Real-Time Distributed Scheduling of Precedence Graphs on Arbitrary Wide Networks

Franck Butelle[1], Mourad Hakem[2] and Lucian Finta[3]
LIPN-CNRS UMR 7030
Université Paris-Nord
99 av. J.B. Clement
93430 Villetaneuse
France
{Franck.butelle,Mourad.Hakem,Lucian.Finta}@lipn.univ-paris13.fr

Abstract

Previous work on scheduling dynamic competitive jobs is focused on multiprocessors configurations. This paper presents a new distributed dynamic scheduling scheme for sporadic real-time jobs with arbitrary precedence relations on arbitrary wide networks. A job is modeled by a Directed Acyclic Graph (DAG). Jobs arrive on any site at any time and compete for the computational resources of the network. The scheduling algorithm developed in this paper is based upon a new concept of Computing Spheres in order to determine a good neighborhood of sites that may cooperate for the execution of a job if it cannot be guaranteed locally. The salient feature of this new concept is that it allows the algorithm to be performed on arbitrary wide networks since it uses a limited number of sites and communication links.

1. Introduction

Real-time systems are those where the correctness of the execution of a task depends not only on the logical results but also on the time they are produced. Examples of such systems include flight control, space shuttle avionics, nuclear power plants, robotics and multimedia. Loosely coupled distributed systems are prevalent and natural candidates for real time applications due to their high performance and reliability.

This paper presents a framework for distributed real-time scheduling of sporadic jobs with deadlines on arbitrary wide networks. A job is a Directed Acyclic Graph (DAG) with arbitrary precedence relations. Each task in the DAG has a given Computational Complexity. Jobs arrive at any time on any site and compete for the computational resources represented by the sites of the network.

Scheduling such jobs with respect to their deadlines is one major challenge in real-time systems.

Most previous researches in the area are restricted to centralized controlled systems and hence are inappropriate for distributed systems. This study differs from previous works for the following reasons:

- Our algorithm uses, for each site, its own local scheduler, and a multi-site mapper. Each site runs the local scheduler independently of the other sites. Thus there is no centralized scheduling control.
- We introduce the concept of Computing Sphere, that is a "good neighborhood": a set of sites that cooperate for the execution of a job and a control structure over this set.

The advantage of Computing Sphere is the use of a limited number of sites and communication links for the execution of a job.

We first introduce some assumptions and notations, next we will present some related work. In section 4 we make a first high level description of the algorithm, detailed in Sections 7 through 10. Section 6 is devoted to the concept of Computing Sphere. Before concluding we introduce some discussions on algorithm generalizations.

2. Assumptions and Notations

The communication network is an arbitrary connected graph composed of an unknown number of sites and bidirectional communication links between them.

1-4244-0910-1/07/$20.00

We assume that each site is composed of two processors, one for the computation of tasks and the other for system management: communications, local scheduler, work distribution, etc.

We assume that a site knows the communication cost (delay) associated to each of its adjacent links. The edges' weights do not satisfy the triangular inequality.

The links are supposed to be faithful, loss-less and order-preserving. The sites are faultless.

For sake of simplicity we will first suppose that all site are identical from Computing Power point of view (i.e. that means that the Computational Complexity of a task represents its execution time).

Any site may receive jobs sporadically. The DAG associated with such a job is denoted by $G = (T, E)$, where T is the set of tasks and E the set of precedence constraints. Each task t_i has a given Computational Complexity $c(t_i)$. Each DAG has a given deadline d.

For sake of simplicity, all weights (Communication Delay, Computational Complexity, etc.) are considered to be non negative throughout the paper.

The surplus I_k of a site k is computed as the ratio of its available (or idle) time divided by the size of the observational window.

3. Related Work

Most of papers on scheduling competitive jobs or tasks deal with centralized control over multiprocessor systems. The case where multiple DAGs are to be scheduled in order to minimize the response time for each one, is not a real-time problem. For instance [7, 8] present a scheduling heuristic based on a dynamic adaptation of the static DLS algorithm presented by Sih and Lee in [11].

