다중 처리기 시스템을 위한 효율적인 리스트 스케줄링 알고리즘

박 경 린† · 추 현 승†† · 이 정 훈†

요 약

비순환 방향 그래프(DAG : Directed Acyclic Graph) 혹은 타스크 그래프로 표현이 된 병렬 프로그램을 다중 처리기 시스템에 효과적으로 스케줄하는 문제는 지난 수 십년 동안 주요 연구 과제였으며, 리스트 스케줄링은 이 문제를 해결하기 위한 대표적인 접근 방법이다. 리스트 스케줄링 알고리즘들은 입력 DAG의 어떤 노드 혹은 엣지들에 우선 순위를 부여한 다음, 이 우선 순위에 위해 스케줄을 작성한다. 본 논문은 효과적인 우선 순위 부여 방식을 가지는 리스트 스케줄링 알고리즘을 제안하고, 제안된 알고리즘에 대하여 최악의 경우 성능과 최적의 조건을 분석한다. 성능 비교 결과는 본 논문에서 제안된 리스트 스케줄링 알고리즘이 통신 오버헤드가 큰 시스템에서 기존의 리스트 스케줄링 알고리즘들보다 더 작은 병렬 수행 시간을 얻음을 보여준다. 또한 성능의 향상은 입력 타스크 그래프가 밀집된 형태일수록 그리고 병렬의 정도가 높을수록 더욱 커짐을 보여준다.

An Efficient List Scheduling Algorithm for Multiprocessor Systems

Gyung-Leen Park† · Hyun-Seung Choo†† · Jeong-Hoon Lee†

ABSTRACT

Scheduling parallel tasks, represented as a Directed Acyclic Graph (DAG) or task graph, on a multiprocessor system has been an important research area in the past decades. List scheduling has been a typical approach for solving the problem. List scheduling algorithms assign priorities to a node or an edge in an input DAG, and then generate a schedule according to the assigned priorities. This paper proposes a list scheduling algorithm with effective method of priority assignments. The paper also analyzes the worst case performance and optimality condition for the proposed algorithm. The performance comparison study shows that the proposed algorithm outperforms existing scheduling algorithms especially for input DAGs with high communication overheads. The performance improvement over existing algorithms becomes larger as the input DAG becomes more dense and the level of parallelism in the DAG is increased.

1. Introduction

Efficient partitioning and scheduling of parallel programs onto processing elements of parallel and distributed computer systems are difficult and important issues in concurrent processing [1-7]. The process consists of partitioning a parallel program's tasks into clusters and efficiently scheduling those clusters among the processing elements of a parallel machine for execution. Once an application program is partitioned into clusters or tasks, it can be represented by a *DAG* (Directed Acyclic

† 정 회 원 : 제주대학교 전산통계학과 교수
†† 종신회원 : 성균관대학교 전기전자 및 컴퓨터공학부 교수
논문접수 : 1999년 5월 12일, 심사완료 : 1999년 7월 20일

Graph), or a *task graph*, which represents the precedence constraints of the program tasks. The goals of the scheduling process are to efficiently utilize resources and to achieve performance objectives of the application (e.g., to minimize program parallel execution time).

Since it has been shown that the multiprocessor scheduling problem is NP-complete, many researchers have proposed scheduling algorithms based on heuristics [8]. The scheduling algorithms can be classified into two general categories : algorithms that employ task duplication and algorithms that do not employ task duplication. Task duplication algorithms attempt to reduce communication overhead by duplicating tasks that would otherwise require interprocessor communications if the tasks were not duplicated [9-18]. One of the major problems with task duplication is the issue of data distribution and preserving of data integrity. This paper assumes that the system does not allow task duplication.

Most of the non-duplication scheduling methods can be classified as either a *clustering algorithm* [19, 20] or a *list scheduling algorithm* [1, 8, 21, 22]. The clustering algorithms basically perform the following operations :

1. Initially, each task is considered to be a cluster.
2. An edge between two clusters is selected according to a priority assigned to the edges by the clustering algorithm.
3. The edge is removed (call *edge zeroing*) if it satisfies certain conditions specified by the algorithm. Once an edge is zeroed, the two clusters connected by that edge will be merged into one cluster.
4. Steps 2 and 3 are repeated until all the edges are examined.
5. The clusters are assigned to the processors in the target system.

The List scheduling algorithms maintain a list of node according to their priorities. A list scheduling algorithm repeatedly carries out the following steps :

1. Tasks ready to be assigned (a task becomes ready for assignment when all of its parents are scheduled) are put onto a priority queue. The priority criteria determines the order in which task are assigned to the processors.
2. Select a "suitable Processing Element (PE)" for assignment. Typically, a suitable PE is one that can execute the task the earliest.
3. Assign the task at the head of the priority queue to this PE.
4. Repeat steps 2 and 3 until the priority queue is exhausted.

Typically, the list scheduling algorithms assume bounded number of processors while clustering algorithms assume an unbounded number of processors. This difference is not significant since these assumptions can be easily removed for each method. A more significant difference between the list scheduling and clustering algorithms is that list scheduling algorithms select *only* a ready node for assignment while clustering algorithms may select *any* node for this purpose.

The critical part in both techniques is the development of a method for assigning priorities to the nodes or the edges of the input DAG. Since a large number of different methods are proposed in the literature, this paper briefly classifies them according to the parameters used for the priority assignment : the node weight, the distance, the critical path, and some combinations of them.

