Université de Picardie Jules Verne - Amiens
Faculté de Mathématiques et d'Informatique

École doctorale Sciences et Santé

Contributions systèmes pour le traitement de grandes masses de données sur grappes

THÈSE

présentée et soutenue publiquement le 18 Décembre 2003

pour l'obtention du

Doctorat de l'Université de Picardie Jules Verne
(spécialité informatique)

par

Olivier COZETTE

Composition du jury

Président :	Jean-Jean-Frédéric Myoupo
Rapporteurs :	Bernard Lecussan (ENSAE, SupAero Toulouse) Raymond Namyst (LaBRI, Université de Bordeaux) Loïc Prylli (CNRS)
Examinateur :	Christophe Cérin (LaRIA, UPJV)
Directeur de thèse :	Gil Utard (LaRIA, UPJV)

Laboratoire de Recherche en Informatique d'Amiens

Cette thèse a pu être mené à bien grâce à un financement du Font Social Europeen et du Conseil Régional de Picardie

Mis en page avec la classe thloria.

"Dieu ne joue pas aux dés".
(Albert Einstein)

Remerciements

Contrairement à l'usage, je préfère décrire mes remerciements en citant chronologiquement les personnes qui m'ont aidé lors du long processus qui conduit à une thèse. Les protagonistes ainsi remercié seront Christophe Cerin, Bernard Lecussan, Jean-Frédéric Myoupo, Raymond Namyst, Loïc Prylli pour le jury, Gil Utard pour la direction, le conseil régional de Picardie et le fond social européen pour le financement, le LaRIA et ses membres, mes parents, les personnes présentes pendant la soutenances et celles qui s'étaient excusées, ainsi que de nombreuses autres personnes dont je ne cite qu'une partie dans la suite.

L'informatique a commencé pour moi à l'age de 10 ans, lorsque mes parents, estimant que je pourrais être intéressé par l'informatique, m'ont offert mon premier ordinateur : un TO7/70. Les premiers cours du plan "Informatiques pour Tous" m'ont appris un premier langage de programmation (le Basic). J'ai ensuite commencé à faire de nombreux programmes dont principalement des jeux. Puis cet ordinateur vieillissant, je suis passé, à la suite de quelques cours, dont une colonie et la défunte option informatique du lycée, au PC puis au (Turbo) Pascal. J'ai ensuite voulu commencer l'exploitation du son et aller plus vite pour certains jeux de ma conception. Cette motivation ajoutée à la curiosité du départ, m'ont poussé à apprendre l'assembleur x86 grâce à quelques programmes disponible dans certaines revues.

Les langages ne me donnaient pas assez de possibilités d'accéder aux ressources de l'ordinateur tels que le graphisme avancé ou la souris. Pour palier ce manque, j'ai étudié et programmé de nombreux périphériques au cours de mes années de DEUG. Cela a été réalisé en partie grâce à "La Bible du PC" et les manuels des processeurs Intel, mais la majorité est le résultat de nombreuses heures de reverse engineering du BIOS et d'autres programmes. A cette époque, j'ai trouvé un autre passionné d'informatique, Jérôme Cailly, qui était axé sur le développement de moteur 3D, mais avec qui j'ai pu échanger ma passion et gagner en motivation. Je me suis ensuite dit qu'il était amusant de concevoir un petit système d'exploitation. En effet, je savais comment programmer les disques IDE, souris, cartes graphiques, lecteurs de disquettes et autres contrôleurs DMA. C'est ainsi que des efforts de conceptions, de débogages et de reverse engineerings m'ont permis de déboucher, au début de ma maîtrise de mathématiques, sur un petit système qui démarre avec un mini environnement graphique et sans aide d'un autre OS.

C'est à ce stade que je suis arrivé en DEA d'informatique, sans aucune étude d'informatique avant (j'ai fait un cursus en mathématiques pures). Cette absence de formation universitaire dans le domaine, n'a pas rebuté Gil Utard, qui a choisi de me prendre en stage de DEA puis en thèse au sein du LaRIA grâce à une allocation de la région Picardie. Il m'a ainsi permit d'accomplir cette thèse, en procurant le matériel, m'orientant sur les idées porteuses et m'initiant au travail de chercheur qu'est la conception d'articles. J'ai ainsi utilisé ce matériel pour l'adapter à nos travaux grâce aux nombreux codes ouverts tels que Linux et aux nombreuses documentations disponibles sur le matériel tels que les processeurs Intel, AMD, les cartes LSILOGIC ou les cartes Myrinet.

