UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

DIAGNOSTIC DES PANNES DANS LES SYSTÈMES MULTIPROCESSEURS

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR

SAMUEL GUILBAULT

DÉCEMBRE 2006

UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

Département d'informatique et d'ingénierie

Ce mémoire intitulé :

DIAGNOSTIC DES PANNES DANS LES SYSTÈMES MULTIPROCESSEURS

présenté par

Samuel Guilbault

pour l'obtention du grade de maître ès science (M.Sc.)

a été évalué par un jury composé des personnes suivantes :

Dr. Andrzej Pelc ..... Directeur de recherche

Dr. Karim El Guemhioui ..... President du jury

Dr. Jurek Czyzowicz ..... Membre du jury

Mémoire accepté le : 1 décembre 2006

À mon fils Malcolm dont les nombreux sourires m'ont donné l'énergie de persévérer dans mon rôle de père, d'étudiant, et dans la poursuite de ma carrière.

Remerciements

À Andrzej Pelc pour son soutien, sa patience et ses conseils.
À Claire Sévigny pour avoir corrigé mes travaux.
À Denise Fortin pour avoir corrigé mes travaux.
À Gisèle Tétrault pour avoir corrigé mes travaux.
À Karine Fortin pour m'avoir supporté et encouragé.
À Éric Vachon et à Cédric Bastien pour leurs conseils.
Merci !

Table des matières

Liste des figures

Liste des tableaux

Résumé

Les systèmes multiprocesseurs sont vulnérables aux défectuosités et la tolérance aux pannes est devenue importante pour les systèmes avec un grand nombre de processeurs reliés ensemble par un réseau interconnecté. La complexité montante des systèmes multiprocesseurs a déclenché une recherche intense sur les problèmes de détection de fautes, de diagnostics et de reconfiguration. Le diagnostic du système est une importante technique d'amélioration de l'intégrité et de la disponibilité du système.

Ce mémoire présente des algorithmes de diagnostics de pannes dans les sytèmes multiprocesseurs. Plus spécifiquement, les diagnostics proposés sont déterministes et utilisent aussi bien l'approche adaptative que non-adaptative dans certains cas. Nous avons amélioré l'algorithme de Blecher[6] en trouvant divers algorithmes qui sont plus efficaces tant sur le plan du nombre de liens (densité du réseau) nécessaire pour son fonctionnement que du nombre de rondes (temps d'exécution). Chacun de ces algorithmes est optimal en nombre de tests.

Abstract

Multiprocessor systems are vulnerable to failures, and fault-tolerance becomes important, particularly for systems with a large number of processors connected in a network. The rising complexity of multiprocessor systems stimulated intense research on the problems of fault detection, diagnosis and reconfiguration. The diagnosis of the system, i.e., locating all faulty processors, is an important technique in the improvement of the integrity and availability of the system.

This master's thesis presents deterministic diagnosis algorithms for multiprocessor systems. Most of the proposed algorithms use the adaptive approach. We improved the algorithm of Blecher [6] by designing various diagnosis methods which work for sparser networks and have shorter execution time in the worst case. Each of our algorithms is optimal in the number of tests.

Chapitre 1

Introduction

1.1 Le diagnostic des pannes

Depuis plusieurs années, les systèmes multiprocesseurs prennent un essor considérable dans la recherche en informatique. En effet, les avantages apportés par un système comprenant plusieurs processeurs joignant leurs ressources dans un travail collectif sont significatifs. Toutefois, ces mêmes avantages peuvent être complètement perdus si certains processeurs en panne sont autorisés à corrompre le système et à disséminer des fausses informations.

Puisque de telles situations sont courantes et peuvent avoir parfois des conséquences très graves pour un réseau et son fonctionnement, il importe de pouvoir, dans un premier temps, diagnostiquer ces pannes pour ainsi les éviter ou encore les réparer. Dans les deux cas, la localisation de pannes est la pierre angulaire et est donc indispensable.

Toutefois, il n'y a pas qu'une seule technique de diagnostic car, comme dans tout problème, nous devons définir les hypothèses de travail qui vont à leur tour déterminer les types de diagnostic. Par exemple, est-ce que nous imposons une borne maximale sur le nombre de pannes dans un réseau ou encore, supposons-nous une probabilité qu'un processeur tombe en panne ? Est-ce que nous pouvons faire des tests séquentiellement ou encore faut-il les faire tous en même temps ? Est-ce que le temps pour trouver tous les processeurs fautifs est notre préoccupation première ou est-ce que nous préférons économiser le nombre de tests ?

Puisqu'il y a tant d'hypothèses possibles, la recherche dans ce domaine est vaste et en pleine effervescence et il est intéressant de découvrir de nouvelles méthodes de diagnostic ainsi que de nouveaux problèmes au fur et à mesure que les systèmes multiprocesseurs évoluent.

1.2 Définition du problème

La nature des systèmes multiprocesseurs les rend vulnérables aux pannes. Une panne peut avoir des conséquences très graves dans un système multiprocesseur et c'est pourquoi le diagnostic doit être effectué de façon efficace, rapide et correcte.

Certains chercheurs ont proposé divers modèles de diagnostic de systèmes multiprocesseurs. Dans certain cas, les algorithmes applicables pour ces modèles sont déterministes et dans d'autre cas, ils sont probabilistes. De plus, certains algorithmes sont adaptatifs, c'est-à-dire qu'ils tirent des conclusions des tests précédents qu'ils ont effectués pour améliorer la rentabilité de leurs tests subséquents, alors que d'autres sont non-adaptatifs et ne tiennent pas compte des tests effectués au préalable.

Bien que les algorithmes applicables pour les modèles déterministes soient moins fréquents dans les cas réels, ils sont souvent plus simples à bâtir. De plus, par la suite, certains algorithmes déterministes peuvent être adaptés partiellement au modèle probabiliste en effectuant quelques modifications.

Nous proposons d'étudier les techniques actuelles de diagnostics déterministes ainsi que certaines techniques probabilistes de pannes avec une approche non-adaptative ou adaptative dans les systèmes multiprocesseurs.

Notre contribution sera au niveau de la création et l'analyse des algorithmes déterministes plus efficaces tant sur le plan de la densité des systèmes que sur la rapidité du diagnostic. Nous allons également préserver l'optimalité du nombre de messages utilisés lors de ces diagnostics.

1.3 Présentation des résultats

Nos résultats sont concentrés autour de trois mesures de qualité des algorithmes de diagnostic adaptatif : le nombre de tests, le temps de fonctionnement et la densité du réseau auquel l'algorithme est applicable. Il existe dans la littérature un algorithme de diagnostic conçu par Blecher [6] qui utilise $n+t-1$ tests dans un réseau de n processeurs avec au plus t processeurs défectueux. Le nombre de $n+t-1$ tests a été prouvé optimal. Cependant, l'algorithme de Blecher fonctionne seulement dans le graphe complet et utilise un temps $O(n)$. Nos résultats visent l'amélioration de cet algorithme dans deux directions différentes, tout en gardant la borne inférieure du point de vue du nombre de tests. D'une part, nous concevons des algorithmes applicables dans des réseaux avec peu de liens et d'autre part, nous proposons des algorithmes de diagnostic travaillant dans le réseau complet mais beaucoup plus rapides que celui de Blecher.

Nous avons trouvé un algorithme optimal en nombre de tests ainsi qu'en nombre de rondes dans une grille $m \times n$. Nous avons également démontré la borne inférieure sur la densité d'un réseau pouvant permettre la faisabilité du t-diagnostic et nous avons trouvé des algorithmes fonctionnant dans des réseaux de densité améliorée tout en respectant la borne inférieure sur le nombre de tests. Dans certains cas spécifiques, nous avons obtenu des algorithmes fonctionnant dans un réseau de densité optimale (i.e. : réseau raréfié au maximum). Finalement, nous avons trouvé des algorithmes qui sont très rapides tout en respectant la borne inférieure sur le nombre de tests.

Au chapitre 4, nous présentons l'algorithme Diagnostic NonAda-GrilleTest qui a permis de faire le diagnostic non-adaptatif pour un graphe en grille $m \times n$ en $2n$ tests pour $t \leq 2$. L'algorithme Ada-GrilleTest a permis de faire le diagnostic en $n+1$ tests pour $t \leq 2$, dans la même grille. Ces deux résultats sont optimaux en nombre de tests. De plus, l'algorithme Ada-GrilleTest fait le diagnostic en nombre optimal de rondes, soit 3 rondes.

Au chapitre 5, nous présentons trois algorithmes qui ont permis de travailler dans un graphe de faible densité. Chacun de ces algorithmes comporte des avantages différents. Le premier présenté est l'algorithme Ada-Diagnostic-Régulier. L'avantage est qu'il s'applique dans un graphe régulier avec nt liens. Le second présenté est l'algorithme Ada-Blecher-Raréfié qui permet de faire le diagnostic dans un graphe irrégulier de $nt-t$ liens.

Finalement, le dernier présenté fonctionne dans les graphes les plus raréfiés qui admettent un t-diagnostic, soit les graphes avec $\lceil \frac{nt}{2} \rceil$ liens pour $t < 2\sqrt{n} - 2$.

Au chapitre 6, nous présentons deux algorithmes de diagnostic rapides. L'algorithme Diagnostic Adaptatif Blecher-Rapide fonctionne en $O(\log n + t)$ rondes pour $n \geq 2t + 1$ et le deuxième, l'algorithme Diagnostic Balayage-Rapide fonctionne en $O(\log t)$ rondes pour $n \geq 2t^2 + 2t$ et $t \geq 18$.

Au chapitre 7, nous présentons les résultats des simulations. D'abord, nous comparons l'algorithme de Blecher [6] avec l'algorithme Ada-Diagnostic-Optimal. Nous pouvons voir que l'algorithme que nous avons proposé nécessite en moyenne moins de tests que l'algorithme de Blecher pour de petits t. Nous comparons aussi l'algorithme Diagnostic Adaptatif de Blecher-Rapide avec l'algorithme Diagnostic Adaptatif Balayage-Rapide nécessite moins de tests en moyenne que l'algorithme que nous avons proposé. Toutefois, notre algorithme est nettement plus rapide.

Chapitre 2

État des connaissances

2.1 Revue de la littérature

Les systèmes multiprocesseurs sont vulnérables aux fautes et la tolérance aux pannes est devenue importante pour les systèmes avec un grand nombre de processeurs reliés ensemble par un réseau interconnecté. La complexité montante des systèmes multiprocesseurs a déclenché une recherche intense sur les problèmes de détection de fautes, de diagnostics et de reconfiguration. Le diagnostic du système est une importante technique d'amélioration de l'intégrité et de la disponibilité du système. La majorité des recherches algorithmiques concernant le diagnostic des systèmes multiprocesseurs était effectuée dans les années 1980-1999. Il y a cependant quelques articles plus récents sur le sujet [9, 20, 21]

2.2 Les différents modèles utilisés pour le diagnostic de processeurs

Un des objectifs d'un modèle dans le cadre du diagnostic de processeurs est de définir le comportement d'un processeur lorqu'il devient défectueux. Idéalement, un processeur fautif devrait agir de façon prévisible comme le font les bons processeurs mais ce n'est généralement pas le cas. Dans le cadre d'un diagnostic de processeurs, le modèle est une description des résultats de tests en tenant compte du statut des processeurs qui testent

et des processeurs qui sont testés. De plus, dans une procédure de diagnostic, il faut connaître la nature des fautes. Nous avons trois classes de fautes, les fautes passagères qui n'arrivent qu'une fois et disparaissent d'elles-mêmes, les fautes intermittentes qui apparaissent et disparaissent continuellement et les fautes permanentes qui sont présentes jusqu'à ce qu'elles soient réparées.

Certains modèles ont été introduits pour permettre de déterminer le comportement d'un processeur fautif. Les modèles les plus simples sont les modèles HK1 et HK 2 qui ont été introduits par Kreutzer et Hakimi [14] appelés communément modèles d'invalidation réflectifs et irréflectifs (Voir fig.2.1).

Le modèle HK1 est un modèle dans lequel un processeur fautif teste toujours les bons processeurs comme mauvais, les mauvais aléatoirement et les bons testeurs donnent toujours un diagnostic correct. Le modèle HK2 est un modèle dans lequel un processeur fautif teste toujours les autres processeurs comme mauvais et les bons testeurs donnent toujours un diagnostic correct.

Vient ensuite le modèle BGM introduit par Barsi, Grandoni et Maestrini [2] dans lequel un processeur fautif teste toujours un autre mauvais processeur comme mauvais, un bon processeur aléatoirement et les bons testeurs donnent toujours un diagnostic correct.

Toutefois, ces trois modèles assument que le comportement d'un processeur fautif reste prévisible dans une certaine mesure. Cela est peu réaliste et c'est pourquoi notre choix s'est porté sur le quatrième modèle, soit le modèle PMC introduit par Preparata, Metze et Chien [25]. Notons que ce modèle est chronologiquement le premier modèle introduit dans la littérature.

Dans le modèle PMC, le système multiprocesseur est représenté par le graphe non orienté $G = (V, E)$ dans lequel les noeuds dans V représentent des processeurs et les arêtes dans E représentent les liens de communication. Chaque noeud est soit correct (dénoté par +) ou défectueux (-). De plus, nous allons considérer un graphe orienté avec étiquettes sur les arcs $G^* = (V, E^*)$ qui est appelé attribution de tests dans lequel $(v_i, v_j) \in E^*$ seulement si $\{v_i, v_j\} \in E$. Le processeur v_i est dit testeur du processeur v_j (processeur testé ou sujet). La restriction ci-dessus indique que les tests peuvent être

accomplis seulement sur les liens du système. Pour un test $(v_i, v_j) \in E^*$, l'étiquette $w(v_i, v_j) = 0$ si v_i évalue v_j comme un processeur correct et $w(v_i, v_j) = 1$ si v_i évalue v_j comme un processeur défectueux. Alors nous disons que v_i a un arc 0 (respectivement 1) vers v_j. La collection de tous les résultats de tests est appelée un syndrome (Voir fig.2.2).

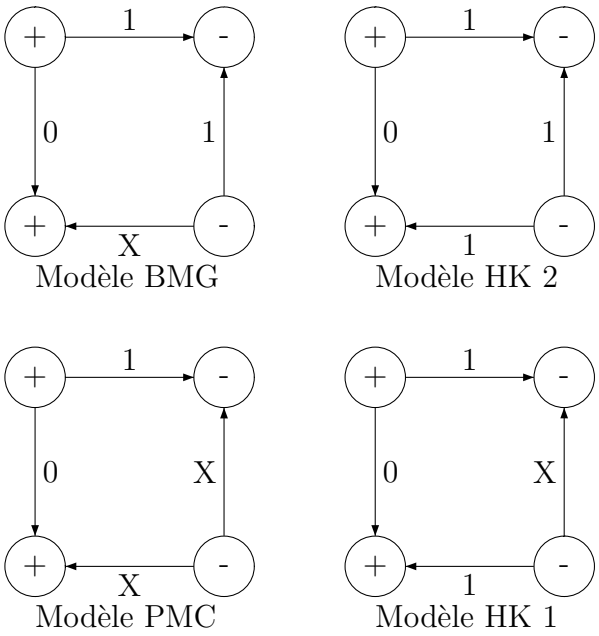

FIG. 2.1 – Exemple de différents modèles

Trois hypothèses sont faites dans le modèle PMC que nous utilisons :

1. Les fautes sont permanentes.
2. Les résultats des tests exécutés par les processeurs corrects sont toujours exacts : un processeur correct évalue un sujet correct comme correct et un sujet défectueux comme défectueux.
3. Les résultats des tests exécutés par les processeurs défectueux sont complètement non fiables : un testeur défectueux peut donner n'importe quel résultat de tests, peu importe le statut du processeur testé.

Plus précisément, les étiquettes décrivant les résultats des tests doivent satisfaire la condition suivante :

$$w(v_i, v_j) = \begin{cases} 0 & \text{si } v_i \text{ et } v_j \text{ sont corrects} \\ 1 & \text{si } v_i \text{ est correct et } v_j \\ & \text{est fautif} \\ 0 \vee 1 & \text{Pour les autres cas} \end{cases}$$

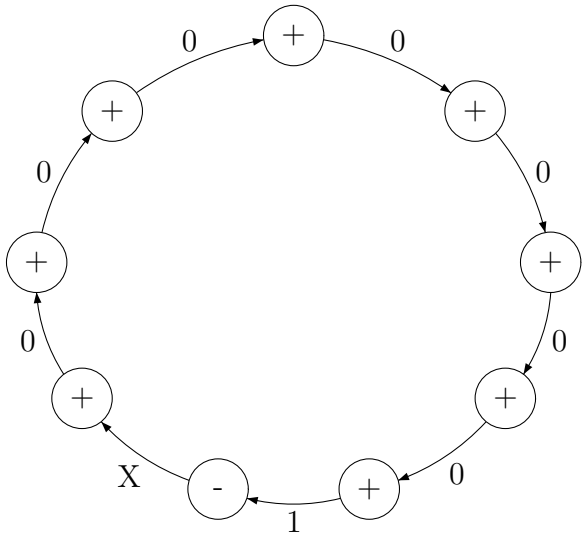

FIG. 2.2 – Exemple des résultats de tests dans le modèle PMC

Plusieurs variations de ce modèle ont été étudiées et on a proposé des algorithmes efficaces de diagnostics [1, 7, 8, 12, 24]. Des chercheurs ont enquêté tant dans le pire cas de localisation des fautes [6, 12, 22] que sous la supposition de leur distribution aléatoire [7, 24].

2.3 Les tests

Après avoir caractérisé le comportement des différents processeurs défectueux selon différents modèles, il importe de déterminer les méthodes de tests permettant d'identifier ces processeurs défectueux. La nature des tests effectués est un point majeur dans le domaine de l'informatique pratique. Principalement, un processeur teste un autre processeur en lui donnant certaines entrées pour un problème de calcul et en comparant les sorties avec des résultats supposés corrects.

Le choix de la méthode de tests est une phase très importante du diagnostic des systèmes multiprocesseurs car, selon le choix que nous ferons, certains processeurs peuvent passer certains tests et en faillir d'autres. Il est important de choisir au départ quel genre de contrôle nous voulons avoir sur le système en sélectionant de manière appropriée les tests. Nous présenterons ici trois différentes approches : l'autodiagnostic, les tests en groupe ainsi que les tests par comparaison. Ensuite, nous proposerons un exemple de tests qui satisfait le modèle PMC.

2.3.1 L'autodiagnostic

L'autodiagnostic peut être fait par chaque processeur sur lui-même en utilisant une série de tests. Dans ce cas spécifique, un testeur vérifiant le statut du processeur s'autodiagnostiquant ne fait que demander le statut de celui-ci.

L'avantage d'une telle technique est qu'un processeur qui n'est pas occupé peut utiliser ses temps libres pour s'autodiagnostiquer et ainsi ne pas encombrer le réseau. Kuhl et Reddy [16] ont proposé un système d'autodiagnostic qui permet à un processeur de s'évaluer comme défectueux ou comme bon.

2.3.2 Les tests en groupe

Les tests en groupe ne permettent que de déterminer si un groupe de processeurs est fautif ou correct. Pour être capable de valider un seul processeur, cela peut demander plusieurs tests. Également, l'exécution d'un test peut demander plusieurs processeurs et si l'un d'entre eux est défectueux, cela peut causer le rejet du test en groupe.

Si plusieurs processeurs sont requis pour faire le test en groupe, alors nous appellerons ce test MIPT (i.e. :Multiple Invalidation Per Test). Toutefois, le modèle PMC s'intéresse plutôt au SIPT (i.e. : Single Invalidation Per Test).

Les avantages d'utiliser le MIPT est que, généralement, l'implantation en est plus simple, il est plus rapide à exécuter et le test peut suffire dans certains cas spécifiques, ce qui fait gagner beaucoup de temps en terme de nombres de tests à effectuer.

2.3.3 Les tests par comparaison

Les tests par comparaison se font en assignant une tâche à deux processeurs différents et en comparant les résultats obtenus. Par un seul test, nous pouvons détecter une panne, mais nous ne pouvons pas la localiser. Par contre, si nous utilisons plus de deux processeurs, un processeur peut diagnostiquer jusqu'à $\lceil n/2 \rceil - 1$ processeurs en utilisant certaines techniques. Malek [18] a introduit une approche par comparaison qui modélise les tests sous forme de graphe non-orienté (Voir fig.2.3).

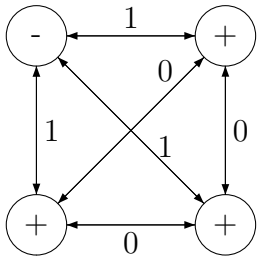

FIG. 2.3 – Exemple de tests par comparaison

2.3.4 Les tests pour le modèle PMC

Il y a des exemples naturels et pratiques de tests qui satisfont le modèle PMC. Chaque processeur P choisit un problème de calcul suffisamment complexe pour qu'un processeur en panne ne puisse pas (en général) le résoudre correctement. Au lieu d'un seul problème, on peut déterminer une série de plusieurs problèmes.

Ensuite, le processeur P envoie le problème au processeur testé Q et attend la réponse. En attendant, il résout le problème lui-même. Si les deux réponses sont identiques (ou les deux suites de réponses à la série de problèmes sont identiques), P considère Q comme bon, sinon, il le considère comme fautif.

