AN EFFICIENT TASKS SCHEDULING ALGORITHM FOR DISTRIBUTED MEMORY MACHINES WITH COMMUNICATION DELAYS

***FATMA A. OMARA AND AMIN ALLAM**

**Computer Science Dept., Faculty of Computers & Information, Cairo University, Cairo, Egypt.*

e-mail: Fatma_omara@hotmail.com

Abstract: The scheduling of multiple interacting tasks of a single parallel program is considered the most important issue to exploit the true performance of the multiprocessor system. The problem is to find a schedule that will minimize the execution time (Make_Span) of a program. On the other hand, task scheduling on a multiprocessor system with and without communication delays is known to be NP-complete problem. Consequently, many heuristic algorithms have been developed, each of which may find optimal scheduling under different circumstances. One of the well known iterative algorithms is the hill-climbing. This algorithm starts with a complete solution and searches to improve this solution by choosing a better neighbor based on a cost function. This will lead to a local optimum which is considered the main drawback of this algorithm. The research in this study concerns to develop an efficient iterative algorithm for scheduling problem based on the hill-climbing. The developed algorithm satisfies a local optimum that is very close to the global one in a reasonable amount of time. In most experiments, it satisfies the actual global optimum.

Keywords: *Multiprocessors, scheduling, directed acyclic graph, communication delay*

1. INTRODUCTION

Parallel processing is a promising approach to meet the computational requirements of a large number of current and emerging applications that would be either inefficient or impractical executed using sequential processing like weather modeling, fluid flow, image processing, real-time and distributed database systems [1, 2]. However, it poses a number of problems that are not encountered in sequential processing such as designing a parallel algorithm for the application, partitioning the application into tasks, coordinating communication and synchronization, and scheduling the tasks onto the machine [3]. Most of these problems have been reported in the literature [1, 4]. On the other hand, scheduling and allocation of multiple interacting tasks of a single parallel program is a highly important issue since an inappropriate scheduling of tasks can fail to exploit the true performance of the system and can offset the grain from parallelization [3].

An abstract model of partitioning a parallel program can be represented by a *Directed Acyclic Graph* (DAG), in which the node weights represent processing times and the edge

weights represent data dependencies, as well as, the communication times between tasks. The objective of task scheduling is to find an assignment of tasks to the available processors and an execution order such that parallel program execution time is minimized [5]. Generally, the scheduling problem exists in two types: static and dynamic. According to the static scheduling, the characteristics of a parallel program such as task processing times, communication, data dependencies, and synchronization requirement are known before program execution [3]. According to the dynamic scheduling, a few assumptions about the parallel program can be made before execution, then, scheduling decisions have to be made on-the-fly [6]. The study in this research concerns only the static task scheduling. On the other hand, task scheduling on a multiprocessor system with and without communication delays is known to be NP-complete problem except for some special cases [5][3] [7]. Consequently, heuristic methods have to be used. Although they may find only approximations of the optimum, but they will do it in a reasonable amount of time [8]. Heuristic algorithms may be divided in two main classes. First, the general purpose optimization algorithms independent of the given optimization problem and, on the other hand, the heuristic approaches especially designed for each specific scheduling problem [7]. The common categorized heuristic algorithms of the second class are priority-based [9], cluster-based [4], and task duplication-based algorithms [10]. Although these algorithms are considered greedy algorithms, where the optimum task scheduling of a parallel program is progressively constructed and allocating of tasks into the available processors is done without back tracking(i. e., greedy algorithms), they are easy to implement and have a polynomial complexity [11]. According to priority-based algorithms, a priority is assigned to each task and then tasks are assigned to different processors one by one according to that priority. The schedulers based on priority are classified according to the particular heuristic used to assign priorities to the tasks and to select the "best" processor to run the task [9]. The task priority can be defined in several ways; the length of the ***C**ritical **P**ath*[1] (CP) [12], the amount of required communications [13], the number of successors, the task execution time, the task mobility etc. [14, 15] [16]. According to the cluster-based schemes, the tasks in the DAG are divided into a set of clusters, each formed on the basis of interdependence of the task and then allocate each cluster to a processor so as to achieve a minimal communication overhead [4],. The task duplication-based Scheduling algorithms based on duplicating some tasks such that the wait state can be reduced while data is being transferred between processors of the system to satisfy precedence constraints [2, 10, 17-19]. A recent survey of various scheduling algorithms of the second class and their functionalities was found in [3]. The main drawback of the second class heuristic algorithms is that their limited applicability [8].

