Optimized Parallelization Heuristic for Task Scheduling

Waheed Aslam Ghumman
Technische Universität Dresden
Fakultät Informatik
Nöthnitzer Str. 46, Dresden
Germany
Waheed-Aslam.Ghumman@mailbox.tu-dresden.de

Pedro López-Garcia
Fundación IMDEA Software
Facultad de Informática (UPM)
Campus Montegancedo, Madrid
Spain
pedro.lopez@imdea.org

Abstract: Distributed and multiprocessor computing is a basic need for increasingly complex computational requirements. Scheduling the tasks efficiently is as important in multiprocessor systems as getting the correct results. In this paper, we present a heuristic algorithm for optimized parallelization of tasks in multiprocessor environments. This heuristic can be applied in such multiprocessor environments where task and resource information is completely known at time of scheduling or incomplete task information is known at scheduling time. For the case of incomplete task information, we have given a method of using static cost analysis to get different sets of parallel tasks and one set among these alternatives is chosen in runtime based on the value of input parameter. We have verified our approach using a prolog (CLP) program for random sets of tasks and resource with 100% positive results. Our heuristic algorithm is quite simple yet effective which makes it different than already existing scheduling algorithms, heuristics and genetic algorithms.

Key–Words: Task scheduling heuristic, optimization in task scheduling, optimization heuristic algorithm.

1 Introduction

One of the essential parts of any computer system is a mechanism for allocating the processors of the system among the various competitors for their services [9]. The main criterion for organization of batch processing in homogeneous computing systems (HOS) is minimum time to process a batch of jobs. In real time systems the task completion time is as important as the correctness of the results computed by the system. The examples of this type of real time system are command and control systems, process control systems, flight control systems, the space shuttle avionics system, future systems such as the space station, space-based defense systems such as Strategic Defense Initiative (SDI), and large command and control systems [12]. Therefore, research of parallel processing technology on multiprocessor systems is gaining more and more focus. Task scheduling methods have been widely discussed to optimize the processing time.

Scheduling a set of tasks consists of planning the order of execution of task requests so that the timing constraints are met [3]. Multiprocessors have emerged as a powerful computing means for running real-time applications, especially where a uniprocessor system would not be sufficient enough to execute all the tasks by their deadlines [3]. In this paper, we present a heuristic algorithm which optimizes the processing time for parallelization of batch of tasks in a multiprocessor environment. Many heuristic algorithms exist for specific instances of the task scheduling problem, but are inefficient for a more general case [8]. Andrew J. Page et al. [10] present an algorithm to dynamically schedule heterogeneous tasks on heterogeneous processors in a distributed system. The scheduler operates in an environment with dynamically changing resources and adapts to variable system resources. Most of the parallelization heuristics for static and dynamic are based on genetic algorithms [7, 2, 6, 14] but our heuristic algorithm is based on a very simple yet efficient method.

In this paper, we introduce a simple and effective heuristic algorithm for optimized task scheduling. Our approach targets such multiprocessor environments in which:

1. Tasks are known in advance with complete information of their required processing time and available processors/resources, OR

2. Tasks are known in advance with incomplete information. Number of tasks, number of processors, nature of each task is known but processing time may depend on runtime input.

In first case (i.e. tasks with complete information), our heuristic algorithm can be applied directly.
But in second case (i.e. tasks with incomplete information), we use cost analysis to compute the cost function for each task, which can be used to calculate exact task cost in runtime and to schedule tasks by using our heuristic algorithm (more details in section 2). In section 2, we will discuss about computing cost functions for incomplete information tasks. In section 3, we will present a very simple and quick method for task scheduling which can be used in such cases where absolute optimization in task scheduling is not required (although in many cases this method returns the optimized results). Then, in section 4, we will extend this simple method and we will present our heuristic algorithm to get the optimized parallelization for task scheduling in a multiprocessor environment. In section 5 we give some conclusions and future work related to this paper.