Ce type de tests correspond bien au modèle PMC. Si P et Q sont bons, les résultats seront identiques. Si P est bon et Q est fautif, ils seront différents. Donc, un bon processeur donne un résultat de tests qui est correct. Si P est lui-même fautif, alors il

peut envoyer des données incorrectes à Q ou mal résoudre ses propres problèmes. Par conséquent, une telle situation rend le résultat du diagnostic de Q imprévisible.

2.4 Le diagnostic

Le but d'un diagnostic est de trouver tous les processeurs défectueux dans le système, basé sur le syndrome obtenu. Un système avec un graphe de tests $G^* = (V, E^*)$ est t-diagnosticable, si tous les processeurs défectueux dans le système peuvent toujours être identifiés à condition que leur nombre n'excède pas t. Nous exigeons que le diagnostic soit correct et complet, c'est-à-dire que tous les processeurs défectueux doivent être diagnostiqués comme défectueux et tous les processeurs sans faute comme corrects. Un algorithme de diagnostic travaillant pour n'importe quelle configuration d'au plus t processeurs défectueux est appelé un t-diagnostic. Dans ce qui suit, nous allons utiliser les termes "pannes" ou "fautes" au lieu de processeurs défectueux.

Preparata et al. [25] ont étudié des distributions de tests fournissant des renseignements suffisants pour faire correctement un t-diagnostic non-adaptatif (i.e. : One Step t-Diagnosis). Ils ont prouvé que si t est la borne supérieure du nombre de processeurs défectueux, alors il est nécessaire pour un système de contenir n processeurs avec $n \geq 2t+1$ afin qu'il puisse être t-diagnosticable. De plus, il est nécessaire pour chacun des processeurs d'être évalué par au moins t autres processeurs distincts. Hakimi et Amin [12] ont donné une caractérisation de systèmes t-diagnosticables.

Le diagnostic non-adaptatif est un diagnostic où l'utilisateur fournit tout d'abord le graphe de tests à effectuer puis collecte les résulats des tests pour déterminer quels sont les bons processeurs ainsi que les mauvais processeurs en analysant le syndrome obtenu. La borne inférieure pour ce type de diagnostic est de nt tests. Il est intéressant de noter que dans ce type de diagnostic, il est impossible d'avoir moins que nt tests puisque nous ne pouvons pas déterminer à l'avance les résultats obtenus.

Nakajima [13] a introduit en premier le diagnostic adaptatif, où les tests futurs peuvent être déterminés dynamiquement, sur la base des résultats des tests précédents. L'aspect important de cette méthode est d'identifier un processeur correct le plus tôt

possible et ensuite l'utiliser pour diagnostiquer d'autres processeurs dans le système. Hakimi et Nakajima [12, 19] ont proposé un algorithme de diagnostic adaptatif pour le graphe complet prenant au plus $n + 2t - 2$ tests pour identifier tous les processeurs défectueux, en supposant que le nombre de processeurs défectueux est au plus t et qu'il y a n processeurs.

Blecher [6] et Wu [26] ont amélioré la précédente borne à $n + t - 1$ tests. De plus, Blecher a démontré que ceci est aussi la borne inférieure sur le nombre de tests pour identifier toutes les unités défectueuses en pire cas. Toutefois, ces résultats supposent que le système est modélisé par un graphe complet. Pour un nombre élevé de processeurs, des systèmes complètement interconnectés sont difficiles à implanter, ce qui restreint de façon significative l'applicabilité du diagnostic ci-dessus. Toutefois, étant donné que n'importe quel graphe peut être plongé dans un graphe complet, cette borne inférieure sera vraie pour tout graphe.

Jusqu'ici, nous avons parlé du diagnostic déterministe car, dans le cadre de notre mémoire, c'est le type de diagnostic qui nous intéresse principalement. Toutefois, le diagnostic probabiliste permet d'utiliser moins de tests en moyenne. Son désavantage est qu'il ne fonctionne pas toujours mais seulement avec forte probabilité. Dans le diagnostic probabiliste, nous ne faisons plus aucune supposition quant à la borne sur le nombre de processeurs défectueux dans le système, mais une probabilité d'échec est attribuée à chaque processeur. Toutefois, le diagnostic probabiliste ne garantit pas un succès. On dit qu'un diagnostic probabiliste est presque certain si la probabilité qu'il soit correct est d'au moins $1 - \frac{1}{n}$ pour un système de n processeurs. Il s'ensuit que, lorsque le diagnostic est fait dans les grands réseaux, les chances de succès croissent.

Le modèle probabiliste a une limitation majeure. Il requiert que nous puissions assigner des probabilités de pannes aux processeurs pour modéliser les processeurs défectueux et ainsi avoir un diagnostic de qualité. Toutefois, dans l'environnement pratique, l'assignation de probabilité est un problème non résolu. L'algorithme de majorité non-adaptative proposé par Blough, Sullivan et Masson [7] a permis d'atteindre la borne de $O(n\ log\ n)$ tests avec une probabilité de succès presque certaine. Dans le scénario adaptatif, le nombre de tests peut être réduit à $O(n)$.

Dahbura et Masson [11] ont introduit un algorithme permettant de trouver tous les processeurs défectueux dans un environnement déterministe avec un t-diagnostic non-adaptatif en utilisant au plus $O(n^{2.5})$ tests. Dahbura a ensuite proposé une modification à l'algorithme original pour s'appliquer dans l'environnement probabiliste.

2.5 Différentes topologies de réseaux

Bien que certains algorithmes aient été créés pour fonctionner dans des réseaux complets (i.e. cliques) tel que l'algorithme de diagnostic adaptatif de Blecher [6], certaines topologies de réseaux de plus faible densité peuvent être intéressantes du fait qu'elles possèdent des propriétés spécifiques facilitant leur implantation.

Bjorklund [9] a donné un algorithme permettant de diagnostiquer un hypercube en $n + t - 1$ tests. Il a également donné un algorithme pour diagnostiquer l'hypercube en 4 rondes.

Yamada, Nomura et Ueno [20] ont donné des algorithmes adaptatifs pour les CCC (i.e. : Cube Connected Cycles) en utilisant le nombre minimal de tests. Ils ont également amélioré la précédente borne en montrant que 3 rondes sont suffisantes et nécessaires pour diagnostiquer un hypercube assumant que le nombre de fautes n'exède pas $\log N - \lceil \log(\log N - \lceil \log \log N \rceil + 4) \rceil + 2$. Ils ont également diagnostiqué un CCC de dimension supérieure à 3 en 3 rondes.

Okashita, Araki, et Shibata [21] ont considéré quant à eux le diagnostic adaptatif des réseaux modélisés sous forme de papillons (i.e. : Butterfly Network) en utilisant peu de tests.

2.6 Le temps dans le diagnostic

Le temps est un autre paramètre qui est intéressant dans un diagnostic. Nous considérons généralement qu'un test se fait dans une unité de temps (ronde) et que les tests

(u, v) et (u', v'), où tous les processeurs u, v, u', v' sont distincts, peuvent être exécutés en même temps.

Le temps peut être un paramètre extrêmement important dépendamment des contextes. Par exemple, considérons un robot qui prend des échantillons dans un environnement radioactif et dont certains des processeurs sont défectueux. Ce robot doit effectuer un diagnostic et s'assurer d'éliminer de son réseau les processeurs défectueux. Toutefois, étant donné qu'il est dans un environnement extrêmement influent sur le fonctionnement des processeurs, il doit s'assurer de faire le diagnostic très rapidement afin qu'un processeur ne change pas de statut durant l'exécution du diagnostic.

Beigel, Kosaraju et Sullivan [4] ont montré un diagnostic adaptatif en un nombre constant de rondes. Beigel, Margulis et Spielman [5] ont démontré que dans un environnement déterministe, on peut faire un diagnostic en 84 rondes pour n'importe quel nombre de processeurs dans le réseau. Ils ont également démontré que, si le diagnostic est probabiliste, 32 rondes seulement sont suffisantes et que la borne inférieure générale possible est de 3 rondes.

Beigel, Hurwood et Kahale [3] ont finalement introduit une méthode de diagnostic permettant d'optimiser les précédents résultats. Ils ont démontré que 4 rondes sont suffisantes et nécessaires pour faire un diagnostic si $2\sqrt{2n} \leq t \leq 0.03n$, que 5 rondes sont nécessaires si $t \geq 0.49n$ et que 10 rondes sont suffisantes pour $t < \frac{n}{2}$ si on utilise une approche probabiliste. Finalement, 13 rondes sont suffisantes pour tous $t < \frac{n}{2}$, même avec une approche déterministe.

Toutefois, tous ces algorithmes ne tiennent pas compte du nombre de tests.

2.7 Le diagnostic distribué

Un des désavantages du modèle PMC est que nous avons besoin d'un évaluateur centralisé pour regrouper et analyser les syndromes obtenus pour identifier les bons et les mauvais processeurs. Cet évaluateur central doit non seulement être extrêmement fiable et efficace mais il doit aussi avoir une communication fonctionnelle avec chacun des autres processeurs du système. Cette approche est difficile et dispendieuse à implanter dans les systèmes actuels et est, par conséquent, un point faible de ce modèle. Toutefois, il existe une alternative à cette méthode : le diagnostic distribué.

Le diagnostic distribué est une méthode qui permet à chaque processeur d'avoir son propre diagnostic du système. Les processeurs communiquent mutuellement les résultats de tests et chaque bon processeur doit faire un diagnostic correct du système. Il y a un problème toutefois avec cette approche. L'utilisateur ne peut pas savoir si le processeur auquel il s'adresse est un processeur fautif ou correct et ne peut pas, par conséquent, se fier au diagnostic de ce processeur sans le tester au préalable.

Kuhl et Reddy [17] ont introduit le terme diagnostic distribué. Selon leur modèle, un processeur a les informations concernant son voisinage uniquement. Ce qui veut dire qu'il n'a qu'un accès direct au processeur incident et un accès indirect aux autres processeurs du système. Dans ce modèle, un processeur devrait être capable indépendamment de diagnostiquer le reste du système.

Kreutzer et Hakimi [15] ont déterminé que le nombre minimal de liens entre l'évaluateur et le reste des processeurs du système dans un environnement où chaque processeur a un diagnostic complet du système est de $t + 1$ liens vers des différents processeurs dans le cas où l'utilisateur peut tester les processeurs et de $2t + 1$ liens dans le cas où l'utilisateur doit compiler des statistiques pour savoir qui dit la vérité et qui ment.

Ciompi et al. [10] ont suggéré une approche où les tests sont faits comme dans le cas général mais les résultats ne sont pas envoyés directement à l'évaluateur. Un consensus distribué semblable à l'accord byzantin est utilisé pour retourner les résultats et tous les processeurs ont la même évaluation du système.

Chapitre 3

Méthodologie, modèle et terminologie

3.1 Objectifs

Notre premier objectif est l'analyse de la densité minimale théorique d'un réseau pour que le diagnostic utilisant le nombre minimal de tests soit possible dans l'environement déterministe, ainsi que la conception des algorithmes de diagnostic déterministes travaillant dans des réseaux à faible densité et dans certains cas avec densité optimale.

Le deuxième objectif est la construction des algorithmes rapides de diagnostic, tout en gardant le nombre minimal possible de tests. Le temps et le nombre de tests devaient être évalués analytiquement.

Le troisième objectif est la construction d'un simulateur qui permet de comparer nos algorithmes de façon expérimentale avec ceux déjà existants, tant sur le plan du nombre moyen de tests que sur le plan du nombre de rondes.

3.2 Méthodologie

Les deux premiers objectufs ont été réalisés à l'aide des outils analytiques. Dans le cas du premier objectif, nous avons établi une borne inférieure sur la densité du réseau permettant un diagnostic qui utilise un nombre minimal de tests. Nous avons construit des réseaux de faible densité et conçu des algorithmes travaillant pour ces réseaux et utilisant un nombre minimal de tests. L'exactitude de ces algorithmes a été prouvé analytiquement. Dans le cas du deuxième objectif, nous avons conçu un algorithme rapide utilisant un nombre minimal de tests. Son temps de fonctionnement ainsi que le nombre de tests utilisé ont été estimés à l'aide d'une analyse combinatoire.

La méthodologie utilisée dans le cas du troisième objectif était expérimentale. Nous avons conçu un simulateur et nous avons effectué des simulations de nos algorithmes pour obtenir des résultats statistiques comparant leur efficacité moyenne.

Le simulateur a été créé en langage Java avec un environnement convivial. Nous avons implanté l'algorithme de diagnostic de Blecher [6] ainsi que ceux parmis les algorithmes de diagnostic que nous avons trouvés dont le fonctionnement diffère de façon significative de l'algorithme de Blecher. Nous avons comparé le nombre de tests moyen et le nombre de rondes de ces algorithmes.

Le simulateur a les propriétés suivantes :

1. Un paramètre n, soit le nombre de processeurs du réseau pouvant être ajusté aux valeurs $n < 1000000$.
2. Un paramètre t, soit le nombre de processeurs défectueux en pire cas pouvant être ajusté aux valeurs $t < 500000$ et ne pouvant en aucun cas égaler ou dépasser 50 % du nombre total de processeurs.
3. Un paramètre x, soit un nombre fixe de processeurs défectueux et ne pouvant en aucun cas dépasser le paramètre t.
4. Un générateur aléatoire de pannes.
5. Une sélection des algorithmes que nous désirons exécuter sur le graphe donné.
6. Un résultat statistique sur le nombre d'étapes ainsi que le nombre moyen de tests.

3.3 Modèle

Nous utilisons le modèle PMC résumé ci-dessous.

On considère un graphe orienté $G^* = (V, E^*)$, dont les sommets sont des processeurs. Un arc (v_i, v_j) signifie que le processeur v_i effectue un test sur le processeur v_j. Le résultat de ce test peut être 0 (Le processeur v_i considère le processeur v_j comme bon) ou 1 (Le processeur v_i considère le processeur v_j comme mauvais). Le résultat du test (v_i, v_j) est dénoté par $S(v_i, v_j)$. Les résultats de tests sont sujets aux règles suivantes :

$$S(v_i, v_j) = \begin{cases} 0 & \text{si } v_i \text{ et } v_j \text{ sont corrects} \\ 1 & \text{si } v_i \text{ est correct et } v_j \\ & \text{est fautif} \\ 0 \vee 1 & \text{Pour les autres cas} \end{cases}$$

Autrement dit, les bons processeurs sont complètement fiables comme testeurs et les mauvais processeurs sont complètement non fiables comme testeurs.

Pour un ensemble de tests donnés, représenté par un graphe de tests $G^* = (V, E^*)$, chaque ensemble de résultats de tous ces tests est appelé un syndrome. Formellement, un syndrome est une fonction $S : E \rightarrow \{0, 1\}$. Un syndrome S est dit compatible avec l'ensemble de pannes F s'il peut être généré en supposant que les mauvais processeurs sont exactement ceux dans F.

On dit qu'un t-diagnostic est possible s'il n'existe pas deux ensembles de pannes F_1 et F_2 de grandeur au plus t, tel que $F_1 \neq F_2$ et F_1 et F_2 sont compatibles avec le même syndrome. La tâche du t-diagnostic dans ce cas est de trouver, pour chaque syndrome S compatible avec un ensemble de pannes F de grandeur au plus t, cet unique ensemble F. Pour qu'un t diagnostic soit possible, nous devons imposer la borne $t < \frac{n}{2}$.

3.4 Terminologie

Rappelons la terminologie suivante :

$G = (V, E)$: Graphe orienté de tests G avec un ensemble de processeurs V et un ensemble d'arcs E. Un processeur v_i est dit adjacent à un processeur v_j s'il y a un lien direct entre v_i et v_j.

test(v_i, v_j) : Le processeur v_i effectue un test sur le processeur v_j.

$S(v_i, v_j)$: Le résultat du test(v_i, v_j). Est égal à 0 si v_i évalue v_j comme bon et 1 autrement.

Chapitre 4

Diagnostic optimal des systèmes multiprocesseurs en grille

Le but de ce chapitre est de construire et analyser des algorithmes de diagnostic qui fonctionnent pour une classe de réseaux importants : les grilles. Puisque chaque grille a des sommets de degré 2, nous nous limitons au 2-diagnostic.

4.1 Algorithme Diagnostic NonAda-RingTest

Dans cette section, nous présentons un algorithme optimal non-adaptatif pour un anneau de grandeur n dénoté par $\mathcal{R}_n$. Ce résultat nous servira pour analyser le cas de la grille.

Algorithme NonAda-RingTest($\mathcal{T}_n$)
Faire un test sur chaque voisin gauche et droit de chaque processeur de l'anneau.

Théorème 4.1.1. *Un anneau $\mathcal{R}_n$ avec $n \geq 5$ processeurs peut être correctement diagnostiqué en utilisant l'algorithme Diagnostic NonAda-RingTest, en supposant que le nombre de fautes est au plus 2. Le nombre de tests est égal à $2n$.*

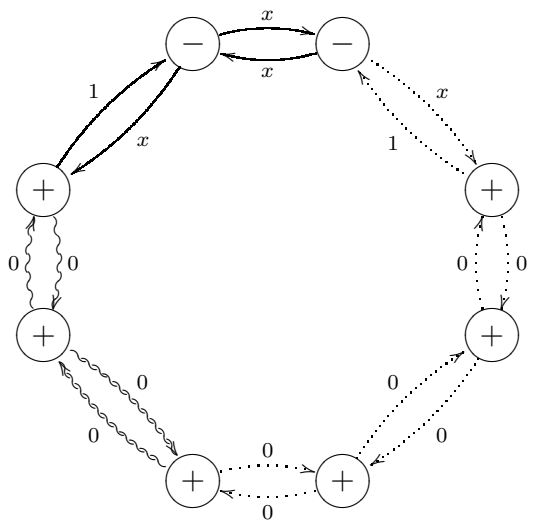

FIG. 4.1 – Cas 1 : où un bon processeur est testé par 2 bons processeurs

Démonstration. Si nous parvenons à identifier avec certitude un processeur correct dans l'ensemble de processeurs, alors il existe une séquence de tests qui pourra identifier avec succès 2 processeurs défectueux. Un processeur correct a deux arêtes incidentes. Il teste correctement ses deux voisins. Par induction, nous pouvons démontrer qu'il y aura une séquence de bons processeurs sur chaque côté du processeur de départ qui finira par trouver un processeur défectueux ou par revenir au point de départ. Puisqu'il y a deux processeurs défectueux en pire cas et que nous avons deux chemins possibles, nous pouvons localiser les processeurs défectueux. Il reste à montrer que nous pouvons identifier un processeur correct.

Si un processeur est testé par ses deux arêtes incidentes comme correct, alors ce processeur doit être bon puisque $t \leq 2$. S'il n'était pas bon, cela impliquerait qu'il y a au moins 3 processeurs défectueux. De plus, si deux processeurs se testent mutuellement comme corrects et qu'il existe un lien 1 n'étant pas incident à ces deux processeurs, alors les deux processeurs doivent être corrects. En effet, si deux processeurs se testent correctement, c'est qu'ils sont soit tous les deux bons, soit tous les deux mauvais. Mais étant donné qu'il y a un lien 1 ailleurs, il doit y avoir un autre processeur mauvais. Par conséquent, ces deux processeurs doivent être bons, sinon cela impliquerait qu'il y a au moins 3 processeurs défectueux.

Ces deux cas couvrent tous les cas possibles. Dans n'importe quelle configuration d'anneaux où $n \geq 2t + 1$ et où il y a au moins un processeur défectueux, il doit y avoir au moins un lien mutuel qui donne 0 et une arête avec étiquette 1 qui n'est pas incidente à ces deux processeurs. L'autre cas couvre le cas ou aucune faute n'est trouvée. □

Lemme 4.1.2. *[25] Le nombre minimal de tests nécessaires pour effectuer correctement un t-diagnostic non-adaptatif dans chaque graphe de n processeurs avec au plus t fautes est tn*

Démonstration. Supposons que v est un processeur tel que l'ensemble S_v de testeurs de v a moins que t éléments. Considérons le syndrome suivant : $S(u, w) = 1$ si $w \in S_v$ et $S(u, w) = 0$ si $w \notin S_v$. Ce syndrome est compatible avec deux ensembles de pannes différents : S_v et $S_v \cup \{v\}$. Chacun d'eux a au plus t éléments. Donc, le t-diagnostic est impossible. Cela implique que chaque processeur doit être testé par au moins t autres. Donc il y a au moins nt tests. □

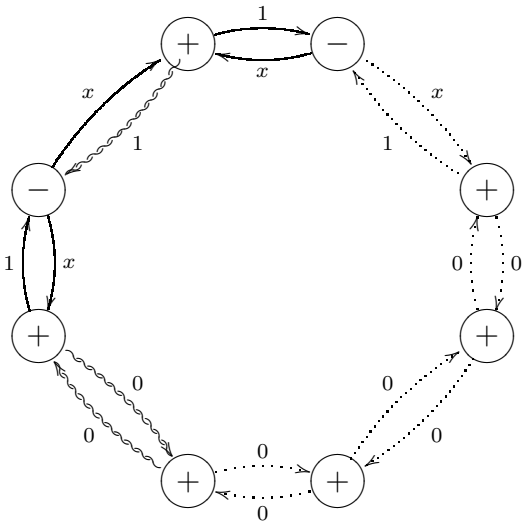

FIG. 4.2 – Cas 2 : où 2 bons processeurs se testent mutuellement

4.2 Diagnostic pour les systèmes en grille

Dans cette section, nous donnons deux algorithmes optimaux, un non-adaptatif et l'autre adaptatif pour des grilles de k × m. Puisque le degré de chaque coin d'une grille

d'un minimum de 2×2 est de 2 et est minimal, nous devons assumer que $t = 2$. Nous avons aussi besoin d'assumer que $n \geq 2t + 1 = 5$. Donc, nous ne pouvons effectuer de diagnostic pour une grille de 2×2 puisque $2 \times 2 = 4 < 5$ et, par conséquent, la grille minimum utilisable sera la grille de 2×3. Ci-dessous, nous considérons des grilles de $k \times m$, où $k \geq 2$, $m \geq 3$. Nous dénotons $n = km$.

