REDUCING INTER-PROCESSOR COMMUNICATION OVERHEAD BY TASK REPLICATION

By

JUN SU

Bachelor of Science

Shenyang Agricultural University

Shenyang, China

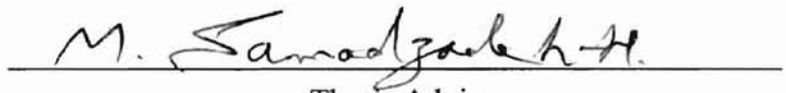
1992

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
December 2002

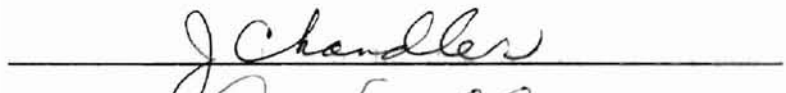
REDUCING INTER-PROCESSOR COMMUNICATION OVERHEAD

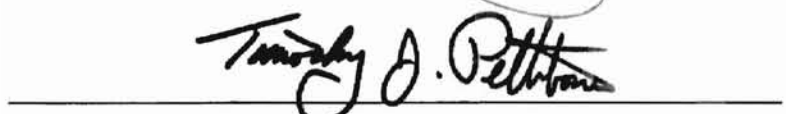
BY TASK REPLICATION

Thesis Approved:



Thesis Adviser





Dean of the Graduate College

PREFACE

In parallel processing, a program is first broken into a set of interdependent subprograms, tasks, processes, or threads. The program components together with the dependence relationships among them are generically called a task system. Each task in the resulting task system is then allocated to an available processing element on a multiprocessor system by a scheduling algorithm. The purpose of such a scheduling algorithm is to utilize the parallelism of the task system to minimize the length of time required for executing all of the tasks, without violating the precedence relations among the tasks. The computation time of the tasks and the overheads from various sources constitute the overall execution time of the task system. If any pair of tasks with a precedence relations between them are allocated to different processors, communication between the two processors is incurred. Such overhead, which is called inter-processor communication overhead, is added to the total execution time of the whole program.

The main objective of this thesis was to design and implement a multiprocessor scheduling algorithm which could reduce the inter-processor communication overhead. Based on Hu's Algorithm, the algorithm that was designed and implemented as part of this thesis adopts the approach of replicating selected tasks and allocating the replicated tasks to appropriate idle processors. Experimentations were conducted using a wide range of randomly generated task systems of different sizes. The results from these experimentations displayed a trend of increasing number of savings of inter-processor

communication realized by the algorithm with the growth of task system sizes. The trade-offs between the overhead savings and the number of task replications are also shown.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Mansur H. Samadzadeh for his continuous supervision, guidance, and dedication during my thesis work. Without his patience and encouragement, it would have been impossible to accomplish this research work. I also appreciate the time and effort of my other committee members Drs. John P. Chandler and Douglas Heisterkamp.

I am grateful to my parents for their moral and financial support, trust, and devoted love throughout my graduate studies. Their love can inspire me to meet any challenge. I also express thanks to my friends and colleagues who intellectually stimulated and helped me at all times.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

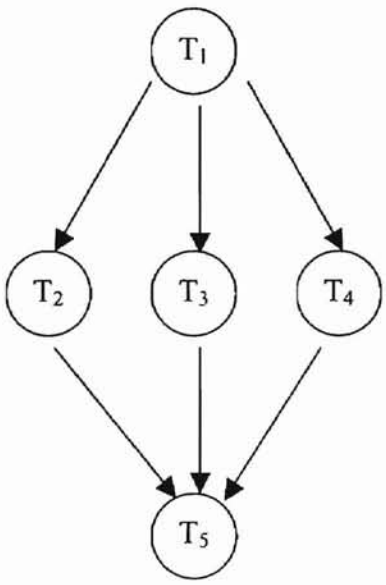
In computing complicated problems, parallel processing is an important approach that can be used to reduce the execution time of programs [Hockney and Jesshope 81] [Hwang and Briggs 84]. In parallel processing, the executions of the instructions of a program are divided among multiple computing units with the goal of minimizing the overall processing time of the program. Typically, a program is first partitioned into a number of subprograms, tasks, processes, or threads that can be processed concurrently on the different computing units of a multiprocessor system, while staying within the constraints of the precedence relations or interdependence among the tasks.

The objective of parallel processing is to utilize the inherent parallelism of a problem to maximum degree with the smallest possible overhead. The overhead comes from different sources, depending on the nature and characteristics of the task system. One type of such overhead is the communication overhead.

A task system consists of a number of tasks which have interdependence relations among them [Fox et al. 88]. The size of a task can be as small as a single statement or as large as an entire program, depending on the granularity of program decomposition. Therefore, the communication cost between all pairs of interdependent tasks could add up

to be considerable. In addition, the communication overhead between interdependent tasks on different processors could be even more significant.

After the program partitioning or decomposition stage, the resulting tasks are assigned to different processors for execution by a multiprocessor scheduling algorithm. When a task system (a set of tasks with precedence relations among them) is assigned to multiple processors, if any two tasks have an interdependence relation and they are allocated to different processors, then information needs to be conveyed between the two processors and thus inter-processor communication is required as overhead. Figure 1 illustrates an example of such a situation. By this schedule, task T_3 on processor P_2 and task T_4 on processor P_3 will need communication from their predecessor T_1 which is on processor P_1. Therefore, two instances of inter-processor communication, P_1 to P_2 for T_3 and P_1 to P_3 for T_4, are incurred. Similarly, T_5 also requires two instances of inter-processor communication from its predecessors T_3 and T_4 (P_2 to P_1 and P_3 to P_1). In this



Processor P_1	T_1	T_2	T_5
Processor P_2	∅	T_3	∅
Processor P_3	∅	T_4	∅

Figure 1. An Example of a Task System and Its Schedule on Three Processors

thesis, we focus on the first two instances of inter-processor communication which could be eliminated by task replication.

Excessive overhead, in the form of inter-processor communication, can add considerable extra time to the total execution time. Related research work has been done to design and implement scheduling algorithms to reduce the inter-processor communication overhead. One method is to allocate tasks to carefully selected processors [Bhat 2000]. The objective of this thesis was to utilize another approach to address this issue. The alternative adopted here is to duplicate appropriate tasks and assign them to idle processors, thus potentially curtailing the inter-processor communication overhead.

The rest of the thesis is organized as follows. Chapter II provides a brief literature review of the DAG scheduling problem and the inter-processor communication issue. Chapter III describes the approach of task replication and the proposed algorithm, the major contribution of the thesis. Chapter IV discusses the implementation issues of the algorithm. Chapter V contains the report of the experimentations carried out as part of this thesis. Chapter VI outlines the summary and future work for the thesis.

CHAPTER II

LITERATURE REVIEW

This chapter reviews DAG (Directed Acyclic Graph, which is the task system representation used in this thesis), general issues on DAG scheduling, the approach of Hu's Algorithm which is the basis of the proposed algorithm and the problem of inter-processor communication overhead.

2.1 Task Systems and DAGs

In parallel processing, a problem that has potential parallelism to be extracted and exploited, is first decomposed into a task system. Various formats are available to represent a task system [Aho et al. 86] [Gurd et al. 85]. Directed Acyclic Graphs (DAG), data flow graphs, and Petri nets are some examples of commonly used graphical representations of task systems. We used the DAG format to study scheduling algorithms in this thesis. The term DAG scheduling refers to scheduling a task system in DAG format.

DAGs are one form of precedence graphs widely used for the study of scheduling algorithms for parallel programs [Coffman 76]. A DAG can be represented as G = (V, E), where V is a set of nodes and E is a set of connecting edges (directed arcs). Each node represents a task which in turn is a set of instructions which must be executed on the

same processor sequentially without preemption. Each directed edge specifies a precedence between the incident tasks. For each edge, the ending node (child node) cannot start execution before it obtains data from the start node (parent node). For example, in the DAG shown in Figure 2, task T_2 cannot begin execution before both tasks T_4 and T_5 finish execution and send all the required data to task T_2. The node weights could be used to represent task processing times for the corresponding tasks. Alternatively, the edge weights could be used to represent communication costs between the pairs of connected tasks. A node with no parent is an entry node and a node with no child is an exit node. Without less of generality, we will study DAGs with single exit nodes.

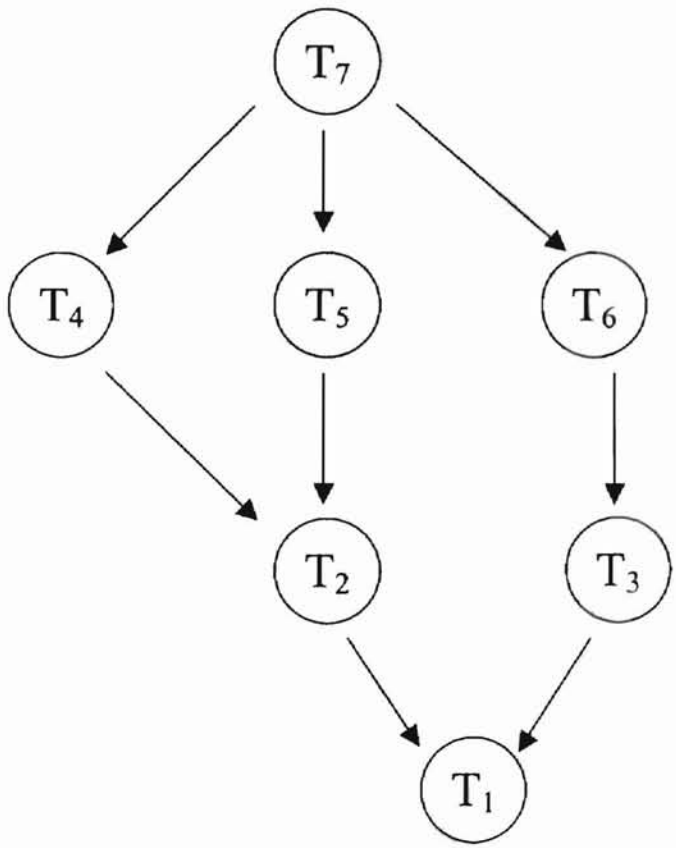


Figure 2. An Example of a DAG (Directed Acyclic Graph)

A rooted tree is a special type of DAG in which each node can have any number of predecessors but only one successor at most. This type of a simplified DAG is a major subject for the study of scheduling algorithms. Figure 3 shows a rooted tree.

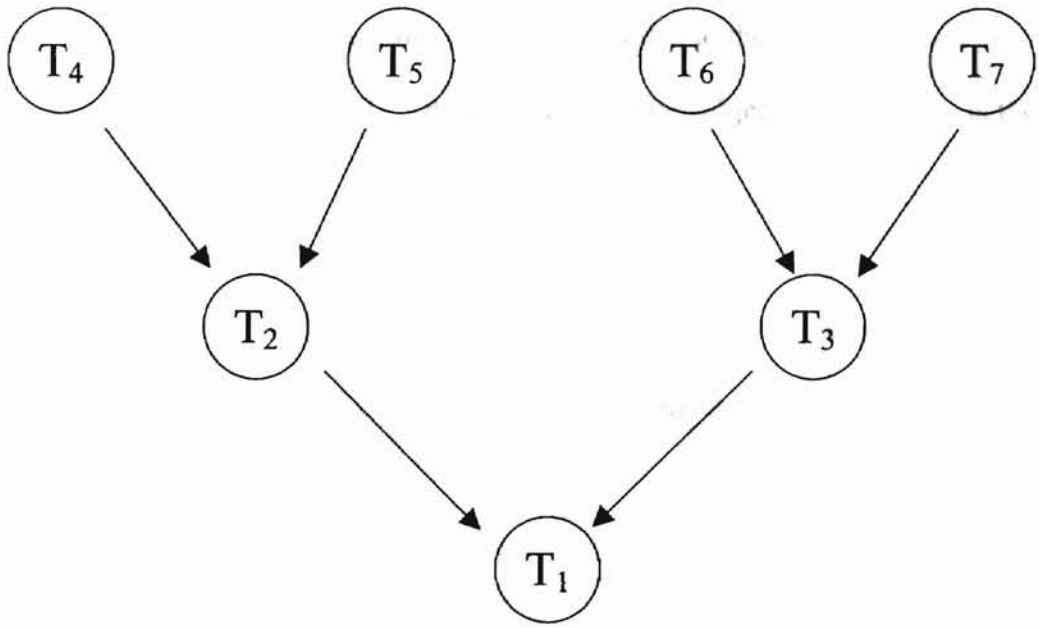


Figure 3. An Example of a Rooted Tree

2.2 DAG Scheduling

After a problem is decomposed and represented as a task system, each task is assigned to one of a set of processors on a multiprocessor system by a scheduling algorithm. The scheduling algorithm must not violate the precedence relation or the precedence graph. The main goal of DAG scheduling is to minimize the total execution time of a task system, which is known as schedule length.

