

Student Work

---

5-1-2001

**A maximal chain approach for scheduling tasks in a multiprocessor system.**

Sachin Pawaskar

Follow this and additional works at: <https://digitalcommons.unomaha.edu/studentwork>

---

**Recommended Citation**

Pawaskar, Sachin, "A maximal chain approach for scheduling tasks in a multiprocessor system." (2001). *Student Work*. 3576.

<https://digitalcommons.unomaha.edu/studentwork/3576>

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact [unodigitalcommons@unomaha.edu](mailto:unodigitalcommons@unomaha.edu).



**A MAXIMAL CHAIN APPROACH FOR SCHEDULING  
TASKS IN A MULTIPROCESSOR SYSTEM**

**A THESIS**

**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**

**Sachin Pawaskar**

**May 2001**

UMI Number: EP74774

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI EP74774

Published by ProQuest LLC (2015). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 - 1346

# THESIS ACCEPTANCE

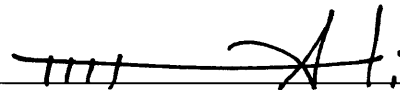
Acceptance for the faculty of the Graduate College,  
University of Nebraska, in partial fulfillment of the  
Requirements for the degree Masters of Science in Computer Science  
University of Nebraska at Omaha.

Committee

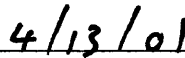




Chairperson



Date





## **Acknowledgements**

Hesham Ali, my advisor, who guided me through the process from beginning to end, Thank you for your encouragement and patience. Thank you for all your time and effort.

Kelly, my wife, without your support completing this work would not have been possible. Thanks for everything you did so that I would have more time to devote to this effort.

My mother Sharmila Pawaskar and my father Sudhakar Pawaskar for encouraging me throughout my life. My sister Suparna for all the happy childhood memories.

My daughter Sihley and son Samuel for bringing such tremendous joy and happiness in my life.

## **Abstract**

### A MAXIMAL CHAIN APPROACH FOR SCHEDULING TASKS IN A MULTIPROCESSOR SYSTEM

Sachin Pawaskar, MS

University of Nebraska, 2001

Advisor: Dr. Hesham Ali

The scheduling problem has been an interesting problem for quite some time. But recently, in the era of parallel and distributed computing it has seen increased activity and many researchers have focused their attention on the scheduling problem once again. Task scheduling is one of the most challenging problems in parallel and distributed computing. It is known to be NP-complete in its general form as well as several restricted cases. Researchers have studied restricted forms of the problem by constraining either the task graph representing the parallel tasks or the computer model. For example, the 2-processor problem has a polynomial-time algorithm.

In an attempt to solve the problem in the general case, a number of heuristics have been developed. 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 thesis, we study the scheduling problem for a fixed number of processors  $m$ . In the proposed work, we are suggesting generating the maximal chain and reducing the problem to  $(m-1)$  processors until we apply the two-processor scheduling algorithm on the remaining tasks. This way we can reduce an  $m$ -processor problem to a  $(m-1)$ -processor problem. We are trying a new heuristic approach, which tries to reduce the problem to a 2-processor problem; from then on it is just a matter of merging the

maximal chain and the derived  $(m-1)$  processor schedule. The motivation for reducing it to a 2-processor problem is because there are well known polynomial algorithms to solve this problem. The two-processor algorithm that we will be using is one of the famous algorithms by Coffman and Graham, Sethi, and Gabow.

The proposed heuristic will be compared with other well-known heuristics such as List scheduling heuristics. A user-friendly Graphical User Interface will be developed to simplify the use of the developed algorithm. The GUI will allow the user to create a task graph by plotting the nodes and the edges and then there will be various menu items, which will help in generating the labels, maximal chains and schedules for the plotted graph. The User will be able to save the plotted graph and functionality will be provided to copy, cut and paste entire graphs and portions of the graph.

# Table of Contents

1	Introduction .....	1
1.1	The Scheduling Problem .....	1
1.2	Task Scheduling Model .....	3
2	Basic Terminology and Problem Definition .....	5
2.1	Basic Terminology .....	5
2.2	Problem Definition .....	11
2.2.1	Target Machine: .....	12
2.2.2	Parallel Program Tasks: .....	13
2.2.3	Execution and Communication Cost .....	15
2.2.4	The Schedule.....	16
2.2.5	Performance Measures.....	17
3	Survey of Previous Work .....	19
3.1	The NP-Completeness of the Scheduling Problem .....	19
3.2	Special cases of the scheduling problem.....	22
3.2.1	Program and Machine Models.....	23
3.2.2	Scheduling Tree-Structured Task Graphs.....	23
3.2.3	Scheduling Interval-Ordered Tasks .....	26
3.2.4	Arbitrary task graph on two processors .....	28
3.3	List Scheduling Heuristics.....	32
3.3.1	List Scheduling .....	33
4	Proposed Solution .....	35
4.1	Motivation: .....	35
4.2	Program and Machine Models.....	36
4.3	Maximal chain scheduling heuristic.....	37
4.3.1	The Algorithm.....	38
4.3.2	Example .....	41
4.4	Conjecture: All maximal chains .....	45
5	Package.....	46
5.1	The Graph-It application .....	46
5.2	Random graph generator .....	48
5.3	Application Customization .....	49
5.4	Program specifications .....	50
6	Implementation and Results.....	51
6.1	Experiment-1 .....	53
6.1.1	3-processor scheduling for task graphs (25 to 35 nodes).....	53
6.1.2	3-processor scheduling for task graphs (40 to 90 nodes).....	54
6.1.3	3-processor scheduling for task graphs (100 to 400 nodes).....	55

6.2	Experiment-2 .....	56
6.2.1	Small task graphs (25 nodes) on 3,4 and 5 processors .....	57
6.2.2	Medium task graphs (100 –200 nodes) on 3,4 and 5 processors .....	64
6.2.3	Large task graphs (300 - 400 nodes) on 3,4 and 5 processors .....	78
7	Conclusions and Future Research .....	92
7.1	Conclusions .....	92
7.2	Future Research .....	93
	References.....	95

## List of Figures and Tables

Figure 1.1 The Scheduling System .....	2
Figure 2.1 A task graph.....	5
Figure 2.2 Maximal Chain in a task graph.....	7
Figure 2.3 Graph showing the tasks levels .....	8
Figure 2.4 Graph showing the tasks co-levels .....	9
Figure 2.5. Target Machine.....	13
Figure 2.6. Task graph with node numbers, execution times & communication costs. ...	14
Figure 2.7 A Scheduler .....	17
Table 3.1 Complexity comparison of scheduling problem.....	22
Figure 3.2. A tree-structured task graph. ....	24
Figure 3.3. A schedule for the task graph given in Figure 3.2 on 3 processors.....	26
Figure 3.4. An interval-ordered task graph.....	27
Figure 3.5. A schedule for the interval order given in Figure 3.4 on 3 processors.....	28
Figure 3.6. A task graph.....	31
Figure 3.7. A 2-processor schedule for the task graph given in Figure 3.6 .....	31
Figure 3.8. A task graph consisting of twelve tasks. ....	34
Figure 3.9 The level and co-level of tasks in task graph of Figure 3.8.....	34
Figure 4.1 A task graph.....	42
Figure 4.2: Maximal chain for task graph in Figure 4.1 .....	42
Figure 4.3: Simple Merge of maximal chain and 2-processor schedule.....	43
Figure 4.4: Schedule after Merge and Check Violations/Feasibility.....	44
Figure 4.5: Schedule after Merge and Check Violation/Feasibility and Optimization.....	44

Figure 5.1 Graph-It graphical program.....	47
Figure 5.2 Random graph generation dialog box.....	49
Figure 5.3 Customize dialog box .....	50
Figure 6.1. Graphs with 25 nodes on 3 processors .....	58
Figure 6.2. Graphs with 25 nodes on 4 processors .....	60
Figure 6.3. Graphs with 25 nodes on 5 processors .....	62
Figure 6.4. Graphs with 100 nodes on 3 processors .....	65
Figure 6.5. Graphs with 100 nodes on 4 processors .....	67
Figure 6.6. Graphs with 100 nodes on 5 processors .....	69
Figure 6.7. Graphs with 200 nodes on 3 processors .....	72
Figure 6.8. Graphs with 200 nodes on 4 processors .....	74
Figure 6.9. Graphs with 200 nodes on 5 processors .....	76
Figure 6.10. Graphs with 300 nodes on 3 processors .....	79
Figure 6.11. Graphs with 300 nodes on 4 processors .....	81
Figure 6.12. Graphs with 300 nodes on 5 processors .....	83
Figure 6.13. Graphs with 400 nodes on 3 processors .....	86
Figure 6.14. Graphs with 400 nodes on 4 processors .....	88
Figure 6.15. Graphs with 400 nodes on 5 processors .....	90

# 1 Introduction

Scheduling is a classical field with several interesting problems and results. A scheduling problem emerges whenever there is a choice. The choice could be the order in which a number of tasks can be performed, and/or in the assignment of tasks to servers for processing. A problem may involve jobs that need to be processed in a manufacturing plant, bank customers waiting to be served by tellers, aircrafts waiting for landing clearances, or program tasks to be run on a parallel or a distributed computer. Clearly, there is a fundamental similarity to scheduling problems regardless of the difference in the nature of the tasks and the environment.

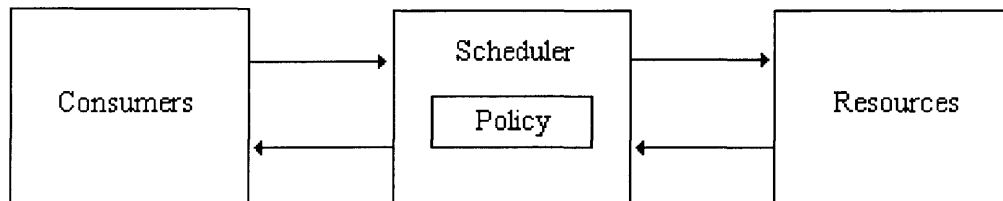
## 1.1 The Scheduling Problem

The scheduling problem has been described in a number of different ways in different fields. The classical problem of job sequencing in production management has influenced most of what has been written about this problem. Most manufacturing processes involve several operations to transform raw material into a finished product. The problem is to determine some sequences of these operations that are preferred according to certain (e.g. economic) criteria. The problem of discovering these preferred sequences is referred to as the sequencing problem. Over the years, several methods have been used to deal with the sequencing problem such as complete enumeration, heuristic rules, integer programming, and sampling methods. It is clear that complete enumeration is impractical because the problem is exponential, which means that it requires too much time, sometimes years of computation time would be required even for a small number of



tasks. Hence optimal solutions cannot be obtained in real time. However, heuristic methods have been used to deal with most general case of the problem.

In general, the scheduling problem assumes a set of resources and a set of consumers serviced by these resources according to a certain policy. Based on the nature of and the constraints on the consumers and the resources, the problem is to find an efficient policy for managing the access to and the use of the resources by various consumers to optimize some desired performance measure such as schedule length. Accordingly, a scheduling system can be considered as consisting of a set of consumers, a set of resources, and a scheduling policy as shown in Figure 1.1. Examples of consumers are a task in a program, a job in a factory, or a customer in a bank. Examples of resources are a processing element in a computer system, a machine in a factory, or a teller in a bank. First-come-first-served is an example of a scheduling policy. Scheduling policy performance varies with different circumstances. While first-come-first-served may be appropriate in a bank environment, it may not necessarily be the best policy to be applied to jobs on a factory floor.



**Figure 1.1 The Scheduling System**

Performance and efficiency are two parameters used to evaluate a scheduling system. We should evaluate a scheduling system based on the goodness of the produced schedule and the efficiency of the policy.

In this thesis, we are concerned with scheduling program tasks on parallel and distributed systems. The tasks are the consumers and will be represented using directed graphs called task graphs. Task graphs are used because we can represent precedence relationships between tasks. The processing elements are the resources and their interconnection networks will be represented using undirected graphs. The “scheduler” generates a schedule using a timing diagram called the Gantt chart. The scheduler performs allocation, which means it will tell which tasks go on which processor, but does not give their order. Whereas “scheduling” will perform allocation as well as provide an order for the tasks on the individual processors. The Gantt chart illustrates the allocation of the parallel program tasks onto the target machine processors and their execution order. A Gantt chart consists of a list of all processors in the target machine and, for each processor, a list of all tasks allocated to that processor ordered by their execution time. The term tasks, nodes and jobs will be regarded as equivalent to the term “consumers”. Also, resources may be referred to as processors or processing elements.

## **1.2 Task Scheduling Model**

The model that we will study in this thesis is deterministic and static in the sense that all information governing the scheduling decisions are assumed to be known in advance. In particular, the task graph representing the parallel program and the target machine is assumed to be available.

There are four components in any scheduling system: the target machine, the parallel tasks, the generated schedule, and the performance criterion. In our task-scheduling model we will ignore the communication delays and consider all tasks to have the same unit execution time. Also most of the time, we deal with the same machine, i.e. multiple processors on the same machine. Nowadays we have such similar environments that it leads to almost same communication delay times. We will discuss and define the scheduling problem in more detail later in the thesis.

## 2 Basic Terminology and Problem Definition

In this section we define a few terms that will be used in the later sections of this thesis.

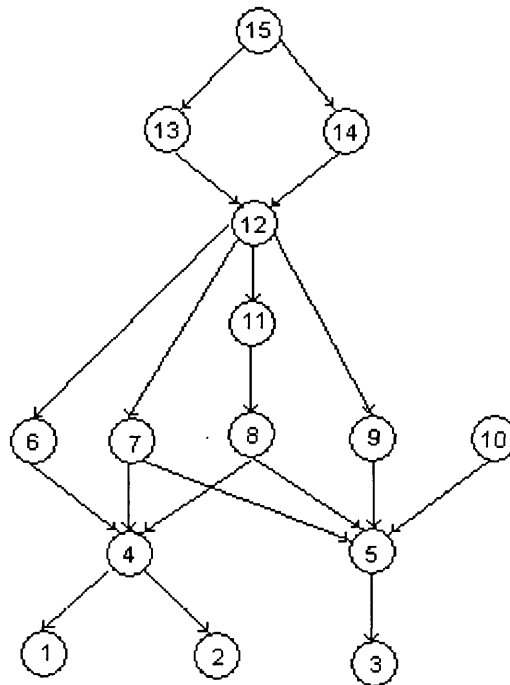
We will also define the scheduling problem in its most general form and then we will study some of the special cases of this problem and some of the classical algorithms that have been published to solve these special cases.

### 2.1 Basic Terminology

In this section we define a few terms that will be used in the later sections of this thesis.

**Graph:** A graph is a pervasive data structure in computer science. A graph is usually defined as  $G = (V, E)$ , where  $|V|$  is the vertices and  $|E|$  is the Edges of the graph.

**Task Graph:** A task graph is a directed acyclic graph. A directed edge  $(i, j)$  between two tasks  $t_i$  and  $t_j$  specifies that  $t_i$  must be completed before  $t_j$  can begin.



**Figure 2.1 A task graph**

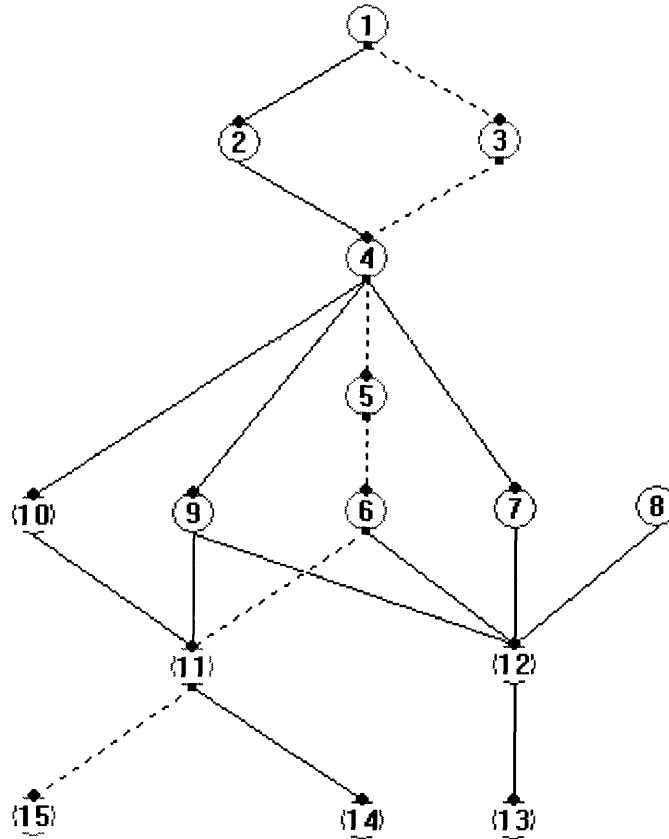
**Density or Sparseness:** The density or sparseness of a graph is computed as a ratio of the number of edges  $|E|$  in the graph as a percentage to the maximum number of edges that graph can have which is  $|V|^2$ . So a graph with density of 0.5 will have half the number of maximum edges possible for that graph.

**Sparse Graph:** A graph  $G = (V, E)$  is said to be sparse if  $|E|$  is much less than  $|V|^2$ .

**Dense Graph:** A graph  $G = (V, E)$  is said to be dense if  $|E|$  is close to  $|V|^2$ .

**Task Level:** 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. Given the task graph in Figure 2.1, we can say that nodes 1,2 and 3 are at level 1, 4 and 5 are at level 2, nodes 6,7,8,9 and 10 are at level 3, and so on.

**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  $S$ . Given the task graph in Figure 2.1, a maximal chain consists of tasks 1,3,4,5,6,11,15 as shown in Figure 2.2.



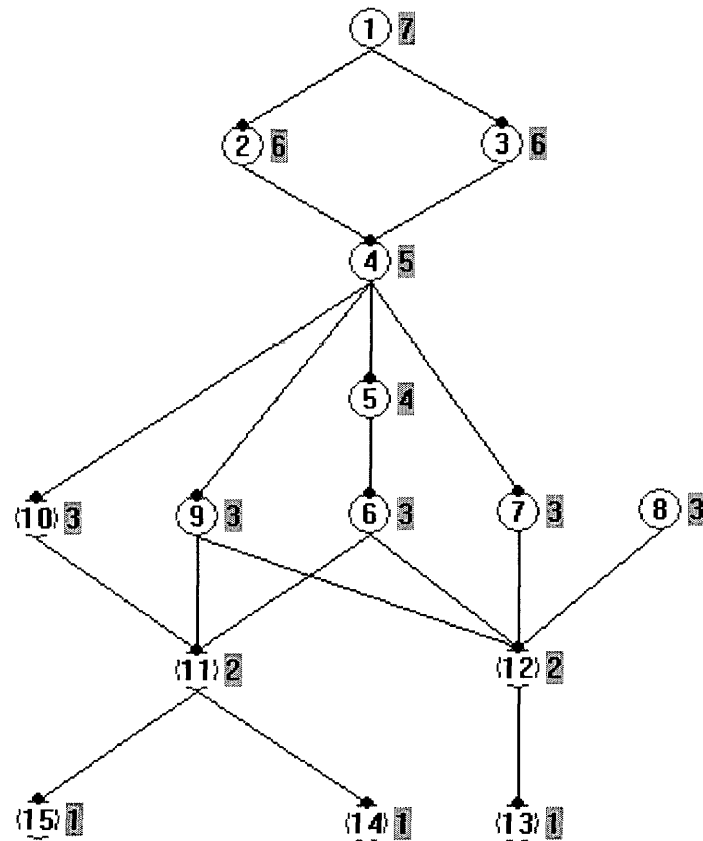
**Figure 2.2 Maximal Chain in a task graph**

**(Maximal Chain is 1,3,4,5,6,11,15)**

**Ready Task:** Let a task be ready when it has no predecessors or when all of its predecessors have already been executed.

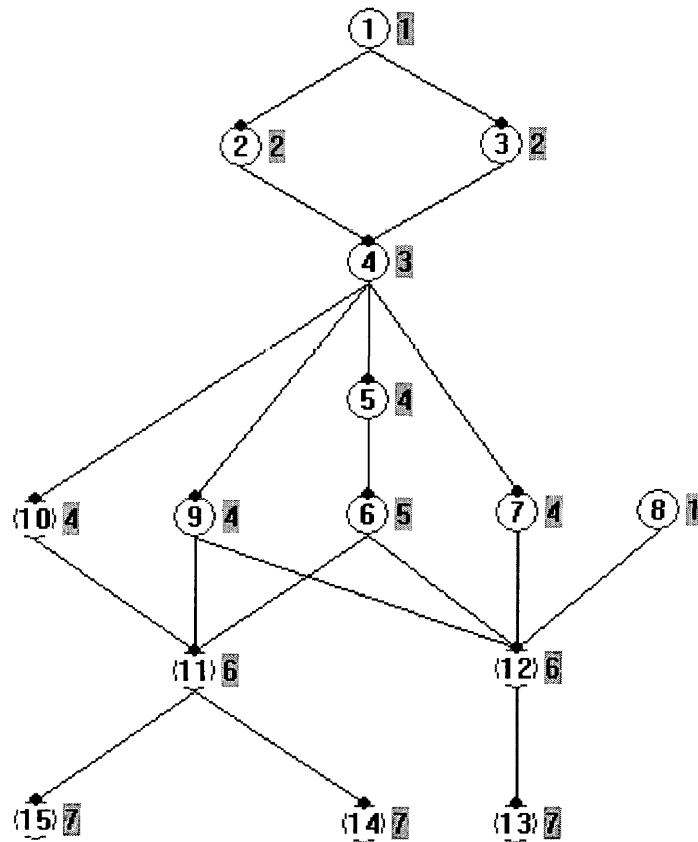
**Maximal Task:** Given a task graph  $G = (V, A)$ , let  $S$  be an arbitrary subset of tasks in  $G$ ; then we say that  $u \in S$  is a maximal task in  $S$  if there does not exist task  $v \in S$  such that  $v$  precedes  $u$  in  $G$ . Given the task graph in Figure 2.1, we can say that nodes 10 and 15 are Maximal tasks.

**Level:** The level of a node in a task graph is defined as the length (number of nodes) of the longest path from the node to an exit node (an exit node is the one with no successors). Figure 2.3 shows the task levels for a given task graph.



**Figure 2.3 Graph showing the tasks levels**

**Co-Level:** The co-level of a node in a task graph is defined in the same way as a level except that the lengths are measured from the starting points of the task graph rather than from the exit node. Figure 2.4 shows the Co-Levels for a given task graph.



**Figure 2.4 Graph showing the tasks co-levels**

**Schedule Length or Schedule Time:** Given a task graph  $G = (T, A)$  and its schedule on  $m$  processors,  $f$ , the length of schedule  $f$  of  $G$  is the maximum finishing time of any task in  $G$ . Formally,  $\text{length}(f) = t_{\max}$  where  $t_{\max} = \text{maximum} \{t + T_{ij}\}$  where  $f(i) = ((j, t) \forall i \in T, 1 \leq j \leq m$ . Note  $T_{ij}$  is the execution time of task  $i$  on processor  $j$ .

**List Scheduling:** 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 tasks contend for processors, the selection of tasks to be immediately processed is done on the basis of priority with the higher-priority tasks being assigned processors first. If there is more than one task of a given priority, ties are broken randomly.



**Path Length:** The length of a path in a task graph is the summation of the weights of all nodes along the path including the initial and final node.

**Ready Time:** The ready time of a processor is the time when the processor has finished its assigned task and is ready for another task.

**Ready Queue:** The ready queue is a queue of ordered ready tasks. Tasks are ordered according to their levels; the task with the highest level is scheduled first. Tasks at the same level are ordered according to the number of immediate successors; the task with the greatest number of immediate successors is scheduled first.

**Assigned task:** The Assigned task is the highest-priority task selected from the ready queue.

**Idle time slot:** The idle time slot is the time interval between the ready time of a processor and the assigned task's starting time.

**Assigned processor:** The assigned processor is the one chosen to execute the assigned task.

**First Ready Processor:** The first ready processor is the first processor in the set of all processors to become ready after the assigned task is scheduled on the assigned processor.

**Parallel Program:** A parallel program is modeled as a partially ordered set (poset)  $(T, <)$ , where  $T$  is a set of tasks. The relation  $u < v$  implies that the computation of task  $v$  depends on the results of the computation of task  $u$ , i.e. task  $u$  must be computed before task  $v$ , and the result of the computation of task  $u$  must be known by the processor computing task  $v$ .

**NP-completeness:** Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial time factor. That is, if  $L_1 \preceq L_2$ , then  $L_1$  is not more than a polynomial factor harder than  $L_2$ , which is why the “less than or equal to” notation is mnemonic. We can now define the set of NP-complete languages, which are the hardest problems in NP.