For real-time centralized systems, [9] presents a scheduling algorithm that also takes into account a reliability measure of the system.

In the case of real-time distributed scheduling of mutually independent tasks, many papers have been published. Krithi Ramamritham et al. describe in [10] a set of heuristics to schedule tasks with deadlines in a distributed system. In [5, 12, 3], flexible algorithms, that combine focused addressing and bidding are proposed.

To our knowledge, the algorithm [4] is the only one to address the case of real-time competitive DAGs in a distributed environment: when a DAG cannot be guaranteed locally, it is distributed using focused addressing and bidding scheme among a subset of sites. Selection of sites is based on the surplus of each site that is broadcasted over all the network periodically. Unfortunately, in [4] the description of the distributed scheduling algorithm is too succinct, no details are given on what is really sent and how decision is taken. Due to lack of details, we cannot implement and compare this algorithm with the one we propose.

In our paper, we propose a new scheme to distribute a DAG when it cannot be guaranteed locally. Selection of good sites to send tasks to be executed, is based on the concept of Computing Spheres. Our network may be unbounded since we never broadcast over all the network.

4. A High-Level Description of our RTDS Algorithm

Our algorithm RTDS (Real-Time Distributed Scheduling) will be described from the point of view of a site k.

- Construction of the Potential Computing Sphere rooted in k (this is done only once since no site/links failure may occur).
- Upon reception of a DAG
  1. Test if the job (DAG) may be locally guaranteed
  2. Construction of the Available Computing Sphere
  3. Trial-Mapping construction by the Mapper
  4. Mapping validation
  5. Mapping execution after receiving a permutation and tasks code.

5. Local Scheduler

When a new job arrives on site k, local test is performed. It consists on verifying if all tasks of the job may be scheduled in-between tasks already accepted to be scheduled on site k before deadline d.

A similar test will be done for a subset of those tasks during Trial-Mapping validation.

6. Computing Spheres

A DAG arriving on site k is either executed locally or, if possible, executed over a particular neighborhood of k named Computing Sphere (or CS for short). The sites in the Computing Sphere are "close" to k: in terms of hops and communication delay. During the CS construction, a subsequent communication control structure is devised, allowing local broadcast. In the CS the following nice properties exist: each site has a unique minimum communication delay path to k, the diameter in terms of hops is bounded by a constant h and minimum communication delay path between any pair of sites of the Potential Computing Sphere of k (PCS).

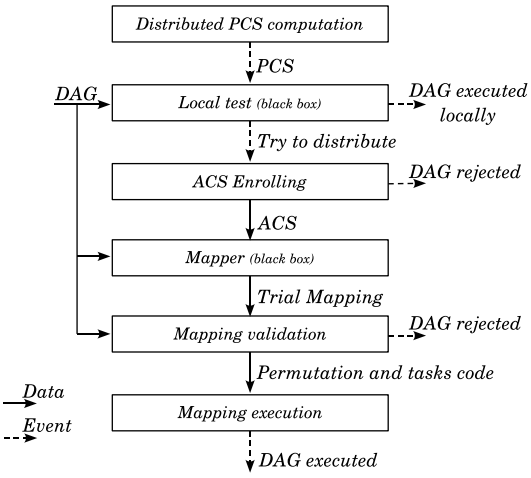

Figure 1. Algorithm overview

7. Distributed PCS Construction Algorithm

The Potential Computing Sphere of a site k is a set of sites that may collaborate for the execution of a job (DAG). It is constructed at the initialization of the system. Obviously, for two sites k and j one may have a non-empty intersection of their PCS. A site in this intersection may participate to the execution of DAGs arriving on k and j.

Several known algorithms may be adapted to achieve PCS construction.