2 Using Cost Analysis for Parallelization

As discussed in above section that we will compute cost functions for incomplete information tasks. We assume that such tasks which do not have shared resources (e.g. variables) are strictly independent and we can run as many tasks in parallel as possible. We omit the effect of maximizing the resource utilization on system performance. However, if effect of maximizing the resource utilization is important then we can consider using existing analysis and approaches for maximum achievable system throughput in combination with our heuristic. For example, Parsons [11] studied bounds on the achievable system throughput considering memory demand of parallel tasks. We will use cost analysis given by Saumya Debray and Nai-wei Lin [5] and Saumya Debray and Pedro López-Garcia [4] compute the cost functions of tasks and we will further use this task cost in our heuristic.

2.1 Annotation of Cost Intervals

We can use dependency graph to get the sets of parallel tasks as described in [1]. An initial set $S_0$ of parallel tasks includes such tasks which are not dependent on any other task and each task in set $S_1$ of parallel tasks (succeeding the initial set $S_0$) is dependent on one or more tasks of first set $S_0$, similarly each task in a set of parallel tasks $S_n$ will be dependent on one or more tasks from the set of parallel tasks $S_{n-1}$.

**Theorem 1** If $S_i$ is a set of independent tasks which can be processed in parallel to each other where $S_i \in \{S_0, S_1, \ldots, S_n\}$ then each task $T_j \in S_{i+1}$ is dependent on at least one task $T_i \in S_i$.

**Proof:** Proof of this theorem is trivial as if a task $T$ from a set $S_{i+1}$ is not dependent on any task in its preceding task set $S_i$ then task $T$ can be moved to $S_i$.

As, each set $S_i$ has some tasks which can be processed in parallel, so we will focus on parallelizing such tasks from $S_i$. As mentioned above, we will use cost analysis of Sumaya Debray and others [5], [4] to find the cost function of input parameters for each task. After getting the cost function for each task, we will compare these cost functions. We need these comparison, so that we can arrange these tasks in some order and then we may apply some further method to parallelize them. In runtime environment, cost function for each task will return a positive real number $R^+$ (based on the value of input parameter).

As Tarski[13] showed that the theory of real closed fields coincides with the theory of reals, so we can use properties of real closed fields to give few axioms which can be used to accelerate comparison process of tasks’ cost. Suppose, $C(T)$ represents cost of task $T$ then:

TC1: $C(T_\alpha) \leq C(T_\beta) \Rightarrow C(T_\alpha) \leq C(T_\gamma)$
TC2: $C(T_\alpha) \leq C(T_\beta) \Rightarrow C(T_\alpha) + C(T_\gamma) \leq C(T_\beta) + C(T_\gamma)$
TC3: $C(T_\alpha) \leq C(T_\beta)$ and $C(T_\gamma) \leq C(T_\delta)$ implies $C(T_\alpha) \times C(T_\gamma) \leq C(T_\beta) \times C(T_\delta)$

**Example 1**

For example we have two tasks $T_1$ and $T_2$ with cost functions e.g. $C(T_1) = x \times 3$ and $C(T_2) = x^2$. We will compare these cost functions to know which function results in greater cost. Cost function may result in constant, linear, quadratic, logarithmic or exponential functions. We can use standard mathematical techniques to compare each type of cost function. A graphical comparison for cost functions of $T_1$ and $T_2$ is shown in figure 1. As we can see from the figure 1 that we may get multiple answers as a result of comparing two cost functions, so we may get some intervals. And depending on the value of $x$ (input variable), we can decide which task will be greater for that value of $x$. These static cost analysis of our tasks will give some comparisons of tasks in form of intervals which will be checked on runtime to decide the exact cost of task. We can divide these comparison intervals.
Figure 1: Comparison for cost functions of two tasks

\[ T_1 = 3x \]
\[ T_2 = x^2 \]

\[ T_1 \geq T_2 \text{ if } x \in [0, 3] \]
\[ T_1 < T_2 \text{ otherwise} \]

