UNE IMP

EHE CWI

★QMIPS★

INR UER

ZAR UTO

Quantitative
Modeling
In
Parallel
Systems

Characterization, Approximate Analysis and Simulation of Task Graph Models Representing the Dependable Execution of Parallel and Distributed Programs

Erol Gelenbe
Department of Electrical Engineering
Duke University
Durham, N.C. 27708-0291, USA

Sophie Chabridon
EHEI
Université René Descartes
45 rue des Saints-Pères
75006 Paris, France

D W3.T1.17-1.v1

29 September 1994

Distribution Level: All partners

Approved by: E. Gelenbe

The QMIPS Project: The QMIPS project is a collaborative research project supported by the CEC as ESPRIT-BRA project no 7269. It is being carried out by the following organisations: CWI (Amsterdam), EHEI (University of Paris V), Imperial College (London), INRIA (Sophia-Antipolis), University of Erlangen, University of Newcastle, University of Torino and University of Zaragoza.

Abstract In this paper, we use task graph models to represent the behaviour of parallel programs. These models are characterized by execution times of the tasks, and by the precedence relation between the tasks. The latter can be represented by a probabilistic ordering, or can be provided with a specific known ordering for a given application. When failures occur in the processing system, we consider a recovery mechanism based on failure detection, and subsequent task restart. Both of these operations take additional processing times which are explicitly represented in the task graph characterization. Failures themselves are represented in the model by variable failure rates. We report on the design, analysis and simulation of novel algorithms which will ensure that application software runs correctly on an MIMD system in which processing units (PU) may fail. These algorithms are based on certain existing tasks which are selected within the program, which we call *agents*. Their role is to carry out failure detection and if necessary restart of other tasks, as soon as they have completed their own specific assigned work. The effect of these algorithms is evaluated using analytical approximations and simulation as a function of failure rates, and other system parameters. The comparison of the simulation results with the approximate analytical results, shows a very good level of accuracy for this degree of complexity, which indicates that simple analytical formulae can be used to obtain robust first-order estimates of program execution times with and without failures. We also provide specific examples of task graphs for two well known computations (matrix multiplication and the Fast Fourier Transform) and their parallel implementation. Finally we provide simulation results which evaluate the proposed failure detection and recovery algorithms for the specific case of the FFT algorithm.

1 Introduction

Fault-tolerance and reliability have always been of essential importance to designers and users of computers. With the increasing variety of parallel systems, from tightly coupled shared/distributed memory SIMD and MIMD architectures to loosely coupled "farms" of workstations, these concerns assume even greater significance.

Parallel and distributed systems, because of their size and complexity, are more prone than sequential machines to incidents which impair program execution. In addition to conventional failures which can occur in such systems, specific events may occur which can cause a particular application program's execution to fail. For instance: certain processing units (PU's) may be pre-empted unexpectedly by other programs, certain processing units may be turned off by users who are unaware of the ongoing application, and workload surges from higher priority applications may slow down a particular application's work on certain PUs, creating excessive delay for some tasks, which can then be interpreted as a failure. In some cases, such surges are known to eliminate certain processes from memory.

Task graphs are convenient models of the execution of parallel programs [1, 2] and have been used widely to estimate the performance of parallel architectures and systems. They will be used in this paper as the basic tool for modeling the workload in a parallel system with failures. As such, we can view this paper as a continuation of our previous work based on task graph characterization, in the direction of systems with failures.

For sequential processing, a variety of techniques such as log-based recovery and recovery block mechanisms, have been suggested and anlyzed [3, 4, 5, 6]. In recent research [7, 8, 9, 10] new techniques have been developed which guarantee that a parallel program of the fork-join or threads and barriers type can be modified so that processor failures can be detected, and processing rescheduled when processor failures occur. In order to illustrate the <u>need</u> for the failure detection and recovery algorithms in distributed programs, we first present simulations of parallel programs running on a MIMD architecture with failures. We assume that the machine running the program has more processors than are needed at any time by the program, so that failures are the only limiting factor on performance. The program is composed of a set of interdependent tasks, and when a PU on which a task is running fails, then the corresponding task is disabled. When a PU fails it remains unavailable for some time F, after which it becomes operational again (it is "repaired"). We consider two alternatives concerning task restart after the PU is again operational:

- (1) In the case with <u>automatic restart</u>, the task which was running on the failed PU is automatically restarted when the PU becomes operational again. Consequently, even without failure detection, tasks are restarted and the program as a whole has a chance to complete. The program's time to completion will depend on the failure rate and on the down time F.

- (2) If there is <u>no automatic restart</u> then – since we do not have failure detection – the program as a whole will have failed when any processor fails.

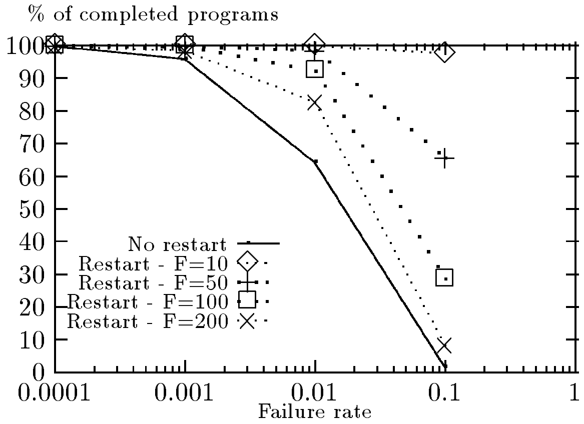

Figure 1 - Effects of failures without detection.

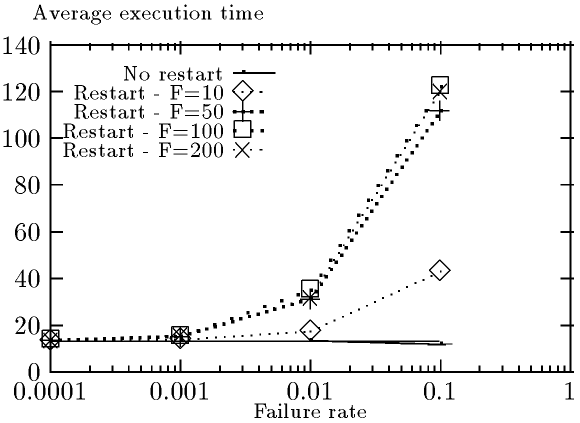

Figure 2 - Average execution time of completed programs.

