10-1-2003

A "Maximal Tree" Approach For Scheduling Tasks In A Multiprocessor System.

Haiying Sun

A "Maximal Tree" Approach For Scheduling Tasks In A Multiprocessor System

A Thesis -Equivalent Project

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

University of Nebraska at Omaha

by

Haiying Sun

October 2003

PROJECT ACCEPTANCE

Acceptance for the faculty of the Graduate College, University of Nebraska, in partial fulfillment of the requirements for the degree (Master of Science), University of Nebraska at Omaha.

Committee

Name		Department
Hesham Ali	10/3/03	CSCI
Deepak Khazanchi	10/3/03	ISQA
Prithviraj Dasgupta	10/03/03	CSCI

Chairperson [illegible]

Date 10/3/03

TABLE OF CONTENTS

A "Maximal Tree" Approach For Scheduling Tasks In A Multiprocessor System

Haiying Sun, MS

University of Nebraska, 2003

Advisor: Dr. Hesham H.Ali

Abstract:

The problem of scheduling tasks across distributed system has been approved to be NP-complete in its general case. When communication cost among system processors is not considered, polynomial-time optimal algorithms for solving scheduling problem are exit only in three special cases. In attempting to solve the problem in the general case, a number of heuristics have been developed. These algorithms intend to reduce the input task graph to one of the special cases and then optimal scheduling can be obtained accordingly. In this paper, we study all these heuristics, and present a improved heuristic ---"Maximal Tree" graph approach for scheduling general task graph in the parallel system. A package is developed for comparing the proposed heuristic with three other algorithms, List, Maximal chain and Augmentation. A number of experimental studies have been conducted to compare the proposed technique with these known heuristics. Finally, the conclusion of the algorithm is most efficient for a certain kind of task graph was made accordingly.

Keywords: NP-complete, task graph , Multiproccssor Scheduling , maximal tree, maximal chain

1. Introduction

Scheduling is used to efficiently assign time and resources to more than one task simultaneously. It is a classical field with several interesting problems and results. Scheduling problem emerges whenever there is a choice as to the order in which a number of tasks can be performed, and/or in the assignment of tasks to servers for processing [1]. It has tremendous range of applicability. For example, transportation systems such as railroads and bus networks use and rely on schedules, alone with jobs that need to be processed in a manufacturing plant, bank customers waiting to be served by tellers, aircraft waiting for landing clearances, or program tasks to be run on a parallel or distributed computer. Schedules are used to improve efficiency and profitability in addition to prevention of accidents and overloads where applicable.

The problem of scheduling tasks on parallel and distributed processor architecture has been studied several decades. It has been proved that this problem is NP-complete, i.e., in general, no optimal solution can be achieved with the polynomial computation time. When the communication cost among system processors is not considered, optimal algorithm are known for the following few cases: 1) Interval ordered task graph ; 2) Tree-structured task graph 3) Two processor system . In attempting to solve the problem in the general case, a number of heuristics have been developed. These algorithms intend to reduced the input task graph to one of the above special cases and then optimal scheduling can be employed to solve the reduced problem and then a solution to the original problem can be obtained accordingly. The max-chain approach is reducing an m-processor problem to (m-1) processors until we apply the two –processor scheduling algorithm on the remaining tasks. Augmentation approach is adding edges to the partially ordered task graph until it has interval order. Then we could apply interval-order scheduling algorithm on the augmented task graph. All these heuristics do not guarantee an optimal solution to the problem, but they attempt to find near-optimal solutions most of the time.

In this paper, we propose a new scheduling heuristic that generates the "maximum trees" to find a near optimal schedule of a given set of partially ordered tasks. The proposed algorithm is based on the reducing partial of the input task graph to "tree special case ". This approach contains two steps: the first step is finding the "maximal tree" inside the input graph and using Hu's algorithm to schedule the tree. The second step is using the List algorithm to schedule the left tasks into the available time slots and finding the final schedule.

In this paper, we assume that **1)** the input task form a partially ordered set (poset), P = (V,<), where V is a set of tasks. The relation u<v in P implies that the computation of task v depends on the results of task u. The poset P can be represented by a directed acyclic graph (dag), called task graph, which is defined as follows. A dag G = (V,A) represents the poset P if there is a one-to- one correspondence between the vertices in V and the elements of P such that (u,v)$\in$P, or u<v if and only if there is a directed path from u to v in G.[2]

2) The processors are identical and each task can be processed by any processor in the same amount of time(one unit of time).

3) The communication between processors is ignored .

4) A timing diagram called the Gantt chart is used to represent the final schedule. A Gantt chart consists of a list of all processors and, for each processor, a list of all tasks allocated to that processor ordered by their execution time.

5) The tree-structured task graphs in this paper are all in – forest, i.e., each task has at most one immediate successor.

The rest of paper is organized as follows. In order to provide a good background, this thesis will review the optimal scheduling algorithms and related definitions in chapter two .A summaries and analysis of previous research will be presented. This would cover the research, background and theories that have been used to compile this research. These summaries will provide insight into how scheduling is perceived and dealt with when particular situations arise. Once the background information has been presented, the heuristics that will be used in this research will be fully explained in detail. There will be more than three heuristic used to build a comparative study. In addition, the results of every heuristic will be analyzed separately. In chapter 5 , we will present the packages we developed for the comparison study .The last chapter will contain the conclusion. The conclusion summarizes the results and performance measurements .

2. Optimal task scheduling algorithms

As we mentioned above, the problem of scheduling tasks in a distributed system has been approved to be NP-complete in its general case. When communication cost among system processors is not considered, polynomial-time optimal algorithms for solving scheduling problem are exit only in three special cases. These special cases are:

1) When there are two processors available;
2) When the input task graph is a tree;
3) When the input task graph has interval order

In this section we'll introduce three optimal algorithms for solving task scheduling problem in the above three special cases.

2.1 Hu's algorithm for scheduling a tree –structured task graph

This algorithm called the level algorithm which can be used to solve the scheduling problem in linear time when the task graph is either an in-forest, i.e., each task has at most one immediate successor, or an out-forest, i.e., each task has at most one immediate predecessor, and all tasks have the same execution time. In this paper, we assume that the task graph is an in-forest of n tasks as shown in Figure 1.[2]

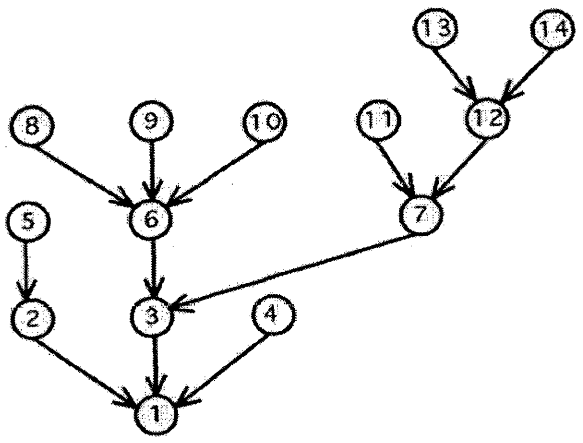


Figure 1: A tree structured task graph

We define task level as follows: Let the level of a node x in a task graph be the maximum number of nodes (including x) on any path from x to a terminal task. In a tree, there is exactly one such path. A terminal task is at level 1. We define ready task as: Let a task be ready when it has no predecessors or when all of its predecessors have already been executed.

Hu's Algorithm works in two steps : First The level of each node in the task graph is calculated as given above and used as each node's priority. Second, whenever a processor becomes available, assign it the unexecuted ready task with the highest priority.

2.2 Coffman/Graham Algorithm for scheduling task graph in a two processor system

This algorithm is proven to achieve an optimal two processor schedule if the input is given as a DAG. This algorithm has a complexity $O(n^2)$.Coffman/Graham's algorithm first derive a linear schedule from the task graph.

And second, this schedule is distributed to the two processors. To get the linear schedule, the graph is traversed bottom-up and all vertices are labeled successively. The order of labeling is such that the vertex to be labeled next has all its successors labeled and those labels from the lexicographically smallest set of all such vertices. After all the vertices have been labeled , move the first ready-task to processor1; if another ready-task is available :then move it to processor 2.

Coffman and Graham algorithm:

a) Assign the number 1 to one of the terminal tasks.

b) Let labels 1,2... j -1 have been assigned. Let S be the set of unassigned tasks with no unlabelled successors. We next select an element of S to be assigned label j. For each node x in S, define l(x) as follows: Let $y_1, y_2, \dots y_k$ be the immediate successors of x. Then l(x) is the decreasing sequence of integers formed by ordering the set $\{L(y_1), L(y_2)...L(y_k)\}$' Let x be an element of S such that $\forall x'$ in S, $l(x) \leq l(x')$ (lexicographically). Define L(x) to be j.

c)When all tasks have been labeled, use the list $(T_n, T_{n-1}.... T_1)$ where for all i, $1 \leq i \leq n$, $L(T_i) = i$ to schedule the tasks.