Figure 1: Comparison for cost functions of two tasks in two sets, one set \( CI_1 \) contains such cost intervals in which \( T_1 \geq T_2 \) and second set \( CI_2 \) contains such cost intervals in which \( T_1 < T_2 \). So, we will get two alternatives as ordered sets of parallel tasks:

1. \( S = T_1, T_2 \) when \( T_1 \geq T_2 \)
2. \( S = T_2, T_1 \) when \( T_1 < T_2 \)

Using above cost analysis, we may have more than one alternatives as set of parallel tasks which are dependent on value of input parameter in runtime environment. But only one set among all alternatives will be selected and then our heuristic for parallelization will be applied.

3 Towards Automatic and Optimized Parallelization in Task Scheduling

In above section, we have illustrated that task cost can be calculated using cost analysis. Moreover, using dependency graph, we get sets parallel tasks i.e. each set contains such tasks which can be processed in parallel. So, we will focus on only one set of parallel tasks. Generally, if number of resources are greater than the number of required resources then there is no need to apply any method for parallelization as each task can get available resource without any wait. But if number of resources are lesser than required then some method for assigning the resources to tasks is desired that ends up in an optimized time.

3.1 Method for Resource Assignment

We will first illustrate the method of assigning the resources to tasks assuming that all the given in a set \( S_i \) can be executed in parallel mode depending on the availability of resources.

Example 2

For example, we have 2 resources and we have 5 tasks \( S = \{T_1(8), T_2(5), T_3(1), T_4(10), T_5(6)\} \). \( T_1(8) \) represents the task cost(load) of 8 units of time for \( T_1 \) and \( T_5(6) \) represents the task load of 6 units of time for \( T_5 \). We will assign resources to tasks in the same order as given in their task loads set \( S \) i.e. according to this set first element \( T_1(8) \) represents the first, so we will first assign resource to \( T_1 \) and then if there is any other free resource, it will be assigned to next task and when one task has been completed then free resource will be assigned to next task in the tasks set \( S \). Resource assignment and task completion for set \( S \) is shown in figure 2.

We can see from the figure 2 that we have total task loads of 30 units of time (i.e. \( 8+5+1+10+6=30 \)) and we have two resources, so we start assigning resources to tasks in the order as given in the task load set \( S \). Resource 1 is assigned to \( \{T_1(8)\} \). As Resource 2 is free, it is assigned to \( T_2(5) \). After 5 units of time Resource 2 will be available for next assignment, so Resource 2 is assigned to \( \{T_3\} \). Similarly, we will continue assigning free resources to next task in \( S \) until set is empty. In the end, total time taken for all tasks is equal to the maximum of the times when any resource finished its last task. In this example, total time is maximum of 14 and 16 i.e. maximum of times when Resource 1 and Resource 2 finished their last task respectively. So, in this example, total time taken for all tasks while executing in parallel and using 2 resources is 16 units of time.

Definition 2 Total Processing Time (\( T_P \))

Suppose we have:
• \( N \) number of processors as \( \{P_1, P_2, ..., P_N\} \) and we have a set of tasks \( S = \{T_1(L_1), T_2(L_2), ..., T_M(L_M)\} \) (arranged in a certain order), where \( T_i(L_i) \) represents a task \( T_i \) with task load \( L_i \) \((i \in \{1, 2, ..., M\})\),

• If we start executing tasks in parallel on all processors such that every free processor is assigned a new task immediately from the task set \( S \),

\[ RT_j \in (RT_1, RT_2, ..., RT_K) \]

where \( RT_j \) represents the time at which resource \( j \) processes all the tasks assigned to it.

Then total processing time \( T_P \) is maximum of all the times at which resources finish processing their last task i.e. \( T_P = \max(\{RT_1, RT_2, ..., RT_K\}) \).