The first class heuristic algorithms are iterative algorithms where they depart from an initial solution and try to improve it [20]. The initial solution in iterative algorithms is found using either ***L**argest **P**rocessing **T**ime* (LPT) [21] or the length of the ***C**ritical **P**ath* (CP) [12] and then the tasks are exchanged between processors in the system to improve locally a solution. A well known iterative algorithm is called hill-climbing was proposed by Bokhari [22]. The hill-climbing algorithm starts with a complete solution and searches to improve this solution by choosing a better neighbor. The quality of a solution using hill-climbing algorithm is defined by a cost function. This solution leads directly to a local

[1] CP is defined as the largest sum of execution times till an end task in DAG.

weights represent data dependencies, as well as, the communication times between tasks. The objective of task scheduling is to find an assignment of tasks to the available processors and an execution order such that parallel program execution time is minimized [5]. Generally, the scheduling problem exists in two types: static and dynamic. According to the static scheduling, the characteristics of a parallel program such as task processing times, communication, data dependencies, and synchronization requirement are known before program execution [3]. According to the dynamic scheduling, a few assumptions about the parallel program can be made before execution, then, scheduling decisions have to be made on-the-fly [6]. The study in this research concerns only the static task scheduling. On the other hand, task scheduling on a multiprocessor system with and without communication delays is known to be NP-complete problem except for some special cases [5][3] [7]. Consequently, heuristic methods have to be used. Although they may find only approximations of the optimum, but they will do it in a reasonable amount of time [8]. Heuristic algorithms may be divided in two main classes. First, the general purpose optimization algorithms independent of the given optimization problem and, on the other hand, the heuristic approaches especially designed for each specific scheduling problem [7]. The common categorized heuristic algorithms of the second class are priority-based [9], cluster-based [4], and task duplication-based algorithms [10]. Although these algorithms are considered greedy algorithms, where the optimum task scheduling of a parallel program is progressively constructed and allocating of tasks into the available processors is done without back tracking(i. e., greedy algorithms), they are easy to implement and have a polynomial complexity [11]. According to priority-based algorithms, a priority is assigned to each task and then tasks are assigned to different processors one by one according to that priority. The schedulers based on priority are classified according to the particular heuristic used to assign priorities to the tasks and to select the "best" processor to run the task [9]. The task priority can be defined in several ways; the length of the ***C**ritical **P**ath*[1] (CP) [12], the amount of required communications [13], the number of successors, the task execution time, the task mobility etc. [14, 15] [16]. According to the cluster-based schemes, the tasks in the DAG are divided into a set of clusters, each formed on the basis of interdependence of the task and then allocate each cluster to a processor so as to achieve a minimal communication overhead [4],. The task duplication-based Scheduling algorithms based on duplicating some tasks such that the wait state can be reduced while data is being transferred between processors of the system to satisfy precedence constraints [2, 10, 17-19]. A recent survey of various scheduling algorithms of the second class and their functionalities was found in [3]. The main drawback of the second class heuristic algorithms is that their limited applicability [8].

The first class heuristic algorithms are iterative algorithms where they depart from an initial solution and try to improve it [20]. The initial solution in iterative algorithms is found using either ***L**argest **P**rocessing **T**ime* (LPT) [21] or the length of the ***C**ritical **P**ath* (CP) [12] and then the tasks are exchanged between processors in the system to improve locally a solution. A well known iterative algorithm is called hill-climbing was proposed by Bokhari [22]. The hill-climbing algorithm starts with a complete solution and searches to improve this solution by choosing a better neighbor. The quality of a solution using hill-climbing algorithm is defined by a cost function. This solution leads directly to a local

[1] CP is defined as the largest sum of execution times till an end task in DAG.

optimum, which is considered the main drawback of this algorithm. The present research concerns to develop an efficient iterative algorithm based on hill-climbing for task scheduling on multiprocessor.

Recently, *Genetic Algorithms* (GAs) are introduced by Holland [23]. They have been applied to combinatorial optimization problems in various fields including scheduling [24] [25, 26]. GAs are considered global search techniques to explore different regions of the search space simultaneously by keeping track of a set of potential solutions of diverse characteristics, called a population. Therefore, GAs are widely recognized as effective techniques in solving numerous optimization problems, because they can potentially locate better solutions at the expense of longer running time. Another merit of a genetic search is that its inherent parallelism can be exploited so as to further reduce its running time. Recently, a parallel genetic algorithm for scheduling has been proposed [15].

2. MULTIPROCESSOR TASK SCHEDULING PROBLEM

The model of multiprocessor systems that is considered in this study was described as follows [27]; the time required for executing a unit of task on a processor is assumed a unit, and the time required for transmitting a unit of data from one processor to another is also assumed to be a unit. A communication from/to a processor P is overlapped with the computation on P, and simultaneously, with the other communication from/to P. Assume $P = \{P1, P2, P3... Pm\}$ denotes the set of m processors, and $V = \{v1, v2, v3... vn\}$ denotes a set of n vertices representing a set of tasks. The precedence constraint among tasks in V can be represented in the form of a Directed Acyclic task Graph (DAG) $G = (V, E)$, where each directed edge $(v, w) \in E$ intuitively implies that the execution of w needs the outcome of the execution of v, i.e. *v is a predecessor of w,* and *w is a successor of v.* Each vertex $v \in V$ is given an integral cost $\tau(v)$ representing the processing time of task v on a processor, and each edge $(v, w) \in E$ is given an integral cost $c(v, w)$ representing the size of data to be transmitted from task v to task w; i.e., the communication from v to w takes 0 step if those vertices are assigned to the same processor, and it takes $c(v, w)$ steps if they are assigned to different processors. A vertex with no predecessor is called an entry vertex, and a vertex with no successor is called an exit vertex. Most of research works assumed that G contains exactly one entry vertex V_s and exactly one exit vertex V_t without loss of generality. According to the present study, this consideration is not used where DAG with arbitrary structure can be used.