The methods based on the node weight, such as that in HNF (Heavy Node First), assign a higher priority to a node with a larger computation cost [1]. The distance (defined as the sum of computation and communication costs of the nodes on a path) could be either the maximum distance from a root node to the node under consideration (top distance) or the maximum distance from the node being considered to an exit node (bottom distance). For example, HLFET (High Level First with Estimated Time) algorithm assigns a higher priority to a node with a larger bottom distance [8]. A large number of scheduling algorithms use the length of the critical path to assign priorities to the nodes and edges of a DAG. Some examples include Linear Clustering

(LC) [19] and Dominant Sequence Clustering (DSC) [20] algorithms. Finally, some algorithms use combinations of the above parameters to decide the priorities. For example, Critical Path Node-Dominate (CPND) method [22] uses the critical path and the bottom distance for assigning the priorities to the nodes in the input DAG.

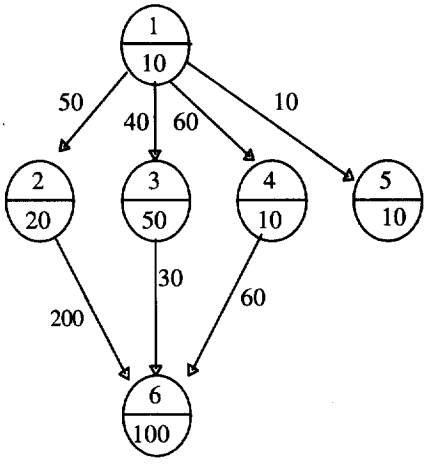
This paper proposes a new list scheduling algorithm, *Decisive Path Scheduling (DPS)* [21], which assigns the priorities to the nodes using the *decisive path* (defined in Section 2). The performance comparison study shows that the proposed algorithm outperforms existing scheduling algorithms especially for input DAGs with high communication overheads. The performance improvement over existing algorithms becomes larger for denser and more parallel DAGs.

The remainder of this paper is organized as follows. Section 2 presents the system model and the problem definition. Section 3 briefly covers the related works. The proposed scheduling algorithm is presented in Section 4. This section also contains the worst case and the optimality analysis of the proposed algorithm. The performance of the proposed algorithm is compared with that of the typical existing algorithms in Section 5. Finally, Section 6 concludes this paper.

2. System model and problem definition

A parallel program is usually represented by a Directed Acyclic Graph (DAG), which is also called a task graph. As defined in [16], a DAG consists of a tuple (V, E, T, C), where V, E, T, and C are the set of task nodes, the set of communication edges, the set of computation costs associated with the task nodes, and the set of communication costs associated with the edges, respectively. The computation cost and the communication cost represent the time for executing the task and that for communication delay, respectively. $T(V_i)$ is a computation cost for task V_i and $C(V_i, V_j)$ is the communication cost for edge $E(V_i, V_j)$ which connects task V_i and V_j. The edge $E(V_i, V_j)$ represents the precedence constraint between the node V_i and V_j. In other words, task V_j can start the execution only after the output of V_i is available to V_j. When the two tasks, V_i and V_j, are assigned to the same processor, $C(V_i, V_j)$ is assumed to be zero since intra-processor communication cost is negligible compared with the interprocessor communication cost. The weights associated with nodes and edges are obtained by estimation [23].

This paper defines two relations for precedence constraints. The $V_i \Rightarrow V_j$ relation indicates the strong precedence relation between V_i and V_j. That is, V_i is an immediate parent of V_j and V_j is an immediate child of V_i. The terms *iparent* and *ichild* are used to represent immediate parent and immediate child, respectively. The $V_i \rightarrow V_j$ relation indicates the weak precedence relation between V_i and V_j. That is, V_i is a parent of V_j but not necessarily the immediate one. $V_i \rightarrow V_j$ and $V_j \rightarrow V_k$ imply $V_i \rightarrow V_k$. $V_i \Rightarrow V_j$ and $V_j \Rightarrow V_k$ do not imply $V_i \Rightarrow V_k$, but imply $V_i \rightarrow V_k$. The relation $\rightarrow$ is transitive, and the relation $\Rightarrow$ is not. A node without any parent is called an *entry node* and a node without any child is called an *exit node*.



(Figure 1) The sample DAG

Graphically, a node is represented as a circle with a dividing line in the middle. The number in the upper portion of the circle represents the node ID number and the number in the lower portion of the circle represents the computation cost for the node. For example, for the sample DAG in (Figure 1), the entry node is node num-

ber 1 which has a computation cost of 10. In the graph representation of a DAG, the communication cost for each edge is written on the edge itself. For each node, *incoming degree* is the number of input edges and *outgoing degree* is the number of output edges.

For example, in (Figure 1), the incoming and outgoing degrees for the node V_1 are 0 and 4, respectively. A few terms are defined here for a more precise presentation.

Definition 1 : A node is called a *fork node* if its outgoing degree is greater than 1.

Definition 2 : A node is called a *join node* if its incoming degree is greater than 1.

Note that the fork node and the join node are not exclusive terms, which means that one node can be both a fork and also a join node ; i.e., both of the node's incoming and outgoing degrees are greater than one. Similarly, a node can be neither a fork nor a join node ; i.e., both of the node's incoming and outgoing degrees are one. For the sample DAG in (Figure 1), V_1 is a fork node while V_6 is a join node. Nodes V_2, V_3, V_4, and V_5 are neither fork nodes nor join nodes.

Definition 3 : The *Earliest Start Time, EST(V_i, P_k)*, and *Earliest Completion Time, ECT(V_i, P_k)*, are the times that a task V_i starts and finishes its execution on processor P_k, respectively. When the information on the processor is not necessary, they are denoted just as EST(V_i) and ECT(V_i), respectively.

Definition 4 : The critical path is the longest path from an entry node to an exit node in the graph. A *Critical Path Including Communication cost (CPIC)* is the length of the critical path including communication costs in the path while a *Critical Path Excluding Communication cost (CPEC)* is the length of the critical path excluding communication costs in the path. For the sample DAG in (Figure 1) as an example, CPIC is $T(V_1) + C(V_1, V_2) + T(V_2) + C(V_2, V_6) + T(V_6)$, which is 380. CPEC is $T(V_1) + T(V_3) + T(V_6)$, which is 160.