The scheduling problems fall into two categories: static and dynamic. In static scheduling, the required information for scheduling a task system is known before

program execution and the scheduling is done at compile time [Chu et al. 84] [Gajski and Peir 85]. Such scheduling information include task processing times, communication costs, and the precedence relation. On the contrary, in dynamic scheduling such information is not known before program execution. Thus scheduling is done during program execution and the scheduling overhead is added to the total execution time of the program [Ahmad and Ghafoor 91] [Palis et al. 95]. The subject of this thesis is the static scheduling problem (hereafter referred to as scheduling).

Scheduling algorithms for DAGs are usually developed based on a number of assumptions about the structure and characteristics of task systems and the target machines. These assumptions include uniform task computation costs, zero inter-process communication costs, and an unlimited number of processors. It has been proven however that the general scheduling problem is NP-complete except for a number of special cases [Garey and Johnson 79]. Ullman showed that scheduling a DAG with uniform computation costs to an arbitrary number of processors is NP-complete [Ullman 75]. Scheduling a DAG with one or two unit computation costs to two processors has also been shown to be NP-complete [Coffman 76] [Ullman 75]. Garey et al. showed that scheduling an opposing forest (the disjoint union of an in-forest and an out-forest) with equal node weight to an arbitrary number of processors is also NP-complete [Garey et al. 83].

Given that scheduling a DAG is generally an NP-complete problem, optimal polynomial time algorithms do exist under three special circumstances [Kwok and Ahmad 99]: 1) scheduling equally weighted rooted trees on unlimited number of processors [Hu 61]. 2) scheduling arbitrary DAGs with uniform computation costs on

two processors [Coffman and Graham 72]. 3) scheduling interval-ordered, equally weighted DAGs on arbitrary number of processors [Papadimitriou and Yannakakis 79]. All of the above cases are based on the assumption that the inter-process communications are zero.

2.3 Hu's Algorithm

In this section, Hu's Algorithm for multiprocessor scheduling as well as some of its extended versions are discussed.

2.3.1 Algorithm A

Hu [Hu 61] designed a polynomial-time algorithm to provide optimal schedules for rooted-tree structured DAGs with equally weighted nodes. The critical part of the algorithm is the labeling process, in which each node is associated with a level number which is equal to the length of its path to exit node plus one. Thus the tasks in the whole DAG are partitioned into groups of a number of levels. After that, the tasks in the same level are assigned to the same time slot on different processors. The algorithm is outlined in Figure 4.

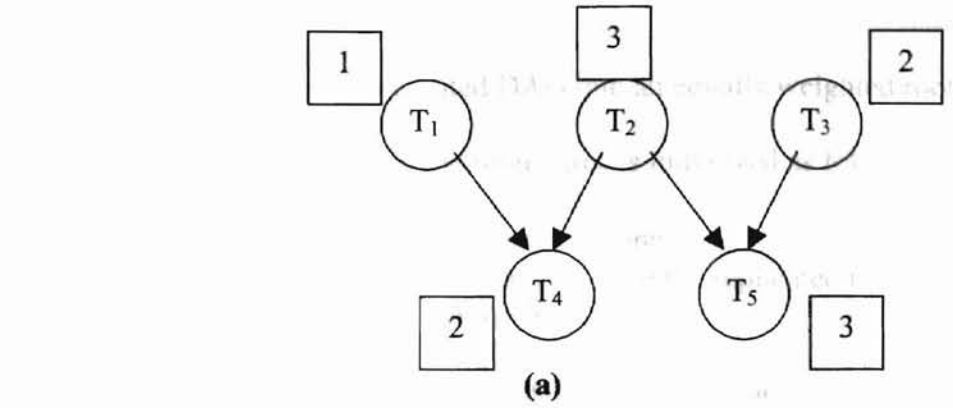
```
/* The label of each node in the rooted tree is initialized as follows. */
   Each node is assigned a label as follows:
        • The label of the root node is set to one.
        • The label of any other node is set to one plus the label of its unique
          successor node.

   L denotes the value of the maximum label among the initialized labels of the
   tasks.
   w_i denotes the set of tasks with label i.
   p denotes the number of processors in the machine.
/* Following the procedure below, the label of each task in the rooted tree is
   finalized. */
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 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
        if n = L then
           select any node from the set w_L as the victim;
           change the node's label from L to L+1;
           increase L by 1;
        end
        goto L1;

/* Assign the tasks in each group to time units on different processors. */
L3:
        /* Form the schedule as follows: */
        for i = L, L-1, ...,1 do
          execute a task in the set w_i in the (L-i+1)th unit of time on one
          of the p processors;
        end
```

Figure 4. Algorithm A: Scheduling a Rooted Tree on p Processors [Hu 61]

2.3.2 Algorithms B and C

Algorithm A, mentioned above in Subsection 2.3.1, has two important limitations: first, it can only schedule rooted trees. Second, the rooted tree must have equally weighted nodes [Mandyam and Samadzadeh 92]. To address these drawbacks, Algorithms B and C extend and modify Algorithm A to schedule DAGs based on the the same general approach as Hu's Algorithm.

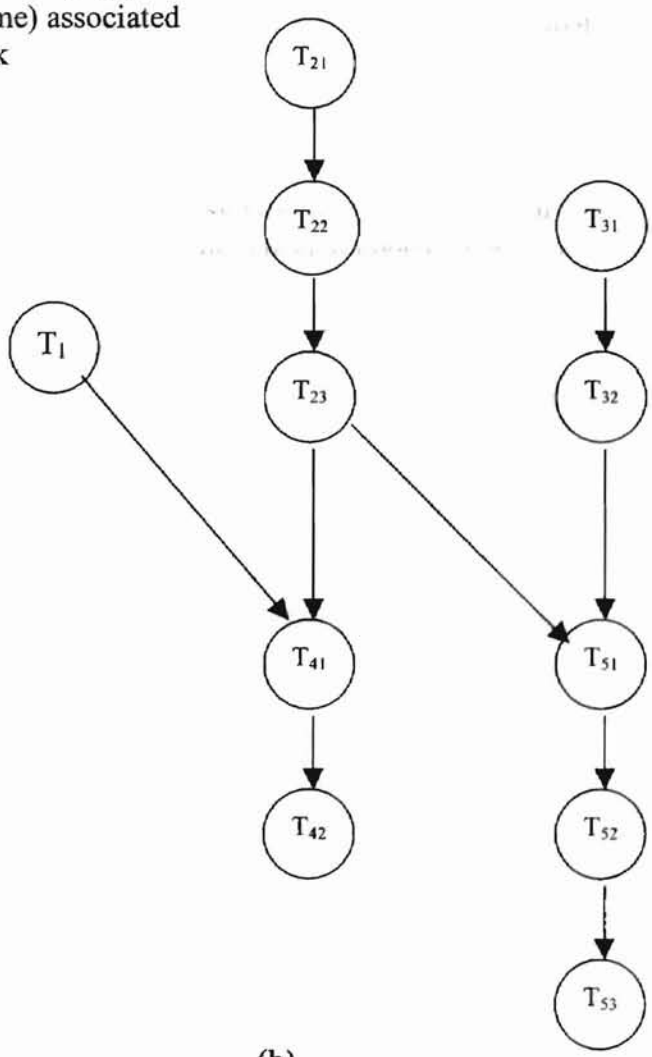
Algorithm B adds a preprocessing phase that converts an arbitrary, weighted DAG into an equally weighted rooted tree (see Figure 5 for an example). Then a procedure similar to Algorithm A is used to schedule the resulting task system (Figure 6). Algorithm C schedules an equally weighted DAG directly without the conversion stage as in Algorithm B. Algorithm C follows a modified labeling algorithm based on Algorithm A (Figure 7).



(a)

: a task

: the weight or cost (e.g., processing time) associated with each task



(b)

Figure 5. Converting a Weighted DAG to an Equally Weighted Rooted Tree [Gonzalez 77]

Preprocess the arbitrary weighted DAG into an equally weighted rooted tree.

/* The label of each node in the rooted tree is initialized as follows. */
Each node is assigned a label as follows:

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

L denotes the value of the maximum label among the initialized labels of the tasks.
w_i denotes the set of tasks with label i.
p denotes the number of processors in the machine.

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 6. Algorithm B: Scheduling a Rooted Tree with Repeated Nodes on p Processors [Hu 61]

```
/* The label of each node in the DAG is initialized as follows. */
   Each node is assigned a label as follows:
        • The label of the root node is set to one.
        • The label of any other node is set to one plus the label of its successor
          node. If a node has more than one successors, the maximum label is
          considered.

   L denotes the value of the maximum label among the initialized labels of the
   tasks.
   w_i denotes the set of tasks with label i.
   p denotes the number of processors in the machine.
/* Following the procedure below, the label of each task in the rooted tree is
   finalized. */
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 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
           change the node's label from n to n+1;
        end
        if n = L then
           select any node from the set w_L as the victim;
           change the node's label from L to L+1;
           increase L by 1;
        end
        goto L1;

/* Assign the tasks in each group to time units on different processors. */
L3:
        /* Form the schedule as follows: */
        for i = L, L-1, ..., 1 do
          execute a task in the set w_i in the (L-i+1)th unit of time on one
          of the p processors;
        end
```

Figure 7. Algorithm C: Scheduling a DAG on p Processors Directly [Hu 61]

2.4 Reducing Inter-Processor Communication Overhead

This section introduces the problem of inter-processor communication overhead and reviews a recent research work that addressed this issue.

2.4.1 Inter-Processor Communication

The algorithms mentioned in Section 2.3 provide optimal execution times for the scheduling of the respective task systems [Mandyam and Samadzadeh 92]. However, none of these algorithms takes into consideration the issue of inter-processor communication overhead.

Depending on the degree of interdependence among the tasks, a precedence graph could be sparsely or densely connected. Naturally, a highly-connected or dense task system tends to incur more inter-processor communication in multiprocessor scheduling. The overhead could have significant impact on the overall performance of a program.

2.4.2 Algorithm D

Recently, efforts have been made to study various methods to decrease the inter-processor communication overhead without impairing the optimality of algorithms A, B, and C mentioned in Section 2.3. A new algorithm, Algorithm D [Bhat 2000] , is put forward to achieve this goal.

Algorithm D follows the general approach of Hu's Algorithm, except that each task is assigned to the processor which has the highest number of its predecessors. A simulation program has shown that in most cases scheduling a DAG by Algorithm D incurs less inter-processor communication overhead than scheduling a DAG using Hu's

Algorithm and its variations. The cost of this overhead savings is the time complexity of computing the processor of each task and the space complexity of maintaining the information to do so. Figure 8 provides the details of Algorithm D.

```
/* The label of each node in the DAG is initialized as follows. */
   Each node is assigned a label as follows:
        • The label of the root node is set to one.
        • The label of any other node is set to one plus the label of its successor
          node. If a node has more than one successors, the maximum label is
          considered.

   L denotes the value of the maximum label among the initialized labels of the
   tasks.
   w_i denotes the set of tasks with label i.
   p denotes the number of processors in the machine.
/* Following the procedure below, the label of each task in the rooted tree is
   finalized. */
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 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
        if n = L then
           select any node from the set w_L as the victim;
           change the node's label from L to L+1;
           increase L by 1;
        end
        goto L1;

/* Assign the tasks in each group to time units on different processors. */
L3:
        /* Form the schedule as follows: */
        for i = L, L-1, ..., 1 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 on processor, select any one of the processors with predecessors
          assigned;
          execute a task in the set w_i in the (L-i+1)th unit of time on its processors;
        end
```

Figure 8. Algorithm D: Scheduling a DAG on p Processors While Reducing Inter-Processor Communication Overhead [Bhat 2000]

CHAPTER III

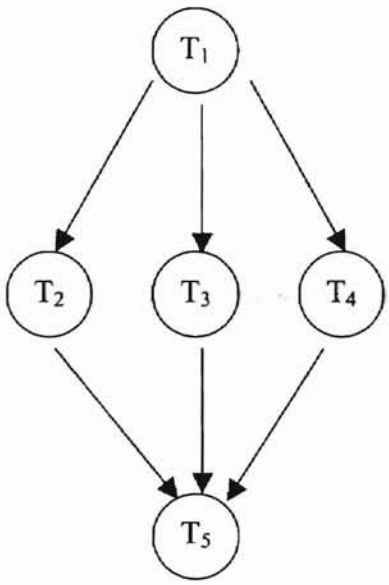
TASK REPLICATION APPROACH AND ALGORITHM

In this chapter, task replication as a method of reducing inter-processor communication overhead is illustrated. The proposed algorithm based on this approach, which is the main contribution of this thesis work, is also outlined.

3.1 Reducing Inter-Processor Communication by Task Replication

In some cases, inter-processor communication overhead can be reduced by task replication. Without changing the main method of Hu's Algorithm, the proposed algorithm realizes overhead savings by replicating some of the tasks in the task system. Under certain conditions, a task may be replicated and assigned to an idle processor, thereby making communication unnecessary if one of its successors is on the same processor.

