'International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering' CIS2E 06, December 4 - 14, 2006
Published in : Innovations & advanced techniques in computer & information sciences & engineering, Springer, pp 301-307, (06-2007).

An efficient fault-tolerant scheduling algorithm for precedence constrained tasks in heterogeneous distributed systems

M. Nakechbandi[*], J.-Y. Colin[*], J.B. Gashumba[**]

LITIS Université du Havre, 25 rue Philippe Lebon, BP 540, 76058, Le Havre cedex, France
[(*)]{moustafa.nakechbandi, jean-yves.colin}@univ-lehavre.fr
[(**)] jb.gashumba@free.fr

Abstract

In this paper, we propose an efficient scheduling algorithm for problems in which tasks with precedence constraints and communication delays have to be scheduled on an heterogeneous distributed system with one fault hypothesis. Our algorithm combines the DSS_OPT algorithm and the eFRCD algorithm. To provide a fault-tolerant capability, we employ primary and backup copies. In this scheme, backup copies can overlap other backup copies, and backup copies are not established for tasks that have more than one primary copy. The result is a schedule in polynomial time that is optimal when there is no failure, and is a good resilient schedule in the case of one failure.

Keywords: Scheduling, Heterogeneous Distributed System, Fault-tolerance, Precedence Constraints, Communication Delays, Critical-Path Method.

I. INTRODUCTION

Heterogeneous distributed systems have been increasingly used for scientific and commercial applications, some of the current ones include Automated Document Factories (ADF) in banking environments where several hundred thousands documents are produced each day on networks of several multiprocessors servers, or high performance Data Mining (DM) systems [10] or Grid Computing [9, 11]. However, using efficiently these heterogeneous systems is a hard problem, because the general problem of optimal scheduling of tasks is NP-complete [13].

When the application tasks can be represented by Directed Acyclic Graphs (DAGs), many dynamic scheduling algorithms have been devised. For some examples, see [2, 3, 7]. Also, several static algorithms for scheduling DAGs in meta-computing systems are described in [1, 4, 6, 13]. Most of them suppose that tasks compete for limited processor resources, and thus these algorithms are mostly heuristics. In [5] is presented an optimal polynomial algorithm that schedules the tasks and communications of an application on a Virtual Distributed System with several clusters' levels, although, in [8] is studied a static scheduling problem where the tasks execution times are positive independent random variables, and the communication delays between the tasks are perfectly known.

This review shows that a lot of works has been done concerning heterogeneous distributed scheduling.

However, the problems with fault tolerant aspect are less studied. Reliable execution of a set of tasks is usually achieved by task duplication and backup copy [16, 17, 18]. Inspired with the works of Xiao Qin, Hong Jiang on Real-Time Heterogenous Systems [16], we propose in this article a good algorithm to solve the problem of one fault tolerant system using tasks duplication and backup copies technics.

This paper has three main parts: In the first one, we present the problem, in the second part, we present a solution to the problem, and in the third part, we discuss the advantages and disadvantages of the proposed solution.

II. THE CENTRAL PROBLEM

In this part, we present the following problem : Given an application T, we want to build an off-line scheduling of the tasks of this application in an heterogeneous distributed System with some possibilities of limited failures. We suppose that only one permanent failure can occur without any possibility of a temporary failure. This constitute one hypothesis which we call "1-failure". Note that the processing time of the application has to be minimized.

2.1 The Distributed Servers System

We call Distributed Servers System (DSS) a virtual set of geographically distributed, multi-users, heterogeneous or not, servers. Therefore, a DSS has the following properties:
First, the processing time of a task on a DSS may vary from a server to another. This may be due to the processing power available on each server of the DSS for example. The processing time of each task on each server is supposed known. Second, although it may be possible that some servers of a DSS are potentially able to execute all the tasks of an application, it may also be possible in some applications that some tasks may not be executed by all servers. This could be due to the fact that specific hardware is needed to process these tasks and that this hardware is not available on some servers. Or it could be that some specific data needed to compute these tasks are not available on these servers for some reason. Or it could be that some user input is needed and the user is only located in a geographically specific place. Obviously, in our problem we suppose that the needs of each task of an application are known, and that at least one server of the DSS may process it, else there is no possible solution to the scheduling problem.

Furthermore, an important hypothesis is that the concurrent executions of some tasks of the application on a server have a negligible effect on the processing time of any other task of the application on the same server. Although apparently far-fetched, this hypothesis may hold if the server is a multiprocessors architecture with enough processors to simultaneously execute all the tasks of the application that are to be processed concurrently. Or it may be that the server is a time-shared, multi-user system with a permanent heavy load coming from other applications, and the tasks of an application on this server represent a negligible additional load compared to the rest.

In addition, in the network interconnecting the servers of a DSS, the transmission delay of a result between two tasks varies depending on the tasks and on their respective sites.