On Figure 1 we plot results from simulations we have carried out; we will not discuss the simulation model used in detail, since our purpose here is merely to motivate the results presented in the following Sections. These simulations deal with the percentage of programs which complete successfully, against the failure rate γ, for 1000 simulated programs selected at random. In order to normalize the time units involved, each individual task of a program has a unit (1) average execution time. The failure rate γ ranges from 0.0001 to 0.1 in logarithmic scale. Each simulated program contains $K = 50$ tasks. In order to concentrate the analysis on the reliability issues, these simulations assume that the number of PUs is unlimited and that communications times between tasks are small enough to be neglected. A program is stopped if it does not complete after a time $T_{max} = 200$. The results shown concern the case when there is no restart, and with automatic restart for four values of F: $10, 50, 100, 200$. In the case with restart, the percentage of completed programs decreases significantly when $F = 200$; for $\gamma = 0.1$, less than 10 % of the programs complete. When tasks are not restarted automatically, the percentage of completed programs decreases quickly as the failure rate increases, and almost no programs complete when $\gamma = 0.1$. Figure 2 shows the average execution time of the *successfully completed programs*, in the presence of failures and without the benefit of detection algorithms. Without restart, programs which complete in the presence of failures are simply those for which no PUs fail during their execution. Shorter programs are more likely to avoid failures: average execution time of completed programs decreases from 13.17 when $\gamma = 0.0001$ to 11.6 when $\gamma = 0.1$. With automatic restart the programs' average execution time increases rapidly as a function of the failure rate, attaining more than 10 times their original value when $\gamma = 0.1$. These simulations strongly motivate the need for better methods for handling failures, which will detect failures and reschedule tasks rapidly.

2 Detection and Recovery Algorithms

A distributed application is composed of many interdependent tasks, and each task is a granule of sequential code which is executed on a single PU. Tasks are related to each other by precedence constraints, by the exchange of data, or of messages, and by the transfer of control. Whenever a task successfully completes, it sends a message to its successor tasks, which requires a *perfectly reliable communication medium.* Another alternative, is that the task places information in shared memory, to indicate that it has completed and that it thus enables other tasks' execution; this assumes that there is a *small failsafe shared memory* which any task can interrogate. Throughout this study, we will assume that either one of these two options is available for tasks to communicate reliably.

We propose to select a subset of the tasks in the program, and make them responsible for executing the monitoring – i.e. failure detection and task reallocation – algorithm. These selected tasks will complete the computation they would carry out normally, and then actively detect failures, and assist the rest of the program's recovery from failures. We will call these selected tasks *agents*. Clearly, an agent can only examine its "sibling or successor tasks" for failure, i.e. those for which it is not a successor. The schemes we propose will guarantee full dependability only if at least one of the In particular we will consider the following alternatives for chosing the agents:

- **Alternative 1** – Here only tasks which have no successor (i.e. the "leaves" of the task graph) are agents. Each task which has finished execution and which has no successor in the task graph will first examine the other leaves of the task graph. An agent will not examine a task which has been – or is being – examined by another agent. If one of the leaves of a task is shown not to have completed, the agent will "investigate" it by examining its predecessors until all tasks are accounted for either as "not yet started", "running but unfinished" or "failed". The agent will then reassign the failed task which it has detected (if it finds one) to a PU which is up, and then the task will be restarted.

  This will work very well when failures are infrequent, since it introduces little overhead. When failures are more frequent, this approach may delay detection for too long; in fact, it may delay it "for ever" since none of the leaves may be reached.

- **Alternative 2** – Here we choose a certain number of intermediate tasks also to be agents, in addition to the leaves. The purpose is to detect failures soon enough to avoid that the distributed application fails completely before the leaves are reached. In most cases, any failure will be detected by the intermediate tasks without having to wait for leaves to be activated.

  Let the "rank" of a task be the longest path from itself back to a source task (a source being a task without predecessors). We may select some ranks, and assign the role of agent to some tasks in each rank. An agent will examine other tasks having its own rank, and then those of lesser rank, in order to detect failures. We call this "look-back detection".

- **Alternative 3** – This is an extension of Alternative 2. Here, agents will carry out "look-ahead" for failures, in addition to the "look-back" of Alternative 2. When an agent finds a task of the same rank which has completed its execution, it will test its successors. The purpose is to detect failures as soon as possible and to favor the rapid successful completion of the whole program.

Three options can be combined with Alternatives 1,2 and 3.

- **Active Wait Option (W)** – Detection can be improved by charging the leaf task which completes earliest, with carrying out failure detection periodically until the whole program completes. In fact, this can be extended to all completed leaf tasks.
- **Restart Option (R)** – The purpose here is to protect from complete program failure. Thus if a program does not complete execution by some predetrmined date T_{max}, it is restarted from scratch. Clearly a small value of T_{max} will lead to "false alarms", while a very large value will lead to excessive waits when the program has failed.
- **W+R Option** – Combining the two previous options, we obtain a scheme in which some or all leaf tasks are kept alive as failure detectors, and in addition the program as a whole is restarted after a time T_{max}.

The assumptions concerning the manner in which failures may occur, as well as their nature (intermittent or permanent) are important. We consider the following relevant failure assumptions:

- *Failure Assumption A (FAA)* – Tasks cannot autonomously restart. Any task whose processor *has not failed* by the end of its computational step will not subsequently fail until all of the program terminates. Thus any task which carries out the role of "agent" is preserved from failures. This is clearly a "best case" assumption, and would be valid if the agents' detection and restart functions were carried out on highly reliable special PUs.
- *Failure Assumption B (FAB)* – Tasks cannot autonomously restart. Agents which have terminated their normal work, and which are now carrying out failure detection and restart for other tasks, can fail as well. This assumption is more realistic.
- *Failure Assumption C (FAC)* – Under failure assumption C, tasks which fail can restart autonomously without the help of agents, simply because when the corresponding PU is repaired the failed task immediately restarts autonomously. Note that restarts can also be caused by agents and agents themselves be restarted after a failure. This is the most complex scenario.

Failure Assumption A is overly optimistic with respect to the fact that agents are protected from failure, while Assumption C is perhaps too complex to be implemented. Failure Assumption B, where agents can also fail, and tasks cannot restart autonomously, describes a realistic operating condition.

