REDUCING PROCESSOR COMMUNICATION OVERHEAD IN MULTIPROCESSOR SCHEDULING

By

SURAJ SRINIVAS BHAT

Bachelor of Science

Bangalore University

Karnataka, India

1994

Submitted to the faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May 2000

REDUCING PROCESSOR COMMUNICATION OVERHEAD

IN MULTIPROCESSOR SCHEDULING

Thesis Approved:

Mansur H. Samadzadeh

Thesis Advisor

Blayne E. Mayfield

Jacques E. LaFrance

Wayne B. Powell

Dean of the Graduate College

PREFACE

The parallelism within an algorithm at any stage of execution can be defined as the number of independent operations that can be performed in parallel. These independent operations can be simultaneously scheduled on multiple processors. The scheduling algorithm used for this purpose influences the time taken to complete the entire set of operations. The efficiency and overhead of such a schedule depends on many issues such as task decomposition, allocation of tasks to processors, and processor communication overhead. In general, a given problem can be decomposed into smaller tasks/processes/operations, with interdependencies among them (i.e., the precedence relation among the tasks). Using known algorithms, the maximum possible parallelism can be extracted from these interdependencies. Subsequently, the tasks can be allocated to a set of available processors. The objective of the proposed thesis was to improve such parallel executions (in terms of multiprocessor performance) by reducing the communication overhead among processors. Research has been mainly conducted on deterministic scheduling algorithms for multiprocessors. These algorithms can schedule a set of tasks, given in the form of a directed acyclic graph (DAG), on a given number of processors. Communication overhead between processors can be an important distinguishing factor among possible schedules of a given task system. The purpose of the thesis was to reduce the communication overhead between processors, which could result in faster execution times for algorithms.

A simulation program was developed to study the effects of reducing processor communication overhead in multiprocessor scheduling. The program implements the proposed algorithm (algorithm D) which generates a schedule for a task system based on each task's predecessors. The test suite consisted of task systems obtained from various sources, but mostly from a task system generator program, which was designed and implemented as part of this work. It was found that algorithm D reduces the inter-processor communication overhead in most of the cases.

ACKNOWLEDGEMENTS

I would like to convey my sincere appreciation to my major advisor Dr. Mansur H. Samadzadeh for his supervision, advice, patience, guidance, and constructive criticism. His words of encouragement and moral support are greatly appreciated. My sincere appreciation extends to my other committee members Drs. Blayne E. Mayfield and Jacques LaFrance for serving on my committee. Their time and effort are greatly appreciated.

I would like to extend special appreciation and gratitude to my parents, who always believed in me and my abilities, for their moral and financial support and encouragement. I am also grateful to my friends and colleagues for their invaluable suggestions and support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Parallel processing offers one possible solution to meeting the ever-increasing demand for computational speed in solving complicated problems. Parallel processing is generally used to increase the overall throughput of a system. In parallel processing that utilizes the method of function parallelism, a program is divided into a number of processes, tasks, or threads, which can run concurrently on the available processors. All processors in a parallel processing system normally reside in a single machine, unless the system is distributed or loosely coupled.

Implementing parallel algorithms for a given problem, in order to attain high performance has multiple parts to it. The first step in the implementation of parallel algorithms is decomposition. Problem decomposition involves identifying sequential units of computation, called "tasks", in the given problem. The second step involves the identification of dependencies among the decomposed tasks and thus determining the extent and degree of parallelism existent in the given problem. The last step involves the allocation of the tasks to the available processors.

A problem partitioned into tasks forms a task system. A task system is represented by a directed acyclic graph (DAG) or a rooted tree. A DAG consists of a set of nodes and a set of directed arcs (edges) connecting them. A rooted tree is a directed graph in which

each node has at most one successor (the root has no successors) and any number of predecessors (including zero predecessors for the leaves).

Having decomposed a problem into a task system, the individual tasks must be assigned to the set of available processors, using a scheduling algorithm. The goal is to reduce, and ideally to minimize, the time taken to complete the execution of the task system. Tasks that are on the same path of a DAG may not in general be assigned to the same processor one after another. Since a task needs to pass along data to its successors, a schedule will involve a certain amount of communication overhead in terms of information transferred among processors. The thesis focuses on reducing the communication overhead among processors. Since this type of inter-processor communication in general comprises unproductive time, curtailing the communication between processors can lead to a potentially significant gain by eventually increasing the throughput of the system.

The rest of the thesis is organized as follows. Chapter II contains basic algorithms used. Chapter III describes the design and implementation issues of the simulation program. Chapter IV describes the experimentation conducted and chapter VI contains the summary, conclusion and future work.

CHAPTER II

BASIC ALGORITHMS

This chapter reviews the basic model used for task system representation in the context of three scheduling algorithms. The three basic algorithms are described in the following subsections, with each algorithm having advantages over the preceding one.

2.1 Scheduling Algorithm A

A task system given in the form of a rooted tree can be scheduled by using Algorithm A. This algorithm (Figure 1) follows the same broad approach used in Hu's algorithm for multiprocessors [Mandyam and Samadzadeh 92b].

Algorithm A has certain limitations. Primarily, it is restricted to scheduling task systems that are rooted trees. However, a DAG need not necessarily be a rooted tree, because the nodes of a DAG can have more than one successor. A second restriction is that the rooted tree should have nodes with equal weights, which need not be true for a general DAG.

2.2 Scheduling Algorithm B

To address the problems of Algorithm A, Algorithm B was developed [Hu 61] [Mandyam and Samadzadeh 92b]. This algorithm (Figure 2) first preprocesses a given

DAG. A rooted tree is obtained with equally weighted nodes by replication of some of the nodes in the DAG.

Algorithm B presumes that a given task graph has already been preprocessed so that a corresponding rooted tree (which is equivalent to the original DAG) with equally weighted nodes, is at hand.

2.3 Scheduling Algorithm C

The only limitation of Algorithm B is its replication (repetition) of nodes during the preprocessing phase. To overcome this problem, a multiprocessor scheduling algorithm (Algorithm C) was developed [Mandyam and Samadzadeh 92a] (see Figure 3). This algorithm schedules a task system, given in the form of a DAG (and not necessarily a rooted tree), on multiprocessors. The advantage of Algorithm C over Algorithm B is that Algorithm C schedules the DAG directly, and the intermediate phase of converting a DAG with multiple successor nodes into a rooted tree containing only equally weighted nodes, is therefore removed.

The tasks are labeled using the labeling scheme mentioned below.
Every node of a given graph is assigned a label as follows.

- The root node label is set to 1.
- The label of any other node is set to 1 plus the label of its single successor node.

Using this labeling arrangement, a label table is created for the rooted tree.

Let L denote the value of the maximum label in the label table, w_i denotes the subset of the nodes or tasks with label i and $|w_i|$ denotes the number of tasks in w_i.
The width of the graph is defined as

$$w_G = \max\{|w_1|, |w_2|, \ldots, |w_L|\}$$

The tasks are then grouped into sets, for scheduling on p processors, using the following algorithm.

```
L1:
   if |w_i| ≤ p for i = L, ..., 1 then
      goto L3
   else if for some i, |w_i| > p then
      n = i;
L2:
   if n ≠ L then
       find a internal node from w_n that does not have any
       predecessors in w_{n+1};
       change the node's label from n to n+1;
   end /* then */
   if n = L then
        select any node from the set w_L as the victim;
        /* since all are leaf vertices in w_L
        change the node's label from L to L+1 */
        increment L by 1;
    end /* then */
    goto L1;
L3:
   form the schedule as follows
   for i = 1, 2, ..., L do
      execute a task in the set w_i in the (L-i+1)th unit of
      time on one of the p processors;
      /* if fewer than p tasks are available, the
      remaining processors will be idle */
   end  /* do */
```

Figure 1. Algorithm A. Scheduling a Rooted Tree on *p* Processors [Hu 61]

Consider a rooted tree with replicated nodes that corresponds to a given DAG that has been preprocessed. Each node in the graph is assigned a label as follows.

- The root node label is set to 1.
- The label of any other node (including the replicated nodes) is set to 1 plus the label of its single successor node.

Using this labeling arrangement, a label table is created for the rooted tree.

Let L denote the value of the maximum label in the label table, let w_i denote the subset of the nodes or tasks with label i, and let $| w_i |$ denote the number of tasks in w_i.
The width $\mathbf{w_G}$ of the graph is defined as in Algorithm A.

repeat
Select at most p tasks from w_i for i = L, ..., 1 such that:
they are leaf nodes **or**
all their predecessors have been assigned in an interval previous to the current time interval;

if the predecessor of a task is a repeated node, **then**
any counterpart of the repeated node can be considered the predecessor of the task;

if a repeated node needs to be selected, **then**
if any of its counterparts has been selected earlier or in the current interval, **then**
discard it from the current set w_i;
else select it for the current time interval;
end;

if all tasks in the set w_i have been tried for selection,
then examine the next set w_{i+1};

Schedule p (or fewer) tasks on p processors during the current interval;
until (all tasks have been scheduled);

Figure 2. Algorithm B. Scheduling a Rooted Tree with Repeated Nodes on *p* Processors (originally in [Hu 61] and modified in [Mandyam and Samadzadeh 92b])

The tasks are labeled according to the same labeling scheme used for Algorithm A. The tasks are grouped into sets, for p processors, using the following algorithm.

Let L denote the value of the maximum label in the label table.

```
L1:
    if |w_i| ≤ p for i = L, ..., 1 then
       goto L3
    else if for some i, |w_i| > p then
       n = i;
L2:
    if n ≠ L then
        find an internal node from w_n that does not have any
        predecessors in w_{n+1};
        if no such node is available in w_n then
           n = n + 1;
           goto L2
        end /* then */
        change the node's label from n to n+1;
    end /* then */
    if n = L then
         select any node from the set w_L as the victim;
         /* all nodes in w_L are leaf nodes */
         /* change the related node's label from L to L+1 */
         increment L by 1;
     end /* then */
     goto L1;
L3:
    form the schedule as follows
    for i = 1, 2, ..., L do
       execute a task in the set w_i in the (L-i+1)th unit of
       time on one of the p processors;
       /* if fewer than p tasks are available, the
       remaining processors will be idle */
    end  /* do */
```

Figure 3. Algorithm C. Scheduling a DAG on *p* Processors Directly (originally in [Hu 61] and modified in [Mandyam and Samadzadeh 92b])

CHAPTER III

DESIGN AND IMPLEMENTATION ISSUES

This chapter describes the design and implementation issues of the simulation program (i.e., the software tool created as part of this thesis). The simulation program was implemented in C under Solaris 7 (SunOS v5.7).

3.1 Overview of the Simulation

The simulation program was developed as a tool that aids in the study of the effects of inter-processor communication overhead in multiprocessor scheduling. The input to the simulation is a random task system (DAG) obtained by using the random task generator program, which was also designed and implemented as part of this research effort. Task systems from various other sources, including those from previous research papers [Hu 61] [Samadzadeh 92] [Ananthaneni 97] and the Internet [Battista, et al. 96] were also used as inputs to test the program, to provide greater variability.

The random task system generator program creates task systems by placing dependencies among tasks that are located at different levels of the task system. The number of tasks in the system is used as a primary constraint. All other elements, including the number of levels in the task system, the number of tasks in each level, the number of dependencies (i.e., the number of successors of each task in the task system except the root), and the successor tasks, are randomly generated.

Consider the DAG given in Figure 4. The format of the input file for the given DAG is shown in Figure 5. In Figure 5, the first line indicates the number of tasks in the task system. The rest of the lines contain task name, task level, task weight, number of successors, and successor names, for all tasks in the system.

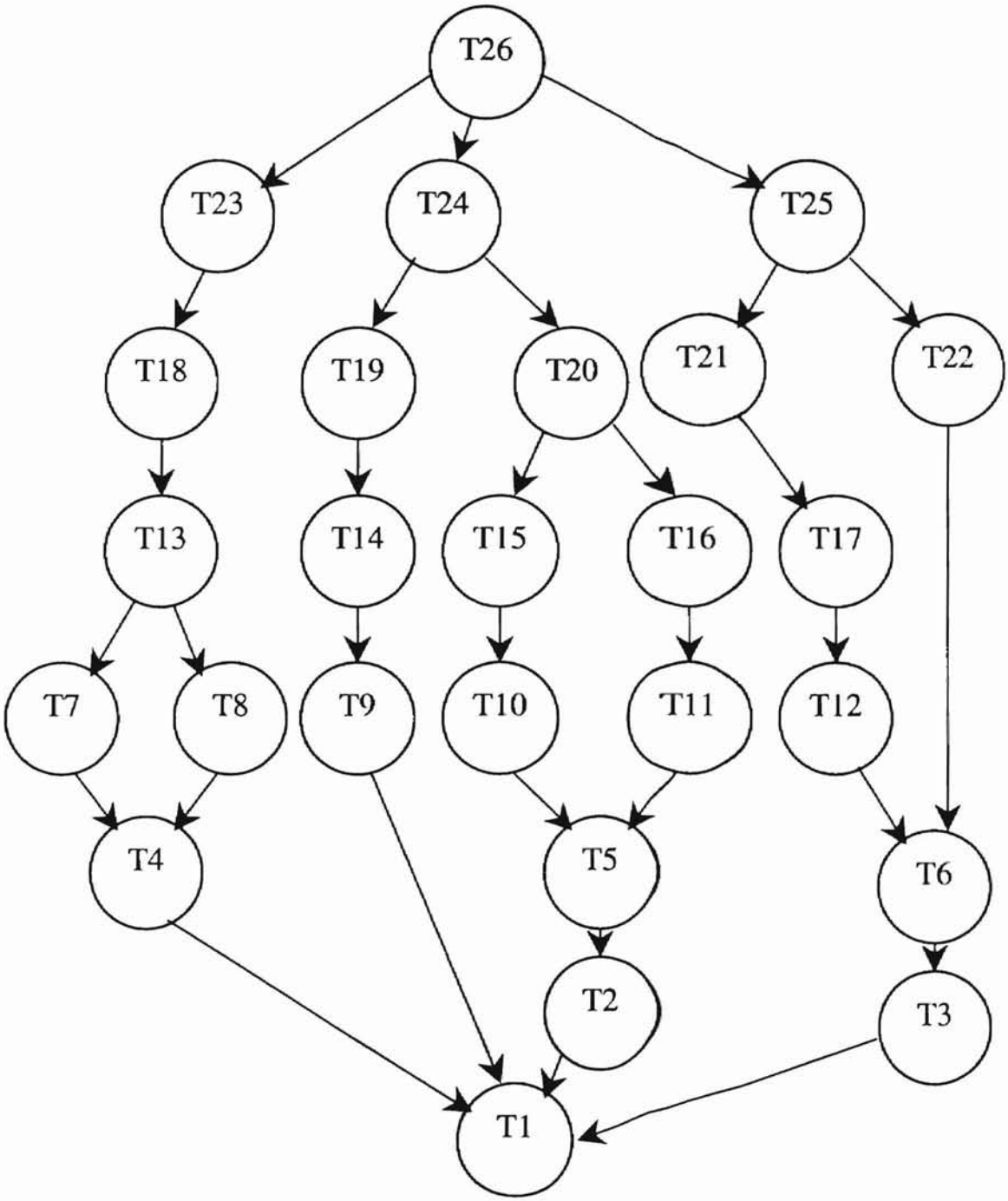

Figure 4. Example of a DAG [Mandyam and Samadzadeh 92b]

