A Task Scheduling Algorithm for Minimum Busiest Processor Idle Time

Emile Haddad

TR 93-32

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

December 12, 1993

A Task Scheduling Algorithm

for Minimum Busiest Processor Idle Time

Emile Haddad
Computer Science Department
Virginia Polytechnic Institute and State University
2990 Telestar Court, Falls Church, Virginia 22042.
Tel. (703) 698-6023, E-mail: haddad@vtopus.cs.vt.edu

Abstract

This paper provides a heuristic to minimize the idle time of the busiest processor in a system in which the number of modules to be executed by every processor has been predetermined for a given precedence graph. Except for the busiest processor, the assignment of modules to the other processors is as evenly as possible. Each of the modules involved is of unit size. An exhaustive enumeration solution of the problem is of NP-complete complexity. The heuristic presented in this paper is of polynomial time .

1. Introduction

Task scheduling has been a widely investigated problem in the area of parallel processing. A typical scheduling assignment of modules to processors is to minimize the total time to execute the interactive modules without any constraint on the number of modules that each processor must handle. The resulting optimal schedule will heavily depend on the number of processors involved and the precedence of modules as given in the task graph. Haddad in [2] investigates a system with unit-length modules in which the number of modules assigned to each processor has been predetermined for optimizing an objective function consisting of a linear combination of four performance and cost metrics representing workload completion time , communication cost, processor utilization cost, and processor idle time cost. The resulting optimal load distribution assigns to the busiest processor a specified peak load, with the ramaining modules distributed as evenly as possible to the other processors. The implied time-constraint is that the total execution time of any other processor may not exceed the total execution time of the busiest processor. Execution time is the actual amount of time during which the processor is busy plus the amount of synchronization delay in which the processor has

to wait for a predecessor module to complete its execution . The goal of the scheduling is to minimize the idle time of the busiest processor, which results in minimizing the overall completion time (turnaround time) of the total workload

2. Description of the Scheduling Algorithm

We first define the following terms which are employed in the description of the algorithm to follow:

Time slot : Each time unit in the scheduling.

Predecessor of module A : a module that has to complete before module A can be executed.

$T_{k.delay}$ *: the number of time units during which processor k is idle.*

X(x) : the predetermined number of modules assigned to the busiest processor.

The description of the scheduling algorithm is as follows:

At every time slot, we assign the modules that do not have any predecessors (call these $\{M_i'\}$), to the processors. We keep track of the number of modules that has been assigned to each processor so that it will not exceed its prescribed upper limit . Once a processor reaches its limit, we eliminate it from the set of available processors, $\{P_i'\}$, i.e., the processors that can still be assigned some modules. We repeat this process of assigning modules for each time slot to $\{P_i'\}$ until all the processors but the first have reached their limits (To ensure that the other processors reach their limits before P_1 does, we always assign the modules to the other processors when $|\{P_i'\}| > |\{M_i'\}|$). Let us denote this time slot in which all the other processors have reached their limits as saturation point.

It may seem that the next step of the algorithm is to simply assign all the remaining modules sequentially to P_1. However, since some of these modules can actually be executed in parallel assuming some of the other processors had not reached their limits, we can move some module already assigned to some other processor, P_j, in a previous time slot, to P_1 if P_1 is idle at that time slot. We then assign one of the modules that can be executed in parallel after the saturation point to P_j. This reallocation of modules is to reduce the number of idle slots of P_1 (yet retain the required number of modules of P_j) and hence is unnecessary when P_1 has no idle slots. Therefore, when P_1 has some idle slot(s), the next step in the algorithm is to assign all sequential modules starting from this time slot to P_1 until $|\{M_i'\}| > 1$ or $\{M_i'\} = \emptyset$, i.e., until we find modules that can be executed in parallel, or until there are no more modules left. If there are no more modules left, then we will no longer be able to perform reallocation and thus the algorithm is terminated. Otherwise, we take the module assigned to some P_j in the idle slot of P_1 and assign this module to P_1 instead. We then take two modules from $\{M_i'\}$, and assign each of

them to P_j and P_1 at some appropriate time slot k, where k > SaturationPoint. To remove P_1's other idle slots, we first check if there is a module in $\{M_i^{'}\}$ that can be executed at time slot k, i.e., parallel with P_j and P_1 in the previous step. If not, we repeat the previous step of assigning all the sequential modules to P_1 until $|\{M_i^{'}\}| > 1$ or $\{M_i^{'}\} = \emptyset$. We then repeat the reallocation step until there are no more modules left (in which the algorithm terminates), or until P_1 no longer has any idle slots. In this case, the final schedule is obtained by arbitrarily assigning any ready module to P_1.