A language  $L = \{0, 1\}^*$  is **NP-complete** if

1.  $L \in \text{NP}$ , and
2.  $L' \preceq L$  for every  $L' \in \text{NP}$ .

If a language  $L$  satisfies property 2, but not necessarily property 1, we say that  $L$  is **NP-hard** [TCR90].

## 2.2 Problem Definition

In this section we describe the general model to formulate the scheduling problem. The problems that we survey in the next chapter can be represented as special cases of the model. Our model of the problem is deterministic in the sense that all information governing the scheduling decisions are assumed to be known in advance. In particular, the task graph representing the parallel program and the target machine is assumed to be available before the program starts execution.

There are four components in any scheduling system:

- 1) The Target Machine.
- 2) The Parallel Tasks (represented as a task graph).
- 3) The Generated Schedule.
- 4) The Performance criterion.

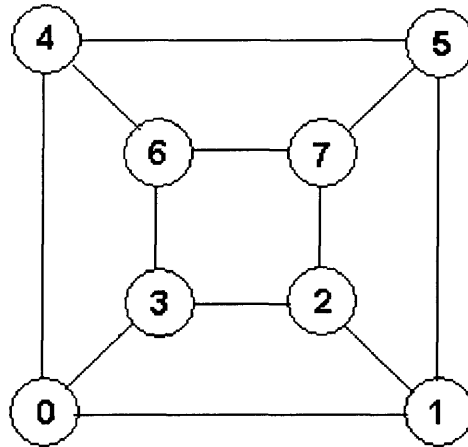
We show each one of these components and show how the program and target machine parameters can be used to estimate execution times. For this thesis, we will assume there are no communication delays.

### 2.2.1 Target Machine:

The target machine is assumed to be made up of  $m$  heterogeneous processing elements connected using an arbitrary interconnection network. Each processing element can run one task at a time and all tasks can be processed by any processing element. Formally, the target machine characteristics can be described as a system  $(P, [P_{ij}], [S_i], [I_i], [B_i], [R_{ij}])$  as follows:

1.  $P = \{P_1, \dots, P_m\}$  is a set of processors forming the parallel architecture.
2.  $[P_{ij}]$  is an  $m \times m$  interconnection topology matrix.
3.  $S_i, 1 \leq i \leq m$ , specifies the speed of processor  $P_i$ .
4.  $I_i, 1 \leq i \leq m$ , specifies the startup cost of initiating a message on processor  $P_i$ .
5.  $B_i, 1 \leq i \leq m$ , specifies the startup cost of initiating a process on processor  $P_i$ .
6.  $R_{ij}$  is the transmission rate over the link connecting two adjacent processors  $P_i$  and  $P_j$ .

The connectivity of the processing elements can be represented using an undirected graph called the target machine graph. Figure 2.5 shows an example of a target machine consisting of eight processors ( $m = 8$ ) forming a three dimensional hypercube. Processors are occasionally referred to simply by their indices (i.e., 1 may be used rather than  $P_1$ , especially when target machine nodes are conveniently labeled with integers).



**Figure 2.5. Target Machine.**

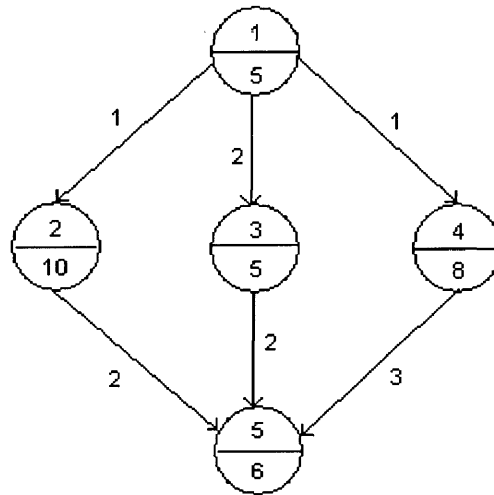
### 2.2.2 Parallel Program Tasks

A parallel program is modeled as a partially ordered set (poset)  $(T, <)$ , where  $T$  is a set of tasks. The relation  $u < v$  implies that the computation of task  $v$  depends on the results of the computation of task  $u$ , i.e. task  $u$  must be computed before task  $v$ , and the result of the computation of task  $u$  must be known by the processor computing task  $v$ .

The characteristics of a parallel program can be defined as the system  $(T, <, [D_{ij}], [A_i])$  as follows:

1.  $T = \{t_1, \dots, t_n\}$  is a set of tasks to be executed.
2.  $<$  is a partial order defined on  $T$  which specifies operational precedence constraints. That is,  $t_i < t_j$  signifies that  $t_i$  must be completed before  $t_j$  can begin.
3.  $[D_{ij}]$  is an  $n \times n$  matrix of communication data where  $D_{ij} \geq 0$  is the amount of data required to be transmitted from task  $t_i$  to task  $t_j$ ,  $1 \leq i, j \leq n$ .
4.  $[A_i]$  is an  $n$  vector of the amount of computations, i.e.,  $A_i > 0$  is the number of instructions required to execute  $t_i$ ,  $1 \leq i \leq n$ .

The partial order  $<$  is represented as a directed acyclic graph called a task graph. A directed edge  $(i, j)$  between two tasks  $t_i$  and  $t_j$  specifies that  $t_i$  must be completed before  $t_j$  can begin. Figure 2.6 shows an example of a task graph consisting of five nodes ( $n = 5$ ), where each node represents a task.



**Figure 2.6. Task graph with node numbers, execution times & communication costs.**

The number shown in the upper portion of each node is the node number, the number in the lower portion of a node  $i$  represents the parameter  $A_i$  (the amount of computation needed by task  $t_i$ ), and the number next to an edge  $(i, j)$  represents the parameter  $D_{ij}$ . For example,  $A_1 = 5$ ,  $D_{45} = 3$ . Tasks are referred to simply by their indices (e.g., 1 may be used rather than  $t_1$ , especially when graph nodes are more conveniently labeled with integers). As part of this thesis for the proposed heuristic, we will ignore the communication delay and also assume that each task takes the same amount of time to process.

### 2.2.3 Execution and Communication Cost

Given a parallel program model and the description of the target machine that will execute the program, task execution time and the communication delay can be obtained as follows:

1.  $T_{ij}$ : the execution time of task  $i$  when executed on processor  $j$ . It can be computed as follows.

$$T_{ij} = A_i / S_j + B_j$$

Where  $A_i$  is a function of the computational complexity of the task. Thus,  $A_i(S, x)$  will be used to model the movement of  $S$  bytes of data to  $x$  replicated tasks; even when  $x = 1$ ,  $A_i(S, 1)$  is an accurate model.

2.  $C(i_1, i_2, j_1, j_2)$ : Communication delay between tasks  $i_1$  and  $i_2$  when they are executed on processing elements  $j_1$  and  $j_2$ , respectively. It reflects the target machine performance parameters as well as the size of the data to be transmitted and can be computed as follows:

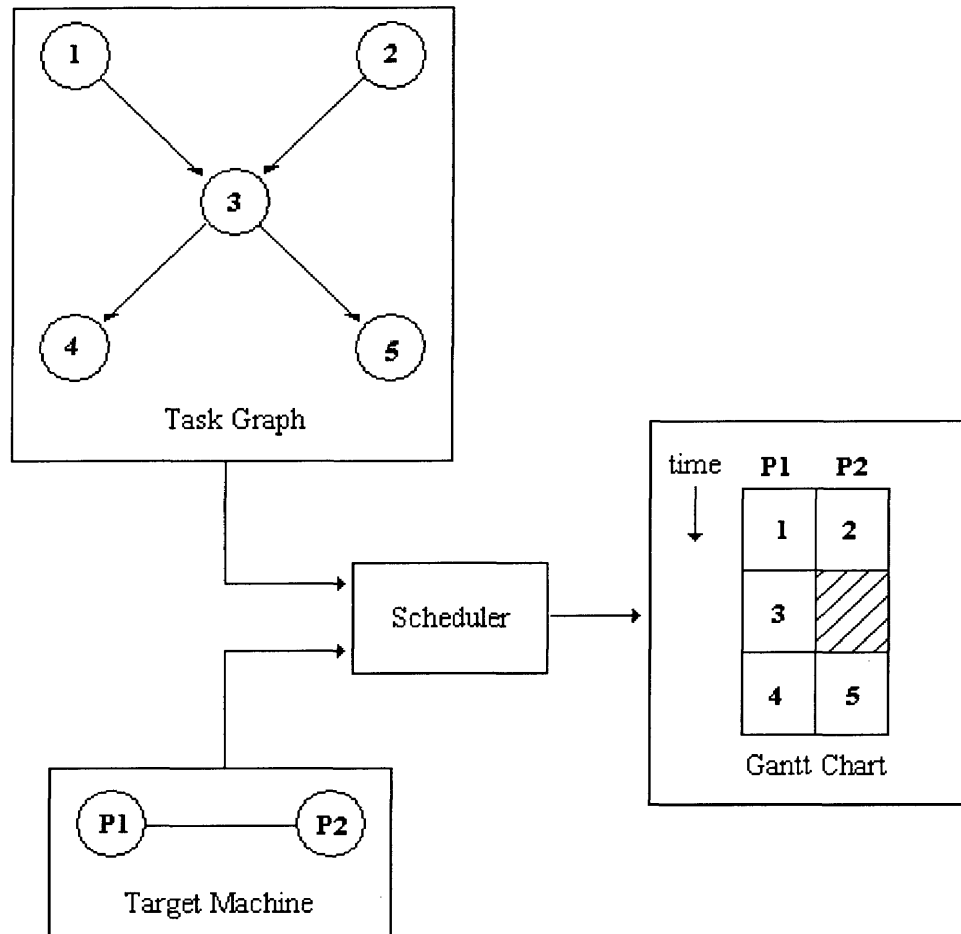
Suppose that  $j_1$  and  $j_2$  are two adjacent processing elements, then the communication delay of a message sent from task  $i_1$  running on  $j_1$  to task  $i_2$  running on  $j_2$  over a free link is:

$$C(i_1, i_2, j_1, j_2) = D_{i_1 i_2} / R_{j_1 j_2} + I_{j_1}.$$

We will ignore contention delay. Also note that for the purpose of this thesis, we will assume execution time is the same for all tasks and there is no communication delay.

### 2.2.4 The Schedule

A schedule of the task graph  $G = (T, A)$  on a target machine made up of  $m$  processors is a function  $f$  that maps each task to a processor and a starting time. Formally,  $f: T \rightarrow \{1, 2, \dots, m\} \times [0, \infty)$ . If  $f(v) = (i, t)$ , for some  $v \in T$  we say that task  $v$  is scheduled to be processed by processor  $p_i$  starting at time  $t$ . Note that there exists no  $u, v \in T$  such that  $f(v) = f(u)$ . If  $v < u$  and  $f(v) = (i, t_1)$ ,  $f(u) = (j, t_2)$ , then  $t_1 < t_2$ . A schedule  $f$ , is a feasible schedule if it preserves all the precedence relations and communication restrictions. The Gantt chart gives an informal notion of the schedule where the start and finish times for all tasks can be easily shown. Figure 2.7 shows a scheduling system where the input is a task graph and a target machine description while the output schedule is shown in the form of a Gantt chart.



**Figure 2.7 A Scheduler**

**that takes task graph and target machine topologies as inputs, and produces a Gantt chart as output.**

### 2.2.5 Performance Measures

Now in light of the description of the scheduling problem, we would like to find efficient algorithms for scheduling the tasks on the available processors to optimize some desired performance measure. There are several performance criteria such as: balancing the load, utilization of processors and minimizing completion time. In this thesis, our scheduling goal is to minimize the total completion time of a parallel program. This performance



measure is known as the schedule length or maximum finishing time. Schedule length can be described as follows. Given a task graph  $G = (T, A)$  and its schedule  $f$ , on  $m$  processors, the length of schedule  $f$  of  $G$  is the maximum finishing time of any task in  $G$ . Formally,  $\text{length}(f) = t_{\max}$  where  $t_{\max} = \text{maximum } \{t + T_{ij} \}$  where  $f(i) = ((j,t) \forall i \in T, 1 \leq j \leq m$ . Note  $T_{ij}$  is the execution time of task  $i$  on processor  $j$  in our case  $T_{ij} = 1$  unit.

**[RLA94]**

### 3 Survey of Previous Work

In general, the time complexity of an algorithm refers to its execution time as a function of its input. We specify the complexity of a scheduling algorithm as a function of the number of tasks and the number of processors. A scheduling algorithm whose time complexity is bounded by a polynomial is called a polynomial-time algorithm. An optimal algorithm is considered to be efficient if it runs in polynomial time. Inefficient algorithms are those, which require a search of the whole enumerated space and have an exponential time complexity. The problem of scheduling parallel programs tasks on multiprocessor systems is known to be NP-complete in its general form. There are few known polynomial-time scheduling algorithms even when severe restrictions are placed on the task graph representing the program and the parallel processor models. In general we can say classify the scheduling problems as follows:

- 1) NP-Complete scheduling problems.
- 2) Scheduling problems that have heuristics algorithms
- 3) Special case scheduling problems, which have polynomial time optimal algorithms.

#### 3.1 The NP-Completeness of the Scheduling Problem

In this section, a list of some of the NP-Complete results in the scheduling problem is provided. There is a host of important problems that are roughly equivalent in complexity. These problems form a class called the NP-complete problems. This class of problem includes many classical problems in combinatorics, graph theory, and computer science such as the traveling salesman problem, the Hamilton circuit problem, and integer

programming. The best-known algorithms for these problems could take exponential time on some inputs. The exact complexity of these NP-complete problems has yet to be determined and remains the foremost open problem in theoretical computer science.

Either all of these problems have polynomial-time solutions or none of them do.

Formally, a problem  $P_0$  is NP-complete if  $P_0$  is in **NP** and  $P_0$  is NP-hard. A problem is in **NP** if it is accepted by a non-deterministic Turing machine in polynomial time. A  $P_0$  is NP-hard if any problem in **NP** polynomially transforms to  $P_0$ .

It has been shown that the problem of finding an optimal schedule for a set of tasks is NP-complete in the general case and in several restricted cases, even when communication among tasks is not considered. The following are formal definitions of some versions of the scheduling problem proven to be NP-complete.

**(P1):** General scheduling problem. Given a set  $T$  of  $n$  tasks, a partial order  $<$  on  $T$ , weight  $A_i$ ,  $1 \leq i \leq n$ , and  $m$  processors, and a time limit  $k$ , does there exist a total function  $h$  from  $T$  to  $\{0, 1, \dots, k-1\}$  such that:

- i) if  $i < j$ , then  $h(i) + A_i \leq h(j)$
- ii) for each  $i$  in  $T$ ,  $h(i) + A_i \leq k$
- iii) for each  $t$ ,  $0 \leq t \leq k$ , there are at most  $m$  values of  $i$  for which  $h(i) \leq t < h(i) + A_i$ ?

The following problems are special cases of P1.

**(P2):** Single Execution Time Scheduling. We restrict P1 by requiring  $A_i = 1$ ,  $1 \leq i \leq n$ . (all tasks require one time unit)

**(P3):** Two Processor, one or two time-units scheduling. We restrict P1 by requiring  $m=2$ , and  $A_i$  in  $\{1, 2\}$ ,  $1 \leq i \leq n$ . (all tasks require one or two time units, and there are only two processors)

**(P4):** Two Processor, interval-order scheduling. We restrict P1 by requiring the partial order  $<$  to be an interval order and  $m=2$ .

**(P5):** Single Execution Time, Opposing Forests. We restrict P1 by requiring  $A_i = 1$ ,  $1 \leq i \leq n$ , and the partial order  $<$  to be an opposing forest.

Problem P1 was proven to be NP-complete by Richard M. Karp in 1972.

Problems P2 and P3 were proven to be NP-complete by Jeffery D. Ullman in 1975

[Ullm75]. Problem P4 was proven to be NP-complete by Christos H. Papadimitriou and

Mihalis Yannakakis in 1979 [PaYa 79]. In 1983 Michael R. Garey, David S. Johnson,

Robert E. Tarjan, and Mihalis Yannakakis proved that P5 is also NP-complete [GJTY83].

The following table 3.7 summarizes the complexity of several versions of the scheduling problem when communication is not considered and the target machine is fully connected. Note that  $n$  is the number of tasks and  $e$  is the number of arcs in the task graph.

Task Graph	Task Execution Time	Number of Processors	Complexity
Tree	Identical	Arbitrary	$O(n)$
Interval order	Identical	Arbitrary	$O(n)$
Arbitrary	Identical	2	$O(e + n\alpha(n))$
Arbitrary	Identical	Arbitrary	NP-complete
Arbitrary	1 or 2 time units	$\geq 2$	NP-complete
Opposing forest	Identical	Arbitrary	NP-complete
Interval order	Arbitrary	$\geq 2$	NP-complete
Arbitrary	Arbitrary	Arbitrary	NP-complete

**Table 3.1 Complexity comparison of scheduling problem  
(Assuming fully connected target machine and zero communication)**

### 3.2 Special cases of the scheduling problem.

In this section some efficient optimal algorithms are presented that have polynomial time complexity. Polynomial algorithms can be obtained in the following cases:

- 1) When the task graph is a tree.
- 2) When the task graph is an interval order, and
- 3) When there are only two processors available.

In the three cases all tasks are assumed to have the same execution time, and communication costs between tasks is not considered. For a task graph that is a tree that executes all tasks in one time unit, Hu introduced a linear algorithm that uses level number equal to the length of the longest path from the node to the ending node as a priority number. Papadimitriou and Yannakakis showed that interval ordered tasks can be

scheduled in linear time on an arbitrary number of processors. Coffman and Graham gave a polynomial-scheduling algorithm that allows an optimal length schedule for an arbitrary task graph on a two-processor system. [RLA94]

### 3.2.1 Program and Machine Models

In this chapter, we assume a restricted version of the general model introduced in the previous chapter. Recall that the target machine model was represented as the 6-tuple  $(P, [P_{ij}], [S_i], [I_i], [B_i], [R_{ij}])$ , while the parallel program tasks were represented as the 4-tuple  $(T, <, [D_{ij}], [A_i])$ . The assumptions are as follows:

#### Target Machine

- $P_{ij} = 1, 1 \leq i, j \leq m$ . (A fully connected system)
- $S_i = \text{Constant}, 1 \leq i \leq m$ . (Identical processing elements)
- $I_i = 0, 1 \leq i \leq m$ . (No startup costs for initiating a message)
- $B_i = 0, 1 \leq i \leq m$ . (No startup costs for initiating a task)

#### Program Tasks

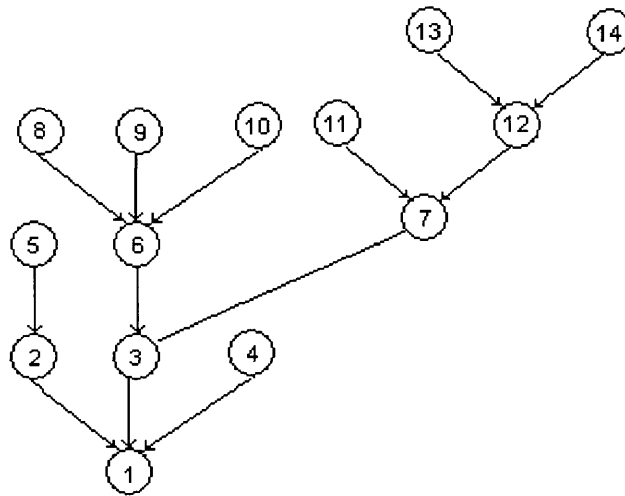
- $D_{ij} = 0, 1 \leq i, j \leq n$ . (No communication between task)

It follows that the execution time,  $T_{ij}$  of task  $i$  on processor  $j$ , will be the same for all processors and is equal to  $A_i / S_j$ . Assuming that  $S_j = 1$ , we can use  $A_i$  as the execution time of task  $i$ .

### 3.2.2 Scheduling Tree-Structured Task Graphs

In a classic paper, Hu, presented an 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 section, we assume that the task graph is an in-forest of  $n$  tasks as shown in Figure 3.2. There is no loss of generality if we assume that all the tasks have unit execution times. The algorithm given in this section is linear for determining a minimal-length schedule [Hu 61].



**Figure 3.2. A tree-structured task graph.**

### **Task Level**

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.

### **Ready Tasks**

Let a task be ready when it has no predecessors or when all of its predecessors have already been executed.

**Algorithm 3.1**

1. The level of each node in the task graph is calculated as given above and used as each node's priority.
2. Whenever a processor becomes available, assign it the unexecuted ready task with the highest priority.

Algorithm 3.1 can be used in the case of out-forest with a simple modification. However, this algorithm will not produce an optimal solution in the case of an opposing forest. An opposing forest is the disjoint union of an in-forest and an out-forest. Scheduling an opposing forest is proven to be NP-complete.

**Example 3.1**

Consider the problem of scheduling the task graph given in Figure 3.2 on a fully connected target machine of three identical processors. Applying Algorithm 3.1, we first compute the level at each node. At the beginning, among all the ready tasks (4, 5, 8, 9, 10, 11, 13, 14), tasks 13, 14 with level 5 are assigned first as shown in the resulting schedule in Figure 3.3. Following the path from tasks 13 or 14 to the terminal node 1, it can be noticed that regardless of the number of processors available, at least five units of time will be required to execute all of the tasks in the system. Having only three processors, the optimal schedule length is 6.



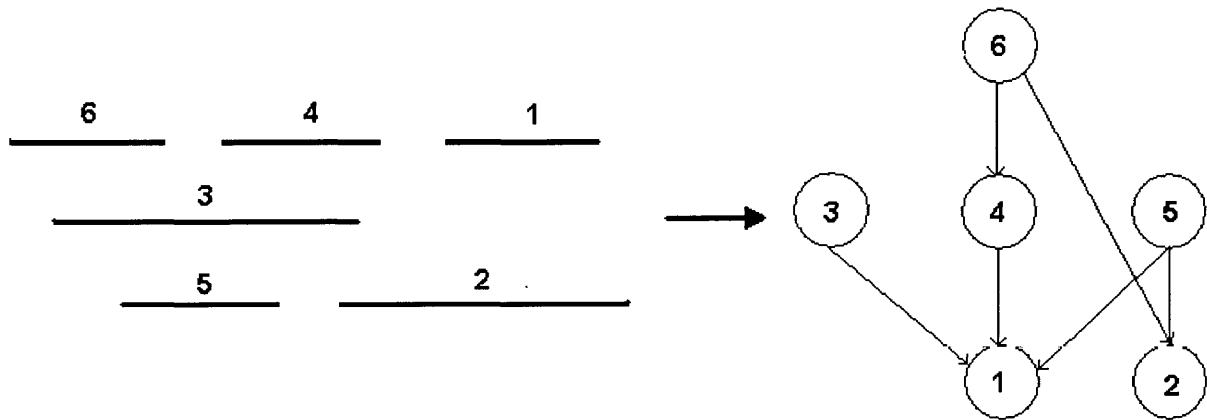
P1	P2	P3
14	13	11
12	10	9
8	7	5
6	4	2
3		
1		

**Figure 3.3.** A schedule for the task graph given in Figure 3.2 on 3 processors.

### 3.2.3 Scheduling Interval-Ordered Tasks

In this section, we deal with a special class of system tasks called interval-ordered tasks.

The term interval-ordered tasks is used to indicate that the task graph which describes the precedence relations among the system tasks is an interval order. A task graph is an interval order when its elements can be mapped into intervals on the real line and two elements are related if and only if the corresponding intervals do not overlap. Formally, a partially ordered set  $(V, <)$  is an interval order if its elements can be mapped into intervals on the real line  $(\mathbb{R}, <)$ , such that  $\forall x, y \in V, x < y$  if and only if the interval assigned to  $x$  completely precedes the interval assigned to  $y$ . In other words, if the interval that is assigned to each element  $v \in V$  is represented by a left point  $L(v)$  and a right point  $R(v)$ , then  $x < y$  if and only if  $R(x) < L(y)$ . An example of an interval order and its corresponding interval representation is given in Figure 3.4.



**Figure 3.4. 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  $u, v \in V$ , either  $N(v) \cap N(u)$  or  $N(u) \cap N(v)$  where  $N(v) = \{u: (v,u) \in A\}$ . 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 Algorithm 3.2 [PaYa 79].

**Algorithm 3.2**

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, since the number of successors of all tasks can be computed in  $O(|A|)$  time and sorting the tasks, according to the number of successors, can be done in  $O(|V|)$  time by bucket sort. Implementing the schedule in Step 2 can be done in  $O(|V|)$  time.

**Example 3.2**

Consider the problem of scheduling the interval order given in Figure 3.4 on a fully connected target machine of three identical processors. Applying Algorithm 3.2, Figure 3.5 shows the resulting schedule.

