UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Swann Perarnau

Thèse dirigée par **Denis Trystram**
et codirigée par **Guillaume Huard**

préparée au sein **Laboratoire d'Informatique de Grenoble**
et de **l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Environnements pour l'analyse expérimentale d'applications de calcul haute performance

Thèse soutenue publiquement le **1er Décembre 2011**,
devant le jury composé de :

M. Noël De Palma
Professeur UJF, Président
M. Raymond Namyst
Professeur Université de Bordeaux 1, Rapporteur
M. Jean-Louis Pazat
Professeur INSA Rennes, Rapporteur
M. Michel Syska
Maître de Conférences Université de Nice, Examinateur
M. Denis Trystram
Professeur Grenoble INP, Directeur de thèse
M. Guillaume Huard
Maître de Conférences UJF, Co-Directeur de thèse

Remerciements

Je voudrais, pour commencer, adresser mes remerciements aux membres du jury. Je remercie les deux rapporteurs Raymond Namyst et Jean-Louis Pazat, pour avoir accepté d'évaluer mon travail en profondeur. Merci aussi à Michel Syska et au président du jury Noël De Palma d'avoir assisté à ma soutenance. J'ai particulièrement apprécié les questions du jury lors de cette dernière et vous en remercie tous une fois encore.

Cette thèse n'aurait jamais pu prendre la forme qu'elle a aujourd'hui sans mes deux directeurs, Denis Trystram et Guillaume Huard. Denis, lorsque tu m'as expliqué pour la première fois cette histoire de génération aléatoire de graphes, jamais je n'aurais imaginé découvrir un problème aussi intéressant à tous les niveaux. J'ai beaucoup appris à tes côtés et je t'en remercie. Guillaume, je n'ai jamais regretté d'être venu te voir pour faire de la recherche les mains dans le cambouis. En quatre ans, tu m'as vraiment appris ce métier, les bons côtés comme les moins drôles. Nos nombreuses discussions, qu'elles concernent le travail ou des trucs de geeks, resteront de super moments.

J'ai rencontré un nombre certain de personnes formidables dans le bâtiment de l'ancien laboratoire ID. Je pense bien sûr à tous ceux avec qui j'ai eu la chance de travailler, mais aussi à tous ceux qui ont croisé ma route pour partager un troll de cafétéria ou une partie de coinche. Alors merci à Brice, Erik, Marin, Marc, Frédéric, Vincent, Jean-Marc, Olivier, Joseph, Yves, Grégory, Vania, Pierre-François, Nicolas et tous les autres. Je garde une place à part pour ceux qui avec qui j'ai partagé un bureau : Xavier, Charbel et Marie, parce que me supporter plus de 8 heures d'affilée c'est toujours un exploit. Et enfin merci Jean-Noël, tu es un ami précieux. Les nombreux aller-retours entre nos deux bureaux pour parler de tout, et surtout de rien, sont de très bons souvenirs.

Un *immense* merci, pour finir, à tous mes amis et à ma famille pour votre soutien. Même si je ne le dis pas souvent, tout ça c'est aussi grâce à vous.

Sommaire

Remerciements iii

Sommaire v

1 Introduction 1

I Contrôler l'utilisation des ressources matérielles 5

2 Architecture d'une machine à mémoire partagée 7

3 Fonctionnement d'un système d'exploitation moderne 25

4 Émuler l'indisponibilité du processeur 37

5 Donner à l'utilisateur le contrôle du cache 59

II Génération de graphes de tâches pour la simulation d'ordonnanceurs 85

6 Notions d'ordonnancement 87

7 Génération de graphes pour la simulation d'ordonnanceurs 95

8 Influences des graphes générés sur les ordonnanceurs 113

Conclusion et Perspectives 123

Bibliographie 127

Table des figures 135

Table des matières 139

Résumés 143

Introduction 1

Sommaire

1.1 Axes d'étude **2**

1.1.1 Contrôler l'utilisation d'une ressource matérielle 2

1.1.2 Maitriser les entrées lors de la simulation d'un ordonnanceur 3

1.2 Guide de lecture et contributions **3**

Le TOP500 [MSSD10] recense deux fois par an les machines les plus performantes au monde en se basant sur une application de référence : LINPACK [DLP03]. En juin 2011, le *K computer*, supercalculateur Japonais construit par Fujitsu fût déclaré l'ordinateur le plus puissant de la planète, avec une performance atteignant 8.162 petaflops. La structure de ce système est typique des machines pour le calcul haute performance (HPC). Plus de 500 000 nœuds sont interconnectés par un réseau complexe (tore à six dimensions) et chacun d'entre eux est composé d'un processeur SPARC64 VIIIfx et de 16 GiB de mémoire physique. Ce processeur comprend 8 cœurs et deux niveaux de caches, le dernier niveau (L2) étant partagé par l'ensemble des cœurs.

Le développement d'applications pour ce type de machines soulève essentiellement deux problèmes : l'utilisation efficace d'un nœud de calcul, et la distribution du travail entre les nœuds. Au niveau du nœud, il s'agit ainsi d'analyser l'architecture à disponibilité et d'optimiser l'application pour en tirer le meilleur. Par exemple, la hiérarchie mémoire peut être mise à contribution pour faciliter la communication entre les processus légers s'exécutant sur le même processeur. Entre les nœuds, des algorithmes de répartition du travail s'assurent que chacun d'entre eux reçoive suffisamment de données à traiter, tout en optimisant les transferts pour éviter de surcharger le réseau.

Mais les machines parallèles peuvent aussi prendre d'autres formes, comme de petits systèmes composés de plusieurs processeurs et partagés par plusieurs utilisateurs simultanément. Il convient alors d'adapter le fonctionnement de l'application à un environnement dynamique, où la quantité disponible de ressources varie au cours du temps.

Pour répondre à ces nombreuses problématiques, de nouveaux environnements se sont développés tels que OpenMP [DM98], Intel TBB [Rei07] ou Cilk [Blu95] et des algorithmes de distribution des ressources complexes tels que HEFT [THW02] ou HBMCT [RH04]. Pour comprendre ces solutions, les améliorer ou concevoir de nouveaux algorithmes plus efficaces, il est nécessaire d'expérimenter sur celles-ci. Cette expérimentation permet de confronter une application à un environnement ou à des paramètres particuliers, parfois différents de ceux pour lesquels elle fut conçue, pour analyser ainsi sa performance et ses points faibles.

Il est néanmoins indispensable, du point de vue de la démarche scientifique, que ces

expérimentations soient réalisées à l'aide d'outils précis, validés et permettant de mettre en place des conditions expérimentales reproductibles. Nous regroupons sous ce terme de conditions expérimentales l'ensemble des paramètres pertinents d'une expérience : l'état de la machine utilisée (dédiée ? occupation des processeurs ?), les caractéristiques des données en entrées (structurées, générées aléatoirement ?), le fonctionnement des logiciels présents ou encore les caractéristiques matérielles. Tous ces paramètres doivent être maitrisés lors de l'étude d'une application.

Cette maitrise devient particulièrement critique dans le contexte du calcul haute performance ou dans les environnements distribués et dynamiques. Les applications étudiées sont en effet conçues pour tirer le maximum d'un système, ce qui les rend sensibles aux modifications, même légères de l'environnement. Le bruit système, par exemple, terme recouvrant l'ensemble des évènements activant le système d'exploitation durant l'exécution d'une application est un phénomène connu pour modifier le comportement (et particulièrement la performance) d'applications réalisant beaucoup de synchronisations [TEFK05]. Dès lors, il devient difficile de mettre en place des conditions expérimentales particulières sur un système de façon reproductible et sans perturber outre mesure le comportement des applications étudiées.

C'est à cette dernière problématique que s'intéresse cette thèse : la conception d'environnements pour l'analyse expérimentale d'application, leur validation et leur utilisation pour la compréhension et l'optimisation d'applications.

1.1 Axes d'étude

Les deux problématiques que nous avons dégagées plus haut ont guidé cette thèse : l'optimisation de l'utilisation d'un nœud de calcul, que nous assimilons à une machine à mémoire partagée et l'étude d'algorithmes de répartition du travail, autrement dit d'ordonnanceurs pour machines parallèles. Sur ces deux axes, nous nous intéressons à la conception d'environnements pour évaluer expérimentalement des solutions existantes.

1.1.1 Contrôler l'utilisation d'une ressource matérielle

La performance d'une application s'exécutant sur un nœud de calcul dépend fortement de sa capacité à exploiter au mieux l'ensemble du matériel présent sur le système. Un certain nombre de modèles font ainsi un parallèle direct entre la disponibilité d'une ressource et la performance d'une application. Il suffit alors de contrôler, pour une ressource donnée, la quantité utilisable pour obtenir des informations capitales sur le comportement d'une application.

Malheureusement, peu de techniques permettent aujourd'hui à un expérimentateur de limiter l'utilisation d'une ressource sur un système réel. La pile logicielle, et principalement le système d'exploitation, en contrôlent en effet l'accès et cela avec des objectifs bien différents. Ainsi, si plusieurs applications sont présentes sur la machine le système d'exploitation se charge de répartir équitablement les ressources entre elles. De même, les abstractions fournies par le système pour implémenter des applications correspondent peu aux besoins d'un expérimentateur : le principe de mémoire virtuelle par exemple abstrait la hiérarchie mémoire d'une machine comme une mémoire infinie

avec des temps d'accès uniformes. De plus, certains composants matériels comme les caches sont conçus pour être transparents du point de vue de l'application.

Pour fournir à l'expérimentateur un environnement de contrôle des ressources matérielles, nous nous sommes donc intéressés à une modification du système d'exploitation. Le contrôle de deux ressources en particulier a été étudié : le temps processeur obtenu par une application et le contrôle de la hiérarchie mémoire, à savoir la quantité de cache disponible pour une application. Nous montrerons soit par coopération avec des mécanismes avancés du système soit par remplacement de certaines fonctionnalités, qu'il est possible de construire un environnement de contrôle du matériel qui soit précis et qui fournisse des performances reproductibles.

1.1.2 Maitriser les entrées lors de la simulation d'un ordonnanceur

L'ordonnancement est un domaine riche et toujours d'actualité dans le calcul haute performance. Que ce soit pour distribuer du travail à différents nœuds de calcul ou pour répartir les ressources à différents utilisateurs, des problèmes et de nouveaux algorithmes pour les résoudre voient régulièrement le jour.

Dans ce contexte, la validation expérimentale de ces algorithmes par simulation prend une part de plus en plus importante dans la recherche. Cette simulation repose bien sûr sur un environnement de simulation comme Simgrid [CLQ08], mais comporte aussi une partie génération des données en entrées. La qualité de cette génération est capitale dans l'analyse des performances des algorithmes. En effet, certaines caractéristiques de ces données générées peuvent influer fortement sur la qualité des solutions calculées par des ordonnanceurs. Malheureusement, peu de travaux proposent une analyse en profondeur de cette influence et des méthodes de génération *ad-hoc* sont le plus souvent utilisées, sans effort de validation ou d'analyse des résultats.

Nous nous sommes donc intéressés à l'établissement d'un environnement de génération aléatoire des données en entrées d'un ordonnanceur (graphe de tâche) et à une analyse des résultats générés et de leur influence sur la performance d'algorithmes d'ordonnancement. Nous montrerons ainsi comment la modification de certains paramètres de la génération entraîne des phénomènes d'inversion dans la comparaison d'algorithmes : alors que sur la méthode de génération A un algorithme 1 était meilleur qu'un algorithme 2, l'utilisation de la méthode B montre ce dernier (2) comme meilleur que 1.

1.2 Guide de lecture et contributions

Ce manuscrit est découpé en deux parties, correspondant chacune à un des axes d'études précédemment cité.

Le chapitre 2 présente le modèle de machine à mémoire partagée utilisé pour décrire un nœud de calcul et en détaille les principales caractéristiques architecturales. Cela comprend l'organisation des unités d'exécution, la hiérarchie mémoire, l'influence de cette hiérarchie sur la performance des applications et les moyens existants pour détecter les caractéristiques de la machine.

Le chapitre 3 détaille le fonctionnement d'un système d'exploitation moderne comme on en trouve dans le HPC. Le fonctionnement de l'ordonnanceur système, responsable de la répartition du processeur aux applications, et du gestionnaire de mémoire virtuelle y sont détaillés.

Le chapitre 4 présente notre première contribution : un environnement de reproduction d'une charge processeur. Cet environnement permet d'émuler sur une machine dédiée un déséquilibre dans le temps processeur disponible pour une application, et cela de façon reproductible. Ce travail a été publié en version courte à PPoPP en 2010 [PH10a] et en version longue à IPDPS la même année [PH10b].

Le chapitre 5 présente notre deuxième contribution : un environnement permettant à une application de contrôler la quantité de cache qu'elle utilise, et cela pour chacune des structures de données qu'elle contient. Basé sur des techniques de partitionnement de cache, ce travail a été publié à ICS en 2011 [PTH11].

Le chapitre 6 rappelle quelques notions de base en ordonnancement et en théorie des graphes. Nous y détaillons notamment le fonctionnement des ordonnanceurs par liste et quelques caractéristiques des graphes de tâches pouvant influer sur la performance d'un tel algorithme.

Le chapitre 7 présente notre troisième contribution : un outil de génération aléatoire de graphes pour l'ordonnancement. Cet outil fournit un ensemble d'algorithmes de génération de référence, dont l'implémentation a été validée par comparaison avec les propriétés théoriques connues des modèles de génération utilisés. Le contenu de ce chapitre a été publié à Simutools en 2010 [CMP+10] et en français dans un article court à RENPAR l'année précédente [CMP+09].

Enfin, le chapitre 8 détaille comment la performance d'un algorithme d'ordonnancement est influencée par la méthode de génération utilisée. Ces résultats ont été publiés en version courte à ROADEF en 2011 [PTV11] et en version longue à RENPAR la même année [QP11].

I

Contrôler l'utilisation des ressources matérielles

Architecture d'une machine à mémoire partagée 2

Sommaire

2.1 Une architecture organisée hiérarchiquement 8
2.1.1 Multiprocesseurs et Multicœurs 9
2.1.2 Parallélisme au sein d'un cœur 9
2.1.3 Accès non uniformes à la mémoire 10
2.2 Architecture d'un cache 11
2.2.1 Principes de localité 11
2.2.2 Fonctionnement des caches 12
2.2.3 Gestion du cache 12
2.2.4 Adressage 13
2.2.5 Hiérarchie et partage des caches entre cœurs 14
2.3 Interactions entre caches et applications 15
2.3.1 Classification des défauts de cache 15
2.3.2 Préchargement automatique 16
2.3.3 Distance de réutilisation 16
2.3.4 *Working set* 17
2.4 Informations fournies par le matériel 17
2.4.1 Compteurs de performance matériels 17
2.4.2 Niveaux de trace et utilisations des compteurs 18
2.4.3 Identification des caractéristiques de la machine 19
2.4.4 Détection logicielle de caractéristiques matérielles 20

Il est capital, avant d'envisager le contrôle des ressources matérielles d'une machine à mémoire partagée, de comprendre en détail comment elles fonctionnent et interagissent avec les applications présentes sur le système. Nous rappelons donc dans ce chapitre comment s'organise une machine à mémoire partagée d'un point de vue architectural et les divers composants matériels influant sur la performance d'une application. Après une description de la structure hiérarchique d'un nœud de calcul, nous nous intéresserons notamment au fonctionnement des caches mémoires et des modèles de performance qui en découlent. Nous terminerons en détaillant une fonctionnalité devenue critique dans l'analyse de performance des applications : la détection des caractéristiques de la machine et le décompte des évènements matériels.

2.1 Une architecture organisée hiérarchiquement

Dans le cadre de cette thèse, nous considérons une machine à mémoire partagée comme un système composé de plusieurs unités de calcul ayant toutes accès à une même mémoire centrale. Cette architecture dispose aussi d'un ensemble de mémoires intermédiaires, appelées *caches*, gardant les données récemment accédées au plus proche des unités de calcul. Il existe bien évidemment des architectures qui ne correspondent pas exactement à ce modèle de machine à mémoire partagée. Dans le processeur Cell d'IBM par exemple, certaines des unités de calcul, les SPE, n'ont pas accès directement à la mémoire centrale. Certaines architectures, comme le processeur expérimental Tera [ACC+90] par exemple, ne disposent d'aucun cache (ce que l'on retrouve encore dans les systèmes embarqués). Cependant, notre modèle correspond aux architectures les plus communes, que ce soit dans les grands centres de calcul ou les machines destinées aux particuliers.

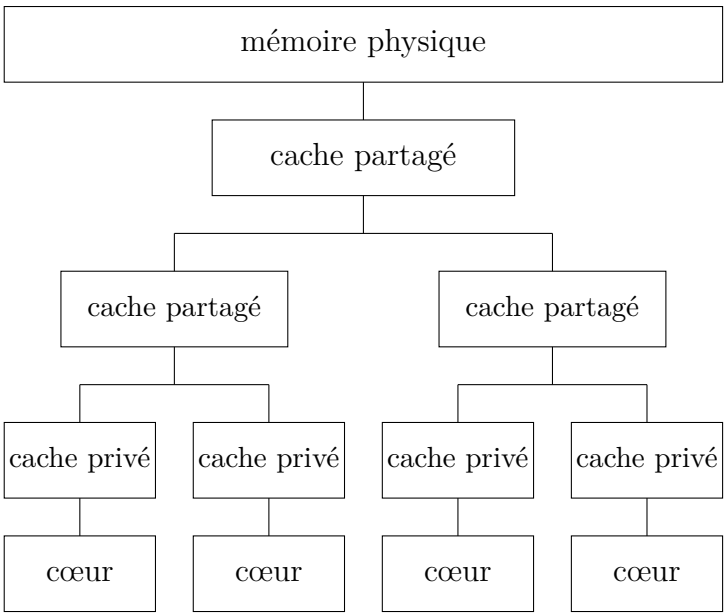


FIGURE 2.1 – Schéma général d'une machine à mémoire partagée.

Dans ce modèle, trois composants ont une influence sur les performances d'une application. Nous appellerons ici *cœur* l'unité minimale nécessaire à l'exécution d'un flot d'instructions. Ce composant comprend les mécanismes de décodage des instructions, les unités fonctionnelles permettant leur exécution ainsi que le support au système d'exploitation et au contrôle des périphériques (tables d'interruptions et les registres spéciaux associés). Deuxième élément, des bancs mémoire conservent instructions et données nécessaires aux programmes et sont connectés par un *bus* (ou canal) aux unités de calcul. Finalement, des caches se placent entre les cœurs et la mémoire pour accélérer certains accès. Certains de ces caches sont exclusifs (ou *privés*) à chaque coeur, tandis que d'autres sont partagés entre plusieurs unités de calcul.

Ces composants sont organisés différemment d'une architecture à une autre. Ainsi, certaines machines disposent de caches partagés par plusieurs cœurs, permettant des stratégies fines de partage du travail. Sur d'autres systèmes la mémoire centrale sera

découpée en différents bancs, chacun des cœurs étant plus proche (en temps d'accès) d'un banc que des autres. Nous détaillons dans la suite quelques-unes des organisations les plus courantes et leurs conséquences sur la performance des applications.

2.1.1 Multiprocesseurs et Multicœurs

Nous désignerons dans la suite par *puce* ou *processeur* l'unité marchande en vigueur chez les fondeurs. Cette notion de puce est centrale à un certain nombre de concepts, tirés de l'organisation physique des cœurs sur les machines qui nous intéressent. Pendant de nombreuses années, la plupart des processeurs grand public ne contenaient qu'un cœur. Pour réaliser une machine à mémoire partagée avec plusieurs unités de calcul il était donc nécessaire d'utiliser plusieurs puces. Un tel système était dit *multiprocesseur*.

Deux organisations de ces multiprocesseurs existent : dans une première version, proche de notre modèle, les canaux d'accès à la mémoire de toutes les puces se rejoignent en un concentrateur avant d'atteindre celle-ci. Un tel schéma, que l'on retrouve notamment dans les processeurs Intel Pentium et les premières générations de Core 2, permet un contrôle centralisé de tous les accès à la mémoire et donc simplifie grandement la conception des processeurs. Cette concentration est aussi la faiblesse du système, puisque toutes les unités de calcul du système se retrouvent en compétition sur la bande passante mémoire. Des phénomènes de contention peuvent alors apparaître : une application sera ralentie par le trop grand nombre d'accès à la mémoire effectué sur le même canal.

Dans une deuxième version, que l'on retrouve par exemple sur les processeurs AMD Opteron ou les derniers Intel Xeon, un banc mémoire est présent pour chaque puce, qui est donc dotée d'un contrôleur mémoire distinct. Un réseau d'interconnexion permet alors à tous les cœurs d'avoir accès à l'ensemble de la mémoire. Bien que plus complexe à mettre en œuvre, cette organisation à l'avantage d'éviter, dans certains cas, la compétition sur le bus mémoire entre les puces. Nous reviendrons un peu plus tard sur l'inconvénient majeur de cette organisation : l'apparition de plusieurs vitesses d'accès à la mémoire (en fonction du banc auquel accède un cœur).

2.1.2 Parallélisme au sein d'un cœur

Un programme en exécution peut être visualisé comme un flux d'instructions. Quelques registres, comme le pointeur d'instruction courante ou encore le pointeur de pile, dictent la façon dont ce flux d'instructions se construit et évolue au cours du temps. D'un point de vue architectural, tout programme en exécution est identifié par ces registres, dont il suffit de préserver l'état pour pouvoir arrêter ou reprendre l'exécution à n'importe quel moment. Nous appellerons dans la suite de ce chapitre *fil d'exécution* ou *thread* un programme en exécution sur un cœur.

Afin d'augmenter la quantité d'instructions exécutées sur une période de temps donnée, deux formes de parallélisme peuvent exister à l'intérieur d'un cœur : le parallélisme niveau instructions et le parallélisme niveau fils d'exécution.

Le parallélisme niveau instructions concerne la mise en place d'un certain nombre de techniques pour que plusieurs instructions d'un même thread puissent être décodées et exécutées en simultané. Cela comprend par exemple un pipeline qui sépare les

opérations de lecture, décodage, exécution et écriture du résultat d'une instruction en étages, de façon à pouvoir décoder une instruction pendant qu'une autre est exécutée.

Les gains qu'il est possible d'obtenir à l'aide de ces techniques restent faibles, les dépendances entre les instructions étant très nombreuses (les instructions de branchement posent notamment de nombreux problèmes).

Le parallélisme niveau fil d'exécution consiste alors à améliorer les techniques précédentes de façon à ce qu'un cœur exécute en simultané des instructions provenant de plusieurs threads (`Simultaneous MultiThreading` ou SMT). Ce mécanisme permet ainsi de mieux utiliser les pipelines et les unités fonctionnelles présents dans les cœurs. En revanche, certaines portions du matériel doivent être dupliquées, notamment les registres d'état du processeur pour pouvoir conserver le résultat de certaines instructions et l'état de chacun des threads.

Cette technologie est connue surtout dans sa version développée par Intel avec le Pentium 4 (`Hyper-Threading`) mais se retrouve dans des processeurs plus récents, comme le I7 ou encore le POWER7 d'IBM, spécialement développé pour le calcul haute performance. Son avantage majeur reste l'économie réalisée en matériel puisqu'il est ainsi possible d'exécuter plus de threads avec le même nombre de cœurs.

Le SMT reste néanmoins encore le sujet de beaucoup de débats concernant sa performance. En effet, les processus s'exécutant en simultané sont en compétition sur l'utilisation des unités fonctionnelles du cœur. Cette contention, à l'intérieur même de la mécanique d'exécution des instructions, peut avoir des conséquences importantes si les deux processus s'exécutant en simultané ne sont pas bien choisis. C'est au système d'exploitation que revient cette tâche, et il ne dispose que de peu d'informations sur l'affinité entre les différents fils d'exécution présents sur le système. Différentes études [CSL+04, CDL+02] ont ainsi montré que l'HyperThreading pouvait nuire à la performance de certaines applications multimédia notamment. Il n'est ainsi pas rare de voir le mécanisme désactivé sur des plates-formes pour le HPC, ce qui est par ailleurs le cas des machines utilisées dans cette thèse.

2.1.3 Accès non uniformes à la mémoire

Nous avons mentionné précédemment que sur certains systèmes multiprocesseurs, chaque puce possédait son propre banc mémoire et qu'un réseau d'interconnexion assurait l'accès à ces mémoires pour les autres cœurs. Dans une telle organisation, certains accès mémoire deviennent ainsi *locaux* et donc rapides, tandis que d'autres sont *distants* et plus lents. On parle alors d'une machine avec accès mémoire non uniformes ou de machine NUMA (*Non Uniform Memory Access*).

Il devient donc nécessaire de prendre en compte la topologie de la plate-forme lors du déploiement d'une application. De ce point de vue, la machine est alors découpée de la façon suivante. Premièrement, les différents bancs mémoire sont identifiés de manière unique (on parle de nœuds mémoire). Chaque cœur est alors placé dans une classe, selon le nœud mémoire le plus proche. Ainsi, les cœurs d'une même classe se comportent comme un système UMA, tandis que les accès mémoire vers un nœud appartenant à une autre classe sont plus lents.

Afin d'identifier la différence de vitesse d'accès entre un nœud NUMA local et un autre, la notion de *facteur NUMA* a été introduite :

Définition 1 *Appelons L_n^c le temps d'accès (latence), depuis le cœur c à un nœud NUMA n. Le facteur NUMA $F(c,n)$ est calculé de la façon suivante : $F(c,n) = \frac{L_n^c}{L_{loc}^c}$, où loc est le noeud NUMA local à c.*

Intuitivement, ce facteur NUMA donne le prix à payer lorsqu'un accès mémoire touche un nœud NUMA distant plutôt qu'un nœud local. En réalité, un tel facteur ne dépend pas que de la topologie de la machine puisque toute activité sur les autres cœurs peut influer sur la vitesse d'accès à un nœud. Les opérations de communication collectives en sont un bon exemple, le facteur NUMA ressenti dépendant de la quantité de données échangées et de la bande passante disponible entre les nœuds. Pour ces mêmes raisons, la caractérisation d'une machine NUMA se fait généralement à l'aide d'une table de facteurs pour chacun des couples (cœur,nœud) et sous différentes conditions en terme d'activité sur la machine. Nous considérerons cependant dans la suite de cette thèse que les machines observées ne relèvent pas de cette classe de systèmes.

2.2 Architecture d'un cache

Un grand nombre d'applications sont ralenties par l'incapacité du système à rapatrier suffisamment vite sur un cœur les données devant être traitées. Il est donc capital de trouver des mécanismes permettant à la fois de limiter la quantité de données qu'il est nécessaire de charger depuis la RAM et d'accélérer les accès restants.

De l'observation des comportements en mémoire des applications se sont dégagés deux principes, à l'origine de la plupart des solutions à ces deux problèmes.

2.2.1 Principes de localité

On distingue dans la plupart des applications deux phénomènes : la localité spatiale et la localité temporelle.

On désigne par localité spatiale le fait qu'une application accède généralement à des données proches les unes des autres en mémoire dans un laps de temps assez court. Ce phénomène apparaît naturellement du fait que les données de même nature sont généralement placées côte à côte en mémoire.

Un parcours de tableau représente un exemple très simple de ce genre de phénomène. Un algorithme pour rechercher le minimum dans un tableau d'entiers, par exemple, va naturellement parcourir la totalité du tableau de façon séquentielle. Celui-ci étant placé dans un espace contigu en mémoire, le deuxième élément, très proche du premier, sera accédé juste après ce dernier et ainsi de suite.

La localité temporelle quand à elle désigne la propension d'une application à réutiliser dans un laps de temps assez court une donnée. Ce phénomène apparaît lui aussi naturellement dans une application, une donnée n'étant rarement utilisée qu'une seule fois sur la totalité d'une exécution. Si l'on considère par exemple un produit naïf de deux matrices A et B, la même ligne de A sera utilisée pour calculer chacun des éléments d'une ligne de la matrice résultat.

Certaines optimisations deviennent évidentes si l'on prend en compte ces principes de localité : la nécessité de charger plus d'une donnée à la fois depuis la mémoire et la conservation des données récemment accédées.

2.2.2 Fonctionnement des caches

Les caches répondent à ces deux impératifs. Comme nous l'avons déjà signalé, il s'agit d'une petite mémoire placée entre un cœur et la RAM (généralement à l'intérieur des puces). Cette mémoire conserve les données récemment accédées pour profiter de la localité temporelle et les considère par petits blocs de mémoire contigüe pour user de la localité spatiale. Ces groupes sont par ailleurs désignés sous le terme *lignes de cache*.

Ces caches ont un comportement relativement simple du point de vue de l'application. Lorsqu'un cœur exécute des instructions il déclenche naturellement des accès aux données. Placé entre ce dernier et la mémoire, le cache intercepte chacun de ces accès. Si la donnée demandée est présente en cache, il y a *hit* et le cache fournit immédiatement cette dernière au cœur, sinon se produit ce que l'on nomme un *miss* (ou défaut de cache). Dans ce cas, un accès vers la mémoire est effectué pour rapatrier une ligne entière.

2.2.3 Gestion du cache

Il existe trois stratégies déterminant comment une nouvelle ligne mémoire est placée en cache. Ces trois classes de caches correspondent à un compromis entre coût d'implémentation et efficacité du cache. En effet, la mécanique d'identification de la ligne associée à l'adresse demandée par le cœur et la recherche de cette ligne dans le cache se déclenchent à chaque accès mémoire. Il est donc capital que cette phase soit la plus rapide possible, et surtout plus rapide qu'un accès direct à la mémoire. Lors de la conception d'un cache, la minimisation des circuits nécessaires à son fonctionnement est donc au cœur des préoccupations.

De ce point de vue, la classe la plus efficace est celle des caches à correspondance directe (*direct mapped* en anglais). Une ligne mémoire ne peut prendre alors qu'un seul emplacement en cache. Celui-ci est fixé et fonctionne selon une correspondance cyclique : la ligne mémoire numéro i sera placée dans la ligne de cache numéro $i \bmod L$ sur les L disponibles. Ces types de caches sont néanmoins peu efficaces sur certains types d'accès mémoire notamment ceux où le processeur effectue des sauts de taille constante dans la mémoire (une partie du cache n'est alors jamais utilisée).

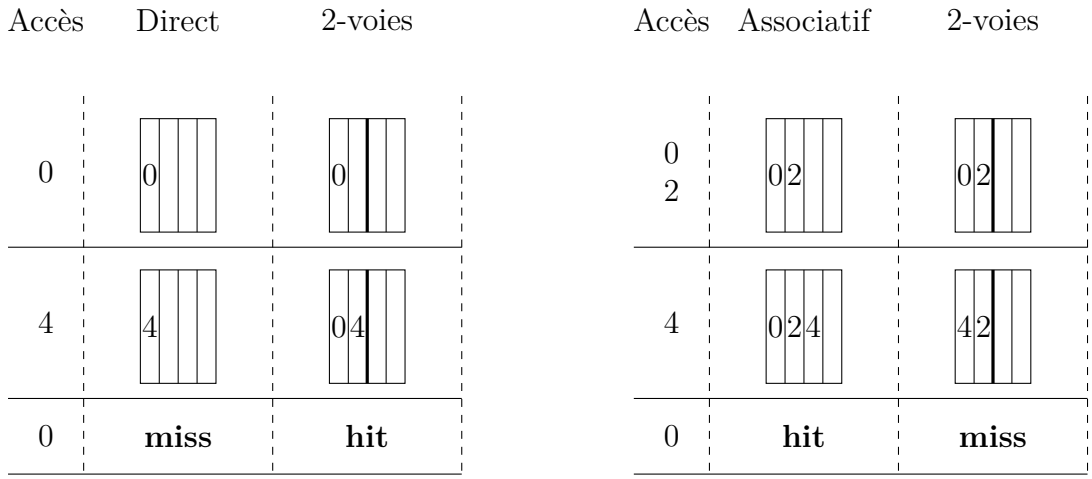
À l'opposé, un cache complètement associatif (*fully associative* en anglais) permet à une ligne mémoire de se placer dans n'importe quelle ligne de cache. Ces caches sont très efficaces puisqu'un nombre maximal de lignes mémoire peuvent y être sauvegardées. Malheureusement, ils coûtent excessivement cher en circuits, chaque adresse mémoire accédée devant être recherchée dans l'ensemble du cache (plus le cache est gros et plus il devient lent).

Enfin, la plupart des caches rencontrés de nos jours sont dit associatifs N-voies (N-*way associative* en anglais) car une ligne mémoire ne peut aller que dans N lignes de cache différentes. On appelle le groupe de lignes de cache que peut rejoindre une certaine ligne mémoire un *ensemble associatif*. Ce type de cache représente un compromis entre les deux premières solutions, étant moins efficace pour préserver les données récemment accédées, mais bien moins coûteux en circuit.

Dès lors qu'un cache est associatif, un algorithme doit être mis en place pour décider quelle ligne de cache sera remplacée lors d'un nouveau chargement. L'algorithme

privilégié pour un tel choix se nomme LRU (pour *Least Recently Used*) et consiste à éliminer la ligne qui a été utilisée le plus loin dans le temps. En pratique, une approximation de cet algorithme est utilisée pour des questions de coûts d'implémentation. Ces variantes sont généralement nommées pseudo-lru, la plus connue utilisant un arbre de décision binaire [KMCV10].

La figure 2.2 illustre le remplacement d'une ligne dans les trois types de caches que nous venons d'expliquer, sur des séquences de lectures de la taille d'une ligne dans un tableau et indiquées par un indice. Le contenu des trois caches, possédant tous quatre lignes mais respectivement à correspondance directe, associatif 2-voies et complètement associatif, est indiqué à différents instants. Nous illustrons une séquence sur laquelle un cache complètement associatif est plus efficace qu'un cache 2-voies, et une autre séquence pour laquelle ce dernier est meilleur qu'un cache à correspondance directe.



(a) Comparaison entre cache direct et cache associatif 2-voies

(b) Comparaison entre associatif 2-voies et associatif complet

FIGURE 2.2 – Comportement des différents types de caches sur des séquences d'accès à des lignes contigües en mémoire.

2.2.4 Adressage

Lorsqu'un processeur décode et exécute des instructions, les adresses mémoire manipulées ne correspondent pas directement à des emplacements en RAM. Il s'agit là d'une des conséquences des mécanismes de *virtualisation* présents sur la plupart des architectures.

Cette virtualisation consiste à fournir à chaque thread en activité l'illusion que la mémoire est de taille maximale (fonction de la taille des registres du système) et qu'il est le seul à la manipuler. Pour permettre cette illusion, des composants matériels comme la MMU assurent de manière automatique la traduction entre les adresses virtuelles contenues dans le programme et les adresses physiques correspondant à la RAM et

aux périphériques. Nous reviendrons sur le rôle du système d'exploitation dans ces mécanismes au prochain chapitre.

Il est donc possible de rencontrer dans un processeur des caches identifiant les lignes mémoire sauvegardées par leurs adresses virtuelles, et d'autres par leurs adresses physiques. Chacun de ces modes de fonctionnement possède ses avantages et ses inconvénients.

Il est nécessaire de comprendre que la traduction d'une adresse virtuelle en une adresse physique est un processus coûteux. Divers composants matériels doivent être interrogés tour à tour et, dans le pire des cas, une interruption et la participation du système d'exploitation sont nécessaires. C'est pour cela qu'il peut sembler avantageux d'éviter l'activation de ces mécanismes à chaque accès au cache.

Néanmoins, un cache fonctionnant avec seulement des adresses virtuelles pose d'autres problèmes dans un système où plusieurs processus peuvent s'exécuter sur les différents cœurs. En effet, la même adresse virtuelle peut être utilisée pour accéder à des informations complètement différentes dans deux processus différents. Il est alors impératif que le système d'exploitation nettoie l'ensemble du cache à chaque fois qu'un nouveau processus vient à s'exécuter sur l'un des cœurs utilisant ce dernier. Il s'agit là d'une contrainte forte, aux conséquences importantes sur la performance du système. C'est pour cela qu'il existe aussi des variantes des caches adressés virtuellement manipulant en plus de l'adresse un identifiant de processus [Cor10a]. Dans ce cas, la coopération du système d'exploitation est toujours nécessaire (il faut indiquer au cache que le processus a changé) mais le nettoyage complet n'est plus indispensable.

Un cache adressé physiquement règle les différents problèmes que nous venons d'identifier. Ainsi, une multitude de processus peuvent se partager le cache sans que cela provoque de conflits. Malheureusement, le décodage complet de l'adresse virtuelle utilisée par le processeur requiert un certain nombre de cycles, ce qui limite la technique aux niveaux les plus éloignés du processeur dans la hiérarchie. De plus, les processus présents sur le système entrent alors en compétition pour l'utilisation du cache, un phénomène aux multiples conséquences et sujet d'un grand nombre de publications [KCS04, ZJS10, STW92, SRD04]. Nous reviendrons d'ailleurs sur certaines des solutions à ce problème dans un prochain chapitre.

Enfin, il est intéressant de souligner que les caches adressés virtuellement ont tendance à disparaître des architectures grand public, principalement à cause de la multiplication des cœurs et du partage des caches entre ces derniers. L'adressage physique est en effet plus simple à mettre en œuvre dans ces cas.

2.2.5 Hiérarchie et partage des caches entre cœurs

La plupart des processeurs récents possèdent plusieurs caches organisés en niveaux. Le niveau le plus proche d'un cœur, nommé L1, est le plus petit d'entre eux. Cette taille se justifie de deux façons : il existe peu de place assez près d'un cœur pour y placer une mémoire, et le temps nécessaire pour rechercher une ligne dans un cache augmente avec la taille de ce dernier (surtout pour les caches associatifs). Les niveaux suivants étant plus éloignés des cœurs, ils disposent de plus d'espace et peuvent être plus complexes en circuits, ils sont donc de plus en plus gros. Notons que le terme LLC pour *Last Level Cache* désigne le niveau de cache le plus éloigné des cœurs et qu'il

s'agit de plus en plus souvent du troisième niveau (L3).

Bien qu'il soit nécessaire de fournir à chacun des cœurs un accès à cette hiérarchie, il serait généralement trop coûteux en terme d'espace de la dupliquer en totalité. Les niveaux les plus gros de la hiérarchie sont donc généralement partagés entre plusieurs cœurs.

Ces caches partagés ne sont pas pour autant plus simples, puisqu'il est courant de ne les partager qu'entre certains cœurs (et non pas la totalité de la puce). Il est ainsi possible de rencontrer un processeur multicœur où chaque cœur possède un cache L1 privé, un cache L2 partagé avec un autre cœur et un niveau L3 partagé par toute la puce.

De nombreux travaux ont montré l'utilisation de cette hiérarchie des caches, et leur partage pour l'accélération des applications parallèles. On y trouve notamment des techniques de préchargement par cœur interposé [JLLS06] : un fil d'exécution, appelé *helper thread*, est utilisé pour accéder à l'avance aux données nécessaires à un fil s'exécutant sur un autre cœur mais partageant le même cache. Bien que nécessitant une synchronisation très fine entre les fils d'exécution, cette technique améliore grandement l'efficacité d'une application. Elle rejoint aussi les algorithmes parallèles à fenêtre [TDR10], où les données très proches en mémoire (dans une fenêtre) sont traitées parallèlement sur les différents cœurs d'une machine.

2.3 Interactions entre caches et applications

La hiérarchie des caches joue un rôle déterminant dans la performance d'une application. En effet, selon l'organisation des données en mémoire ou l'ordre des accès à cette mémoire, une architecture de cache donnée va être plus ou moins efficace pour accélérer l'application.

2.3.1 Classification des défauts de cache

De manière générale, la capacité d'une application à utiliser le cache est mesurée par le nombre de défauts de cache (à chacun des niveaux) déclenchés par celle-ci lors de son exécution. Le modèle 3C [HS89] définit les trois raisons pouvant entraîner un défaut de cache et donc les pistes à suivre pour optimiser une application :

- défaut de capacité (*Capacity miss*) apparaît lorsque le cache n'est pas assez grand pour contenir tous les emplacements mémoire séparant deux accès à la même ligne.
- défaut obligatoire (*Compulsory miss*) est déclenché lorsqu'un emplacement mémoire est accédé pour la première fois.
- défaut par conflit (*Conflict miss*) résulte de l'éviction d'une ligne de cache à cause d'une trop faible associativité : il a été nécessaire de supprimer une ligne d'un ensemble associatif alors que des données plus anciennes existaient dans d'autres ensembles.

On retrouve dans de nombreux domaines des travaux visant à diminuer l'occurrence de ces évènements, comme par exemple le préchargement matériel ou logiciel pour les *compulsory miss*, la réorganisation des données en mémoire et la modification de l'ordre des accès par la compilation [BAM+96] ou encore des politiques d'attribution de la

mémoire physique optimisées pour le cache au sein des systèmes d'exploitation (nous en reparlerons en détail dans le chapitre suivant).

2.3.2 Préchargement automatique

Dans sa version la plus simple, le préchargement automatique consiste à systématiquement charger la ligne mémoire adjacente à celle ayant provoqué un défaut de cache. Cette technique profite, au même titre que les lignes de cache, de la localité spatiale de l'application s'exécutant.

Il n'est pas rare que ce préchargement soit complété par un mécanisme spéculatif. Dans ce cas, un matériel dédié au sein du contrôleur mémoire est chargé de détecter si les accès mémoire réalisés correspondent à des accès séquentiels avec sauts (chaque accès est séparé de la même distance). Ce matériel est généralement capable de détecter plusieurs flux d'accès de ce type en parallèle et donc d'améliorer substantiellement la performance d'une application. Malheureusement, ce mécanisme peut s'avérer trop agressif si les accès mémoire de l'application ne respectent pas un schéma séquentiel. Pour plus d'informations sur le fonctionnement précis de ces mécanismes de préchargement, se référer (par exemple) aux manuels de chez Intel [Cor10b].

2.3.3 Distance de réutilisation

Par souci de simplicité, de nombreux modèles considèrent les défauts par conflits comme inexistants (comme sur un cache à associativité complète). Dès lors, la compréhension de la performance d'une application passe par la séparation des défauts de cache entre les obligatoires (pour lesquels des optimisations spécifiques comme le préchargement logiciel existent) et ceux de capacité (nécessitant plus d'analyse et de plus amples modifications de l'application). À ce titre, la distance de réutilisation est le modèle le plus utile.

Définition 2 *La distance de réutilisation est la fonction qui à chaque accès mémoire d'une exécution associe le nombre d'instructions touchant la mémoire le séparant de l'accès précédent à la même donnée.*

Intuitivement, cette distance de réutilisation détermine si un accès mémoire déclenchera un défaut ou non. En effet, dans le cas d'un cache complètement associatif implémentant l'algorithme LRU parfaitement, il faut exactement autant d'accès à des données qu'il y a de lignes de cache pour évincer une donnée de ce dernier. Cette distance nous indique donc si suffisamment de données différentes ont été chargées en cache entre deux accès à la même ligne pour provoquer son éviction. Ainsi, dans le cas d'un cache idéal (associatif complet avec LRU), une application dont toutes les distances de réutilisation non infinies (premier accès à une donnée) seraient inférieures à la taille du cache ne ferait que des défauts de cache obligatoires.

Bien que les hypothèses derrière cette définition puissent paraître excessives, plusieurs travaux ont montré la pertinence de ce modèle dans la prédiction de performance d'une application [BD01, SY05].

2.3.4 *Working set*

Fortement liés à la distance de réutilisation, les *working sets* représentent eux aussi de façon intéressante l'influence du cache sur une application.

Définition 3 *Le working set d'une application désigne un intervalle de valeurs possibles pour la quantité de ressources allouées pour lesquelles la performance du programme est stable. Par extension, le terme de working set désigne aussi la quantité minimale de ressources pour laquelle la performance de l'application atteint un certain seuil.*

Si l'on mesure pour une application donnée sa performance en fonction de la quantité de cache qui lui est attribuée, on obtient généralement une courbe proche d'une fonction constante par morceaux. La performance de l'application n'étant pas nécessairement modifiée d'un niveau d'attribution des ressources à un autre, certains plateaux apparaissent sur cette courbe. Un working set désigne communément ce type de plateaux ainsi que la quantité minimale de ressources à partir de laquelle ce dernier apparaît.

Bien que cette notion ait été développée à l'origine pour répondre aux problématiques d'allocation de mémoire physique dans les systèmes d'exploitation et pour des périodes de temps réduites, des travaux ont démontré son utilité pour l'analyse des besoins en cache [Dre07, BKSL08].

Nous détaillerons dans un prochain chapitre certaines des applications possibles de cette notion. Il existe néanmoins une relation évidente entre la distance de réutilisation et ces working sets : connaitre la distribution des distances de réutilisation d'une zone mémoire nous donne la quantité de cache nécessaire pour qu'elle reste en cache.

Plus formellement, si on nomme $H(d)$ le nombre d'accès mémoire avec une distance de réutilisation de d, alors le nombre de défauts de cache Q ayant lieu avec un cache de taille C est de : $Q(C) = \sum_{d=C+1}^{\infty} H(d)$ (le premier accès à un élément possède une distance de réutilisation infinie). Ainsi, les working sets d'une application peuvent être déterminés en connaissant la distribution des distances de réutilisation : s'il existe un intervalle $[i, j]$ pour lequel H est nul (*i.e.* $\forall d \in [i, j]\ H(d) = 0$), alors Q restera constant pour C dans cet intervalle. La figure 2.3 illustre plus précisément cette relation.

2.4 Informations fournies par le matériel

L'analyse des performances d'une application nécessite une certaine capacité à mesurer le comportement du matériel sous-jacent et à appréhender l'influence de la topologie de la machine sur ce comportement. De ce besoin sont nées deux technologies capitales : les compteurs de performance matériels et les techniques de détection du matériel.