For example, in Figure 9 task T_1 is replicated twice and the two replicated tasks are allocated to processors P_2 and P_3. As a result, task T_3 on P_2 and task T_4 on P_3 now could get the required data from their respective processors, and no inter-processor communication would be required. In this case, the inter-processor communication overhead is reduced by two instances for the cost of two replications of task T_1.



Processor P_1	T_1	T_2	T_5
Processor P_2	T_1	T_3	Ø
Processor P_3	T_1	T_4	Ø

Figure 9. Reducing Inter-Processor Communication by Task Replication

Task replication does not reduce inter-processor communication in all situations. Improper replication does not reduce and it may even increase the inter-processor communication. The purpose of the proposed algorithm is to discriminate among different situations and to decide in which cases replication should be done, which tasks and how many instances are to be replicated, and which processors they should be allocated to.

3.2 Algorithm

The goal of the proposed algorithm was to realize inter-processor communication savings by task replication, and to study the trade-off between the overhead savings thus achieved and the number of task replications. In this algorithm, after scheduling the tasks by Hu's Algorithm (see Section 2.3 for details), the tasks are replicated when appropriate and the replicated tasks are assigned to appropriate processors.

The proposed algorithm was designed under two constraints: first, the tasks in the DAG are assumed to be equally weighted, and second, we assume the availability of a sufficient number of processors on the target multiprocessor machine, i.e., the number of processors is equal to the width of the DAG.

Replicating a task and allocating it to an idle processor has two consequences depending on the in degree and out degree of the node representing the task in the corresponding precedence graph which represent the number of predecessor and successor tasks of the given task. The two consequences are an increase in the number of inter-processor communications on the one hand and a decrease in the number of inter-processor communications on the other hand. All we need to compute is whether a task replication produces a net reduction on the overhead. This is the basic idea of the proposed algorithm. If a replication could reduce the overhead ultimately, it is done and the corresponding overhead savings are calculated. The total overhead savings by the proposed algorithm are computed by the comparison with basic Hu's Algorithm. The proposed algorithm is outlined in Figure 10.

The running time for the algorithm can be analyzed as follows (see Figure 10 for details of the algorithm). A DAG is represented by G = (V, E) where V is the set of nodes and E is the set of edges. If T_i denotes the number of tasks in level i, for $1 \leq i \leq L$, and the average number of idle processors in each level is denoted by R, the number of runs of the innermost loop is obtained by: $T_1*R + T_2*R + \ldots + T_L*R = |V|*R$. Since in each innermost loop the successors and predecessor lists are traversed for each task, which add up to 2*|E|, the time complexity of the algorithm can be represented as O(|V|*|E|*R). In terms of space complexity, each task needs to maintain a list of

```
Step 1: Schedule and allocate the tasks to p processors following Hu's Algorithm.

   L denotes the value of the maximum time unit of the schedule.
Step 2:
/* Following the procedure below, the appropriate tasks are replicated and
   allocated to appropriate processors. */
 for i = 2, ..., L do
    for each idle processor P do
       for each task t in level i  do
            S denotes the number of t's successors on P.
            R denotes the number of t's predecessors.
            Rp denotes the number of t's predecessors on P.
            if S > R - Rp
            then
                  duplicate t to P at time unit i;
                  if t has any predecessor r at level i+1 which has no predecessor
                        and P is idle at time unit i+1
                  then
                     duplicate r to P at time unit i+1;
                     break;
                  end
             else if S > 0 and S = R - Rp
                  then
                     if t has any predecessor r at level i+1 which has no
                           predecessor and P is idle at time unit i+1
                     then
                           duplicate t to P at time unit i;
                           duplicate r to P at time unit i+1;
                           break;
                     end
                  end
       end
    end
 end

/* Execute all the tasks including the replicated ones in each time unit. */
for i = L, L-1, ..., 1 do
      execute the tasks including the replicated tasks in the (L-i+1)th unit of time;
end
```

Figure 10. Proposed Algorithm: Scheduling an Equally Weighted DAG on p Processors

successors in addition to the predecessor list. Therefore, the extra space requirement in addition to that of Hu's Algorithm is O(|V|+|E|).

CHAPTER IV

IMPLEMENTATION ISSUES

This chapter addresses the implementation issues of the proposed algorithm, including the input formats and the structure of the simulation program. The program was developed in C++ under Solaris 7 (SunOS v5.8).

4.1 Input Formats to the Program

The input to the simulation program is a set of randomly generated, equally weighted DAGs. The sources of the DAGs are two random task system generator programs: TGFF and TGG.

TGFF (Task Graphs For Free) [Dick et al. 98] is a pseudo-random task-graph generator program for use in scheduling and allocation research. Based on the parameters specified by the user, the program generates DAGs of different types meeting a user's requirements. The type of DAGs used in this implementation had multiple entry nodes and a single exit node. TGFF generates a graph file and an equivalent text file containing the generated DAG. The text file was used as the input to the simulation program. Figure 11 shows a standard text file containing a DAG generated by the TGFF program.

```
ARC a0_0   FROM t0_0   TO t0_1
ARC a0_1   FROM t0_0   TO t0_2
ARC a0_2   FROM t0_1   TO t0_2
ARC a0_3   FROM t0_1   TO t0_3
ARC a0_4   FROM t0_2   TO t0_3
ARC a0_5   FROM t0_0   TO t0_3
ARC a0_6   FROM t0_3   TO t0_4
ARC a0_7   FROM t0_3   TO t0_5
ARC a0_8   FROM t0_3   TO t0_6
ARC a0_9   FROM t0_3   TO t0_7
ARC a0_10  FROM t0_3   TO t0_8
ARC a0_11  FROM t0_8   TO t0_9
ARC a0_12  FROM t0_8   TO t0_10
ARC a0_13  FROM t0_8   TO t0_11
ARC a0_14  FROM t0_8   TO t0_12
ARC a0_15  FROM t0_8   TO t0_13
ARC a0_16  FROM t0_7   TO t0_14
ARC a0_17  FROM t0_6   TO t0_14
ARC a0_18  FROM t0_1   TO t0_14
ARC a0_19  FROM t0_5   TO t0_14
ARC a0_20  FROM t0_4   TO t0_14
ARC a0_21  FROM t0_14  TO t0_15
ARC a0_22  FROM t0_14  TO t0_16
ARC a0_23  FROM t0_14  TO t0_17
ARC a0_24  FROM t0_14  TO t0_18
ARC a0_25  FROM t0_14  TO t0_19
```

Figure 11. An Example of a DAG generated by TGFF [Dick et al. 98] (the graph is given as a list of directed edges which contains 20 nodes and 26 edges in this example.)

TGG, Task Graph Generation [Samadzadeh 91], generates single-entry, single-exit DAGs in adjacency matrix format. A seed for the random number generator used in TGG, the desired number of nodes of the generated DAG, and the number of graphs to be generated are specified as inputs to the task generation program. Each DAG generated is an upper triangular adjacency matrix. Figure 12 is a typical output of the generator.

```
0111111111111110
 001100110001101
  00110010011010
   0111011011001
    011101111001
     01010010100
      0110010111
       011011100
        01101100
         0101110
          011011
           01111
            0111
             010
              01
               0
```

Figure 12. An Example of a DAG generated by TGG [Samadzadeh 91] (the graph is given as an adjacency matrix which contains 16 rows and 16 columns in this example.)

4.2 Algorithm Implementation

First, a randomly generated task system is read from the output file of a random task system generator program. The task system is in the form of a single rooted, equally weighted DAG. Subsequently, the task system is scheduled to a multiprocessor system by following both Hu's Algorithm and the proposed algorithm, assuming that the target machine has the same number of processors as the width of the generated DAG. Based on the allocation by Hu's Algorithm, the proposed algorithm replicates the selected tasks in the task system and allocates the replicated tasks to the appropriate processors. Finally, the total number of inter-processor communication overhead instances are calculated for the schedule generated by each algorithm, showing the savings by the

proposed algorithm over Hu's Algorithm. The number of task replications is also calculated.

The simulation program, designed and implemented as part of this thesis work, is called Task_System. It was written in C++ and consists of 836 lines of code, 7 methods and 4 classes.

The major methods of Task_System are as listed below. Read_TGFF_Graph() and Read_TGG_Graph() are used to read input (the format of DAGs) from the two random task system generators. Find_Level_Width() labels the level of each task node following the labeling process of Hu's Algorithm [Hu 61]. The width of the DAG is also determined. Assign_by_Hu() follows Hu's Algorithm to schedule the task system to a multiprocessor system. Assign_by_Proposed() duplicates the appropriate tasks to appropriate processors following the proposed algorithm. Total_OH() computes the number of overhead instances for each of the two algorithms.

Four classes were designed to represent the task systems and implement the algorithm. The Task_Node is the object representing a task in the task system. Task_List class is a linked list of the Task_Node objects. Task class contains information about each task including task number, its level number, its assigned processor, and the lists of its predecessors and successors. Task_System class represents the entire task system, which consists of Task objects, the number of tasks, DAG width, the number of levels, and the allocated processors of the scheduled tasks.

CHAPTER V

EXPERIMENTATION

The experiment conducted using the simulation program called Task_System consists of two sets of runs, each using the two different task system generators described in Chapter IV as sources of input. In each set of runs, for each different task system size range, the average savings of inter-processor communication instances by the proposed algorithm over Hu's Algorithm, and the overhead savings per task replication were output.

Each pseudo-randomly generated task system was scheduled by Hu's Algorithm and the proposed algorithm. The inter-processor communication overhead was calculated based on the schedule generated by each algorithm, on the assumption that each processor-to-processor communication instance counts as one unit. The number of task replications done by the proposed algorithm was also counted. Then the average savings accomplished by the proposed algorithm over Hu's Algorithm, and the savings per task replication were obtained as the final results.

For the TGFF program, ten distinct task systems of each size range, starting from 10 – 19, were generated using ten different, widely distributed seeds for the pseudo-random number generator. Then the simulation program was run for each generated DAG. The results were averaged and the value for the DAGs of that size range was

obtained. Then the size range of the DAGs was incremented by 10 each time, and the corresponding results were obtained and averaged for each size range. The maximum size range of the task systems is 190 – 199. The maximum in-degree and out-degree of each DAG are equal to the number of the nodes in that DAG.

For the TGG program, the procedure was the same except that the exact size of the generated task systems can be specified as a parameter to the task system generator program. The sizes of DAGs generated by TGG range from 10 to 200, incrementing by 10 each time.

To automate the entire process, a few scripts and programs were also written to work with each DAG generator program and the Task_System simulation program. These programs generated the input files containing the appropriate parameters to the DAG generator programs, and wrote the final results to the output file. The results are plotted in Figures 13 through 16.

Figures 13 and 14 plot the average number of inter-processor communication instances saved by the proposed algorithm compared with Hu's Algorithm for different sizes of task systems. Both diagrams show that with the increase of task system size, the number of overhead savings accomplished by the proposed algorithm tends to increase. For TGFF, the graph exhibited a tendency for the amount of savings to fluctuate up and down over a mildly ascending line. For TGG, the number of savings appeared to climb steadily with the increase of the number of tasks in the DAGs.

Figures 15 and 16 exhibit the behavior of the average overhead savings per task replication by the proposed algorithm with the increase of task system sizes. For TGFF, the numbers were between 0.91 and 1.63. For TGG, the numbers were from 1.37 to

18.38 based on the increase of the number of tasks in the DAGs, displaying a clear ascending trend.

The substantial differences of the run results from TGFF and TGG can be attributed to the different characteristics of the DAGs generated by the two task system generators. DAGs generated by TGG have a single entry node, which makes the replication of the entry node very effective since the entry node usually has no predecessors but many successors. The TGFF-generated DAGs, which have multiple entry nodes, do not have this privilege and thus replications are not done in them as much as in the TGG-generated DAGs.

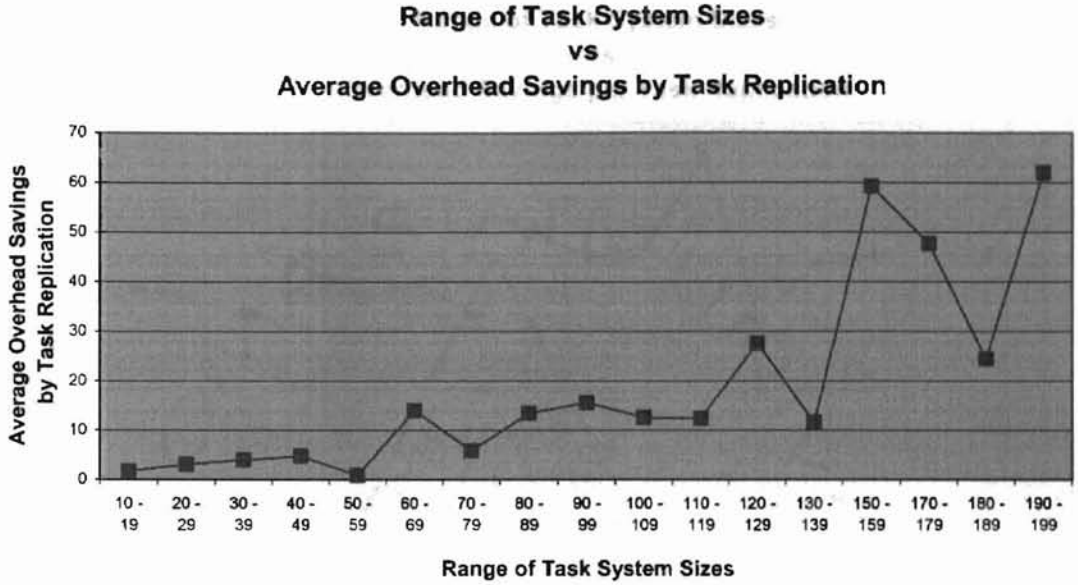


Figure 13. Average Inter-Processor Communication Overhead Savings by the Proposed Algorithm with Increasing Number of Tasks in Task Systems, with Task Systems Generated Using TGFF (number of task systems in each range is 10).