P1	P2	P3
6	5	3
4	2	
1		

**Figure 3.5. A schedule for the interval order given in Figure 3.4 on 3 processors**

**3.2.4 Arbitrary task graph on two processors**

There are no known polynomial algorithms for scheduling task graphs where all tasks have the same exact execution time on a fixed number of processors,  $m$ , if  $m > 2$ . The

first polynomial time algorithm for  $m = 2$ , based on matching techniques, was presented by Fujii. The time complexity of Fujii's algorithm is  $O(n^{2.5})$ . Improved algorithms have been obtained by Coffman and Graham, Sethi, and Gabow [**Coff 76, Seth 82, Gabo 82**]. The time complexity of these three algorithms are  $O(n^2)$ ,  $O(\min(en, n^{2.61}))$ , and  $O(e + n\alpha(n))$ , respectively, where  $n$  is the number of nodes and  $e$  is the number of arcs in the task graph.

In this section the Coffman and Graham algorithm is presented. The approach is to assign labels giving priority to tasks, a list for scheduling the task graph is constructed from the labels. Labels from the set  $\{1, 2, \dots, n\}$  are assigned to each task in the task graph by the function  $L(*)$  as explained in the following Algorithm 3.3.

**Algorithm 3.3**

1. Assign the number 1 to one of the terminal tasks.
2. Let labels  $1, 2, \dots, j - 1$  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  $\{L(y_1), L(y_2), \dots, L(y_k)\}$ . Let  $x$  be an element of  $S$  such that  $\forall x' \text{ in } S, l(x) \leq l(x')$  (lexicographically). Define  $L(x)$  to be  $j$ .

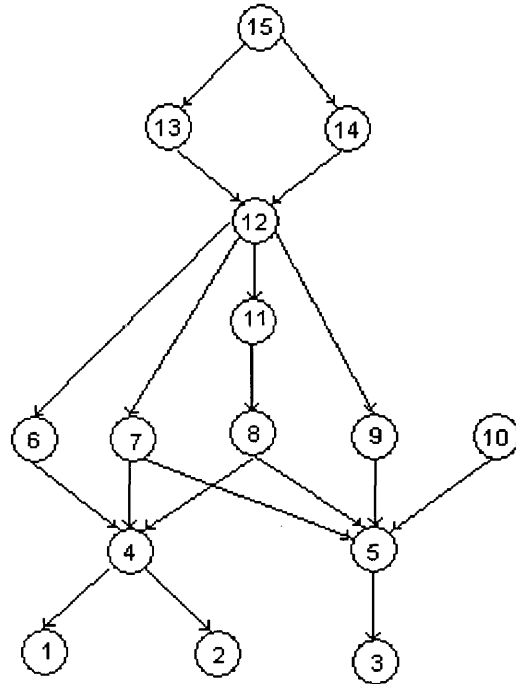
When all tasks have been labeled, use the list  $(T_n, T_{n-1} \dots T_1)$  where for all  $i, 1 \leq i \leq n, L(T_i) = i$  to schedule the tasks.

Since each task executes for one unit time, processors 1 and 2 both become available at the same time. It is assumed that processor 1 is scheduled before processor 2.

The algorithm basically assigns a label 1 to one of the terminal tasks. It then computes the set  $S$  of unassigned tasks with no unlabelled successors. It then selects an element from set  $S$ , which is lexicographically the smallest. The lexicographical elements are themselves formed from labels of the element's immediate successors. This element selected from the unassigned task set is assigned the next label and so on. When all the tasks have been labeled, the list of the labeled tasks is used as a priority queue to schedule the tasks.

### **Example 3.3**

To understand the algorithm let us examine the task graph given in Figure 3.6. The three terminal tasks are assigned the labels 1, 2, 3 respectively. At this point, the set  $S$  of unassigned tasks with no unlabeled successors becomes  $\{4,5\}$ . Also, it can be noticed that  $l(4) = \{2,1\}$  and  $l(5) = \{3\}$ . Since  $\{3\} > \{2,1\}$  (lexicographically), we assign labels 4, 5 to the tasks as given in the Figure 3.6. The algorithm continues until all tasks are labeled. The number within each node in Figure 3.6 indicates its label. Task 15 with the highest label is scheduled first on processor 1, and then task 10 is scheduled on processor 2. After task 15 and 10 are complete, the only ready tasks are 13 and 14. Note that a task is called ready when all of its predecessors have been executed. Figure 3.7 shows the output schedule of the task graph in Figure 3.6 on two processors using Algorithm 2.1



**Figure 3.6. A task graph.**

P1	P2
15	10
14	13
12	
11	9
8	7
6	5
4	3
2	1

**Figure 3.7. A 2-processor schedule for the task graph given in Figure 3.6**

### 3.3 List Scheduling 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. One may question how we can restrict a parallel computer to have only two processors in the era of massively parallel architecture with hundreds of processors, or how we can neglect the effect of communication delay in distributed-memory systems. To provide useful solutions to the scheduling problem, restrictions on the parallel program and target machine must be relaxed. Recent research in this area has emphasized heuristic approaches. Now, what is a heuristic? 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.

One class of scheduling heuristics in which many schedulers are classified is list scheduling. In list scheduling each task is assigned a priority, then a list of tasks is constructed in decreasing priority order. 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. The schedulers in this class differ in the way they assign priorities to tasks.

### 3.3.1 List Scheduling

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 tasks contend for processors, the selection of tasks to be immediately processed is done on the basis of priority with the higher-priority tasks being assigned processors first. If there is more than one task of a given priority, ties are broken randomly. Algorithm 3.4 presents a generic procedure of list scheduling.

#### Algorithm 3.4

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.

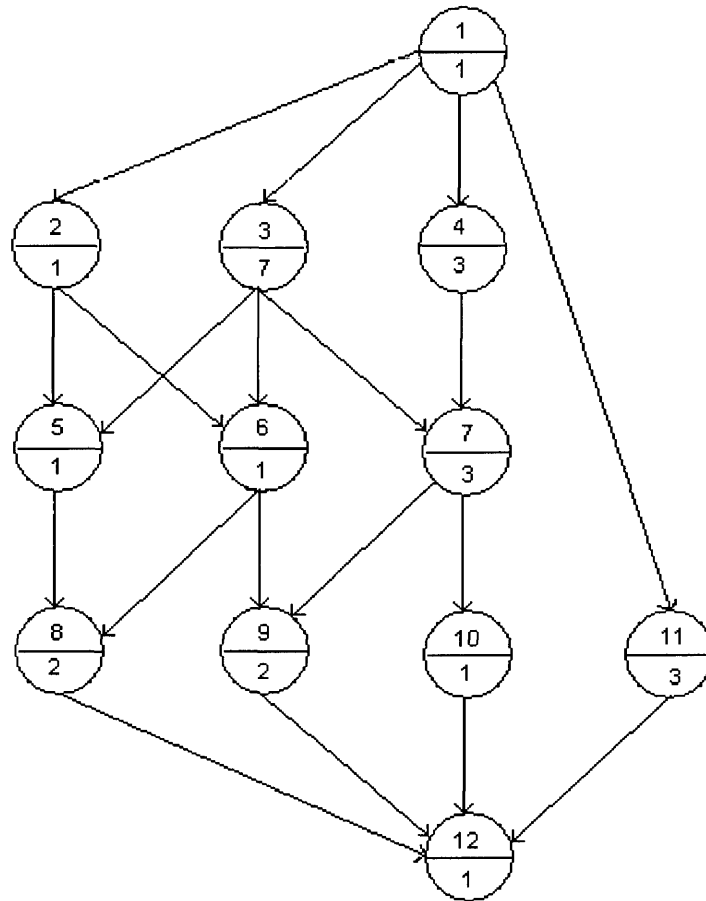
Priority assignment results in different schedules because nodes are selected in a different order. The level and co-level of a task are two examples of task priority.

#### Example 3.4

Consider the task graph given in Figure 3.8. The number given in the upper portion of a node is its label, while the number given in the lower portion is the execution time, which will also be referred to as task size. Note that communication is not considered here; this



is why there are no weights on the edges of the task graph. Figure 3.9 shows the levels and co-levels of the tasks of the task graph given in Figure 3.8.



**Figure 3.8. A task graph consisting of twelve tasks.**

Task	1	2	3	4	5	6	7	8	9	10	11	12
Level	14	5	13	9	4	4	6	3	3	2	4	1
Co-level	1	2	8	4	9	9	11	11	13	12	4	14

**Figure 3.9 The level and co-level of tasks in task graph of Figure 3.8**

## 4 Proposed Solution

In this section, we will study the proposed maximal chain scheduling heuristic. The maximal chain scheduling heuristic algorithm can be treated as an algorithm, which divides the  $n$  processor scheduling problem into an  $(n-1)$  processors + 1 processor scheduling problem. The maximal chain tasks are scheduled on 1 processor (P1) and the remaining tasks from the task graph are processed on the remaining  $(n-1)$  processors. This is done recursively until we reach 2 processors for which we have well known optimal algorithms that we can apply as we studied in the previous chapter.

### 4.1 Motivation:

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 one of the special cases. We also took advantage of the maximal clique, because it gave us the basics that those tasks, which participated in the maximal chain had to be performed in that precedence and hence the schedule length has to be at least the length of the maximal clique. After that it was just going to be a matter of merging the maximal clique with the schedule for the remaining tasks. The merging was not as simple as we hoped it would be, we had to check for violations and adjust the schedule to remove these violations. As a final pass, we ran the final schedule against an optimizer/compacting routine, which checks every idle time slot, it goes through every time slot and moves appropriate tasks to this idle time slot, only if it does that violate the precedence relationships for that task and graph.

As part of this thesis, we will implement another, well known heuristic. We have chosen to implement the List scheduling heuristic. This will enable us to run the task graph against different heuristics so that results can be compared with the proposed heuristic. As part of this thesis, a graphical user program will be written, the graphical program will enable the user to create a task graph and then run the n-processor scheduling algorithm on it. One will be able to compare the results with other well-known heuristics.

Our model of the problem is deterministic in the sense that all information governing the scheduling decisions are assumed to be known in advance. In particular, the task graph representing the parallel program and the target machine is assumed to be available before the program starts execution.

## 4.2 Program and Machine Models

In this chapter, we assume a restricted version of the general model introduced in chapter 2. Recall that the target machine model was represented as the 6-tuple  $(P, [P_{ij}], [S_i], [I_i], [B_i], [R_{ij}])$ , while the parallel program tasks were represented as the 4-tuple  $(T, <, [D_{ij}], [A_i])$ . The assumptions are as follows:

### Target Machine

- $P_{ij} = 1, 1 \leq i, j \leq m$ . (A fully connected system)
- $S_i = \text{Constant}, 1 \leq i \leq m$ . (Identical processing elements)
- $I_i = 0, 1 \leq i \leq m$ . (No startup costs for initiating a message)
- $B_i = 0, 1 \leq i \leq m$ . (No startup costs for initiating a task)

## Program Tasks

- $D_{ij} = 0, 1 \leq i, j \leq n$ . (No communication between task)

It follows that the execution time,  $T_{ij}$  of task  $i$  on processor  $j$ , will be the same for all processors and is equal to  $A_i / S_j$ . Assuming that  $S_j = 1$ , we can use  $A_i$  as the execution time of task  $i$ .

### 4.3 Maximal chain scheduling heuristic

As mentioned previously, there are many heuristic algorithms available for the scheduling problem. Some of them do perform well in some special cases. But in general they do not perform well, always. The proposed scheduling algorithm uses the maximal chain approach to solve the problem in a more general way. The main objective function is to minimize the time of completion of the tasks to be scheduled; in other words the shortest schedule. The proposed 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 famous Coffman and Graham algorithm. We present the maximal chain algorithm below in section 4.3.1. The algorithm itself consists of 3 different pieces, the maximal chain algorithm, the 2-processor algorithm (Coffman and Graham algorithm discussed in the previous chapter) and the Merge routine which not only merges the maximal chain and the  $(n-1)$  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.

### 4.3.1 The Algorithm

Given a Task Graph  $G (V, E)$  where  $V$  is the number of tasks and  $E$  is the number of edges between the nodes. Also  $N$  is the number of processors.

**Step 1:** If  $N = 2$  go to step 3.

- a) Given the Graph  $G (V, E)$ , Assign a priority to each task in the task graph using labels.
- b) Given the Graph  $G (V, E)$ , Find the Maximal Chain  $M$  for this task graph.
- c) Generate the sub-graph  $G_s = G - M$
- d) Repeat Step 1, with  $G = G_s$  and  $N = N - 1$ ;
- e) Merge  $(M, L)$  where  $M$  is the Maximal chain and  $L$  is the schedule for graph  $G_s$  for  $N$  processors. Assign the Maximal chain tasks to be processed by processor 1 and the remaining schedule to be processed by processors 2 to  $N$ . The resulting schedule  $S = M \cup L$ .
- f) Check the schedule for feasibility; this is done by making sure that the schedule does not have any violations. If we have violations, we move the entire chain down the appropriate time slots so that the task precedence rules are maintained and the violations no longer exist.

**Step 2:**

Given task graph  $G'$  and  $N' = 2$ . Apply the Coffman and Graham algorithm for 2 processors, which is as follows:

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

- b) Let labels  $1, 2, \dots, j-1$  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  $\{L(y_1), L(y_2), \dots, L(y_k)\}$ . Let  $x$  be an element of  $S$  such that  $\forall x'$  in  $S$ ,  $l(x) \leq l(x')$  (lexicographically). Define  $L(x)$  to be  $j$ .

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

As mentioned earlier, the maximal chain scheduling heuristic that we propose 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  $(n-1)$  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.

The approach to the maximal chain scheduling heuristic algorithm is to assign labels giving priority to tasks, and then a list for scheduling the task graph is constructed from the labels. Labels from the set  $\{1, 2, \dots, n\}$  are assigned to each task in the task graph by the function  $L(*)$  as explained in the following Algorithm 4.1.

**Algorithm 4.1.**

1. Assign the number 1 to one of the terminal tasks.
2. Let labels  $1, 2, \dots, j-1$  be 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  $\{L(y_1), L(y_2), \dots, L(y_k)\}$ . Let  $x$  be an element of  $S$  such that  $\forall x'$  in  $S$ ,  $l(x) \leq l(x')$  (lexicographically). Define  $L(x)$  to be  $j$ .

Once we have assigned a priority to all the tasks, generate the maximal chain for the task graph. This is done as explained in Algorithm 4.2.

**Algorithm 4.2.**

1. Let Maximal chain be  $M = \{\text{null}\}$ ,
2. Pick the task  $T_i$  with the highest label. (This will be a task which will have no predecessors.). The maximal chain  $M = M \cup \{T_i\}$ .
3. 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.

Once we break down the problem into  $1 + (n-1)$  processors, eventually we will reach 2-processors, for which we use the optimal Coffman and Graham algorithm presented in the previous chapter. Now given the maximal chain and the  $(n-1)$  processor schedule, all that needs to be done is to merge them maintaining the feasibility of the schedule based on the precedence of the tasks in the task graph and optimizes the schedule wherever possible. We present the Merge routine below in Algorithm 4.3.

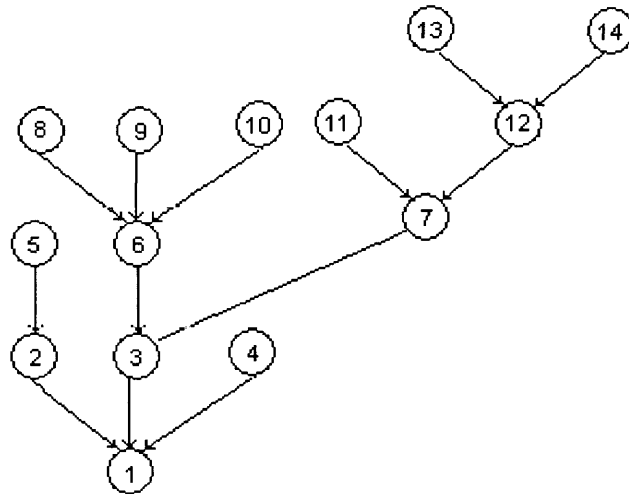
### Algorithm 4.3

1. Let  $M$  be the maximal chain and  $S$  be the  $(n-1)$  processor schedule.
2. Assign the tasks in the maximal chain to processor  $P_1$  and the tasks of the  $(n-1)$  processor schedule to processors  $P_2$  to  $P_n$ .
3. We examine every task from the beginning of the schedule. If a task violates any of the precedence rules  $T(i,j)$  of the task graph  $G$  then move that task  $T_i$  and the tasks below it on that processor  $P_x$  down the below the task that it violates  $T_j$ .  
Note that the tasks  $I$  and  $j$  will be on different processors, because within the processor they will already be satisfying the precedence rules.
4. After all the violations are removed, we examine each idle time slot on each of the processors  $P_1$  to  $P_n$  from the beginning in sequence. If we find an idle slot, we try to find a task below it which can be moved to the idle time slot without violating any of the precedence rules for the tasks  $T(i, j)$ .

#### 4.3.2 Example

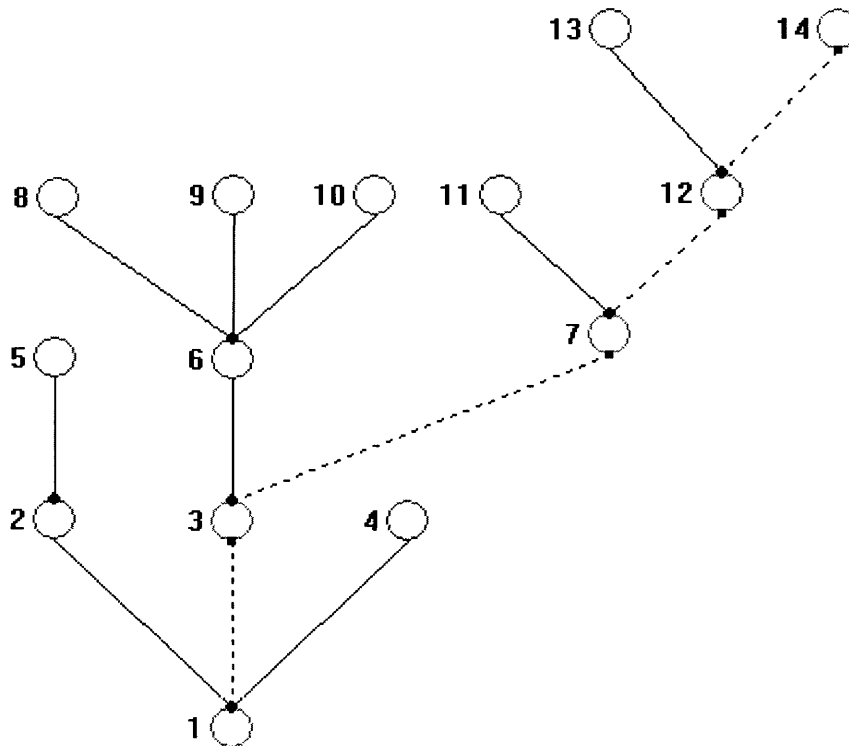
To understand the maximal chain scheduling heuristic algorithm, let us examine the task graph as shown in Figure 4.1. We first assign labels to each of the tasks in the task graph, which becomes the priority of that task. Based on the priority of the tasks we first find the maximal chain for that graph.





**Figure 4.1 A task graph.**

For the maximal chain (such as [14, 12, 7, 3, 1] in Figure 4.2) we take the remaining tasks (13, 11, 10, 9, 8, 6, 5, 4, 2, 1) and create a sub-graph  $G'$ .



**Figure 4.2: Maximal chain for task graph in Figure 4.1**

We perform the 2-processor scheduling algorithm (Coffman and Graham) on it. We then assign the tasks on the maximal chain to the third processor and then save the schedule length as shown in Figure 4.3. This is the first part of our Merge process. We now need to formally check this schedule for violations to make sure that the schedule is feasible.

P1	P2	P3
14	13	11
12	10	9
7	8	5
3	4	6
1	2	

**Figure 4.3: Simple Merge of maximal chain and 2-processor schedule.**

As we run this schedule through the feasibility check, from the top of the schedule to the bottom of the schedule, we find that task 3 cannot be run in parallel with task 6, because task 3 is a successor to task 6 or in other words task 6 precedes task 3, hence there is an obvious violation. We move all the tasks from 3 onwards one time slot down to fix this violation, which gives us the schedule as shown in Figure 4.4, which is an optimal feasible schedule.

P1	P2	P3
14	13	11
12	10	9
7	8	5
	4	6
3	2	
1		

**Figure 4.4: Schedule after Merge and Check Violations/Feasibility.**

In this example based on our algorithm, we will now have a feasible schedule. We will run this through our optimize routine, which will find the first idle time slot at T4 for processor P1. We will find that we can put task 2 which is below it in time slot 5 on processor P2 in this time slot without violating the precedence relationships of the task graph, hence our final schedule will be as shown in Figure 4.5. This does not improve the schedule length in this case, but will in certain other cases.

P1	P2	P3
14	13	11
12	10	9
7	8	5
2	4	6
3		
1		

**Figure 4.5: Schedule after Merge and Check Violation/Feasibility and Optimization.**

#### 4.4 Conjecture: All maximal chains

It was suggested, by Dr. Hesham Ali that since, the optimal algorithm presented by Coffman and Graham for 2-processor scheduling is truly based on identifying the maximal chain and then assigning the remaining tasks appropriately to get the two processor schedule. If we take this one step further, we can say that, finding all the maximal chains and performing our proposed maximal chain heuristic on it should yield us the minimum optimal schedule most of the times. Merging the maximal chain and the (n-1) processor schedule for all possible maximal chains in the task graph should provide us with the maximal chain that would give us the smallest /optimal schedule length. The crucial part of the algorithm is the merging of the maximal chain and the (n-1) processor schedule. Our merge technique may not be the best. We used the “All maximal chains” routine for testing purposes on graphs with small number of nodes. The “All maximal chains” algorithm is exponential. [AliH2001]

## **5 Package**

A graphical user program has been implemented as part of the thesis, this graphical program will enable the user to create a task graph and then run the n-processor scheduling algorithm on it, one will be able to compare the results with other well-known heuristics.

### **5.1 The Graph-It application**

The graphical package is called Graph-It and looks as shown in Figure 5.1 below. The package is a typical Windows MDI application, which has a “Title Bar” that displays the name of the package and the name of the graph file that the user is working with. It has a “Status Bar” which displays status information as well as some helpful text. The application provides the user with a “Tool Bar” which enables the user to draw the actual graph (i.e. Nodes and Edges) on the “Graph Client Area”.

The graphical user interface program will be a Multiple Document Interface program and will be based on Microsoft’s Doc/View architecture. One will be able to run the program as standalone or as an embedded server.

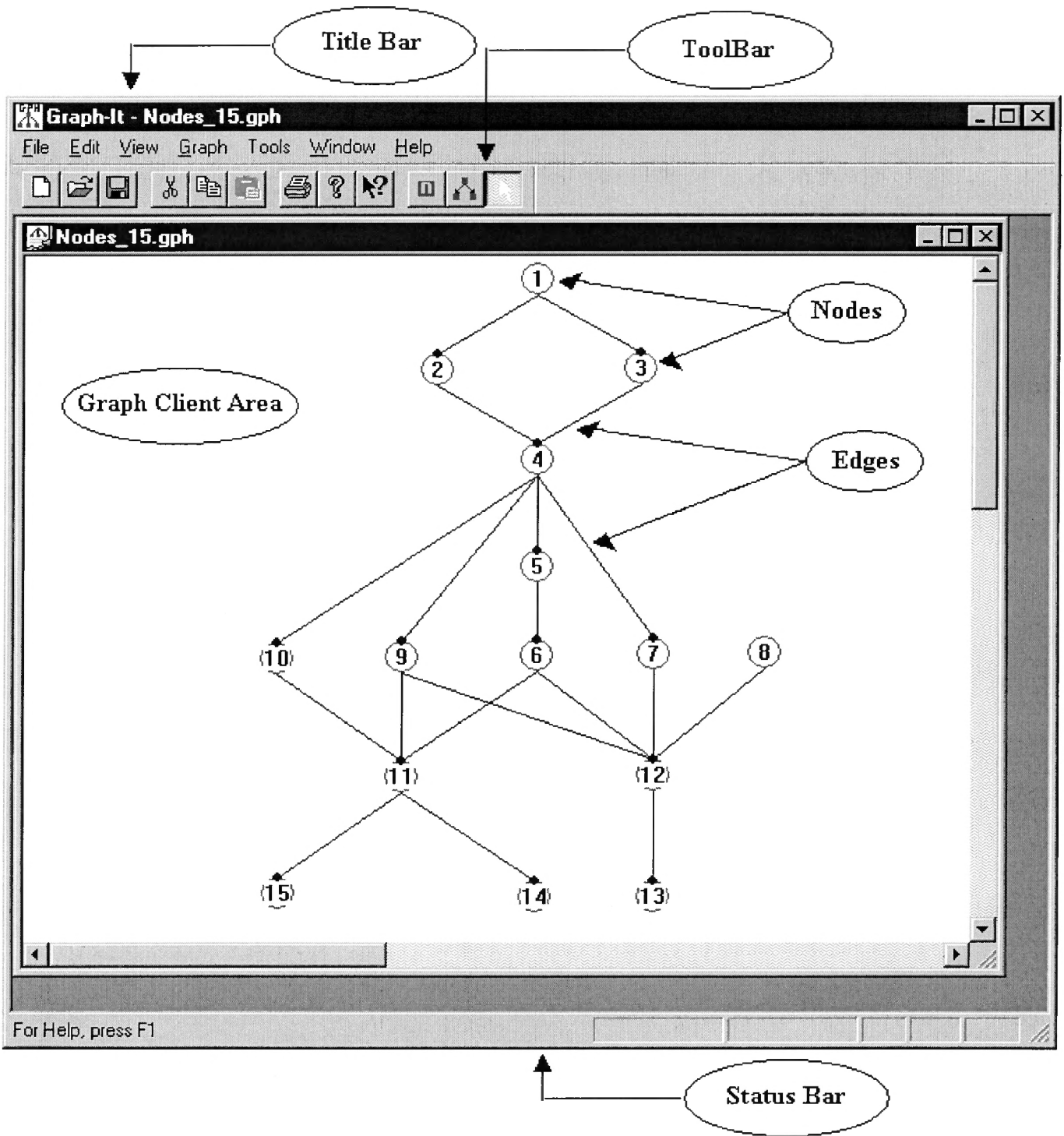


Figure 5.1 Graph-It graphical program

In order to draw a graph the user needs to go in the “Draw Mode”, clicking on the Node button or the Edge button will put the user in the “Draw Mode”. First the user needs to place some nodes on the “Graph Client Area” and then draw edges between the nodes by clicking on the “From Node” first and then dragging the mouse over to the “To Node”. This will draw a directed edge from one node to the other.