Again, we suppose that concurrent communications between tasks of the same application on two servers have a negligible effect on the communication delays between two others tasks located on the same two servers. This hypothesis may hold if the network already has a permanent heavy load due to other applications, and the communications of the application represent a negligible additional load compared to the one already present.

2.2 Directed Acyclic Graph

We now describe the application itself in our problem. An application is decomposed into a set of indivisible tasks that have to be processed. A task may need data or results from other tasks to fulfil its function and then send its results to other tasks. The transfers of data between the tasks introduce dependencies between them. The resulting dependencies form a Directed Acyclic Graph. Because the servers are not necessarily identical, the processing time of a given task can vary from one server to the next. Furthermore, the duration of the transfer of a result on the network cannot be ignored. This communication delay is function of the size of the data to be transferred and of the transmission speed that the network can provide between the involved servers. Note that if two dependent tasks are processed themselves on the same server, this communication delay is considered to be 0.

The central scheduling problem *P* on a Distributed Server System, is represented therefore by the following parameters:

- a set of servers, noted $\Sigma = \{\sigma_1, ..., \sigma_s\}$, interconnected by a network,
- a set of the tasks of the application, noted I = {1,..., n}, to be executed on Σ. The execution of task i, i ∈ I, on server σ_r, $\sigma_r \in \Sigma$, is noted i/σ_r. The subset of the servers able to process task i is noted Σ_i, and may be different from Σ,
- the processing times of each task *i* on a server σ_r is a positive value noted π_{i/σ_r}. The set of processing times of a given task *i* on all servers of Σ is noted $\Pi_i(\Sigma)$. $\pi_{i/\sigma_r} = \infty$ means that the task *i* cannot be executed by the server σ_r.
- a set of the transmissions between the tasks of the application, noted U. The transmission of a result of an task *i*, *i* ∈ I, toward a task *j*, *j* ∈ I, is noted (i, j). It is supposed in the following that the tasks are numbered so that if (*i, j*) ∈ U, then i < j,
- the communication delays of the transmission of the result (i, j) for a task i processed by server σ_r toward a task j processed by server σ_p is a positive value noted $c_{i/\sigma_r, j/\sigma_p}$. The set of all possible communication delays of the transmission of the result of task i, toward task j is noted $\Delta_{i,j}(\Sigma)$. Note that a zero in $\Delta_{i,j}(\Sigma)$ mean that i and j are on the same server, i.e. $c_{i/\sigma_r, j/\sigma_p} = 0 \Rightarrow \sigma_r = \sigma_p$. And $c_{i/\sigma_r, j/\sigma_p} = \infty$ means that either task *i* cannot be executed by server σ_r, or task *j* cannot be executed by server σ_p, or both.

Let $\Pi(\Sigma) = \bigcup_{i \in I} \Pi_i(\Sigma)$ be the set of all processing times of the tasks of *P* on Σ.

Let $\Delta(\Sigma) = \bigcup_{(i,j) \in U} \Delta_{i,j}(\Sigma)$ be the set of all communication delays of transmissions (i, j) on Σ.

The central scheduling problem *P* on a distributed servers system DSS can be modelled by a multi-valued DAG $G = \{I, U, \Pi(\Sigma), \Delta(\Sigma)\}$. In this case we note $P=\{G, \Sigma\}$.

Example: In the following example we consider a problem *P* of an application with nine tasks that has to be processed by a set of 3 servers on an heterogeneous distributed system. The architecture of distributed system and the task graph of this application are represented by the figure 1 and figure 2. It is supposed that the 3 servers are distributed on two different networks.

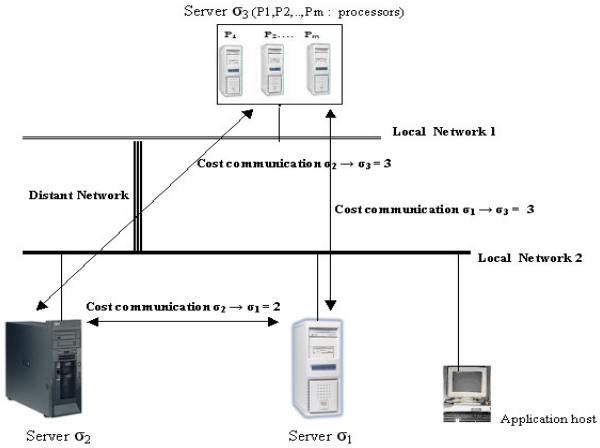

***Figure 1**: Distributed system architecture for the application example.*

The Matrix cost communication of our example is presented in table 1.

Network delay between $\sigma_i \rightarrow \sigma_j$	Server σ_1	Server σ_2	Server σ_3
Server σ_1	0	2	3
Server σ_2	2	0	3
Server σ_3	3	3	0

***Table 1**:* Cost communication *between servers* (*distance* $\sigma_r \rightarrow \sigma_p$ *)*

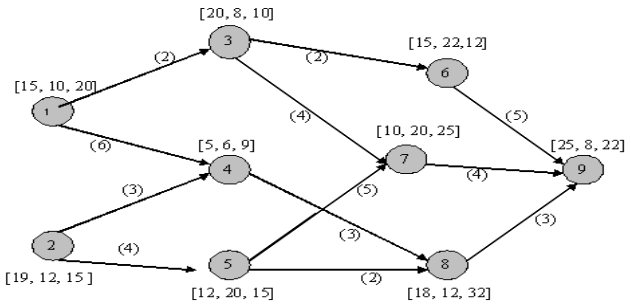

Figure 2: *Example of a multi-valued DAG*

In this example there are 9 tasks, the label [x, y, z] on task is the processing cost on the 3 servers'. For example on the task 1 we have the label [15, 10, 20], that is mean the processing time of task 1 on server σ_1 is 15, on server 2 is 10 on server 3 is 20. The table 2 indicate the complete communication delays for the problem *P*, this effective communication delay between two tasks is due to the multiplication of cost communication (data in tables 1) and volume communication data between tow tasks. For example, if task 1 is executed on server σ_1, task 3 is executed on server σ_2 the communication between tasks 1 and 3 noted $C_{1/\sigma_1,3/\sigma_2} = 2*2 = 4$, because the cost communication ($\sigma_1 \rightarrow \sigma_2$) is 2, the volume of data between task1 and task 3 is 2 also.

(1, 3)	σ_1	σ_2	σ_3
σ_1	0	4	6
σ_2	4	0	6
σ_3	6	6	0

(3, 6)	σ_1	σ_2	σ_3
σ_1	0	4	6
σ_2	4	0	6
σ_3	6	6	0

(5, 8)	σ_1	σ_2	σ_3
σ_1	0	4	6
σ_2	4	0	6
σ_3	6	6	0

(1, 4)	σ_1	σ_2	σ_3
σ_1	0	12	18
σ_2	12	0	18
σ_3	18	18	0

(3, 7)	σ_1	σ_2	σ_3
σ_1	0	8	12
σ_2	8	0	12
σ_3	12	12	0

(6, 9)	σ_1	σ_2	σ_3
σ_1	0	10	15
σ_2	10	0	15
σ_3	15	15	0

(2, 4)	σ_1	σ_2	σ_3
σ_1	0	6	9
σ_2	6	0	9
σ_3	9	9	0

(4, 8)	σ_1	σ_2	σ_3
σ_1	0	6	9
σ_2	6	0	9
σ_3	9	9	0

(7, 9)	σ_1	σ_2	σ_3
σ_1	0	8	12
σ_2	8	0	12
σ_3	12	12	0

(2, 5)	σ_1	σ_2	σ_3
σ_1	0	8	12
σ_2	8	0	12
σ_3	12	12	0

(5, 7)	σ_1	σ_2	σ_3
σ_1	0	10	15
σ_2	10	0	15
σ_3	15	15	0

(8, 9)	σ_1	σ_2	σ_3
σ_1	0	6	9
σ_2	6	0	9
σ_3	9	9	0

Table 2: *Complete communication times for the problem P*

2.3 Definition of a feasible solution

We note PRED(*i*), the set of the predecessors of task *i* in *G*:

$$\text{PRED}(i) = \{\, k / k \in I \text{ et } (k,i) \in U \,\}$$

And we note SUCC(*i*), the set of the successors of task *i* in *G*:

$$\text{SUCC}(i) = \{\, j / j \in I \text{ et } (i,j) \in U \,\}$$

A feasible solution S for the problem P is a subset of executions $\{ i/\sigma_r , i \in I \}$ with the following properties:

- each task *i* of the application is executed at least once on at least one server σ_r of Σ_i,
- to each task *i* of the application executed by a server σ_r of Σ_i, is associated one positive execution date t_{i/σ_r},
- for each execution of a task *i* on a server σ_r, such that PRED(*i*) ≠ ∅, there is at least an execution of a task *k*, $k \in$ PRED(*i*), on a server σ_p, $\sigma_p \in \Sigma_\kappa$, that can transmit its result to server σ_r before the execution date t_{i/σ_r}.

The last condition, also known as the Generalized Precedence Constraint (GPC) [5], can be expressed more formally as:

$$\forall i/\sigma_r \in S \begin{cases} t_{i/\sigma_r} \geq 0 & \text{if PRED}(i) = \varnothing \\ \forall k \in \text{PRED}(), \exists \sigma_p \in \Sigma_k \,/\, t_{i/\sigma_r} \geq t_{k/\sigma_p} + \pi_{k/\sigma_p} + c_{k/\sigma_p,i/\sigma_r} & \text{else} \end{cases}$$

It means that if a communication must be done between two scheduled tasks, there is at least one execution of the first task on a server with enough delay between the end of this task and the beginning of the second one for the communication to take place. A feasible solution *S* for the problem *P* is therefore a set of executions i/σ_r of all i tasks, $i \in I$, scheduled at their dates t_{i/σ_r}, and verifying the Generalised Precedence Constraints GPC. Note that, in a feasible solution, several servers may simultaneously or not execute the same task. This may be useful to generate less communications. All the executed tasks in this feasible solution, however, must respect the Generalized Dependence Constraints.

2.4 Optimality Condition

Let *T* be the total processing time of an application (also known as the makespan of the application) in a feasible solution *S*, with *T* defined as:

$$T = \max_{i/\sigma_r \in S} (t_{i/\sigma_r} + \pi_{i/\sigma_r})$$

A feasible solution *S** of the problem *P* modelled by a DAG $G = \{I, U, \Pi(\Sigma), \Delta(\Sigma)\}$ is optimal if its total processing time *T** is minimal. That is, it does not exist any feasible solution *S* with a total processing time *T* such that $T < T^*$.

III. PROPOSAL SOLUTION

Our algorithm has two phases: the first one is for the scheduling of primary copies where we use the DSS-OPT algorithm [15] and the second one is for the scheduling of the backup copies in which a variant of the eFRCD algorithm [16] is used.

3.1. Primary Copies Scheduling (The DSS_OPT Algorithm):

We schedule primary copies of tasks in our algorithm with DSS-OPT algorithm [15]. The DSS-OPT algorithm is an extension of PERT algorithms type. In its first phase, it computes the earliest feasible execution date of each task on every server, and in its second phase it builds a feasible solution (without fault) starting from the end of the graph with the help of the earliest dates computed in the first phase.

Let *P* be a DSS scheduling problem, and let $G = \{I, U, \Pi(\Sigma), \Delta(\Sigma)\}$ be its DAG.

One can first note that there is an optimal trivial solution to this DSS scheduling problem. In this trivial solution, all possible tasks are executed on all possible servers, and their results are then broadcasted to all other tasks that may need them on all others servers. This is an obvious waste of processing power and communication resources, however, and something better and more efficient is usually needed.

So, we now present DSS_OPT(P) algorithm's that builds an optimal solution for problem P.

DSS_OPT has two phases:
The first phase, DSS_LWB(P), computes the earliest feasible execution dates b_{i/σ_r} for all possible executions i/σ_r of each task i of problem P.

The second phase determines, for every task i that does not have any successor in P, the execution i/σ_r ending at the earliest possible date b_{i/σ_r}. If several executions of task i end at the same smallest date b_{i/σ_r}, one is chosen, arbitrarily or using other criteria of convenience, and kept in the solution. Then, for each kept execution i/σ_r that has at least one predecessor in the application, the subset L_i of the executions of its predecessors that satisfy GPC(i/σ_r) is established. This subset of executions of predecessors of i contains at least an execution of each of its predecessors in G. One execution k/σ_p of every predecessor task k of task i is chosen in the subset, arbitrarily or using other criteria of convenience, and kept in the solution. It is executed at date b_{k/σ_p}. The examination of the predecessors is pursued in a recursive manner until the studied tasks do not present any predecessors in G. The complete algorithm is the following

Input: $G = \{I, U, \Pi(\Sigma), \Delta(\Sigma)\}$
Output: A feasible solution

DSS_OPT (*P*)
1: DSS_LWB (P) // first phase
2: $T = \max_{\forall i/\mathrm{SUCC}(i)=\varnothing} \min_{\forall \sigma_r \in \Sigma_i} (r_{i/\sigma_r})$
3: for all tasks i such that SUCC(i) = ∅ // second phase
4: $L_i \leftarrow \{ i/\sigma_r / \sigma_r \in \Sigma_\iota$ and $r_{i/\sigma_r} \le T \}$
5: $i/\sigma_r \leftarrow$ keepOnefrom(L_i)
6: schedule (i/σ_r)
end DSS_OPT

DSS_LWB(*P*)
1: **For** each task i where PRED(i) = ∅ **do**
2: **for each** server σ_r such that $\sigma_r \in \Sigma_i$ **do**
3: $b_{i/\sigma_r} \leftarrow 0$
4: $r_{i/\sigma_r} \leftarrow \pi_{i/\sigma_r}$
end for
5: mark (i)
end for

6: **while** there is a non marked task i such that all its predecessors k in G are marked **do**

7: **for** each server σ_r such that $\sigma_r \in \Sigma_i$ **do**
$b_{i/\sigma_r} \leftarrow \max_{\forall k \in \mathrm{PRED}(i)} \min_{\forall \sigma_p \in \Sigma_k} (b_{k/\sigma_p} + \pi_{k/\sigma_p} + c_{k/\sigma_p, i/\sigma_r})$
9: $r_{i/\sigma_r} \leftarrow b_{i/\sigma_r} + \pi_{i/\sigma_r}$
end for

10: mark (i)
end while

end DSS_LWB(*P*)

schedule(i/σ_r)
1:execute the task i at the date b_{i/σ_r} on the server σ_r
2: **if** PRED(i) ≠ ∅ then
3: **for** each task k such that $k \in$ PRED(i) **do**
4: $L_k^{i/\sigma_r} \leftarrow \{ k/\sigma_q$ / $\sigma_p \in \Sigma_\kappa$ and
$b_{k/\sigma_p} + \pi_{k/\sigma_p} + c_{k/\sigma_p, i/\sigma_r} \le b_{i/\sigma_r} \}$
5: $k/\sigma_q \leftarrow$ keepOneFrom(L_k^{i/σ_r})
6: schedule (k/σ_q)
end for
end if
end schedule

keepOneFrom(L_i)
return an execution i/σ_r of task i in the list of the executions L_i.
end keepOneFrom.

Because the computed execution time of each task on each server is its earliest execution time on this server, and because only the copy with the ealiest ending time, of each task without any successor, is used in the solution calculated by DSS_OPT, and finally because all other copies are used only if they ensure that the final copies receives their data in time else they are not used, it follows that the feasible solution computed by DSS_OPT is optimal in execution tile. Summarizing the above discussion, we have the following theorem.

Theorem : The feasible solution calculated by DSS_OPT algorithm is optimal

Numerical example:
We consider here the problem P definite in figure 1 and 2, the DSS-OPT algorithm uses DSS_LWB to compute the earliest possible execution date of all tasks on all possible servers, resulting in the following values b and r (*Table 3)*:

1	b_1	r_1
σ_1	0	15
σ_2	0	10
σ_3	0	20
4	b_4	r_4
σ_1	18	23
σ_2	12	18
σ_3	20	29

2	b_2	r_2
σ_1	0	19
σ_2	0	12
σ_3	0	15
5	b_5	r_5
σ_1	19	31
σ_2	12	32
σ_3	15	30

3	b_3	r_3
σ_1	14	34
σ_2	10	18
σ_3	16	26
6	b_6	r_6
σ_1	22	37
σ_2	18	40
σ_3	24	36

7	b_7	r_7
σ_1	31	41
σ_2	32	52
σ_3	30	55

8	b_8	r_8
σ_1	31	49
σ_2	32	44
σ_3	30	62

9	b_9	r_9
σ_1	49	74
σ_2	49	57
σ_3	53	75

Table 3: *The earliest possible execution date of all tasks on all possible servers for the problem P*

It then computes the smallest makespan of any solution to the P problem :
$T = \max_{\forall i/\mathrm{SUCC}(i)=\varnothing} \min_{\forall \sigma_r \in \Sigma_i} (r_{i/\sigma_r}) = \min(74,57,75) = 57$

In our example, the task 9 does not have any successor. The list L_9 of the executions kept for this task in the solution is reduced therefore to the execution $9/\sigma_2$. Thus $L_9 = \{9/\sigma_2\}$. The execution of task 9 on the server σ_2 is

scheduled at date 49. Next, The tasks 6, 7 and 8 are the predecessors in G of task 9. For the task 6, the execution $6/\sigma_3$ may satisfy the Generalised Precedence Constraints relative to $9/\sigma_2$. Therefore, this execution is kept and is scheduled at date 24 (b_{6/σ_3}). For task 7, execution $7/\sigma_1$ is kept and is scheduled at date 31…

The table 4 presents the final executions i/σ_r kept by the DSS_OPT(P) algorithm, with their date of execution, in an optimal solution S.

	$1/\sigma_2$	$2/\sigma_1$	$2/\sigma_2$	$3/\sigma_2$	$4/\sigma_2$	$5/\sigma_1$	$5/\sigma_2$	$6/\sigma_3$	$7/\sigma_1$	$8/\sigma_2$	$9/\sigma_2$
b_{i/σ_r}	0	0	0	10	12	19	12	24	31	32	49
r_{i/σ_r}	10	19	12	18	18	31	32	36	41	44	57

***Table 4**: final executions i/σ_r kept by the DSS_OPT(P) algorithm*

We obtain (figure 3) the following optimal scheduling by DSS_OPT(P) algorithm:

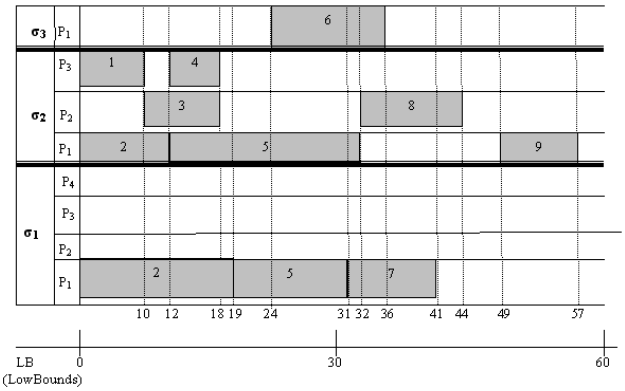

Figure 3 : DSS_OPT algorithm scheduler

3.2 Backup Copies Scheduling (eFRCD Algorithm)

We have just been building a scheduling of the primary copies, with DSS-OPT algorithm. Now, we schedule back-up copies of the tasks with the eFRCD algorithm [16], which allows us to survive failures with the hypothesis [1-failure]. Note that the eFRCD Algorithm was originally proposed to schedule tasks with deadlines in a real-time system with 1-fault hypothesis. We slightly modified it, drooping deadlines, and using earliest date of execution of the tasks and the strong copies[15] definition in ordering list priority. The modified eFRCD algorithm is detailed below.

Some notices and definitions:

- The OL (Ordered List) used by eFRCD to define priorities of tasks will be determined by earliest start execution task's dates by ascending order on all different servers where primary copies are affected with DSS-OPT algorithm scheduling.
- Backup copies are not established for tasks that have already two primary copies with DSS-OPT Algorithm scheduling even it can exist for these tasks the "backup" with good results compared to primary copies.
- **Strong copy definition**: Given a task i, i^P is a strong primary copy, if and only if only a failure of its executing processor may forbid the execution of this primary copy. Because if its executing processor is not in failure, then for each of its predecessors in the DAG whenever the failure is, there is at least one primary or backup copy able to send it its result on time. For example, on figure 3, all primary copies of tasks 1 and 2 are strong copies, because they have no predecessor. The primary copy of task 3 on server σ_2 is strong too because only a failure of σ_2 may forbid its execution. However, the primary copy of task 6 on server σ_3 will not be executed at the scheduled date if there is a failure on σ_2, as it must receive the result of task 3, and there is no other copy of task 3 available. Note that as backup copies of its predecessors are added by eFRCD algorithm, it is possible for an initially non strong primary copy to become strong.

Modified eFRCD Algorithm

Input: Scheduling established by DSS-OPT algorithm
Output: Backup copies scheduling.

Algorithm

1. order the tasks by their earliest start execution as established by DSS-OPT algorithm.
   This will generate an Ordered List noted OL;
2. SL ← Strong-copies in OL; // SL: Strong-copies List
   OL ← (OL – SL)
   while SL != ϕ do
     for each primary copy i^P of a task i in SL do
       if a backup copy i^B of task i can be established, then
3.   schedule i^B on server σ_r where it can be scheduled at low cost
       end if
     end for
4.   delete from SL tasks whose backup are scheduled
   end while
   while OL != ϕ do
     for each primary copy i^P of task i in OL do
       if a backup copy i^B of task i can be established, then
5.   schedule i^B on server σ_r where it can be scheduled at low cost
       end if
       delete from OL tasks whose backup are scheduled
     end for
   end while
End of algorithm

Numerical example:

The execution of our algorithm with the example DAG of figure 2 gives the following results:

1. Ordered List (OL) with precedence constraints by DSS-OPT: {1, 3, 4, 6, 7, 8, 9}; (tasks 2 and 5 are not considered because they have more than one primary copies).

Tasks number	1		3		4		6		7		8		9	
Candidate_servers for backup copies	1	3	1	3	1	3	1	2	2	3	1	3	1	3
Costs on servers	15	20	20	10	5	9	15	22	20	25	18	32	25	22
Low cost Server	1		3		1		1		2		1		3	

Table 5: Table of costs for backup copies on servers

2. Strong primary copies of the list SL = {1, 3, 4, 8};
3. Backup copy scheduling of tasks in list SL:
   - Backup copy scheduling of task 1: server σ_1 , bounds 0 to 15;
   - Backup copy scheduling of task 3: server σ_3, bounds 22 to 32;

- Backup copy scheduling of task 4: server σ_1, bounds from 19 to 24 considering task 2 finish date;
- Backup copy scheduling of task 8: server σ_1, bounds from 31 to 49 considering task5 and backup of task4 finish time.

4. New List OL: {6, 7, and 9}
   - 4.1. Strong primary copies of the list: {7};
   - 4.2. Backup copy scheduling of task 7: server σ_2 , schedule from 31 to 51.
5. New List OL: { 6, 9}
   - Strong primary copies of the list: {};
   - Backup copy scheduling of task 6: server σ_1, schedule from 38 to 53;
   - Backup copy scheduling of task 9: server σ_3, schedule task 9 from max(34+0, 41+12, 49+9) =58, so finish date of 9^B = 58 + 22 = 80.

The next table 6 indicates the Lower Bounds for backup copies for the problem P:

Tasks / Servers	1^B		3^B		4^B		6^B		7^B		8^B		9^B	
	s	f	s	s	s	f	s	f	s	f	s	f	s	f
σ_1	0	15			19	24	38	53			31	49		
σ_2									31	51				
σ_3			22	32									58	80

Table 6: Start dates (s) and finish (f) dates table for back-up copies

Finally, we obtain (figure 4) the following resilient scheduling in the case of one failure :

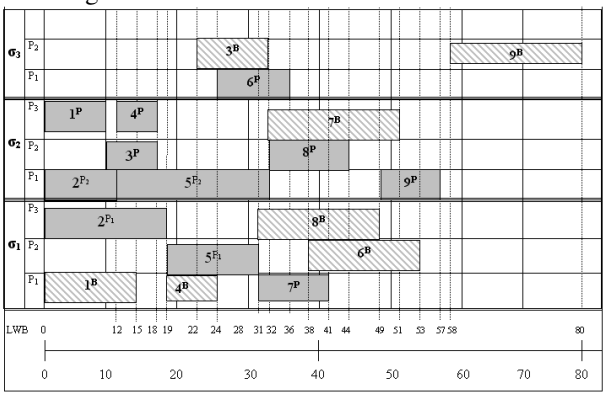

Figure 4 : Modified eFRCD algorithm scheduler

Notes on the figure 4:

- In the last figure σ_1 use 3 processing units, σ_2 use 3 processing units also, but σ_3 use only 2 processing.
- i^P is a primary copy of task i, i^{P1} is the first primary copy of task i, i^{P2} is the second primary copy of task i.
- i^B is the backup copy of task i.

IV. ALGORITHM ANALYSIS

The most computationally intensive part of DSS_OPT(*P*) is the first part DSS_LWB(*P*). In this part, for each task *i*, for each server executing *i*, for each predecessor *j* of *i*, for each server executing *j*, a small computation is done. Thus the complexity of DSS_LWB(P) is $O(n^2s^2)$, where *n* is the number of tasks in *P*, and *s* is the number of servers in DSS. Thus, the complexity of the DSS_OPT(*P*) algorithm is $O(n^2s^2)$. eFRCD is polynomial also, for the reason that eFRCD has two loops (line 3 and line 5), each loop need n*s operations, thus the complexity of eFRCD is $O(n^2s)$. Consequently the global complexity of our new algorithm is $O(n^2s^2)$.

This algorithm has two advantages:

- When there is no fault on servers, our algorithm is optimal as it uses the optimal solution computed by DSS-OPT.
- When there is a fault on one server, our solution is a good one because before the fault the solution by DSS-OPT is better than eFRCD, and after the fault it uses a solution computed by eFRCD, and eFRCD builds a good solution in this case.

Furthermore, note that our solution is guaranteed to finish if one fault occurs, because every tasks has two or more scheduled copies on different servers in the final solution. If more than one fault occur, the solution may still finish, but there is no guaranty now.

The strong copies notion of eFRCD algorithm has been considered in our algorithm because this notion gives an enhanced order of priority. A strong copy is scheduled with a high priority because it is considered as a task without precedence constraints of starting.

We do not establish backup copies for tasks which have already two primary copies with DSS-OPT Algorithm scheduling even it can exist for these tasks the "backup" with good results compared to primary copies. Related to our hypothesis “1-fault”, this option gives scheduling facilities . In our further works, we will study the case of backup copies for tasks, which have more than one primary copy.

The model of failure, as it features at most 1 crash, may seem poor. However, if the probability of any failure is very low, and the probabilities of failures independent, then the probability of two failures will be smaller indeed. Furthermore, the algorithm may be extended to 2 or more failures, by using two or more backup copies.

Finally, the problem solved by this new algorithm suffers from one strong assumption, namely that an unbounded number of tasks may be processed on each server in parallel without any effect on the tasks’ processing time. We mention two cases where this assumption may be satisfied: (a) on systems with enough parallel processors, requiring n processors per server to guarantee schedulability, or (b) constant heavy load on the servers. This limit the usefulness of the direct application of this algorithm to these few cases. However, this algorithm and its model are similar in idea and assumption to the classical Critical-Path Method (or PERT method) [19] as that CPM/PERT method do not consider resources constraints in order to get earliest execution dates. By the way these CPM/PERT results are then used in some real-life systems as the priority values of tasks in some list-scheduling algorithms, the result found by our algorithm may be used as the first step of a list scheduling algorithm, in which the earliest execution dates of primary and backup copies are used as priority values to schedule these copies on the servers of a real-life system.

V. Conclusion

In this paper, we have proposed an efficient scheduling algorithm in which tasks with precedence constraints and communication delays have to be scheduled on an heterogeneous distributed system environment with one fault hypothesis. Our algorithm has combined the DSS-OPT algorithm and the eFRCD algorithm. To provide a fault-tolerant capability, we employed primary and backup copies. But no backup copies were established for tasks which have more than one primary copy.

The result have been a schedule in polynomial time that is optimal when there is no failure, and is a good resilient schedule in the case of one failure. The one fault-tolerant capability was incorporated by using modified eFRCD and the system reliability was further enhanced by using DSS-OPT algorithm where we have optimal scheduling in case of no fault. The execution dates of the primary and backup copies may be used as priority values for list scheduling algorithm in cases of real-life, limited resources, systems.

In our future work, we intend to study the same problem with sub-networks failures. Also, we intend to consider the problem of non permanent failures of servers. Finally, we want to consider the problem of the partial failure of one server, in which one server is not completely down but loses the ability to execute some tasks and keeps the ability to execute at least one other task.

References

[1] J.-Y. Colin, P. Chrétienne (1991). "Scheduling with Small Communication Delays and Task Duplication", Operations Research, vol. 39, n o 4, 680-684, 1991.

[2] M. Maheswaran and H. J. Siegel, "A Dynamic matching and scheduling algorithm for heterogeneous computing systems", *Proceedings of the 7th IEEE Heterogeneous Computing Workshop(HCW '98), pp. 57-69, Orlando, Florida 1998.*

[3] M. Iverson, F. Özgüner, "Dynamic, Competitive Scheduling of Multible DAGs in a Distributes Heterogeneous Environment", *Proceedings of the 7th IEEE Heterogeneous Computing Workshop (HCW '98), pp. 70 – 78, Orlando, Florida 1998.*

[4] H. Topcuoglu, S. Hariri, and M.-Y. Wu., "Task scheduling algorithms for heterogeneous processors". *In8th Heterogeneous Computing Workshop (HCW' 99), pages 3–14, April 1999.*

[5] J.-Y. Colin , M. Nakechbandi, P. Colin, F. Guinand, "Scheduling Tasks with communication Delays on Multi-Levels Clusters", *PDPTA'99 : Parallel and Distributed Techniques and Application, Las Vegas, U.S.A., June 1999.*

[6] A. H. Alhusaini, V. K. Prasanna, C.S. Raghavendra, "A Unified Resource Scheduling Framework for Heterogeneous, Computing Environments", *Proceedings of the 8th IEEE Heterogeneous Computing Workshop, Puerto Rico, 1999, pp.156-166.*

[7] H. Chen, M. Maheswaran, "Distributed Dynamic Scheduling of Composite Tasks on Grid Computing Systems", *Proceedings of the 11th IEEE Heterogeneous Computing Workshop, p. 88b-98b, Fort Lauderdale, 2002.*

[8] M. Nakechbandi, J.-Y. Colin , C. Delaruelle, "Bounding the makespan of best pre-scheduling of task graphs with fixed communication delays and random execution times on a virtual distributed system", *OPODIS02, Reims, December 2002*.

[9] Christoph Ruffner, Pedro José Marrón, Kurt Rothermel, "An Enhanced Application Model for Scheduling in Grid Environments", *TR-2003-01, University of Stuttgart, Institute of Parallel and Distributed Systems (IPVS), 2003.*

[10] P. Palmerini, "On performance of data mining: from algorithms to management systems for data exploration*", PhD. Thesis: TD-2004-2, Universit`a Ca'Foscari di Venezia,2004.*

[11] Srikumar Venugopal, Rajkumar Buyya and Lyle Winton, "A Grid Task Broker for Scheduling Distributed Data-Oriented Applications on Global Grids", *Technical Report, GRIDS-TR-2004-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, February 2004.*

[12] M.J. Flynn, "Some computer organization and their e_ectiveness.", *IEEE Transations on Computer, pages 948-960, September 1972.*

[13] Yu-Kwong Kwok, and Ishfaq Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors", *ACM Computing Surveys (CSUR), 31 (4): 406 - 471, 1999.*

[14] M.R. Garey and D.S. Johnson.,"Computers and Intractability, a Guide to the Theory of NP-Completeness*W. H. Freeman Company, San Francisco, 1979.*

[15] J.-Y. Colin , M. Nakechbandi, P. Colin,"A multi-valued DAG model and an optimal PERT-like Algorithm for the Distribution of Applications on Heterogeneous, Computing Systems", *PDPTA'05, Las Vegas, Nevada, USA, June 27-30, 2005.*

[16] X. Qin and H. Jiang, "A Novel Fault-tolerant Scheduling Algorithm for Precedence Constrained Tasks in Real-Time Heterogeneous Systems" , *Parallel Computing, vol. 32, no. 5-6, pp. 331-356, June 2006.*

[17] B. Randell, "System structure for software fault-tolerance", *IEEE Trans. Software Eng. 1(2) June 1975, 220-232.*

[18] L. Chen, A. Avizienis, "N-version programming: a fault tolerant approach to reliability of software operation", *Proceeding of the IEEE Fault-Tolerant Computing Symposium, 1978 pp. 3-9.*

[19] MR Garey, David S. Johnson: Two-Processor Scheduling with Start-Times and Deadlines. SIAM J. Comput. 6(3): 416-426 (1977).