African University of Science and Technology

Scheduling in Distributed Systems

Pr. Ousmane THIARE

http://www.ousmanethiare.com

August 18, 2014

Outline

Outline

Issues and assumptions

Be a set of tasks (work) interdependent and a fixed number of processors. We want to determine the order of execution of the tasks so that the execution of all the shortest time possible, knowing that we can perform some tasks in parallel.
Assumptions applicable to a schedule:

- H1 - Static execution: we known set of tasks, their duration and structure of dependency graphs (that is to say the set of pairs (T_i, T_j) such that T_i must be completed for T_j can begin).
- H2 - time invariant: the duration of a task is the same regardless of the context in which it runs.
- H3 - indivisibility: the tasks are not pre-emptive (not fragmentable).

Issues and assumptions

Be a set of tasks (work) interdependent and a fixed number of processors. We want to determine the order of execution of the tasks so that the execution of all the shortest time possible, knowing that we can perform some tasks in parallel.
Assumptions applicable to a schedule:

- **H1 - Static execution:** we known set of tasks, their duration and structure of dependency graphs (that is to say the set of pairs (T_i, T_j) such that T_i must be completed for T_j can begin).
- **H2 - time invariant:** the duration of a task is the same regardless of the context in which it runs.
- **H3 - indivisibility:** the tasks are not pre-emptive (not fragmentable).

Issues and assumptions

- **H4 - immediate communication:** there is no delay in communications. T_j can begin as soon as T_i finished.
- **H5 - number of processors is sufficient:** whatever the proposed scheduling, we have enough processors.
- **H6 - lack of priority:** there is no a priori means of setting priorities on tasks.
- **H7 - resources are sufficient :** tasks are never blocked by lack of resources (disk, memory, ...) and processors are powerful enough to support them.

In the following, unless otherwise specified, the assumptions H2, H3, H7 will be checked.

Outline

System tasks

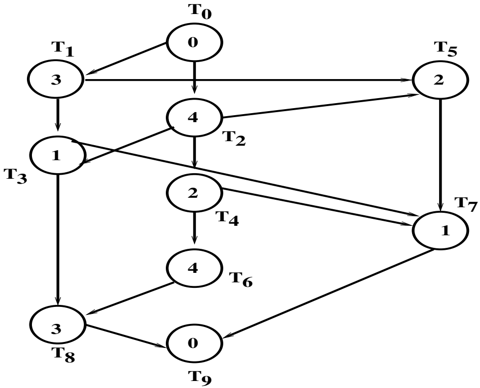
Take

- a set of tasks $\{T_1, \cdots, T_n\}$ and a set of runtimes: $\{ex(T1), \cdots, ex(T_n)\}$ without communication
- and a precedence relation $<<$ as $T_i << T_j$ if T_i must be completed for T_j can begin

We called precedence graph, a graph in which:

- nodes represent tasks;
- two fictitious tasks T_0 said initial task, and T_{n+1} called final task with zero duration are added;
- nodes take the duration of the task which they are derived.

System tasks



Outline

System tasks

A scheduling on p processors is defined as an application Ord defined on $\{T_1, \cdots, T_n\}$ to $(N, [1, \cdots, p])$ associating each T_i the couple (start(T_i),processor(T_i)) where early (T_i) is the start date of T_i and processor(T_i) the processor assigned as:

- if $T_i << T_j$ then $start(T_j) - start(T_i) \geq ex(T_i)$
- si processeur(T_j)=processeur(T_i) then
  - $start(T_i) + ex(T_i) \leq start(T_j)$ or
  - $start(T_j) + ex(T_j) \leq debut(T_i)$

Condition 2 ensures that two tasks can not be carried out simultaneously on the same processor.
Note simply the t_i the start time (T_i), also called *potential*.