A schedule S of G onto P is a relation $\Re \subseteq V \times P \times (N^+ \cup \{0\})$, where N^+ is the set of natural numbers that is used to represent the start time of the tasks; i.e., $(u, p, t) \in \Re$ implies that p is a processor to which task u is assigned and t is the time at which the execution of u on p starts. If we restrict $\Re$ to be a function from V to $P \times (N^+ \cup \{0\})$ then we say that the model does not allow duplication of tasks; otherwise, we say that it allows duplication of tasks. A feasible schedule is a schedule satisfying the following two conditions:

1. For any $v, w \in V$, if $(v, p, t_v) \in \Re$ and $(w, p, t_w) \in \Re$, then $t_v + \tau(v) \leq t_w$ or $t_w + \tau(w) \leq t_v$; i.e., the execution of two tasks assigned to the same processor must not be overlapped.
2. For any $(v, w) \in E$, if $(v, p_v, t_v) \in \Re$ and $(w, p_w, t_w) \in \Re$ then $t_w \geq t_v + \tau(v) + \delta(v, w)$, where $\delta(v, w) = c(v, w)$ if $p_v \neq p_w$ and $\delta(v, w) = 0$, otherwise; i.e., the assignment must satisfy the precedence constraint.

The Make_Span of schedule S is defined as $\max_{v \in V}\{tv + \tau(v) : (v, p_v, t_v) \in \Re\}$. Therefore, the multiprocessor scheduling problem is defined as how to find a feasible schedule with the minimum Make_Span.

3. OUR ENHANCED HILL-CLIMBING ALGORITHM

A well known hill-climbing algorithm for static scheduling was proposed by Bokhari [22]. This algorithm starts with a complete solution and search to improve this solution by choosing a better neighbor [22]. It consists of improving a current solution by local transformations. If the quality of the new solution is better (according to a predefined objective function) than the current one, the new solution is kept and it becomes the current solution. Otherwise, the current solution is not altered. This process is repeated until the quality of the solution is not improved for a predefined number of iterations. The quality of a solution using hill-climbing algorithm is defined by a predefined function. The main drawback of the hill-climbing algorithm is that it sticks with a local optimum rather than a global one [8, 11]. On the other hand, the hill-climbing approach improves a solution very fast unless it reaches a local optimum, i.e., it is considered a good search technique for convex spaces. According to this study, another hill-climbing algorithm has been developed.

Our developed algorithm is based on hill-climbing with some modifications that have been added to enhance the chance of moving from a local optimum to the global one. These modifications overcome the hill-climbing main drawback, and in the same time, keep its advantages. Also, our algorithm is based on modifying the objective function such that the comparison between two similar solutions determines which solution has better quality depending on internal characteristics of these two solutions even if they may be considered the same from the point of view of the basic objective function. Refinement the objective function and a similar issue called monitoring are discussed in [11]. Our idea which is used to move the solution from a local optimum is that carefully examining the solution characteristics that highly cause sticking to a local optimum, and concentrate on doing local transformations that have high probability to improve these characteristics. The details of our algorithm are discussed in the following sections.

3.1 The Solution Encoding and the Corresponding Schedule

A valid solution is encoded in two parts [28]. These parts are:

Sp1[1 2 ..... n], and Sp2[1 2 .....n]

where Sp1[i], $1 \leq i \leq n$ is the task ID which has order i in the scheduling. Tasks IDs are numbers from 1 to n. Similarly, Sp2[i], $1 \leq i \leq n$ is the processor ID to which the task with ID = i will be allocated on it. Processors IDs are numbers from 1 to m.

Most previous works chose an encoding that consists only of Sp1 part of the solution, which is a permutation of the tasks that obeys the precedence constraints, and followed the rules of allocating a task to a processor that allows the **E**arliest **S**tart (ES) execution time of this task (i.e., ES approach). The drawback of ES approach is that it does not guarantee the optimum task allocation according to the given task order. Therefore, the optimum solution may remain hidden and unreachable in many cases [29]. Our algorithm, however, avoids this drawback so as not to lose the hope of obtaining the optimum solution by using Sp1, and Sp2.

The following pseudo code describes how to construct a schedule from a specific valid solution encoding:

```
for i = 1 to n
    begin
        Allocate task Sp1[i] to processor Sp2[Sp1[i]] such that it is started as
    early as possible while preserving all the precedence constraints
end
```

3.2 The Initial Solution Encoding

The initial solution may be constructed using a greedy algorithm. However, experiments show that the overheads of applying a greedy algorithm are usually greater than the benefits of starting with a good initial solution. According to our algorithm, a random valid initial solution is chosen as follows:

```
for i = 1 to n
  begin
    Sp2[i] = random number from 1 to m
  end
for i = 1 to n
  begin
      Sp1[i] = a task ID with the property that all tasks which directly precede it
    in DAG have IDs that exists in Sp1[1, ..., (i-1)]
  end
```

3.3 The Objective Function (The Solution Fitness)

Although the main objective function of the scheduling algorithm is to minimize the schedule Make_Span. The problem with this assumption is that several solutions may have the same schedule Make_Span, but one of them (the hidden one) may be the best one in the aspect of being easy modifiable to a new solution that has less schedule Make_Span.

