit from one processor to another. Therefore, it is negligible. A distributed memory multiprocessor is then defined as $S = (P, \tau)$, where $P = \{P_1, P_2, ..., P_m\}$ 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 distributed memory multiprocessor is identical.
2. The intra-processor communication overhead is negligible, that is, $\tau(P_i, P_i) = 0$.
3. The communication subsystem is contention free.
4. A processor can send messages to some or all processors in a distributed memory multiprocessor simultaneously.
5. The system overhead, such as initialization of a send communication primitive, is negligible.

In a real machine, such as an NCUBE-2 on which we will present some experimental results, some of the above assumptions may not be valid. Assumption 2 is a realistic 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 an 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 (that is, the number of bytes per message is large), then the system overhead does not play a major role. Our experimental results on an NCUBE-2 show that our heuristics behave close to (and highly correlated to) the simulation results (which do not include the system overhead) and produce very good mappings.

3. BTDH: BOTTOM-UP TOP-DOWN DUPLICATION HEURISTIC

In the following, we will 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, that is, $e(T_n, P_x) = \max\{f(T_p, P_k) + \tau(P_k, P_x) \times \eta(T_p, T_n) \mid \forall T_p \in \theta(T_n, P_x)\}$.

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 as follows:

```
Algorithm BTDH($T_n$, $P_x$)
 1. repeat
 2.  { $e\_time = e(T_n, P_x)$, $T_{top} = previous(T_n, P_x)$, $T_{end} = T_n$, $weight = 0$.
 3.    $idle\_time = e(T_{end}, P_x) - f(T_{top}, T_x)$.
 4.    loop
 5.      { if (($\exists\ T_i \in \theta(T_n, P_x)$) ) and ($T_i$ is not scheduled on $P_x$))
 6.          then { Duplicate $T_i$ before $T_n$ on $P_x$.
 7.                 Recompute the earliest start time of tasks from $T_i$ to $T_{end}$ on $P_x$.
 8.                 if ($e(T_{end}, P_x) \le e\_time$) then { $T_n = T_i$, exit. }
 9.                 $weight = weight + \mu(T_i)$.
10.                 if ($weight < idle\_time$) then $T_n = T_i$.
                      else /* No predecessors of $T_{end}$ is duplicated */
11.                     { Remove all the tasks between $T_{top}$ and $T_{end}$.
12.                       $e(T_{end}, P_x) = e\_time$.
13.                       $T_n = next(T_{end}, P_x)$, recompute $e(T_n, P_x)$, exit.}
                 }
14.          else if ($T_n \ne T_{end}$) then $T_n = next(T_n, P_x)$.
```

```
          else /* No predecessors of $T_{end}$ is duplicated */
15.           { Remove all the tasks between $T_{top}$ and $T_{end}$.
16.             $e(T_{end}, P_x)$ = $e\_time$.
17.             $T_n$ = $next(T_{end}, P_x)$, recompute $e(T_n, P_x)$, exit.}
18.       } forever.
19.  } until ($T_n = \emptyset$).
20. return (the earliest start time of the last task scheduled on $P_x$).
end_of_algorithm_BTDH
```

T_{end} (on P_x) represents the task for which the earliest start time is being considered for possible reduction (loop between lines 4-18). 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 continue to fit 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 Tend (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 14 else part).

In Case 2 and Case 3, the duplicated tasks between T_{top} and T_{end} are removed, and the algorithm proceeds to minimize the earliest start time of the 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 complexity of algorithm *BTDH* is $O(r^3)$, where r is the maximum number of predecessors of tasks in a DAG.

An example is given in Fig. 1 to show the behavior of algorithm *BTDH*. The DAG is given in Fig. 1(a). The target computer, which has four processors with complete connections between processors, is shown in Fig. 1(b). The initial status is shown in Fig. 1(c). In Fig. 1(c), the current scheduling lengths of P_0, P_1, P_2, and P_3 are 32, 31, and 40 time units, respectively. Symbol ϕ_i denotes the idleness of i time units. For example, in Fig. 1(c), ϕ_{37} in P_3 denotes that P_3 is idled for 37 time units before task T_{10} is executed. In the following, we assume that task P_{10} is scheduled on processor P_3, and we show how BTDH duplicates some predecessors of task T_{10} on P_3 step by step.

Step 1 : $\theta(T_{10}, P_3) = \{T_8\}$ and $e(T_{10}, P_3) = 37$. Since duplicating T_8 before T_{10} does not increase the value of $e(T_{10}, P_3)$, T_8 is duplicated on processor P_3 Fig. 1(d), and BTDH tries to minimize $e(T_8, P_3)$.

Step 2 : $\theta(T_8, P_3) = \{T_4\}$ and $e(T_8, P_3) = 27$. Since duplicating T_4 before T_8 does not increase the value of $e(T_8, P_3)$, T_4 is duplicated on processor P_3 Fig.

1(e), and BTDH tries to minimize $e(T_4, P_3)$.

Step 3: $\theta(T_4, P_3) = \{T_2\}$ and $e(T_4, P_3) = 22$. Since duplicating T_2 before T_4 does not increase the value of $e(T_4, P_3)$, T_2 is duplicated on processor P_3 Fig. 1(f), and BTDH tries to minimize $e(T_2, P_3)$.

Step 4: $\theta(T_2, P_3) = \{T_1\}$ and $e(T_2, P_3) = 19$. Since duplicating T_1 before T_2 does not increase the value of $e(T_2, P_3)$, T_1 is duplicated on processor P_3 Fig. 1(g), and BTDH tries to minimize $e(T_1, P_3)$.

Step 5: Since $\theta(T_1, P_3) = \emptyset$, BTDH tries to minimize $e(T_2, P_3)$.

Step 6: Since $\theta(T_2, P_3) = \emptyset$, BTDH tries to minimize $e(T_4, P_3)$.

Step 7: Since $\theta(T_4, P_3) = \emptyset$, BTDH tries to minimize $e(T_8, P_3)$.

Step 8: Since $\theta(T_8, P_3) = \emptyset$, BTDH tries to minimize $e(T_{10}, P_3)$.

Step 9: $\theta(T_{10}, P_3) = \{T_7\}$ and $e(T_{10}, P_3) = 28$. Since duplicating T_7 before T_{10} does not increase the value of $e(T_{10}, P_3)$, T_7 is duplicated on processor P_3 Fig. 1(h), and BTDH tries to minimize $e(T_7, P_3)$.

Step 10: Since $\theta(T_7, P_3) = \emptyset$, BTDH tries to minimize $e(T_{10}, P_3)$.

Step 11: Since $\theta(T_{10}, P_3) = \emptyset$, BTDH terminates its duplicating process.

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, an LSA can be described as follows:

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

Since BTDH is a task duplication heuristic, it can be applied to an LSA in two ways. (1) BTDH is used as a pure optimization algorithm for an 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 an 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 functions at a given time, the sequence of (task, processor) pairs provides us with an order to select tasks for optimization. The algorithm is given as follows:

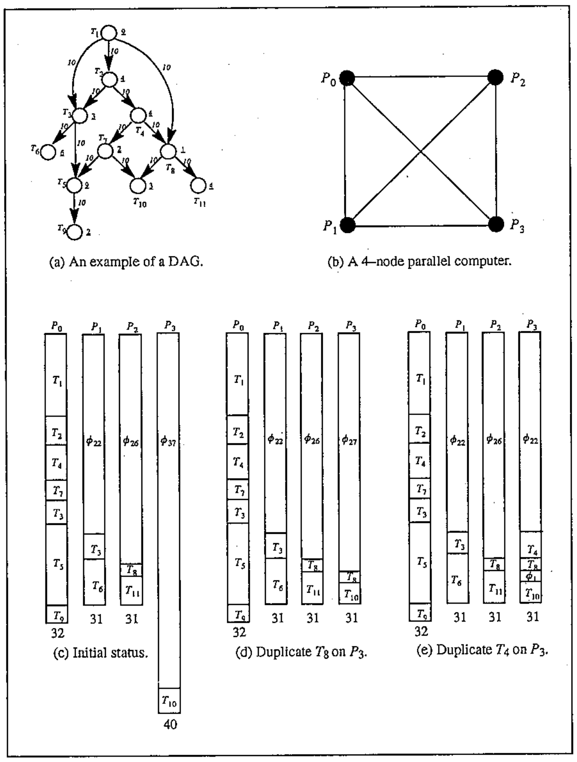

(a) An example of a DAG.
(b) A 4-node parallel computer.
(c) Initial status.
(d) Duplicate T_8 on P_3.
(e) Duplicate T_4 on P_3.

Fig. 1. An example of applying BTDH.