Nous dénoterons G une grille avec n processeurs et G' le graphe résultant de G par l'élimination d'un coin et de ses arêtes incidentes.

4.2.1 Algorithme Diagnostic NonAda-GrilleTest

Procédure NonAda-GrilleTest($\mathcal{G}_n$)

Étape 1 :
Si (n est pair) **Alors**
Construire un cycle hamiltonien dans G.
Sinon
Construire un cycle hamiltonien dans G'.

Étape 2 :
Appliquer l'algorithme NonAda-RingTest sur l'anneau trouvé.

Étape 3 :
Si (n est impair) **Alors**
tester le coin enlevé au début par ses deux voisins dans G.

Lemme 4.2.1. *Dans une grille $k \times m$ ayant un nombre de processeurs pairs, il existe un cycle hamiltonien. (Voir fig.4.3)*

Démonstration. Présentons la grille comme une matrice $(k \times m)$ k étant pair, le processeur du coin bas gauche étant la coordonnée $(1, 1)$ et celle du processeur haut droit comme (k, m). La séquence $(1, 1)$, $(2, 1)$, ..., $(m, 1)$, $(m, 2)$, $(m - 1, 2)$,..., $(2, 2)$, $(2, 3)$, $(3, 3)$, ..., $(2, k - 1)$, $(3, k - 1)$, ..., $(m, k - 1)$, (m, k), $(m - 1, k)$, ..., $(1, k)$, $(1, k - 1)$, ..., $(1, 1)$ est un cycle hamiltonien pour une grille minimum de 4×4. Le cas de la grille 2×4 est évident. □

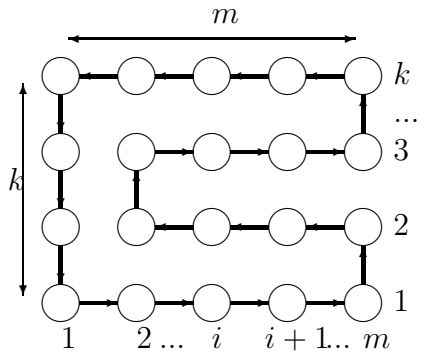

FIG. 4.3 – Un cycle hamiltonien pour une grille avec un nombre de noeuds pair

Lemme 4.2.2. *Soit G' le graphe résultant d'une grille $k \times m$, où $n = km$ est impair, par l'élimination d'un coin et de ses arêtes incidentes. Il existe un cycle hamiltonien dans G'. (Voir fig.4.4)*

Démonstration. Présentons la grille comme une matrice $(k \times m)$, le processeur du coin bas gauche étant la coordonnée $(1, 1)$ et celle du processeur haut droit comme (k, m). Enlevons le processeur $(1, k)$ de cette matrice. La séquence $(1, k-1)$, $(1, k-2)$, ..., $(1, 1)$, $(2, 1)$, $(3, 1)$, ..., $(m, 1)$, $(m, 2)$, $(m-1, 2)$, ..., $(2, 2)$, $(2, 3)$, ..., $(m-1, 3)$, $(m, 3)$, ..., $(m-1, k-2)$, $(m, k-2)$, $(m, k-1)$, (m, k), $(m-1, k)$, $(m-1, k-1)$, $(m-2, k-1)$, ..., $(2, k-1)$, $(1, k-1)$ est un cycle hamiltonien pour une grille minimum 5×3. Le cas de la grille 3×3 est évident. □

Théorème 4.2.3. *Soit $k \geq 2, m \geq 3$.Le diagnostic non-adaptatif peut être fait correctement dans la grille $k \times m$ en utilisant la procédure NonAda-GrilleTest en supposant que le nombre de fautes est au plus 2. Le nombre de tests est égal à $2n$.*

Démonstration. Nous savons que l'algorithme non-adaptatif pour les anneaux de n processeurs utilise $2n$ tests. Par conséquent, tel que prouvé dans les lemmes 4.2.1 et 4.2.2, nous pouvons utiliser l'algorithme non-adaptatif pour les anneaux dans ce contexte.

Si le nombre de processeurs dans la grille est pair, alors nous appelons l'algorithme pour le cycle hamiltonien trouvé. Sinon, nous appelons l'algorithme pour le cycle hamiltonien

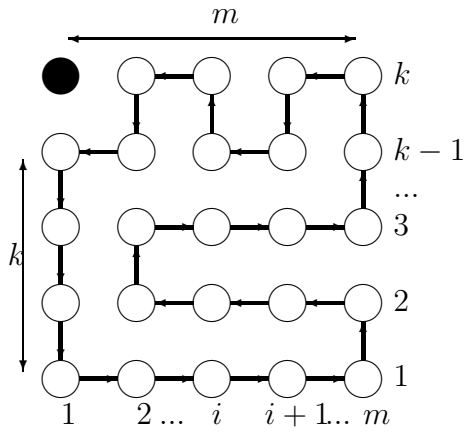

FIG. 4.4 – Un cycle hamiltonien pour une grille avec un nombre de noeuds impair

trouvé pour G' et nous ajoutons deux tests vers le processeur enlevé par ses deux voisins.

Le nombre de tests dans les deux cas est $2n$. Dans le second cas, si deux processeurs défectueux sont identifiés dans le cycle, le processeur enlevé est correct. Sinon, un de ses voisins est correct et le statut du processeur enlevé est celui indiqué par ce voisin. □

Corollaire 4.2.4. *Le lemme 4.1.2 et le théorème 4.2.3 impliquent que le nombre minimum de tests nécessaires pour faire un 2-diagnostic non-adaptatif dans n'importe quelle grille avec $n \geq 5$ processeurs est $2n$.*

4.2.2 Algorithme Diagnostic Ada-GrilleTest

Procédure Ada-GrilleTest($\mathcal{G}_n$)

Étape 1 :
Si (n est pair) **Alors**
Construire un cycle hamiltonien dans G.
Sinon

Construire un cycle hamiltonien dans G'.

Étape 2 :
Appliquer l'algorithme Ada-RingTest de [23] sur l'anneau trouvé.

Étape 3 :
Si (n est impair et le nombre de processeurs défectueux est ≤ 1) **Alors**
tester le coin enlevé au début par le voisin correct trouvé dans G.

Théorème 4.2.5. *Soit $k \geq 2, m \geq 3$. Le diagnostic adaptatif peut être fait correctement dans la grille $k \times m$ en utilisant la procédure Ada-GrilleTest en supposant que le nombre de fautes est au plus 2. Le nombre de tests en pire cas est $n + 1$*

Démonstration. La preuve de l'efficacité de l'algorithme Ada-RingTest a été faite dans l'article de Kranakis, Pelc et Spatharis [23]. Il a été prouvé qu'on peut trouver le nombre de processeurs défectueux avec $t \leq 2$ en $n + 1$ tests dans un anneau de grandeur n.

Si le nombre de processeurs dans la grille est pair, alors il existe un cycle hamiltonien que nous pouvons utiliser pour appeler la procédure Ada-RingTest utilisant $n + 1$ tests. Sinon, il existe un cycle hamiltonien dans G' que nous pouvons utiliser pour appeler la procédure Ada-RingTest utilisant $n - 1 + 1$ tests. Ensuite, nous pouvons vérifier le processeur enlevé avec un seul processeur correct si nécessaire. Le nombre total de tests est donc $n - 1 + 1 + 1 = n + 1$. □

Théorème 4.2.6. *Le nombre minimum de tests nécessaires pour faire un 2-diagnostic adaptatif dans n'importe quelle grille avec $n \geq 5$ processeurs est $n + 1$.*

Démonstration. La borne inférieure de $n+t-1$ a été prouvée par Blecher [6] et Wu [26] pour le graphe complet. Elle est donc valide aussi pour la grille. Pour $t = 2$, cette borne inférieure est donc $n + 1$. □

4.2.3 Diagnostic adaptatif rapide pour les grilles

Nous utiliserons les résultats sur le diagnostic des cycles prouvés par Kranakis, Pelc et Spataris [23] et déduirons le théorème suivant.

Théorème 4.2.7. *L'algorithme Ada-RingTest permet de diagnostiquer un cycle avec un nombre de processeurs impairs en* 4 *rondes au plus et un cycle avec un nombre de processeurs pairs en* 3 *rondes au plus.*

Démonstration. Dans la phase 1 de l'algorithme Ada-RingTest, tous les tests peuvent être effectués en deux rondes dans un cycle pair et en trois rondes dans un cycle impair.

La phase 2 de l'algorithme Ada-RingTest ne demande qu'un test supplémentaire. □

Théorème 4.2.8. *L'algorithme Ada-GrilleTest permet de diagnostiquer une grille en* 3 *rondes.*

Démonstration. Puisque l'algorithme Ada-GrilleTest utilise un cycle plongé dans le graphe G ou G' et que le cycle est toujours un cycle pair, le théorème 4.2.7 permet de diagnostiquer ce cycle en 3 rondes. Dans le cas de la grille paire, puisque le problème se résume au problème du cycle pair, la preuve est faite.

Dans le cas de la grille impaire G, le graphe G' est un graphe où l'on peut plonger un cycle avec un nombre de processeurs pair supérieur ou égal à 8.

Dans ce cas, il y a 4 situations possibles pour diagnostiquer la grille G en utilisant la procédure Ada-RingTest sur G'.
Voici les 4 situations possibles :

cas 1 : $a \rightarrow^1 b \rightarrow^1 c \rightarrow^1 d \rightarrow^0 e$
Alors, faire le test(e, d).
Dans ce cas, il y a deux fautes dans l'anneau et le processeur dans $G \backslash G'$ n'a pas besoin d'être diagnostiqué.

cas 2 : $a \rightarrow^1 b \rightarrow^1 c \rightarrow^0 d$
Alors faire le test(d, c).
Dans ce cas, la situation problématique arrive lorsque le processeur b et le processeur d sont adjacents au processeur dans $G \backslash G'$. Dans ce cas, faire le test en parallèle du processeur d vers le processeur dans $G \backslash G'$ et également prendre un autre processeur que le processeur b ou d que nous appelerons d' adjacent au processeur c. Ce processeur doit exister puisque le processeur c ne se situe pas sur un coté d'une grille de dimension minimale 3×3 et que c a donc 4 processeurs adjacents. Également, ce processeur doit être bon puisque la faute restante ne peut être que parmi c et le processeur dans $G \backslash G'$.

cas 3 : $a \rightarrow^1 b \rightarrow^0 c \rightarrow^1 d \rightarrow^0 e$
Alors faire le test(e, d).
Dans ce cas, les deux fautes se trouvent dans l'anneau et le processeur dans $G \backslash G'$ n'a pas besoin d'être testé puisqu'il doit être bon.

cas 4 : $a \rightarrow^1 b \rightarrow^0 c \rightarrow^0 d$
Alors faire le test(d, c).
Dans ce cas, il y a au moins un processeur adjacent au processeur dans $G \backslash G'$ qui doit être bon. Celui-ci fait le test sur le processeur dans $G \backslash G'$. Si ce processeur est le processeur d, cela implique que le processeur c n'est pas sur un côté de la grille et qu'il a, par conséquent, 4 voisins. Dans ce cas, tester dans $G \backslash G'$ avec d et tester c par l'un de ses autres voisins corrects. □

Le théorème que nous laissons sans démonstration montre que l'algorithme de diagnostic adaptatif Ada-GrilleTest est optimal sur le nombre de rondes.

Théorème 4.2.9. *[5] Le nombre de rondes minimum pour faire n'importe quel diagnostic adaptatif est 3.*

Chapitre 5

Diagnostic adaptatif optimal dans les réseaux à faible densité

Dans ce chapitre, nous atteindrons le but de diminuer la densité du réseau dans lequel un t-diagnostic utilisant le nombre minimal de $n + t - 1$ tests peut fonctionner. Nous construisons trois réseaux : un avec nt liens, un avec $nt - t$ liens et le dernier avec $\lceil \frac{nt}{2} \rceil$ liens. Pour tous ces réseaux, nous construisons des algorithmes de t-diagnostic utilisant $n + t - 1$ tests. Le nombre de liens dans le premier réseau est légèrement plus grand que dans le deuxième mais il a l'avantage d'être régulier, ce qui n'est pas le cas du deuxième réseau. Par contre, le troisième réseau est optimal du point de vue de la densité et il est régulier mais notre t-diagnostic fonctionne uniquement pour $t < 2\sqrt{n} - 2$.

5.1 Architecture de graphe complet

Nous commençons par la présentation de l'algorithme de diagnostic adaptatif de Blecher [6], qui fonctionne dans un graphe complet et qui utilise $n + t - 1$ tests en pire cas à condition que $t < \frac{n}{2}$

Algorithme Diagnostic-Adaptatif de Blecher($\mathcal{T}_n$)

Soit 1,2,...,n une énumération des processeurs. Effectuer les tests (2, 1), (3, 1), (4, 1), (5,1)..., jusqu'au premier des événements suivants.

Événement A : t tests donnent le résultat 0.
Alors le processeur 1 doit être bon, car autrement il y aurait plus de t mauvais processeurs.

Tous les processeurs i, tel que le test (i, 1) a été effectué et a donné le résultat 1, sont mauvais. Soit T l'ensemble de ces processeurs. Effectuer tous les tests (1, j) pour $j \notin T \cup \{1\}$. Diagnostiquer j comme bon si le résultat est 0 et comme mauvais si le résultat est 1.

Événement B : Le nombre de résultats 1 dépasse le nombre de résultats 0.
Dans ce cas, s'il y a m résultats 1, alors il y a $m-1$ résultats 0. Soit A l'ensemble des processeurs j parmi {2,. . .,2m}, pour lesquels $S(j,1) = 0$ et B l'ensemble des processeurs j parmi {2,. . .,2m}, pour lesquels $S(j,1) = 1$. Parmi les processeurs 1, 2, . . ., $2m$, il y a au moins m mauvais processeurs. Puisque $t < \frac{n}{2}$, on a $t - m < \frac{n-2m}{2}$. Donc, on peut appliquer l'algorithme récursivement, pour effectuer le diagnostic dans le système de $n - 2m$ processeurs avec au plus $t - m$ mauvais processeurs. Il reste à diagnostiquer les processeurs 1, 2, . . ., $2m$. Choisir n'importe quel processeur i diagnostiqué comme bon dans l'appel récursif et effectuer le test(i, 1). Il y a deux cas :

Cas 1. $S(i,1) = 0$.
Effectuer tous les tests (i,j), pour les processeurs dans A. Diagnostiquer le processeur 1 comme bon, tous les processeurs $j \in A$ selon le résultat du test (i,j) et tous les processeurs dans B comme mauvais. Puisque le processeur i est bon, ce diagnostic doit être correct.

Cas 2. $S(i,1) = 1$.
Effectuer tous les tests (i,j), pour les processeurs dans B. Diagnostiquer le processeur 1 comme mauvais, tous les processeurs $j \in B$ selon le résultat du test (i,j) et tous les processeurs dans A comme mauvais. Puisque le processeur i est bon, ce diagnostic doit être correct.

Théorème 5.1.1. *[6] L'algorithme Diagnostic-Adaptatif de Blecher effectue correctement un t-diagnostic, pour $t < \frac{n}{2}$, et utilise au plus $n + t - 1$ tests en pire cas.*

Démonstration. L'exactitude a été justifiée dans la formulation de l'algorithme. Il reste à estimer le nombre de tests.

Démonstration.

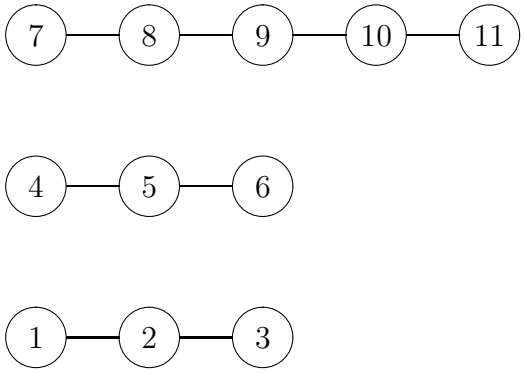

FIG. 5.1 – Répartition des processeurs pour un 1-diagnostic où $n = 11$.

Événement A. Le nombre total de tests effectués est $t + |T| + n - |T| - 1 = n + t - 1$.
Événement B. Le nombre de tests avant l'appel récursif est $2m - 1$. Le nombre de tests pendant l'appel récursif est au plus $(n - 2m) + (t - m) - 1$. Le nombre de tests après l'appel récursif est au plus $1 + m$. Le total est donc au plus $n + t - 1$. □

5.2 Densité minimum d'un graphe pour un t-diagnostic

Dans cette section, nous allons établir la densité minimum d'un graphe pour accomplir un t-diagnostic. Nous allons tout d'abord traiter le cas du 1-diagnostic et ensuite les autres cas.

Théorème 5.2.1. *Soit $n = 3x + r$, où $r < 3$ et $x \geq 1$. Alors il existe un graphe avec $2x + r$ arêtes qui permet un 1-diagnostic en au plus n tests.*

Considérons une énumération des processeurs de 1 à n. Considérons une partition de tous les processeurs en ensemble de grandeur 3, le dernier ensemble étant de grandeur $3 \leq x \leq 5$. Dans chaque ensemble, les processeurs sont reliés en ligne. (voir Fig. 5.1). Dans chaque ligne de processeurs $[a_1, a_2, a_3, \ldots]$, le premier processeur teste le suivant. Si $S(a_1, a_2) = 1$, alors faire test (a_3, a_2). Il y a deux cas possibles :

Cas 1 : $S(a_3, a_2) = 1$.
Alors, le processeur a_2 doit être mauvais. Autrement, cela impliquerait que les processeurs a_1 et a_3 sont mauvais mais $t \leq 1$.

Cas 2 : $S(a_3, a_2) = 0$.
Alors, le processeur a_1 doit être mauvais sinon, cela implique que nous avons $t > 1$.

Si $S(a_1, a_2) = 0$, alors faire test(a_2, a_3) et si la ligne contient plus de 3 processeurs, effectuer les tests (a_i, a_{i+1}) pour $i < 5$. Aussitôt que $S(a_i, a_{i+1}) = 1$, le processeur a_{i+1} doit être mauvais, sinon, cela impliquerait que $t > 1$.

L'exactitude a été démontrée dans la formulation de l'algorithme, il nous reste à estimer le nombre d'arêtes et le nombre de tests. Soit $n = 3x+r$, où $r < 3$. Notre graphe a alors $2(x-1)+3+r-1 = 2x+r$ arêtes.

Pour estimer le nombre total de tests, nous devons estimer le nombre de tests sur chaque ligne. Il y a deux cas possibles sur une ligne. Le premier cas est que lorsque nous faisons test(a_1, a_2), test(a_2, a_3), etc en séquences sur les lignes, un résultat de tests égale 1. Dans ce cas, seulement cette ligne aura besoin de 3 tests en pire cas et chacune des autres en aura besoin de 3 en pire cas puisque la faute se situe sur la ligne ayant un résultat de test 1. Nous aurons donc $2(x-1)+3+r \leq 3x+r$ pour $x > 0$. Dans le cas ou nous n'aurons aucun résultat 1 sur chaque ligne, alors le deuxième processeur de chaque ligne devra tester le premier pour un total de n tests. □

Théorème 5.2.2. *Soit $n = 3x + r$, où $r < 3$ et $x \geq 1$. Alors le nombre minimum d'arêtes dans un graphe de n noeuds qui permet un 1-diagnostic est $2x + r$.*

Démonstration. Nous avons démontré que c'est possible avec $2x+r$ arêtes. Il nous reste maintenant à démontrer que ce n'est pas possible avec moins d'arêtes.

Supposons que nous ayons $2x+r-1$ arêtes, alors cela implique qu'un de nos processeurs est soit isolé ou qu'un groupe de 2 processeurs au maximum existe quelque part. Or, nous savons que pour faire un diagnostic, nous avons besoin de $2t + 1$ processeurs reliés entre eux avec un degré d'au moins t. Ce processeur ou ce groupe de 2 processeurs ne répond pas à cette contrainte. Contradiction ! □

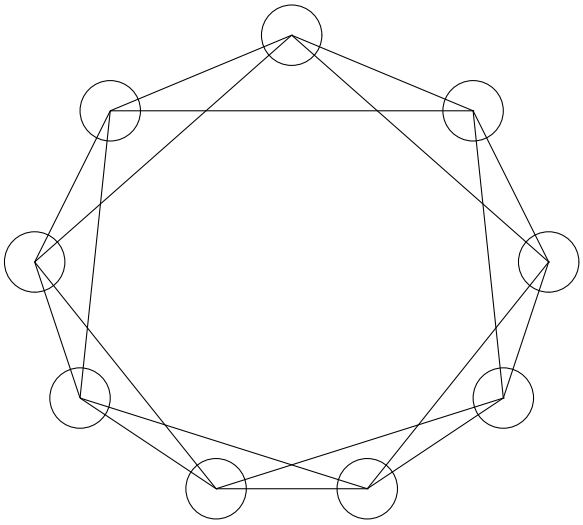

FIG. 5.2 – Graphe régulier t-diagnosticable avec nt liens où $t = 4$.