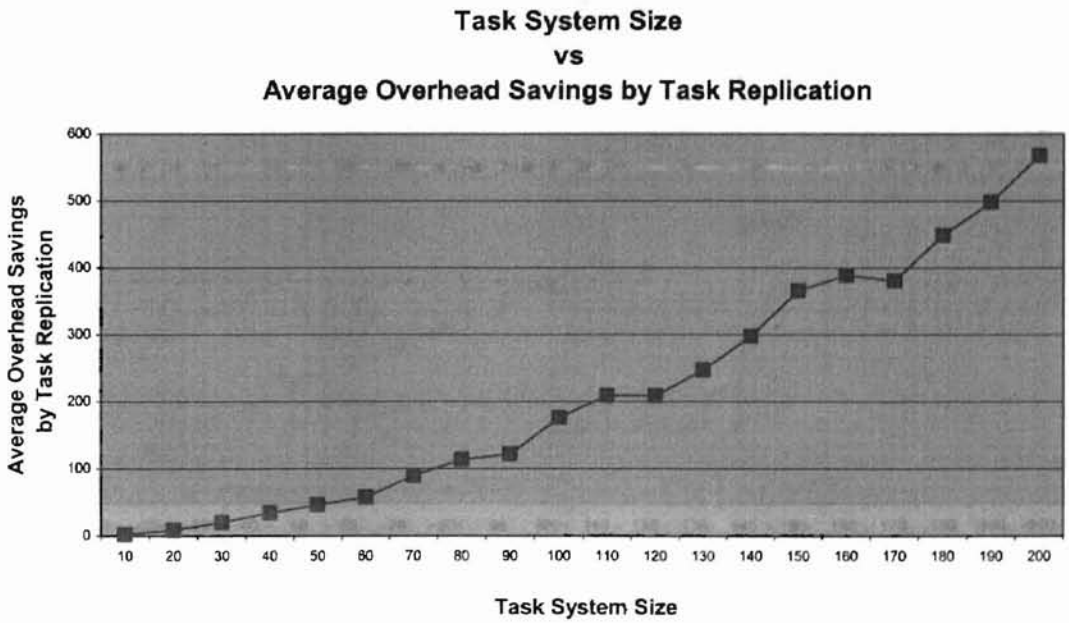


Figure 14. Average Inter-Processor Communication Overhead Savings by the Proposed Algorithm with Increasing number of Tasks in Task Systems, with Task Systems Generated Using TGG (number of task systems in each size is 10).

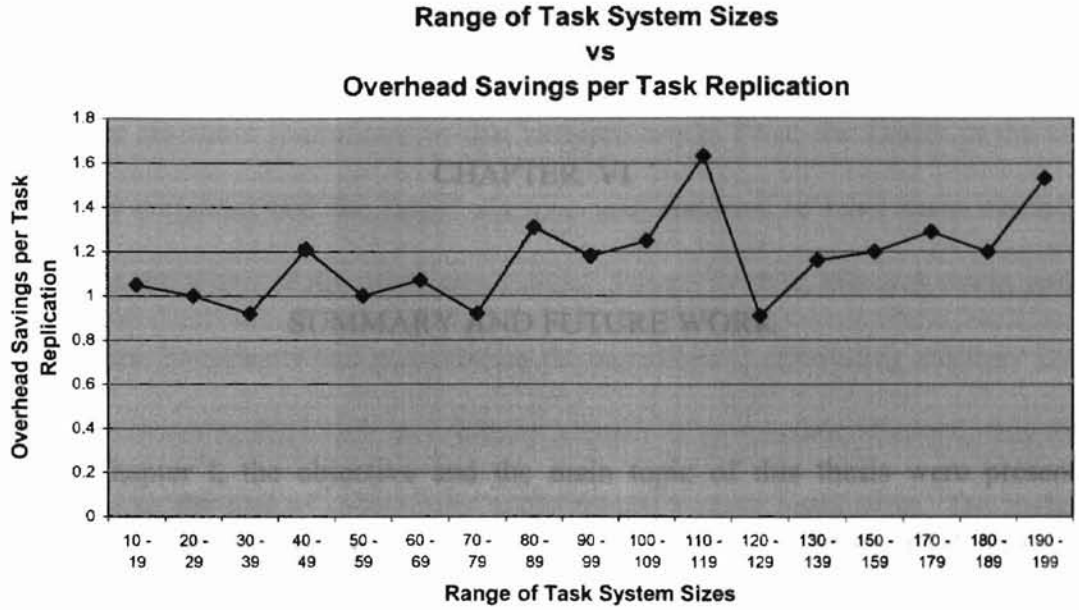


Figure 15. Average Inter-Processor Communication Overhead Savings Per Task Replication with Increasing Number of Tasks in Task System, with Task Systems Generated Using TGFF (number of task systems in each range is 10).

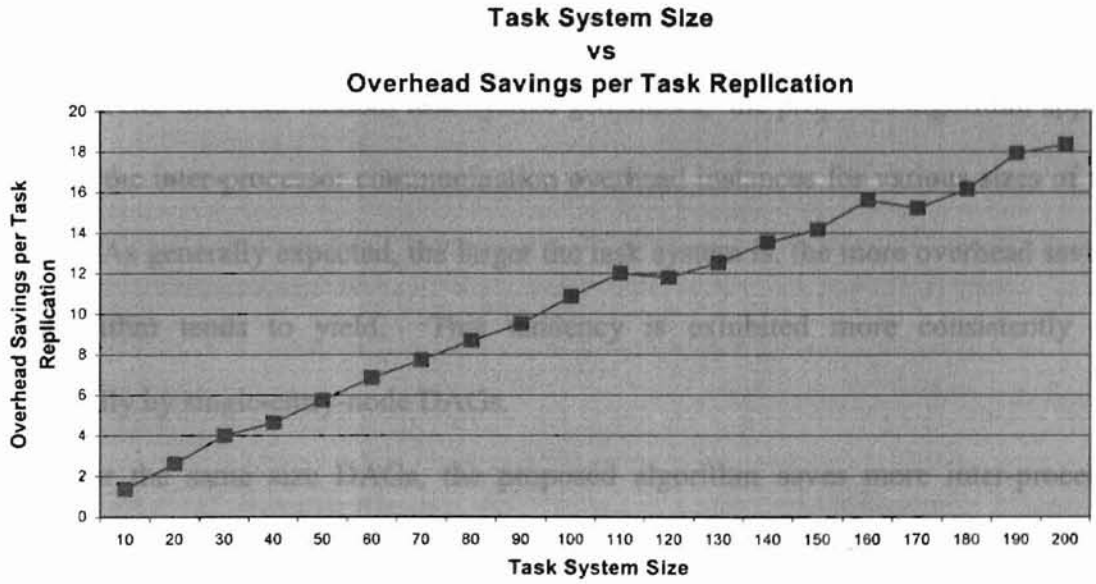


Figure 16. Average Inter-Processor Communication Overhead Savings Per Task Replication with Increasing Number of Tasks in Task System, with Task Systems Generated Using TGG (number of task systems in each size is 10).

CHAPTER VI

SUMMARY AND FUTURE WORK

In Chapter I, the objective and the main topic of this thesis were presented. Chapter II reviewed the previous research work on the subject. Chapter III described the approach adopted by this thesis and outlined the proposed algorithm. In Chapter IV, the relevant issues on the implementation of the algorithm were covered, and the details of the algorithm were given. In Chapter V, the experimentation procedure was explained and the test run results were illustrated and analyzed briefly.

Based on the experiments using the Task_System simulation program, which took input from two different random task system generators, the proposed algorithm appears to reduce the inter-processor communication overhead instances for various sizes of task systems. As generally expected, the larger the task system is, the more overhead savings the algorithm tends to yield. This tendency is exhibited more consistently and dramatically by single-entry-node DAGs.

For the same size DAGs, the proposed algorithm saves more inter-processor communication in the single-entry-node DAGs than the multiple-entry-node DAGs. In terms of overhead savings per replication, the result was close to one for multiple-entry-node DAGs (ranging from 0.91 to 1.63). For single-entry-node DAGs, the overhead

savings per replication grow quickly with the increase of task system sizes, climbing from 1.37 to 18.38.

There are some limitations on this research work: First, the DAGs in the study were equally weighted and the target machine was assumed to have same number of processors as the width of the respective DAGs. Future work in this area could include removing these constraints and generalizing the algorithm to scheduling arbitrary DAGs to a multiprocessor system with an arbitrary number of processors. Second, this thesis did not consider the cost of extra buffer space caused by task replication. The trade-off study in the future work could include this factor as part of the costs of task replication. Third, the algorithm uses the first feasible task sequence for replication. Future research could look into the possibility of using a greedy algorithm to select the most overhead-saving task for each replication.

REFERENCES

[Aho et al. 86] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[Ahmad and Ghafoor 91] I. Ahmad and A. Ghafoor, "Semi-Distributed Load Balancing for Massively Parallel Multicomputer Systems", *IEEE Transactions on Software Engineering*, Vol. 17, No. 10, pp. 987-1004, October 1991.

[Bhat 2000] S. S. Bhat, "Reducing Processor Communication Overhead in Multiprocessor Scheduling", Master of Science Thesis, Computer Science Department, Oklahoma State University, Stillwater, OK, 2000.

[Chu et al. 84] W. W. Chu, M.-T. Lan, and J. Hellerstein, "Estimation of Intermodule Communication (IMC) and Its Applications in Distributed Processing Systems", *IEEE Transactions on Computers*, Vol. C-33, No. 8, pp. 691-699, August 1984.

[Coffman 76] E. G. Coffman, Jr., *Computer and Job-Shop Scheduling Theory*, John Wiley, New York, NY, 1976.

[Coffman and Graham 72] E. G. Coffman and R. L. Graham, "Optimal Scheduling for Two-Processor Systems", *Acta Informatica*, Vol. 1, No. 3, pp. 200-213, February 1972.

[Dick et al. 98] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task Graphs for Free", *Proceedings of the Sixth International Workshop on Hardware/Software Codesign*, pp. 97-101, Seattle, WA, March 1998.

[Fishburn 85] P. C. Fishburn, *Interval Orders and Interval Graphs*, John Wiley & Sons, New York, NY, 1985.

[Fox et al. 88] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors: General Techniques and Regular Problems*, Vol. I, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Gajski and Peir 85] D. D. Gajski and J. Peir, "Essential Issues in Multiprocessors", *IEEE Computer*, Vol. 18, No. 6, pp. 9-27, June 1985.

[Garey and Johnson 79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, CA, 1979.

[Garey et al. 83] M. R. Garey, D. Johnson, R. Tarjan, and M. Yannakakis, "Scheduling Opposing Forests", *SIAM Journal on Algebraic Discrete Methods*, Vol. 4, No. 1, pp. 72-92, March 1983.

[Gonzalez 77] M. J. Gonzalez, "Deterministic Processor Scheduling", *Computing Surveys*, Vol. 9, No. 3, pp. 173-204, September 1977.

[Gurd et al. 85] J. R. Gurd, C. C. Kirkham, and I. Watson. "The Manchester Prototype Dataflow Computer", *Communications of the ACM*, Vol. 28, No. 1, pp. 34-52, January 1985.

[Hockney and Jesshope 81] R. W. Hockney and C. R. Jesshope, *Parallel Computers*, Hilger, Bristol, UK, 1981.

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

[Hwang and Briggs 84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Co., New York, NY, 1984.

[Kwok and Ahmad 99] Y. K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors", *ACM Computing Surveys*, Vol. 31 , No. 4, pp. 406-471, December 1999.

[Mandyam and Samadzadeh 92] 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, MO, March 1992.

[Palis et al. 95] M. A. Palis, J.-C. Liou, S. Rajasekaran, S. Shende, and D.S.L. Wei, "Online Scheduling of Dynamic Trees", *Parallel Processing Letters*, Vol. 5, No. 4, pp. 635-646, December 1995.