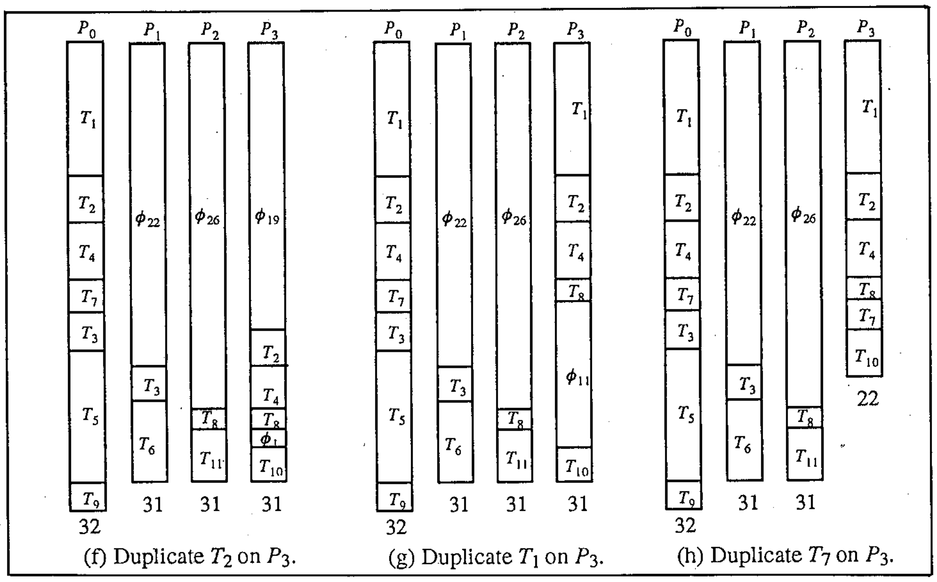

(f) Duplicate T_2 on P_3.
(g) Duplicate T_1 on P_3.
(h) Duplicate T_7 on P_3.

Fig. 1 (cont'd). An example of applying BTDH.

```
Algorithm LSA-BTDH:
Phase 1: Use algorithm LSA to schedule a DAG on a distributed memory multiprocessor,
         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 Phase 2 to Phase 4 until $Q$ is empty.
end_of_algorithm_LSA_BTDH:
```

Algorithm *LSA-BTDH* is a generic term. The term *LSA* can be replaced with any list scheduling algorithm. For example, if BTDH is used as an optimization algorithm for HLFET, then the scheduling algorithm is HLFET-BTDH. The time complexity of algorithm *LSA-BTDH* is upper bounded by $O(Nr^3) + O(L)$, where N is the number of tasks of a DAG, r is the maximal number of predecessors of any task of a DAG, and $O(L)$ is the time complexity of an LSA.