Definition 5 : The *level* of a node is recursively defined as follows. The level of an entry node, V_1, is one. Let $Lv(V_i)$ be the level of V_i. Then $Lv(V_1) = 1$. $Lv(V_j) = Lv(V_i) + 1$, $V_i \Rightarrow V_j$, for non-join node V_j. $Lv(V_j) = Max(Lv(V_i)) + 1$, $V_i \Rightarrow V_j$, for join node V_j. For example, the level of node V_1, V_2, V_3, V_4, V_5, and V_6 in the sample DAG are 1, 2, 2, 2, 2, and 3, respectively. Even if there was an edge from node V_1 to V_6, the level of V_6 would still be 3 since $Lv(V_6) = Max(Lv(V_i)) + 1$, $V_i \Rightarrow V_6$, for join node V_6. The level of a DAG is the maximum level of the nodes in the DAG.

Definition 6 : The *top distance* for any give node is the longest distance from an entry node to that node, excluding the computation cost of the node itself. The *bottom distance* for any given node is the longest distance from that node to an exit node, including the computation cost of the node itself. For example, the top distance of V_6, $TD(V_6)$, is 280 which is $T(V_1) + C(V_1, V_2) + T(V_2) + C(V_2, V_6)$. The bottom distance of V_2, $BD(V_2)$, is 320 which is $T(V_2) + C(V_2, V_6) + T(V_6)$.

Definition 7 : The *Decisive Path (DP)* to node V_i is the path which decides the top distance of V_i. For example, the decisive path to V_6 , $DP(V_6)$, is the path through V_1, V_2, and V_6 since the path decides the top distance of V_6. The decisive path is defined for every node in the DAG. For example, $DP(V_3)$ is the path through V_1 and V_3. The critical path becomes a special case of the decisive path defined for an exit node.

The multiprocessor scheduling process becomes a mapping of the task nodes in the input DAG to the processors in the target system with the goal of minimizing the execution time of the entire program. This paper assumes a complete graph for the target system ; i.e., any processor can communicate with any other processor. Interested readers may refer to [24] for topology issues. The execution time of the entire program after scheduling is called the *parallel time* to be distinguished from the completion time of an individual task node.

3. The related work

As discussed in the introduction, the critical issue in list scheduling and clustering algorithms is the method by which the priorities of the nodes or edges of the input DAG are decided. Since most of the scheduling algorithms use certain properties of the input DAG for deciding the priorities, we classify the existing scheduling algorithms into four categories according to the properties used : node weights, distances, the critical path, and some combinations of these parameters. This section briefly covers a typical scheduling algorithm in each category. The algorithms are used later for performance comparison against the proposed method.

The Heavy Node First(HNF) algorithm [1] assigns the nodes in a DAG to the processors, level by level. At each level, the scheduler selects the eligible nodes for scheduling in descending order based on computational weight, with the heaviest node (i.e. the node which has the largest computation cost) selected first. The node is selected arbitrarily if multiple nodes at the same level have the same computation cost. The selected node is assigned to a processor which provides the earliest start time to the node.

The High Level First with Estimated Time (HLFET) algorithm [8] also assigns the nodes in a DAG to the processors, level by level. At each level, the scheduler assigns a higher priority to a node with a larger bottom distance. The node with the highest priority is assigned a processor which provides the earliest start time for the node.

The Linear Clustering (LC) algorithm [19] is a traditional critical path based scheduling method. The scheduler identifies the critical path, removes the nodes in the path and the edges attached to the nodes from the DAG, and assigns them to a linear cluster. The process is repeated until there are no task nodes remaining in the DAG. The clusters are then scheduled onto a processor.

The Dominant Sequence Clustering (DSC) [20] algorithm is based on the dominant sequence which is a dynamic version of the critical path. The dominant sequence is the longest path of the task graph for un-scheduled nodes [20]. Initially, the dominant sequence is same as the critical path for the original input DAG. At each step, the scheduler selects one edge in the dominant sequence and zeros it if the edge zeroing reduces the length of the dominant sequence. The scheduler identifies a new dominant sequence since the edge zeroing may change the longest path. The operations are repeatedly carried out until all the edges are examined.

In the Critical Path Node-Dominate (CPND) algorithm [22], the nodes in the input DAG are classified into three categories : Critical Path Node (CPN), In Branch Node (IBN), and Out Branch Node (OBN). A CPN is the node on the critical path while an IBN is a nodes which is not a CPN but from which there is a path reaching a CPN. An OBN is a node which is neither a CPN nor an IBN. The CPND algorithm tries to schedule the CPNs first. If there are unscheduled IBNs which are parents of a CPN, they are scheduled in the descending order of their bottom distances. OBNs are scheduled after all CPNs and IBNs are scheduled, also in the descending order of their bottom distances. CPND algorithm obtains a schedule using a FAST (Fast Assignment using Search Technique) scheduler [22]. A series of optimizations are then applied to the original schedule to improve the performance of the application. In this paper we use the original, un-optimized CPND schedules since we are interested in investigating the effectiveness of the priority assignment methods. The optimization routines can also be applied to the proposed algorithm later on.

The time complexity and the priority criteria for the aforementioned algorithms are summarized in <Table 1> The information for the proposed algorithm (DPS) is also included in this table and will be discussed shortly.