Here we present a PCS construction algorithm based on an interrupted All-Pairs Shortest Paths algorithm. The interruption is made in order to limit network flooding.

First we briefly describe the algorithm of [2] and next how we adapt it.

7.1. Distributed All-Pairs Shortest Paths Algorithm [2]

Start conditions: each site starts with a vector of distances (delays) to all directly attached sites (immediate neighbors). Each node maintains a routing table consisting of route lines like $<$*destination, distance, next Hop*$>$.

Send step: Every site sends its routing table lines to all immediate neighbors. These updates are sent periodically. Updates are send out whenever destination vectors entries in the routing table change.

Receiving step: Update lines of the routing table with the received tuples.

7.2. Our Adaptation of All-Pairs Shortest Paths Algorithm [2]

The periodic updates sent by this algorithm is useless since we do not cope neither with topological changes nor topological failures. We construct the PCS by stopping the previous algorithm after a given number of phases.

First, we adapt it slightly to make it organized into logical phases. Site k starts its PCS construction by sending its routing table. A phase is composed of send step and reception of all neighbor routing tables. This ensure us that each new phase makes distances being accurate one hop further.

So, if the algorithm is stopped after h phases, the distances computed so far by the algorithm, on each site, will contain the accurate minimum distance to any site up to h hops away from this site.

In fact, the algorithm is stopped after $2h$ phases, such that each node of the PCS of k discovers a path to every other nodes of the PCS of k (which will be of hop-radius h).

8. Available Computing Sphere Construction

When a new job (DAG) cannot be guaranteed locally, ACS construction starts (see Figure 1).

ACS members are dynamically selected as a subset of the PCS rooted at site k. Its construction is done by marking/enrolling sites of PCS using the site's routing table.

Mutual exclusion for enrollment from initiator is guaranteed by a lock variable on each local site.

Each enrolled site sends a message back to the initiator k with its surplus.

It also keep a $lock$ value associated to initiator k. If a locked site receive an enrollment message, this message is ignored until reception of unlocking message from k. The set of enrolled sites is called ACS.

9. Trial-Mapping Construction by the Mapper

We do not describe here the details of this algorithm. Almost any heuristic can be adapted to our purpose (see for example [6]).

The Mapper have the following inputs: a DAG to partition using a list of sites with their associated surplus in descending order. The Mapper decides if the DAG is rejected or computes a mapping. Trial-Mapping construction/validation delay should be short enough to avoid deadline missing of the job.

A Trial-Mapping M is a set of three functions: $S : T \rightarrow U$ (where $U = 1, \ldots, |U|$ is a set of logical processors/sites and $S(t)$ is the logical site to which task t is assigned), $r : T \rightarrow \mathcal{R}^+$ where $r(t)$ is the release time of the task t and $d : T \rightarrow \mathcal{R}^+$ where $d(t)$ is the deadline of the task t. Recall that, from the DAG definition, we also have $c : T \rightarrow \mathcal{R}^+$ where $c(t)$ is the Computational Complexity of the task t.

Trial-Mapping M is constructed by the mapper based on PCS and ACS knowledge: surplus of each site j of the ACS and distances between any pair of sites in the ACS.

10. Trial-Mapping Validation

The previous computed mapping have still to be validated by several sites in ACS since the Mapper knows only the surplus but not the exact start and end of idle intervals for a given processor. To do so we broadcast in ACS the mapping M.

Each site j, in ACS, try to endors the role of each logical processor. The initiator k expects, from each site j, the list of logical processors ids that j may endors.

More precisely, upon reception of M, a site j tries to validate all tasks assigned to a logical site i for each $i \in U$. Let T_i be the set of tasks assigned to logical site i (i.e. $T_i = \{t \in T \ / \ S(t) = i\}$). A set of tasks T_i is locally satisfiable by a local scheduler iff each task t of T_i may be executed with respect to its release $r(t)$ and deadline $d(t)$. The list of sites i, for which the set T_i is locally satisfiable, is sent back to the initiator k.