2.3 Scheduling an interval-ordered task graph

The interval order has a nice structure that is established by the following property. If N(v) denotes the set of successors of task v, then for any interval order P = (V, A) and u, v ε V, either $N(v) \subseteq N(u)$ or $N(u) \subseteq N(v)$ where $N(v) = \{u: (v,u) \varepsilon A\}$. This property implies that for any interval-ordered pair of tasks u and

v, either the successors of u are also successors of v or the successors of v are also successors of u. Since each task corresponds to an interval on the real line, then $R(u) \leq R(v)$ would imply that any successor x of v is a successor of u as well. In this case, $v < x$ implies that $L(x) > R(v) \geq R(u)$, then $u < x$ implies that $v < x$ for any successor x of u. This property of interval-ordered tasks makes it possible to apply a simple greedy algorithm to find an optimal schedule in the case when the execution time of all tasks is the same. At any given time, if there is more than one task ready for execution, picking the task with the maximum number of successors will always lead to the optimal solution, since its set of successors
will include the set of successors of other ready tasks. This idea is reflected in the following algorithm.

1. The number of successors of each node is used as its priority.

2. Whenever a processor becomes available, assign it the unexecuted ready task with the highest priority.

The above algorithm solves the unit execution time scheduling problem for interval order (V, A) in $O(|A| + |V|)$ time.

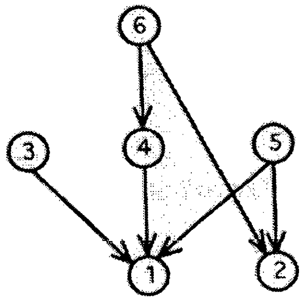


Figure 2: An interval-ordered task graph

3. Survey of previous Heuristics

As shown in the previous section, optimal schedules can be obtained in very restricted special cases. These special cases are by far different from real world situations. For solving the problem in the general case, recent research in this area has emphasized heuristic approaches. A heuristic produces an answer in less than exponential time, but does not guarantee an optimal solution. Therefore "near optimal" means the solutions obtained by a heuristic fall near the optimal solution most of the time. Intuition is most often used to come up with heuristics that make use of special parameters that affect the system in an indirect way. A heuristic is said to be better than another heuristic if solutions fall closer to optimality more often, or if the time taken to obtain a near-optimal solution is less. In this section, we'll study three heuristics, List, Augmentation and Max Chain. The latter two algorithms intend to reduce the input task graph to one of the three special cases and then optimal scheduling can be obtained accordingly.

3.1 List algorithm

List scheduling is a class of scheduling heuristics in which tasks are assigned priorities and placed in a list ordered in decreasing magnitude of priority. Whenever a processor is available, a ready task with the highest priority is selected from the list and assigned to that processor. If more than one task has the same priority, a task is selected arbitrarily.

List Algorithm: [4]

1. Each node in the task graph is assigned a priority. A priority queue is initialized for ready tasks by inserting every task that has no immediate predecessors. Tasks are sorted in decreasing order of task priorities.
2. As long as the priority queue is not empty do the following:
   - i. Obtain a task from the front of the queue.
   - ii. Select an idle processor to run the task.
   - iii. When all the immediate predecessors of a particular task are Executed, that successor is ready to be inserted into the priority queue.

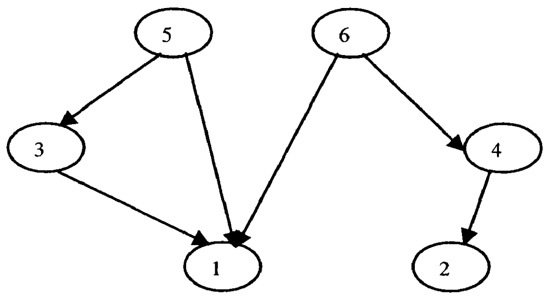


Figure 3: A task Graph

Example:

For the above task graph, we use the number of successors as the priority of each node .And sorted them with the decreasing order of their priority, insert them into priority queue, as shown in follows:

6	5	3	4	1	2

Task 3 and 4 have the same priority, task 2 and 1 have the same priority.

When a processor is available, we obtain a task from the front of the priority queue and assign it into that processor.

Resulting Schedule:

	P1	P2
1	6	5
2	3	4
3	1	2

3.2 Interval Order Augmentation Algorithm

The graph augmentation algorithm based on the fact that given any general directed acyclic graph, an augmented interval-ordered graph, or augmented interval order, exists.The Graph augmentation algorithm first augmented the input graph G into the interval- ordered graph G' .then apply interval ordered graph scheduling algorithm. The algorithm itself consists of *2* different pieces, the augment graph algorithm, and the interval ordered graph algorithm.

Interval Order Augmentation Algorithm: [5]

1. If the input task graph (G) has no interval order, adding edges until we achieve interval ordered task graph (G').
2. The number of successors of each node in G' is used as its priority.
3. Whenever a processor becomes available, assign it the ready task with the highest priority.

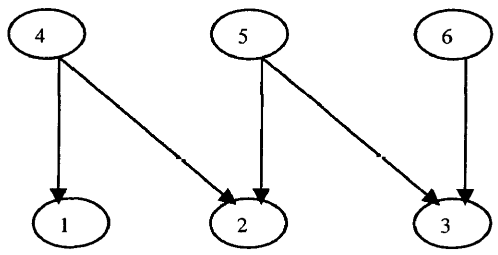


Figure 4: A partial-ordered task graph

Example:

Schedule the task graph as shown in Figure 4.

Step 1: Augment the above graph to achieve interval order

We should add as less edges as possible to achieve interval order, because the more edges we add, the schedule will be less efficient. In this case, we just need to add an edge form task node 4 to task node 3 to achieve interval order.

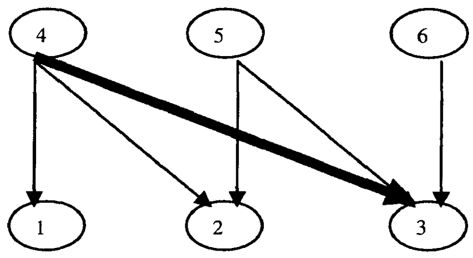


Figure 5 :An interval ordered task graph. (Figure 4 after augmentation.)

Step 2: Sort the nodes according the descending order of their priority. We use the successor number of each node as their priority. Whenever a processor becomes available, assign it the ready task with the highest priority.

The sorted nodes should be as follows:

4	5	6	1	2	3

Task node 1, 2 and 3 has the same priority.

Resulting Schedule:

	P1	P2	P3
1	4	5	6
2	1	2	3

3.3 Maximal Chain algorithm [7]

The Maximal chain algorithm first finds a maximal chain in the given task graph and then takes the remaining tasks and passes it through an (n-1) scheduling algorithm, this is done recursively until we reach the 2-processor scheduling algorithm, which we will solve using the Coffman and Graham algorithm.

The maximal chain of the task graph G is defined as follows: Given a task graph G = (V, A), let S be a subset of tasks in G from the root node to a terminal node in sequence; then we say that S is a maximal chain in G if there does not exist another chain S' in G such that S' has a higher number of tasks than S.

The maximal chain scheduling heuristic consists of three main sub-algorithms, which are as follows:

1) The maximal chain

2) The 2-processor Coffman-Graham algorithm

3) The Merge routine, which not only merges the maximal chain and the (n-l) processor schedule, but also maintains the feasibility of the schedule based on the task graph precedence of the tasks and optimizes the schedule wherever possible.

Algorithm:

1. Use Coffman-Graham algorithm to assign priority to each task node.

2. Find the maximal chain. Let Maximal Chain be M = {null}, Pick the task T_i with the highest label. (This will be a task which will have no predecessors.). The maximal chain $M = M \cup \{ T_i \}$. From the list of successors tasks S' of this task T_i find the task with the next highest label . Let this be task T_j .With this task T_j repeat from step 2 until we have a task, which has no successors.

3. Let M be the maximal chain and S be the (n-1) processor schedule. Assign the task in the maximal chain to processor P_1, Repeat step 2, keep finding maximal chain and assign the whole chain to the available processor until there are only 2 processor left.

4. Using Coffman-Graham algorithm, schedule the rest of tasks to the remaining two processors.

5. Merge the schedules. In this step, we need to examine every task if it violates any of the precedence rules, remove all the violations and idle time slots.