As an illustration, (Figure 2), presents the schedule obtained by each algorithm for the sample DAG of (Figure 1). In this example, P_i represents processing element i ; PT is the Parallel Time of the DAG ; and [EST(V_i, P_k), i, ECT(V_i, P_k)] represents the earliest start time and earliest completion time of task i. In the first

〈Table 1〉 Characteristics of scheduling algorithms

ALGORITHM	PRIORITY CRITERIA	COMPLEXITY
HNF	Level and Node Weight	O(VlogV)
HLFET	Level and Bottom Distance	$O(V^2)$
LC	Critical Path	$O(V^3)$
DSC	Dominant Sequence	$O(V^2logV)$
CPND	Critical Path and Bottom Distance	$O(V^2)$
DPS	Decisive Path	$O(V^2)$

line of (Figure 2), (a), for example, [0, 1, 10] represents that task V_1 starts and completes its own execution at time 0 and 10 respectively, on processor P1. The figure also shows the delay due to communication time. In (Figure 2).(b) as an example, the start time of task V_6 is 140 since its immediate parent V_4 completes its execution at time 80 and the communication takes 60 time units. In this example, the proposed algorithm provides the best parallel time compared to the other algorithms under consideration.

p1 : [0, 1, 10] [10, 3, 60] [60, 2, 80] [140, 6, 240]
p2 : [70, 4, 80]
p3 : [20, 5, 30]

(a) The Schedule by HNF(PT = 240)

p1 : [0, 1, 10] [10, 2, 30] [30, 3, 80] [140, 6, 240]
p2 : [70, 4, 80]
p3 : [20, 5, 30]

(b) The Schedule by HLFET(PT = 240)

p1 : [0, 1, 10] [10, 2, 30] [140, 6, 240]
p2 : [50, 3, 100]
p3 : [70, 4, 80]
p4 : [20, 5, 30]

(c) The Schedule by LC(PT = 240)

p1 : [0, 1, 10] [10, 2, 30] [50, 3, 100] [100, 4, 110] [110, 6, 210]
p2 : [20, 5, 30]

(d) The Schedule by DSC(PT = 210)

p1 : [0, 1, 10] [10, 2, 30] [30, 3, 80] [140, 6, 240]
p2 : [70, 4, 80]
p3 : [20, 5, 30]

(e) The Schedule by CPND(PT = 240)

p1 : [0, 1, 10] [10, 2, 30] [30, 4, 40] [40, 3, 90] [90, 6, 190]
p2 : [20, 5, 30]

(f) The Schedule by DPS(PT = 190)

(Figure 2) Schedules for the sample DAG

4. The proposed algorithm

4.1 Motivation

The basic heuristic behind various multiprocessor scheduling algorithms is that we can reduce the parallel time by first scheduling the task node which will have the most impact on the parallel time. For example, HNF first schedules the heaviest node (the node with the highest computation time), assuming that the heaviest node has more effect on the parallel time than others. The DSC, LC, and CPND algorithms focus on the critical path since it will most likely decide the parallel time of the application. The proposed algorithm, DPS, focuses on the "decisive path" since the length of the decisive path to a node most often determines its starting time. Note that the critical path is a special decisive path defined only for an exit node.

4.2 Algorithm description

A high level description of the proposed algorithm is presented in (Figure 3). In step (1), DPS transforms an input DAG to a DAG with only one entry node and only one exit node. The transformation can be done simply by adding a dummy entry node and a dummy exit node with computation costs of zero. The dummy entry node is connected to the actual entry nodes with communication costs of zero. Similarly, the dummy exit node is connected to the actual exit nodes in the same way. Any task graph with multiple entry nodes and/or exit nodes can be scheduled by DPS algorithm since the task graph can be easily transformed into a task graph with only one entry node and one exit node in step (1) without violating any constraint in the original task graph. Step (2) identifies the decisive paths to all the nodes in the transformed input DAG. The decisive path to the dummy exit node becomes the critical path of the DAG.

Step (3) builds the "task_queue" which queues all the DAG nodes, prioritized based on the lengths of their decisive paths. The priorities are decided as follows : DPS puts the CPNs into the task_queue in the ascending order of their top distances (parents first) if there is no IBN for a given CPN. If there are some IBNs reaching a CPN, the IBN belonging to the decisive path of the CPN is selected first among the un-queued IBNs. The

same procedure is carried out recursively if an IBN has parents which are not queued yet. After all the parents are queued in the task_queue, the CPN is inserted, as shown in the search_and_put() procedure . Finally, DPS assigns the task_queue tasks (in FIFO order) to the processing elements (PEs). At each step of the assignment, the selected PE provides the earliest start time for the task under consideration, taking into account all the communications from the task's parents (i.e., find a suitable PE for assignment). If the completion time of a task is larger than the sum of all the computation costs of the nodes, DPS assigns all the nodes to one processor and Sexits from the algorithm as shown in steps (7) and (8).

DPS Algorithm

(1) Transform the input DAG so that the DAG has only one entry and only one exit node ;
(2) Identify the decisive path to each node ;
    /* the decisive path to the exit node becomes the critical path, CP */
(3) task_queue = build_task_queue(CP) ;
(4) for each task,V_i, in the task_queue in a FIFO manner
(5) find the suitable processor for V_i ;
(6) schedule V_i on the suitable processor ;
(7) if $ECT(V_i) \geq \Sigma T(V_k)$, $\forall V_k$
(8) uni_schedule() ;
(9) exit(0) ;
(11) endif
(12) endfor

build_task_queue(CP)

/* Let CPN be a set of nodes belonging to CP. NQ is a set of nodes which are not in the task queue yet. Initially, NQ contains all the nodes in the input DAG. */
(13) while ($NQ \neq \varnothing$)
(14) for each task V_i, $distance[V_i] < distance[V_j]$, $\forall V_j$, V_i, $V_j \in CPN$, V_i, $V_j \in NQ$
(15) if $\forall V_k \notin NQ$, $V_k \Rightarrow V_i$
(16) put V_i into the task queue ;
(17) NQ = NQ - {V_i} ;
(18) else
(19) search_and_put(V_i) ;
(20) endif
(21) endfor
(22) endwhile
(23) return the task queue ;

search_and_put(V_i)