When k has received all the lists from the sites of ACS, it computes a maximum coupling (classical problem in graph theory solved in polynomial time, see e.g. [1]). If the cardinality of the maximum coupling is less than $|U|$ then no combination satisfy all T_i ($i \in U$) of M thus the DAG is rejected and ACS members are unlocked.

If a subset of size $|U|$ of the maximum coupling is found, it gives a permutation of the sites that are able to achieve DAG's computation.

11. Distributed Execution

k sends the previous permutation with the tasks codes to all sites in the ACS.

A site j receiving the permutation learn if it is involved in the execution or not (in this case the lock of j is released). If it is, let i be the logical site identity assigned to j. j executes tasks $t \in T_i$ with respect to their release $r(t)$ and deadline $d(t)$.

The lock of j is immediately released after the insertion of all tasks of T_i in its own scheduling plan.

12. Mapper Instance and Detailed Example

The mapper implementation needs to specify how:

- a task t_i is selected for assignment to some processor
- a processor p_i is selected to execute task t_i
- the release r_i (resp. deadline d_i) is assigned to task t_i.

Here we give only a simple proposal for this three points: our goal is not to give their best implementation, but only a possible implementation in order to detail an example.

Task t_i selection is done by list scheduling based on critical path: the priority of a task t_i is the length of the longest path from t_i to a sink task in the graph (node weights only, t_i included). The list contains only free tasks (task with all predecessors already mapped).

The processor selection is done in a greedy manner: p_i is the processor that allows the earliest finishing time for the execution of the selected task t_i.

Communication delay between t_j (immediate predecessor of t_i) and t_i (denoted by $\omega(p(t_j), p(t_i))$) is taken into account as being (over-estimated by) the computed diameter (in terms of delay) of the current ACS.

Surplus of processor p_i (i.e. I_i) is used for the evaluation of the execution duration of t_i: the execution of t_i is the ratio of Computational Complexity $c(t_i)$ over I_i (the surplus of processor p_i). Start time of t_i on p_i is not sooner than the end of the previous task mapped on p_i, nor before the end of the communications from immediate predecessors of t_i (if any).

Let $\mathcal{S}$ be the schedule computed above and $\mathcal{M}$ be its makespan. Release time r_i (and resp. deadline d_i) is given by the starting (and resp. finishing time) of t_i on p_i. Formally, let $\Gamma^-(t_i)$ and $\Gamma^+(t_i)$ denote the sets of immediate predecessors and successors of t_i respectively.

$$\forall t_i \in DAG,\ d_i = r_i + c(t_i)/I_j \tag{1}$$

$$r_i = \begin{cases} r & \text{if } \Gamma^-(t_i) = \phi \\ \max_{t_j \in \Gamma^-(t_i)} \{d_j + \omega\left(p(t_j), p(t_i)\right)\} & \text{otherwise} \end{cases} \tag{2}$$

These values do not take into account the deadline d of the job. We first introduce an example and then we show how releases and deadlines are scaled.

12.1. Example

Consider the example job of Fig. 2, we assume that surpluses for processors p_1 and p_2 are $I_1 = 0.5$ and $I_2 = 0.4$. We consider the deadline of the job is $d = 66$ and for sake of simplicity its release is $r = 0$. We suppose that the computed diameter of the ACS is equal to 3.