(2) BTDH is used with an LSA to form a new scheduling algorithm: Let (T_n, P_k) be the best (task, processor) pair found from the ready to schedule task list and the available processors list 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 available processors list after some tasks are scheduled, BTDH is also applied to the pair (T_n, P_m), where P_m is in the available processors list, and some of the immediate predecessors of T_n are duplicated on P_m. For those (T_n, processor) pairs to 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 ready to schedule task list
         and the available processor list 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 available
         processor list, 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 $(T_n, P_k)$ 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 ready to schedule task list and the available processor list.
Phase 6: Repeat Phase 1 to Phase 5 until all tasks are scheduled.
end_of_algorithm_LSA/BTDH:
```

Algorithm *LSA/BTDH* is a generic term. The term *LSA* can be replaced with any list scheduling algorithm. For example, if BTDH is used with the ETF to form a scheduling algorithm, then the scheduling algorithm is ETF/BTDH. The time complexity of LAS/BTDH is upper bounded by $O((N^2 + MN)r^3)$, where M is the number of processors of a distributed memory multiprocessor, N is the number of tasks of a DAG, and r is the maximum number of predecessors of tasks of a DAG.

5. SIMULATION RESULTS

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 which can affect the scheduling length of a scheduling algorithm. In this paper, we will focus on the relationship of CCR, GP, and PN to the scheduling length or speedup.

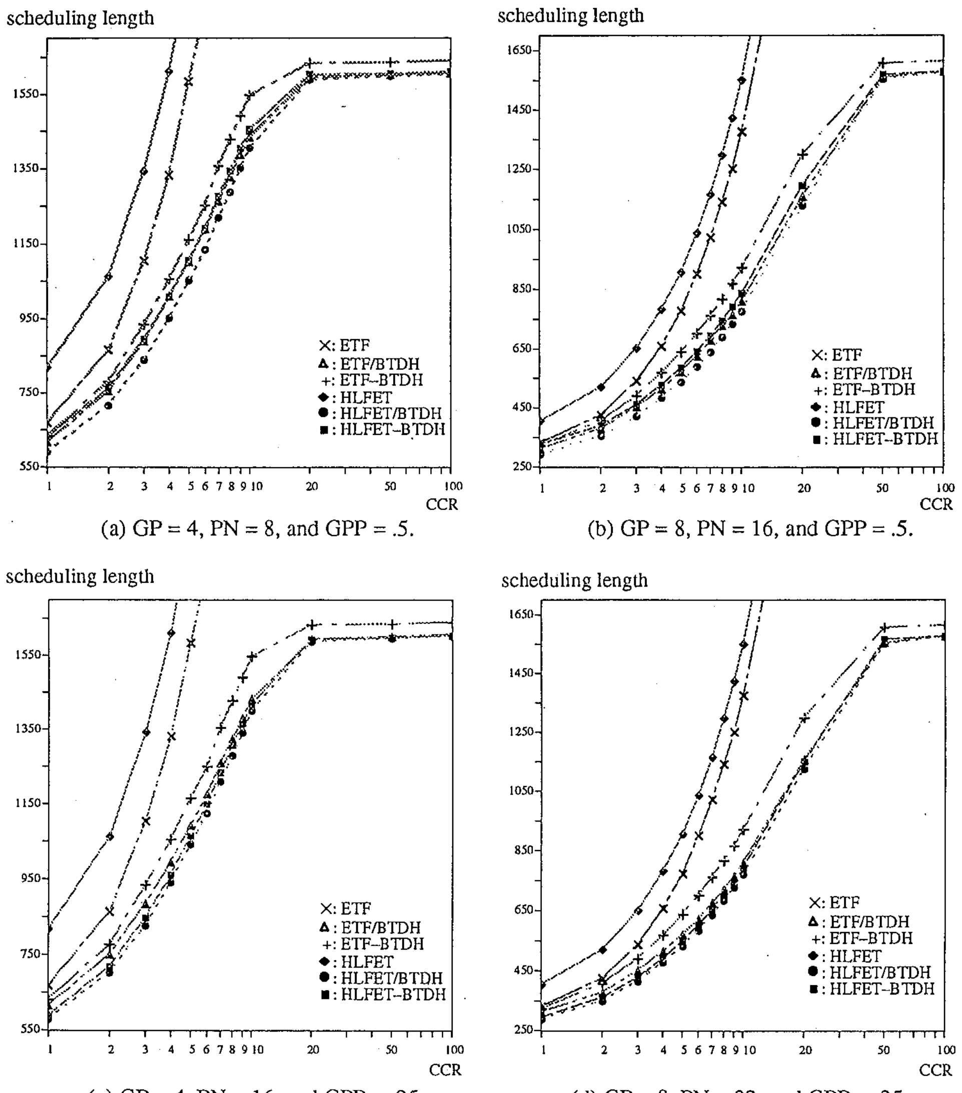
In our simulation, we assume that the system overhead is negligible. In a real situation, especially for small size problems, the system overhead such as the initiation of a communication primitive may have some effect on scheduling length of scheduling algorithms. In this case, as will be seen in Section 6, the system overhead sometimes will greatly offset the speedups of scheduling algorithms. However, the results of 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.

Since it is important to use a broad range of DAGs as the test samples, in our simulation, we set the values of CCR = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 50, 100}, 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. The number of tasks in each DAG is equal to 300. Each task in a DAG has at most 4 immediate predecessors and 4 immediate successors. The number of immediate predecessors and immediate successors are uniformly distributed between 0 and 4. The computation costs of tasks are uniformly distributed from 1 to 10 time units. The average computation cost of the DAGs generated is around 1670 time units. For simplicity, the communication cost and the computation cost use the same time measurement. However, in this section, we did not evaluate the performance of scheduling algorithms for the cases where $0 < \text{CCR} < 1$. In Section 6, we will give the experimental results for those cases.

Let GPP be the ratio of GP to PN, that is, GPP = GP / PN. Since 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 which can be achieved by a scheduling algorithm for a given DAG on a distributed memory multiprocessor. Therefore, GPP can be treated as an efficiency (of system utilization) measure. 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. In Fig. 2, the scheduling length for scheduling algorithms with GPP = 0.5 and GPP = 0.25 are shown. From Fig. 2, we can see that LSA/BTDH and LSA/BTDH outperformed LSA for CCR ≥ 1. When the value of CCR increased, the difference of scheduling length between LSA and LSA/BTDH (or LSA-BTDH) increased as well. Moreover, the scheduling length of



(a) GP = 4, PN = 8, and GPP = .5.

(b) GP = 8, PN = 16, and GPP = .5.

(c) GP = 4, PN = 16, and GPP = .25.

(d) GP = 8, PN = 32, and GPP = .25.

Fig. 2. The scheduling lengths of LSAs for complete networks, where CCR = 1, ..., 100 and GPP < 1.

LSA-BTDH and LSA/BTDH converge asymptotically when the value of CCR is over a threshold (In our case, CCR $\geq$ 50). This is because that when the value of CCR is over a threshold, the value of *idle_time* in algorithm *BTDH* is greater than or equal to the total computation cost of all predecessors of a task T_i. This implies that all the predecessors of T_i can be duplicated in front of T_i when BTDH is applied. In Fig. 2, the scheduling lengths of ETF, ETF/BTDH, ETF-BTDH, HLFET, HLFET/BTDH, and HLFET-BTDH have the following order:

$$\begin{aligned}&(\text{HLFET/BTDH})_{\textit{scheduling length}} \leq (\text{ETF/BTDH})_{\textit{scheduling length}}\\ &\leq (\text{HLFET-BTDH})_{\textit{scheduling length}} \leq (\text{ETF-BTDH})_{\textit{scheduling length}}\\ &\leq (\text{ETF})_{\textit{scheduling length}} \leq (\text{HLFET})_{\textit{scheduling length}},\end{aligned}$$