Therefore, our developed algorithm is based on how to discover the hidden and unreachable optimum solution by finding the objective function using two phases; The

Ordinary Phase and the Local_Optimum_Skipping Phase. The function of the Ordinary Phase is to define the best solution according to a basic solution criteria (will be defined later). The algorithm starts with the Ordinary Phase and remains there until the local optimum is reached; i.e., no improvement in the basic solution criteria is encountered for a specific number of iterations. In this case, the algorithm starts the Local_Optimum_Skipping Phase to improve the basic solution criteria. The main function of the Local_Optimum_Skipping Phase is how to select an optimum solution if there are two competing solutions have been encountered according to the basic solution criteria.

Our algorithm operates as follows:

Generate Initial Solution S_0
Current_Phase := Ordinary, S := S_0, Idle_Count := 0
Repeat
Compute a neighboring solution S' by local transformation
if (Compare_Basic_Criteria (S, S') = S') do
S := S', Idle_Count := 0, Current_Phase := Ordinary
else if (Compare_Basic_Criteria (S, S') = S) do
Idle_Count ++
else if (Compare_Basic_Criteria (S, S') = equal) do
Idle_Count ++
if (Compare_Hidden_Criteria (S, S', Current_Phase) ≠ S) do
S := S'
end if
end if
if (Current_Phase = Ordinary and Idle_Count > MAX_IDLE_PHASE1) do
Current Phase := Local_Optimum_Skipping
end if
Until Stopping criteria(Max number of iterations or time limit)

The following algorithm defines the function of Compare_Hidden_Criteria (S_1, S_2, Current_Phase):

Compare_Hidden_Criteria (S_1, S_2, Current_Phase)
if (Current_Phase = Ordinary) then
return Compare_Hidden_Criteria_Ordinary (S_1, S_2)
else
return Compare_Hidden_Criteria_Local_Optimum_Skipping(S1, S_2)

3.4 The Basic Solution Criteria

The following algorithm explains how to compare between two solutions based on the basic criteria in the Ordinary Phase:

Compare_Basic_Criteria (S_1, S_2)
if Schedule Make_Span of S_1 ≠ Schedule Make_Span of S_2
return solution with the smaller value
if Number of tasks that finish at the schedule Make_Span end time is different for S_1 and S_2

return solution with the smaller value
return "equal"

Figure 1 illustrates the principle of the comparison according to condition (1) and condition (2).

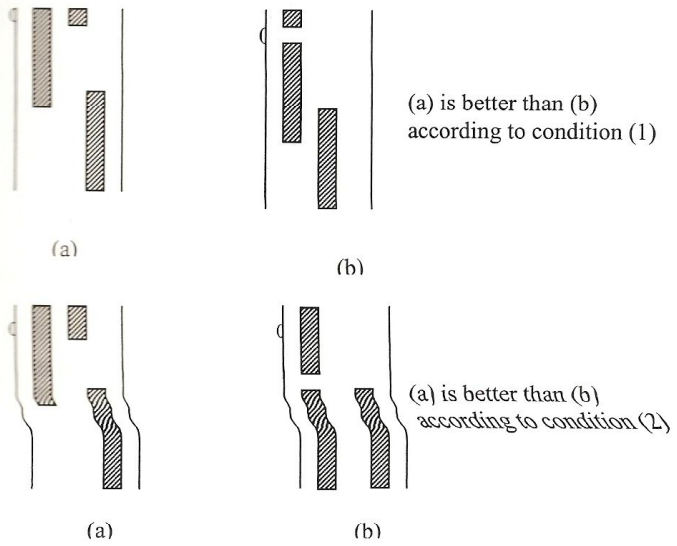

Fig. 1: Basic Criteria in the Ordinary Phase.

3.5 The Hidden Solution Criteria in the Ordinary Phase

The following algorithm explains how to compare the hidden criteria of two solutions that are equal in the basic criteria in the Ordinary Phase.

Compare_Hidden_Criteria_Ordinary (S_1, S_2)
if Processor Idle Time of $S_1 \neq$ Processor Idle Time of S_2
return solution with the smaller value
if the sum of squared distance between each processor's end time and the whole schedule Make_Span is different for S_1 and S_2 (Load Balance Parameter)
return solution with the smaller value
return "equal"

Figure 2 illustrates the principle of comparing between two solutions S_1 and S_2 ordinary according to the hidden criteria in the Ordinary Phase.