Outline

Dates earlier / later than

The previous example, calculate the minimum time before a task can run

$\triangleright(t_1 = 0, p_1)$ and $(t_2 = 0, p_2)$ because nothing precedes these tasks

$\triangleright t_3 =?$ Or

$$\left\{ \begin{array}{ccc} t_1 \ll t3 & \Leftrightarrow & t_1 + ex(T_1) \leq t_3 \\ \text{and} & & \\ t_2 \ll t_3 & \Leftrightarrow & t_2 + ex(T_2) \leq t_3 \end{array} \right. \tag{1}$$

whence $t_3 = max(t_1 + ex(T_1), t_2 + ex(T_2))$

$\triangleright t_8 = max(t_3 + ex(T_3), t_6 + ex(T_6)) =$
$max(max(t_1 + ex(T_1), t_2 + ex(T_2)), t_6 + ex(T_6)) = \cdots = 10$

Dates earlier / later than

Bellmann's algorithm implements this calculation :

$t_0 = 0$; mark T_0

While there are unmarked vertices do

whether T_j unlabeled vertex whose all predecessors T_k are marked (there are at least one if there is a cycle in the graph) then

$t_j = max\{t_k + ex(T_k)\}$

mark T_j

Note in the example, T_1 can start at t = 1 without affecting the application execution time.

On the other side, as soon as its start date is greater than 1s, the entire application is delayed.

Note that for $T_{n+1} = 9$, we obtain $t_9 = 13s$

Dates earlier / later than

Critical Path: The minimum duration of the application is then the maximum value of the paths leading from T_0 to T_{n+1}. It is called the critical path.
In the example $\Rightarrow T_0, T_2, T_4, T_6, T_8, T_9$ for 13s (which is the start date of T_9).
Dates earlier / later than Be called:

- earliest date t_i for T_i, la maximum value of all the paths of T_0 to T_i (the earliest date t_i for T_i is calculated so obvious by the algorithm Bellmann).

Dates earlier / later than

Critical Path: The minimum duration of the application is then the maximum value of the paths leading from T_0 to T_{n+1}. It is called the critical path.
In the example $\Rightarrow T_0, T_2, T_4, T_6, T_8, T_9$ for 13s (which is the start date of T_9).
Dates earlier / later than Be called:

- earliest date t_i for T_i, la maximum value of all the paths of T_0 to T_i (the earliest date t_i for T_i is calculated so obvious by the algorithm Bellmann).
- latest date d_i for T_i, t_{n+1} - the maximum value of all the paths of T_i to T_{n+1}

Dates earlier / later than

Indeed, the value of a path from T_i to T_{n+1} is the time it takes at least the corresponding branch to execute.

i	1	2	3	4	5	6	7	8	9
$Dtard_i$	1	0	9	4	10	6	12	10	13

It is easy to show that for the task T_i of the critical path: $t_i = di$

Outline

Optimality / minimality of a scheduling

- **Total execution time of a task system:** The total execution time of execution of a system of tasks is the time between the start of T_0 (initial task) and the end of T_{n+1} (final task)
- **Average execution time of a task in a task system:** The average execution time of a task in a task system is the average execution time of each task.
- **Optimality / minimality of scheduling:** A scheduling O is minimal if for any number of processors, there is no other scheduling whose total execution time is less than O. Any algorithm that provides a scheduling as the execution time of the system is equal to the execution time of the critical path is minimized.

Optimality / minimality of a scheduling

- A scheduling O is optimal if for a given number of processors, there is no other scheduling whose total execution time is less than O.
- Indeed, it may not exist minimum solution. For example, if the sum of the durations of tasks divided by the number of processors is greater than the duration of the critical path, then there can not exist a minimum solution.
- The purpose of a scheduling algorithm will be to find to any tasks system, a minimum scheduling if possible, if not optimal. It will also seek to minimize the average execution time of tasks.

Outline

Case of a static execution (H1) without communication (H4)

If any such scheduling a sufficient number of processors is available that: $\forall i, t_i \leq start(T_i) \leq d_i$ is minimal.
Thus, if we have enough processors, it will suffice to allocate at random to the task T_i as soon as the earliest date t_i is reached but no later than that date at the latest di is reached, so that scheduling is minimal.