3 Approximate Analysis and Simulations

We simulate the following model of a set of parallel programs running on a distributed architecture. Each program is represented by a *Task-graph*; these models have been successfully used to represent parallel programs and to study their performance [1, 2, 30, 31]. In the task-graph model, a program is represented by a set of K tasks numbered $\{1, \ldots, \mathrm{K}\}$: each task is an indivisible unit of code and data which has to be executed sequentially. An acyclic graph G with K nodes represents the precedence relations between the tasks. The graph is acyclic because we are dealing with a computation which must eventually terminate. Each task i has an execution time t_i on a PU, and it may be executed on any one of the PUs in the same amount of time. When a task finishes its execution it informs all of its successors in the precedence graph, either by sending a message or through a shared memory. A task can begin execution when it is informed that all of its predecessors have terminated their execution.

In the simulations, each task graph representing a program, is generated at random. A probability p will be selected, and used to determine whether some task j, with $j > i$ is a successor of task i. By not allowing a task j to be a successor of some task $k < j$, we will guarantee that the graph is acyclic. By choosing a small value of p, we will simulate programs which are very "parallel" since any task will have few successors, while a larger value of p will limit the amount of parallelism in the program. For the sake of simplicity, the failure process is assumed to be Poisson, and all and durations are exponentially distributed unless otherwise stated.

The simulations have to account for the time spent by an agent both in detection, and in failure recovery: each time a task is tested by an agent, a constant time w is added to the execution time of the agent. When a task is restarted after a failure is detected, it takes an additional constant time C for the agent to do so.

3.1 Approximate Analytical Results

Let us present a simple analysis of the main trade-offs of "agent based" recovery, using the task-graph model of parallel programs. As before, the average task execution time is normakized to be 1. The analysis will be based on results reported in [1, 2].

The smallest or best possible execution time of a parallel program E is obtained when the program has access to an unlimited number of processing units, assuming that communication times can be neglected. It has been shown [2] that the smallest possible execution time of a parallel program represented by a task graph is given approximately by the following expression:

$$E = \frac{2Kp}{p+1}. \tag{1}$$

Extrapolating from this formula, the average execution time of a program with failure detection, but without any failures, is approximately:

$$E_d = \frac{2Kp(1 + \pi D)}{p+1}, \tag{2}$$

where π is the probability that a particular task is an agent, and D is the time that each agent spends doing failure detection. Since all tasks, including agents, will have to be tested and because there are πK agents, each agent spends on the average a time:

$$D = \frac{wK}{\pi K} = \frac{w}{\pi} \tag{3}$$

so that the average program execution time without failures but with failure detections is obtained as:

$$E_d = \frac{2Kp(1+w)}{p+1}. \qquad (4)$$

This is identical to the formula we would have obtained if we had added the time spent in failure detection to each task's execution time.

Let us now consider the possibility of failures, and let f be the probability that a failure occurs during a tasks execution. Then the total average task execution time X becomes:

$$X = 1 + D + Xf + Rf, \qquad or \qquad X = \frac{1+D+Rf}{1-f} \qquad (5)$$

where D is the failure detection time, and R is the time it takes the agent which detects the failure to reschedule the task. This is because with probability $(1-f)$, $X = 1+D$, while with probability f we have $X = 1+D+X+R$, since failures can repeat in the future even after a task has been rescheduled. Assuming agents also can fail, detection and restart can only be accomplished by non-failed agents, so:

$$D = \frac{w}{\pi(1-f)}, \qquad and \qquad R = \frac{C}{\pi(1-f)} \qquad (6)$$

Thus the total average program execution time in the presence of failures E_F is estimated as:

$$E_F = \frac{2Kp[1 + \frac{w}{\pi(1-f)} + \frac{C}{\pi(1-f)}]}{(1-f)(p+1)}. \qquad (7)$$

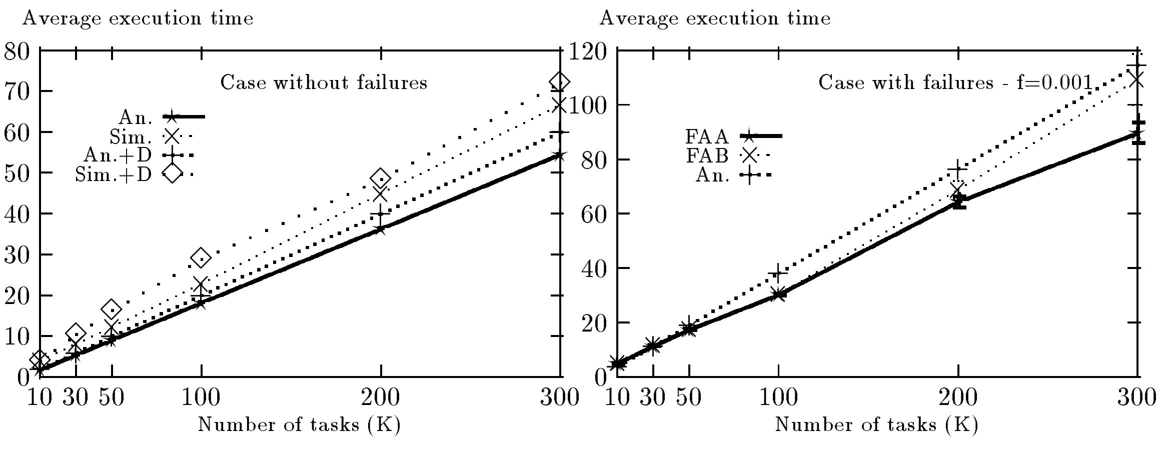

Figure 3 - Comparison of analytical and simulation results.

We compare these approximate formulae with simulation results on Figure 3 for values of K in the range 10 to 300, p = 0.1, π=1 and f = 10^{-3}. the detection cost w is taken equal to 0.1 and the restart cost C is 1. The analytical results presented here are simple but they provide a useful bounds for the cost of the detection algorithm and of the failures. They can be therefore be used as "fast" estimators of performance under failure detection and recovery, to complement the simulation results.