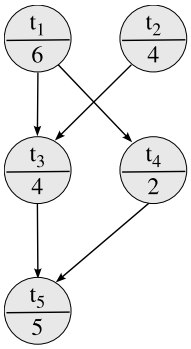

Figure 2. A task graph instance

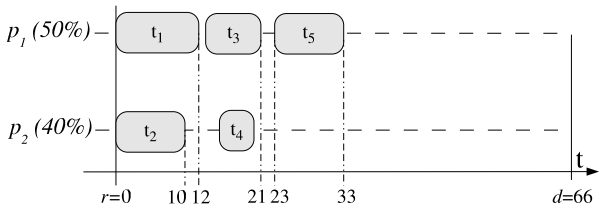

Figure 3. The schedule $\mathcal{S}$ computed by the Mapper

12.2. Adjustment of the Parameters r_i and d_i

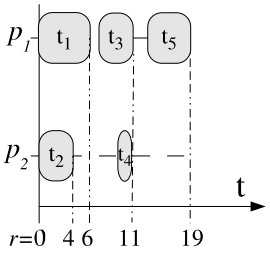

Figure 4. The schedule $\mathcal{S}^*$ computed by the Mapper

Let $\mathcal{S}^*$ be a schedule constructed using the same mapping for the tasks as in $\mathcal{S}$, but with processors surpluses equal to 100% (see Figs. 3 and 4). The makespan of $\mathcal{S}^*$ is denoted by $\mathcal{M}^*$. $\mathcal{M}^*$ is the lower bound of $\mathcal{M}$ for the same mapping.

i) If $\mathcal{M}^* > d - r$ then, the job is rejected.

ii) If $\mathcal{M} \leq d - r$ then update $d(t_i)$ and $r(t_i)$ in topological order, for each task using equations (3) and (5).

iii) If $\mathcal{M}^* \leq d - r \leq \mathcal{M}$ then update $d(t_i)$ for all tasks in reverse topological order using equation (4) and then update $r(t_i)$ using equation (5) for all tasks in topological order.

$$d(t_i) \leftarrow r + (d_i - r)\frac{d - r}{\mathcal{M}} \tag{3}$$

Let η be the maximum number of tasks belonging to any critical path in the schedule $\mathcal{S}^*$. In order to respect communication delays, the deadlines $d(t)$ are scaled by the laxity: $\ell(t) = (d - r - \mathcal{M}^*)/\eta$
Thus,

if $\Gamma^+(t_i) = \emptyset$ then $d(t_i) \leftarrow d$, otherwise,

$$d(t_i) \leftarrow \min_{t_j \in \Gamma^+(t_i)} \{d(t_j) - \ell(t_j) - c(t_j) - \omega(p_i, p_j)\} \tag{4}$$

$$r(t_i) \leftarrow \begin{cases} r & \text{if } \Gamma^-(t_i) = \emptyset \\ \max_{t_j \in \Gamma^-(t_i)}\{d(t_j) + \omega(p_j, p_i)\} & \text{otherwise} \end{cases} \tag{5}$$

For our example, the adjusted values are in Table 1. $\mathcal{M} = 33$ and the scaling factor is $\frac{d-r}{\mathcal{M}} = 2$ (see Table 1).

Table 1. Adjusted $r(t_i)$ and $d(t_i)$

t_i	r_i	d_i	$r(t_i)$	$d(t_i)$
1	0	12	0	24
2	0	10	0	20
3	13	21	24	42
4	15	20	27	40
5	23	33	43	66

13. Discussion

We present in this section some generalizations of our RTDS algorithm.

- *Preemptive Case:* This algorithm may provide better results in the preemptive case.
- *Uniform Machines:* For sake of simplicity, the presented algorithm deals with the case of identical machines. It is easily extendable to the case of uniform machines (related machines) by scaling the site surplus by its Computing Power.

- *Local knowledge of k:* If k is one of the processors selected by the mapper to execute some tasks of the current job, the mapper may use the local knowledge (local idle intervals) instead of using surplus of k only.

- *Laxity Dispatching:* For the case (*iii*) in the previous section, the extra laxity $d - r - \mathcal{M}^*$ is equitably scattered over all tasks belonging to the longest (in terms of number of tasks) critical paths of the schedule $\mathcal{S}^*$. For each such tasks, the busyness $(1 - I)$ of the processor where this task is executed may be used in order to ponderate the extra laxity scattering (tasks on busy processors receive more extra laxity).