[Papadimitriou and Yannakakis 79] C. H. Papadimitriou and M. Yannakakis, "Scheduling Interval-Ordered Tasks", *SIAM Journal on Computing*, Vol. 8, No. 3, pp. 405-409, August 1979.

[Samadzadeh 91] F. A. Samadzadeh, "Task Graph Generation Program", Software created at Computer Science Department, Oklahoma State University, Stillwater, OK, 1991.

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

[Ullman 75] J. Ullman, "NP-Complete Scheduling Problems", *Journal of Computer and System Sciences*, Vol. 10, No. 3, pp. 384-393, October 1975.

APPENDICES

APPENDIX A

GLOSSARY

DAG	Directed acyclic graph.
In/Out Degree	The number of predecessors/successors of a node in a DAG.
Interval-Ordered DAG	Every two precedence-related nodes x, y can be mapped to two intervals on the real number line such that the interval assigned to x completely precedes the interval assigned to y [Fishburn 85].
Opposing Forest	The disjoint union of an in-forest (each task has at most one immediate successor) and an out-forest (each task has at most one immediate predecessor) [Garey et al. 83].
Parallel Processing	Utilizing the parallelism of a program to execute the concurrent tasks in the program on different processing units.
Precedence Graph	A graphical representation of a task system depicting the precedence relation among the tasks.
Rooted Tree	A precedence graph in which each node has at most one successor and any number of predecessors.
Schedule	Allocation of tasks to processing units.
Task	One sequential computation unit of a program or a problem.
Task System	A set of tasks with a precedence relation among them.
TGFF	Task Graphs For Free, a random task system generator software that generates multiple-entry, single-exit DAGs. Developed by Dick et al. at Princeton University [Dick et al. 98].
TGG	Task Graph Generation, a random task system generator software that generates single-entry, single-exit DAGs. Developed by Samadzadeh at Oklahoma State University [Samadzadeh 91].

APPENDIX B

PROGRAM LISTINGS

This appendix consists of two parts. The first part is the simulation program called Task_System that implements the proposed algorithm and Hu's Algorithm, and calculates the corresponding inter-processor communication overhead for each algorithm. The second part consists of the supplementary programs and scripts used to work with the simulation program and the task system generator programs to automate the experimentation and outputting of the results.

PART ONE

The first part of Appendix B is the simulation program that implements the proposed algorithm and Hu's Algorithm, and calculates the corresponding inter-processor communication overhead for each algorithm.

```
/*********************************************************************/
/*  Program: Reducing Inter-Processor Communication Overhead         */
/*           By Task Replication                                     */
/*  Author:  Jun Su                                                  */
/*  Advisor: Dr. Mansur H. Samadzadeh                                */
/*  Date:    July 2002                                               */
/*********************************************************************/

/*********************************************************************
/*     The purpose of the program is to implement an algorithm designed
/*  to reduce Inter-Processor Communication Overhead by the approach
/*  of task replication.
/*     First a random generated task system is read from a random task
/*  system generator program, in the form of single rooted, equally
/*  weighted DAG. Then the task system is scheduled to a multiprocessor
/*  system, following Hu's Algorithm and the proposed algorithm,
/*  assuming the system has same number of processor as the width of
/*  the schedule DAG. Based on the allocation by Hu's Algorithm.The
/*  proposed algorithm realizes the Inter-Processor Communications by
/*  replicating some tasks in the task system and allocating them to
/*  appropriate processors.  The proposed algorithm has the same
/*  scheduling length as Hu's Algorithm. Finally, the total number of
/*  Inter-Processor Communication Overhead is calculated respectively
/*  for both algorithm, showing the savings by the proposed algorithm
/*  over Hu's Algorithm. The number of replications is also shown.
/*     The major methods of Task_System are as follows:
/*  Read_TGFF_Graph()and Read_TGG_Graph() are used to input from the
```

```
/*  two random task system generator, in the format of DAG.
/*  Find_Level_Width() finds the level of each task node and the width
/*  in the DAG. Assign_by_Hu() follows Hu's Algorithm to schedule the
/*  task system to a multiprocessor system. Assign_by_Proposed()
/*  duplicates the appropriate tasks to the calculated processors.
/*  Total_OH() shows the number of overhead instance for each of the
/*  two algorithms.
/*  Four classes are designed to represent the whole task system and
/*  implement the algorithms. The Task_Node is the object representing
/*  a task in the task system. The Task_List class is a linked list of
/*  Task_Node objects. The Task class contains the task number, level
/*  number, its assigned processor, and the lists of its predecessors
/*  and successors. The Task_System class represents the scheduled task
/*  system, which holds information including each Task objects, the
/*  number of tasks,DAG width, maximum level number, and the allocation
/*  of the scheduled tasks.
/*********************************************************************/


#include <cstdlib>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
using namespace std;

/* The class Task_Node is used to store the task number of a task in a Task_List. It
contains the task number of the task and a pointer to another task node.*/
class Task_Node
{
          int num; //the task number
          Task_Node *next;
          friend class Task_List;
public:
    /* constructor */
          Task_Node(int x) { num=x; next=NULL; }

    /* return the task number. */
          int get_num() const { return num;}

};


/* The class Task_List is a linked list holding the numbers of Task_Nodes
representing the tasks. The iter pointer is an iterator class used to traverse the list.
*/
class Task_List
{
          Task_Node *head; //head pointer
          Task_Node *iter; //iterator class
          int size; //the size of the linked list

public:
          /* constructors */
          Task_List()
               {
                              head=iter=NULL;
                              size=0;
                    }
          Task_List(int x)
               {
                              iter=head=new Task_Node(x);
                    size=1;
                      }

          /* return the size of the list */
          int getsize() const { return size; }

          /* test if the Task_List is empty */
        bool is_empty() const { return head==NULL; }
```

```
        /* search for a task number in the list */
          bool search(const int );

          /* add a task number to head of the list */
          void add(const int);

        /* append a task number to the end of the list */
          void append(const int);

        /* remove the head node of the list and return the pointer pointing
           to it */
          Task_Node* dequeue();

        /* print the whole list */
          void print();

          /* The methods below are used to traverse the Task_List by an
         iterator. */

        //return the task number of task node pointed by the interator
          int get_iter() const { return iter->num;}

        //test if the interator reach the end of the list
          bool is_end() const { return iter==NULL;}


        // move the iterator to the next node in the list
          void iterate() { iter=iter->next;}

          // reset the iterator to the head of the list
        void reset_iter() {iter=head;}

};


/* search the Task_List for a particular task by the given task number. */
bool Task_List::search(const int i)
{
          while (iter!=NULL)
          {
                    if (iter->num==i)
                    {
                              reset_iter();
                              return true;
                    }
                    iterate();
          }

          reset_iter();
          return false;

}


/* Add a task into the head of the Task_List. */
void Task_List::add(const int i)
{
          Task_Node *n=new Task_Node(i);
          if (head==NULL )
          {
                    head=n;
                    n->next=NULL;
          }
          else
          {
                    n->next=head;
                    head=n;
          }

          size++;
          iter=head;
```

```
}

/* Append a task at the end of the Task_List. */
void Task_List::append(const int i)
{
          Task_Node *n=new Task_Node(i);
          if (is_empty())
          {
                    head=n;
                    n->next=NULL;
          }
        else
          {
                    while (iter->next!=NULL) iterate();
                    iter->next=n;
                    n->next=NULL;
          }

          reset_iter();
          size++;

}


/* Remove head task from the Task_List and return a pointer to it. */
Task_Node* Task_List::dequeue()
{
          Task_Node* temp=head;
          head=head->next;
          size--;
          return temp;
}


/* Print out the Task_List. */
void Task_List::print()
{
                    while (!is_end())
                    {
                    cout << iter->num << " ";
                    iterate();
                    }
                    reset_iter();
}


/* The class Task store all information about a task, including task number,
   its level number, which processor it is assigned and its predecessors and
   successors by Task_List class. */
class Task
{
          /* the task number */
          int task_num;

          /* the list of predecessors */
          Task_List* Pred;

          /* the list of successors */
          Task_List* Succ;

          /* level number by Hu's labeling algorithm */
          int level;

          /* the processor the task is assigned to. */
          int processor;

public:
          /* constructor */
          Task(){level=0; Pred=NULL; Succ=NULL; };
```

```
        /* set the task number */
        void set_task_num(const int j) {task_num=j;}

        /* return the task number */
        int get_task_num() const { return task_num;}

      /* Add a predecessor to the predecessor list. */
      void add_pred(const Task&);

      /* Add a successor to the successor list. */
      void add_succ(const Task&);

        /* add task t to the predecessor list of the invoking task
           and add the invoking task to the successor list of task t.*/
        void connect(Task& t) {       add_pred(t); t.add_succ(*this);        }

        /* Return the predecessor list. */
        Task_List* get_Pred() { return Pred; }

        /* Return the successor list. */
        Task_List* get_Succ() { return Succ; }

        /* Set the level number. */
        void set_level(const int l) { level=l;}

        /* return the level number. */
        int get_level() const { return level;}

        /* set the processor number assigned for the task. */
        void set_processor(const int p) { processor=p;}

      /* return the processor number assigned for the task. */
        int get_processor() const { return processor;}

        /* return the total number of predecessors of the task. */
        int get_pred_num() const;

      /* return the total number of successors of the task. */
        int get_succ_num() const;

};


/* return the total number of predecessors. */
int Task::get_pred_num() const
{
        if (!Pred) return 0;
        return Pred->getsize();
}


/* return the total number of successors. */
int Task::get_succ_num() const
{
        if (!Succ) return 0;
        return Succ->getsize();
}

/* Add a task to the predecessor list. */
void Task::add_pred(const Task& t)
{
                if (Pred==NULL)
                        Pred=new Task_List(t.get_task_num());
                else
                        Pred->add(t.get_task_num());
}


/* Add a task to the successor list. */
void Task::add_succ(const Task& t)
{
```

```
                    if (Succ==NULL)
                              Succ=new Task_List(t.get_task_num());
                    else
                              Succ->add(t.get_task_num());
}


/* The Task_System class represents a task system in DAG format. It consists of a list of
tasks, the root task, the total number of levels, the width of the DAG, an two-dimention
array holding scheduing information including the time-unit/processor combination for
each task. */

class Task_System
{
          /* the list of tasks in the DAG */
          Task *T;

          /* the number of tasks in the system */
          int size;

          /* the number of the root task */
          int root;

          /* the width of the DAG */
          int width;

        /* the number of levels in the DAG */
          int L;

        /* the allocation of tasks for each time unit and each processor */
          int* slot;

public:
        /* read the task system from the task system generator TGFF */
          void Read_TGFF_Graph();

        /* read the task system from the task system generator Task Graph
           Generator */
          void Read_TGG_Graph();

          /* get the total number of levels and the width of the DAG. */
          void Find_Level_Width();

        /* get the total number of tasks in the task system. */
        int get_total_num() const { return size; }

          /*schedule and allocate each task to a processor by Hu's Algorithm.*/
          void Assign_by_Hu(ofstream&);

          /* schedule and allocate each task to a processor by the proposed
         algorithm. */
          void Assign_by_Proposed(int&, ofstream&);

          /* compute the total inter-processor communication overhead for the
         task system. */
          int Total_OH() const;

};


/* Read a task system randomly generated by TGFF program. The task system is
   in file "seed.tgff".
   The format of DAG is a number of arcs connecting pairs of task nodes. */
void Task_System::Read_TGFF_Graph()
{
          //open the file which contains DAG generated by TGFF program
          ifstream ifile("seed.tgff");

          Task_List end;   //the end list contains end nodes in arcs
          Task_List start; //the start list contains start nodes in arcs
```

```
        char *input = new char[80];
        char *task_temp= new char[80];
        int max=0;

      //get start and end node for each arc and add to the Task_List start
      //and end accordingly until the end of file
        while (true)
        {
                do
                {
                        ifile >> input;
                } while (strcmp(input, "FROM") && strcmp(input, "TO"));

                ifile >> task_temp;
                string task(task_temp);
                task.erase(0,3);
                int i=atoi(task.c_str());
                if (!strcmp(input, "FROM"))
                  end.append(i);
                if (!strcmp(input, "TO"))
                {
                        start.append(i);
                        if (i>max) max=i;
                        ifile.ignore(80,'\n');
                        if (ifile.peek()=='\n')
                                break;
                }
        }//end while true

        size=max+1;  //the size of the task system
        T=new Task[size]; //allocate memory for the tasks

      //set the task number for each task
        int j=0;
        for ( j=0; j<size; j++)
                T[j].set_task_num(j);

      //add precedence relations to each task
        for ( j=0; j<start.getsize(); j++)
        {
                T[end.get_iter()].connect(T[start.get_iter()]);
                start.iterate();
                end.iterate();
        }
        start.reset_iter();
        end.reset_iter();

        root=0;
        T[0].set_level(1); //set level of root task as one

        ifile.close();

}//end of Read_TGFF_Graph


/* Read a task system generated by task_graph.pas program. The format is an upper
triangular adjacency matrix.   */
void Task_System::Read_TGG_Graph()
{
    /* open the file which contains the generated DAG in adjacency matrix
       format */
        ifstream ifile("graphs.out");

        Task_List end; //the end list contains end nodes in arcs
        Task_List start; //the start list contains start nodes in arcs

        int max=0;
        char *input = new char[80];
        ifile.seekg(3,ios::beg);

        ifile >> max; //the number of tasks specified by the input file
```

```
        size=max;
        T=new Task[size]; //allocate memory for the tasks

        //set the task number for each task
        int j=0;
        for ( j=0; j<size; j++)
                T[j].set_task_num(j);

        //add precedence relations to each task
        char c;
        int i=0;
        for ( i=0; i<size; i++)
        {
                ifile.ignore(80, '\n');
                for (j=0; j<size; j++)
                {
                        ifile.get(c);
                        if (j<i) continue;
                        if (c=='1')          T[j].connect(T[i]);
                }//end for j=0

        }//end for i=0

        root=size-1;
        T[size-1].set_level(1); //set root task level as one

}//end of Read_TGG_Graph


/* The method calculate the level number for each task and the width for
   the whole task system.  A queue of Task_Node, implemented by Task_List,
   is utilized to accomplish the objective. The width of the DAG is the
   maximum number of task in each level.    */
void Task_System::Find_Level_Width()
{
        L=0;//number of levels

      //Q is a queue to hold each task, starting from root node
        Task_List Q;

        Q.append(root); //append the root task to the queue

      /* Each time task i is dequeued. all its predecessors's level are set
         one more than i's level. If any predecessor is not in the Q, append
         it. Keep dequeuing until Q is empty. */
         while(!Q.is_empty())
         {
                Task_Node* dequeued=Q.dequeue();

          //get the predecessor of the dequeued node
                Task_List* pred=T[dequeued->get_num()].get_Pred();

                while (pred && !pred->is_end())
                {
                        int pred_num=pred->get_iter();

                    //level of the dequeued node
                        int curr_level=T[dequeued->get_num()].get_level();

                        /* set the predecessor level as one plus the level of
                       the dequeued node. */
                        if ((curr_level+1)>T[pred_num].get_level())
                                T[pred_num].set_level(curr_level+1);

                    //keep track of the maximum level number L
                        if ((curr_level+1)>L)
                                L=curr_level+1;

                    //if the predecessor is not in Q, append it to Q
                        if (!Q.search(pred_num))
                                Q.append(pred_num);
```

```
                        pred->iterate();

                }//end while pred

                if (pred) pred->reset_iter();

        }//end while Q is empty


      // find the width of the task system
        int *W=new int[L+1];
        int i=0;
        for (i=1;i<L+1; i++)
                W[i]=0;

        for (i=0; i<size; i++)
                W[T[i].get_level()]++;

        width=W[1];
        for ( i=1; i<L+1;i++)
        {
                if (width<W[i])
                        width=W[i];
        }
        delete W;

    //allocate memory for each time unit and each processor
        int slot_sum=width*L;
        slot=new int[slot_sum];

        for (i=0; i<slot_sum; i++) //idle processor represented as -1
                slot[i]=-1;

}//end of Find_Level_Width()


/* Allocate each task in the task system in a multiprocessor system, given the number of
processor is width of the task system. The schedule is printed out. */
void Task_System::Assign_by_Hu(ofstream& out) {

        int i,j;

        int *P=new int[L+1];//array to keep track of which processor is idle
        for (i=1;i<L+1;i++)
                P[i]=1;

        for ( int t=0; t<size; t++)
        {
                int slot_num=L*(P[T[t].get_level()]-1)+T[t].get_level()-1;
                slot[slot_num]=t;
                T[t].set_processor(P[T[t].get_level()]);
                P[T[t].get_level()]++;
        }
        delete P;

        for (i=0; i<width; i++)
        {
                for (j=L; j>0; j--)
                        out << setw(3) << slot[i*L+j-1] << " ";
                out << endl;
        }

}//end method Assign_by_Hu


/* The proposed algorithm: replicating selected tasks to calculated idle
   processor could reduce communication overhead to certain extent.
   Given a task system scheduled by Hu's Algorithm, follow the steps below:
   for i=2, ..., L do
      for each idle processor P
```

```
                for each task t in level i {
                        S denotes the number of t's successors on P;
                          R denotes the number of t's prdecessors;
                          Rp denotes the number of t's predecessors on P;
                           if S > R - Rp
                                 then {
                                       duplicate t to P at time unit I
                                             if t has any predecessor p at level i+1
                                                which as no predecessor and P is idle
                                                at time unit i+1
                                             then
                                                duplicate p to P at time unit i+1
                                                break
                                             }
                                    else if S>0 and S=R-Rp
                                       then {
                                             if t has any predecessor p at level i+1
                                                which as no predecessor and P is idle
                                                at time unit i+1
                                             then {  duplicate t to P at time unit i
                                                     duplicate p to P at time unit i+1
                                                     break     }
                }

*/
void Task_System::Assign_by_Proposed(int& dup, ofstream& out)
{
          int i,j;
          for (i=2; i<=L; i++) // for each level
          {
                    int num=1;//the number of tasks in each level, at least one;

            //the number of the idle processors at the beginning,at least two
                    int p=2;

                    while (slot[L*(p-1)+i-1]!=-1 && p<=width)
                    {
                              num++;
                              p++;
                    }//find the available slot

                    if(num>=width) continue; //if level i has no idle processor

                    for (; p<=width; p++) //for each available slot
                    {
                              for (j=0; j<num; j++) //for each task in that level
                              {
                                        int t=j*L+i-1;

                                        int S=0;
                                        int Rp=0;

                                        int R=T[slot[t]].get_pred_num(); //get R

                                        Task_List* succ=T[slot[t]].get_Succ();
                                        Task_List* pred=T[slot[t]].get_Pred();

                                        while (succ && !succ->is_end()) //calculate S
                                        {
                                                  int s=succ->get_iter();
                                                  if (T[s].get_processor()==p)
                                                            S++;
                                                  if (succ) succ->iterate();
                                        }//end while

                                        if (succ) succ->reset_iter();


                                        while (pred && !pred->is_end()) //calculate Rp
                                        {
                                                  int s=pred->get_iter();
```

```
                if (T[s].get_processor()==p)
                        Rp++;
                if (pred) pred->iterate();

        }//end while

        if (pred) pred->reset_iter();

        if (S > (R - Rp))
        {
        //duplicate t to the open slot;
                slot[L*(p-1)+i-1]=slot[t];

        dup++;

                Task_List* pred=T[slot[t]].get_Pred();
                for (int m=0; m<width; m++)
                {
                if (pred && pred->search(slot[m*L+i]))
                {
                if(T[slot[m*L+i]].get_Pred()==NULL
              && slot[(p-1)*L+i]==-1 )
                {
              /* duplicate t's predecessor to
                 the open slot; */
                        slot[L*(p-1)+i]=slot[m*L+i];

                        dup++;
                        break;
                }//end if

                }//end if

                }//end for m=0

                break;
        }//end if S>R-Rk

        else if (S>0 && S==R-Rp) {

                Task_List* pred=T[slot[t]].get_Pred();
                for (int m=0; m<width; m++)
                {
                if (pred && pred->search(slot[m*L+i]))
                {
                if (T[slot[m*L+i]].get_Pred()==NULL
              && slot[(p-1)*L+i]==-1 )
                {

            //duplicate t to the open slot;
                        slot[L*(p-1)+i-1]=slot[t];

           /* duplicate t's predecessor to
              the open slot;*/
                        slot[L*(p-1)+i]=slot[m*L+i];

                        dup=dup+2;
                        break;
                }//end if

                }//end if

                }//end for m=0

                break;
        }//end if S>0

        else ;

}//end for each task in that level
```

```
                }//end for each available slot

        }//end for i=0


        //print out the schedule
        for (i=0; i<width; i++)
        {
                for (j=L; j>0; j--)
                        out << setw(3) << slot[i*L+j-1] << " ";
                out << endl;
        }

}//end method Assign_By_Proposed


/* The method which computes the total number of communication overhead among
   the processors. The calculation is based on the precedence between the
   tasks already scheduled on different slot. */
int Task_System::Total_OH() const
{
        int OH=0;

        for (int i=0; i<L; i++)       {

                for (int j=0; j<width; j++)
                {
                        int temp=slot[j*L+i];
                        if (temp==-1) continue;

                        Task_List* pred=T[temp].get_Pred();

                        while (pred && !pred->is_end())
                        {
                                int p=pred->get_iter();
                                int k=0;
                                for ( k=i+1; k<L; k++)
                                {
                                        if (slot[j*L+k]==p)
                                                break;
                                }

                                if (k==L) OH++;
                                pred->iterate();

                        }//end while

                        if (pred) pred->reset_iter();

                }//end inner for

        }//end outer for

        return OH;
}


/* The main function to call the methods and display the results. */
int main(int argc, char* argv[]) {

        //the output file which shows the schedules of two algorithms
        ofstream outfile("schedule", ios::out | ios::app);

        /* the output file which shows the overhead savings vs number of
       replications. */
        ofstream ofile("result.txt", ios::out | ios::app);

        Task_System G;
        int Total_Dup=0; //total number of task replications

        if (!strcmp(argv[1], "pascal"))
```

```
                G.Read_TGG_Graph();
        else if (!strcmp(argv[1], "tgff"))
                G.Read_TGFF_Graph();
        else {
                cout << "Invalid Graph Generation Program!" << endl;
                exit(-1);
        }

    // label the task level number and find the width of the task system
        G.Find_Level_Width();

        /* schedule by Hu's Algor and output the schedule to the output file
       accordingly */
        G.Assign_by_Hu(outfile);
        int Hu_OH=G.Total_OH();
        outfile << "Total OH by Hu's Algor is "<< Hu_OH << "." << endl;

        /* schedule by Proposed Algorithm and output the schedule to the output
       file accordingly */
        G.Assign_by_Proposed(Total_Dup, outfile);
        int Proposed_OH=G.Total_OH();
        outfile << "Total OH by Proposed Algorithm is "<< Proposed_OH << "."<< endl;

        //calculate overhead savings by the proposed algorithm
        int Total_Savings = Hu_OH - Proposed_OH;
        ofile << G.get_total_num() << '\t' << Total_Dup << '\t' <<
          Total_Savings<< endl;

        outfile.close();
        ofile.close();
        return 0;
}
```