Théorème 5.2.3. *Chaque graphe qui assure un t-diagnostic pour $t \geq 2$ doit avoir au minimum $\lceil \frac{nt}{2} \rceil$ liens.*

Démonstration. Supposons un graphe qui a moins de $\lceil \frac{nt}{2} \rceil$ liens et qui est t-diagnosticable pour $t \geq 2$. Alors, il existe un processeur v qui a moins que t arêtes incidentes. Par conséquent, v est un processeur tel que l'ensemble S_v de testeurs de v a moins que t éléments. Considérons le syndrome suivant : $S(u, w) = 1$ si $w \in S_v$ et $S(u, w) = 0$ si $w \notin S_v$. Ce syndrome est compatible avec deux ensembles de pannes différents : S_v et $S_v \cup \{v\}$. Chacun d'eux a au plus t éléments. Donc, le t-diagnostic est impossible. □

5.3 Nouvelles architectures de graphes pour le diagnostic optimal

Dans cette section, nous proposons différentes architectures de graphes avec $O(nt)$ liens. Nous présentons également un algorithme pour chacune de ces architectures qui utilise $n + t - 1$ tests en pire cas à condition que $t < \frac{n}{2}$.

5.3.1 Algorithme Diagnostic Ada-Diagnostic-Régulier

Nous présentons ici un algorithme qui permet de faire le diagnostic d'un graphe G régulier avec nt liens(régulier sauf dans le cas où n et t sont impairs, auquel cas il y a un noeud de degré différent que les autres) en utilisant le nombre minimal de tests où G est défini comme suit :
Supposons une énumération de processeurs de 1 à n. Chaque processeur possède un lien vers les t processeurs suivants et les t processeurs précédents où le processeur successif à n est le processeur 1 et le processeur précédent 1 est le processeur n. (voir Fig. 5.2).

Procédure Ada-Diagnostic-Régulier($\mathcal{G}_n$)

Étape 1 :
Effectuer les tests (2, 1), (n, 1), (3, 1), (n-1,1)..., jusqu'au premier des événements A ou B de l'algorithme de Blecher présenté à la section précédente (voir fig.5.3).

Étape 2 :
Si (événement = B) **Alors**
Passer à l'étape 3.
Sinon
Le bon processeur est trouvé. Ce processeur va tester tous ses testeurs. Chaque bon processeur identifié pourra à son tour identifier d'autres processeurs et ainsi de suite jusqu'à ce que le graphe soit complètement diagnostiqué.

Étape 3 :
Soit x le processeur qui devait être le prochain testeur, X_g et X_d étant l'ensemble des processeurs sur lequel x possède des liens respectivement à gauche et à droite qui n'ont jamais été testés ou testeurs. x devient le nouveau processeur testé. x_{g_i} étant le i^{eme} processeur de X_g en partant de x.

Tant Que(événement $\neq$ A et événement $\neq$ B) **Faire**
Si ($|X_g| > 0$) **Alors**
Effectuer le test (x_{g_1}, x)
Sinon Si ($|X_d| > 0$) **Alors**
Effectuer le test (x_{d_1}, x)
Sinon Fin Tant Que **Retour à l'étape 2.**

Aller à l'étape 2.

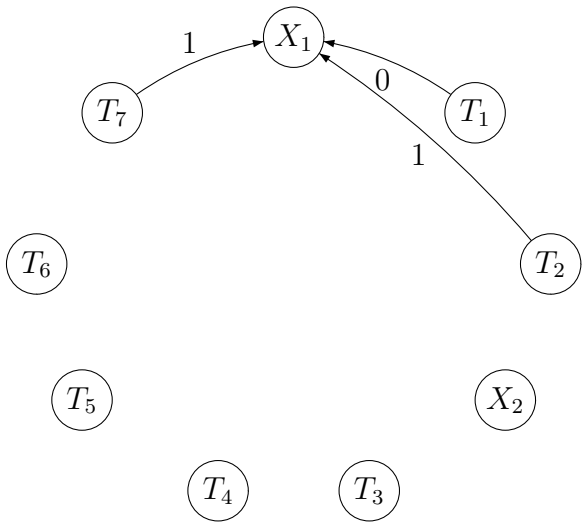

FIG. 5.3 – Graphe de test partiel pour un t-diagnostic.

Théorème 5.3.1. *Le graphe régulier G est t-diagnosticable avec $n + t - 1$ tests en utilisant la procédure Ada-Diagnostic-Régulier.*

Démonstration. Deux choses doivent être démontrées dans la preuve. Premièrement, qu'il est toujours possible d'effectuer un test vers le processeur X en partant d'un processeur qui n'a ni été testé, ni été testeur, tant que l'événement A ou B n'est pas survenu et deuxièmement, lorsqu'un bon processeur est trouvé, qu'il peut initier une séquence de tests qui identifiera tous les processeurs qui n'ont toujours pas été diagnostiqués.

Soit k, le nombre d'événements B qui se sont produits, m_i le nombre de tests 1 sur le processeur testé lors du i^{eme} événement B. Alors $\sum_{i=1}^{k} m_i \leq t$. Étant donné que dans un événement B, nous avons exactement un test 0 de moins qu'un test 1, nous pouvons en déduire l'équation $\sum_{i=1}^{k}(m_i - 1) \leq t - k$. nous avons k processeurs X testés. Par conséquent, nous aurons un maximum de $t + (t - k) + k = 2t$ tests pour trouver un bon processeur.

Au k^{ieme} événement B, nous aurons un bloc de $2(\sum_{i=1}^{k} m_i)$ processeurs dont chaque membre aura participé d'une quelconque façon à un test (en étant testeur ou testé). Nous pouvons éliminer ce bloc de processeurs puisque chaque processeur a déjà fait une action et doit être diagnostiqué plus tard. Par conséquent, si nous choisissons un nouveau processeur X à tester, il aura au moins $2t - 2(\sum_{i=1}^{k} m_i)$ processeurs qui ont un lien sur

le processeur X et qui ne sont pas dans le bloc éliminé. Supposons que $\sum_{i=1}^{k} m_i = 2t$ et que nous ayons un $k+1^{ime}$ événements B avec au moins un test 1. Cela implique que nous ayons t processeurs fautifs dans les k premiers événements et 1 processeur fautif dans l'événement actuel. Or nous aurons donc $t+1$ processeurs fautifs. Contradiction ! Il existe donc toujours un lien d'un processeur qui n'a pas agi jusqu'à maintenant pour tester le processeur X si l'événement A n'est pas survenu.

Finalement, lorsqu'un bon processeur est trouvé, il y a deux cas possibles pour compléter la phase de tests.

Cas 1 : Aucun événement B n'est survenu. Alors, le processeur X est bon et a exactement $2t$ processeurs incidents. Puisque chaque processeur a également $2t$ processeurs incidents et que nous avons au maximum t fautes, une séquence de tests pour obtenir le diagnostic peut être faite en partance du processeur X.

Cas 2 : Il y a eu i événements B successifs tel que $i > 0$. Soit le processeur α_i, le processeur testé au i^{eme} événement B. Le processeur α_i ou l'un de ses testeurs doit être bon. Ce bon processeur possède certainement un lien vers α_{i-1} car autrement, cela impliquerait que nous ayons plus de t fautes puisque t fautes sépareraient α_{i-1} du bon processeur plus au moins un mauvais processeur pour l'événement $i-1$.

Le bon processeur (α_i ou un de ses testeurs) effectue le test sur α_{i-1}. Si α_{i-1} est mauvais, alors le bon processeur va tester les testeurs de α_{i-1} jusqu'à ce qu'un bon processeur soit trouvé. Ce bon processeur fera le reste des tests pour les testeurs à l'événement $i-1$. Si α_{i-1} est bon, alors c'est α_{i-1} qui fera ces tests.

Nous reprenons le processus récursivement pour tester α_{i-2} et ainsi de suite jusqu'a ce que nous ayons identifié le processeur α_0 et ses testeurs. □

5.3.2 Algorithme Diagnostic Adaptatif Ada-Blecher-Raréfié

Dans cette section, nous présentons un graphe G avec $nt-t$ liens ainsi qu'un algorithme qui effectue correctement le t-diagnostic avec au plus $n+t-1$ tests dans ce graphe pour tous $t < \frac{n}{2}$. L'algorithme utilisé est inspiré de l'algorithme proposé par Blecher [6] qui a été introduit dans la section 5.1. L'algorithme nécessite toutefois une légère modification par rapport à la version de Blecher.

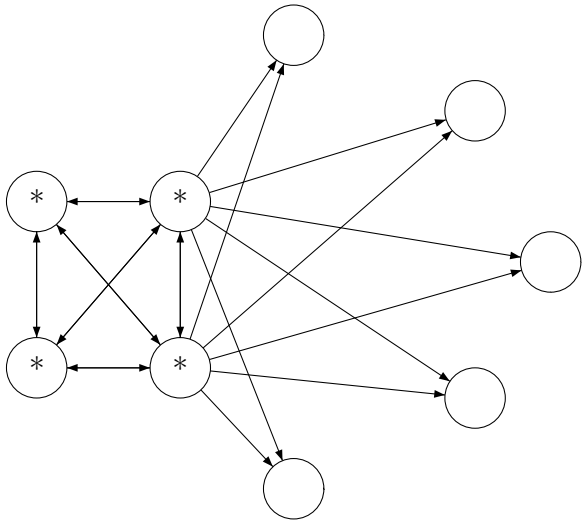

FIG. 5.4 – Graphe t-diagnosticable avec $nt - t$ liens où $n = 9, t \leq 2$.

Le graphe proposé (voir Fig. 5.4) est un graphe comprenant un sous-graphe complet contenant $2t$ processeurs. Nous aurons t processeurs de ce sous-graphe qui seront reliés aux $n - 2t$ autres processeurs qui sont externes au sous-graphe. Ce graphe contient donc un sous- graphe ayant $\frac{2t*(2t-1)}{2} = 2t^2 - t$ liens et le reste du graphe ayant $(n-2t)t = nt-2t^2$ liens. Pour un total de $2t^2 - t + nt - 2t^2 = nt - t$ liens.

Procédure Ada-Blecher-Raréfié($\mathcal{G}$)

Étape 1 :

Appliquer l'algorithme de Blecher sur le sous-graphe complet de $2t$ processeurs jusqu'à l'événement A ou jusqu'à ce que les $2t$ processeurs aient été testeurs ou testés.

Si (Événement = A) **Alors**

Diagnostiquer le reste des processeurs du sous-graphe complet et du sous-graphe externe

avec le bon processeur trouvé.

Sinon

Il y a t mauvais processeurs dans la clique.

Les processeurs externes sont bons.

Diagnostiquer les processeurs de la clique à l'aide d'un processeur externe jusqu'à ce qu'un bon processeur soit trouvé. Ce bon processeur fera le reste du diagnostic de la clique.

Théorème 5.3.2. *L'algorithme Ada-Blecher-Raréfié permet de faire le diagnostic dans le graphe G en utilisant $n + t - 1$ tests.*

Démonstration. Si l'événement A survient, alors la validité de l'algorithme est justifié comme suit. Puisqu'il y a au moins t liens en partance de t processeurs de la sous-clique vers chaque processeur externe de la sous-clique, soit ces t processeurs de la sous-clique sont défectueux, soit il existe un chemin connexe d'un bon processeur de la sous-clique vers ceux qui sont externes. Le seul cas problème est si l'événement A ne survient pas.

C'est dans ce cas où l'algorithme Ada-Blecher-Raréfié est modifié par rapport à l'algorithme de Blecher [6] pour qu'un des processeurs externes aux $2t$ processeurs du sous-graphe complet s'occupe de faire le test qui permet d'identifier les mauvais processeurs ainsi que les bons dans le sous-graphe complet. En effet, ces processeurs externes au sous-graphe complet doivent tous être bons puisque nous savons qu'il y a t mauvais processeurs dans le sous-graphe complet. De plus, n'importe quel test de l'un des $n - 2t$ processeurs externes au sous-graphe complet vers un des $2t$ processeurs du sous-graphe permet un diagnostic complet de la section d'un appel récursif de ce sous-graphe et de ses $2m$ processeurs. Puisque, dans cet appel récursif, nous savons qu'il y a m fautes et que nous pouvons identifier m bons processeurs, le reste des tests pour les autres appels récursifs seront faits par un bon processeur faisant partie des $2t$ processeurs du sous-graphe complet.

Il y a 6 cas possibles pour le test d'un processeur parmi les $2m$ d'un appel récursif. Soit x, le processeur testé dans cet appel récursif.

Cas 1. Nous testons comme bon un processeur ayant testé comme 0 le processeur x
Alors, puisque le processeur x est bon, tous les testeurs l'ayant testé comme mauvais sont mauvais, il y en a m, alors les testeurs l'ayant testé comme bon doivent être bons.

Cas 2. Nous testons comme mauvais un processeur ayant testé comme 0 le processeur x
Alors le processeur x doit être mauvais. S'il était bon, cela supposerait que les m testeurs qui l'ont testé comme mauvais sont mauvais et nous aurions $m + 1$ mauvais processeurs. Puisque le processeur x est mauvais, tous les testeurs qui l'ont testé comme bon sont

mauvais.

Cas 3. Nous testons comme bon un processeur ayant testé comme 1 le processeur x
Alors, les m testeurs qui ont testé le processeur x comme bon sont mauvais. Donc les testeurs qui ont testé le processeur x comme mauvais doivent être bons.

Cas 4. Nous testons comme mauvais un processeur ayant testé comme 1 le processeur x
Alors, le processeur x doit être bon. S'il était mauvais, cela impliquerait que les m processeurs ayant testé comme 0 sont mauvais ainsi que le processeur que nous venons d'identifier. Puisque le processeur x est bon, alors les m processeurs l'ayant testé comme mauvais sont mauvais et les $m-1$ processeurs l'ayant testé comme bon sont bons.

Cas 5. Nous testons comme bon le processeur x
Alors les m processeurs l'ayant testé comme mauvais sont mauvais et tous les autres sont bons.

Cas 6. Nous testons comme mauvais le processeur x
Alors les $m-1$ processeurs l'ayant testé comme bon sont mauvais ainsi que le processeur x et tous les autres sont bons.

Il reste à estimer le nombre de tests. Supposons que nous travaillons seulement avec la sous-clique de grandeur n', nous aurons :
Événement A. Le nombre total de tests effectués est $t + |T| + n' - |T| - 1 = n' + t - 1$.
Événement B. Le nombre de tests avant l'appel récursif est $2m - 1$. Le nombre de tests pendant l'appel récursif est au plus $(n' - 2m) + (t - m) - 1$. Le nombre de tests après l'appel récursif est au plus $1 + m$. Le total est donc au plus $n' + t - 1$.
Finalement, nous aurons un test de plus en pire cas pour chacun des $n - n'$ processeurs externes à la sous-clique. Ce qui donne $n - n' + n' + t - 1 = n + t - 1$ tests en pire cas.

Nous avons donc un t-diagnostic en $n + t - 1$ tests dans un graphe irrégulier de $nt - t$ liens. □

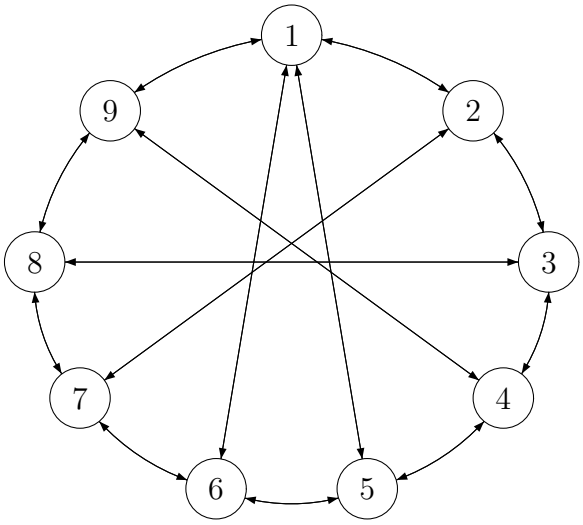

FIG. 5.5 – Un graphe t-diagnosticable contenant $\lceil \frac{nt}{2} \rceil$ liens où $t = 3$

5.3.3 Algorithme Diagnostic Adaptatif Ada-Diagnostic-Optimal

Dans cette section, nous présentons un graphe régulier avec $\lceil \frac{nt}{2} \rceil$ liens ainsi qu'un nouvel algorithme qui effectue correctement le t-diagnostic avec au plus $n + t - 1$ tests pour tous $t \in \mathbf{N}$ tel que $2 \leq t < 2\sqrt{n} - 2$.

Nous dénoterons par G un graphe en anneau $(g_1, g_2, \ldots, g_n)$ dont chaque processeur a un lien bidirectionnel avec les $\lfloor \frac{t}{2} \rfloor$ processeurs suivants et précédents. Si t est impair, alors chaque processeur aura un lien pour former un couple avec un autre processeur. Un processeur ne peut être que dans un couple. Si n est impair, alors un et un seul processeur sera dans 2 couples (voir Fig. 5.5). Nous dénoterons par F, l'ensemble des processeurs incidents à une arête étiquetée 1 ; par k le nombre de couples de deux processeurs consécutifs, incidents à une arête étiquetée 1 ; par S, une suite maximale $(s_i, s_{i+1}, ..., s_j)$ de processeurs consécutifs tels que toutes les arêtes $S_k \rightarrow S_{k+1}$ sont étiquetées 0. Une telle suite est appelée une bonne suite. $|S|$ dénote le nombre de processeurs dans une bonne suite.

Algorithme Ada-Diagnostic-Optimal($\mathcal{G}_n$)

Étape 1 :

$i \leftarrow 1$

$k \leftarrow 0$
Tant que($i \leq n$) **Faire**
  Test(g_i, g_{i+1}).
  Si $S(g_i, g_{i+1}) = 1$ **Alors**
  Mettre couple (g_i, g_{i+1}) dans F
  $k \leftarrow k + 1$
  $i \leftarrow i + 2$
  Sinon $i \leftarrow i + 1$

Étape 2 :
Si $(k \geq 1)$ **Alors**
Trouver une bonne suite S tel que $|S| \geq t - k + 1$ processeurs ;
Diagnostiquer le dernier processeur de S comme bon.
Sinon Terminer l'algorithme : tous les processeurs sont corrects.

Étape 3 :
À partir d'un bon processeur, diagnostiquer le permier processeur de chaque couple dans F. Diagnostiquer tous les processeurs qui suivent ce premier s'il est mauvais jusqu'à ce qu'un bon processeur soit trouvé.

Lemme 5.3.3. *Si $t < 2\sqrt{n} - 2$, alors à l'étape 2 on trouve une bonne suite de longueur au moins $t - k + 1$ dont le dernier processeur est bon.*

Démonstration. Si $t < 2\sqrt{n} - 2$ alors $(t + 2)^2 - 4n < 0$. Cette inéquation implique que l'inéquation $k^2 - (t + 2)k + n > 0$ est vraie pour tout k. Donc $\frac{n-2k}{k} > t - k$ pour tout k. Cela implique l'existence d'une bonne suite de longueur plus grande que $t - k$, donc au moins $t - k + 1$. Le dernier processeur de cette suite (voir Fig. 5.6) doit être bon. Sinon, tous les processeurs de cette suite seraient en panne, mais il y a, en plus, au moins k processeurs en panne (au moins un dans chaque couple de F), ce qui donnerait un total d'au moins $t - k + 1 + k = t + 1$ processeurs en panne. Contradiction !

□

Théorème 5.3.4. *Si $t < 2\sqrt{n} - 2$, alors à la fin de l'algorithme Ada-Diagnostic-Optimal tous les processeurs sont diagnostiqués et le nombre total de tests est $n + t - 1$.*

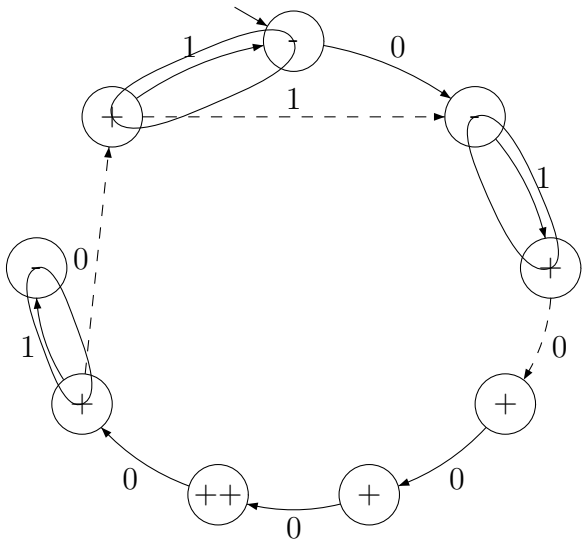

FIG. 5.6 – 3-diagnostic respectant la condition $t < 2\sqrt{n} - 2$

Démonstration. Il s'ensuit du lemme 5.3.3 qu'à la fin de l'étape 2, au moins un bon processeur est trouvé puisque dans la séquence la plus longue, si le dernier processeur était mauvais, cela impliquerait que nous ayons plus de t fautes.

Il reste à estimer le nombre de tests. Pour la première étape, le nombre de tests est de $n - k + 1$ en pire cas (le $+1$ étant le cas où le dernier processeur teste le premier comme mauvais). Ensuite, nous aurons besoin d'au plus $2k - 2$ tests pour identifier chaque processeur de chaque couple (Le premier couple ayant été identifié dans l'étape 1 par le dernier processeur de la séquence correcte). Nous aurons besoin de $t - k$ tests supplémentaires pour les $t - k$ fautes potentielles qui n'ont pas été diagnostiquées dans les couples de F.

