Computer Science and Engineering

Task Scheduling on a Multiprocessor

Roberto Hoyos Morales
October 7, 2004

Definition:

- A set T of *n* tasks, *m* processors, length of execution time l(t) for each task t in T, and a deadline D. All values (m, l(t) for all t, and D are integers).
- Is there a schedule for the tasks in T on the m processors that completes the execution of all tasks that meets the deadline of D?

Objectives:

1. To show that the **Task Scheduling on a Multiprocessor** problem is NP-complete.
2. Description of a solution with run-time that is polynomial in m, D, and n.

In this paper I solve the problem of *Task Scheduling in Multiprocessors*, based upon the following assumptions:

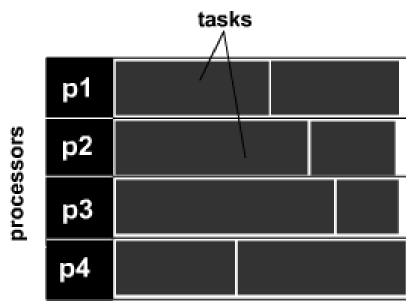
1.- All the task involved are independent from each other.
2.- All processors are homogeneous
3.- *Partition* is a NP-Complete problem (meaning we take the proof for granted)

Understanding the problem

First Approach. Intuition.

Would it be possible to arrange a number of tasks (in this case 8) with different processing times for each one in a way that can be within the limit set by the deadline?

The answer is yes, since for this particular example I could obtain an answer.



But how can I express it in a more general and abstract way?

The first thing is to acknowledge the time we require for *all* of the processes involved.

That sum *must* be less or equal to D x p (*deadline* ***times*** *number of processors)*, since the amount of time of parallel work represented in linear time will be enhanced by any processor added. Thus: $\mathbf{\Sigma\ l(n) \leq D \times p}$

The problem is how to order or divide such tasks so that they can become a number of sets with similar or equal length of time. Every set having less or equal time to D.

I know this first draft to grasp a solution is not *mathematics-proof*, but it gives us a general idea of the problem we are dealing with. And since we don't know its complexity it has laid down the foundation from which we can figure out a solution. This is where *black magic* operates, and where we know that a problem for which we know its complexity (*partition*) is involved, and can help in getting a solution.

Keeping Things Simple

Let us start with the minimum definition of a multiprocessor layout: two processors. This is vital because I intend to prove that we can reduce *partition* into a *two-proccessor task scheduling* (***2p-TS***), and then use that result as way to prove that multiprocessor task scheduling (***mp-TS***) can be obtained from ***2p-TS,*** that is, reducing ***2p-TS*** to ***mp-TS.***

Q.- Is *two-processor task scheduling* an NP-complete problem?

Considering (P, s) as a *partition*, where 'P' is for *Partitioned set* and 's' for *segment length*. Let D (length of time until deadline) be half of the sum of all the tasks 'r' in the set P. This is basically what I did above with a number *p* of processors.

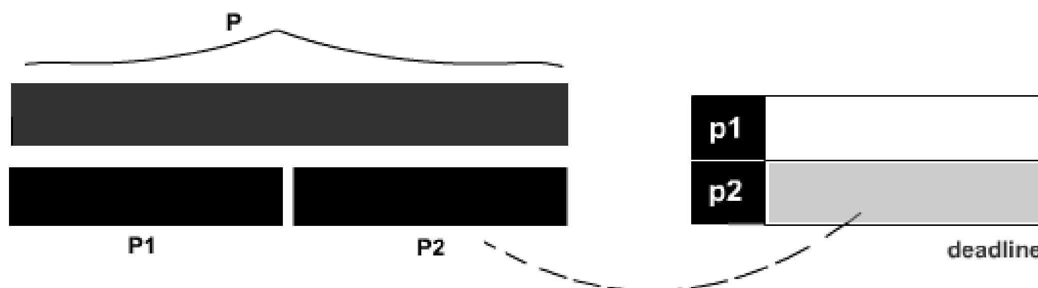
Again, our basic formula now has a fixed number of processors: 2.

$$\Sigma\, l(r) \leq D \times p \quad \rightarrow \quad \Sigma\, l(r) \leq D \times 2$$

and D can be obtained by simple algebra,

$$\frac{\Sigma\, l(r)}{2} \leq D$$

If (P,s) is a partition, then it must have two different sets P_1 and P_2 with their sums less than or equal to D (and both sums actually are equal). And this can easily be mirrored like a ***2p-TS*** triplet (P,s,D), where D is deadline, <u>because subsets P_1 and P_2 can work like a distinct processor schedule in our layout of two *chips*</u>. The sums of every 'r'(task) in each subset P_1 and P_2 will be less or equal to D, as we can know by our formula, and by consequence it will fit with the requirements. So, if there is a solution for (P,s), that solution can work the same for (P,s,D). The next graphic tries to illustrate what I mention above.



So: *two-processor task scheduling* is, indeed, NP-complete.

Will this work with more processors?

The answer is likely to be yes, since we can take any triplet of a ***2p-TS*** (P,s,D) and convert it to a ***mp-TS*** (P^1,s^1,D). But there is to be some adjustments if (P^1,s^1,D) is to be used like a (P,s,D):

We are adding a number *p minus 2* processors to the layout, and because of that every *chip* added must represent a corresponding number of sets {q} of limit D(each set for one processor) in order to keep proportions.

P^1 will be the same set P *but also* a number of {q} sets defined by the number of processors added.

$$P^1 = P \cup \{q\} \cup \{q\} \cup \{q\} \cup \ldots \{q\}$$

where {q} ***can not*** be in the set P because the resulting partitions need to be independent from each other, and by consequence, $\Sigma\ l(q) = D$

With this in mind we can get the very same partition that we obtain with a two processor layout *and* another partition with the rest of the processors (actually p-2) that are working with ‘q’ in the limit imposed by D. This allows us to get to the (P,s,D) form at any given moment if we combine a subset of *p* processors with a subset *p-2*,

$p-(p-2) = 2$ Which is the case from which we came in the first place.

Summary:

We have converted a problem of known complexity (*partition*) into a problem of unknown complexity (the task scheduling for just two processors) and from that approach we have, again, obtained a solution for a problem that resembles it very much.

This is just one part of the business, although. The next phase is to actually write a program that executes the above.

Down to programming; Down to Business

Input: m, D and n (number of *processors*, *deadline* and number of *tasks*, respectively; all integers)

```
//----------------------
//computation phase

maxTime = m x D
sum = 0
sum2 = 0

         for (i=1 to n) {
Ω (n)
            sum = l(n) + sum
         }

if (sum ≤ maxTime) {

    func = choice() //is the sum of sets partitionable in D?

        If (func == 1){
        //if it is then do it!
      partitionSet(m,D,n)

    } else {
     failure
  }

} else {
    failure
}

//----------------------
//verification phase

         for (i=1 to m){

Ω (n)       sum2 = l(n i) + sum2  //sum the tasks of each single processor

            if (sum2 > D)
               failure

         //otherwise continue
}

printSchedule()
success
```

The only real ‘issue’ is to perform the partition (arrangement and division of the tasks so that they fit in a given limit) in a non-deterministic way. But this would be no worse than Ω (n). Thus I shall say that the whole program is in the *k*n running time. But I might be wrong.