PART TWO

The second part of Appendix B consists of the supplementary programs and scripts used to work with the simulation program and the task system generator programs to automate the experimentation and outputting of the results.

1) Supplementary Programs for Test Runs Using TGFF (Task Graphs For Free) [Dick et al. 98] as Input.

```
/*********************************************************************/
/*  Program: Generating Parameter File for TGFF Program              */
/*  Author:  Jun Su                                                  */
/*  Advisor: Dr. Mansur H. Samadzadeh                                */
/*  Date:    July 2002                                               */
/*********************************************************************/


#include <fstream.h>

int main()
{
        int NUM, count;

      //seeds for TGFF
        int seed[]={0, 5, 36, 99, 162, 381, 747, 1893, 6790, 34905};

      //the file which contains the minimum number of tasks and count of runs
        ifstream in("run");

      //the parameter file for TGFF
        ofstream out("seed.tgffopt");

        in >> NUM >> count;
        out << "seed " << seed[count] << endl; //seeds
        out << "tg_cnt " << 1 << endl;  //number of task system in the file
        out << "task_cnt " << NUM << " 0" << endl; //minimum number of tasks
        out << "task_degree " << NUM << " " << NUM << endl; //in and out degree

        out << endl << "tg_write" << endl;

        in.close();
        out.close();
        return 0;
}


/*********************************************************************/
/*  Program: Outputing Final Data(TGFF)                              */
/*  Author:  Jun Su                                                  */
/*  Advisor: Dr. Mansur H. Samadzadeh                                */
/*  Date:    July 2002                                               */
/*********************************************************************/

#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
```

```
int main()
{
          // the input file which contains the minimum task number and the
        // count of runs
          ifstream infile("run");

        //intermediate input file which contains the run result for each DAG
          ifstream in("result.txt");

        //the output file which contains the final data
          ofstream out("result", ios::out | ios::app);

          out.setf(ios::fixed, ios::floatfield);
          out.setf(ios::showpoint);

          //number of replications and total overhead savings
          int dup=0, savings=0;
          int temp;
          int run[20];
          int dup_sum[20], savings_sum[20];
          double avg[20], savings_avg[20];
          int i;

          for (i=0;i<20;i++)
        {
                    dup_sum[i]=0;
                    savings_sum[i]=0;
                    run[i]=0;
          }

          while(in)
        {
                    in >> temp;
                    if (temp>=200) {
                              in.ignore(80, '\n');
                              continue;
                    }
                    in >> dup;
                    in >> savings;
                    in.ignore(80,'\n');
                    dup_sum[temp/10]+=dup;
                    savings_sum[temp/10]+=savings;
                    run[temp/10]++;
          }// end while

          for (i=0; i<20; i++)
          {
                    //discard the results if the total run count for that DAG size
                //is smaller than 5
                    if (run[i]<5) continue;

                    if (dup_sum[i]==0) {
                              avg[i]=0;
                              savings_avg[i]=0;
                    }
                    else
                    {
                              avg[i]=(double)savings_sum[i]/(double)dup_sum[i];
                              savings_avg[i]=(double)savings_sum[i]/(double)run[i];
                    }

                    out  << setw(3) << i*10 << " - " << setw(3) << i*10+9 << "\t"
                     <<  setprecision(2) << avg[i] << "\t" << savings_avg[i]
                     << endl;

          }//end for

          return 0;
```

```
}

/*********************************************************************/
/*  Program: Looping (UNIX Shell Script)                             */
/*  Author:  Jun Su                                                  */
/*  Advisor: Dr. Mansur H. Samadzadeh                                */
/*  Date:    July 2002                                               */
/*********************************************************************/

NUM=5
while ((NUM <= 100))
  do
     count=0

     while ((count < 10))
        do
           print $NUM $count > run
           rand
           tgff seed
           task tgff
           ((count=count+1))
        done

     ((NUM=NUM+5))
  done
output
rm result.txt
rm seed*
```