For the example illustrated in Figure 5, the task names range from T1 to T26. Task T1 is always the root and has no successors. Task T26, which is on level 8, can have successors from tasks T25 to T1, which are all in levels below level 8. Thus the maximum number of successors of any other task, ranges from one to the total number of tasks in the levels below the level of the task under consideration.

```
26
T1 1 2 0
T2 2 1 1 T1
T3 2 1 1 T1
T4 2 2 1 T1
T9 2 2 1 T1
T5 3 1 1 T2
T6 3 1 1 T3
T7 3 2 1 T4
T8 3 2 1 T4
T14 3 2 1 T9
T10 4 2 1 T5
T11 4 2 1 T5
T12 4 2 1 T6
T13 4 2 2 T7 T8
T19 4 2 1 T14
T22 4 2 1 T6
T15 5 2 1 T10
T16 5 2 1 T11
T17 5 2 1 T12
T18 5 2 1 T13
T20 6 2 2 T15 T16
T21 6 2 1 T17
T23 6 2 1 T18
T24 7 2 2 T19 T20
T25 7 2 2 T21 T22
T26 8 2 3 T23 T24 T25
```

Figure 5. Format of the Input File for the DAG in Figure 4

3.2 Algorithm Implementation

The program's algorithm consists of four main functions. They are gen_sets(), Hu_schedule(), random_schedule(), and the proposed_schedule(). The function gen_sets() implements Hu's algorithm (actually Algorithm C). All functions schedule the tasks on p processors and compute the inter-processor communication overhead incrementally when each task is executed. Each function calls the sub program compute_overhead() to compute the inter-processor communication overhead. In Hu's algorithm, each task from each task set (a set of tasks that do not have dependencies among them) is selected in the order of its appearance in the task set and is assigned to the first available processor.

The function random_schedule() implements the random algorithm. In this function each task from each task set is chosen and is assigned to a randomly selected processor. For randomizing the schedule, the function rand() provided by the C compiler is used.

The function proposed_schedule() implements the proposed algorithm shown in Figure 6, and also the worst case algorithm. This function schedules the tasks based on their predecessors already assigned to the processors.

Each task from each set is selected in the order of its appearance in the task set. For the best case, each task from each set is assigned to a processor that has the highest number of that task's predecessors. For the worst case, each task from each set is assigned to a processor that has the lowest number of its predecessors or zero predecessors assigned. For the best case and the worst case, the function calls the sub function CheckForPred() to find the processor to which the highest number of

predecessors of a task have been assigned, and the lowest number of predecessors of a task or no predecessors of a task have been assigned, respectively. Thus the proposed algorithm attempts to schedule tasks that are on the same path of a DAG to the same processor one after another, to the extent possible, to reduce the inter-processor communication overhead incurred. The worst case algorithm on the other hand, attempts to schedule tasks that are on the same path of a DAG to different processors, to the extent possible, to increase the inter-processor communication overhead incurred.

The program uses four main structures. The structure Node stores attributes of each task in the DAG. These attributes include task name, task weight, task level, number of successors of the task, number of predecessors of the task, successor names, and predecessor names.

The attributes of each level in the DAG are stored in the Width structure. The processor structure is used to execute the tasks scheduled on each processor. Considering the worst case, the memory allocated to the Width and processor structure is the same as the memory allocated to the Node structure, i.e., each task in the DAG has the maximum number of successors possible. The precedence structure is used to store the highest and the lowest number of predecessors of a task assigned for the given number of processors. This structure is used by the function proposed_schedule() for the best case schedule for and the worst case schedule.

The tasks are labeled using the labeling scheme mentioned below.
Every node of a given graph is assigned a label as follows.

- The root node label is set to **1**.
- The label of any other node is set to 1 plus the label of its single successor node.

Using this labeling arrangement, a label table is created for the rooted tree.

Let L denote the value of the maximum label in the label table, w_i denotes the subset of the nodes or tasks with label i, and $|w_i|$ denotes the number of tasks in w_i.

The width of the graph is defined as

$$w_G = \max\{|w_1|, |w_2|, \ldots, |w_L|\}$$

The tasks are then grouped into sets, for scheduling on p processors, using the following algorithm.

```
L1:
    if |w_i| ≤ p for i = L, ..., 1 then
       goto L3
    else if for some i, | w_i| > p then
       n = i;
L2:
    if n ≠ L then
        find an internal node from w_n that does not have any predecessors in w_{n+1};
        change the node's label from n to n+1;
    end /* then */
    if n = L then
         select any node from the set w_L as the victim;
         /* since all are leaf nodes in w_L, change the node's label from L to L+1 */
         increment L by 1;
     end /* then */
     goto L1;
L3:
    form the schedule as follows
    for i = 1, 2, ..., L do
       assign a task in the set w_i to one of the p processors which has the
       highest number of its predecessors assigned;
       if the numbers of predecessors of the task are equally assigned to more
       than one processor, select any one of the processors with
       predecessors assigned;
       compute the communication overhead for task execution;
       execute a task in the (L-i+1)th unit of time;
        /* if fewer than p tasks are available, the remaining processors will be idle */
    end  /* do */
```

Figure 6. Proposed Algorithm. Scheduling a DAG on p Processors

CHAPTER IV

EXPERIMENTATION

This chapter contains a description of the experiment conducted using the simulation tool. In the sets of runs comprising the experimentation, the input task systems (i.e., the test suite) were obtained by using the random task system generator which was designed and implemented as part of this thesis work. Task systems from various other sources, including those from previous research papers [Hu 61] [Samadzadeh 92] [Ananthaneni 97] and the Internet [Battista, et al. 96] were also used as input.

Twenty task system sets (each being a set of task systems containing the same number of tasks) were generated using the random task system generator, with each task system set containing 20 task systems. The number of tasks in each task system ranged between 25 (the first task set) to 120 (the last task set), in steps of 5. In each task system, the number of levels in the DAG, the number of tasks in each level, the number of successors, and the successor names were randomly generated. The number of levels in each task system ranged from 2 to number of tasks in the system. The number of tasks in each level ranged from 1 to the number of tasks remaining to be assigned minus the number of levels remaining. The number of successors of each task (except the root) ranged from 1 to the total number of tasks in the levels below the level of the task under consideration. Each task had at least one successor in the level just below itself.

The simulation program was executed several times for each task system, by varying the number of processors used from 2 to the maximum of 50 and the maximum width of the task systems. Fifty was the maximum number of processors used if the maximum width of the task system was greater than 50. Although these numbers were arbitrarily chosen, the idea was to cover as much of the spectrum of reasonable situations as possible.

The task systems were scheduled using Hu's algorithm, the random algorithm, algorithm D, and the worst case algorithm. The inter-processor communication overhead was calculated as the task systems were scheduled on the processors according to the schedule generated by each algorithm. In very few cases Hu's algorithm and the random algorithm performed better in reducing the inter-processor communication overhead, than algorithm D.

For the same schedule length, if tasks that are on the same path of a DAG are assigned to the same processor one after another to the extent possible, the inter-processor communication overhead for the task system will be less than other alternatives. Algorithm D assigns each task from each task set (i.e., a set of tasks in a task system that do not have dependencies among them) to the first available slot for a given number of processors in the order of their appearance in the task set with the specific processor selected based on where each task's predecessor has been scheduled previously. Algorithm D considers the past scheduling decisions to reduce the inter-processor communication overhead. This scheme sometimes proves to be less efficient than assigning tasks using the random algorithm. Hu's algorithm selects each task from each task set (i.e., a set of tasks in a task system that do not have dependencies among them) in the order of their appearance in the task set for assignment, and the first

available processor is selected for all the available times. Thus sometimes Hu's algorithm (first available task selection and processor selection) and the random algorithm (arbitrary task selection and processor allocation) assigns tasks that are on the same path of a DAG to the same processor, thus ending up with less overhead than algorithm D.

Figure 7 through 10 show the average inter-processor communication overhead with increasing number of tasks executed (25 to 120 in steps of 5) on a given number of processors (2 to 12) as scheduled by Hu's algorithm, the random algorithm, algorithm D and the worst case algorithm. With the increase in the size of the task system executed, the average inter-processor communication overhead was generally found to be increasing. As the figures suggest, Hu's algorithm and algorithm D performed better than the random algorithm and the worst case algorithm in terms of the communication overhead incurred.

Figure 11 shows the average inter-processor communication overhead for all the task systems executed on a given number of processors in the range 2 to Wmax (maximum width of each task system), where Wmax is less than or equal to 50 as scheduled by Hu's algorithm, the random algorithm, algorithm D, and worst case algorithm. The worst case algorithm and the random algorithm behaved expectedly more erratically than Hu's algorithm and algorithm D. Also, as indicated by the figure, the average inter-processor communication overhead generated by algorithm D was less than the average inter-processor communication overhead generated by the other three algorithms.

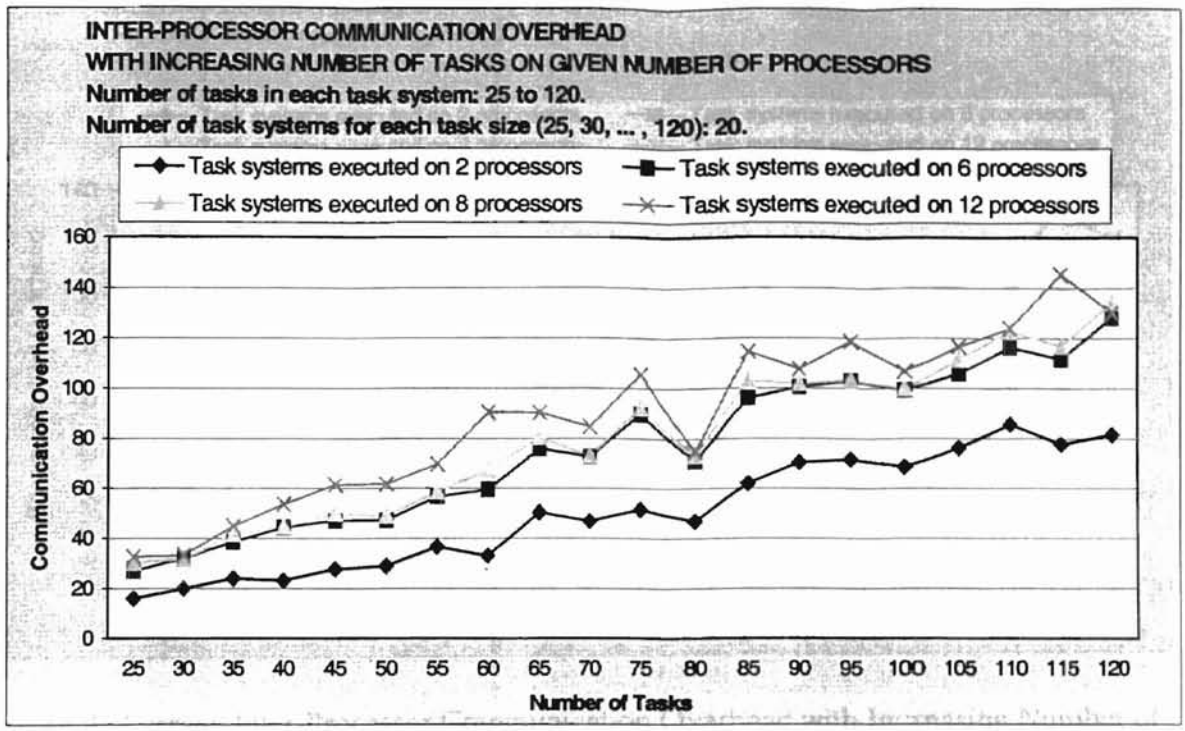

Figure 7. Average Inter-Processor Communication Overhead with Increasing number of Tasks Executed on the Given Number of Processors by Hu's Algorithm

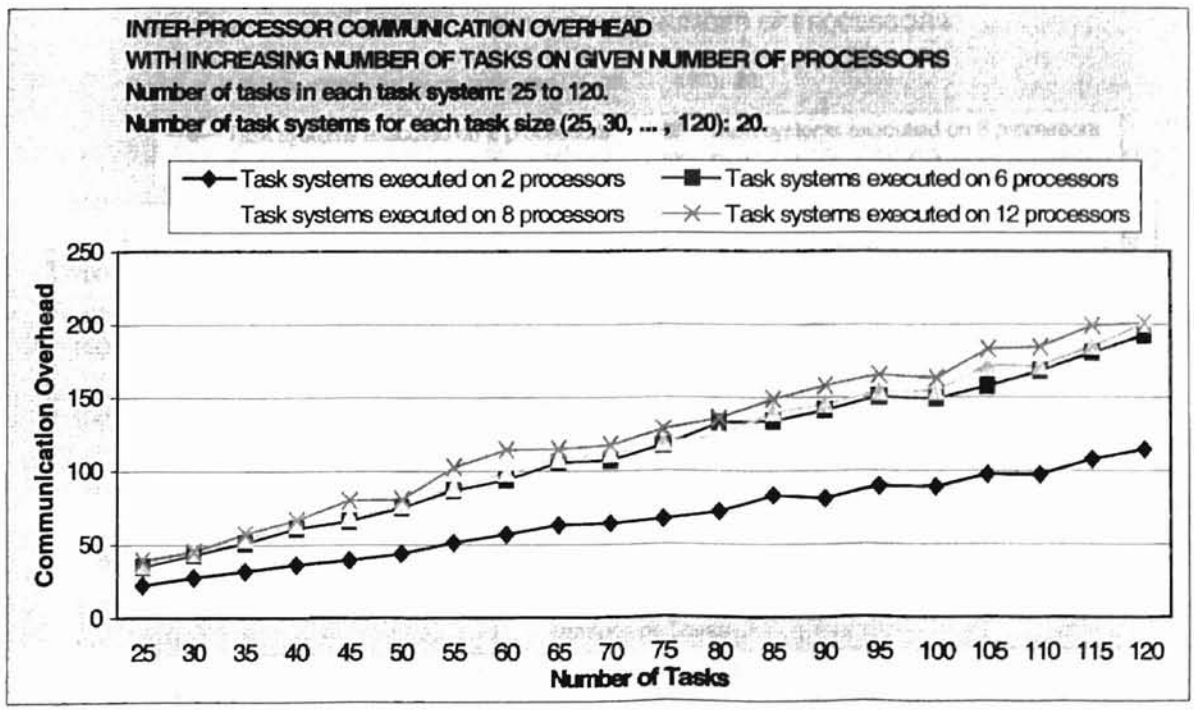

Figure 8. Average Inter-Processor Communication Overhead with Increasing number of Tasks Executed on the Given Number of Processors by the Random Algorithm