where (scheduling algorithm)$_{\textit{scheduling length}}$ is the scheduling length 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 interesting to see if the scheduling lengths for the case where GPP < 1 are less than the scheduling lengths for the case where GPP = 1. In Fig. 3, the speedups for scheduling algorithms with CCR = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 50}, GP = 8, and GPP = {0.25, 0.5, 1} are shown. From Fig. 3, we can see that, in general, the more processors we use, 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 Fig. 3(f), 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 : 1.11 : 1. Therefore, GP can accurately predict the number of processors to be used for a DAG such that a good scheduling length and 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 which can be achieved for a scheduling algorithm, the case where GPP = 1 seems likely to be the most economical way to use processors. In Fig. 4, the scheduling lengths for scheduling algorithms with GPP = 1 are shown. From Fig. 4, we can see that LSA/BTDH and LSA-BTDH outperformed LSA for CCR $\geq$ 2. When the value of CCR increased, the difference of scheduling length between LSA and LSA/BTDH (or LSA-BTDH) increased as well. Moreover, the scheduling length of LSA-BTDH and LSA/BTDH converge asymptotically when the value of CCR is over a threshold. (This is not clear from Fig. 4(c) and Fig. 4(d). In our simulation, the scheduling lengths of LSA-BTDH and LSA/BTDH for the cases where GP = 16 and GP = 32 converge asymptotically when CCR $\geq$ 100 and CCR $\geq$ 300, respectively.) This is because when the value of CCR is over a threshold, the value of *idle_time* in algorithm *BTDH* is greater than or equal to the total computation cost of all predecessors of a task T_i. This implies that all the predecessors of T_i can be duplicated in front of T_i when BTDH is applied. In Fig. 4, the scheduling lengths of HLFET, ETF, LSA/BTDH, and LSA-BTDH, in general, have the following order:

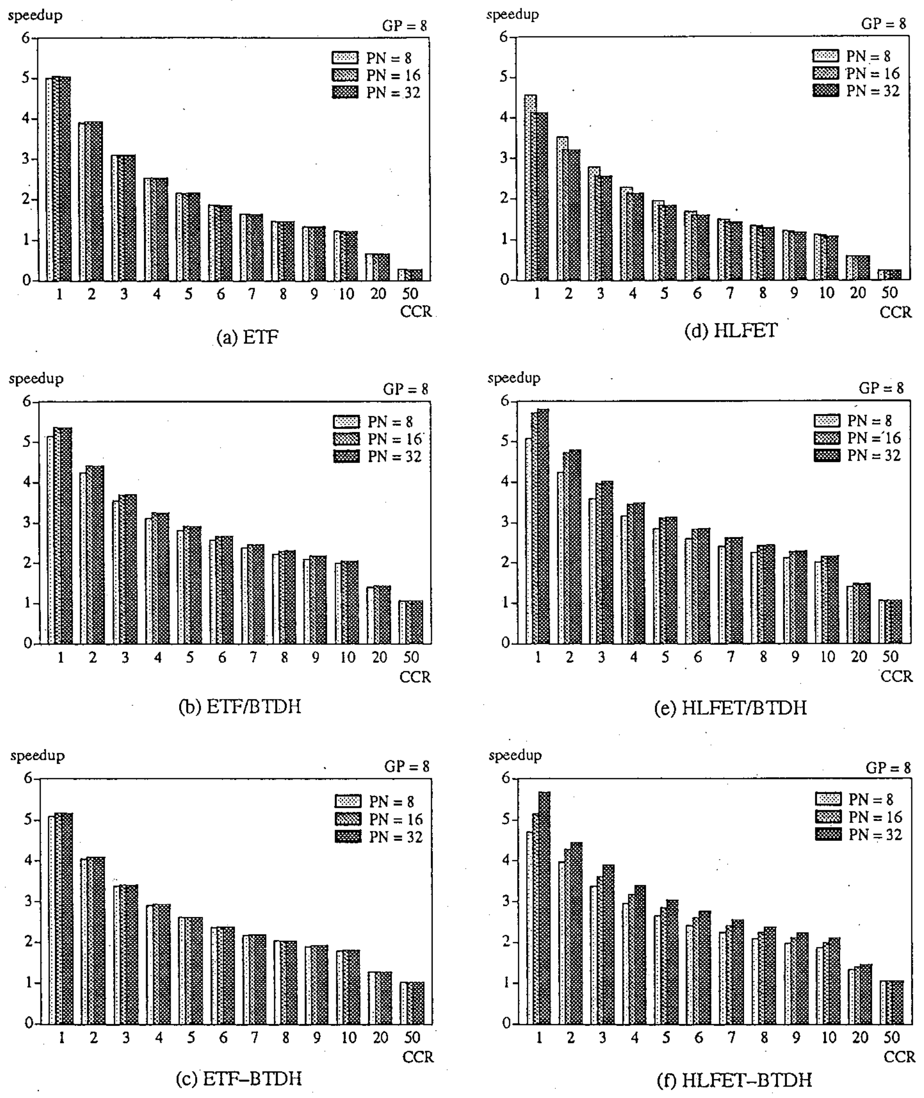


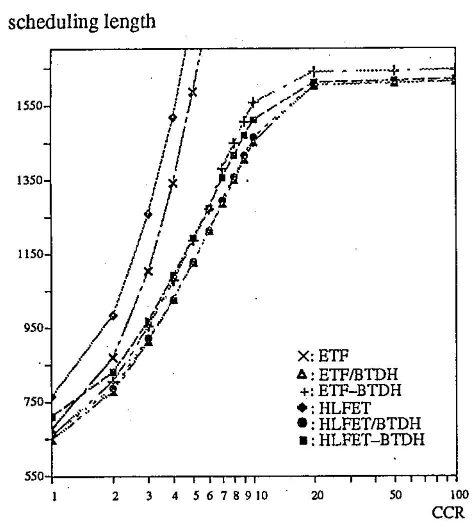
Fig. 3. The speedups of LSAs for complete networks, where CCR = 1, ..., 50 and GPP $\leq$ 1.

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

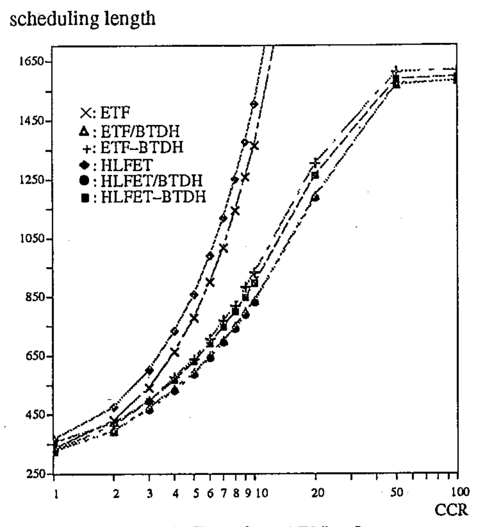
where $(\text{scheduling algorithm})_{scheduling\ length}$ is the scheduling for a scheduling algorithm. The difference between HLFET/BTDH and ETF/BTDH is negligible.

5.3 GPP > 1

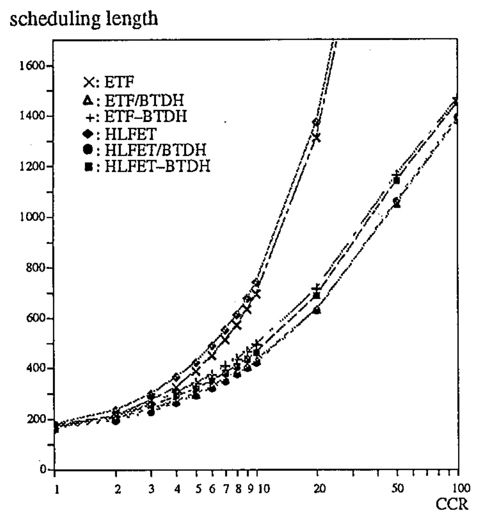
GPP > 1 implies that the PN used in a scheduling algorithm is less than the GP of a DAG. In Fig. 5, the scheduling lengths for scheduling algorithms with GPP = {2, 4, 8, 16} are shown. From Fig. 5, we can see that the scheduling lengths



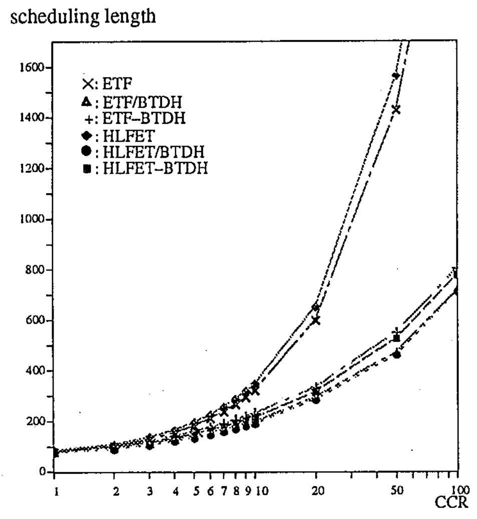
(a) GP = 4 and PN = 4.



(b) GP = 8 and PN = 8.



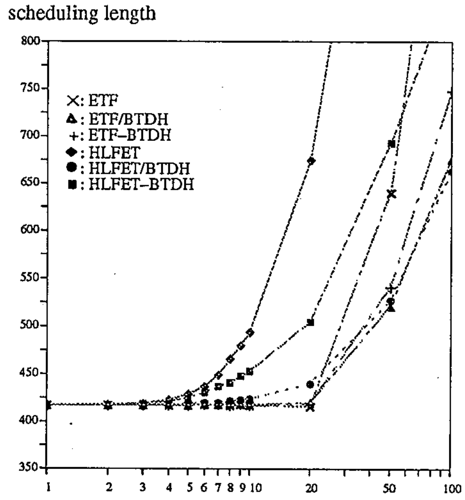
(c) GP = 16 and PN = 16.



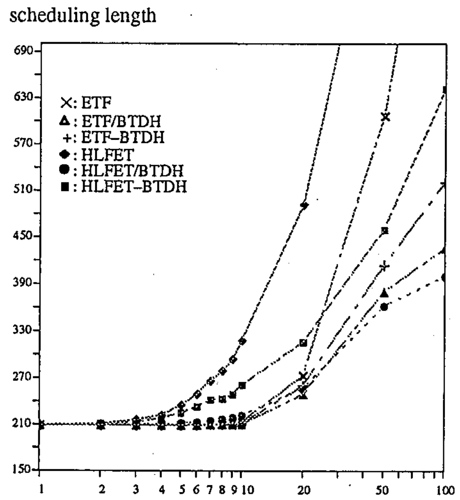
(d) GP = 32 and PN = 32.

Fig. 4. The scheduling length of LSAs for complete networks, where CCR = 1, ..., 100 and GPP = 1.

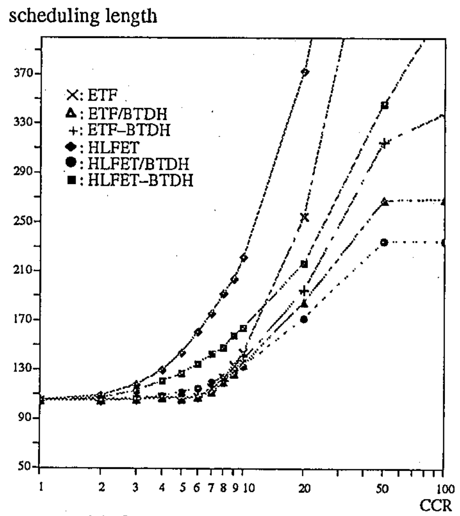
of ETF are almost the same as those of LSA/BTDH and LSA-BTDH when CCR ≤ 5 and GPP = 2, or CCR ≤ 10 and GPP = 4, or CCR ≤ 20 and GPP ≥ 8. This implies that when GPP is large enough, and CCR is less than a threshold, the scheduling lengths of ETF are almost the same as those of LSA/BTDH and LSA-BTDH. If the value of CCR is large, the scheduling lengths of ETF are increased dramatically. However, the scheduling lengths of LSA-BTDH and LSA/BTDH converge asymptotically when the value of CCR is over a threshold. (However, we do not show the convergence in Fig. 5(a) and Fig. 5(b). In our simulation, the



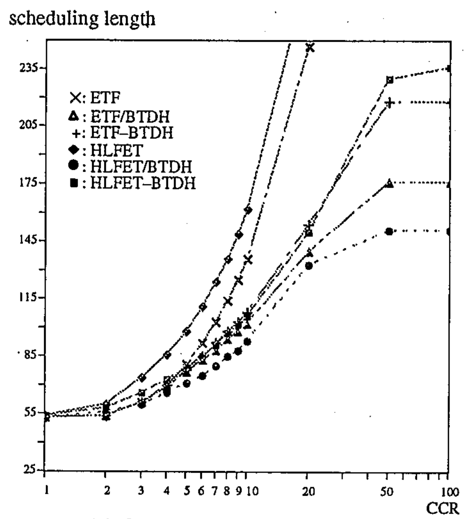
(a) GP = 64, PN = 4, and GPP = 16.



(b) GP = 64, PN = 8, and GPP = 8.



(c) GP = 64, PN = 16, and GPP = 4.



(d) GP = 64, PN = 32, and GPP = 2.

Fig. 5. The scheduling length of LSAs for complete networks, where CCR = 1, ..., 100 and GPP > 1.

scheduling lengths of LSA-BTDH and LSA/BTDH for the cases where GPP = 16 and GPP = 8 converge asymptotically when CCR ≥ 300 and CCR ≥ 100, respectively.) This is because when the value of CCR is over a threshold, the value of *idle_time* in algorithm *BTDH* is greater than or equal to the total computation cost of all predecessors of a task T_i. This implies that all the predecessors of T_i can be duplicated in front of T_i when BTDH is applied.

5.4 Comparisons of Execution Time of Scheduling Algorithms

The execution time needed for each scheduling algorithm to schedule the test samples on a distributed memory multiprocessor 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 a better scheduling length than can LSAs in a reasonable time.

Table 1. The execution time of scheduling algorithms for the test samples.

(a) The execution time (in second) of scheduling algorithms, where GPP < 1.

	4–processor	8–processor	16–processor	32–processor
