Computer Science Faculty Proceedings & Presentations

Department of Computer Science

11-2004

A Maximal Chain Approach for Scheduling Tasks in a Multiprocessor Systems

Sachin Pawaskar
University of Nebraska at Omaha, spawaskar@unomaha.edu

Hesham Ali
University of Nebraska at Omaha, hali@unomaha.edu

A MAXIMAL CHAIN APPROACH FOR SCHEDULING TASKS IN A MULTIPROCESSOR SYSTEM

Sachin Pawaskar and Hesham H. Ali
Department of Computer Science
University of Nebraska at Omaha
Omaha, NE 68182
sachinpawaskar@msn.com | hesham@unomaha.edu

ABSTRACT

Scheduling dependent tasks is one of the most challenging versions of the scheduling problem in parallel and distributed systems. It is known to be computationally intractable in its general form as well as several restricted cases. As a result, researchers have studied restricted forms of the problem by constraining either the task graph representing the parallel tasks or the computer model. Also, in an attempt to solve the problem in the general case, a number of heuristics have been developed. In this paper, we study the scheduling problem for a fixed number of processors m. In the proposed work, we approach the problem by recursively reducing the m-processor scheduling to (m-1)-processor scheduling until we apply the optimal two-processor scheduling algorithm when m equals two. This is accomplished by identifying a maximal chain C in the task graph G and merging the (m-1) processor scheduling of (G-C) and the 1-processor scheduling of C. A number of experiments were conducted to compare the suggested approach with the standard list-scheduling algorithm. Based on the outcome of the conducted experiments, the proposed algorithms outperformed or matched the performance of the list heuristic almost all the time.

KEY WORDS

Task Scheduling, Heuristics, Parallel Processing, Optimal algorithms.

1. Introduction

Scheduling is a classical field with several interesting problems and results. Due to its wide range of applications, the scheduling problem has been attracting many researchers from a number of fields. 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.

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 [1,2]. However, many heuristic methods have been used to deal with most general case of the problem. Such methods include traditional priority-based algorithms [3], task merging techniques [4], critical path heuristics [3,5]. In addition, distributed algorithms have been designed to address different versions of the scheduling problem [6].

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 (schedule) for managing the access to and the use of the resources by various consumers to optimize some desired performance measure such as the total service time (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.
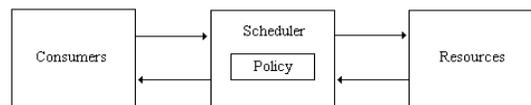

Figure 1: The Scheduling System

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. Performance and efficiency are two parameters used to evaluate a scheduling system. It's customary to evaluate a scheduling system based on the goodness of the produced schedule and the efficiency of the policy.

In this paper, we are concerned with scheduling dependent 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 to 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.

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 paper.

2. Basic Terminology & Problem Definition

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

Task Graph: A task graph G=(T,A) is a directed acyclic graph. For a pair of tasks $t_i, t_j \in T$, a directed edge $(i, j) \in A$ between the two tasks specifies that t_i must be completed before t_j can begin. Figure 2 shows a task graph.

Density or Sparseness: The density or sparseness of a graph G=(T,A) is computed as a ratio of the number of edges |A| in the graph as a percentage to the maximum

number of edges that graph can have which is of order (|T| * |T-1|) / 2. So a graph with density of 0.5 will have half the number of maximum edges possible for that 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. Given the task graph in Figure 2, 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 = (T, 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 is a subset of S'. A maximum chain is the maximal chain with the higher number of tasks. Given the task graph in Figure 2, a maximum chain consists of tasks 15, 14, 12, 11, 8, 4 and 1. Also, tasks 10, 5 and 3 form a maximal chain.
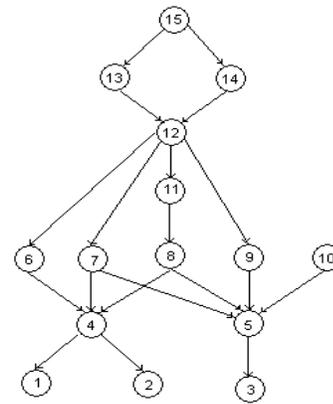

Figure 2: A Task graph

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.