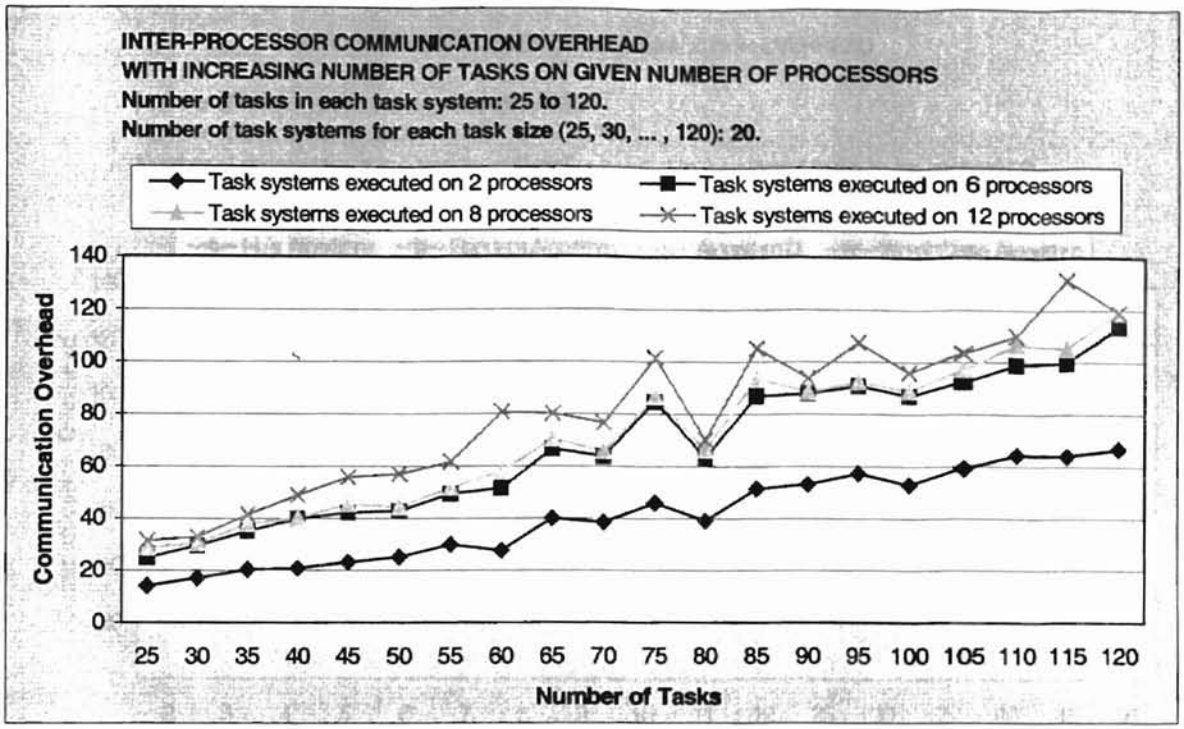

Figure 9. Average Inter-Processor Communication Overhead with Increasing Number of Tasks Executed on the Given Number of Processors by Algorithm D

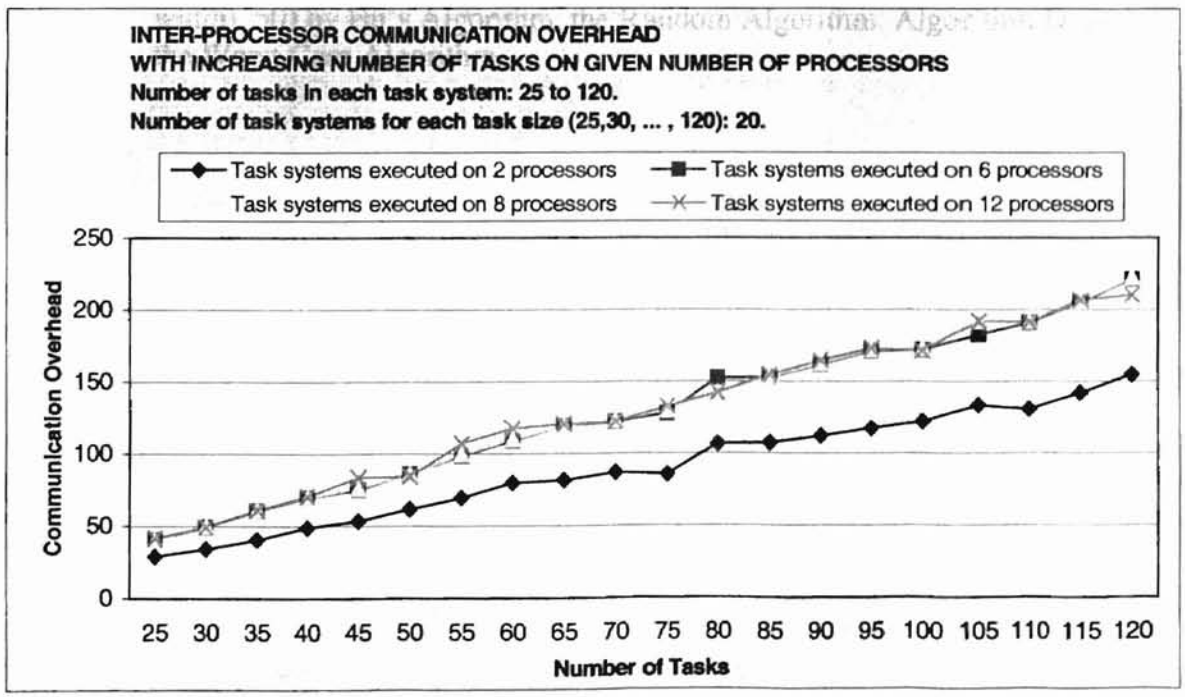

Figure 10. Average Inter-Processor Communication Overhead with Increasing Number of Tasks Executed on the Given Number of Processors by the Worst Case Algorithm

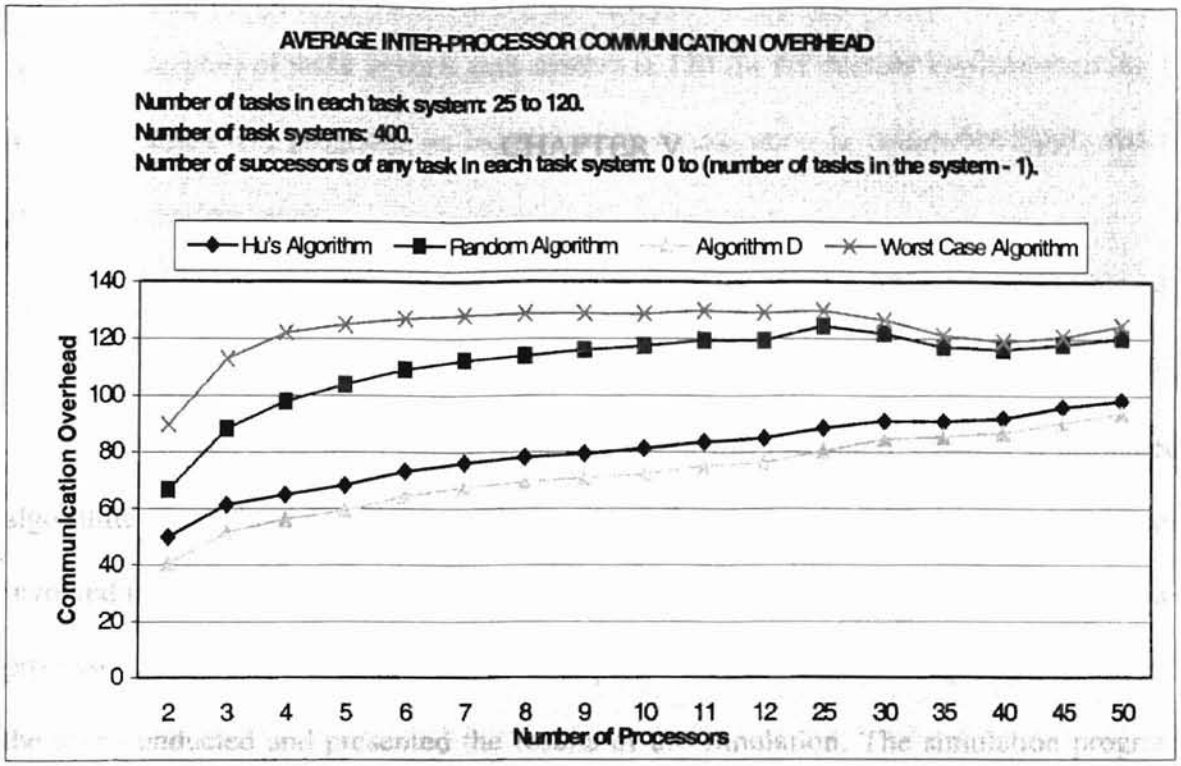

Figure 11. Average Inter-Processor Communication Overhead When Executed on a Given Number of Processors in the Range 2 to Max(**W**max (maximum width), 50) by Hu's Algorithm, the Random Algorithm, Algorithm D, and the Worst Case Algorithm

CHAPTER V

SUMMARY AND FUTURE WORK

In Chapter I the main objective of this thesis was presented. Chapter II presented algorithms A, B, and C. Chapter III discussed the design and implementation issues involved in this work including the proposed algorithm (algorithm D) for reducing inter-processor communication overhead in multiprocessor scheduling. Chapter IV described the tests conducted and presented the results of the simulation. The simulation program takes a random task system as input, schedules and executes it according to Hu's algorithm, a random algorithm, algorithm D, and the worst case algorithm, and calculates the communication overhead for each schedule as the tasks are allocated to the available processors.

It was found that algorithm D reduces the inter-processor communication overhead in most of the cases and the percentage reduction in inter-processor communication overhead was generally independent of the number of tasks in a task system. The overhead introduced by algorithm D is the time complexity of selecting a processor for the next schedulable task and the space complexity of keeping enough information around to do so. Algorithm D can be further improved by introducing more heuristics into the process of selecting a processor for a task ready to be dispatched.

It should be noted that algorithm D does not always produce the optimal schedule for a given task system. Limitations of the simulation program include the following: the

maximum number of tasks in each task system is 120 (in the current implementation), it is not an interactive program, its input format specification is somewhat rigid, and it assumes error-free input.

Future work in this area includes the following. Instead of considering abstract task systems, actual programs can be considered as the basis for the generation of the task systems. The performance of algorithm D when task weights are variable can also be studied. Task systems with multiple roots can be considered as well. In Hu's algorithm, when selecting a task from each task set as victim to be moved it to a higher numbered task set, the choice could be based on the number of predecessors of the task under consideration in the higher numbered task set. A graphical user interface can be developed to visualize the variations in the inter-processor communication.

REFERENCES

[Ananthaneni 97] Kamalakar Ananthaneni, "Generation of Maximally Parallel Task Systems", Master of Science Thesis, Computer Science Department, Oklahoma State University, Stillwater, Oklahoma, 1997.

[Battista et al. 96] G. Di Battista, A. Garg, G. Liotta, A. Parise, R. Tamassia, E. Tassinari, F. Vargiu, and L. Vismara. "Drawing Directed Graphs: An Experimental Study", in S. North, editor, Graph Drawing (*Proc. GD '96*), Lecture Notes in Computer Science, Springer-Verlag, 1997. (See also a preliminary version in Technical Report CS-96-24, Center for Geometric Computing, Department of Computer Science, Brown University, 1996.)

[El-Rewini 94] Hesham El-Rewini, *Task Scheduling in Parallel and Distributed Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[Hu 61] T. C. Hu."Parallel Sequencing and Assembly Line Problems", *Operations Research* Vol. 9, No. 6, pp. 841-848, April 1961.

[Mandyam and Samadzadeh 92a] S. Mandyam and Mansur H. Samadzadeh, "Implementation of Regular Expression Transformation Algorithms on the Hypercube," *Proceedings of the Twenty-Third Annual Pittsburgh Conference on Modeling and Simulation,* Volume 23, Part 3, pp. 1587-1594, Published and Distributed by: Instrument Society of America, Edited by: William G. Vogt and Marlin H. Mickle, Pittsburgh, Pennsylvania, April 30- May 1, 1992.

[Mandyam and Samadzadeh 92b] S. Mandyam and Mansur H. Samadzadeh, "Scheduling Algorithms for Precedence Graphs," *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing (SAC'92),* pp. 747-756, Edited by: Hal Berghel, Ed Deaton, George Hedrick, David Roach, and Roger Wainwright, Kansas City, Missouri, March 1992.

[Samadzadeh 92] Farideh A. Samadzadeh, "Scheduling Algorithms for Parallel Execution of Computer Programs," Ph.D. Dissertation, Computer Science Department, Oklahoma State University, Stillwater, OK, 1992.

[Samadzadeh, et al. 94] F. A. Samadzadeh, A. A. Thobani, and Mansur H. Samadzadeh, "Implementation and Performance Evaluation of a Self-Timed Multiprocessor Scheduler on a Shared-Memory Machine," *Proceedings of the Thirteenth Annual IEEE International Phoenix Conference on Computers and Communications* (IPCCC'94), pp. 54-60, Phoenix, Arizona, April 1994.

APPENDICES

APPENDIX A

GLOSSARY

Communication Overhead	Unproductive time due to processor communication.
DAG	Directed Acyclic Graph.
Deterministic Scheduling	When the information needed for scheduling tasks is known á priori.
Precedence Graph	A task system in which a precedence relation exits among the tasks. This precedence relation specifies the general order of task executions.
Rooted Tree	A directed graph similar to a precedence graph in which each node has at most one successor (and any number of predecessors, including zero predecessors).
Schedule	A description of the work to be carried out by each processor as a function of time.
Tasks	Sequential units of computation, identified by problem or program decomposition.
Task set	A set of tasks that in a task system do not have dependencies among them. Tasks that have the same distance from the root in a rooted tree. Tasks at the same level of a rooted tree.