Ces travaux m'ont permit de faire plusieurs collaborations. Parmi elles, on trouve l'équipe du LIP6 travaillant sur des cartes haut débit (PCI-DDC) dont j'ai apprécié l'aspect complet de la recherche allant du silicium au pilote de périphérique. La principale et la plus importante collaboration est celle avec Abdou Guermouche qui a été grandement facilité par son entrain et son efficacité. Cette collaboration a nécessité quelques passages au LIP, au cours desquels j'ai été agréablement accueilli. Au cours de cette thèse j'ai également enseigné à Amiens, puis à Saint-Quentin et à Beauvais. Ces enseignements ont été facilité par les diverses équipes pédagogiques et en particulier par Stéphane Nègre qui m'a beaucoup appris au points de vu pédagogie lors de mon passage à Saint-Quentin.

Une thèse ne se résume pas au simple travail, elle s'accompagne de multiples phases de questions et de découragement. J'ai ainsi eu les réponses à une bonne partie de ces questions au sein du LaRIA et plus particulièrement par les doctorants et certains enseignants, je tiens particulièrement à remercier Bernard Jurkowiak que j'ai si souvent empêché de travailler par nos nombreux débats. Une vue sur la thèse extérieure est également essentielle pour garder un certain recul est un bon moral, je remercie pour cela de nombreux amis dont Patrick Bonhoure, Eddy Godelle et surtout Sandrine Dubois pour nos nombreuses discutions.

Ce travail de thèse a débouché sur l'écriture d'articles, la présentation d'exposés et la rédaction du manuscrit de thèse, avec de nombreux remaniements et corrections pour ce dernier. Lors de ces étapes j'ai naturellement été beaucoup aidé par Gil Utard, mais aussi par Cyril Randriamaro, Christophe Cerin puis pour les corrections du manuscrit de thèse de Béatrice Vedel, Sabine Bruaux et Bernard Jurkowiak.

J'ai ensuite soumis ce manuscrit et le travail qu'il représente à un Jury que je remercie dans sa globalité pour son intérêt à mon travail. Ce jury était composé de trois rapporteurs qui sont Bernard Lecussan, Raymond Namyst et de Loïc Prylli que je remercie pour leurs lectures assidues et leurs remarques constructives sur le manuscrit et le travail, d'un président Jean-Frédéric Myoupo qui m'avait quelques années plus tôt admis en DEA, de Christophe Cerin et bien sûr de Gil Utard.

Table des matières

Introduction

De nombreuses applications manipulent de grandes masses de données, c'est particulièrement le cas dans le domaine du calcul *scientifique* et *technique* (calcul des structures, simulation, conception assistée par ordinateur), *médical* (imagerie), ou *commercial* (*data-mining*). Ces applications ont longtemps été réservées aux architectures dédiées. L'émergence d'un nouveau type d'architecture basée sur des PC standard interconnectés par des réseaux rapides, représente une alternative bon marché. Néanmoins, ces nouvelles architectures n'ont pas été conçues à priori pour le traitement de grandes masses de données et souffrent donc de sérieuses limitations pour les E/S intensives et le calcul *out-of-core*. Cette thèse se concentrera sur ce dernier point à travers l'étude et la modification des mécanismes de pagination et des entrées/sorties distantes.

Traitement de grandes données

La recherche en parallélisme s'est attachée à construire avec succès des solutions matérielles et logicielles pour le calcul et la communication des nouvelles architectures parallèles. De ce fait, elles sont aujourd'hui capables d'atteindre des puissances de calcul de plusieurs dizaines de Téraflops (milliard d'opérations par seconde). Un point crucial pour exploiter pleinement ces machines est de pouvoir traiter des masses de données qui se mesurent en Gigaoctets, voir en Téraoctets. Ces quantités, que l'on rencontre souvent dans les applications scientifiques (simulation par exemple), financières ou commerciales (*data-mining* par exemple) sont largement supérieures à la mémoire centrale disponible sur ces machines parallèles et obligent à utiliser les disques pour les accueillir. Les problèmes induits concernent alors l'organisation et l'accès aux données. On remarque en effet que le temps d'accès aux disques est largement prédominant sur le temps de calcul ou de communication.