For the rest of the paper, we assume that 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. As in the standard scheduling system, our system has four components: the target machine, the parallel tasks (represented as a task graph), the generated schedule and the performance criterion. The minimization of the schedule length is the performance criterion considered in our scheduling model.

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 known results as follows:

1) The NP-Completeness of several versions of the scheduling problems [1,3].
2) Optimal "efficient" algorithms, for solving restricted versions of the scheduling problems [2,3,8,9,10].
3) Heuristic algorithms for tackling more general cases of the scheduling problems [3,4,5,7].

Table 1 summarizes the complexity of several versions of the scheduling problem when 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. Note also that the results in Table 1 are obtained when communication costs are not considered. Forest and interval-order are special classes of task graphs. For more detailed definition and the formal discussion of NP-completeness please refer [1,3].

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	≥ 2	NP-complete
Opposing forest	Identical	Arbitrary	NP-complete
Interval order	Arbitrary	≥ 2	NP-complete
Arbitrary	Arbitrary	Arbitrary	NP-complete

Table 1: Complexity comparison of scheduling problem

As mentioned earlier a number of scheduling heuristic have been developed to deal with many versions of the scheduling problem. Among the developed heuristics, List scheduling has been used often due to its simplicity and over all good results. 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. In this paper, we will use the list scheduling heuristic to access the goodness of the proposed algorithm.

3. Proposed Solution

In this section, we will study the proposed maximal chain scheduling heuristic. As mentioned previously, there are many heuristic algorithms developed for dealing with the scheduling problem. Most of these heuristics perform well in some cases while performing poorly in others. The proposed scheduling algorithm employs a theoretical concept in dealing with the scheduling problem by using one of the few known optimal algorithms namely the 2-procesor scheduling algorithm. The algorithm selects a special maximal chain from the input task graph. Using the selected chain, the n-processor scheduling problem is reduced to two scheduling problems: 1) An (n-1)-processor scheduling problem, and 2) A simple1-processor scheduling problem. The maximal chain tasks are scheduled on 1-processor (P1) and the remaining tasks from the task graph are scheduled using the same algorithm to the remaining (n-1) processors. The two assignments, resulting from solving the two scheduling problems, are then merged together to satisfy the precedence relations and perform any needed reassignments of tasks for optimization purposes. The partitioning process is repeated recursively until we reach the base case of 2-processor scheduling for which we have well know optimal algorithm that we can apply as we discussed in the previous section.
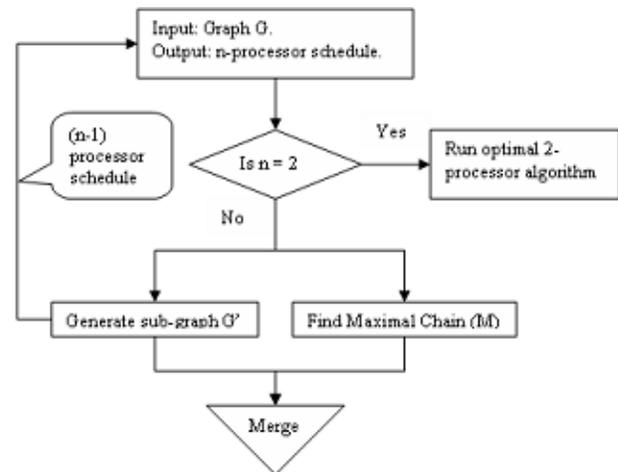

Figure 3: The Proposed Approach

The motivation for this approach stemmed from the fact that well-known polynomial (and optimal) algorithms are known for special cases of the scheduling problem. The algorithm uses a maximal clique since it does provides a lower bound on the schedule. The process of merging the (n-1) processor scheduling with the maximal chain is rather non-trivial since it needs to resolve any potential violations of the precedence relations. After merging the two schedules and resolving any violations, an optimizer/compacting routine is called to reduce the length of the obtained feasible schedule by moving tasks to appropriate slots without violating the task dependencies. To assess the performance of the proposed algorithm, we also implemented a basic standard well-known scheduling heuristic. We have selected the List scheduling heuristic for this purpose.