Donc nous avons $(n - k + 1) + (2k - 2) + (t - k) = n + t - 1$ tests. □

Le théorème suivant dû à Blecher [6], que nous laissons sans démonstration, montre que l'algorithme Ada-Diagnostic est optimal du point de vue du nombre de tests effectués.

Théorème 5.3.5. *Chaque algorithme qui effectue un t-diagnostic dans un système de n processeurs doit utiliser au moins $n + t - 1$ tests en pire cas.*

Chapitre 6

Diagnostic adaptatif optimal rapide

Dans ce chapitre, nous atteindrons le but de diminuer le temps de t-diagnostic adaptatif tout en gardant le nombre minimal de $n+t-1$ tests. Nous présentons deux algorithmes travaillant dans un graphe complet. Le premier fonctionne pour $t < \frac{n}{2}$ et travaille en temps $O(\log n + t)$ tandis que le deuxième fonctionne seulement si $n > 2t^2 + 2t$, mais travaille en temps $O(\log t)$. Nous considérons ce dernier algorithme comme la contribution la plus importante de ce mémoire.

6.1 Algorithme rapide pour les cas généraux

Dans cette section, nous allons présenter un algorithme inspiré de celui de Blecher [6] mais modifié afin de réduire son temps de fonctionnement. L'algorithme Blecher-Rapide fera le diagnostic en $O(\log n + t)$ rondes pour $t < \frac{n}{2}$.

6.1.1 Algorithme Diagnostic Adaptatif Blecher-Rapide

Algorithme Diagnostic Adaptatif Blecher-Rapide($\mathcal{T}_n$)

Phase 1 : Identification

Appliquer l'algorithme de Blecher jusqu'à ce qu'au moins un bon processeur soit trouvé.

Phase 2 : Diagnostic

$K \leftarrow$ L'ensemble des bons processeurs identifiés en phase 1.

$F \leftarrow$ L'ensemble des fautes identifiées en phase 1.

Tant que $(|K| + |F|) < n$, Faire

Chaque bon processeur dans K teste un processeur

qui n'est pas dans K ou dans F.

$F \leftarrow$ L'ensemble des fautes trouvées jusqu'à maintenant.

$K \leftarrow$ L'ensemble des bons processeurs trouvés jusqu'à maintenant.

Théorème 6.1.1. *Si $t < \frac{n}{2}$, alors l'algorithme Diagnostic-Adaptatif-Blecher-Rapide utilise $n + t - 1$ tests et travaille en temps $O(\log n + t)$.*

Démonstration. Le nombre de tests a été justifié par la démonstration du theorème 5.1.1 dans le chapitre précédent. Il nous reste donc à estimer le nombre de rondes.

La phase 2 de l'algorithme diffère de celui de l'algorithme de Blecher [6] en ce sens que, dans l'algorithme de Blecher [6], le processeur identifié comme bon fait tous les tests. Dans notre cas, nous utilisons les autres bons processeurs pour effectuer des tests en parallèle.

Supposons que nous avons f fautes identifiées en phase 1. La phase 1 prend alors au plus $t + f$ tests puisque nous avons f fautes et t tests supplémentaires pour trouver un bon processeur, donc $t + f$ rondes. Les rondes en phase 2 peuvent être divisées en deux classes : les rondes blanches, où il n'y a que des bons processeurs diagnostiqués et les rondes noires, où il y a au moins un mauvais processeur diagnostiqué. Le nombre de rondes noires est au plus $t - f$ et le nombre de rondes blanches est au plus $\log_2 n$, car dans une ronde blanche, le nombre de processeurs identifiés est doublé. Alors en phase 2, nous aurons $\log_2 n + t - f$ rondes en pire cas à effectuer pour identifier les processeurs restants.

Nous aurons donc en pire cas $(\log_2 n) + (t - f) + (t + f) = \log_2 n + 2t$ rondes pour faire le diagnostic. Donc le temps du diagnostic est $O(\log n + t)$. □

6.2 Diagnostic rapide pour peu de pannes

Dans cette section, nous allons présenter un algorithme qui permet de faire le diagnostic dans un graphe complet en utilisant $n + t - 1$ tests tout en utilisant seulement $O(\log\ t)$ rondes, pour $n \geq 2t^2 + 2t$.

Nous allons définir une 0-ligne parfaite de longueur m comme étant une suite des processeurs $p_1, p_2..., p_m$, tel que tous les tests $p_i \rightarrow p_{i+1}, i < m$ ont été effectués et $S(p_i, p_{i+1}) = 0$, pour tout $i < m$. Nous allons également définir une 0-ligne imparfaite de façon semblable comme une 0-ligne parfaite mais avec la différence que l'avant-dernier processeur aura testé le dernier processeur comme mauvais au lieu de bon. Nous utiliserons l'appelation 0-ligne pour inclure l'une ou l'autre de ces notions.

Voici quelques autres notions dont nous aurons besoin dans ce chapitre :
L : Ensemble de lignes de processeurs tel que le processeur 1 de chaque ligne a testé le processeur 2 qui a testé le processeur 3 et ainsi de suite.
L_i : La i^{eme} ligne de processeurs de l'ensemble de lignes L.
$|L_i|$: Nombre de processeurs dans la ligne L_i.
$L_i(k)$: Le k^{eme} processeur de la ligne L_i.
L^{max} : La plus grande ligne appartenant à L.
$L_i[v_k, v_j]$: Segment de ligne appartenant à une ligne L_i partant du processeur v_k et se terminant au processeur v_j.
L_i^f : Le f^{eme} segment de ligne L_i où $L_i^0 = L_i[0, h], L_i^1 = L_i[h + 1, 2h], L_i^3 = L_i[2h + 1, 3h]$ et ainsi de suite, où h est la longueur d'un segment.
$L_i^f(x)$: Le x^{eme} processeur d'un segment L_i^f (ce processeur est égal à $L_i(fh + x)$).

6.2.1 Algorithme Diagnostic Adaptatif Ligne-Rapide

Dans cette section, nous présentons un algorithme qui permet de faire le diagnostic d'une 0-ligne contenant t pannes en $O(\log\ t)$ rondes en utilisant en pire cas $t + 1$ tests (Voir fig.6.1). Nous supposons l'existence d'un processeur x diagnostiqué comme bon, en dehors de la ligne.

Algorithme Diagnostic Adaptatif Ligne-Rapide($\mathcal{T}_n$)

Soit $1, 2, \ldots, n$ une énumération des processeurs. Supposons que ces processeurs sont dans une 0-ligne, 1 étant le premier processeur de la 0-ligne, 2 étant le deuxième et ainsi de suite. Supposons que X est un bon processeur qui a testé le processeur 1 comme mauvais.

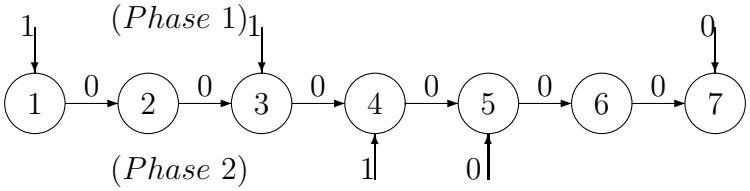

FIG. 6.1 – Exemple d'analyse d'une ligne où $t = 4$

Phase 1 : Sectionnement

$i \leftarrow 2$
Faire test $(X, min(2, n))$.
Tant que $S(X, min(2^i - 1, n)) = 1$ Faire
  $i \leftarrow i + 1$
  Faire $test(X, min(2^i - 1, n))$

Phase 2 : Dichotomie

$b \leftarrow 2^{i-1} - 1$
$f \leftarrow min(2^i - 1, n)$
$m \leftarrow \frac{b+f}{2}$
Tant que $b < f$
  Faire test(X, m).
  Si $S(X, m) = 0$ Alors
    $f \leftarrow m$
  Si non
    $b \leftarrow m + 1$
  $m \leftarrow \lfloor \frac{b+f}{2} \rfloor$

Théorème 6.2.1. *Supposons qu'il y a t fautes dans la 0-ligne parfaite T_n. L'algorithme Ligne-Rapide accomplit correctement le diagnostic de T_n, et utilise exactement $2\lfloor \log_2(t+1) \rfloor$ tests et prend $2\lfloor \log_2(t+1) \rfloor$ rondes.*

Démonstration. Nous devons tout d'abord démontrer l'exactitude en prouvant qu'à la fin de l'algorithme, tous les processeurs seront diagnostiqués.

Observons d'abord que tous les processeurs qui suivent un processeur diagnostiqué comme bon par X, doivent être bons. Il reste à démontrer que tous les processeurs précédant un processeur diagnostiqué comme mauvais par X doivent être mauvais.

Supposons qu'un bon processeur précède le processeur identifié par X comme mauvais dans la ligne, alors tous les processeurs qui suivent ce bon processeur doivent également être bons puisque nous avons une séquence de tests 0. Cela implique que le processeur testé comme mauvais par le processeur X est bon et que, par conséquent, le processeur X est mauvais. Toutefois, par définition, le processeur X est bon. Contradiction !

Nous avons démontré l'exactitude de l'algorithme, il nous reste donc à estimer le temps et le nombre de tests.

Tout d'abord, un seul test est effectué par ronde. Par conséquent, le nombre de tests doit être égal au nombre de rondes.

À la phase 1 de l'algorithme, nous utilisons $\lfloor \log_2(t+1) \rfloor$ tests. Puis, une approche dichotomique est faite sur les processeurs situés entre le dernier test et le test précédent. Soit $2^i - 2^{i-1} = 2^{i-1} = 2^{\lfloor \log_2(t+1) \rfloor}$ processeurs à tester. Nous aurons donc $\log_2(2^{\lfloor \log_2(t+1) \rfloor}) = \lfloor \log_2(t+1) \rfloor$ tests à effectuer en phase 2. Par conséquent, le total sera $2\lfloor \log_2(t+1) \rfloor$ tests à effectuer. □

Corollaire 6.2.2. *L'algorithme Ligne-Rapide($\mathcal{T}_n$) est aussi ou plus efficace lorsqu'il teste une 0-ligne imparfaite.*

Démonstration. Il y a deux situations possibles à vérifier pour une séquence. Si le processeur X effectue un test précédant le dernier résultat de tests (qui a donné un 1) et que ce résultat donne 0, alors le dernier processeur doit être mauvais et il n'occasionnera aucun coût supplémentaire pour être identifié. Si l'avant-dernier processeur est testé comme

mauvais par le processeur X (impliquant que tous les autres processeurs le précédant sont mauvais selon la preuve du théorème 6.2.1), alors nous aurons $\lfloor \log_2(t+1) \rfloor$ tests pour identifier cette séquence et nous aurons un test supplémentaire pour identifier le dernier processeur. Soit $\lfloor \log_2(t+1) \rfloor + 1 \leq 2\lfloor \log_2(t+1) \rfloor$ pour $t \geq 1$. □

Corollaire 6.2.3. *L'algorithme Ligne-Rapide($\mathcal{T}_n$) utilise $t+1$ tests en pire cas.*

Démonstration. $2\lfloor \log(t+1) \rfloor \leq t+1$, pour chaque $t \geq 1$. □

6.2.2 Algorithme Diagnostic Adaptatif Balayage-de-Lignes

Dans cette section, nous présentons un algorithme qui permet de faire le diagnostic d'un ensemble de $x \leq t$ 0-lignes contenant au plus t pannes au total en $O(\log\ t)$ rondes et qui utilise au maximum $t-1$ tests (Voir fig.6.2). Nous supposons l'existence de x processeurs diagnostiqués comme bons à l'extérieur de ces lignes et nous supposons également que chaque processeur qui débute une 0-ligne aura été identifié comme fautif.

Algorithme Diagnostic Adaptatif Balayage-de-Lignes($\mathcal{T}_N$)

Soit X, un ensemble de bons processeurs où $|X| = x$, $\mathcal{L}_i$ une i^{eme} 0-ligne de processeurs où $0 \leq i < x$, $\mathcal{L}_i(k)$ le k^{eme} processeur de la i^{eme} ligne où $0 \leq k \leq \chi$ tel que $\log_{\frac{32}{31}} t + 5 \leq \chi < 2(\log_{\frac{32}{31}} t + 5))$, P_r, le nombre de 0-lignes où $S(X_i, \mathcal{L}_i(r)) = 1$, $\tau(r)$, le nombre de tests maximum pouvant être utilisé à la ronde r, $|S(X, \mathcal{L}(k)) = \delta|$, le nombre de tests que chaque processeur de l'ensemble X a effectué sur chaque r^{eme} processeur de l'ensemble $\mathcal{L}$ de lignes et dont le résultat a donné δ.

$r \leftarrow 1$
$\tau(r) \leftarrow t - P_0 - 1$
Tant que $(\tau(r) < 32P_r)$ faire
    Faire Test$(X_i, \mathcal{L}_i(r))$ en parallèle pour $i \leq x$, pour tous i dans $\mathcal{L}$.
    $\tau(r+1) \leftarrow \tau(r) - P_r$
    $r \leftarrow$ r + 1
    $P_r \leftarrow |S(X, \mathcal{L}(r)) = 1|$
    $\mathcal{L} \leftarrow \mathcal{L} \backslash \{i \in \mathcal{L} | S(X_i, \mathcal{L}_i(r)) = 1\}$

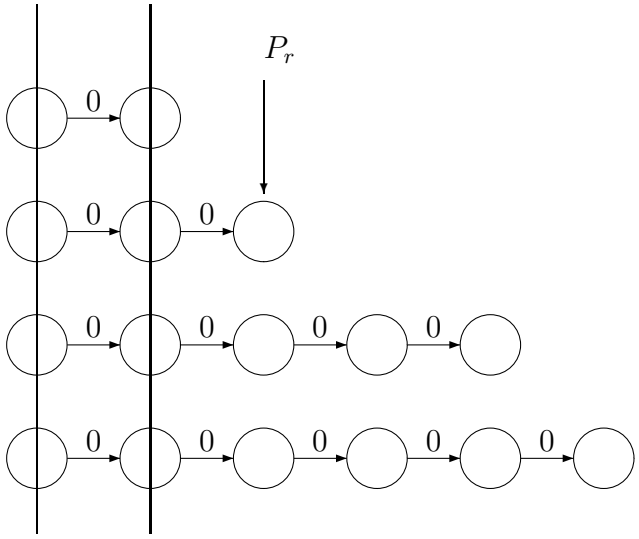

FIG. 6.2 – Exemple du Balayage-de-Lignes pour $r = 2$ et $P_r = 3$

$X \leftarrow X \backslash \{i \in \mathcal{L} | S(X_i, \mathcal{L}_i(r)) = 1\}$
Appeler Algorithme Ligne-Rapide($\mathcal{T}_P$) en parallèle pour les P_r lignes.

Théorème 6.2.4. *L'algorithme Balayage-de-Lignes permet de faire un diagnostic dans un ensemble $|\mathcal{L}|$ de 0-lignes en $C \log_2 t$ rondes, pour une certaine constante C en utilisant au plus $t - 1$ tests pour t pannes.*

Démonstration. Supposons qu'il y a P_r 0-lignes avec un total d'au plus $\tau(r)$ pannes où $\tau(r) \leq t$ pannes. Alors, en utilisant l'algorithme Ligne-Rapide en parallèle pour toutes les 0-lignes, on peut faire le diagnostic avec $\sum_{i=1}^{P_r}(2\lfloor \log(\tau_i + 1) \rfloor \leq \sum_{i=1}^{P_r}(4 \log \tau_i)$ tests, où τ_i est le nombre de pannes dans la i^{me} ligne (en temps $O(\log \tau(r))$ (Voir fig.6.3).

Pour calculer le nombre de tests, nous profitons du fait algébrique suivant :
Fait : Si $x_1 + \ldots + x_k = x$ alors $\log x_1 + \ldots + \log x_k \leq k \log \frac{x}{k}$.

Puisque $\tau_1 + \ldots + \tau_{P_r} = \tau(r)$, le diagnostic peut être fait avec au plus $4P_r \log \frac{\tau(r)}{P_r}$ tests.

Notons que nous pouvons resserrer la base du logarithme à $\frac{5}{4}$ en faisant quelques manipulations algébriques plus complexes. Cela permettrait de faire fonctionner l'algorithme Balayage-Rapide pour $t \geq 18$.

Soit $r_0 = \lceil \log_{\frac{32}{31}} t \rceil$. Nous allons montrer qu'il existe un $r \leq r_0$ tel que soit $\tau(r) = 0$, soit $\tau(r) \geq 32P_r$. Supposons que ce n'est pas vrai. Alors pour chaque $r \leq r_0$ on a $\tau(r) \leq \frac{31}{32}\tau(r-1)$. Donc $\tau(r_0) = 0$. Contradiction.

Cela implique 2 différentes situations : soit l'algorithme Ligne-Rapide est appelé avant $C \log_2 t$ rondes ou soit après $C \log_2 t$ rondes, tous les processeurs défectueux sont trouvés. Nous considérons ces deux cas.

Cas 1 : Si $\tau(r) \geq 32P_r$ pour une certaine ronde r où $r < C \log t$. Alors le diagnostic peut être fait avec au plus $\tau(r) - 1$ tests. En effet, pour $\tau(r) = 32P_r$ cela doit être vrai, car $4P_r \log \frac{32P_r}{P_r} \leq 32P_r$; donc, à plus forte raison, cela est vrai pour $\tau(r) > 32P_r$, car $\lim_{x\to\infty} \frac{\log x}{x} = 0$.

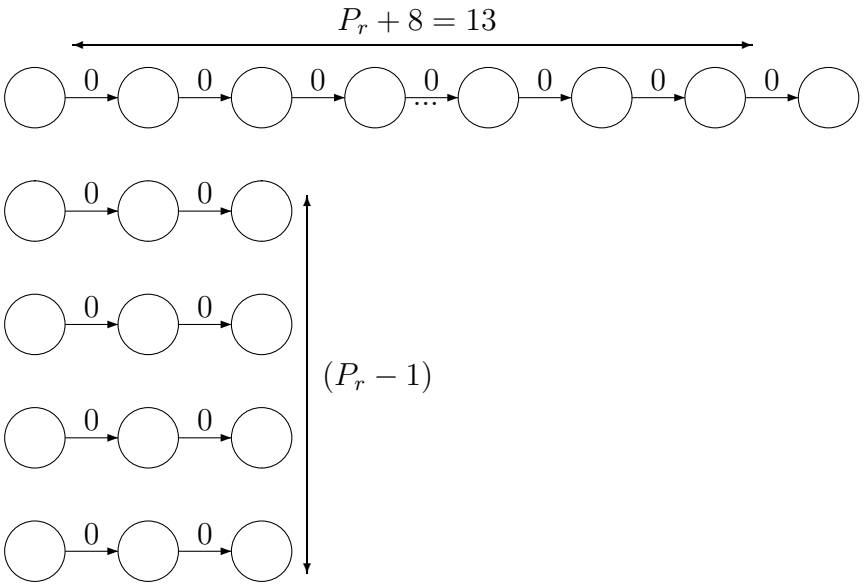

FIG. 6.3 – Exemple de la pire répartition de processeurs pour $t = 5P_r$ où $r = 0$ et $P_r = 5$.

Cas 2 : Si $\tau(r) < 32P_r$, pour chaque $r < C \log t$, alors la condition nécessaire pour appeler l'algorithme Ligne-Rapide n'a pas été rencontrée. Toutefois, dans un tel cas, nous aurons au maximum $\log_{\frac{32}{31}} t$ rondes durant l'algorithme Balayage-de-Lignes.

Il nous reste à démontrer que, dans un tel cas, les nombre de tests utilisés est inférieur à t. Pour la ronde $r = 0$, le nombre de processeurs identifiés est égal à P_0 puisque chaque premier processeur de chaque ligne appartenant à $\mathcal{L}$ a été identifié comme fautif avant l'appel de l'algorithme (assomption de base pour l'algorithme Diagnostic Adaptatif Balayage-de-Lignes).

Nous aurons donc $\sum_{i=0}^{r} P_i$ processeurs identifiés en rondes r et nous aurons utilisé $\sum_{i=0}^{r-1} P_i$ tests. Or $P_r > 0$ car sinon, cela implique que tous les processeurs ont été diagnostiqués. Par conséquent, $\sum_{i=0}^{r} P_i - \sum_{i=0}^{r-1} P_i = P_r$ tests restants pour avoir t tests au total. Puisque $P_r > 0$, nous aurons en pire cas effectué $t - 1$ tests. □

6.2.3 Algorithme Diagnostic Adaptatif Balayage-Rapide

Dans cette section, nous présentons un algorithme qui fonctionne en $O(\log\ t)$ rondes et qui utilise $n + t - 1$ tests en pire cas.

Supposons que $t \geq 167$. Nous dénoterons par $\mathcal{L}$ l'ensemble des lignes avec au moins $t+1$ processeurs. Également, le réseau devra respecter la condition $|\mathcal{L}| = \lfloor \frac{n}{t+1} \rfloor$, $n \geq 2t^2 + 2t$, $|\mathcal{L}| \geq 2t$ et $t \geq \lceil \log_{\frac{32}{31}} t \rceil + 5$.