APPENDIX B

PROGRAM LISTINGS

This appendix has two programs listed. The first program is the random task system generator, which generates the input to the second program. The second program listed is reducing processor communication overhead in multiprocessor scheduling which executes the input.

```
////////////////////////////////////////////////////////////////////////////
//                Program : Random Task System Generator
//                 Author : Suraj S Bhat
//             Instructor : Dr. Mansur H. Samadzadeh
//                   Date : January 2000
//   Programming Language : C
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// The program algorithm consists of 2 main functions getlevel() and
// gen_dependencies().The function getlevel() randomly generates the
// number of levels (2 to number of tasks in the system) in the task
// system and the number of tasks in each level. The number of tasks
// in each level range from 1 to the number of tasks remaining to be
// assigned minus the number of levels remaining. This function is used to
// generate dependencies among various tasks. The aim is to generate the
// number of successors, which is done randomly. The successor names are also
// chosen randomly, but the properties of the DAG are maintained. In each set
// of the initial set representation, it is checked to see whether any of the
// tasks have dependencies in the same task set. Thus, when the task system
// is being randomly generated, in each step of its process of generation, it
// is checked to see whether the DAG properties are being violated or not.
////////////////////////////////////////////////////////////////////////////


#include <stdio.h>
#include <stdlib.h>

//MAX is the number of nodes or tasks in the system.
#define MAX 50

// Structure holds the attributes of all tasks of the task system or DAG.
struct task {
   char task_name[80]; // name of the task
   int task_level;     // task level
   int task_weight;    // task weight
   int predecessors;   // number of predecessors
   char pred[200][20]; // predecessor names
   char suc[200][20];  // successor names
   int index;          // number of successors
} tasks[MAX + 1];

struct level {
```

```
    char task_name[MAX + 1][20]; // the task names in each level of the task
                                 // system.
    int index;                   // the number of tasks in each level of the
                                 // task system.
}levels[MAX + 1];

// This array is used to stack all the probable successors of each task in
// each level of the task system.
char stack[200][20];

// function prototypes for all the functions used.
void initialize(void);
void getname(void);
int getlevel(void);
void getweight(void);
void getpredecessors(void);
void get_pred_names(int);
int Strcmp(const char[], const char[]);
void getprint(int);
void gen_dependencies(int );
int stack_up(int);
int get_task(char[]);
int isrepeater(int, int[]);
void print_it(int);



main()
{
 int i;

  // Number of levels in the task system range from 2 to the number of tasks in
  // the system. Since tasks systems with only one root is considered and the
  // number of tasks in the task system are greater than 2, the above mentioned
  // range is considered.
 int levels;

 // Number of DAGS to be generated.
 int Number_of_DAGS = 20;

 // The srand() function uses the argument seed(time(0)) as a seed for a new
 // sequence of pseudo-random numbers to be returned by subsequent calls to
 // rand()
 srand(time(0));

 // This for loop calls all the functions and generates the DAGs.
 for (i = 1; i < Number_of_DAGS; i++)
  {
   initialize();
   getname();
   levels = getlevel();
   getweight();
   gen_dependencies(levels);
   print_it(i);
  }
}

////////////////////////////////////////////////////////////////////////////
// This function prints all the task names and the task variables to the
// outputfile.
////////////////////////////////////////////////////////////////////////////
void print_it(int i)
{
```

```
  register int t;
  FILE *fp;
  char file[80];

  // The randomly generated task system is stored in a file. The file name is
  // generated using sprintf. Thus each time the function is called a new file
  // name is generated with the prefix DAG and the value of i concatenated.
  sprintf (file, "%s%d", "DAG", i);

  // Opening file to write.
  if ((fp=fopen(file, "w"))==NULL) {
        printf("cannot open file. \n");
        exit(1);
  }

 // The number of nodes in the system is printed in the output file first.
 // The program implementing the proposed algorithm reads this first
 // and allocates memory accordingly.
 fprintf (fp,"%d\n", MAX);
 for (t = 1; t < MAX + 1; t++)
  {
   fprintf (fp,"%s ", tasks[t].task_name);
   fprintf (fp,"%d ", tasks[t].task_level);
   fprintf (fp,"%d ", tasks[t].task_weight);
   fprintf (fp,"%d", tasks[t].index);
   for (i = 1; i < tasks[t].index + 1; i++)
     fprintf(fp, " %s", tasks[t].suc[i]);
   fprintf(fp, "\n");
  }
 fclose(fp);
}

////////////////////////////////////////////////////////////////////////////
//Initialize the variables of the "tasks" and "levels" structure.
////////////////////////////////////////////////////////////////////////////
void initialize(void)
{
  register int t;

  for (t = 1; t < MAX + 1; t++)
    {
     tasks[t].index = 0;
     levels[t].index = 0;
    }
}

////////////////////////////////////////////////////////////////////////////
//This function generates task names from T1 to TMAX.
//It copies the task names to the task_name variable of the task structure.
////////////////////////////////////////////////////////////////////////////
void getname(void)
{
register int t;
char task[80];
  for (t = 1; t < MAX + 1 ; t++)
    {
     sprintf(task, "%s%d", "T", t);
     strcpy(tasks[t].task_name, task);
    }
}


////////////////////////////////////////////////////////////////////////////
```

```
// This function randomly generates the number of levels (2 to number of tasks
// in the system) in the task system and the number of tasks in each level.
// The number of tasks in each level range from 1 to the number of tasks
// remaining to be assigned minus the number of levels remaining.
////////////////////////////////////////////////////////////////////////////
int getlevel(void)
{
  // variables used in the for loop for generating the number of tasks in
  // each level and the task names.
  register int i,j,t;

 // the number of tasks in each level.
  int no_tasks;

 // variable used to as a counter put the successor names into the level
 // structure.
  int big = 1;

 // counter used to start the insertion of tasks in the level structure
 // from level 2
  int counter = 2;

 // number of levels in the DAG, range from 2 to the number of levels in the
 // DAG
  int no_level;

  int max;

 // number of tasks in each level range from 1 to the number of tasks
  // remaining to be assigned minus the number of levels remaining.
  int level_tasks = 0;

  int MAX_tasks = 0;

  no_level = 2 + rand() % (MAX - 1);

// The level one will have only one task, i.e., T1, in all DAGs. The level one
// will have no successors, only predecessors, because it is the root.

  tasks[1].task_level = 1;
  levels[1].index++;
  strcpy(levels[1].task_name[1], tasks[1].task_name);

  MAX_tasks = MAX - 1;

// The following for loop randomly generates the number of tasks in each
// level. The number of tasks in each level range from 1 to the number of
// tasks remaining to be assigned minus the number of levels remaining. Then
// the tasks are copied into the level structure, thus when the level
// structure is accessed, each index would contain  the number of tasks in
// that level and the task names.

  for (t = 2; t < no_level + 1; t++)
    {

      // the number of tasks in each level are randomly generated
      // using the following equation
      max = MAX_tasks % (no_level - t + 1);
      no_tasks = 1 + rand() % (max + 1);
      MAX_tasks = MAX_tasks - no_tasks;

      level_tasks = no_tasks + counter;

       if ((t == no_level && level_tasks <= MAX) || (level_tasks > MAX+1))
```

```
        {
         level_tasks = MAX + 1;
         no_tasks = MAX - counter + 1;
        }

       // this inner for loop is for copying the task names to each level.
       // the variable task_name is printed in the output file by the print_it()
       // function
        for (i = counter; i < level_tasks + 1; i++)
          {
            // the task level of each task is inserted in the structure task the
            // variable task_level is printed in the output file by the
            // print_it() function .

            tasks[i].task_level = t;

            strcpy(levels[t].task_name[big], tasks[i].task_name);
            big++;
          }
          big = 1;
          levels[t].index = no_tasks;
          counter = counter + no_tasks;
           if (level_tasks == MAX + 1)
             break;
    }

   return no_level;
}

// the weight of each task is randomly generated in the range 1 to 3.
void getweight(void)
{
  register int t;
  for (t = 1; t < MAX + 1; t++) {
     tasks[t].task_weight = 1 + rand() % 3;
   }
}

////////////////////////////////////////////////////////////////////////////
// This function is used to generate dependencies among various tasks.
// The aim is to generate the number of successors which is done
// randomly. The successor names are also chosen randomly, but the
// properties of the DAG are maintained. It is checked to see if tasks
// are dependent on themselves i.e., the node points to itself. In each
// set of the initial set representation, it is checked to see whether any
// of the tasks have dependencies in the same task set.
////////////////////////////////////////////////////////////////////////////
void gen_dependencies(int no_levels)
{
  // variables used in the for loop for generating successor predecessor
  // relationship between tasks in the task system.
  register int i, j, p, m, n;

 // The number of successors of each task (except the root) range
 // from 1 to the total number of tasks in the levels below the
 // level of the task considered. Each task has at least one successor
 // in the level just below the level of the task considered.
 int successors;

 // variable names used as index or flags
 int big, suc_name, task_number, check;

 // store the successors of the task (integers from 1 to MAX). This array
```

```
// is used to check whether the randomly generated successor has been
// repeated or not.
int array[MAX];

for (i = no_levels; i > 2; i--)
 {
  for (j = 1; j < (levels[i].index + 1); j++)
    {
     // tasks below level i are stacked or stored, and big is the number
     // of tasks in the stack. The stack is later used to get the successor
     // name (randomly generated)
     big = stack_up(i);

     // number of successors of a task could be from 1 to the number of
     // tasks in the stack.
     successors = 1 + rand() % big;

      if (successors >= big)
        successors = 1 + rand() % (big - 1);

     // get the task number. To get the index of the task structure into
     // which the number of successors and successor names are stored.
     task_number = get_task(levels[i].task_name[j]);

     for ( n = 1; n < MAX + 1; n++)
      array[n] = 0;

     for ( p = 1; p < successors + 1; p++)
       {
        // The successor name is chosen randomly. The successor could be
        // any task available in the array stack.
        suc_name = 1 + rand() % (big -1);

        // Atleast one successor is chosen from the level n-1, to maintain
        // the task level dependency between all tasks in the task system.
         if (p == 1)
          suc_name = (big - levels[i-1].index) + rand() % levels[i-1].index;

        // since the suc_name is generated randomly, it is checked whether
        // it has been repeated or not. if repeated the index is decrement
        // until a valid successor name is found.
         check = isrepeater(suc_name, array);

        // If the suc_name is not repeated then the task from the stack is
        // copied to the task structure.
        if (check == 1)
         {
          array[p] = suc_name;

          // for each task the successors are copied to the suc variable
          strcpy(tasks[task_number].suc[p],stack[suc_name]);

          // the number of successors are incremented for the task.
          // or tasks[task_number].index = successors;
          tasks[task_number].index++;
         }
        else
          --p;
       }
    }
}

// all the tasks in level 2 will always have T1 as its successor, because
```

```
    // the task system has only one root i.e., T1.
    if (i == 2)
    {
     for (p = 1; p < (levels[i].index + 1); p++)
      {
       task_number = get_task(levels[i].task_name[p]);
       strcpy(tasks[task_number].suc[1],levels[i-1].task_name[1]);
       tasks[task_number].index++;
      }
    }
}

////////////////////////////////////////////////////////////////////////////
// This function is used to check whether the randomly generated successor has
// been repeated or not.
////////////////////////////////////////////////////////////////////////////
int isrepeater(int suc_name,int array[])
{
 register int t = 0;
  for (t = 1; t < MAX + 1; t++)
   {
     if (suc_name == array[t])
     return 0;
   }
  return 1;
}

////////////////////////////////////////////////////////////////////////////
// returns the task number. i.e., for example if T5 is sent to the function
// it returns 5 to the function called.
////////////////////////////////////////////////////////////////////////////
int get_task(char temp[])
{
 register int m;
  for (m = 1;m < MAX + 1; m++)
   {
    if (Strcmp(tasks[m].task_name, temp) == 0)
        return m;
   }
}

////////////////////////////////////////////////////////////////////////////
// All the tasks in the levels below the level i that could be the successors
// of the task, are stacked up. The function returns the number of tasks
// in the stack to the called function.
////////////////////////////////////////////////////////////////////////////
int stack_up(int i)
{
 register int t,j;

 // The number of elements in the stack
 int big = 1;
 for (t = 1; t < MAX; t++)
   strcpy(stack[t],"");
 for(t = 1; t < i; t++)
  {
   for(j = 1; j < levels[t].index + 1; j++)
    {
     strcpy(stack[big],levels[t].task_name[j]);
     big++;
    }
  }
 return big;
```

```
}

////////////////////////////////////////////////////////////////////////////
// String compare function
////////////////////////////////////////////////////////////////////////////
int Strcmp(const char Lhs[], const char Rhs[])
{
  int i;

  for (i = 0; Lhs[i] == Rhs[i]; i++)
       if (Lhs[i] == '\0')
          return 0;

  return Lhs[i] - Rhs[i];
}
```

```
////////////////////////////////////////////////////////////////////////////
//                Program : Reducing Processor Communication Overhead in
//                          Multiprocessor Scheduling
//                  Author: Suraj S Bhat
//              Instructor: Dr. Mansur H. Samadzadeh
//                    Date: January 2000
//  Programming Language : C
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// The program algorithm consists of four main functions. They are gen_sets(),
// Hu_schedule(), random_schedule() and the proposed_schedule(). The gen_sets()
// implements the Hu's algorithm or Algorithm C. All functions schedule the
// tasks on p processors and compute communication overhead when each task is
// executed. Each function calls compute_overhead() to compute the communication
// overhead  because of  task allocation to each processor. In Hu's schedule,
// each task from each set is selected on a first come first serve basis and the
// task is assigned to the first available processor.
// The random_schedule() implements the random algorithm. In this function each
// task from each task set is chosen and is assigned to a randomly selected
// processor. For randomizing the schedule, the rand() function provided by the
// C compiler is used. The prudent_schedule() implements the proposed algorithm
// and also the Worst case algorithm. The function schedules the tasks based on
// its predecessors assigned to the p processors.
//    Each task from each set is selected on a first come first serve basis. For
// the best case, each task from each set is assigned to a processor, which has
// the highest number of the task's predecessors. For the worst case, each task
// from each set is assigned to a processor, which has the lowest number of its
// predecessors assigned or zero predecessors assigned. For the best case and
// the worst case, the function calls CheckForPred() to find the processor to
// which the highest number of predecessors of a task has been assigned and the
// lowest number of predecessors of a task or no predecessors of a task has been
// assigned, respectively.
//   The program uses four main structures. The structure Node stores attributes
// of each task in the DAG. These attributes include the task name, the task
// weight, the task level, the number of successors of the task, the number of
// predecessors of the task, the successor names and the predecessor names.
// Memory equivalent to the number of tasks in the structure times the size of
// the structure is allocated using malloc().
//    The attributes of each level in the DAG are stored by the Width structure.
// The processor structure is used to execute the tasks scheduled on each
// processor. Considering the worst case, the memory allocated to the Width and
// processor structure is same as the memory allocated to the Node structure,
// i.e., each task in the DAG has the maximum number of successors possible. The
// precedence structure is used to store the highest and the lowest number of
// predecessors of a task assigned to the given processors. This structure is
// used by the function prudent_schedule for the best case schedule and the
// worst case schedule.
////////////////////////////////////////////////////////////////////////////


#include <stdio.h>
#include <stdlib.h>

// Number of processors in the system
#define PROCESSORS 3

// Stores all the attributes of all tasks of the DAG.
struct Node {
   char task_name[80]; // task name is stored
   int task_weight;    // weight of the task is stored
   int task_level;     // level of the task in DAG
   int successors;     // successors of the task in the DAG
   int predecessors;   // predecessors of the task in the DAG
   int suc[25][6];     // names of the successors are stored
   int pred[25][6];    // names of the predecessors are stored
};
```

```
// Width structure stores attributes of each level in the DAG.
struct Width {
   int num_tasks;          // number of tasks in the level
   char task_names[40][6]; // names of the tasks in the level
};

// This structure is used to store the initial set representations of the DAG.
struct last {
  int Num_tasks;
  char Task_names[40][6];
} last[25];

// This structure is used to execute the tasks on the processors.
struct processor {
  char time_unit[25][25];
};

// This structure is used to check whether a time unit of a processor has
// been allocated to a task or not.
struct CHECK {
  char open_close[80];
} CHECK[100];

// This structure is used to find the largest and the smallest number of