Example:

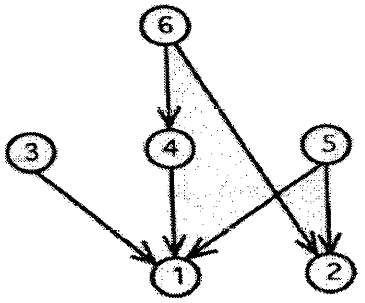
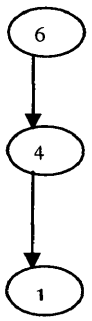


Figure 6: A task graph

We suppose to schedule the above task graph into 3 processor.

1) Assign label: just as shown in the above figure, according Coffman-Graham algorithm assign priorities to all the nodes. In this graph, the priorities of each node are same as their ID: node1 labeled to be 1; node 2 labeled to be 2....node 6 labeled to be 6.
2) Find the maximum chain and schedule it into processor 1:

   The maximum chain of the above Figure is 6 → 4 → 1



	P1	*P2*	*P3*
1	*6*		
2	*4*		
3	*1*		

3) Use Coffman-Graham algorithm schedule the rest of tasks to the left two processors

	P1	*P2*	*P3*
1		*5*	*3*
2		*2*	
3			

4) Merge the schedules; remove any violations of precedence rules and idle time slots.

Resulting Schedule:

	P1	*P2*	*P3*
1	*6*	*5*	*3*
2	*4*	*2*	
3	*1*		

4. Proposed Solution

In this section, we will study the proposed "Maximal Tree" scheduling heuristic. We could also name this new algorithm to be "splitting graph scheduling heuristic", because in this proposed solution, we first divide the input task graph in to a tree-like graph and a common graph. Then we could apply either Hu's algorithm or the other List algorithms to get the schedule of the tasks in the tree-like graph. For the remaining tasks, we will use list algorithms to assign them into the available time slots and get the final schedule.

The motivation for this heuristic came from the fact that we already have well known polynomial algorithms for special cases. We felt that it was a matter of degenerating or reducing the problem to the special cases. We also took advantage of the maximal chain algorithm, we notice that in the maximal chain algorithm, we have to move precedence violations in the final step. We proposed that instead of finding the maximum chain, we finding the maximal tree inside the graph. The maximal tree includes all the nodes in the maximal chain and all their predecessors in the graph. We schedule all the tasks on the tree into available processors according Hu's algorithm. After that it was just going to be a matter of merging the "maximal tree" schedule with the schedule for the remaining tasks. We use the List algorithm to schedule the remaining tasks into the Tree's schedule. We check every idle time slot in the schedule of the maximal tree, and assign the ready task in the remaining graph to it. The maximal tree algorithm should yield a better solution than maximal chain and list algorithm. The schedule length has to be at least the length of the maximal chain. The same thing is for the "maximal tree", the schedule length should has to be at least the length of the schedule length of the "maximal tree ", because the "maximal tree" contains the maximal chain and all the predecessors of the nodes in the maximal chain. Implementing and performing our proposed maximal tree heuristic on it should yield us same or better schedule than maximal chain heuristic since we take care all the predecessors of the nodes in the maximal chain in the first assignment, so we do not need to remove any violations in the final merging process. Using maximal tree technique, we'll force the schedule length as shorter as possible.

4.1 Maximal Chain and Maximal Tree

We define maximal chain and maximal tree as follows:

Maximal Chain: Given a task graph G = (V, A), let S be a subset of tasks in G from the root node to a terminal node in sequence; then we say that S is a maximal chain in G if there does not exist another chain S' in G such that S' has a higher number of tasks than S.

Maximal tree: Given a task graph G = (V, A), let S be a subset of tasks in G from the root node to a terminal node in sequence; then we say that S is a maximal tree in G if S contains all the nodes in the maximum chain and all their precedence in G. We search the predecessors from the top node of the maximal chain, and count each predecessor only once.

Example 1:

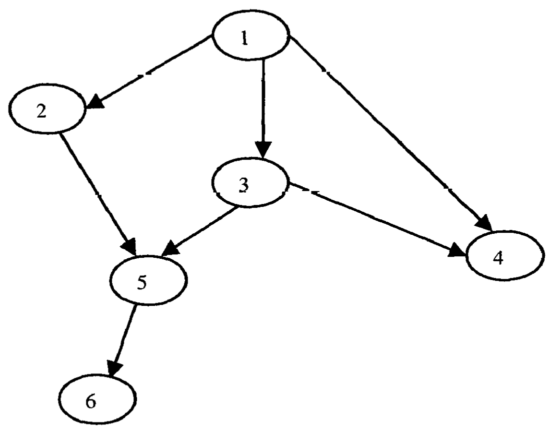


Figure 6 A task graph

The maximal chain of the above figure is: 1→3→5→6

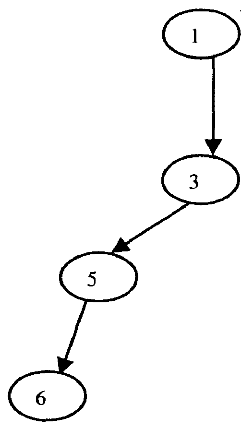


Figure 7: Maximal chain of Figure 6.

The maximal tree of Figure 6 is: **1→3→2→5→6**

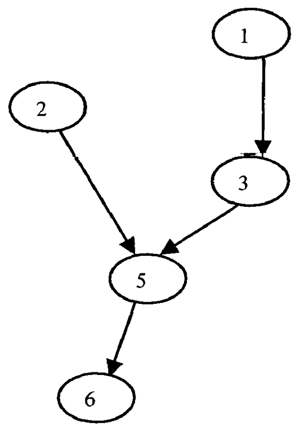


Figure 8: Maximal Tree of figure 6.

Example 2:

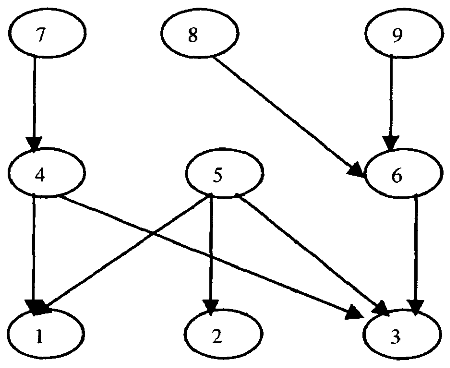


Figure 9: A task graph

The max chain of the above figure is: 9→6→3

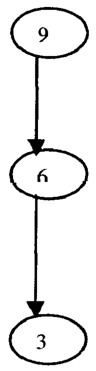


Figure 10: the maximal chain of Figure 9

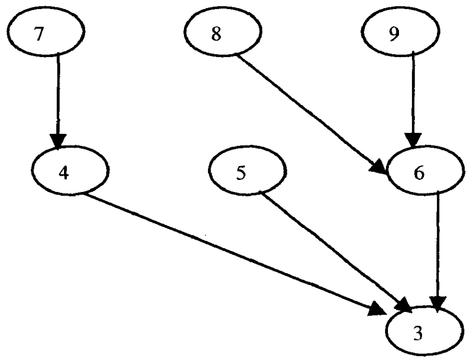


Figure 11: the maximal tree of Figure 9

We search maximal tree in the graph G, using the following approach:

Lets M is the maximal chain of the G, and MT is the maximal tree of the G. We first lets MT={M} , From the top node of the maximal chain M , we search for its all predecessors P(x) in graph G , if P(x) is not found in MT , then we put P(x) in the Maximal tree: MT={MT} ∪ P(x)

In the process for finding maximal tree, we follow the "Start from the highest label of the maximal tree, and each predecessor only count once" rule.