Algorithme Diagnostic Adaptatif Balayage-Rapide($\mathcal{T}_n$)

Phase 1 : Identification de groupes

Considérons une énumération de processeurs de 1 à n. Considérons une partition $\mathcal{L}$ de tous les processeurs en ensembles de grandeur $t+1$ tel que $n = \mathcal{L}(t+1) + r$ où $r < t+1$ et $\mathcal{L} \geq 1$. Chaque processeur parmi les r processeurs restant sera ajouté dans les r premiers ensembles. Les ensembles ainsi formés seront numérotés de 1 à $|\mathcal{L}|$. Chacuns de ces ensembles peut être visualisée comme une ligne dont le premier processeur sera testeur du deuxième qui sera testeur du troisième et ainsi de suite.

Nous dénoterons par $\mathcal{L}_i$ la i^{eme} ligne de processeurs. Nous dénoterons également $\beta = \left\lfloor \frac{\mathcal{L}_i}{\log_{\frac{32}{31}} t+5} \right\rfloor$, le nombre de segments de ligne dans une ligne et $\mathcal{L}_i^f(j)$, un segment de ligne où $0 \leq f \leq \beta$ comprenant χ processeurs où $\chi = x(\log_{\frac{32}{31}} t + 5) + r'$. Chaque processeur parmi les r' processeurs restants sera ajouté dans les x premiers segments des $\log_{\frac{32}{31}} t + 5$ segments de lignes, de façon uniforme (Voir fig.6.4).

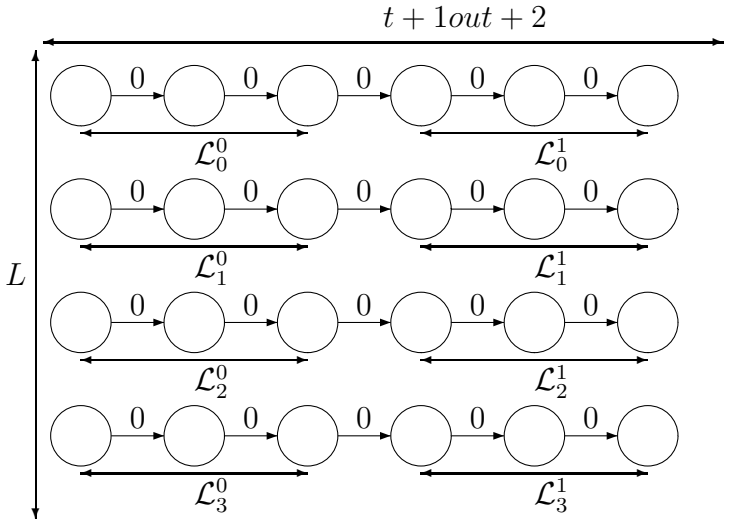

FIG. 6.4 – Exemple de segments de lignes d'un graphe où $\mathcal{L}_i^k$ veux dire le k^{eme} segment de la i^{eme} ligne.

$j \leftarrow 1$
$\lambda \leftarrow 0$
Faire en parallèle test$(\mathcal{L}_i^f(0), \mathcal{L}_i^f(1))$
Tant que $j < \chi$
  Si $S(\mathcal{L}_i^f(j-1), \mathcal{L}_i^f(j)) = 0$ ou
  (test$(\mathcal{L}_i^f(j-1), \mathcal{L}_i^f(j))$ n'a pas eu lieu) Alors
    Faire en parallèle test$(\mathcal{L}_i^f(j), \mathcal{L}_i^f(j+1))$ pour chaque segment qui respecte la condition.
  Sinon
    $\lambda \leftarrow \lambda + 1$ pour chaque segment qui ne respecte pas la condition.

$j \leftarrow j + 1$
Faire pour chacun de ces segments le test($\mathcal{L}_i^f(\chi)$,$\mathcal{L}_i^{f+1}(0)$) où $f < \beta$.

Nous dénoterons ϕ, le nombre des lignes $\mathcal{L}_i$ dont tous les segments $\mathcal{L}_i^f$ sont des 0-lignes parfaites, γ le nombre de ces tests qui donne un résultat 1 et δ le nombre de lignes où $\lambda > 0$.

Lemme 6.2.5. *La phase* 1 *de l'algorithme Diagnostic Adaptatif Balayage-Rapide prend exactement* $n - |\mathcal{L}| - \lambda - ((|\mathcal{L}| - \phi)\beta)$ *tests et se fait en* χ *rondes en pire cas.*

Démonstration. Le nombre de rondes est justifié par la formulation de l'algorithme, il nous reste à estimer le nombre de tests.

Nous avons $|\mathcal{L}|$ lignes $\mathcal{L}_i$ de processeurs dont le dernier processeur de chaque ligne ne teste aucun processeur durant cette phase. Nous avons donc $n-|\mathcal{L}|$ tests. Toutefois, nous enlevons à ce résultat le nombre de tests que nous n'aurons pas faits suite à des résultats de tests 1, soit λ tests. Également, nous avons β segments par ligne. Nous aurons donc omis également de faire les tests d'un segment vers un autre sur les lignes ayant au moins un test avec un résultat 1, ce qui nous donne $(|\mathcal{L}| - \phi)$ lignes multipliées par β segments pour un total de tests de $n - |\mathcal{L}| - \lambda - \Big((|\mathcal{L}| - \phi)\beta\Big)$. □

Lemme 6.2.6. *La phase* 1 *de l'algorithme Diagnostic Adaptatif Balayage-Rapide permet de trouver au moins* t *0-lignes parfaites de processeurs.*

Démonstration. Puisque nous avons au moins $2t$ lignes et que nous avons au plus t fautes, il doit exister t lignes qui ne contiennent pas de faute. □

Phase 2 : Complément de séquences
Pour les $|\mathcal{L}| - \delta - \gamma$ 0-lignes parfaites identifiées en phase 1, faire test($\mathcal{L}_i(|\mathcal{L}_i|)$,$\mathcal{L}_i(0)$).

Lemme 6.2.7. *La phase* 2 *de l'algorithme Diagnostic Adaptatif Balayage-Rapide prend exactement* $|\mathcal{L}| - \delta - \gamma$ *tests et se fait en une ronde.*

Démonstration. Directe. □

Lemme 6.2.8. *Suite à la phase* 2 *de l'algorithme Diagnostic Adaptatif Balayage-Rapide, au moins* $2t$ *bons processeurs sont diagnostiqués.*

Démonstration. Suite à la phase 2, au moins $t \times (t+1) = t^2 + t \geq 2t$ processeurs sont diagnostiqués comme bons. □

Phase 3 : Test de séquence restante
Tester les $\delta + \gamma$ processeurs $\mathcal{L}_i(0)$ qui n'ont pas été testés en phase 2 ainsi que les λ processeurs suivant un résultat de test 1. Tester également les γ processeurs $\mathcal{L}_i^f(0)$ qui suivent dans une ligne $\mathcal{L}_i$ un $S(\mathcal{L}_i^f(\chi),\mathcal{L}_i^{f+1}(0)) = 1$ pour $f < \beta$.

Lemme 6.2.9. *La phase* 3 *de l'algorithme Diagnostic Adaptatif Balayage-Rapide prend exactement* $\delta - t + \lambda + \Big((|\mathcal{L}| - \phi)\beta\Big) + 2\gamma$ *tests et se fait en une ronde.*

Démonstration. Le nombre de rondes est justifié par la formulation de l'algorithme. Il nous reste à démontrer le nombre de tests.

Il y a exactement $\delta + \gamma$ tests sur le premier processeur de chaque ligne qui n'ont pas été complétés en phase 2. Il faut également faire λ tests pour identifier les λ processeurs qui suivent un test donnant le résultat 1 dans chaque ligne. De plus, il faut $\Big((|\mathcal{L}| - \phi)\beta\Big)$ tests pour identifier les premiers processeurs de chaque segment qui ont eu un résultat de tests 1. Finalement, il y a les γ tests pour identifier le premier processeur d'un segment qui a été diagnostiqué comme mauvais par le dernier processeur du segment précédent lorsque chaque segment de la ligne était une 0-ligne parfaite. Pour un grand total de $\delta - t + \lambda + \Big((|\mathcal{L}| - \phi)\beta\Big) + 2\gamma$ tests. □

Corollaire 6.2.10. *La phase* 3 *de l'algorithme Diagnostic Adaptatif Balayage-Rapide permet de créer des 0-lignes d'une longueur maximale de* χ *processeurs.*

Démonstration. Directe. □

Lemme 6.2.11. *À la fin de la phase 3 de l'algorithme Diagnostic Adaptatif Balayage-Rapide($\mathcal{T}_n$), le nombre de tests effectués est exactement* $n + \gamma$.

Démonstration. En phase 1 de l'algorithme, nous aurons exactement $n - |\mathcal{L}| - \lambda - ((|\mathcal{L}| - \phi)\beta)$ tests. En phase 2 de l'algorithme, nous aurons exactement $|\mathcal{L}| - \delta - \gamma$ tests. Finalement, en phase 3 de l'algorithme, nous aurons exactement $\delta - t + \lambda + \Big((|\mathcal{L}| - \phi)\beta\Big) + 2\gamma$ tests. Nous aurons donc $n - |\mathcal{L}| - \lambda - ((|\mathcal{L}| - \phi)\beta) + (|\mathcal{L}| - \delta - \gamma) + \delta + \lambda + \Big((|\mathcal{L}| - \phi)\beta\Big) + 2\gamma = n + \gamma$ tests. □

Phase 4 : Filtrage des 0-lignes commençant par un mauvais processeur
Soit P_0, le nombre de segments de lignes de processeurs qui ont été testés en phase 3 et dont le résultat du test a donné 1. Appeler l'algorithme Diagnostic Adaptatif Balayage-de-Lignes($\mathcal{T}_{P_0}$).

Théorème 6.2.12. *L'algorithme Diagnostic Adaptatif Balayage-Rapide($\mathcal{T}_n$) utilise $n + t - 1$ tests pour identifier t fautes et prend au maximum $C \log_2 t + 3$ rondes pour $n \geq 2t^2 + 2t$ et $t \geq 18$.*

Démonstration. Nous avons démontré que dans l'algorithme Balayage-de-Lignes nous utilisons au plus $t - 1$ tests pour identifier t fautes.

Ce que nous devons démontrer est que nous pourrons épargner γ tests à raison de 1 par chacun des γ segments de lignes et que, par conséquent, lorsque nous avons la situation où il y a γ segments de lignes, nous utiliserons en fait $t - \gamma - 1$ tests au lieu de $t - 1$ tests lors de l'appel de l'algorithme Balayage-de-Lignes.

Notons que les γ segments de lignes ont tous une longueur minimale de $\log_{\frac{32}{31}} t + 5$ et que chacune d'entre eux est une 0-ligne imparfaite. L'algorithme Balayage-de-Lignes prend au plus $\log_{\frac{32}{31}} t$ rondes en identifiant à chaque fois un seul processeur par segment de ligne.

Si l'algorithme Ligne-Rapide n'est pas appelé après $\log_{\frac{32}{31}} t$ rondes, cela veut dire que nous avons identifié toutes les fautes. Par conséquent, le dernier processeur de chacun des γ segments de lignes doit être bon. Ce qui permet d'identifier le processeur suivant comme mauvais, diagnostic qui ne coûte aucun test. Cela nous permet de sauver un test pour chacun des γ segments de lignes.

Par contre, si l'algorithme Ligne-Rapide est appelé, nous avons au moins 5 processeurs pour chacune des γ 0-lignes puisque l'algorithme prend au plus $\log_{\frac{32}{31}} t$ et que nous avons des segments de lignes de grandeur $\log_{\frac{32}{31}} t+5$ processeurs. Il y a deux situations possibles. La première est que tous les processeurs non diagnostiqués de ce segment de lignes sont mauvais, auquel cas nous aurons alors sauvé au moins un test par γ segments de lignes puisque pour $t > 5$ fautes, nous aurons toujours au moins $t-1$ tests. La seconde situation est que nous aurons au moins un bon processeur dans le segment de ligne. Dans ce cas, l'avant-dernier processeur doit être bon et nous aurons alors sauvé également 1 test pour

identifier le dernier processeur. Alors nous aurons sauvé au minimum 1 test, peu importe ce qui se passe pour chacun des γ segments de lignes donc, nous aurons sauvé au moins γ tests au total. Finalement, puisque selon le théorème 6.2.4, l'algorithme Balayage-de-Lignes prend au plus $t-1$ tests, nous aurons à la suite de la phase 4 de l'algorithme $n+\gamma+t-1-\gamma = n+t-1$ tests. □

Chapitre 7

Résultats de simulations

7.1 Description des simulations

Pour chaque simulation, nous avons choisi un paramètre t et un paramètre n adapté en fonction de t. Nous avons effectué 100 simulations pour chaque cas. Les simulations étaient effectuées avec une distribution aléatoire uniforme des processeurs fautifs et chaque processeur fautif attribuait un résultat bon ou mauvais en testant un autre processeur avec une probabilité de 50% que ce processeur soit bon et de 50% qu'il soit mauvais. Nous avons fait ensuite la moyenne des 100 résultats obtenus pour chaque cas.

7.2 Résultats de simulation des algorithmes de diagnostic pour les graphes à faible densité

Dans cette section, nous présentons les résultats obtenus par la simulation de l'algorithme diagnostic adaptatif de Blecher [6] et celui que nous avons construit, soit le diagnostic Ada-Diagnostic Optimal (voir tableau. 7.1). Nous comparons le nombre moyen de tests dans chaque cas.

Nous pouvons conclure que l'algorithme de diagnostic adaptatif de Blecher est généralement un peu plus coûteux en nombre moyen de tests que l'algorithme Ada-Diagnostic Optimal lorsque le paramètre t grandit mais qu'il est moins coûteux si t est vraiment petit par rapport à n.

Algorithme Diagnostic Adaptatif de Blecher		
n	t	# de tests
40	10	48
5000	10	4 579
700	50	746
50 000	50	49 197
3 000	100	3 091
200 000	100	198 109
65 000	500	65 476
500 000	500	499 608
Algorithme Diagnostic Ada-Diagnostic-Optimal		
n	t	# de tests
40	10	40
5000	10	4 999
700	50	700
50 000	50	49 999
3 000	100	3 000
200 000	100	199 999
65 000	500	65 000
500 000	500	499 999

TAB. 7.1 – Simulation des algorithmes fonctionnant dans des réseaux à faible densité.

Nous pouvons expliquer ces résultats par le fait que l'algorithme de diagnostic adaptatif de Blecher commence par identifier un bon processeur, ce qui lui prendra au moins t tests et possiblement $2t$ tests en pire cas. L'algorithme va arrêter lorsqu'il a trouvé toutes les fautes. Pour un t assez grand, il est probable qu'au moins un mauvais processeur sera identifié parmi les derniers. Ce qui causera un dépassement de n et va donc situer notre nombre de tests entre n et $n + t - 1$ tests très probablement. Par contre, pour un t très petit par rapport à n, il est probable qu'il trouvera le dernier mauvais processeur bien avant les t derniers. Par conséquent, dans ce cas, le nombre de tests sera inférieur à n.

L'algorithme Ada-T-Diagnostic Optimal, quant à lui, est stable sur le nombre de tests. Il fait presque toujours $n - k$ tests pour identifier k couples. Si les k couples ont leur premier processeur bon, le résultat tendra vers n. Il ne sera pratiquement jamais inférieur mais généralement très peu supérieur. Cela s'explique par le fait que la probabilité est

faible que deux couples soient subséquents ou encore que deux fautes soient subséquentes.

Rappelons finalement que l'algorithme de Blecher fonctionne seulement dans un graphe complet pour $t < \frac{n}{2}$ mais que l'algorithme Ada-T-Diagnostic Optimal fonctionne dans un graphe avec $\lceil \frac{nt}{2} \rceil$ liens mais seulement pour $t < 2\sqrt{n} - 2$.

7.3 Résultats de simulation des algorithmes en temps rapide

Dans cette section, nous présentons les résultats obtenus par la simulation de l'algorithme Diagnostic Adaptatif de Blecher-Rapide et celui que nous avons construit, soit le diagnostic Balayage-Rapide (voir tableau. 7.2), aussi bien du point de vue du nombre de tests que du point de vue de temps (en moyenne).

Pour le nombre de rondes, nous avons entre t rondes et $2t$ rondes pour identifier un bon processeur. Puisque t est relativement petit par rapport à n, le nombre de rondes pour trouver un bon processeur sera plus près de t que de $2t$. Il reste ensuite un temps logarithmique en n pour identifier le reste du réseau. Ce qui explique que le nombre de rondes est généralement légèrement supérieur à t.

L'algorithme Balayage-Rapide utilise toujours au moins n tests avant de diagnostiquer les fautes sauf dans des cas extrêmement peu probables tels qu'une ligne complète de t fautes. Puisque les t fautes sont généralement dipersées et que le résultat en est des 0-lignes imparfaites ou encore dont le premier processeur est le seul mauvais, le nombre de tests avoisine n.

Pour le nombre de rondes, nous pouvons voir que plus t est élevé par rapport à $\log_{\frac{5}{4}} t$(base utilisée pour l'algorithme que nous laissons sans démonstration), plus l'algorithme est performant sur le nombre de rondes.

Rappelons finalement que l'Algorithme Diagnostic de Blecher-Rapide fonctionne pour $t < \frac{n}{2}$ mais que l'algorithme Balayage-Rapide fonctionne seulement pour $n \geq 2t^2 + 2t$ où $t \geq 18$.

Algorithme Diagnostic Blecher-Rapide			
n	t	# de tests	# de rondes
1 000	20	984	29
10 000	20	9 568	32
5 500	50	5 462	61
100 000	50	98 278	65
21 000	100	20 971	113
200 000	100	198 122	116
510 000	500	509 917	518
900 000	500	899 798	519
Algorithme Diagnostic Balayage-Rapide			
n	t	# de tests	# de rondes
1 000	20	1 003	26
10 000	20	10 003	27
5 500	50	5 507	36
100 000	50	100 006	35
21 000	100	20 009	58
200 000	100	200 006	57
510 000	500	510 031	54
900 000	500	900 022	54

TAB. 7.2 – Simulation des algorithmes fonctionnant en temps rapide.

Chapitre 8

Conclusion

Nous avons montré des algorithmes adaptatifs et non-adaptatifs optimaux sur le plan du nombre de tests et du temps pour les grilles. Nous avons également trouvé différentes topologies de réseaux avec peu de liens et nous avons construit des algorithmes faisant le diagnostic optimal sur le nombre de tests dans ces réseaux.

Le tableau récapitulatif 8.1 compare la performance de nos algorithmes et spécifie les réseaux dans lesquels ils sont applicables.

Plusieurs problèmes intéressants restent ouverts. Ci dessous, nous en énumérons quelques uns :

1. Trouver un t-diagnostic utilisant $n + t - 1$ tests applicable dans chaque graphe t-connexe.
2. Trouver une borne inférieure non-constante sur le temps de fonctionnement d'un algorithme de t-diagnostic utilisant $n + t - 1$ tests.
3. Trouver un algorithme utilisant $n+t-1$ tests et travaillant en temps $O(\log t)$ pour chaque $t < \frac{n}{2}$.

Tableau récapitulatif des algorithmes				
Chapitre Section	Algorithmes	Nombre de liens du réseau à n processeurs	Borne supérieure sur le nombre de pannes	temps
4.1	NonAda-RingTest	n (Anneau)	2	$O(n)$
4.2.1	NonAda-GrilleTest	$O(n)$ (Grille)	2	$O(n)$
4.2.2	Ada-GrilleTest	$O(n)$ (Grille)	2	3
5.3.1	Ada-Diagnostic-Régulier	nt (Régulier)	$t < \frac{n}{2}$	$O(n)$
5.3.2	Ada-Blecher-Raréfié	$nt - t$	$t < \frac{n}{2}$	$O(n)$
5.3.3	Ada-Diagnostic-Optimal	$\lceil \frac{nt}{2} \rceil$	$t < 2\sqrt{n} - 2$	$O(n)$
6.1.1	Blecher-Rapide	$\frac{n\times(n-1)}{2}$ (Clique)	$t < \frac{n}{2}$	$O(\log n + t)$
6.2.3	Balayage-Rapide	$\frac{n\times(n-1)}{2}$ (Clique)	t tel que : $n > 2t^2 + 2t$	$O(\log t)$

TAB. 8.1 – Tableau récapitulatif des algorithmes utilisant $n+t-1$ tests en pire cas créés dans le cadre du mémoire.