Outline

Case of a static execution (H1) without communication (H4)

It then removes the hypothesis H6 often introducing priorities between tasks and managing a list of runnable tasks (that is to say those whose earliest date has passed without the task is started).

First solution in the general case

We apply the previous algorithm, and then when you have to assign a processor becomes available, it assigns it to a next executable task prioritization. If there are no priorities, we can take in order

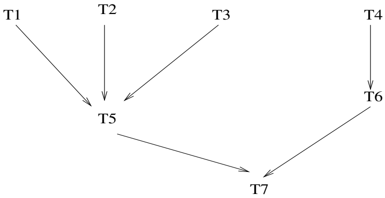
- tasks whose dates are later than most outdated
- those with later dates are closest
- and finally those whose dates are earlier than most exceeded

Case of a static execution (H1) without communication (H4)

Unfortunately, this algorithm does not always provide the optimal schedule (in fact, we can show that the calculation of the optimal schedule is NP-complete). Are there special cases optimal algorithm?.

Case of an anti-tree: The graph is such that each task has only one successor.

Example:



Case of a static execution (H1) without communication (H4)

Case of an anti-tree

One can show that in this case, regardless of the number of processors, scheduling the date later than is optimal.

Thus, in the example, if each task at the same time:

T_1, T_2, T_3, T_4, T_5, T_6, T_7 is great. But also T_2, T_4, T_3, T_1, T_6, T_5, T_7.

In addition, the construction of this list is O(n).

Case of a static execution (H1) without communication (H4)

Case of an arbitrary graph and 2 processors

We classify the tasks according to their dates at the latest. Then for two tasks with the same date at the latest, we apply the following rule:

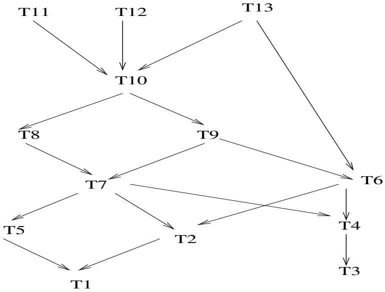
Rule

If all successors of T_i is strictly included in the set of successors of T_j then T_j must be a higher priority than T_i so that we can run the successors of T_j that are not successors of T_i

Case of a static execution (H1) without communication (H4)

Case of an arbitrary graph and 2 processors

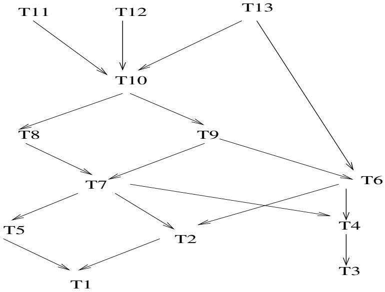
Example: Tasks unit length is assumed (for simplicity of illustration)



Case of a static execution (H1) without communication (H4)

Case of an arbitrary graph and 2 processors

Example: Tasks unit length is assumed (for simplicity of illustration)



Case of a static execution (H1) without communication (H4)

For example, T6 and T7 are the same "level". As $succ(T_6) \subset succ(T_7)$, T7 will be a priority. Intuitively, performing T_7 before T_6 was more likely to allow the continuation of dependent tasks of T_7 that do not depend to T_6: Example T_5.

Case of a static execution (H1) without communication (H4)

Coffman and Graham algorithm, also said labeling algorithm provides a list of priority which respects the previous rule and more reflects the priorities successors. Durations of tasks do not matter.
Choose a terminal task Ti : ET[Ti] = 1
For k=2 to N do whether $S = \{TE1, TE2, \cdots, TEp\}$ labelled all tasks /* that is to say those who have no successors or whose successors are all labeled*/
For every TEi of S do
Compute the list L(TEi) = descending ordered list of labels successors of TEi
Determine TEm such that L(TEm) is less than or equal to all L (TEI) in lexicographical order
ET[TEm] = k

Case of a static execution (H1) without communication (H4)

Coffman and Graham algorithm This algorithm provides a list of tasks in increasing priority.
It then suffices to schedule using the priority and then scheduling is optimal.