For example:

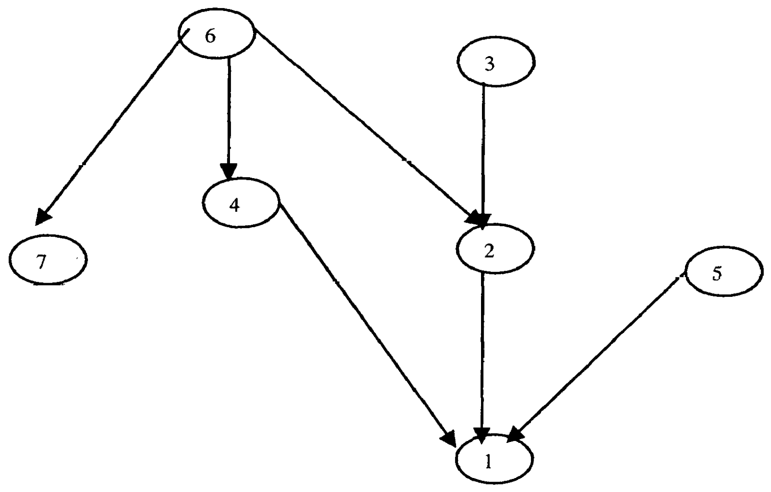


Figure 11: A task graph

Figure 11 shows a task graph and Node 6 is the predecessor of both node 2 and node 1. We suppose the maximal chain M of Figure 11 is 3→2→1 , First of all, we put all the nodes in the maximal chain into the maximal tree → MT={3,2,1}, then we search from the top of the maximal chain , which is node 3 , node 3 does not have any predecessors, so we continue to search the next node in the maximal chain, which is node 2. Node 2 has two predecessors , node 6 and node 3 , because node 6 is not in M, , so we include node 6 in to MT ={3,2,1,6}. Then we continue to search the next node in the M , which is node 1 , Node 1 has five predecessors : node 4 , node 6 , node 5 , node 3 and node 2 , because node 6 , node 3 and node 2 are already in the maximal tree, we will not count them again , at this point , we just need to put node 5 and node 4 into the Maximal tree. MT = {3, 2, 1, 6, 5, 4}. So the maximal tree of Figure 11 is as follows:

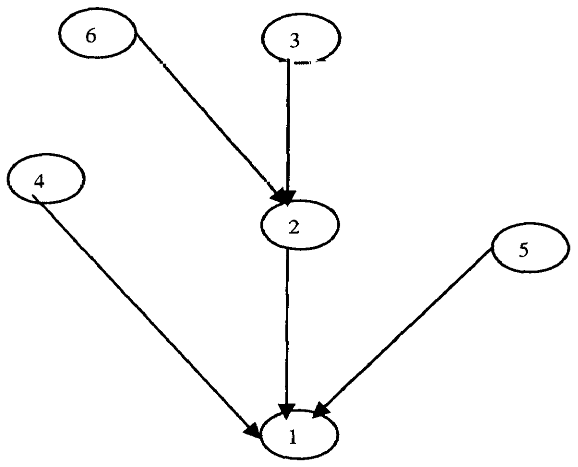


Figure 12: The Maximal Tree of the Graph in Figure 11.

According Hu's algorithm, the priority of a node x in a task graph is the maximum number of nodes (including x) on any path from x to a terminal task. In a tree, there is exactly one such path. In the above example, we ignore the edge between node 6 and 4, because we search the precedence from the highest node of maximal chain and count the predecessors only once, node 6 will always has the higher priority than the node 4, so even we did not count the edge 6→4, we will not worry that it will cause any violations of the precedence rule with Hu's algorithm. But to make sure that all the precedence rules are satisfied, after we sorted the nodes with their priority, we will go over the ordered nodes in the "Maximal Tree", check if there are any violations of precedence rule with their original predecessors. In case there exists any violations, we will remove them first, before we start to schedule them.

4.2 "Maximal Tree "Algorithm:

The maximal tree scheduling heuristic that we propose consists of three main sub-algorithms, which are as follows:

1) The maximal tree

2) Hu's algorithm

3) The List algorithm

Algorithm 4.2.1 for finding maximal tree

1. Assign the number 1 to one of the terminal tasks.

2. Let labels 1, 2. ..J -1 has been assigned. Let S be the set of unassigned tasks with no unlabelled successors. We next select an element of S to be assigned label j. For each node x in S, define l(x) as follows: Let y_1, y_2....y_k be the immediate successors of x. Then l(x) is the decreasing sequence of integers formed by ordering the set {L (y_1), L (y_2)...L (y_k). Let x be an element of S such that V x' in S, l(x) $\leq$ l(x') (lexicographically). Define L(x) to be j.

3. Let Maximal Chain be M = {null},

4. Pick the task T_i with the highest label. (This will be a task which will have no predecessors.). The maximal chain M = M $\cup$ { T_i }.

5. From the list of successors tasks S' of this task $T_{i,}$ find the task with the next highest label. Let this be task T_j .With this task T_j repeat from step 2 until we have a task, which has no successors.

6. Let Maximal Tree be MT = {null},

7. Find the Maximal chain M and MT = {M}

8. From the highest label task $x \in M$, find all the predecessors of x, P(x) in the graph G, MT = {M} $\bigcup$ P(x).

Once we find the "maximal tree" inside a graph, we could apply algorithm 4.2.2 to schedule the "maximal tree" into the processors.

Algorithm 4.2.2 for scheduling the "maximal tree".

1. Let the level of a node x in a task graph be the maximum number of nodes (including x) on any path from x to a terminal task. A terminal task is at level 1. Sorted all the nodes in MT according its task level in the descending order.

2. Go over the order of nodes; remove the violations of the precedence rule.

2. Whenever a processor becomes available, assign it the unexecuted ready task with the highest priority

After we get the schedule for the "maximal tree", then the following job is using algorithm 4.2.3 to assign the remaining tasks into the schedule.

Algorithm 4.2.3 for merging the remaining tasks.

1. A priority queue is initialized for ready tasks by inserting every task that has no immediate predecessors. Tasks are sorted in decreasing order of task priorities.

2. As long as the priority queue is not empty do the following:

a. Obtain a task from the front of the queue.

b. Select an idle time slot to run the task.

c. When all the immediate predecessors of a particular task are executed, that successor is ready to be inserted into the priority queue.

Example 1:

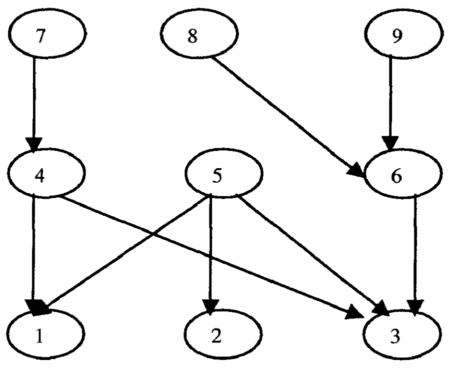
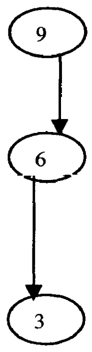
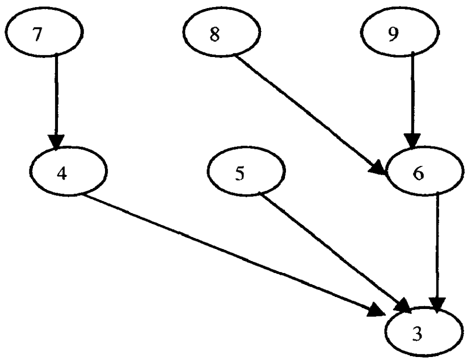


Figure 9: A task graph

1) For the above task graph, we first find the maximal chain which is as follows:



2) Then according maximal chain, we find the maximal tree, which as follows:



3) Then we use algorithm 4.2.2 to schedule the above tree into the processors, we assume we have 3 processors here:

The sorted nodes according algorithm 4.2.2

9	8	7	6	5	4	3

We go over the above order, to make sure there are not any precedence violations according their original predecessors .There are no violations in this case. Then we schedule the "maximal tree" as follows:

	P1	*P2*	*P3*
1	*9*	*8*	*7*
2	*6*	*5*	*4*
3	*3*		

4) There are two nodes left which as follows:

In the final step, we use the list algorithm to merge the left nodes into the available time slots:

Resulting schedule:

	P1	*P2*	*P3*
1	*9*	*8*	*7*
2	*6*	*5*	*4*
3	*3*	*2*	*1*

Example 2:

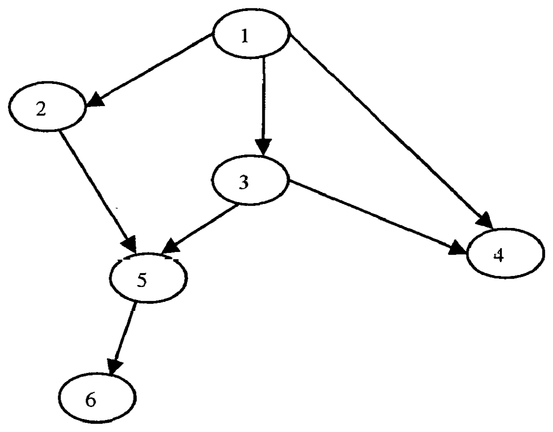
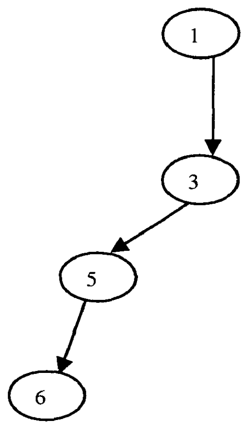


Figure 6.