Annexe A

Code de Simulateur.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.util.*;

public class Simulateur extends JApplet implements ActionListener
{
  private final static int  posStartX_c = 0;
  private final static int  posStartY_c = 0;
  private final static int  posEndX_c = 700;
  private final static int  posEndY_c = 550;
  private final static int  NbStats_c = 2;
  private final static int  NB_TESTS = 0;
  private final static int  NB_ROUNDS = 1;
  private final static int  GOOD = 0;
  private final static int  BAD = 1;
  private final static int  UNDEFINED = -1;
  private Graphics2D g2d;
  private Container container;
  private GridBagLayout displayLayout = new GridBagLayout();
  private GridBagConstraints displayConstraints = new GridBagConstraints();
  private JPanel buttonsPanel;
  private JPanel displayPanel;
  private JPanel textPanel;
  private JPanel analysePanel;
  private JPanel selectionPanel;
  private JPanel parameterSubPanel;
  private JPanel parameterSubPanel2;
  private JPanel algorithmSelectionSubPanel;
  private JPanel algorithmChoiceSubPanel;
  private JButton graphButton[];
```

```
private JTextArea infoTextArea;
private JTextArea analyseTextArea;
private JTextField nValue;
private JTextField tValue;
private JTextField xValue;
private JTextField repetition;
private long demoDelay = 10;
private JRadioButton densiteSlct;
private JRadioButton timeSlct;
private ButtonGroup groupBRadio;
private JCheckBox blecher;
private JCheckBox blecherRapide;
private JCheckBox balayageRapide;
private JCheckBox ntDemi;
private String graphButtonName[] = { "Exécuter", "Clear", "", ""};
private boolean networkStatus[];
private int networkAnalysis[];
private boolean networkStatus2[];
private int networkAnalysis2[];
private int statisticsResult[][];
private int statisticsResult2[][];
private int results[];
private int results2[];
private int capacity = 0;
private int restingCapacity = 0;
private int cpt = 0;

public void init(){
  container = getContentPane();
  container.setLayout( displayLayout );
  SetPanels();
  nValue.setText("100");
  tValue.setText("10");
  xValue.setText("0");
  repetition.setText("100");
  super.setSize( 1024, 768 );
  super.setVisible( true );
}

public void paint( Graphics g ){
  super.paint( g );
  g2d = ( Graphics2D ) g;
}

private void TracerContour(){
  g2d.setPaint( Color.black );
  g2d.setStroke( new BasicStroke( 1.0f ) );
  g2d.draw( new Rectangle2D.Double( posStartX_c, posStartY_c,
                                    posEndX_c, posEndY_c ) );
}
```

```
private int[] ExecuteBalayageRapide(int n, int t){
  results2 = new int[2];
  results2[NB_TESTS] = 0;
  results2[NB_ROUNDS] = 0;
  final double log = (Math.log10((double)t) / Math.log10((double)1.25));
  final int logInt = (int)Math.ceil(log);
  final int a  = n / (t+1);
  final int b  = (t+1)/(logInt + 5);
  final int c  = logInt + 5;
  final int c1 = c + ((t+1) % c);
  final int c11 = c + ((t+2) % c);
  final int nbNodeOutOfLine = n % (t+1);
  int Vgood = -1;
  int nouveauVecteur = 0;
  Vector<Vector> zeroLignes = new Vector<Vector>(t);
  int lambda;
  int j;
  int L[][][] = InitialeBalayageDeLigne(n, t);
  int G[] = new int[n];
  int G1[] = new int[n];

  for( int i = 0; i < n; ++i )
  {
      G[i] = UNDEFINED;
  }
  lambda = 0;

  for( int i1 = 0; i1 < a; ++i1 )
  {
      for( int i2 = 0; i2 < b; ++i2 )
      {
          results2[NB_TESTS]++;
          G[L[i1][i2][1]] = DoPmcTest(L[i1][i2][0], L[i1][i2][1]) ? GOOD : BAD;
      }
  }

  for( int i1 = 0; i1 < a; ++i1 )
  {
      for( int i2 = 0; i2 < b; ++i2 )
      {
          j = 1;
          while( j < c11 )
          {
              if( L[i1][i2][j] != -2 )
              {
                  if( G[L[i1][i2][j]] != BAD && L[i1][i2][j+1] != -2 )
                  {
                      results2[NB_TESTS]++;
                      G[L[i1][i2][j+1]] = DoPmcTest(L[i1][i2][j], L[i1][i2][j+1]) ? GOOD : BAD;
                  }
                  else
```

```
                {
                    lambda++;
                }
            }
            j++;
        }
    }
}
results2[NB_ROUNDS] += (c11-1);

boolean is0Ligne;
boolean is0SegmentLigne;
results2[NB_ROUNDS]++;
results2[NB_ROUNDS]++;
for( int i1 = 0; i1 < a; ++i1 )
{
    is0Ligne = true;
    for( int i2 = 0; i2 < b; ++i2 )
    {
        is0SegmentLigne = true;
        for( int i3 = 0; i3 < c11; ++i3 )
        {
            if( L[i1][i2][i3] != -2 )
            {
                if( G[L[i1][i2][i3]] == BAD )
                {
                    is0SegmentLigne = false;
                    break;
                }
            }
        }
        if( !is0SegmentLigne )
        {
            is0Ligne = is0SegmentLigne;
            break;
        }
    }
    if( is0Ligne )
    {
        for( int i2 = 0; i2 < b; ++i2 )
        {
            int Vi = 0;
            for( int i3 = 0; i3 < c11; ++i3 )
            {
                if( L[i1][i2][i3] != -2 )
                {
                    Vi = i3;
                }
            }
            boolean is0LigneParfaite = true;
            if( i2 < (b-1) )
```

```
                {
                    results2[NB_TESTS]++;
                    G[L[i1][i2+1][0]] = DoPmcTest(L[i1][i2][Vi], L[i1][i2+1][0]) ? GOOD : BAD;
                    if( G[L[i1][i2+1][0]] == BAD )
                    {
                        is0LigneParfaite = false;
                    }
                }
                else
                {
                    if( is0LigneParfaite )
                    {
                        results2[NB_TESTS]++;
                        G[L[i1][0][0]] = DoPmcTest(L[i1][i2][Vi], L[i1][0][0]) ? GOOD : BAD;
                        if( Vgood < 0 )
                        {
                            Vgood = L[i1][i2][Vi];
                        }
                        if( G[L[i1][0][0]] == GOOD )
                        {
                            for( int i4 = 0; i4 < b; ++i4 )
                            {
                                for( int i5 = 0; i5 < c11; ++i5 )
                                {
                                    if( L[i1][i4][i5] != -2 )
                                    {
                                        networkAnalysis2[L[i1][i4][i5]] = GOOD;
                                    }
                                }
                            }
                        }
                        else
                        {
                            networkAnalysis2[L[i1][0][0]] = BAD;
                            zeroLignes.addElement( new Vector<Integer>(t+2) );
                            for( int i4 = 0; i4 < b; ++i4 )
                            {
                                for( int i5 = 0; i5 < c11; ++i5 )
                                {
                                    if( L[i1][i4][i5] != -2 )
                                    {
                                        zeroLignes.lastElement().addElement
                                            (new Integer(L[i1][i4][i5]));
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
```

```
}

for( int o = 0; o < n; ++o )
{
    G1[o] = G[o];
}

results2[NB_ROUNDS]++;
for( int i1 = 0; i1 < a; ++i1 )
{
    for( int j1 = 0; j1 < b; ++j1 )
    {
        for( int k1 = 0; k1 < c11; ++k1 )
        {
            if(L[i1][j1][k1] != -2)
            {
                if( G[L[i1][j1][k1]] == UNDEFINED )
                {
                    results2[NB_TESTS]++;
                    G[L[i1][j1][k1]] = DoPmcTest(Vgood, L[i1][j1][k1]) ? GOOD : BAD;
                    networkAnalysis2[L[i1][j1][k1]] = G[L[i1][j1][k1]];
                    if( networkAnalysis2[L[i1][j1][k1]] == GOOD )
                    {
                        if( k1+1 < c11 && L[i1][j1][k1+1] != -2 ) k1++;
                        while( G[L[i1][j1][k1]] != UNDEFINED )
                        {
                            if( G[L[i1][j1][k1]] == GOOD )
                            {
                                networkAnalysis2[L[i1][j1][k1]] = GOOD;
                                if( k1+1 < c11 && L[i1][j1][k1+1] != -2 ) k1++;
                                else break;
                            }
                            else if( G[L[i1][j1][k1]] == BAD )
                            {
                                networkAnalysis2[L[i1][j1][k1]] = BAD;
                                break;
                            }
                        }
                    }
                    else
                    {
                        zeroLignes.addElement( new Vector<Integer>(t+2) );
                        zeroLignes.lastElement().addElement
                            (new Integer(L[i1][j1][k1]));

                        if( k1+1 < c11 && L[i1][j1][k1+1] != -2 ) k1++;
                        while( G[L[i1][j1][k1]] != UNDEFINED )
                        {
                            if( G[L[i1][j1][k1]] == GOOD )
                            {
                                zeroLignes.lastElement().addElement
```

```
                                    (new Integer(L[i1][j1][k1]));
                                if( k1+1 < c11 && L[i1][j1][k1+1] != -2 ) k1++;
                                else break;
                            }
                            else if( G[L[i1][j1][k1]] == BAD )
                            {
                                zeroLignes.lastElement().addElement
                                    (new Integer(L[i1][j1][k1]));
                                networkAnalysis2[L[i1][j1][k1]] = BAD;
                                break;
                            }
                        }
                    }
                }
            }
        }
    }
}

int T_r = n + t - results2[NB_TESTS] - 1;
while( T_r < (5 * zeroLignes.size()) && zeroLignes.size() > 0 )
{
    results2[NB_ROUNDS]++;
    for( int i = 0; i < zeroLignes.size(); ++i )
    {
        zeroLignes.elementAt(i).remove(0);
    }

    for( int i = 0; i < zeroLignes.size(); ++i )
    {
        T_r--;
        results2[NB_TESTS]++;
        networkAnalysis2[(Integer)zeroLignes.elementAt(i).firstElement()] =
        DoPmcTest(Vgood, (Integer)zeroLignes.elementAt(i).firstElement()) ? GOOD : BAD;
        if( networkAnalysis2[(Integer)zeroLignes.elementAt(i).firstElement()] == GOOD )
        {
            for( int k = 1; k < zeroLignes.elementAt(i).size(); ++k )
            {
                networkAnalysis2[(Integer)zeroLignes.elementAt(i).elementAt(k)] =
                    G[(Integer)zeroLignes.elementAt(i).elementAt(k)];
            }
        }
    }

    int C = 0;
    while( C < zeroLignes.size() )
    {
        if( networkAnalysis2[(Integer)zeroLignes.elementAt(C).firstElement()] == GOOD )
        {
            zeroLignes.remove(0);
        }
```

```
            else
            {
                C++;
            }
        }
    }

    int nbBads;
    int nbTests;
    int maxRound = 0;
    double log2;
    for( int i = 0; i < zeroLignes.size(); ++i )
    {
        nbBads = -1;
        for( int k = 0; k < zeroLignes.elementAt(i).size(); ++k )
        {
            networkAnalysis2[(Integer)zeroLignes.elementAt(i).elementAt(k)] =
                DoPmcTest(Vgood, (Integer)zeroLignes.elementAt(i).elementAt(k)) ? GOOD : BAD;

            if( networkAnalysis2[(Integer)zeroLignes.elementAt(i).elementAt(k)] == GOOD )
            {
                nbBads = k - 1;
            }
        }

        nbTests = 0;
        if( nbBads != -1 )
        {
            log2 = (Math.log10((double)(nbBads+1)) / Math.log10((double)2));
            if( nbBads == zeroLignes.elementAt(i).size()-1 )
            {
                nbTests = ((int)log2) +
                    ((G[(Integer)zeroLignes.elementAt(i).lastElement()] == BAD) ? 1 : 0 );
            }
            else
            {
                nbTests = (2 * (int)log2);
            }
        }
        if( maxRound < nbTests )
        {
            maxRound = nbTests;
        }
        results2[NB_TESTS] += nbTests;
    }
    results2[NB_ROUNDS] += maxRound;
    return results2;
}

private int[] ExecuteNtDemi(int n, int t){
    results2 = new int[2];
```

```
results2[NB_TESTS] = 0;
results2[NB_ROUNDS] = 0;
int F[] = new int[t+1];
int B[] = new int[t+1];
int nbFaults = 0;
int lgSequence = 1;
int lgLongestSequence = 0;
int Vgood = -1;
int VgoodPotential = -1;

for( int i = 0; i < t+1; ++i )
{
    F[i] = -1;
    B[i] = -1;
}

boolean testIsGood;
int f = 0;
int Vi;
for( Vi = 0; Vi < n && Vgood == -1; ++Vi )
{
    results2[NB_TESTS]++;
    testIsGood = DoPmcTest(Vi, (Vi + 1)%n);
    if( !testIsGood )
    {
        F[f] = Vi;
        B[f] = (Vi+2)%n;
        f++;
        nbFaults++;
        lgSequence = 1;
        if( VgoodPotential == Vi ) lgLongestSequence--;
        if( lgLongestSequence <= (t - nbFaults))
        {
            Vi++;
        }
    }
    else
    {
        lgSequence++;
        if( lgSequence > lgLongestSequence )
        {
            lgLongestSequence = lgSequence;
            VgoodPotential = Vi + 1;
        }
    }

    if( lgLongestSequence > (t - nbFaults))
    {
        Vgood = VgoodPotential;
        networkAnalysis[Vgood] = GOOD;
    }
```

```
}

if( networkAnalysis[(Vi+1)%n] != UNDEFINED ) Vi++;
for( Vi = Vi + 1; Vi <= n; ++Vi )
{
    if( networkAnalysis[Vi%n] == UNDEFINED )
    {
        results2[NB_TESTS]++;
        networkAnalysis[Vi%n] = DoPmcTest(Vgood,(Vi%n)) ? GOOD : BAD;
    }
}

if( networkAnalysis[0] == UNDEFINED )
{
    results2[NB_TESTS]++;
    networkAnalysis[0] = DoPmcTest(Vgood,0) ? GOOD : BAD;
}
Vi = 0;
while( Vi < (F[0] + 1) )
{
    if( networkAnalysis[Vi] == BAD )
    {
        if( networkAnalysis[Vi+1] == UNDEFINED )
        {
            results2[NB_TESTS]++;
            networkAnalysis[Vi+1] = DoPmcTest(Vgood,(Vi+1)) ? GOOD : BAD;
        }
    }
    else
    {
        if( Vi+1 == F[0]+1 )
        {
            if(  networkAnalysis[Vi+1] == UNDEFINED )
            {
                networkAnalysis[Vi+1] = BAD;
            }
        }
        else
        {
            if(  networkAnalysis[Vi+1] == UNDEFINED )
            {
                networkAnalysis[Vi+1] = GOOD;
            }
        }
    }
    Vi++;
}

nbFaults = 0;
int i;
for( i = 0; i < n; ++i )
```

```
{
    if( networkAnalysis[i] == BAD )
    {
        nbFaults++;
        if( nbFaults == t )
        {
            for( i = 0; i < n; ++i )
            {
                if(networkAnalysis[i] != BAD )
                {
                    networkAnalysis[i] = GOOD;
                }
            }
            results2[NB_ROUNDS] = results2[NB_TESTS];
            return results2;
        }
    }
}

boolean lineCompleted;
int Vj;
i = 0;
if( F[0] != -1 )
{
    while(B[i] != -1)
    {
        Vj = B[i];
        if( networkAnalysis[Vj] == UNDEFINED )
        {
            results2[NB_TESTS]++;
            networkAnalysis[Vj] = DoPmcTest(Vgood,Vj) ? GOOD : BAD;
        }

        if( networkAnalysis[Vj] == GOOD )
        {
            if( F[i+1] != -1 )
            {
                if( networkAnalysis[(F[i+1]+1)%n] == UNDEFINED  )
                {
                    networkAnalysis[(F[i+1]+1)%n] = BAD;
                    nbFaults++;
                    if( nbFaults == t )
                    {
                        for( i = 0; i < n; ++i )
                        {
                            if(networkAnalysis[i] != BAD )
                            {
                                networkAnalysis[i] = GOOD;
                            }
                        }
                        results2[NB_ROUNDS] = results2[NB_TESTS];
```

```
                return results2;
            }
        }
    }
    else
    {
        for( int h = B[i]+1; h < Vgood; ++h )
        {
            if( networkAnalysis[h] == UNDEFINED )
            {
                networkAnalysis[h] = GOOD;
            }
        }
    }

    for(int w = Vj+1; w < F[i+1]+1; ++w)
    {
        if( networkAnalysis[w] == UNDEFINED )
        {
            networkAnalysis[w] = GOOD;
        }
    }
}
else
{
    nbFaults++;
    lineCompleted = false;
    do
    {
        Vj++;
        if( nbFaults < t )
        {
            if( networkAnalysis[Vj] == UNDEFINED )
            {
                results2[NB_TESTS]++;
                networkAnalysis[Vj] = DoPmcTest(Vgood,Vj) ? GOOD : BAD;
            }
            if( networkAnalysis[Vj] == BAD )
            {
                nbFaults++;
                if( nbFaults == t )
                {
                    for( i = 0; i < n; ++i )
                    {
                        if(networkAnalysis[i] != BAD )
                        {
                            networkAnalysis[i] = GOOD;
                        }
                    }
                    results2[NB_ROUNDS] = results2[NB_TESTS];
                    return results2;
```

```
                        }
                    }
                    else
                    {
                        if( F[i+1]+1 != Vj )
                        {
                            if( networkAnalysis[(F[i+1]+1)%n] == UNDEFINED )
                            {
                                networkAnalysis[(F[i+1]+1)%n] = BAD;
                                nbFaults++;
                                if( nbFaults == t )
                                {
                                    for( i = 0; i < n; ++i )
                                    {
                                        if(networkAnalysis[i] != BAD )
                                        {
                                            networkAnalysis[i] = GOOD;
                                        }
                                    }
                                    results2[NB_ROUNDS] = results2[NB_TESTS];
                                    return results2;
                                }
                            }

                            for(int w = Vj+1; w < F[i+1]+1; ++w)
                            {
                                if( networkAnalysis[w] == UNDEFINED )
                                {
                                    networkAnalysis[w] = GOOD;
                                }
                            }
                        }
                        if( F[i+1]+1 == Vj ) lineCompleted = true;
                    }
                    else
                    {
                        if( networkAnalysis[Vj] == UNDEFINED )
                        {
                            networkAnalysis[Vj] = GOOD;
                        }
                    }
                }
                while((nbFaults < t) && (networkAnalysis[Vj] == BAD) && (!lineCompleted));
            }
            i++;
        }
    }
    else
    {
        for( int h = 0; h < Vgood; ++h )
```

```
        {
            if( networkAnalysis[h] == GOOD )
            {
                if( networkAnalysis[h+1] == UNDEFINED )
                {
                    networkAnalysis[h+1] = GOOD;
                }
            }
            else
            {
                if( networkAnalysis[h+1] == UNDEFINED )
                {
                    results2[NB_TESTS]++;
                    networkAnalysis[h+1] = DoPmcTest(Vgood,h+1) ? GOOD : BAD;
                }
                if( networkAnalysis[h+1] == BAD )
                {
                    nbFaults++;
                    if( nbFaults == t )
                    {
                        for( i = 0; i < n; ++i )
                        {
                            if(networkAnalysis[i] != BAD )
                            {
                                networkAnalysis[i] = GOOD;
                            }
                        }
                        results2[NB_ROUNDS] = results2[NB_TESTS];
                        return results2;
                    }
                }
            }
        }
    }

    results2[NB_ROUNDS] = results2[NB_TESTS];
    return results2;
}

private int[] ExecuteBlecherRapide(int n, int t){
    results = new int[2];
    results[NB_TESTS] = 0;
    results[NB_ROUNDS] = 0;
    int Vgood = -1;
    int nbGoodProcessorsFound = 0;
    int nbBadProcessorsFound = 0;
    boolean allFound = false;

    Vgood = BlecherFindGood( 0, n, t, 0 );
    int i;
    int Vj = 0;
```

```
if( restingCapacity > 0 )
{
    results[NB_ROUNDS]++;
    for( i = 0; i < restingCapacity; ++i )
    {
        while( networkAnalysis[Vj] != UNDEFINED && Vj < n )
        {
            Vj++;
        }
        if( Vj < n )
        {
            results[NB_TESTS]++;
            networkAnalysis[Vj] = DoPmcTest(Vgood, Vj) ? GOOD : BAD;
        }
    }
}

for( int k = 0; k < n; ++k )
{
    if(networkAnalysis[k] == GOOD) nbGoodProcessorsFound++;
    if(networkAnalysis[k] == BAD) nbBadProcessorsFound++;
}

if( (Vj + 1) < n )
{
    int k = nbGoodProcessorsFound;
    int nbGoodProcessorsFoundTemp = k;
    for( i = Vj + 1; i < n; ++i )
    {
        while( networkAnalysis[i] != UNDEFINED && i < n )
        {
            i++;
        }
        if( i < n )
        {
            results[NB_TESTS]++;
            nbGoodProcessorsFoundTemp--;
            if(DoPmcTest(Vgood, i))
            {
                networkAnalysis[i] = GOOD;
                nbGoodProcessorsFound++;
            }
            else
            {
                networkAnalysis[i] = BAD;
                nbBadProcessorsFound++;
            }

            if( nbGoodProcessorsFoundTemp == -1 )
            {
                results[NB_ROUNDS]++;
```

```
                nbGoodProcessorsFoundTemp = nbGoodProcessorsFound - 1;
            }
        }
    }
  }
  return results;
}

private int BlecherFindGood( int Vj, int n, int t, int p )
{
  int goods[] = new int[t];
  int bads[] = new int[t];
  int nbGood = 0;
  int nbBad = 0;
  int Vi = Vj + 1;
  int Vgood = -1;

  for( int i = 0; i < t; ++i )
  {
      goods[i] = -1;
      bads[i] = -1;
  }

  boolean isGood;
  while( (nbGood >= nbBad) && (nbGood < t) )
  {
      isGood = DoPmcTest(Vi,Vj);
      results[NB_TESTS]++;
      results[NB_ROUNDS]++;

      if( isGood )
      {
          goods[nbGood] = Vi;
          nbGood++;
      }
      else
      {
          bads[nbBad] = Vi;
          nbBad++;
      }
      Vi++;
  }

  if( nbGood != t )
  {
      int nbGoodProcessorsFound = 1;
      Vgood = BlecherFindGood( Vi, n, t-nbBad, p + 1);
      for( int i = 0; i < Vgood; ++i )
      {
          if(networkAnalysis[i] == GOOD) nbGoodProcessorsFound++;
      }
```