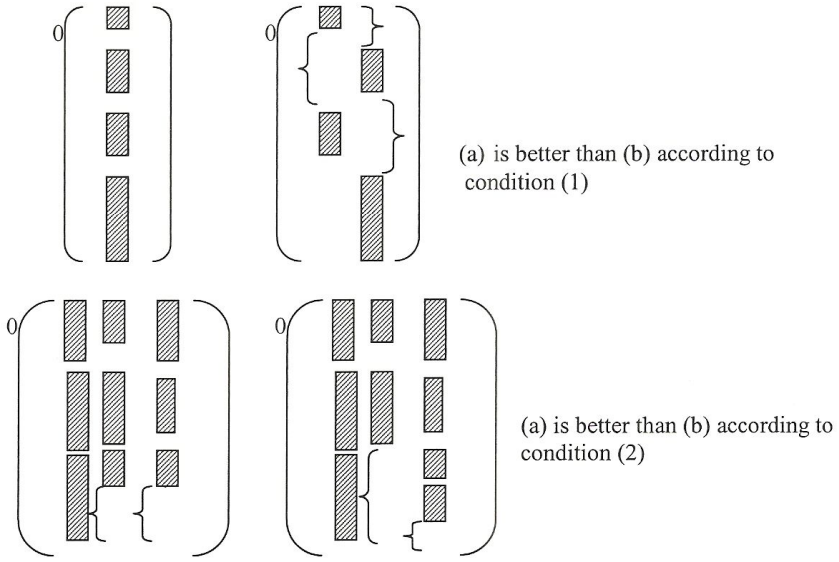

Fig. 2: The Hidden Solution Criteria in the Ordinary Phase.

3.6 The Hidden Solution Criteria in the Local_Optimum_Skipping Phase

Before explaining the algorithm of the Hidden Solution Criteria in The Local_Optimum_Skipping Phase, we need to state some definitions.

The Earliest Block Start Time; it is the earliest time of a task such that it can start under the current configuration. Assuming that this task will be located on the same processor and all the immediately preceding tasks of that task will remain at their places.

An Immediately Blocking task of a task X; it is a task that immediately precedes the task X such that if it is removed, the earliest block start time of task X will decrease, or the number of immediately blocking tasks of the task X will be decreased by 1. (Recursive definition)

The Critical Blocking Path; it consists of some tasks, and it is defined as follows:

Add a task which finishes its execution at the schedule Make_Span.
Add an immediately blocking task of the last added task.
Repeat step (2) until there is no immediately blocking task for the last added task.

The algorithm which is used to compare the Hidden Solution Criteria of two solutions that are equal in the basic criteria in the Local Optimum Skipping Phase is as follow:

3.7 Compare_Hidden_Criteria_Local_Optimum_Skipping (S_1, S_2)

for i := 1 to Max Critical path size
if (Critical path size of S_1 and Critical Path size of S_2 < i)
return "equal"
if (Critical path size of S_1 < i)
return S_1
if (Critical path size of S_2< i)
return S_2
if (Earliest Block start time of i^{th} task of critical path of S_1 differs from its equivalence of S_2)
return solution with the smaller value
if (the number of immediately blocking tasks of i^{th} task of critical path of S_1 differs from its equivalent of S_2)
return solution with the smaller value
end For
return "equal" otherwise

3.8 The Local Transformation

A local transformation is done by obtaining a neighbor of the current solution so as to compare it against the old solution. A local transformation should be fast enough because it is repeated many times. Therefore, it is perfect to make it at most O(n). Also, a local transformation should generate a solution that is very close to the old solution because a long jump will often generate no better solution. A local transformation will be done by making a change in the SP1 or the SP2 part of the encoded solution.

3.8.1. Local Transformation by Changing SP1 part

A change in the SP1 part can be done by using a 1-Or-Opt (1-Swap) neighbor [7]. This is done by moving a single task from one position to another while preserving precedence constraints. This can be done by the following procedure:

Move a task ID at location SP1[old] to a new location SP1[new] such that one of the following conditions must be satisfied before moving:

1) new > old, for all old < i ≤ new : (Sp1[old], i) ∉ E.

2) new < old , for all new ≤ i < old : (i, Sp1[old]) ∉ E.

3.8.2. Local Transformation by Changing SP2 part

A change in the SP2 part can be done by moving a given task on another processor (movement strategy) or by exchanging the processors of two tasks (exchange strategy) [8]. Because these approaches make a big difference (long jump) between the old solution and the new solution, we introduce a new strategy.

Our local transformation strategy appears to be more complicated than the previous two strategies. However, the resulting solution will have a very small meaningful difference than the old one. Our strategy is similar to the genetic crossover operator made by Hou [26] and it can be illustrated as follows:

Given a random order r (from 1→n) and random two processors (Pr_1, Pr_2)

for i := r to n do
current_task := SP1[i]
if (SP2[current_task] = Pr_1) then SP2 [current_task] := Pr_2
else if (SP2[current_task] = Pr_2) then SP2 [current_task] := Pr_1
end

3.8.3. Local Transformation Enhancements

The local transformation has been enhanced by some kinds of analyzing the schedule of the old solution. The idea of this enhancement is taken from the monitoring procedures in [11], and can be done as follows:

While constructing the schedule (3.1), two arrays are constructed; Idle[1,.., Max_Idle] and Block[1, .., Max_Block], where Max_Idle, and Max_Block are the number of entries in the two arrays. An entry Idle[i] where $1 \leq i \leq$ Max_Idle, contains information about an idle slot within a processor. This information includes the task ID which starts immediately after the idle slot of this processor (Idle[i].Pidle), and the task ID of the immediately blocking task Idle[i].Pblocking

An entry Block[i], where $1 \leq i \leq$ Max_Block contains information about a situation in which some task lies on the critical blocking path Block[i].Pblocked, and its earliest block start time is not equal to its current start time. The information of the entries in this array are used only if the algorithm is currently running in the local optimum skipping phase, that is , if the algorithm operates in the ordinary phase, Max_Block should equal 0.