ETF	not available	0.26 – 0.41	0.42 – 0.59	0.75 – 1.24
ETF/BTDH	not available	9.13 – 18.07	14.50 – 59.15	31.00 – 216.17
ETF-BTDH	not available	0.77 – 1.48	1.03 – 3.29	1.64 – 8.68
HLFET	not available	0.07 – 0.08	0.11 – 0.12	0.19 – 0.21
HLFET/BTDH	not available	7.44 – 22.38	10.36 – 64.89	11.62 – 184.69
HLFET-BTDH	not available	0.71 – 3.04	1.10 – 10.88	1.77 – 39.08

(b) The execution time (in second) of scheduling algorithms, where GPP = 1.

	4–processor	8–processor	16–processor	32–processor
ETF	0.21 – 0.31	0.26 – 0.35	0.44 – 0.65	0.88 – 1.43
ETF/BTDH	5.71 – 9.44	11.37 – 29.43	30.7 – 93.03	160.66 – 276.22
ETF-BTDH	0.4 – 0.91	0.53 – 2.12	0.82 – 2.53	1.37 – 10.8
HLFET	0.05 – 0.06	0.07 – 0.08	0.11 – 0.11	0.17 – 0.18
HLFET/BTDH	4.65 – 9.24	5.39 – 18.49	6.22 – 55.04	7.88 – 91.00
HLFET-BTDH	0.25 – 0.86	0.35 – 2.95	0.51 – 9.04	0.77 – 14.54

(c) The execution time (in second) of scheduling algorithms, where GPP > 1.

	4–processor	8–processor	16–processor	32–processor
ETF	0.22 – 0.63	0.30 – 0.75	0.51 – 1.00	1.11 – 1.53
ETF/BTDH	6.24 – 16.53	13.9 – 38.17	50.36 – 100.38	199.28 – 279.04
ETF-BTDH	0.33 – 1.00	0.46 – 2.84	0.74 – 5.47	1.41 – 2.39
HLFET	0.05 – 0.07	0.08 – 0.08	0.10 – 0.11	0.15 – 0.16
HLFET/BTDH	3.16 – 7.45	3.71 – 19.84	4.19 – 37.06	4.55 – 10.14
HLFET-BTDH	0.13 – 0.87	0.18 – 2.63	0.27 – 5.32	0.44 – 1.06

5.5 Discussion

From the above comparisons, we have the following conclusions:

1) Given a DAG, the GP of a DAG can accurately predict the number of processors to be used such that a good scheduling length and better resource utilization can be achieved simultaneously.
2) If GPP is very large, to use BTDH with ETF may gain nothing in terms of speedup or scheduling length; that is, it is not necessary to use BTDH.
3) In general, LSA/BTDH can produce better scheduling length than can LSA-BTDH, LSA-BTDH can produce better scheduling length than can ETF, and ETF can produce better scheduling length than can HLFET, that is, $(\text{LSA/BTDH})_{\textit{scheduling length}} \leq (\text{LSA-BTDH})_{\textit{scheduling length}} \leq (\text{ETF})_{\textit{scheduling length}} \leq (\text{HLFET})_{\textit{scheduling length}}$, where $(\text{scheduling algorithm})_{\textit{scheduling length}}$ is the scheduling length for a scheduling algorithm.
4) In terms of time complexity, in general, we have $(\text{LSA/BTDH})_{\textit{time}} \geq (\text{LSA-BTDH})_{\textit{time}} \geq (\text{ETF})_{\textit{time}} \geq (\text{HLFET})_{\textit{time}}$, where $(\text{scheduling algorithm})_{\textit{time}}$ 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, FFT programs were implemented. The programs were written in C language by using the *Single Program Multiple Data* (SPMD) programming model. Since BTDH is a task duplication heuristic, and we used the SPMD programming model, statements in an FFT program were implemented as procedure calls. The overall program design flow is given in Fig. 6. In the DAG generation phase, we analyze the sequential program, choose the grain size, and partition and transform the sequential program to a DAG. In the scheduling phase, the DAG is scheduled and a *configuration table* is generated. The configuration table contains information of (task, processor) pairs, the earliest start time of tasks on processors, and the execution order of tasks on processors. In the communication analysis phase, the communication needed among processors is analyzed, and the information for each task which needs to communicate with other tasks is added to the configuration table. According to the configuration table from the communication analysis phase, in the code generation phase, the parallel codes of the sequential program are generated. The parallel program is then executed on an NCUBE-2 to evaluate its execution time. All phases, except the DAG generation phase, are performed automatically. The DAG generation phase is performed in a semi-automatic way. Since the grain size of a DAG determines the value of CCR, in the following, we will describe how the DAG generation phase is performed for the test samples in detail. However, for other phases, we do not describe their implementation in this paper. For FFT programs, we generate DAGs with $\text{CCR} > 1$, $0.5 \leq \text{CCR} < 1$, and $0 \leq \text{CCR} < 0.5$ and compare the performance of scheduling algorithms for each case.

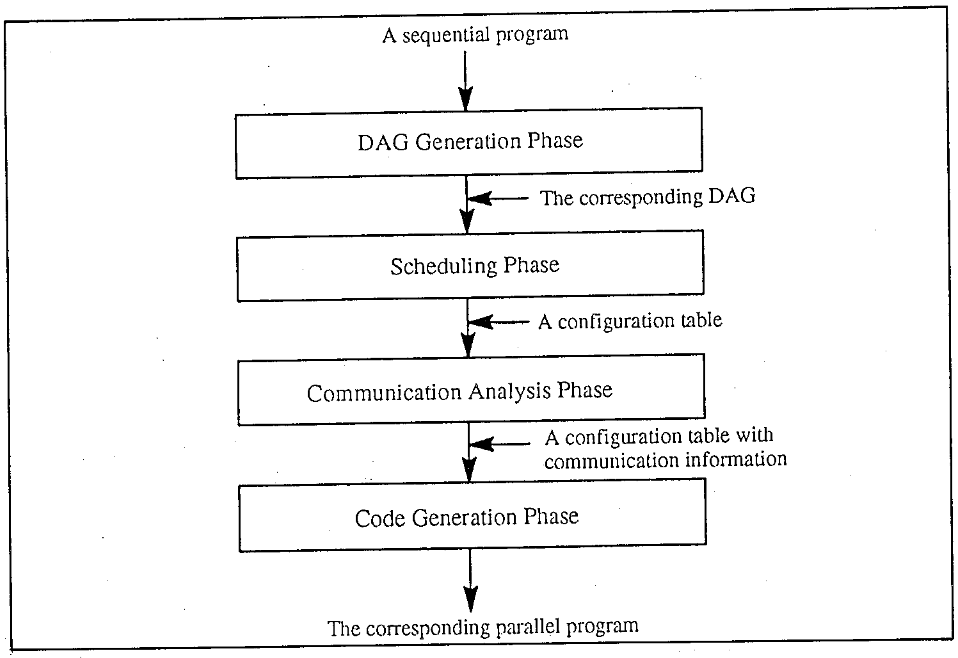


Fig. 6. The parallel program design flow.

6.1 The performance of Scheduling Algorithms for FFT

An 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. $\omega_n = e2^{\pi i/n}$; $\omega = 1$;
6. for $k = 0$ to $n/2-1$ do
7. { $Y[k] = Y^{(0)}[k] + \omega * Y^{(1)}[k]$; $Y[k+n/2] = Y^{(0)}[k] - \omega * Y^{(1)}[k]$; $\omega = \omega * \omega_n$; }
8. return ($Y$); /* $Y$ is assumed to be column vector */
end_of_algorithm_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 Fig. 7. In Fig. 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 desired value of CCR, array A can be split into an appropriate number of subarrays.