Réseau de PC haut débit

Les machines parallèles ont été par le passé essentiellement conçues avec des composants propriétaires (processeur, réseau, système). De telles architectures étaient onéreuses. Ce type d'architecture est généralement réservé aux grands centres de calcul. D'autre part elles ont du mal à suivre les progrès d'intégration des composants. Ainsi, de nombreux constructeurs (TMC, Intel par exemple !) ont arrêté le développement de telles machines ou ont traversé de graves crises commerciales et financières (voir les déboires de Cray).

Une alternative est l'utilisation de grappes de PC (appelées aussi *cluster*s) interconnectés par des réseaux locaux à très hautes performances (réseau Myrinet, ATM, Ethernet Gigabit, ...) avec des débits de plusieurs Gigabits par seconde et des temps de latences inférieurs à 5 μs. L'ensemble opère le plus souvent sous Linux, avec une couche système additionnelle de virtualisation (environnement multi-threads, bibliothèques de communication, dispositif d'équilibrage dynamique de charge, ...).

D'un point de vue économique, l'objectif est de montrer qu'il est possible aujourd'hui d'offrir des puissances de calcul qu'il n'était possible jusqu'alors d'obtenir qu'avec des machines complexes et chères. Avec la réduction des prix des réseaux à haut débit et les évolutions des systèmes, chaque laboratoire scientifique peut disposer de moyens de calcul performants à partir de son équipement informatique de base.

Recherche sur les grappes

La recherche sur les grappes à plus d'une dizaine d'années, elle a commencé par l'assemblage de composants couramment utilisés pour les stations de travail. Des systèmes ouverts tels Linux, ainsi qu'un grand nombre de matériels programmables et documentés (documentations complétées par du reverse engineering), ont permis de modifier les systèmes pour les adapter aux calculs sur ces grappes. Ces adaptations se sont divisées en quatre grands axes de recherches : les mécanismes de communications rapides, les mémoires virtuellement partagées, les systèmes de fichiers parallèles et la gestion des processus globaux.

Les mécanismes de communication : Les premiers mécanismes de communication étaient basés sur le protocole TCP/IP et des cartes Ethernet. Le problème était le faible débit et la latence importante. Cette dernière étant pénalisante pour les applications fortement couplées. Afin d'augmenter le débit et de diminuer la latence, de nombreuses autres implémentations de communications basées sur le passage de messages ont été conçues. Chacune introduisant un nouveau protocole. Afin d'obtenir des latences plus faible, l'idée était d'augmenter la réactivité à l'arrivé des données, c'est le cas de Gamma[22] qui appelle la procédure de gestion des messages avant la fin de l'arrivée de celui-ci. Une autre idée vient du coût des copies de données par rapport au débit réseau, les implémentations de protocoles de passages de messages rapides tels que Gamma permettent d'éviter cela grâce à l'écriture directe des données dans l'espace utilisateur. Puis des cartes réseau plus évoluées ont permis d'embarquer les routines de communication dans la carte réseau, permettant ainsi de faire des protocoles de passage de messages sans appel système, les rendant beaucoup plus réactifs. Cette dernière technique a vu le jour sous la forme de BIP [59] puis a été suivie par d'autres implémentations telles que GM [64], PM [50, 55], Active Message [12] ou des implémentations de la pile TCP/IP dans la carte réseau (Trapeze [7]). Ces divers protocoles n'ont pas la même interface. Néanmoins, chacun de ces protocoles propose une implémentation haut niveau exhibant une interface de passage de messages au standard MPI [23].

La mémoire virtuelle partagée : Il est plus évident de programmer une grappe si l'on dispose de mémoire partagée, la programmation étant effectuée en divisant en petite tâches appelées *thread* déjà largement utilisé dans les systèmes à temps partagés. Pour cela, de nombreux systèmes de mémoire partagées ont été conçus. Une première classe est basée sur la gestion par le matériel, c'est le cas des multiprocesseurs (*SMP*), de SCI[11], Sebring [78] et d'autres projets permettant d'établir un pont entre les bus de plusieurs machines.