Once the Graph is drawn, the user can choose to run various graph algorithms on it (currently for this thesis this is restricted to scheduling algorithms). The application also provides the user with some user interface functions that the user can perform on the graph, some of which are listed below.

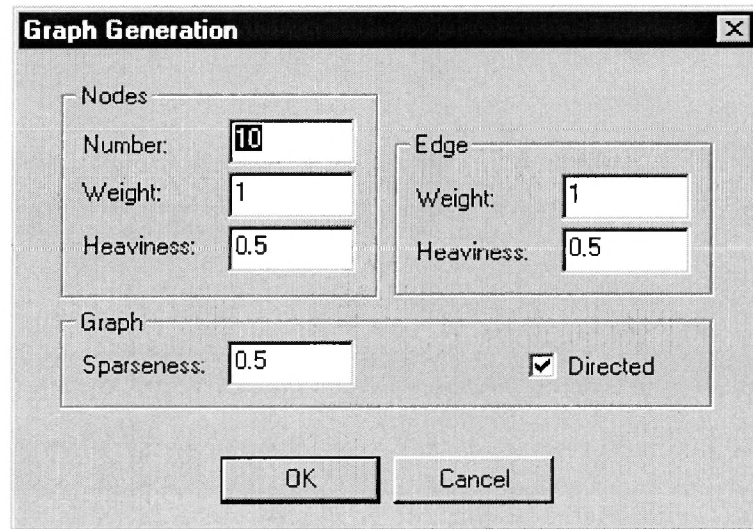
- 1) Manipulate the color of the nodes and edges.
- 2) Manipulate the shape of the nodes and edges.
- 3) Copy/Move/Delete certain nodes or edges or a sub-graph.
- 4) Rename the name assigned to nodes, edges and the graph.
- 5) View the graph as a matrix.

## **5.2 Random graph generator**

The application also provides a Random Graph Generation Algorithm under the Tools menu. This menu option opens a dialog box shown in Figure 5.2. This dialog box allows the user can key in the following parameters for the random graph.

- a) Number of nodes in the graph.
- b) Give a weight/heaviness factor to the nodes and edges
- c) Give a sparseness factor to the graph, which determines the density of the graph.

d) Make the graph either a directed or undirected graph.



**Figure 5.2 Random graph generation dialog box**

After the user specifies the parameters and clicks on the “OK” button, the user will be prompted to select a filename for the random graph. The random graph will then be stored in the user specified file.

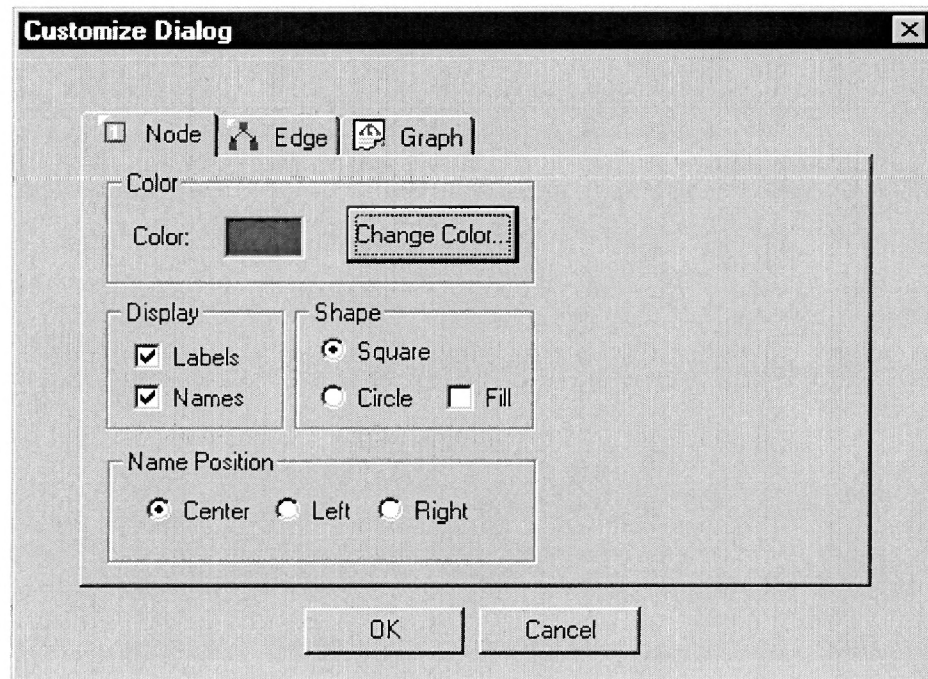
### 5.3 Application Customization

The graphical application also has a “Customize...” menu option, which will bring up the “Customize Dialog” as shown in Figure 5.3. This dialog box allows the user to choose application specific defaults, which will be applied to all the graphs that are created. The user can always override these defaults for a specific graph or a specific node or a specific edge. The following customizations are allowed.

- a) Default colors for nodes and edges.
- b) Default shape (Square, Circle, Filled-Square and Filled-Circle) for the nodes.
- c) Default display names/labels for nodes and edges.



- d) Default position (Center, Left and Right) to display the node names.
- e) Edge display flag to indicate whether the application should display edge names.



**Figure 5.3 Customize dialog box**

#### **5.4 Program specifications**

The program specifications for the graphical user interface are as follows:

- 1) The program is written under the Windows operating system (Windows 95, Windows 98 and Windows NT 4.0).
- 2) The program makes use of the Microsoft Foundation class library.
- 3) The program is written using MS Visual Studio VC++.
- 4) The program includes features such as Object Linking and Embedding.
- 5) The program is COM (Component Object Model) compliant.
- 6) The program was written on a Pentium Pro 180MHz machine, which had 4 GB of hard disk space and 64MB of RAM.

## 6 Implementation and Results

We ran various experiments 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 sparseness of the graph 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 the lower density will most likely take less scheduling time as compared with a graph of higher density.

We ran different experiments as explained below:

1) Experiment-1 was run on graphs with 25, 35, 40, 50, 60, 70, 80, 90, 100, 200, 300 and 400 nodes with densities varying from 0.1 to 0.8. Also it was noted that for graphs with about 35 nodes and a medium density of 0.4, 0.5 the optimal algorithm took too long to run (more than 2 hours). Hence the optimal algorithm could not be run on all the graphs especially those with higher number of nodes having medium to high density.

The naming convention used for the various graphs in this experiments is as follows GXXX\_Y, where XXX denotes the number of nodes in the graph and Y denotes the density/sparseness of the graph. So graph G200\_4 would have 200 nodes and a density of 0.4.

2) Experiment-2 was run on graphs with 25, 100, 200, 300, 400 nodes with densities varying from 0.1 to 0.8. We generated 80 graphs for each of the above-mentioned

number of nodes. 10 graphs were generated for each density/sparseness between 0.1 and 0.8, hence the 80 graphs. This experiment was conducted on a total of 400 graphs having high number of nodes. We divide the tasks graphs in this experiment into small graphs (25 nodes), medium graphs (100 – 200 nodes) and large graphs (300 – 400 nodes).

The naming convention used for the various graphs in this experiments-2 is as follows GXXX\_Y\_Z, where XXX denotes the number of nodes in the graph and Y denotes the density/sparseness of the graph and Z denotes the graph sequence. So graph G200\_4\_1 would have 200 nodes and a density of 0.4 and would be the first graph in that series.

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. The reason this is important is that this increases the density of the graphs generated.

The graphs used in the experiments are not transitively reduced graphs. We would also like to define the density of the graphs used in the experiments as the ratio of the number of edges  $|E|$  in the graph as a percentage to the maximum number of edges that the graph can have in our case  $(n*(n-1)) / 2$  (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 Experiment-1

Experiment-1 was run on graphs with 25, 35, 40, 50, 60, 70, 80, 90, 100, 200, 300 and 400 nodes with densities varying from 0.1 to 0.8. Also it was noted that for graphs with about 35 nodes and a medium density of 0.4, 0.5 the optimal algorithm took too long to run (more that 2 hours).

### 6.1.1 3-processor scheduling for task graphs (25 to 35 nodes)

Experiment for 3-processor scheduling using graphs with small number of nodes (25 to 35) with density from 0.1 to 0.5.

Graphs	Optimal	Maximal	List
G25_1.txt	9	9	10
G25_2.txt	9	9	9
G25_3.txt	11	11	11
G25_4.txt	12	12	12
G25_5.txt	16	18	18
G35_1.txt	12	12	12
G35_2.txt	12	13	12
G35_3.txt	20	20	20
G35_4.txt		20	20
G35_5.txt		21	20

### 6.1.2 3-processor scheduling for task graphs (40 to 90 nodes)

Experiment for 3-processor scheduling using graphs with medium number of nodes (40 to 90) with density from 0.1 to 0.8. Note that, the optimal “All maximal chains” algorithm was not run on this set of graphs.

Graphs	Maximal	List
G40_1.txt	13	14
G40_2.txt	16	17
G40_3.txt	19	19
G40_4.txt	18	18
G40_5.txt	26	26
G40_6.txt	27	27
G40_7.txt	30	30
G40_8.txt	34	34
G50_1.txt	20	19
G50_2.txt	21	20
G50_3.txt	20	21
G50_4.txt	26	26
G50_5.txt	28	28
G50_6.txt	36	36
G50_7.txt	37	37
G50_8.txt	42	42
G60_1.txt	19	21
G60_2.txt	22	24
G60_3.txt	22	23
G60_4.txt	26	25
G60_5.txt	37	37
G60_6.txt	46	46
G60_7.txt	41	41
G60_8.txt	50	50

G70_1.txt	23	24
G70_2.txt	22	25
G70_3.txt	29	30
G70_4.txt	35	35
G70_5.txt	38	37
G70_6.txt	39	39
G70_7.txt	52	52
G70_8.txt	55	55
G80_1.txt	27	28
G80_2.txt	31	32
G80_3.txt	33	33
G80_4.txt	38	38
G80_5.txt	53	53
G80_6.txt	52	52
G80_7.txt	54	54
G80_8.txt	64	64
G90_1.txt	30	31
G90_2.txt	35	33
G90_3.txt	34	35
G90_4.txt	43	42
G90_5.txt	53	53
G90_6.txt	62	62
G90_7.txt	67	67
G90_8.txt	71	71

### 6.1.3 3-processor scheduling for task graphs (100 to 400 nodes)

Experiment for 3-processor scheduling using graphs with large number of nodes (100 to 400) with density from 0.1 to 0.8. Note that, the optimal “All maximal chains” algorithm was not run on this set of graphs.

Graphs	Maximal	List
G100_1.txt	33	35
G100_2.txt	33	35
G100_3.txt	40	43
G100_4.txt	52	51
G100_5.txt	58	58
G100_6.txt	64	64
G100_7.txt	77	77
G100_8.txt	84	84
G200_1.txt	67	68
G200_2.txt	67	73
G200_3.txt	90	91
G200_4.txt	97	97
G200_5.txt	108	108
G200_6.txt	134	134
G200_7.txt	154	154
G200_8.txt	166	166
G300_1.txt	100	101
G300_2.txt	103	108
G300_3.txt	126	129
G300_4.txt	152	152
G300_5.txt	172	172
G300_6.txt	207	207
G300_7.txt	225	225
G300_8.txt	254	254
G400_1.txt	134	135
G400_2.txt	142	147
G400_3.txt	165	166
G400_4.txt	201	201
G400_5.txt	231	231
G400_6.txt	263	263
G400_7.txt	288	288
G400_8.txt	323	323

## 6.2 Experiment-2

Experiment-2 was run on graphs with 25, 100, 200, 300, 400 nodes with densities varying from 0.1 to 0.8. We generated 80 graphs for each of the above-mentioned number of nodes. 10 graphs were generated for each density/sparseness between 0.1 and 0.8, hence the 80 graphs. This experiment was conducted on a total of 400 graphs having high number of nodes.

The naming convention used for the various graphs in this experiments-2 is as follows GXXX\_Y\_Z, where XXX denotes the number of nodes in the graph and Y denotes the density/sparseness of the graph and Z denotes the graph sequence. So graph G200\_4\_1 would have 200 nodes and a density of 0.4 and would be the first graph in that series.

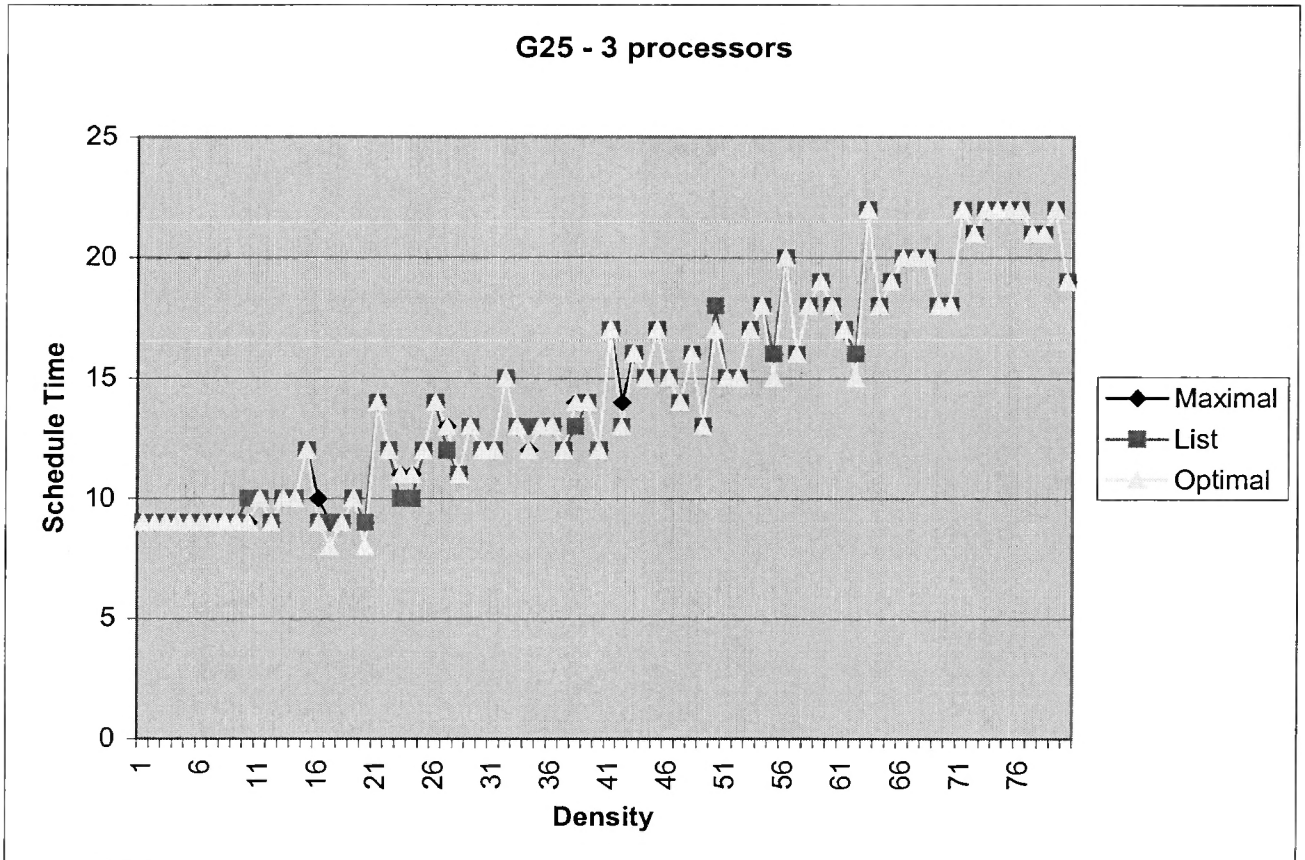
### 6.2.1 Small task graphs (25 nodes) on 3,4 and 5 processors

Experiment for 3-processor scheduling using graphs with 25 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List	Optimal
G25_1_1.txt	9	9	9
G25_1_2.txt	9	9	9
G25_1_3.txt	9	9	9
G25_1_4.txt	9	9	9
G25_1_5.txt	9	9	9
G25_1_6.txt	9	9	9
G25_1_7.txt	9	9	9
G25_1_8.txt	9	9	9
G25_1_9.txt	9	9	9
G25_1_10.txt	9	10	9
G25_2_1.txt	10	10	10
G25_2_2.txt	9	9	9
G25_2_3.txt	10	10	10
G25_2_4.txt	10	10	10
G25_2_5.txt	12	12	12
G25_2_6.txt	10	9	9
G25_2_7.txt	9	9	8
G25_2_8.txt	9	9	9
G25_2_9.txt	10	10	10
G25_2_10.txt	9	9	8
G25_3_1.txt	14	14	14
G25_3_2.txt	12	12	12
G25_3_3.txt	11	10	11
G25_3_4.txt	11	10	11
G25_3_5.txt	12	12	12
G25_3_6.txt	14	14	14
G25_3_7.txt	13	12	13
G25_3_8.txt	11	11	11
G25_3_9.txt	13	13	13
G25_3_10.txt	12	12	12
G25_4_1.txt	12	12	12
G25_4_2.txt	15	15	15
G25_4_3.txt	13	13	13
G25_4_4.txt	12	13	12
G25_4_5.txt	13	13	13
G25_4_6.txt	13	13	13
G25_4_7.txt	12	12	12
G25_4_8.txt	14	13	14
G25_4_9.txt	14	14	14
G25_4_10.txt	12	12	12

G25_5_1.txt	17	17	17
G25_5_2.txt	14	13	13
G25_5_3.txt	16	16	16
G25_5_4.txt	15	15	15
G25_5_5.txt	17	17	17
G25_5_6.txt	15	15	15
G25_5_7.txt	14	14	14
G25_5_8.txt	16	16	16
G25_5_9.txt	13	13	13
G25_5_10.txt	18	18	17
G25_6_1.txt	15	15	15
G25_6_2.txt	15	15	15
G25_6_3.txt	17	17	17
G25_6_4.txt	18	18	18
G25_6_5.txt	16	16	15
G25_6_6.txt	20	20	20
G25_6_7.txt	16	16	16
G25_6_8.txt	18	18	18
G25_6_9.txt	19	19	19
G25_6_10.txt	18	18	18
G25_7_1.txt	17	17	17
G25_7_2.txt	16	16	15
G25_7_3.txt	22	22	22
G25_7_4.txt	18	18	18
G25_7_5.txt	19	19	19
G25_7_6.txt	20	20	20
G25_7_7.txt	20	20	20
G25_7_8.txt	20	20	20
G25_7_9.txt	18	18	18
G25_7_10.txt	18	18	18
G25_8_1.txt	22	22	22
G25_8_2.txt	21	21	21
G25_8_3.txt	22	22	22
G25_8_4.txt	22	22	22
G25_8_5.txt	22	22	22
G25_8_6.txt	22	22	22
G25_8_7.txt	21	21	21
G25_8_8.txt	21	21	21
G25_8_9.txt	22	22	22
G25_8_10.txt	19	19	19





**Figure 6.1. Graphs with 25 nodes on 3 processors**

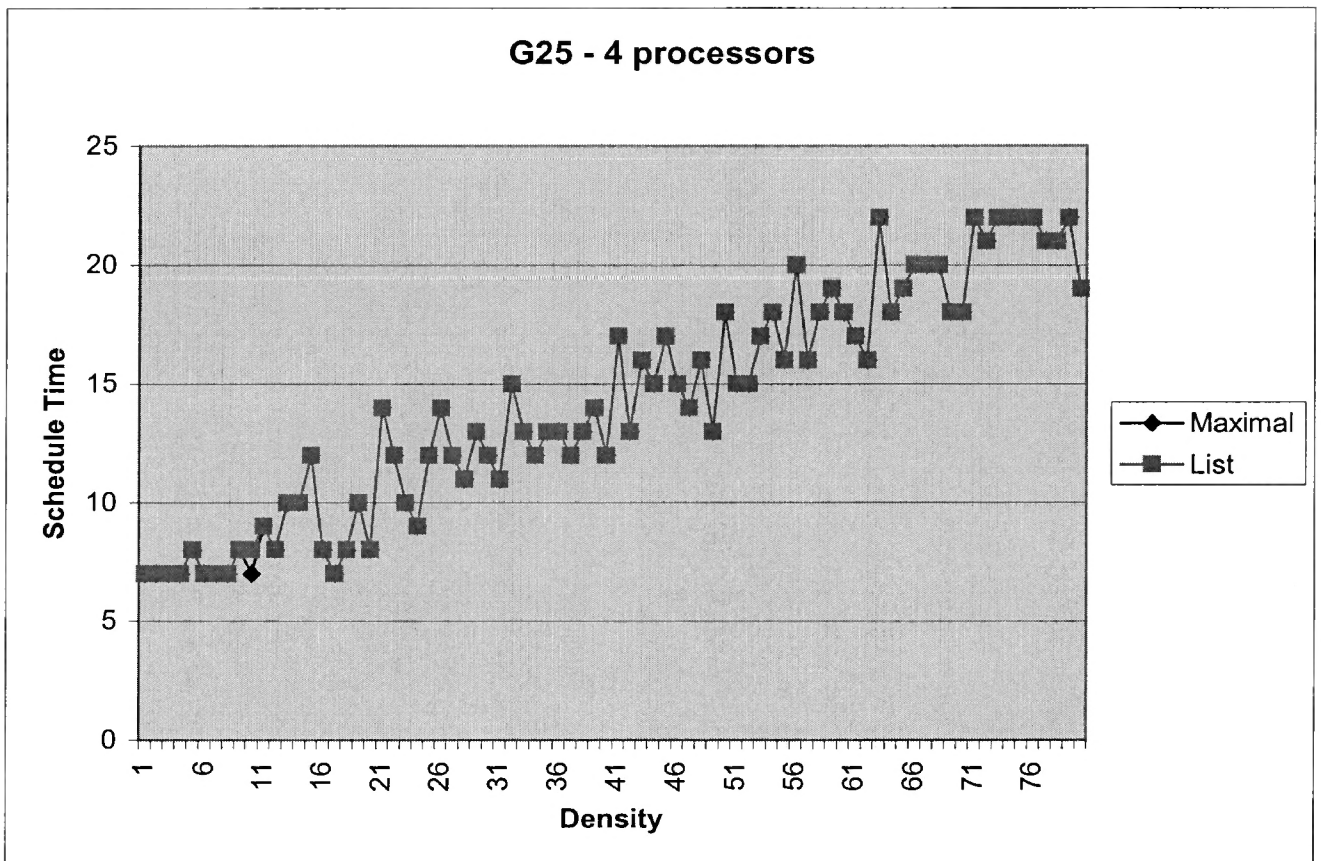
From the above experiments and results we can draw the following conclusions:

- 1) It can be concluded that the optimal all chain scheduling heuristic performs better or equal to than the List scheduling and the Maximal chain scheduling heuristic in most cases.
- 2) The optimal all chain algorithm, Maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance when the density/sparseness of the graphs is greater than 0.4 for most of the cases.