3.2 Comparing Detection Algorithms

Let us now evaluate and compare the failure detection Algorithms 1, 2 and 3. We evaluate the overhead due to each detection algorithm when there is no failure. Figure 4 presents simulation measurements for the average execution time of 10 different programs as a function of the program's size in number of tasks K. The detection and restart durations are respecively $w = 0.1$ and $C = 1$. When there is no failure, Algorithm 1 is the one which has smallest overhead. Algorithm 3 has more overhead than Algorithm 2. For $K = 200$, the average execution time obtained with algorithm 3 and $\pi = 1$ is 180 when it is only 40 without detection. Thus, Algorithm 3 increases the average execution time by 140, i.e. more than five times the cost of 28 of Algorithm 2 with $\pi = 1$.

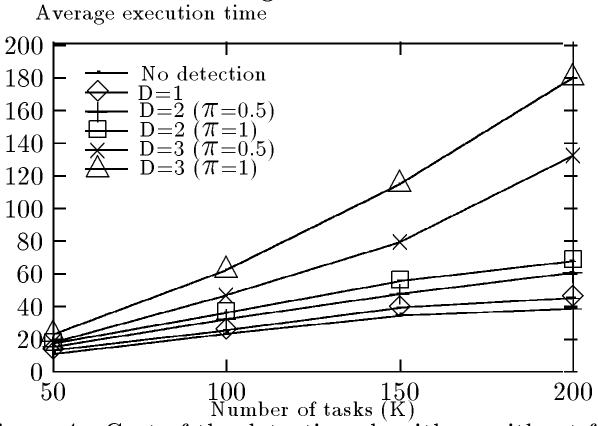

Figure 4 - Cost of the detection algorithms without failures.

Figures 5 and 6 present the average execution times over 10 programs as a function of the failure rate γ under failure assumptions FAA and FAB respectively. The average down period F is taken equal to 100. The figures show only one curve for algorithms 2 and 3 corresponding to the value of π which gave the lowest execution time. Under FAA, algorithm 2 gives the best result with $\pi = 0.25$ for $\gamma = 0.0001$ and $\gamma = 0.001$, and with $\pi = 1$ for $\gamma = 0.01$ and $\gamma = 0.1$. Still under FAA, algorithm 3 is plotted with $\pi = 0.25$ for $\gamma = 0.0001, 0.001$ and 0.01 and with $\pi = 1$ for $\gamma = 0.1$.

For failure assumption FAB, algorithm 2 gives the best result with $\pi = 0.25$ for $\gamma = 0.0001$ and $\gamma = 0.001$, with $\pi = 1$ for $\gamma = 0.01$ and with $\pi = 0.75$ for $\gamma = 0.1$; algorithm 3 is plotted with $\pi = 0.25$ for each value of γ.

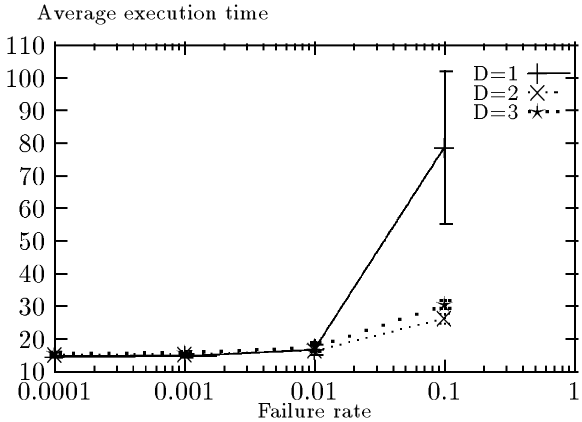

Figure 5 - Average execution time under Failure Assumption A.

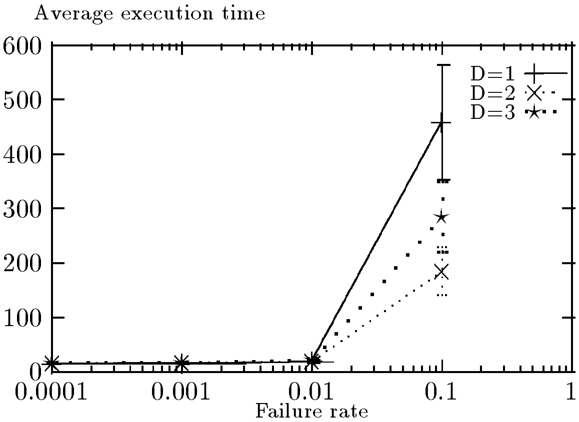

Figure 6 - Average execution time under Failure Assumption B.

Figure 7 shows results obtained under failure assumption B with and without communication overhead. Communication overhead is involved each time a task completes and informs its successor tasks of this. If a successor task is in the queue of another processor, then the inter-processor communication will require an exponential random time of average Z. On Figure 7, we have considered that $Z = 1$. When $\gamma = 0.1$, the execution time with communication is almost twice the time without communication delays.

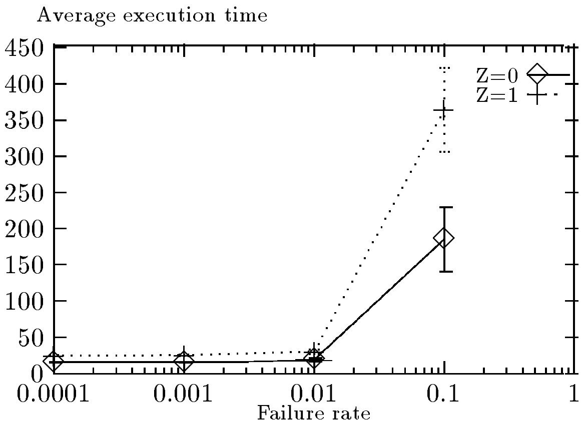

Figure 7 - Execution times with communication delays (FAB).

4 An Example: A Task-Graph for Matrix Multiplication

As an example of the construction of task graphs corresponding to a parallel computation we will consider matrix multiplication. Parallel matrix multiplication is very broadly used and easily implemented on a MIMD machine. Consider the matrix product: $C = A$ x B, where A, B and C are square matrices of size n by n. The n^2 terms c_{ij} of C are computed using the formula:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} * b_{kj}$$

Clearly the total number of operations necessary to compute the matrix C is given by $n^2(2n-1)$. At this point, we construct the task graph corresponding to this computation. For the sake of illustration, we present two different graphs on Figure 1 for $n = 4$. In Graph 1, the computation of one term c_{ij} requires only one task which performs n multiplications and $(n-1)$ additions. If each operation has an execution time of t, each task will then require $(2n-1)t$ time units.