// predecessors of a task allocated to any of the processors.
// The largest number is used to allocate the task to the processor when
// the proposed algorithm is implemented.
// The smallest number is used to allocate the task to the processor in the
// case of WORST_CASE_SCHEDULE of the tasks.
struct precedence {
  int big;
  int small;
} PRED[100];

// Number of sets or levels in the DAG before and after the implementation
// of the Hu's algorithm.
int SETS;

// function prototypes for the program.
void gen_sets(int MAX, int level, struct Node *node, struct Width *width);
int prudent_schedule(struct processor *P, struct Width *width, struct Node
*node, int MAX, int worst_case);
int Hu_schedule(struct processor *P, struct Width *width, int MAX,
struct Node *node);
int random_schedule(struct processor *P, struct Width *width, int MAX,
struct Node *node);
void print_schedule(struct processor *P);
void print_star();
int SMALL();
int BIG();
int CheckForPred(struct processor *P, char *temp, struct Node *node, int MAX,
int index, int row, struct Width *width, int worst_case);
int isrepeater(int suc_name, int array[], int MAX);
int compute_overhead(struct processor *P, char *temp, int index,
struct Node *node, int t, int MAX);
void scheduler(struct processor *P, struct Width *width, struct Node *node,
int MAX);
struct processor *get_memory(int );
void init_processor(struct processor *P);
struct Node *get_struct(int );
struct Width *get_width(int );
void label_table(int MAX, struct Node *node, struct Width *width);
void read_file(int MAX, FILE *fp, struct Node *node);
```

```
int get_level(int max, struct Node *node);
void init_pred( int max, struct Node *node);
void get_pred(int MAX, struct Node *node);
void init_width(int MAX, struct Width *width);
void delete(struct Width *width, int victim, int index);
int get_task_num(char *temp, int MAX, struct Node *spade);
int get_task(char *temp, struct Width *width, int index);
int check_pred(char *temp, int MAX, int index, struct Width *width,
struct Node *node);

main()
{
 // File pointer to open and read the input file.
 FILE *fp;

 // MAX variable is the number of nodes in the DAG.
 int MAX;

 struct Node *node;
 struct Width *width;
 struct processor *P;

 // reading the input file.
 if((fp = fopen("DAG16","r")) == NULL)
   {
     printf( "The input file is not found. Aborting ....\n");
     exit(0);
   }

 // Reading the number of nodes in the DAG.
 fscanf(fp, "%d", &MAX);

 // Memory is allocated to the structures Node, Width and processor according
 // to the value of MAX.
 node = get_struct(MAX);
 width = get_width(MAX);
 P = get_memory(MAX);

 //The variable task_names is initialized.
 init_width(MAX, width);

 //The remaining part of the input file is read.
 read_file(MAX, fp, node);

 // The input file is closed.
 fclose(fp);

 // The label_table is generated according to algorithm C or Hu's algorithm.
 label_table(MAX, node, width);

 // The scheduler produces all the four types of schedules.
 scheduler(P,width,node, MAX);
}

////////////////////////////////////////////////////////////////////////
// Allocating memory to the Width structure.
////////////////////////////////////////////////////////////////////////
struct Width *get_width(int MAX)
{
  struct Width *width;

  if ((width = (struct Width *)malloc( MAX * sizeof(struct Width))) == NULL) {
     printf("Allocation error - aborting.");
```

```
      exit(1);
    }
    return width;
}

////////////////////////////////////////////////////////////////////////////
// Initializing the values of num_tasks and task names variables of the
// Width structure.
////////////////////////////////////////////////////////////////////////////
void init_width(int MAX, struct Width *width)
 {
    register int i,j;
    for (i = 1;i < MAX + 1; i++)
     {
      width[i].num_tasks = 0;
      for (j = 1; j < MAX; j++)
        strcpy(width[i].task_names[j], "");
     }
 }

////////////////////////////////////////////////////////////////////////////
// The label table initializes the task systems by implementing the Hu's
// algorithm. It generates the final set representation to the given
// number of processors.
////////////////////////////////////////////////////////////////////////////
void label_table(int MAX, struct Node *node, struct Width *width)
{
  int max_width;
  int level;
  struct processor *P;

  // level - The number of levels in the system
  level = get_level(MAX, node);

  // This function copies all the data from the Node structure to Width
  // and other structures as required.
  initialize(MAX, node, width);

  // This function generates the final set representation of the DAG, i.e., it
  // implements Algorithm C or Hu's Algorithm for given number of processors
  gen_sets(MAX, level, node, width);
}

////////////////////////////////////////////////////////////////////////////
// All four types of schedules are obtained using this function.
// 1. Hu's schedule is obtained and is directly scheduled on the given
//    number of processors.
// 2. In Random schedule, from each set, the available tasks are randomly
//    chosen for task allocation and the processors to which the tasks
//    allocated are also randomly chosen.
// 3. Proposed Schedule: Tasks are scheduled according to predecessors of
//    each task.
// 4. Worst schedule: This schedule is exactly opposite to the Prudent
//    schedule.
////////////////////////////////////////////////////////////////////////////
void scheduler(struct processor *P, struct Width *width, struct Node *node,
int MAX)
{
  register int i,j;

  int execution_time = 0; // Communication Overhead

  // Used to store the width structure before each of the schedules are
```

```
 // implemented.
 char backup[100][25];

 // Variable used as a counter while storing task names in the array "backup"
 int counter = 1;

 // When prudent assignment of tasks is to be done the variable worst_case
 // is set to 0 and for worst case assignment of tasks the variable is
 // set to 1.
 int worst_case = 0;

 // Saving the width structure before scheduling using Hu's algorithm.
 for ( i = 1; i < SETS + 1; i++)
   for ( j = 1; j < width[i].num_tasks + 1; j++) {
      strcpy(backup[counter],width[i].task_names[j]);
      counter++;
 }

// Communication overhead is obtained after scheduling according to
// Hu's algorithm.
execution_time = Hu_schedule(P,width,MAX, node);

// The schedule and the communication overhead is printed after Hu's schedule
// is implemented on the given number of processors.
print_star();
printf("Tasks scheduled on the processors, before prudent assignment\n");
printf("                of tasks (Hu's algorithm)\n");
print_star();
print_schedule(P);
print_star();
printf("The processor communication overhead before prudent assignment\n");
printf ("                          of tasks(Hu's algorithm)\n");
print_star();
printf ("Processor Communication %d\n", execution_time);

// The saved status of the width structure before scheduling Hu's
// algorithm is copied back to the Width structure.
counter = 1;
for ( i = 1; i < SETS + 1; i++)
 for ( j = 1; j < width[i].num_tasks + 1; j++) {
    strcpy(width[i].task_names[j], backup[counter]);
    counter++;
  }

// Communication overhead is obtained after scheduling according to
// random algorithm.
execution_time = random_schedule(P, width,MAX,node);

// The saved status of the width structure is restored.
counter = 1;
for ( i = 1; i < SETS + 1; i++)
  for ( j = 1; j < width[i].num_tasks + 1; j++) {
     strcpy(width[i].task_names[j], backup[counter]);
     counter++;
   }

// The schedule and the communication overhead is printed after the
// implementation of the random schedule.
print_star();
printf("Tasks scheduled on the processors, before prudent assignment\n");
printf("                of tasks (randomly)\n");
print_star();
print_schedule(P);
```

```
print_star();
printf("The processor communication overhead before prudent assignment\n");
printf ("                          of tasks\n");
print_star();
printf ("Processor Communication %d\n", execution_time);

// worst_case is set to 0 for prudent assignment of tasks.
worst_case = 0;

// The communication overhead is obtained after the implementation of
// prudent or proposed schedule on the given number of processors.
execution_time = prudent_schedule(P, width, node, MAX, worst_case);

// The schedule and communication overhead is printed after the
// implementation of prudent schedule on the given number of processors.
print_star();
printf("Tasks scheduled on the processors, after prudent assignment\n");
printf("                          of tasks\n");
print_star();
print_schedule(P);

print_star();
printf("The processor communication overhead after prudent assignment\n");
printf ("                          of tasks\n");
print_star();
printf ("Processor Communication %d\n", execution_time);

// The saved status of the width structure before scheduling prudent
// algorithm is copied back to the Width structure.
counter = 1;
for ( i = 1; i < SETS + 1; i++)
  for ( j = 1; j < width[i].num_tasks + 1; j++) {
     strcpy(width[i].task_names[j], backup[counter]);
     counter++;
  }

// worst_case is set to 1 for worst case assignment of tasks.
worst_case = 1;

// The communication overhead is obtained after the implementation of
// worst case schedule on the given number of processors.
execution_time = prudent_schedule(P, width, node, MAX, worst_case);

// The schedule and communication overhead is printed after the
// implementation of worst case schedule on the given number of processors.
print_star();
printf("Tasks scheduled on the processors, after worst case assignment\n");
printf("                          of tasks\n");
print_star();
print_schedule(P);
print_star();
printf("The processor communication overhead worst case assignment\n");
printf ("                          of tasks\n");
print_star();
printf ("Processor Communication %d\n", execution_time);

}

////////////////////////////////////////////////////////////////////////////
// The schedule obtained from Hu's algorithm is directly scheduled on to the
// processors also the communication overhead when each task is allocated is
// taken into account. The function returns communication overhead for the
// schedule.
```

```
////////////////////////////////////////////////////////////////////////////
int Hu_schedule(struct processor *P, struct Width *width, int MAX,
struct Node *node)
{
 register int i, j;
 int count = 0;  // count is the communication overhead for each task

 // Counter is the total communication overhead at any point of time.
 int counter = 0;

  for(i = SETS; i > 0; i--) {
    for (j = 1; j < PROCESSORS + 1; j++)
      {
       // if the number of tasks in the set is less than the number of
       // processors, the remaining tasks are made NULL.
       if (j > width[i].num_tasks)
         strcpy(width[i].task_names[j], "NULL");

       // The tasks are scheduled on the processor.
       strcpy(P[i].time_unit[j], width[i].task_names[j]);

       // each task is sent to the function compute_overhead to get the
       // communication overhead because of the task allocation to the
       // processor.
       count = compute_overhead(P, P[i].time_unit[j], j, node, i, MAX);

       // The counter keeps track of the total communication overhead
       // at any given time.
       counter = counter + count;
      }
  }
return counter;
}

////////////////////////////////////////////////////////////////////////////
// The schedule is obtained by randomly selecting each task from each task set
// and the time unit slot of each processor is also selected randomly, thus to
// get a totally random schedule, (psuedorandom number generator provided by
// the C complier is used). The function returns the communication overhead for
// the schedule.
////////////////////////////////////////////////////////////////////////////
int random_schedule(struct processor *P, struct Width *width, int MAX,
struct Node *node)
{
register int i,j,k,m;

// When a task is selected for allocating a time slot on the processor the
// array is filled with the task number. for instance if T5 is selected,
// 5 is filled in the array. Thus when the same task is selected randomly,
// the array is checked, to make sure the same task is not allocated again.
int array[100];

// When a time slot has been allocated a task (randomly), the array process
// is filled with that slot number. Thus when the same slot is chosen again
// randomly, the array process is checked to see whether the slot has
// allocated or not.
int process[100];

// task_name is the index for the Width structure variable Task_names.
// check is used check the validity of the chosen task.
// index is used check the validity of the chosen time slot of the processor.
int task_name, check, index;
```

```
// Variable is used when the index of the Width structure do not match
// the index of the processor structure.
int send = 0;

// Variable is used to get the total communication overhead because of the
// schedule.
int counter = 0;

// Variable is used to get the communication overhead because of the task
// allocation to a processor.
int count = 0;

// The array and process variables are initialized to 0.
  for(i = SETS; i > 0; i--) {
   for(m = 1; m < MAX + 1; m++) {
     array[m] = 0;
     process[m] = 0;
   }

// The random scheduling begins here.
    for (j = 1; j < PROCESSORS + 1; j++)
      {
         // The task name is chosen randomly, the range would be from
         // 1 to the given number of processors.
         task_name =  1 + rand() % PROCESSORS;

         // If the task_name is greater than the number of tasks in the set
         // then the task_names is set to NULL.
         if (task_name > width[i].num_tasks)
            strcpy(width[i].task_names[task_name], "NULL");

         // It is checked whether the task_name has been already selected.
         check = isrepeater(task_name, array, MAX);

         // If not then proceed.
            if(check == 1)
              {

                // Find a slot on any processor, that is valid, loop until
                // the slot is found.
                for (k = 1; k < PROCESSORS + 1; k++)
                  {

                    // The time slot is randomly chosen.
                    index = 1 + rand() % PROCESSORS;

                    // Check whether the slot has already been taken.
                    check = isrepeater(index, process, MAX);

                    // If the slot has not been taken proceed.
                    if (check == 1) {

                      // The arrays process and array are filled with the
                      // time slot selected and the task selected respectively.
                      process[j] = index;
                      array[j] = task_name;

                      // The task is allocated the time unit on the processor.
                      strcpy(P[i].time_unit[index],
                            width[i].task_names[task_name]);
                      send = j;

                      if (j != index)
```

```
                        send = index;

                      // Communication overhead because of the task allocation.
                      count = compute_overhead(P, P[i].time_unit[index],
                                               send, node, i, MAX);

                      // The communication overhead at any given time.
                      counter = counter + count;
                      break;
                    }

                    // If the time slot selected is not valid then --k.
                    else
                     --k;
                  }
               }

             // If the task name has been already selected, then --j.
             else
               --j;
       }
}
return counter;
}

////////////////////////////////////////////////////////////////////////////
// This fuction is called during printing the schedule and the communication
// overhead for a schedule.
////////////////////////////////////////////////////////////////////////////
void print_star()
{
printf("*************************************************************\n");
}

////////////////////////////////////////////////////////////////////////////
// The schedules are printed. They are:
//  1. Hu's algorithm Schedule
//  2. Random schedule
//  3. Proposed algorithm schedule
//  4. worst case schedule
////////////////////////////////////////////////////////////////////////////
void print_schedule(struct processor *P)
{
register int i,j,k;
for ( i = 1; i < PROCESSORS + 1; i++) {
  printf ("Processor P%d: ", i);
   for (k = SETS; k > 0; k--) {
      for (j = i; j < i+1 ; j++)
    printf("%4s ", P[k].time_unit[j]);
     }
printf("\n");
 }
}

////////////////////////////////////////////////////////////////////////////
// The best case and the worst case schedule are obtained using this function
// For the best case or prudent algorithm, the task is allocated to the
// processor which has the highest number of the task's predecessors.
// For the worst case algorithm, the task is allocated to the processor
// which has the lowest number of the task's predecessors.
////////////////////////////////////////////////////////////////////////////
int prudent_schedule(struct processor *P, struct Width *width, struct Node
*node
```

```
,int MAX, in_t worst_case)
{
register int i,j,k,m,l;

// task_name is the index for the Width structure variable Task_names.
// index is used check the validity of the chosen task
int task_name, index;

// Variable is used to get the total communication overhead because of the
// schedule.
int counter = 0;

// Variable is used to get the communication overhead because of the task
// allocation to a processor.
int count = 0;

// Variable is used when the index of the Width structure does not match
// the index of the processor structure.
int send;

for( i = SETS; i > 0; i--) {
  for ( m = 1; m < PROCESSORS + 1; m++)

  // The time unit slots are open for each processor.
    strcpy(CHECK[m].open_close,"open");

   for ( j = 1; j < PROCESSORS + 1; j++)
     {
       task_name = j;

       // If the task name is greater than the number of tasks in the set
       // then the empty slots are set to NULL.
       if (task_name > width[i].num_tasks)
         strcpy(width[i].task_names[task_name], "NULL");

       // The first set of tasks are leaf nodes and hence are scheduled
       // on a first come first basis.
       if (i != SETS)
        {
         // The time unit slot of a processor is chosen which is based on
         // the value of the worst_case.
         index = CheckForPred(P, width[i].task_names[j], node, MAX,j,i,width,
                              worst_case);

         // If the task name is NULL then it is checked to find an open slot
         // on any other processor.
         if (Strcmp(width[i].task_names[j], "NULL") == 0)
          {
           for(l = 1; l < PROCESSORS + 1;l++)
             if (Strcmp(CHECK[l].open_close, "open") == 0)
              {
               index = l;
               break;
              }
          }

         // Allocate the task to the index retuned by the function
         // compute_overhead
         strcpy(P[i].time_unit[index], width[i].task_names[j]);

         // The slot flag is set to "closed"