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. The target machine is composed of m identical fully connected processors. The input tasks are assumed to require the same amount of computation time and communication overhead among tasks assigned to different processors is ignored. The main objective function is to minimize the time of completion of the tasks to be scheduled; in other words the shortest schedule.

The details of the algorithm are given below. The basic algorithm consists of 3 different steps, the maximal chain algorithm, the 2-processor algorithm 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.

The Algorithm
Given a Task Graph G (T, A) where T is the number of tasks and A is the number of directed edges between the nodes. Also N is the number of processors.
Step 1:
If N = 2 go to step 2.
a) Given the Graph G = (T, A), assign a priority (label) to each task in $t \in T$ (Perform Algorithm Assign_Labels)
b) Find the Maximal Chain C for this task graph G (Perform Algorithm Generate_Maximal_Chain)
c) Generate the sub-graph $G_s = G - C$
d) Repeat Step 1, with $G = G_s$ and N = N – 1;
e) Merge the Maximal chain C and the schedule S for graph G_s for N processors. (Perform Algorithm Merge_Schedules)
Step 2:
Given task graph G' and N' = 2. Apply any optimal 2-processor scheduling algorithm (such as Coffman and Graham algorithm), which is as follows:
a) Assign lexicographical labels to all the tasks. (Perform Algorithm Assign_Labels)
b) Use the list $(t_n, t_{n-1} \dots t_1)$ where for all i, $1 \le i \le n$, $L(ti) = i$ 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 generation of the maximal chain
2) The optimal 2-processor 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,…,n} are assigned to each task in the task graph by the function L(*) as explained below.

Algorithm (Assign_Labels)
1) Assign the number 1 to one of the terminal tasks.
2) Let labels 1, 2… j – 1 be assigned. Let S be the set of unassigned tasks with no unlabelled successors.
   a. We next select an element of S to be assigned label j.
   b. For each node x in S, define L(x) as follows: Let y_1, y_2…y_k be the immediate successors of x. Then L(x) is the decreasing sequence of integers formed by ordering the set {L(y_1), L(y_2)…L(y_k)}.
   c. Let x be an element of S such that $\forall$ x' in S, L(x) ≤ L(x') (lexicographically).
   d. 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 below.

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

Algorithm (Merge_Schedules)
1) Let C 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 relations 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 relations.

Example

To understand the maximal chain scheduling heuristic algorithm, let us examine the task graph as shown in Figure 4. 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.
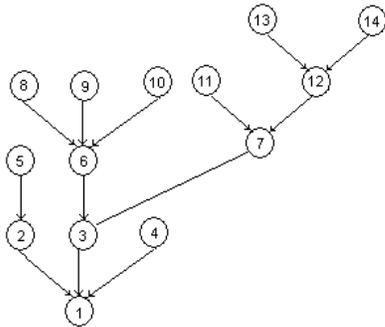

Figure 4: A task graph

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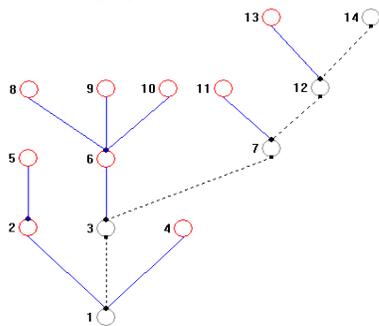

Figure 5: Maximal chain for task graph in Figure 4

We perform the optimal 2-processor scheduling algorithm 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 6. 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 6: 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 7, which is an optimal feasible schedule.

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

Figure 7: 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 4 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 8. 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 8: Schedule after Merge and Check Violation/Feasibility and Optimization.