In Graph 2, c_{ij} is computed in two parts. First the n products $a_{ik} * b_{kj}$ are computed for $k = 1$, ..., n. Then, the n terms previously obtained are added in a tree fashion requiring $\log_2 n$ steps. This means that one term c_{ij} is obtained using $2n-1$ tasks. In this case, to compute all the n^2 terms of the matrix C, $n^2(2n-1)$ tasks are necessary. In the first part, each task performs one multiplication; this gives an execution time of t. In the second part, each task performs one addition and has also an execution time of t.

Consequently Graph 2 will result in lower execution times than Graph 1 if a large number of processors is available. To quantify the number of processors necessary, we need to evaluate the degree of parallelism of each task graph, which is the maximum number of tasks present in the same generation of the graph. For Graph 1, there are n^2 tasks that can be computed in parallel using n^2 processors in only $(2n-1)*t$ time units. For Graph 2, the first generation is the one with the largest number of tasks which is n^3. If n^3 processors are available, then the total execution time of the second graph is $(\log_2 n + 1) * t$ time units. So the first graph has an execution time of $O(n)$ and needs no more than n^2 processors while the second has an execution time in $O(log_2 n)$ if n^3 processors are available. However, since both task graphs involve the same number $n^2(2n-1)$ of operations, if only n^2 processors are available the second task graph will have the same execution time of $(2n-1) * t$ units as the first graph.

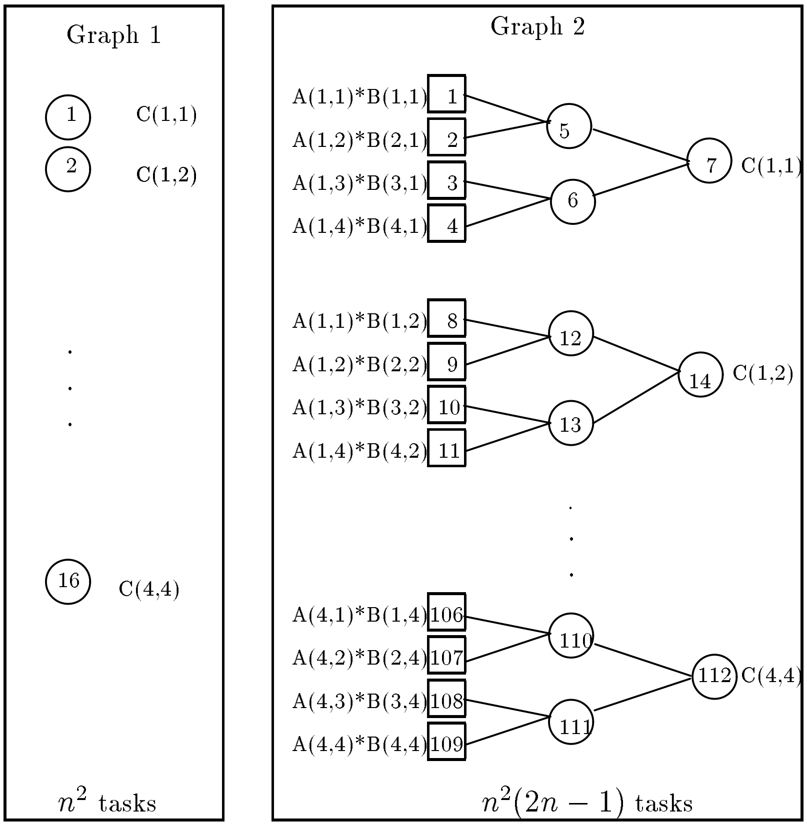

Figure 8. Two different task graphs for parallel matrix multiplication with $n = 4$.

5 An example: Dependable Execution of the Parallel FFT Algorithm

In order to proceed further in the understanding of how our proposed failure detection and recovery algorithms work, and the manner in which they affect a practical parallel application, we will consider the discrete Fourier transform which is widely used in a variety of application areas, ranging from statistics to signal processing. Much effort has been spent by researchers to find efficient implementations of this algorithm, especially on parallel machines. In particular, the Fast Fourier Transform algorithm [19] is an efficient version which we consider in parallel form in this section.

We first describe the FFT algorithm, its parallelization as well as the corresponding task graph structure. Then we conduct simulations of the execution of the FFT task graph on a set of failing parallel processors with our failure detection and recovery algorithms.

For a given complex vector X, its discrete Fourier transform is given on a finite number of input samples by:

$$y(k) = \sum_{i=0}^{N-1} x(i)\omega_N^{ik}, \qquad k = 0, 1, ..., N-1$$

where $\omega_N = e^{j(2\Pi/N)}$ is the Nth of unity. Clearly the computation of all the components of the complex vector Y requires N^2 complex multiplications and additions. It is possible to rewrite the above equation as:

$$y(k) = y_{even}(k) + \omega^k * y_{odd}(k), \qquad 0 \leq k \leq N/2 - 1$$

$$y(k + N/2) = y_{even}(k) - \omega^k * y_{odd}(k), \qquad 0 \leq k \leq N/2 - 1$$

where Y_{even} is the Fourier transform of the even points of X, and Y_{odd} is the Fourier transform of the odd points of X.

The Fast Fourier Transform (FFT) [19] uses the two previous recursive equations. It computes the discrete Fourier transform of a vector using $1/2N \log_2 N$ instead of N^2 operations. We show on Figure 2 an example of a task graph for the FFT algorithm for $N = 16$ with $Nlog_2N = 64$ tasks. The connections between the tasks represent the flow of data during the computation. Initially, a source task receives the component of X whose index is the bit-reversal-permutation of the task number decremented by 1. For example, the task 2 ($2 - 1 = 1$ being coded 0001 on $log_2N = 4$ bits) will receive the component $x(8)$. Then each task performs one complex multiplication and one complex addition. If each task has an execution time t, the total execution time of the parallel FFT algorithm is $log_2N * t$ time units with at least N processors.

Before we go into the simulation results, let us discuss the algorithmic specifics.