```
        if(capacity > 0)
        {
            results[NB_ROUNDS]++;
        }
        capacity -= nbGoodProcessorsFound;
        if( capacity < 0 ) restingCapacity = capacity * -1;

        results[NB_TESTS]++;
        if( DoPmcTest(Vgood,Vj) )
        {
            networkAnalysis[Vj] = GOOD;
            for( int i = 0; i < nbBad; i++ )
            {
                networkAnalysis[bads[i]] = BAD;
            }
        }
        else
        {
            networkAnalysis[Vj] = BAD;
            for( int i = 0; i < nbGood; i++ )
            {
                networkAnalysis[goods[i]] = BAD;
            }
        }
    }
    else
    {
        Vgood = Vj;
        capacity = p;
        networkAnalysis[Vj] = GOOD;
        for( int i = 0; i < nbBad; i++ )
        {
            networkAnalysis[bads[i]] = BAD;
        }
    }

    return Vgood;
}

private boolean DoPmcTest(int Vi, int Vj){
    if( networkStatus[Vi] == true )
    {
        return networkStatus[Vj];
    }
    else
    {
        return ((int)(Math.random() * 100)) < 50 ? true : false;
    }
}
```

```
private boolean ValidationParameters(int n, int t, int x, int r){
  boolean isValidated = true;

  if( (r > 1000) || (n <= (2 * t)) )
  {
      isValidated = false;
  }
  if( n >= 1000000 )
  {
      isValidated = false;
  }
  if( timeSlct.isSelected() && (n < ((2*(t*t))+(2 * t))) && balayageRapide.isSelected())
  {
      isValidated = false;
  }
  if( densiteSlct.isSelected() && ((((t+2)*(t+2))-(4*n)) >= 0) && ntDemi.isSelected())
  {
      isValidated = false;
  }
  if( !isValidated )
  {
      infoTextArea.append("Les paramètres n[" + n + "] et t[" + t +
          "]\nsont incorrects pour exécuter l'algorithme");
  }
  return isValidated;
}

private void AddGridBagComponent( GridBagLayout gbTraces, GridBagConstraints gbConstraints,
                                  Component component, int ligne,
                                  int colonne, int largeur, int hauteur, int weightx,
                                  int weighty ){

  displayConstraints.fill = GridBagConstraints.BOTH;
  gbConstraints.weightx = weightx;
  gbConstraints.weighty = weighty;
  gbConstraints.gridx = colonne;
  gbConstraints.gridy = ligne;
  gbConstraints.gridwidth  = largeur;
  gbConstraints.gridheight = hauteur;
  gbTraces.setConstraints( component, gbConstraints );
}

private class RadioButtonManager implements ItemListener  {
    public void itemStateChanged( ItemEvent evenement ){
        if( evenement.getSource() == densiteSlct )
        {
            blecherRapide.setVisible( false );
            balayageRapide.setVisible( false );
            ntDemi.setVisible( true );
            blecher.setVisible( true );
            nValue.setText("100");
```

```
            tValue.setText("10");
            xValue.setText("0");
        }
        else if( evenement.getSource() == timeSlct )
        {
            blecherRapide.setVisible( true );
            balayageRapide.setVisible( true );
            ntDemi.setVisible( false );
            blecher.setVisible( false );
            nValue.setText("5100");
            tValue.setText("50");
            xValue.setText("0");
        }
    }
};

private boolean IsResultsSimilar( boolean origine[], int resultats[] ){
  if( origine.length == resultats.length )
  {
      for( int i = 0; i < origine.length; ++i )
      {
          if(!( ( (origine[i] == true)  && (resultats[i] == GOOD) ) ||
              ( (origine[i] == false) && (resultats[i] == BAD)  ) ))
          {
              return false;
          }
      }
  }
  return true;
}

private int[][][] InitialeBalayageDeLigne(int n, int t){
  final double log = (Math.log10((double)t) / Math.log10((double)1.25));
  final int logInt = (int)Math.ceil(log);
  final int a  = n / (t+1);
  final int b  = (t+1)/(logInt + 5);
  final int c  = logInt + 5;
  final int c1 = c + ((t+1) % c);
  final int c11 = c + ((t+2) % c);
  final int nbNodeOutOfLine = n % (t+1);

  int L[][][] = new int[a][b][c11+1];
  for( int i = 0; i < a; ++i )
  {
      for( int j = 0; j < b; ++j )
      {
          for( int k = 0; k < c11+1; ++k )
          {
              L[i][j][k] = -2;
          }
      }
```

```
  }

  int i1;
  int j1;
  int k1;
  int cmpt = 0;
  for( i1 = 0; i1 < a; ++i1 )
  {
      for( j1 = 0; j1 < b; ++j1)
      {
          if( j1 == b - 1 )
          {
              for( k1 = 0; k1 < c1; ++k1 )
              {
                  L[i1][j1][k1] = cmpt++;
              }
              if( i1 < nbNodeOutOfLine ) // Ligne t+2
              {
                  L[i1][j1][k1] = cmpt++;
              }
          }
          else
          {
              for( k1 = 0; k1 < c; ++k1 )
              {
                  L[i1][j1][k1] = cmpt++;
              }
          }
      }
  }
  return L;
}

//---------------------------------------------------------------------------

public void actionPerformed( ActionEvent event ){
  int buttonEventNb = 0;
  for( int i = 0; i < graphButtonName.length && buttonEventNb == 0; ++i ){
    if( event.getSource() == graphButton[i] )
      buttonEventNb = i;
  }

  analyseTextArea.setText("");
  switch( buttonEventNb ){
    case 0 :
      int n = Integer.parseInt(nValue.getText());
      int t = Integer.parseInt(tValue.getText());
      int x = Integer.parseInt(xValue.getText());
      if( x == 0 ) x = t;
      int r = Integer.parseInt(repetition.getText());
```

```
if( ValidationParameters( n, t, x, r ) )
{
    statisticsResult = new int[r][NbStats_c];
    statisticsResult2 = new int[r][NbStats_c];
    boolean notSimilar;
    for( int j = 0; j < r; j++ )
    {
        notSimilar = false;
        networkStatus = new boolean[n];
        networkAnalysis = new int[n];
        networkStatus2 = new boolean[n];
        networkAnalysis2 = new int[n];
        for( int i = 0; i < n; ++i )
        {
            networkStatus[i] = true;
            networkAnalysis[i] = UNDEFINED;
            networkStatus2[i] = true;
            networkAnalysis2[i] = UNDEFINED;
        }
        for( int i = 0; i < x && i < t; ++i )
        {
            int nbAleat = (int)(Math.random() * n);
            networkStatus[nbAleat] = false;
            networkStatus2[nbAleat] = false;
        }
        if( densiteSlct.isSelected() )
        {
            boolean selected = false;
            if( blecher.isSelected() )
            {
                selected = true;
                int resultats[] = ExecuteBlecherRapide(n, t);
                if( !IsResultsSimilar(networkStatus, networkAnalysis) )
                {
                    notSimilar = true;
                }
                statisticsResult[j][NB_TESTS] = resultats[NB_TESTS];
                statisticsResult[j][NB_ROUNDS] = resultats[NB_ROUNDS];
            }

            if( ntDemi.isSelected() )
            {
                selected = true;
                networkStatus = networkStatus2;
                networkAnalysis = networkAnalysis2;
                int resultats2[] = ExecuteNtDemi(n, t);
                if( !IsResultsSimilar(networkStatus, networkAnalysis) )
                {
                    notSimilar = true;
                }
                statisticsResult2[j][NB_TESTS] = resultats2[NB_TESTS];
```

```
                statisticsResult2[j][NB_ROUNDS] = resultats2[NB_ROUNDS];
            }

            if( !selected )
            {
                infoTextArea.setText("\nAucun algorithme n'a été sélectionné");
            }
        }
        else if( timeSlct.isSelected() )
        {
            boolean selected = false;
            if( blecherRapide.isSelected() )
            {
                selected = true;
                int resultats[] = ExecuteBlecherRapide(n, t);
                if( !IsResultsSimilar(networkStatus, networkAnalysis) )
                {
                    notSimilar = true;
                }
                statisticsResult[j][NB_TESTS] = resultats[NB_TESTS];
                statisticsResult[j][NB_ROUNDS] = resultats[NB_ROUNDS];
            }

            if( balayageRapide.isSelected() )
            {
                selected = true;
                networkStatus = networkStatus2;
                networkAnalysis = networkAnalysis2;
                int resultats2[] = ExecuteBalayageRapide(n, t);
                if( !IsResultsSimilar(networkStatus, networkAnalysis) )
                {
                    notSimilar = true;
                }
                statisticsResult2[j][NB_TESTS] = resultats2[NB_TESTS];
                statisticsResult2[j][NB_ROUNDS] = resultats2[NB_ROUNDS];
            }

            if( !selected )
            {
                infoTextArea.setText("\nAucun algorithme n'a été sélectionné");
            }
        }
        if( notSimilar ) j--;
    }

    int sumTest;
    int sumRound;
    int averageTest;
    int averageRound;
    if( densiteSlct.isSelected() )
    {
```

```
        if( blecher.isSelected() )
        {
            sumTest = 0;
            sumRound = 0;
            averageTest = 0;
            averageRound = 0;
            analyseTextArea.append("\n\n ALGORITHME DE BLECHER \n");
            for( int i = 0; i < r; ++i )
            {
                sumTest += statisticsResult[i][NB_TESTS];
                sumRound += statisticsResult[i][NB_ROUNDS];
            }

            averageTest = sumTest / r;
            averageRound = sumRound / r;

            analyseTextArea.append("\n N               = " + n);
            analyseTextArea.append("\n T               = " + t);
            analyseTextArea.append("\n Nombre de Tests = " + averageTest);
            analyseTextArea.append("\n Sur               " + r + " itérations ! \n");
        }

        if( ntDemi.isSelected() )
        {
            sumTest = 0;
            sumRound = 0;
            averageTest = 0;
            averageRound = 0;
            analyseTextArea.append("\n\n ALGORTIHME ADA-T-DIADNOSTIC-OPTIMAL \n");
            for( int i = 0; i < r; ++i )
            {
                sumTest += statisticsResult2[i][NB_TESTS];
                sumRound += statisticsResult2[i][NB_ROUNDS];
            }

            averageTest = sumTest / r;
            averageRound = sumRound / r;

            analyseTextArea.append("\n N               = " + n);
            analyseTextArea.append("\n T               = " + t);
            analyseTextArea.append("\n Nombre de Tests = " + averageTest);
            analyseTextArea.append("\n Sur             " + r + " itérations ! \n");
        }
    }
    else
    {
        if( blecherRapide.isSelected() )
        {
            sumTest = 0;
            sumRound = 0;
            averageTest = 0;
```

```
                averageRound = 0;
                analyseTextArea.append("\n\n ALGORITHME DE BLECHER RAPIDE \n");
                for( int i = 0; i < r; ++i )
                {
                    sumTest += statisticsResult[i][NB_TESTS];
                    sumRound += statisticsResult[i][NB_ROUNDS];
                }

                averageTest = sumTest / r;
                averageRound = sumRound / r;

                analyseTextArea.append("\n N                = " + n);
                analyseTextArea.append("\n T                = " + t);
                analyseTextArea.append("\n Nombre de Tests  = " + averageTest);
                analyseTextArea.append("\n Nombre de Rounds = " + averageRound);
                analyseTextArea.append("\n Sur " + r +        " itérations ! \n");
            }

            if( balayageRapide.isSelected() )
            {
                sumTest = 0;
                sumRound = 0;
                averageTest = 0;
                averageRound = 0;
                analyseTextArea.append("\n\n ALGORITHME BALAYAGE-RAPIDE \n");
                for( int i = 0; i < r; ++i )
                {
                    sumTest += statisticsResult2[i][NB_TESTS];
                    sumRound += statisticsResult2[i][NB_ROUNDS];
                }

                averageTest = sumTest / r;
                averageRound = sumRound / r;

                analyseTextArea.append("\n N        =         " + n);
                analyseTextArea.append("\n T        =         " + t);
                analyseTextArea.append("\n Nombre de Tests  = " + averageTest);
                analyseTextArea.append("\n Nombre de Rounds = " + averageRound);
                analyseTextArea.append("\n Sur " + r +        " itérations ! \n");
            }
        }
      }

    break;
    case 1 :
      analyseTextArea.setText( "" );
    break;
    case 2 :
      analyseTextArea.setText( "" );
    break;
    case 3 :
```

```
      analyseTextArea.setText( "" );
    break;
  }
}

//---------------------------------------------------------------------------

private void SetPanels()
{
  analysePanel = new JPanel();
  analysePanel.setLayout( new GridLayout(1,1));
  AddGridBagComponent( displayLayout, displayConstraints, analysePanel, 0, 0, 2, 8, 1, 1 );
  container.add( analysePanel );

  buttonsPanel =  new JPanel();
  buttonsPanel.setLayout( new GridLayout( 2, 2, 5, 5 ) );
  AddGridBagComponent( displayLayout, displayConstraints, buttonsPanel, 0, 2, 1, 1, 0, 0 );
  container.add( buttonsPanel );

  selectionPanel =  new JPanel();
  selectionPanel.setLayout( new GridLayout( 4, 1) );
  AddGridBagComponent( displayLayout, displayConstraints, selectionPanel, 1, 2, 1, 1, 0, 0 );
  container.add( selectionPanel );

  textPanel = new JPanel();
  textPanel.setLayout( new GridLayout(1,1));
  AddGridBagComponent( displayLayout, displayConstraints, textPanel, 2, 2, 1, 6, 0, 0 );
  container.add( textPanel );

  graphButton = new JButton[ graphButtonName.length ];
  for( int i = 0; i < graphButtonName.length; ++i )
  {
    graphButton[i] = new JButton( graphButtonName[i] );
    graphButton[i].addActionListener( this );
    buttonsPanel.add( graphButton[i] );
  }

  infoTextArea = new JTextArea();
  textPanel.add( infoTextArea );

  analyseTextArea = new JTextArea();
  analysePanel.add( analyseTextArea );

  parameterSubPanel = new JPanel();
  FlowLayout flowLayout = new FlowLayout();
  parameterSubPanel.setLayout(flowLayout);
  flowLayout.setAlignment( FlowLayout.LEFT);
  selectionPanel.add(parameterSubPanel);
  parameterSubPanel.add( new JLabel("n: "));
  nValue = new JTextField("",5);
  parameterSubPanel.add( nValue );
```

```
    parameterSubPanel.add( new JLabel("t: "));
    tValue = new JTextField("",5);
    parameterSubPanel.add( tValue );
    parameterSubPanel.add( new JLabel("x: "));
    xValue = new JTextField("",5);
    parameterSubPanel.add( xValue );

    FlowLayout flowLayout2 = new FlowLayout();
    parameterSubPanel2 = new JPanel();
    parameterSubPanel2.setLayout(flowLayout2);
    flowLayout2.setAlignment( FlowLayout.LEFT);
    selectionPanel.add(parameterSubPanel2);
    parameterSubPanel2.add( new JLabel("repetition: "));
    repetition = new JTextField("",5);
    parameterSubPanel2.add( repetition );

    algorithmChoiceSubPanel = new JPanel();
    algorithmChoiceSubPanel.setLayout( new FlowLayout());
    selectionPanel.add(algorithmChoiceSubPanel);
    densiteSlct = new JRadioButton( "Densité des réseaux", true);
    algorithmChoiceSubPanel.add( densiteSlct );
    timeSlct = new JRadioButton( "Temps d'exécution", false);
    algorithmChoiceSubPanel.add( timeSlct );
    RadioButtonManager gestionnaire = new RadioButtonManager();
    densiteSlct.addItemListener(gestionnaire);
    timeSlct.addItemListener(gestionnaire);
    groupBRadio = new ButtonGroup();
    groupBRadio.add(densiteSlct);
    groupBRadio.add(timeSlct);

    algorithmSelectionSubPanel = new JPanel();
    algorithmSelectionSubPanel.setLayout( new GridLayout(4, 1));
    selectionPanel.add(algorithmSelectionSubPanel);
    blecher = new JCheckBox("Blecher");
    algorithmSelectionSubPanel.add(blecher);
    ntDemi = new JCheckBox("Ada-T-Diagnostic-Optimal");
    algorithmSelectionSubPanel.add(ntDemi);
    blecherRapide = new JCheckBox("Blecher-Rapide");
    algorithmSelectionSubPanel.add(blecherRapide);
    balayageRapide = new JCheckBox("Balayage-Rapide");
    algorithmSelectionSubPanel.add(balayageRapide);

    blecher.setSelected( true );
    ntDemi.setSelected( true );
    blecherRapide.setSelected( true );
    balayageRapide.setSelected( true );
    balayageRapide.setVisible( false );
    blecherRapide.setVisible( false );
  }
}
```

Bibliographie

[1] BARBORAK M., DAHBURA A., and MALEK M.. The consensus problem in fault-tolerant computing. *ACM Comput. Surv.*, 25(2) :171–220, 1993.

[2] BARSI F., GRANDONI F. and MAESTRINI P. A theory of diagnosability of digital systems. *IEEE Trans. Comput. C-25, 6*, pages 585–593, 1976.

[3] BEIGEL R., HURWOOD W. and N. KAHALI. Fault diagnosis in a flash. *Proceedings of the 36th Annual Symposium on Foundations of Computer Science, IEEE*, pages 571–580, 1995.

[4] BEIGEL R., KOSARAJU S.R., and SULLIVAN G.F.. Locating faults in a constant number of parallel testing rounds. In *SPAA '89 : Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 189–198, New York, NY, USA, 1989. ACM Press.

[5] BEIGEL R., MARGULIS G., and SPIELMAN D.A. Fault diagnosis in a small constant number of parallel testing rounds. In *SPAA '93 : Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 21–29, New York, NY, USA, 1993. ACM Press.

[6] BLECHER P.M. On a logical problem. *Discr. Math. 43*, pages 107–110, 1983.

[7] BLOUGH D.M., SULLIVAN G.F. and MASSON G.M. Efficient diagnosis of multiprocessor systems under probabilistics models. *IEEE Trans. Comput. 41*, pages 1126–1136, 1992.

[8] BLOUGH D.M., SULLIVAN G., and MASSON G. System diagnosis, fault tolerant computer system design. *D.K. Pradhan,Prentice-Hall*, pages 478–536, 1996.

[9] BJORKLUND A. Optimal adaptive fault diagnosis of hypercubes, Proc. Scandinavian Workshop on Algorithm Theory, SWAT 2000, LNCS 1851, 527-534.

[10] CIOMPI P., GRANDONI F., and SIMONCINI L. Distributed diagnosis in multiprocessor system : The muteam approach. *In the 11th International IEEE Symposium on Fault-Tolerant Computing*, pages 25–29, 1981.

[11] DAHBURA A. and MASSON G.M. An o($n^{2.5}$) fault identification algorithm for diagnosable systems. *In the 14th International IEEE Symposium on Fault-Tolerant Computing*, pages 428–433, 1984.

[12] HAKIMI S.L. and AMIN A.T. Characterization of connection assignment of diagnosable systems. *IEEE Trans. Comput. 23*, pages 86–88, 1974.

[13] HAKIMI S.L. and NAKAJIMA K. On adaptive system diagnosis. *IEEE Trans. Comput. 33*, pages 234–240, 1984.

[14] KREUTZER S. and HAKIMI S. Adaptative fault identification in two new diagnostic models. *Proceedings of the 21st Allerton Conference on Communication, Control and Computing*, pages 353–362, 1983.

[15] KREUTSER S. and HAMIKI S. Distributed diagnosys and the system user. *IEEE Trans. Comput., Vol. C-37, No. 1*, pages 71–78, 1988.

[16] KUHL J. and REDDY S. Distributed fault-tolerance for large multiprocessor systems. *In proceeding of the 7th anual symposium on computer*, pages 23–30, 1980.

[17] KUHL J. and REDDY S. Fault Diagnosis in fully distributed system. *In the 11th International IEEE Symposium on Fault-Tolerant Computing*, pages 100–105, 1981.

[18] MALEK M. A comparison connection assignment for diagnosis of multiprocessor systems. *In Proceedings of the 7th Anual Symposium on Computer Architecture*, pages 31–36, 1980.

[19] NAKAJIMA K. A new approch to system diagnosis. *Proc. 19th Annual Allerton Conf. Commun., Contr. and Comput.*, pages 697–706, 1981.

[20] NOMURA K., YAMADA T., UENO S. On adaptive fault diagnosis for multiprocessor systems. Proc. ISAAC 2001, LNCS 2223, 86-98.

[21] OKASHITA A., ARAKI T., SHIBATA Y. An optimal adaptive diagnosis of butterfly networks IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences E86A (2003), 1008-1018.

[22] PELC A. Undirected graph models for system-level fault diagnosis. *IEEE Trans. Comput. 40*, pages 1271–1276, 1991.

[23] PELC A. et SPATHARIS A. KRANAKIS E. Optimal adaptive fault diagnosis for simple multiprocessor systems. 1999.

[24] PELC A. et UPFAL E. Reliable fault diagnosis with few tests. *Combin. Probab. Comput. 7*, pages 323–333, 1998.

[25] PRAPARATA F.P., METZE G. and CHIEN R.T. On the connection assignment problem of diagnosable systems. *IEEE Trans. Elect. Comput. 12*, pages 848–854, 1967.

[26] WU C.-C. Partial solution to problem 81-6. *J. of Algorithms 3*, pages 379–380, 1982.