Step 1: To schedule the above graph, we need first find its maximal chain which is as follows:

The maximal chain of the above figure is: 1→3→5→6



Then we find the max tree which as follows:

The maximal tree of Figure 6 is: **1→3→2→5→6**

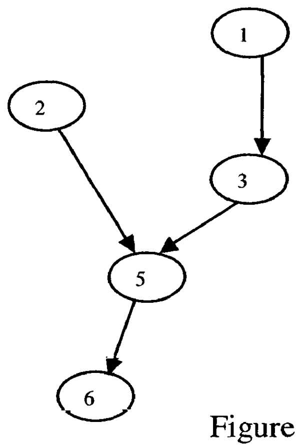


Figure 8. Maximal Tree of figure 6.

Step 2: We use the above algorithm 4.2.2 to schedule the above tree:

	P1	*P2*	*P3*
1	*1*		
2	*3*	*2*	
3	*5*		
4	*6*		

Step 3: We merge the left node 4 into the available time slot according algorithm 4.2.3

Resulting schedule:

	P1	P2	P3
1	1		
2	3	2	
3	5	4	
4	6		

The schedule length has to be at least the length of the maximal tree, so applying maximal tree heuristic should yield us better schedule than other heuristic. Using "maximal tree technique", we force the schedule length as shorter as possible.

5. Package

The project is implemented as a C++ program. C++ was chosen as the implementation language since is an object-oriented language which is platform neutral with widely available free implementations. It also has rich API that provides core data structures that are useful. This program is run on Microsoft Visual C++, which has rich compiling tools and easy to use.

Several programs been implemented as part of this paper. These programs will enable the user to create a task graph and then run the n-processor scheduling algorithm on it. It can compare the results with other algorithms (List, Maximal chain, Augmentation).

This package includes the five parts:

5.1 Random DAG Generator:

To test the efficiency of above algorithms, a set of directed acyclic graphs is required

Since the "randomness" of the graphs is usually not well-defined concept, we here use the mathematical algorithms that mimic randomness. Microsoft Visual C++ provides the standard class library which has random functions that we could take advantage of.

Random DAG Generator allows us randomly generate a task graph by:

Input: a) Number of nodes in the graph, and

b) The Density/ Sparseness of the graph

The density of graph is varies from 0.1 to 0.9

Output: A random generated Task Graph with the processor Number. (Text File) It must be noted that the graphs generated had transitive precedence edges, which implies if $A \rightarrow B$ and $B \rightarrow C$, then even though by transitivity it implies that $A \rightarrow C$, the random graph generator, does not take this into account and may possibly create such transitive edges. This increase the density of the graphs generated. We define the task nodes as an integer number in this package and define the edge as two nodes; the edge direction is from the first node to the second node.

The example of the random generated task graph (text file):

```
5
5
1 2
3 4
2 4
1 4
2 3
```

The first 5 is the nodes number, the second 5 is the edge number, in this package we use an integer to represent node, and two integer numbers represent the edge. The edge direction is from the first node to the second one.

The above text file represents the following task graph

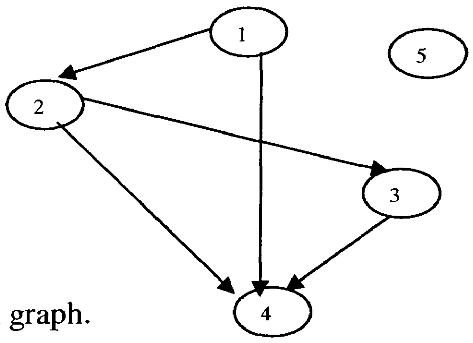


Figure 13:

A random generated task graph.

5.2 List Algorithm Package

This program read in the task graph file and the processor number , apply List algorithm to schedule the task graph and output the schedule.

Input: a) A Task Graph File (text file)

b) Processor Number

Output: The schedule

In the List package , we have the following program files:

Vertex .h : It is the defination of the nodes in the task graph. We define the vertex consist of node Id , which represented by an interger and the labels which are used to represent its priority or if it has be scheduled in the graph computation. We reload all the mathematical operators in this program .

Edge.h: It is the defination of the edge in the task graph . We define the edge as two nodes , the direction is from the first node to the second one . We also reload all the mathematical operators in this program .

Task.h : It is the defination of task graph object , the task object is built by nodes and edges .

ListSchedue.cpp: This class is the core program of this package. It reads into a text file , and according the input data ,set nodes and edges and build the task graph.

Then it will perform the related graph computation , for example , search for the successors or the precedessors ,assign priorities ect. Finally it will implement the the list heuristic and output theschedule. The two most important functions in this class are :

getPriority(...) which is used to assign the priorities to each task nodes

ListSchedule(...) which is used to implement List schedule heuristic .

We use Queue data structure to manipulate the ready tasks. We also use Stack here for assisting graph computaion.

Other functions in this class are all related to the graph computation.

5.3 Maximal Chain Algorithm Package

This program read in the task graph file and the processor number , apply Maximal Chain algorithm to schedule the task graph and output the schedule.

Input: a) A Task Graph File

b) Processor Number

Output: The schedule

In the Maximal Chain package , we have the following header files for the defining the task graph.

Vertex .h , **Edge.h , Task.h** , which are same as in the List package

MaxChainFinal.cpp: This class is the core program of this package. It reads into a text file , and according the input data ,set nodes and edges and build the task graph.Then it will perform the related graph computation .Finally it will implement the the Maximal Chain heuristic and output the schedule.

The main functions in this class are :

AssignLabels(...) which use Coffman-Graham algorithm to assign priority to each task node.

FindMaximalChain(...) which is used to find maximal chain in the task graph according thc algorithm mentioned in the previous section.

MaxChainSchedule(….) which is used to implement maximal chain schedule heuristic and output the final schedule.

getSchedule(…) which is used to implement 2-processor 's Coffman-Graham algorithm.

adjustSchedule (…..) which is used to remove the idle time slots.

We use Queue and Stack to manipulating the related graph computaion.

5.4 Maximum Tree Algorithm Package

This program read in the task graph file and the processor number , apply Maximal Chain algorithm to schedule the task graph and output the schedule.

Input: a) A Task Graph File

b) Processor Number

Output: The schedule

In the Maximal Tree package , we have the following header files for the defining the task graph.

Vertex .h , **Edge.h , Task.h** , which are same as in the List package

The core program in this package is :

MaxTreeSchedule.cpp: This class is the core program of this package. It reads into a text file , and according the input data ,set nodes and edges and build the task graph. Then it will perform the related graph computation , for example , search for the successors or the precedessors ,assign priorities ect. Finally it will implement the the the proposed heuristic and output the schedule. The most important functions in this class are :

FindMaximalTree(…) which used the algorithm 4.2.2 to find the maximal tree in the task graph.

getPriority(….) which is used to assign the priority to each task nodes according Hu's algorithm.

MaxTreeSchedule (….) which is used to implement maximal tree schedule heuristic and output the final schedule.

getReadyTask(..) which is used to manipulate the ready taska.

We use Queue and Stack to manipulate tasks in the scheduling process and assist the related graph computaion.

5.5 Interval order augmentation Algorithm Package

This program read in the task graph file and the processor number , apply Interval Order Augmentation Schedule algorithm to schedule the task graph and output the schedule.

Input: a) A Task Graph File

b) Processor Number

Output: The schedule

In this package , we defined the Vertex, Edge and Task which is same as in the previous package. The core program in this package is :

AugmentF1.cpp: This class is the core program of this package. It first read into a text file , and according the input data , build the task graph and then implent Augmentation schedule heuristic.

The main functions in this class are :

CheckInterOrder(...) which used to check if the input task has interval order.

Augment(....) which is used to augment input task graph into the inteval ordered graph .

Schedule (....) which is used to implement interval-ordered task graph optimal algorithm to output schedule .

We use Queue and Stack data structures here to manipulate tasks in the scheduling process and assist the related graph computaion.

5.6 Program Specifications :

The program specifications are as follows:

1) The program is written under the Windows operating system (Windows 2000)

2) The program makes use of the Microsoft Foundation class library.

3) The program is written using MS Visual Studio VC++ 6.0.

4) The program was written on an Intel ® Pentium, 4 CPU 1400 MHz machine, which had 5 GB of hard disk space and 261,424 KB of RAM.

6. Implementation Results and Comparative Studies:

We ran various simulations on the different scheduling heuristics using different graphs. The two most important properties of the graphs that the algorithms were tested against were:

a) Number of nodes in the graph, and

b) The Density/ Sparseness of the graph

The density of graph is varies from 0.1 to 0.9 , it implies that the graph having 0.1 sparseness would have fewer edges and hence less density and as the density increases the number of edges increase and so the sparseness of the graph reduces. It also implies that the graph with lower density will most likely take less scheduling time as compared with a graph of higher density.

We ran different simulations as explained below:

1) Simulation-1 was run on graphs with 10, 20, 25, 30, 35,40,50 nodes with density varying from 0.1 to 0.8. Also the percentage of augmented edges is in between 0.1 to 0.45.