Experiment for 4-processor scheduling using graphs with 25 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List
G25_1_1.txt	7	7
G25_1_2.txt	7	7
G25_1_3.txt	7	7
G25_1_4.txt	7	7
G25_1_5.txt	8	8
G25_1_6.txt	7	7
G25_1_7.txt	7	7
G25_1_8.txt	7	7
G25_1_9.txt	8	8
G25_1_10.txt	7	8
G25_2_1.txt	9	9
G25_2_2.txt	8	8
G25_2_3.txt	10	10
G25_2_4.txt	10	10
G25_2_5.txt	12	12
G25_2_6.txt	8	8
G25_2_7.txt	7	7
G25_2_8.txt	8	8
G25_2_9.txt	10	10
G25_2_10.txt	8	8
G25_3_1.txt	14	14
G25_3_2.txt	12	12
G25_3_3.txt	10	10
G25_3_4.txt	9	9
G25_3_5.txt	12	12
G25_3_6.txt	14	14
G25_3_7.txt	12	12
G25_3_8.txt	11	11
G25_3_9.txt	13	13
G25_3_10.txt	12	12
G25_4_1.txt	11	11
G25_4_2.txt	15	15
G25_4_3.txt	13	13
G25_4_4.txt	12	12
G25_4_5.txt	13	13
G25_4_6.txt	13	13
G25_4_7.txt	12	12
G25_4_8.txt	13	13
G25_4_9.txt	14	14
G25_4_10.txt	12	12
G25_5_1.txt	17	17

G25_5_2.txt	13	13
G25_5_3.txt	16	16
G25_5_4.txt	15	15
G25_5_5.txt	17	17
G25_5_6.txt	15	15
G25_5_7.txt	14	14
G25_5_8.txt	16	16
G25_5_9.txt	13	13
G25_5_10.txt	18	18
G25_6_1.txt	15	15
G25_6_2.txt	15	15
G25_6_3.txt	17	17
G25_6_4.txt	18	18
G25_6_5.txt	16	16
G25_6_6.txt	20	20
G25_6_7.txt	16	16
G25_6_8.txt	18	18
G25_6_9.txt	19	19
G25_6_10.txt	18	18
G25_7_1.txt	17	17
G25_7_2.txt	16	16
G25_7_3.txt	22	22
G25_7_4.txt	18	18
G25_7_5.txt	19	19
G25_7_6.txt	20	20
G25_7_7.txt	20	20
G25_7_8.txt	20	20
G25_7_9.txt	18	18
G25_7_10.txt	18	18
G25_8_1.txt	22	22
G25_8_2.txt	21	21
G25_8_3.txt	22	22
G25_8_4.txt	22	22
G25_8_5.txt	22	22
G25_8_6.txt	22	22
G25_8_7.txt	21	21
G25_8_8.txt	21	21
G25_8_9.txt	22	22
G25_8_10.txt	19	19



**Figure 6.2. Graphs with 25 nodes on 4 processors**

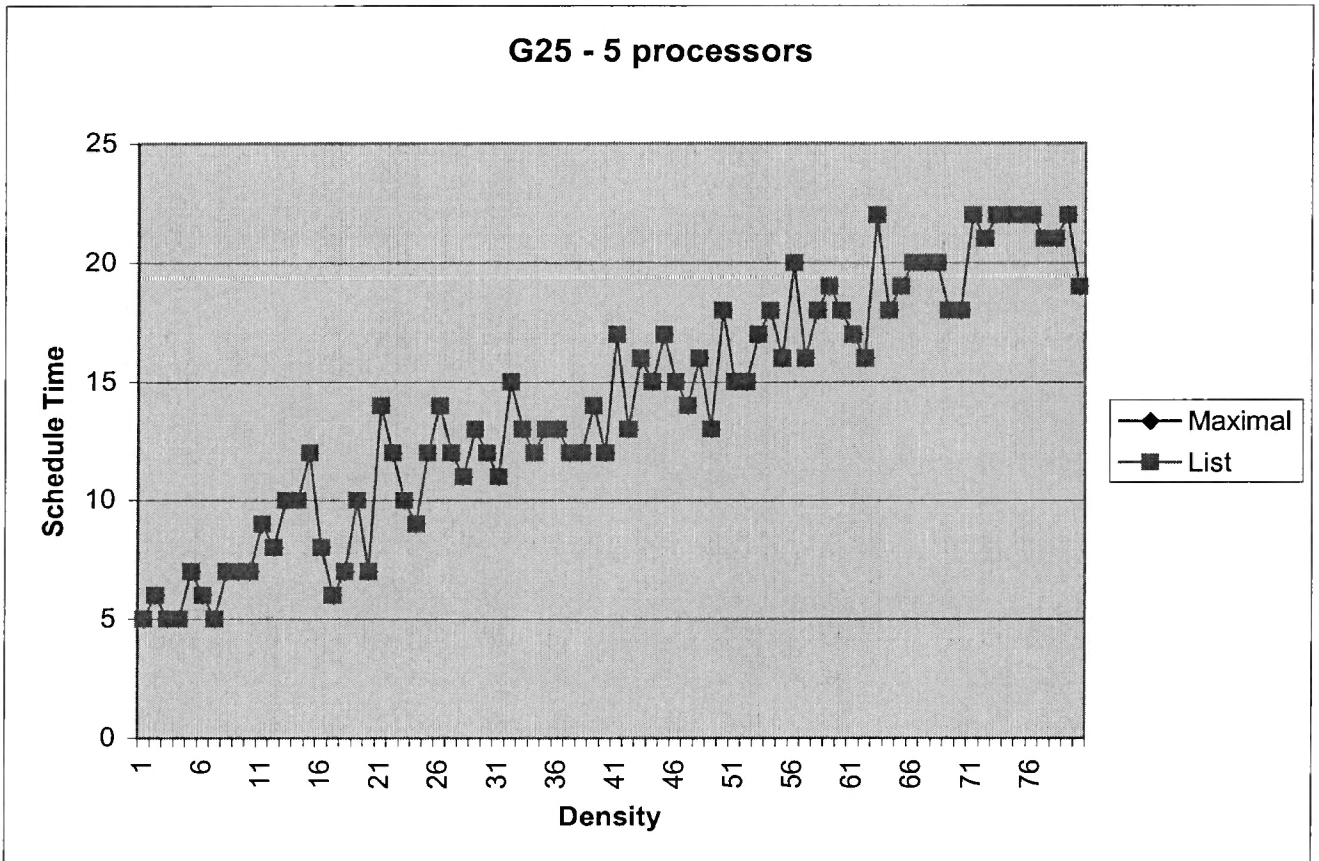
From the above experiments and results we can draw the following conclusions:

- 1) The maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance for most of the cases.

Experiment for 5-processor scheduling using graphs with 25 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List
G25_1_1.txt	5	5
G25_1_2.txt	6	6
G25_1_3.txt	5	5
G25_1_4.txt	5	5
G25_1_5.txt	7	7
G25_1_6.txt	6	6
G25_1_7.txt	5	5
G25_1_8.txt	7	7
G25_1_9.txt	7	7
G25_1_10.txt	7	7
G25_2_1.txt	9	9
G25_2_2.txt	8	8
G25_2_3.txt	10	10
G25_2_4.txt	10	10
G25_2_5.txt	12	12
G25_2_6.txt	8	8
G25_2_7.txt	6	6
G25_2_8.txt	7	7
G25_2_9.txt	10	10
G25_2_10.txt	7	7
G25_3_1.txt	14	14
G25_3_2.txt	12	12
G25_3_3.txt	10	10
G25_3_4.txt	9	9
G25_3_5.txt	12	12
G25_3_6.txt	14	14
G25_3_7.txt	12	12
G25_3_8.txt	11	11
G25_3_9.txt	13	13
G25_3_10.txt	12	12
G25_4_1.txt	11	11
G25_4_2.txt	15	15
G25_4_3.txt	13	13
G25_4_4.txt	12	12
G25_4_5.txt	13	13
G25_4_6.txt	13	13
G25_4_7.txt	12	12
G25_4_8.txt	12	12
G25_4_9.txt	14	14
G25_4_10.txt	12	12
G25_5_1.txt	17	17

G25_5_2.txt	13	13
G25_5_3.txt	16	16
G25_5_4.txt	15	15
G25_5_5.txt	17	17
G25_5_6.txt	15	15
G25_5_7.txt	14	14
G25_5_8.txt	16	16
G25_5_9.txt	13	13
G25_5_10.txt	18	18
G25_6_1.txt	15	15
G25_6_2.txt	15	15
G25_6_3.txt	17	17
G25_6_4.txt	18	18
G25_6_5.txt	16	16
G25_6_6.txt	20	20
G25_6_7.txt	16	16
G25_6_8.txt	18	18
G25_6_9.txt	19	19
G25_6_10.txt	18	18
G25_7_1.txt	17	17
G25_7_2.txt	16	16
G25_7_3.txt	22	22
G25_7_4.txt	18	18
G25_7_5.txt	19	19
G25_7_6.txt	20	20
G25_7_7.txt	20	20
G25_7_8.txt	20	20
G25_7_9.txt	18	18
G25_7_10.txt	18	18
G25_8_1.txt	22	22
G25_8_2.txt	21	21
G25_8_3.txt	22	22
G25_8_4.txt	22	22
G25_8_5.txt	22	22
G25_8_6.txt	22	22
G25_8_7.txt	21	21
G25_8_8.txt	21	21
G25_8_9.txt	22	22
G25_8_10.txt	19	19



**Figure 6.3. Graphs with 25 nodes on 5 processors**

From the above experiments and results we can draw the following conclusions:

- 1) The maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance for most of the cases.

From the above experiments and results on small task graphs (25 node graphs) when run on 3, 4 and 5 processors we can draw the following overall conclusions:

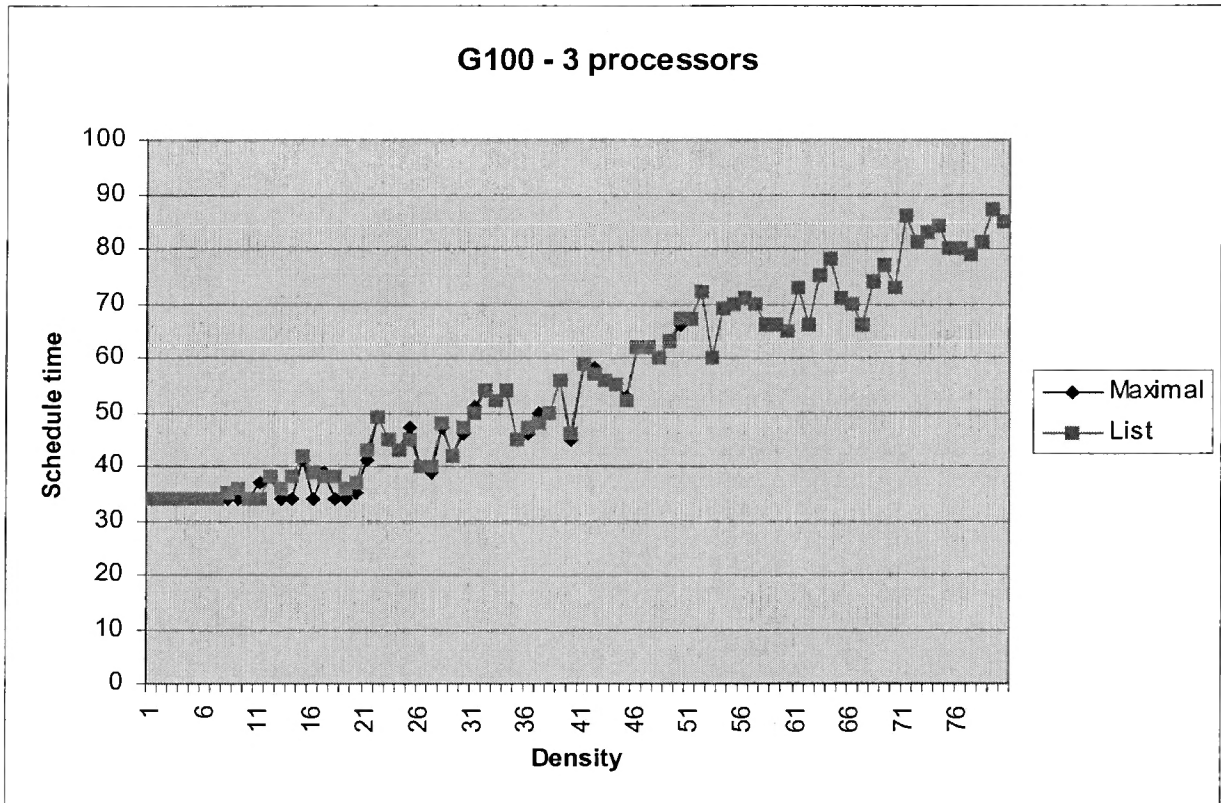
- 1) It can be concluded that as the number of processors increases for graphs between density of 0.1 and 0.2, the schedule length reduces slightly. The percentage of reduction in schedule length decreases as the number of processors increases from 3 to 5.
- 2) It can be concluded that as the number of processors increases for graphs with density greater than 0.2, the schedule length does not reduce at all for most graphs. This implies that as the density of the graphs increases, adding more processors will not help to reduce the schedule length.

## 6.2.2 Medium task graphs (100 –200 nodes) on 3,4 and 5 processors

Experiment for 3-processor scheduling using graphs with 100 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List
G100_1_1.txt	34	34
G100_1_2.txt	34	34
G100_1_3.txt	34	34
G100_1_4.txt	34	34
G100_1_5.txt	34	34
G100_1_6.txt	34	34
G100_1_7.txt	34	34
G100_1_8.txt	34	35
G100_1_9.txt	34	36
G100_1_10.txt	34	34
G100_2_1.txt	37	34
G100_2_2.txt	38	38
G100_2_3.txt	34	36
G100_2_4.txt	34	38
G100_2_5.txt	41	42
G100_2_6.txt	34	39
G100_2_7.txt	39	38
G100_2_8.txt	34	38
G100_2_9.txt	34	36
G100_2_10.txt	35	37
G100_3_1.txt	41	43
G100_3_2.txt	49	49
G100_3_3.txt	45	45
G100_3_4.txt	43	43
G100_3_5.txt	47	45
G100_3_6.txt	40	40
G100_3_7.txt	39	40
G100_3_8.txt	47	48
G100_3_9.txt	42	42
G100_3_10.txt	46	47
G100_4_1.txt	51	50
G100_4_2.txt	54	54
G100_4_3.txt	52	52
G100_4_4.txt	54	54
G100_4_5.txt	45	45
G100_4_6.txt	46	47
G100_4_7.txt	50	48
G100_4_8.txt	50	50
G100_4_9.txt	56	56
G100_4_10.txt	45	46

G100_5_1.txt	59	59
G100_5_2.txt	58	57
G100_5_3.txt	56	56
G100_5_4.txt	55	55
G100_5_5.txt	53	52
G100_5_6.txt	62	62
G100_5_7.txt	62	62
G100_5_8.txt	60	60
G100_5_9.txt	63	63
G100_5_10.txt	66	67
G100_6_1.txt	67	67
G100_6_2.txt	72	72
G100_6_3.txt	60	60
G100_6_4.txt	69	69
G100_6_5.txt	70	70
G100_6_6.txt	71	71
G100_6_7.txt	70	70
G100_6_8.txt	66	66
G100_6_9.txt	66	66
G100_6_10.txt	65	65
G100_7_1.txt	73	73
G100_7_2.txt	66	66
G100_7_3.txt	75	75
G100_7_4.txt	78	78
G100_7_5.txt	71	71
G100_7_6.txt	70	70
G100_7_7.txt	66	66
G100_7_8.txt	74	74
G100_7_9.txt	77	77
G100_7_10.txt	73	73
G100_8_1.txt	86	86
G100_8_2.txt	81	81
G100_8_3.txt	83	83
G100_8_4.txt	84	84
G100_8_5.txt	80	80
G100_8_6.txt	80	80
G100_8_7.txt	79	79
G100_8_8.txt	81	81
G100_8_9.txt	87	87
G100_8_10.txt	85	85



**Figure 6.4. Graphs with 100 nodes on 3 processors**

From the above experiments and results we can draw the following conclusions:

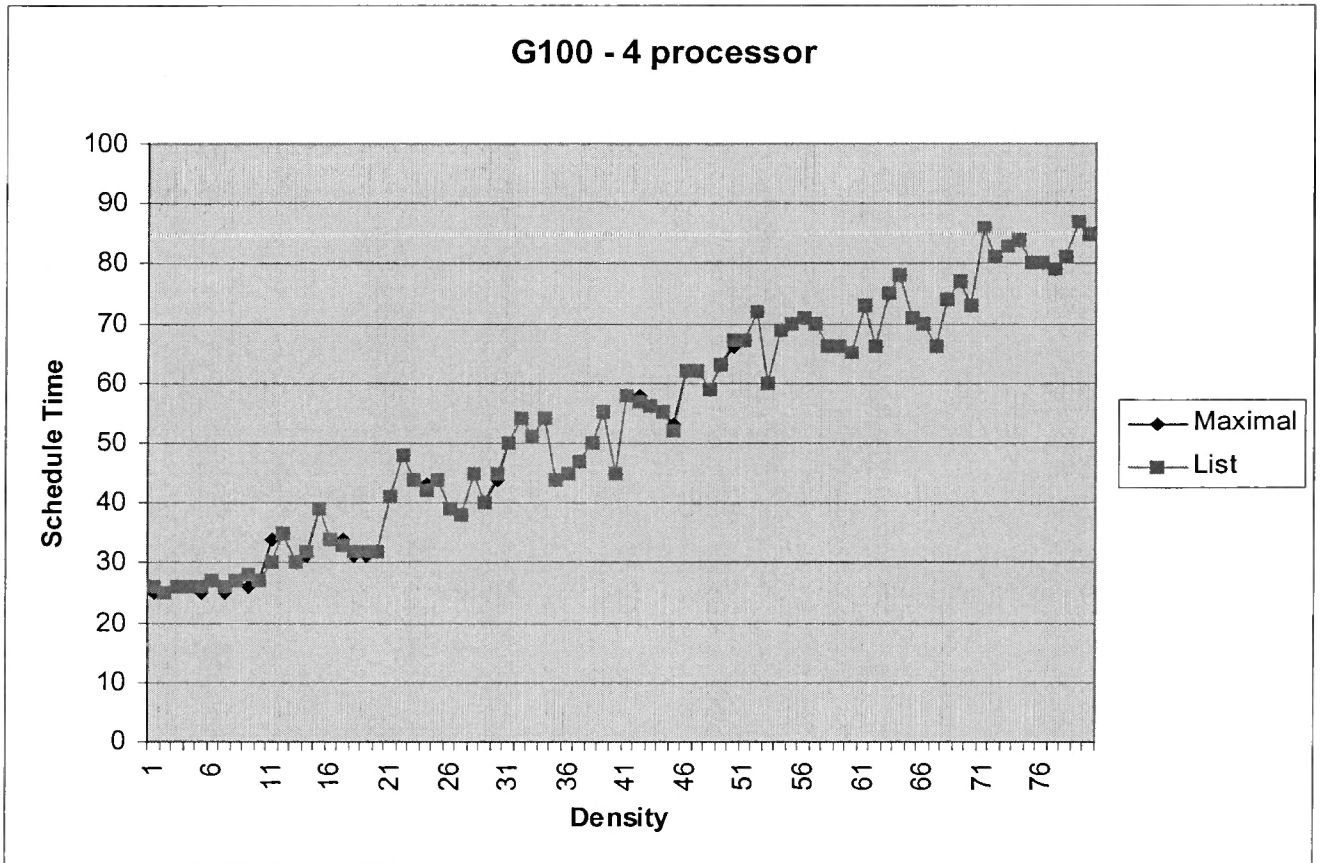
- 1) It can be concluded that the maximal chain scheduling heuristic performs slightly better than the List scheduling when the density/sparseness of the graphs is between 0.1 and 0.4 for most of the cases.
- 2) The maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance when the density/sparseness of the graphs is greater than 0.4 for most of the cases.



Experiment for 4-processor scheduling using graphs with 100 nodes of density from 0.1 to 0.8.

Graphs	Maximal List	
G100_1_1.txt	25	26
G100_1_2.txt	25	25
G100_1_3.txt	26	26
G100_1_4.txt	26	26
G100_1_5.txt	25	26
G100_1_6.txt	27	27
G100_1_7.txt	25	26
G100_1_8.txt	27	27
G100_1_9.txt	26	28
G100_1_10.txt	27	27
G100_2_1.txt	34	30
G100_2_2.txt	35	35
G100_2_3.txt	30	30
G100_2_4.txt	31	32
G100_2_5.txt	39	39
G100_2_6.txt	34	34
G100_2_7.txt	34	33
G100_2_8.txt	31	32
G100_2_9.txt	31	32
G100_2_10.txt	32	32
G100_3_1.txt	41	41
G100_3_2.txt	48	48
G100_3_3.txt	44	44
G100_3_4.txt	43	42
G100_3_5.txt	44	44
G100_3_6.txt	39	39
G100_3_7.txt	38	38
G100_3_8.txt	45	45
G100_3_9.txt	40	40
G100_3_10.txt	44	45
G100_4_1.txt	50	50
G100_4_2.txt	54	54
G100_4_3.txt	51	51
G100_4_4.txt	54	54
G100_4_5.txt	44	44
G100_4_6.txt	45	45
G100_4_7.txt	17	17
G100_4_8.txt	50	50
G100_4_9.txt	55	55
G100_4_10.txt	45	45
G100_5_1.txt	58	58

G100_5_2.txt	58	57
G100_5_3.txt	56	56
G100_5_4.txt	55	55
G100_5_5.txt	53	52
G100_5_6.txt	62	62
G100_5_7.txt	62	62
G100_5_8.txt	59	59
G100_5_9.txt	63	63
G100_5_10.txt	66	67
G100_6_1.txt	67	67
G100_6_2.txt	72	72
G100_6_3.txt	60	60
G100_6_4.txt	69	69
G100_6_5.txt	70	70
G100_6_6.txt	71	71
G100_6_7.txt	70	70
G100_6_8.txt	66	66
G100_6_9.txt	66	66
G100_6_10.txt	65	65
G100_7_1.txt	73	73
G100_7_2.txt	66	66
G100_7_3.txt	75	75
G100_7_4.txt	78	78
G100_7_5.txt	71	71
G100_7_6.txt	70	70
G100_7_7.txt	66	66
G100_7_8.txt	74	74
G100_7_9.txt	77	77
G100_7_10.txt	73	73
G100_8_1.txt	86	86
G100_8_2.txt	81	81
G100_8_3.txt	83	83
G100_8_4.txt	84	84
G100_8_5.txt	80	80
G100_8_6.txt	80	80
G100_8_7.txt	79	79
G100_8_8.txt	81	81
G100_8_9.txt	87	87
G100_8_10.txt	85	85



**Figure 6.5. Graphs with 100 nodes on 4 processors**

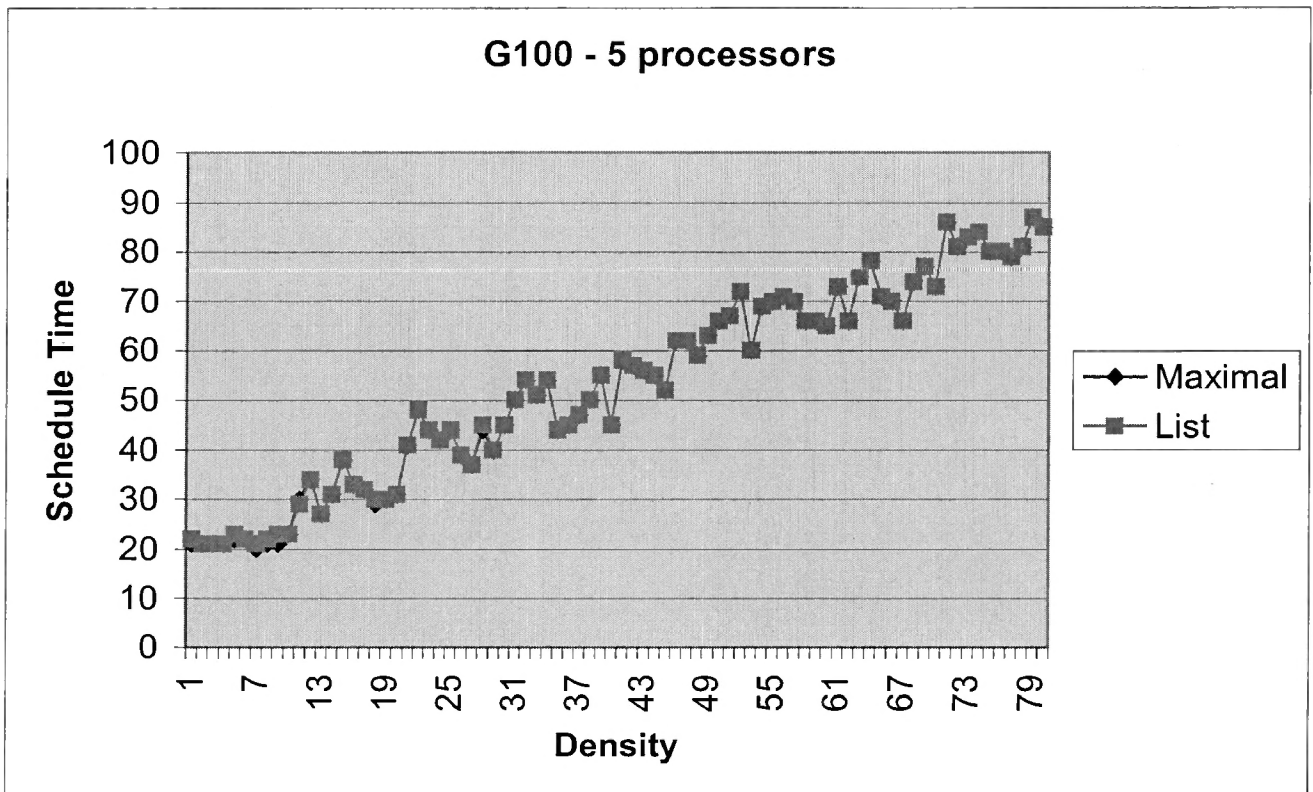
From the above experiments and results we can draw the following conclusions:

- 1) It can be concluded that the maximal chain scheduling heuristic performs slightly better than the List scheduling when the density/sparseness of the graphs is between 0.1 and 0.3 for most of the cases.
- 2) The maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance when the density/sparseness of the graphs is greater than 0.3 for most of the cases.