(24) for V_d, $distance[V_d] + C(V_d, V_i) \geq distance[V_o] + C(V_o, V_i)$, $\forall V_o$, $V_d \Rightarrow V_i$, $V_o \Rightarrow V_i$, V_d, $V_o \in NQ$
    /* V_d is the iparent of and in the decisive path to V_i */
(25) search_and_put(V_d) ;
(26) put V_d into the task queue ;
(27) NQ = NQ - {V_d} ;
(28) endfor

uni_schedule()

(29) remove the schedule obtained so far ;
(30) schedule all the tasks on one processor ;

(Figure 3) Description of the DPS algorithm

Step (2) takes $O(V^2)$ time for identifying the decisive paths to all the nodes. Step (3) also takes $O(V^2)$ time since it examines all the edges in the input DAG. If roughly estimated, the complexity of *build_task_queue* in step (3) would be O(VE) since the while loop in step (13) takes O(V) and the for loop in step (14) examines all the edges. However, note that all the edges associated with *each node* are examined in step (14). The number of edges examined becomes the number of the edges in the input DAG. Therefore, the complexity of the routine becomes O(E) which is $O(V^2)$. Step (5) takes O(V) time since |V| processors are enough for this scheduler. Thus, the time complexity of the DPS algorithm becomes $O(V^2)$.

4.3 Analysis of the proposed algorithm

The worst case analysis of the scheduling algorithm is important especially for real-time systems. At first, we will show the worst case performance and the optimality condition of the DPS algorithm for a tree structured input DAG. The tree structured input DAG means a task graph which does not contain a join node. Then the worst case performance analysis for a general input DAG is presented. The notations used in the proofs are first summarized :

- V_r : the entry node.
- V_e : the exit node.
- $V_{k,a}$: node V_k whose level is a.
- V_p : an iparent[1] of V_e, which means that V_p is the exit node in the original input DAG before the transformation.
- $LDP(V_i)$: the length of the decisive path to the task node V_i.
- $DPN(V_i)$: a set of nodes on the decisive path to the task node V_i.
- FN : a set of fork nodes.

For a tree structured input DAG not containing a join node, the worst case parallel time obtained by the DPS algorithm is $\max_p\{\Sigma T(V_j) + \Sigma C(V_i, V_j)\}$, $V_i \Rightarrow V_j$, $V_i \in$

1) Please refer to section 2 for the definitions of iparent and ichild.

FN, V_i, $V_j \in DPN(V_p)$, $\forall V_p$, $V_p \Rightarrow V_e$. That is, the worst case parallel time is the largest $ECT(V_p)$ which is the sum of computation costs of the nodes on the path to V_p plus the sum of the communication costs from *only* the fork nodes on the path. Theorem 1 proves this assertion by induction. The proof basically says that, for any ichild[2] V_j of V_i, $ECT(V_j) = ECT(V_i) + T(V_j)$, if V_i is not a fork node while $ECT(V_j) = ECT(V_i) + C(V_i, V_j) + T(V_j)$ in the worst case if V_i is a fork node with the basis that $ECT(V_r) = T(V_r)$.

Theorem 1 : For a tree structured input DAG which does not contain a join node, the worst case parallel time obtained by DPS, PT(DPS), is $\max_p\{\sum T(V_j) + \sum C(V_i, V_j)\}$, $V_i \Rightarrow V_j$, $V_i \in FN$, V_i, $V_j \in DPN(V_p)$, $\forall V_p$, $V_p \Rightarrow V_e$.

Proof :

The parallel time is the largest $ECT(V_p)$, $\forall V_p$, $V_p \Rightarrow V_e$, since $C(V_p, V_e) = 0$ and $T(V_e) = 0$. We are going to show that $ECT(V_p) = \sum T(V_j) + \sum C(V_i, V_j)$, $V_i \Rightarrow V_j$, $V_i \in FN$, V_i, $V_j \in DPN(V_p)$, in the worst case.

1) **Basis** : For the entry node V_r, $ECT(V_r) = T(V_r)$.
2) **Inductive Hypothesis** : for $\forall V_j$, $V_i \Rightarrow V_j$
   2.1) $ECT(V_j) = ECT(V_i) + T(V_j)$, if $V_i \notin FN$.
   2.2) $ECT(V_j) = ECT(V_i) + C(V_i, V_j) + T(V_j)$ in the worst case, if $V_i \in FN$.
3) **Inductive Step** : Let P_k be the processor where V_i has been scheduled.
   3.1) If $V_i \notin FN$, $V_i \Rightarrow V_j$, the suitable PE obtained by step (5) in the algorithm will be P_k since P_k gives the earliest start time $EST(V_j) = ECT(V_i)$. Then $ECT(V_j) = ECT(V_i) + T(V_j)$.
   3.2) If $V_i \notin FN$, $V_i \Rightarrow V_j$, the suitable PE obtained by step (5) will be P_k if P_k provides a start time for V_j which is earlier than $ECT(V_i) + C(V_i, V_j)$. Otherwise, step (5) will return another processor where $ECT(V_j) = ECT(V_i) + C(V_i, V_j) + T(V_j)$. Thus, it is guaranteed that $ECT(V_j) = ECT(V_i) + C(V_i, V_j) + T(V_j)$ in the worst case.

□

It is obvious that the parallel time obtained by the DPS scheduler is always less than or equal to the sum of the computation costs of the task nodes in any DAG due to steps (7) and (8) of the algorithm (Figure 3). We will also prove that the parallel time obtained by the proposed algorithm is always less than or equal to the length of the critical path, CPIC, for any input DAG. Note that the parallel time is the same as $ECT(S_e)$, and the CPIC is the same as $LDP(S_e)$. Therefore, proving that the parallel time is always less than or equal to CPIC is equivalent to proving $ECT(S_e) \leq LDP(S_e)$.

Theorem 2 : For any input DAG, when using the DPS algorithm for scheduling, $ECT(S_e) \leq LDP(S_e)$.

Proof by induction :

1) Basis : $ECT(V_{k,2}) \leq LDP(V_{k,2})$.

At level one, $ECT(V_r) = LDP(V_r) = T(V_r)$ since V_r is the dummy entry node. Then $ECT(V_{k,2}) = ECT(V_r) + T(V_r)$ if $V_{k,2} \in CPN$. If $V_{k,2} \notin CPN$, the suitable PE is the processor where V_r is scheduled if $EST(V_{k,2}) \leq ECT(V_r) + C(V_r, V_{k,2})$. Otherwise $V_{k,2}$ will be scheduled on a different processor where $ECT(V_{k,2}) = ECT(V_r) + C(V_r, V_{k,2}) + T(V_{k,2})$. Therefore, it is guaranteed that $ECT(V_{k,2}) \leq ECT(V_r) + C(V_r, V_{k,2}) + T(V_{k,2})$. Thus, $ECT(V_{k,2}) \leq LDP(V_{k,2})$ since $LDP(V_{k,2}) = LDP(V_r) + C(V_r, V_{k,2}) + T(V_{k,2})$ for any $V_{k,2}$.

2) Inductive Hypothesis : if $ECT(V_{k,i}) \leq LDP(V_{k,i})$ then $ECT(V_{k,i+1}) \leq LDP(V_{k,i+1})$

3) Inductive Step :

$V_{k,i+1}$ will be scheduled on the processor where $V_{k,i}$, $V_{k,j} \Rightarrow V_{k,i+1}$, has been scheduled if $EST(V_{k,i+1}) \leq ECT(V_{k,j}) + C(V_{k,i}, V_{k,i+1})$. Otherwise $V_{k,i+1}$ will be scheduled on a different processor where $ECT(V_{k,i+1}) = ECT(V_{k,i}) + C(V_{k,i}, V_{k,i+1}) + T(V_{k,i+1})$. So it is guaranteed that $ECT(V_{k,i+1}) \leq ECT(V_{k,j}) + C(V_{k,j}, V_{k,i+1}) + T(V_{k,i+1})$. Thus, $ECT(V_{k,i+1}) \leq LDP(V_{k,i+1})$ if $ECT(V_{k,i}) \leq LDP(V_{k,j})$ since $LDP(V_{k,i+1}) = LDP(V_{k,j}) + C(V_{k,i}, V_{k,i+1}) + T(V_{k,i+1})$.

According to the inductive step, the completion time of any node is shorter than the length of the decisive path to that node, including the exit node. That is, the parallel time is always less than or equal to the CPIC. □

5. Performance comparison

We generated random DAGs to compare the perfor-

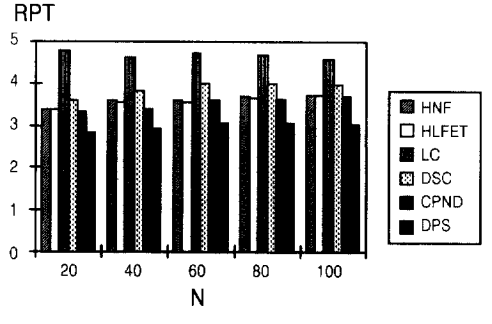
mance of the proposed DPS algorithm with that of the existing scheduling algorithms through a simulation study. We used four parameters the effects of which we were interested to investigate :

1. The number of DAG nodes : DAGs of varying sizes, including DAGs with 20, 40, 60, 80, and 100 were considered.
2. The CCR (Communication to Computation Ratio) : CCR is the ratio of the average communication cost to the average computation cost. CCR values of 0.1, 0.5, 1.0, 5.0, and 10.0 were considered.
3. The depth or maximum level of the DAG : We were interested to investigate the effect of the degree of parallelism in a DAG on the scheduling algorithms. For a fixed number of nodes, a DAG with a shorter depth (maximum level or level of the dummy exit node) displays more parallelism compared to a DAG with a longer depth. If K is the average number of siblings at a level, and N is the number of DAG nodes, then the average depth of the DAG will be N/K. Thus, for a fixed number of DAG nodes, if the average number of siblings at the same level (K) is small, the DAG represents a tall and lean graph which has a low degree of parallelism. On the other hand, a large value of K generates DAGs with more parallelism among the siblings. In our studies, we ranged the number of siblings (K) from 2 to 10.
4. The average out-degree of a node : The average out-degree of a node controls the density and amount of communication among the nodes. The larger the average out-degree, the denser the DAG is and more communications are generated. We considered the average out-degrees of 2 to K.

There are 25 combinations of the DAG sizes and the CCR values (5×5). Since there are 9 levels (from 2 to 10) for each combination and each level K has (K-1) cases of outgoing degrees (from 2 to K), there are 45 (1 + 2 +...+ 9) cases for each combination. Since we generated 5 random DAGs for each case, the number of DAGs used for the performance comparison study is 5,625 (25 × 45 × 5). The scheduling algorithms discussed in section 3 ; i.e., HNF, HLFET, LC, DSC, and CPND, were compared against the DPS algorithm.

For performance comparison, we define a *normalized* performance measure named *Relative Parallel Time (RPT)*, which is the ratio of the parallel time to CPEC. For example, if the parallel time obtained by the DPS is 200 and CPEC is 100, RPT of DPS is 2.0. If LC provides a parallel of 250 for the same DAG, then its RPT is 2.5. A smaller RPT value is indicative of a shorter parallel time. The RPT of any scheduling algorithm can not be lower than one since CPEC is the lower bound for completion time of the DAG.

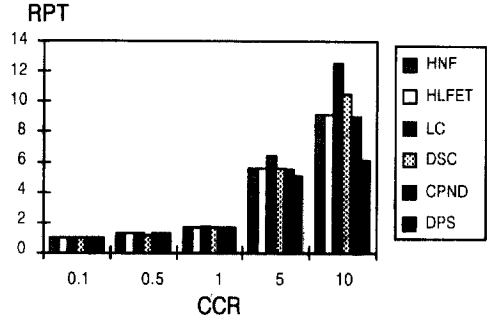
(Figure 4) compares the performance of the scheduling algorithms with respect to the number of DAG nodes. Each case in (Figure 4) shows an average RPT value from 1125 runs with varying CCR, K, and average out-degree values. The average values of CCR and K turned out to be 3.3 and 5, respectively. As shown in (Figure 4), the number of nodes does not significantly affect the relative performance of scheduling algorithms. In other words, the performance comparison shows similar patterns regardless of N. The pattern shows that for the same set of DAGs, DPS provides a shorter parallel time than the existing algorithms.



(Figure 4) Performance comparison with respect to N (for average CCR = 3.3 and K = 5)

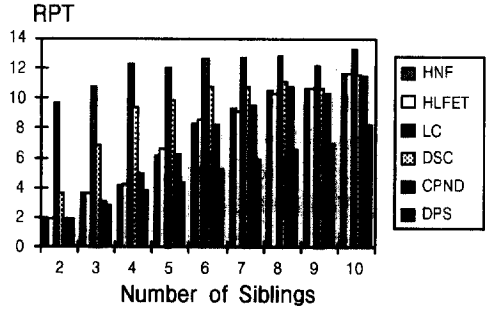
(Figure 5) depicts the RPT values for varying CCR values. When CCR is less than one, DSC slightly outperforms the other algorithms. When CCR is one, all the algorithms perform evenly. However, as the CCR

value is increased, DPS outperforms the other algorithms. The performance gap becomes larger as CCR values are increased.



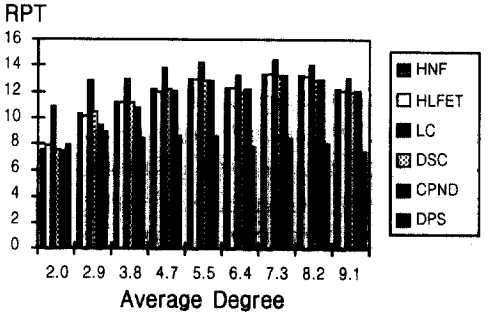
(Figure 5) Performance comparison with respect to CCR (for N = 100 and average K = 5)

(Figure 6) shows the effect of the degree of parallelism in the DAG (represented by K = average number of sibling nodes at each level) on the scheduling algorithms. Recall that for a fixed number of nodes in the DAG, a smaller K value results in a more serial DAG, while a larger K results in a more parallel DAG. In all cases the proposed DPS algorithm outperforms the other scheduling algorithms, however, the performance gap becomes more pronounced for DAGs with a higher degree of parallelism. This is an important result because it shows that the decisive path heuristic does a good job of discriminating the nodes in the difficult case of having many parallel nodes as well as the easy case of having many serial nodes in the input DAG.



(Figure 6) Performance comparison with respect to number of siblings (K) (for N = 100 and CCR = 10)

Finally, (Figure 7) depicts the performance results when the amount of communication, represented by the average out-degree of the nodes, in the DAG is varied. It seems the studied scheduling algorithms are not sensitive to the degree of communication (or dependency) in the DAG. The relative performances remain fixed for varying average node out-degrees. However, in almost all cases, DPS outperforms the other algorithms.



(Figure 7) Performance comparison with respect to average out-degree of a node (for N = 100, CCR = 10, and K = 10)

6. Conclusion

One of the critical issues in a list scheduling algorithm is the development of a method for assignment of priorities to the nodes or edges of an input DAG. In this paper we proposed a novel method, called decisive path scheduling, for determining node priorities in a list scheduling algorithm. Through an extensive performance study, it is shown that the proposed algorithm outperforms many of the existing list scheduling , as well as clustering, algorithms. The paper also establishes an optimality condition and provides a worst-case analysis of the proposed algorithm for a tree structured DAG which does not contain a join node.

Acknowledgment

We would like to express our appreciation to Dr. Tao Yang and his research group for providing the source code for the DSC schedulers which was used in our performance comparison study.

References

[1] B. Shirazi, M. Wang, and G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling," *Journal of Parallel and Distributed Computing*, Vol.10, No.3, 1990, pp.222-232.

[2] B. Shirazi, A. R. Hurson, "Scheduling and Load Balancing : Guest Editors' Introduction," *Journal of Parallel and Distributed Computing*, Dec. 1992, pp. 271-275.

[3] B. Shirazi, A. R. Hurson, "A Mini-track on Scheduling and Load Balancing : Track Coordinator's Introduction," *Hawaii Int'l Conf. on System Sciences (HICSS-26)*, Jan. 1993, pp.484-486.

[4] B. Shirazi, A. R. Hurson, K. Kavi, "Scheduling & Load Balancing," *IEEE Press*, 1995.

[5] B. Shirazi, H.-B. Chen, and J. Marquis, "Comparative Study of Task Duplication Static Scheduling versus Clustering and Non-Clustering Techniques," *Concurrency : Practice and Experience*, Vol.7(5), Aug. 1995, pp.371-389.

[6] M. Y. Wu, A dedicated track on "Program Partitioning and Scheduling in Parallel and Distributed Systems," in the *Hawaii Int'l Conference on Systems Sciences*, Jan. 1994.

[7] T. Yang and A. Gerasoulis, A dedicated track on "Partitioning and Scheduling for Parallel and Distributed Computation," in the *Hawaii Int'l Conference on Systems Sciences*, Jan. 1995.