- *Communication Delays:* The Communication Delays model only distance (propagation delay) between sites: data transfer rate (throughput) of the links and data volumes between tasks are not considered for sake of simplicity.

  To be more realistic, data volumes may be easily taken into account (decoration of the arcs in the DAG) when all throughputs are identical: communication delays should be adjusted by the ratio data volume over throughput. This adjustment should be done as well for communication delays due to result sending from predecessor to successor task, as in the case of task code sending from k to the processors that participate in the computation of the job (i.e. the job release must be augmented by the computation time taken by the mapper, the time taken by Trial-Mapping validation and also by the dispatching of tasks code).

14. Conclusion

In this paper, we have presented a new approach for real-time distributed scheduling sporadic competitive jobs on arbitrary wide networks. Jobs have deadlines and are composed of tasks with arbitrary precedence relations. The mean features of our algorithms are:

- It is a distributed scheduling scheme and there is no centralized scheduling control

- The jobs may arrive sporadically on any site

- The new concept of Computing Sphere introduced in this paper reduces the communication overhead and allow the algorithm to use a limited number of sites and communications links to distribute a job (when the job is rejected locally). This leads to an increase of the number of accepted (executed) jobs.

References

[1] C. Berge. *Graphes et Hypergraphes*. Monographies universitaires de mathématiques. Dunod, Paris, 1970. English translation : Graphs and Hypergraphs (North-Holland, Amsterdam 1973).

[2] D. P. Bertsekas and R. G. Gallager. Distributed asynchronous bellman-ford algorithm. In *Data Networks*, chapter 5.2.4, pages 325–333. Prentice Hall, Englewood Cliffs, 1987.

[3] A. K. Bhattacharjee, K. Ravindranath, A. Pal, and R. Mall. Ddsched: a distributed dynamic real-time scheduling algorithm. *Progress in computer research*, pages 170–184, 2001.

[4] S. Cheng, J. A. Stankovic, and K. Ramamritham. Dynamic scheduling of groups of tasks with precedence constraints in distributed hard real-time systems. In *IEEE Real-Time Systems Symposium*, pages 166–174, 1986.

[5] M. Hakem and F. Butelle. A new on-line scheduling algorithm for distributed real-time system. In *Proc. of the 3rd Int. Symp. and School on Advanced Distributed Systems*, volume 3061 of *Lecture Notes in Computer Science*, pages 241–251, 2004.

[6] M. Hakem and F. Butelle. Efficient critical task scheduling parallel programs on a bounded number of processors. In *Proc. of the 17th Int. Conf. on Parallel and Distr. Comp. and Syst. (PDCS'05–IASTED)*, pages 139–144, 2005.

[7] M. Iverson and F. Ozguner. Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment. In *HCW '98: Proceedings of the Seventh Heterogeneous Computing Workshop*, page 70, Washington, DC, USA, 1998. IEEE Computer Society.

[8] M. Iverson and F. Ozguner. Hierarchical, competitive scheduling of multiple dags in a dynamic heterogeneous environment. *Distributed Systems Engineering*, 6(3):112–120, 1999.

[9] X. Qin and H. Jiang. Dynamic, reliability-driven scheduling ofi parallel real-time jobs in heterogeneous systems. In *Proc. Of the 30 th International Conference on Parallel Processing*, pages 113–122, 2001.

[10] K. Ramamritham, J. A. Stankovic, and W. Zhao. Distributed schedulings of tasks with deadlines and resource requirements. In *Proc. IEEE trans. on Computers*, volume 38, pages 1110–1123, august 1989.

[11] G. Sih and E. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. on Parallel and Dist. Systems*, 4(2):75–87, 1993.

[12] J. A. Stankovic and S. C. Krithivasan Ramamritham. Evaluation of a flexible task scheduling algorithm for distributed hard real time systems. In *Proc. IEEE trans. on Computers*, volume C-34, pages 1130–143, december 1985.