Experiment for 5-processor scheduling using graphs with 100 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List
G100_1_1.txt	21	22
G100_1_2.txt	21	21
G100_1_3.txt	21	21
G100_1_4.txt	21	21
G100_1_5.txt	22	23
G100_1_6.txt	22	22
G100_1_7.txt	20	21
G100_1_8.txt	21	22
G100_1_9.txt	21	23
G100_1_10.txt	23	23
G100_2_1.txt	30	29
G100_2_2.txt	34	34
G100_2_3.txt	27	27
G100_2_4.txt	31	31
G100_2_5.txt	38	38
G100_2_6.txt	33	33
G100_2_7.txt	32	32
G100_2_8.txt	29	30
G100_2_9.txt	30	30
G100_2_10.txt	31	31
G100_3_1.txt	41	41
G100_3_2.txt	48	48
G100_3_3.txt	44	44
G100_3_4.txt	42	42
G100_3_5.txt	44	44
G100_3_6.txt	39	39
G100_3_7.txt	37	37
G100_3_8.txt	44	45
G100_3_9.txt	40	40
G100_3_10.txt	45	45
G100_4_1.txt	50	50
G100_4_2.txt	54	54
G100_4_3.txt	51	51
G100_4_4.txt	54	54
G100_4_5.txt	44	44
G100_4_6.txt	45	45
G100_4_7.txt	47	47
G100_4_8.txt	50	50
G100_4_9.txt	55	55
G100_4_10.txt	45	45
G100_5_1.txt	58	58

G100_5_2.txt	57	57
G100_5_3.txt	56	56
G100_5_4.txt	55	55
G100_5_5.txt	52	52
G100_5_6.txt	62	62
G100_5_7.txt	62	62
G100_5_8.txt	59	59
G100_5_9.txt	63	63
G100_5_10.txt	66	66
G100_6_1.txt	67	67
G100_6_2.txt	72	72
G100_6_3.txt	60	60
G100_6_4.txt	69	69
G100_6_5.txt	70	70
G100_6_6.txt	71	71
G100_6_7.txt	70	70
G100_6_8.txt	66	66
G100_6_9.txt	66	66
G100_6_10.txt	65	65
G100_7_1.txt	73	73
G100_7_2.txt	66	66
G100_7_3.txt	75	75
G100_7_4.txt	78	78
G100_7_5.txt	71	71
G100_7_6.txt	70	70
G100_7_7.txt	66	66
G100_7_8.txt	74	74
G100_7_9.txt	77	77
G100_7_10.txt	73	73
G100_8_1.txt	86	86
G100_8_2.txt	81	81
G100_8_3.txt	83	83
G100_8_4.txt	84	84
G100_8_5.txt	80	80
G100_8_6.txt	80	80
G100_8_7.txt	79	79
G100_8_8.txt	81	81
G100_8_9.txt	87	87
G100_8_10.txt	85	85



**Figure 6.6. Graphs with 100 nodes on 5 processors**

From the above experiments and results we can draw the following conclusions:

- 1) It can be concluded that the maximal chain scheduling heuristic performs slightly better than the List scheduling when the density/sparseness of the graphs is between 0.1 and 0.2 for most of the cases.
- 2) The maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance when the density/sparseness of the graphs is greater than 0.2 for most of the cases.

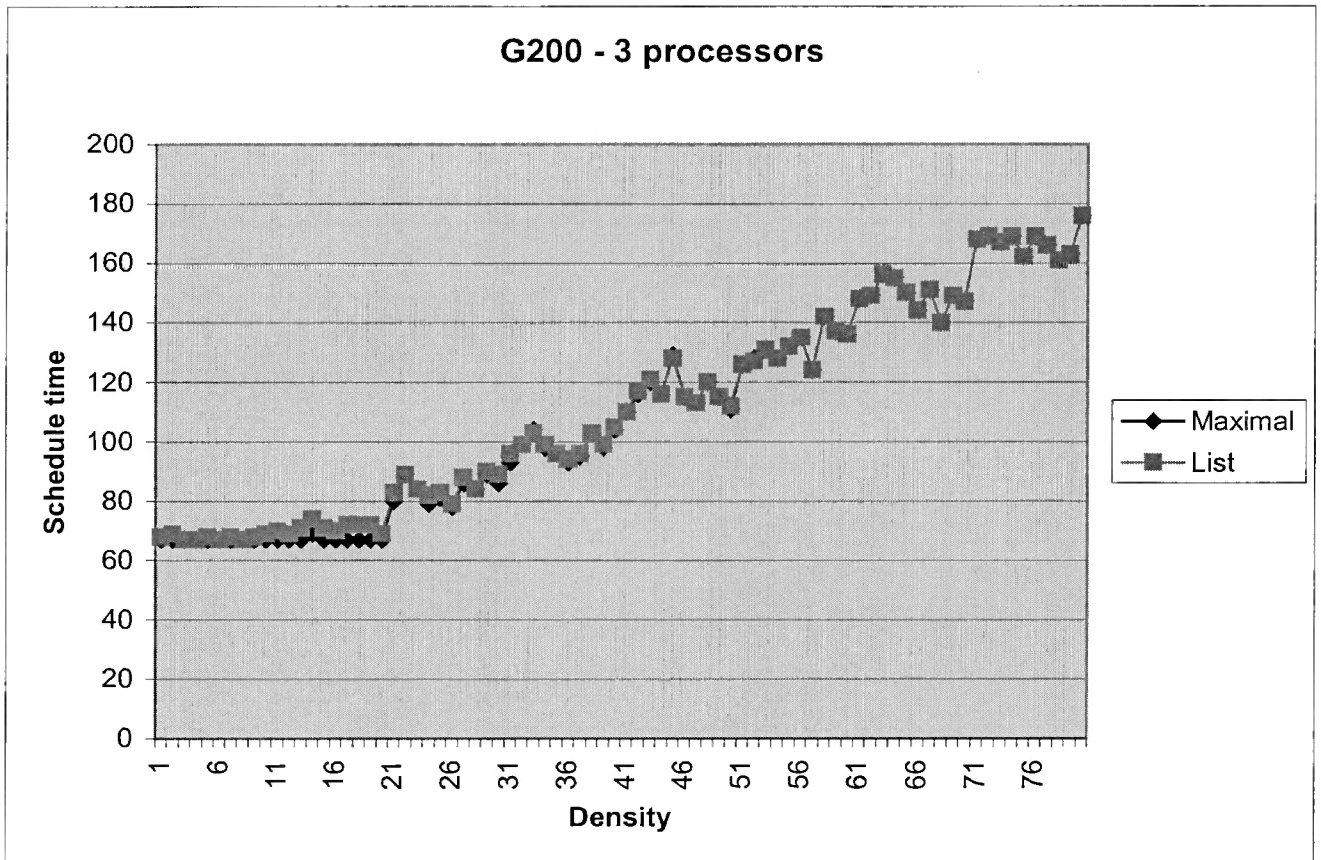
From the above experiments and results for medium task graphs (100 node graphs) when run on 3, 4 and 5 processors we can draw the following overall conclusions:

- 1) It can be concluded that as the number of processors increases for graphs between density of 0.1 and 0.4, the schedule length reduces slightly. The percentage of reduction in schedule length decreases as the number of processors increases from 3 to 5.
- 2) It can be concluded that as the number of processors increases for graphs with density greater than 0.4, the schedule length does not reduce at all for most graphs. This implies that as the density of the graphs increases, adding more processors will not help to reduce the schedule length.

Experiment for 3-processor scheduling using graphs with 200 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List
G200_1_1.txt	67	68
G200_1_2.txt	67	69
G200_1_3.txt	67	67
G200_1_4.txt	67	67
G200_1_5.txt	67	68
G200_1_6.txt	67	67
G200_1_7.txt	67	68
G200_1_8.txt	67	67
G200_1_9.txt	67	68
G200_1_10.txt	67	69
G200_2_1.txt	67	70
G200_2_2.txt	67	69
G200_2_3.txt	67	71
G200_2_4.txt	69	74
G200_2_5.txt	67	71
G200_2_6.txt	67	70
G200_2_7.txt	67	72
G200_2_8.txt	67	72
G200_2_9.txt	67	72
G200_2_10.txt	67	69
G200_3_1.txt	80	83
G200_3_2.txt	89	89
G200_3_3.txt	84	84
G200_3_4.txt	79	82
G200_3_5.txt	81	83
G200_3_6.txt	78	79
G200_3_7.txt	86	88
G200_3_8.txt	84	84
G200_3_9.txt	89	90
G200_3_10.txt	86	89
G200_4_1.txt	93	96
G200_4_2.txt	99	99
G200_4_3.txt	104	103
G200_4_4.txt	98	99
G200_4_5.txt	96	96
G200_4_6.txt	93	94
G200_4_7.txt	95	96
G200_4_8.txt	103	103
G200_4_9.txt	98	99
G200_4_10.txt	104	105
G200_5_1.txt	110	110

G200_5_2.txt	116	117
G200_5_3.txt	120	121
G200_5_4.txt	116	116
G200_5_5.txt	129	128
G200_5_6.txt	115	115
G200_5_7.txt	113	113
G200_5_8.txt	120	120
G200_5_9.txt	115	115
G200_5_10.txt	111	112
G200_6_1.txt	126	126
G200_6_2.txt	128	127
G200_6_3.txt	131	131
G200_6_4.txt	128	128
G200_6_5.txt	132	132
G200_6_6.txt	135	135
G200_6_7.txt	124	124
G200_6_8.txt	142	142
G200_6_9.txt	137	137
G200_6_10.txt	136	136
G200_7_1.txt	148	148
G200_7_2.txt	149	149
G200_7_3.txt	156	156
G200_7_4.txt	155	155
G200_7_5.txt	150	150
G200_7_6.txt	144	144
G200_7_7.txt	151	151
G200_7_8.txt	140	140
G200_7_9.txt	149	149
G200_7_10.txt	147	147
G200_8_1.txt	168	168
G200_8_2.txt	169	169
G200_8_3.txt	167	167
G200_8_4.txt	169	169
G200_8_5.txt	162	162
G200_8_6.txt	169	169
G200_8_7.txt	166	166
G200_8_8.txt	161	161
G200_8_9.txt	163	163
G200_8_10.txt	176	176



**Figure 6.7. Graphs with 200 nodes on 3 processors**

From the above experiments and results we can draw the following conclusions:

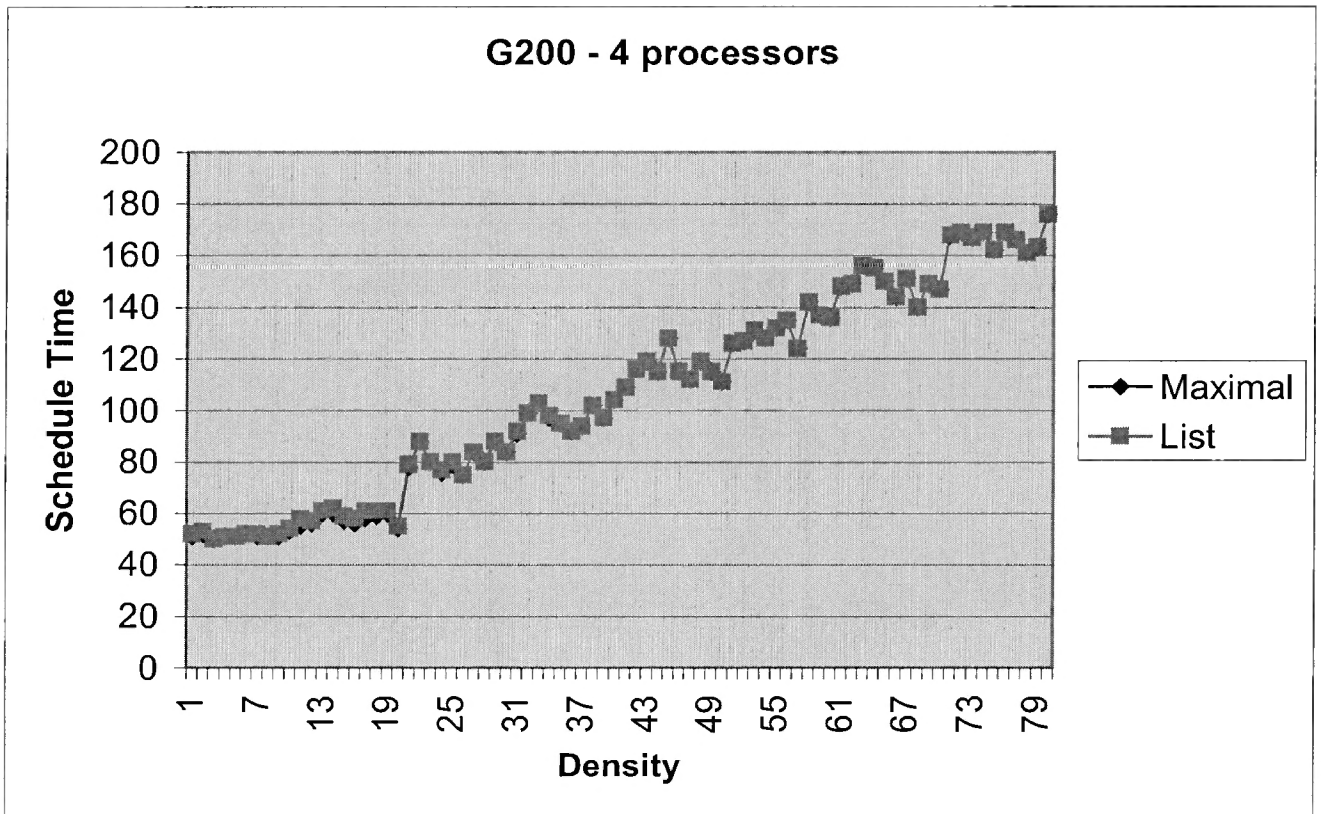
- 1) It can be concluded that the maximal chain scheduling heuristic performs slightly better than the List scheduling when the density/sparseness of the graphs is between 0.1 and 0.4 for most of the cases.
- 2) The maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance when the density/sparseness of the graphs is greater than 0.4 for most of the cases.

Experiment for 4-processor scheduling using graphs with 200 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List
G200_1_1.txt	51	52
G200_1_2.txt	52	53
G200_1_3.txt	50	50
G200_1_4.txt	51	51
G200_1_5.txt	51	51
G200_1_6.txt	52	52
G200_1_7.txt	51	52
G200_1_8.txt	51	51
G200_1_9.txt	51	52
G200_1_10.txt	53	54
G200_2_1.txt	55	58
G200_2_2.txt	56	57
G200_2_3.txt	60	61
G200_2_4.txt	60	62
G200_2_5.txt	57	59
G200_2_6.txt	56	58
G200_2_7.txt	58	61
G200_2_8.txt	59	61
G200_2_9.txt	60	61
G200_2_10.txt	54	55
G200_3_1.txt	78	79
G200_3_2.txt	88	88
G200_3_3.txt	80	80
G200_3_4.txt	76	77
G200_3_5.txt	79	80
G200_3_6.txt	75	75
G200_3_7.txt	84	84
G200_3_8.txt	80	80
G200_3_9.txt	88	88
G200_3_10.txt	84	84
G200_4_1.txt	91	92
G200_4_2.txt	99	99
G200_4_3.txt	103	103
G200_4_4.txt	97	98
G200_4_5.txt	95	95
G200_4_6.txt	92	92
G200_4_7.txt	94	94
G200_4_8.txt	102	102
G200_4_9.txt	97	97
G200_4_10.txt	104	104
G200_5_1.txt	109	109

G200_5_2.txt	116	116
G200_5_3.txt	119	119
G200_5_4.txt	115	115
G200_5_5.txt	128	128
G200_5_6.txt	115	115
G200_5_7.txt	112	112
G200_5_8.txt	119	119
G200_5_9.txt	115	115
G200_5_10.txt	111	111
G200_6_1.txt	126	126
G200_6_2.txt	127	127
G200_6_3.txt	131	131
G200_6_4.txt	128	128
G200_6_5.txt	132	132
G200_6_6.txt	135	135
G200_6_7.txt	124	124
G200_6_8.txt	142	142
G200_6_9.txt	137	137
G200_6_10.txt	136	136
G200_7_1.txt	148	148
G200_7_2.txt	149	149
G200_7_3.txt	156	156
G200_7_4.txt	155	155
G200_7_5.txt	150	150
G200_7_6.txt	144	144
G200_7_7.txt	151	151
G200_7_8.txt	140	140
G200_7_9.txt	149	149
G200_7_10.txt	147	147
G200_8_1.txt	168	168
G200_8_2.txt	169	169
G200_8_3.txt	167	167
G200_8_4.txt	169	169
G200_8_5.txt	162	162
G200_8_6.txt	169	169
G200_8_7.txt	166	166
G200_8_8.txt	161	161
G200_8_9.txt	163	163
G200_8_10.txt	176	176





**Figure 6.8. Graphs with 200 nodes on 4 processors**

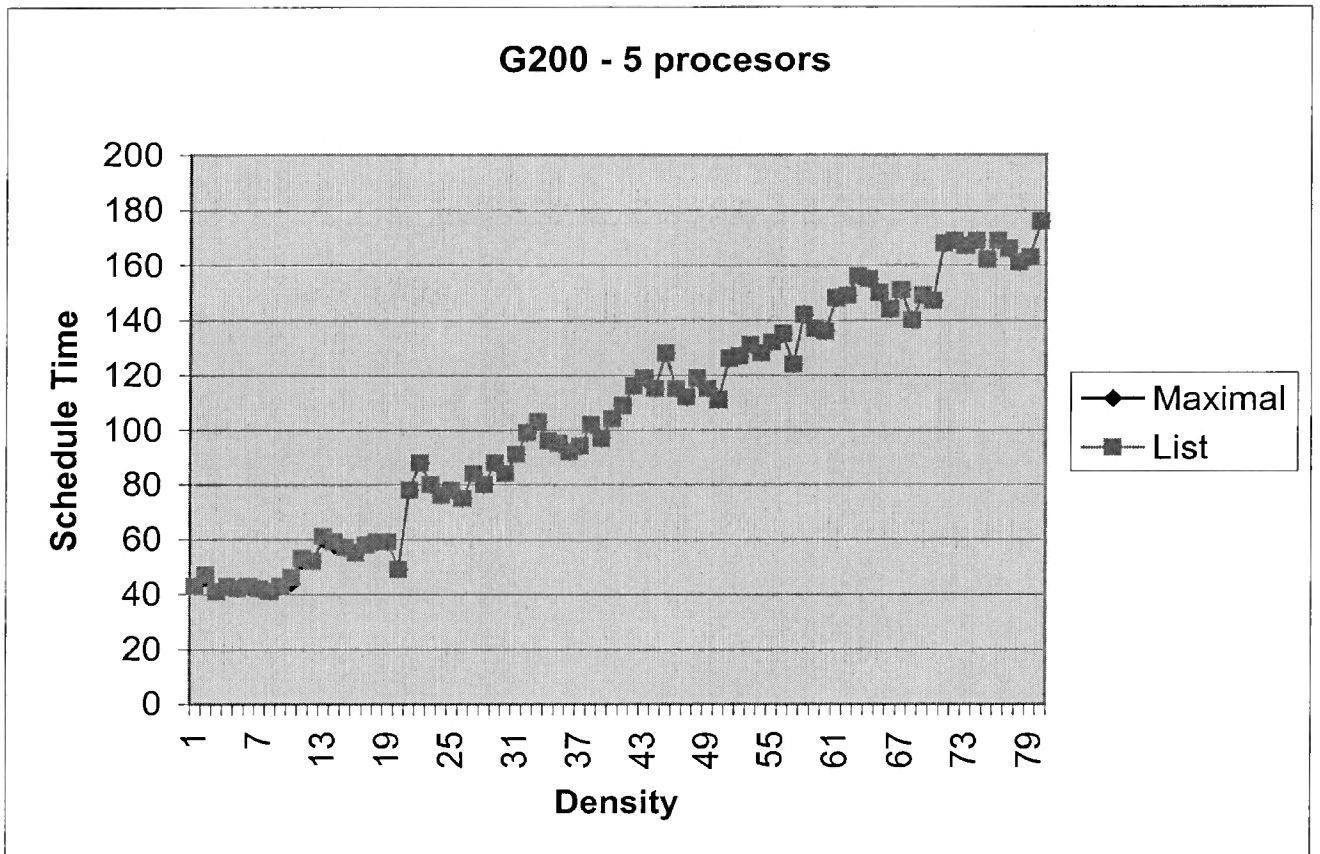
From the above experiments and results we can draw the following conclusions:

- 1) It can be concluded that the maximal chain scheduling heuristic performs slightly better than the List scheduling when the density/sparseness of the graphs is between 0.1 and 0.3 for most of the cases.
- 2) The maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance when the density/sparseness of the graphs is greater than 0.3 for most of the cases.

Experiment for 5-processor scheduling using graphs with 200 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List
G200_1_1.txt	43	43
G200_1_2.txt	46	47
G200_1_3.txt	41	41
G200_1_4.txt	43	43
G200_1_5.txt	42	42
G200_1_6.txt	43	43
G200_1_7.txt	42	42
G200_1_8.txt	41	41
G200_1_9.txt	43	43
G200_1_10.txt	44	46
G200_2_1.txt	52	53
G200_2_2.txt	52	52
G200_2_3.txt	60	61
G200_2_4.txt	58	59
G200_2_5.txt	57	57
G200_2_6.txt	55	55
G200_2_7.txt	58	58
G200_2_8.txt	59	59
G200_2_9.txt	59	59
G200_2_10.txt	49	49
G200_3_1.txt	78	78
G200_3_2.txt	88	88
G200_3_3.txt	80	80
G200_3_4.txt	76	76
G200_3_5.txt	78	78
G200_3_6.txt	75	75
G200_3_7.txt	84	84
G200_3_8.txt	80	80
G200_3_9.txt	88	88
G200_3_10.txt	84	84
G200_4_1.txt	91	91
G200_4_2.txt	99	99
G200_4_3.txt	103	103
G200_4_4.txt	96	96
G200_4_5.txt	95	95
G200_4_6.txt	92	92
G200_4_7.txt	94	94
G200_4_8.txt	102	102
G200_4_9.txt	97	97
G200_4_10.txt	104	104
G200_5_1.txt	109	109

G200_5_2.txt	116	116
G200_5_3.txt	119	119
G200_5_4.txt	115	115
G200_5_5.txt	128	128
G200_5_6.txt	115	115
G200_5_7.txt	112	112
G200_5_8.txt	119	119
G200_5_9.txt	115	115
G200_5_10.txt	111	111
G200_6_1.txt	126	126
G200_6_2.txt	127	127
G200_6_3.txt	131	131
G200_6_4.txt	128	128
G200_6_5.txt	132	132
G200_6_6.txt	135	135
G200_6_7.txt	124	124
G200_6_8.txt	142	142
G200_6_9.txt	137	137
G200_6_10.txt	136	136
G200_7_1.txt	148	148
G200_7_2.txt	149	149
G200_7_3.txt	156	156
G200_7_4.txt	155	155
G200_7_5.txt	150	150
G200_7_6.txt	144	144
G200_7_7.txt	151	151
G200_7_8.txt	140	140
G200_7_9.txt	149	149
G200_7_10.txt	147	147
G200_8_1.txt	168	168
G200_8_2.txt	169	169
G200_8_3.txt	167	167
G200_8_4.txt	169	169
G200_8_5.txt	162	162
G200_8_6.txt	169	169
G200_8_7.txt	166	166
G200_8_8.txt	161	161
G200_8_9.txt	163	163
G200_8_10.txt	176	176



**Figure 6.9. Graphs with 200 nodes on 5 processors**

From the above experiments and results we can draw the following conclusions:

1. It can be concluded that the maximal chain scheduling heuristic performs slightly better than the List scheduling when the density/sparseness of the graphs is between 0.1 and 0.2 for most of the cases.
2. The maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance when the density/sparseness of the graphs is greater than 0.2 for most of the cases.