The Optimal maximal chains Conjecture

The underlying principle behind the optimal 2-processor scheduling algorithm can be viewed as identifying the maximal chain (clique) of tasks whose removal guarantees the minimum 1-processor scheduling of the remaining tasks. Then, an optimal 2-processor schedule can be obtained by merging the chain with the minimum 1-processor schedule of the remaining tasks. In this case, identifying such a chain is easy since it has to be a maximum chain or a chain with the maximum number of tasks. If we take this principle one step further, it can conjectured that for a set of tasks T, if we identify a maximal chain C in an n-processor scheduling problem whose removal results in a minimum (n-1)-processor scheduling of the remaining tasks T-C, then an optimal n-processor schedule can be obtained by merging C with the optimal (n-1)-processor schedule of the remaining tasks. This approach would require a) finding such a clique; and b) design an optimal merge algorithm. We have tested this concept by generating all maximal chains and trying

several merge algorithms. We used the "All maximal chains" routine for testing purposes on graphs with small number of nodes. Applying this approach on many random task graphs produced the optimal schedule almost all the times. Clearly, generating all maximal chains will strip the approach its desired polynomial complexity. We are currently working on developing a polynomial algorithm to find such a clique and on developing a refined merge algorithm [11]. The above conjecture provided the basic foundation for the developed heuristic whose results are reported in the next section.

4. Implementation and Results

Various experiments were run on the maximal scheduling heuristics and the list scheduling heuristic using different graphs. The two most important properties of the graphs that the algorithms were tested against were:

a) Number of nodes in the graph, and
b) The Density/Sparseness of the graph

The density of the graph varies from 0.1 to 0.9, it implies that the graphs having 0.1 densities would have fewer edges and hence less density and as the density increases the number of edges increase and so the density of the graph increases. 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 tried all densities, but it is important to note that in most real situations we will encounter graphs with lower densities, in which case our algorithm produces better results. Also note that the same task graphs were used for comparing our algorithm with the list-scheduling algorithm.

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 that 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 Gaaa_b, where aaa denotes the number of nodes in the graph and b denotes the density 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 Gaaa_b_c, where aaa denotes the number of nodes in the graph and b denotes the density of the graph and c 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.

All the graphs used in these experiments were randomly generated by a random graph generator program, which was specifically written for this purpose.

It must be noted that the graphs generated had transitive precedence edges, which implies if A → B and B → C, then even though by transitivity it implies that A → 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.
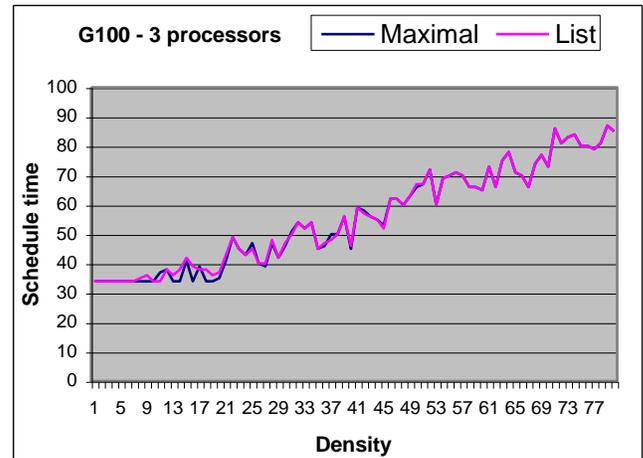

Figure 9. Graphs with 100 nodes on 3 processors

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

1) The maximal chain scheduling heuristic performs slightly better that 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 that 0.4 for most of the cases.
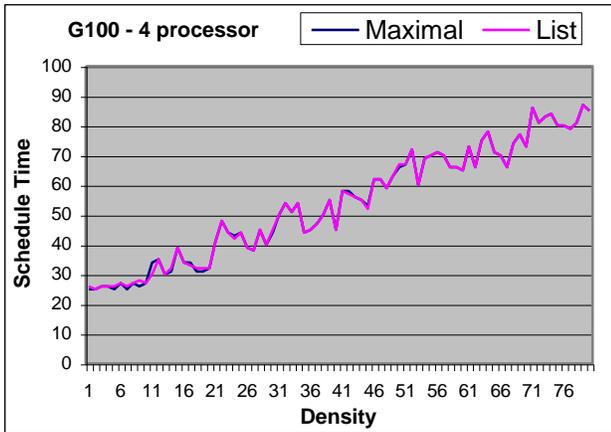
Figure 10. Graphs with 100 nodes on 4 processors