The naming conversion used for the various graphs in this simulation is as follows Dag xxx-y-z, where xxx denotes the number of nodes in the graph ; Y denotes the processor number , which is three in this Simulation; and z denotes the density of the graph.

2) Simulation-2 was run on 11 typical graphs from "Task Scheduling in Parallel and Distributed System ".This simulation used to comparing the efficiency of the four Heuristics.

The naming conversion used for the various graphs in this simulation is as follows Figure p xxx, where xxx denotes the page number of the task graph form "Task Scheduling in Parallel and Distributed System ". For example: Figure P12 means the input graph is from page number 12 of the book.

3) Simulation-3 was run on graphs with nodes 40, 50 60 and 70. with the density varying from 0.1 to 0.9 Processor number is 3, 4 and 5.

The naming conversion used for the various graphs in this Simulation is as follows Dag xxx-y-z, where xxx denotes the number of nodes in the graph,
Y denotes the processor number, and z denotes the density of the graph.
For example: dag50-5-9 means the input task graph has 50 nodes, density is 90% and run on 5 processors. This experiment is also used to comparing the efficiency of the four heuristics.

4) Simulation-4 was run on graphs with nodes 15. with the density with task graphs with nodes number 15, density varying from 0.04 to 0.14 and processor number is 3. The percentage of the maximal tree is varying from 13% to 100%.

The naming conversion used for the various graphs in this simulation is as follows Dag xxx-y-z, where xxx denotes the number of nodes in the graph,
Y denotes the density, and z denotes the number of nodes on the tree.
For example: dag15-.26-9 means the input task graph has 15 nodes, density is 26% and the number of nodes on the tree are 9.
It must be noted that the graphs generated had transitive precedence edges, which implies if $A \rightarrow B$ and $B \rightarrow C$, then even though by transitivity it implies that

$A \rightarrow C$, the random graph generator, does not take this into account and may possibly create such transitive edges. This increase the density of the graphs generated. The graphs used in the simulations are not transitively reduced graph. We would also like to define the density of graphs used in the simulations as ratio of the number of edges $|E|$ in the graph as a percentage to the maximal number of edges that the graph can have. In our case, the maximal number of edges is $(n\times(n-1) \div 2)$ (because we do not consider nodes having edges on to themselves), where n is the number of nodes in the graph. The graphs that we generated have more density because these graphs are not transitively reduced.

6.1 Simulation 1

Goal:

Clarify the relationship in between the number of adding edges and the length of the output schedule.

Input: 1) random generated Dag file with density 10% to 80%

2) Augment percentage

Output: Time of the schedule

Result: Table 1:

Input File	**Time**	**Augment percentage**
Dag35-3-1	14	10%
Dag35-3-1	17	20%
Dag35-3-1	21	30%
Dag35-3-1	28	40%
Dag50-3-1	22	10%
Dag50-3-1	24	15%
Dag50-3-1	26	20%
Dag50-3-1	29	25%
Dag50-3-1	33	30%
Dag50-3-1	36	35%
Dag50-3-1	40	40%
Dag50-3-1	49	45%
Dag10-3-2	6	10%
Dag10-3-2	7	20%
Dag10-3-2	7	30%
Dag10-3-2	10	40%

Table II:

Input file	**Time**	**Augment percentage**
Dag20-3-3	11	10%
Dag20-3-3	13	20%
Dag20-3-3	17	30%
Dag20-3-3	20	35%
Dag30-3-4	20	10%
Dag30-3-4	22	15%
Dag30-3-4	24	20%
Dag30-3-4	26	25%
Dag30-3-4	30	30%
Dag25-3-7	22	5%
Dag25-3-7	22	8%
Dag25-3-7	23	10%
Dag25-3-7	25	15%
Dag25-3-8	24	5%
Dag25-3-8	25	10%
Dag40-3-5	27	10%
Dag40-3-5	30	15%
Dag40-3-5	35	20%
Dag40-3-5	40	25%

Conclusion:

From the above simulation result, It can be concluded that the more edges we added the longer length of the output schedule.

6.2 Simulation 2

Goal:

Comparing the efficiency of Augmentation, List, Max Chain and Max Tree heuristics with 11 typical task graphs in the "Task Scheduling in Parallel and Distributed System".

Input: 1) 11 typical task graphs

2) Processor number is 3

Output: Gatt Chart

Implement Result:

Simulation- 2: Simulations with 11 typical task graphs, processor number is 3

Algorithms	Augment Algorithm	List Algorithm	Max Chain Algorithm	Max Tree Algorithm
Figure p12 Time	3	3	3	3
Figure P19 Time	6	6	7	6
Figurc P21 Time	3	3	3	3
Figure P24 Time	8	7	7	7
Figure P42 Time	3	3	3	3
Figure P46g Time	4	5	4	4
Figure P46h Time	5	5	5	5
Figure P58 Time	6	6	6	6
Figure P70 Time	5	5	5	5
Figure P76 Time	4	4	4	4
Figure P83 Time	5	5	5	5

Conclusions from Simulation 2:

From above simulation results we can draw the following conclusions:

1) It can be concluded that maximal tree algorithm always outputs the best schedule.

2) All these four heuristics have the same performance in Figure p12, p21, p46h, P42, p58, p70, p76 and Figure p83. For the figure p19, List and Max Tree and Augmentation heuristics work better than Maximal Chain heuristic. In Figure P46g List has the worst performance.

3) In figure 24, Augmentation algorithm has the worst performance.

6.3 Simulation 3

Goal:

Comparing the efficiency of augmentation, List, Max Chain and Max Tree heuristics with task graphs with nodes number 40,50,60,70, density varying from 0.1 to 0.9, and processor number is 3, 4 and 5.

Input: 1) Task graphs

2) Processor number

Output: Schedule time

Result of Simulation 3

Simulations of graph (node number 40-70, density 0.1—0.9, Processor number 3)

Algorithms	Augmentation	List	Max Chain	Max Tree
Dag40-3-1	15	18	15	15
Dag40-3-2	19	21	18	18
Dag40-3-3	22	22	21	20
Dag40-3-4	25	25	24	24
Dag40-3-5	27	27	26	26
Dag40-3-6	33	33	33	33
Dag40-3-7	33	33	33	33
Dag40-3-8	36	36	36	36
Dag40-3-9	38	38	38	38
Dag50-3-1	20	22	19	18
Dag50-3-2	25	25	23	22
Dag50-3-3	28	27	26	25
Dag50-3-4	28	29	28	28
Dag50-3-5	33	33	34	33
Dag50-3-6	37	38	37	37
Dag50-3-7	40	40	40	40
Dag50-3-8	42	42	42	42
Dag50-3-9	48	48	48	48
Dag60-3-1	23	25	22	21
Dag60-3-2	29	32	27	27
Dag60-3-3	29	30	26	26
Dag60-3-4	38	38	37	37
Dag60-3-5	40	40	40	40
Dag60-3-6	44	44	44	44
Dag60-3-7	53	53	53	53
Dag60-3-8	53	53	53	53
Dag60-3-9	57	57	57	57
Dag70-3-1	30	27	28	26
Dag70-3-2	36	37	34	34
Dag70-3-3	39	39	36	36
Dag70-3-4	42	42	42	40
Dag70-3-5	43	44	43	43
Dag70-3-6	50	51	49	49
Dag70-3-7	60	60	60	60
Dag70-3-8	64	64	64	64
Dag70-3-9	68	68	68	68

Simulations of graph (node number 40-70, density 0.1—0.9, Processor number 4)

Algorithms	Augmentation	List	Max Chain	Max Tree
Dag40-4-1	15	16	12	12
Dag40-4-2	19	19	18	18
Dag40-4-3	20	21	20	20
Dag40-4-4	25	25	24	24
Dag40-4-5	27	27	26	26
Dag40-4-6	33	33	33	33
Dag40-4-7	33	33	33	33
Dag40-4-8	36	36	36	36
Dag40-4-9	38	38	38	38
Dag50-4-1	16	20	16	15
Dag50-4-2	24	24	21	21
Dag50-4-3	27	26	25	25
Dag50-4-4	28	28	28	28
Dag50-4-5	33	33	33	33
Dag50-4-6	37	38	37	37
Dag50-4-7	40	40	40	40
Dag50-4-8	42	42	42	42
Dag50-4-9	48	48	48	48
Dag60-4-1	19	22	19	17
Dag60-4-2	28	30	26	25
Dag60-4-3	26	26	25	25
Dag60-4-4	38	38	37	37
Dag60-4-5	40	40	40	40
Dag60-4-6	44	44	44	44
Dag60-4-7	53	53	53	53
Dag60-4-8	53	53	53	53
Dag60-4-9	57	57	57	57
Dag70-4-1	25	25	25	22
Dag70-4-2	33	35	31	30
Dag70-4-3	35	36	35	35
Dag70-4-4	41	41	40	40
Dag70-4-5	43	44	42	42
Dag70-4-6	50	50	49	49
Dag70-4-7	60	60	60	60
Dag70-4-8	64	64	64	64
Dag70-4-9	68	68	68	68