The usage of these information in the local transformation is done for some probability as follow:

For the SP1 port:
Make one of the following changes:
Move the task Idle[i].Pidle to the right, where i is random 1 → Max_Idle.
Move the task Idle[i].Pblocking to the left, where i is random 1 → Max_Idle.
Move the task Block[i].Pblocked to the left, where i is random 1→ Max_Block.
For the SP2 port:

Make the same changes as we did before in (3.4.2) with the exception that r, Pr_1, Pr_2 are not random and are chosen such that one of the following conditions is satisfied:

(1) SP1[r] = Idle[i].Pidle
Pr_1 = Sp2[Idle[i].Pidle], Pr_2 = SP2[Idle[i].Pblocking].
(2) SP1[r] = Block[i].Pblocked
Pr_1 = SP2[Block[i].Pblocked]

Pr_2 = random number from 1 →m.

4. EXPERIMENTAL RESULTS

We have implemented our algorithm on an AMD Athlon XP processor (1.7 GHz) using task graphs taken from a Standard Task Graph Set Archive [30]. This Standard Task Graph Set consists of task graphs generated randomly and modeled from actual application programs without communication delays (i.e., 0 communication delays). Also, we reapplied these task graphs using our algorithm with considering random communication costs are distributed uniformly between 0 and a specified maximum delay for each experiment.

Our experimental results are very close to that given in [30] with respect to the solution quality (the closeness to the optimum) and also the rate of solution improvement. In addition to, the experimental results for iterative techniques given in [30] are restricted to problems with number of tasks at most 500 [7, 8, 11, 28]. Although our algorithm has a computer memory overhead that depends linearly on the number of tasks, it provides high quality solutions even if the number of tasks increases hugely.

In general, according to our algorithm the execution time of each experiment increases if one of the following criteria increases:

1) Number of tasks (n).
2) Number of processors (m).
3) Number of communication edges (Ce).
4) Maximum communication delays (Cd).

Each of the following graphs (Fig. 3 to 7) illustrates how the solution improves with respect to the Make_Span for five task graphs. The most important observation is that our algorithm does not stick with a local optimum unless it is very close to the global one. For most experiments, it reaches the global optimum in a reasonable amount of time that depends on the previous parameters.

Therefore, a parallel implementation of our algorithm will give better Make_Span. The proposed pseudo code for such parallelism could be as follows:

1- Generate Initial Solution (S) in master processor,
2- Execute our algorithm on all processors (Including master processor) using S as the initial solution for a fixed number of iterations or fixed amount of time.
3- S : = best solution obtained from the solutions generated by all processors.
4- Go to step 2 (if stopping condition is not satisfied).

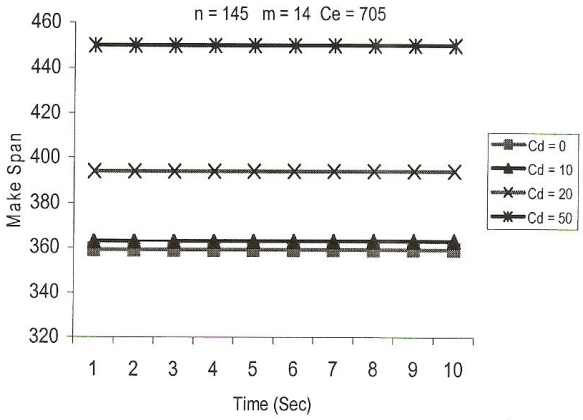

Fig. 3: Improving the Make_Span for the First Task Graph.

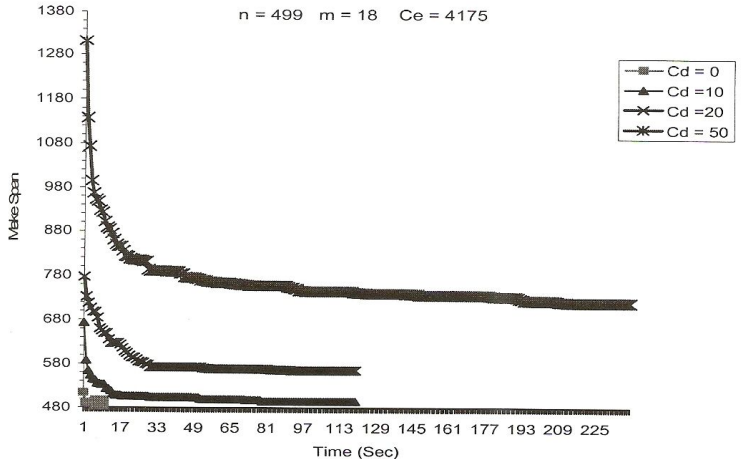

Fig. 4: Improving the Make_Span for the Second Task Graph.