Static execution (H1) with communication (NO-H4)

Assumption Communication between tasks takes place only at the end of the task for issuing the start of the receiving task. Thus, the relation of precedence (see "task system") is assigned a communication relationship: each arc in the precedence graph is replaced by an arc of communication.
We will speak about communication graph.

Outline

Static execution (H1) with communication (NO-H4)

Assumption We introduce the communication function $C(T_i, T_j)$ which gives the communication time between task T_i and task T_j:

$$C(T_i, T_j) = \begin{cases} c_{i,j} \text{ if } processeur(T_i) \neq processeur(Tj) \\ 0 \text{ if } processeur(T_i) = processeur(T_j) \end{cases} \tag{2}$$

In fact, $C(T_i, T_j)$ is $c_{i,j}$ (given by the network) if T_i and T_j are not on the same processeur, 0 otherwise (Considering they communicate through shared memory and that this is "instantaneous").

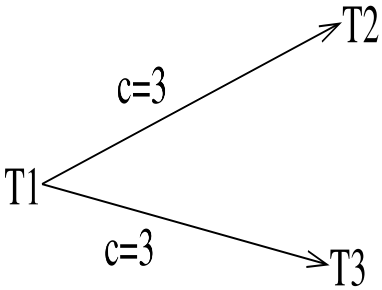
Static execution (H1) with communication (NO-H4)

The relation of precedence becomes:

$$(T_i \to T_j) \Rightarrow \begin{cases} sta(T_j) \geq sta(T_i) + ex(T_i) \, \text{if} \, proc(T_i) = proc(T_j) \\ sta(T_j) \geq sta(T_i) + ex(T_i) + ci, j \, \text{if} \, proc(T_i) \neq proc(T_j) \end{cases}$$

Static execution (H1) with communication (NO-H4)

Hence, one solution is to try to put on a single processor tasks that communicate with it but it does not optimize a situation such as this:



Indeed, a priori, we can start T_2 and T_3 at the same time. Indeed, it is either T_3 is on the same processor as T_1 and then T_2 must wait 3 seconds communication or vice versa is that T_3 must wait.

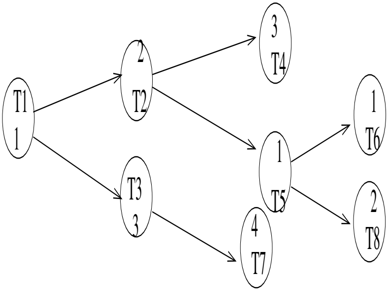
Static execution (H1) with communication (NO-H4)

One solution is to duplicate T_1: on two different processors (eg p_1 and p_2) T_1 is started and once it ends, you can start T_2 on one of these two processors (eg p_1) and T_3 on other (p_2 in this case).
As there is no satisfactory algorithm where tasks are not duplicated, we assume sIn result they are.

Outline

Static execution (H1) with communication (NO-H4)

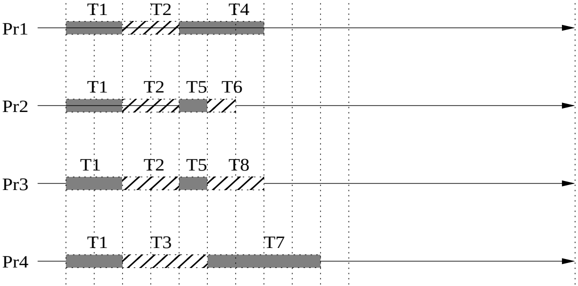
Case of a tree of precedence.
It is possible, as well as each task has a predecessor to associate a processor with each leaf and execute without delay the path from the root to the leaf.
Example:



Static execution (H1) with communication (NO-H4)

Case of a tree of precedence.

gives us:



Outline

Static execution (H1) with communication (NO-H4)

Case of an arbitrary graph

Idea: transform the graph into a tree. For each task T_i, the critical path leading to it is determined. But one task T_j, the predecessor of T_i can be put on the same processor as T_i.