2) Supplementary Programs for Test Runs Using TGG (Task Graph Generation) [Samadzadeh 91] as Input.

```
/*********************************************************************/
/*  Program: Generating Parameter File for TGG Program               */
/*  Author:  Jun Su                                                  */
/*  Advisor: Dr. Mansur H. Samadzadeh                                */
/*  Date:    July 2002                                               */
/*********************************************************************/

#include <fstream.h>
#include <stdlib.h>

int main()
{

          int NUM,count;

          //seeds for TGG Program
          double seed[]={0.0005, 0.12, 2.4, 14.9, 89.32, 578.01,
                    1238.3, 9962, 23973.4, 897865.32};

          //the file containing the minimum number of tasks and count of runs
          ifstream in("run");

         //the parameter file for TGG
          ofstream out("infile");

          in >> NUM;
          in >> count;
          out << seed[count] << endl;
          out << 1 << endl;
          out << "E" << endl;
          out << NUM << endl;
          out << NUM << endl;

          out.close();
          return 0;
```

```
}


/*********************************************************************/
/*  Program: Outputing Final Data(TGG)                               */
/*  Author:  Jun Su                                                  */
/*  Advisor: Dr. Mansur H. Samadzadeh                                */
/*  Date:    July 2002                                               */
/*********************************************************************/

#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>

int main()
{
        // the input file which contains the minimum task number and the
        // count of runs
          ifstream infile("run");

        //intermediate input file which contains the run result for each DAG
          ifstream in("result.txt");

        //the output file which contains the final data
          ofstream out("result", ios::out | ios::app);

          out.setf(ios::fixed, ios::floatfield);
          out.setf(ios::showpoint);

          //number of replications and total overhead savings
          int dup=0, savings=0;
          int NUM;
          int dup_sum=0, savings_sum=0;

        while(in)
        {
                    in >> NUM;
                    in >> dup;
                    in >> savings;
                    in.ignore(80,'\n');
                    dup_sum+=dup;
                    savings_sum+=savings;
          }
          double avg=(double)savings_sum/(double)dup_sum;
          double savings_avg=(double)savings_sum/(double)10;

        infile >> NUM;
          out  << NUM << "\t" <<  setprecision(2) << avg << "\t" << savings_avg
             << endl;

        return 0;
}

/*********************************************************************/
/*  Program: Looping (UNIX Shell Script)                             */
/*  Author:  Jun Su                                                  */
/*  Advisor: Dr. Mansur H. Samadzadeh                                */
/*  Date:    July 2002                                               */
/*********************************************************************/

NUM=10
while ((NUM <= 200))
  do
     count=0

     while ((count < 10))
        do
           print $NUM $count > run
           rand
           a.out < infile
           task pascal
```

```
        ((count=count+1))
      done

   output
   rm result.txt
   rm infile
   ((NUM=NUM+10))
done
```

APPENDIX C

OUTPUT LISTINGS

This appendix consists of two output listings. The first part shows the experimentation results, including the data for Average Overhead Savings by Replication and Overhead Savings per Replication for TGFF and TGG, respectively. The second part includes a fragment of the schedules by Hu's Algorithm and the proposed algorithm for the DAGs generated by TGFF and TGG, respectively.

PART ONE

The first part of Appendix C shows the experimentation results, including the data for Average Overhead Savings by Replication and Overhead Savings per Replication for TGFF and TGG, respectively.
For each size range, ten task systems are generated randomly by TGG and TGFF. The data for each size range is obtained by averaging the data for the ten task systems of the same size range.

Number of Tasks in the task systems	Overhead Savings per Replication for all task systems in the range	Average Overhead Savings by Replication for all task systems in the range
10 - 19	1.05	1.67
20 - 29	1.00	3.11
30 - 39	0.92	3.92
40 - 49	1.21	4.73
50 - 59	1.00	0.86
60 - 69	1.07	14.00
70 - 79	0.92	5.83
80 - 89	1.31	13.45
90 - 99	1.18	15.56
100 - 109	1.25	12.60
110 - 119	1.63	12.36
120 - 129	0.91	27.60
130 - 139	1.16	11.60
150 - 159	1.20	59.22
170 - 179	1.29	47.70
180 - 189	1.20	24.43
190 - 199	1.53	62.00

Data Output by the Proposed Algorithm using TGFF Task Systems as Input.

Number of Tasks in the task systems	Overhead Savings per Replication for all task systems in the size	Average Overhead Savings by Replication for all task systems in the size
10	1.37	2.60
20	2.62	8.90
30	4.00	20.40
40	4.64	34.30
50	5.74	46.50
60	6.85	57.50
70	7.72	89.60
80	8.68	113.70
90	9.52	121.90
100	10.85	175.80
110	12.01	209.00
120	11.80	208.80
130	12.52	246.70
140	13.52	297.50
150	14.17	365.50
160	15.61	388.70
170	15.23	380.70
180	16.15	448.90
190	17.92	498.10
200	18.38	567.90

Data Output by the Proposed Algorithm using TGG Task Systems as Input.

PART TWO

The second part of Appendix C includes a fragment of the schedules by Hu's Algorithm and the proposed algorithm for the DAGs generated by TGFF and TGG, respectively.
Each row represents a processor.
Each column represents a time unit.
-1 denotes idle on a particular processor during a particular time unit.
OH denotes Inter-Processor Communication Overhead.

1) Schedules by Hu's Algorithm and the Proposed Algorithm using TGFF as input.

```
 16   10    9    8    7    5    3    2    1    0
 17   11   -1   -1   -1    6    4   -1   -1   -1
 18   12   -1   -1   -1   -1   -1   -1   -1   -1
 19   13   -1   -1   -1   -1   -1   -1   -1   -1
 20   14   -1   -1   -1   -1   -1   -1   -1   -1
 21   15   -1   -1   -1   -1   -1   -1   -1   -1
Total OH by Hu's Algorithm is 18.
 16   10    9    8    7    5    3    2    1    0
 17   11   -1    8   -1    6    4   -1   -1   -1
 18   12   -1   -1   -1   -1   -1   -1   -1   -1
 19   13   -1   -1   -1   -1   -1   -1   -1   -1
 20   14   -1   -1   -1   -1   -1   -1   -1   -1
 21   15   -1   -1   -1   -1   -1   -1   -1   -1
Total OH by Proposed Algorithm is 17.

 15   14   12   11    9    1    0
 16   -1   13   -1   10    2   -1
 17   -1   -1   -1   -1    3   -1
 18   -1   -1   -1   -1    4   -1
 19   -1   -1   -1   -1    5   -1
 20   -1   -1   -1   -1    6   -1
 21   -1   -1   -1   -1    7   -1
 22   -1   -1   -1   -1    8   -1
Total OH by Hu's Algorithm is 31.
 15   14   12   11    9    1    0
 16   -1   13   -1   10    2   -1
 17   -1   -1   -1   -1    3   -1
 18   -1   -1   -1   -1    4   -1
 19   -1   -1   -1   -1    5   -1
 20   -1   -1   -1   -1    6   -1
 21   -1   -1   -1   -1    7   -1
 22   -1   -1   -1   -1    8   -1
Total OH by Proposed Algorithm is 31.

  7    6    3    2    1    0
  8   -1    4   -1   -1   -1
  9   -1    5   -1   -1   -1
 10   -1   -1   -1   -1   -1
 11   -1   -1   -1   -1   -1
 12   -1   -1   -1   -1   -1
 13   -1   -1   -1   -1   -1
Total OH by Hu's Algorithm is 10.
  7    6    3    2    1    0
  8   -1    4   -1   -1   -1
```

```
 9  -1   5  -1  -1  -1
10  -1  -1  -1  -1  -1
11  -1  -1  -1  -1  -1
12  -1  -1  -1  -1  -1
13  -1  -1  -1  -1  -1
Total OH by Proposed Algorithm is 10.

22  16  10   9   1   0
23  17  11  -1   2  -1
24  18  12  -1   3  -1
25  19  13  -1   4  -1
26  20  14  -1   5  -1
27  21  15  -1   6  -1
28  -1  -1  -1   7  -1
29  -1  -1  -1   8  -1
30  -1  -1  -1  -1  -1
31  -1  -1  -1  -1  -1
32  -1  -1  -1  -1  -1
33  -1  -1  -1  -1  -1
Total OH by Hu's Algorithm is 35.
22  16  10   9   1   0
23  17  11  -1   2  -1
24  18  12  -1   3  -1
25  19  13  -1   4  -1
26  20  14  -1   5  -1
27  21  15  -1   6  -1
28  -1  -1  -1   7  -1
29  -1  -1  -1   8  -1
30  -1  -1  -1  -1  -1
31  -1  -1  -1  -1  -1
32  -1  -1  -1  -1  -1
33  -1  -1  -1  -1  -1
Total OH by Proposed Algorithm is 35.

19  12   7   2   1   0
20  13   8   3  -1  -1
-1  14   9   4  -1  -1
-1  15  10   5  -1  -1
-1  16  11   6  -1  -1
-1  17  -1  -1  -1  -1
-1  18  -1  -1  -1  -1
Total OH by Hu's Algorithm is 16.
19  12   7   2   1   0
20  13   8   3  -1  -1
-1  14   9   4  -1  -1
-1  15  10   5  -1  -1
-1  16  11   6  -1  -1
19  17  -1  -1  -1  -1
-1  18  -1  -1  -1  -1
Total OH by Proposed Algorithm is 15.

28  27  17   8   7   6   5   1   0
-1  -1  18   9  -1  -1  -1   2  -1
-1  -1  19  10  -1  -1  -1   3  -1
-1  -1  20  11  -1  -1  -1   4  -1
-1  -1  21  12  -1  -1  -1  -1  -1
-1  -1  22  13  -1  -1  -1  -1  -1
```

```
 -1   -1   23   14   -1   -1   -1   -1   -1
 -1   -1   24   15   -1   -1   -1   -1   -1
 -1   -1   25   16   -1   -1   -1   -1   -1
 -1   -1   26   -1   -1   -1   -1   -1   -1
 -1   -1   29   -1   -1   -1   -1   -1   -1
 -1   -1   30   -1   -1   -1   -1   -1   -1
 -1   -1   31   -1   -1   -1   -1   -1   -1
 -1   -1   32   -1   -1   -1   -1   -1   -1
 -1   -1   33   -1   -1   -1   -1   -1   -1
 -1   -1   34   -1   -1   -1   -1   -1   -1
 -1   -1   35   -1   -1   -1   -1   -1   -1
 -1   -1   36   -1   -1   -1   -1   -1   -1
 -1   -1   37   -1   -1   -1   -1   -1   -1
 -1   -1   38   -1   -1   -1   -1   -1   -1
Total OH by Hu's Algorithm is 52.
 28   27   17    8    7    6    5    1    0
 28   -1   18    9   -1   -1   -1    2   -1
 28   27   19   10   -1   -1   -1    3   -1
 28   27   20   11   -1   -1   -1    4   -1
 28   27   21   12   -1   -1   -1   -1   -1
 28   -1   22   13   -1   -1   -1   -1   -1
 28   -1   23   14   -1   -1   -1   -1   -1
 -1   -1   24   15   -1   -1   -1   -1   -1
 28   -1   25   16   -1   -1   -1   -1   -1
 28   27   26   -1   -1   -1   -1   -1   -1
 -1   -1   29   -1   -1   -1   -1   -1   -1
 -1   -1   30   -1   -1   -1   -1   -1   -1
 -1   -1   31   -1   -1   -1   -1   -1   -1
 -1   -1   32   -1   -1   -1   -1   -1   -1
 -1   -1   33   -1   -1   -1   -1   -1   -1
 -1   -1   34   -1   -1   -1   -1   -1   -1
 -1   -1   35   -1   -1   -1   -1   -1   -1
 -1   -1   36   -1   -1   -1   -1   -1   -1
 -1   -1   37   -1   -1   -1   -1   -1   -1
 -1   -1   38   -1   -1   -1   -1   -1   -1
Total OH by Proposed Algorithm is 39.

  9    2    1    0
 -1    3   -1   -1
 -1    4   -1   -1
 -1    5   -1   -1
 -1    6   -1   -1
 -1    7   -1   -1
 -1    8   -1   -1
Total OH by Hu's Algorithm is 12.
  9    2    1    0
  9    3   -1   -1
  9    4   -1   -1
  9    5   -1   -1
  9    6   -1   -1
  9    7   -1   -1
  9    8   -1   -1
Total OH by Proposed Algorithm is 6.

 27   17   16   15    8    2    1    0
 28   18   -1   -1    9    3   -1   -1
 29   19   -1   -1   10    4   -1   -1
```

```
 30   20   -1   -1   11    5   -1   -1
 31   21   -1   -1   12    6   -1   -1
 32   22   -1   -1   13    7   -1   -1
 33   23   -1   -1   14   -1   -1   -1
 34   24   -1   -1   -1   -1   -1   -1
 35   25   -1   -1   -1   -1   -1   -1
 36   26   -1   -1   -1   -1   -1   -1
Total OH by Hu's Algorithm is 45.
 27   17   16   15    8    2    1    0
 28   18   -1   -1    9    3   -1   -1
 29   19   -1   15   10    4   -1   -1
 30   20   -1   -1   11    5   -1   -1
 31   21   -1   15   12    6   -1   -1
 32   22   -1   15   13    7   -1   -1
 33   23   -1   -1   14   -1   -1   -1
 34   24   -1   -1   -1   -1   -1   -1
 35   25   -1   -1   -1   -1   -1   -1
 36   26   -1   -1   -1   -1   -1   -1
Total OH by Proposed Algorithm is 42.

  4    3    1    0
  5   -1    2   -1
  6   -1   -1   -1
  7   -1   -1   -1
  8   -1   -1   -1
  9   -1   -1   -1
Total OH by Hu's Algorithm is 7.
  4    3    1    0
  5   -1    2   -1
  6   -1   -1   -1
  7   -1   -1   -1
  8   -1   -1   -1
  9   -1   -1   -1
Total OH by Proposed Algorithm is 7.

 24   23   22   14   11    1    0
 -1   -1   25   15   12    2   -1
 -1   -1   26   16   13    3   -1
 -1   -1   27   17   -1    4   -1
 -1   -1   28   18   -1    5   -1
 -1   -1   29   19   -1    6   -1
 -1   -1   30   20   -1    7   -1
 -1   -1   -1   21   -1    8   -1
 -1   -1   -1   -1   -1    9   -1
 -1   -1   -1   -1   -1   10   -1
Total OH by Hu's Algorithm is 57.
 24   23   22   14   11    1    0
 24   -1   25   15   12    2   -1
 24   23   26   16   13    3   -1
 24   23   27   17   -1    4   -1
 24   23   28   18   -1    5   -1
 24   23   29   19   -1    6   -1
 24   23   30   20   -1    7   -1
 24   23   -1   21   -1    8   -1
 24   23   -1   -1   -1    9   -1
 24   23   -1   -1   -1   10   -1
Total OH by Proposed Algorithm is 36.
```