         strcpy(CHECK[index].open_close, "closed");
        }
```

```
        // The first task set is allocated to the processors on first come
        // first serve basis. The processors are also chosen on first come
        // first serve basis.
        else
          strcpy(P[i].time_unit[j], width[i].task_names[j]);
        send = j;
        if (j != index)
        send = index;

        // Communication overhead because of the task allocation.
        count = compute_overhead(P, P[i].time_unit[index], send, node, i, MAX);

       // Total communication overhead at any point of time.
        counter = counter + count;
      }
   }
 return counter;
}

////////////////////////////////////////////////////////////////////////
// This function checks for predecessors of each task assigned on any of the
// processors.
// returns the processor to which the largest number of predecessors of a
// task assigned, when the worst case = 0;
// returns the processor to which the lowest number of predecessors of a
// task assigned, when the worst case = 1;
////////////////////////////////////////////////////////////////////////
int CheckForPred(struct processor *P, char *temp, struct Node *node, int MAX,
int index, int row, struct Width *width, int worst_case)
{
register int i, j, k, n, m, l;

// The tasks scheduled on a particular processor is stored in the array str.
char str[40][10];

// A counter used for storing the task names in the variable str.
int point;

// Variable would have the task number.
int task;

// Variable to store the initial index.
int initial;

// Variable to check whether a task's predecessor has matched any of the
// tasks scheduled previously on the processor.
int flag;

// The initial value of the index is saved.
initial = index;

// If the task is NULL then index is sent back to the called function.
if (Strcmp(temp, "NULL") == 0)
return index;

// The task number is found, for instance for task_name T5, the task number
// would be 5
task = get_task_num(temp, MAX, node);

for (i = 1; i < PROCESSORS + 1; i++)
 {
  // Initializing the variables big and small of the precedence structure.
```

```
  PRED[i].big = 0;
  PRED[i].small = 0;

  // If the time slot of the processor is open, then all the tasks
  // scheduled on the processor is copied to the array str.
  if (Strcmp(CHECK[i].open_close, "open") == 0)
   {
    point = 1;
    for (k = SETS; k > row; k--)
      for (j = i; j < i + 1; j++)
        {
          strcpy(str[point], P[k].time_unit[j]);
          point++;
        }

    // If any of the predecessors of the task match the tasks scheduled
    // on the processor, then big or small is incremented, depending on
    // the best case schedule or the worst case schedule respectively.
    for ( m = 1; m < node[task].predecessors + 1; m++)
      {
       for (n = 1;n < point; n++)
         {
           if ((Strcmp(str[n], node[task].pred[m]) == 0) && worst_case == 0)
             PRED[i].big++;
           else if ((Strcmp(str[n], node[task].pred[m]) == 0) &&
                     worst_case == 1)
             PRED[i].small++;
         }
      }
   }
 }

  flag = 0;

// For prudent assignment of tasks.
  if (worst_case == 0)
    {
      // Check whether the task has predecessors on any processors.
      for(m = 1; m < PROCESSORS + 1; m++)
        {
         if (PRED[m].big == 0)
            flag++;
        }

       // If the task don't have any predecessors assigned then return
       // the initial value of index.
       if(flag == PROCESSORS)
        {
           for(l = 1; l < PROCESSORS + 1;l++)
             if (Strcmp(CHECK[l].open_close, "open") == 0)
              {
               initial = l;
               break;
              }
         return initial;
        }
       // Return the index of the processors to which the largest number
       // of the task's predecessors has been assigned.
         return BIG();
    }
// Worst case assignment of tasks.
  else if (worst_case == 1)
    {
```

```
      // Check whether the task has predecessors on any processors.
      for(m = 1; m < PROCESSORS + 1; m++)
        {
         if (PRED[m].small == 0)
            flag++;
        }

       // If the task donot have any predeceesors assigned then return
       // the initial value of index.
       if(flag == PROCESSORS)
        {
           for(l = 1; l < PROCESSORS + 1;l++)
             if (Strcmp(CHECK[l].open_close, "open") == 0)
              {
               initial = l;
               break;
              }
         return initial;
        }
       // Return the index of the processors to which the lowest number
       // of the task's predecessors has been assigned.
         return SMALL();
    }
}

////////////////////////////////////////////////////////////////////////////
// Return the index of the processors to which the highest number
// of the task's predecessors has been assigned.
////////////////////////////////////////////////////////////////////////////
int BIG()
{
  register int i;
  int big = 0;
  int ret;
  for ( i = 1; i < PROCESSORS + 1; i++)
    {
      if (Strcmp(CHECK[i].open_close, "open") == 0)
      if (PRED[i].big > big)
       {
         big = PRED[i].big;
         ret = i;
       }
    }
  return ret;
}

////////////////////////////////////////////////////////////////////////////
// Return the index of the processors to which the lowest number
// of the task's predecessors has been assigned.
////////////////////////////////////////////////////////////////////////////
int SMALL()
{
  register int i;
  int big = 999;
  int ret;
  for ( i = 1; i < PROCESSORS + 1; i++)
    {
      if (Strcmp(CHECK[i].open_close, "open") == 0)
      if (PRED[i].small < big)
       {
         big = PRED[i].small;
         ret = i;
       }
```

```
    }
  return ret;
}

////////////////////////////////////////////////////////////////////////
// Checks whether the task's predecessors has been assigned to a particular
// processor. This function is called to compute communication overhead
// whenever a task is scheduled to a processor.
////////////////////////////////////////////////////////////////////////
int compute_overhead(struct processor *P, char *temp, int index,
struct Node *node, int t, int MAX)
{
 register int i,j,k = 1;

 // Used to store the tasks scheduled on a particular processor.
 char str[50][16];

 // A counter used for storing the task names in the variable str.
 int point;

 // Variable would have the task number.
 int task;

 // The total communication overhead of all the tasks scheduled is stored
 // in this variable.
 int counter = 0;

 // If the task_name is NULL or if the first set of tasks are being scheduled
 // return counter.
 if ( t == SETS || (Strcmp(temp, "NULL") == 0))
   return counter;

 // Copy all the tasks scheduled on the processor to the variable str.
 for(i = SETS; i > 0; i--) {
   for(j = 1; j < PROCESSORS + 1; j++) {
       if(j != index) {
        strcpy(str[k], P[i].time_unit[j]);
         k++;
         point++;
    }
 }
}

 // The task number of the task_name is found.
 task = get_task_num(temp, MAX, node);

 // The communication overhead is calculated for the particular processor
 // i.e., if the task's predecessor matches any of the tasks scheduled
 // the variable counter is incremented.
   for(j = 1; j < node[task].predecessors + 1; j++)
     {
      for(k = 1; k < point + 1; k++)
         if (Strcmp(str[k], node[task].pred[j]) == 0)
           counter++;
     }
  return counter;
}

////////////////////////////////////////////////////////////////////////
// This function checks whether a task has been assigned to a processor.
// If the task has been assigned to a processor the function returns 0 else
// returns 1.
////////////////////////////////////////////////////////////////////////
```

```
int isrepeater(int suc_name, int array[], int MAX)
 {
   register int t;
     for (t = 1; t < MAX; t++)
       {
         if(suc_name == array[t])
           return 0;
       }
    return 1;
 }

////////////////////////////////////////////////////////////////////////
//Memory is allocated to the structure processor.
////////////////////////////////////////////////////////////////////////
struct processor *get_memory(int MAX)
{
  struct processor *node;

  if ((node = (struct processor *)malloc( MAX * sizeof(struct processor)))
      == NULL) {
     printf("Allocation error - aborting.");
     exit(1);
  }
  return node;
}

////////////////////////////////////////////////////////////////////////
// This function implements Hu's algorithm, where the number of tasks in each
// task set would be less than or equal to the number of given processors.
////////////////////////////////////////////////////////////////////////
void gen_sets(int MAX, int level, struct Node *node, struct Width *width)
{
  register int i,j,k,n,m;
  int max_width; // maximum width or the largest task set of the DAG
  int level;     // number of levels in the DAG

  // When a set would contain all leaf nodes, a node is randomly chosen as
  // a victim to move to the higher level.
  int victim;

  struct Temp *temp;

  // Variable to check a node has been found or not.
  int node_found = 0;

  // Predecessors of each task is found.
  get_pred(MAX, node);

  // Number of levels in the DAG
  level = get_level(MAX, node);
  i = level;

  // Implementation of Hu's algorithm begins here.
  while(i > 0)
   {
    // If |Wi| <= P for i = L,.....,1 then schedule it
    // else If for some i, i.e., if the number of tasks in the set is greater
    // than the number fo processors then go further
     if (width[i].num_tasks > PROCESSORS)
       {

      // if n is not equal to L, where L is the number of levels in the DAG
        if (i != level)
```

```
    {
     n = i;
     node_found = 0;
     while(node_found == 0 && n <= level)
      {
    // find a node from Wn that does not have predecessors in Wn+1
    // if no such node is available in Wn then n = n + 1
    // if a node is found change the node's label from n to n+1
       for (k = 1; k < width[n].num_tasks + 1; k++)
        {
         node_found = check_pred(width[n].task_names[k], MAX, n, width,
                                 node);
         if (node_found == 1) {
          n++;
          width[n].num_tasks++;
          j = width[n].num_tasks;
          strcpy(width[n].task_names[j], width[n-1].task_names[k]);

          // The victim is moved from one level to a higher level.
          victim = get_task(width[n].task_names[j], width,n-1);

          // And it is deleted from the present level.
          delete(width, victim, n-1);
          break;
          }
        }
       if(node_found == 1)
        break;
       else
         n++;
       if(n == level)
        {
         i = level;
         break;
        }
     }
    }
 // if n = L then
 //   select any node from the set WL as the victim, since all are leaf
 //   nodes in this set
 //   change the node's label from L to L+1
 //   increment L by 1
   if ( i == level)
    {
     // A victim is chosen randomly.
     victim =  1 + rand() % width[i].num_tasks;

     // Number of levels in the DAG is incremented.
     level++;

     // Number of tasks in the new level is incremented
     width[level].num_tasks++;
     j = width[level].num_tasks;

     // Victim is copied to the new level and deleted from the
     // previous level.
     strcpy(width[level].task_names[j], width[i].task_names[victim]);
     delete(width, victim, i);
    }
    i = level;
  }

// If the number of tasks in the set is less than the number of processors
```

```
     // then the set is untouched.
     else if (width[i].num_tasks <= PROCESSORS)
        i--;
   }

// Each task and its predecessor names are printed.
print_star();
printf(" Node name and its predecessor\n");
print_star();

for(i = 1; i < MAX + 1; i++) {
   printf ("%s --> ",node[i].task_name);
   for (k = 1; k < node[i].predecessors + 1;k++)
     printf( "%s, ",node[i].pred[k]);
printf("\n");
}

// The final set representation is printed.
SETS = level;
print_star();
printf(" final set representation\n");
print_star();
 for (m = 1; m < level+1; m++) {
  printf ("W%d --> {", m);
    for (k = 1; k < width[m].num_tasks + 1; k++)
        printf("%s, ", width[m].task_names[k]);
    printf("}\n");
  }

// The initial set representation is printed.
level = get_level(MAX,node);
print_star();
printf(" Initial set representation\n");
print_star();
  for(i = 1; i < level + 1; i++){
   printf ("W%d --> {", i);
      for (j = 1; j < last[i].Num_tasks + 1; j++)
         printf("%s, ", last[i].Task_names[j]);
  printf("}\n");
  }
}

////////////////////////////////////////////////////////////////////////////
// Returns the task number of the task name.
////////////////////////////////////////////////////////////////////////////
int get_task(char *temp, struct Width *width, int index)
 {
   register int i;
   for (i = 1; i < width[index].num_tasks + 1; i++)
      if(Strcmp(temp, width[index].task_names[i]) == 0)
        return i;
 }

////////////////////////////////////////////////////////////////////////////
// This function checks whether the node's (the node is in the set Wn)
// predecessors exist in the set Wn+1.
////////////////////////////////////////////////////////////////////////////
int check_pred(char *temp, int MAX, int index, struct Width *width,
               struct Node *node)
  {
    register int i,j, k;

    // Task set is stored in the array str.
```

```
    char str[25][6];

    // Task number of the task name is stored.
    int task;

    for (i = 1; i < width[index + 1].num_tasks + 1;i++) {
       strcpy(str[i], width[index + 1].task_names[i]);
    }

    // If any of the tasks in the set match the predecessors of the
    // task then return 0 else return 1.
       task = get_task_num(temp, MAX, node);
       for (j = 1; j < node[task].predecessors + 1; j++)
        {
           for (k = 1;k < width[index+1].num_tasks+1; k++)
            {
             if(Strcmp(str[k],node[task].pred[j]) == 0)
              return 0;
            }
        }
         return 1;
  }

////////////////////////////////////////////////////////////////////////
// The task name in the form of string is sent to this function and the number
// corresponding to that task is sent back to the called function.
////////////////////////////////////////////////////////////////////////
int get_task_num(char *temp, int MAX, struct Node *spade)
 {
  register int i;
   for(i =1; i < MAX + 1;i++)
    {
     if (strcmp(temp,spade[i].task_name) == 0)
        return i;
    }
 }

////////////////////////////////////////////////////////////////////////
// When a task is moved from a lower level to a higher level, the task has to
// be deleted from the lower level. This function deletes the victim from the
// lower level. The victim is selected by random or on first come first
// serve basis.
////////////////////////////////////////////////////////////////////////
void delete(struct Width *width, int victim, int index)
 {
   register int i,j = 1;
   char temp[25][6];

   for(i = 1; i < width[index].num_tasks + 1;i++)
     {
       if ( i != victim)
        {
         strcpy(temp[j], width[index].task_names[i]);
         j++;
        }
     }
   for(i = 1; i < width[index].num_tasks + 1; i++)
      strcpy(width[index].task_names[i], "");

   width[index].num_tasks--;
   for(i = 1; i < width[index].num_tasks + 1; i++)
      strcpy(width[index].task_names[i], temp[i]);
 }
```

```
////////////////////////////////////////////////////////////////////////////
// The predecessors of each task is found and stored in the array pred[i][j]
// of the Node structure.
////////////////////////////////////////////////////////////////////////////
void get_pred(int MAX, struct Node *node)
{
  register int i, index, j;
  int level, pred, temp;
  level = get_level(MAX, node);
  init_pred(MAX, node);
  for(i = 1; i < MAX + 1; i++)
   {
     for (j = 1; j < node[i].successors + 1; j++)
      {
        index = get_num_task(node[i].suc[j], MAX, node);
        node[index].predecessors++;
        temp = node[index].predecessors;
        strcpy(node[index].pred[temp], node[i].task_name);
      }
   }
}

////////////////////////////////////////////////////////////////////////////
// The variable predecessors of the structure Node is set to 0.
////////////////////////////////////////////////////////////////////////////
void init_pred( int max, struct Node *node)
{
 register int i;
 for (i = 1; i < max + 1; i++)
     node[i].predecessors = 0;
}

////////////////////////////////////////////////////////////////////////////
// The string compare function.
////////////////////////////////////////////////////////////////////////////
int Strcmp(const char Lhs[], const char Rhs[])
{
  int i;

  for (i = 0; Lhs[i] == Rhs[i]; i++)
       if (Lhs[i] == '\0')
          return 0;

  return Lhs[i] - Rhs[i];
}

////////////////////////////////////////////////////////////////////////////
// All the tasks in the system are copied into the structures Width and last
// for further operations on them, and last is used to store the initial
// representations of the task sytstem. i.e., before implementing Hu's
// algorithm.
////////////////////////////////////////////////////////////////////////////
initialize(int MAX, struct Node *node, struct Width *width)
{
 register int i, j, count = 1;
 int level;
 level = get_level(MAX, node);
 for (i = 1; i < level + 1; i++)
  {
   last[i].Num_tasks = width[i].num_tasks;
   for (j = 1; j < width[i].num_tasks + 1; j++)
     {
```

```
      strcpy(width[i].task_names[j], node[count].task_name);
      strcpy(last[i].Task_names[j], node[count].task_name);
      count++;
     }
  }
}

////////////////////////////////////////////////////////////////////////
// Returns the number of levels in the DAG.
////////////////////////////////////////////////////////////////////////
int get_level(int max, struct Node *node)
{
  return node[max].task_level;
}

////////////////////////////////////////////////////////////////////////
// Reads the input file.
////////////////////////////////////////////////////////////////////////
void read_file(int MAX, FILE *fp, struct Node *node)
{
 register int i, j;
 char *temp;
 strcpy(temp, " ");
printf("*****************************************************************\n");
printf("Task_name Task_level Task_weight Task_succesors Successors\n");
printf("*****************************************************************\n");
for (i = 1; i < MAX + 1; i++)
 {
 fscanf (fp, "%s",node[i].task_name);
 fscanf (fp, "%d", &node[i].task_level);
 fscanf (fp, "%d", &node[i].task_weight);
 fscanf (fp, "%d", &node[i].successors);
 printf ("%6s %8d %10d %14d %9s",node[i].task_name, node[i].task_level,
         node[i].task_weight, node[i].successors, temp);
   for (j = 1; j < node[i].successors + 1; j++) {
       fscanf (fp , "%s", node[i].suc[j]);
       printf (" %s", node[i].suc[j]);
    }
 printf("\n");
 }
}

////////////////////////////////////////////////////////////////////////
// Allocate memory to the structure Node.
////////////////////////////////////////////////////////////////////////
struct Node *get_struct(int MAX)
{
  struct Node *node;

  if ((node = (struct Node *)malloc( MAX * sizeof(struct Node))) == NULL) {
     printf("Allocation error - aborting.");
     exit(1);
  }
  return node;
}
```