[8] T. L. Adam, K. Chandy, and J. Dickson, "A Comparison of List Scheduling for Parallel Processing System," *Communication of the ACM*, Vol.17, No.12, Dec. 1974, pp.685-690.

[9] Gyung-Leen Park, B. Shirazi, and J. Marquis, "DFRN : A New Approach for Duplication Based Scheduling for Distributed Memory Systems," *International Parallel Processing Symposium*, pp.157-166, Geneva, Switzerland, April 1997.

[10] Gyung-Leen Park, B. Shirazi, and J. Marquis, "A Scalable Task Duplication Scheduling for Message Passing Systems," *International Conference on Parallel and Distributed Systems*, pp.122-129, Barcelona, Spain, June 1997.

[11] Gyung-Leen Park, B. Shirazi, and J. Marquis, "Comparative Study of Static Scheduling with Task Duplication for Message Passing Multicomputer Systems," *International Symposium on Solving Irregularly Structured Problems in Parallel*, pp.123-134, Paderborn, Germany, June 1997.

[12] I. Ahmad and Y. K. Kwok, "A New Approach to Scheduling Parallel Program Using Task Duplication," *Proc. of Int'l Conf. on Parallel Processing*, Vol.II, Aug. 1994, pp.47-51.

[13] H. Chen, B. Shirazi, and J. Marquis, "Performance Evaluation of A Novel Scheduling Method : Linear Clustering with Task Duplication," *Proc. of Int'l Conf. on Parallel and Distributed Systems*, Dec. 1993, pp.270-275.

[14] Y. C. Chung and S. Ranka, "Application and Performance Analysis of a Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed-Memory Multiprocessors," *Proc. of Supercomputing'92*, Nov. 1992, pp.512-521.

[15] J. Y. Colin and P. Chretienne, "C.P.M. Scheduling with Small Communication Delays and Task Duplication," *Operations Research*, 1991, pp.680-684.

[16] S. Darbha and D. P. Agrawal, "SDBS : A task duplication based optimal scheduling algorithm," *Proc. of Scalable High Performance Computing Conf.*, May 1994, pp.756-763.

[17] B. Kruatrachue and T. G. Lewis, "Grain Size Determination for parallel processing," *IEEE Software*, Jan. 1988, pp.23-32.

[18] S. Darbha and D. P. Agrawal, "A Fast and Scalable Scheduling Algorithm for Distributed Memory Systems," *Proc. of Symp. On Parallel and Distributed Processing*, Oct. 1995, pp.60-63.

[19] S. J. Kim and J. C. Browne, "A general approach to mapping of parallel computation upon multiprocessor architectures," *Proc. of Int'l Conf. on Parallel Processing*, Vol.III, 1988, pp.1-8.

[20] T. Yang and A. Gerasoulis, "DSC : Scheduling Parallel tasks on an Unbounded Number of Processors," *IEEE Trans. On Parallel and Distributed Systems*, Vol.5, No.9, pp.951-967, Sep. 1994.

[21] Gyung-Leen Park, B. Shirazi, J. Marquis, and Hyunseung Choo, "Decisive Path Scheduling : A

New List Scheduling Method," *International Conference on Parallel Processing*, pp.472-480, Chicago, USA, Aug. 1997.

[22] Y.-K. Kwok, I. Ahmad, and J. Gu, "FAST : A Low-Complexity Algorithm for Efficient Scheduling of DAGs on Parallel Processors," *Proc. of Int'l Conf. on Parallel Processing*, Vol.II, 1996, pp.150-157.

[23] M. Y. Wu and D. D. Gajski, "Hypertool : A Programming Aid for Message-Passing Systems," *IEEE Trans. on Parallel and Distributed Systems*, Vol.1, No.3, Jul. 1990, pp.330-340.

[24] Hyunseung Choo, Hee Yong Youn, Gyung-Leen Park, and Behrooz Shirazi, "Efficient Processor Allocation Scheme for Multi Dimensional Interconnection Networks," *26th International Conference on Parallel Processing*, pp.114-117, Chicago, USA, Aug. 1997

박 경 린

e-mail : glpark@cheju.cheju.ac.kr

1986년 중앙대학교 전자계산학과 졸업(학사)

1988년 중앙대학교 전자계산학과 대학원(공학석사)

1992년 텍사스 주립대(알링턴) 전산공학과 대학원(공학석사)

1997년 텍사스 주립대(알링턴) 전산공학과 대학원 (공학박사)

현재 제주대학교 자연과학대학 전산통계학과 조교수

관심분야 : 분산/병렬 처리 시스템, 오류 허용 시스템, 성능 평가 등

추 현 승

e-mail : choo@yurim.skku.ac.kr

1988년 성균관대학교 이과대학 수학과 졸업(학사)

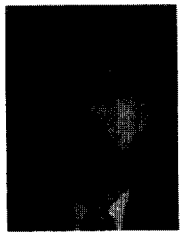
1990년 텍사스 주립대(달라스) 전자계산학과(공학석사)

1996년 텍사스 주립대(알링턴) 전산공학과(공학박사)

1997년~1998년 특허청 심사4국 컴퓨터심사담당관실 심사관

현재 성균관대학교 전기전자 및 컴퓨터공학부 조교수

관심분야 : ATM, 병렬 및 분산 처리, 알고리즘 해석, 고속통신망 등



이 정 훈

e-mail : jhlee@venus1.cheju.ac.kr

1984년~1988년 서울대학교 컴퓨터공학과(학사)

1988년~1990년 서울대학교 컴퓨터공학과(석사)

1990년~1992년 대우통신 전송 연구실

1992년~1996년 서울대학교 컴퓨터공학과(박사)

1996년~1997년 대우통신 광통신연구실(선임)

현재 제주대학교 전산통계학과 조교수

관심분야 : 실시간 통신, 분산 시스템, 멀티미디어 통신