**Definition 3 Optimized Processing Time**

(\( T_{opt} \)) Suppose we have \( N \) number of processors as \( \{P_1, P_2, ..., P_N\} \) and we have \( M \) number of tasks represented as their loads i.e. \( \{T_1(L_1), T_2(L_2), ..., T_M(L_M)\} \), then optimized processing time \( (T_{opt}) \) is calculated as following:

\[
T_{opt} = \max \left( \frac{\sum_{i=1}^{N} T_i(L_i)}{N} \right)
\]

It is not necessary that in every case, \( T_{opt} \) is exactly equal to \( \frac{\sum_{i=1}^{N} L_i}{N} \). For example if we have two tasks with task load 5 and 1 and we have two processors then \( T_{opt} = \frac{5+1}{2} = 3 \) but if we assign tasks to processors e.g. first task with 5 task load is assigned to one processor and other task with task load of 1 is assigned to other processor. So \( T_P = \max(5, 1) = 5 \) which is greater than \( T_{opt} \) and can not be improved in any case.

### 3.2 Optimizing Total Processing Time

We present here a method for optimizing the \( T_P \). According to this method, we can optimize the \( T_P \) if we arrange the set of task loads of all tasks in descending order and then start assigning the tasks to free resources(processors). Continuing with example 2 in which we have 2 resources and 5 tasks \( S = \{T_1(8), T_2(5), T_3(1), T_4(10), T_5(6)\} \), we will arrange this set in descending order of task loads, so new set will be:

\[ S_{desc} = \{T_4(10), T_1(8), T_5(6), T_2(5), T_3(1)\} \]

Resource assignment for this order is shown in figure 3. As we can see that this order has resulted in \( T_P = T_{opt} = \frac{30}{2} = 15 \).

---

**Figure 3: Optimizing resource assignment using descending order method**

<table>
<thead>
<tr>
<th>Resource 1</th>
<th>T1(5)</th>
<th>T2(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource 2</th>
<th>T1(8)</th>
<th>T2(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>14</td>
</tr>
</tbody>
</table>

---

**Figure 4: Resource assignment using descending order method**

Now we consider another example with six tasks having task loads \( S = \{T_1(2), T_2(3), T_3(5), T_4(2), T_5(2), T_6(2)\} \) and we have two resources.

We will apply the method of arranging the order of resource assignment to these tasks in descending order of their task loads e.g.

\[ S_{desc} = \{T_3(5), T_2(3), T_1(2), T_4(2), T_5(2), T_6(2)\} \]

and then start assigning the resources to these tasks, we still get \( T_P = 9 \) as shown in figure 4.

As \( (T_P = 9) \neq (T_{opt} = \frac{16}{2} = 8) \) for \( S_{desc} \), so we used a prolog program which generated all the permutations of the ordered set \( S \) and then we tried to find out any possibly more optimal order by assigning resources for each permutation and getting its \( T_P \). We found out that there are some more optimal solutions. For example if we arrange these task loads as:

\[ S_{new} = \{T_3(5), T_1(2), T_4(2), T_5(2), T_2(3), T_6(2)\} \]

and we assign the resources to these six tasks in this order then we get \( T_P = 8 \) as shown in figure 5.

---

**Figure 5: Optimized resource allocation**

<table>
<thead>
<tr>
<th>Resource 1</th>
<th>T2(5)</th>
<th>T2(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource 2</th>
<th>T1(2)</th>
<th>T2(5)</th>
<th>T3(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

---
OptimizedOrdered_Set ParallelizationHeuristic (Task_Load_Set S, Num_Of_Resources N)
{
  S_Desc = Arrange_Descending(S).
  T_P = AssignResources(S_Desc, N).