2.4.1 Compteurs de performance matériels

Conçus au départ pour permettre à un fondeur de tester et valider une architecture, les compteurs de performance matériels enregistrent certains événements et permettent à un processus de capturer ces informations. Un compteur de performance matériel

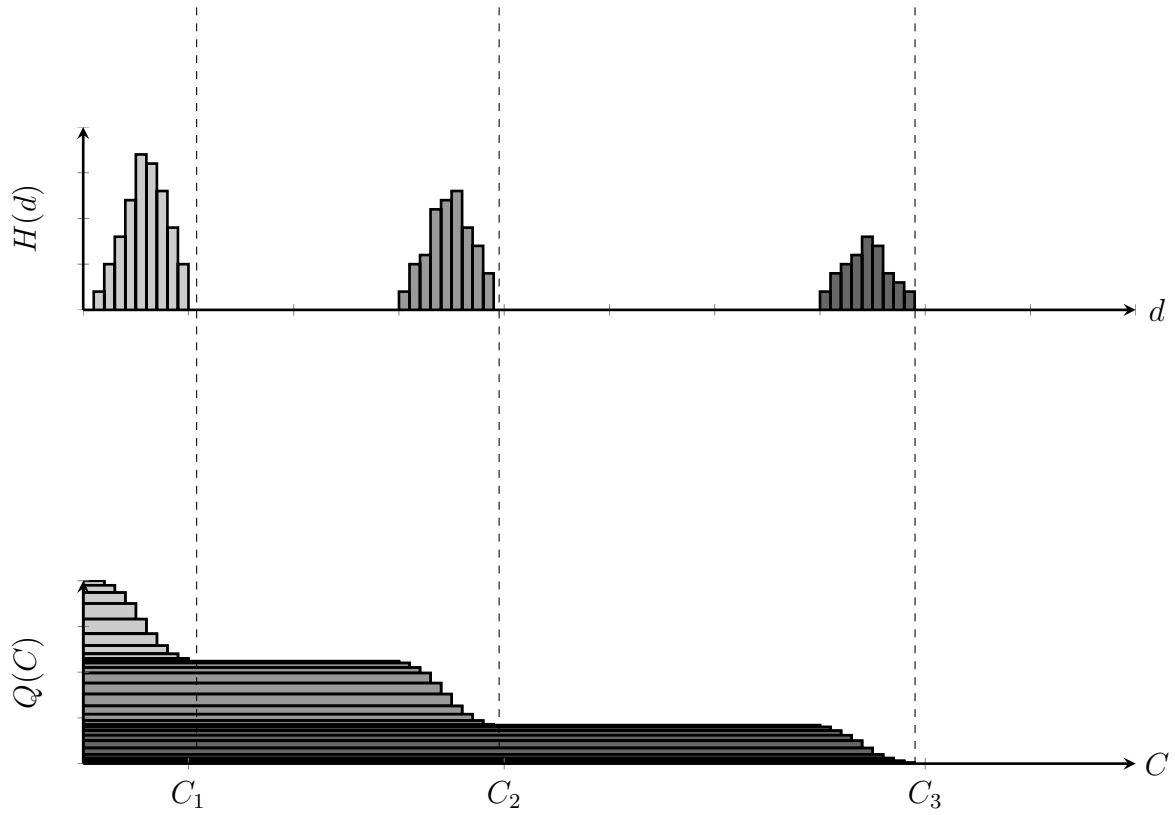


FIGURE 2.3 – Relation entre distribution des distances de réutilisation (H) et défauts de cache (Q). Les working sets apparaissent lorsqu'aucun accès mémoire ne bénéficie d'une augmentation de la taille du cache.

est un registre spécifique du processeur servant à comptabiliser certains événements matériels comme les cycles écoulés, le nombre d'instructions exécutées ou le nombre de défauts de cache déclenchés.

On distingue généralement deux types de compteurs : les compteurs fixés, en lecture seule, qui enregistrent un évènement particulier de manière systématique et les compteurs configurables, plus lesquels l'utilisateur spécifie l'évènement à compter à l'aide d'instructions privilégiées.

Il est aussi possible, sur certaines architectures, de configurer un compteur pour qu'il génère une interruption lorsque sa valeur dépasse un certain niveau. Ce mécanisme permet notamment d'enregistrer l'adresse de l'instruction ayant provoqué le dépassement et ainsi de mesurer à moindre coût si certaines instructions d'un programme ne déclenchent pas plus d'évènements que d'autres. La table 2.1 donne un (petit) extrait de la liste des évènements disponibles sur un processeur moderne.

2.4.2 Niveaux de trace et utilisations des compteurs

La plupart des architectures distinguent, lors de l'enregistrement d'un évènement matériel, si ce dernier a été déclenché durant l'exécution du système d'exploitation ou d'un programme utilisateur. Il est ainsi possible de s'abstraire lors de la mesure d'une application des problèmes que soulève le système. Il faut néanmoins être pru-

Code	Description
0x40000024	Tics horloge observés durant l'exécution
0x40000025	Accès au cache L2
0x40000026	Défauts de cache L2
0x40000036	Instructions complétées
0x4000001b	Instructions de préchargement déclenchées
0x40000043	Cycles passés avec les interruptions masquées

TABLE 2.1 – Quelques évènements matériels disponibles sur un processeur AMD Opteron 6174 (4 compteurs configurables disponibles).

dent, puisque les évènements matériels ayant lieu en noyau peuvent parfaitement être liés à une organisation particulière de l'application. Par exemple, l'organisation des lectures/écritures d'un fichier peut influer sur les temps d'attente du processeur lors de transferts entre la mémoire et les disques.

Enfin, ces compteurs ne sont pas sans inconvénients. Premièrement, ils nécessitent une modification du système d'exploitation, puisque les instructions de contrôle sont privilégiées. Ensuite, la liste des évènements mesurables ainsi que le nombre et type de compteurs changent d'un processeur à l'autre. Voilà pourquoi la plupart des utilisateurs se reposent sur des outils ou bibliothèques spécialisés tels que PAPI [BDG+00], perf [Lin], Likwid [THW10] ou VTune [Rei05].

A titre d'exemple, la figure 2.4 donne un extrait de code C utilisant PAPI pour mesurer les défauts de cache déclenchés durant son exécution.

```
#include <papi.h>

int main(void) {
    long long val[2];
    int events[2] = { PAPI_L1_DCM, PAPI_L2_TCM};
    PAPI_library_init(PAPI_VER_CURRENT);
    PAPI_start_counters(events,2);

    do_stuff();

    PAPI_stop_counters(val,2);
    printf("L1d miss %lld, L2 miss %lld\n,val[0],val[1]);
    return 0;
}
```

FIGURE 2.4 – Exemple de code C mesurant les défauts de cache durant son exécution à l'aide de PAPI.

2.4.3 Identification des caractéristiques de la machine

La topologie de la machine et la complexité du partage des caches devenant nécessaire à la compréhension des performances d'une application et à son optimisation, la

niveau	taille	associativité	indexation
L1 Data	32 KB	8	physique
L1 Instruction	32 KB	8	physique
L2	3072 KB	12	physique
L3	16 MB	16	physique

Table 2.2 – Caractéristiques des caches d'une puce Intel Xeon X7460

question de leur détection se pose.

Deux approches coexistent sur ce problème : l'identification par le fabricant de la puce et la détection par l'expérimentation. La première approche est simple sur le papier : le fabricant fournit une instruction spécifique ne servant qu'à cela. On retrouve ce principe derrière l'instruction `cpuid` dans les processeurs Intel.

Cette instruction permet, en lui fournissant certains paramètres, de récupérer des codes identifiant le type d'architecture sous-jacent. Ces codes sont ensuite à reporter dans une table de traduction afin d'obtenir les caractéristiques précises. C'est justement à ce niveau que la technique montre sa principale faiblesse : la table n'existe que sur papier, dans la documentation du fabricant. Ainsi, chaque utilisateur nécessitant cette information se retrouve obligé de conserver à l'intérieur de ses outils toute la table s'il souhaite les rendre portables. Ce problème est aussi accentué par l'existence d'erreurs sur certains processeurs (retour de mauvais codes classiquement). On peut alors trouver dans le code de certains outils une multitude de cas particuliers de traductions visant à corriger ces erreurs.

À titre d'exemple, la figure 2.5 et le tableau 2.2 illustrent la hiérarchie mémoire d'une machine possédant 4 puces de 6 cœurs ainsi que ses caractéristiques, comme le reporte l'outil HWLoc [BCOM+10] à partir de `cpuid`.

2.4.4 Détection logicielle de caractéristiques matérielles

Pour compléter cette identification de la topologie, il est aussi utile de réaliser des mesures expérimentales des temps d'accès aux différentes ressources du système. Une image précise de ces différentes latences permet par exemple de réaliser de meilleures optimisations en cache (nous en verrons quelque unes dans un chapitre ultérieur) ou simplement de mieux comprendre les performances d'une application.

Ces mesures expérimentales reposent toutes sur l'utilisation d'applications très spécifiques (appelées microbenchmarks), ne réalisant qu'un seul type d'opération et un contrôle très fin de leur exécution. Certaines fonctionnalités du système d'exploitation sont alors mises à contribution (les ordonnanceurs temps-réel par exemple dont nous parlerons au chapitre suivant) afin de stabiliser les résultats et de s'assurer que l'on mesure bien ce que l'on souhaite.

Détaillons par exemple le code nécessaire à une mesure de la latence des caches (et de la mémoire physique). Le principe de ce code, popularisé par Ulrich Drepper [Dre07], est de mesurer le temps d'exécution d'un programme réalisant des lectures aléatoires dans une zone mémoire de taille fixée. En faisant varier cette taille, le programme se retrouve capable d'utiliser pleinement certains niveaux du cache mais pas d'autres. Il faut néanmoins qu'un nombre important d'accès ait lieu pour que le temps moyen

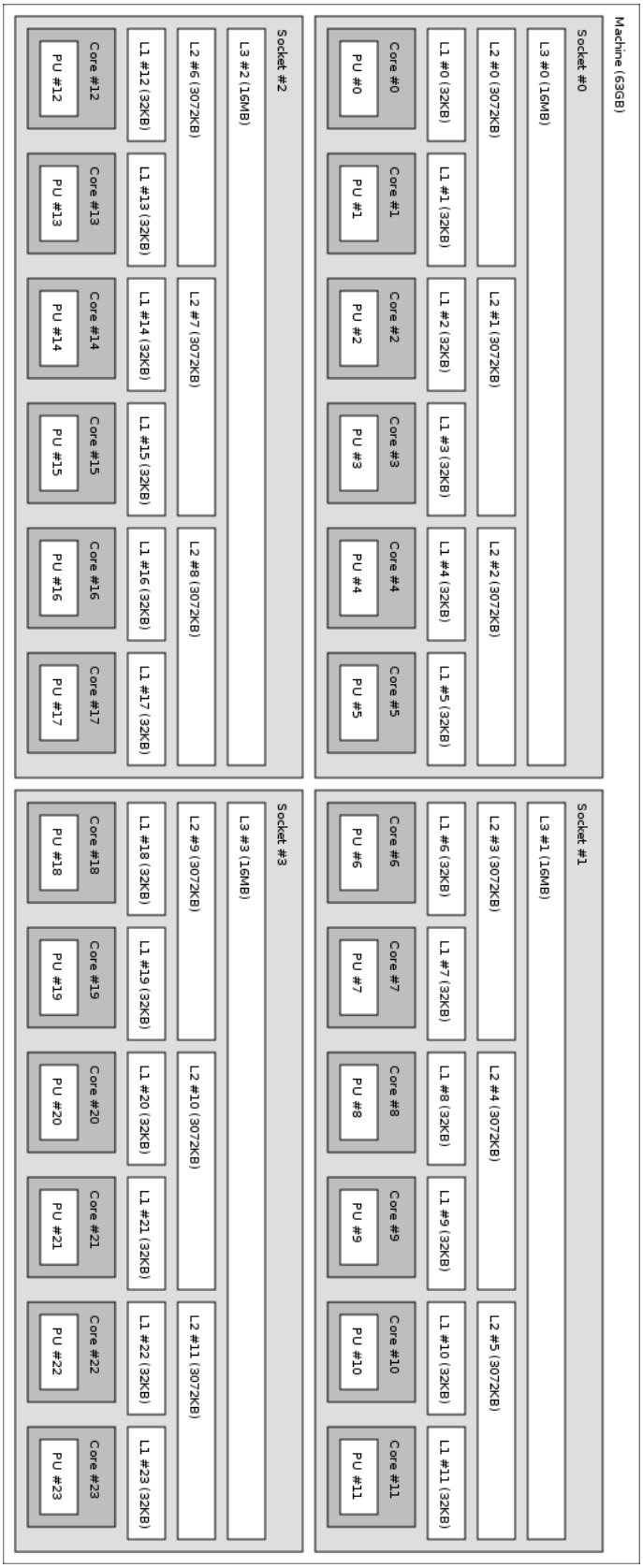


FIGURE 2.5 – Hiérarchies processeur et mémoire d'une machine à 4 processeurs de 6 cœurs (Intel Xeon X7460)

d'une lecture se stabilise ($n \log n + (k-1)n \log n$ pour k accès à chaque élément, d'après le *Coupon collector problem*).

La figure 2.7 présente le code nécessaire pour une telle expérience. Notons que chaque élément de la zone mémoire est ici une structure de 64 octets, afin que chaque lecture touche une ligne de cache différente.

La figure 2.6 présente le temps d'accès moyen à un élément de la région mémoire en fonction de sa taille sur la machine présentée un figure 2.5. Afin de stabiliser les résultats, le programme était fixé sur un seul cœur et était ordonnancé en temps-réel. Chaque point est la moyenne de 30 résultats. Les intervalles de confiance à 95% sont trop petits pour être visibles.

Ce type de résultat laisse apparaître les différentes latences d'accès à la mémoire d'une machine. Lorsque la zone mémoire est de petite taille, un nombre suffisant d'accès la stockeront en cache L1, et le temps moyen d'accès par élément restera très faible (si le nombre d'accès total est suffisamment important). Une zone de plus en plus grande se placera en cache L2, puis L3, avant de ne plus pouvoir être cachée et que le programme accède en majorité à la RAM. Il est ainsi possible de constater une différence de l'ordre d'un facteur 20 entre un accès au L1 et la RAM sur ce processeur.

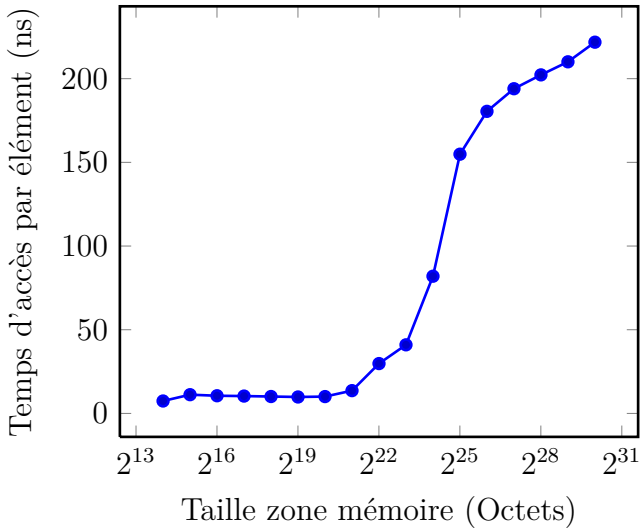


FIGURE 2.6 – Durée moyenne d'une lecture aléatoire selon la taille de la zone mémoire.

```
/* make CFLAGS="-O3 -D_GNU_SOURCE" LDFLAGS="-lgsl -lgslcblas -lm -lrt"
* SCHED_FIFO appliqué par chrt en ligne de commande.
*/

#include <sched.h>
#include <stdlib.h>
#include <time.h>
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

struct elem {
    int v;
    struct elem *n;
    char pad[64-sizeof(int)-sizeof(struct elem *)];
};

int main(int argc, char *argv[]) {
    unsigned long i,max;
    assert(argc == 2);

    /* on récupère la taille comme une puissance de 2 */
    int log = atoi(argv[1]);
    int taille = 1<<log;
    assert(taille > 0  && taille < (1<<27));
    /* plus simple que la véritable formule et tout aussi efficace pour k=4 */
    max = 3L*log*taille;
    struct elem *tableau = malloc(taille * sizeof(struct elem));
    assert(tableau != NULL);
    struct timespec debut, fin;

    /* randomisation de la liste
    * mt19937 est un générateur de qualité (Mersenne twister)
    */
    gsl_rng *r = gsl_rng_alloc(gsl_rng_mt19937);
    for(i = 0; i < taille; i++)
        tableau[i].v = i;
    gsl_ran_shuffle(r,(void *)tableau, taille,sizeof(struct elem));
    struct elem *cur = &(tableau[tableau[0].v]);
    for(i= 0; i < taille; i++) {
        cur->n = &(tableau[tableau[i].v]);
        cur = cur->n;
    }
    cur->n = &(tableau[tableau[0].v]);
    gsl_rng_free(r);

    /* volatile évite que gcc considère la boucle et la somme
     * comme inutile */
    volatile int somme=0;
    cur = &(tableau[tableau[0].v]);
    clock_gettime(CLOCK_REALTIME, &debut);
    for (i = 0; i < max; ++i) {
        somme += cur->v;
        cur = cur->n;
    }
    clock_gettime(CLOCK_REALTIME, &fin);

    long long int temps_nano = (fin.tv_nsec - debut.tv_nsec) +
        1e9* (fin.tv_sec - debut.tv_sec);

    free(tableau);
    printf("%d %lu %Ld\n", taille, max, temps_nano);
    return 0;
}
```

FIGURE 2.7 – Code de mesure de la vitesse d'accès aux différents niveaux de cache d'une machine.

Fonctionnement d'un système d'exploitation moderne 3

Sommaire

3.1 Abstractions offertes par le système 26
3.1.1 Processus et threads 26
3.1.2 Mémoire virtuelle 27
3.1.3 Groupes de processus 27
3.2 Ordonnancement système 28
3.2.1 Ordonnancement équitable sous Linux 29
3.2.2 Ordonnancement par groupes de processus 30
3.2.3 Ordonnancement Temps-réel 31
3.3 Modèle d'utilisation du processeur 32
3.3.1 Résolution de l'ordonnanceur 32
3.3.2 Charge processeur 32
3.3.3 Mesure de charge 33
3.4 Gestion de la mémoire virtuelle 33
3.4.1 Pagination 34
3.4.2 Gestion de la mémoire physique 34
3.4.3 Coloration de page 35
3.4.4 Gestion d'une mémoire virtuelle avec coloration 35

Nous nous intéressons dans cette thèse à des machines disposant d'un système d'exploitation complet et moderne. Ce système d'exploitation est responsable d'un certain nombre de services du point de vue des utilisateurs, notamment l'abstraction d'une partie de la complexité du matériel, l'isolation entre les applications en présence et une certaine qualité de service au niveau du partage des ressources. Ces services, de par leur implémentation et les politiques de contrôle qui leur sont associées, ont une influence sur la performance d'une application.

Nous détaillerons donc dans ce chapitre certaines des abstractions présentées par un système d'exploitation aux applications. Nous étudierons ensuite en détail deux composants particuliers d'un système d'exploitation : l'ordonnanceur système, responsable de la répartition du processeur aux applications en activité et le gestionnaire de mémoire virtuelle. Pour finir, nous établirons un modèle de l'utilisation des processeurs d'une machine par une application, à partir des différentes politiques d'ordonnancement en activité.

Lorsqu'il s'agira de détailler certaines technologies, nous nous concentrerons sur leur équivalent sous Linux, le système d'exploitation le plus populaire dans le domaine du calcul haute performance (91% du Top500 en Juin 2011).

3.1 Abstractions offertes par le système

La plupart des systèmes d'exploitation confinent chaque application dans une forme de machine abstraite. On retrouve autour de cette notion de machine abstraite les concepts de processus et de mémoire virtuelle par exemple. Ce mécanisme permet notamment à plusieurs applications d'exister simultanément sur la machine sans pour autant complexifier leur programmation. Cependant, le système d'exploitation, en implémentant ces différentes abstractions, devient responsable d'une partie de la performance des applications en exécution. De plus, le système implémente généralement un contrôle de la répartition du matériel entre les applications présentes, influençant d'autant plus sur leur capacités. Nous détaillons ici certaines des abstractions offertes par un système à une application.

3.1.1 Processus et threads

L'unité de base représentant, dans un système POSIX, une application est le processus. Cette notion de processus englobe l'ensemble des composants nécessaires à une communication entre le système d'exploitation et l'application. Le système d'exploitation associe par exemple à un processus la liste de fichiers ouverts par le programme (dont les entrées/sorties standards), le code du programme à exécuter, les arguments que ce programme a reçu au démarrage, le chemin courant dans le système de fichier ou encore les variables d'environnement disponibles. Dans sa version la plus courante, un processus identifie aussi un fil d'exécution. En effet, la plupart des systèmes sauvegardent l'état courant d'un fil d'exécution dans l'espace mémoire d'un processus.

Lorsque plusieurs fils d'exécution se partagent une même mémoire, on parle alors de processus légers ou de threads. Ce partage de la mémoire peut notamment être utilisé pour la communication ou la synchronisation entre les threads. Les détails exacts de ce partage varient d'un système à un autre, de même que la relation entre un thread et un processus. Sous Linux, tous les fils d'exécution sont considérés égaux pour ce qui est de la répartition des ressources.

Notons enfin l'existence d'une hiérarchie entre les processus : un processus est nécessairement créé par un autre et peut en créer autant qu'il le souhaite. Les processus présents sur un système forment donc une arborescence, la racine étant un processus particulier car initialisé par le système d'exploitation. Cette hiérarchie est aussi complétée par les processus légers, qui appartiennent tous à même processus et donc au même nœud de l'arbre. Certains mécanismes avancés utilisent cette hiérarchie pour isoler certains processus des autres, en redéfinissant la racine du système de fichier par exemple (`chroot`). Ces modifications se propagent alors automatiquement à tous les descendants du processus cible dans la hiérarchie.

3.1.2 Mémoire virtuelle

Comme nous l'avons déjà énoncé, un processeur manipule des adresses dites virtuelles lorsqu'il décode des instructions. Une unité matérielle spécifique (MMU) traite de manière semi-automatique la traduction de ces adresses en leurs correspondants véritables (physiques).

On dénomme communément par espace d'adressage virtuel l'ensemble des emplacements en mémoire virtuelle auxquels il est possible d'associer une adresse. Ainsi, il est théoriquement possible d'adresser 4 GiB de mémoire sur une architecture dite 32 bits (les registres communs ont une taille de 32 bits), et 4 PiB (2^{64} octets) sur les architectures 64 bits (en pratique la plupart des systèmes se limitent à 2^{48}).

La mémoire virtuelle fonctionne en pratique par un découpage de la mémoire en blocs contigus appelés *pages* et une gestion par le système d'exploitation de la correspondance entre des pages physiques et des pages virtuelles. Malgré les nombreux avantages de la technique, elle reste coûteuse à de nombreux égards, notamment en mémoire système : le système conserve pour chaque page physique un certain nombre d'informations sur leur utilisation. Le mécanisme matériel est aussi particulièrement sollicité.

Du point de vue des applications, et nous reviendrons sur ce point plus tard, la mémoire virtuelle a une influence critique sur leur performance en cache. En effet, une application ne maîtrise pas la façon dont le système associe les pages virtuelles aux pages physiques, alors que sa performance en cache (indexé physiquement la plupart du temps) est directement liée aux adresses physiques utilisées.

Enfin, la plupart des systèmes d'exploitation modernes considèrent qu'un espace d'adressage est associé à un processus particulier. Cet espace contient donc le code et les données (statiques et dynamiques) du programme s'exécutant. Ainsi, tous les threads d'un même processus ont accès au même espace d'adressage et peuvent donc communiquer en utilisant les variables globales du programme.

3.1.3 Groupes de processus

Il peut être intéressant pour l'utilisateur de spécifier qu'un ensemble de processus ne devrait pas recevoir plus d'un certaine quantité de mémoire ou être considéré comme un seul lors de la répartition des processeurs. Ce besoin est notamment exprimé par des entreprises louant du matériel, pour contrôler finement ce qu'obtient un client, ou des administrateurs système pour éviter qu'un utilisateur ne monopolise une machine.

A ce titre, certains systèmes d'exploitation comme Linux fournissent une notion de groupes de processus (à ne pas confondre avec les groupes d'utilisateurs). Les différentes politiques de gestion des ressources du système sont alors adaptées pour prendre en compte ces groupes. Nous étudierons notamment le cas de l'ordonnanceur système dans la section suivante.

Dans sa version la plus souple, un groupe de processus est défini comme n'importe quel ensemble, dynamique, de processus. Le système garanti généralement que tous les processus appartenant à un groupe doivent être traités de manière équitable pour l'attribution des ressources. Par souci de simplicité, l'appartenance à un groupe est héréditaire : tout processus créé hérite des groupes de son père. Il est bien sûr possible

pour un processus de s'isoler dans un nouveau groupe sur simple demande au système d'exploitation.

Il en découle que les groupes forment entre eux une hiérarchie, distincte d'une quelconque hiérarchie entre processus : tous les processus démarrent dans un groupe *racine* et peuvent créer ou rejoindre des groupes dynamiquement pour s'isoler les uns des autres. Notons tout de même que cette hiérarchie peut être particulièrement complexe sur un système comme Linux. En effet, Linux permet de placer un processus dans un groupe différent pour chaque gestionnaire de ressource considéré. Par exemple, deux processus peuvent se placer dans un même groupe pour la répartition des processeurs, mais dans deux groupes séparés pour celle de la mémoire.

Du point de vue du calcul haute performance, les groupes de processus présentent plusieurs intérêts. Tout d'abord, un gestionnaire de ressource de type `batch scheduler`, responsable de la répartition des nœuds d'un supercalculateur, peut utiliser les groupes pour répartir équitablement aux différents utilisateurs les ressources de calcul. Ensuite, certains systèmes comme Linux fournissent des statistiques d'utilisation des ressources de la machine par groupe. Ce type d'information peut s'avérer utile pour vérifier la consommation d'un utilisateur ou d'une application en particulier. Enfin, les groupes fournissent une forme de virtualisation à moindre coût, en permettant d'isoler du point de vue de certaines ressources système, les différentes applications de la machine.

3.2 Ordonnancement système

L'ordonnanceur système est responsable, sur une machine, de la répartition des unités d'exécution aux différents fils d'exécution en activité (processus ou threads). Il s'agit la plupart du temps à la fois d'un objectif d'efficacité (utiliser au maximum les unités disponibles) et un objectif d'équité (répartir équitablement les unités aux différents threads).

Par souci de simplification, nous nommerons dans la suite de ce chapitre *threads système* ou *tâches* les différents fils d'exécution pris en considération par un ordonnanceur. Ces threads peuvent être dans trois états à l'intérieur du système. Premièrement, il peuvent être *bloqués* en attente d'une ressource ou d'un évènement et il est dans ce cas impossible de leur attribuer une unité d'exécution. Deuxièmement, un thread *prêt* peut être choisi par l'ordonnanceur. Enfin, un thread *actif* désigne une tâche en cours d'exécution. De même, nous appellerons *processeur* toute unité matérielle permettant l'exécution indépendante d'un thread système. Ainsi, une machine possédant deux puces bi-cœurs, chacun des cœurs disposant d'une technologie SMT permettant deux processus en simultané est composée, d'un point de vue système, de 8 processeurs.

Les différentes classes d'ordonnanceurs système se différencient essentiellement sur deux points : l'algorithme décidant du fil d'exécution auquel attribuer l'unité d'exécution considérée et le type d'évènements entraînant la réactivation d'un ordonnanceur (pour effectuer un nouveau choix). Dans tous les cas, ce type de fonctionnement est désigné par le terme de *temps partagé* : l'ordonnanceur découpe de fait l'attribution d'un processeur en périodes de temps et les distribue ensuite aux différents threads. Nous appellerons l'intervalle de temps séparant deux réveils de l'ordonnanceur *jiffy* en référence à la nomenclature en vigueur sous Linux. Notons que cette période de temps est définie de façon relativement floue : si le processus actif venait à réaliser un appel sys-

tème le bloquant, l'ordonnanceur s'activerait de nouveau, mettant fin prématurément à la période de temps en cours.

L'ordonnanceur joue donc un rôle capital dans la performance des applications. Tout d'abord parce qu'il est responsable de la quantité de temps processeur qu'une application peut obtenir. Ensuite, parce qu'un certain nombre de travaux ont montré l'influence du *bruit système* sur la performance d'une application de calcul haute performance. Le bruit système désigne l'ensemble des évènements, ordonnanceur en tête, pouvant désynchroniser plusieurs threads d'une même application. L'ordonnanceur se déclenchant régulièrement, même lorsqu'aucune répartition n'est nécessaire, et de façon désynchronisée sur chacun des processeurs, il peut induire des latences sur les opérations de communication/synchronisation collectives d'une application. Sur des applications effectuant beaucoup de synchronisations, ce type de latence a une grande influence sur la performance. Enfin, il n'est pas rare qu'un ordonnanceur privilégie certaines classes d'applications dans l'attribution des processeurs. Les applications réalisant un grand nombre d'entrées/sorties par exemple sont souvent favorisées de façon à compenser leur difficulté à compléter les quantums de temps attribués par l'ordonnanceur (cela augmente par la même occasion leurs chances d'effectuer de nouvelles requêtes bloquantes rapidement). On retrouve aussi ce type de problématique dans les travaux autour des cœurs dédiés aux entrées/sorties : l'application sacrifie alors certains cœurs pour améliorer ses accès disque.

3.2.1 Ordonnancement équitable sous Linux

La majorité des systèmes d'exploitation implémente plus d'un type d'ordonnanceur. En effet, la norme POSIX recommande qu'il soit possible pour l'utilisateur de spécifier la politique d'ordonnancement à appliquer à une tâche. Dans cette norme, la politique par défaut est de répartir le plus équitablement possible le temps processeur aux différentes tâches prêtes. Pour cela, l'ordonnanceur est réveillé plusieurs centaines de fois par seconde (entre 200 et 1000 fois sous Linux) pour équilibrer au mieux la répartition des *jiffies*.

Nous détaillons ici l'implémentation de cet ordonnanceur pour le noyau Linux, dans sa dernière refonte (issue du noyau version 2.6.23), implémentant par ailleurs le principe des priorités entre tâches des systèmes POSIX : le *Completely Fair Scheduler*. Cet ordonnanceur possède deux particularités : chaque processeur conserve les threads prêts sous la forme d'un arbre binaire (rouge-noir) indexé par leur temps d'exécution, et l'ordonnanceur ne possède pas de période de réveil fixée (on parle de *tick-less scheduler*).

L'algorithme utilisé sur chacun des processeurs est très simple : à chaque réveil, l'ordonnanceur comptabilise le temps d'exécution du thread courant. Il replace ensuite ce dernier dans l'arbre binaire ordonné. La tâche la plus à gauche dans l'arbre est alors celle qui s'est le moins exécutée, et elle recevra donc le processeur. Le réveil d'un ordonnanceur s'effectue régulièrement lorsqu'un thread réalise des appels systèmes (ou qu'une interruption se déclenche) et est garanti par une alarme dans le cas où aucun autre événement n'aurait lieu sur une période de temps assez large.

Un tel algorithme d'ordonnancement réparti donc le temps processeur sur de longues périodes. En effet, l'équité entre les tâches ne sera obtenue qu'après un grand nombre d'activations, le temps que chacune des tâches ait utilisé approximativement le même

nombre de *jiffies*. De ce point de vue, aucune garantie n'est fournie sur le temps nécessaire à un tel équilibrage par le noyau Linux.

Cet algorithme est aussi complété par une gestion de priorités entre les threads : plus une tâche possède une priorité élevée et moins son temps d'exécution véritable est pris en compte dans son temps d'exécution utilisé. Le système est ainsi configuré pour qu'une tâche de priorité 0 obtienne 10% de temps processeur de plus qu'une tâche de priorité 1. Le même genre de mécanisme est utilisé pour privilégier les tâches interactives (souvent bloquées sur des entrées/sorties) afin d'améliorer la réactivité du système. Le temps processeur alloué à une tâche dépend ainsi du nombre de tâches prêtes, de leurs priorités respectives et de leur nature.

Notons enfin que certaines des variables de l'ordonnanceur sont modifiables à l'exécution, notamment la quantité minimale de temps processeur à accorder à une tâche et le temps maximal entre deux réveils.

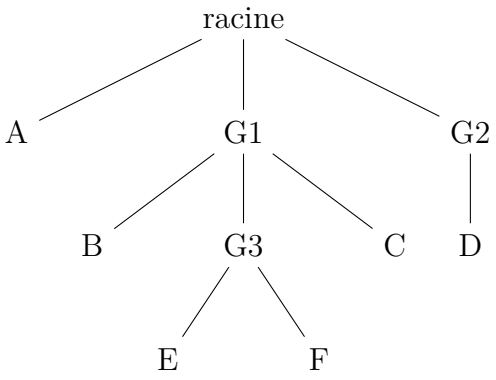
3.2.2 Ordonnancement par groupes de processus

Si une politique de répartition équitable semble correspondre à la plupart des besoins d'un utilisateur, notamment en terme d'interactivité, elle n'est pas sans défauts. En effet, cette répartition ne prend pas en compte l'équité entre plusieurs utilisateurs sur un système partagé : il suffit de créer plus de tâches qu'un autre utilisateur pour obtenir plus de temps processeur que lui. Pour régler ce type de problème, certains systèmes implémentent un ordonnancement prenant en compte les groupes de processus. Il s'agit alors de répartir le temps processeur équitablement entre des groupes, et à l'intérieur d'un groupe entre les tâches. L'administrateur peut ensuite mettre en place des politiques de création de groupes par utilisateur ou par type d'application.

Dans le cas de Linux, les groupes possèdent leur propre priorité. Alors que les priorités POSIX établissent des seuils d'attribution (un niveau de priorité équivaut à 10% de processeur en plus), un groupe obtient un temps processeur directement proportionnel à sa priorité. Pour un niveau donné de la hiérarchie, un groupe A obtient ainsi une portion du temps processeur T_P du groupe père P en fonction de sa priorité $prio_A$ et de la somme des priorités des groupes au même niveau S_g : $T_A = \frac{prio_A}{S_g} * T_P$.

Les groupes formant une hiérarchie, l'ordonnanceur redistribue le temps processeur de manière récursive à travers celle-ci. Le groupe racine reçoit donc tout le processeur, les groupes au niveau inférieur se partagent ce temps de manière proportionnelle à leurs priorités respectives et ainsi de suite à l'intérieur de chaque groupe. De ce point de vue, si un groupe contient à la fois des processus et des groupes, alors chacun des processus est considéré comme un groupe distinct, avec la même priorité par défaut que le groupe racine.

Calculer le temps processeur que va obtenir une tâche demande donc de connaître les priorités de toutes les tâches et groupes entre la racine et le niveau de hiérarchie de son groupe. La figure 3.1 donne un exemple de cette répartition sur une configuration simple, en détaillant les priorités de chacun et le temps processeur accordé en conséquence.



Process/Grp.	Priorité	Temps processeur (%)
A	2	50
B	1	5
C	1	5
D	10	25
E	4	7.5
F	4	7.5
G1	1	25
G2	1	25
G3	3	15

FIGURE 3.1 – Répartition du temps processeur dans une hiérarchie de groupes de processus avec priorités.

3.2.3 Ordonnancement Temps-réel

La norme POSIX définit en plus d'un ordonnanceur classique avec répartition équitable (`SCHED_OTHER`) deux types d'ordonnanceurs dits *temps-réel* : `SCHED_FIFO` et `SCHED_RR`. Ces deux ordonnanceurs reposent sur un contrat simple : toute tâche indiquée comme temps-réel ne peut être interrompue que par une tâche de même nature et de priorité au moins égale. Ce type de tâche peut donc s'exécuter sans craindre d'être perturbée par des processus de moindre importance. Ce type d'utilisation est très populaire pour la gestion des applications multimédia par exemple, afin de garantir une qualité de service constante.

La politique d'ordonnancement `SCHED_FIFO`, comme son nom l'indique, fournit le processeur à la première tâche prête. La tâche conserve ensuite le processeur tant qu'aucune autre tâche de plus haute priorité n'arrive. Il lui est néanmoins possible de relâcher le processeur volontairement via l'appel système `yield()`. Une telle politique est particulièrement utile pour des tâches ne devant pas être perturbées, et est utilisée à de multiples reprises dans cette thèse pour stabiliser les résultats expérimentaux. En effet, une telle politique d'ordonnancement permet à une tâche de s'abstraire de l'ordonnanceur, celui-ci ne s'activant qu'à la demande explicite de la tâche et de limiter par la même occasion le bruit système. Ce type d'utilisation comporte néanmoins des risques, puisqu'elle monopolise les processeurs, empêchant toute autre tâche de s'exécuter (y compris des services comme `ssh`). Il peut alors être difficile pour un utilisateur externe de reprendre le contrôle de la machine en cas d'erreur, celle-ci n'étant pas en mesure de répondre à la moindre requête.

L'autre politique (`SCHED_RR`) garantit à l'inverse que toutes les tâches au plus haut niveau de priorité actif obtiendront au bout d'un certain temps le processeur. Il s'agit donc essentiellement d'un temps partagé classique, mais pour lequel le quantum de temps alloué est de longueur fixée et est attribué à une tâche en une seule fois. Ce type d'ordonnanceur garantit donc que toute tâche obtiendra le processeur pendant une grande période de temps et sans pouvoir être interrompue. Cela est idéal pour des travaux sensibles aux perturbations, mais n'offre aucune assurance sur le temps séparant deux périodes d'exécution.

3.3 Modèle d'utilisation du processeur

La politique d'ordonnancement par répartition équitable du temps processeur est la plus utilisée dans le domaine du HPC. Cette dernière étant responsable du temps processeur qu'un processus sera capable d'obtenir, il peut être intéressant d'en tirer différentes métriques sur l'utilisation d'un cœur de calcul ou la performance d'une application.

3.3.1 Résolution de l'ordonnanceur

Comme nous l'avons déjà indiqué, un ordonnanceur ne peut répartir équitablement le temps processeur aux différentes tâches que sur une grande période de temps. Nous utilisons ce constat pour définir pour un ordonnanceur donné la notion de résolution :

Définition 4 *La résolution d'un ordonnanceur définit la période de temps minimale à considérer pour que sa politique de répartition atteigne son comportement asymptotique (ou du moins s'en approche suffisamment).*

Un ordonnanceur ne garantissant sa politique de répartition des ressources que de manière asymptotique, cette notion de résolution est capitale dans la compréhension du temps processeur obtenu par une application ou utilisé sur un processeur donné. Bien que peu de systèmes disposent de définition précise de cette résolution, il est considéré acceptable de la définir comme une période de 1 à 3 secondes, en fonction de la configuration de l'ordonnanceur et du nombre d'applications présentes sur le système.

3.3.2 Charge processeur

Une fois définie la résolution d'un ordonnanceur, nous pouvons nous intéresser à l'activité d'un processeur en particulier ou à sa charge :

Définition 5 *La charge absolue d'un processeur correspond à la proportion de temps durant laquelle il a été utilisé, sur une période de temps supérieure à la résolution de l'ordonnanceur. Formellement, si sur une période de temps mesurée en jiffies T_{tot}, le nombre de jiffies durant lesquels le processeur était utilisé par une application ou le système est T_u alors la charge du processeur vaut :*

$$charge\ absolue = \frac{T_u}{T_{tot}}$$

Nous appelons cette charge *absolue* par opposition à la charge du processeur *relative* à une application : le temps durant lequel le processeur a été utilisé par l'application en question sur une période de temps.

De la même façon il est possible de définir un profil de charge sur une période de temps de plusieurs fois la résolution de l'ordonnanceur :

Définition 6 *Le profil de charge d'un processeur sur un intervalle de temps est la fonction constante par morceaux représentant l'évolution temporelle de la charge de ce dernier. Par extension, le profil de charge de plusieurs processeurs est l'ensemble de leurs profils de charge.*

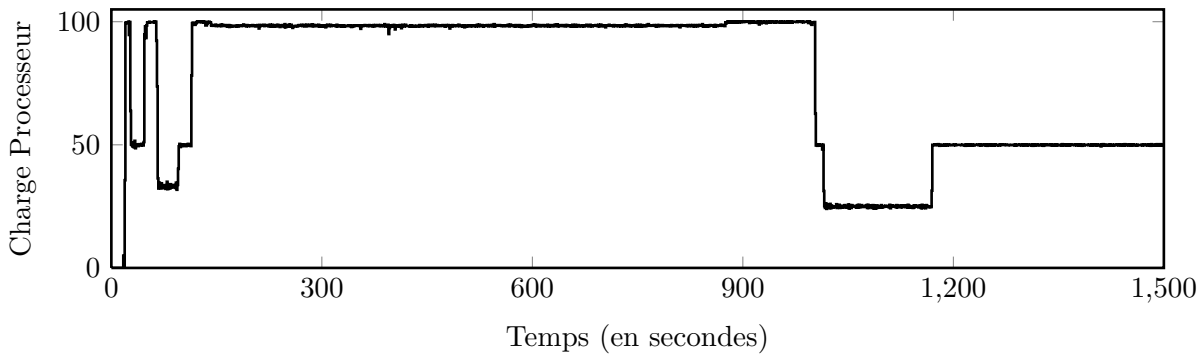


FIGURE 3.2 – Exemple de profil de charge.

3.3.3 Mesure de charge

Si les définitions précédentes pourraient nous permettre de mieux comprendre la répartition d'un processeur aux différentes applications en activité, encore faut-il pouvoir les mesurer. À ce titre, la plupart des systèmes d'exploitation fournissent des interfaces pour obtenir les quantités de temps processeur accordées aux différentes applications. Sur la plupart des systèmes UNIX, le système de fichier virtuel `procfs` fournit ces informations, sous la forme d'un fichier par processus affichant le nombre de *jiffies* attribués. C'est ce système de fichier qui est utilisé par des outils comme `ps`,`top`, ou `sysstat` [God08] pour mesurer l'utilisation processeur de chaque programme au cours du temps.

Pour obtenir cette mesure, ces outils nécessitent néanmoins l'exécution répétée de code sur la machine cible. En effet, le système de fichier ne fournit qu'une information brute : combien de *jiffies* au total a obtenu un processus. Pour calculer la charge processeur relative à un processus, il faut donc régulièrement relire cette information et ainsi comptabiliser le nombre de *jiffies* utilisés entre les deux lectures. Cette exécution répétée de code perturbe nécessairement la mesure effectuée, celle-ci utilisant du temps processeur. La plupart des outils proposent donc de configurer l'intervalle de temps séparant deux lectures et permettent donc une mesure à une fréquence supérieure à la résolution de l'ordonnanceur, ce qui donne de bons résultats. Un exemple de profil de charge est donné sur la figure 3.2.

3.4 Gestion de la mémoire virtuelle

Comme nous l'avons déjà expliqué, la mémoire virtuelle est un mécanisme d'abstraction de la mémoire aux nombreux avantages. Elle permet notamment à un processus d'utiliser plus de mémoire qu'il n'y a de RAM dans la machine ou d'isoler chaque application dans un espace d'adressage indépendant des autres. Pour fonctionner, ce mécanisme se repose sur une coopération étroite entre certains composants matériels et le système d'exploitation ayant pour but la distribution de la mémoire physique aux différentes applications. Bien évidemment, ce dispositif est fortement sollicité durant l'exécution d'un programme. Son implémentation a donc influence importante sur la performance des applications présentes sur le système.

3.4.1 Pagination

Pour gérer la distribution de la mémoire physique aux processus actifs sur la machine, le système d'exploitation utilise un mécanisme appelé pagination. Ce mécanisme repose sur un découpage de la mémoire (virtuelle et physique) en blocs continus de taille fixée : des pages. La distribution des pages physiques disponibles aux pages virtuelles dont ont besoin les processus actifs s'effectue en coopération avec un composant matériel bien spécifique : la MMU (pour *Memory Management Unit*).

On peut résumer cette coopération de la façon suivante. Chaque fois que le processeur accéde à une adresse virtuelle particulière, il demande à la MMU de traduire cette adresse en adresse physique. Pour cela, le matériel interroge automatiquement la table des pages du processus courant. La table des pages est une forme d'annuaire placé en mémoire et géré par le système d'exploitation. En parcourant celle-ci, la MMU trouve la page physique associée à l'adresse virtuelle demandée. Cette adresse physique est ensuite passée au bus mémoire pour effectuer le transfert mémoire.

Il arrive régulièrement que la table des pages ne contienne pas d'association avec une page physique pour l'adresse demandée. Dans ce cas, le matériel déclenche un déroutement du flux d'exécution du processeur et demande au système d'exploitation de satisfaire cette demande de mémoire physique. On parle alors de défaut de page.

Ces défauts de pages ont un coût important pour l'application. Premièrement, ils provoquent un déroutement pour donner la main au système d'exploitation. Deuxièmement, le gestionnaire de mémoire physique se doit de trouver un emplacement en mémoire pour la page virtuelle demandée, une opération potentiellement longue (voir sous-section suivante). C'est pour cela que le système d'exploitation comme les applications sont généralement optimisés pour déclencher le moins de défauts de page possible.

3.4.2 Gestion de la mémoire physique

Le gestionnaire de mémoire physique a donc la responsabilité au sein du système d'exploitation de répartir au mieux la mémoire disponible. On distingue généralement deux types de besoins en mémoire physique : le système d'exploitation lui-même, qui peut vouloir allouer dynamiquement de la mémoire pour ses structures internes ; et les applications qui s'exécutent sur la machine et dont les besoins apparaissent essentiellement à travers les défauts de page.

Nous ne détaillerons pas ici le gestionnaire de mémoire destiné au système d'exploitation lui-même. Il suffit de savoir que la mémoire allouée par ce biais est considérée comme prioritaire sur les applications et n'est donc jamais nettoyée de manière automatique (le système d'exploitation n'est pas sensé épuiser sa propre mémoire).

Pour ce qui est du gestionnaire destiné aux applications, il est composé de deux parties : un allocateur pour la mémoire libre et un nettoyeur automatique. Le système maintient en permanence une liste des pages physiques de la machine, afin de garder trace des propriétaires des pages occupées et de la liste des pages libres. Pour conserver en permanence une partie de la mémoire libre, le nettoyeur automatique se déclenche régulièrement pour vérifier l'état de la mémoire. Dans le cas où la mémoire libre vient à manquer, il choisit alors certaines pages pour les évincer. Ces pages sont alors sauvegardées sur disque si elles ont été écrites puis simplement effacées de la mémoire.

L'allocateur de pages libres se contente quand à lui de répondre aux requêtes lorsque le système créé un nouveau processus et lui alloue des pages ou lorsqu'un défaut de page est déclenché. Dans de rares cas (charge importante de la machine), cet allocateur peut aussi entraîner un nettoyage des pages occupées par d'autres processus, en suivant le même algorithme que précédemment. Différents algorithmes pour l'allocation et l'éviction mémoire sont apparus au cours du temps, nous pouvons citer notamment les *Buddy Systems* [Knu97] pour la première et LRU ou des algorithmes plus simples comme *Clock* ou *Seconde Chance* [TW97] pour l'éviction.

3.4.3 Coloration de page

Nous avons vu précédemment que les caches mémoires présents sur un processeur peuvent fonctionner soit sur des adresses physiques soit sur des adresses virtuelles. Dans le cas d'une indexation physique, la plus répandue dans les processeurs actuels, la façon dont le système d'exploitation associe aux pages virtuelles des pages physiques peut influer sur la capacité d'un processus à utiliser le cache.

