MASTER 2 SRO 2007-2008

Problèmes d'ordonnancement tolérant aux panes

7 février 2008

Guillaume TATTI
Nicoleta POPA
Bertrand SAINTES
Maxime MICHELIER

Tuteur

M. Moustafa NAKECHBANDI : HDR à l'Université du Havre

INTRODUCTION 4

1. PRÉSENTATION DU PROJET 4
2. RÉSUME DU TRAVAIL RÉALISÉ 5

1. SIMULATION DE PANNES ET RÉSISTANCE AUX PANNES DE RÉGIONS 6

1.1. SIMULATION DE PANNE SUR UNE MACHINE 6
1.2. RÉSISTANCE AUX PANNES DE RÉGIONS 6
1.2.1 DÉFINITION 6
1.2.2 Mise en place 7
1.2.3 Fonctionnement 7
1.2.4 Critiques et observations 8

2. QUALITÉ DES GRAPHES DE TÂCHES 10

2.1 DESCRIPTION 10
2.2 TYPES DE GRAPHES DE TÂCHES 11
2.2.1 GRAPHE BUTTERFLY 11
2.2.2 GRAPHE GRILLE 11
2.2.3 GRAPHE SÉRIE-PARALLÈLE 12
2.3 LE FICHIER DE TRACE 12
2.4. RÉALISATION 13
2.4.1 TEST_DSS_BUTTERFLY 13
2.4.2 TEST_DSS_GRILLE 13
2.4.3 TEST_DSS_SERIEPARALLELE 14

3. INTERFACE GRAPHIQUE 15

3.1 DESCRIPTION 15
3.2 NAVIGATION 15
3.3. SAISIE MANUELLE 15
3.4 UTILISER UN FICHIER 18
3.5 LE DIAGRAMME DE GANTT 19
3.5.1 DESCRIPTION 19
3.5.2 MISE EN ŒUVRE 20

4. STATISTIQUES 22

4.1 MAKESPAN EN FONCTION DU NOMBRE DE MACHINES 22
4.2. MAKESPAN EN FONCTION DU NOMBRE DE TÂCHES 23

4.2.1 GRAPHE ALÉATOIRE 24
4.2.2 GRAPHE GRILLE 25
4.2.3 GRAPHE SÉRIE-PARALLÈLE 26
4.2.4 GRAPHE BUTTERFLY 27
4.2.5 COMPARAISON DES COURBES 28
4.3 INTERPRÉTATION 29
4.3.1 GRAPHE ALÉATOIRE 29
4.3.2 GRAPHE GRILLE 30
4.3.3 GRAPHE SÉRIE-PARALLÈLE 31
4.3.4 GRAPHE BUTTERFLY 32

CONCLUSION 33

PROBLÈMES D'ORDONNANCEMENT TOLÉRANT AUX PANNES

Introduction

1. Présentation du projet

L'ordonnancement de tâches est un travail récurent en informatique. En effet, le processeur passe son temps à allouer ou désallouer des ressources pour l'exécution de tâches données. Ce fonctionnement dit séquentiel peut être amélioré en parallélisant l'exécution de tâches indépendantes sur différentes machines. Plusieurs algorithmes existent pour produire un ordonnancement optimal à un problème donné dans un environnement distribué. Toutefois, la distribution de tâches sur des machines différentes, probablement éloignées les unes des autres implique de prendre en compte le facteur panne aussi bien au niveau des machines elles-mêmes qu'au niveau des moyens de communication. En effet, plus le nombre de machines augmente plus on prend le risque de subir une panne de l'une d'entre elles, il en va de même pour les moyens de communication.

Une solution proposée par J-Y COLIN, M NAKECHBANDI et J-B GASHUMBA permet d'obtenir un ordonnancement efficace résistant à une panne à partir d'un ordonnancement optimal. Malheureusement cette solution ne prend en compte qu'une seule panne machine. Qu'advient-il de notre ordonnancement lorsqu'un ensemble de machines situées dans une même région tombe en panne ? Comment s'assurer d'obtenir un résultat final ? Afin de répondre à cette problématique, la solution avancée consiste à améliorer l'algorithme de J-Y COLIN, M NAKECHBANDI et J-B GASHUMBA en ajoutant la notion de région et en étendant la notion de panne non plus à une seule machine mais une région.

2. Résume du travail réalisé

Ainsi, nous verrons dans une première partie l'analyse de l'existant ainsi que les différents choix de développement qui ont été fait. Puis, nous étudierons le fonctionnement du nouvel algorithme résistant à une panne de région. De plus, nous verrons ensuite les différents types de graphes que le programme est capable de générer. Ensuite, nous aborderons le travail réalisé sur l'interface homme machine afin de simplifier l'utilisation du programme. Enfin, dans une dernière partie, nous étudierons les différents résultats statistiques obtenus en exécutant à plusieurs reprises notre programme.

1. Simulation de pannes et résistance aux pannes de régions

1.1. Simulation de panne sur une machine

La mise en place d'une méthode permettant de simuler une panne survenant sur une machine permet de tester la validité d'un ordonnancement produit par eFRCD.

En entrée, on fournit un ordonnancement produit par eFRCD et on récupère en sortie un ensemble de tâches correspondant au chemin parcouru pour arriver au bout du programme que l'on a voulu paralléliser avec nos algorithmes d'ordonnancement DSS_OPT et eFRCD.

Pour y parvenir, on commence par "élaguer" notre ordonnancement retourné par eFRCD en retirant toutes les tâches qui s'exécutent sur la machine que l'on souhaite mettre en panne. Ensuite on parcourt de manière itérative ce nouvel ensemble épuré en marquant les tâches par lesquelles on passe afin d'éviter de passer plusieurs fois sur une même tâche. En effet, on sait par hypothèse que la tâche d'indice i ne possède pas de prédécesseurs dans les tâches d'indice i+1. Ainsi, en parcourant notre ordonnancement de manière itérative, on est assuré de respecter les contraintes relations entre toutes les tâches. Une fois arrivé au bout de notre itération, on vérifie que notre chemin passe bien par toutes les tâches du programme ce qui nous garantit que l'on n'en a omis aucune.

1.2. Résistance aux pannes de régions

1.2.1 Définition

Par hypothèse, on définit une région comme un ensemble de machines. Chaque région est composée d'au moins une machine et une même machine ne peut appartenir qu'à une seule région. Ainsi, au maximum, il y'a autant de régions que de machines et au minimum deux régions.

De ce fait, la résistance à une panne de région implique que notre algorithme doit être en mesure de fournir un ordonnancement capable de terminer son traitement quelque soit la région qui tombe en panne.

1.2.2 Mise en place

La modélisation des régions dans notre système passe par la génération d'un nouveau tableau qui vient s'ajouter aux matrices déjà utilisées par l'algorithme eFRCD résistant à une panne. Ce tableau, dont la taille correspond au nombre de machines, contient dans la case i le numéro de la région à laquelle appartient la machine i comme le résume le schéma suivant :

0	……	i **… …**	MAX_PROCS
indice_de_région		R	indice_de_région

On constate sur ce tableau que la machine i appartient à la région R. Ce tableau nous permet ensuite d'accéder à l'information très simplement. Plusieurs questions se posent à propos de la méthode utilisée pour répartir les machines dans les différentes régions comme nous le verront plus tard (cf sous-partie Critiques et observations).

Afin de simplifier encore l'utilisation de notre tableau, une méthode statique a été ajoutée, intitulée estDansLaRegion(id_machine). Cette dernière retourne le numéro de la région à laquelle appartient la machine dont l'identifiant est passé en paramètre.

1.2.3 Fonctionnement

Pour parvenir à un ordonnancement résistant aux pannes de régions, il faut appliquer deux conditions principales lorsque l'on souhaite placer les copies des tâches principales.

Tout d'abord, il faut s'assurer que la copie d'une tache i nommée i' s'exécute bien sur une machine différente que celle qui accueille la tâche i originelle et que la tâche originelle et sa copie respecte bien les contraintes de relations avec leurs prédécesseurs (cf Rapport sur l'algorithme eFRCD). En appliquant cella, on obtient un ordonnancement résistant à une panne de machine.

Maintenant, en étoffant la condition suivante, on obtient un ordonnancement résistant à une panne de région. Il faut s'assurer que la machine qui exécutera la copie de la tâche i se trouve dans une région différente que celle qui exécute la tâche i. Or, on sait que l'algorithme eFRCD fournit un ordonnancement efficace résistant à une panne de machine. Ainsi, en appliquant cette condition à eFRCD lorsqu'on élit la machine qui permettra d'exécuter le plus rapidement la copie de la tâche i, on s'assure de déterminer la meilleure machine située dans une région différente pour notre copie de la tâche i. De ce fait, on obtient un algorithme produisant un ordonnancement résistant à une panne de région efficace. En résumé, une petite modification de l'algorithme eFRCD nous a permis d'obtenir le nouvel algorithme voulu, efficace de surcroît.

1.2.4 Critiques et observations

- ***Représenter la distance entre les régions***

Plusieurs remarques peuvent être émises aussi bien sur la manière de représenter les données du problème que sur la façon dont on les traite.

Tout d'abord, l'ajout de la notion de régions n'a que partiellement été représentée. En effet, elle a uniquement ajouté un tableau à nos données initiales mais n'a aucunement modifiée les coûts de communication entre les machines. Ainsi, si on prend un ensemble de régions numérotées de 0 à N, on peut supposer que la

région R_0 est plus éloignée de la région R_i que de la région R_1. De ce fait, le coût pour transmettre une unité de données entre deux machines situées dans des régions éloignées devrait logiquement être plus important que deux machines situées dans des régions proches. Or, ces données sur les coûts de communication entre les machines ne sont pas modifiées. Il serait peut-être judicieux d'incorporer cette notion de distance quand on génère les coûts de communication entre les machines afin d'obtenir une situation plus réaliste.

- ***Répartir les machines dans les régions***

Pour qu'une répartition des machines soit valide, il faut que chaque tâche puisse être exécutée par au moins deux machines distinctes situées dans deux régions différentes. Cette répartition est grandement facilitée par le fait que la partie de l'algorithme chargée d'initialiser nos données impose que toutes les machines puissent exécuter toutes les tâches, même si les temps d'exécution vont varier selon les machines pour une tâche donnée. Dans cette situation, une répartition valide peut se résumer à la condition suivante : *nombre_de_régions >= 2*

Pour illustrer cella, prenons un exemple. Soit un problème d'ordonnancement de n tâches et de m machines disponibles. Si toutes les machines sont capables d'exécuter les n tâches, le nombre de régions nécessaires pour obtenir un ordonnancement résistant à une panne de région est de 2, puisque si la région R_0 tombe en panne, toutes les tâches perdues pourront être exécutées par des machines situées dans la région R_1.

Maintenant, le problème se complique si on modifie les données initiales en spécifiant que les machines ne sont plus capables d'exécuter toutes les tâches. En effet, trouver une répartition valide pour la tâche i ne signifie pas qu'elle sera

nécessairement valide pour la tâche i+1. Plusieurs solutions peuvent être envisagées pour résoudre ce problème. Il est sûrement possible d'appliquer les méthodes utilisées pour un problème classique de l'intelligence artificielle qu'est le problème des N-reines puisqu'il s'agit également d'un problème de répartition. En effet, on pourrait se pencher sur l'implémentation d'un algorithme avec backtracking, ou bien un recuit-simulé qui donne de très bons résultats sur ce type de problème. On constate donc que la mise en place des régions dans notre algorithme, simple en apparence, fait apparaître un second problème qui mérite de la réflexion et du travail si on souhaite l'implémenter complètement.

Pour l'instant, l'algorithme qui gère la répartition des régions se contente de générer une répartition aléatoire et de vérifier que pour chaque tâche, il existe au moins deux machines distinctes situées dans deux régions distinctes capables de l'exécuter. Si cette condition n'est pas remplie, l'algorithme recommence jusqu'à trouver une répartition valide, ce qui garantit que la seconde partie de l'algorithme puisse trouver un ordonnancement résistant à une panne de région.

2. Qualité des graphes de tâches

2.1 Description

Un problème rencontré a été la génération des graphes des tâches des problèmes d'ordonnancement : cette génération étant faite aléatoirement, on obtenait des graphes de tâches difficiles à exploiter.

Un des buts du projet a été de créer des algorithmes qui permettent de générer des graphes de tâches avec une structure précise, et d'étudier l'impact de la structure sur l'ordonnancement.

2.2 Types de graphes de tâches

2.2.1 Graphe Butterfly

Un graphe Butterfly est une structure récursive caractérisée par ce que l'on va appeler l'ordre du graphe. Un graphe Butterfly d'ordre N et en fait composer de 2 graphes Butterfly d'ordre (N-1).

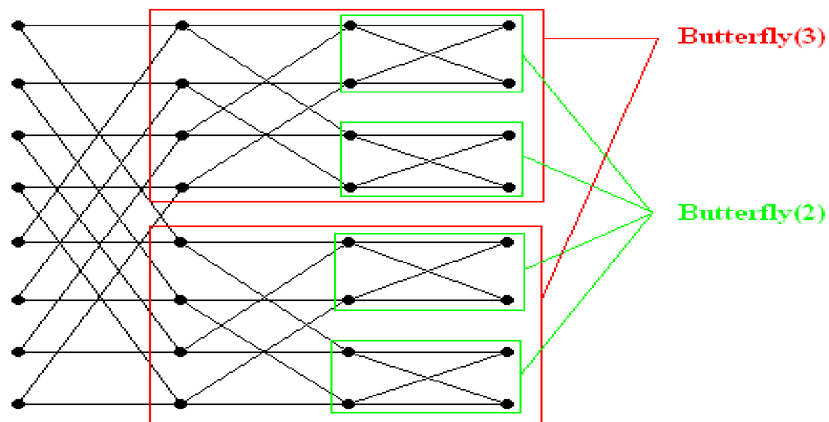


Figure1. Graphe Butterfly d'ordre 4

Un graphe Butterfly d'ordre N peut être séparé en N couches qui contiennent chacune $2^{(N-1)}$ tâches. Il comporte donc en tout ($N * 2^{(N-1)}$) tâches.

2.2.2 Graphe Grille

Un graphe Grille est caractérisé par le nombre de lignes L et le nombre de colonnes C du graphe, et comporte alors LxC tâches.

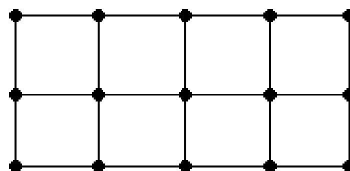


Figure 2. Graphe Grille à 3 lignes et 5 colonnes

2.2.3 Graphe Série-Parallèle

Un graphe Série-Parallèle est caractérisé par un nombre de tâches du graphe, ainsi que par le nombre de ce que l'on appellera "séries", des groupes de tâches parallèles, séparés par une tâche qui agit comme une sorte de "goulot d'étranglement" : cette tâche dépend de toutes les tâches de la série précédente, et est nécessaire à l'exécution des tâches de la série suivante.

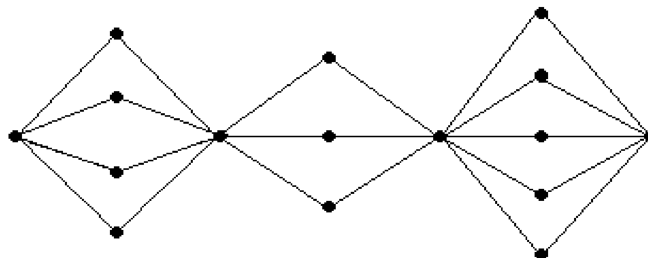


Figure 3. Graphe Série-Parallèles de 16 tâches et 3 séries.

2.3 Le fichier de trace

Afin de garder la trace du type de graphe de tâches utilisé, le fichier de trace généré par l'algorithme d'ordonnancement, et représentant un problème d'ordonnancement, a été complété.

La partie réservée pour la matrice de lien, qui représente le graphe de tâches, se présente maintenant sous cette forme :

```
[Matrice de liens :]
[Type de graphe :] <type de graphe>
<coefficients de la matrice de liens>
```

où <type de graphe> peut être "Aléatoire", "Butterfly", "Grille", ou "Série-Parallèle".

2.4. Réalisation

3 nouvelles classes, dérivant de Test_DSS_avec_retard, permettent de générer des graphes de tâches de type défini en surchargeant la méthode initMatrice() :

2.4.1 Test_DSS_Butterfly

Cette classe génère des graphes de tâches de type Butterfly. Le programme principal attend 2 arguments : l'ordre du graphe Butterfly des tâches, et le nombre de machines.

Exemple d'exécution : java Test_DSS_Butterfly 4 5

2.4.2 Test_DSS_Grille

Cette classe génère des graphes de tâches de type Grille. Le programme principal attend 3 arguments : le nombre de lignes et le nombre de colonnes du graphe de tâches, ainsi que le nombre de machines.

Exemple d'exécution : java Test_DSS_Grille 3 5 4

2.4.3 Test_DSS_SerieParallele

Cette classe génère des graphes de tâches de type Série-Parallèle. Le programme principal attend 3 arguments : le nombre de tâches, le nombre de séries du graphe de tâches, et le nombre de machines.

Exemple d'exécution : java Test_DSS_SerieParallele 16 3 5

De plus, afin de pouvoir spécifier le type de graphe de tâches, un champs statique "type graphe" a été rajouté à la classe Test_DSS_avec_retard, qui sera initialisé à "Aléatoire", "Butterfly", "Grille", ou "Série-Parallèle". Afin de pouvoir écrire et charger le type de graphe avec le fichier de trace, les méthodes "genereTrace" et "lireTrace" ont été modifiées.

3. Interface graphique

3.1 Description

L'interface graphique permet à la fois, de créer des problèmes d'ordonnancement et de lancer l'algorithme d'ordonnancement, qui résous ensuite ces problèmes. Elle offre deux choix : la saisie manuelle des paramètres ou le chargement de données à partir d'un fichier de trace.

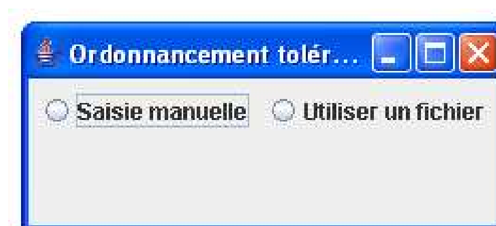
Après la résolution du problème d'ordonnancement, un diagramme de Gantt apparaît afin de donner un aperçu visuel de l'ordonnancement.

3.2 Navigation

Pour lancer l'interface graphique de l'application il faut saisir en ligne de commande:

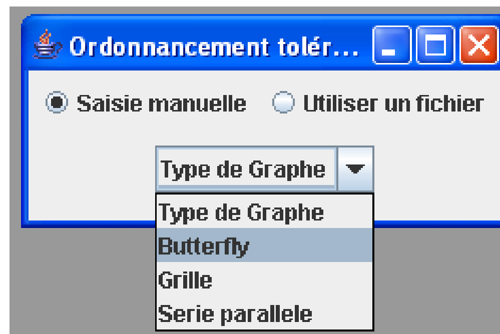
```
java Interface
```

Suite à cette commande la fenêtre suivante apparaitra à l'écran :



3.3. Saisie Manuelle

Une fois le choix "Saisie manuelle" effectué l'application proposera une nouvelle option "Type de Graphe" pour choisir le type de graphe de tâches sur lequel on veut dérouler l'algorithme d'ordonnancement. On peut choisir entre trois types de graphe de tache : Butterfly, Grille, Série parallèle.



Après qu'on ait choisi le type de graphe de tâches l'interface graphique permet de saisir les paramètres spécifiques à chaque type de graphe.

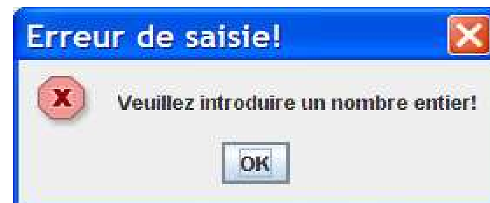
Pour les graphes Butterfly il faut saisir l'ordre du graphe et le nombre de machines.

Pour les graphes Grille il faut saisir le nombre de lignes, le nombre de colonnes et le nombre de machines.

Pour les graphes Série parallèle il faut saisir : le nombre de taches, le nombre de série et le nombre de machines.

Pour pouvoir lancer l'algorithme, tous les champs doivent être remplis. Une fois que tous les champs ont été renseignés, le bouton "Lancer" devient actif et les algorithmes de création du problème et de sa résolution peuvent être lancés.

Pour éviter les erreurs de saisie et le lancement de l'application avec des paramètres erronés une vérification de champs de saisie est effectue et le lancement de l'application n'est pas possible.

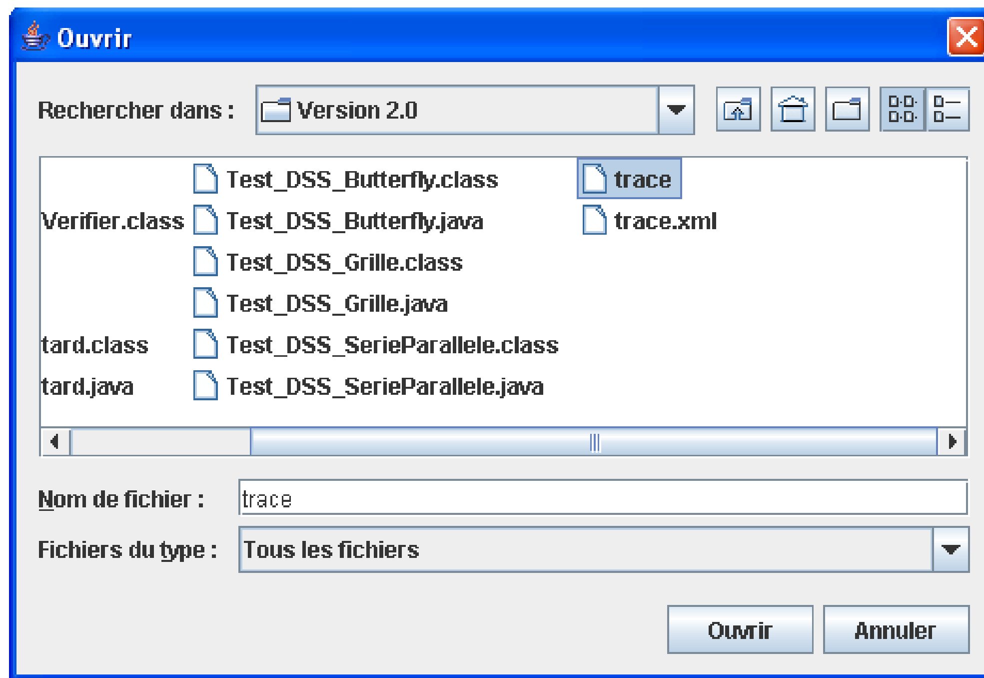


Les paramètres de la solution seront affichés en mode console, et un ficher "trace" décrivant le problème d'ordonnancement, ainsi qu'un fichier "trace.xml" décrivant l'ordonnancement solution, seront générés dans le répertoire de l'application. De plus, le diagramme de Gantt de l'ordonnancement solution s'affichera.

3.4 Utiliser un fichier

Une fois le choix "Utiliser un ficher" effectué l'application proposera un bouton "Ouvrir un fichier".

Ce bouton permet de choisir le fichier de trace à utiliser pour charger un problème.

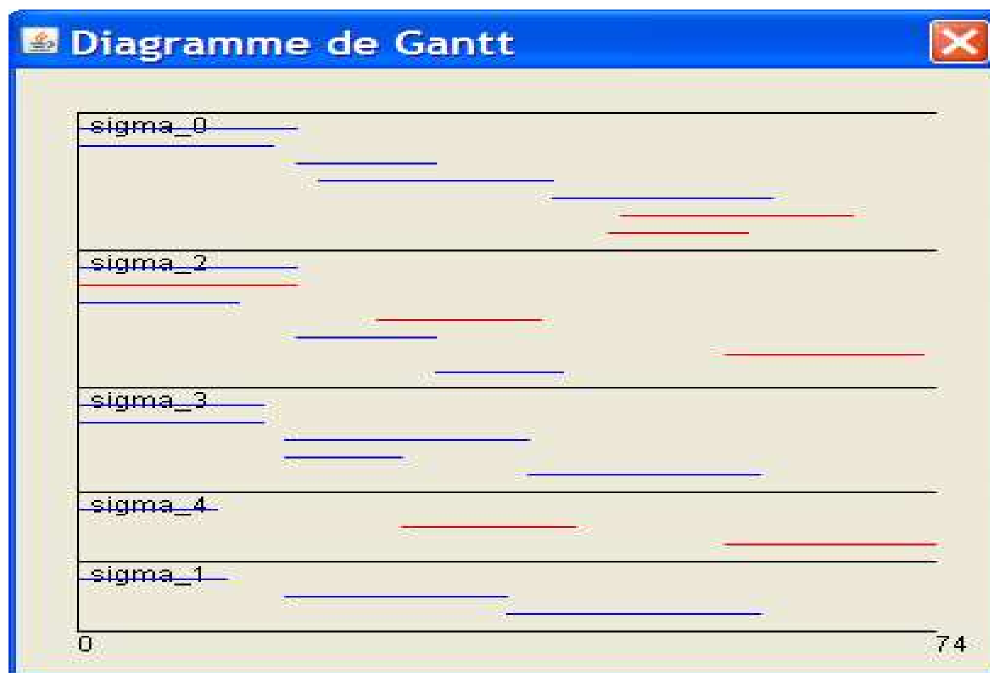


Une fois le fichier choisi, le problème d'ordonnancement est créée, résolu et un diagramme de Gantt sera affiché. La solution est également affichée dans la console.

3.5 Le Diagramme de Gantt

3.5.1 Description

L'affichage du diagramme de Gantt permet d'avoir un aperçu visuel global du chargement des ressources.



Les lignes bleues représentent les tâches initiales et les lignes rouges représentent les backups des tâches initiales. Grâce à se diagramme, on peut savoir quand et sur quelle ressource une tâche s'exécute, et on peut également évaluer le nombre de tâches qui s'exécutent sur chaque ressource à un instant donné.

3.5.2 Mise en œuvre

L'application génère lors de son lancement un fichier de trace "trace.xml" en format XML, sous la forme suivante :

```
<schedule name="trace.xml" makespan="256">
    <resources>
        <resource id="0" name="sigma_0" />
        <-- etc … -->
    </resources>
    <tasks>
        <task id="0" name="tâche_0" backup="0" start="0" duration="15"
                 resource="0" suivants="64,96" />
        <-- etc … -->
    </tasks>
</schedule>
```

La balise <schedule/> est globale et contient les informations sur les ressources et sur les tâches. Les attributs de cette balise sont "name": le nom du fichier XML et "makespan" : le temps total d'exécution.

La balise <resources/> contient la liste des ressources sous forme de balise <resource/>. Les attributs de la balise <resource/> sont l'identifiant de la ressource et son nom.

La balise <tasks/> contient la liste des ressources sous forme de balise <task/>. Les attributs de la balise <task/> sont l'identifiant de la tâche, son nom, le niveau de backup, le début de la tâche, sa durée, l'identifiant de la ressource où cette tâche est effectue et les identifiants des tâches qui la suivent (la tâche peut ou pas avoir des tâches qui la suivent).

Ce fichier sert ensuite à l'affichage du diagramme de Gantt. Le fichier est exploité à l'aide de la librairie open source jdom – www.jdom.org. Cette librairie offre une solution java pour la manipulation facile des fichiers XML, et a permis de créer la

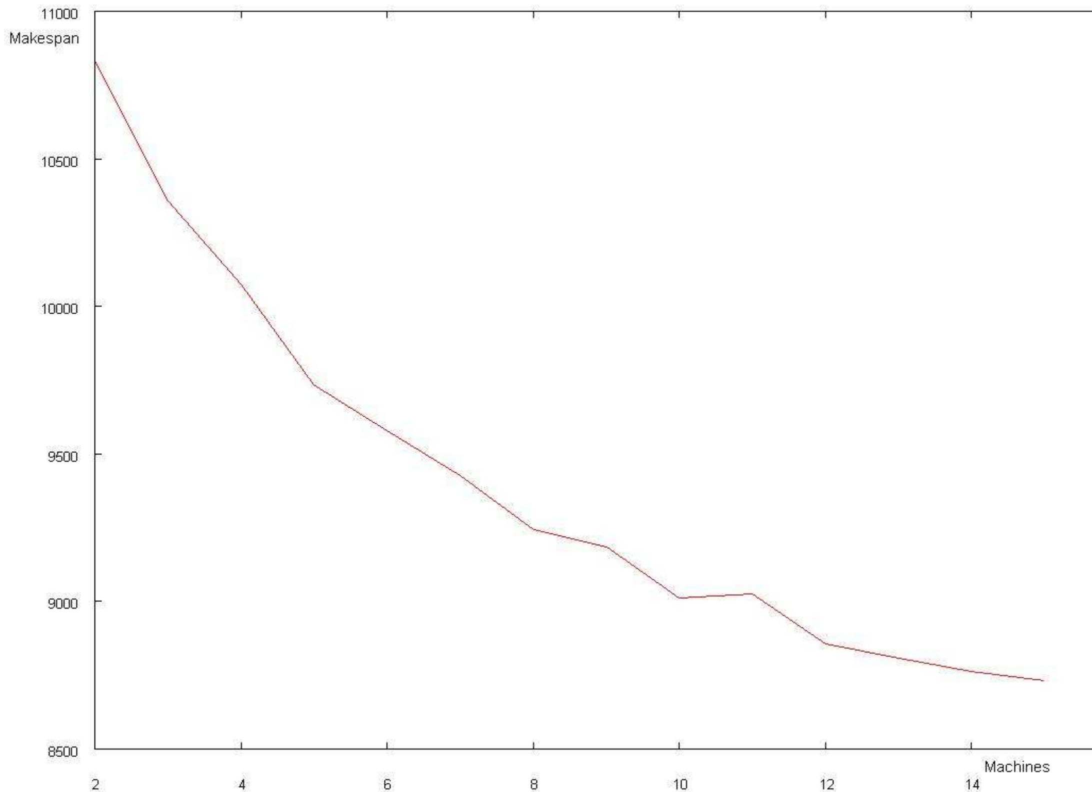
classe "Schedule", qui est chargé de récupérer les informations de l'ordonnancement contenues dans le fichier "trace.xml".

La classe "DiagrammeGantt" crée une fenêtre contenant l'affichage du diagramme de Gantt par rapport au chargement des ressources. Cette classe récupère les informations du fichier "trace.xml" sous la forme d'un objet Schedule, et affiche ces informations.

Les tâches sont affichées par rapport à la ressource où sont exécutées. Le tâches initiales sont dessines en bleu et tâches de type backup sont dessines en rouge. Le graphique ne contient ni les dates de début ni les dates de fin des tâches car le but est d'avoir une vision globale sur le planning, et pour un nombre assez important de tâches, il deviendrait vite illisible.

4. Statistiques

4.1 Makespan en fonction du nombre de machines



Le premier modèle de distribution que l'on a étudié consistait à étudier le makespan d'un ordonnancement avec un nombre de tâches fixe, en augmentant le nombre de machines. Cella modélise une distribution des tâches sur un grand nombre de machines, afin d'améliorer les performances.

Cette courbe met en évidence que la distribution des tâches effectuées par l'algorithme n'est pas adapté à un grand nombre de machines ; en effet, à partir de 8-10 machines, on peut observer que le makespan ne diminue quasiment plus. Cella est due à une hypothèse de départ, qui est que les machines peuvent exécuter un nombre infini de tâches simultanément, ce qui limite l'effet de l'augmentation du nombre de machines.

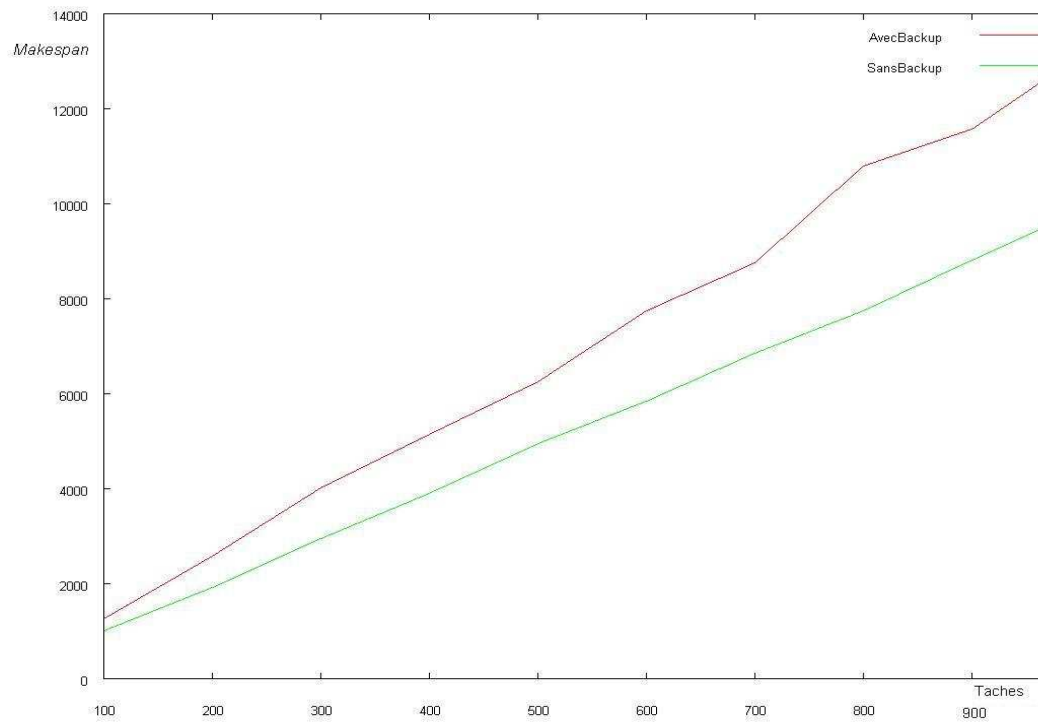
4.2. Makespan en fonction du nombre de tâches

Nous nous sommes donc plutôt penchés sur un deuxième modèle, avec un nombre limité de machines, mais aux capacités énormes, et nous avons étudié l'évolution du makespan quand on augmente le nombre de tâches, pour chaque type de graphe, Aléatoire, Butterfly, Grille, et Série-Parallèle.

Nos statistiques étudie l'évolution de makespan par rapport au nombre de taches selon le type de graphes. Mais pour cella il nous faut un nombre de machines fixe, nous nous sommes alors demandé quel était le nombre réaliste de machines qui permettrait tout de même d'avoir un makespan honorable sur un nombre de taches conséquent. Pour cella, nous avons choisis de faire nos tests sur 5 machines.

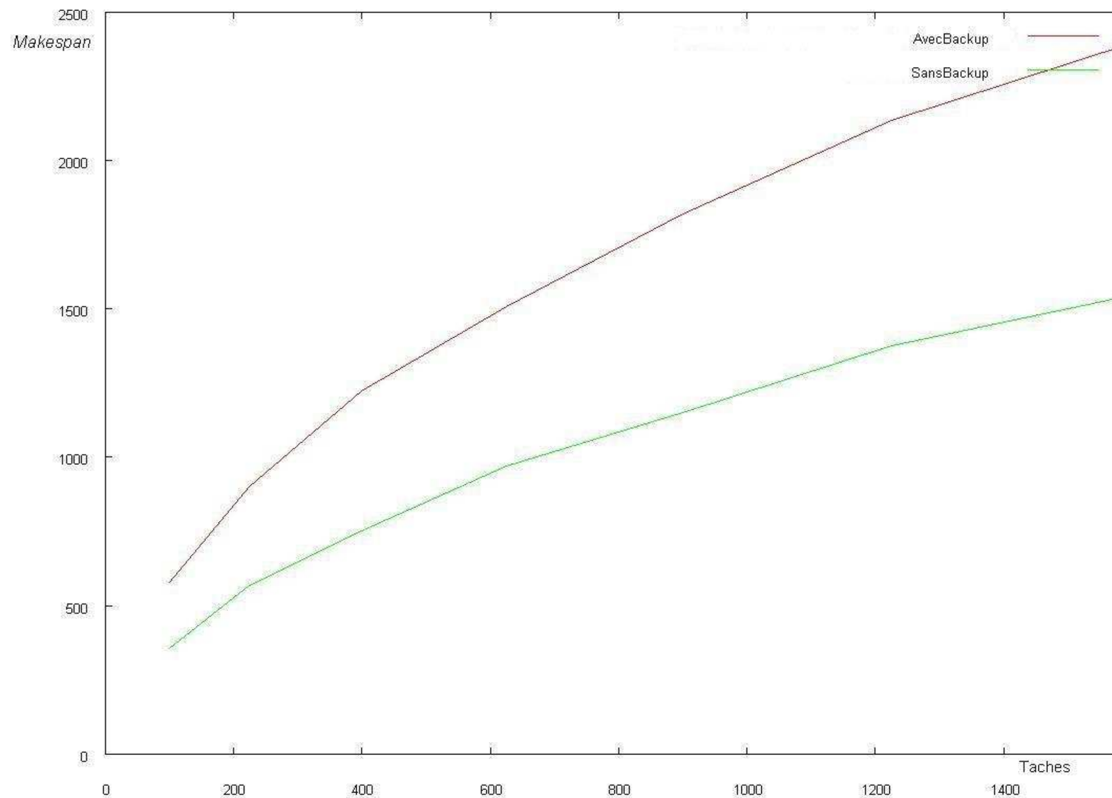
Nous avons ensuite comparé le makespan entre les algorithmes avec backup et sans backup, en augmentant le nombre de tâches.

4.2.1 Graphe Aléatoire



On observe que le makespan en fonction du nombre de tâches pour un graphe de tâches aléatoire suit une évolution linéaire.

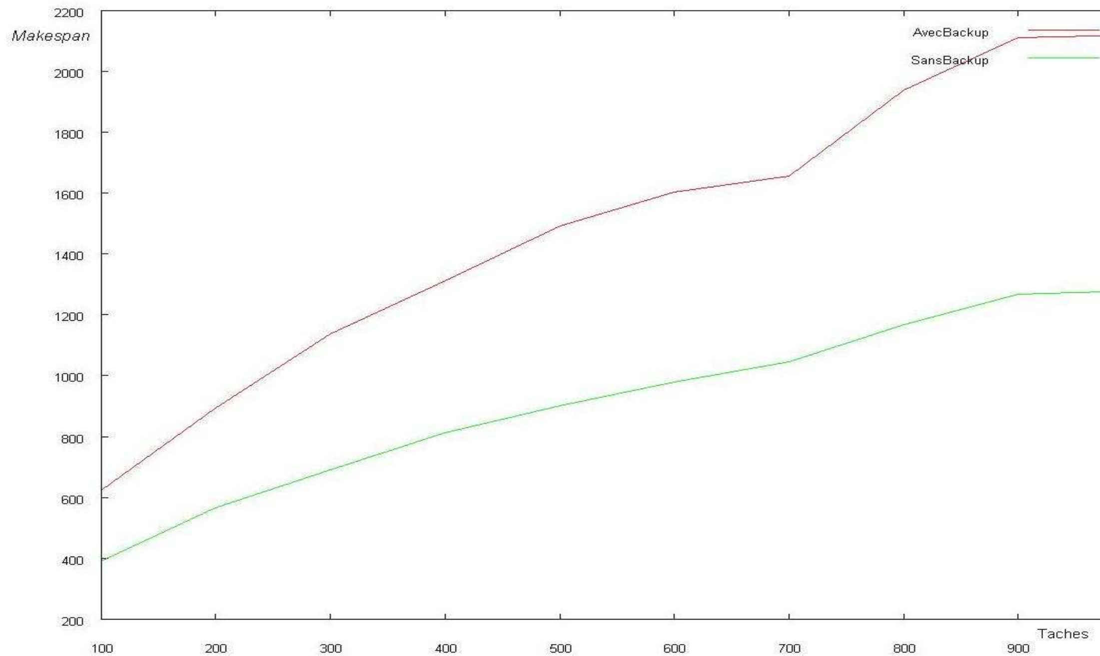
4.2.2 Graphe Grille



Avec un graphe grille, le makespan en fonction du nombre de tâches suit une évolution logarithmique : par rapport à un graphe de tâche aléatoire, le makespan augmente de moins en moins quand on augmente le nombre de tâches.

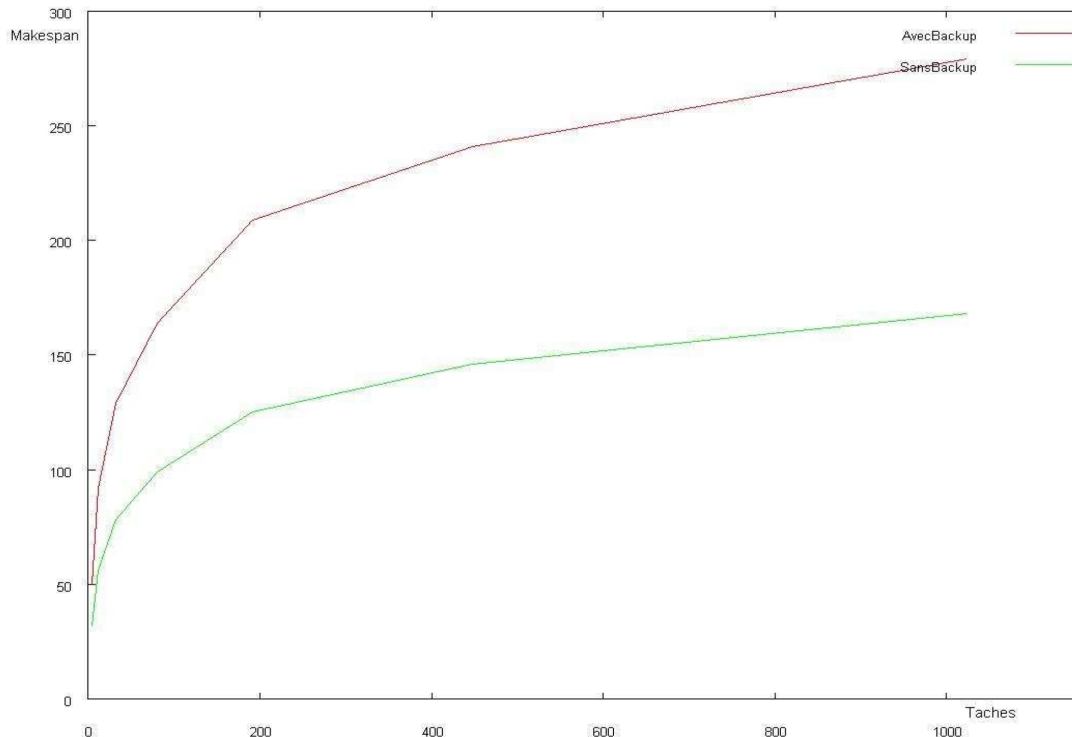
De plus, les performances sont bien supérieures aux résultats obtenues avec des graphes aléatoires : pour 1000 tâches, le graphe aléatoire à un makespan de 13000, pour 2000 avec un graphe grille.

4.2.3 Graphe Série-Parallèle



Les résultats obtenus avec ce graphe nous montrent des similarités avec la grille. Les makespan obtenus sont du même ordre, bien que la grille ait un léger avantage.

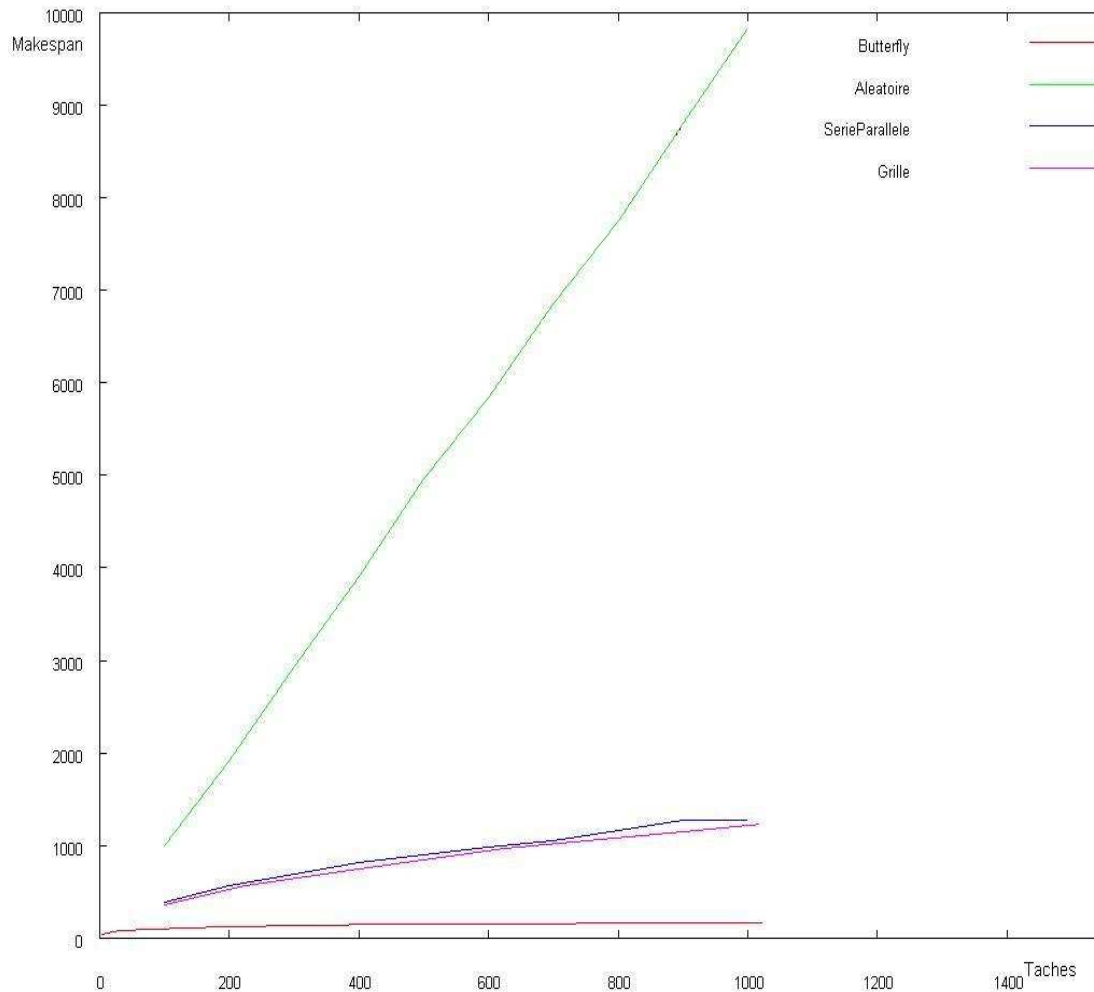
4.2.4 Graphe Butterfly



Le makespan obtenus pour peu de taches est très faible, et sa croissance est rapidement stabilisée. La différence entre les résultats obtenus avec les autres types de graphe et ce graphe sont très net. Pour 1000 taches, le makespan est ici à peine supérieur à 150 (le makespan pour 1000 taches dépasse les 10000 pour un graphe aléatoire).

D'une manière générale, on a pu observer que le rapport entre les makespan obtenus avec backup et sans backup va de 1,5 à 2 au maximum. L'utilisation du backup ne requiert donc pas de calcul excessif.

4.2.5 Comparaison des courbes



On voit nettement que les graphes aléatoires ne sont pas adaptés a l'algorithme d'ordonnancement, le makespan obtenus pour 100 taches est déjà de 1000 et il augmente de façon linéaire. Les résultats obtenus avec un graphe grille et série parallèle sont relativement similaires. Les meilleures performances sont, de loin, obtenues avec le graphe butterfly.

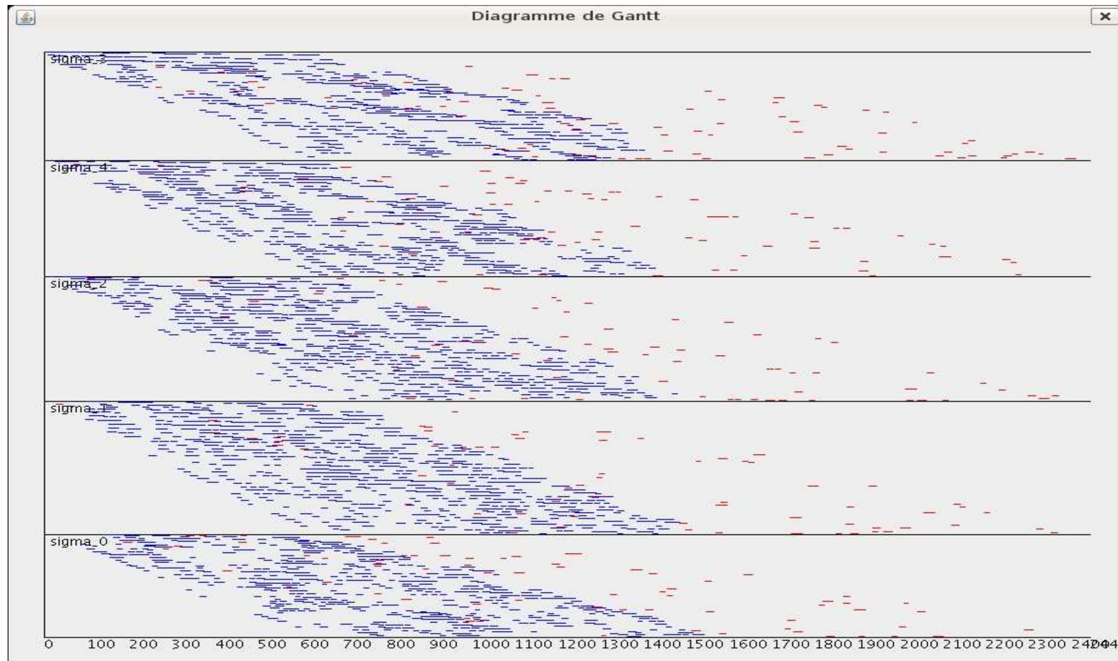
4.3 Interprétation

Afin d'interpréter les données statistiques, nous avons utilisé le diagramme de Gantt afin de mieux évaluer la répartition des tâches de l'ordonnancement.

4.3.1 Graphe Aléatoire

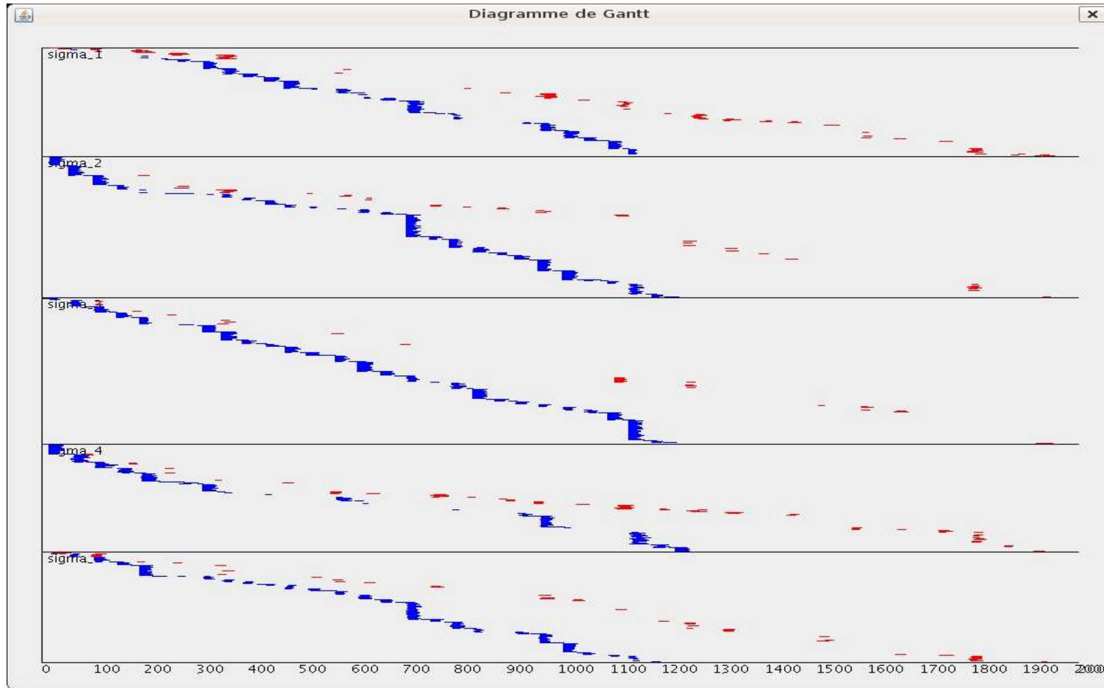
Les mauvaises performances peuvent s'expliquer ainsi : la génération aléatoire du graphe provoque trop de dépendances, ce qui fait que peu de tâches peuvent s'exécuter simultanément.

4.3.2 Graphe Grille



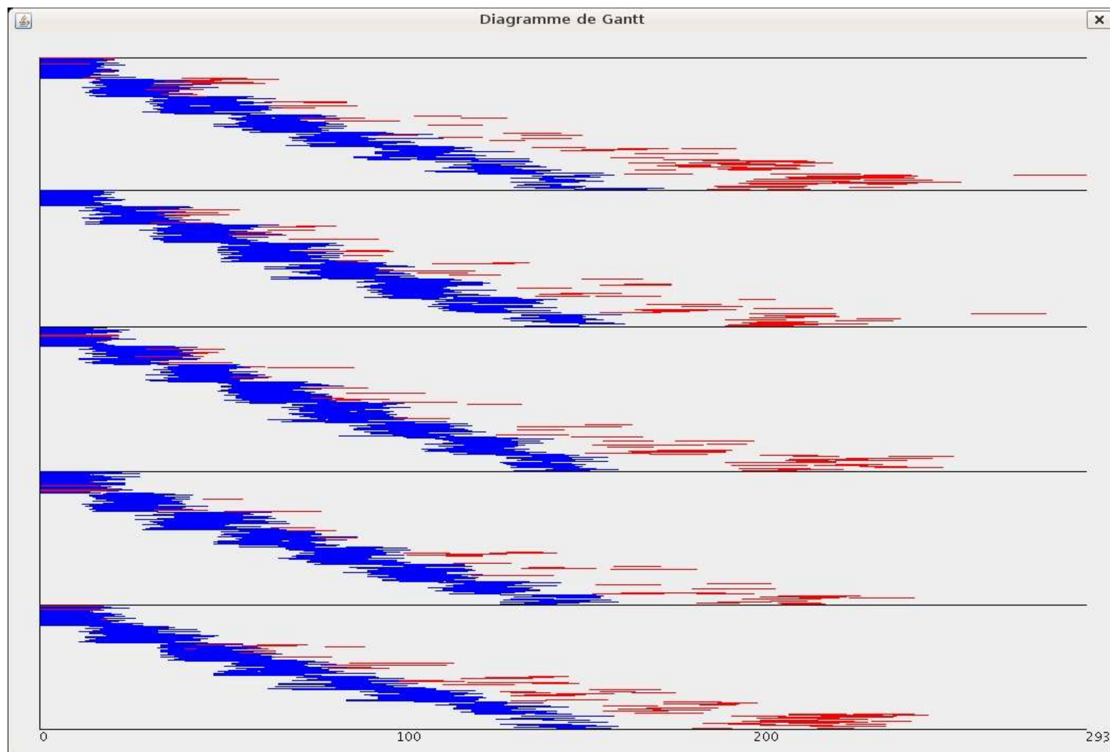
Dû à la structure du graphe, il y a une mauvaise répartition des tâches tout au long de l'ordonnancement : peu de tâches s'exécutent au début et à la fin, alors que les machines sont très chargées en milieu d'algorithme.

4.3.3 Graphe Série-Parallèle



La structure du graphe Série-Parallèle permet une meilleure parallélisation des tâches sur les différentes machines. Cependant, les performances sont réduites à cause des trop nombreuses dépendances entre les machines, qui ne permettent pas d'enchaîner l'exécution d'une couche tant que la précédente n'est pas terminée: on observe alors des trous, où certaines machines n'exécutent rien du tout.

4.3.4 Graphe Butterfly



On peut observer dans le diagramme de Gantt la structure en couche du graphe Butterfly. Contrairement au graphe Grille, dont les couches sont de tailles variables, petites au début et à la fin, grandes en plein milieu, les couches du graphe Butterfly possèdent toutes le même nombre de tâches, ce qui permet une meilleure répartition de l'ordonnancement. De plus, par rapport au graphe Série-Parallèle, le nombre de dépendances est moindre, ce qui permet d'enchaîner plus rapidement l'exécution des différentes couches.

Conclusion

Le projet est bien engagé : un algorithme d'ordonnancement aux pannes de régions a été réalisé, ainsi que des algorithmes de génération de graphes de tâches de type Butterfly, Grille, et Série-Parallèle. Une interface graphique fonctionnelle, qui permet de créer des problèmes d'ordonnancement, et qui affiche le diagramme de Gantt des ordonnancements solutions a également été réalisée, et des données statistiques sur l'algorithme fourni au départ ont été produites.

Reste à présent à tester plus en détail la validité de l'algorithme d'ordonnancement résistant aux pannes de régions, à l'incorporer dans l'interface graphique, et à réaliser de plus amples études statistiques.