5.1 Detailed Algorithimcs of Failure Detection and Recovery

We will discuss several failure detection and recovery algorithms in this section. A first algorithm will be presented (Algorithm 1), and then several other algorithms which are essentially refinements of the basic scheme will be discussed.

The basic idea in all of these algorithms is that some – or all – of the tasks in the application are empowered with the failure detection and recovery procedure. Any task which is allowed to play this additional role (i.e. additional in the sense that this failure detection and recovery is carried out in addition to the ordinary computational function of the task) is said to be an **agent**.

Algorithm 1 – In this basic algorithm the tasks which have no successor (i.e. the "leaves" of the task graph) are allowed to carry out failure detection and recovery – they are the only agents in the application. This activity will be carried out in the following manner. After the leaf task i sets $V(i)$ to FINISHED, it will first examine the other leaves of the task graph. For each leaf seen not to be in the BLOCKED state, the agent will examine its predecessors until all tasks are accounted for as BLOCKED, READY, FINISHED, or ACTIVE but unfinished. This last case covers tasks which may have been stopped due to processor failure. The agent will examine the corresponding processors and reinitialize all such tasks by stopping their current execution and rescheduling them on another processor.

This algorithm is efficient when failures are rare but it will delay the detection of failures till at least one leaf completes execution. Indeed, it may happen that no leaf will ever be executed due to preceding failures and the application may be stopped as a whole. To remedy this deficiency, it is necessary to select intermediate tasks that will run the detection algorithm in addition to the leaves. The following algorithms differ in the way the selection of agents is performed and in the manner in which they follow the edges of the task graph.

Algorithm 2 – Here, we select agents from a certain number of "ranks", where the rank of a task is the longest path from itself back to a source task (one without predecessors). Each task having those selected ranks is an agent. An agent then begins examining its sibling tasks for failure, i.e. those for which it is neither a predecessor nor a successor, and then proceeds to smaller ranks as the need arises.

Algorithm 3 – When an agent finds a task of the same rank which has completed its execution, it will proceed to test in turn the successors of this completed task. The intention is to detect failures as soon as possible and to favor the rapid and successful completion of the whole program. Agents are still selected by choosing some of the ranks. Thus while Algorithms 1 and 2 carry out *look back* recovery, Algorithm 3 does look back followed by *look-forward*.

Algorithm 4 – Algorithm 4 first selects the leaves to be agents. Then it selects other tasks to be agents at random with probability π. Note that when $\pi = 1$, all the tasks are agents. Algorithm 4 only uses look-back for detection and recovery.

Algorithm 5 – This algorithm acts in the same way as Algorithm 3 (it uses look-back and look-forward) but in addition to the leaves, it selects agent tasks at random with probability π.

Several variants of these algorithms can also be considered. For instance detection can be improved by forcing the leaf task which is first to complete the detection algorithm, to perform the detection again after some delay. This leaf task can be kept alive as long as the program is not completed as a whole. Furthermore, if a program is not completed after some time-out t_{max} it may be restarted as a whole; this may be necessary when failure rates are very high.

5.2 Simulations for the Dependable Parallel FFT Algorithm

We have run simulations for the case where $N = 256$ generating a graph with 2048 tasks and we have used $M = 256$ processors. Each task has an execution time t corresponding to one complex multiplication and one complex addition; we have taken $3.24E-04$ for the average duration of a complex multiplication and $1.04E-04$ for the average duration of a complex addition (in seconds); these are realistic figures related to current workstation technology. The detection step duration w is taken equal to $6.5E-05$ which corresponds to the average time necessary to test the state of a task. The restarting step duration C is taken equal to $10E-05$; this is a relatively small value which implies that a processor's registers can be very rapidly loaded with the status of a task.

For all the simulations presented, we have computed confidence intervals with the probability of 95 %. The intervals we have thus obtained range from 0 % to 10 % of the average program execution times for all the simulations. They are not directly shown on the figures in order not to clutter the information which is being presented.

We easily see that when no detection algorithm is running, and when failures cannot occur, the total execution time of the graph with $N = 256$ is $(\log_2 N) * t = (\log_2 256) * (3.24E-04 + 1.04E-04) = 3.424E-03$ time units. However when there are no failures but a detection algorithm *is* running, this will generate obvious overhead.

In all the subsequent figures, the symbol D is used to denote the Detection algorithm concerned by a particular curve. Note that this is a change from the notation used in Section 3.

For Algorithms 2, 3, 4 and 5, in addition to the leaves, tasks are selected to be agents using a probabilistic assignment. Therefore we have run simulations with four values of the probability π that an arbitrary task is an agent: 0.25, 0.5, 0.75 and 1. Without failures, the smallest execution time (1.99E-02) is obtained with Algorithm 1 since it has only $N = 256$ tasks (the leaves) running the detection, while the other algorithms also have intermediate tasks running the detection algorithm. The largest execution time (3.78E-01) is encountered when we use Algorithm 5 for $\pi = 0.25$ giving an average detection overhead of $4.096E-02$.

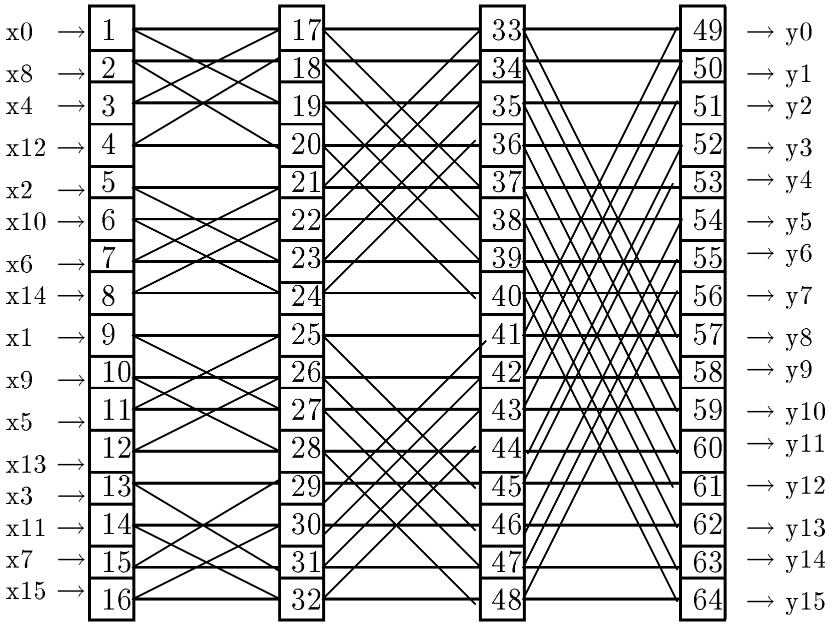

Figure 9. Task graph for the FFT computation

In order to analyze the behavior of the detection algorithms, we have evaluated the FFT execution time with each detection algorithm for three values of the failure rate γ: 0.001, 0.005 and 0.01. We first consider the failure assumption A (FAA) where a task running the detection algorithm is supposed not to fail until it has completed the detection. We have taken for the maximum program duration allowed t_{max} the value 0.5 for $\gamma = 0.001$, 1 for $\gamma = 0.001$ and 1.5 for $\gamma = 0.001$.