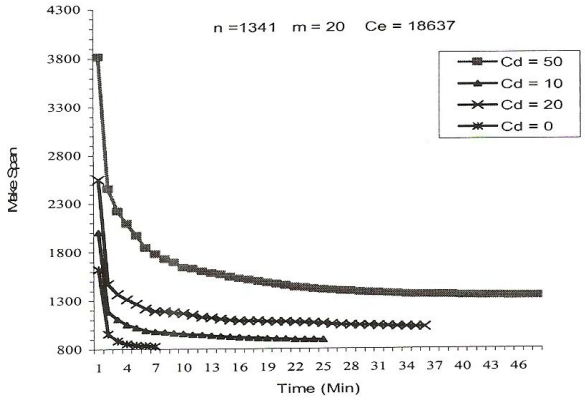

Fig. 5: Improving the Make_Span for the Third Task Graph.

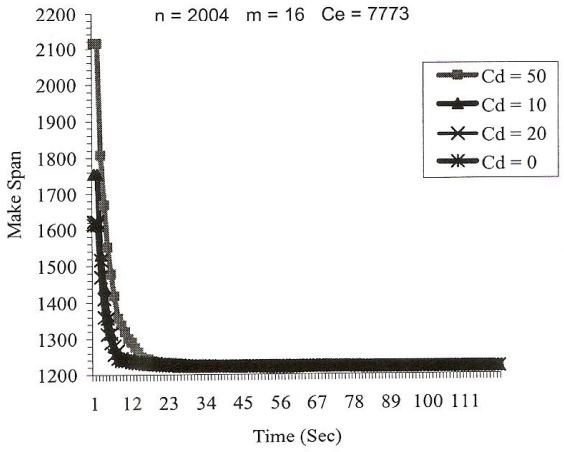

Fig. 6: Improving the Make_Span for the Fourth Task Graph.

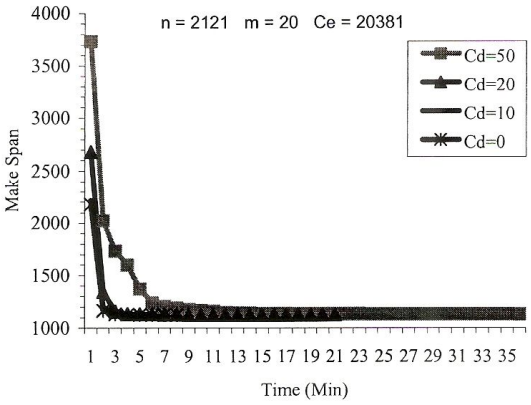

Fig. 7: Improving the Make_Span for the Fifth Task Graph.

5. . CONCLUDING REMARKS AND FUTURE WORK

The scheduling problem is to find a schedule that will minimize the Make_Span of a program. Because task scheduling on a multiprocessor system with and without communication delays is known to be NP-complete problem, many heuristics have been developed. One of the well known iterative algorithms is the hill-climbing. Unfortunately, this algorithm produces a local minimum rather than the required global one. According to this research study, an efficient iterative algorithm based on the hill-climbing has been developed to satisfy a local optimum that is very close to the global one in a reasonable amount of time.

Our algorithm is based on modifying the objective function such that the comparison between two similar solutions determines which solution has better quality depending on internal characteristics of these two solutions even if they may be considered the same from the point of view of the basic objective function.

We have implemented our algorithm using standard task graphs with considering random communication costs. The most important observation is that our algorithm does not stick with a local optimum unless it is very close to the global one. For most experiments, it reaches the global optimum in a reasonable amount of time. Also, a pseudo code for a parallel implementation of our algorithm is presented.

Our hill climbing algorithm can be improved by more meaningful ideas of local transformations. They can be concluded by carefully investigating the solution criteria. Also, the Local_Optimum_Skipping phase can be improved also by adding a helper technique such as tabu search or variable neighborhood search.

REFERENCES

[1] M. Quinn, "Parallel Computing: Theory and Practice", 2nd edition, McGraw-Hill, Inc., New York, NY, 1994.

[2] R. Bajaj, D. Agrawal, "Improving Scheduling of Tasks in a Heterogeneous Environment," IEEE Trans. On Parallel and Distributed Systems, Vol.15 (2), pp. 107-118, 2004.

[3] K. Yu-Kwong, I. Ahmad, "Static Scheduling Algorithms for Allocating Diracted Task Graphs to Multiprocessors," ACM Computing Survey, Vol. 31(4), pp. 406-471, 1999.

[4] A. Gerasoulis, T. Yang, "A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs onto Multiprocessors," Parallel and Distributed Computing Journal, Vol. 16(4), pp. 276-291, 1992 .