Kessler *et al.* [KH92] ont montré qu'un gestionnaire de mémoire virtuelle contribuait jusqu'à 30% au nombre de défauts de cache dus à un conflit lors de l'exécution d'une application. Pour régler ce problème, ils ont proposé un certain nombre d'algorithmes, dont le plus célèbre reste la coloration de page. Cette algorithme est aujourd'hui implémenté dans de nombreux systèmes d'exploitation dont FreeBSD et Windows NT, afin d'améliorer la stabilité des applications et leur performance [KCS04, Dil].

La coloration de page identifie par une couleur le groupe de pages physiques recouvrant les mêmes lignes en cache. Ce recouvrement provient du fonctionnement des ensembles associatifs dans des caches indexés physiquement. En effet, les lignes mémoire étant réparties dans les différents ensembles associatifs de façon cyclique, et un cache ne disposant que de peu d'ensembles, un grand nombre de lignes mémoire se placent dans les mêmes ensembles associatifs. Une page n'étant constituée que d'un nombre entier de lignes mémoire consécutives, un grand nombre de pages occupent les mêmes ensembles. On identifie alors par une couleur ce groupe de pages, mais aussi les ensembles associatifs qu'elles utilisent. Ainsi, seul un accès mémoire à une page d'une certaine couleur peut être mis en cache dans un ensemble associatif de la même couleur.

Par souci de clarté, nous définissons C comme la taille d'un cache, A son nombre de voies (ou associativité), L la taille de ses lignes et P la taille d'une page. Un cache possédant $\frac{C}{AL}$ ensembles associatifs, et une page occupant $\frac{P}{L}$ lignes de cache, le nombre de couleurs dans un cache est de $\frac{C}{AP}$. La figure 3.3 illustre cette association page couleur sur un cache hypothétique avec 8 ensembles associatifs de 8 voies et des pages de 2 lignes.

3.4.4 Gestion d'une mémoire virtuelle avec coloration

Un système d'exploitation prenant en compte la coloration de page lors de l'allocation de pages physiques à un processus réalise généralement un compromis entre deux critères quelque peu antagonistes.

Premièrement, afin d'optimiser le cache utilisable par une application, les pages virtuelles consécutives d'un processus doivent recevoir des couleurs différentes. Bien

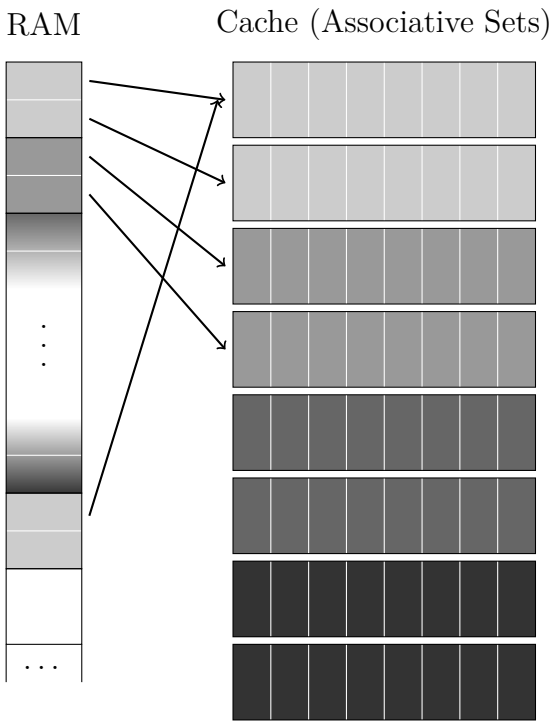


FIGURE 3.3 – Coloration de page sur un système hypothétique avec des pages de 2 lignes, et un cache à 4 couleurs associatif 8 voies. Chaque ligne d'une page occupe un ensemble associatif différent, mais les pages d'une même couleur occupent les mêmes ensembles.

sûr, le nombre de couleurs dans un système est trop faible pour généralement pouvoir donner une couleur différente à chacune des pages virtuelles d'un processus. Cela dit, le gestionnaire mémoire essaye de prendre en compte les couleurs actuellement utilisées par le processus pour équilibrer son allocation entre toutes celles existantes. Bien que cette répartition ne soit pas nécessairement la plus adéquate vis à vis de certains types d'accès mémoire (avec de grands sauts) elle reste efficace dans la plupart des cas [KH92].

Deuxièmement, le système tente de répartir les couleurs équitablement entre des processus s'exécutant sur des cœurs partageant un même cache en donnant, si possible, des couleurs différentes à chacun des processus. En effet, de nombreux travaux ont souligné ces dernières années l'importance de la coloration de page dans l'utilisation équitable d'un cache partagé [LLD+08,Iye04]. La problématique peut se résumer ainsi : si deux processus partagent les mêmes couleurs et s'exécutent sur des cœurs partageant un cache indexé physiquement, alors ils se gêneront mutuellement en déclenchant un excès de défauts de cache de conflit. Différentes solutions ont pu être apportées à ce problème parmi lesquelles un support matériel ou des politiques de gestion de page plus complexes restent les plus populaires (voir chapitre 5).

De manière générale, un contrôle des couleurs associées aux pages virtuelles d'un processus offre de nombreuses possibilités en terme d'optimisation des performances en cache. Il est ainsi possible de limiter la quantité de cache utilisée par des structures de données sans localité ou de limiter l'impact en cache d'un processus sur un autre, et cela de manière transparente, l'application n'ayant pas conscience d'un tel système.

Émuler l'indisponibilité du processeur 4

Sommaire

4.1 Travaux existants 38
4.1.1 Ordonnancement temps-réel 39
4.1.2 Signaux 40
4.1.3 Changement de fréquence du processeur 41
4.2 Générer une charge en coopérant avec le système 42
4.2.1 Partage du temps processeur entre applications 42
4.2.2 Méthodologie retenue 43
4.3 KRASH : une implémentation sous Linux 43
4.3.1 Gestion des groupes sous Linux 44
4.3.2 Mécanisme de supervision 45
4.3.3 Classes d'évènements rencontrés 46
4.3.4 Gestion de la dérive 47
4.3.5 Profil de charge 48
4.4 Évaluation de KRASH 49
4.4.1 Validation expérimentale 50
4.4.2 Comparaison aux autres solutions 52
4.5 Utilisation et extensibilité de la méthode 56
4.5.1 Validation d'applications avec vol de travail 56
4.5.2 Extension aux autres ressources 58

Certaines applications parallèles, lorsqu'elles sont exécutées sur des machines à mémoire partagée, répartissent dynamiquement leur travail au cours du temps. Une telle répartition permet à ces applications d'atteindre de meilleures performances dans des environnements partagés : si un autre utilisateur est présent sur le système ou qu'une autre application utilise une partie des cœurs, une répartition dynamique du travail permettra d'utiliser au mieux les ressources restantes [ABP98, BP98].

Pour tester ce type d'applications, ou réaliser des expériences sur de nouveaux algorithmes de répartition, il peut donc être intéressant de reproduire sur une machine dédiée un déséquilibre dans la répartition du temps processeur. La façon dont une application réagira, ou non, à ce déséquilibre pouvant nous en apprendre sur sa performance et son adaptabilité.

Nous définissons ce déséquilibre comme une charge processeur appliquée à la machine. Cela correspond au fonctionnement normal du système : dans un environnement

où plusieurs applications s'exécutent, l'ordonnanceur est responsable de la répartition effective du processeur à ces dernières. Induire un déséquilibre dans cette répartition revient donc à placer sur le système une charge processeur, de façon à soustraire une partie du temps processeur à d'autres applications. Cette notion de charge appliquée nous permet aussi de définit la qualité de notre dispositif : il doit reproduire les effets d'une application supplémentaire présente sur le système.

Il est néanmoins capital, pour que les résultats expérimentaux tirés d'un tel dispositif puissent être interprétés correctement et reproduits dans un environnement de production, que l'environnement d'expérimentation respecte certains critères. Premièrement, la charge reproduite dans le système doit avoir la même résolution que dans un environnement de production. Deuxièmement, le procédé utilisé pour reproduire cette charge doit être le moins intrusif possible et idéalement, le comportement de l'ordonnanceur système ne doit pas être modifié. Un tel objectif garantit aussi la reproductibilité des résultats dans un environnement de production puisque le système d'exploitation se comportera de la même façon. Troisièmement, la charge processeur produite doit être la plus flexible possible, l'objectif étant de pouvoir reproduire le plus de situations possible. Enfin, ce dispositif doit pouvoir fonctionner quelque soit l'environnement, en terme de charge, de la machine. Autrement dit, quelque soit le nombre d'applications présentes sur le système, la charge induite souhaitée doit pouvoir être atteinte.

Ce chapitre présente donc notre contribution sur ce problème : une méthode performante afin d'obtenir sur une machine à mémoire partagée une charge processeur, et cela de façon reproductible. Après avoir discuté des travaux existants sur la question et de leurs défauts, nous détaillerons la méthodologie que nous avons développé et son implémentation sous Linux. Nous finirons par la validation de cette implémentation et quelques exemples d'utilisation.

4.1 Travaux existants

Divers mécanismes de contrôle des applications ou du matériel permettent de réaliser une charge processeur. Dans la mesure où nous sommes intéressés ici essentiellement par des charges partielles (il reste du temps processeur pour une application cible), nous ne détaillerons pas les applications du domaine des *benchmarks*. En effet, les *benchmarks* processeur tel que Whetstone [CW76], Dhrystone [Wei84], SPEC CPU [Hen06] ou LINPACK [DLP03] visent à mesurer la performance (de crète ou soutenue) d'un système face à une application particulière dans le but de comparer les systèmes entre eux. Bien que ces applications génèrent une charge importante sur la machine, celle-ci n'est pas contrôlée et n'a donc que peu d'intérêt pour notre objectif de reproduction d'un profil de charge.

Un ensemble de travaux en particulier nous intéressera ici : le projet Wrekavoc [CJ06a, ODJ09, CDGJ10]. Les différentes publications liées à ce projet ont en effet couvert un large panel des solutions de génération de charge processeur. Wrekavoc est un outil développé au sein de l'équipe AlGorille (INRIA Loria) pour permettre l'émulation, sur un ensemble de machines dédiées, d'un système hétérogène. Cette reproduction d'hétérogénéité ne se limite pas au processeur : les performances réseau sont notamment au cœur des préoccupations du projet. Concernant la reproduction d'une hétérogénéité

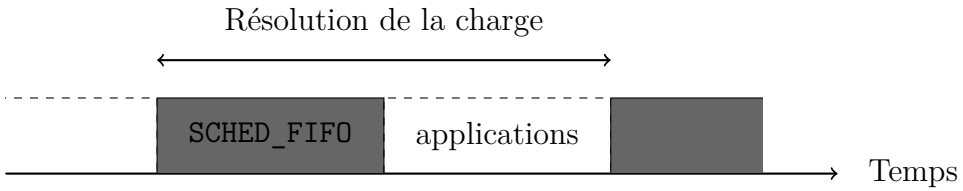


FIGURE 4.1 – Répartition du temps processeur pour une charge de 50% infligée à l'aide d'un processus `SCHED_FIFO` simple.

processeur, Wrekavoc permet en théorie d'imiter, du point de vue d'une application, des processeurs de plus faible puissance que ceux présents sur les machines cibles. Cet objectif d'imitation ne rejoint pas exactement le nôtre : les différents auteurs de Wrekavoc considèrent qu'une charge doit reproduire un processeur de fréquence inférieure alors que nous la considérons comme la reproduction d'une répartition du temps processeur à différentes applications. Bien que cette différence puisse sembler mineure, nous montrerons plus tard que sur certaines expériences, nos conclusions divergent significativement.

4.1.1 Ordonnancement temps-réel

Une première technique permettant de créer une charge artificielle sur une machine concerne la politique d'ordonnancement `SCHED_FIFO`. En effet, cette dernière permet à un processus de monopoliser le processeur aussi longtemps que voulu. Pour réaliser n'importe quelle charge sur un processeur, il suffit donc qu'un processus ordonnancé en temps-réel alterne entre des phases d'activité (une boucle infinie par exemple) et des phases de sommeil, pour permettre aux autres processus d'utiliser le temps processeur restant.

La difficulté majeure de cette technique tient dans la distribution du temps processeur aux autres applications. Le cas le plus extrême survient lorsque la charge est appliquée en une seule fois : le processus de charge monopolise le processeur par grandes phases (équivalentes à la moitié de la résolution), une fois par période de temps considérée. Ce procédé est résumé en figure 4.1. Cette solution ne satisfait malheureusement pas nos contraintes : les applications réalisant de nombreuses entrées/sorties sont parfois incapables d'utiliser le temps processeur restant en entier, et se retrouvent à percevoir une charge supérieure à celle voulue.

Il est donc nécessaire pour obtenir une charge de qualité à l'aide de cette technique que le processus de charge alterne bien plus d'une fois entre activité et sommeil sur la période d'une résolution. Ce comportement reproduirait plus fidèlement le fonctionnement normal du système. C'est cette solution qui fut retenue durant un certain temps par les auteurs de Wrekavoc. Le processus de charge, après une phase de calibration afin de déterminer un temps d'exécution adéquat pour les périodes d'activité, alternait de manière systématique entre exécution et sommeil. Deux problèmes techniques rendent néanmoins cette solution peut enviable : les mécanismes du système d'exploitation lui-même permettant à un processus de s'endormir sont implémentés de façon à ce qu'une demande de sommeil trop courte ne soit pas satisfaite (le processus reste en attente active sur l'horloge au lieu de bloquer). Cela empêche la mise en place de

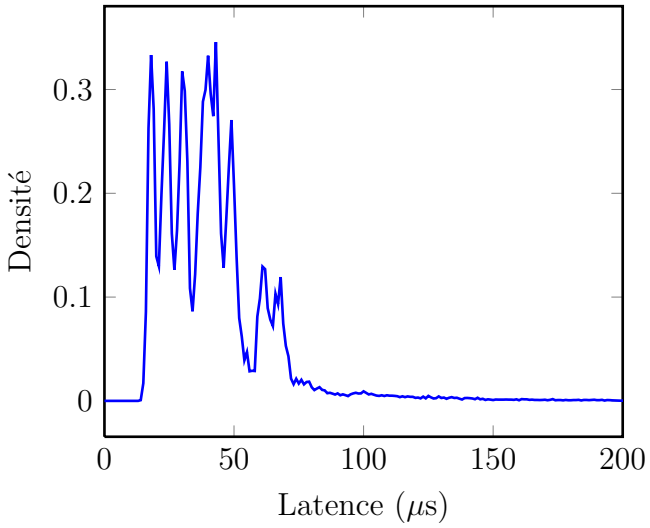


FIGURE 4.2 – Latences mesurées par `cyclictest` sur des périodes de sommeil de 1ms, en priorité temps-réel et à l'aide de `clock_nanosleep`.

charges trop faibles. De plus, la précision des opérations d'endormissement et de réveil d'un processus est très mauvaise. Il en résulte une charge de mauvaise précision, ce qui ici encore a des répercussions sur certaines applications.

Les développeurs du noyau Linux fournissent quelques outils permettant d'évaluer la précision de certains mécanismes du noyau tels que les signaux POSIX, les sémaphores ou les méthodes d'endormissement. Pour illustrer la précision de ces dernières, nous donnons en figure 4.2 un histogramme des latences mesurées sur une demande d'endormissement de 1ms, répétée 10000 fois sur une machine dédiée, en priorité `SCHED_FIFO`. Notons que le noyau ne fournit aucune garantie sur les changements de contexte associés : il n'est absolument pas certain qu'une autre application puisse s'exécuter durant ces périodes de sommeil.

Les auteurs de Wrekavoc ont d'ailleurs admis que l'utilisation des priorités temps-réel posaient certains problèmes pour la précision et les effets de bords de la charge obtenue [ODJ09], notamment vis à vis des applications utilisant le réseau, et nous confirmerons ces résultats plus tard vis à vis des entrées/sorties.

4.1.2 Signaux

La deuxième technique mise en œuvre par Wrekavoc utilise les signaux POSIX. Ces signaux permettent entre autre d'exercer une forme de contrôle d'un processus vers d'autres. Deux signaux sont utilisés ici, *SIGSTOP* qui stoppe un processus, et *SIGCONT* qui en redémarre un.

La méthode utilisée par Wrekavoc repose sur un processus superviseur, inspectant de manière régulière le système de fichier `/proc` pour connaître la liste des processus actifs et le temps processeur qu'ils ont utilisé (à la manière de `top`). Ce superviseur envoie ensuite des signaux aux différents processus afin de les stopper lorsqu'ils dépassent le temps processeur autorisé. Afin d'optimiser cette méthode, Wrekavoc créé en réalité autant de superviseurs que de processus actifs sur le système, et les place en priorité temps-réel.

Cette méthode n'applique tout simplement pas une charge processeur à une machine. Si chaque processus se voit bien dans l'incapacité d'utiliser plus que la limite autorisée de temps processeur, chaque processeur du système peut en revanche être, en fin de compte, utilisé complètement : il suffit que l'application cible créé plus de processus qu'il n'y a de processeurs dans le système. Prenons par exemple le cas de deux processus s'exécutant sur un même processeur : charger la machine à 50% avec cette méthode revient à stopper les processus s'il reçoivent plus de la moitié du temps processeur. Cela ne changera rien à leur performance, puisqu'ils occupaient déjà le processeur à ce niveau. Pourtant, avec une charge de 50 % sur la machine, on s'attendrait à ce que les processus restants ne reçoivent plus que 25% du processeur pour chacun.

4.1.3 Changement de fréquence du processeur

La plupart des processeurs d'aujourd'hui possèdent des mécanismes de changement de fréquence (aussi appelés *Dynamic Frequency Scaling*). Ce type de technologie est surtout présente pour réduire la consommation énergétique des processeurs lorsqu'ils sont sous-utilisés.

Le nombre d'instructions qu'un processeur est capable d'exécuter par seconde est directement lié à sa fréquence. Nous pouvons donc considérer qu'il s'agit d'un mécanisme de charge : un processeur dont la fréquence est diminuée de moitié exécute moitié moins d'instructions, et correspond donc à un processeur chargé à 50%. Cette technique a déjà été utilisée par exemple dans des études sur l'influence du processeur sur la performance réseau d'une machine [SM03].

Malheureusement, cette méthode comporte de nombreux désavantages. Tout d'abord, le nombre de niveaux auxquels il est possible de baisser la fréquence d'un processeur est très limité. La plupart des constructeurs n'implémentent que quelques paliers, et leur nombre dépend fortement du modèle de processeur considéré. La plage de charges processeur qu'il est donc possible de simuler est assez étroite. Notons aussi que jusqu'à très récemment, tous les cœurs d'un processeur devaient être placés à la même fréquence, il était donc impossible d'utiliser une telle méthode si la charge à reproduire n'était pas la même sur chaque cœur.

Enfin, on peut s'interroger sur l'équivalence entre baisser la fréquence d'un processeur et limiter le temps processeur disponible. En effet, le changement de fréquence ne concerne que le processeur, pas la vitesse d'accès à la RAM ni celle des disques. En conséquence, un programme s'exécutant sur une fréquence de processeur inférieure peut se comporter comme si la mémoire était plus rapide. Cela change aussi la façon dont les communications et le calcul se superposent, et donc le comportement général des applications présentes sur le système. Nous montrerons dans la suite que cette méthode engendre des différences notables dans le comportement des applications par rapport à une charge processeur classique.

Notons que la dernière version de Wrekavoc utilise cette technique, en alternant entre plusieurs fréquences régulièrement pour simuler une fréquence donnée s'il n'existe pas de palier à ce niveau.

4.2 Générer une charge en coopérant avec le système

Les principaux défauts des différentes techniques présentées précédemment proviennent de leur incapacité à préserver le comportement normal du système d'exploitation lorsque plusieurs applications sont en concurrence (exactement ce que l'on cherche à reproduire). Pour mieux comprendre le problème et les solutions envisageables, nous nous attardons ici au comportement exact de quelques classes d'applications dans ce type de conditions.

4.2.1 Partage du temps processeur entre applications

Nous l'avons déjà dit, l'ordonnanceur système cherche principalement à distribuer équitablement le processeur au cours du temps. Il découpe donc le temps en intervalles d'utilisation exclusive et répartit ces intervalles aux différentes applications. L'objectif d'équité est alors atteint sur une longue période de temps, que nous avons appelé résolution. Intéressons nous plus précisément au cas où deux applications s'exécutent en concurrence.

Si les deux applications réalisent uniquement du calcul, elles sont actives durant toute leur exécution et peuvent donc se répartir les *jiffies* de façon cyclique. Nous donnons en figure 4.3 un schéma de cette répartition sur la période d'une résolution. Pour reproduire un tel comportement face à une seule application ne réalisant que du calcul, il suffit donc d'exécuter un simple processus en concurrence. Bien sûr, si plus de processus sont en exécution un tel système ne convient pas, mais l'idée reste la même : il faut occuper une certaine proportion de *jiffies* au cours du temps.

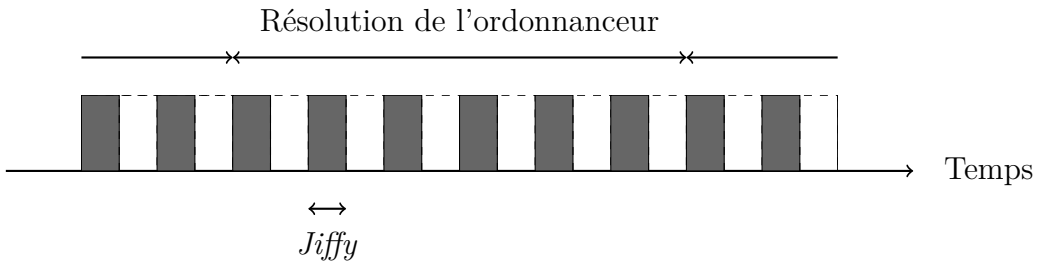


FIGURE 4.3 – Répartition du temps processeur entre deux processus ne réalisant que des calculs.

Il n'en reste pas moins que la plupart des applications ne réalisent pas uniquement du calcul, mais effectuent aussi des entrées/sorties ou des opérations de synchronisation. Ces opérations bloquent alors les processus et empêchent l'ordonnanceur de leur attribuer des *jiffies* (et terminent par ailleurs les périodes d'utilisation du processeur en cours). Pour équilibrer alors le temps processeur, les politiques d'ordonnancement sont configurées pour attribuer plus souvent un *jiffy* à ces applications, de façon à ce qu'elles obtiennent plus de *jiffies* à leur réveil. La répartition obtenue alors est illustrée en figure 4.4.

Reproduire une telle répartition des *jiffies* s'avère complexe : toute stratégie visant à court-circuiter l'ordonnanceur risque de perturber les optimisations mises en place

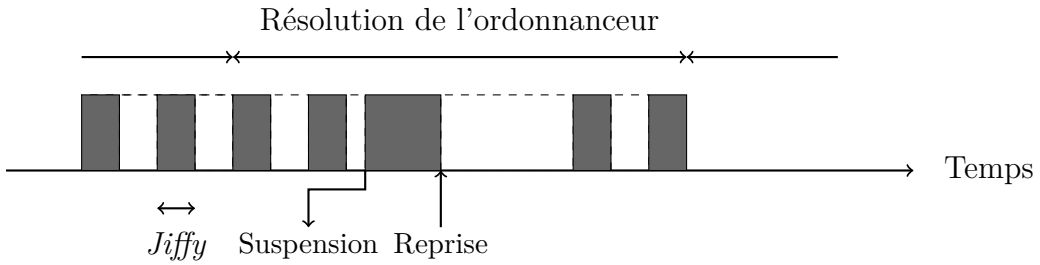


FIGURE 4.4 – Répartition du temps processeur entre un processus de calcul et un processus réalisant des entrées/sorties. Pour compenser la période de suspension, le processus se voit accorder plus de temps processeur à la reprise.

par ce dernier et une modification des conditions matérielles entraîne nécessairement une modification de la réactivité des applications et donc de la distribution des *jiffies* dans le temps.

4.2.2 Méthodologie retenue

Nous cherchons donc une méthode capable d'occuper autant de temps processeur que voulu tout en laissant l'ordonnanceur système fonctionner normalement. De plus, la charge doit pouvoir varier dynamiquement, afin de simuler par exemple l'arrivée et le départ d'applications. Au vu des problèmes rencontrés par les autres méthodes, il apparaît intéressant d'utiliser directement les mécanismes d'ordonnancement système pour arriver à nos fins.

Deux procédés doivent donc être mis en place : un superviseur et un contrôle de l'ordonnanceur. Le superviseur aura pour mission de surveiller la charge actuelle générée et de la modifier en fonction des conditions de la machine et du profil de charge qu'il faut reproduire, et cela indépendamment d'un processeur à l'autre. Le mécanisme de contrôle de l'ordonnanceur doit faire en sorte d'utiliser une portion du temps processeur (fonction de la charge) pour lui-même.

4.3 KRASH : une implémentation sous Linux

Malheureusement, l'ordonnanceur standard d'un système d'exploitation (même avec priorités) interdit la création d'une charge trop faible ou trop forte : tous les processus obtiennent quoiqu'il en soit du temps processeur. De plus, le superviseur peut difficilement agir à la résolution de l'ordonnanceur de peur de dégrader fortement la performance des autres applications.

L'ordonnancement par groupe, par contre, nous donne une solution élégante à tous ces problèmes. En effet, dans le cadre des groupes de processus, il est parfaitement possible de placer un processus unique comme demandant autant de temps processeur que tous les autres processus présents dans le système : il suffit qu'il soit dans un autre groupe et au même niveau de la hiérarchie. Par un tel mécanisme, l'ordonnanceur système lui-même nous garantit alors une répartition du temps processeur adéquate et à la bonne résolution.

Notre processus de charge se résume alors aux étapes suivantes. Premièrement, placer tous les processus présents sur le système dans un seul et même groupe (que nous appellerons *base*), au niveau 1 de la hiérarchie (sous le groupe racine). Créer ensuite au même niveau un groupe par processeur cible et y placer un processus occupant systématiquement tout le temps processeur disponible (appelons le *poids*). Chacun de ces groupes est alors restreint au processeur correspondant. Dès lors, l'ordonnanceur système considère qu'il n'existe que deux groupes actifs par processeur (la base et le poids), et donc chacun reçoit la moitié du temps CPU. Enfin, un processus de supervision, se réveille lorsque nécessaire (changement de charge) pour ajuster la priorité du groupe de chaque poids.

Ce calcul de la priorité à appliquer est très simple : le temps processeur que reçoit un groupe ne dépend que de sa propre priorité et de la somme des priorités de tous les autres groupes actifs sur le processeur. Dans notre configuration les seuls groupes actifs sont *base* et le groupe *poids*, et nous cherchons à partir d'un niveau de charge, la priorité du groupe poids à appliquer :

$$charge = \frac{poids_{prio}}{poids_{prio} + base_{prio}}$$

D'où le résultat suivant :

$$poids_{prio} = \frac{base_{prio}}{1 - \text{charge}} \times \text{charge}$$

Afin de valider cette méthodologie, nous l'avons implémentée sous Linux. Le noyau Linux est en effet l'un des rares systèmes d'exploitation à disposer d'une gestion complète des groupes de processus et de l'ordonnanceur par groupes associé. Cette technologie, assez récente dans son implémentation (la première version remonte à 2007), porte le nom de `control groups`.

Notre implémentation, en C++, porte le nom de KRASH et permet donc de reproduire une charge processeur hétérogène (différente sur chaque cœur) et dynamique sur un système Linux. Nous détaillons dans la suite les différents composants de cette implémentation, de la gestion des groupes à la structure du superviseur en passant par les technicités des différentes phases de fonctionnement de l'outil et le format choisi pour décrire le profil de charge à reproduire.

4.3.1 Gestion des groupes sous Linux

L'interface principale pour la gestion des groupes sous Linux repose sur des fichiers virtuels. Chaque dossier représente ainsi un groupe, contenant un certain nombre de fichiers correspondants aux propriétés du groupe qu'il est possible de contrôler. En fonction du fichier considéré, le noyau redéfinit la sémantique de deux opérations : la lecture et l'écriture. Ainsi, lorsque le fichier `cpu.shares` est lu, la priorité du groupe correspondant est donnée comme un entier positif. L'écriture d'un entier dans ce fichier permet quand à elle de donner une nouvelle valeur à cette priorité. Mais dans le cas du fichier `tasks`, l'écriture d'un entier est interprétée comme l'identifiant d'un processus qu'il faut placer dans ce groupe. Un processus est ainsi migré d'un groupe à l'autre en écrivant son identifiant dans le fichier `tasks` du groupe destination.

Certains développeurs du noyau Linux participent à l'écriture d'une interface de plus haut niveau, la bibliothèque `libcgroup`. Il s'agit d'une bibliothèque C permettant de créer de manière transparente un groupe et d'interroger ses différentes propriétés (priorité, permissions, processus contenus). Cette interface possède néanmoins un défaut majeur : la lecture des propriétés est réalisée en une seule fois, puis mémorisée en interne tant que l'utilisateur ne demande pas de mise à jour explicitement. Les valeurs ainsi retournées lors de l'interrogation des propriétés d'un groupe peuvent donc ne pas être à jour par rapport au système de fichier (ce dernier pouvant être modifié sans utiliser la bibliothèque). L'utilisation de cette bibliothèque simplifiant grandement l'interaction avec le système de fichier, notamment au niveau de la détection de la hiérarchie présente sur le système,KRASH l'utilise pour agir sur les groupes.

Notons tout de même deux problèmes liés à l'utilisation d'un système de fichier virtuel comme interface pour la gestion des groupes. Premièrement, une telle interface ne permet pas d'effectuer d'opérations atomiques : la lecture de la liste des processus d'un groupe par exemple peut retourner des identifiants de processus ayant disparus dans le temps nécessaire à la complétion de l'opération. Cela peut être particulièrement gênant lors de la migration des tâches d'un groupe à un autre, puisque le procédé peut obtenir des identifiants de processus qu'il n'est pas possible d'écrire dans la nouvelle liste sans déclencher d'erreur. Deuxièmement, certains processus particuliers ne peuvent être migrés en dehors du groupe racine sous Linux. Il s'agit des processus liés directement au noyau et généralement chargés d'opérations sensibles telles que la gestion du cache disque (*swap*), ou une partie de la pile réseau mais n'occupant néanmoins que peu de temps processeur en général. Une erreur est donc déclenchée lorsqu'une application tente de migrer ces processus vers un autre groupe que la racine. Notre méthode de génération de charge démarrant par la migration de tous les processus utilisateurs dans le groupe base, KRASH est conçu avec pour utiliser les groupes avec une certaine souplesse (interprétation des erreurs, tentatives multiples sur une opération par exemple).

4.3.2 Mécanisme de supervision

Le processus superviseur est responsable de la majeure partie du fonctionnement de notre mécanisme de charge. Son comportement peut se résumer à quatre phases : la prise en compte du profil de charge à reproduire, la configuration du système, la phase de supervision à proprement parler et le nettoyage du système.

Au vu de la méthodologie retenue, un profil de charge se résume à une liste de dates pour lesquelles il est nécessaire de modifier la priorité d'un des groupes contenant les poids. Il est en effet inutile pour le superviseur d'être en activité durant toute la période de charge, l'ordonnanceur système garantissant lui-même la charge infligée à partir des priorités en place. Le superviseur mémorise donc une liste d'actions, composées d'une date et de la charge à infliger, lors de sa première phase. Cette liste est bien sûr ordonnée par rapport aux dates, et chaque action est spécifique à un processeur en particulier.

La phase de configuration correspond aux premières étapes de notre méthodologie de génération de charge. Tous les processus actifs sur le système sont donc placés dans un nouveau groupe (base) situé sous le groupe racine. Un poids et son groupe sont alors créés et verrouillés (avec les `cpusets`) sur chacun des processeurs. À ce niveau là

de l'installation, les processus chargés sont donc opposés sur chaque processeur à un seul autre groupe, au même niveau de la hiérarchie.

La phase de supervision commence alors. Le superviseur inspecte la liste des actions pour déterminer si une charge doit être appliquée (on peut considérer que sur chaque processeur une action existe à la date 0), exécute les opérations nécessaires (modification de la priorité du groupe poids sur le processeur correspondant) puis s'endort jusqu'à la date de la prochaine action. Il s'agit donc d'un procédé évènementiel, le superviseur n'agissant que lorsqu'un évènement telle qu'une nouvelle action requière son attention. Cette phase ne s'arrête qu'à la fin du profil de charge.

La phase de nettoyage consiste simplement à éliminer les poids du système, détruire les groupes correspondants et replacer les processus présents dans la base au niveau de la racine. Notons au passage que de nouveaux processus ont très bien pu apparaître durant l'exécution de notre mécanisme de charge, et qu'il faut donc bien déplacer tous les processus présents et pas seulement ceux déplacés initialement. La base peut alors être détruite afin de replacer le système dans la configuration initiale.

4.3.3 Classes d'évènements rencontrés

Le superviseur fonctionne globalement comme un système évènementiel : il n'agit que lorsqu'une action est à réaliser ou qu'un évènement particulier a lieu sur la machine chargée. Dans cette implémentation, un certain nombre d'évènements peuvent donc venir activer le superviseur. Nous détaillons ici leur type et la réaction de ce dernier.

Changement de charge : il s'agit là de l'action la plus courante. Lorsque la charge d'un processeur doit être modifiée, le superviseur est réveillé et doit alors calculer la nouvelle priorité du groupe concerné. En utilisant les informations sur les groupes conservées en mémoire, ce calcul est extrêmement rapide. La nouvelle priorité est alors mise à jour sur le système et le superviseur se remet en attente.

Terminaison : le profil de charge peut contenir une action spéciale (`kill`) servant de marqueur de fin de profil. Ce dernier déclenche la terminaison de l'outil à une date relative au démarrage. Cet évènement déclenche donc le nettoyage complet du système et l'arrêt de l'outil.

Modifications des groupes : il est techniquement possible qu'un utilisateur créé, détruise ou modifie un des groupes présents sur le système durant l'exécution de notre outil. À cet effet, KRASH active au démarrage un certain nombre de mécanismes de surveillance du système de fichier virtuel. Ainsi, les modifications apportées par un utilisateur seront prises en compte dans la suite du fonctionnement de l'outil. Certains de ces évènements sont néanmoins critiques, comme la destruction d'un groupe poids, et l'outil déclenche donc son arrêt total dans ces cas là.

Erreurs diverses : il est aussi possible qu'un utilisateur supprime directement un des processus poids. Cet type d'évènement est capturé par le superviseur et déclenche aussi la terminaison de l'outil.

Contrôles au démarrage : certaines versions du noyau Linux, notamment les premières ayant implémenté l'ordonnancement de groupe, contenaient des erreurs importantes et rendaient KRASH inefficace. Pour permettre aux utilisateurs de détecter de telles erreurs, le superviseur peut déclencher au démarrage une phase de test. Durant cette phase, le superviseur vérifie qu'un processus sous son contrôle perçoit le bon niveau de charge sur une période de quelques secondes. La charge à tester est choisie de manière arbitraire, mais différente de 50% (valeur qu'il est possible d'obtenir même avec un ordonnanceur invalide). Pour obtenir une telle fonctionnalité, certains évènements particuliers ont donc été ajoutés (démarrage et arrêt de la charge test, lancement de l'application test). Ces tests comprennent aussi une phase de vérification de l'interface de modification des groupes notamment au niveau du temps nécessaire au changement de leur priorité.

Tous ces évènements, et les actions qui en découlent, constituent l'essentiel du travail du superviseur. Bien qu'ils soient tous très différents dans leur nature, la bibliothèque utilisée pour la gestion de la boucle évènementielle `libev` nous permet de les gérer de manière transparente, par l'installation de *callbacks* (codes dédiés) appelés directement à chaque évènement. Cette bibliothèque fournit une interface simple, portable et de qualité, notamment en terme de passage à l'échelle (grand nombre d'évènements) [Leh].

4.3.4 Gestion de la dérive

Indépendamment de la qualité de la boucle évènementielle, la performance de KRASH dépend fortement de sa capacité à exécuter rapidement les différentes actions nécessaires lors de son fonctionnement. En effet, le nombre d'opérations à réaliser croît fortement avec le nombre de processeurs sur la machine et la précision du profil de charge à reproduire. Ainsi, si KRASH génère une charge variant toutes les secondes sur 10 processeurs il doit modifier la priorité de 10 groupes par seconde.

Malheureusement, la modification de la priorité d'un groupe est une opération relativement longue. L'implémentation des groupes sous Linux est telle qu'un utilisateur malicieux pourrait déclencher des attaques de type *déni de service* en modifiant trop rapidement les priorités des groupes. Pour limiter ce type d'attaques, le système d'exploitation limite l'intervalle de temps séparant deux accès aux priorités (le seuil est fixé à 0.25ms par défaut) en ignorant toute modification durant cet intervalle. Pour garantir la prise en compte de la nouvelle charge, KRASH réécrit donc la nouvelle priorité d'un groupe jusqu'à ce que le système la retourne en lecture. Cette vérification compense aussi les erreurs présentes dans certaines versions du noyau.

La conséquence directe de ces problèmes et des vérifications effectuées par KRASH est l'apparition d'un phénomène de dérive d'horloge : lorsque le superviseur se réveille pour réaliser une action, la date est naturellement dépassée de quelques millisecondes, auxquelles s'ajouteront tous les retards dus aux vérifications successives, entraînant jusqu'à plusieurs secondes de décalage au total. Pour compenser un tel phénomène, il est nécessaire que le superviseur rattrape le retard accumulé. Ainsi, à chaque réveil, plutôt que de traiter uniquement l'évènement ayant déclenché ce réveil, KRASH inspecte la liste des actions et considère toutes celles qui auraient dues être réalisées.

Bien entendu, il est parfaitement inutile de modifier la charge d'un seul processeur deux fois d'affilée lors de l'activation du superviseur. Ce dernier effectue donc un filtrage

```
cgroup_root = /
all_name = alltasks
cpu {
        profile {
                0 {
                        0         70
                        60        30
                        120       50
                }
                1 {
                        0         70
                        10        30
                }
        }
}
kill 150
```

FIGURE 4.5 – Exemple de profil de charge KRASH.

des actions considérées, de façon à ne conserver qu'une seule modification de charge par processeur : la plus proche dans le temps. Ce choix n'est pas sans conséquences, puisqu'il arrive alors que certains niveaux de charge dans un profil dynamique soient complètement ignorés dans la réalité. Néanmoins, cette solution est celle qui permet de respecter au mieux le profil de charge original, quelque soit l'utilisation qu'en fait un expérimentateur.

Notons aussi que sur des profils raisonnables (avec des intervalles de temps de plusieurs secondes entre deux changements de charge) ce type de phénomène est extrêmement rare.

4.3.5 Profil de charge

Pour que KRASH puisse reproduire une charge, il est nécessaire que l'utilisateur lui fournisse un profil. Ce dernier prend la forme d'un fichier en entrée du programme, respectant un certain format. Comme nous cherchons à reproduire une charge avec la même résolution que l'ordonnanceur Linux, notre profil de charge ne permet de modifier cette dernière qu'une seule fois par seconde au maximum par processeur. Au vu des problèmes mentionnés précédemment, cette résolution reste raisonnable.

Le format de fichier utilisé est très simple, ayant pour but de décrire la fonction constante par morceau qu'est la charge d'un processeur. Il s'agit donc essentiellement d'une liste de dates (relatives au démarrage de l'outil) où la charge doit être modifiée. Par exemple, si pour le processeur 0 à la date 0 la charge est de 50% et à la date 10 la charge est de 30%, alors KRASH occupera 50% du temps processeur pour les dix premières secondes suivant son démarrage, puis 30% jusqu'à son arrêt.

Un exemple simple de profil est donné en figure 4.5. Si nous détaillons cet exemple ligne par ligne, nous obtenons les informations suivantes :

– `cgroup_root` nous donne le chemin, depuis la racine du système de fichiers pour

les groupes, jusqu'au groupe des processus devant être chargés. Il est ainsi possible de n'infliger une charge processeur qu'à une sous partie de la hiérarchie des groupes.
- `all_name` spécifie le nom du groupe qui sera créé pour contenir toutes les tâches chargées.
- `cpu` définit la partie du profil concernant la charge processeur.
- `profile` identifie le début du profil de charge à proprement parler.
- `0` est l'identifiant du premier processeur à charger.
- Viennent ensuite la liste des dates où modifier la charge. Ainsi, sur cet exemple le processeur 0 sera chargé à 70% au démarrage et cela pendant une minute.
- Le mot clé `kill` en fin de profil indique que la génération de charge doit s'arrêter au bout de 150 secondes.

4.4 Évaluation de KRASH

Nous validons dans cette section notre implémentation du générateur de charge sous Linux. Nous présentons d'abord une série d'expériences validant que cette implémentation est capable de générer une charge dynamique de qualité. Nous comparons ensuite cette implémentation aux autres méthodes existantes, autant en termes de fonctionnalités que de performance.

Lors de ces expériences, nous cherchons à évaluer certains critères en particulier :
- reproductibilité : le générateur devrait être capable de reproduire une charge donnée quelles que soient les conditions (nombre de processus dans le système, quantité de mémoire disponible, etc).
- intrusivité : le générateur ne doit en aucun cas altérer le comportement du système. En particulier, la politique d'ordonnancement en vigueur ne doit pas être modifiée et aucune dégradation de performance additionnelle ne devrait impacter les autres composants du système (dégradation de la performance des entrées/sorties par exemple).
- précision et réactivité : nous cherchons un générateur capable de reproduire une charge processeur réaliste dans un environnement très dynamique. Autrement dit, le générateur devrait être capable de produire sa charge avec une bonne précision autant en terme de niveau de charge que de date de changement de charge.
- polyvalence : la charge processeur infligée par le générateur devrait être faible et prise en compte dans le procédé de génération. De même, la plage de niveaux de charge applicables devrait être la plus large possible.

Pour le reste de cette section, la machine utilisée est un système à mémoire partagée comprenant 8 processeurs bi-cœurs AMD Opteron 875 (2.2 GHz) et de 32 GiB de RAM. Cette machine fonctionne sous une distribution Debian (instable) utilisant un noyau Linux version 2.6.30. Ce noyau est configuré pour la gestion des groupes et l'ordonnanceur associé. Dans les cas où nous n'utilisons qu'une portion des cœurs du système, ils sont choisis de façon à utiliser le moins de processeurs possible : 8 cœurs utilisés signifie que seuls les 4 premiers processeurs sont actifs. Du point de vue des entrées/sorties, les deux premiers processeurs sont plus efficaces que les autres. Les résultats présentés sont tous tirés d'une moyenne sur 30 exécutions.

Les mesures de charge ont été réalisées à l'aide de Sysstat [God08]. Cet outil possède

une résolution d'une seconde, comparable aux autres outils existants sur les systèmes Unix. Une telle résolution étant d'un niveau similaire à l'ordonnanceur système, cet outil est amplement suffisant pour notre évaluation.

4.4.1 Validation expérimentale

Nous validons ici la précision, réactivité, reproductibilité et polyvalence de KRASH. Nous ne traiterons l'intrusivité de l'outil que dans la section suivante, en la comparant à celle des autres outils existants.

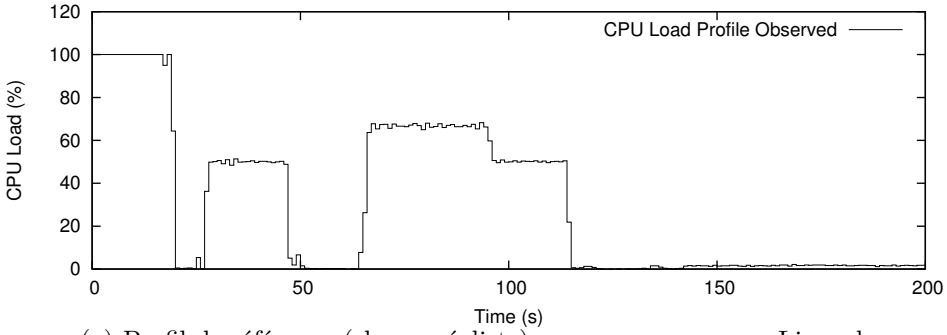
Pour cette validation, nous commençons par créer un profil de charge réaliste qu'il faudra reproduire avec KRASH. Ce profil de charge est créé en exécutant une application de référence (une instance de Linpack [DLP03]) et en lançant sur le système d'autres processus, plus courts en temps d'exécution, de façon à imiter un environnement partagé par plusieurs utilisateurs.

Le scénario d'exécution est le suivant : exécuter notre application de référence sur 8 cœurs avec la priorité par défaut ; lancer une tâche (boucle infinie) en temps-réel durant 15 secondes puis la supprimer ; attendre 10 secondes ; lancer une instance du programme EP des NAS [Dav91] sur 8 cœurs pendant 20 secondes (puis l'éliminer) ; attendre 16 secondes ; exécuter une nouvelle instance de EP sur 8 cœurs avec une priorité plus élevée. Lorsque cette instance se termine (30 secondes plus tard), en démarrer une autre avec la priorité par défaut et l'arrêter 20 secondes après. À ce point, Linpack s'exécute seul sur le système.

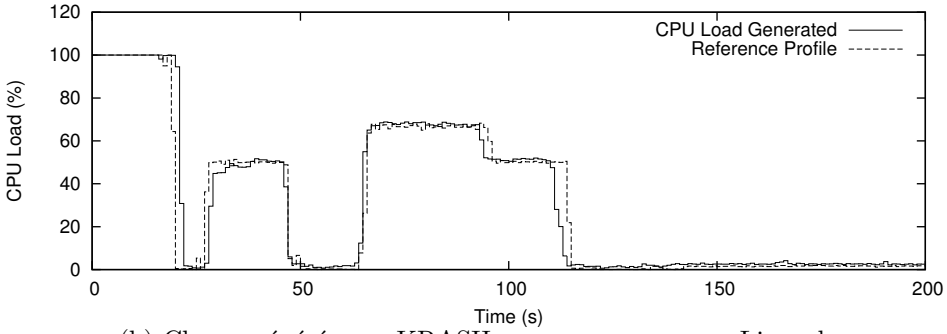
En mesurant le temps processeur alloué à Linpack, nous avons obtenu un profil de la charge qui lui était opposé par les différentes applications exécutées en concurrence. Ce profil est reproduit en figure 4.6a, nous l'appellerons le profil de référence. Notons d'ailleurs que les différentes applications utilisées n'ont eu un impact que sur le temps processeur alloué à Linpack, leurs empreintes mémoire étant très faibles.

La première expérience de validation consiste à reproduire le profil de charge que nous venons de mesurer à l'aide de KRASH et d'exécuter à nouveau une instance de Linpack. Si la charge est correctement reproduite par KRASH, Linpack sera perturbé de la même manière et nous obtiendrons le même profil de charge des deux cotés. En comparant la charge observée et le profil de référence, nous pourrons ainsi vérifier le fonctionnement de KRASH. La figure 4.6b nous donne la charge que nous avons mesurée lors de cette expérience (traits pleins), et le profil de référence en surimpression (pointillés). Nous pouvons constater qu'ils sont très proches l'un de l'autre : en moyenne, l'erreur entre la charge infligée et le profil est de 2% avec un écart-type de 1%. Cela confirme la précision et la réactivité de KRASH : l'environnement généré est très proche de l'environnement réel correspondant.

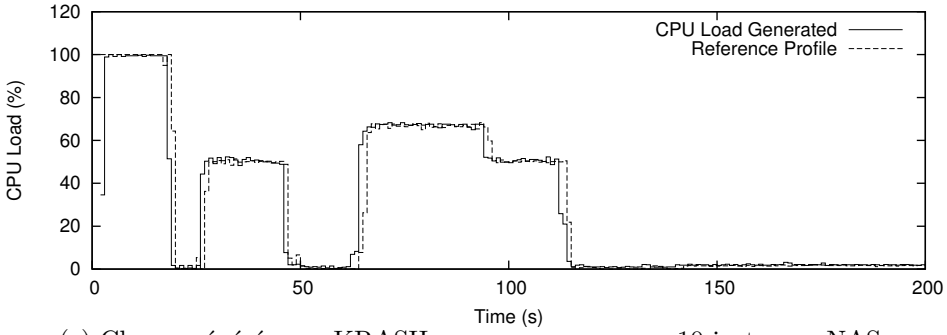
Notre deuxième expérience a pour but de valider la reproductibilité de KRASH : nous voulons vérifier que l'outil génère le bon profil de charge quel que soit le nombre et le type des applications en concurrence. Pour cela, nous remplaçons dans l'expérience précédente Linpack par 10 instances concurrentes des NAS EP. Il y a donc bien plus de processus que précédemment et au comportement différent. La charge induite par KRASH est présentée en figure 4.6c. Une fois encore, la charge réalisée par l'outil est de très bonne qualité : l'erreur moyenne est de 2% avec un écart-type de 1%. Nous pouvons donc dire que KRASH est capable de reproduire un profil de charge de manière précise



(a) Profil de référence (charge réaliste) en concurrence avec Linpack



(b) Charge générée par KRASH en concurrence avec Linpack



(c) Charge générée par KRASH en concurrence avec 10 instances NASs

et cela quel que soit le type d'applications en concurrence.

4.4.2 Comparaison aux autres solutions

Nous avons présenté un certain nombre de techniques permettant de reproduire une charge processeur et nous avons validé la qualité de notre solution. Après un rapide rappel des différentes caractéristiques de chaque solution, nous réalisons ici une comparaison de leur performance et intrusivité.

	Profil de charge dynamique	Effets de bord sur ordonnanceur	Reproductible	Intrusivité du superviseur	Résolution
KRASH	Oui	Négligeable	Oui	Négligeable	Celle de l'ordo.
Signaux	Non implémenté	Faibles	Non	Attente active	Supérieure à l'ordo.
Priorité temps-réel	Non implémenté	Élevés	Oui	Réveil régulier	Mauvaise
Fréquence	Non implémenté	Moyens	Oui	Négligeable	Supérieure à l'ordo.

Table 4.1 – Caractéristiques des différentes méthodes de génération existantes.

La table 4.1 donne un résumé des différents points que nous avons abordés plus tôt. Nous nous focalisons ici sur les implémentations réalisées par le projet Wrekavoc de ces différentes solutions. Comme aucune de ces dernières n'implémente de génération de charge dynamique, nous les comparerons à KRASH sur une charge constante.

Malheureusement, l'utilisation d'une charge constante ne nous permet pas de réaliser une telle comparaison à l'aide du profil de charge résultant. En effet, toutes les méthodes évaluées possèdent une résolution suffisante pour produire une telle charge. Nous pouvons néanmoins mesurer la précision de chaque méthode : l'application qui sera placée en concurrence de la charge ralentira en fonction de cette dernière. Nous pouvons aussi évaluer l'intrusivité du générateur : si une charge est générée sur d'autres ressources (entrées/sorties par exemple), cela affectera notre application. Dans les deux cas, nous pouvons utiliser le temps d'exécution (et donc le ralentissement) de l'application chargée pour vérifier le comportement de la méthode de génération. Pour ce qui est de l'intrusivité du générateur, nous pouvons la mesurer directement en observant le superviseur s'il existe.

Précision et Surcharge

Pour cette expérience, nous utilisons une instance de NAS EP sur 8 cœurs. Ce programme est un benchmark uniquement processeur, ne nécessitant que de peu de mémoire et n'effectuant que très peu d'entrées/sorties. Quelque soit la méthode de génération utilisée, seule la précision et l'intrusivité de la charge vont donc influer sur son temps d'exécution. Nous avons mesuré ce temps sur notre machine sans charge et pour chacune des méthodes. Ces dernières sont configurées pour utiliser 50% du temps processeur durant toute l'expérience et sont démarrées avant l'exécution de EP. Nous avons déduit du temps d'exécution de notre application le ralentissement observé. Comme il s'agit d'une application n'ayant que des besoins en temps processeur, la charge perçue devrait être de 50%. Lorsque la méthode de génération comprend un superviseur, nous avons aussi mesuré sa propre charge. Les résultats sont présentés dans la table 4.2.

	Temps d'exécution NAS EP		Charge Perçue		Surcharge	Résolution
	moy.	écart type	moy.	écart type		estimée
Sans charge	21.4	0.1	NA	NA	NA	NA
KRASH	41.9	0.9	48.9	1.1	1% sur un cœur	1
Signaux	47.6	4.5	55	3.9	15% par processus	2
Priorités	47.1	3.8	54.5	3.2	NA	1
Fréquence	45.3	0.3	52.7	0.5	NA	NA

TABLE 4.2 – Comparaison de la précision des différentes méthodes de générations.

La résolution que nous indiquons dans cette table est une estimation basée sur l'implémentation choisie pour chaque méthode. La version de Wrekavoc utilisant les signaux ne stoppe les processus chargés que sur des périodes d'une seconde, ce qui en ajoutant les opérations internes au superviseur nous donne une résolution plus proche de 2 secondes que de 1. Les priorités temps-réel quand à elles, en utilisant des périodes d'endormissement suffisamment courtes (de l'ordre de la microseconde) permettent dans le cas général d'obtenir une résolution proche de la seconde sur des applications n'utilisant que le processeur. Comme KRASH repose intégralement sur l'ordonnanceur Linux, il possède la même résolution. Enfin, nous ne reportons pas de résolution pour le changement de fréquence puisqu'il s'agit d'une modification matérielle réalisée en une seule opération (aucune activité durant l'expérience après installation).

Notre application cible n'utilisant que le processeur, la charge perçue devrait être très proche de 50%. KRASH et le changement de fréquence produisent les meilleurs résultats : notre méthode est meilleure en moyenne mais le changement de fréquence est plus stable. Pour ce qui est des résultats observés avec les signaux et les priorités, nous supposons que les grandes variations sont dues à une plus mauvaise distribution des *jiffies* au cours du temps, étant donné que les deux méthodes sont en concurrence avec l'ordonnanceur Linux.

Effets de bord

Une méthode de génération de charge processeur ne devrait pas influer outre mesure sur la performance des autres ressources du système. Nous cherchons ici à vérifier la présence d'effets de bord sur deux types d'applications particulières : celles effectuant des communications réseaux, et celles réalisant des entrées/sorties disque. Dans les deux cas, l'ordonnanceur système est configuré pour entrelacer d'autres tâches durant ces opérations pour ensuite privilégier les applications qui étaient bloquées. Une telle stratégie permet en général de maximiser le nombre d'opérations bloquantes réalisées par ces applications et donc de maintenir une bonne performance lorsque d'autres processus sont en concurrence. Par contre, si une méthode de génération de charge vient à perturber ces optimisations de l'ordonnanceur, les applications seront fortement perturbées au niveau temps d'exécution.

Notre première expérience se concentre sur l'aspect entrées/sorties. Nous utilisons la commande `dd` pour copier un fichier présent sur le disque dur de notre machine de test. Pour supprimer les effets de certains composants (cache écriture en tête), la commande est lancée de façon à ne terminer que lorsque toutes les écritures auront été portées sur le disque (option `fdatasync`). Nous demandons aussi à l'outil de ne réaliser ces écritures que par petits blocs de 1 kiB, de façon à déclencher un flux continu de

requêtes au système. La table 4.3 présente les résultats observés sur notre machine chargée à 50%, avec la copie d'un fichier de 1GiB.

	Temps de copie d'un fichier		Ralentissement
	moyenne	écart type	
Sans charge	10.2	0.8	1
KRASH	20.5	0.5	2
Signaux	24.9	1.7	2.4
Priorités	36.6	1.8	3.6
Fréquence	24.3	1.8	2.4

Table 4.3 – Effets de bord sur les E/S pour les différentes méthodes de génération.

Comme nous pouvons le constater, KRASH est le seul outil à produire une charge de 50%. Il s'agit de la seule méthode permettant à l'ordonnanceur de réaliser toutes les optimisations que nous avons détaillées précédemment. La méthode du changement de fréquence devrait aussi permettre ce comportement, malheureusement une dégradation de la fréquence du processeur provoque une dégradation de la réactivité du système (ralentissement des interruptions matérielles) et donc une charge perçue plus importante. Les deux autres méthodes produisent des résultats au moins aussi mauvais. Il s'agit là de la conséquence directe de leur influence sur l'ordonnanceur système. Les signaux envoyés peuvent ainsi interrompre notre application entre deux opérations d'écriture bloquantes, déclenchant deux pertes de temps processeur : au signal et au réveil avec le lancement de l'écriture. Les priorités temps-réel bloquant complètement l'activité de l'ordonnanceur durant l'exécution du processus de charge, la perturbation des optimisations de l'ordonnanceur est encore plus importante.

Intéressons nous maintenant à la performance des communications réseau. Nous utilisons comme application cible le benchmark NAS DT. Il s'agit d'une application réalisant des communications MPI point-à-point bloquantes et intensives entre tous ses processus. Nous exécutons ce programme avec 80 processus et une topologie de communication aléatoire. Les résultats observés sont reportés dans la table 4.4.

	Temps d'exécution		Ralentissement
	moyenne	écart type	
Sans charge	2.9	0.5	1
KRASH	6.2	0.8	2.1
Signaux	NA	NA	> 100
Priorités	11.3	3.2	3.9
Fréquence	4.4	0.6	1.5

Table 4.4 – Effets de bord de la génération de charge sur NAS DT.

Commençons par noter que la méthode de changement de fréquence ne produit un ralentissement que de $1,5$ seulement au lieu du 2 attendu avec une charge de 50%. Ce résultat surprenant peut s'expliquer par un mauvais recouvrement des communications

par les calculs sur une machine non chargée : si les différents processus participants n'utilisent pas assez le processeur, certaines communications ne seront pas complètement masquées. Dans ce genre de situation, un processeur plus lent permet une meilleure répartition et donc un ralentissement proportionnellement plus faible. Avec KRASH, le processeur est mieux utilisé par les processus poids, et le nombre d'opportunités de recouvrement n'augmente pas, amenant bien à une charge de 50%, comme cela se passerait avec une véritable application. Les priorités temps-réel quand à elles dégradent deux fois trop le temps d'exécution des applications, pour les mêmes raisons que dans le cas des entrées/sorties : les processus de DT ne sont pas privilégiés en sortie d'opération bloquante. Enfin, l'implémentation par Wrekavoc de la technique des signaux montre clairement ses limites lors de cette expérience. La charge produite par les 80 processus superviseurs lancés en parallèle de l'application dégrade la performance de cette dernière d'un facteur supérieur à 100. Dans ce cas précis, nous avons choisi d'arrêter l'expérience au-delà d'un certain temps d'exécution dépassé et c'est pour cela que le temps d'exécution n'est pas renseigné. Cela confirme ce que nous disions plus tôt dans ce chapitre sur la capacité de cette méthode à limiter un nombre de processus supérieur à un par cœur.

Pour compléter ces analyses, nous réalisons une dernière expérience sur la compilation parallèle d'un noyau Linux à l'aide de `gcc`. Ce type d'activité comprend la création d'un nombre élevé de processus avec des durées d'exécution très variées et des besoins tant en calcul qu'en entrées/sorties (au total 30000 fichiers d'une taille totale de 300MiB sont accédés). Comme précédemment, la charge infligée est de 50%, et la table 4.5 donne les temps d'exécution obtenus.

	Temps d'exécution		Ralentissement
	moyenne	écart type	
Sans charge	197	3	1
KRASH	387	7	2
Signaux	NA	NA	> 100
Priorités	558	5	2.8
Fréquence	392	21	2

TABLE 4.5 – Effets de bord sur une compilation gcc.

Dans cette expérience la performance des entrées/sorties est moins critique que précédemment. Le changement de fréquence et KRASH obtiennent donc de bons résultats avec un temps d'exécution deux fois plus long. Malheureusement, et pour les mêmes raisons que lors des autres expériences, la méthode des priorités temps-réel obtient un ralentissement plus élevé que souhaité $(2,8)$. Finalement, les signaux présentent les mêmes défauts qu'avec l'expérience du réseau : pour découvrir les processus créés et les ralentir, les superviseurs génèrent une charge trop importante sur le système. Ainsi, et même si le nombre de processus actifs durant la compilation ne dépasse jamais le nombre de cœurs de la machine, la création et la destruction très rapide de nouveaux superviseurs pour des processus se terminant très vite et l'inspection permanente de `/proc` consomment trop de ressources.

4.5 Utilisation et extensibilité de la méthode

Nous venons de valider notre méthode de charge et son implémentation autant en termes de qualité de la charge générée que d'intrusivité du procédé. KRASH nous permet donc de reproduire un profil de charge très général sur une machine dédiée et quelque soit le nombre de processus s'exécutant en concurrence. Nous l'avons laissé entendre à divers moments, ce type d'outil nous permet entre autre d'expérimenter sur des applications répartissant dynamiquement leur travail. Nous donnons un exemple de ce genre d'expériences dans la suite de cette section, avant de discuter plus généralement de la méthode et de son extensibilité.

4.5.1 Validation d'applications avec vol de travail

Beaucoup d'algorithmes parallèles reposent sur une répartition équitable du travail à réaliser entre les différents processeurs participant au calcul. Si cette découpe est généralement simple à réaliser et efficace sur des machines homogènes, elle souffre de problèmes de performance dès qu'un des participants au calcul ne peut s'exécuter à la même vitesse que les autres. Ainsi, le temps de complétion d'une application fonctionnant de cette façon peut être impacté sévèrement par un processeur plus lent que les autres ou par la présence d'une autre application sur la machine. Les architectures hétérogènes ou les systèmes partagés étant de plus en plus courants, d'autres politiques de répartition du travail sont venues attaquer ce genre de problèmes, parmi lesquelles se trouve le vol de travail.

Le vol de travail porte très bien son nom : dans ce type de répartition, un participant n'ayant plus rien à calculer vole une partie du travail à un autre, de façon à accélérer la terminaison de l'application. Cette technique repose essentiellement sur une description des opérations qu'il reste à réaliser (on parle généralement de tâches) et d'une politique de vol particulière (quelle quantité de travail voler et à qui ?). Les environnements permettant de concevoir ce type d'applications sont nombreux, dont les plus connus sont Cilk [Blu95] et Intel TBB [Rei07]. L'équipe MOAIS développe elle aussi une bibliothèque de ce type : XKAAPI [BLTG09, TDG+10]. Pour valider de tels environnements, une première étape peut être de vérifier que les calculs à réaliser sont répartis correctement sur un environnement hétérogène.

Nous réalisons donc ici une expérience visant à comparer deux versions d'un même algorithme parallèle : le calcul du préfixe. Dans une première version, le calcul à réaliser est découpé de façon équitable entre tous les processus participants. Dans la seconde version la totalité des opérations est contenue dans le premier processus et les autres participants viennent voler du travail lorsqu'ils n'en ont plus. Pour créer un environnement hétérogène, nous utilisons KRASH pour charger chacun des cœurs de la machine de façon à ce que plus le nombre de processeurs participants au calcul augmente et plus le dernier processeur ajouté est lent. Ainsi le cœur 0 n'est pas chargé, le 1 est chargé à 50%, le 2 à 75% et ainsi de suite. La figure 4.6 illustre les temps d'exécution obtenus sur cette expérience (même machine que précédemment).

Comme nous pouvons le constater, le temps d'exécution de l'algorithme avec découpe statique se comporte de moins en moins bien, tandis que l'algorithme avec vol de travail parvient à améliorer ses performances lorsque plus de processeurs sont attribués

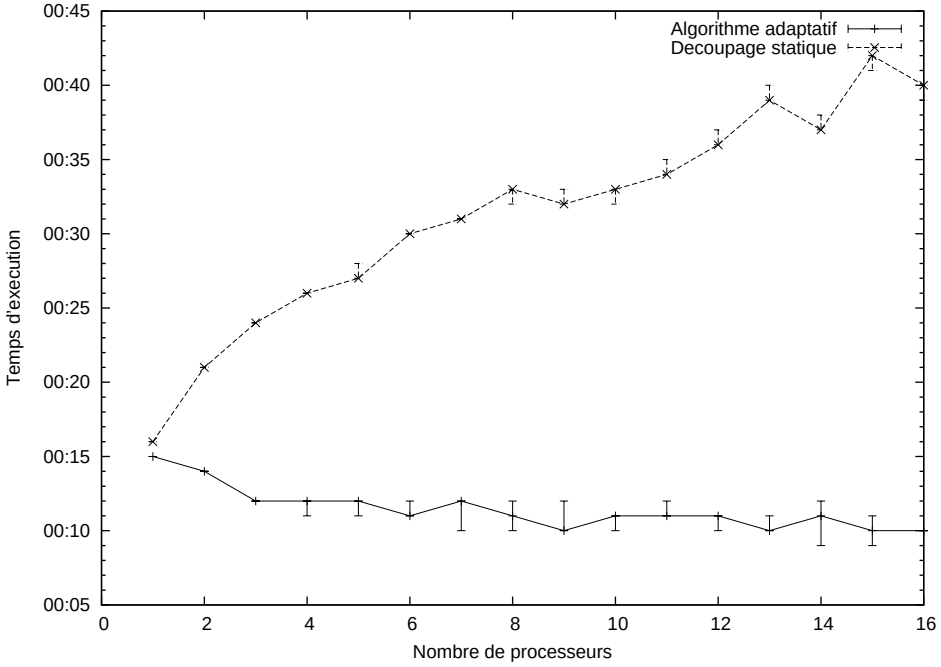


FIGURE 4.6 – Temps d'exécution des algorithmes parallèles sur machine rendue hétérogène par injection de charge.

au calcul. Bien que simple, cette expérience illustre certaines des utilisations possibles de notre méthode de génération de charge.

4.5.2 Extension aux autres ressources

Nous avons pu démontrer l'utilité de notre méthode de génération et sa performance. En se basant sur les mécanismes de répartition équitable du processeur, nous avons ainsi pu concevoir un outil pour contrôler l'utilisation de cette ressource par une application cible, et ainsi reproduire un environnement complexe de façon précise et reproductible. Mais le processeur n'est pas la seule ressource dont le système contrôle la répartition. On peut penser par exemple à la mémoire physique ou à l'accès au disque.

Qui plus est, la technique utilisée dans notre implémentation sous Linux se prête à ce type d'extension. Linux est un des rares systèmes ayant en effet étendu la plupart de ses politiques de répartition à la gestion des groupes de processus et à proposer un contrôle de ces dernières par l'utilisateur. Ainsi, le gestionnaire de mémoire physique permet de limiter le nombre de pages utilisées par un groupe de processus, le gestionnaire d'entrées/sorties peut limiter la banque passante disque disponible tandis que l'ordonnanceur de tâches temps-réel permet de gérer plusieurs groupes équitablement.

Bien que nous nous soyons limités durant cette thèse au cas de la charge processeur pour ce qui est des ressources contrôlées directement par le système, nous avons conçu KRASH pour pouvoir l'étendre sans difficultés à ces autres ressources, notamment du point de vue du passage à l'échelle sur le nombre d'opérations à réaliser dans un court intervalle.

Donner à l'utilisateur le contrôle du cache 5

Sommaire

5.1 Contrôler l'utilisation du cache 60
5.1.1 Partitionnement matériel 61
5.1.2 Support du système d'exploitation 61
5.1.3 Outils pour l'utilisateur 62
5.2 Coloration de page en espace utilisateur 62
5.2.1 Colorer l'espace d'adressage d'un processus 63
5.2.2 Gestion des hiérarchies de caches 64
5.3 CControl : un environnement de coloration de page sous Linux 64
5.3.1 Gestion de la mémoire sous Linux 65
5.3.2 Organisation générale de CControl 65
5.4 Validation de CControl 68
5.4.1 Environnement utilisé 68
5.4.2 Résultats 69
5.5 Conditions expérimentales et première optimisation . . . 71
5.5.1 Machines utilisées 71
5.5.2 Optimisation de NAS MG 72
5.6 Extraction d'isosurface 72
5.6.1 MT parallèle pour un cache partagé 73
5.6.2 Analyse des défauts de caches avec la distance de réutilisation 74
5.6.3 Éviter la pollution du cache 76
5.7 Stencil multi-résolutions 77
5.7.1 Mesure des besoins en cache 77
5.7.2 Redistribution du cache 79
5.8 Partitionner pour une hiérarchie de caches 80
5.8.1 Coloration par niveaux 80
5.8.2 Amélioration du stencil 82

Dès lors qu'une application réutilise certaines de ces données ou les partage entre différents threads, la hiérarchie de caches présente sur la machine joue un rôle dans sa performance. Un cache partagé entre plusieurs cœurs accélérera par exemple l'accès au travail commun à plusieurs threads tandis que la taille des caches influera sur la quantité de données qu'il est possible de conserver près des cœurs.

Nous avons déjà présenté plusieurs modèles qui visent à expliciter l'influence d'un cache sur une application. Malheureusement, il est difficile d'utiliser ces modèles aujourd'hui sur de véritables programmes. À titre d'exemple, la distance de réutilisation de l'ensemble des accès mémoire d'un programme ne peut être calculée qu'en capturant ces derniers pour ensuite simuler un cache. Il s'agit d'un procédé bien trop coûteux en calcul et en mémoire pour être utilisé sur des applications réelles. De plus, l'utilisateur ne dispose aujourd'hui que de très peu de méthodes pour améliorer l'utilisation qui est faite du cache par une application.

Pour répondre à ces problèmes, nous nous intéressons dans ce chapitre à la prise de contrôle, par un expérimentateur, de l'utilisation du cache par une application. Par ce contrôle, nous cherchons à remplir deux objectifs précis. Premièrement, un utilisateur doit pouvoir limiter la quantité de cache disponible pour une application. La démarche est similaire au chapitre précédent : nous cherchons à étudier comment la performance d'une application est impactée par la limitation d'une ressource matérielle. Deuxièmement, nous souhaitons pouvoir optimiser l'utilisation du cache, en laissant l'application en décider. Il s'agit là d'exploiter la connaissance du programmeur pour mieux tirer partie du matériel. Enfin ce contrôle devrait être applicable à un maximum d'architectures, de façon à pouvoir l'utiliser le plus largement possible (y compris sur des systèmes préexistants).

Ce chapitre présente donc notre contribution à ces questions. Après une analyse des différentes méthodes existantes pour contrôler le cache d'une application, nous détaillons un environnement permettant d'exercer ce contrôle au niveau utilisateur. Cet environnement a été implémenté et validé sous Linux. Nous illustrons ensuite plusieurs applications de cet environnement, pour l'analyse expérimentale des besoins d'un programme en cache (de manière globale et par structure de donnée) et l'optimisation de certaines applications.

5.1 Contrôler l'utilisation du cache

Sur la plupart des machines à mémoire partagée pour le calcul haute performance, ni l'architecture ni le système d'exploitation ne placent de limitations à la quantité de cache qu'une application est capable d'utiliser : seuls les schémas d'accès à la mémoire et le placement sur les cœurs déterminent la façon dont l'application sollicitera la hiérarchie des caches.

Il existe néanmoins des travaux à de nombreux niveaux instaurant une forme de contrôle sur l'application. Parmi ces travaux, les techniques de partitionnement de cache présentent un intérêt particulier.

Définition 7 *On désigne par partitionnement de cache toute technique visant à séparer un ou plusieurs caches en parties indépendantes. Cette indépendance concerne notamment la politique d'éviction du cache : une ligne rentrant dans une partition ne peut entrainer l'éviction d'une ligne d'une autre partition.*

Le partitionnement de cache répond naturellement à nos objectifs. Ainsi, si un utilisateur peut contraindre une application à n'utiliser qu'une partition de cache de petite taille, notre objectif de limitation de la quantité de cache disponible pour une application est atteint. De même, en redistribuant le cache sous la forme de plusieurs partitions,

il semble possible de contrôler et d'optimiser l'utilisation du cache par un programme. Nous détaillons dans la suite les différents travaux existants sur le partitionnement de cache, en les classant en trois catégories : les techniques nécessitant une modification de l'architecture, les techniques se plaçant au niveau du système d'exploitation et enfin celles donnant la main à l'utilisateur.

5.1.1 Partitionnement matériel

Une solution naturelle pour partitionner un cache matériel est de modifier directement l'architecture. Un tel changement garantit notamment l'efficacité du partitionnement : étant implémenté en matériel, il coûte peu en nombre de cycles et fonctionne automatiquement sans avoir à modifier ni le système d'exploitation ni les applications exécutées.

La plupart des travaux de ce type se focalisent néanmoins sur des problématiques de qualité de service et d'équité entre plusieurs applications [STW92, HRIM06, CCR+00, SRD01, CS07, QP06]. Il s'agit alors de garantir au niveau architectural que deux processus partageant un même cache ne se polluent pas mutuellement. Les techniques utilisées laissent alors peu de contrôle aux applications, pour se focaliser sur un moyen automatique de comprendre les besoins de ces dernières et d'y réagir.

Le principal défaut de ces travaux reste leur disponibilité : à l'exception de quelques rares architectures, aucun mécanisme de partitionnement de cache matériel n'est implémenté sur les systèmes d'aujourd'hui. Parmi les quelques exemples connus, notons le processeur SPARC64 VIIIfx qui fournit deux partitions en cache partagé et un jeu d'instructions spécial pour indiquer laquelle doit recevoir les données [Mar09].

5.1.2 Support du système d'exploitation

Plus proches de nos objectifs, certains travaux proposent la mise en place au niveau du système d'exploitation d'un mécanisme de partitionnement du cache. Ainsi, Soares *et al.* en 2008 [STS08] ont proposé un système détectant les schémas d'accès à la mémoire les plus néfastes. Après avoir observé les pages virtuelles déclenchant le plus de défauts de cache, ces dernières sont isolées dans une petite partition de façon à limiter leur impact sur le reste de l'application (un mécanisme dénommé `pollute buffers` dans l'article).

Dans l'esprit des mécanismes de partitionnement matériel, d'autres travaux [LLD+08, CJ06b] ont aussi implémenté une coloration de page classique pour distribuer le cache à différents processus. Nous retrouvons là un des problèmes des travaux précédents : ils se concentrent sur la question du partage d'un cache entre plusieurs applications (sans rapport les unes avec les autres) alors que nous cherchons le contrôle, par une application, du cache qu'elle utilise.

Enfin, certains travaux [BAM+96,SCE99] se sont intéressés à la coopération entre un compilateur et le système d'exploitation : en analysant statiquement une application, le compilateur peut dans un premier temps optimiser les accès mémoire de l'application puis indiquer au système les partitions à utiliser pour obtenir une performance optimale.

5.1.3 Outils pour l'utilisateur

Trois travaux se rapprochent nettement de nos objectifs : ULCC [DWZ11], SoftOLP [LLD+09] et le `Cache Pirating` [ENBSH11]. Ces trois travaux mettent en place un mécanisme pour limiter, et pour les deux premiers redistribuer, le cache disponible à une application.

ULCC est un outil pour redistribuer le cache partagé d'un processeur multicœur aux différentes structures de données d'une application parallèle. Ce travail se focalise sur un objectif quelque peu différent du notre : le partitionnement du cache est réalisé de manière automatique. Une fois que le programmeur a indiqué quelles structures de données sont partagées par les threads de l'application et lesquelles sont privées, un partitionnement préservant les premières dans le cache partagé est calculé. Nous verrons cependant par la suite que ce partitionnement n'est pas nécessairement le plus efficace : en fonction des schémas d'accès à ces structures partagées, il peut être bénéfique d'en privilégier une par rapport aux autres (en lui donnant plus de cache). Deuxième problème, les auteurs de ULCC considèrent que le programmeur connait parfaitement les besoins en cache de son application, alors que nous nous intéressons aussi à des expériences permettant de déterminer ces besoins.

Soft-OLP est très proche de ULCC dans son fonctionnement : l'outil partitionne le cache d'une application au niveau de ces objets. Néanmoins ce partitionnement est réalisé de manière automatique, après évaluation des distances de réutilisation de chacun des objets créés durant une exécution. Cette mesure est réalisée par instrumentation binaire de l'application, ce qui la ralentit de manière conséquente. Les auteurs admettent ainsi des ralentissements de l'ordre de 50 à 80 fois le temps d'exécution normal des applications instrumentées, ce qui les oblige à ne mesurer les distances de réutilisation que sur des problèmes de petite taille. L'outil extrapole alors les besoins de l'application, en supposant que les instances mesurées sont représentatives du comportement réel de l'application. Nous retrouvons là des problèmes incompatibles avec nos objectifs de contrôle du cache : l'application ne peut réaliser ce partitionnement elle-même.

Enfin, le piratage du cache (`Cache Pirating`) est une technique visant à "voler" le cache d'une application. Elle consiste à placer sur un cœur partageant le même cache que le programme cible (comme un `helper thread`) un outil réalisant un grand nombre d'accès mémoire. En contrôlant avec précision le `working set` du programme pirate, les auteurs démontrent que l'application ciblée est limitée dans son utilisation du cache. Ce travail est néanmoins restreint à plusieurs niveaux : il ne permet de perturber que le cache partagé entre plusieurs cœurs et ne peut limiter trop fortement la quantité de cache disponible. Ainsi, les auteurs remarquent que leur outil obtient des résultats trop instables au dessus de 75% de cache piraté. Bien entendu, cette technique ne permet pas non plus de donner le contrôle du cache à l'application cible.

5.2 Coloration de page en espace utilisateur

La plupart des solutions de partitionnement de cache que nous venons de présenter fonctionnent sur le principe de la coloration de page. Il s'agit d'une solution simple, ne demandant aucune modification du matériel et possédant une bonne granularité. Chaque couleur correspond en effet à une section de cache indépendante des autres :

toutes les pages d'une même couleur se placent dans les mêmes ensembles associatifs. Il est ainsi impossible pour deux pages de couleurs différentes d'entrer en conflit en cache. Pour compléter le dispositif, l'association entre pages virtuelles d'un processus et couleurs peut être contrôlée au niveau du système d'exploitation. Notre objectif étant de fournir ce contrôle à un expérimentateur ou au concepteur d'une application, il nous reste à transférer une interface de manipulation des couleurs du système à l'espace utilisateur.

5.2.1 Colorer l'espace d'adressage d'un processus

Généralement, lorsqu'un système implémente la coloration de page, la politique de répartition des couleurs est inaccessible à l'utilisateur. De plus cette politique fonctionne à une granularité très fine : chaque page de l'espace d'adressage virtuel est coloriée indépendamment.

Notre environnement fournit à l'utilisateur la capacité d'indiquer au système les couleurs à utiliser pour répondre aux besoins en mémoire physique d'une application. L'interface mise à disposition est simple : nous fournissons des primitives d'allocation de mémoire dynamique qui demandent en argument un ensemble de couleurs. Cet ensemble sera communiqué au système, qui limitera alors aux couleurs spécifiées la mémoire physique utilisée sur la nouvelle zone mémoire. Au sein d'une allocation, les couleurs autorisées sont utilisées de manière cyclique. Par exemple, si l'utilisateur demande une allocation de quatre pages avec uniquement les couleurs 1 et 2 alors le système utilisera pour cette allocation une séquence de pages de couleurs 1,2,1,2. Nous illustrons cet exemple en figure 5.1 dans un système à 4 couleurs. Une telle interface permet par exemple de spécifier des couleurs différentes pour deux structures de données de façon à ce qu'elles ne se polluent pas mutuellement en cache. Nous montrerons plus tard l'utilisation de cette interface pour améliorer substantiellement la performance d'applications parallèles.

Ce mécanisme de coloration est statique : l'utilisateur ne peut modifier les couleurs utilisées par une zone mémoire. Pourtant, certaines applications parallèles présentent des changements de phases, lors desquelles les schémas d'accès à la mémoire sont modifiés. Dans ces cas-là le programmeur peut, s'il le souhaite, effectuer une recoloration manuelle d'une structure en effectuant une nouvelle allocation et en copiant les données dans celle-ci.

Bien entendu, le partitionnement de cache induit un partitionnement de la mémoire. Autrement dit, comme pour toutes les techniques de partitionnement, la mémoire disponible pour une partition est limitée aux pages physiques des bonnes couleurs. Ainsi, une petite partition de cache ne contient que peu de couleurs et donc peu de pages physiques, ce qui limite la taille des allocations mémoire qui peuvent y entrer. Cette limitation peut sembler forte mais notre environnement ne réalise finalement qu'une redistribution des pages utilisées par une application. Un programme ayant assez de mémoire sans le partitionnement devrait donc en disposer de suffisamment avec notre environnement. Néanmoins, la taille de certaines partitions en nombre de couleurs pourrait devoir être augmentée plus que nécessaire (vis à vis de le réutilisation) pour des structures de données occupant beaucoup de mémoire.

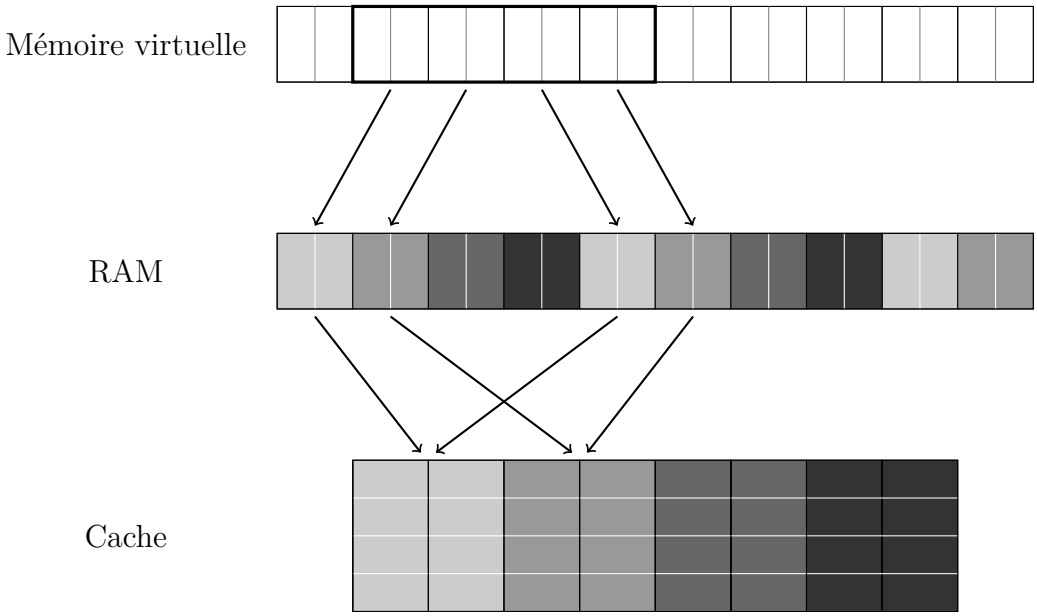


FIGURE 5.1 – Illustration de la coloration d'un espace d'adressage. L'application perçoit une zone contigüe mais le système lui fournit des pages physiques de seulement deux couleurs sur les quatre disponibles.

5.2.2 Gestion des hiérarchies de caches

Nous l'avons déjà dit, la plupart des processeurs modernes disposent d'une hiérarchie de caches. Certains caches (les plus éloignés des cœurs) peuvent être partagés tandis que d'autres sont privés à chaque cœur. Du point de vue de la coloration de pages, chaque niveau de la hiérarchie dispose d'une définition différente des couleurs : l'associativité et la taille du niveau de cache considéré changent la façon dont les pages physiques se placent en cache.

Nous considérerons dans un premier temps que notre environnement ne fonctionne que sur la coloration du niveau de cache le plus éloigné des cœurs. Il s'agit généralement du niveau le plus gros et contenant le plus de couleurs, ce qui nous offre la plus grande flexibilité. Nous reviendrons en fin de chapitre sur cette question, en discutant notamment de la possibilité de partitionner tous les niveaux de caches simultanément. Pour ce qui suit, il suffit de savoir que les niveaux inférieurs de la hiérarchie se comportent comme un cache classique lorsque seul le niveau le plus proche de la mémoire est partitionné.

5.3 CControl : un environnement de coloration de page sous Linux

Nous avons implémenté notre environnement de contrôle du cache sous Linux. Ce système présente l'avantage de ne posséder aucune gestion des couleurs, ce qui nous évite d'avoir à désactiver un mécanisme préexistant comme ce serait le cas sous FreeBSD par exemple. Cette implémentation repose sur la capacité de chargement dynamique de code sous Linux, qui permet d'injecter sous la forme de "module noyau"

notre code et de créer ainsi des interfaces système supplémentaires sans avoir à modifier en profondeur le noyau.

Les détails de notre implémentation étant directement liés à l'organisation du noyau en terme de gestion mémoire, nous détaillons les mécanismes existants d'allocation mémoire sous Linux avant de présenter notre environnement.

5.3.1 Gestion de la mémoire sous Linux

Lorsqu'un processus effectue une allocation mémoire en utilisant les fonctions POSIX de la bibliothèque C GNU, deux évènements peuvent avoir lieu. Si l'allocation est de grande taille (plus grande qu'une page), la bibliothèque effectuera un appel système `mmap`. Cet appel système permet notamment de demander l'allocation d'une nouvelle zone mémoire dans l'espace d'adressage d'un processus. Dans l'autre cas, la bibliothèque utilisera une zone mémoire provenant d'une réserve de pages précédemment allouée. Cette réserve est gérée dynamiquement, et de nouvelles pages sont forcément allouées au bout d'un certain temps. En conséquence, toute allocation mémoire finit nécessairement par faire appel au système pour demander des pages virtuelles supplémentaires.

Cette demande est toujours gérée de la même façon par le noyau : Linux crée ou étend une `area` ou zone mémoire. Cette structure représente une région de l'espace d'adressage du processus qui possède le même gestionnaire mémoire. Une fois les pages données au processus, le noyau Linux rend la main à ce dernier sans y avoir touché. Pour chaque `area`, un gestionnaire mémoire particulier est en charge des défauts de page. Ainsi, quand un processus accède pour la première fois à une page, le noyau renvoie le défaut de page vers le gestionnaire de la zone mémoire en question. Généralement, une page physique sera allouée au processus, mais ce type d'évènement peut aussi déclencher des accès au disque ou des opérations sur le réseau. Autrement dit, la gestion de l'espace d'adressage virtuel sous Linux fonctionne en deux étapes : une zone de mémoire virtuelle est activée (elle est alors accessible par le processus), puis un gestionnaire mémoire fournit des pages physiques au fur et à mesure que le processus y accède.

Dernier point, des modules noyau chargés à l'exécution peuvent modifier cette gestion de la mémoire virtuelle. Par exemple, un module peut ajouter au noyau des fichiers spéciaux (virtuels) ayant leur propre gestionnaire de défauts de page. Une fois qu'un processus a couplé ces fichiers à son espace d'adressage, un accès à la zone mémoire en question activera le gestionnaire spécialisé, contenu dans le module. C'est la technique que nous avons utilisée pour ajouter de la coloration de page au noyau.

5.3.2 Organisation générale de CControl

Notre environnement est séparé en deux parties : un module noyau qui implémente la coloration de page et en exporte un mécanisme de contrôle en espace utilisateur, et un couple de bibliothèques C simplifiant cette interface en implémentant des allocateurs mémoire avec coloration. Nous détaillons ici leur fonctionnement interne.

Module

Le module noyau a pour principal rôle de faire l'interface entre le système d'exploitation et l'utilisateur final de notre environnement. Le système permet ainsi à un module de réserver de la mémoire physique pour son usage exclusif et de fournir cette mémoire aux utilisateurs dans certaines conditions. Le fonctionnement de ce module se résume à quatre phases : la mise en place (réservation mémoire notamment), la réponse aux demandes de l'utilisateur, la réponse aux défauts de pages et enfin le nettoyage du système.

Lors du chargement, l'utilisateur précise au module la quantité de mémoire à réserver sur la machine. Le module alloue alors cette mémoire auprès du gestionnaire de mémoire physique par blocs contigus de grande taille (au minimum une page par couleur). Les allocations sont réalisées par blocs pour garantir l'obtention d'un nombre égal de pages pour toutes les couleurs : pour obtenir deux pages de la même couleur, le gestionnaire de mémoire physique de Linux nous oblige à allouer toutes les pages présentes entre ces dernières. Ces blocs de mémoire sont ensuite découpés et leur pages indexées par couleur. Pour finir cette initialisation, le module met en place un premier fichier spécial, capable de recevoir l'appel système `ioctl`.

En utilisant cet appel, l'utilisateur peut demander la création ou la destruction d'une partition de cache. Pour la création d'une partition, l'utilisateur fournit l'ensemble des couleurs à utiliser et la taille de l'espace en mémoire physique que le processus occupera. Le module créé alors un nouveau fichier spécial et lui réserve suffisamment de pages physiques aux couleurs spécifiées. Un code identifiant de manière unique le fichier spécial (numéro de `device`) est alors retourné à l'utilisateur. Dans le cas d'une demande de destruction, l'utilisateur fournit l'identifiant du fichier spécial lié à la partition et ce dernier est supprimé par le module qui récupère ainsi les pages physiques associées.

Une fois le fichier spécial d'une partition créé, un processus peut le coupler à son espace d'adressage avec l'appel système `mmap`. Lors de cet appel, Linux crée une nouvelle zone mémoire dans l'espace d'adressage de l'application, et charge notre module de la gestion des défauts de page sur cette zone. Cette gestion est très simple, puisque le noyau fournit au gestionnaire une description précise de la page virtuelle ayant été accédée. Il suffit alors au module de retrouver la page correspondante qui avait été réservée lors de la création de la partition.

Enfin, lorsque l'utilisateur demande le déchargement du module, la mémoire physique est rendue au système d'exploitation et tous les fichiers spéciaux sont supprimés.

Interfaces pour l'utilisateur

Deux interfaces sont disponibles pour l'utilisateur. Une première interface permet à un programmeur d'insérer dans son application un contrôle des couleurs utilisées pour certaines portions de son espace d'adressage. Cette interface fonctionne à la manière de la bibliothèque `libhugetlbfs` [Var11] sous Linux : elle fournit des fonctions pour créer et supprimer une partition en cache (une zone), puis une interface proche des fonctions d'allocation dynamique standard pour utiliser cette partition comme un allocateur mémoire.

Une zone correspond exactement à un fichier spécial sur le système. Elle est créée

en spécifiant les couleurs à utiliser et la taille mémoire à occuper. Une fois mise à disposition par le module noyau, un allocateur mémoire est mis en place à l'intérieur de la zone, de façon à ce qu'un programme y accède comme s'il réalisait des opérations d'allocation normales. Cette allocateur mémoire est simplifié au maximum, pour des questions de performance : il s'agit d'une simple liste chaînée de blocs libres, l'algorithme de choix d'un bloc lors d'une allocation étant réduit à choisir le premier bloc suffisamment grand. La figure 5.2 donne un exemple d'utilisation de cette interface, pour obtenir une partition de la moitié du cache et allouer un tableau de caractères à l'intérieur.

```
#include<ccontrol.h>

void do_stuff(char *t, size_t s);

int main(void) {
    char *t;
    struct ccontrol_zone *z;
    color_set c;
    size_t array_size = 100;
    /* la zone doit être légèrement plus grande
    * que le tableau.
    */
    size_t zone_size = array_size + 20;

    /* utilisons la première moitié du cache */
    COLOR_ZERO(&c);
    for(int i = 0; i < ccontrol_numcolors()/2; i++)
        COLOR_SET(i,&c);

    z = ccontrol_new(); // structure de contrôle
    ccontrol_create_zone(z,&c,zone_size); // création de la partition en noyau

    /* allocation d'un tableau de char dans la partition */
    t = (char *) ccontrol_malloc(z,array_size*sizeof(char));

    do_stuff(t,100);

    ccontrol_free(z,t); // libération du tableau
    ccontrol_destroy_zone(z); // destruction de la partition
    ccontrol_delete(z); // libération de la structure de contrôle
    return 0;
}
```

FIGURE 5.2 – Exemple de code utilisant notre interface de contrôle du cache, créant une partition de la moitié du cache et allouant un tableau de caractères à l'intérieur.

La variable d'environnement POSIX `LD_PRELOAD` permet de modifier les bibliothèques qui seront liées dynamiquement avec un programme. Cette variable est souvent utilisée pour interposer du code entre une application et une bibliothèque comme la bibliothèque standard C. Le principe est très simple : au démarrage d'une application, une phase d'édition de liens à lieu pour fournir à l'application certaines fonctions. Les bibliothèques pointées par la variable `LD_PRELOAD` sont prioritaires par rapport à celles des chemins contenus dans la variable `LD_LIBRARY_PATH`. Ainsi, si une bibliothèque contient une fonction respectant exactement le prototype des fonctions de la bibliothèque standard C, elle intercepte les appels provenant de l'application. Notre deuxième interface est une bibliothèque fournissant les fonctions d'allocation dynamique standard du C (`malloc,realloc,calloc,free`), mais alloue toutes les requêtes

dans une seule partition. L'utilisateur peut, au moyen de variables d'environnements, spécifier les couleurs à utiliser et la taille de la zone mémoire nécessaire. Cette interface permet donc de limiter le cache utilisé par une application, et cela sans en modifier le code source.

Pour simplifier l'utilisation de cette deuxième interface et le chargement/déchargement du module noyau, un outil en ligne de commande est aussi fournit. A titre d'exemple, la succession de commandes pour utiliser notre deuxième interface depuis cet outil est illustrée en figure 5.3.

```
~$ ccontrol load --mem 1G #chargement module avec 1GiB de RAM
~$ ccontrol exec --size 900M --colors "1,3,5,7-42" ./mytool
~$ ccontrol unload
```

FIGURE 5.3 – Commandes `shell` pour limiter le cache d'une application en utilisant la bibliothèque d'interception des allocations dynamiques.

5.4 Validation de CControl

Notre implémentation doit être validée sur deux aspects. Premièrement, nous devons vérifier sa capacité à fournir de la mémoire physique à une application. Ensuite, il faut s'assurer que la coloration de page est implémentée correctement et permet de partitionner le cache. Ainsi, une application utilisant notre interface de contrôle du cache doit être capable d'utiliser tout le cache si elle utilise toutes les couleurs et n'obtenir que la moitié de ce dernier si elle le demande.

5.4.1 Environnement utilisé

Les expériences suivantes ont été réalisées sur un système comprenant 4 Intel Xeon E5530. Chaque puce contient 4 cœurs, avec un cache L1 de 32 KiB pour les données, un L2 unifié, associatif 8 voies de 256 KiB et un cache L3 partagé entre tous les cœurs de 8 MiB, associatif 16 voies. Toutes les lignes de ces caches font 64 octets. Le cache L3 comprend donc 128 couleurs de 64 KiB chacune.

Pour valider correctement notre implémentation, une application capable d'utiliser tout le cache disponible et dont le comportement est facile à vérifier doit être utilisée. Nous avons choisi ici le programme déjà présenté en chapitre 2 (page 23) : les accès aléatoires sur une zone mémoire de taille variable. Pour rappel, ce programme alloue une région mémoire d'une taille configurée par l'utilisateur, la transforme en liste chaînée circulaire dont les éléments font la taille d'une ligne de cache, ordonne aléatoirement ces éléments et réalise ensuite un grand nombre de parcours de la liste. Rappelons que la vitesse moyenne d'accès à un des éléments de la liste dépend de la taille de la zone mémoire allouée. Le nombre d'accès effectués par l'application est suffisamment grand pour que la zone mémoire se place rapidement en cache si elle est suffisamment petite. Ainsi, plus la région grandit et plus les accès aléatoires accèdent aux caches les plus éloignés du cœur, prenant donc plus de temps.

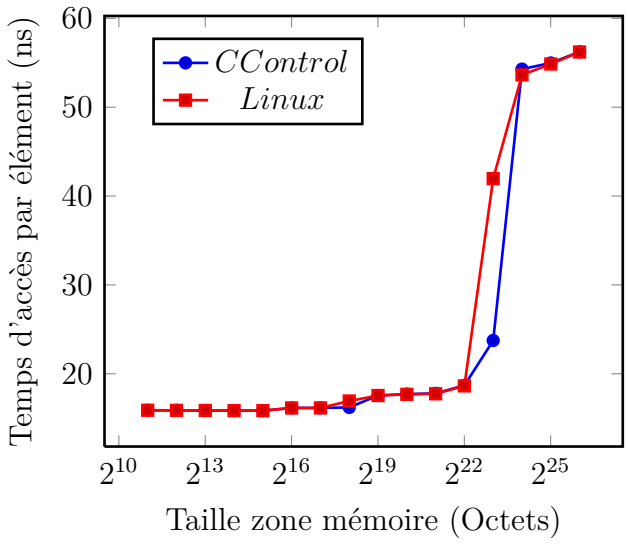


FIGURE 5.4 – Lectures aléatoires : temps d'accès par élément sur une zone mémoire de taille croissante. Comparaison de performance entre les allocations Linux et par CControl.

Sur notre système, l'application devrait comporter 3 `working sets` différents en fonction de la taille du tableau alloué : un premier lorsque le tableau est plus petit que le cache L2, un autre pour une taille comprise entre le cache L2 et le cache L3 et enfin un dernier lorsque le tableau est plus grand que le cache L2. Un autre plateau de performance pourrait être constaté lorsque le tableau est plus petit que le cache L1 mais en pratique la différence de performance entre les niveaux L1 et L2 est trop faible.

Chaque point indiqué dans les graphiques qui suivent est la moyenne de 100 exécutions de l'application. Le programme était fixé sur un seul cœur et exécuté en temps-réel avec priorité maximale. Les intervalles de confiance sont trop petits pour être affichés. Pour ces expériences, la taille de la partition de cache utilisée est indiquée en nombre de couleurs.

5.4.2 Résultats

Dans une première expérience, nous comparons les différents plateaux de performance de notre application en fonction de la stratégie d'allocation de la zone mémoire. Soit elle est allouée classiquement par le système (`mmap` et allocateur Linux) soit elle est contrôlée par notre environnement tout en ayant accès à tout le cache (les 128 couleurs sont données dans l'ordre à l'application). La figure 5.4 donne les résultats obtenus.

Dans une telle configuration, notre allocation mémoire se comporte comme si le système implémentait une coloration de page classique. Toutes les couleurs sont données à l'application et en même proportion. Le noyau Linux n'implémentant pas ce type de coloration, les couleurs qu'il alloue à l'application sont moins bien réparties. Il devrait en résulter une légère différence de performance entre les deux allocations : le temps d'accès moyen à un élément doit croître plus vite lorsque la taille de la zone accédée arrive aux alentours de la taille du cache L3 dans le cas de l'allocation Linux. Cette expérience valide aussi la performance de notre gestionnaire de défauts de page dans

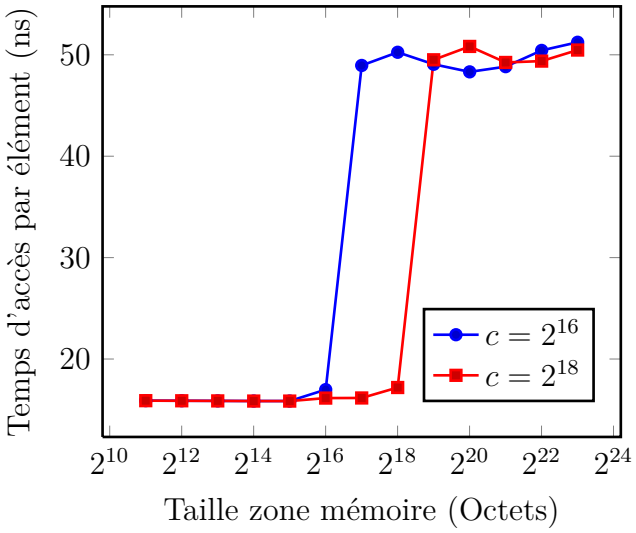


FIGURE 5.5 – Lectures aléatoires : temps d'accès par élément sur une zone mémoire de taille croissante. Comparaison de performance entre deux tailles de partition en cache.

le module : si la performance du code que nous avons ajouté au noyau est mauvaise, l'application se comportera moins bien en utilisant notre environnement.

Nous observons sur cette expérience les trois plateaux attendus : à chaque fois que la zone mémoire atteint une taille équivalente à celle d'un des niveaux de cache, le temps d'accès moyen par élément augmente. Ainsi, une légère diminution de la performance apparaît pour une taille de 2^{17}, qui est la taille du cache L2, et un changement plus important de performance autour de 2^{23} (taille du cache L3). Nous pouvons aussi remarquer qu'avec CControl la performance du programme change moins rapidement autour de la taille du cache L3, notre allocateur répartissant mieux les couleurs aux différentes pages virtuelles du processus que ne le fait Linux. Cette expérience valide donc la performance de notre environnement de contrôle du cache pour ce qui est de fournir une coloration de page correcte.

En deuxième expérience, nous validons le partitionnement de cache à proprement parler : en faisant varier la taille du cache disponible pour l'application. Si nous configurons notre programme pour ne demander qu'une partie des couleurs disponibles à notre environnement, alors le changement de performance dû à la sortie du cache L3 devrait apparaître plus tôt. Nous comparons donc deux tailles de partitions différentes pour voir si le temps d'accès moyen à un élément augmente lorsque la taille de la zone mémoire approche celle de la partition. La figure 5.5 donne la performance observée.

Comme nous pouvons le voir, lorsque la partition de cache fait 2^{16} octets (soit une seule couleur) la perte de performance apparaît sur une tableau de 2^{17} octets et un comportement similaire survient pour une partition 4 fois plus grande. Cette expérience valide donc que CControl est capable de partitionner le cache L3 de notre architecture, et donc qu'il fonctionne correctement.

5.5 Conditions expérimentales et première optimisation

Maintenant que notre environnement de contrôle du cache a été validé, nous nous intéressons à son utilisation. À travers trois classes d'applications, nous montrerons dans les sections suivantes comment exploiter CControl pour analyser une application et l'optimiser en redistribuant le cache disponible à plusieurs structures de données.

5.5.1 Machines utilisées

Dans les expériences qui vont suivre, quatre machines au total ont été utilisées. La diversité de ces machines permet tout d'abord de vérifier que notre environnement de contrôle du cache fonctionne sur différentes architectures.

La machine que nous venons d'utiliser pour la validation de CControl fait partie des systèmes de nous réutiliserons. Nous la désignerons par la suite sous le nom de M_{valid}. Malheureusement, nous avons perdu l'accès à cette machine durant l'été 2011. Pour la remplacer, un des systèmes de Grid5000 fut utilisé. Ce dernier est composé de deux puces Intel Xeon E5520 (4 cœurs chacune) et de 24 GiB de RAM. Les processeurs E5520 sont strictement équivalents, pour notre utilisation, aux processeurs E5530 de Grimage (seule la fréquence des cœurs change). Ils possèdent ainsi la même hiérarchie mémoire : même nombre de niveaux, même partage, même tailles et donc même nombre de couleurs. Ce nouveau système sera nommé M_{rempl} pour la suite de ce chapitre.

Les deux systèmes précédents sont basés sur des architectures Nehalem. Ces architectures sont particulières du point de vue mémoire. En effet, les processeurs en question possèdent des contrôleurs mémoire dédiés et des mécanismes de préchargement spéculatif plus agressifs que ceux d'architectures plus anciennes comme les Core2. Malheureusement, les compteurs de performance matériels permettant de mesurer les défauts de cache ne comptabilisent pas l'activité du préchargement. Ce phénomène nous empêche, sur certaines applications, de visualiser clairement l'influence du cache sur leurs performances. Pour palier ce problème, nous avons utilisé pour une de nos expériences une machine comprenant un Intel Core2 Duo. La puce contient deux cœurs, chacun possédant un cache L1 de donnée de 32 KiB et un L2 unifié associatif 16 voies de 4 MiB. Tous les caches ont des lignes de 64 octets. Du point de vue de la coloration, le cache L2 contient donc 64 couleurs, de 64 KiB chacune. Nous appellerons cette machine M_{simple}.

Enfin, pour tester certaines hypothèses sur le partitionnement de plusieurs niveaux de cache en simultané, nous avons utilisé la machine M_{part}. Ce système, que nous avons déjà décrit dans le chapitre 2, comprend 4 Intel Xeon X7460, chacun contenant 6 cœurs et 3 niveaux de cache. Le cache L1 est privé à chaque cœur, de taille 32 KiB. Le L2 est partagé par 2 cœurs et fait 3 MiB en taille avec une associativité 12-voies. Enfin le L3 est partagé par les 6 cœurs d'une puce, associatif 16-voies et de 16 MiB en taille. Nous avons choisi d'utiliser cette machine en raison du nombre important de couleurs du L3 et de la présence de deux niveaux de cache de grande taille.

	Défauts de cache L_3	Performance (MOP/s)
Linux	$2.4 \cdot 10^8$	4620
CControl	$2.3 \cdot 10^8$	4752
CControl optimisé	$2.2 \cdot 10^8$	4853

Table 5.1 – Performance de NAS MG en fonction de la politique d'allocation du cache.

5.5.2 Optimisation de NAS MG

Nous avons déjà présenté les `NAS Parallel Benchmarks` : une suite d'applications pour l'évaluation de performance de supercalculateurs. Développés par une division de la NASA, ils représentent certains des algorithmes les plus utilisés dans le calcul haute performance et sont très utilisés en recherche pour l'évaluation de nouvelles méthodes de parallélisation ou d'analyse des programmes. Nous cherchons ici à optimiser un des programmes NAS : MG. Il s'agit d'un programme de résolution d'une équation de Poisson discrète (selon la technique multi-grille). Sans entrer dans les détails, notons juste que ce programme est connu pour être intensif en mémoire et donc dépendre sensiblement de la hiérarchie mémoire de la machine pour sa performance.

Nous utilisons ici la version 2.3-OpenMP de ce programme, modifiée pour utiliser notre environnement de contrôle du cache si besoin. Ces modifications sont légères : moins de 30 lignes de code ont été changées. La machine utilisée est M_{rempl}. Les résultats présentés sont les moyennes de 30 exécutions par paramétrage de l'expérience. La classe A est choisie pour la taille des données en entrée.

Nous avons connaissance d'une stratégie d'optimisation simple de ce programme. Trois structures de taille importante sont accédées en parallèle dans ce dernier : les tableaux multi-dimensionnels U, V et R. Le tableau V est principalement écrit tandis que les deux autres sont principalement utilisés en lecture. Pour éviter que les écritures sur V ne polluent le cache inutilement, nous décidons de l'isoler dans une partition de cache séparée et d'utiliser le reste du cache pour les deux autres tableaux. Une première partition de 120 couleurs est donc utilisée pour U et R tandis que V est alloué dans les 8 couleurs restantes.

Nous comparons cette stratégie d'allocation aux deux autres stratégies possibles : des allocations mémoire normales (Linux) et l'utilisation d'une seule partition avec toutes les couleurs pour les trois tableaux. L'application est exécutée sur 4 cœurs de la même puce. Le tableau 5.1 reporte les défauts de cache observés et la performance calculée par l'application. Celle-ci mesure en effet son temps d'exécution durant sa phase de calcul et en déduit un nombre d'opérations par seconde en millions. Comme nous pouvons le constater, CControl nous permet d'améliorer les performances de notre application, avec 8% de diminution des défauts de cache, pour une performance augmentée de 5%.

5.6 Extraction d'isosurface

L'extraction d'isosurface est un filtre très classique du domaine de la visualisation scientifique. Il permet de mieux comprendre la structure d'un champ scalaire inclus dans un maillage tridimensionnel en visualisant les surfaces ayant la même valeur sca-

laire. L'un des algorithmes les plus utilisés pour cette opération se nomme `Marching Tetrahedron` [JH04] (MT). Pour chaque cellule du maillage, cet algorithme lit les coordonnées et la valeur scalaire des points et calcule une triangulation de l'isosurface passant par cette cellule.

Nous présentons dans cette section comment une analyse de la performance en cache de la version séquentielle de cet algorithme peut nous aider pour sa parallélisation. Nous disposons en effet de deux méthodes pour paralléliser cet algorithme, selon que l'on privilégie la diminution des défauts de cache ou la simplicité de la synchronisation entre threads. Avant d'aller plus loin, il faut néanmoins comprendre la performance en cache de cet algorithme.

Les défauts de cache induits par MT peuvent être analysés comme suit. Le maillage est constitué de deux tableaux multidimensionnels : un tableau stockant, pour chaque point, ses coordonnées et un scalaire et un tableau contenant, pour chaque cellule, les indices de ses points (*cf.* figure 5.6). Le procédé de construction du maillage fait en sorte que l'ordre des points et des cellules ait un certaine localité : les points et cellules proches les uns des autres dans le maillage auront des indices proches dans le tableau (et donc en mémoire). Ainsi, si les cellules sont traitées dans l'ordre de leurs indices, une certaine localité apparaît sur les accès aux points : des cellules consécutives auront régulièrement des points en commun ou des points placés dans la même ligne de cache. Cette localité peut même être améliorée par des techniques de tri des tableaux de points et de cellules [TDR10].

5.6.1 MT parallèle pour un cache partagé

Comme chaque cellule peut être traitée indépendamment, il est facile de paralléliser cet algorithme. Il suffit de diviser la séquence de cellule à traiter en morceaux continus et d'assigner à chaque cœur un morceau. Le travail à réaliser variant en fonction de la cellule à traiter, nous avons utilisé un ordonnanceur par vol de travail pour équilibrer dynamiquement la charge de chacun des cœurs. Lorsqu'un cœur n'a plus de cellules à traiter il sélectionne ainsi un autre cœur au hasard pour lui voler la moitié du travail restant. Ce schéma parallèle utilise efficacement les caches privés d'un processeur multicœur : chaque cœur traite des cellules proches les unes des autres et maximise ainsi la réutilisation des points chargés dans son cache privé. Cependant, les cœurs opèrent sur des portions éloignées les unes des autres dans la séquence des cellules, ce qui réduit la possibilité pour deux cœurs de partager des points. En conséquence, cet algorithme parallèle, que nous appellerons NoWindow, n'utilise pas efficacement le dernier niveau de cache d'un processeur multicœur qui est généralement partagé entre tous les cœurs.

Pour améliorer la réutilisation des données stockées dans le cache partagé, nous étudions un autre algorithme nommé SlidingWindow, introduit récemment [TDG+10]. Une fenêtre glissante est introduite dans cet algorithme et restreint les cœurs de façon à ce qu'ils traitent des cellules proches dans la séquence de traitement globale. Chaque cœur continue de traiter des séquences continues de cellules pour utiliser au mieux son cache privé, mais ces séquences sont désormais plus petites et plus proches les unes des autres. Pour implémenter cet algorithme par vol de travail, le cœur au début de la séquence des cellules est doté d'un statut spécial et nommé `master`. Lorsqu'un cœur

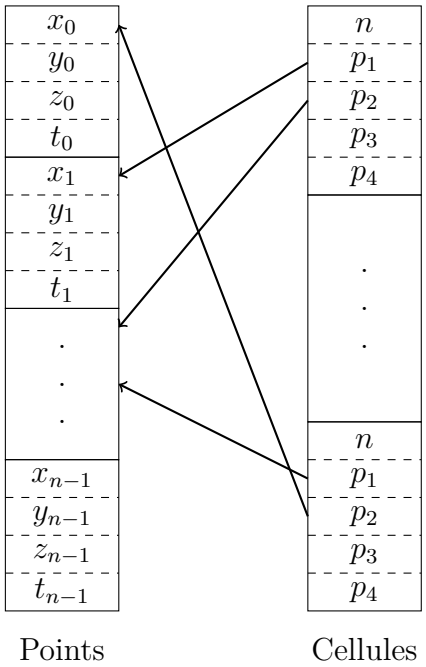


FIGURE 5.6 – Structure du maillage : le tableau de points contient les coordonnées et un scalaire (t) tandis que le tableau de cellules conserve les indices des points composant chacune d'entre elles.

vole le `master`, il ne peut récupérer que des cellules dans la fenêtre. Les autres vols se déroulent comme dans l'algorithme NoWindow. La fenêtre est avancée au fur et à mesure que les éléments du début sont traités. Tous les cœurs travaillent donc à l'intérieur de la fenêtre pendant l'exécution de l'application.

5.6.2 Analyse des défauts de caches avec la distance de réutilisation

L'algorithme SlidingWindow améliore l'utilisation du cache partagé mais augmente les coûts de synchronisation par rapport à NoWindow : les cœurs volent des quantités de travail plus petites. Nous aimerions, bien sûr, n'utiliser SlidingWindow que si la quantité de défauts de cache sur le niveau partagé par les cœurs diminue suffisamment. Pour prédire le gain de performance de cet algorithme, nous utilisons ici la fonction Q des `working sets` de l'algorithme séquentiel traitant les cellules dans l'ordre global.

Soit $H(d)$ le nombre d'opérations en mémoire avec une distance de réutilisation de d dans l'algorithme séquentiel. Le nombre de défauts de cache sur un cache complètement associatif de taille C est donné par $Q(C) = \sum_{d=C+1}^{\infty} H(d)$. Nous supposons ici que l'algorithme séquentiel exhibe une bonne localité, autrement dit que les cellules proches dans la séquence utilisent des points communs et que les cellules éloignées ne partagent aucun point.

Nous considérons tout d'abord l'algorithme NoWindow s'exécutant sur p cœurs partageant un cache de taille C. Dans ce cas, comme chaque cœur traite des points différents, la distance de réutilisation d'un accès est équivalente à celle sur l'algorithme

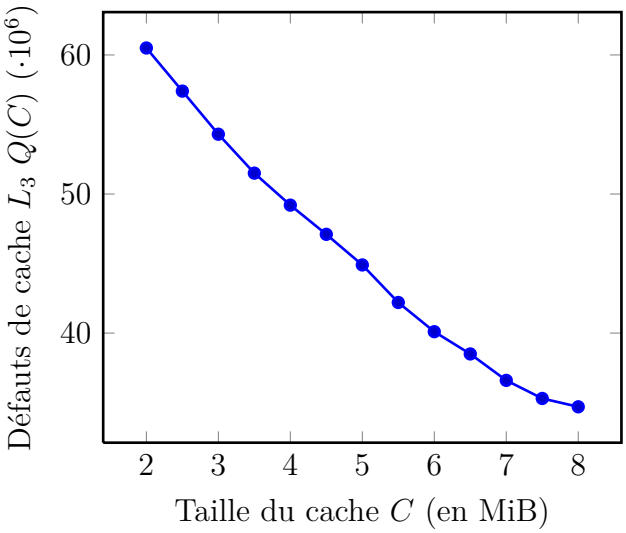


FIGURE 5.7 – Nombre de défauts de cache (pour le cache partagé) de l'algorithme MT séquentiel en fonction de la taille du cache.

séquentiel multipliée par p : chaque accès par un cœur est suivi de $p-1$ accès indépendants en parallèle (si l'on suppose que tous les cœurs avancent à la même vitesse). Ainsi, $H_{\text{no-win}}(d) = H(\frac{d}{p})$ et le nombre de défauts de cache de l'algorithme NoWindow est

$$Q_{\text{no-win}}(C) = \sum_{d=C+1}^{\infty} H\left(\frac{d}{p}\right) = \sum_{d=\frac{C}{p}+1}^{\infty} H(d) = Q\left(\frac{C}{p}\right).$$

L'algorithme NoWindow induit donc autant de défauts de cache que l'algorithme séquentiel sur un cache p fois plus petit.

Intéressons nous maintenant à l'algorithme SlidingWindow lorsqu'il traite des éléments à une distance de m au maximum. Soit $r(m)$ le nombre maximum d'accès mémoires distincts lors du traitement de $m-1$ éléments consécutifs de la séquence de cellules. Dans le pire des cas, lorsque le dernier élément de la fenêtre est traité, tous les autres éléments ont déjà été traités, en accédant au maximum à $r(m)$ emplacements mémoire distincts par rapport à l'algorithme séquentiel. Ainsi, la distance de réutilisation d'un accès a augmenté au maximum de $r(m)$. Le nombre de défauts de cache de l'algorithme SlidingWindow est donc

$$Q_{\text{w}}(C) \leq \sum_{d=C+1}^{\infty} H(d - r(m)) = Q(C) + \sum_{d=C+1-r(m)}^{C} H(d).$$

Comme nous avons considéré que la séquence originelle possède une bonne localité, $r(m)$ est petit devant m et $H(d)$ est petit pour un d grand. En conséquence, le dernier terme de notre formule est petit et l'algorithme SlidingWindow induit approximativement le même nombre de défauts de cache que l'algorithme séquentiel.

Nous vérifions ces résultats expérimentalement en mesurant le nombre de défauts de cache de l'algorithme séquentiel $Q(C)$ pour des tailles de cache C entre 2 MiB et 8 MiB sur la machine M_{valid} (*cf.* figure 5.7). Nous avons utilisé un maillage de 150 000 000 cellules. Le nombre de défauts de cache est bien plus important sur un cache de

	Défauts de cache L_3	Temps (ms)	Accélération
Séquentiel ($C = 2$MB)	$60.5 \cdot 10^6$	5015	0.66
Séquentiel ($C = 8$MB)	$34.7 \cdot 10^6$	3320	1.00
NoWindow	$55.3 \cdot 10^6$	1137	2.92
SlidingWindow	$38.4 \cdot 10^6$	964	3.44

Table 5.2 – Performance des deux algorithmes MT parallèles NoWindow et SlidingWindow comparée à la version séquentielle.

	Défauts de cache L_3	Temps (ms)
Linux	$37.1 \cdot 10^6$	4124
CControl	$34.7 \cdot 10^6$	3320
CControl optimisé	$23.7 \cdot 10^6$	3090

Table 5.3 – Performance de l'algorithme MT séquentiel en fonction de la politique d'allocation du cache.

2 MiB que sur un cache de 8 MiB. Nous nous attendons donc à ce que l'algorithme SlidingWindow déclenche bien moins de défauts de cache que NoWindow sur les 4 cœurs d'une même puce. Cette attente est confirmée expérimentalement (*cf.* Table 5.2) : NoWindow déclenche le même nombre de défauts de cache que l'algorithme séquentiel avec quatre fois moins de cache. En conséquence, son accélération est inhibée par son manque de localité. À l'opposé, SlidingWindow déclenche légèrement plus de défauts que l'algorithme séquentiel et offre ainsi une très bonne accélération. En conclusion, nous pouvons dire que CControl nous a permis de vérifier sur une machine réelle une analyse théorique de ces différents algorithmes.

5.6.3 Éviter la pollution du cache

En examinant les schémas d'accès à la mémoire de l'algorithme MT, on peut remarquer que seuls les accès aux points présentent une certaine localité. Ni la lecture de la séquence de cellules ni l'écriture des triangles générés ne possèdent la moindre localité, et gâchent de l'espace en cache. Pour éviter ces défauts, notre implémentation de MT utilise des instructions non temporelles pour effectuer les écritures. Ces instructions indiquent au processeur que les écritures réalisées n'ont pas besoin d'être conservées en cache et peuvent directement être placées en RAM, ce qui évite la pollution du cache par des données sans réutilisation. Nous aimerions pouvoir faire de même pour la lecture de la séquence de cellules. Malheureusement, aucune opération de lecture non temporelle n'est disponible sur notre architecture. Cependant, notre environnement de contrôle du cache nous permet d'éviter cette pollution du cache en isolant les structures de données accédées sans réutilisation dans une partition de cache la plus petite possible. Nous avons donc ajouté cette optimisation pour le tableau des cellules du maillage dans notre implémentation et nous comparons les performances de l'application selon trois stratégies d'allocations mémoire disponibles dans la table 5.3.

Dans la première méthode, nommée Linux, les allocations sont réalisées à l'aide d'un simple `malloc`. Dans la seconde, dénotée CControl, nous allouons les structures à l'aide de CControl en utilisant toutes les couleurs disponibles pour obtenir une meilleure

utilisation du cache que sous Linux. Enfin, la stratégie nommée CControl optimisé réserve 100 couleurs au tableau de points tandis que le tableau des cellules et celui des triangles sont isolés dans les 28 couleurs restantes. Comme nous l'espérions, la distribution du cache de façon à privilégier le tableau exhibant le plus de réutilisation nous apporte une amélioration substantielle de la performance de l'application : les défauts de cache sont diminués de 36% entraînant une accélération de l'application d'un facteur 1.33.

5.7 Stencil multi-résolutions

Pour démontrer tout le potentiel de l'utilisation de notre environnement de contrôle du cache, nous avons programmé une application jouet inspirée des filtres classiquement rencontrés en visualisation scientifique. Notre application utilise simultanément trois matrices résidant en mémoire pour calculer les éléments d'une matrice résultat. Les matrices en entrée forment une grille multi-résolutions, composée d'une grande matrice ($Y \times X$ éléments, chacun de la taille d'une ligne de cache), d'une matrice de taille moyenne (un quart de la grande) et d'une petite (un seizième de la grande). La matrice résultat est, quand à elle, de même taille que la grande matrice. Chaque élément résultat est une combinaison linéaire de neuf points provenant de chaque matrice en entrée, aux mêmes coordonnées (interpolées pour les matrices plus petites).

Cette application est intéressante pour deux raisons : elle est extrêmement intensive en mémoire et présente simultanément des `working sets` de différentes tailles. Notre `stencil` à neuf points forme une croix (un élément central, deux éléments au dessus, deux en dessous, à gauche et à droite) et est compris dans 5 lignes d'une matrice. Ainsi, dans une configuration idéale, si cinq lignes de chaque matrice en entrée sont conservées en cache durant le calcul, l'application aura une localité maximale. Cela se traduit par un besoin en cache de $X \times L \times 5$ octets pour la grande matrice, moitié moins pour la moyenne et un quart de cet espace pour la plus petite. Bien sûr, si ces `working sets` n'utilisent pas de couleurs disjointes, ils se pollueront mutuellement en cache durant l'exécution de l'application.

La figure 5.8 illustre notre application, avec le `stencil` à 9 neuf points formant une croix lue dans chacune des 3 matrices pour calculer une seule cellule de la matrice résultat. Les besoins en cache de chaque matrice sont aussi affichés. Les matrices sont nommées ici de la plus petite M_1 ($X/4$ par $Y/4$) à la plus grande M_3 (X par Y), la matrice résultat étant M_r.

5.7.1 Mesure des besoins en cache

Pour comprendre les besoins en cache de chaque structure de donnée à l'intérieur de l'application, nous avons besoin de redéfinir les notions de distance de réutilisation et des `working sets` associés pour les restreindre à une région mémoire spécifique à l'intérieur de l'espace d'adressage de l'application.

Ainsi, la distance de réutilisation d'un accès mémoire relativement à une région est défini par le nombre d'accès mémoire dans la région spécifiée avant un accès au même emplacement. Le nombre d'accès à l'intérieur de notre région dont la distance de réutilisation est supérieure à la taille du cache peut alors être interprété comme le nombre

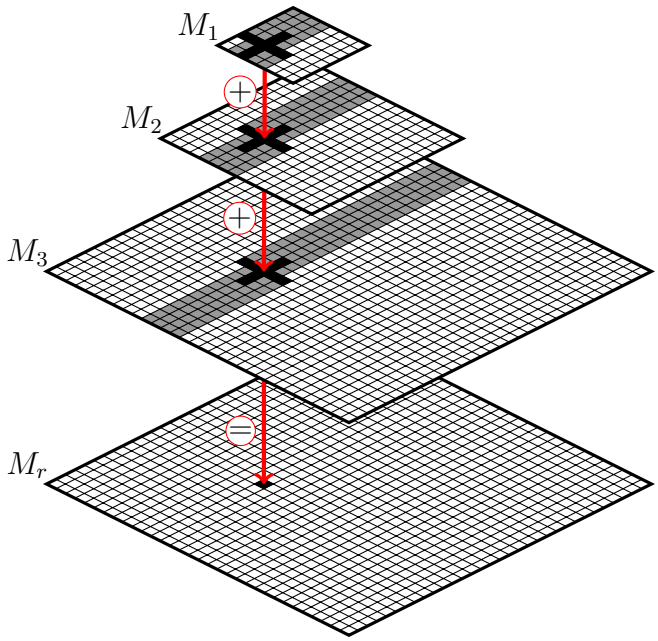


FIGURE 5.8 – Stencil multi-résolutions : 9 cellules des matrices M_1,M_2 et M_3 sont sommées dans une cellule de M_r. Les zones grisées représentent les besoins en cache de chaque matrice.

de défauts de cache déclenchés lorsqu'on isole la région mémoire dans une partition dédiée. De la même façon, les `working sets` d'une région mémoire représentent les différents niveaux de performance atteignables en fonction de la taille de la partition de cache isolant cette région du reste des accès de l'application.

Nous avons conçu une expérience pour mesurer ces `working sets`, pour chacune des structures de données à l'intérieur d'une application. Pour commencer, nous isolons une structure cible dans une partition de cache. Ensuite, toutes les autres structures de données sont placées dans une seconde partition de taille fixée et la plus petite possible. Il suffit alors de mesurer les défauts de cache déclenchés par l'application en faisant varier la taille de la partition dédiée d'une exécution à l'autre. Les structures de données placées dans la même partition de taille fixée génèrent systématiquement le même nombre de défauts de cache. Si des variations dans le nombre de défauts de cache sont observées, elles ne peuvent résulter que du changement de taille de la partition dédiée à une seule structure. L'expérience est répétée pour chacune des structures d'intérêt dans l'application.

Notons tout de même que cette expérience ne permet pas de comparer directement le nombre de défauts de cache déclenchés sur une taille de partition donnée pour deux isolations de structures différentes. En effet, pour chacune des structures isolées, la composition de la partition de taille fixée change, ce qui rend impossible de comparer le nombre de défauts de cache directement. Ainsi, seule la forme de courbe des défauts de cache pour une structure isolée en particulier présente un intérêt.

Nous utilisons cette fois la machine M_{simple}. Du point de vue de la coloration, son cache L2 (il n'y a pas de cache L3) contient 64 couleurs, de 64 KiB chacune. La figure 5.9 nous donne les `working sets` que nous avons mesurés pour chacune des matrices de notre application, sur une taille de partition variant entre 8 et 56

	Défauts de cache L_2	Temps (ms)
Linux	$3.6 \cdot 10^6$	139
CControl	$3.3 \cdot 10^6$	78
CControl optimisé	$2.2 \cdot 10^6$	57

TABLE 5.4 – Performance de notre application *stencil* en fonction de la politique d'allocation du cache.

couleurs. Les autres matrices sont confinées dans une partition de 8 couleurs. Pour ces expériences, l'application était configurée avec $X = 7168$ et $Y = 100$.

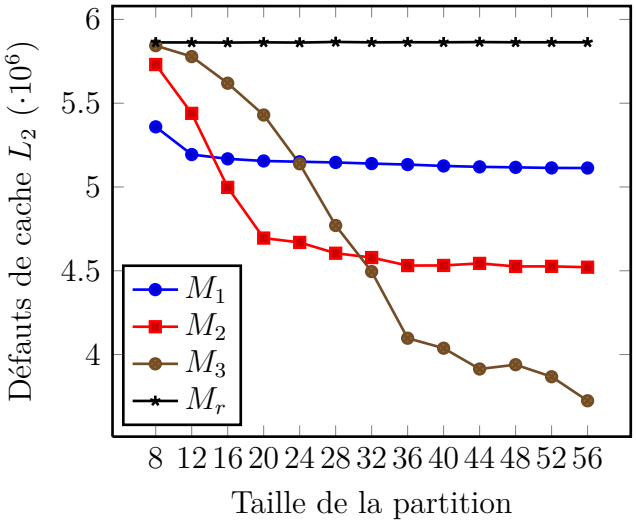


FIGURE 5.9 – Stencil : Défauts de cache L_2 en fonction de la taille de la partition en cache pour chacune des matrices.

Les `working sets` mesurés correspondent très bien à notre analyse théorique de l'application : chaque matrice requiert cinq de ces lignes en cache, excepté pour M_r qui ne présente aucune localité (le programme ne fait qu'écrire dedans). Comme chaque matrice M_i est deux fois plus grande que M_{i+1} (en X), elle nécessite deux fois plus de cache.

5.7.2 Redistribution du cache

Au vu des besoins en cache de chaque matrice, nous configurons notre application pour utiliser une partition de cache différente pour chacune d'entre elles. Ainsi, nous pouvons donner exactement la bonne quantité de cache à chaque matrice et améliorer la performance de notre application.

Pour ce partitionnement, nous choisissons de donner 9 couleurs à M_1, 18 à M_2, 35 pour M_3 et les 2 restantes à M_r. La table 5.4 présente les performances obtenues par notre application, en fonction des trois allocations mémoire disponible : Linux pour l'allocation classique sans notre environnement, CControl pour une allocation avec une seule partition utilisant toutes les couleurs et CControl optimisé pour la version avec redistribution du cache à chaque structure.

Nous pouvons constater que notre environnement de contrôle du cache nous a permis d'améliorer de 38% le nombre de défauts de cache en L2 de notre application par rapport à l'allocation standard. Cette baisse du nombre de défauts de cache s'accompagne d'une accélération d'un facteur 2.4 du temps d'exécution totale de l'application.

Bien sûr, notre programme a été conçu spécifiquement pour montrer les gains de performance qu'il était possible d'obtenir avec CControl. La taille des matrices est ici configurée pour qu'une allocation standard ne puisse utiliser tout le cache bien qu'une allocation précise des couleurs le permette. Nous montrerons néanmoins dans la section suivante que ce type d'amélioration peut aussi s'appliquer à des applications plus complexes, et qu'il n'est pas nécessaire de comprendre parfaitement les schémas d'accès à la mémoire d'une application pour obtenir des résultats intéressants.

5.8 Partitionner pour une hiérarchie de caches

Nous considérions jusqu'alors que le partitionnement de cache ne s'effectuait qu'au niveau du cache le plus éloigné (typiquement L3) du processeur. Si ce type de partitionnement promet déjà des résultats intéressants, il est parfaitement possible d'envisager de contrôler plusieurs niveaux simultanément. Seule contrainte, les différents niveaux doivent être indexés physiquement pour pouvoir appliquer la technique de la coloration de page. Nous approfondissons donc le partitionnement multi-niveaux dans cette section, en étudiant son impact sur notre application *jouet* : le stencil multi-résolutions.

5.8.1 Coloration par niveaux

Avant de rentrer dans la partie expérimentale à proprement parler, il est nécessaire de comprendre comment la coloration de page peut être appliquée à plusieurs niveaux de cache. Pour chaque niveau adéquat, nous utilisons la définition classique de la coloration. Cela signifie que, pour chaque niveau, les groupes d'ensembles associatifs partageant les mêmes pages physiques appartiennent à la même couleur. En conséquence, chaque page physique se voit associée à une couleur par niveau de cache.

La plupart du temps, chaque niveau de cache d'une architecture dispose de caractéristiques différentes (taille, associativité) et donc d'un nombre de couleurs différent. Il existe néanmoins une correspondance entre les couleurs de chaque niveau : si le niveau L2 possède n couleurs, alors la couleur i du niveau L3 correspondra à la couleur $i \ mod \ n$ en L2. La figure 5.10 illustre ce recouvrement des couleurs sur une hiérarchie fictive comprenant un cache de 4 couleurs et un cache plus petit de 2 couleurs seulement, avec des pages occupant deux ensembles associatifs.

Le schéma de partitionnement que nous avons appliqué jusqu'alors ne permet pas de séparer plusieurs niveaux de la hiérarchie en même temps. Ainsi, si nous choisissons de partitionner selon le dernier niveau (plus éloigné d'un cœur), en regroupant des couleurs consécutives dans la même partition alors ce groupe risque de s'étendre sur toutes les couleurs du niveau inférieur. En effet, l'association cyclique entre les couleurs d'un niveau et celles du niveau inférieur signifie que des couleurs consécutives ne se recouvrent pas plus bas dans la hiérarchie. Dans un tel cas, les autres niveaux ne sont alors pas partitionnés du tout et des accès concurrents à plusieurs partitions se gêneront à travers la hiérarchie (à l'exception du dernier niveau).

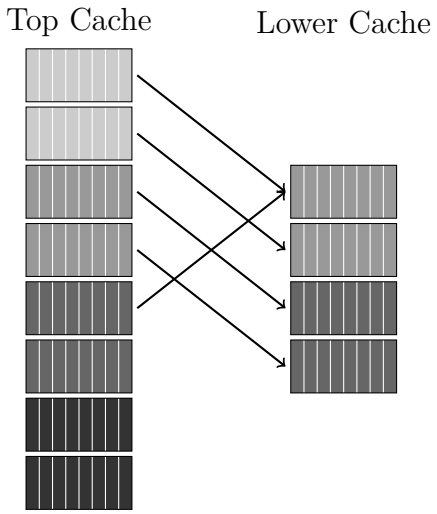


FIGURE 5.10 – Coloration de page à travers une hiérarchie de cache. Les couleurs du plus haut niveau se recouvrent dans le niveau inférieur, qui en contient moins.

Prenons comme exemple une architecture possédant 16 couleurs en L3 et 4 en L2. Si nous séparons le L3 en deux en utilisant des couleurs consécutives, la première partition contient les couleurs 0 à 8 et la deuxième le reste. Malheureusement, sur cette architecture les couleurs 0,4,8 et 12 en L3 correspondent à la couleur 0 en L2, les 1,4,9 et 13 à la couleur 1 et ainsi de suite. En conséquence, nos deux partitions en L3 utilisent les quatre couleurs du L2.

Il est toujours possible de partitionner tous les niveaux de cache simultanément. En effet, si le partitionnement est réalisé à partir des couleurs du plus petit niveau (plus proche d'un cœur), il est préservé à travers l'ensemble de la hiérarchie. L'inconvénient majeur de cette solution étant bien sûr que le plus petit niveau est celui avec le moins de couleurs et donc permettant le moins de flexibilité dans le partitionnement. Dans le cas général cette solution n'est d'ailleurs pas la meilleure, puisqu'il peut parfaitement être judicieux de laisser certaines structures de données recouvrir plusieurs partitions dans les niveaux inférieurs de la hiérarchie si leurs besoins en cache et leur utilisation s'y prêtent. Ainsi, une structure de donnée nécessitant $x\%$ du cache L3 pour fonctionner au mieux ne requiert pas forcement $x\%$ du cache L2 : un `working set` adéquat en L3 peut ne pas tenir en L2. Un algorithme pour résoudre ce problème devient alors très complexe, puisqu'il s'agit de prendre en compte les besoins en cache de chaque structure, le recouvrement des couleurs d'un niveau à l'autre et le potentiel de gain d'un partitionnement.

Nous illustrerons dans la suite les améliorations possibles à l'aide d'un algorithme très simple : le partitionnement est réalisé sur le dernier niveau mais avec un ordre des couleurs modifié. Cet ordre place de manière consécutive des couleurs du dernier niveau se recouvrant dans les niveaux inférieurs. En utilisant cet ordre, il devient possible de partitionner plusieurs niveaux en même temps. Il suffit de dimensionner les partitions de façon à ce que toutes les couleurs du dernier niveau correspondant à la même couleur au niveau inférieur soient dans la même partition.

5.8.2 Amélioration du stencil

Nous utilisons cette fois une architecture plus propice au partitionnement de plusieurs niveaux de cache : M_{part}. Du point de vue de la coloration, le cache L3 de cette machine contient 256 couleurs et le L2 64. Nous illustrons ici deux contextes pour lesquels le partitionnement des niveaux L2 et L3 présente un intérêt : dans le premier cas il s'agit de mieux répartir le partitionnement L3 en L2 et dans le second de simplement partitionner le L2 plutôt que le L3.

Commençons par utiliser des tailles de matrices de sorte qu'une ligne de m_3 occupe 1.6 MiB de mémoire ($X = 26214$). Nous en déduisons que M_3 nécessite 8 MiB en cache, moitié moins pour M_2 et 2 MiB pour M_1. Ainsi, un bon partitionnement pour notre architecture serait de 128 couleurs pour M_3, 64 pour M_2, 32 pour M_1 et le reste pour M_r. La figure 5.11 présente les `working sets` mesurés pour cette configuration de matrices. Chaque point est la moyenne de 30 exécutions. Ces résultats sont très proches de notre analyse des besoins en cache de l'application, chaque structure de donnée ayant approximativement besoin du nombre de couleurs prévu.

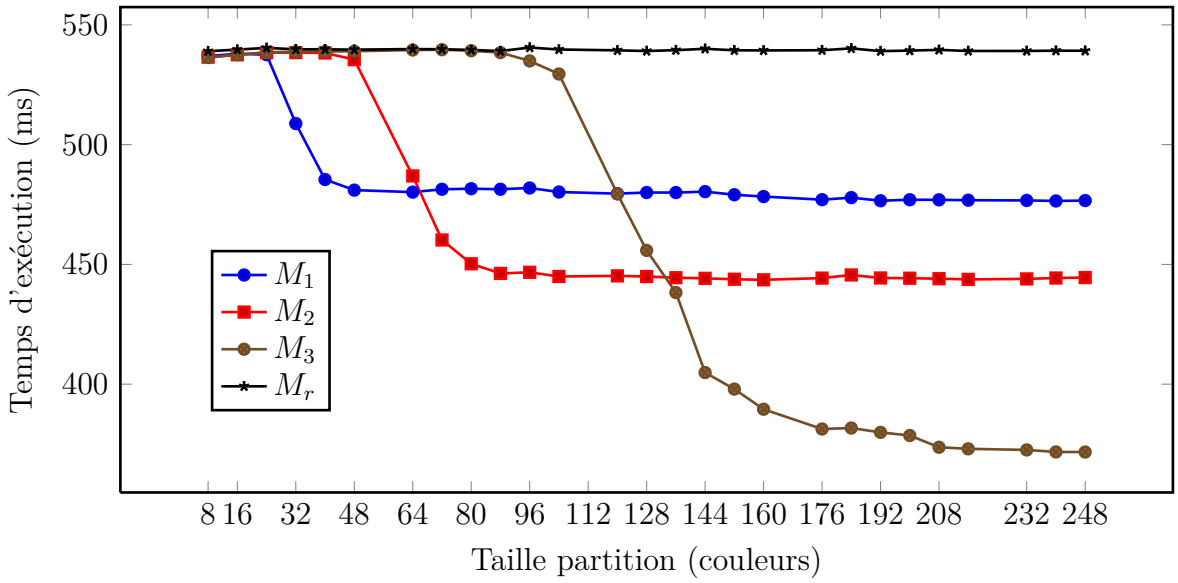


FIGURE 5.11 – Stencil : temps d'exécution en fonction de la taille du cache, pour chaque matrice.

Au vu de ces besoins en cache, nous appliquons dans un premier temps un partitionnement uniquement en L3, sans prise en compte des couleurs utilisées en L2. Pour la version optimisée, nous attribuons les 36 premières couleurs à M_1, ensuite 74 couleurs à M_2, 144 à M_3 et les deux dernières à M_r. La performance obtenue est reportée dans la table 5.5, aux cotés de la performance obtenue avec Linux et l'utilisation d'une seule partition.

En utilisant notre environnement de contrôle du cache nous obtenons une meilleure utilisation du L3, avec 16% de défauts de cache en moins et un temps d'exécution amélioré de 15%. Malheureusement, ce partitionnement ne fonctionne pas particulièrement bien en L2. En effet, les 3 MiB de cache ne permettent en aucun cas de sauvegarder l'ensemble des lignes nécessaires pour les matrices, et ce premier partitionnement ne

	Cycles en attente	Défauts de cache L3	Temps (ms)
Linux	$9.5 \cdot 10^8$	$2.7 \cdot 10^6$	384
CControl	$9.3 \cdot 10^8$	$3.0 \cdot 10^6$	377
CControl optimisé	$8.0 \cdot 10^8$	$2.6 \cdot 10^6$	324
Optimisation L2	$7.9 \cdot 10^8$	$2.6 \cdot 10^6$	320

TABLE 5.5 – Stencil, M_3 nécessitant 8 MiB de cache. Comparaison entre les différentes politiques d'allocation de cache.

	Cycles en attente	Défauts de cache L2	Temps (ms)
Linux	$3.1 \cdot 10^8$	$9.4 \cdot 10^6$	132
CControl	$3.1 \cdot 10^8$	$9.3 \cdot 10^6$	133
L2 Optim.	$2.9 \cdot 10^8$	$7.7 \cdot 10^6$	128

TABLE 5.6 – Stencil, M_3 occupant 1.6 MiB en cache. Comparaison entre un contrôle du cache L2, une seule partition L3 et Linux.

réalisant aucun contrôle sur le L2, les partitions traversent toutes les couleurs de ce dernier.

Pour améliorer ce partitionnement, nous décidons donc de l'activer aussi sur le cache L2. Ainsi, en changeant l'ordre d'attribution des couleurs, nous obtenons des partitions en L2 de 9 couleurs pour M_1, 18 pour M_2 et 36 pour M_3. Et il reste ainsi 1 couleur pour M_r. La performance de cette nouvelle optimisation est aussi indiquée dans la table précédente. Comme elle permet de mieux utiliser le L2, une ligne de chaque matrice y est conservée, une petite amélioration des performances de l'application apparaît.

Le partitionnement du L2 peut aussi sembler intéressant lorsque les besoins en cache de l'application y tiennent complètement, le partitionnement du L3 n'offrant alors aucun bénéfice direct. Pour tester cette hypothèse, nous configurons notre application pour que 5 lignes de la matrice M_3 occupent 1.6 MiB ($X = 5242$). Les matrices M_2 et M_1 nécessitent alors 800 KiB et 400 KiB respectivement. Comme l'ensemble de ces besoins peuvent être placés en cache L2, nous le partitionnons directement. La table 5.6 donne les temps d'exécution et le nombre de cycles processeur perdus en attente d'une ressource pour cette allocation du cache, comparée à Linux et un contrôle du cache avec une seule partition.

Comme nous pouvions nous y attendre, le partitionnement du cache L2 permet un gain de performance, avec 23% de défauts de cache L2 en moins et 4% d'amélioration du temps d'exécution. Bien entendu, la présence du cache L3 sur cette architecture limite fortement le coût d'une mauvaise utilisation du cache L2, ce qui explique que le gain en temps d'exécution ne soit pas plus important.

II

Génération de graphes de tâches pour la simulation d'ordonnanceurs

Notions d'ordonnancement 6

Sommaire

6.1 Définitions **87**
6.1.1 Classification des problèmes d'ordonnancement 88
6.1.2 Ordonnancement avec dépendances 89
6.2 Complexité et Approximabilité **90**
6.2.1 Généralités 90
6.2.2 Heuristiques 90
6.3 Influence d'un graphe sur la performance d'un algorithme **92**
6.3.1 Longueur des chemins 92
6.3.2 Antichaîne 93

Nous nous intéressons dans cette partie à la simulation d'algorithmes d'ordonnancement, et plus particulièrement à la question de la génération des entrées de ces algorithmes : les graphes de tâches. Il est néanmoins indispensable, avant de pouvoir aborder les difficultés que pose ce sujet, de rappeler les définitions de base et quelques théorèmes essentiels du domaine de l'ordonnancement. Nous nous limiterons naturellement à la partie de ce domaine qui nous intéresse plus particulièrement, à savoir l'ordonnancement pour machines parallèles.

Ce chapitre est donc un tour d'horizon de la question théorique de l'ordonnancement d'un graphe de tâches sur machines parallèles. Nous commencerons par un rappel de la définition générale du problème de l'ordonnancement et de la notation classique utilisée dans la littérature pour classifier les nombreuses versions de ce dernier. Nous nous intéresserons ensuite à quelques résultats en complexité et approximation, avant d'étudier la forme la plus populaire d'heuristiques pour la résolution de ces problèmes : les ordonnanceurs par liste. Nous terminerons en discutant plus en détails de l'importance de certaines caractéristiques des graphes de tâches vis à vis de la performance d'un ordonnanceur.

6.1 Définitions

De manière générale, un problème d'ordonnancement est défini comme la répartition d'un ensemble de ressources à un ensemble de travaux, avec pour objectif l'optimisation d'un ou de plusieurs critères de performance.

Définition 8 *Formellement, étant donnés un ensemble de machines* $P = (p_0, p_1, ..., p_m)$ *et un ensemble de travaux (ou tâches)* $J = (j_0, j_1, ..., j_n)$ *à réaliser, un ordonnancement*

est constitué d'une fonction σ qui associe à chaque tâche une date de démarrage et d'une fonction π qui lui alloue un processeur. Cet ordonnancement respecte un certain nombre de contraintes, comme par exemple le fait qu'une seule tâche puisse utiliser une machine donnée à un instant précis.

Classiquement, le nombre m indique le nombre de processeurs et n le nombre de tâches. Les indices i servent alors à identifier les tâches et les j les machines. La représentation la plus commune d'un ordonnancement prend alors la forme d'un diagramme de Gantt, que nous illustrons sur un exemple en figure 6.1.

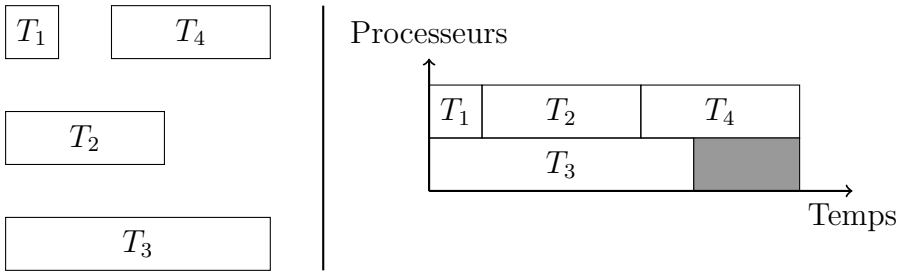


FIGURE 6.1 – Diagramme de Gantt d'un ordonnancement de quatre tâches. L'espace grisé indique un temps d'inactivité d'un processeur.

6.1.1 Classification des problèmes d'ordonnancement

Une instance d'un problème d'ordonnancement est donc caractérisée par la description des tâches, des ressources et des objectifs. Pour simplifier cette caractérisation, Graham *et al* [Gra69] ont proposé la notation à trois champs $\alpha \mid \beta \mid \gamma$. Dans cette notation, le champ α définit le type de machines utilisées, le champ β les caractéristiques d'une instance du problème et le dernier champ la ou les fonctions objectif qu'il faut optimiser. Nous détaillons ici quelques-unes des notations, et donc des problèmes, rencontrés dans la littérature.

Pour ce qui est du α, trois notations sont particulièrement utilisées : les symboles P, Q et R. Le symbole P désigne ainsi les problèmes pour lesquels des processeurs équivalents sont utilisés pour exécuter les tâches. Isolé, ce symbole indique un nombre arbitraire de machines (paramètre de l'instance), pour un nombre m de processeurs fixé, il est noté Pm. Chaque tâche i peut donc être exécutée sur chaque processeur, et cela dans en un même temps noté p_i. On parle aussi de processeurs identiques pour ce modèle. Le symbole Q désigne lui des processeurs hétérogènes uniformes : chaque processeur j dispose d'une certaine vitesse de traitement des instructions s_j et exécute donc une tâche i contenant o_i opérations en $p_{ij} = o_i s_j$. Enfin, dans un modèle où les processeurs sont hétérogènes non uniformes (R), le temps d'exécution d'une tâche est indépendant d'un processeur à l'autre, et se nomme donc p_{ij}. Notons aussi l'utilisation du 1 comme symbole pour le premier champ, qui désigne alors que la machine ne comprend qu'un seul processeur.

Suivant le modèle d'application considéré, les caractéristiques des tâches peuvent être nombreuses. Parmi celles déjà mentionnées, le deuxième champ de la classification de Graham contient ainsi la notation p_i pour le temps d'exécution d'une tâche, particulièrement dans le cas de tâches unitaires : $p_i = 1$ signifie que toutes les tâches

prennent exactement le même temps pour s'exécuter. Les dates de disponibilité sont notées r_i et celles d'échéance (*deadline date*) d_i. Enfin, la notation *prec*, très classique, désigne la présence de dépendances entre les tâches.

Parmi les fonctions objectifs utilisées, les dates de terminaisons sont les plus populaires. On note classiquement $C_i = \sigma(i) + p_{i\pi(i)}$. Un objectif très classique de la littérature est la minimisation de la date de terminaison (*makespan* en anglais) : $C_{max} = \max C_i$. On retrouve aussi la minimisation de la somme des dates de terminaison, très utile pour étudier le temps de complétion moyen des tâches.

Notons enfin que dans certains modèles les tâches ne sont pas séquentielles : elles peuvent par exemple nécessiter un nombre de processeurs fixé (*tâches rigides*), pouvoir s'exécuter sur le nombre fixe de processeurs spécifié par l'ordonnanceur (*tâches modelables*) ou être déroulées sur un nombre variable de processeurs au cours du temps (*tâches malléables*).

6.1.2 Ordonnancement avec dépendances

Le graphe de tâches, objet de notre étude, apparaît par la prise en compte de dépendances entre les différentes tâches constituant une instance d'un problème. Ces dépendances expriment une relation d'ordre dans l'exécution : si la tâche j dépend de la tâche i, alors j ne pourra démarrer (ou être ordonnancée) qu'une fois que la tâche i aura été exécutée.

Cette relation de dépendance entre les tâches est représentée très naturellement par un graphe orienté sans cycles (DAG) : les sommets représentent les tâches à exécuter, et les arêtes les dépendances. Ainsi, une arête du sommet i au sommet j indique que la tâche i précède la tâche j. Ce graphe est généralement annoté, pour prendre en compte les coûts en ressources des tâches et aussi des communications (équivalentes à des dépendances). Notons par ailleurs que certains problèmes d'ordonnancement contraignent leur instance à des structures des graphes plus simples (nous en donnons quelques exemples dans le chapitre suivant).

Pour la suite de ce document, nous introduisons un certain nombre de termes liés à cette représentation en DAG et à l'ordonnancement avec dépendances. Un graphe G est défini par $G = (V, E)$, V étant l'ensemble des sommets (ou nœuds) et E l'ensemble des arêtes. Deux sommets i et j sont reliés par une arête si $(i, j) \in E$. Dans la suite de ce document, nous ne ferons référence qu'à des graphes orientés. Dans de tels graphes, le sommet i est nommé *prédécesseur direct* (ou père) du sommet j s'il existe un *arc* de i vers j : $(i, j) \in E$. Par opposition, j est alors un *successeur direct* (ou fils) de i.

Un chemin dans un graphe orienté est une suite de sommets reliés deux à deux par un arc : $s_1, s_2, ..., s_k$ avec $(s_i, s_i + 1) \in E$. Un cycle désigne alors un chemin dont les premiers et derniers sommets sont confondus. Les graphes manipulés dans ce document étant des modèles de dépendances entre des tâches, il sont sans cycles : tout cycle de dépendance rendrait l'exécution d'une partie des tâches impossible.

Enfin, par souci de simplicité, nous utiliserons par la suite les termes de graphe pour désigner les graphes orientés sans cycles et d'arête pour désigner une relation entre deux sommets (arc). Un arête *sortante* (respectivement *entrante*) d'un sommet i désigne alors un arc dont l'origine (respectivement l'arrivée) est i. Le nombre d'arêtes sortantes d'un nœud est nommé *degré sortant* et l'opposé *degré entrant*. Le *degré* du

sommet est donc la somme du degré entrant et du degré sortant. Un sommet dont le degré sortant est nul est appelé *puits* tandis que son opposé est désigné sous le nom de *source*.

Du point de vue de l'ordonnancement, nous identifierons les tâches dont toutes les dépendances ont été résolues (prédécesseurs exécutés) par le terme de tâches *prêtes*.

6.2 Complexité et Approximabilité

Nous détaillons ici quelques résultats élémentaires sur la complexité et l'approximabilité des problèmes d'ordonnancement pour machine parallèles. Pour une étude plus détaillée de la question, voir le livre [Dro09] par exemple.

6.2.1 Généralités

La classification de Graham permet aussi de mettre en évidence une relation de complexité entre les différents problèmes d'ordonnancement. Il apparaît assez évident en effet que la généralisation d'un problème d'ordonnancement le rende plus complexe. Ainsi, les problèmes dans R sont au moins aussi difficiles à résoudre que les problèmes dans Q, puisque tout problème dans ce dernier peut être réduit à une question similaire dans R. Il en est de même pour l'ajout de contraintes de précédence entre les tâches, ou le passage d'un nombre fixé de processeurs à un nombre arbitraire (paramètre de l'instance). Il est ainsi possible d'établir une hiérarchie des problèmes d'ordonnancement en fonction de leur complexité et des réductions existantes d'un problème à l'autre.

La plupart des problèmes d'ordonnancement qui nous intéresserons plus tard sont NP-difficiles, ce qui rend généralement impossible leur résolution exacte en un temps raisonnable. Il existe néanmoins une classe de problèmes, dit NP-difficiles au sens faible pour lesquels un algorithme pseudo-polynomial existe. Un tel algorithme résout alors le problème en un temps polynomial en la taille de l'instance et de la valeur d'une partie de l'instance (intuitivement la complexité dépend du codage de l'instance).

Lorsque le problème est prouvé NP-difficile au sens fort, seule la recherche d'un algorithme d'approximation reste envisageable. Si un tel algorithme existe, sa qualité s'exprime par son ratio d'approximation : le rapport entre la solution obtenue en pire cas et la solution optimale correspondante.

Parmi les problèmes les plus classiques, notons ainsi que $P2 \mid\mid C_{\max}$ est un problème NP-difficile tandis que $P \mid\mid C_{\max}$ l'est au sens fort, de même $P \mid \text{prec} \mid C_{\max}$ (qui en est une généralisation).

6.2.2 Heuristiques

Au vu de la complexité des problèmes d'ordonnancement, la plupart des solutions apportées dans la littérature prennent la forme d'heuristiques. Il s'agit alors de construire, lorsque c'est possible, une solution avec garantie : cette dernière est éloignée de la solution optimale par un facteur constant.

Une des classes d'algorithmes les plus utilisée est celle des ordonnanceurs par liste (ou `list schedulers`). Il s'agit d'un algorithme glouton, répartissant les tâches prêtes

selon un ordre préétabli au fur et à mesure que les processeurs se libèrent. Le fonctionnement exact de ces algorithmes se résume ainsi :

Algorithm 1 Ordonnancement de liste

Require: un graphe de tâches G, un nombre de machines m.
  Construire une liste L ordonnée des tâches prêtes.
  while L non vide **do**
    for all tâche terminée **do**
      Placer successeurs dans la liste selon l'ordre
    for all processeur libre **do**
      Placer la tâche de plus haute priorité
    avancer dans le temps

Tout le fonctionnement de ce type d'algorithme repose donc sur la construction de la liste ordonnée des tâches. À titre d'exemple, l'algorithme LPT (`Longuest Processing Time`) utilise par exemple le temps d'exécution des tâches pour ce tri, de sorte que les tâches les plus longues soient exécutées en priorité. À l'opposé, SPT (`Shortest Processing Time`) trie les tâches par ordre croissant des temps d'exécution. La figure 6.2 donne un exemple de ces ordonnancements sur le même jeu de tâches.

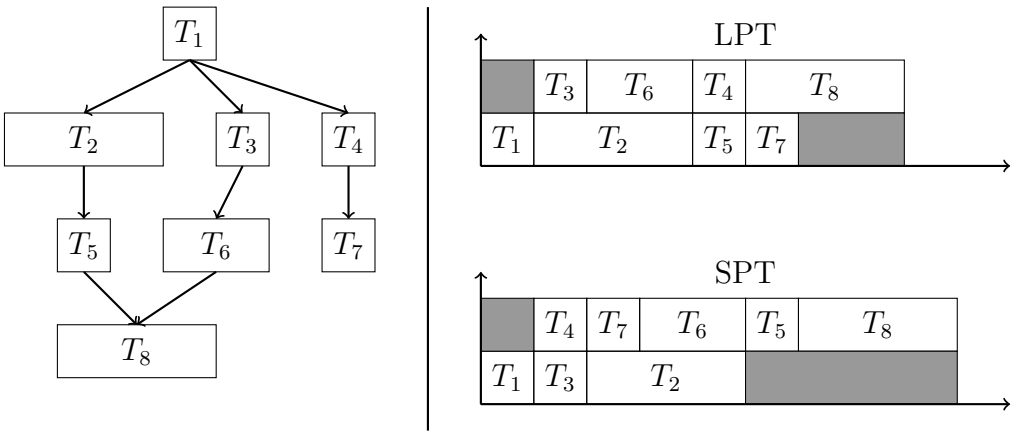


FIGURE 6.2 – Comparaison d'un ordonnancement par LPT et par SPT du même graphe de tâche sur 2 processeurs.

Vis à vis du problème $P \mid \text{prec} \mid C_{\max}$, Graham montra en 1969 [Gra69] que dans le pire des cas, un algorithme de liste donne une solution à un rapport $2 - \frac{1}{m}$ de l'ordonnancement optimal. Il est facile de prouver ce théorème par un argument de surface, en détaillant l'occupation des machines entre temps d'activité (*Act*) et temps d'inactivité des m processeurs (*Idle*) :
Si l'on considère un diagramme de Gantt d'un ordonnancement, alors la surface occupée par une solution est de $m \times C_{\max}$. Nous avons donc :

$$m \times C_{\max} = \text{Idle} + \text{Act}$$

Notons $C^*_{\max}$ le temps de complétion optimal. Au maximum, le temps d'activité des processeurs se réduit à $m \times C^*_{\max}$. Autrement dit, dans le pire des cas (en terme d'occupation), la solution optimale du problème consiste à utiliser la totalité des processeurs

en continu.

$$\text{Act} \leq mC^*_{\max}$$

De même, le temps d'inactivité des processeurs peut être borné par $(m-1) \times C^*_{\max}$. Ce résultat vient du constat suivant : $C^*_{\max}$ est borné par le chemin de plus grand poids dans le graphe de tâches. Ce chemin, que l'on appelle *chemin critique* interdit en effet tout ordonnancement terminant plus rapidement (puisqu'il faut au moins qu'un processeur l'exécute en entier et il ne peut être parallélisé). Un processeur ne peut être inactif que durant l'exécution d'une des tâches de ce chemin. En effet, si un processeur avait été disponible à un autre moment, l'algorithme de liste aurait dû lui allouer une tâche du chemin critique, puisqu'elle était forcément disponible. Au maximum, l'inactivité des processeurs correspond à un système où à chaque fois qu'un processeur exécute une tâche du chemin critique, tous les autres processeurs sont en attente. D'où :

$$\text{Idle} \leq (m-1)C^*_{\max}$$

Il suffit alors de simplifier notre première inégalité :

$$\begin{aligned} mC_{\max} &= \text{Idle} + \text{Act} \\ mC_{\max} &\leq (m-1)C^*_{\max} + mC^*_{\max} \\ C_{\max} &\leq (2 - \frac{1}{m})C^*_{\max} \end{aligned}$$

6.3 Influence d'un graphe sur la performance d'un algorithme

Il apparaît évident en analysant la preuve précédente que le chemin critique est une caractéristique à prendre en compte dans l'évaluation de performance des ordonnanceurs par liste. Nous approfondissons dans cette section cette analyse, ainsi que d'autres caractéristiques des graphes influant la performance d'un algorithme.

6.3.1 Longueur des chemins

Tout graphe orienté sans cycle contient un (ou plusieurs) chemin de longueur maximale et ce chemin est calculable en temps polynomial (avec un parcours dans l'ordre topologique du graphe par exemple). La figure 6.3 donne un exemple de graphe avec son plus long chemin.

Comme nous l'avons déjà dit, du point de vue de l'ordonnancement ce chemin présente la particularité de borner le temps de complétion d'une instance. En effet, pour exécuter la totalité d'un graphe de tâches, toutes les tâches doivent être exécutées en respectant leurs dépendances. La dernière tâche du chemin le plus long ne peut donc être exécutée tant que toutes les autres tâches du chemin ne l'ont été. Sur une infinité de processeurs, ce temps est suffisant pour exécuter tous les autres chemins présents dans le graphe, puisqu'ils sont plus courts. Cela n'est cependant vrai que si l'on considère toutes les tâches comme nécessitant le même temps processeur.

En effet, dès qu'un problème d'ordonnancement s'appuie sur des temps d'exécution arbitraires sur chaque tâche, le chemin le plus long (du point de vue de la structure du

graphe) ne correspond plus forcément au chemin critique (du point de vue de l'ordonnancement). Dans le cadre de l'analyse de graphes pour l'ordonnancement, il devient alors intéressant de prendre en compte le poids des nœuds dans le calcul du chemin le plus long (cela ne le complique en rien par ailleurs).

Néanmoins, pour certains problèmes d'ordonnancement, cela ne suffit pas. En effet, dans les problèmes avec coûts de communication par exemple, le temps d'exécution d'un chemin dépend du temps des communications entre chaque tâche. Ce temps pouvant être variable (modèles avec contention, suppression du coût si la communication est sur la même machine), il n'est pas possible de le prendre en compte dans le calcul de la longueur d'un chemin.

Dans ces derniers cas, d'autres mesures peuvent compenser le manque d'information concernant le chemin critique, comme le nombre d'arêtes ou la distribution des degrés, souvent liées à ce dernier mais plus facilement calculable.

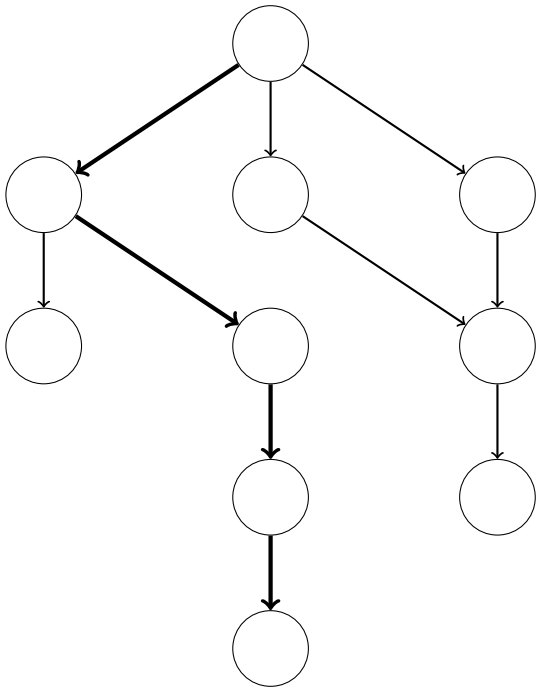


FIGURE 6.3 – Un DAG et son plus long chemin.

6.3.2 Antichaîne

Un ordonnanceur par liste ne laissant jamais un processeur libre si une tâche est prête, sa performance dépend fortement du nombre de tâches disponibles à un instant donné. Ainsi, un graphe constitué de m chaînes indépendantes et de même longueur sera facilement exécutable sur m processeurs. À l'opposé, si le graphe ne contient qu'un seul chemin, il sera difficile d'obtenir la moindre amélioration de performance en utilisant plus de processeurs.

Le *parallélisme* d'un graphe correspond ainsi à une certaine répartition des dépendances dans le graphe. Cette dernière peut se caractériser en premier abord par l'analyse des degrés des nœuds du graphe : une distribution importante de degrés sortants

élevés et de degrés entrants faibles indique clairement qu'un graphe se parallélisera convenablement.

La notion d'antichaîne est aussi capitale pour ce genre d'analyse. Une antichaîne est définie comme un ensemble de nœuds n'étant pas reliés deux à deux par une arête. Il s'agit donc d'un ensemble de nœuds qu'il est possible d'exécuter en parallèle. La longueur de la plus grande antichaîne dans un graphe orienté sans cycle est par ailleurs calculable en temps polynomial, ce qui rend son utilisation aisée dans l'analyse de la difficulté d'un ensemble de graphes de tâches. La figure 6.4 donne un exemple de graphe et de son antichaîne maximale.

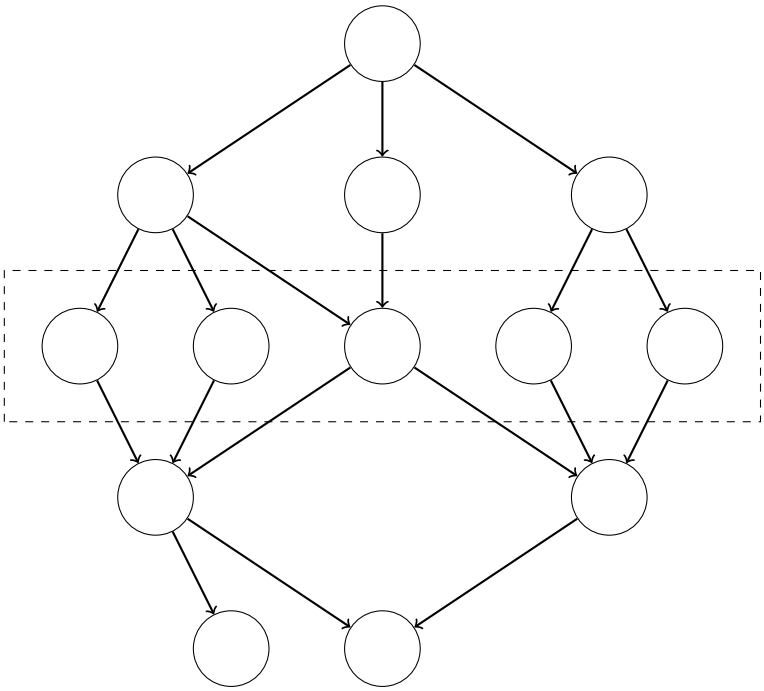


FIGURE 6.4 – Un DAG et sa plus longue antichaîne.

Génération de graphes pour la simulation d'ordonnanceurs 7

Sommaire

7.1 Générateurs et collections de graphes 96
7.1.1 Algorithmes de génération aléatoire 96
7.1.2 Collections de graphes pour la simulation 98
7.1.3 Structures classiques de graphes 99
7.2 GGen : un environnement de génération aléatoire de DAG 99
7.2.1 Organisation de l'environnement 100
7.2.2 Algorithmes de génération implémentés 101
7.3 Caractéristiques des graphes générés 103
7.3.1 Modèles d'Erdős 103
7.3.2 Layer-by-Layer 105
7.3.3 Fan-in/Fan-out 107
7.3.4 Random Orders 109
7.3.5 Conclusion 109

La validation d'un algorithme d'ordonnancement passe par de nombreuses phases. Bien sûr, une preuve théorique des performances attendues peut être apportée dans un premier temps. Mais la simulation d'un algorithme est aussi de plus en plus utilisée. Cette simulation sert plusieurs objectifs : elle permet tout d'abord de vérifier l'implémentation de l'ordonnanceur, soit par l'utilisation de jeux de données aléatoires soit en vérifiant son comportement sur des entrées connues. Mais la simulation peut aussi être utilisée pour étudier le comportement d'un algorithme dans un environnement pour lequel il n'a pas encore été étudié théoriquement.

Dans tous ces cas, l'expérimentateur a besoin de jeux de données spécifiés le plus précisément possible et d'outils pour analyser ces jeux. Or la plupart des travaux que l'on peut rencontrer dans la littérature pèchent soit par des imprécisions sur les jeux de données utilisés soit par un manque d'analyse de ces entrées. Nous détaillons donc dans ce chapitre un environnement que nous avons conçu pour générer aléatoirement des graphes de tâches et les analyser en détails. Après un tour d'horizon des outils existants pour la génération de graphes ainsi que les autres classes de jeux de données rencontrées dans la littérature, nous présentons notre environnement et les nombreux algorithmes mis en œuvre. Nous détaillons ensuite la validation de cet environnement, à travers l'analyse précise des caractéristiques des graphes générés.

Nbre de nœuds	1	2	3	4	5	6	7	8	9	10
Nbre de DAGs	1	2	6	31	302	5984	243668	20286025	3424938010	1165948612902

Table 7.1 – Nombre de graphes dirigés acycliques non étiquetés.

7.1 Générateurs et collections de graphes

De nombreux domaines de l'informatique utilisent des générateurs de graphes. En simulation des réseaux par exemple, le problème de la modélisation de la topologie d'un réseau par un graphe est sujet d'un grand nombre de travaux. Les outils de visualisation de graphes peuvent aussi avoir besoin de tels générateurs pour valider leur implémentation. Nous ne détaillerons néanmoins ici que les générateurs et collections directement liés à la simulation d'ordonnanceurs.

Idéalement, un ordonnanceur devrait être validé face à toutes les configurations d'entrées possibles. Cela requiert un générateur aléatoire de graphes capable de générer toutes les structures de graphes possibles avec la même probabilité. La notion de *structure de graphes* est formellement définie par l'isomorphisme de graphes. Ainsi, deux graphes G et H sont isomorphes s'il existe une association $\varphi : V(G) \to V(H)$ telle que $(u, v) \in E(G) \iff (\varphi(u), \varphi(v)) \in E(H), \forall u, v \in V(G)$. Un algorithme d'ordonnancement générera probablement la même solution pour deux graphes isomorphes, même si les annotations sont différentes. Par exemple, deux graphes isomorphes dont les annotations sont tirées aléatoirement selon une distribution uniforme induiront généralement la même performance pour l'ordonnanceur (cela revient à réaliser deux tirages des annotations sur un seul graphe).

Malheureusement, le nombre de graphes dirigés acycliques non isomorphes rend inapplicable toute méthode de génération basée sur l'énumération de l'ensemble des graphes possibles, même pour un nombre de nœuds extrêmement faible. La table 7.1 donne le nombre de DAGs non isomorphes comprenant jusqu'à 10 nœuds (information reproduite depuis la séquence A003087 dans [SP95]). Il n'existe pas, pour autant que nous le sachions, d'algorithme capable de générer uniformément de tels graphes.

7.1.1 Algorithmes de génération aléatoire

Une génération parfaitement aléatoire de graphes non isomorphes n'étant pas disponible, les concepteurs d'algorithmes d'ordonnancement ont recours à des générateurs spécifiques aux types d'entrées pour lesquelles ces derniers ont été conçus.

Niveau par niveau

Un des générateurs les plus connus dans cette catégorie est l'algorithme *niveau par niveau* (Layer-by-Layer), conçu spécifiquement pour valider des algorithmes d'ordonnancement sur machines parallèles [YG94]. Il repose sur un concept de *niveau* : un groupe de nœuds indépendants dans le graphe, tel que s'il existe un chemin d'un nœud du niveau a à un nœud du niveau b il n'existe aucun chemin d'une tâche de b vers une de a. L'algorithme repose ensuite sur le tirage aléatoire des arêtes possibles du graphe, de façon à ce que chaque arête potentielle existe avec probabilité p. Nous donnons ici

l'algorithme complet, en utilisant la fonction `Random()` qui retourne un nombre réel distribué uniformément sur l'intervalle $[0, 1[$.

Algorithm 2 Algorithme Layer-by-Layer.

Require: $n, k, p \in \mathbb{N}$.
  Distribuer n nœuds entre k ensembles différents notés $L_1, \ldots, L_k$.
  Soit $layer(v)$ le niveau du nœud v.
  Soit M une matrice d'adjacence $n \times n$ initialisée à zéro.
  for i = 1 **to** n **do**
    for j = 1 **to** n **do**
      if $layer(j) > layer(i)$ **then**
        if Random() < p **then**
          M[i][j] = 1
        else
          M[i][j] = 0
  return un DAG aléatoire avec k niveaux et n nœuds.

Cette méthode, bien que simple, est très pratique pour contrôler la difficulté des graphes générés : le nombre de niveaux limite la longueur du chemin critique. Elle correspond aussi intuitivement à une succession d'étages de calculs indépendants, une structure que l'on retrouve dans certaines applications parallèles.

Algorithme G(n, p)

Paul Erdős and Alfréd Rényi ont défini en 1959 [ER59] une méthode très simple pour générer un graphe aléatoirement. Cette méthode a depuis été adaptée aux DAGs [AVAM92], et est généralement référencée sous le nom de G(n, p). Pour un nombre n de nœuds, l'algorithme G(n, p) génère un DAG dont chacune des $\frac{n(n-1)}{2}$ arêtes possibles est présente avec probabilité p.

Algorithm 3 Algorithme G(n, p)

Require: $n \in \mathbb{N}, p \in \mathbb{R}$.
Ensure: un graphe avec n nœuds.
  Soit M une matrice d'adjacence $n \times n$ initialisée à zéro.
  for i = 1 **to** n **do**
    for j = i+1 **to** n **do**
      if Random() < p **then**
        M[i][j] = 1
      else
        M[i][j] = 0
  return le DAG représenté par M.

Cet algorithme possède un grand nombre de propriétés connues, sur lesquelles nous reviendrons dans la partie implémentation.

Task Graphs For Free

Une des seules méthodes de génération aléatoire de graphes dont il existe une implémentation publiquement accessible nous vient de Dick *et al.* [DRW98]. Cette méthode construit par incréments le graphe résultat en alternant aléatoirement entre deux phases : une d'expansion du graphe et l'autre de contraction. L'objectif de cette méthode est d'obtenir plus de contrôle sur les degrés entrants et sortants des nœuds. Nous décrivons ici une version ayant comme paramètre le nombre maximal de sommets du graphe plutôt que de le choisir aléatoirement comme dans la présentation originale.

Algorithm 4 Algorithme Fan-in/Fan-out

Require: $n, id, od \in \mathbb{N}$.
Ensure: un graphe avec au moins n nœuds, où chaque sommet possède un degré sortant $\leq od$ et un degré entrant $\leq id$.
Initialiser $G = (V, E)$, avec $E = \emptyset$ et $V = \emptyset$.
Ajouter un nœud dans G.
while $|V| \leq n$ **do**
  if Random() < 0.5 **then** {Expansion}
    Trouver le sommet v avec la plus grande différence entre son degré sortant et od. Soit m_o cette différence.
    Ajouter un nombre aléatoire de sommets (entre 1 et m_o) à V et les arêtes de v aux nouveaux nœuds.
  else {Contraction}
    Trouver l'ensemble S des nœuds avec un degré sortant $< od$.
    Sélectionner un sous ensemble T de S de taille maximale id.
    Ajouter un nouveau sommet v et les arêtes (t, v) pour chaque $t \in T$.

Les phases de contraction et d'expansion sont ici un modèle du comportement de certaines applications parallèles, qui contiennent des communications regroupant les résultats avant de redéployer du calcul.

7.1.2 Collections de graphes pour la simulation

Si les algorithmes de génération aléatoires peuvent mettre en évidence des erreurs subtiles dans un ordonnanceur, la simulation face à des jeux de données bien identifiées possède aussi des avantages. Par exemple, il est bien plus facile de comparer la performance de deux travaux si les simulations ont utilisé les mêmes entrées.

Une de ces collections fut proposée en 2002 par Tobita et Kasahara [TK02] : le `Standard Task Graph Set`. Il s'agit là d'une collection de graphes générés selon des méthodes bien connues et censées être représentatives des graphes de tâches que l'on peut trouver dans un système en production. Ces méthodes comportent des générateurs aléatoires, notamment $\mathsf{G(n,p)}$ et Layer-by-Layer mais aussi des générateurs déterministes, reproduisant la structure d'un algorithme parallèle particulier. L'intérêt principal de cette collection réside néanmoins ailleurs, dans la mise à disponibilité de solutions optimales pour certains problèmes d'ordonnancement pouvant utiliser ces graphes. Ces solutions étant parfois difficiles à calculer, leur mise à disponibilité facilite grandement le travail d'autres chercheurs.

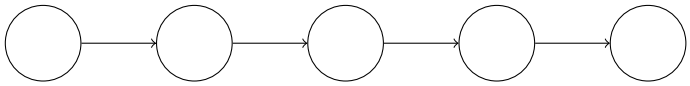


FIGURE 7.1 – Chaîne de 5 tâches.

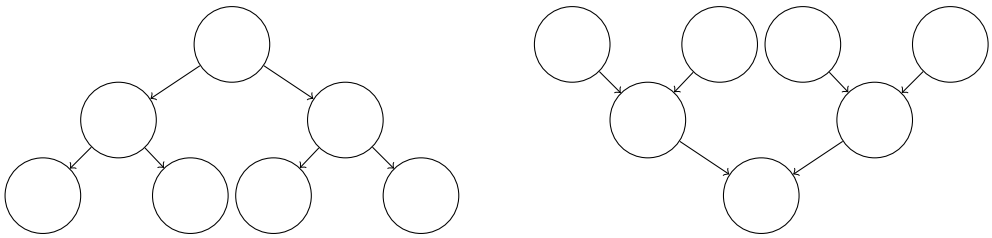


FIGURE 7.2 – Arbre (à gauche) et anti-arbre (à droite) de 7 tâches.

Notons aussi l'existence de bases de données d'instances tirées de systèmes en production, qui peuvent alors être traduites en graphes de tâches par les expérimentateurs intéressés. Une des bases les plus connue est certainement la Parallel Workloads Archive [Fei09], maintenue by Dror Feitelson. Elle contient les enregistrements des soumissions de travaux sur de nombreuses machines parallèles, et détaille par exemple des caractéristiques comme les dates d'arrivées, le nombre de processeurs utilisés, le temps d'exécution, la quantité de mémoire utilisée.

7.1.3 Structures classiques de graphes

Enfin, de nombreux problèmes d'ordonnancement limitent la structure des graphes à quelques formes très régulières. Les diverses propriétés de ces structures permettent alors de simplifier le problème, ainsi que la conception et la validation d'ordonnanceurs.

La chaîne de tâches désigne ainsi un ensemble de tâches dont les dépendances forment une chaîne : chaque tâche ne possède qu'un prédécesseur et un successeur (mis à part un sommet source et un sommet puits). La figure 7.1 illustre cette structure. Les problèmes d'ordonnancement font généralement référence à des groupes de structures de ce type, de taille variable.

Deux autres structures très populaires sont les arbres et anti-arbres. Un arbre est un graphe dont les sommets ont au maximum un degré entrant de 1, tandis qu'un anti-arbre se caractérise par un degré sortant au maximum de 1. A noter qu'il existe aussi des variantes de problèmes d'ordonnancement pour des graphes au degré limité arbitrairement. La figure 7.2 illustre ces deux structures arborescentes.

Dernière structure particulière, le graphe *fork-join* reproduit un schéma d'application parallèle très classique : une phase de déploiement de travaux indépendants (*fork*) suivie d'un regroupement des résultats (*join*). Cette structure est illustrée en figure 7.3.

7.2 GGen : un environnement de génération aléatoire de DAG

Le premier reproche que l'on peut adresser aux différentes méthodes de génération régulièrement utilisées dans la littérature est l'absence d'implémentation de référence

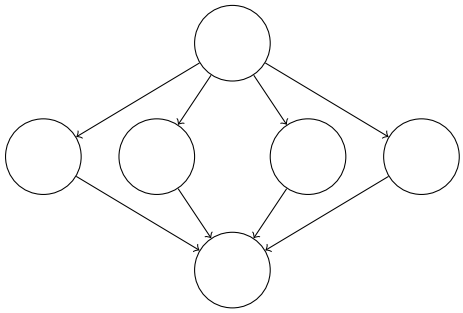


FIGURE 7.3 – *Fork-join* de 6 tâches.

accessible publiquement. Cette absence crée plusieurs problèmes dans le travail de recherche autour d'un ordonnanceur. Tout d'abord, sans implémentation publique il n'est pas possible de s'assurer que les expérimentations d'un article ne contiennent pas d'erreur d'implémentation. Il en effet rare que les auteurs discutent de la validation de leur implémentation. Ensuite, mise à part pour les propriétés connues des algorithmes utilisés, il est difficile d'analyser en détails si la méthode de génération choisie n'introduit pas de biais dans une comparaison d'algorithmes par exemple. Nous montrerons dans le prochain chapitre que de tels biais peuvent être complexes à détecter. Enfin certains algorithmes, dans leur présentation originale, présentent des imprécisions qui n'apparaissent que lors de leur implémentation. Il est alors nécessaire de disposer d'une référence pour normaliser les implémentations et éviter des incohérences entre les travaux de différents expérimentateurs.

GGen est donc un environnement de génération et d'analyse de graphes orientés sans cycles destiné à la simulation d'ordonnanceurs. Nous présentons ici son organisation et les algorithmes qui y sont implémentés.

7.2.1 Organisation de l'environnement

La majorité des problèmes d'ordonnancement, et donc des instances en entrée d'un algorithme, comportent des annotations sur les nœuds ou les arêtes. Nous avons choisi de séparer la génération de ces annotations de la génération des structures de graphes. Cette séparation permet entre autre de réutiliser une collection de structures d'une campagne de simulation à l'autre, et d'observer l'influence de la distribution des poids sur les résultats (méthode notamment utilisée dans le chapitre 8). GGen considère donc séparément les graphes et les *propriétés* ou annotations de ces graphes.

Notre environnement est organisé en cinq parties :

- Génération de graphes : une implémentation des algorithmes de génération aléatoire utilisés dans la littérature, sous la même interface.
- Analyse de graphes : des méthodes d'analyse classiques comme le chemin le plus long, l'arbre couvrant minimal ou les distributions des degrés.
- Annotations aléatoires : l'ajout à un graphe d'une propriété, soit sur les nœuds soit sur les arêtes, en suivant une distribution aléatoire (pareto, exponentielle, gaussienne,...).
- Analyse des annotations : extraction de quelques statistiques sur les annotations présentes dans le graphe, comme la moyenne, l'écart type, le minimum et le

maximum d'une propriété.
- Transformation de graphes : il est courant de transformer certains graphes de façon à ce qu'ils ne possèdent qu'une unique source ou un seul puits. Toutes les méthodes de génération ne résultant pas en ce type de graphes, GGen est capable de rajouter ces nœuds supplémentaires.

Un des objectifs majeurs de cet environnement étant de fournir une implémentation de qualité des méthodes de génération aléatoire, GGen repose sur des bibliothèques reconnues pour ce qui est des structures de données manipulées et des générateurs aléatoires utilisés au cœur des algorithmes. La bibliothèque C Igraph [CN06], conçue pour la recherche dans les réseaux, fournit ainsi des structures de graphes efficaces, certains des algorithmes d'analyse et la possibilité de lire et d'écrire les graphes depuis/vers un grand nombre de format de fichiers tandis que la bibliothèque GSL (GNU Scientific Library) [Gou09] fournit les générateurs de nombres aléatoires nécessaires.

Toutes ces fonctionnalités sont accessibles à travers deux interfaces : une bibliothèque C permettant d'inclure GGen dans un programme complet, et une interface en ligne de commande destinée à simplifier la création et la manipulation de jeux de données complets pour une campagne de simulation. Cette deuxième interface fonctionne à travers un format de description de graphe simple et largement répandu : DOT [GN00]. Chaque invocation du programme manipulant un graphe, il est ainsi possible d'enchainer les commandes pour créer, générer et annoter les graphes.

7.2.2 Algorithmes de génération implémentés

En plus des algorithmes listés précédemment ($\mathsf{G}(\mathsf{n},\mathsf{p})$, Layer-by-Layer, Fan-in/Fan-out), GGen implémente deux générateurs de graphes que l'on peut retrouver dans la littérature.

Modèle $G(n, M)$

Cette méthode de génération est une version alternative du modèle $\mathsf{G}(\mathsf{n},\mathsf{p})$: au lieu de considérer chaque arête comme présente avec probabilité p, un nombre M d'arêtes sont choisies uniformément parmi toutes celles possibles. Bien que les deux modèles soient complètement équivalents (il est possible de calculer p en fonction d'un M voulu), cette méthode est plus simple d'utilisation lorsqu'un expérimentateur souhaite un nombre d'arêtes précis dans les graphes générés. Un algorithme simple pour cette méthode est décrit ci-après, avec `RInt(n)` la fonction qui retourne un entier tiré uniformément entre 0 et n.

Algorithm 5 Algorithme G(n, M)

Require: $n, M \in \mathbb{N}$.
Ensure: un graphe avec n nœuds.
  Soit A une matrice d'adjacence $n \times n$ initialisée a zéro.
  e = 0
  while e < M **do**
    i = RInt(n)
    j = RInt(n)
    if i < j **and** $M[i][j] \neq 1$ **then**
      M[i][j] = 1
      e = e + 1
  return le DAG représenté par M.

Notons que cet algorithme perd en efficacité si le nombre d'arêtes demandé est élevé : au fur et à mesure que l'on ajoute une arête, la probabilité de vouloir en rajouter une déjà présente augmente. Pour pallier ce problème, une solution élégante consiste à l'inverser si le paramètre M est supérieur à $\frac{n(n-1)}{4}$. Au lieu de rajouter M arêtes à une matrice d'adjacence vide, il suffit de supprimer des arêtes dans la matrice d'adjacence pleine (bien sûr il ne faut en supprimer que $\frac{n(n-1)}{2} - M$).

Ordres aléatoires

La méthode Random Orders [Win85] provient d'un domaine proche de celui des DAG : l'étude des ordres partiels sur un nombre fini d'éléments. En effet, tout graphe orienté sans cycle peut être interprété comme un ordre partiel entre les sommets : une arête de i vers j signifie que i est inférieur à j (on parle de l'ordre topologique du graphe). Inversement, un ordre partiel peut être traduit en DAG, en considérant qu'il n'existe une arête entre deux sommets que s'ils sont comparables. La méthode Random Orders génère donc un graphe en deux étapes : un ordre partiel est généré par intersection d'un nombre configurable d'ordre totaux générés aléatoirement puis cet ordre partiel est traduit en graphe. L'intersection d'un ensemble d'ordres totaux est définie comme la relation pour laquelle deux éléments ne sont comparables que s'ils sont dans le même ordre pour tous les ordres intersectés. On appelle dimension d'un ordre partiel P le nombre minimal d'ordres totaux qu'il faut intersecter pour obtenir P.

Algorithm 6 Algorithme Random Orders

Require: $n, k \in \mathbb{N}$.
Ensure: un graphe avec n nœuds traduit d'un ordre de dimension au plus k.
  Générer k ordres totaux (permutations aléatoires des n sommets).
  Intersecter les k ordres générés pour obtenir un ordre partiel.
  return la traduction de l'ordre partiel en graphe.

La traduction d'un ordre partiel en DAG n'est pas unique : toute une classe de graphes peut être générée selon que l'on traduit l'ensemble des relations entre les sommets ou non. Par exemple, si l'ordre partiel contient $a < b$ et $b < c$, les arêtes (a, b)

et (b, c) doivent être rajoutées mais on peut choisir de ne pas inclure (a, c) (qui existe par transitivité). GGen implémente cette traduction en générant la totalité des arêtes possibles : le DAG généré est la fermeture transitive de l'ensemble des graphes possibles. Au vu de la méthode de génération, il s'agit de la solution la plus efficace sans compter que la boite à outils DOT nécessaire à l'interprétation du format fournit un outil d'élimination des arêtes superflues (`tred` pour réduction transitive).

Notons que ce modèle de génération aléatoire de graphes est équivalent à un algorithme ordonnant n points placés aléatoirement dans un espace de dimension k (une arête n'existe que si les coordonnées d'un point sont toutes inférieures strictement à celles d'un autre point).

7.3 Caractéristiques des graphes générés

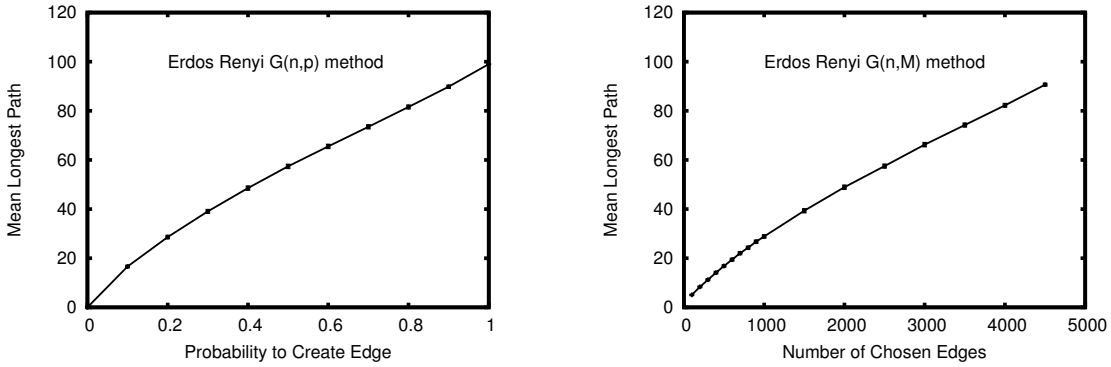
Les différentes méthodes de génération disponibles dans notre environnement produisent des structures radicalement différentes. Puisque la performance d'un algorithme d'ordonnancement peut dépendre des caractéristiques des graphes en entrée, il nous parait important de réaliser une étude de chacune des méthodes de génération. Nous analysons ici trois caractéristiques essentielles : le chemin le plus long, le nombre d'arêtes et le degré sortant.

Le chemin le plus long d'un graphe, ou *chemin critique*, est défini comme le chemin contenant le plus de nœuds dans le graphe. Nous avons déjà discuté de son importance dans la performance d'algorithmes d'ordonnancements, puisqu'il constitue une borne inférieure sur le temps de complétion d'un graphe. Le nombre d'arêtes est lui aussi important vis à vis d'un ordonnanceur : facile à mesurer, il donne une indication sur la complexité du graphe. La proportion d'arêtes par rapport au nombre de tâches est ainsi souvent utilisée pour évaluer la difficulté d'une instance. Enfin, la distribution du degré sortant permet d'estimer le nombre de processeurs utilisables par un ordonnanceur.

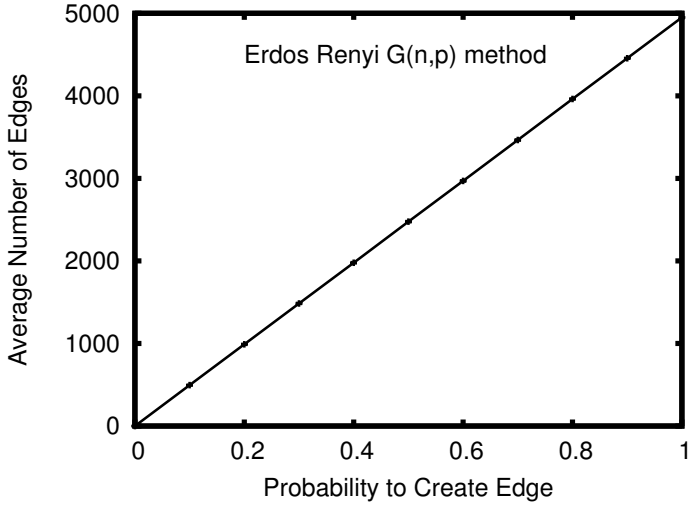
Pour chacune des méthodes de génération, nous avons généré pour chaque paramètre un millier de graphes avec exactement 100 nœuds (sauf pour Fan-in/Fan-out, qui donne environ 102 ± 2 nœuds). Ces mesures ont été réalisées à l'aide des algorithmes d'analyse de GGen. Chaque figure est représentée avec des intervalles de confiance à 95% mais ceux-ci étant très petits, il n'apparaissent clairement que sur la figure 7.6c. Notons enfin que la longueur du chemin le plus long est indiquée dans la suite en nombre d'arêtes et non pas de nœuds (un chemin de 100 sommets est donc de longueur 99).

7.3.1 Modèles d'Erdős

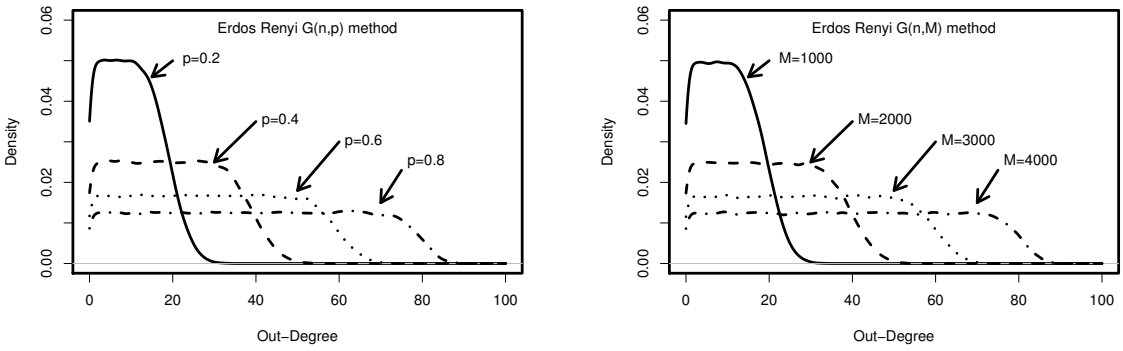
Les algorithmes $\mathsf{G}(\mathsf{n}, \mathsf{p})$ et $\mathsf{G}(\mathsf{n}, \mathsf{M})$ étant très proches, nous les analysons conjointement. Tout d'abord, la figure 7.4a donne la longueur moyenne du chemin le plus long sur ces deux méthodes, en fonction de la probabilité d'apparition d'une arête pour $\mathsf{G}(\mathsf{n}, \mathsf{p})$, et du nombre d'arêtes sélectionnées pour $\mathsf{G}(\mathsf{n}, \mathsf{M})$. Bien évidemment, plus les paramètres p ou m sont élevés et plus le chemin le plus long est grand. Certains cas particuliers sont gérés sans déroulement complet de l'algorithme par GGen, notamment pour $p = 0$ et $p = 1$ qui apparaissent respectivement comme le graphe sans arêtes (et donc un chemin critique de 0) et le graphe *complet* (chemin critique de 99). Le parallèle



(a) Chemin critique



(b) Nombre d'arêtes



(c) Distribution des degrés sortants

FIGURE 7.4 – Caractéristiques des graphes provenant de $\mathsf{G}(\mathsf{n},\mathsf{p})$ et $\mathsf{G}(\mathsf{n},\mathsf{M})$.

entre les deux algorithmes apparaît aussi nettement : par exemple, une probabilité de $0,2$ correspond à environ 1000 arêtes dans le graphe pour $\mathsf{G(n,p)}$. Or en observant les deux courbes, nous pouvons constater que $\mathsf{G(n,M)}$ donne la même longueur de chemin critique pour 1000 arêtes que $\mathsf{G(n,p)}$ pour $p = 0,2$.

La figure 7.4b fait le parallèle entre le paramètre p et le nombre d'arêtes dans le graphe résultat, illustrant cette relation entre les deux modèles. Notons que cette relation peut aussi être extraite d'une analyse simple de $\mathsf{G(n,p)}$: l'espérance du nombre d'arêtes d'un graphe est de p fois le nombre maximal d'arêtes possibles, autrement dit $p \times \frac{n(n-1)}{2}$.

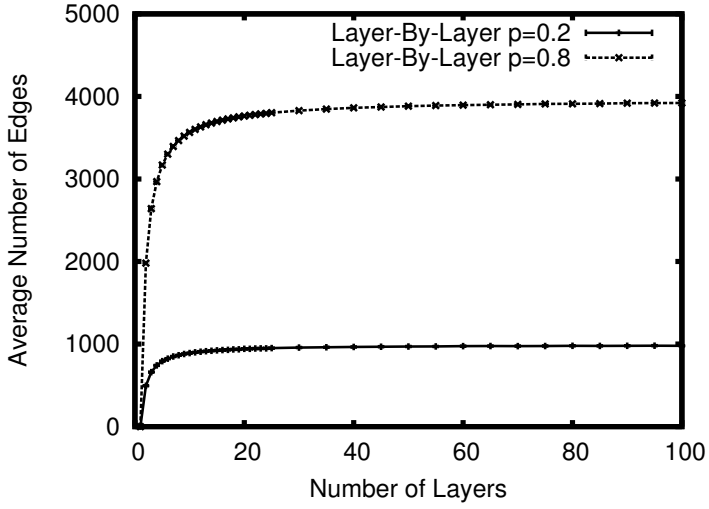
Enfin, la figure 7.4c montre côte à côte plusieurs distributions des degrés sortants des deux méthodes, et illustre à nouveau l'équivalence entre ces dernières. Il apparaît sur ces distributions que nos deux méthodes résultent en un nombre équivalent de sommets avec un degré sortant entre 1 et $p(n-1)$. Ce phénomène s'explique facilement sur la méthode $\mathsf{G(n,p)}$: dans la matrice d'adjacence considérée par l'algorithme, le sommet i peut posséder $n-i$ arêtes sortantes. Le nombre véritable d'arêtes créées suit donc une distribution binomiale $B(n-i,p)$ (loi correspondant au nombre de succès dans une suite de $n-i$ tests à probabilité p de réussite). La distribution du degré sortant d'une collection de graphes correspond donc à un processus de choix aléatoire uniforme de i entre 0 et $n-1$ et tirage suivant la binomiale $B(n-i,p)$. Ce qui donne une courbe avec un plateau central, les proportions de degré sortant étant approximativement les mêmes entre 1 et $p(n-1)$.

7.3.2 Layer-by-Layer

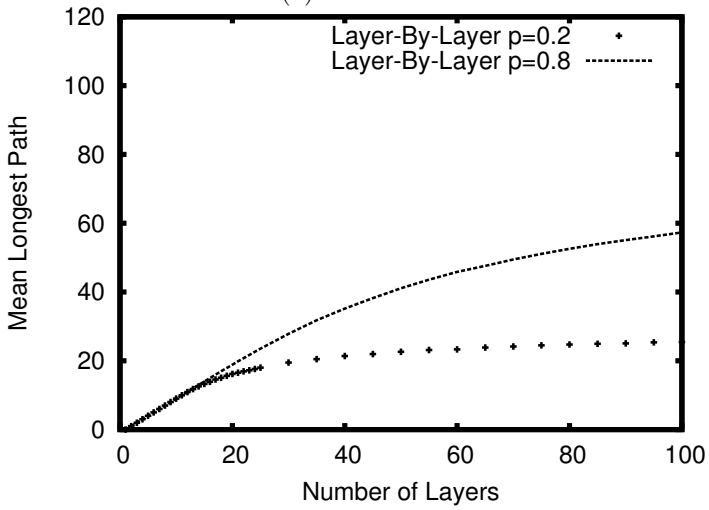
L'algorithme de niveau par niveau qu'implémente GGen est très proche de $\mathsf{G(n,p)}$ dans son fonctionnement. Au lieu de considérer toutes les arêtes du DAG complet comme équiprobables, Layer-by-Layer commence par répartir équitablement les sommets du graphes entre les différents niveaux. Chaque niveau correspondant à un certain nombre de nœuds ne pouvant partager une arête, le nombre de niveaux donné en paramètre du générateur influe directement sur le nombre d'arêtes possibles. La figure 7.5a illustre l'évolution de ce dernier en fonction du nombre de niveaux et de la probabilité d'apparition d'une arête.

Nous pouvons constater que le nombre d'arêtes atteint très vite un plateau quel que soit le nombre de niveaux, et que le paramètre p possède une influence forte sur le nombre d'arêtes. Ces courbes peuvent s'expliquer par une analyse théorique de l'algorithme Layer-by-Layer. La première étape de ce dernier répartit uniformément les sommets du graphe dans k niveaux. Il y a donc en moyenne $\frac{n}{k}$ nœuds par niveau. Le premier niveau peut ensuite être originaire de $\frac{n}{k}(n-\frac{n}{k})$. En effet, chaque nœud du niveau peut posséder une arête vers tous les sommets des niveaux inférieurs. En étendant cette analyse à l'ensemble du graphe nous obtenons que le nombre d'arêtes possibles au total est de approximativement $\frac{n^2(k-1)}{2k}$. L'espérance du nombre d'arêtes des graphes générés est donc de p fois ce nombre. Le terme $\frac{k-1}{2k}$ tendant rapidement vers $\frac{1}{2}$ quand k augmente, le nombre d'arêtes des graphes générés atteint rapidement un plateau autour de $p\frac{n^2}{2}$.

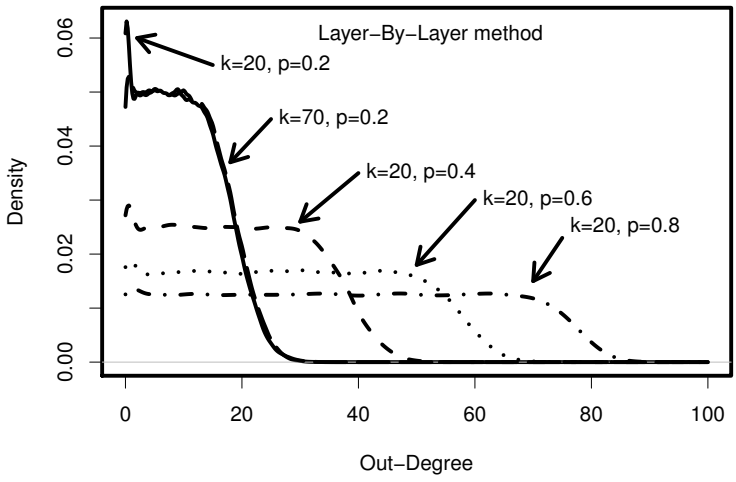
La figure 7.5b présente ensuite la longueur du chemin le plus long pour un paramètre p de 0,2 et 0,8. Cette dernière est limitée fortement par le nombre de niveaux :



(a) Nombre d'arêtes



(b) Chemin critique



(c) Distributions des degrés sortants

FIGURE 7.5 – Caractéristiques des graphes provenant de Layer-by-Layer.

les plus longs chemins permis par l'algorithme Layer-by-Layer ne peuvent en effet que traverser un sommet par niveau au maximum. De plus, les arêtes étant réparties uniformément entre les nœuds, rien ne garantit que de tels chemins existent quelle que soit la probabilité d'apparition des arêtes. La moyenne du chemin le plus long est donc nettement inférieure au nombre de niveaux.

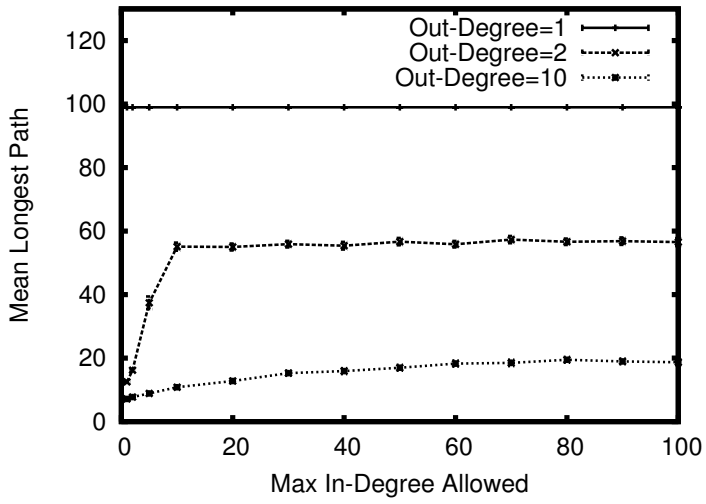
Enfin, la distribution des degrés sortants sur les graphes générés est présentée en figure 7.5c pour différents nombres de niveaux et probabilités. Nous retrouvons là des courbes similaires aux méthodes $\mathsf{G}(\mathsf{n},\mathsf{p})$ et $\mathsf{G}(\mathsf{n},\mathsf{M})$, à la différence d'une proportion plus importante de sommets sans arêtes. Ce phénomène peut s'expliquer par le fonctionnement des niveaux, qui réduisent fortement le nombre potentiel d'arêtes sur certains sommets (notamment ceux des derniers niveaux). Notons par ailleurs que le nombre de niveaux influe peu sur cette distribution, les deux courbes pour $p = 0,2$ étant très proches l'une de l'autre.

7.3.3 Fan-in/Fan-out

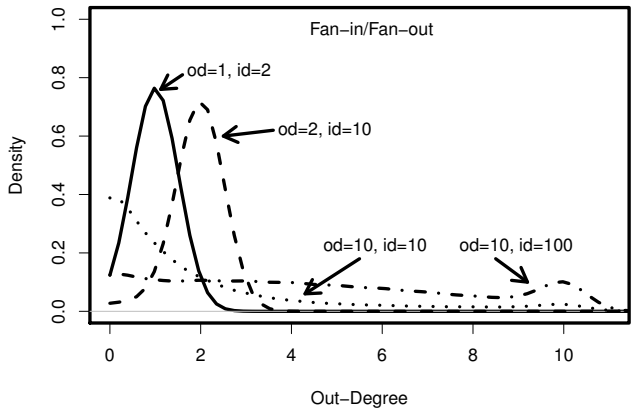
L'algorithme Fan-in/Fan-out décrit précédemment permet la génération de graphes en fixant des degrés entrants et sortants maximaux pour l'ensemble des sommets. Ces contraintes engendrent naturellement des limites sur la longueur du chemin le plus long. Nous présentons quelques exemples de cette influence dans la figure 7.6a. Trois courbes apparaissent sur cette figure. La courbe du haut indique que lorsque le degré sortant maximal autorisé est de 1, la méthode ne génère que des chaînes : tous les nœuds ne forment qu'un seul chemin de longueur maximale. Les deux courbes indiquent que cette structure en chaîne se dégrade rapidement avec l'augmentation du degré sortant limite. Plus ce dernier est élevé, et plus Fan-in/Fan-out génère des graphes avec un chemin critique court. Cela s'explique en partie par la façon dont l'algorithme choisit les nœuds auxquels rajouter des arêtes. En effet, lors d'une phase d'expansion, les sommets possédant le degré le plus faible sont sélectionnés en priorité ce qui a pour effet de *tasser* le graphe (l'expansion se fait plus souvent en largeur qu'en profondeur).

Ce phénomène est confirmé par l'observation de la distribution des degrés sortants en figure 7.6b. Lorsque le degré sortant maximal autorisé est faible, Fan-in/Fan-out sature la plupart des sommets en arêtes (c'est notamment le cas pour $FiFo(1,2)$ et $FiFo(2,10)$). L'augmentation de cette limite entraîne néanmoins une répartition plus uniforme des arêtes entre les différents nœuds : toutes les valeurs de degré sortant autorisées apparaissent sur $FiFo(10,10)$ et $FiFo(10,100)$. Notons d'ailleurs que lorsque le degré entrant maximal est élevé les phases de contractions génèrent un grand nombre d'arêtes, ce qui peut augmenter sensiblement la proportion de sommets avec un degré sortant important.

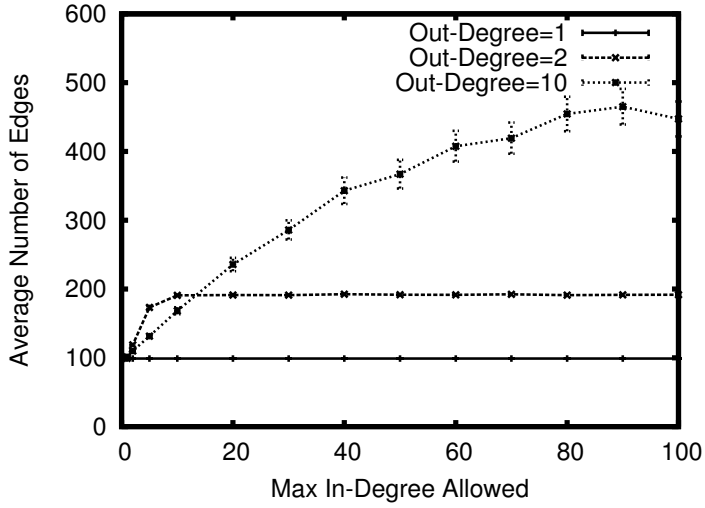
De manière générale, Fan-in/Fan-out produit des graphes comprenant peu d'arêtes. Nous avons pu voir que les degrés sortants des sommets étaient relativement faibles, et cela est confirmé par la figure 7.6c. Nous pouvons observer que pour de petites valeurs de od le nombre d'arêtes ne change pas en augmentant id. Ce comportement est parfaitement logique, puisqu'un graphe ne peut posséder sur cette méthode plus de $n \times od$ arêtes. En revanche, il est plus difficile d'expliquer le très faible nombre d'arêtes présentes pour des paramètres plus larges. La justification la plus raisonnable reste de constater que Fan-in/Fan-out ne créé des arêtes que par l'ajout de nœuds dans le



(a) Chemin critique



(b) Distributions des degrés sortants



(c) Nombre d'arêtes

FIGURE 7.6 – Caractéristiques des graphes provenant de Fan-in/Fan-out.

graphe. Ainsi, une phase d'expansion résulte au maximum en *od* nouvelles arêtes (et *id* nouvelles pour la phase de contraction). Ce nombre est bien inférieur au nombre d'arêtes qui pourraient être créées lors de l'ajout d'un nœud dans un DAG quelconque, et donc le nombre d'arêtes créées au total est faible pour Fan-in/Fan-out.

7.3.4 Random Orders

En intersectant un certain nombre d'ordres totaux, l'algorithme Random Orders fonctionne en réalité par destruction du DAG contenant toutes les arêtes possibles. Ainsi, ce dernier est généré si un seul ordre est utilisé. L'intersection d'un deuxième ordre total, permutation aléatoire du premier, a pour effet mécanique de supprimer un nombre important d'arêtes, et il en est de même pour tout ajout d'un autre ordre total. Ce phénomène est illustré par la figure 7.7a qui donne le nombre moyen d'arêtes dans les graphes générés en fonction du nombre d'ordres intersectés.

Peter Winkler, premier utilisateur de ce modèle de génération d'ordres partiels, a longuement étudié certaines caractéristiques des ordres générés. Une des questions les plus étudiées nous concerne directement : quelle est la longueur de la plus grande sous séquence croissante d'éléments ? Cette longueur, aussi appelée *hauteur*, désigne une succession d'éléments formant un ordre total, s'il sont considérés à part. Il s'agit donc de la longueur du chemin critique des graphes correspondants à l'ordre généré. Dans un article de 1988 [BW88], Winkler et Bollobás ont prouvé le théorème suivant :

- Soit $H_k(n)$ la longueur du chemin critique pour un graphe avec n sommets générés par intersection de k ordres,
- Il existe des constantes $c_1, c_2, \ldots$ telles que $c_k < e$ pour tout k et $\lim_{k\to\infty} c_k = e$,
- $\lim_{n\to\infty} H_k(n) = c_k n^{1/k}$.

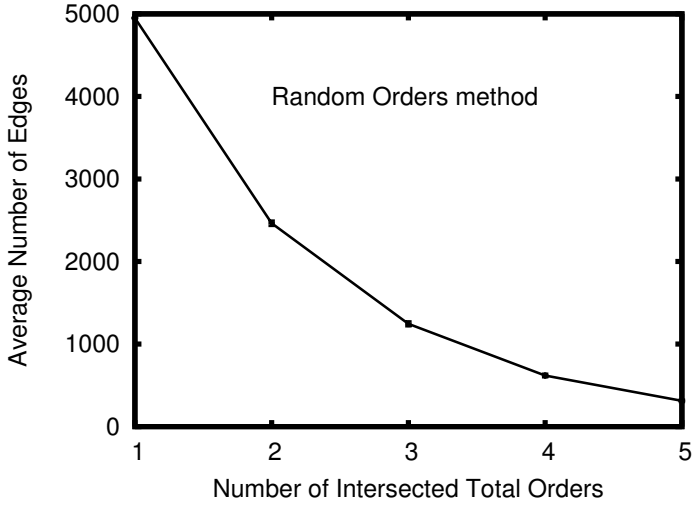
Ce résultat nous donne une certaine estimation de la longueur du chemin le plus long atteignable à l'aide du nombre d'ordres intersectés. Malheureusement, seules les valeurs des deux premières constantes sont connues : $c_1 = 1$ et $c_2 = 2$, et la preuve que ces constantes croissent avec l'augmentation de k n'est pas apportée. La figure 7.7b illustre nos mesures de cette longueur pour un nombre d'ordres entre 1 et 5.

Enfin, la figure 7.7c présente la distribution des degrés sortants sur les graphes générés pour différentes valeurs de k. Il apparaît encore que l'augmentation du nombre d'ordres totaux intersectés influe fortement sur le degré des nœuds, conséquence naturelle de la diminution du nombre d'arêtes dans les graphes générés.

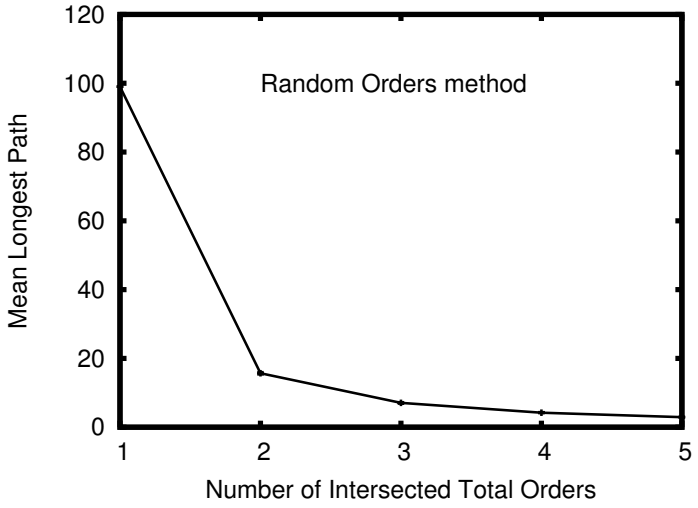
7.3.5 Conclusion

Il apparaît évident au vu des analyses précédentes que ces différentes méthodes de génération produisent des graphes bien distincts. Il est donc important de comprendre les particularités de chaque méthode lors de leur utilisation pour l'analyse d'algorithmes d'ordonnancement.

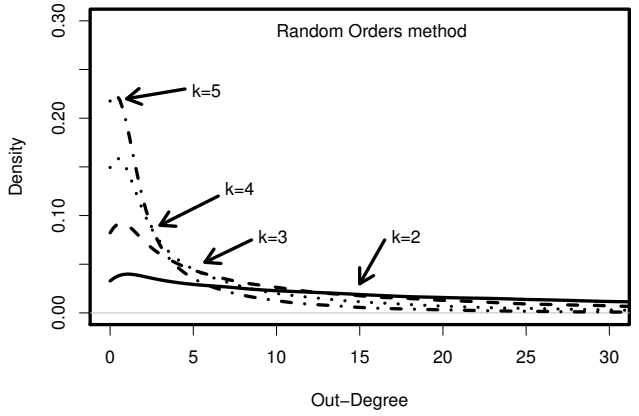
Du point de vue du chemin le plus long, les méthodes d'Erdős et Layer-by-Layer sont très similaires. Plus la probabilité d'apparition des arêtes (ou leur nombre) augmente et plus la longueur moyenne de ce chemin croît. Dans le cas de Layer-by-Layer, le nombre de niveaux joue tout de même un rôle important puisqu'il établit une limite sur cette longueur. Le comportement des deux autres algorithmes est moins facile à



(a) Nombre d'arêtes



(b) Chemin critique



(c) Distributions des degrés sortants

FIGURE 7.7 – Caractéristiques des graphes provenant de Random Orders.

anticiper. Ainsi, l'influence de chacun des paramètres de Fan-in/Fan-out sur la longueur du chemin est difficile à identifier. Un degré sortant trop faible provoque des graphes proches de chaînes, mais le degré entrant apparaît jouer un rôle important lorsque les contraintes sur le premier paramètre sont relâchées. Random Orders génère quant à elle des graphes avec des chemins très courts. Ces observations se complètent sur les autres caractéristiques de graphes que nous avons mesurées.

Influences des graphes générés sur les ordonnanceurs 8

Sommaire

8.1 Première étude de cas : ordonnancement par liste 113
8.1.1 Algorithmes considérés 114
8.1.2 Simulation 114
8.1.3 Analyse 114
8.2 Sensibilité des ordonnanceurs à la méthode de génération 115
8.2.1 Algorithmes étudiés 116
8.2.2 Expériences réalisées 116
8.2.3 Performance des groupes témoins 117
8.2.4 Sensibilité à la distribution des calculs 118
8.2.5 Sensibilité à la distribution des communications 120

Nous avons pu constater que d'une méthode de génération à l'autre, les propriétés des graphes générés peuvent varier considérablement. Or ces propriétés peuvent avoir une influence sur la performance d'un algorithme d'ordonnancement. Il est ainsi évident que des graphes possédant un chemin critique petit pourront être plus faciles à ordonnancer.

Nous réalisons dans ce chapitre deux études de cas sur l'influence d'une méthode de génération sur la performance d'ordonnanceurs. Dans une première partie, nous évaluons la performance de quelques ordonnanceurs à liste sur un problème simple, sans annotations ni sur les nœuds ni sur les arêtes (tâches unitaires avec précédence). Il s'agit là d'évaluer si un changement de structure des entrées influe sur la performance d'algorithmes très simples. Dans un deuxième temps, nous nous intéressons à la simulation d'algorithmes plus complexes, destinés à l'ordonnancement sur des plates-formes distribuées, avec prise en compte des coûts de communications. Il s'agit alors d'évaluer si la performance d'un algorithme varie en fonction des annotations utilisées, et si cela peut perturber la comparaison de plusieurs algorithmes.

8.1 Première étude de cas : ordonnancement par liste

Nous nous intéressons donc ici à l'influence de la structure des graphes sur la performance d'algorithmes d'ordonnancement assez simple : les ordonnanceurs par liste.

Suivant la notation standard, ces algorithmes doivent résoudre le problème NP-difficile $P \mid p_i = 1; prec \mid C_{\max}$. Il s'agit donc d'ordonnancer un graphe de tâches possédant toutes le même poids sur un jeu de machines parallèles identiques pour minimiser la date de terminaison de la dernière tâche.

8.1.1 Algorithmes considérés

Nous avons déjà présenté les ordonnanceurs par liste et discuté de leur performance en général (borne de Graham). Pour rappel, ce type d'ordonnanceur fonctionne en quelques étapes simples :

1. Construction d'une liste à priorité de toutes les tâches du graphe en fonction d'une métrique (propre à chaque algorithme).
2. À chaque étape de l'ordonnancement :
   (a) Choisir dans la liste une tâche de plus haute priorité dont les prédécesseurs ont été exécutés ;
   (b) Assigner cette tâche à une ressource disponible (processeur).

Chaque algorithme est donc défini principalement par la métrique utilisée pour construire la liste à priorité. Nous nous intéresserons ici à quatre algorithmes en particulier : BottomLevel, OutDegree, MinDegree et Random qui utilisent les stratégies suivantes :

- **BottomLevel** : la priorité d'une tâche est définie par la longueur du chemin le plus long partant de cette tâche.
- **OutDegree** : les tâches sont triées par ordre croissant du nombre de successeurs.
- **MinDegree** : les tâches sont triées par ordre décroissant du nombre de successeurs.
- **Random** : la prochaine tâche à ordonnancer est tirée aléatoirement parmi l'ensemble des tâches prêtes.

8.1.2 Simulation

Nous avons simulé les algorithmes précédents sur différents jeux d'entrées générées avec GGen. Chaque algorithme a été simulé sur des graphes provenant de chaque méthode. Pour chaque méthode analysée, 1000 graphes de 100 nœuds ont été générés. Pour la simulation de l'ordonnanceur Random, chaque graphe a été utilisé 20 fois. Au total, plus de 1500000 simulations ont ainsi été réalisées. La table 8.1 présente les résultats de simulation obtenus.

8.1.3 Analyse

La première observation que nous pouvons faire, c'est que la performance d'un algorithme peut beaucoup varier d'une méthode de génération à l'autre. Ainsi sur 4 processeurs, utiliser Random Orders plutôt que $\mathsf{G}(\mathsf{n}, \mathsf{p})$ peut faire varier de 20% le *makespan* obtenu. Cette différence est encore plus accrue sur 16 processeurs, où elle monte jusqu'à un facteur 3,5.

	OutDegree		BottomLevel		MinDegree		Random	
	moy.	σ	moy.	σ	moy.	σ	moy.	σ
GNP(100,0.25)	36	3	35	3	37	3	36	3
GNM(100,300)	25	< 0.5	25	< 0.5	27	1	26	1
FiFo(100,10,10)	28	1	28	1	29	2	29	2
Layer(100,10,0.5)	26	< 0.5	26	< 0.5	27	1	26	1
RandomOrders(100,2)	25	1	25	< 0.5	29	1	27	1
GNP(100,0.25)	35	3	35	3	35	3	35	3
GNM(100,300)	12	2	12	2	13	2	12	2
FiFo(100,10,10)	12	2	12	2	13	2	13	2
Layer(100,10,0.5)	10	< 0.5	10	< 0.5	10	< 0.5	10	< 0.5
RandomOrders(100,2)	17	2	17	2	17	2	17	2

TABLE 8.1 – *Makespan* obtenu par simulation d'un ensemble d'ordonnanceurs par liste, en utilisant 4 (haut) et 16 processeurs (bas) sur 1000 graphes générés aléatoirement.

Cette variation des résultats s'explique très facilement par une analyse théorique du problème d'ordonnancement considéré. En effet, la performance des algorithmes pour le problème $P \mid p_j; prec \mid C_{\max}$ dépend fortement de la longueur du chemin critique. Ainsi, lorsque le nombre de processeurs disponibles est suffisamment grand, la solution optimale est égale à la somme des poids sur le chemin le plus long du graphe. Puisque toutes les tâches ont le même poids de 1, le *makespan* optimal est égal à la longueur du chemin critique du graphe.

Ainsi, sur un petit nombre de processeurs, la performance de nos algorithmes est supérieure au chemin critique pour les paramètres choisis sur chacune des méthodes (les longueurs des chemins critiques sont indiquées en section 7.3). Lorsque nous doublons le nombre de processeurs, la performance des algorithmes se rapproche de la performance optimale. Or les paramètres choisis pour $\mathsf{G}(\mathsf{n}, \mathsf{p})$ résultent dans de plus grands chemins critiques que le reste des méthodes. En conséquence, la performance obtenue par nos algorithmes est bien moindre face aux graphes de $\mathsf{G}(\mathsf{n}, \mathsf{p})$ que de Random Orders par exemple. Ces résultats confirment l'importance des choix de la méthode de génération et des paramètres de génération adéquats lors de la simulation d'ordonnanceurs : un mauvais jeu de paramètres peut influer de manière importante sur la performance obtenue.

Notons aussi que les algorithmes que nous avons simulés produisent des résultats relativement proches les uns des autres. Si la borne de Graham nous garantit que la différence entre les résultats de deux algorithmes ne peut dépasser un ratio de 2, il est intéressant de constater qu'en pratique ce ratio peut être bien inférieur.

8.2 Sensibilité des ordonnanceurs à la méthode de génération

Nous étudions maintenant une deuxième forme d'influence des méthodes de génération de graphes sur la performance des ordonnanceurs. Il s'agit ici d'évaluer la *sensibilité* d'un algorithme d'ordonnancement à certains paramètres du graphe en entrée : quelle variation subit la performance de l'algorithme lorsqu'un paramètre du

graphe est modifié.

Cette étude de sensibilité peut être justifiée de plusieurs manières. Tout d'abord, il peut être intéressant d'évaluer comment un algorithme précis réagit lorsque ses entrées subissent une modification importante. Ainsi, si l'algorithme a été déployé dans un environnement particulier mais que les caractéristiques de cet environnement sont modifiées (les utilisateurs changent de comportement par exemple) il peut être utile de voir comment la performance de l'ordonnanceur sera modifiée. Ensuite, cela peut servir à étudier la performance d'un ordonnanceur dans un environnement différent de celui pour lequel il a été conçu.

Nous avons réalisé cette étude sur des algorithmes destinés à l'ordonnancement sur plate-forme distribuée : les graphes sont annotés pour posséder des coûts en calcul et en communication. Il s'agit alors d'observer si la modification de la distribution utilisée pour ces annotations modifie la performance des algorithmes considérés.

8.2.1 Algorithmes étudiés

Dans la littérature, de nombreux algorithmes d'ordonnancement prenant en compte les communications existent. Dans la suite, nous nous intéressons plus particulièrement à deux catégories, les algorithmes de *list-scheduling* et de *clustering*.

Parmi les algorithmes de *list-scheduling*, nous comparons les algorithmes HEFT [THW02], CPOP [THW02] et HBMCT [RH04]. Ces algorithmes fonctionnent de la même façon que ceux présentés plus haut, ne différant que par la métrique utilisée pour trier les tâches. Pour les algorithmes de *clustering*, le fonctionnement est différent. Le but de ces algorithmes est de regrouper les tâches qui génèrent des communications impactant le temps d'exécution total sur une infinité de machines. Pour cela, ces algorithmes regroupent de manière itérative sur les mêmes processeurs les tâches concernées par les communications les plus importantes. Les groupes formés, appelés *clusters*, sont ainsi fusionnés, à la condition que le chemin critique n'augmente jamais lors d'un regroupement. Si ce regroupement est toujours réalisé à partir d'un ordonnancement sur une infinité de machines, il n'atteint pas nécessairement un nombre de clusters inférieur au nombre de processeurs disponibles dans le problème considéré. La plupart des algorithmes considèrent alors que le problème n'a pas de solution.

Nous avons choisi comme algorithme de *clustering* DSC [YG94] qui fournit un ordonnancement quel que soit le nombre de processeurs. Et c'est l'implémentation de PYRROS [TA92] que nous avons utilisée pour nos simulations.

Pour simuler ces algorithmes, nous avons eu recours à Simgrid [CLQ08]. Cet outil permet la réalisation d'un simulateur gérant finement les coûts des communications. Pour modéliser ces coûts, Simgrid prend en entrée une description de la plate-forme, fournissant les caractéristiques des liens de communication et la puissance des machines. L'outil peut alors simuler de la contention réseau et impacter les temps de communication sur la disponibilité des processeurs pour l'ordonnancement.

8.2.2 Expériences réalisées

L'objectif de nos simulations est d'établir si un des algorithmes précédemment cité est sensible à certaines caractéristiques des applications. Pour cela, nous allons com-

Entrées	Nb tâches	Nb Comm.		Coût en calcul		Coût en Comm.		CCR
		moy	écart-type	moy	écart-type	moy	écart-type	
T_{small}	500	746	27	9.98	5.7	0.5	0.2	≈ 20
T_{big}						10.2	5.1	≈ 1

TABLE 8.2 – Récapitulatif des caractéristiques des jeux de données de référence. Les coûts sont donnés en moyenne sur une tâche/arête.

parer la performance des algorithmes selon que l'on utilise un jeu d'entrées *test* (ou témoin) ou une version modifiée de ce dernier. Deux types de paramètres vont être modifiés par la suite : la distribution des coûts en calcul et celle des coûts en communication des graphes de tâches considérés par les ordonnanceurs.

La plate-forme d'exécution de référence est un groupe de machines homogènes reliées entre elles par un lien privilégié (pas de contention sur les communications). Cette plate-forme nommée *clique* dispose d'un réseau gigabit. Les caractéristiques (latence, débit) du réseau ne seront pas modifiées par la suite. Nous appelons dans les paragraphes suivants *temps de communications* le temps moyen que prennent les transferts de données lorsqu'ils sont simulés sur cette plate-forme.

La caractéristique principale qui a déterminé notre choix de graphes de tâches de référence est le ratio calculs/communications. En effet, ce ratio a une influence capitale sur la capacité d'un ordonnanceur à distribuer les tâches sur les différentes machines disponibles. Nous disposons donc de deux jeux d'entrées servant de référence : ils partagent les mêmes caractéristiques en nombre de nœuds ou d'arêtes, mais contiennent des coûts en calcul et en communication différents. Le tableau 8.2 récapitule leurs caractéristiques ainsi que le ratio temps moyen de calcul d'une tâche sur temps moyen de communication (CCR).

Les graphes ont été générés à l'aide de l'algorithme Layer-by-Layer, en générant des graphes à 500 nœuds et 100 niveaux. La probabilité d'apparition d'une arête a été choisie de façon à obtenir une moyenne de 3 arêtes par sommet. La formule exacte est : $p = \frac{3l}{n(l-1)}$ avec l le nombre de niveaux. Chaque groupe témoin contient 100 graphes. Les annotations suivent sur les jeux témoins une distribution uniforme.

8.2.3 Performance des groupes témoins

Avant de réaliser l'analyse de sensibilité proprement dite, il nous faut observer la performance de nos algorithmes sur les groupes témoins. Nous avons omis les intervalles de confiance, trop petits pour être visualisés.

Les figures 8.1, 8.2 nous donnent respectivement les accélérations (*speed-up*) de nos algorithmes sur T_{small} et T_{big} avec un nombre de machines variant entre 1 et 50. Ces courbes mettent en valeur plusieurs caractéristiques des algorithmes choisis. Sur T_{small} avec un nombre de machines faibles (inférieur à 10), la plupart des algorithmes fournissent un *speed-up* quasi optimal. Lorsque le nombre de machines augmente, des différences de performance plus marquées apparaissent. On peut aussi noter que HBMCT et DSC perdent en performance lorsque le nombre de machines est important.

Lorsque les coûts en communications sont importants (comme sur T_{big} , avec CCR égal à 1), les différences de comportement entre algorithmes apparaissent plus nette-

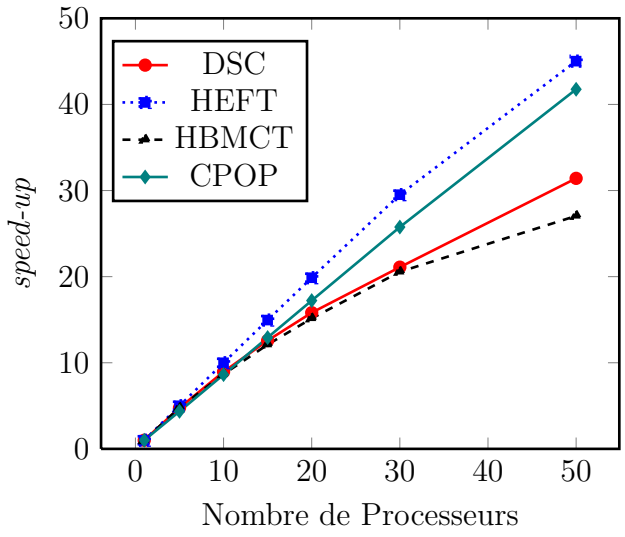


FIGURE 8.1 – Accélération des algorithmes en fonction du nombre de processeurs (sur T_{small}).

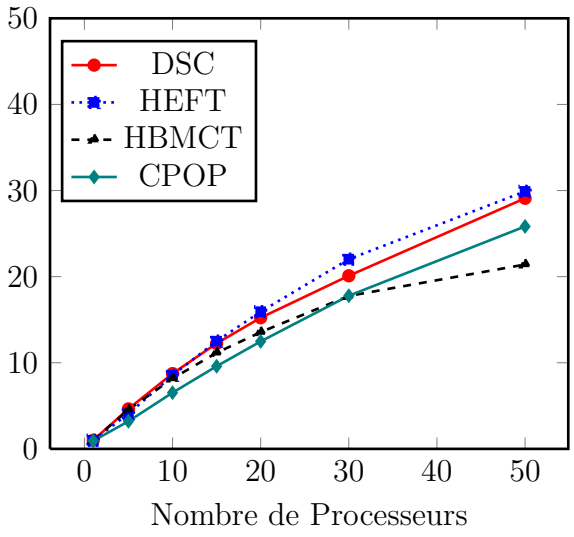


FIGURE 8.2 – Accélération des algorithmes en fonction du nombre de processeurs (sur T_{big}).

ment. DSC qui réalise une diminution importante du nombre de communications avec le *clustering* de tâches, est ainsi faiblement impacté. Au contraire, les algorithmes de liste voient leur efficacité décroître significativement avec le nombre de machines.

Les comportements que nous venons d'observer nous servent de base pour analyser la sensibilité des algorithmes étudiés. Nous définissons la sensibilité d'un algorithme pour le reste de ce chapitre comme la dégradation (ou amélioration) de sa performance entre son exécution sur le jeu d'entrées modifiées et celle sur le groupe témoin correspondant. Afin de simplifier l'interprétation de cette dégradation, nous la normalisons par la valeur de référence. Plus formellement, le calcul est le suivant :
Sensibilité $= \frac{C_{\max} - C_{\max_\text{ref}}}{C_{\max_\text{ref}}}$ avec :

- $C_{\max}$: le temps d'exécution sur jeu d'entrées modifiées.
- $C_{\max_\text{ref}}$: le temps d'exécution obtenu sur le groupe test.

Ainsi, un algorithme avec un sensibilité de 0.1 sur une des modifications a vu son temps de complétion (observé sur le groupe modifié) augmenter de 10% par rapport au temps sur le groupe témoin.

8.2.4 Sensibilité à la distribution des calculs

Comme première étude de sensibilité, nous nous intéressons au comportement de nos algorithmes d'ordonnancement lorsque la distribution des coûts d'exécution des tâches change. Ce changement de la loi de distribution modifie la répartition entre petites tâches et grandes tâches (en temps d'exécution) dans les graphes en entrée. Nous avons choisi de passer ainsi d'une loi uniforme à une loi exponentielle (paramétrée pour conserver le même temps d'exécution moyen sur les tâches). Aucun des algorithmes étudiés ne prend en compte cette distribution des coûts lors du calcul d'un d'ordonnancement. En conséquence, nous pourrions nous attendre à ce que tous réagissent, en moyenne, de la même façon face à ces modifications.

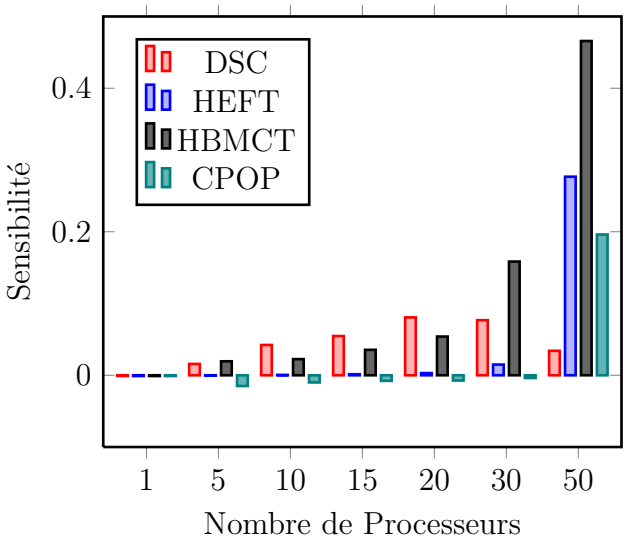


FIGURE 8.3 – Sensibilité à la distribution du calcul en fonction du nombre de processeurs (sur T_{small}).

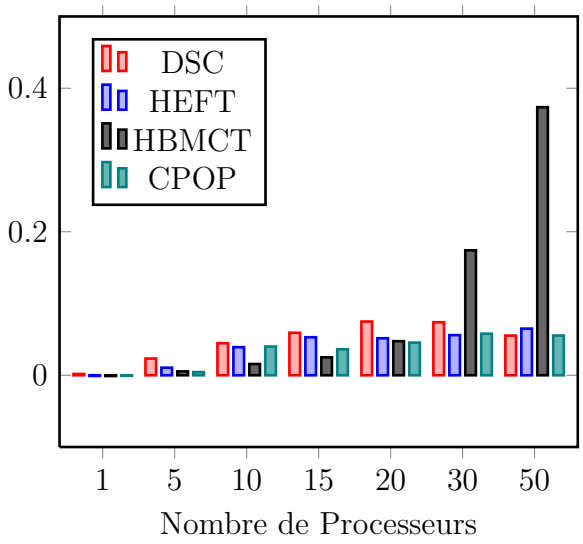


FIGURE 8.4 – Sensibilité à la distribution du calcul en fonction du nombre de processeurs (sur T_{big}).

La figure 8.3 montre la sensibilité observée pour nos algorithmes sur une modification du groupe témoin avec peu de communications (T_{small}). Nous détaillons cette sensibilité en fonction du nombre de processeurs utilisés pour l'ordonnancement. La sensibilité sur une machine (sans coûts de communications), nous confirme que les graphes donnés aux algorithmes possèdent bien une quantité de travail équivalente au groupe témoin (différence inférieure à 2%).

Face à notre jeu de données modifiées, trois phénomènes sont à observer. Premièrement, HBMCT subit une variation très importante de sa performance par rapport au groupe témoin (plus de 40 %). Cette sensibilité se manifeste surtout lorsque 50 machines sont utilisées et est négligeable en dessous du seuil de 30 machines. Ce seuil correspond à un nombre de machines équivalent au parallélisme des graphes en entrée (cf figure 8.1). Cela signifie que l'algorithme est instable quand il dispose de plus de machines que nécessaire pour calculer son ordonnancement. Cette sensibilité, que l'on retrouve dans une moindre mesure chez CPOP et HEFT entraîne un éloignement de DSC qui est faiblement impacté par cette modification. Ainsi, HBMCT passe pour le temps de complétion sur 50 processeurs de 185 (159 pour DSC) à 271 (respectivement 164).

La figure 8.4 présente la même analyse de sensibilité avec une modification de T_{big}. Dans ce cas, les communications ont plus d'influence sur le temps de complétion que les temps d'exécution des tâches. Notons alors que la sensibilité des algorithmes diminue fortement, même si HBMCT reste plus sensible que les autres. Ces modifications n'altèrent pas pour autant la comparaison que nous pouvions réaliser précédemment : HEFT reste plus performant que DSC et CPOP meilleur que HBMCT.

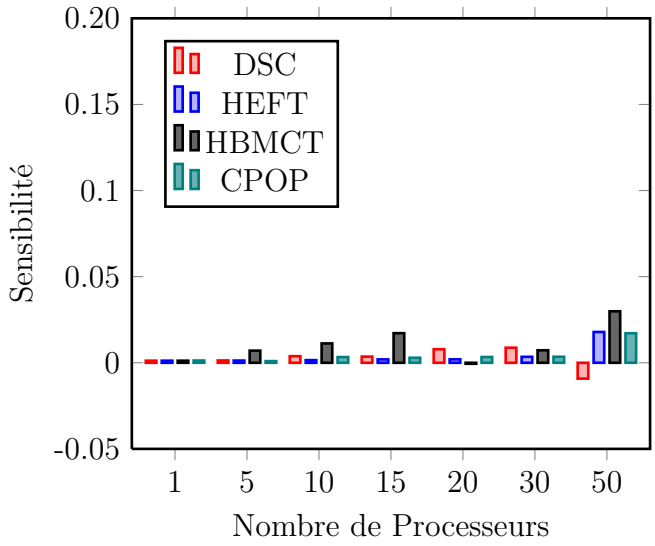


FIGURE 8.5 – Sensibilité à la distribution des communications en fonction du nombre de processeurs (sur T_{small}).

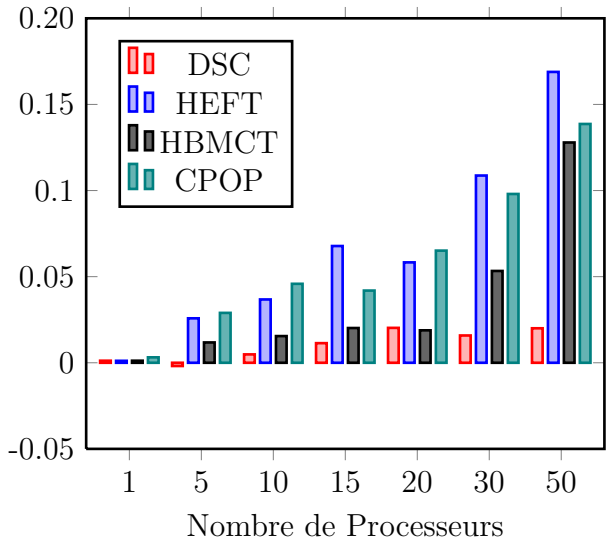


FIGURE 8.6 – Sensibilité à la distribution des communications en fonction du nombre de processeurs (sur T_{big}).

8.2.5 Sensibilité à la distribution des communications

Nous nous intéressons cette fois à la sensibilité des algorithmes choisis à la distribution des communications. Comme précédemment, nous avons modifié les groupes témoins pour passer d'une loi uniforme à une loi exponentielle.

La figure 8.5 montre l'évolution de la sensibilité en fonction du nombre de processeurs avec en entrée une modification de T_{small}. Sur cette figure, la quantité de communication est peu élevée (CCR à 1). Dans ce cas, la sensibilité à la loi de distribution des communications est négligeable quel que soit le nombre de machines.

La figure 8.6 montre la même évolution lorsque les communications sont plus importantes (CCR à 1). Dans ce cas les algorithmes de *list-scheduling* sont significativement plus sensibles. Notons que DSC, un des algorithmes offrant de bonnes performances durant l'analyse des groupes témoins, est le moins sensible à cette modification des communications. Cela peut s'expliquer par un nombre de communications plus faible que les autres algorithmes. En effet, il regroupe en priorité les tâches qui communiquent le plus sur le même processeur et donc les communications les plus importantes seront rarement effectuées, quelle que soit leur distribution (impact faible).

Ces différences de sensibilité ne sont pas sans conséquences : dans certains cas, alors que l'analyse sur les groupes témoins nous donnait un algorithme comme étant meilleur qu'un autre, le jeu de données modifié inverse cette relation. La figure 8.7 met en valeur l'inversion constatée entre DSC et HEFT sur les jeux de données T_{big}. Il apparaît clairement que la sensibilité de HEFT joue fortement sur les différences de performance entre ces deux algorithmes.

Il est finalement intéressant d'observer que des algorithmes conçus pour résoudre des problèmes d'ordonnancement avec coûts de communications peuvent se montrer assez sensibles à la distribution de ces derniers. C'est notamment le cas lorsque le nombre de machines disponibles est plus important que le parallélisme des graphes

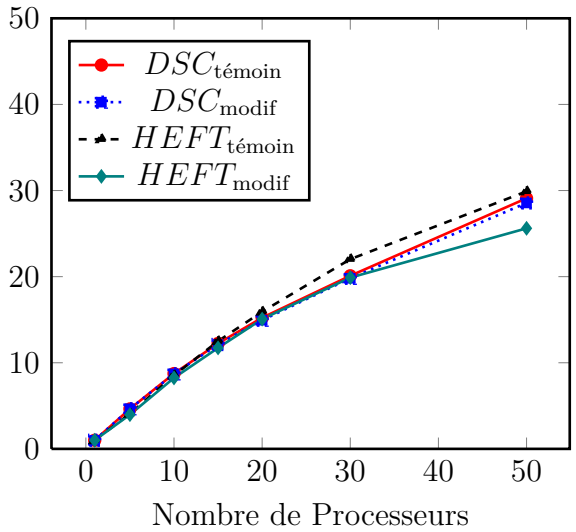


FIGURE 8.7 – Accélérations de DSC et HEFT sur les jeux de données T_{big}.

en entrées. Enfin, la distribution des coûts de calculs semble elle aussi jouer un rôle, surtout lorsque les communications sont faibles.

Nous avons réalisé des expériences similaires avec d'autres entrées. Nous avons ainsi utilisé des graphes avec un nombre de niveaux calculé pour un degré moyen par nœud de 5. Nous avons aussi testé d'autres CCR (2 et 10) et une distribution de pareto pour les jeux de données modifiés. Dans tous ces cas, les résultats obtenus sont similaires : certains algorithmes présentent une sensibilité plus importante que les autres, ce qui modifie la différence de performance entre certains d'entre eux, entraînant parfois des inversions.

Conclusion et Perspectives

La conception et l'utilisation d'environnements d'expérimentation font partie intégrante de la démarche scientifique. Nous avons montré dans ce document comment de tels environnements permettent l'analyse et parfois même l'optimisation d'applications. Ce travail s'est déroulé sur deux axes d'études complémentaires : l'analyse de l'utilisation des ressources matérielles d'un nœud de calcul par une application réelle, et l'étude des performances d'algorithmes de répartition du calcul en fonction des caractéristiques de leurs entrées.

Nous avons ainsi conçu, validé et utilisé des environnements de contrôle de l'attribution des ressources matérielles d'une machine à mémoire partagée. Dans le chapitre 4 nous avons développé une méthodologie pour appliquer une charge processeur précise et reproductible sur une machine dédiée. Cette méthodologie, basée sur une coopération avec le système d'exploitation, a été implémentée et validée sous Linux. Nous avons démontré que cette implémentation se comportait bien mieux que les autres solutions existantes. À ce sujet il est intéressant de noter que notre méthodologie, après sa publication en 2010, a été réimplémentée et reconnue comme efficace par le projet Wrekavoc [BNG10, CDGJ10], auquel nous nous sommes comparés.

Dans le chapitre 5 nous avons démontré l'utilité de permettre à une application de contrôler son utilisation en cache. Ce contrôle permet à la fois à un expérimentateur d'analyser les besoins en cache d'une application et au concepteur de cette dernière de l'optimiser. Nous avons implémenté cette fonctionnalité sous Linux, sans avoir à modifier en profondeur le noyau (simplement par ajout de code et redirection des allocations mémoire). En utilisant cette implémentation, nous avons présenté une méthodologie efficace pour analyser les besoins en cache de chacune des structures de données d'une application, basée sur la notion de `working set`. Cette analyse nous a permis d'optimiser plusieurs applications, d'abord en isolant les structures ne présentant aucune réutilisation et ensuite en redistribuant le cache aux différentes structures en fonction de leurs besoins.

Nous avons aussi montré l'importance de la méthode de génération des graphes de tâches utilisés pour la simulation d'ordonnanceurs sur la qualité des solutions observées. Dans le chapitre 7 nous avons présenté un environnement de génération de DAGs unifiant les méthodes les plus classiques de la littérature. Nous avons ensuite analysé ces méthodes en s'appuyant sur leurs propriétés théoriques et démontré des différences notables dans les caractéristiques des graphes générés.

Enfin, dans le chapitre 8 nous avons analysé l'influence de la méthode de génération des graphes de tâches sur la performance d'ordonnanceurs. Nous avons notamment identifié la sensibilité de certains algorithmes : changer la distribution utilisée pour annoter les graphes modifie de manière importante la qualité des solutions obtenues. Cette sensibilité entraîne en particulier des phénomènes d'inversion. Alors qu'un algorithme semblait plus performant qu'un autre sur une première campagne de simulation, sa

sensibilité inverse cette comparaison sur une deuxième campagne aux caractéristiques très proches de la première.

La majeure partie des résultats présentés ici a été publiée dans des conférences internationales. De plus, les trois environnements développés durant cette thèse sont accessibles librement sur internet (open source) :

- KRASH : `http://krash.ligforge.imag.fr/`
- CControl : `http://ccontrol.ligforge.imag.fr/`
- GGen : `http://ggen.ligforge.imag.fr/`

Perspectives

Si nous avons conçu les méthodes de contrôle de l'utilisation des ressources matérielles d'une machine indépendamment du système d'exploitation considéré, leurs implémentations reposent sur des fonctionnalités propres à Linux. Il serait intéressant d'étudier l'application de ces méthodes sur d'autres systèmes comme ceux des BlueGene ou du K Computer. Sur ces machines, le système d'exploitation est généralement simplifié au maximum, ce qui complique l'application de techniques complexes comme l'ordonnancement par groupe.

Ce dernier subit par ailleurs des modifications intéressantes sous Linux : certains développeurs étudient la possibilité d'ajouter à chaque groupe de processus une limite sur la bande passante processeur autorisée. Cela simplifierai grandement notre travail puisque les processus poids ne seraient plus nécessaires pour appliquer une charge.

Certaines architectures comme le SPARC64 VIIIfx proposent des mécanismes de partitionnement de cache contrôlés par des instructions spéciales. Il serait intéressant d'étudier l'application de notre méthode de mesure des `working sets` d'un programme à ce type d'architecture. Par ailleurs, nous gagnerions à automatiser cette méthode : le procédé d'isolation d'une structure en particulier et l'exécution sur plusieurs tailles de partition est à l'heure actuelle entre les mains de l'expérimentateur. Nous pourrions envisager l'utilisation de techniques de compilation pour identifier automatiquement les structures d'intérêt et lancer leur analyse. Notons par ailleurs que certaines architecture récentes fonctionnent avec des caches plus complexes que ceux présentés dans ce document. Le Sandy Bridge d'Intel par exemple fonctionne avec un cache L3 découpé en autant de morceaux que de cœurs. Chaque cœur possède en réalité un morceau à proximité, ce qui signifie que certaines lignes de cache sont plus proches que d'autres en temps d'accès. Il serait intéressant d'étudier l'application de nos méthodes d'optimisation à ce type d'architectures.

Pour ce qui est de l'analyse de sensibilité des algorithmes d'ordonnancement, cette dernière mériterait d'être creusée, notamment en expérimentant avec des topologies de plates-formes différentes de la clique. Une analyse plus théorique des phénomènes à l'origine de cette sensibilité et des inversions qui l'accompagnent sont aussi des pistes envisageables.

En ce qui concerne notre environnement de génération de graphes, plusieurs pistes s'offrent à nous. Tout d'abord, l'intégration de la notion de campagne expérimentale dans l'outil. Au jour d'aujourd'hui, l'interface proposée permet de construire une campagne de génération de graphes et de s'attaquer à leur analyse, mais le procédé reste délicat (sur la gestion des générateurs aléatoires notamment). Les utilisateurs gagneraient à pouvoir spécifier une commande comme "générer 1000 graphes selon la

méthode $\mathsf{G}(\mathsf{n}, \mathsf{p})$ et mesurer la longueur moyenne du chemin le plus long" plutôt que de faire appel à l'outil 1000 fois. Ce genre de campagne pourrait ensuite être lancée automatiquement sur des plates-formes comme Grid5000. Enfin, d'autres algorithmes d'analyse et de génération de DAGs pourrait être ajoutés en fonction des besoins des utilisateurs.

Bibliographie

[ABP98] N.S. Arora, R.D. Blumofe et C.G. Plaxton : Thread scheduling for multiprogrammed multiprocessors. *In Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129. ACM, 1998.

[ACC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield et B. Smith : The tera computer system. *In Proceedings of the 4th international conference on Supercomputing*, pages 1–6. ACM, 1990.

[AVAM92] Almeida, Vasconcelos, Arabe et Menasce : Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems. *SC Conference*, 0:683–691, 1992.

[BAM+96] E. Bugnion, J.M. Anderson, T.C. Mowry, M. Rosenblum et M.S. Lam : Compiler-directed page coloring for multiprocessors. *ACM SIGOPS Operating Systems Review*, 30(5):255, 1996.

[BCOM+10] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault et R. Namyst : hwloc : a generic framework for managing hardware affinities in hpc applications. *In 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186. IEEE, 2010.

[BD01] K. Beyls et E. D'Hollander : Reuse distance as a metric for cache behavior. *In Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 350–360, 2001.

[BDG+00] S. Browne, J. Dongarra, N. Garner, G. Ho et P. Mucci : A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh et Kai Li : The parsec benchmark suite : characterization and architectural implications. *In Proceedings of 17th International Conference on Parallel Architecture and Compilation Techniques*, pages 72–81, 2008.

[BLTG09] X. Besseron, C. Laferrière, D. Traoré et T. Gautier : X-kaapi : Une nouvelle implémentation extrême du vol de travail. *Proceedings des 19èmes rencontres francophones du parallélisme (RenPar'19)*, 2009.

[Blu95] Robert D. Blumofe : Executing multithreaded programs efficiently. Technical Report MIT/LCS/TR-677, Massachusetts Institute of Technology, septembre 1995.

[BNG10] Tomasz Buchert, Lucas Nussbaum et Jens Gustedt : Accurate emulation of cpu performance. *In 8th International Workshop on Algorithms,*

Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar'2010), Ischia, Italy, 2010.

[BP98] R.D. Blumofe et D. Papadopoulos : The performance of work stealing in multiprogrammed environments. *In ACM SIGMETRICS Performance Evaluation Review*, volume 26, pages 266–267. ACM, 1998.

[BW88] Béla Bollobás et Peter Winkler : The longest chain among random points in euclidean space. *Proceedings of the American Mathematical Society*, 103(2):pp. 347–353, 1988.

[CCR+00] Derek Chiou, Derek Chiouy, Larry Rudolph, Larry Rudolphy, Srinivas Devadas, Srinivas Devadasy, Boon S. Ang et Boon S. Angz : Dynamic cache partitioning via columnization. *In Proceedings of the Design Automation Conference*, 2000.

[CDGJ10] Louis-Claude Canon, Olivier Dubuisson, Jens Gustedt et Emmanuel Jeannot : Defining and controlling the heterogeneity of a cluster : The wrekavoc tool. *J. Syst. Softw.*, 83:786–802, May 2010.

[CDL+02] Yen-Kuang Chen, Eric Debes, Rainer Lienhart, Matthew Holliman et Minerva Yeung : Evaluating and improving performance of multimedia applications on simultaneous multi-threading. *Parallel and Distributed Systems, International Conference on*, 0:529, 2002.

[CJ06a] Louis-Claude Canon et Emmanuel Jeannot : Wrekavoc : a tool for emulating heterogeneity. *In 15th IEEE Heterogeneous Computing Workshop (HCW 06)*, 2006.

[CJ06b] Sangyeun Cho et Lei Jin : Managing distributed, shared l2 caches through os-level page allocation. *In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, 2006.

[CLQ08] Henri Casanova, Arnaud Legrand et Martin Quinson : SimGrid : a Generic Framework for Large-Scale Distributed Experiments. *In 10th IEEE International Conference on Computer Modeling and Simulation*, mars 2008.

[CMP+09] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent et Frédéric Wagner : Comment rater la validation de votre algorithme d'ordonnancement. *In French RENPAR Conference, Poster Session/Short Paper*, 2009.

[CMP+10] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent et Frédéric Wagner : Random graph generation for scheduling simulations. *In International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, 2010.

[CN06] Gabor Csardi et Tamas Nepusz : The igraph software package for complex network research. *InterJournal Complex Systems*, 1695, 2006.

[Cor10a] Intel Corporation : Intel architecture software developer's manual, volume 3 : System programming guide, March 2010.

[Cor10b] Intel Corporation : Intel architectures optimization reference manual, March 2010.

[CS07] J. Chang et G.S. Sohi : Cooperative cache partitioning for chip multiprocessors. *In Proceedings of the 21st annual international conference on Supercomputing*, pages 242–252. ACM, 2007.

[CSL+04] Onur Celebioglu, Amina Saify, Tau Leng, Jenwei Hsieh, Victor Mashayekhi et Reza Rooholamini : The performance impact of computational efficiency on hpc clusters with hyper-threading technology. *Parallel and Distributed Processing Symposium, International*, 15:250b, 2004.

[CW76] H.J. Curnow et B.A. Wichmann : A synthetic benchmark. *The Computer Journal*, 19(1):43, 1976.

[Dav91] David Bailey et al. : The NAS parallel benchmarks. Rapport technique RNR-91-002, NAS Systems Division, janvier 1991.

[Dil] Matthew Dillon : Design elements of the FreeBSD VM system. `http://www.freebsd.org/doc/en/articles/vm-design`.

[DLP03] Jack Dongarra, Piotr Luszczek et Antoine Petitet : The LINPACK benchmark : past, present and future. *Concurrency and Computation : Practice and Experience*, 15(9):803–820, août 2003.

[DM98] L. Dagum et R. Menon : Openmp : an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1): 46 –55, jan-mar 1998.

[Dre07] Ulrich Drepper : What every programmer should know about memory. `http://www.akkadia.org/drepper/`, 2007.

[Dro09] M. Drozdowski : *Scheduling for Parallel Processing*. Springer-Verlag New York Inc, 2009.

[DRW98] Robert P. Dick, David L. Rhodes et Wayne Wolf : TGFF : Task Graphs For Free. *In Proceedings of the 6th International Workshop on Hardware/Software Codesign*, pages 97–101, Washington, DC, USA, mars 1998. IEEE Computer Society.

[DWZ11] Xiaoning Ding, Kaibo Wang et Xiaodong Zhang : Ulcc : a user-level facility for optimizing shared cache performance on multicores. *In Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPOPP)*, pages 103–112, 2011.

[ENBSH11] D. Eklov, N. Nikoleris, D. Black-Schaffer et E. Hagersten : Cache pirating : Measuring the curse of the shared cache. Rapport technique, Uppsala University, 2011.

[ER59] Paul Erdős et Alfréd Rényi : On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[Fei09] Dror Feitelson : The parallel workloads archive logs. `http://www.cs.huji.ac.il/labs/parallel/workload/logs.html`, octobre 2009.

[GN00] Emden R. Gansner et Stephen C. North : An open graph visualization system and its applications to software engineering. *Software : Practice and Experience*, 30(11):1203–1233, 2000.

[God08] Sebastien Godard : SYSSTAT. `http://sebastien.godard.pagesperso-orange.fr/`, 2002-2008.

[Gou09] Brian Gough : *GNU Scientific Library Reference Manual (3rd Ed.)*. Network Theory Ltd., 2009.

[Gra69] Ronald L. Graham : Bounds on multiprocessor timing anomalies. *SIAM J. Appl. Math*, 17(2):416–429, 1969.

[Hen06] J.L. Henning : Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[HRIM06] L.R. Hsu, S.K. Reinhardt, R. Iyer et S. Makineni : Communist, utilitarian, and capitalist cache policies on cmps : caches as a shared resource. *In Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 13–22. ACM, 2006.

[HS89] M.D. Hill et A.J. Smith : Evaluating associativity in cpu caches. *Computers, IEEE Transactions on*, 38(12):1612–1630, 1989.

[Iye04] Ravi R. Iyer : CQoS : a framework for enabling QoS in shared caches of cmp platforms. *In Proceedings of the 18th International Conference on Supercomputing*, pages 257–266, 2004.

[JH04] Christopher Johnson et Charles Hansen : *Visualization Handbook*. Academic Press, Inc., 2004.

[JLLS06] C. Jung, D. Lim, J. Lee et Y. Solihin : Helper thread prefetching for loosely-coupled multiprocessor systems. *In Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[KCS04] Seongbeom Kim, Dhruba Chandra et Yan Solihin : Fair cache sharing and partitioning in a chip multiprocessor architecture. *In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[KH92] R. E. Kessler et Mark D. Hill : Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10:338–359, 1992.

[KMCV10] K. Kedzierski, M. Moreto, F.J. Cazorla et M. Valero : Adapting cache partitioning algorithms to pseudo-lru replacement policies. *In Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[Knu97] Donald E. Knuth : *The Art of Computer Programming*. Addison-Wesley, 1997.

[Leh] Marc Lehmann : Benchmarking libevent against libev. `http://libev.schmorp.de/bench.html`.

[Lin] Linux kernel developpers : Perf. `http://perf.wiki.kernel.org/`.

[LLD+08] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang et P. Sadayappan : Gaining insights into multicore cache partitioning : Bridging the gap between simulation and real systems. *In Proceedings of the 14th International Conference on High-Performance Computer Architecture*, pages 367–378, 2008.

[LLD+09] Qingda Lu, Jiang Lin, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang et P. Sadayappan : Soft-olp : Improving hardware cache performance through software-controlled object-level partitioning. *In Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 246–257, 2009.

[Mar09] T. Maruyama : Sparc64™ viiifx : Fujitsu's new generation octo core processor for peta scale computing. *In Hot Chips*, volume 21, pages 23–25, 2009.

[MSSD10] Hans Meuer, Erich Strohmaier, Horst Simon et Jack Dongarra : 35th release of the TOP500 list of fastest supercomputers, 2010.

[ODJ09] Jens Gustedt Olivier Dubuisson et Emmanuel Jeannot : Validating Wrekavoc : a tool for heterogeneity emulation. *In 18th IEEE Heterogeneous Computing Workshop (HCW 09)*, 2009.

[PH10a] Swann Perarnau et Guillaume Huard : Krash : Reproducible cpu load generation on many-core machines. *In ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), Poster Session/Short Paper*, 2010.

[PH10b] Swann Perarnau et Guillaume Huard : Krash : Reproducible cpu load generation on many-core machines. *In IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[PTH11] Swann Perarnau, Marc Tchiboukdjian et Guillaume Huard : Controlling cache utilization of hpc applications. *In International Conference on Supercomputing (ICS)*, 2011.

[PTV11] Swann Perarnau, Denis Trystram et Jean-Marc Vincent : GGen : Génération aléatoire de graphes pour l'ordonnancement. *In French ROADEF Conference, Short Paper*, 2011.

[QP06] M.K. Qureshi et Y.N. Patt : Utility-based cache partitioning : A low-overhead, high-performance, runtime mechanism to partition shared caches. *In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432. IEEE Computer Society, 2006.

[QP11] Jean-Noël Quintin et Swann Perarnau : Sensibilité des algorithmes d'ordonnancement. *In French RENPAR Conference*, 2011.

[Rei05] J. Reinders : *VTune Performance Analyzer Essentials*. Intel Press, 2005.

[Rei07] J. Reinders : *Intel threading building blocks*. O'Reilly, 2007.

[RH04] Sakellariou Rizos et Zhao Henan : A hybrid heuristic for dag scheduling on heterogeneous systems. *In International Parallel and Distributed Processing Symposium, 2004. Proceedings*, april 2004.

[SCE99] Timothy Sherwood, Brad Calder et Joel S. Emer : Reducing cache misses using hardware and software page placement. *In Proceedings of the 13th International Conference on Supercomputing*, pages 155–164, 1999.

[SM03] Ravi Iyer Srihari Makineni : Measurement-based analysis of tcp/ip processing requirements. *In 10th International Conference on High Performance Computing (HiPC 2003)*, 2003.

[SP95] Neil James Alexander Sloane et Simon Plouffe : *The Encyclopedia of Integer Sequences*. Academic Press, 1995.

[SRD01] G.E. Suh, L. Rudolph et S. Devadas : Dynamic cache partitioning for simultaneous multithreading systems. *In Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 116–127. Citeseer, 2001.

[SRD04] G.E. Suh, L. Rudolph et S. Devadas : Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[STS08] Livio Soares, David K. Tam et Michael Stumm : Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. *In 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–269, 2008.

[STW92] H.S. Stone, J. Turek et J.L. Wolf : Optimal partitioning of cache memory. *Computers, IEEE Transactions on*, 41(9):1054–1068, 1992.

[SY05] M. Snir et J. Yu : On the theory of spatial and temporal locality. Rapport technique, University of Illinois at Urbana-Champaign, 2005.

[TA92] Yang Tao et Gerasoulis Apostolos : Pyrros : static task scheduling and code generation for message passing multiprocessors. *In Proceedings of the 6th international conference on Supercomputing*, ICS '92, pages 428–437, New York, USA, 1992. ACM.

[TDG+10] Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier, Fabien Le Mentec et Bruno Raffin : A work stealing scheduler for parallel loops on shared cache multicores. *In Proceedings of the 4th Workshop on Highly Parallel Processing on a Chip (HPPC 2010)*, 2010.

[TDR10] Marc Tchiboukdjian, Vincent Danjean et Bruno Raffin : Binary mesh partitioning for cache-efficient visualization. *Transactions on Visualization and Computer Graphics*, 16(5):815 –828, sep. 2010.

[TEFK05] D. Tsafrir, Y. Etsion, D.G. Feitelson et S. Kirkpatrick : System noise, os clock ticks, and fine-grained parallel applications. *In Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312. ACM, 2005.

[THW02] Haluk Topcuouglu, Salim Hariri et Min-you Wu : Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13:260–274, March 2002.

[THW10] J. Treibig, G. Hager et G. Wellein : Likwid : A lightweight performance-oriented tool suite for x86 multicore environments. *In Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[TK02] Takao Tobita et Hironori Kasahara : A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5):379–394, 2002.

[TW97] A.S. Tanenbaum et A.S. Woodhull : *Operating systems : design and implementation*. Prentice Hall, 1997.

[Var11] Various people : Libhugetlbfs. `http://sourceforge.net/projects/libhugetlbfs`, 2006-2011.

[Wei84] R.P. Weicker : Dhrystone : a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[Win85] Peter Winkler : Random orders. *Order*, 1(4):317–331, décembre 1985.

[YG94] Tao Yang et Apostolos Gerasoulis : Dsc : Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5:951–967, 1994.

[ZJS10] E.Z. Zhang, Y. Jiang et X. Shen : Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs ? *In ACM SIGPLAN Notices*, volume 45, pages 203–212. ACM, 2010.

Table des figures

2.1 Schéma général d'une machine à mémoire partagée. 8
2.2 Comportement des différents types de caches sur des séquences d'accès à des lignes contigües en mémoire. 13
2.3 Relation entre distribution des distances de réutilisation (H) et défauts de cache (Q). Les working sets apparaissent lorsqu'aucun accès mémoire ne bénéficie d'une augmentation de la taille du cache. 18
2.4 Exemple de code C mesurant les défauts de cache durant son exécution à l'aide de PAPI. 19
2.5 Hiérarchies processeur et mémoire d'une machine à 4 processeurs de 6 cœurs (Intel Xeon X7460) 21
2.6 Durée moyenne d'une lecture aléatoire selon la taille de la zone mémoire. 22
2.7 Code de mesure de la vitesse d'accès aux différents niveaux de cache d'une machine. 23

3.1 Répartition du temps processeur dans une hiérarchie de groupes de processus avec priorités. 31
3.2 Exemple de profil de charge. 33
3.3 Coloration de page sur un système hypothétique avec des pages de 2 lignes, et un cache à 4 couleurs associatif 8 voies. Chaque ligne d'une page occupe un ensemble associatif différent, mais les pages d'une même couleur occupent les mêmes ensembles. 36

4.1 Répartition du temps processeur pour une charge de 50% infligée à l'aide d'un processus `SCHED_FIFO` simple. 39
4.2 Latences mesurées par `cyclictest` sur des périodes de sommeil de 1ms, en priorité temps-réel et à l'aide de `clock_nanosleep`. 40
4.3 Répartition du temps processeur entre deux processus ne réalisant que des calculs. 42
4.4 Répartition du temps processeur entre un processus de calcul et un processus réalisant des entrées/sorties. Pour compenser la période de suspension, le processus se voit accorder plus de temps processeur à la reprise. 43
4.5 Exemple de profil de charge KRASH. 48
4.6 Temps d'exécution des algorithmes parallèles sur machine rendue hétérogène par injection de charge. 57

5.1 Illustration de la coloration d'un espace d'adressage. L'application perçoit une zone contigüe mais le système lui fournit des pages physiques de seulement deux couleurs sur les quatre disponibles. 64

5.2 Exemple de code utilisant notre interface de contrôle du cache, créant une partition de la moitié du cache et allouant un tableau de caractères à l'intérieur. 67
5.3 Commandes `shell` pour limiter le cache d'une application en utilisant la bibliothèque d'interception des allocations dynamiques. 68
5.4 Lectures aléatoires : temps d'accès par élément sur une zone mémoire de taille croissante. Comparaison de performance entre les allocations Linux et par CControl. 69
5.5 Lectures aléatoires : temps d'accès par élément sur une zone mémoire de taille croissante. Comparaison de performance entre deux tailles de partition en cache. 70
5.6 Structure du maillage : le tableau de points contient les coordonnées et un scalaire (t) tandis que le tableau de cellules conserve les indices des points composant chacune d'entre elles. 74
5.7 Nombre de défauts de cache (pour le cache partagé) de l'algorithme MT séquentiel en fonction de la taille du cache. 75
5.8 Stencil multi-résolutions : 9 cellules des matrices M_1,M_2 et M_3 sont sommées dans une cellule de M_r. Les zones grisées représentent les besoins en cache de chaque matrice. 78
5.9 Stencil : Défauts de cache L_2 en fonction de la taille de la partition en cache pour chacune des matrices. 79
5.10 Coloration de page à travers une hiérarchie de cache. Les couleurs du plus haut niveau se recouvrent dans le niveau inférieur, qui en contient moins. 81
5.11 Stencil : temps d'exécution en fonction de la taille du cache, pour chaque matrice. 82

6.1 Diagramme de Gantt d'un ordonnancement de quatre tâches. L'espace grisé indique un temps d'inactivité d'un processeur. 88
6.2 Comparaison d'un ordonnancement par LPT et par SPT du même graphe de tâche sur 2 processeurs. 91
6.3 Un DAG et son plus long chemin. 93
6.4 Un DAG et sa plus longue antichaîne. 94

7.1 Chaîne de 5 tâches. 99
7.2 Arbre (à gauche) et anti-arbre (à droite) de 7 tâches. 99
7.3 *Fork-join* de 6 tâches. 100
7.4 Caractéristiques des graphes provenant de $\mathsf{G(n,p)}$ et $\mathsf{G(n,M)}$. 104
7.5 Caractéristiques des graphes provenant de Layer-by-Layer. 106
7.6 Caractéristiques des graphes provenant de Fan-in/Fan-out. 108
7.7 Caractéristiques des graphes provenant de Random Orders. 110

8.1 Accélération des algorithmes en fonction du nombre de processeurs (sur T_{small}). 118
8.2 Accélération des algorithmes en fonction du nombre de processeurs (sur T_{big}). 118

8.3 Sensibilité à la distribution du calcul en fonction du nombre de processeurs (sur T_{small}). 119
8.4 Sensibilité à la distribution du calcul en fonction du nombre de processeurs (sur T_{big}). 119
8.5 Sensibilité à la distribution des communications en fonction du nombre de processeurs (sur T_{small}). 120
8.6 Sensibilité à la distribution des communications en fonction du nombre de processeurs (sur T_{big}). 120
8.7 Accélérations de DSC et HEFT sur les jeux de données T_{big}. 121

Table des matières

Remerciements **iii**

Sommaire **v**

1 Introduction **1**
1.1 Axes d'étude 2
1.1.1 Contrôler l'utilisation d'une ressource matérielle 2
1.1.2 Maitriser les entrées lors de la simulation d'un ordonnanceur . . 3
1.2 Guide de lecture et contributions 3

I Contrôler l'utilisation des ressources matérielles **5**

2 Architecture d'une machine à mémoire partagée **7**
2.1 Une architecture organisée hiérarchiquement 8
2.1.1 Multiprocesseurs et Multicœurs 9
2.1.2 Parallélisme au sein d'un cœur 9
2.1.3 Accès non uniformes à la mémoire 10
2.2 Architecture d'un cache 11
2.2.1 Principes de localité 11
2.2.2 Fonctionnement des caches 12
2.2.3 Gestion du cache 12
2.2.4 Adressage 13
2.2.5 Hiérarchie et partage des caches entre cœurs 14
2.3 Interactions entre caches et applications 15
2.3.1 Classification des défauts de cache 15
2.3.2 Préchargement automatique 16
2.3.3 Distance de réutilisation 16
2.3.4 *Working set* 17
2.4 Informations fournies par le matériel 17
2.4.1 Compteurs de performance matériels 17
2.4.2 Niveaux de trace et utilisations des compteurs 18
2.4.3 Identification des caractéristiques de la machine 19
2.4.4 Détection logicielle de caractéristiques matérielles 20

3 Fonctionnement d'un système d'exploitation moderne **25**
3.1 Abstractions offertes par le système 26
3.1.1 Processus et threads 26
3.1.2 Mémoire virtuelle 27

3.1.3 Groupes de processus 27
3.2 Ordonnancement système 28
3.2.1 Ordonnancement équitable sous Linux 29
3.2.2 Ordonnancement par groupes de processus 30
3.2.3 Ordonnancement Temps-réel 31
3.3 Modèle d'utilisation du processeur 32
3.3.1 Résolution de l'ordonnanceur 32
3.3.2 Charge processeur 32
3.3.3 Mesure de charge 33
3.4 Gestion de la mémoire virtuelle 33
3.4.1 Pagination 34
3.4.2 Gestion de la mémoire physique 34
3.4.3 Coloration de page 35
3.4.4 Gestion d'une mémoire virtuelle avec coloration 35

4 Émuler l'indisponibilité du processeur 37
4.1 Travaux existants 38
4.1.1 Ordonnancement temps-réel 39
4.1.2 Signaux 40
4.1.3 Changement de fréquence du processeur 41
4.2 Générer une charge en coopérant avec le système 42
4.2.1 Partage du temps processeur entre applications 42
4.2.2 Méthodologie retenue 43
4.3 KRASH : une implémentation sous Linux 43
4.3.1 Gestion des groupes sous Linux 44
4.3.2 Mécanisme de supervision 45
4.3.3 Classes d'évènements rencontrés 46
4.3.4 Gestion de la dérive 47
4.3.5 Profil de charge 48
4.4 Évaluation de KRASH 49
4.4.1 Validation expérimentale 50
4.4.2 Comparaison aux autres solutions 52
4.5 Utilisation et extensibilité de la méthode 56
4.5.1 Validation d'applications avec vol de travail 56
4.5.2 Extension aux autres ressources 58

5 Donner à l'utilisateur le contrôle du cache 59
5.1 Contrôler l'utilisation du cache 60
5.1.1 Partitionnement matériel 61
5.1.2 Support du système d'exploitation 61
5.1.3 Outils pour l'utilisateur 62
5.2 Coloration de page en espace utilisateur 62
5.2.1 Colorer l'espace d'adressage d'un processus 63
5.2.2 Gestion des hiérarchies de caches 64
5.3 CControl : un environnement de coloration de page sous Linux 64
5.3.1 Gestion de la mémoire sous Linux 65
5.3.2 Organisation générale de CControl 65

5.4 Validation de CControl 68
5.4.1 Environnement utilisé 68
5.4.2 Résultats 69
5.5 Conditions expérimentales et première optimisation 71
5.5.1 Machines utilisées 71
5.5.2 Optimisation de NAS MG 72
5.6 Extraction d'isosurface 72
5.6.1 MT parallèle pour un cache partagé 73
5.6.2 Analyse des défauts de caches avec la distance de réutilisation . 74
5.6.3 Éviter la pollution du cache 76
5.7 Stencil multi-résolutions 77
5.7.1 Mesure des besoins en cache 77
5.7.2 Redistribution du cache 79
5.8 Partitionner pour une hiérarchie de caches 80
5.8.1 Coloration par niveaux 80
5.8.2 Amélioration du stencil 82

II Génération de graphes de tâches pour la simulation d'ordonnanceurs 85

6 Notions d'ordonnancement 87
6.1 Définitions 87
6.1.1 Classification des problèmes d'ordonnancement 88
6.1.2 Ordonnancement avec dépendances 89
6.2 Complexité et Approximabilité 90
6.2.1 Généralités 90
6.2.2 Heuristiques 90
6.3 Influence d'un graphe sur la performance d'un algorithme 92
6.3.1 Longueur des chemins 92
6.3.2 Antichaîne 93

7 Génération de graphes pour la simulation d'ordonnanceurs 95
7.1 Générateurs et collections de graphes 96
7.1.1 Algorithmes de génération aléatoire 96
7.1.2 Collections de graphes pour la simulation 98
7.1.3 Structures classiques de graphes 99
7.2 GGen : un environnement de génération aléatoire de DAG 99
7.2.1 Organisation de l'environnement 100
7.2.2 Algorithmes de génération implémentés 101
7.3 Caractéristiques des graphes générés 103
7.3.1 Modèles d'Erdős 103
7.3.2 Layer-by-Layer 105
7.3.3 Fan-in/Fan-out 107
7.3.4 Random Orders 109
7.3.5 Conclusion 109

8 Influences des graphes générés sur les ordonnanceurs 113
8.1 Première étude de cas : ordonnancement par liste 113
8.1.1 Algorithmes considérés 114
8.1.2 Simulation 114
8.1.3 Analyse 114
8.2 Sensibilité des ordonnanceurs à la méthode de génération 115
8.2.1 Algorithmes étudiés 116
8.2.2 Expériences réalisées 116
8.2.3 Performance des groupes témoins 117
8.2.4 Sensibilité à la distribution des calculs 118
8.2.5 Sensibilité à la distribution des communications 120

Conclusion et Perspectives 123

Bibliographie 127

Table des figures 135

Table des matières 139

Résumés 143

Résumé. Les machines du domaine du calcul haute performance (HPC) gagnent régulièrement en complexité. De nos jours, chaque nœud de calcul peut être constitué de plusieurs puces ou de plusieurs cœurs se partageant divers caches mémoire de façon hiérarchique. Que ce soit pour comprendre les performances obtenues par une application sur ces architectures ou pour développer de nouveaux algorithmes et valider leur performance, une phase d'expérimentation est souvent nécessaire. Dans cette thèse, nous nous intéressons à deux formes d'analyse expérimentale : l'exécution sur machines réelles et la simulation d'algorithmes sur des jeux de données aléatoires. Dans un cas comme dans l'autre, le contrôle des paramètres de l'environnement (matériel ou données en entrée) permet une meilleure analyse des performances de l'application étudiée.

Ainsi, nous proposons deux méthodes pour contrôler l'utilisation par une application des ressources matérielles d'une machine : l'une pour le temps processeur alloué et l'autre pour la quantité de cache mémoire disponible. Ces deux méthodes nous permettent notamment d'étudier les changements de comportement d'une application en fonction de la quantité de ressources allouées. Basées sur une modification du comportement du système d'exploitation, nous avons implémenté ces méthodes pour un système Linux et démontré leur utilité dans l'analyse de plusieurs applications parallèles.

Du point de vue de la simulation, nous avons étudié le problème de la génération aléatoire de graphes orientés acycliques (DAG) pour la simulation d'algorithmes d'ordonnancement. Bien qu'un grand nombre d'algorithmes de génération existent dans ce domaine, la plupart des publications repose sur des implémentations ad-hoc et peu validées de ces derniers. Pour pallier ce problème, nous proposons un environnement de génération comprenant la majorité des méthodes rencontrées dans la littérature. Pour valider cet environnement, nous avons réalisé de grande campagnes d'analyses à l'aide de Grid'5000, notamment du point de vue des propriétés statistiques connues de certaines méthodes. Nous montrons aussi que la performance d'un algorithme est fortement influencée par la méthode de génération des entrées choisie, au point de rencontrer des phénomènes d'inversion : un changement d'algorithme de génération inverse le résultat d'une comparaison entre deux ordonnanceurs.

Mots-clés : charge processeur, coloration de page, analyse de performance, génération de graphes, algorithmes d'ordonnancement.

Abstract. High performance computing systems are increasingly complex. Nowadays, each compute node can contain several sockets or several cores and share multiple memory caches in a hierarchical way. To understand an application's performance on such systems or to develop new algorithms and validate their behavior, an experimental study is often required. In this thesis, we consider two types of experimental analysis : execution on real systems and simulation using randomly generated inputs. In both cases, a scientist can improve the quality of its performance analysis by controlling the environment (hardware or input data) used.

Therefore, we discuss two methods to control hardware resources allocation inside a system : one for the processing time given to an application, the other for the amount of cache memory available to it. Both methods allow us to study how an application's behavior change according to the amount of resources allocated. Based on modifications of the operating system, we implemented these methods for Linux and demonstrated their use for the analysis of several parallel applications.

Regarding simulation, we studied the issue of the random generation of directed acyclic graphs for scheduler simulations. While numerous algorithms can be found for such problem, most papers in this field rely on ad-hoc implementations and provide little validation of their generator. To tackle this issue, we propose a complete environment providing most of the classical generation methods. We validated this environment using big analysis campaigns on Grid'5000, verifying known statistical properties of most algorithms. We also demonstrated that the performance of a scheduler can be impacted by the generation method used, identifying a reversing phenomenon : changing the generating algorithm can reverse the comparison between two schedulers.

Keywords: cpu load generation, page coloring, performance analysis, random graph generation, scheduling algorithms.