Une deuxième classe, la plus importante, se base sur une gestion logicielle. Il y eu de nombreuses implémentations telles que Orca, IVY, Alewife et Blizzard-S. Une comparaison en est faite par Schoinas et al[73], ils en déduisent une technique logicielle permettant des performances similaire à une *DSM* matérielle. Cette technique requièrt la modification du code de l'application et est implémenté dans Blizard-E.

Ces implémentations utilisent pour la majorité une cohérence forte des données, dite cohérence séquentielle, ce qui permet de simuler exactement le comportement d'un multiprocesseur (*SMP*). Cette gestion de la cohérence engendre des dégradations de performances importantes qui tombent parfois en dessous d'une exécution séquentielle. Afin de limiter ces pertes, il a été conçu des modèles de cohérence de mémoire dits relâchés [38]. Ces modèles se situent entre la cohérence séquentielle et la communication explicite, permettant ainsi de concilier la facilité d'utilisation de la mémoire partagée avec l'efficacité du passage de messages explicite. De nombreuses implémentations utilisant cette cohérence relâchée ont été faite, parmi elles ont trouve TreadMarks [52], RThread [31].

Quelques travaux connexes, mais ne recréant pas une *DSM*, ont été entrepris sur la gestion de la mémoire dans une grappe. Ainsi, GMS [79] permet une gestion globale de la mémoire en permettant de paginer non plus seulement sur le disque local, mais aussi dans la mémoire des autres machines et de leurs disques. Des travaux étudient aussi le comportement de la mémoire, c'est-à-dire de la pagination au sein de chaque machine, c'est le cas de Glass [39] et Krueger [54] qui proposent des politiques alternatives de gestion de la pagination.

Ces derniers travaux ainsi que beaucoup de *DSM* nécessitent de nombreuses modifications du système, même si certains systèmes à audience relativement confidentielles tels que Mach [72], Grasshopper [3] ou les micronoyaux L3/L4 [43] permettent une modification de la pagination sans modification du noyau du système.

Les systèmes de fichiers parallèles : Comme pour la mémoire, il est plus aisé d'avoir un système de fichiers commun à la grappe, ce qui permet d'éviter de nombreuses communications explicites. Une possibilité est l'utilisation d'un serveur centralisé tel que le *NFS* (SUN) qui donne accès aux fichiers pour toutes les machines, mais qui ne procure pas de cohérence. Ce système a été amélioré pour communiquer plus rapidement avec les clients (DAFS [28]). Il est utile que ce serveur tolère des pannes, ce qui est réalisé au point de vue matériel ou logiciel par des techniques regroupées sous le nom de RAID [71]. Cependant, un serveur centralisé crée un goulot d'étranglement pour la gestion des fichiers. Une autre possibilité consiste en l'utilisation des disques de chaque machine pour faire un système de fichiers global. L'intérêt est l'agglomération des débits des disques et de l'espace de stockage. Afin de profiter au mieux des débits, la distribution des données est essentielle. Ces distributions sont, sur la majorité des systèmes de fichiers parallèles, aiguillées par une interface qui en plus des classiques opérations d'entrées/sorties ajoute des opérations de spécification explicite du placement des données. Les premiers systèmes de fichiers conçus sur ces bases ont été écrit en mode utilisateur (BPFS, PVFS,. . .), pour ne pas avoir à modifier le système d'exploitation. Cependant, l'écriture en mode utilisateur ne permet pas la réutilisation des commandes usuelles. Pour cette raison, certains, comme PVFS, ont été, en partie, réécrits pour pouvoir être intégré dans les systèmes d'exploitation.

Ces diverses recherches ont conduit à l'élaboration de nombreux systèmes de fichiers parallèles parmi lesquels on trouve BPFS [2], PPFS [46], HFS [53], CFS [14], CCFS [21], PVFS [57] ou GFS [68]. Cependant, peu de ces projets perdurent, et il semble que seul deux systèmes de fichiers continuent à être utilisés : PVFS et GFS.

Gestion des processus globaux : Les communications, la mémoire et les fichiers étant gérés, il faut aussi gérer l'application de manière globale. En effet, il faut que les processus puissent migrer d'une machine à une autre, ce que permettent des gestions de THREAD comme PM2[65], qui peuvent être complétées par une gestion des ressources mémoire comme dans Mosix [10]. Il est aussi possible de gérer plusieurs applications au sein de la grappe, ce que permet Score [50] par un ordonnanceur global.