Note that in the step where we assign the modules to the processors, if $|\{P_i^{'}\}| < |\{M_i^{'}\}|$, we can only assign $|\{P_i^{'}\}|$ modules for the processors. Our tie-breaking scheme uses α_i, the longest path from node i to a terminal node, and the numerical label of each module. The nodes with the highest α_i's will be selected. This is based on the justification that these modules have the longest path to a terminal node, and thus delaying its execution can delay any of its successors, which in turn can delay its next successors. This delay can create idle time slots of P_1 that cannot be eliminated even in the reallocation stage of the algorithm. The numerical label is used when there is a tie among α's.

There is also a case where $|\{P_i^{'}\}| > |\{M_i^{'}\}|$. Each time this occurs, we will attempt to distribute the modules evenly among the available processors whose index is greater than 1. This is useful when we attempt to assign all the modules at time slot k, k > SaturationPoint, as parallel as possible to reduce the number of idle slots of P_1. When a situation where $|\{P_i^{'}\}| > |\{M_i^{'}\}|$ occurs more than once, the distribution of the modules to different processors will cause us to take modules from different processors as an attempt to reduce the idle slots of P_1. This creates distinct available processors at time slot k in the swapping process, which causes $\{M_i^{'}\}$ to be executed as parallel as possible, and thus increases the chance of possible reallocation (Recall that we only eliminate an idle slot of P_1 when we can execute some elements of $\{M_i^{'}\}$ in parallel).

The following is a listing of all the variables that are necessary to implement the algorithm:

1. $\{\alpha_i\}$ -- the set of the longest paths from each node i to a terminal node. This can be obtained before the execution of the algorithm using a recursive procedure:

```
function Alpha (Module):  integer;
begin
     if  the successors of Module is nil then
         return 1
     else
         return (1 + max { Alpha's of Module's successors} )
end
```

2. $[x_i']$ - an array representing the number of modules assigned to processors $[P_i]$ so far, i.e., it is an accumulator of the number of modules assigned to each P_i until it reaches its limit, x_i.
3. $\{P_i'\}$ -- the set of processors that have not reached their limits of modules, i.e., the processors with $x_i' < x_i$. In the beginning, none of the processors have been assigned any module, so the accumulator array is initialized to zero, which implies that $\{P_i'\} = \{P_i\}$.
4. $\{M_i'\}$ -- the set of modules that can be executed in the current time, i.e., the modules without any predecessors. This is stored in a heapsort with the root being the module with α_{max}. In the beginning, the modules that do not have any predecessors at this point are the starting modules, and thus we initialize $\{M_i'\}$ to be the set of all starting modules.
5. $[S_{ij}]$ -- the matrix representing the resulting schedule whose element s_{ij} denotes the module assigned to P_i at time j. For convenience, $s_{ij} = 0$ indicates that P_i is idle at time j.
6. LPS -- the index of the last processor selected to execute a module when $|\{P_i'\}| > |\{M_i'\}|$. This is used to keep track of which processors should be used if another case of $|\{P_i'\}| > |\{M_i'\}|$ occurs in the future.

3. The algorithm:

1. Let $[x_i'] = \mathbf{0}$
2. Let $\{P_i'\}$ = all the processors
3. Let $\{M_i'\}$ = all modules without any predecessors
4. Let LPS = 1
5. While $|\{P_i'\}| > 1$ and $\{M_i'\} \neq \emptyset$ do
   - 5.1 Assign the modules to processors:
     - 5.1.1 If $|\{P_i'\}| = |\{M_i'\}|$, simply assign to each P_i the modules in $\{M_i'\}$ and build a new heapsort consisting of the ready successors (the successors without any other predecessors) of these modules.
     - 5.1.2 Else if $|\{P_i'\}| > |\{M_i'\}|$: assign the modules in $\{M_i'\}$ to $\{P_j : P_j \in \{P_i'\}, j > 1\}$, starting from P_j where j > LPS. Build a new heapsort consisting of the ready successors and let LPS = the last selected processor's index. If LPS = p, change LPS to 1.
     - 5.1.3 $|\{P_i'\}| < |\{M_i'\}|$: take from $\{M_i'\}$ the modules with the largest α's (by deleting the root of the heapsort one by one). In the case of a tie, select the ones with the lowest numerical label. Then insert into the heapsort the ready successors of these modules one by one.
   - 5.2 Increase x_i' of each of the assigned processors.
   - 5.3 Let $\{P_i'\}$ = all the processors with $x_i' < x_i$.

6. Let SaturationPoint be the current time unit
7. While $T_{1.delay} > 0$ and $\{M_i'\} \neq \emptyset$ do
   - 7.1 Let IdleSlot = the next time slot where $s_{1j} = 0$
   - 7.2 Assign the modules:
     - 7.2.1 If there exists a module M_i' in $\{M_i'\}$ that can be executed in time slot = SaturationPoint and some k such that $s_{k,IdleSlot} \neq 0$ and $s_{k,SaturationPoint} = 0$ then
       - 7.2.1.1 Let $s_{1,IdleSlot} = s_{k,IdleSlot}$ and $s_{k,IdleSlot} = 0$
       - 7.2.1.2 Let $s_{k,SaturationPoint} = M_i'$
       - 7.2.1.3 Insert the ready successor(s) of M_i' to $\{M_i'\}$ one by one
     - 7.2.2 Else if $|\{M_i'\}| > 1$ then
       - 7.2.2.1 Increment SaturationPoint
       - 7.2.2.2 Find the first k such that $s_{k,IdleSlot} \neq 0$
       - 7.2.2.3 Let $s_{1,IdleSlot} = s_{k,IdleSlot}$ and $s_{k,IdleSlot} = 0$ for some k
       - 7.2.2.4 Remove two modules in $\{M_i'\}$ with the largest α's and assign each of them to P_j and P_1 at time slot = SaturationPoint. In the case of a tie, select the ones with the lowest numerical label.
       - 7.2.2.5 Insert into $\{M_i'\}$ the ready successor(s) of these modules one by one
     - 7.2.3 Else
       - 7.2.3.1 Increment SaturationPoint
       - 7.2.3.2 Remove M_i' from $\{M_i'\}$ and assign it to $s_{1,SaturationPoint}$.
       - 7.2.3.3 Insert the ready successor(s) of M_i' in $\{M_i'\}$ one by one
8. While $\{M_i'\} \neq \emptyset$ do

   Arbitrarily remove any element of $\{M_i'\}$ and insert the ready successors of this element to $\{M_i'\}$. The insertion is performed by merely copying them to the array used to store $\{M_i'\}$.

4. Running Time of the Algorithm

The following approximation on major steps of the algorithm provides an upper bound of the algorithm in terms of the number of comparisons performed:

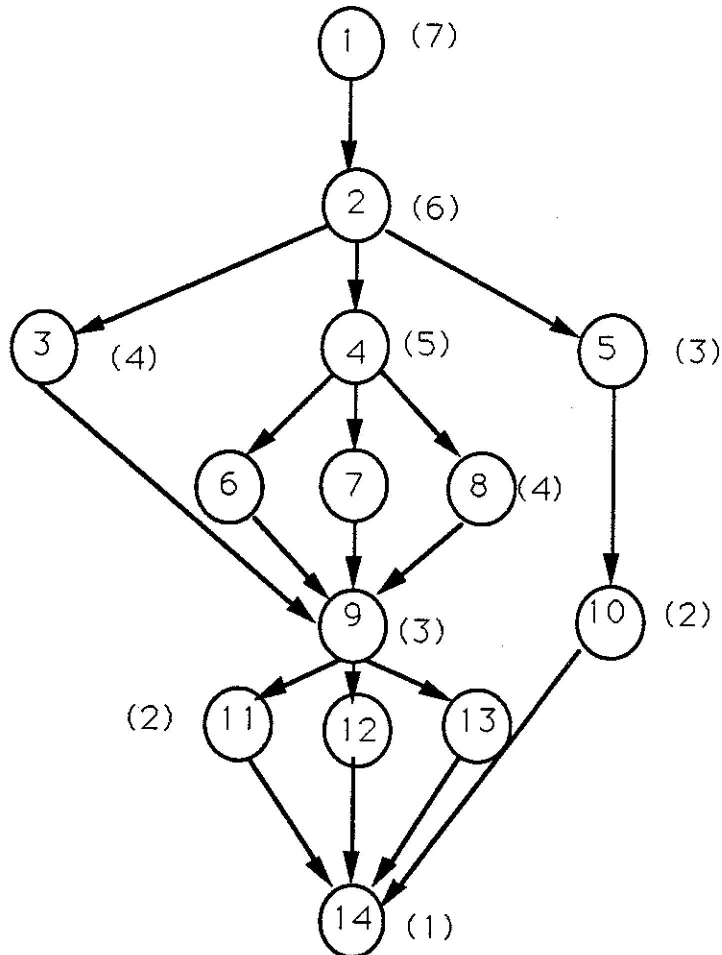
Steps	Upper bound of the number of comparisons	Explanation
3	O (m)	The worst case is when all the modules are the starting modules
5.1.1 & 5.1.2	$O(\lvert\{M_i'\}\rvert)$, simplified to O (m)	The assignment of the modules is $\lvert\{M_i'\}\rvert$ since we need only copy the content of $\{M_i'\}$. Also, building a new heap takes $O(\lvert\{M_i'\}\rvert)$. Since $\lvert\{M_i'\}\rvert \leq m$, we simplify the notation to O (m)
5.1.3	$O((\lvert\{P_i'\}\rvert + \lvert\{M_i'\}\rvert) \lg \lvert\{M_i'\}\rvert)$, simplified to O ((p+m) lg m)	Deleting the max and inserting an element to a heapsort each takes lg $\lvert\{M_i'\}\rvert$ in the worst case. The root is deleted $\lvert\{P_i'\}\rvert$ times and we perform $\lvert\{M_i'\}\rvert$ inserts. Since $\lvert\{M_i'\}\rvert \leq m$ and $\lvert\{P_i'\}\rvert \leq p$, we simplify the first notation to (p+m) lg m).
5.3	$O(\lvert\{P_i'\}\rvert)$, simplified to O (p)	We need to check all elements of $\{P_i'\}$
Entire of **5**	$O(m(\lvert\{P_i'\}\rvert + \lvert\{M_i'\}\rvert) \lg \lvert\{M_i'\}\rvert)$, simplified to O **(m (p+m) lg m)**	The entire step 5 will be executed no more than m times. The worst running time of its statements is O ((p+m) lg m), and thus the total running time for this step is O (m (p+m) lg m)

7.2.1	$\boldsymbol{O}(\lvert\{M_i'\}\rvert)$, simplified to $\boldsymbol{O}(m)$	We need to check individual elements of $\{M_i'\}$ if any can be executed immediately.
7.2.1.3, 7.2.3.3	$\boldsymbol{O}(\lvert\{M_i'\}\rvert \lg \lvert\{M_i'\}\rvert)$, simplified to $\boldsymbol{O}(m \lg m)$	We need to insert the successors one by one
7.2.2.4, 7.2.2.5	$\boldsymbol{O}((2 + \lvert\{M_i'\}\rvert) \lg \lvert\{M_i'\}\rvert)$, simplified to $\boldsymbol{O}(m \lg m)$	See explanation on Step 5.1.3 and remove the constant 2
Entire of **7**	$\boldsymbol{O}(m (2 + \lvert\{M_i'\}\rvert) \lg \lvert\{M_i'\}\rvert)$, simplified to $\boldsymbol{O}(\mathbf{m^2 \lg m})$	See explanation on Step 5
8	$\boldsymbol{O}(\lvert\{M_i'\}\rvert)$, simplified to $\boldsymbol{O}(\mathbf{m})$	Insertion is performed merely by taking each element of $\{M_i'\}$ and copy its ready successors to the heapsort array

Thus the total running time of the algorithm is the maximum of the above, which is $\boldsymbol{O}(m(p+m) \lg m)$ (Recall that $\boldsymbol{O}(f+g) = \max\{\boldsymbol{O}(f), \boldsymbol{O}(g)\}$). Note that, for simplification, many of the upper bounds are approximated higher than the actual bounds. The results, however, is still efficient, as it is a polynomial, which implies that the actual running time of the heuristic is reasonable.

5. Examples

Consider the following task graph:



From the above graph (the numbers in parentheses indicate α's of the nodes) we observe that $m = 14$ and $\alpha_{max} = 7$. Consider the following examples:

1. Let $p = 3$ and $X(x) = \alpha_{max} = 7$. So $x = (7, 4, 3)$.

   After completing steps 1 through 6:

time	1	2	3	4	5
P_1	0	0	3	6	9
P_2	1	0	4	7	10
P_3	0	2	5	8	0

   SaturationPoint = 5

Step 7: ($T_{1.delay} \neq 0$)

After completing the first loop of Step 7:

time	1	2	3	4	5	6
P_1	1	0	3	6	9	11
P_2	0	0	4	7	10	12
P_3	0	2	5	8	0	0

($T_{1.delay} > 0$ and $\{M_i'\} \neq \emptyset$)

The schedule after completing the second loop of Step 7:

time	1	2	3	4	5	6
P_1	1	2	3	6	9	11
P_2	0	0	4	7	10	12
P_3	0	0	5	8	0	13

($T_{1.delay} = 0$) After completing Step 8 we obtain the final (optimal) schedule:

time	1	2	3	4	5	6	7
P_1	1	2	3	6	9	11	14
P_2	0	0	4	7	10	12	0
P_3	0	0	5	8	0	13	0

2. Let $p = 3$ and $X(x) = 6 < \alpha_{max}$. So $x = (6, 4, 4)$.

After completing steps 1 through 6:

time	1	2	3	4	5
P_1	0	0	3	6	0
P_2	1	0	4	7	9
P_3	0	2	5	8	10

SaturationPoint = 5

Step 7: ($T_{1.delay} \neq 0$)

After completing the first loop of Step 7:

time	1	2	3	4	5	6
P_1	1	0	3	6	0	11
P_2	0	0	4	7	9	12
P_3	0	2	5	8	10	0

($T_{1.delay} > 0$ and $\{M_i'\} \neq \emptyset$)

The schedule after completing the second loop of Step 7:

time	1	2	3	4	5	6
P_1	1	2	3	6	0	11
P_2	0	0	4	7	9	12
P_3	0	0	5	8	10	13

($T_{1.delay} > 0$ and $\{M_i'\} \neq \emptyset$)

The schedule after completing the third loop of Step 7:

time	1	2	3	4	5	6	7
P_1	1	2	3	6	0	11	14
P_2	0	0	4	7	9	12	0
P_3	0	0	5	8	10	13	0

Now $\{M_i'\} = \emptyset$ so the above schedule is the final one (with $T_{1.delay} = 1$)

3. Let $p = 2$ and $X(x) = \alpha_{max} = 7$. So $x = (7, 7)$.

After completing steps 1 through 6:

time	1	2	3	4	5	6	7
P_1	0	0	4	6	8	9	11
P_2	1	2	3	7	5	10	12

SaturationPoint = 7

Step 7: ($T_{1.delay} \neq 0$)

After completing the first loop of Step 7:

time	1	2	3	4	5	6	7	8	9
P_1	0	0	4	6	8	9	11	13	14
P_2	1	2	3	7	5	10	12	0	0

Now $\{M_i'\} = \emptyset$ so the above schedule is the final one (with $T_{1.delay} = 2$)

6. Concluding Remarks

The above examples suggest a relationship between p, X(x), α_{max}, and the number of idle time slots of the first processor. Another relevant factor that may not seem obvious is the maximum number of modules that can be executed in parallel if there are enough processors available. When p is less than the maximum number of modules that can be executed in parallel, it will be necessary to execute some modules to the next time slot. Since these modules

may have some successors, delaying their execution can delay the successors of their successors, which may cause an unrecoverable delay in the schedule of the first processor, i.e., a delay that can not be reallocated to any other processor. On the other hand, X(x) and α_{max} are also significant factors: when $X(x) < \alpha_{max}$, we are eventually forced to assign some modules to a processor other than the busiest to satisfy the loading constraint, which creates a time delay in the corresponding time slot of the first processor.

The algorithm presented above was tested against several other examples. In all cases, it provided an optimal schedule, i.e., a schedule in which the amount of idle time of the busiest processor is the minimum possible. Like other NP-complete problems, however, it is not at all trivial to prove that a solution to a graph-related problem is optimal. The proof is therefore still an open problem.

References

[1] S. Bokhari, Assignment Problems in Parallel and Distributed Computing, Kluwer Academic Publishers, 1987.

[2] E. Haddad, "Optimal Distribution of Random Workloads over Heterogeneous Processors with Loading Constraints." *Proc. of the 1992 International Conference on Parallel Processing*, St. Charles, Illinois, August 1992.