Simulations of graph (node number 40-70, density 0.1—0.9, Processor number 5)

Algorithms	Augmentation	List	Max Chain	Max Tree
Dag40-5-1	13	15	11	11
Dag40-5-2	19	19	18	18
Dag40-5-3	20	20	20	20
Dag40-5-4	25	25	24	24
Dag40-5-5	27	27	26	26
Dag40-5-6	33	33	33	33
Dag40-5-7	33	33	33	33
Dag40-5-8	36	36	36	36
Dag40-5-9	38	38	38	38
Dag50-5-1	15	18	15	14
Dag50-5-2	22	23	20	20
Dag50-5-3	27	26	25	25
Dag50-5-4	28	28	28	28
Dag50-5-5	33	33	33	33
Dag50-5-6	37	38	37	37
Dag50-5-7	40	40	40	40
Dag50-5-8	42	42	42	42
Dag50-5-9	48	48	48	48
Dag60-5-1	18	20	16	15
Dag60-5-2	27	30	25	25
Dag60-5-3	26	26	25	25
Dag60-5-4	38	38	37	37
Dag60-5-5	40	40	40	40
Dag60-5-6	44	44	44	44
Dag60-5-7	53	53	53	53
Dag60-5-8	53	53	53	53
Dag60-5-9	57	57	57	57
Dag70-5-1	23	23	23	21
Dag70-5-2	32	34	31	30
Dag70-5-3	35	36	35	35
Dag70-5-4	41	41	40	40
Dag70-5-5	43	43	42	42
Dag70-5-6	50	50	49	49
Dag70-5-7	60	60	60	60
Dag70-5-8	64	64	64	64
Dag70-5-9	68	68	68	68

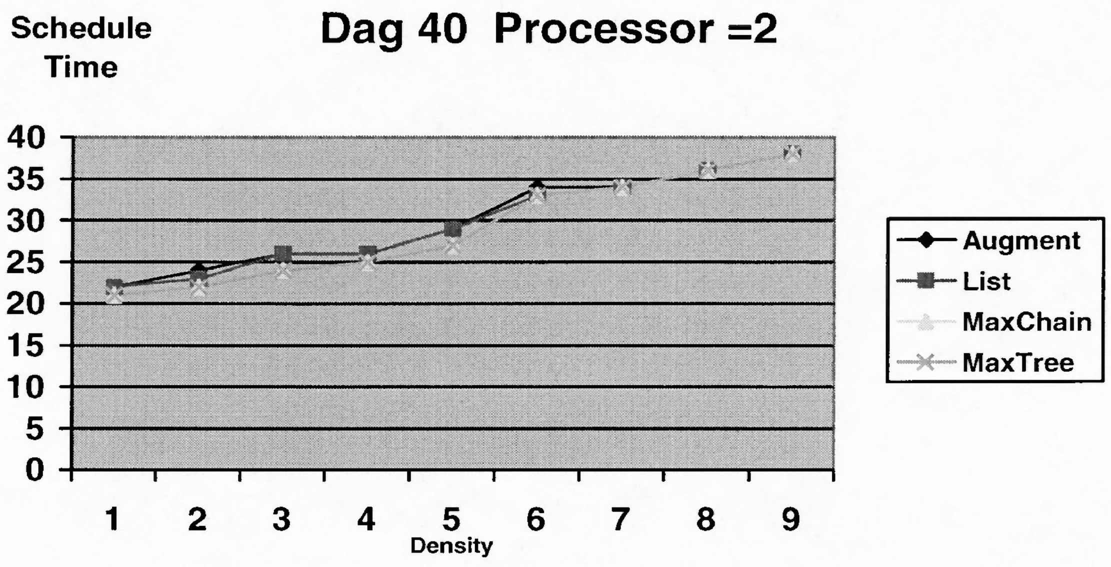
Conclusions from Simulation 3:

From above simulation results we can draw the following conclusions:

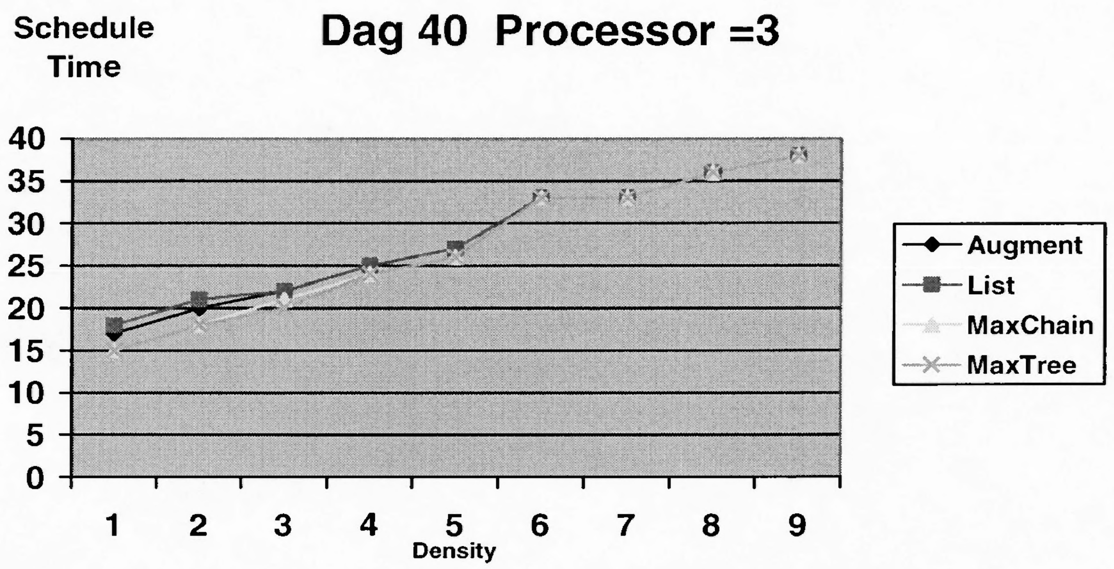
1) Maximal Tree Heuristic always works better than three other heuristics.
2) Maximal Chain Heuristic works better than List and Augmentation heuristics. It yields the best schedule in 85.1% cases. In 99% cases, it works better than List heuristic, and in 100% cases, it works better than Augmentation heuristic in the above experiment.
3) Augmentation Heuristic works better than List heuristic. It yields the best schedule in 58.3% cases. In 96.3% cases, it outputs the better schedule than List Heuristic.
4) List heuristic has the worst performance among all these four algorithms. Only in 33.3% cases, it outputs the most efficient schedule. It works better than Augmentation Heuristic only in 3.7% cases, and in less than 1% cases, it outputs the better schedule than Maximal Chain Heuristic.
5) When the density of the graph is in between 0.7 and 0.9, all these four algorithms have the same performance.
6) With the increasing of the density of the input task graph, all the algorithms have the more similar performance.
7) With the increasing of processor number, when the input density is less than 70%, all the algorithms have the shorter schedule length in most cases. But when the density is above 70%, the schedule length will not be affected.
8) With the increasing of processor number, all the algorithms have the more

similar performance.

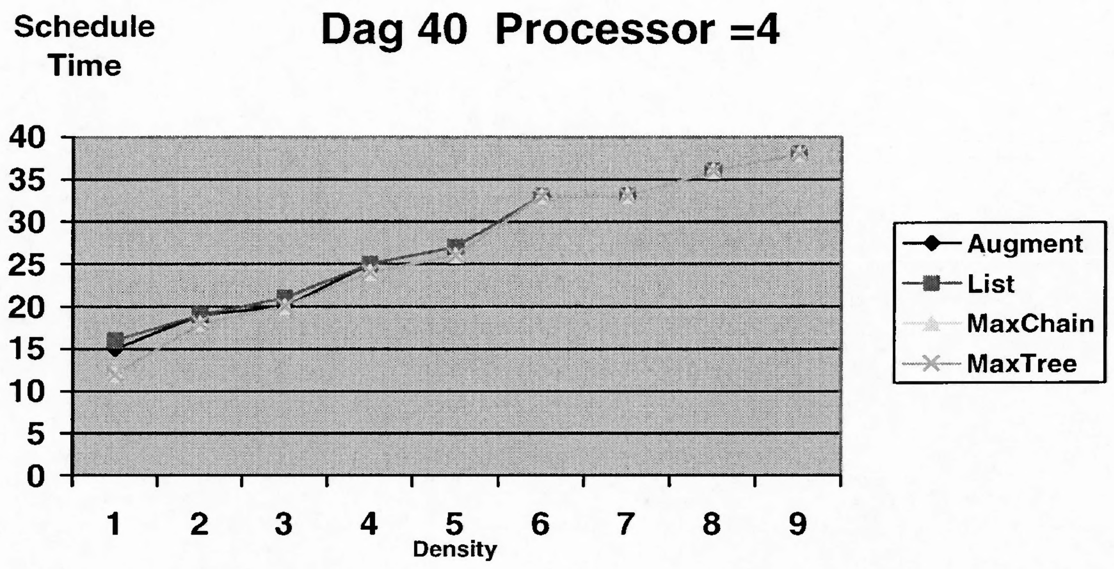
6.4 Comparative Studies:



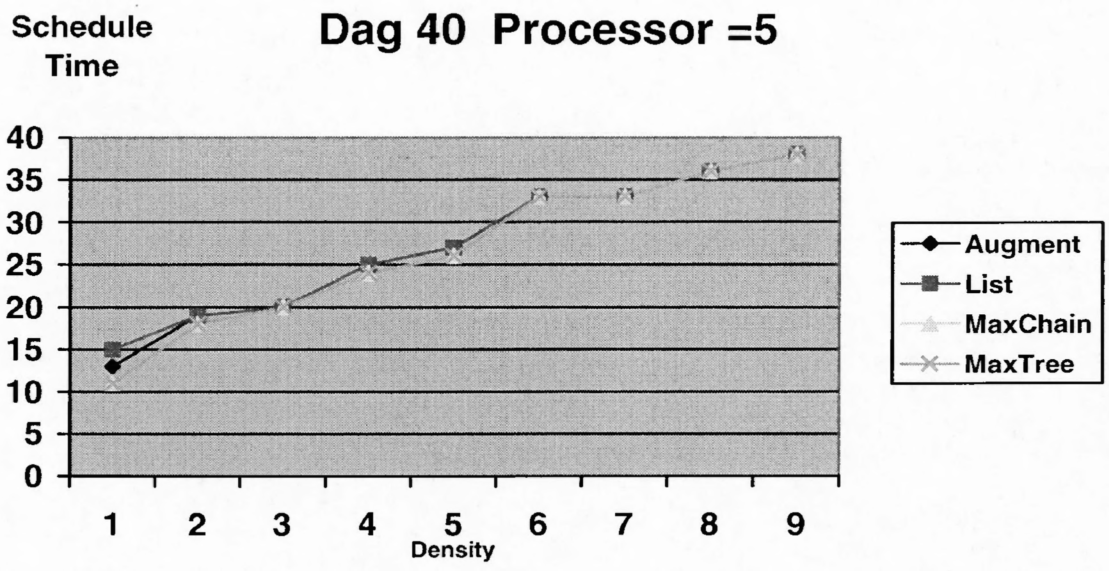
When processor number is two, maximal chain and maximal tree algorithm have the same best performance. List works better than Augmentation algorithm. Augmentation algorithm has the worst performance. All the algorithms have the same performance when density is above 0.7.



When processor number is three, maximal tree algorithm has the best performance. The next efficient algorithm is maximal chain and then is Augmentation algorithm. List algorithm has the worst performance. All the algorithms have the same performance when density is above 0.6.



When processor number is four, maximal chain algorithm and maximal tree algorithm have the same best performance, Augmentation algorithm works better than List algorithm and List algorithm has the worst performance. All the algorithms have the same performance when density is above 0.6.



When processor number is five, maximal chain algorithm and maximal tree algorithm have the same best performance, Augmentation algorithm is the next good algorithm and List algorithm has the worst performance. All the algorithms have the same performance when density is above 0.6.

6.5 Simulation 4

Goal:

Comparing the efficiency of augmentation, List, Max Chain and Max Tree heuristics with task graphs with nodes number 15, density varying from 0.04 to 0.14, and processor number is 3. The percentage of the maximal tree is varying from13% to 100%.

Input: 1) Task graph

- Size = 15
- Density (4% ~ 14%)

- The percentage of the maximal tree (13% ~ 100%)

2) Processor number =3

Output: The length of the schedule time

Simulations of graph (node number is 15, density 4%—14%, Processor Number 3, the percentage of the maximal tree (13% ~ 100%))

Input File	Percentage of tree	Augmentation	List	Max Chain	Max Tree
Dag15-.04-2	13%	6	6	5	5
Dag15-.05-3	23%	6	6	5	5
Dag15-.06-4	26%	6	6	6	6
Dag15-.08-7	46%	6	6	5	5
Dag15-.08-8	53%	7	6	5	5
Dag15-.09-9	60%	6	6	5	5
Dag15-.1-10	66%	6	6	5	5
Dag15-.11-11	73%	6	6	6	5
Dag15-.11-12	80%	7	7	6	5
Dag15-.12-13	86%	6	6	6	5
Dag15-.13-14	93%	7	7	6	6
Dag15-.14-15	100%	8	7	6	6

Conclusions from Simulation 4:

From above simulation results we can draw the following conclusions:

1) When the percentage of the tree is less than 70%, Maximal chain and maximal tree output the same length of schedule. But when the percentage

of the tree is above 70%, Maximal tree algorithm works better than maximal chain algorithm.

2) When the input task graph is the tree (see also Simulation1), both maximal chain and maximal tree yields the same best schedule.

3) Augmentation heuristic yields the worst result when input graph is a tree.

4) Maximal tree always works better than three other algorithms.

7 .Summary

From the above simulation results, we could conclude that Maximal Tree algorithm works better than three other heuristics. Maximal Chain Heuristic works better than Augmentation and List but worse than maximal tree. Augmentation Heuristic works better than List algorithms. List heuristic has the worst performance among all these four algorithms. During the augmentation process, the fewer edges we add, the better schedule we have. When the percentage of the tree is less than 70%, Maximal chain and maximal tree output the same length of schedule. But when the percentage of the tree is above 70%, Maximal tree algorithm works better than maximal chain algorithm.

8. Reference:

1. Task Scheduling in Parallel and Distributed System Hesham El-Rewini/ Theodore G.Lewis Hesham H. Ali p4.
2. A New Approach for Task Scheduling in Distributed System Hesham H. Ali and RajVemulapalli University of Nebraska at Omaha
3. Task Scheduling in Parallel and Distributed System Hesham El-Rewini/ Theodore G.Lewis Hesham Ali p19
5. A Maximal Chain Approach For scheduling tasks In a multiprocessor system Sachin Pawaskar P33 University of Nebraska at Omaha
6. A New Approach for Task Scheduling in Distributed System Hesham H. Ali and Raj Vemulapalli University of Nebraska at Omaha
7. A Maximal Chain Approach For scheduling tasks In a multiprocessor system Sachin Pawaskar
8. Integrated scheduling And Partitioning for Hardware/Software CoDesign Tanvir Sharif, Johnanns Grad , Illinois Institute of Technology
9. Benchmarking ther Task graph Scheduling Algorithms Yu_Kwong Kwok and Ishfag Ahmad
10. Conditional Task Scheduling on Loosely-Coupled Distributed Processors Lin Huang and Michael Oudshoorn , The University of Adelaide
11. On The Relationships Netween Linear Extensions and Multiprocessor Scheduling Jodi Wineman , University of Nebraska at Omaha

12. . “*Connection Scheduling in Web Servers*” M. Crovella, R. Frangioso, and M. Harchol-Balter In USENIX Symposium on Internet Technologies and Systems, June 1999.

13. "On-line machine scheduling with applications to load balancing and virtual circuit routing,"J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts, in Proc. 25th ACM Symp. on Theory of Computing, May 1993.

14. Performance Evaluation for Parallel Systems: A Survey - Hu, Gorton (1997) Lei Hu and Ian Gorton Department of Computer Systems School of Computer Science and Engineering University of NSW.

15. H. Shachnai and T. Tamir, “ Multiprocessor Scheduling with Machine Allotment and Parallelism Constrains” in Algorithmica, vol. 32:4, 651--678, 2002.

16. H. Shachnai and T. Tamir, ``Polynomial Time Approximation Schemes for Class-Constrained Packing Problems ''. J. of Scheduling (invited CONF 2000 issue), vol.4:6, 313--338, 2001. Preliminary version in Proc. of APPROX'00

17. J. Naor, H. Shachnai and T. Tamir, `` Real-time Scheduling with a Budget''. Preliminary version in Proc. of ICALP'03.

18. M. M. Halldorsson, G. Kortsarz, A. Proskurowski, R. Salman, H. Shachnai and J. A. Telle, `` Multicoloring Trees ''. Information and Computation, vol. 180, 113--129, 2003. Preliminary version in Proc. of COCOON'99.