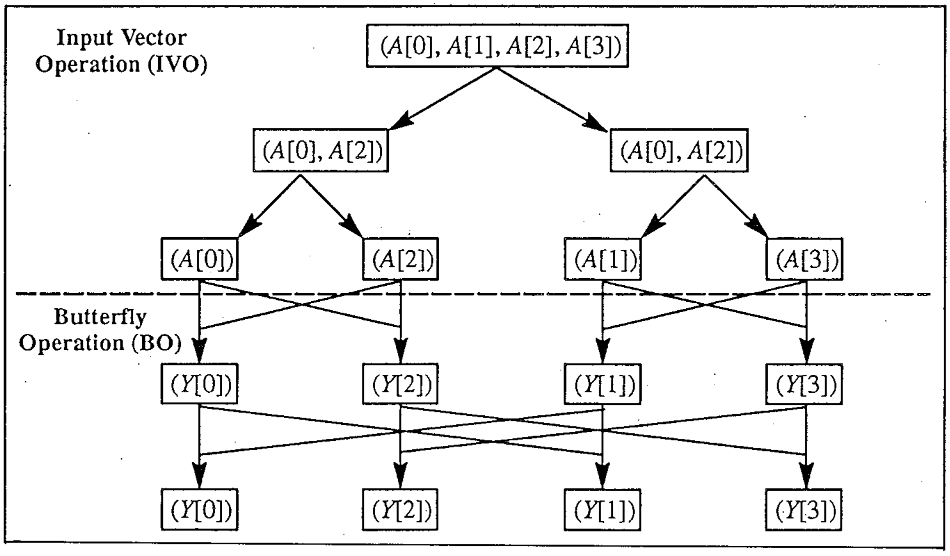


Fig. 7. The behavior of FFT with 4 points.

The DAG of FFT we have used for scheduling algorithms is shown in Fig. 8. In Fig. 8, we have 2^l-1 IVO-task, 2^l *FFT*-task, and $l \times 2^l$ BO-task, where $l > 0$. Each IVO-task with input vector size = k needs to send a vector with size = $k/2$ to its immediate successors. Each *FFT*-task or BO-task with input vector size = k needs to send a vector with size = k to its immediate successors. To obtain the value of CCR of the corresponding DAG of an FFT program, we need to know the computation cost of each IVO-task, FFT-task and BO-task with vector size = k and the communication cost of sending k bytes and $k/2$ bytes of data from one processor to another. Since each IVO-task, FFT-task and BO-task is implemented as a procedure, we measure the execution time of each IVO-task, FFT-task and BO-task with different vector sizes by running the corresponding procedures on an NCUBE-2. We also measure the communication time of sending k bytes and $k/2$ bytes of data from one processor to another on an NCUBE-2. From the measurements, we have found that if the number of FFT-tasks in Fig. 8 is less than or equal to 16 and array A has 1024 elements, then we have CCR < 0.5. If the number of FFT-tasks in Fig. 8 is equal to 32 and array A has 1024 elements, then we have $0.5 \le$ CCR ≤ 1. If the number of FFT-tasks in Fig. 8 is greater than or equal to 64 and array A has 1024 elements, then we have CCR ≥ 1. According to the values of CCR which we want, we generate DAGs with different numbers of tasks.

The experimental results of scheduling FFT programs on an NCUBE-2 are shown in Fig. 9 to Fig. 11. In Fig. 9 to Fig. 11, the input vector size of FFT is 1024. The predicted speedups were obtained by using a simulation approach; that is, the corresponding DAG of an FFT program was scheduled on a simulated hypercube (without system overhead). The real speedups were obtained by executing the corresponding DAG of an FFT program on an NCUBE-2 (with system overhead).

Fig. 9 shows the speedups of scheduling algorithms for a DAG G_4 with 511

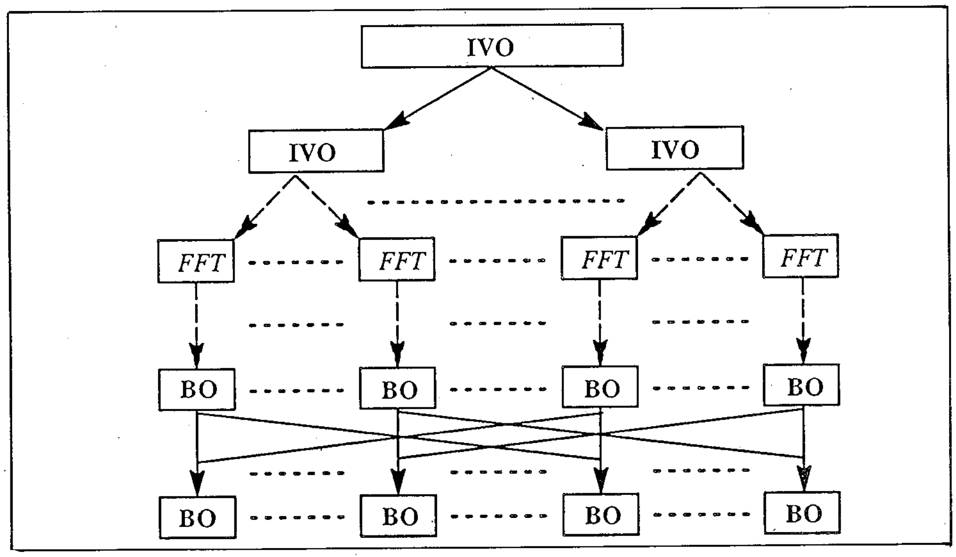


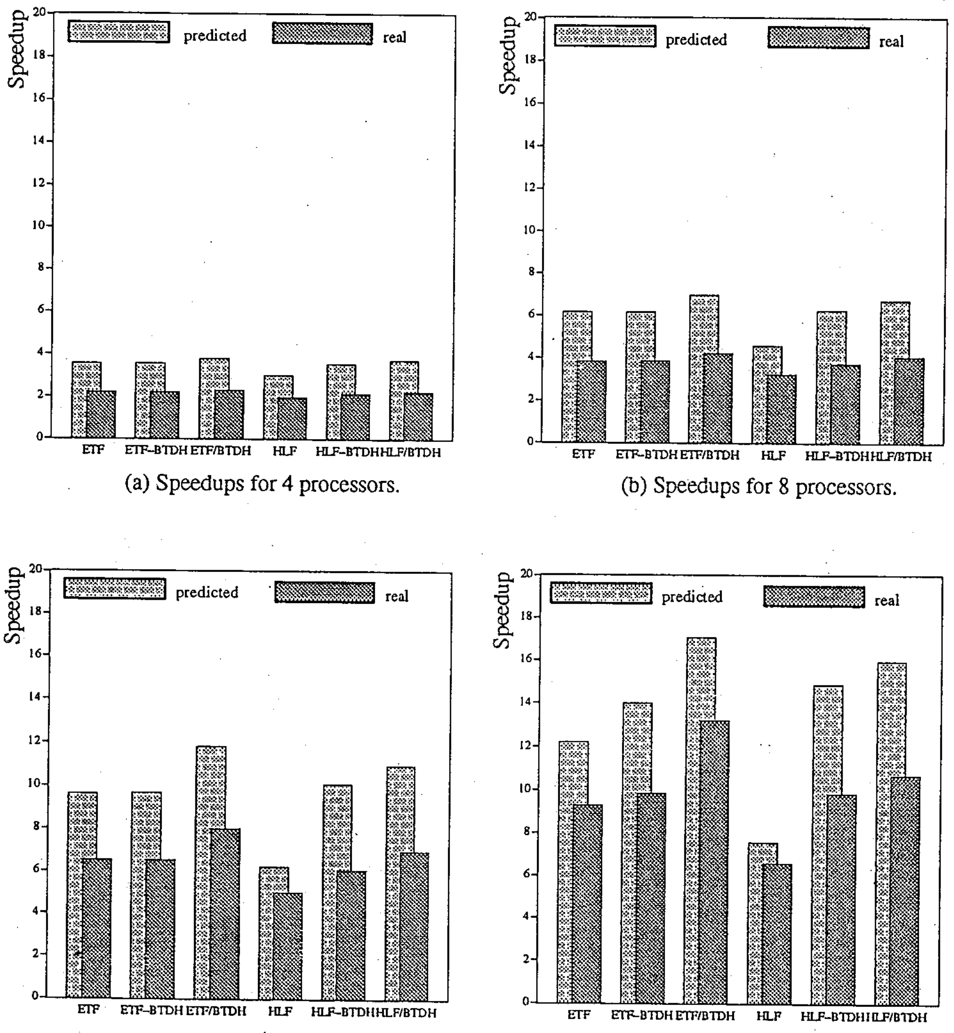
Fig. 8. The DAG generated for scheduling.

tasks (63 IVO-tasks, 64 FFT-tasks, and 384 BO-tasks). The values of CCR and GP of G_4 are 1.01 and 24.7, respectively. From Fig. 9, we can see that the system overhead can greatly offset the speedups we predicted. This is due to the fine grain nature of the FFT-tasks and the BO-tasks (which represent the majority of the tasks). Further, the amount of communication sent to the FFT-tasks and the BO-tasks is relatively small (that is ,the grain size of the communication is small; thus, system overhead plays an important role). Fig. 10 shows the speedups of scheduling algorithms for a DAG G_5 with 223 tasks (31 IVO-tasks, 32 FFT-tasks, and 160 BO-tasks). The values of CCR and GP of G_5 are 0.64 and 19.05, respectively. In Fig. 10, the system overhead can offset some of speedups we predicted (However, it is not so severe as that of G_4). Fig. 11 shows the speedups of scheduling algorithms for a DAG G_6 with 95 tasks (15 IVO-tasks, 16 FFT-tasks, and 64 BO-tasks). The values of CCR and GP of G_6 are 0.47 and 12.18, respectively. From Fig. 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 (that is, the number of bytes transferred per message) increase, 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 Fig. 9 to Fig. 11, in general, the speedups for scheduling algorithms have the following order:

$$(\mathrm{LSA/BTDH})_{speedup} \geq (\mathrm{LSA\text{-}BTDH})_{speedup} \geq (\mathrm{ETF})_{speedup} \geq (\mathrm{HLFET})_{speedup},$$

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



(a) Speedups for 4 processors.

(b) Speedups for 8 processors.

(c) Speedups for 16 processors.

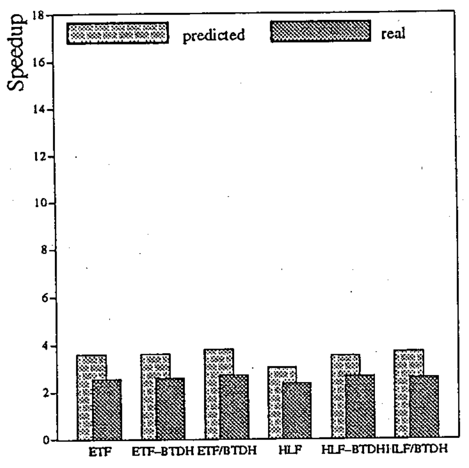
(d) Speedups for 32 processors.

Fig. 9. Speedups of scheduling algorithms for FFT with input vector size = 1024, GP = 24.7, and CCR = 1.01.

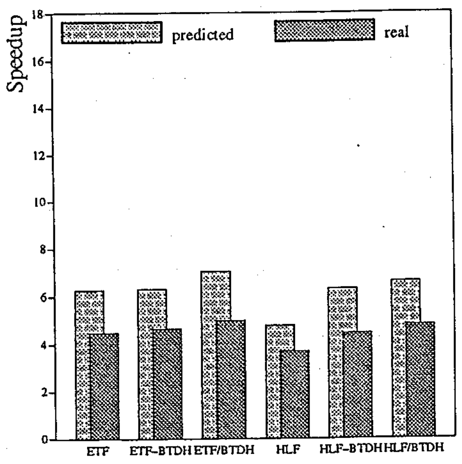
6.2 Discussion

Based on the above experimental results, we can draw the following conclusions.

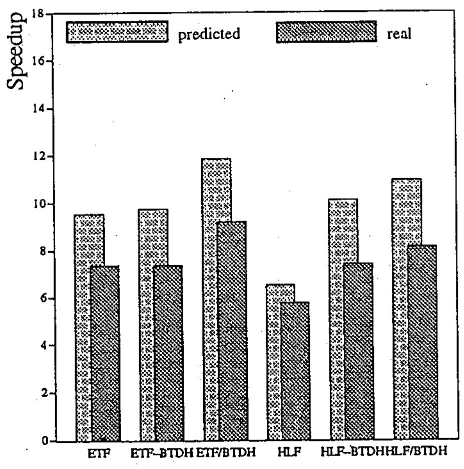
1) Grain size determination has a great impact on speedups of scheduling algorithms. If a DAG is too fine, the system overhead will sometimes greatly offset the speedups of scheduling algorithms. If a DAG is too coarse, the value of GP may be too small; that is, too much parallelism may



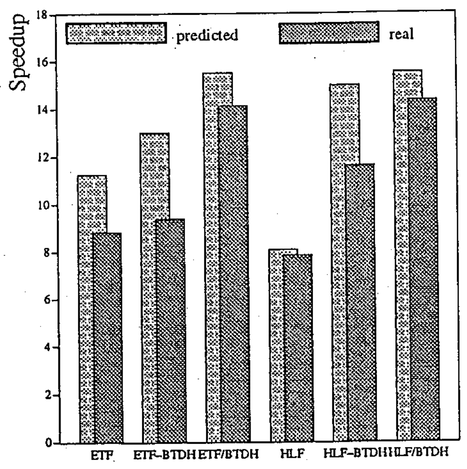
(a) Speedups for 4 processors.



(b) Speedups for 8 processors.



(c) Speedups for 16 processors.

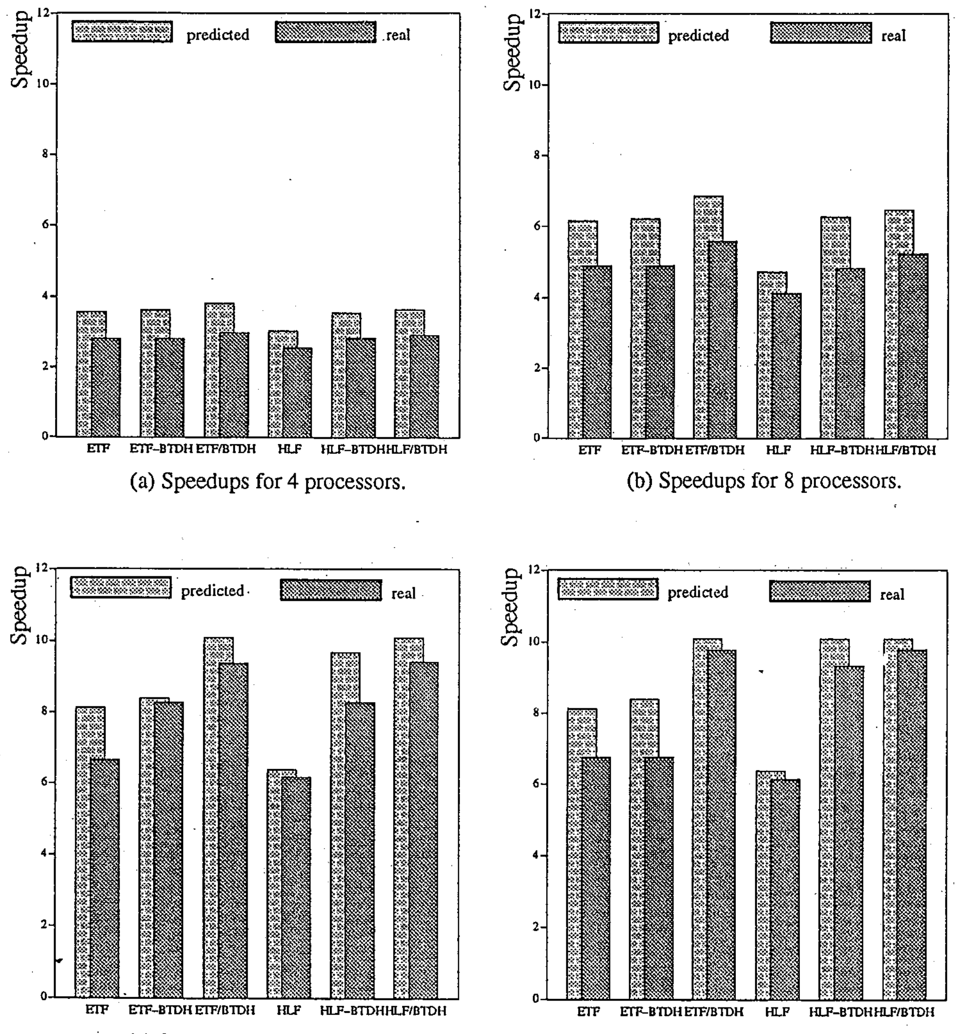


(d) Speedups for 32 processors.

Fig. 10. Speedups of scheduling algorithms for FFT with input vector size = 1024, GP = 19.05, and CCR = 0.64.