Hence the algorithm:

Static execution (H1) with communication (NO-H4)

Case of an arbitrary graph

For each task T_i whose critical paths of all predecessors are determined

- If T_i has no predecessor, it is considered that T_i can be placed on any of the processors, $s = \emptyset$

Static execution (H1) with communication (NO-H4)

Case of an arbitrary graph

For each task T_i whose critical paths of all predecessors are determined

- If T_i has no predecessor, it is considered that T_i can be placed on any of the processors, $s = \emptyset$
- If T_i has a predecessor T_k, we consider that T_i will be the same processor as its predecessor and calculates the earliest date with a time of no communication, s = k

Static execution (H1) with communication (NO-H4)

Case of an arbitrary graph

For each task T_i whose critical paths of all predecessors are determined

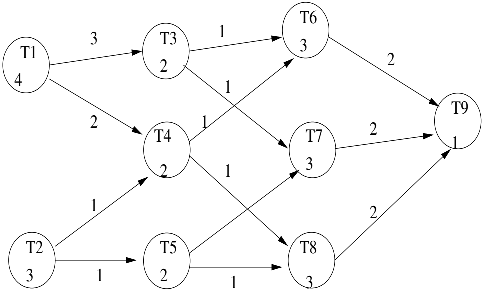
- If T_i has no predecessor, it is considered that T_i can be placed on any of the processors, $s = \emptyset$
- If T_i has a predecessor T_k, we consider that T_i will be the same processor as its predecessor and calculates the earliest date with a time of no communication, s = k
- If T_i is more than one predecessor
  - 1. we calculate all paths leading to T_i assuming that all tasks are on a different processor
  - 2. whether T_{s_m} , the predecessor task of T_i such that $T_0 \to \cdots \to \cdots T_{s_m} \to T_i$ is the critical path
  - 3. we then choose to T_i and T_{s_m} on the same processor, $s = s_m$

Static execution (H1) with communication (NO-H4)

Case of an arbitrary graph

- 4. can then recalculate the critical path (which can have decreased) which is then the earliest date of T_i.

Example:



Static execution (H1) with communication (NO-H4)

Case of an arbitrary graph

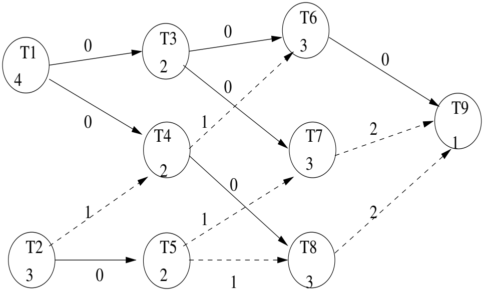
gives:

Tache	1	2	3	4	5	6	7	8	9
D_i	0	0	4	4	3	7	6	6	11
s	-	-	1	1	2	3	3	4	6

Static execution (H1) with communication (NO-H4)

Case of an arbitrary graph

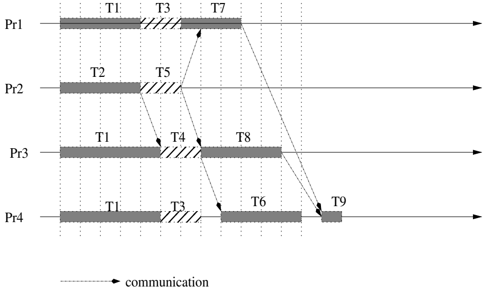
Hence the tree of critical paths (in bold)



Static execution (H1) with communication (NO-H4)

Case of an arbitrary graph

Hence the scheduling



It may be noted that this algorithm gives an allowance of 4 processors while the width of the graph is 3.