APPENDIX C

OUTPUT LI STINGS

This appendix presents two output listings. The first output listing is a sample of the output of the second program in appendix B. The second output listing presents the output for a task system set (a set consisting of 20 task systems).

A sample task system is executed using Hu's algorithm, the random algorithm, the proposed algorithm and the worst case algorithm. The input format, the predecessors of each task in the task system, the initial set representation, the final set representation for schedule on processors from 2 to 5, and the schedule for each number of processors for all the algorithms is summarized.

The input format

task_name	task_level	task_weight	# task_succesors	successors
T1	1	2	0	
T2	2	1	1	T1
T3	2	1	1	T1
T4	2	3	1	T1
T5	3	3	1	T3
T6	4	1	2	T5 T2
T7	4	3	1	T5
T8	4	1	3	T5 T4 T3
T9	5	3	1	T7
T10	5	2	8	T8 T5 T2 T6 T3 T7 T4 T1
T11	6	3	8	T9 T3 T8 T1 T4 T5 T10 T7
T12	6	1	6	T9 T3 T7 T4 T2 T8
T13	7	1	2	T12 T3
T14	7	3	2	T12 T6
T15	7	3	3	T11 T3 T7
T16	8	3	6	T13 T10 T5 T9 T14 T12
T17	8	1	9	T14 T7 T3 T8 T12 T6 T13 T11 T1
T18	9	1	9	T16 T6 T2 T17 T1 T10 T9 T7 T3
T19	9	1	12	T17 T2 T6 T15 T1 T14 T3 T5 T4 T7 T8 T13
T20	10	2	7	T19 T5 T1 T9 T13 T3 T7
T21	11	3	1	T20
T22	11	1	15	T20 T12 T10 T4 T8 T6 T2 T5 T19 T11 T18 T9 T16 T3 T15
T23	11	2	3	T20 T13 T15
T24	11	3	13	T20 T7 T10 T4 T1 T3 T16 T1 T5 T14 T12 T2 T13 T18
T25	11	3	15	T20 T16 T19 T2 T18 T5 T1 T17 T12 T6 T8 T13 T11 T7 T4

Dependencies among tasks (successor-predecessor relationship)	
task name	predecessors of the ask
T1	T2 T3 T4 T10 T11 T17 T18 T19 T20 T24 T25
T2	T6 T10 T12 T18 T19 T22 T24 T25
T3	T5 T8 T10 T11 T12 T13 T15 T17 T18 T19 T20 T22 T24
T4	T8 T10 T11 T12 T19 T22 T24 T25
T5	T6 T7 T8 T10 T11 T16 T19 T20 T22 T25
T6	T10 T14 T17 T18 T19 T22 T25
T7	T9 T10 T11 T12 T15 T17 T18 T19 T20 T24 T25
T8	T10 T11 T12 T17 T19 T22 T25
T9	T11 T12 T16 T18 T20 T22
T10	T11 T16 T18 T22 T24
T11	T15 T17 T22 T25
T12	T13 T14 T16 T17 T22 T24 T25
T13	T16 T17 T19 T20 T23 T24 T25
T14	T16 T17 T19 T24
T15	T19 T22 T23 T24
T16	T18 T22 T24 T25
T17	T18 T19 T25
T18	T22 T24 T25
T19	T20 T22 T25
T20	T21 T22 T23 T24 T25
T21	
T22	
T23	
T24	
T25	

Tasks in each level of the task system are represented by task sets (W1, W2, ..., W11). These task sets represent the task system before implementation of the Hu's algorithm.

Initial Set Representation	
set name	tasks in each set
W1	T1
W2	T2 T3 T4
W3	T5
W4	T6 T7 T8
W5	T9 T10
W6	T11 T12
W7	T13 T14 T15
W8	T16 T17
W9	T18 T19
W10	T20
W11	T21 T22 T23 T24 T25

Tasks in each level of the task system are represented by task sets (W1, W2, ..., W14). These task sets represent the task system after implementation of the Hu's algorithm on a given number of processors in the range 2 to 12 or 2 to Wmax.

Final Set Representation for Two Processors	
set name	tasks in each set
W1	T1
W2	T3 T4
W3	T5 T2
W4	T7 T8
W5	T9 T6
W6	T12 T10
W7	T14 T11
W8	T15 T13
W9	T16 T17
W10	T18 T19
W11	T20
W12	T23 T25
W13	T24 T22
W14	T21

Tasks Scheduled on 2 Processors Using Hu's Algorithm

Processor P1	T21	T24	T23	T20	T18	T16	T15	T14	T12	T9	T7	T5	T3	T1
Processor P2	NULL	T22	T25	NULL	T19	T17	T13	T11	T10	T6	T8	T2	T4	NULL

Inter-Processor Communication Overhead 60

Tasks Scheduled on 2 Processors Using the Random Algorithm

Processor P1	T21	T22	T25	T20	T19	T16	T13	T14	T10	T9	T7	T2	T4	T1
Processor P2	NULL	T24	T23	NULL	T18	T17	T15	T11	T12	T6	T8	T5	T3	NULL

Inter-Processor Communication Overhead 68

Tasks Scheduled on 2 Processors Using the Proposed Algorithm

Processor P1	T21	T24	T23	T20	T19	T17	T15	T14	T10	T6	T7	T5	T3	T1
Processor P2	NULL	T22	T25	NULL	T18	T16	T13	T11	T12	T9	T8	T2	T4	NULL

Inter-Processor Communication Overhead 59

Tasks Scheduled on 2 Processors Using the Worst Case Algorithm

Processor P1	T21	T24	T23	NULL	T18	T16	T15	T14	T12	T9	T7	T5	T3	T1
Processor P2	NULL	T22	T25	T20	T19	T17	T13	T11	T10	T6	T8	T2	T4	NULL

Inter-Processor Communication Overhead 64

Final Set Representation for Three Processors	
set name	tasks in each set
W1	T1
W2	T2 T3 T4
W3	T5
W4	T6 T7 T8
W5	T9 T10
W6	T11 T12
W7	T13 T14 T15
W8	T16 T17
W9	T18 T19
W10	T20
W11	T22 T23 T25
W12	T24 T21

Tasks Scheduled on 3 Processors Using Hu's Algorithm

Processor P1	T24	T22	T20	T18	T16	T13	T11	T9	T6	T5	T2	T1
Processor P2	T21	T23	NULL	T19	T17	T14	T12	T10	T7	NULL	T3	NULL
Processor P3	NULL	T25	NULL	NULL	NULL	T15	NULL	NULL	T8	NULL	T4	NULL

Inter-Processor Communication Overhead 82

Tasks Scheduled on 3 Processors Using the Random Algorithm

Processor P1	T21	T25	T20	T18	NULL	T13	NULL	NULL	T6	NULL	T4	NULL
Processor P2	NULL	T23	NULL	T19	T17	T15	T12	T9	T7	NULL	T3	NULL
Processor P3	T24	T22	NULL	NULL	T16	T14	T11	T10	T8	T5	T2	T1

Inter-Processor Communication Overhead 84

Tasks Scheduled on 3 Processors Using the Proposed Algorithm

Processor P1	T24	T22	T20	T18	T16	T13	T12	T9	T7	NULL	T2	NULL
Processor P2	T21	T23	NULL	NULL	NULL	T15	NULL	NULL	T8	NULL	T4	NULL
Processor P3	NULL	T25	NULL	T19	T17	T14	T11	T10	T6	T5	T3	T1

Inter-Processor Communication Overhead 71

Tasks Scheduled on 3 Processors Using the Worst Case Algorithm

Processor P1	T24	T22	NULL	T19	T17	T15	NULL	T9	T8	T5	T4	NULL
Processor P2	T21	T23	NULL	T18	T16	T13	T11	NULL	T6	NULL	T2	T1
Processor P3	NULL	T25	T20	NULL	NULL	T14	T12	T10	T7	NULL	T3	NULL

Inter-Processor Communication Overhead 95

Final Set representation for Four Processors	
Set Name	Tasks in Each Set
W1	T1
W2	T2 T3 T4
W3	T5
W4	T6 T7 T8
W5	T9 T10
W6	T11 T12
W7	T13 T14 T15
W8	T16 T17
W9	T18 T19
W10	T20
W11	T21 T22 T23 T25
W12	T24

Tasks Scheduled on 4 Processors Using Hu's Algorithm

Processor P1	T24	T21	T20	T18	T16	T13	T11	T9	T6	T5	T2	T1
Processor P2	NULL	T22	NULL	T19	T17	T14	T12	T10	T7	NULL	T3	NULL
Processor P3	NULL	T23	NULL	NULL	NULL	T15	NULL	NULL	T8	NULL	T4	NULL
Processor P4	NULL	T25	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Inter-Processor Communication Overhead 86

Tasks Scheduled on 4 Processors Using the Random Algorithm

Processor P1	NULL	T23	NULL	T18	T16	T14	NULL	T10	T6	NULL	T2	T1
Processor P2	NULL	T21	T20	NULL	T17	T13	T11	NULL	NULL	T5	NULL	NULL
Processor P3	T24	T22	NULL	NULL	NULL	T15	NULL	NULL	T7	NULL	T3	NULL
Processor P4	NULL	T25	NULL	T19	NULL	NULL	T12	T9	T8	NULL	T4	NULL

Inter-Processor Communication Overhead 94

Tasks Scheduled on 4 Processors Using the Proposed Algorithm

Processor P1	T24	T21	T20	T18	T16	T13	T12	T9	T7	NULL	T3	NULL
Processor P2	NULL	T22	NULL	T19	T17	T14	T11	T10	T6	T5	T2	T1
Processor P3	NULL	T23	NULL	NULL	NULL	T15	NULL	NULL	NULL	NULL	NULL	NULL
Processor P4	NULL	T25	NULL	NULL	NULL	NULL	NULL	NULL	T8	NULL	T4	NULL