be lost. In our experimental results, the best speedups, in terms of GPP, of scheduling algorithms were obtained when DAGs with $0.25 \leq \text{CCR} \leq 0.75$ are used.

2) The relative speedup comparisons between different algorithms based on experimental results are similar to those based on simulation results; that is, the simulation results accurately show whether a scheduling algorithm will give a better speedup as compared to another scheduling algorithm. This is also true for cases in which the experimental results are not close to simulation results due to neglect of system overhead in the latter.



(a) Speedups for 4 processors.

(b) Speedups for 8 processors.

(c) Speedups for 16 processors.

(d) Speedups for 32 processors.

Fig. 11. Speedups of scheduling algorithms for FFT with input vector size = 1024, GP = 12.18, and CCR = 0.47.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have discussed the applications of BTDH for LSAs. There are two ways to use BTDH with LSAs. (1) BTDH can be used with an LSA to form a new scheduling algorithm (LSA/BTDH). (2) It can be used as a pure optimization algorithm for an LSA (LSA-BTDH). We have applied BTDH to two LSAs, HLFET and ETF, for both applications. We have studied the performance of BTDH for LSAs using simulation as well as on an NCUBE-2. Three parameters, GP, CCR, and PN, have been simulated. From the simulation, we have drawn the following conclusions. (I) Given a DAG, the GP of a DAG can accurately predict the number of processors to be used such that a good scheduling length and good resource utilization (or efficiency) can be achieved simultaneously. (II) In terms of the scheduling lengths of scheduling algorithms, in general, we have $(\text{LSA/BTDH})_{scheduling\ length} \leq (\text{LSA-BTDH})_{scheduling\ length} \leq (\text{ETF})_{scheduling\ length} \leq (\text{HLFET})_{scheduling\ length}$. In terms of the execution time of scheduling algorithms, in general, we have $(\text{LSA/BTDH})_{time} \geq (\text{LSA-BTDH})_{time} \geq (\text{ETF})_{time} \geq (\text{HLFET})_{time}$. Experimental results of scheduling FFT programs on an NCUBE-2 have been presented. The results confirm our simulation results and show that the speedups of LSA/BTDH and LSA-BTDH are better than those of LSAs.

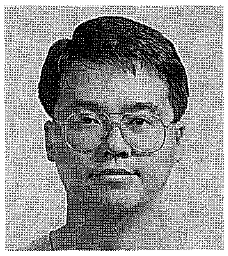
In this paper, we have only used a semi-automatic method to generate DAGs of FFT programs. However, it is important and difficult to translate real programs to DAGs automatically. In the future, we plan to develop a program transformation tool which can translate a sequential program into a corresponding DAG automatically.

REFERENCES

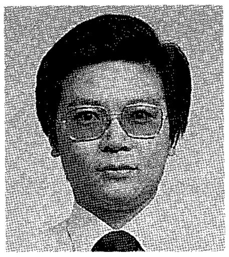
1. Al-Mouhamed, M.A., "Lower bound on the number of processors and time for scheduling precedence graphs with communication costs," *IEEE Transactions on Software Engineering*, Vol. 16, No. 12, 1990, pp. 1390-1401.
2. Anger, F.D., Hwang, J.J. and Chow, Y.C., " Scheduling with sufficient loosely coupled processors," *Journal of Parallel and Distributed Computing*, Vol. 9, 1990, pp. 87-92.
3. Baxter, J. and Patel, J.H., "The LAST algorithm: a heuristic-based static task allocation algorithm," in *Proceedings of International Conference on Parallel Processing*, Vol. 2, 1989, pp. 217-222.
4. Chaudhary, V. and Aggarwal, J.K., "Generalized mapping of parallel algorithms onto parallel architectures," *Proceedings of International Conference on Parallel Processing*, Vol. 2, 1990, pp. 137-141.
5. Chung, Y.C. and Ranka, S., "An optimization approach for static scheduling of directed-acyclic graphs on distributed memory multiprocessors," NPAC-SCCS Report, Syracuse University, 1991.
6. El-Rewini, H. and Lewis, T.G., "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*, Vol. 9, 1990, pp. 138-153.
7. Gupta, R. and Soffa, M.L., "Region scheduling: an approach for detecting and redistributing parallelism," *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, 1990, pp. 421-431.
8. Hwang, J.J. et al, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Journal of Computing*, Vol. 18, 1989, pp. 244-257.
9. Jung, H., Kirousis, L. and Spirakis, P., "Lower bounds and efficient algorithms for multiprocessor scheduling of DAGs with communication delay," in *Proceedings of the ACM Symposiums of Parallel Algorithms and Architectures*, 1989, pp. 254-264.

10. Kim, S.J. and Browne, J.C., "A general approach to mapping of parallel computations upon multiprocessor architectures," in *Proceedings of International Conference on Parallel Processing*, 1988, pp. 1-8.
11. Kruatrachue, B., *Static Task Scheduling and Grain Packing in Parallel Processing Systems*, Ph.D. dissertation, Electrical and Computer Engineering Department, Oregon State University, Corvallis, 1987.
12. Lee, B., Hurson, A.R. and Feng, T.Y., "A vertically layered allocation scheme for data flow systems," *Journal of Parallel and Distributed Computing*, Vol. 11, 1991, pp. 175-187.