From the above experiments and results for the medium task graphs (200 node graphs), when run on 3, 4 and 5 processors we can draw the following overall conclusions:

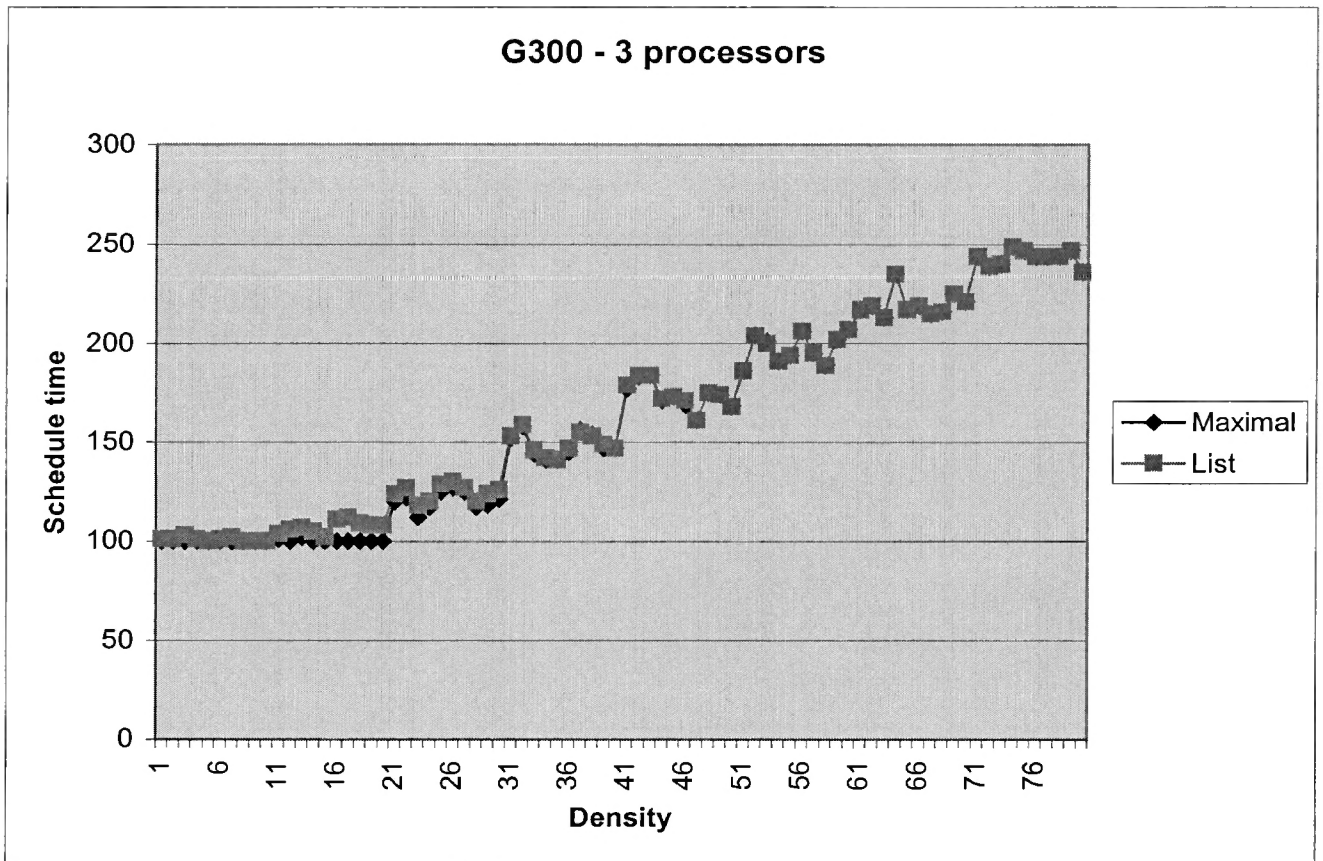
- 1) It can be concluded that as the number of processors increases for graphs between density of 0.1 and 0.4, the schedule length reduces slightly. The percentage of reduction in schedule length decreases as the number of processors increases from 3 to 5.
- 2) It can be concluded that as the number of processors increases for graphs with density greater than 0.4, the schedule length does not reduce at all for most graphs. This implies that as the density of the graphs increases, adding more processors will not help to reduce the schedule length.

### 6.2.3 Large task graphs (300 - 400 nodes) on 3,4 and 5 processors

Experiment for 3-processor scheduling using graphs with 300 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List
G300_1_1.txt	100	101
G300_1_2.txt	100	101
G300_1_3.txt	100	103
G300_1_4.txt	100	101
G300_1_5.txt	100	100
G300_1_6.txt	100	101
G300_1_7.txt	100	102
G300_1_8.txt	100	100
G300_1_9.txt	100	100
G300_1_10.txt	100	100
G300_2_1.txt	101	104
G300_2_2.txt	100	106
G300_2_3.txt	102	107
G300_2_4.txt	100	105
G300_2_5.txt	100	102
G300_2_6.txt	100	111
G300_2_7.txt	100	112
G300_2_8.txt	100	109
G300_2_9.txt	100	108
G300_2_10.txt	100	108
G300_3_1.txt	120	124
G300_3_2.txt	122	127
G300_3_3.txt	112	118
G300_3_4.txt	117	120
G300_3_5.txt	125	129
G300_3_6.txt	127	130
G300_3_7.txt	125	127
G300_3_8.txt	117	120
G300_3_9.txt	118	124
G300_3_10.txt	121	126
G300_4_1.txt	152	153
G300_4_2.txt	158	159
G300_4_3.txt	144	146
G300_4_4.txt	141	142
G300_4_5.txt	141	141
G300_4_6.txt	145	147
G300_4_7.txt	156	155
G300_4_8.txt	154	153
G300_4_9.txt	147	149
G300_4_10.txt	147	147

G300_5_1.txt	177	179
G300_5_2.txt	183	184
G300_5_3.txt	184	184
G300_5_4.txt	171	172
G300_5_5.txt	173	173
G300_5_6.txt	169	171
G300_5_7.txt	161	161
G300_5_8.txt	175	175
G300_5_9.txt	174	174
G300_5_10.txt	168	168
G300_6_1.txt	186	186
G300_6_2.txt	204	204
G300_6_3.txt	201	200
G300_6_4.txt	191	191
G300_6_5.txt	194	194
G300_6_6.txt	206	206
G300_6_7.txt	195	195
G300_6_8.txt	189	189
G300_6_9.txt	202	202
G300_6_10.txt	207	207
G300_7_1.txt	217	217
G300_7_2.txt	219	219
G300_7_3.txt	213	213
G300_7_4.txt	235	235
G300_7_5.txt	217	217
G300_7_6.txt	219	219
G300_7_7.txt	215	215
G300_7_8.txt	216	216
G300_7_9.txt	225	225
G300_7_10.txt	221	221
G300_8_1.txt	244	244
G300_8_2.txt	239	239
G300_8_3.txt	240	240
G300_8_4.txt	249	249
G300_8_5.txt	247	247
G300_8_6.txt	244	244
G300_8_7.txt	244	244
G300_8_8.txt	244	244
G300_8_9.txt	247	247
G300_8_10.txt	236	236



**Figure 6.10. Graphs with 300 nodes on 3 processors**

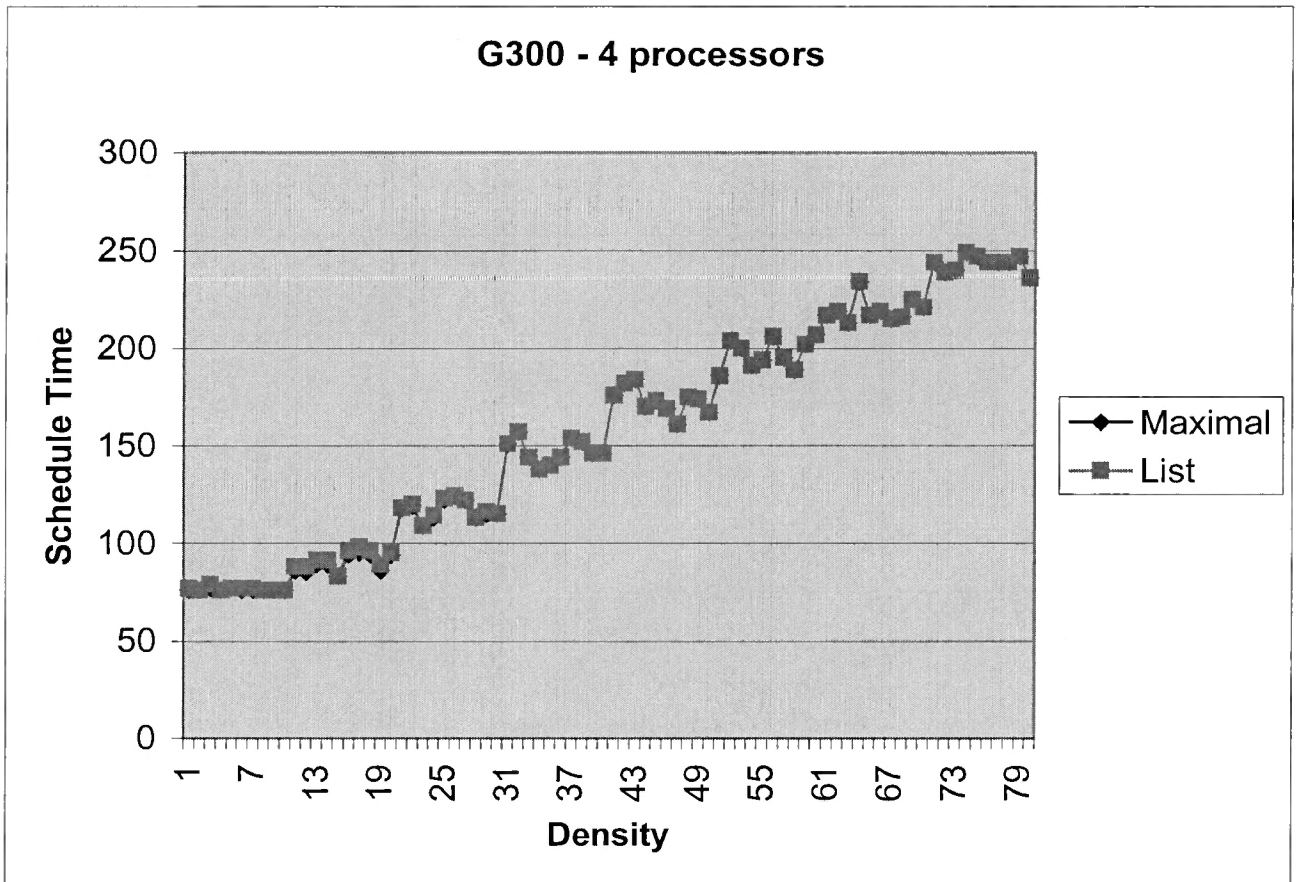
From the above experiments and results we can draw the following conclusions:

- 1) It can be concluded that the maximal chain scheduling heuristic performs slightly better than the List scheduling when the density/sparseness of the graphs is between 0.1 and 0.4 for most of the cases.
- 2) The maximal chain scheduling heuristic and the List scheduling heuristic have the same performance when the density/sparseness of the graphs is greater than 0.4 for most of the cases.

Experiment for 4-processor scheduling using graphs with 300 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List
G300_1_1.txt	76	77
G300_1_2.txt	76	76
G300_1_3.txt	77	79
G300_1_4.txt	76	76
G300_1_5.txt	77	77
G300_1_6.txt	76	77
G300_1_7.txt	76	77
G300_1_8.txt	76	76
G300_1_9.txt	76	76
G300_1_10.txt	76	76
G300_2_1.txt	86	88
G300_2_2.txt	85	88
G300_2_3.txt	89	91
G300_2_4.txt	89	91
G300_2_5.txt	83	83
G300_2_6.txt	94	96
G300_2_7.txt	95	98
G300_2_8.txt	94	96
G300_2_9.txt	86	89
G300_2_10.txt	94	95
G300_3_1.txt	118	118
G300_3_2.txt	119	120
G300_3_3.txt	109	109
G300_3_4.txt	113	114
G300_3_5.txt	122	123
G300_3_6.txt	124	124
G300_3_7.txt	122	122
G300_3_8.txt	113	113
G300_3_9.txt	115	116
G300_3_10.txt	115	115
G300_4_1.txt	151	151
G300_4_2.txt	157	157
G300_4_3.txt	144	144
G300_4_4.txt	138	138
G300_4_5.txt	140	140
G300_4_6.txt	144	144
G300_4_7.txt	154	154
G300_4_8.txt	152	152
G300_4_9.txt	146	146
G300_4_10.txt	146	146
G300_5_1.txt	176	176

G300_5_2.txt	182	182
G300_5_3.txt	184	184
G300_5_4.txt	170	170
G300_5_5.txt	173	173
G300_5_6.txt	169	169
G300_5_7.txt	161	161
G300_5_8.txt	175	175
G300_5_9.txt	174	174
G300_5_10.txt	167	167
G300_6_1.txt	186	186
G300_6_2.txt	204	204
G300_6_3.txt	200	200
G300_6_4.txt	191	191
G300_6_5.txt	194	194
G300_6_6.txt	206	206
G300_6_7.txt	195	195
G300_6_8.txt	189	189
G300_6_9.txt	202	202
G300_6_10.txt	207	207
G300_7_1.txt	217	217
G300_7_2.txt	219	219
G300_7_3.txt	213	213
G300_7_4.txt	234	234
G300_7_5.txt	217	217
G300_7_6.txt	219	219
G300_7_7.txt	215	215
G300_7_8.txt	216	216
G300_7_9.txt	225	225
G300_7_10.txt	221	221
G300_8_1.txt	244	244
G300_8_2.txt	239	239
G300_8_3.txt	240	240
G300_8_4.txt	249	249
G300_8_5.txt	247	247
G300_8_6.txt	244	244
G300_8_7.txt	244	244
G300_8_8.txt	244	244
G300_8_9.txt	247	247
G300_8_10.txt	236	236



**Figure 6.11. Graphs with 300 nodes on 4 processors**

From the above experiments and results we can draw the following conclusions:

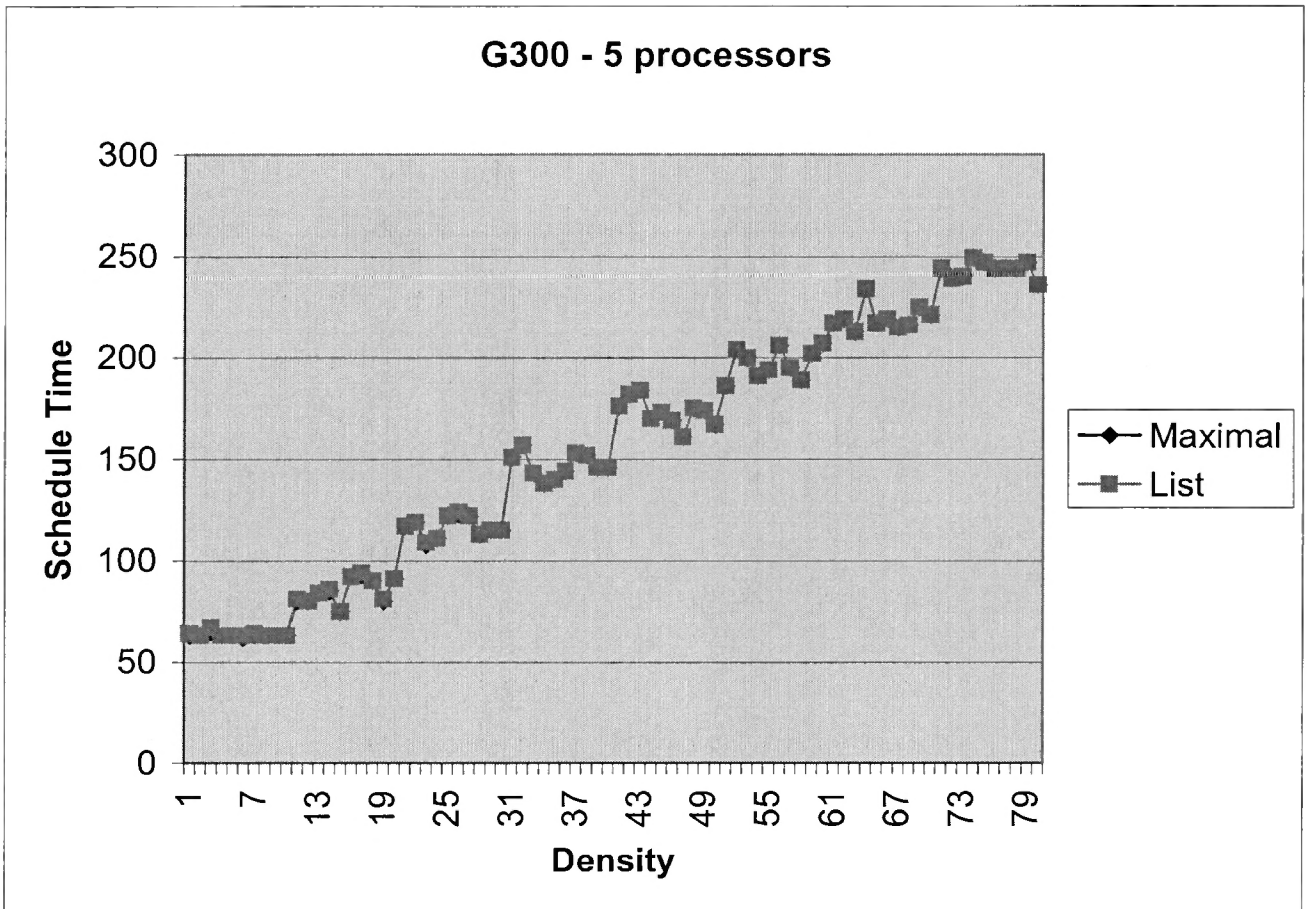
- 1) It can be concluded that the maximal chain scheduling heuristic performs slightly better than the List scheduling when the density/sparseness of the graphs is between 0.1 and 0.3 for most of the cases.
- 2) The maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance when the density/sparseness of the graphs is greater than 0.3 for most of the cases.



Experiment for 5-processor scheduling using graphs with 300 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List
G300_1_1.txt	63	64
G300_1_2.txt	63	63
G300_1_3.txt	65	67
G300_1_4.txt	63	63
G300_1_5.txt	63	63
G300_1_6.txt	62	63
G300_1_7.txt	63	64
G300_1_8.txt	63	63
G300_1_9.txt	63	63
G300_1_10.txt	63	63
G300_2_1.txt	80	81
G300_2_2.txt	80	80
G300_2_3.txt	84	84
G300_2_4.txt	85	86
G300_2_5.txt	75	75
G300_2_6.txt	92	92
G300_2_7.txt	93	94
G300_2_8.txt	90	90
G300_2_9.txt	80	81
G300_2_10.txt	91	91
G300_3_1.txt	117	117
G300_3_2.txt	119	119
G300_3_3.txt	108	109
G300_3_4.txt	111	111
G300_3_5.txt	122	122
G300_3_6.txt	123	124
G300_3_7.txt	122	122
G300_3_8.txt	113	113
G300_3_9.txt	115	115
G300_3_10.txt	115	115
G300_4_1.txt	151	151
G300_4_2.txt	157	157
G300_4_3.txt	143	143
G300_4_4.txt	138	138
G300_4_5.txt	140	140
G300_4_6.txt	144	144
G300_4_7.txt	153	153
G300_4_8.txt	152	152
G300_4_9.txt	146	146
G300_4_10.txt	146	146
G300_5_1.txt	176	176

G300_5_2.txt	182	182
G300_5_3.txt	184	184
G300_5_4.txt	170	170
G300_5_5.txt	173	173
G300_5_6.txt	169	169
G300_5_7.txt	161	161
G300_5_8.txt	175	175
G300_5_9.txt	174	174
G300_5_10.txt	167	167
G300_6_1.txt	186	186
G300_6_2.txt	204	204
G300_6_3.txt	200	200
G300_6_4.txt	191	191
G300_6_5.txt	194	194
G300_6_6.txt	206	206
G300_6_7.txt	195	195
G300_6_8.txt	189	189
G300_6_9.txt	202	202
G300_6_10.txt	207	207
G300_7_1.txt	217	217
G300_7_2.txt	219	219
G300_7_3.txt	213	213
G300_7_4.txt	234	234
G300_7_5.txt	217	217
G300_7_6.txt	219	219
G300_7_7.txt	215	215
G300_7_8.txt	216	216
G300_7_9.txt	225	225
G300_7_10.txt	221	221
G300_8_1.txt	244	244
G300_8_2.txt	239	239
G300_8_3.txt	240	240
G300_8_4.txt	249	249
G300_8_5.txt	247	247
G300_8_6.txt	244	244
G300_8_7.txt	244	244
G300_8_8.txt	244	244
G300_8_9.txt	247	247
G300_8_10.txt	236	236



**Figure 6.12. Graphs with 300 nodes on 5 processors**

From the above experiments and results we can draw the following conclusions:

- 1) It can be concluded that the maximal chain scheduling heuristic performs slightly better than the List scheduling when the density/sparseness of the graphs is between 0.1 and 0.2 for most of the cases.
- 2) The maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance when the density/sparseness of the graphs is greater than 0.2 for most of the cases.

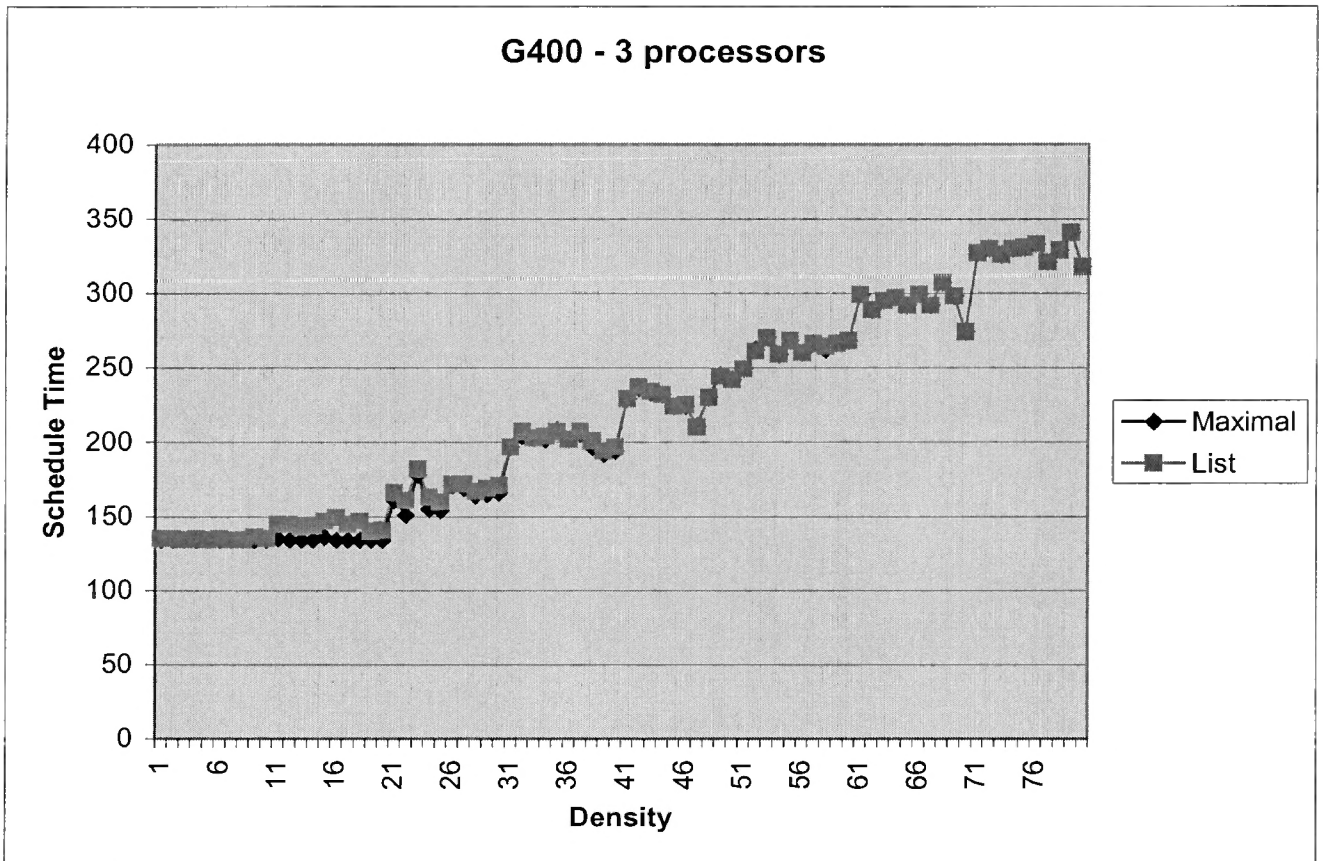
From the above experiments and results for the large task graphs (300 node graphs), when run on 3, 4 and 5 processors we can draw the following overall conclusions:

- 1) It can be concluded that as the number of processors increases for graphs between density of 0.1 and 0.4, the schedule length reduces slightly. The percentage of reduction in schedule length decreases as the number of processors increases from 3 to 5.
- 2) It can be concluded that as the number of processors increases for graphs with density greater than 0.4, the schedule length does not reduce at all for most graphs. This implies that as the density of the graphs increases, adding more processors will not help to reduce the schedule length.