Ces quatre aspects, les communications, la mémoire, le système de fichiers et la gestion globale de l'application permettent de concevoir une machine parallèle à partir d'une grappe de machine, et son regroupées sous le terme de système à image unique. On trouve quelques projets qui rassemblent une partie de ces caractéristiques : Score [50], Mosix [10].

Objectif

Cette thèse a pour objectif d'étudier et de définir des concepts et des outils nécessaires pour résoudre efficacement les problèmes de calculs intensifs sur de grandes masses de données sur les réseaux à haut débit de stations : les grappes (ou *cluster* en anglo-saxon). Les masses de données considérées sont de tailles nettement supérieures à la mémoire physique disponible. Nous étudierons donc les aspects architecturaux et systèmes mis en œuvre. Les applications visées étant essentiellement les problèmes de calcul numérique basé sur l'algèbre linéaire, tels que les modélisations d'écoulement de fluide, et les problèmes de gestion de flux.

La plupart des travaux précédemment cités ont pour objectif d'améliorer les performances d'applications traitant des données présentes en mémoire. Seul les travaux sur les systèmes de fichiers parallèles correspondent en partie à notre objectif de traitement de grandes masses de données. Cependant, ils ont été conçus principalement pour améliorer l'accès aux données (parallélisme des entrées/sorties et gestion de cache), mais il ne tiennent pas compte des traitements à effectuer.

Cette thèse est donc un premier pas dans la définition de nouvelles briques de base pour le calcul *out-of-core* intensif sur grappes. La première est un nouveau mécanisme de gestion de la pagination qui permet d'adapter celle-ci aux calculs. La seconde brique de base est un nouveau mécanisme d'entrées/sorties qui réduit au maximum l'utilisation des ressources de la grappe, à l'instar des protocoles de communication « zéro copie », libérant celles-ci pour le calcul et les communications.

Gestion de la mémoire virtuelle

Puisque l'objectif est de pouvoir traiter de grandes données (e.g. des matrices de très grande taille), une première idée consiste à utiliser les gestionnaires de mémoire virtuelle présents sur tous les systèmes.

Bien entendu, une telle solution est totalement inefficace, de nombreuses études l'ont démontré, telles que celles de Park[67] dans la comparaison avec les entrées/sorties explicites ou Krueger[54] qui montre les limites sur une application. Une première solution consiste à restructurer le code de façon à avoir l'ensemble des données utiles en mémoire centrale à l'instant propice, i.e. d'exploiter la localité temporelle et spatiale de l'application.

La seconde approche consiste à modifier le gestionnaire de pagination pour l'adapter à des applications spécifiques. Cette deuxième approche peut être étendue pour établir un dialogue entre l'application et la pagination. Ce dialogue permettra d'affiner les choix du gestionnaire de pagination. Malheureusement, la pagination est gérée en *mode noyau* (le mode dans lequel s'exécute le système d'exploitation) alors que l'application s'exécute en mode utilisateur. Peu de fonctionnalités sont prévues pour pouvoir transférer la pagination en mode utilisateur. A cette fin, nous présentons un nouveau mécanisme système qui permet de reporter la gestion de la mémoire virtuelle au niveau de l'application même. Nous montrerons ainsi que grâce à la connaissance des accès en mémoire de l'application, nous pouvons dans certains cas réduire significativement les temps d'exécution de l'application, c'est le premier apport de notre travail. Ceci nous permettra de définir une méthode de contrôle générique, permettant d'influer sur la pagination. Ce sera le premier apport majeur de cette thèse.