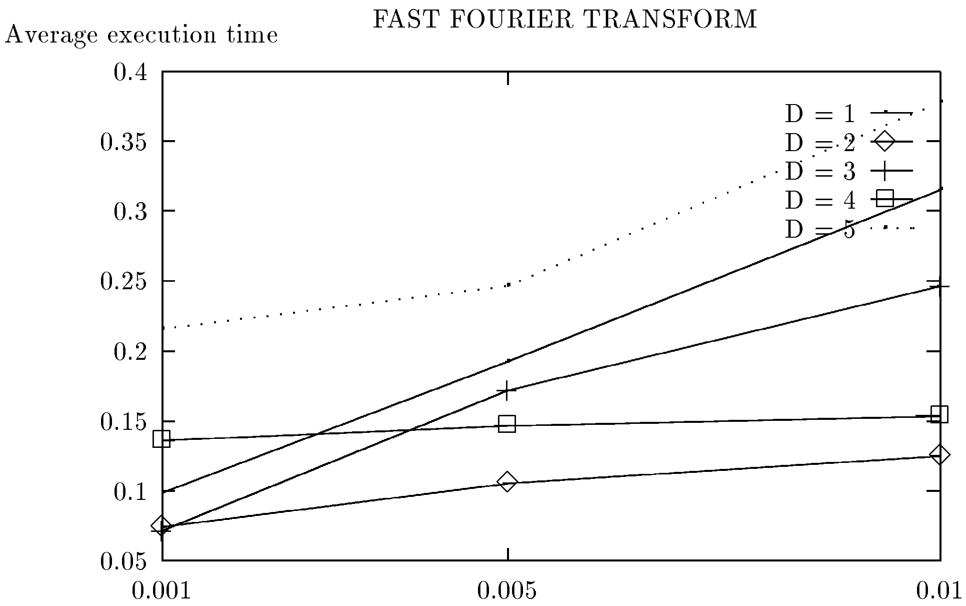

Figure 10. Comparison of the detection algorithms

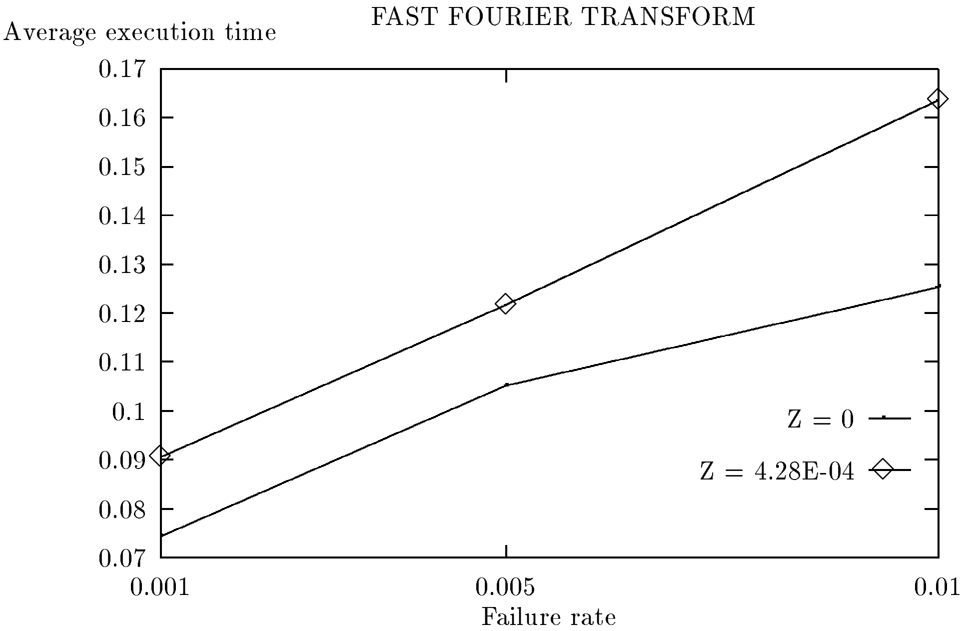

Figure 11. Execution times with and without communication overhead

We show on Figure 10 the average execution times for the parallel FFT algorithm for each of the failure detection and recovery techniques which we consider, as a function of the failure rate. For Algorithms 2 to 5 only the value of π which gave the shortest execution time is shown for each value of failure rate. For Algorithms 2, 3 and 4, $\pi = 0.25$ always provided the smallest execution time for any value of γ. For Algorithm 5, $\pi = 0.75$ appeared to be the best choice for any value of γ.

Algorithm 2 gives the smallest execution times for any value of γ, even though we observe that it results in average job execution times which are very close to Algorithm 3 for low values of γ (e.g. $\gamma = 0.001$). Algorithm 5 gives the largest execution times.

Finally, Figure 11 presents the effect of communication overhead, quantified by the parameter Z representing the time taken to exchange a message from a task to its successor task.

References

[1] E. Gelenbe, *"Multiprocessor performance,"* John Wiley & Sons, New York, 1989.

[2] E. Gelenbe, R. Nelson, T. Philips and A. Tantawi, "Asymptotic processing time of a model of parallel computation", *Proc. National Computer Conference* (U.S.A.), 127–138, 1986.

[3] E. Gelenbe and D. Derochette, "Performance of roll-back recovery systems under intermittent failures", **Comm. ACM,** 21 (6), June 1978, pp 493-499.