2) Schedules by Hu's Algorithm and the Proposed Algorithm using TGG as input.

```
 0   1   2   4   6   7   8   9
-1  -1   3   5  -1  -1  -1  -1
Total OH by Hu's Algorithm is 10.
0   1   2   4   6   7   8   9
0   1   3   5  -1  -1  -1  -1
Total OH by Proposed Algorithm is 7.

  0   1   2   4   5   6   7   8   9
 -1  -1   3  -1  -1  -1  -1  -1  -1
Total OH by Hu's Algorithm is 5.
  0   1   2   4   5   6   7   8   9
  0   1   3  -1  -1  -1  -1  -1  -1
Total OH by Proposed Algorithm is 3.

  0   1   2   5   6   7   8   9
 -1  -1   3  -1  -1  -1  -1  -1
 -1  -1   4  -1  -1  -1  -1  -1
Total OH by Hu's Algorithm is 8.
  0   1   2   5   6   7   8   9
  0   1   3  -1  -1  -1  -1  -1
  0  -1   4  -1  -1  -1  -1  -1
Total OH by Proposed Algorithm is 5.

  0   1   2   3   5   6   4   8   9
 -1  -1  -1  -1  -1  -1   7  -1  -1
Total OH by Hu's Algorithm is 8.
  0   1   2   3   5   6   4   8   9
  0   1  -1  -1  -1  -1   7  -1  -1
Total OH by Proposed Algorithm is 6.

  0   1   2   3   5   7   8   9
 -1  -1  -1   4   6  -1  -1  -1
Total OH by Hu's Algorithm is 9.
  0   1   2   3   5   7   8   9
  0  -1  -1   4   6  -1  -1  -1
Total OH by Proposed Algorithm is 7.

  0   1   3   4   5   6   8   9
 -1   2  -1  -1  -1   7  -1  -1
Total OH by Hu's Algorithm is 10.
  0   1   3   4   5   6   8   9
  0   2  -1  -1  -1   7  -1  -1
Total OH by Proposed Algorithm is 8.

  0   1   3   4   5   7   8   9
 -1   2  -1  -1   6  -1  -1  -1
Total OH by Hu's Algorithm is 10.
  0   1   3   4   5   7   8   9
  0   2  -1  -1   6  -1  -1  -1
Total OH by Proposed Algorithm is 8.

  0   1   2   4   7   8   9
 -1  -1   3   6  -1  -1  -1
 -1  -1   5  -1  -1  -1  -1
```

```
Total OH by Hu's Algorithm is 13.
  0   1   2   4   7   8   9
  0   1   3   6  -1  -1  -1
  0  -1   5  -1  -1  -1  -1
Total OH by Proposed Algorithm is 9.

  0   1   4   5   2   7   8   9
 -1   3  -1  -1   6  -1  -1  -1
Total OH by Hu's Algorithm is 10.
  0   1   4   5   2   7   8   9
  0   3  -1  -1   6  -1  -1  -1
Total OH by Proposed Algorithm is 8.

  0   1   2   3   4   5   6   7   8   9  11  10  13  14  15  16  18  19
 -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  12  -1  -1  -1  17  -1  -1
Total OH by Hu's Algorithm is 23.
  0   1   2   3   4   5   6   7   8   9  11  10  13  14  15  16  18  19
  0   1  -1  -1  -1  -1  -1  -1  -1  -1  -1  12  -1  -1  -1  17  -1  -1
Total OH by Proposed Algorithm is 20.

  0   1   2   3   5   7   9  10  12  13  14  15  16  17  18  19
 -1  -1  -1   4   6   8  -1  11  -1  -1  -1  -1  -1  -1  -1  -1
Total OH by Hu's Algorithm is 32.
  0   1   2   3   5   7   9  10  12  13  14  15  16  17  18  19
  0   1  -1   4   6   8  -1  11  -1  -1  -1  -1  -1  -1  -1  -1
Total OH by Proposed Algorithm is 27.

  0   1   2   4   5   7   9  13  12  15  16  17  18  19
 -1  -1   3  -1   6   8  10  -1  14  -1  -1  -1  -1  -1
 -1  -1  -1  -1  -1  -1  11  -1  -1  -1  -1  -1  -1  -1
Total OH by Hu's Algorithm is 51.
  0   1   2   4   5   7   9  13  12  15  16  17  18  19
  0   1   3   4   6   8  10  -1  14  -1  -1  -1  -1  -1
  0  -1  -1  -1  -1  -1  11  -1  -1  -1  -1  -1  -1  -1
Total OH by Proposed Algorithm is 39.

  0   1   3   4   6   8  10  11  12  13  14  15  16  17  18  19
 -1   2  -1   5   7   9  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
Total OH by Hu's Algorithm is 45.
  0   1   3   4   6   8  10  11  12  13  14  15  16  17  18  19
  0   2  -1   5   7   9  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
Total OH by Proposed Algorithm is 41.

  0   1   4   2   6   8   9  11  12  13  15  16  17  18  19
 -1  -1  -1   3   7  -1  10  -1  -1  14  -1  -1  -1  -1  -1
 -1  -1  -1   5  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
Total OH by Hu's Algorithm is 47.
  0   1   4   2   6   8   9  11  12  13  15  16  17  18  19
  0   1  -1   3   7  -1  10  -1  -1  14  -1  -1  -1  -1  -1
  0   1  -1   5  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
Total OH by Proposed Algorithm is 38.

  0   1   3   2   5   9   6   8  13  15  16  17  18  19
 -1  -1  -1   4   7  -1  10  12  14  -1  -1  -1  -1  -1
 -1  -1  -1  -1  -1  -1  11  -1  -1  -1  -1  -1  -1  -1
Total OH by Hu's Algorithm is 41.
  0   1   3   2   5   9   6   8  13  15  16  17  18  19
```

```
  0  -1   3   4   7  -1  10  12  14  -1  -1  -1  -1  -1
  0  -1  -1  -1  -1  -1  11  -1  -1  -1  -1  -1  -1  -1
Total OH by Proposed Algorithm is 32.

  0   1   2   4   6   8   9  10  11  12  13  15  16  17  18  19
 -1  -1   3   5   7  -1  -1  -1  -1  -1  14  -1  -1  -1  -1  -1
Total OH by Hu's Algorithm is 40.
  0   1   2   4   6   8   9  10  11  12  13  15  16  17  18  19
  0   1   3   5   7  -1  -1  -1  -1  -1  14  -1  -1  -1  -1  -1
Total OH by Proposed Algorithm is 33.

  0   1   2   4   6   7   8  11  12  14  15  16  17  18  19
 -1  -1   3   5  -1   9  10  -1  13  -1  -1  -1  -1  -1  -1
Total OH by Hu's Algorithm is 46.
  0   1   2   4   6   7   8  11  12  14  15  16  17  18  19
  0   1   3   5  -1   9  10  -1  13  -1  -1  -1  -1  -1  -1
Total OH by Proposed Algorithm is 38.

  0   1   2   4   3   6   8   9  13  15  16  17  18  19
 -1  -1  -1  -1   5   7  -1  10  14  -1  -1  -1  -1  -1
 -1  -1  -1  -1  -1  -1  -1  11  -1  -1  -1  -1  -1  -1
 -1  -1  -1  -1  -1  -1  -1  12  -1  -1  -1  -1  -1  -1
Total OH by Hu's Algorithm is 49.
  0   1   2   4   3   6   8   9  13  15  16  17  18  19
  0   1  -1   4   5   7  -1  10  14  -1  -1  -1  -1  -1
  0   1  -1  -1  -1  -1  -1  11  -1  -1  -1  -1  -1  -1
  0  -1  -1  -1  -1  -1  -1  12  -1  -1  -1  -1  -1  -1
Total OH by Proposed Algorithm is 37.

  0   1   2   3   5   4   7   8  10  11  12  14  15  16  17  18  19
 -1  -1  -1  -1  -1   6  -1   9  -1  -1  13  -1  -1  -1  -1  -1  -1
Total OH by Hu's Algorithm is 33.
  0   1   2   3   5   4   7   8  10  11  12  14  15  16  17  18  19
  0   1   2   3  -1   6  -1   9  -1  -1  13  -1  -1  -1  -1  -1  -1
Total OH by Proposed Algorithm is 23.

  0   1   2   5   8   9  11  13  12  15  17  20  21  22  24  25  26  28
29
 -1   3   4   6  -1  10  -1  -1  14  16  18  -1  -1  23  -1  -1  27  -1
-1
 -1  -1  -1   7  -1  -1  -1  -1  -1  -1  19  -1  -1  -1  -1  -1  -1  -1
-1
Total OH by Hu's Algorithm is 128.
  0   1   2   5   8   9  11  13  12  15  17  20  21  22  24  25  26  28
29
  0   3   4   6  -1  10  -1  -1  14  16  18  -1  -1  23  -1  -1  27  -1
-1
  0   1  -1   7  -1  -1  -1  -1  -1  -1  19  -1  -1  -1  -1  -1  -1  -1
-1
Total OH by Proposed Algorithm is 115.
```

VITA

Jun Su

Candidate for the Degree of

Master of Science

Thesis: REDUCING INTER-PROCESSOR COMMUNICATION OVERHEAD BY TASK REPLICATION

Major Field: Computer Science

Biographical:

Personal Data: Born in Shenyang, China, September 22, 1971, son of Yongliang Su and Yuzhuo Pan.

Education: Received Bachelor of Science degree from Shenyang Agricultural University, Shenyang, China in July 1992. Completed the requirements for the Master of Science degree in Computer Science at the Computer Science Department of Oklahoma State University in December 2002.