From the above experiments and results we can draw the following:
1) The maximal chain scheduling heuristic performs slightly better that 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 that 0.3 for most of the cases.
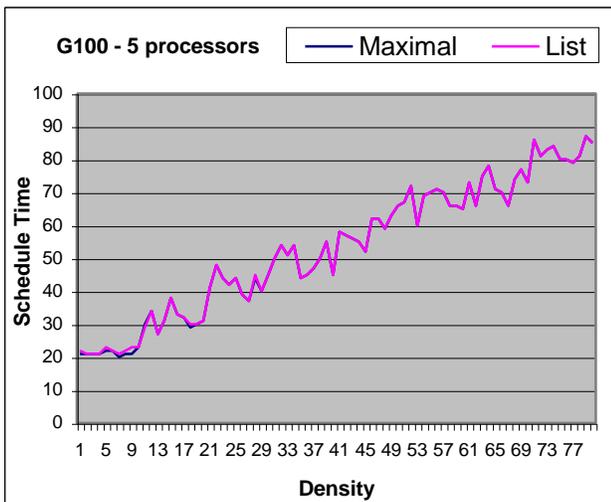

Figure 11. Graphs with 100 nodes on 5 processors

From the above experiments and results we can draw the following:
1) The maximal chain scheduling heuristic performs slightly better that 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 that 0.2 for most of the cases.

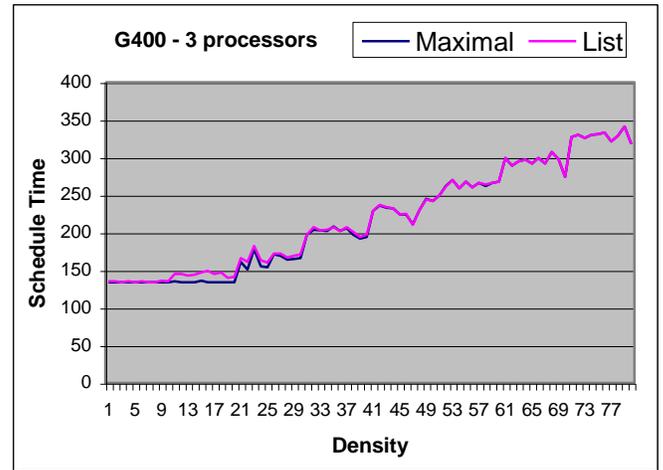From the above experiments and results we can draw the following:

Figure 12. Graphs with 400 nodes on 3 processors

1) The maximal chain scheduling heuristic performs slightly better that 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 that 0.4 for most of the cases.
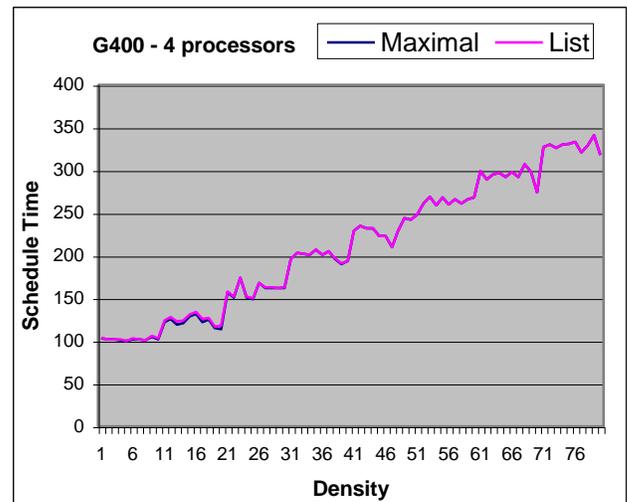

Figure 13. Graphs with 400 nodes on 4 processors

From the above experiments and results we can draw the following:
1) The maximal chain scheduling heuristic performs slightly better that 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 that 0.3 for most of the cases.

Overall, from the above experiments, we can conclude the following:
a) 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.

b) It can also 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.
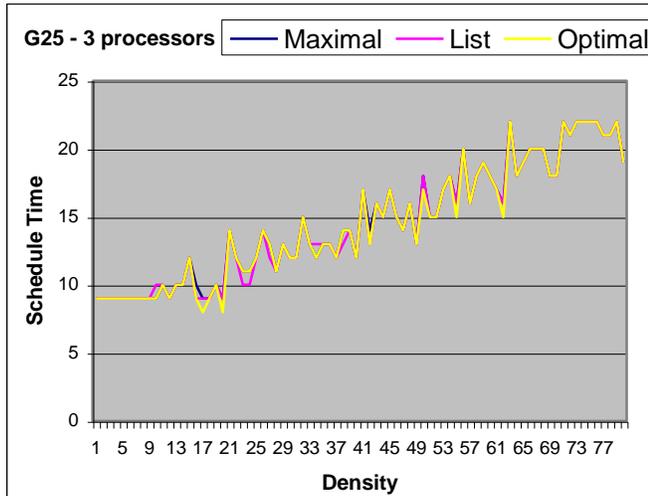

Figure 14. Graphs with 25 nodes on 3 processors (with optimal algorithm)

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

1) 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 chains algorithm, Maximal chain scheduling heuristic performs and the List scheduling heuristic has the same performance when the density/sparseness of the graphs is greater that 0.4 for most of the cases.

5. Conclusions

In this paper we present a novel approach for addressing the n-processor scheduling problem by recursively reducing the problem to the polynomial problem of finding a 2-procesor schedule. We compare the performance of the proposed algorithm, Maximal-Chain scheduling, with the performance of the standard List scheduling algorithm. The introduced approach outperforms the List scheduling algorithm when the density of the graphs is between 0.1 and 0.4 for most of the cases. But for density of the graphs greater that 0.5 both of them have the same performance.

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 units. The All maximal chain scheduling algorithm performed slightly better or the same as the List scheduling

It can be concluded that when the density 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 schedule. 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.

Future research efforts could further investigate enhancements to the Merge routine for merging the maximal chain and the (n-1) processor schedule. One could investigate new approaches for the Merge routine to always obtain an optimal solution. 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.

6. References:

[1] J. Ullman, NP-complete scheduling problems, Journal of Computer and System Sciences, 10, 384-393, 1975.

[2] 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, John Wiley & Sons, A Wiley-Inter-Science publication, 1976.

[3] Hesham El-Rewini, Theodore G. Lewis, Hesham H. Ali: Task Scheduling in Parallel and Distributed Systems, PTR Prentice Hall, Inc. Englewood Cliffs, New Jersey 07632. 1994.

[4] Peter Aronsson and Peter Fritzson: Task Merging and Replication using Graph Rewriting, Tenth International Workshop on Compilers for Parallel Computers, Amsterdam, the Netherlands, Jan 8-10, 2003

[5] A. A. Khan, C. L. McCreary and M. S. Jones, A Comparison of Multiprocessor Scheduling Heuristics, International Conference on Parallel Processing, 1994.

[6] Rong Xie, Daniela Rus and Cliff Stein: Scheduling Multi-Task Agents. In Proceedings of the Fifth IEEE International Conference on Mobile Agents, pages 260-276, Atlanta, Georgia, December, 2001.

[7] H. Ali and H. El-Rewini, Task allocation in distributed systems: A split-graph model, Journal of Combinatorial Mathematics and Combinatorial Computing, 14, 15-32, 1993.

[8] C. Papadimitriou, and M. Yannakakis, Scheduling interval-ordered tasks, SIAM Journal of Computing, 8, 405-409, 1979.

[9] 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.

[10] H. Stone, Multiprocessor scheduling with the aid of network flow algorithms, IEEE Trans. Software Eng., 85-93, 1977.

[11] Hesham Ali, On finding an optimal maximal chain in n-processor scheduling, in preparation.