Inter-Processor Communication Overhead 74

Tasks Scheduled on 4 Processors Using the Worst Case Algorithm

Processor P1	T24	T21	NULL	T19	T16	NULL	T11	NULL	T6	NULL	T3	NULL
Processor P2	NULL	T22	T20	NULL	T17	T15	NULL	T10	NULL	NULL	NULL	NULL
Processor P3	NULL	T23	NULL	T18	NULL	T13	T12	NULL	T7	T5	T4	T1
Processor P4	NULL	T25	NULL	NULL	NULL	T14	NULL	T9	T8	NULL	T2	NULL

Inter-Processor Communication Overhead 112

Final Set Representation for Five Processors	
Set name	Tasks in Each Set
W1	T1
W2	T2 T3 T4
W3	T5
W4	T6 T7 T8
W5	T9 T10
W6	T11 T12
W7	T13 T14 T15
W8	T16 T17
W9	T18 T19
W10	T20
W11	T21 T22 T23 T24 T25

Tasks Scheduled on 5 Processors Using Hu's Algorithm

Processor P1	T21	T20	T18	T16	T13	T11	T9	T6	T5	T2	T1
Processor P2	T22	NULL	T19	T17	T14	T12	T10	T7	NULL	T3	NULL
Processor P3	T23	NULL	NULL	NULL	T15	NULL	NULL	T8	NULL	T4	NULL
Processor P4	T24	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
Processor P5	T25	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Inter-Processor Communication Overhead 92

Tasks Scheduled on 5 Processors Using the Random Algorithm

Processor P1	T23	T20	NULL	NULL	T14	NULL	NULL	NULL	NULL	NULL	NULL
Processor P2	T21	NULL	NULL	T17	NULL	T11	NULL	T8	NULL	NULL	T1
Processor P3	T25	NULL	T18	NULL	NULL	NULL	T9	NULL	NULL	T3	NULL
Processor P4	T24	NULL	NULL	T16	T15	NULL	NULL	T7	T5	T2	NULL
Processor P5	T22	NULL	T19	NULL	T13	T12	T10	T6	NULL	T4	NULL

Inter-Processor Communication Overhead 102

Tasks Scheduled on 5 Processors Using the Proposed Algorithm

Processor P1	T21	T20	T19	T17	T13	T11	T10	T6	T5	T2	T1
Processor P2	T22	NULL	T18	T16	T14	T12	T9	T7	NULL	T3	NULL
Processor P3	T23	NULL	NULL	NULL	T15	NULL	NULL	NULL	NULL	NULL	NULL
Processor P4	T24	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
Processor P5	T25	NULL	NULL	NULL	NULL	NULL	NULL	T8	NULL	T4	NULL

Inter-Processor Communication Overhead 84

Tasks Scheduled on 5 Processors Using the Worst Case Algorithm

Processor P1	T21	T20	T18	NULL	T13	T11	NULL	T8	NULL	T2	NULL
Processor P2	T22	NULL	NULL	T17	T15	NULL	NULL	T7	NULL	T4	NULL
Processor P3	T23	NULL	T19	T16	NULL	T12	NULL	NULL	NULL	NULL	T1
Processor P4	T24	NULL	NULL	NULL	NULL	NULL	T9	T6	T5	NULL	NULL
Processor P5	T25	NULL	NULL	NULL	T14	NULL	T10	NULL	NULL	T3	NULL

Inter-Processor Communication Overhead 118

A sample of 20 task systems (number of tasks in each task system is 25) scheduled using Hu's algorithm, the random algorithm, the proposed algorithm, and the worst case algorithm. The task systems are executed on 2 to Wmax (maximum width) processors. The inter-processor communication overhead for each task system when each algorithm schedules is presented.

Number of Processors	2	3	4	5	6	7	8	9	10	11	12
Inter-processor Communication Overhead for Hu's Algorithm	20.00	23.00	24.00	24.00	25.00	27.00	27.00				
Inter-processor Communication Overhead for the Random Algorithm	20.00	34.00	26.00	38.00	38.00	35.00	38.00				
Inter-processor Communication Overhead for the Proposed Algorithm	16.00	17.00	19.00	16.00	18.00	20.00	20.00				
Inter-processor Communication Overhead for the Worst Case Algorithm	30.00	37.00	42.00	42.00	42.00	42.00	43.00				
Inter-processor Communication Overhead for Hu's Algorithm	25.00	30.00	34.00	34.00							
Inter-processor Communication Overhead for the Random Algorithm	28.00	30.00	35.00	39.00							
Inter-processor Communication Overhead for the Proposed Algorithm	22.00	26.00	28.00	28.00							
Inter-processor Communication Overhead for the Worst Case Algorithm	29.00	36.00	44.00	48.00							
Inter-processor Communication Overhead for Hu's Algorithm	13.00	16.00	19.00	19.00	21.00	21.00	21.00	21.00	21.00	21.00	23.00
Inter-processor Communication Overhead for the Random Algorithm	13.00	16.00	19.00	19.00	20.00	21.00	21.00	21.00	21.00	22.00	23.00
Inter-processor Communication Overhead for the Proposed Algorithm	13.00	17.00	19.00	19.00	21.00	21.00	21.00	21.00	21.00	21.00	23.00
Inter-processor Communication Overhead for the Worst Case Algorithm	13.00	17.00	19.00	20.00	21.00	22.00	21.00	22.00	22.00	22.00	23.00

Inter-processor Communication Overhead for Hu's Algorithm	13.00	12.00	12.00	12.00	12.00						
Inter-processor Communication Overhead for the Random Algorithm	23.00	28.00	29.00	28.00	32.00						
Inter-processor Communication Overhead for the Proposed Algorithm	10.00	11.00	12.00	12.00	12.00						
Inter-processor Communication Overhead for the Worst Case Algorithm	26.00	36.00	38.00	39.00	40.00						

Inter-processor Communication Overhead for Hu's Algorithm	15.00	19.00	19.00	19.00							
Inter-processor Communication Overhead for the Random Algorithm	20.00	29.00	35.00	32.00							
Inter-processor Communication Overhead for the Proposed Algorithm	12.00	12.00	20.00	20.00							
Inter-processor Communication Overhead for the Worst Case Algorithm	33.00	39.00	40.00	43.00							

Inter-processor Communication Overhead for Hu's Algorithm	17.00	25.00	24.00	27.00	25.00	27.00	27.00				
Inter-processor Communication Overhead for the Random Algorithm	22.00	32.00	29.00	41.00	35.00	37.00	35.00				
Inter-processor Communication Overhead for the Proposed Algorithm	13.00	20.00	23.00	27.00	25.00	27.00	27.00				
Inter-processor Communication Overhead for the Worst Case Algorithm	29.00	39.00	40.00	42.00	43.00	43.00	44.00				

Inter-processor Communication Overhead for Hu's Algorithm	19.00	22.00	26.00	29.00	28.00	30.00	30.00	30.00	31.00	31.00	30.00
Inter-processor Communication Overhead for the Random Algorithm	21.00	26.00	28.00	28.00	31.00	33.00	34.00	35.00	35.00	34.00	33.00
Inter-processor Communication Overhead for the Proposed Algorithm	18.00	22.00	26.00	26.00	28.00	29.00	30.00	30.00	29.00	31.00	30.00
Inter-processor Communication Overhead for the Worst Case Algorithm	22.00	28.00	31.00	35.00	33.00	34.00	35.00	35.00	35.00	35.00	36.00

Inter-processor Communication Overhead for Hu's Algorithm	13.00	14.00	14.00								
Inter-processor Communication Overhead for the Random Algorithm	25.00	33.00	40.00								
Inter-processor Communication Overhead for the Proposed Algorithm	9.00	14.00	14.00								
Inter-processor Communication Overhead for the Worst Case Algorithm	35.00	43.00	46.00								

Inter-processor Communication Overhead for Hu's Algorithm	23.00	28.00	29.00	38.00	34.00	40.00	36.00	38.00	38.00	43.00	39.00
Inter-processor Communication Overhead for the Random Algorithm	22.00	35.00	38.00	39.00	41.00	40.00	41.00	45.00	47.00	44.00	47.00
Inter-processor Communication Overhead for the Proposed Algorithm	21.00	28.00	27.00	30.00	32.00	35.00	33.00	38.00	41.00	42.00	40.00
Inter-processor Communication Overhead for the Worst Case Algorithm	31.00	43.00	42.00	45.00	47.00	47.00	47.00	52.00	52.00	52.00	52.00

Inter-processor Communication Overhead for Hu's Algorithm	8.00	13.00	13.00	13.00	13.00	13.00	13.00	13.00	13.00	13.00	13.00
Inter-processor Communication Overhead for the Random Algorithm	25.00	33.00	33.00	33.00	38.00	40.00	37.00	37.00	40.00	37.00	39.00
Inter-processor Communication Overhead for the Proposed Algorithm	12.00	13.00	13.00	13.00	13.00	13.00	13.00	13.00	13.00	13.00	13.00
Inter-processor Communication Overhead for the Worst Case Algorithm	31.00	39.00	42.00	43.00	44.00	44.00	44.00	44.00	44.00	44.00	44.00

Inter-processor Communication Overhead for Hu's Algorithm	18.00	19.00	31.00	32.00	32.00	32.00	32.00	32.00	32.00	32.00	30.00
Inter-processor Communication Overhead for the Random Algorithm	19.00	29.00	36.00	33.00	39.00	40.00	38.00	36.00	38.00	40.00	39.00
Inter-processor Communication Overhead for the Proposed Algorithm	16.00	19.00	22.00	23.00	23.00	23.00	23.00	23.00	23.00	23.00	23.00
Inter-processor Communication Overhead for the Worst Case Algorithm	24.00	34.00	40.00	42.00	44.00	44.00	44.00	44.00	44.00	44.00	44.00

Inter-processor Communication Overhead for Hu's Algorithm	13.00	15.00	15.00	15.00	15.00	17.00	17.00
Inter-processor Communication Overhead for the Random Algorithm	19.00	21.00	28.00	29.00	35.00	34.00	33.00
Inter-processor Communication Overhead for the Proposed Algorithm	11.00	12.00	15.00	16.00	14.00	17.00	17.00
Inter-processor Communication Overhead for the Worst Case Algorithm	31.00	34.00	38.00	39.00	38.00	38.00	39.00

Inter-processor Communication Overhead for Hu's Algorithm	3.00	3.00	3.00
Inter-processor Communication Overhead for the Random Algorithm	24.00	25.00	26.00
Inter-processor Communication Overhead for the Proposed Algorithm	3.00	3.00	3.00
Inter-processor Communication Overhead for the Worst Case Algorithm	31.00	40.00	42.00

Inter-processor Communication Overhead for Hu's Algorithm	24.00	30.00	34.00	34.00	36.00	37.00	40.00	40.00	40.00	39.00	42.00
Inter-processor Communication Overhead for the Random Algorithm	26.00	34.00	38.00	42.00	42.00	46.00	44.00	46.00	47.00	42.00	46.00
Inter-processor Communication Overhead for the Proposed Algorithm	24.00	29.00	34.00	32.00	39.00	39.00	40.00	36.00	37.00	38.00	40.00
Inter-processor Communication Overhead for the Worst Case Algorithm	31.00	41.00	46.00	46.00	52.00	53.00	53.00	49.00	50.00	50.00	50.00

Inter-processor Communication Overhead for Hu's Algorithm	12.00	16.00	17.00	17.00	18.00	18.00
Inter-processor Communication Overhead for the Random Algorithm	19.00	25.00	27.00	34.00	28.00	30.00
Inter-processor Communication Overhead for the Proposed Algorithm	10.00	17.00	14.00	15.00	17.00	17.00
Inter-processor Communication Overhead for the Worst Case Algorithm	26.00	35.00	34.00	35.00	35.00	36.00

Inter-processor Communication Overhead for Hu's Algorithm	22.00	23.00	24.00	25.00	28.00	29.00	29.00
Inter-processor Communication Overhead for the Random Algorithm	24.00	26.00	39.00	39.00	42.00	37.00	37.00
Inter-processor Communication Overhead for the Proposed Algorithm	15.00	21.00	22.00	20.00	20.00	22.00	22.00
Inter-processor Communication Overhead for the Worst Case Algorithm	28.00	38.00	47.00	44.00	45.00	44.00	45.00

Inter-processor Communication Overhead for Hu's Algorithm	6.00	6.00	6.00
Inter-processor Communication Overhead for the Random Algorithm	23.00	28.00	30.00
Inter-processor Communication Overhead for the Proposed Algorithm	6.00	6.00	6.00
Inter-processor Communication Overhead for the Worst Case Algorithm	35.00	41.00	44.00

Inter-processor Communication Overhead for Hu's Algorithm	25.00	32.00	32.00	32.00
Inter-processor Communication Overhead for the Random Algorithm	24.00	28.00	40.00	37.00
Inter-processor Communication Overhead for the Proposed Algorithm	22.00	23.00	23.00	23.00
Inter-processor Communication Overhead for the Worst Case Algorithm	29.00	47.00	52.00	53.00

Inter-processor Communication Overhead for Hu's Algorithm	6.00	6.00	6.00	6.00	6.00
Inter-processor Communication Overhead for the Random Algorithm	25.00	33.00	34.00	36.00	39.00
Inter-processor Communication Overhead for the Proposed Algorithm	6.00	6.00	6.00	6.00	6.00
Inter-processor Communication Overhead for the Worst Case Algorithm	35.00	44.00	45.00	46.00	47.00

Inter-processor Communication Overhead for Hu's Algorithm	21.00	27.00	29.00	31.00	31.00	31.00	33.00	34.00	33.00	33.00	34.00
Inter-processor Communication Overhead for the Random Algorithm	22.00	29.00	32.00	30.00	37.00	39.00	38.00	35.00	39.00	39.00	38.00
Inter-processor Communication Overhead for the Proposed Algorithm	21.00	26.00	26.00	30.00	29.00	27.00	27.00	28.00	27.00	29.00	28.00
Inter-processor Communication Overhead for the Worst Case Algorithm	28.00	34.00	38.00	40.00	44.00	42.00	42.00	42.00	42.00	42.00	42.00

VITA

Suraj Srinivas Bhat

Candidate for the Degree of

Master of Science

Thesis: REDUCING PROCESSOR COMMUNICATION OVERHEAD IN MULTIPROCESSOR SCHEDULING

Major field: Computer Science

Biographical:

Personal Data: Born in Dandeli, India, August 30, 1972, son of Srinivas S. Bhat and Lalita S. Bhat.

Education: Received Bachelor of Engineering in Mechanical Engineering from Bangalore University, Karnataka, India in September 1994. Completed the requirements for the Master of Science degree in Computer Science at the Computer Science Department at Oklahoma State University in May 2000.