  S_Result = S_Desc.
  T_Prev = T_P.
  T_OPT = Sum(S_Desc) / N.

  if (T_P == T_OPT)
    Then
      return S_{Desc}
  Else
    
    foreach element e of S_Desc
    
    /*make a new order set by moving element e to different position and keeping the remaining set in the same order.*/

    for all possible different positions of e in S_Desc
    
    { 
      S_New = moveToNewPosition(S_Desc, e).
      T_New = AssignResources(L_New, N).

      if (T_New == T_OPT)
        Then
          return S_New.

      if (T_New < T_Prev)
        Then
          
          { 
            S_Result = S_New.
            T_Prev = T_New.
          }
      }
    }

  return S_Result.
}

Figure 6: Heuristic algorithm for optimized parallelization
So that means descending order method is not complete yet, there is still something missing which should be improved to find the most optimal solution if it exists. But as we can see that this new optimized task order is almost same as tasks arranged in descending order with just one difference i.e. $T_2(3)$ is moved from second place to fifth place. So, we will consider this fact in next section as a basis of our heuristic algorithm.

4 Optimized Parallelization Heuristic

The main idea of this optimization heuristic is that we arrange the task loads of all tasks (which can be run in parallel) in descending order and if the $T_P$ is greater than $T_{opt}$ then we start changing the position of only one element from the sorted set $S_{desc}$ and keeping the remaining order as same, then we try assigning resources with this new order $S_{new}$ and if new order results in $T_P = T_{opt}$ then we keep this new order as optimized solution otherwise we continue changing position of just one element in $S_{desc}$ and try to find the most optimal solution with that new order. In the same manner, we keep on trying for all elements of list and in the end we return the order of task loads which resulted in minimum possible $T_P$.

To ensure the correctness of our algorithm, we used another prolog program which generates random number of tasks with random amount of task loads and number of available resources and then we apply our heuristic algorithm. We get $T_P$ using our heuristic algorithm and we compared this $T_P$ with another $T_{PP}$ which is calculated by generating all possible permutations of the given task loads set $S$ and then calculates $T_{P_m}$ for each permutation $m$. Then minimum of all $T_{P_m}$ is taken as $T_{PP}$. We found that (for every randomly generated $S$ and random number of resources), our heuristic algorithm returned the $T_P$ which was equal to $T_{PP}$ i.e. minimum possible solution among all permutations. We could not find any single example where this heuristics fails. As proof for correctness and completeness of this method is quite complex, so we can only claim that this heuristic will return positive results in most of the cases but not all. This approach can also be used as a trade-off between scheduling time and processing time i.e. with lesser scheduling time using this approach may result in a bit longer processing time but result of scheduling time plus processing time may result in same. Heuristic algorithm is given in figure 6.

5 Conclusions and Future Work

In this paper, we have presented optimized parallelization heuristic for task scheduling in limited resource based systems. This heuristic can be applied in such multiprocessor environments where task and resource information is completely known at time of scheduling or incomplete task information is known at scheduling time. For the case of incomplete task information, we have given a method of using static cost analysis to get the different sets of parallel tasks as alternative and one set among these alternatives is chosen in runtime based on the value of input parameter. We have verified our approach using s CLP program for random sets of tasks and resource with 100% positive results. But there may be a rare case that our algorithm does not results in the most optimal possible solution, even in that case our heuristic will be very close to that optimal solution. So this approach can have substantial benefits in processing and scheduling time if we are not concerned about absolute optimization. We also present few axioms to compare cost of different tasks, these axioms are based on properties of the real closed fields. These axioms can be really helpful to make scheduling algorithms more efficient. Related future work includes to analyze the first order logics of task scheduling and to analyze the possibility of a complete, sound and correct axiom system for task scheduling. Heuristic algorithm presented in this paper uses straight forward approach to judge the processing time for a given set (of a certain order) of parallel tasks, so an extension to this algorithm can be to devise a mathematical function which takes ordered set of parallel tasks and returns the processing time for that order of parallel tasks in lesser time than straight forward approach.

Acknowledgments: The first author was supported by the European Master’s Program in Computational Logic (EMCL).

References:


