Area Failures and Reliable Distributed Applications

Moustafa Nakechbandi, Jean-Yves Colin

Area Failures and Reliable Distributed Applications

Moustafa NAKECHBANDI, Jean-Yves COLIN

ABSTRACT— **Because fault failures tend to affect whole areas, in some cases, and not only individual computers, we propose a new, efficient scheduling algorithm for problems in which tasks with precedence constraints and communication delays have to be scheduled on a virtual heterogeneous distributed multi areas system subject to the possibility of one complete area failure. Based on an extension of the Critical-Path Method CPM/PERT, our algorithm combines an optimal schedule when there is no failures, with some tasks duplication to provide fault-tolerance in the case of the failure of one area. Backup copies are not established for tasks that have already more than one original copy in different areas. The result is a schedule in polynomial time that is optimal when there is no area failure, and is a good reliable schedule in the case of any one area failure. We finally do some numerical experiments in which we use our algorithm on several semi-random DAGs and compare the optimal solutions with the reliable solutions found by this algorithm.**

KEYWORDS— DAG, scheduling with communication, heterogeneous systems, fault tolerance, catastrophic crash, area failure, reliable applications.

I. INTRODUCTION

Efficiently using heterogeneous systems is a hard problem, because the general problem of optimally scheduling tasks is NP-complete, even when there are no communication delays [8, 10]. When the application tasks can be represented by Directed Acyclic Graphs (DAGs), many static algorithms for scheduling DAGs in meta-computing systems are described in [1], [4], [10], [19]. Reliable execution of a set of tasks is usually achieved by task duplication and backup copies [3], [9], [15], [16].

A very classical and useful tool to study static scheduling problems with DAG is the Critical Path Method (also known as CPM, or PERT method, or CPM/PERT) [2]. Using a relaxation of the constraint on the number of available processors, this method gives results such as a lower bound on the execution time (or makespan) of the application and lower bounds on the execution dates of all tasks of the DAG. Because of the relaxation, tasks can be executed as soon as possible. Improvements and limits of this method to distributed systems with communications delays may be found in [4], [5], [11], for example. The study given in [6] presents the problem of scheduling the tasks of a DAG on the servers of an heterogeneous system. There, the relaxation used in CPM/PERT was replaced by the dual relaxation that each server has no constraint on the number of tasks it can simultaneously process. That is, each server can simultaneously process a non limited number of tasks without loss of performances. Our goal was to compute a lower bound on the execution time of a realistic solution, and compute lower bounds on the execution dates of all tasks of the DAG. In [12], [13], the authors suppose that one server (and at most one) could suffer from a crash fault. The algorithm presented there improved on the one presented in [6] by adding backup copies to the optimal solution build.

However, because heterogeneous systems become geographically larger and larger, they tend to be more influenced by failures that concern whole regions or areas. The failure of a simple DNS server, or an electric shortage affecting an city or region, or even a hacker attack that targets a whole country [18], is sufficient to temporarily render useless all the computing resources of an area. 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 virtual heterogeneous distributed multi-areas system subject to the possibility of one complete area failure. Based on an extension of the Critical-Path Method CPM/PERT, our algorithm combines an optimal schedule with some additional tasks duplication, to provide fault-tolerance. The result is a schedule in polynomial time that is optimal when there is no area failure, and is a good resilient schedule in the case of one area failure.

The rest of this paper is divided into four main parts. In the first one, we present the problem, and in the second one, we present our new algorithm. In the third part, we make some numerical experiments using randomly generated tasks graphs, comparing the optimal solutions with the resilient solutions found by this algorithm. Finally, in the fourth part, we discuss the advantages and disadvantages of the proposed solution.

II. THE CENTRAL PROBLEM

2.1 The Distributed Servers System

We call Distributed Servers System (DSS) a virtual set of geographically distributed, multi-users, heterogeneous or not, servers. The processing time of a task on each server of a DSS is supposedly known. It may vary from one server to another, and some tasks may not be executed on some servers.

The classical CPM/PERT relaxation on the number of processors, is replaced in the DSS problem with the dual relaxation that each server has no constraint on the number of tasks it can simultaneously process. Thus we suppose 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.

The transmission delay of a result between two tasks depends on the tasks and on their respective servers. The communication delay between two tasks executed on the same server is supposed equal to 0.

Moustafa NAKECHBANDI and Jean-Yves COLIN are from LITIS Lab. of Le Havre University, 5, rue Philippe Lebon, BP 540, 76058, Le Havre cedex, France.
e-mail: {moustafa.nakechbandi, jean-yves.colin}@univ-lehavre.fr

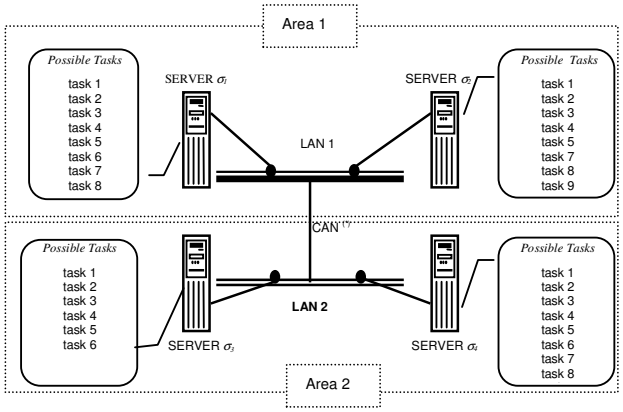


***Figure 1**: Example of Distributed Servers System with the list of the executable services for each server.*
CAN(*) : Campus Area Network.

In Figure 1, if we suppose that the CAN has a speed 1, the LAN 2 has a speed 2 and the LAN 1 has a speed 3, the following matrix gives the communication costs between the servers for one unit of data:

Network delay between $\sigma_i \rightarrow \sigma_j$	Server σ_1	Server σ_2	Server σ_3	Server σ_4
Server σ_1	0	1	3	3
Server σ_2	1	0	3	3
Server σ_3	3	3	0	2
Server σ_4	3	3	2	0

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

Thus, the total communication delay between two tasks is the amount of data from the first task to the second one, time the speed cost between their servers.

A DSS itself may be divide into a set of areas, that will be defined and used later, but that has no effects during the normal processing of an application. In Figure 1, for example, there are two areas, Area 1 and Area 2.

2.2 Directed Acyclic Graph

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 DAG.

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*, $i \in I$, 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*, $i \in$ I, toward a task *j*, $j \in$ I, is noted (i, j).
- The real communication delay, noted $c_{i/\sigma_r, j/\sigma_p}$, of the transmission of the data from *i* to *j* if task *i* is processed by server σ_r and task *j* is processed by server σ_p is a positive value that is in fact the data volume of (i, j) multiplied by the communication cost between the two servers.
- 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 $P=\{G, \Sigma\}$. Figure 2 presents an example of DAG.

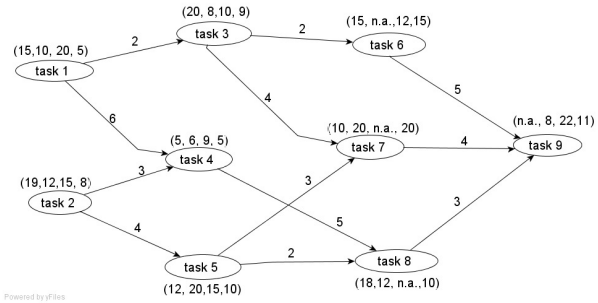


***Figure** 2 : Example of DAG*

n.a.= not allowed, i.e. cannot execute on this server

In this example there are 9 tasks. The label on each task is its processing cost on the 4 servers. For example the label Π_6 = (15, ∞, 12, 15) on task 6 means that the processing time of task 6 on server σ_1 (respectively σ_2, σ_3, σ_4) is 15 (resp. ∞, 12, 15). The label on an arc (i, j) is the data volume from i to j. For example the data volume communicated by task 1 to task 3 is 2. If task 1 is executed on server σ_1 and task 3 is executed on server σ_2, the communication between tasks 1 and 3 noted $c_{1/\sigma1,\ 3/\sigma2} = 1*2 = 2$, because the cost communication between σ1 and σ2 is 1. Also we can see that if task 1 is processed on server σ_1 and task 3 is processed on server σ_4, then $c_{1/\sigma1,\ 2/\sigma2} = 3* 2 = 6$.

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* ∈ PRED(*i*), on a server σ_p, $\sigma_p \in \Sigma_k$, 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:

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

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* ∈ *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*, *Π*(Σ), *Δ*(Σ)} 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**.

2.5. Area Failure

Finally, we now consider a DSS with possibilities of area failures. We suppose that the DSS is composed of a set of areas, noted A={Σ_1, …, Σ_z}. Each area Σ_i is a subset of servers of Σ. Each server belongs to one and only one area. For example in fig.1 we have 2 areas : Area1 = Σ_1=*{σ_1, σ_2}* and Area2 = Σ_2=*{σ_3, σ_4 }*.

One "area failure" of an area means that all servers of this area are unavailable. In our problem, only one area failure at most can occur. We call "failed area" (FA) the area, in which the area failure occurs, if it occurs. To simplify, we suppose that a failed area stay in this state until the end of the execution of the application.

A solution is "one area failure tolerant" or 1FA tolerant if at least one copy of each task of the graph is executed on at least one server outside of the failed area, and the solution is feasible. Note that, for at least one solution to be feasible if there is one area failure, it is obvious that all tasks of the application must be able to be executed on at least two servers in different areas.

III. THE DSS_1FA ALGORITHM

The algorithm proposed here, named DSS_1FA, has two phases: the first one is for the scheduling of original copies where we use the DSS-OPT algorithm [6] and the second one is for adding and scheduling additional backups copies when necessary.

3.1. Scheduling the original copies

We schedule original copies of tasks in our algorithm with the DSS-OPT algorithm [6]. The DSS-OPT algorithm is an extension of CPM/PERT algorithms type to the distributed servers problem. 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 server 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*, *Π*(Σ), *Δ*(Σ)} 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, as soon as possible, and their results are then broadcasted to all others servers. This is an obvious waste of processing power and communication resources, however, and something as optimal, but less wasteful in terms of used resources, is usually needed.

The first phase of the DSS_OPT routine, DSS_LWB(), goes from the initial tasks to the final ones, computing along the way the earliest feasible execution dates b_{i/σ_r} and earliest end date η_{i/σ_r}, for all possible executions i/σ_r of each task *i* of problem *P*.

The second phase of the DSS_OPT routine determines, for every task i that does not have any successor in *G*, i.e. task *i* is a "leaf" or final task, the execution i/σ_r ending at the earliest possible date η_{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 its earliest possible 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.

3.2. Adding backup copies

The ADD_BACKUP_COPIES routine starts from tasks without any predecessors, similarly to DSS_LWB(), and proceed from there to the end of the DAG. First, if there is currently only one copy of a given task, it determines what is the worst possible delay it may encounter if a failure occurs on another server, while satisfying its GPC. It also determines the fastest server (not considering the server executing the only current copy of this task in the current solution) able to execute this task, and adds a backup copy on this server to the solution, again considering the worst

possible delay resulting from this failure, while satisfying the GPC of this copy. Else the task has already several copies in the optimal solution, and the routine determines for each original copy of this task, what is the worst possible delay it may encounter if a failure occurs on another server, while satisfying its GPC.

3.3. DSS_1_AREA_FAILURE algorithm

The complete DSS_1_AREA_FAILURE algorithm is the following:

Input: $G = \{I, U, \Pi(\Sigma), \Delta(\Sigma)\}$
Output: A feasible solution with backup copies
DSS_1FA ()
  DSS_OPT() // first phase
  ADD_BACKUP_COPIES_1FA() // second phase
end DSS_1FA
DSS_OPT()
  DSS_LWB ()

$$T = \max_{\forall i/\mathrm{SUCC}(i)=\emptyset} \min_{\forall \sigma_r \in \Sigma_i} (r_{i/\sigma_r})$$

  for all tasks i such that SUCC(i) = ∅ **do**

$$L_i \leftarrow \{\, i/\sigma_r \,/\, \sigma_r \in \Sigma_i \text{ and } r_{i/\sigma_r} \le T \,\}$$

$$i/\sigma_r \leftarrow \text{keepOnefrom}(L_i)$$

    schedule (i/σ_r)
  end for
end DSS_OPT
DSS_LWB()
  for each task i where PRED(i) = ∅ **do**
    for each server σ_r such that $\sigma_r \in \Sigma_i$ **do**

$$b_{i/\sigma_r} \leftarrow 0$$

$$r_{i/\sigma_r} \leftarrow \pi_{i/\sigma_r}$$

      end for
    mark (i)
    end for
  while there is a non marked task i such that
      all its predecessors k in G are marked **do**
    for each server σ_r such that $\sigma_r \in \Sigma_i$ **do**

$$b_{i/\sigma_r} \leftarrow \max_{\forall k \in \mathrm{PRED}(i)} \left(\min_{\forall \sigma_p \in \Sigma_k} (b_{k/\sigma_p} + \pi_{k/\sigma_p} + c_{k/\sigma_p, i/\sigma_r}) \right)$$

$$r_{i/\sigma_r} \leftarrow b_{i/\sigma_r} + \pi_{i/\sigma_r}$$

    end for
    mark (i)
  end while
end DSS_LWB
schedule(i/σ_r)

  execute the task i at the date b_{i/σ_r} on the server σ_r

  if PRED(i) ≠ ∅ **then**
    for each task k such that $k \in$ PRED(i) **do**

$$L_k^{i/\sigma_r} \leftarrow \{\, k/\sigma_q \;/\; \sigma_p \in \Sigma_\kappa \text{ and } b_{k/\sigma_p} + \pi_{k/\sigma_p} + c_{k/\sigma_p, i/\sigma_r} \le b_{i/\sigma_r} \,\}$$

$$k/\sigma_q \leftarrow \text{keepOneFrom}(L_k^{i/\sigma_r})$$

      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.

ADD_BACKUP_COPIES()
  for each task i such that PRED(i) = ∅ **do**
    if i has only one copy scheduled
        or all copies of i are on servers in the same area
    then
      Let σ_i be the server executing a copy of i
      Let α_i be the area such that $\sigma_i \in \alpha_i$.
      // compute one backup on the fastest server left
      // outside the area α_i of σ_i, if α_i is the failed area
      Let $\sigma_r \notin \alpha_i$ be the fastest server able to execute task i
      Execute a new backup copy of i on σ_r at date 0
    end if
    mark (i)
  end for
  while there is a non marked task i such that all its
      predecessors k in G are marked **do**
    if i has only one copy scheduled
        or all copies of i are on servers in the same area
    then
      Let σ_i be the server executing the copy of i
      Let α_i be the area such that $\sigma_i \in \alpha_i$.
      // First compute the delayed execution date of // task i on this
      // server, if the failure is on an another area
      find the delayed execution date of the copy of i on σ_i
        taking only into account the delayed execution dates of the
        copies and backups of each predecessor of i to verify the GPC
      // Second compute one backup copy on the fastest server left
      // outside area α_i, if α_i is the failed area
      Let $\sigma_r \notin \alpha_i$ be the fastest server able to execute i
      Execute a backup copy of i on σ_r taking only into account the
        delayed execution dates of the copies and backups of each
        predecessor of i to verify the GPC
    else // i has at least two copies scheduled, on servers in separate areas.
        // compute the delayed execution date of the copy of task i on
        // each server, if the failure is on another area
      for each server σ_i executing a copy of i **do**
       Find the delayed execution date of the copy of i on σ_i taking only
        into account the delayed execution dates of the copies and
        backups of each predecessor of i to verify the GPC
      end do
    end if
    mark (i)
  end while
end ADD_BACKUP_COPIES

3.4. 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 2*):

1	b_1	r_1
σ_1	0	15
σ_2	0	10
σ_3	0	20
σ_4	0	5

2	b_2	r_2
σ_1	0	19
σ_2	0	12
σ_3	0	15
σ_4	0	8

3	b_3	r_3
σ_1	11	31
σ_2	11	18
σ_3	9	19
σ_4	5	14

4	b_4	r_4
σ_1	16	21
σ_2	12	18
σ_3	15	24
σ_4	18	23

5	b_5	r_5
σ_1	16	28
σ_2	12	32
σ_3	15	24
σ_4	18	28

6	b_6	r_6
σ_1	20	35
σ_2	∞	∞
σ_3	18	30
σ_4	14	29

7	b_7	r_7
σ_1	28	38
σ_2	30	50
σ_3	∞	∞
σ_4	28	48

8	b_8	r_8
σ_1	28	46
σ_2	30	42
σ_3	∞	∞
σ_4	28	38

9	b_9	r_9
σ_1	∞	∞
σ_2	44	52
σ_3	∞	∞
σ_4	48	56

Table 2: *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 :

$$T = \max_{\forall i/\mathrm{SUCC}(i)=\emptyset} \min_{\forall \sigma_r \in \Sigma_i} (r_{i/\sigma_r}) = \min(\infty, 52, \infty, 56) = 52$$

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 44. Next, The tasks 6, 7 and 8 are the predecessors of task

9. For the task 6, the execution $6/\sigma_4$ may satisfy the Generalised Precedence Constraints relative to $9/\sigma_2$. Therefore, this execution is kept and is scheduled at date 14 (b_{6/σ_4}). For task 7, execution $7/\sigma_1$ is kept and is scheduled at date 28…, the table 3 presents the final executions i/σ_r kept by the DSS_OPT(P) algorithm, with their date of execution, in an optimal solution S.

	$1/\sigma_2$	$1/\sigma_4$	$2/\sigma_2$	$3/\sigma_4$	$4/\sigma_2$	$5/\sigma_1$	$6/\sigma_4$	$7/\sigma_1$	$8/\sigma_2$	$9/\sigma_2$
b_{i/σ_r}	0	0	0	5	12	16	14	28	30	44
r_{i/σ_r}	10	5	12	14	18	28	29	38	42	50

***Table 3**: 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:

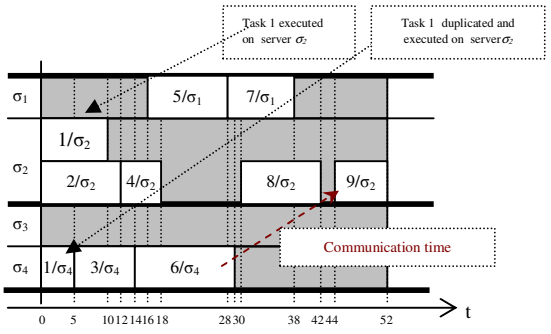


***Figure 3**: DSS_OPT algorithm scheduler*

By adding backup copies using ADD_BACKUP_COPIES we get the following fault-tolerance scheduling (Figure 4.):

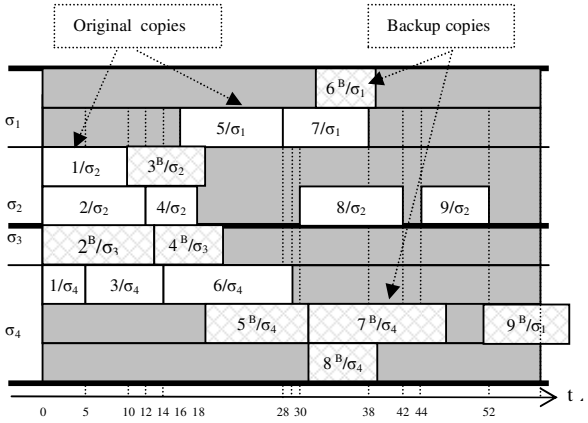


***Figure 4**: Gantt chart given by DSS_1_AREA_FAILURE*

Now we express some proprities on the results found by the proposed algorithm.

Lemma 1: The feasible solution S calculated by the DSS_OPT algorithm is optimal if there is no area failure.
Proof: Because all copies of tasks with at least one successor are scheduled in S only if they ensure, directly or indirectly, that the final copies receives their data in time in the solution, else are not used, it follows that the global makespan of the solution S is the maximal ending date of the copies of the tasks without any successors.
Because only the copy with the earliest ending date of each task without any successor, is used in the solution S, it follows that no possible solution may execute one task without any successor that will end at an earliest date that the one in solution S.
Thus the feasible solution S computed by DSS_OPT is optimal in execution time for the problem without area failure. QED
Theorem 1: The solution calculated by DSS_1FAULT is optimal if there is no area failure.

Proof: Because the copies in the DSS_1FAULT solution come and only come from the DSS_OPT solution, they all will be executed at the same dates if there is no area failure. Because of this and of Lemme 1, it then follows that the solution calculated by DSS_1FAULT is optimal if there is no area failure. QED

Also, in the final solution computed by DSS_1FA(), each task of the DAG has at least two copies (coming from the DSS_OPT() routine), or one copy (coming from the DSS_OPT() routine) and one backup copy (build by the ADD_BACKUP_COPY_1FA() routine) , always executed on different servers.

Furthermore, the execution date of each backup copy and the delayed execution date of each original copy coming from DSS_OPT is always evaluated by ADD_BACKUP_COPIES_1FA() taking into account the delayed execution dates of the copies and the execution dates of the backups copies of each predecessor, using the worst possible case of failure of a predecessor, we have:
Theorem 2: The solution calculated by DSS_1FA is *feasible if there is at most one area failure.*

Also, Let α be the area that contains the servers failures. Because the solution S is feasible when all the servers of one area are unavailable, this solution is also feasible if only one or several servers of area α are unavailable, and if all servers of all others areas are available. Thus:
Theorem 3: Let S be the solution created by DSS_1FA. This solution S is also fault tolerant to the failure of one or several servers, if all servers failures occur in the same area.

The most computationally intensive part of DSS_OPT() is the first part DSS_LWB(). 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() 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() algorithm is $O(n^2s^2)$.

Similarly, in ADD_BACKUP_COPIES_1FA(), for each task *i*, for each copy of task *i* (at most one copy per server), for each predecessor *j* of *i*, for each copy of *j* (at most one per server), one small computation is done. Thus the complexity of ADD_BACKUP_COPIES_1FA() is bounded by $O(n^2s^2)$, where *n* is the number of tasks in *P*, and *s* is the number of servers in DSS. Thus we have:
Theorem 4: The complexity of the DSS_1FAULT algorithm is $O(n^2s^2)$.

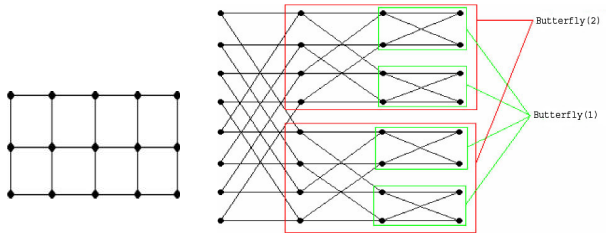
IV. NUMERICAL EXPERIMENTS

4.1 Random graph generator

To evaluate DSS_1FA, we have compared the fault tolerant solutions it generated on some classical problems and DAGs to optimal solutions without fault tolerancy. In our study a semi-random graph generator was implemented

to generate weighted application DAGs with various characteristics. This framework first executes the random graph generator program to construct the application DAGs, which is followed by the execution of the our scheduling algorithms to generate output schedules. We consider two kinds of graphs. The first one is a regular simple two-dimensional grid DAG (see Figure 5. a.), exhibited by the numerical applications, with lot of parallelism and very local communications. The second is the "butterfly" DAG (see Figure 5. b.) present in applications such as the FFT or shuffle algorithms, again with lot of parallelism, but a more complex communication pattern.

The servers performances are independent random values for each task of the DAG, and so are the communication delays. The processing time of a task is a random value generated between 10 and 30. The communication delay between the tasks is also a random value generated between 1 and 10.



a. 2-Dimensional grid DAGs (3 lines, 5 columns)

b. 3-dimensional butterfly DAGs

Figure 5: *Two different kind of graphs*

4.2. Performance Results

In Figure 6 the DAG used is the 2-Dimensional grid DAGs. This kind of graph needs two parameters: the number of lines n and the number of columns m.. Thus a nm-grid graph has $n*m$ vertices. Here the chosen parameters are: (20,15), (20,20), (25,25), (30,25), (30,30), and (40,30), which correspond, respectively to 300, 400, 625, 750, 900, and 1200 tasks.

The Figure 7 uses the butterfly DAGs. This kind of graph needs only one parameters: the butterfly degree n. An n-dimensional butterfly graph has $2^n(n+1)$ vertices. The chosen degrees in this numerical tests are: 4, 5, 6, 7, and 8, which correspond, respectively to 80, 192, 448, 1024, and 2304 tasks.

In all our simulations, we fixed the number of servers to 12 and the number of areas to 3 and each makespan average is computed over 20 random DAGs.

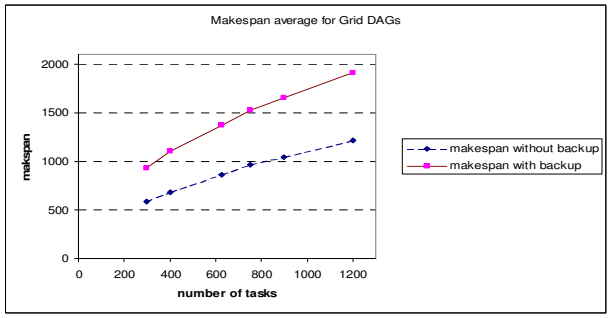


Figure 6 *: Makespan average for 2-Dimensional grid DAGs*

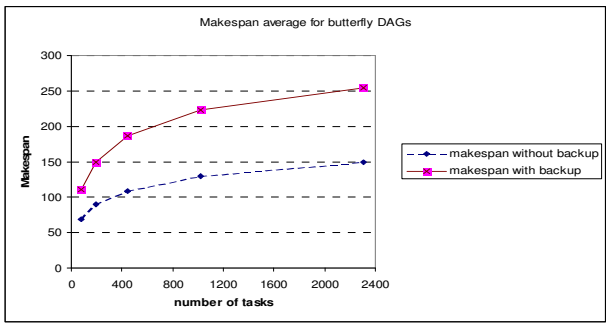


Figure : Makespan average for butterfly DAGs

In both kinds of DAGs (Figure 6 and Figure 7), it is found that the makespan average with backup copies is between 1.5 (usually) and 2 (at most) times the makespan without backup copies.

We got similar results when varying a little the number of servers and number of areas.

Other experiments with totally random graphs and with fork-join graphs yielded similar results, so they are not presented here.

V. ANALYSIS

The model of failure, as it features at most one area failure, may seem limiting. However, if the probability of any area failure is very low, and the probabilities of area failure are independent, then the probability of two failures will be much smaller indeed.

Also, the solution solved by this new algorithm uses the classical CPM/PERT relaxation, namely that an unbounded number of tasks may be processed on each server in parallel without any effect on the tasks' processing time, in the same way the classical CPM/PERT method do not consider resources constraints in order to get earliest execution dates and detect critical paths. This relaxation is not far from the reality, if each server is a multiprocessors architecture for example. Or if each server is a time-shared, multi-users system with a permanent heavy load coming from other applications, and the tasks of an application on each server represent a negligible additional load. Furthermore, even if the above conditions are not met by the real distributed system targeted, the results 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. In the same way these CPM/PERT results are used in some real-life systems as the priority values of tasks in some list-scheduling algorithms for real shared-memory or distributed architectures.

This algorithm has two main advantages:

- when there is no area failure, the DSS_1FA's solution is optimal because it uses the optimal solution computed by DSS-OPT.
- when there is one area failure, the DSS_1FA's solution is certain to finish correctly, because every tasks has two or more scheduled copies on different servers in different areas in the final solution. If more than one area failure occur, the solution may still finish, but there is no guaranty there.

Note also that the solution built gives indications on the sensibility of an application to one area failure when compared to the solution without any area failure, because the makespan in the presence of one area failure is a worst case analysis.

Not considering the areas, one can note that the solution built has fault tolerance to the failure of one individual server. Furthermore, the solution has fault tolerance to the failure of several individual servers, provided that the failed servers are all in the same area.

Another benefit of our algorithm is in using the following idea: suppose that we know that some servers are very likely to have a server failure, for some reason. Even if they are not formally in the same area, it may be worthwhile to group them in a new specific artificial area, made of real areas, to insure that the solution built is able to survive failures of any number of these servers, by using backups outside this artificial area.

VI. CONCLUSION AND FUTURE WORKS

In this paper, we have proposed a polynomial 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. 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 gives earliest execution dates to copies of tasks when there is no failure, and is a good resilient schedule in the case of one failure. Performance evaluation on some DAGs gave an increase in case of one server failure in makespan of 1.5 to 2 times the optimal makespan without server failure.

The execution dates of the original and backup copies may be used as priority values for list scheduling algorithm in cases of real-life, limited resources, and 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] 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, pp.156-166, 1999.

[2] R.E. Bellman. “Dynamic Programming”. *Princeton University Press, Princeton, New Jersey,* 1957.

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

[4] J.-Y. Colin, P. Chrétienne "Scheduling with Small Communication Delays and Task Duplication", *Operations Research,* vol. 39, n o 4, 680684, 1991.

[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.. 1999.

[6] 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, pp. 876-882, 2005.

[7] M.J. Flynn. “Some computer organization and their effectiveness.”, *IEEE Transactions on Computer,* pp. 948-960, September, 1972.

[8] 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.

[9] A. Girault, H. Kalla, and Y. Sorel. J, “A scheduling heuristics for distributed real-time embedded systems tolerant to processor and communication media failures”. *International Journal of Production Research*, 42(14):2877-2898, 2004.

[10] 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.

[11] 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; pp. 225-233, 2002.

[12] M. Nakechbandi, J.-Y. Colin, J.B. Gashumba, "An efficient fault-tolerant scheduling algorithm for precedence constrained tasks in heterogeneous distributed systems"; *CIS2E06 International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering*, 2006. Published in : Innovations & advanced techniques in computer & information sciences & engineering, Springer, 06-2007, pp 301-307, 2007.

[13] M. Nakechbandi, J.-Y. Colin, "An Algorithm and Some Numerical Experiments for the Scheduling of Tasks with Fault-Tolerancy Constraints on Heterogeneous Systems" ; *Workshop on Optimization Issues in Grid and Parallel Computing Environments in HPCS.08*, pp 326-332, Nicosia, Cyprus, 2008.

[14] 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.

[15] 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, 2006.

[16] B. Randell, “System structure for software fault-tolerance”, *IEEE Trans. Software Eng.* 1(2,) pp. 220-232, 1975.

[17] Ch. 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.

[18] A. Saidane, V. Nicomette, and Y. Deswarte, "The Design of a Generic Intrusion-TolerantI Architecture for Web Servers*", IEEE Transactions on dependable and secure computing*, Vol. 6, NO. 1, January-march, 2009.

[19] H. Topcuoglu, S. Hariri, and M.-Y. Wu, “Task scheduling algorithms for heterogeneous processors”. *In 8th Heterogeneous Computing Workshop (HCW’ 99),* pp. 3–14, 1999.

AUTHOR BIOGRAPHIES

Moustafa NAKECHBANDI is Associate Professor at the University of Le Havre, France. He received the "Doctorat de 3ème cycle" in 1979 and the "Doctorat d'Etat" in 1984, both from Besançon University (France). His research interests are the optimization problems relative to parallel computing and the fault-tolerant scheduling in parallel programs.

Jean-Yves COLIN is Assistant Professor at the University of Le Havre, France. He received a Ph.D (1989) in Computer Science from Paris 6 University. His research interests include scheduling in heterogeneous distributed systems, and optimization of parallel programs.