[4] E. Gelenbe, "A model of roll-back recovery with multiple checkpoints", *Proc. ACM-IEEE 2nd International Symposium on Software Engineering,* October 1976, pp 251-255.

[5] E. Gelenbe, "On the Optimum Check-Point Interval", **J. ACM**, 26, 259–270, 1979.

[6] E. Gelenbe and I. Mitrani, "Modeling the Execution of Block Structured Processes with Hardware and Software Fauilures", in G. Iazeolla, P. Courtois, and A. Hordijk (eds.) *Mathematical Computer Performance and Reliability*, North Holland Pub. Co., 1983.

[7] Z. Kedem, K. Palem, and P. Spirakis, "Efficient Robust Parallel Computations", *Proc. 22nd ACM Symp. on Theory of Computing*, 138–148, 1990.

[8] Z. Kedem and K. Palem, "Transformations for the Automatic Derivation of Resilient Parallel Programs", *Proc. 1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 16–25, 1992.

[9] Z. Kedem, K. Palem, M. Rabin, and A. Raghunathan, "Efficient Program Transformations for Resilient Parallel Computation via Randomization", *Proc. 24th ACM Symp. on Theory of Computing*, 306–317, 1992.

[10] Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis, "Resilient Parallel Computing on Unreliable Parallel Machines", pp. 145–172 in: *Lectures on Parallel Computation,* eds. A. Gibbons and P. Spirakis, Cambridge University Press, 1993.

[11] F. Baccelli and A. Makowski, *Queueing models for systems with synchronization constraints*, Proc. IEEE, 77 (1), 1989.

[12] M. Banatre, A. Gefflaut, P. Joubert, P. Lee, C. Morin, *An architecture for tolerating processor failures in shared-memory multiprocessors*, INRIA Report No. 1965, p.1-35, 1993.

[13] P. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.

[14] B. Bhargava and S.-R. Lian, *Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems - An Optimistic Approach*, Proc. 7th IEEE Symposium on Reliable Distributed Systems, 1988.

[15] W. Cellary, E. Gelenbe, J. Morzy *Concurrency Control in Distributed Databases*, Elsevier-North Holland, Amsterdam and New York, 1988.

[16] S. Chabridon and E. Gelenbe, *Dependable execution of distributed programs*, Proc. Massively Parallel Processing Conference '94, (North-Holland Elsevier), Delft, June 21-23, 1994.

[17] W. Chu and K. Leung, *Module replication and assignment for real-time distributed processing systems*, Proc. IEEE, 547–562, 1987.

[18] W. Chu, C. Sit, and K. Leung, *Task response time for real-time systems with resource contentions*, IEEE Trans. on Software Engineering, 17 (10), 1076–1092, 1991.

[19] J.M. Cooley, J.W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Mathematics of Computation, Vol. 19, p.297-301, 1965.

[20] F. Cristian and Farnam Jahanian, *A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations*, Proc. 10th IEEE Symposium on Reliable Distributed Systems, 1991.

[21] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel, *The Performance of Consistent Checkpointing*, Proc. 11th IEEE Symposium on Reliable Distributed Systems, 1992.

[22] E.N. Elnozahy and W. Zwaenepoel, *Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit*, IEEE Transactions on Computers, 41 (5), May 1992.

[23] E. Gelenbe, *Temps d'exécution asymptotique d'un programme parallèle*, Comptes-Rendus Acad. Sci. Paris (Proc. French National Academy of Science), 309 (I), p.399-402, 1989.

[24] E. Gelenbe and Z. Liu, *Performance analysis approximations for parallel processing of concurrent tasks*, in M. Cosnard (ed.) Parallel Computation, North-Holland Pub. Co., 1988.

[25] T.T-Y. Juang and S. Venkatesan, *Crash Recovery With Little Overhead (Preliminary Version)*, Proc. 11th IEEE International Conference on Distributed Computing Systems, 1991.

[26] P.C. Kanellakis, A.A. Shvartsman, *Efficient parallel algorithms can be made robust*, Distributed Computing, p.201-217,1992.

[27] P.C. Kanellakis, A.A. Shvartsman, J.F. Buss, P.L. Radge, Parallel algorithms with processor failures and delays, Brown University Tech. Rep. No. CS-91-54.

[28] R. Koo and S. Toueg, *Checkpointing and Rollback Recovery for Distributed Systems*, IEEE Transactions on Software Engineering, SE-13(1), January 1987.

[29] P-J. Leu and B. Bhargava, *Concurrent Robust Checkpointing and Recovery in Distributed Systems*, Proc. 4th IEEE International Conference on Data Engineering, 1988.

[30] N. Pekergin and J. Vincent, *Stochastic bounds on parallel program execution times*, IEEE Trans. on Software Engineering, 17 (10), p.105-113, 1991.

[31] R.A. Sahner and K.S. Trivedi, *Performance and reliability using directed acyclic graphs*, IEEE Trans. on Software Engineering, 13 (10), p.1105-1114, 1987.

[32] R.D. Schlichting, F.B. Schneider, *Fail-stop processors: An approach to designing fault-tolerant computing system*, ACM Trans. on Computer Systems, Vol. 1, No. 3, p.222-238, 1983.

[33] A. P. Sistla and J.L. Welch, *Efficient distributed recovery using message logging*, Proc. 8th ACM Symposium on Principles of Distributed Computing, 1989.

[34] R.E. Strom, D.F. Bacon, and S.A. Yemini, *Volatile Logging in n-Fault-Tolerant Distributed Systems*, Proc. 18th IEEE Symposium on Fault Tolerant Computing, 1988.

[35] R.E. Strom and S.A. Yemini, *Optimistic Recovery in Distributed Systems*, ACM Transactions on Computer Systems, 3 (3), August 1985.

[36] A. Thomasian and P. Bay, *Analytic queueing network models for parallel processing of task systems*, IEEE Trans. on Computers, 35 (12), p.1045-1054, 1986.

[37] Z. Tong, R.Y. Kain, and W.T. Tsai, *A Low Overhead Checkpointing and Rollback Recovery Scheme for Distributed Recovery*, Proc. 8th IEEE Symposium on Reliable Distributed Systems, 1989.

[38] J.N. Tsitsiklis, C.H. Papadimitriou, P. Humblet, *The performance of a precedence based queueing discipline*, JACM, Vol. 33, 3, p.593-602, July 1986.