13. Lee, C.Y. et al, "Multiprocessor scheduling with interprocessor communication delays," *Operations Research Letters*, Vol. 7, 1988, pp. 141-147.
14. Lin, K.J., Chung, J.Y. and Liu, J., "Scheduling real-time computations on hypercubes with load balancing," in *Proceedings of The Fifth Conference of Distributed Memory Multiprocessors*, 1990, pp. 975-983.
15. Manohara, S. and Thanisch, P., "Assigning dependency graphs onto processor networks," *Parallel Computing*, Vol. 17, 1991, pp. 63-73.
16. Papadimitriou, C.H. and Ullman, J.D., "A communication-time tradeoff," *SIAM Journal of Computing*, Vol. 14, No. 4, 1987, pp. 639-646.
17. Papadimitriou, C.H. and Yannakakis, M., "Towards an architecture-independent analysis of parallel algorithms," *SIAM Journal of Computing*, Vol. 19, 1990, pp. 322-328.
18. Ramamritham, K., Stankovic, J.A. and Shiah, P.F., "Efficient scheduling algorithms for real-time multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, 1990, pp. 184-194.
19. Sih, G.C. and Lee, E.A., "Scheduling to account for interprocessor communication within interconnection-constrained processor networks," in *Proceedings of International Conference on Parallel Processing*, Vol. 1, 1990, pp. 9-16.
20. Stone, H.S., "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Transactions on Software Engineering*, Vol. 3, No. 1, 1977, pp. 85-93.
21. Wu, M.Y. and Gajski, D.D., "Hypertool: a programming aid for message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 3, 1990, pp. 330-343.
22. Adam, T.L., Chandy, K.M. and Dickson, J.R., "A comparison of list schedules for parallel processing systems," *Communication of ACM*, Vol. 17, No. 12, 1974, pp. 685-690.
23. Coffman, E.G. Jr., *Computer Jr., and Job-Shop Scheduling Theory*, Wiley, New York, 1976.
24. Friesen, D.K., "Tighter bound for LPT scheduling on uniform processors," *SIAM Journal of Computing*, Vol. 16, No. 3, 1987, pp. 554-560.
25. Graham, R.L. "Bounds on multiprocessor timing anomalies," *SIAM Journal of Applied Mathematics*, Vol. 17, No. 2, 1969, pp. 416-429.
26. Granski, M., Koren, I. and Silberman, G.M., "The effect of operation scheduling on the performance of a data flow computer, " *IEEE Transactions on Computers*, Vol. 36, No. 9, 1987, pp. 1019-1029.
27. Hochbaum, D.S. and Shmoys, D.B., "A polynomial approximation scheme for scheduling on uniform processors: using the dual approximation ap-

proach," *SIAM Journal of Computing*, Vol. 17, No. 3, 1988, pp. 539-551.
28. Hu, T.C., "Parallel sequencing and assembly line problems," *Operations Research*, 1961, pp. 841-848.
29. Kasahara, H. and Narita, S., "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Transactions on Computers*, Vol. 33, No. 11, 1984, pp. 1023-1029.
30. Leung, J.Y.T. and Young, G.H., "Minimizing schedule length subject to minimum flow time," *SIAM Journal of Computing*, Vol. 18, No. 2, 1989, pp. 314-326.
31. Shirazi, B. and Wang, M., "Analysis and evaluation of heuristic methods for static task scheduling," *Journal of Parallel and Distributed Computing*, Vol. 10, 1990, pp. 222-232.
32. Simons, B.B. and Warmuth, M.K., "A fast algorithm for multiprocessor scheduling of unit-time jobs," *SIAM Journal of Computing*, Vol. 18, No. 4, 1989, pp. 690-710.
33. Garey, M.R. and Johnson, D.S., *Computers and Intractability - A Guide to the Theory of NP- Completeness*, W. H. San Francisco, 1979.

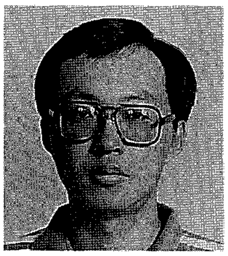


Yeh-Ching Chung (鍾葉青) was born in 1961. He received a B.S. degree in computer science from Chung Yuan Christian University in 1983, and an M.S. and a Ph.D. degrees in computer and information science from Syracuse University in 1988 and 1992, respectively. Since 1992, he has been an Associate Professor in the Department of Information Engineering and Computer Science at Feng Chia University. His research interests include parallel compilers, parallel programming tools, mapping, scheduling, and load balancing.



Chia-Cheng Liu (劉嘉政) received a B.S. degree in computer science from Feng Chia University, a M.S. degree in electrical engineering from National Taiwan University and a Ph.D. degree in computer science from National Tsing Hua University.

He is currently associate professor in the Department of Information Engineering and executive vice director in Information Processing Center, Feng Chia University. His research interests include distributed and multiprocessing systems, computer aid design for VLSI, and parallel algorithms.



Jenshiuh Liu (劉振緒) received B.S. and M.S. degrees in nuclear engineering from National Tsing-Hua University, and M.S. and Ph.D. degrees in computer science from Michigan State University in 1979, 1981, 1987 and 1992, respectively.

Since 1992, he has been an associate professor in the Department of Information Engineering and Computer Science at Feng Chia University, Taiwan. His research interests include parallel and distributed systems, computer networks, parallel and distributed simulation, and performance evaluation.