[5] A. Auyeung, I. Gondra, and K. Dai, "Multi-Heuristic List Scheduling Genetic Algorithm For Task Scheduling," ACM, Vol. 1(58), pp. 721-724, 2003.

[6] A. Palis, C. Liou, S. Rajasekaran, S. Shende, and L. Wei, "Online Scheduling of Dynamic Trees," Parallel Proc. Letter, Vol. 5(4), pp. 635-646, 1995.

[7] T. Daviovic, P. Hansen, and N. Mladenovic, "Variable Neighborhood Search for Multiprocessor Scheduling Problem With Communication Delays," 4th Meta heuristics International Conference, Porto, Portugal, pp. 737-741, 2001.

[8] E-G Talbi and T. Muntean, "General Heuristics for The Mapping Problem", Proc. of The World Transputer Conference, Germany, pp. 1229-1241, 1993.

[9] B. Chen, "A Note on lpt Scheduling", Operational Research Letter, Vol. 14, pp. 139-142, 1993.

[10] S. Darbha and P. Agrawal, "Optimal Scheduling Algorithm for Distributed Memory Machines," IEEE Trans. Parallel and Distributed Systems, Vol. 9(1), pp. 87-95, 1998.

[11] P. Bouvry, J. Chassin and D. Trystram, "Efficient Solutions for Mapping Parallel Programs," CWI-Center for Mathematics and Computer Science, Amsterdam, The Netherlands, published in Euro-Par, pp. 379-390, 1995.

[12] C. Sih and A. Lee,"A Compile Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processors Architectures," IEEE Trans. Parallel and Distributed Systems, Vol. 4(2), pp. 175-187, 1993.

[13] V. Krisshnamoorthy and K. Efe, "Task Scheduling With and Without Communication Delays: A Unified Approach," European Journal of Operational Research, Vol. 89, pp. 366-379, 1996.

[14] K. Kwok and I. Ahmad, "Dynamic Critical Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," IEEE Trans. on Parallel and Distributed Systems, Vol. 7(5), pp. 506-521, 1996.

[15] K. Kwok and I. Ahmad, "Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using a Parallel Genetic Algorithm," Parallel and Distributed Computing Journal, Vol. 47, pp. 58-77, 1997.

[16] K. Kwok and I. Ahmad,"Fastest: A Practical Low-Complexity Algorithm for Compile-Time Assignment of Parallel Programs to Multiprocessors," IEEE Trans. On Parallel and Distributed Systems, Vol. 10(2), pp. 147-159, 1999.

[17] I. Park and Y. Choe, "An Optimal Scheduling Algorithm Based on Task Duplication," IEEE Trans. Computers, Vol. 51(4), pp. 444 – 448, 2002.

[18] I. Ahmad and K. Yu-Kwong, "On Exploiting Task Duplication in Parallel Program Scheduling," IEEE Trans. Parallel and Distributed Systems, Vol. 9(9), pp. 872-892, 1998.

[19] K. Yu-Kwong and I. Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," Parallel and Distributed Computing Journal, Vol. 59 (2), pp. 381-422, 1999.

[20] P. Bouvry, J. Chassin and D. Trystram, "Efficient Solutions for Mapping Parallel Programs," Proc. of Euro-Par, Amsterdam, Netherlands, pp. 379-390, 1995.

[21] K. Friesen, "Tighter Bounds for LPT Scheduling on Uniform Processors," SIAM J. Computer, Vol. 16(3), pp. 554-560, 1987.

[22] H. Bokhari, "On Mapping Problem," IEEE Transaction on Computers, Vol. 30(3), pp. 207-214, 1981.

[23] J. Holland, "Adaptation in Natural and Artificial Systems," Ann Arbor. Univ. Of Michigan Press, 1975.

[24] T. Benten and M. Sait,"Genetic Scheduling of Task Graphs," International Electron Journal, Vol. 77(4), pp. 401-415, 1994.

[25] I. Ahmad and K. Dhodhi, "Multiprocessor Scheduling in a Genetic Paradigm," Parallel Computing, Vol. 22, PP. 395-406, 1996.

[26] H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm form Multiprocessor Scheduling," IEEE Transaction of Parallel and Distributed Systems, Vol. 5(2), pp. 113-120, 1994.

[27] S. Fujita and M. Yamashita, "Approximation Algorithms for Multiprocessor Scheduling," IEICE Transaction of Information and Systems, Vol. E83 (3), pp. 503-509, 2000.

[28] M. Rinehart, V. Kianzad, and S. Bhattacharyya, "A Modular Genetic Algorithm for Scheduling Task Graphs," Technical Report UMIACS-TR-2003-66, Institute for Advanced Computer Studies, University of Maryland at College Park. 2003.

[29] T. Davidovic, "Exhaustive List-Scheduling Heuristic for Dense Task Graphs," YUJOR, Vol. 10(1), 123-136, 2000.

[30] http://www.kasahara.elec.waseda.ac.jp/schedule/, 2005.M. J. Quinn, 1994, Parallel Computing: Theory and Practice, 2nd edition, McGraw-Hill, Inc., New York, NY.