An Efficient Tasks Scheduling Algorithm for Distributed Memory Machines with Communication Delays

Fatma A. Omara and Amin Allam
Department of Computer Science, Faculty of Computers and Information, Cairo University, Egypt

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. Present 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.

Key words: Multiprocessors, scheduling, directed acyclic graph, communication delay

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. 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. Most of these problems have been reported in the literature. 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 gain from parallelization.

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. 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. 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. 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. 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. 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. The common categorized heuristic algorithms of the second class are priority-based, cluster-based and task duplication-based algorithms. Although these algorithms are considered greedy.
The model of multiprocessor systems that is considered in this study was described as follows\textsuperscript{[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 $P = \{P_1, P_2, P_3, \ldots, P_m\}$ denotes the set of $m$ processors and $V = \{v_1, v_2, v_3, \ldots, v_n\}$ 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 $c(v)$ 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 $V_e$ and exactly one exit vertex $V_o$, 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 $R \subseteq V \times P \times (N^+ \cup \{0\})$, where, $N^+$ is the set of natural numbers that is used to represent the start time of the tasks; i.e., $(u, p, t) \in R$ 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 $R$ to be a function from $V$ to $P \times (N^+ \cup \{0\})$ then we say that the model does not allow duplication of tasks.

MULTIPROCESSOR TASK SCHEDULING PROBLEM

The first class heuristic algorithms are iterative algorithms where they depart from an initial solution and try to improve it\textsuperscript{[29]}. The initial solution in iterative algorithms is found using either Largest Processing Time (LPT)\textsuperscript{[30]} or the length of the Critical Path (CP)\textsuperscript{[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\textsuperscript{[29]}. 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 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\textsuperscript{[29]}. They have been applied to combinatorial optimization problems in various fields including scheduling\textsuperscript{[24-26]}. 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\textsuperscript{[18]}. 

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 \((v, p, t_v) \in \mathcal{R}\) and \((w, p, t_w) \in \mathcal{R}\), then \(t_v + t(v) \leq t_w\) or \(t_w + t(v) \leq t_v\); i.e., the execution of two tasks assigned to the same processor must not be overlapped.

2. For any \((v, w) \in E\), if \((v, p_v, t_v) \in \mathcal{R}\) and \((w, p_w, t_w) \in \mathcal{R}\) then \(t_v \geq t_w + t(v) + \delta(v, w)\), 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 \(
\text{max}_{v \in V} \{ t(v) + t(v) : (v, p, t_v) \in \mathcal{R} \} \). Therefore, the multiprocessor scheduling problem is defined as how to find a feasible schedule with the minimum Make_Span.

**OUR ENHANCED HILL-CLIMBING ALGORITHM**

A well-known hill-climbing algorithm for static scheduling was proposed by Bokhari[23]. This algorithm starts with a complete solution and search to improve this solution by choosing a better neighbor[23]. 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[5,11]. 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[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 solution encoding and the corresponding schedule:**
A valid solution is encoded in two parts[39]. These parts are:

\[
\text{Sp1}[1 \ 2 \ \ldots \ n], \text{and Sp2}[1 \ 2 \ \ldots \ n]
\]

Where, \(\text{Sp1}[i]\), \(1 \leq i \leq n\) is the task ID which has order \(i\) in the scheduling. Tasks IDs are numbers from 1 to \(n\). Similarly, \(\text{Sp2}[i]\), \(1 \leq i \leq n\) 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 studies chose an encoding that consists only of \(\text{Sp1}\) 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 Earliest Start (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 \(\text{Sp1}\) and \(\text{Sp2}\).

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

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

**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 exist in Sp1[1, ..., (i-1)]
end

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 S0
• Current_Phase := Ordinary, S := S0, 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 ++
  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
• 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 (S1, S2, Current Phase):
Compare_Hidden_Criteria (S1, S2, Current Phase)
if (Current_Phase = Ordinary) then
    return Compare_Hidden_Criteria_Ordinary (S1, S2)
else
    return Compare_Hidden_Criteria_Local_Optimum_Skipping (S1, S2)

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 (S1, S2):
(1) if Schedule Make_Span of S1 ≠ Schedule Make_Span
    of S2
    return solution with the smaller value
(2) if Number of tasks that finish at the schedule
    Make_Span end time is different for S1 and S2
    return solution with the smaller value
(3) 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
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.

1. Compare_Hidden_Criteria_Ordinary(S₁, S₂)
   (1) if Processor Idle Time of S₁ > Processor Idle Time of S₂
       return solution with the smaller value
   (2) if the sum of squared distance between each processor's end time and the whole schedule Make_Span is different for S₁ and S₂ (Load Balance Parameter)
       return solution with the smaller value
   (3) return "equal"

Figure 2 illustrates the principle of comparing between two solutions S₁ and S₂ ordinary according to the hidden criteria in the Ordinary phase.

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:

1. Add a task which finishes its execution at the schedule Make_Span.
2. Add an immediately blocking task of the last added task.
3. 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:

Compare_Hidden_Criteria_Local_Optimum_Skipping(S₁, S₂)
  for i := 1 to Max Critical path size
    if (Critical path size of Sᵢ and Critical Path size of Sᵢ < i
      return "equal"
    if (Critical path size of Sᵢ < i)
      return Sᵢ
    if (Critical path size of Sᵢ < i)
      return Sᵢ
    if (Earliest Block start time of ith task of critical path of Sᵢ differs from its equivalence of Sᵢ)
      return solution with the smaller value
    if (the number of immediately blocking tasks of ith task of critical path of Sᵢ differs from its equivalent of Sᵢ)
      return solution with the smaller value
  return "equal" otherwise

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.

Local transformation by changing SP1 part: A change in the SP1 part can be done by using a 1-Or-Opt (1-Swap) neighbor. 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 SPI[old] to a new location SPI[new] such that one of the following conditions must be satisfied before moving:

1. new > old, for all old ≤ i = new : (Spi[old], I) ∈ E.
2. new < old, for all new ≤ i < old : (I, Spi[old]) ∈ E.

**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). 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 and it can be illustrated as follows:

Given a random order r (from 1 to n) and random two processors (Pr1, Pr2)
for i := r to n do
    current_task := SPI[i]
    if (SP2[current_task] = Pr1) then SP2[current_task] := Pr2
    else if (SP2[current_task] = Pr2) then SP2[current_task] := Pr1
end

**Local transformation enhancements:** The local transformation has been enhanced by some kinds of analysing the schedule of the old solution. The idea of this enhancement is taken from the monitoring procedures[30] and can be done as follows:

- While constructing the schedule, 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 ≤ i ≤ 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].Pblock).
- An entry Block[i], where, 1 ≤ i ≤ 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:
      1. Move the task Idle[i].Pidle to the right, where i is random 1 – Max_Idle.
      2. Move the task Idle[i].Pblock to the left, where i is random 1 – Max_Idle.
      3. 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 with the exception that r, Pr1, Pr2 are not random and are chosen such that one of the following conditions is satisfied:
      1. SP1[r] = Idle[i].Pidle
         Pr1 = SP2[Idle[i].Pidle], Pr2 = SP2[Idle[i].Pblock].
      2. SP1[r] = Block[i].Pblocked
         Pr1 = SP2[Block[i].Pblocked]
         Pr2 = random number from 1 to n.

**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.

The present experimental results are very close to that given[30] with respect to the solution quality.

![Fig. 3: Improving the Make Span for the first task graph](image-url)
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
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 the work in this paper, 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 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).
REFERENCES