Experiment for 3-processor scheduling using graphs with 400 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List
G400_1_1.txt	134	135
G400_1_2.txt	134	135
G400_1_3.txt	134	134
G400_1_4.txt	134	135
G400_1_5.txt	134	134
G400_1_6.txt	134	135
G400_1_7.txt	134	134
G400_1_8.txt	134	134
G400_1_9.txt	134	136
G400_1_10.txt	134	135
G400_2_1.txt	135	145
G400_2_2.txt	134	145
G400_2_3.txt	134	143
G400_2_4.txt	134	144
G400_2_5.txt	136	147
G400_2_6.txt	134	149
G400_2_7.txt	134	145
G400_2_8.txt	134	147
G400_2_9.txt	134	140
G400_2_10.txt	134	141
G400_3_1.txt	161	166
G400_3_2.txt	151	161
G400_3_3.txt	178	182
G400_3_4.txt	155	163
G400_3_5.txt	154	160
G400_3_6.txt	171	172
G400_3_7.txt	169	172
G400_3_8.txt	164	167
G400_3_9.txt	165	169
G400_3_10.txt	166	171
G400_4_1.txt	197	197
G400_4_2.txt	204	207
G400_4_3.txt	203	203
G400_4_4.txt	202	204
G400_4_5.txt	208	207
G400_4_6.txt	202	202
G400_4_7.txt	206	207
G400_4_8.txt	197	201
G400_4_9.txt	192	194
G400_4_10.txt	194	197
G400_5_1.txt	229	229

G400_5_2.txt	236	237
G400_5_3.txt	233	234
G400_5_4.txt	232	232
G400_5_5.txt	224	224
G400_5_6.txt	224	225
G400_5_7.txt	211	210
G400_5_8.txt	230	230
G400_5_9.txt	245	244
G400_5_10.txt	242	242
G400_6_1.txt	249	249
G400_6_2.txt	262	261
G400_6_3.txt	270	270
G400_6_4.txt	259	259
G400_6_5.txt	268	268
G400_6_6.txt	260	260
G400_6_7.txt	266	266
G400_6_8.txt	262	264
G400_6_9.txt	266	266
G400_6_10.txt	268	268
G400_7_1.txt	299	299
G400_7_2.txt	289	289
G400_7_3.txt	295	295
G400_7_4.txt	297	297
G400_7_5.txt	292	292
G400_7_6.txt	299	299
G400_7_7.txt	292	292
G400_7_8.txt	307	307
G400_7_9.txt	298	298
G400_7_10.txt	274	274
G400_8_1.txt	327	327
G400_8_2.txt	330	330
G400_8_3.txt	326	326
G400_8_4.txt	330	330
G400_8_5.txt	331	331
G400_8_6.txt	333	333
G400_8_7.txt	321	321
G400_8_8.txt	329	329
G400_8_9.txt	341	341
G400_8_10.txt	318	318



**Figure 6.13. Graphs with 400 nodes on 3 processors**

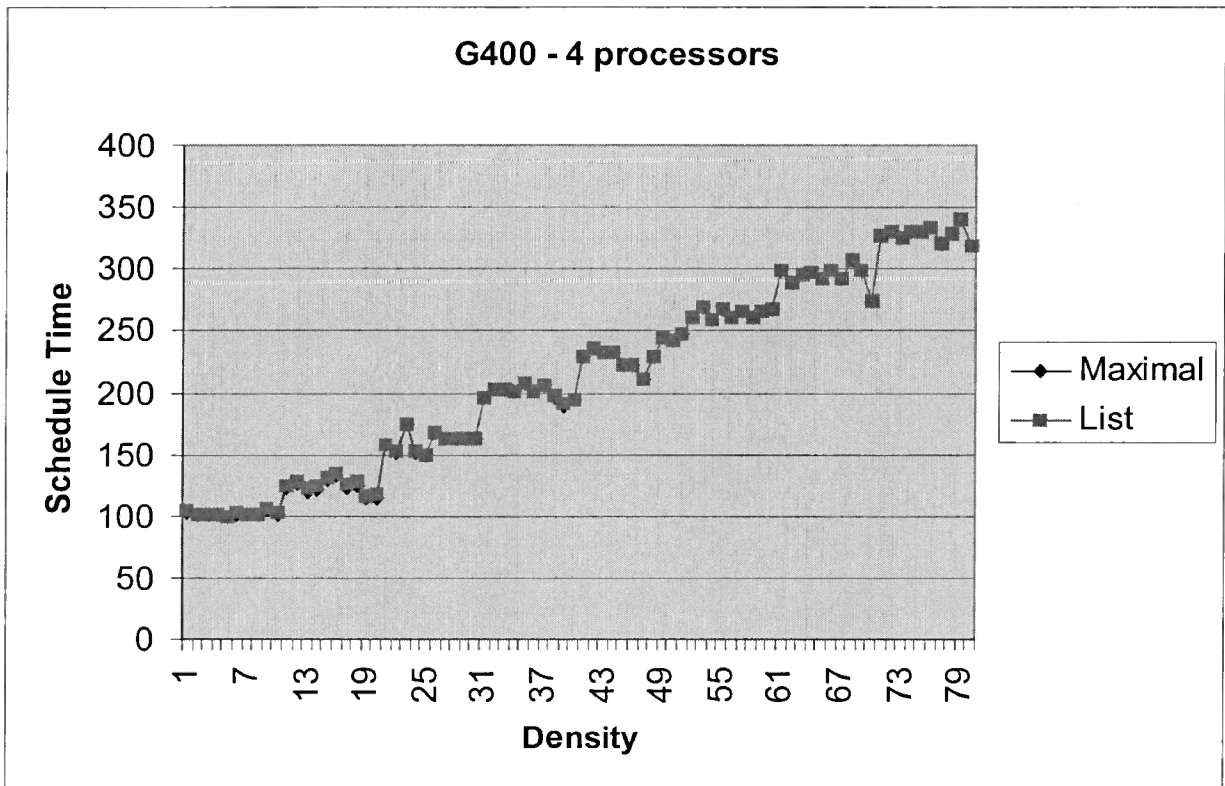
From the above experiments and results we can draw the following conclusions:

- 1) It can be concluded that the maximal chain scheduling heuristic performs slightly better than the List scheduling when the density/sparseness of the graphs is between 0.1 and 0.4 for most of the cases.
- 2) The maximal chain scheduling heuristic and the List scheduling heuristic have the same performance when the density/sparseness of the graphs is greater than 0.4 for most of the cases.

Experiment for 4-processor scheduling using graphs with 400 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List
G400_1_1.txt	103	104
G400_1_2.txt	102	102
G400_1_3.txt	102	102
G400_1_4.txt	101	102
G400_1_5.txt	100	100
G400_1_6.txt	102	103
G400_1_7.txt	102	102
G400_1_8.txt	101	101
G400_1_9.txt	105	106
G400_1_10.txt	102	103
G400_2_1.txt	122	124
G400_2_2.txt	126	128
G400_2_3.txt	119	123
G400_2_4.txt	121	124
G400_2_5.txt	129	131
G400_2_6.txt	132	134
G400_2_7.txt	122	126
G400_2_8.txt	125	127
G400_2_9.txt	115	117
G400_2_10.txt	114	118
G400_3_1.txt	157	158
G400_3_2.txt	151	152
G400_3_3.txt	174	174
G400_3_4.txt	151	152
G400_3_5.txt	149	150
G400_3_6.txt	168	168
G400_3_7.txt	162	163
G400_3_8.txt	162	163
G400_3_9.txt	162	162
G400_3_10.txt	162	163
G400_4_1.txt	196	196
G400_4_2.txt	203	203
G400_4_3.txt	202	202
G400_4_4.txt	201	201
G400_4_5.txt	207	207
G400_4_6.txt	201	201
G400_4_7.txt	205	205
G400_4_8.txt	196	197
G400_4_9.txt	190	191
G400_4_10.txt	194	194
G400_5_1.txt	229	229

G400_5_2.txt	235	235
G400_5_3.txt	232	232
G400_5_4.txt	232	232
G400_5_5.txt	223	223
G400_5_6.txt	223	223
G400_5_7.txt	210	210
G400_5_8.txt	229	229
G400_5_9.txt	244	244
G400_5_10.txt	242	242
G400_6_1.txt	248	248
G400_6_2.txt	261	261
G400_6_3.txt	269	269
G400_6_4.txt	259	259
G400_6_5.txt	268	268
G400_6_6.txt	260	260
G400_6_7.txt	266	266
G400_6_8.txt	261	261
G400_6_9.txt	266	266
G400_6_10.txt	268	268
G400_7_1.txt	299	299
G400_7_2.txt	289	289
G400_7_3.txt	295	295
G400_7_4.txt	297	297
G400_7_5.txt	292	292
G400_7_6.txt	298	298
G400_7_7.txt	292	292
G400_7_8.txt	307	307
G400_7_9.txt	298	298
G400_7_10.txt	274	274
G400_8_1.txt	327	327
G400_8_2.txt	330	330
G400_8_3.txt	326	326
G400_8_4.txt	330	330
G400_8_5.txt	331	331
G400_8_6.txt	333	333
G400_8_7.txt	321	321
G400_8_8.txt	329	329
G400_8_9.txt	341	341
G400_8_10.txt	318	318



**Figure 6.14. Graphs with 400 nodes on 4 processors**

From the above experiments and results we can draw the following conclusions:

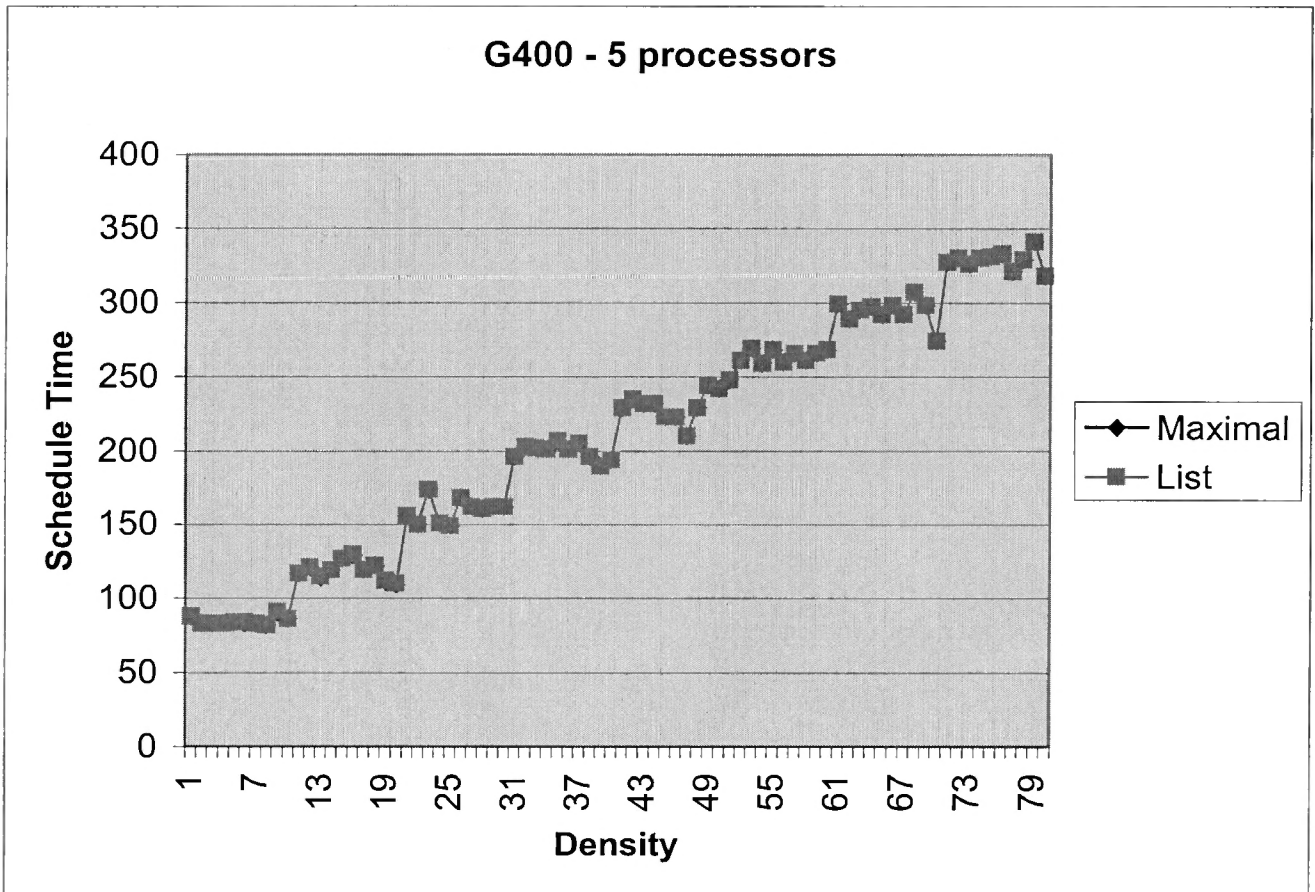
- 1) It can be concluded that the maximal chain scheduling heuristic performs slightly better than the List scheduling when the density/sparseness of the graphs is between 0.1 and 0.3 for most of the cases.
- 2) The maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance when the density/sparseness of the graphs is greater than 0.3 for most of the cases.

Experiment for 5-processor scheduling using graphs with 400 nodes of density from 0.1 to 0.8.

Graphs	Maximal	List
G400_1_1.txt	88	88
G400_1_2.txt	83	83
G400_1_3.txt	83	83
G400_1_4.txt	83	83
G400_1_5.txt	84	84
G400_1_6.txt	83	84
G400_1_7.txt	83	83
G400_1_8.txt	82	82
G400_1_9.txt	90	91
G400_1_10.txt	86	86
G400_2_1.txt	117	117
G400_2_2.txt	121	121
G400_2_3.txt	114	115
G400_2_4.txt	119	119
G400_2_5.txt	127	127
G400_2_6.txt	129	130
G400_2_7.txt	119	119
G400_2_8.txt	122	122
G400_2_9.txt	111	112
G400_2_10.txt	109	110
G400_3_1.txt	156	156
G400_3_2.txt	150	150
G400_3_3.txt	174	174
G400_3_4.txt	151	151
G400_3_5.txt	149	149
G400_3_6.txt	168	168
G400_3_7.txt	162	162
G400_3_8.txt	161	161
G400_3_9.txt	162	162
G400_3_10.txt	162	162
G400_4_1.txt	196	196
G400_4_2.txt	203	203
G400_4_3.txt	202	202
G400_4_4.txt	201	201
G400_4_5.txt	207	207
G400_4_6.txt	201	201
G400_4_7.txt	205	205
G400_4_8.txt	196	196
G400_4_9.txt	190	190
G400_4_10.txt	194	194
G400_5_1.txt	229	229

G400_5_2.txt	235	235
G400_5_3.txt	232	232
G400_5_4.txt	232	232
G400_5_5.txt	223	223
G400_5_6.txt	223	223
G400_5_7.txt	210	210
G400_5_8.txt	229	229
G400_5_9.txt	244	244
G400_5_10.txt	242	242
G400_6_1.txt	248	248
G400_6_2.txt	261	261
G400_6_3.txt	269	269
G400_6_4.txt	259	259
G400_6_5.txt	268	268
G400_6_6.txt	260	260
G400_6_7.txt	266	266
G400_6_8.txt	261	261
G400_6_9.txt	266	266
G400_6_10.txt	268	268
G400_7_1.txt	299	299
G400_7_2.txt	289	289
G400_7_3.txt	295	295
G400_7_4.txt	297	297
G400_7_5.txt	292	292
G400_7_6.txt	298	298
G400_7_7.txt	292	292
G400_7_8.txt	307	307
G400_7_9.txt	298	298
G400_7_10.txt	274	274
G400_8_1.txt	327	327
G400_8_2.txt	330	330
G400_8_3.txt	326	326
G400_8_4.txt	330	330
G400_8_5.txt	331	331
G400_8_6.txt	333	333
G400_8_7.txt	321	321
G400_8_8.txt	329	329
G400_8_9.txt	341	341
G400_8_10.txt	318	318





**Figure 6.15. Graphs with 400 nodes on 5 processors**

From the above experiments and results we can draw the following conclusions:

- 1) It can be concluded that the maximal chain scheduling heuristic performs slightly better than the List scheduling when the density/sparseness of the graphs is between 0.1 and 0.2 for most of the cases.
- 2) The maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance when the density/sparseness of the graphs is greater than 0.2 for most of the cases.

From the above experiments and results for the large task graphs (400 node graphs), when run on 3, 4 and 5 processors we can draw the following overall conclusions:

- 1) It can be concluded that as the number of processors increases for graphs between density of 0.1 and 0.4, the schedule length reduces slightly. The percentage of reduction in schedule length decreases as the number of processors increases from 3 to 5.
- 2) It can be concluded that as the number of processors increases for graphs with density greater than 0.4, the schedule length does not reduce at all for most graphs. This implies that as the density of the graphs increases, adding more processors will not help to reduce the schedule length.

## 7 Conclusions and Future Research

In this section, we list the conclusions that we derived based on the proposed maximal chain heuristic algorithm and the various experiments we conducted and the results obtained from these experiments on various types of tasks graphs. We also suggest future research possibilities based on the proposed maximal chain scheduling heuristic and recursive approach taken to address the multiprocessor scheduling problem.

### 7.1 Conclusions

From the above experiments and results we can draw the following conclusions.

- 1) It can be concluded that the maximal chain scheduling heuristic performs slightly better than the List scheduling when the density/sparseness of the graphs is between 0.1 and 0.4 for most of the cases.
- 2) The maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance when the density/sparseness of the graphs is greater than 0.4 for most of the cases.
- 3) For graphs with small number of nodes (less than 35 nodes), both the maximal chain scheduling heuristic and the List scheduling heuristic gave solutions, which were close to the optimal solution and differed only by 1 or 2 time slots.
- 4) For graphs with small number of nodes (less than 35 nodes), both the optimal all maximal chain scheduling algorithm performed slightly better or the same as the List scheduling heuristic and the maximal chain scheduling heuristic in most cases.
- 5) It can be concluded that when the density/sparseness of the graphs is greater than or equal to 0.5 both the List scheduling heuristic and the Maximal chain heuristic

have the same performance. They both produce the same result in terms of scheduling time for the graphs.

- 6) It can be concluded that when the density/sparseness of the graph increases and the number of processors is increased, the scheduling time is not affected significantly. In fact when the density is 0.4 or greater most of the times they produce the same scheduling time as the 3-processor. Also the List scheduling heuristic and the Maximal chain heuristic have the same performance for graphs with higher density and number of processors greater than 3.

## 7.2 Future Research

Future research efforts could further investigate enhancements to the Merge routine for merging the maximal chain and the  $(n-1)$  processor schedule. One could also investigate the effect of execution time and communication costs for the proposed maximal chain scheduling heuristic, which is based on a recursive approach.

It was suggested, by Dr. Hesham Ali that since, the optimal algorithm presented by Coffman and Graham for 2-processor scheduling is truly based on identifying the maximal chain and then assigning the remaining tasks appropriately to get the two processor schedule, If we take this one step further, we can say that, finding all the maximal chains and performing our proposed maximal chain heuristic on it should yield us the minimum optimal schedule most of the times. We used the “All maximal chains” routine for testing purposes on graphs with small number of nodes, we know for sure that the “All maximal chains” algorithm is exponential. Future research efforts could further investigate this conjecture.

The main concept behind the maximal chain heuristic is that we break down the problem of  $n$  processors into  $1 + (n-1)$  processors. Recursively doing this eventually leads us to the 2-processor scheduling problem for which we have an optimal algorithm (presented by Coffman and Graham). Future research efforts could further investigate this recursive approach for multiprocessor scheduling.

## References

- [AbDa86] S. Abraham and E. Davidson, Task assignment using network flow methods for minimizing communication in n-processor systems, Technical report, Center for Supercomputing Research and Development, University of Illinois, 1986.
- [AEHu92] H. Ali, H. El-Rewini and Y. Huang, Experimental results of a task allocation heuristic using split graph model, Technical report, Depart of Mathematics and Computer Science, University of Nebraska at Omaha, 1992.
- [AIE193] H. Ali and H. El-Rewini, Task allocation in distributed systems: A split-graph model, *Journal of Combinatorial Mathematics and Combinatorial Computing*, 14, (1993). 15-32.
- [AliH2001] Private communication with Dr. Hesham Ali, (1999-2001).
- [Bokh81] S. Bokhari, A shortest tree algorithm for optimal assignments across space and time in distributed processor systems, *IEEE Transactions on Software Engineering*, SE-7, no. 6, (1981).
- [Bokh78] S. Bokhari, Dual processing scheduling with dynamic reassignment, *IEEE Transactions on Software Engineering*, (1978), 254-258.
- [CGS76] E. G. Coffman, R. L. Graham, J. L. Bruno, W. H. Kohler, R. Sethi, K. Steiglitz, and J. D. Ullman: *Computer and Job-Shop Scheduling Theory*, 1976 by John Wiley & Sons, A Wiley-Inter-Science publication
- [ChAb81] T. Chou and J. Abraham, Load balancing in distributed systems, *IEEE Transaction on Software Engineering* SE-8, no. 4, (1981).
- [CHLE80] W. Chu, L. Holloway, M. Lan and K. Efe, Task allocation in distributed data processing, *IEEE Computer*, (1980), 57-69.

- [Efe82] K. J. Efe, Heuristic models of task assignment scheduling in distributed systems, *IEEE Computer*, (1982), 57-56.
- [GaJo79] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, 1979.
- [GJTY83] M. Garey, D. Johnson, R. Tarjan, and M. Yannakakis, Scheduling Opposing Forests, *SIAM Journal. Alg. Disc. Math.*, 4, no. 1, (1983), 72-92
- [Golu80] M. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.
- [Llki92] C. Lee, D. Lee, M. Kim, Optimal Task Assignment in Linear Array Networks, *IEEE Trans. On Computers* 41, no 7, (1992), 877-880.
- [Lo88] V. Lo, Heuristic algorithms for task assignment in distributed systems, *IEEE Trans. On Computers* 37, no 11, (1988), 1384-1397.
- [MLTs82] P. Ma, E. Lee and M. Tsuchiya, A task allocation model for distributed computing systems, *IEEE Trans. Comput.*, (1982), 41-47.
- [PaYa79] C. Papadimitriou, and M. Yannakakis, Scheduling interval-ordered tasks, *SIAM Journal of Computing*, 8, (1979), 405-409
- [RLA94] Hesham El-Rewini, Theodore G. Lewis, Hesham H. Ali: *Task Scheduling in Parallel and Distributed Systems*, 1994 by PTR Prentice Hall, Inc. Englewood Cliffs, New Jersey 07632
- [StBo87] H. Stone and S. Bokhari, Control of distributed processes, *IEEE Computer*, (1987), 85-93.
- [Ston77] H. Stone, Multiprocessor scheduling with the aid of network flow algorithms, *IEEE Trans. Software Eng.*, (1977), 85-93.

- [TCR90] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, Introduction to Algorithms, MIT Press (1990).
- [Traj83] R. Tarjan, Data structures and network algorithms, SIAM, Philadelphia, 1983.
- [Ullm75] J. Ullman, NP-complete scheduling problems, Journal of Computer and System Sciences, 10, (1975), 384-393.