Entrées/Sorties Parallèles

La pagination et le traitement de flux nécessitent d'accéder à des données qui résident sur disques, il faut que ces accès soient les plus performants possibles. D'un point de vue physique, l'évolution des performances des disques, bien que réelle, n'a rien de comparable avec l'évolution des performances des processeurs et des réseaux. La seule solution efficace consiste à *paralléliser* les entrées/sorties en répartissant les données sur tous les disques des stations du réseau, et en effectuant des requêtes simultanées. C'est la solution adoptée pour les RAIDs [71] (Redundant Array of Independant Disks). Cette solution permettant de s'abstraire de la gestion de multiples disques, nous allons pouvoir nous concentrer sur les systèmes de fichiers. Nous allons étudier dans quelles mesures le système de fichiers pourrait être intégré sur les nœuds de communication plutôt que sur les nœuds de calcul. En effet, lors des opérations d'entrées/sorties sur un PC, les données transitent par le processeur hôte, gaspillant inutilement la puissance de calcul et la bande passante de la mémoire. Par ailleurs, les cartes réseaux modernes sont programmables et possèdent aujourd'hui des processeurs élaborés et une mémoire conséquente. Cette caractéristique est utilisée dans cette thèse pour gérer les entrées/sorties parallèles, sans passer par le processeur hôte : les disques sont partagés.

Nous étudions expérimentalement l'impact sur les performances globales que peut offrir une telle architecture système. Nous constatons effectivement les gains permis par cette approche. Nous précisons donc comment intégrer ces techniques dans un système de fichiers parallèle et distribué. C'est le deuxième apport majeur de notre travail.

Contributions

L'ensemble de nos contributions scientifiques est publié dans les actes de plusieurs conférences nationales et internationales. Ainsi, la première partie de cette thèse a donnée lieu à deux articles [19, 20] sur les problèmes liés à la pagination. Nous y avons présenté l'outil *MMUM/MMUSSEL* permettant de contrôler la pagination en mode utilisateur et une utilisation de cet outil pour réduire l'activité pagination d'une factorisation LU. Nous avons soumis nos travaux concernant l'amélioration de cet outil est son utilisation avec le solveur multifrontal *MUMPS*, notre outil nous permettant des gains allant jusqu'à 40% en temps et de l'ordre de 30% en activité de pagination.

La deuxième partie de la thèse a d'abord donnée lieu à deux articles[24, 26] où l'on expose une modélisation et une étude des gains du passage direct des données du disque à la carte réseau. Cette technique pouvant permettre des gains allant jusqu'à 50%, nous l'avons implémenté en mettant le pilote dans la carte réseau (READ^2) et nous avons présenté nos résultats dans deux conférences[25, 27].

Ces contributions, tant sur le plan conceptuel que sur le plan de l'expérimentation, nous permettent d'envisager des améliorations de la pagination pour de nombreuses classes de problèmes, tels qu'une cogestion de l'ordonnancement des applications parallèles ou de permettre une gestion améliorée de la mémoire et des entrées/sorties grâce à notre outil de pagination et notre outil d'accès aux disques distants.

Première partie

Pagination adaptative

Introduction

Une application a besoin d'un certain espace pour ses données. Le système dispose de mémoire physique qu'il peut attribuer à cette application. Mais la taille de cette mémoire peut être inférieure à la taille des données de l'application. Le moyen pour limiter ce problème est de ne stocker qu'une partie des données dans la mémoire. Dans ce cas l'application stocke les données supplémentaires sur un disque et les charges ou les décharges de la mémoire suivant ses besoins. Cette gestion des données en mémoire et des données sur disque peut se faire par deux moyens : les entrées/sorties explicites ou l'utilisation de la mémoire virtuelle. La mémoire virtuelle donne l'illusion d'avoir une mémoire plus grande et c'est le système qui charge/décharge les données de la mémoire.

La mémoire virtuelle rend l'application plus aisée à programmer qu'avec une gestion explicite des entrées/sorties [67]. L'application considère qu'elle n'accède qu'à la mémoire. Le problème est qu'il n'y a pas de dialogue entre le système d'exploitation et l'application : il est donc possible que le système d'exploitation charge des données inutiles ou décharge des données utiles, ce qui engendre une perte de performances par rapport aux entrées/sorties (E/S) explicites.

Dans *Maîs* [17, 16], P. Cadinot et al proposent un mécanisme de pagination à distance qui permet d'utiliser la mémoire des machines distantes comme périphérique de stockage secondaire sur des cartes *FastHSL* [40] d'écritures à distances. Une technique semblable permet une gestion globale de la pagination d'une grappe dans *PGMS*[79].

Néanmoins, ces techniques sont implémentées en mode noyau, ce qui rend leurs modifications et leurs extensions difficiles. De plus, *PGMS* [79] n'est pas disponible pour des systèmes d'exploitation usuels tels que *Windows* ou *Linux*. Une gestion de la pagination en mode utilisateur pourrait limiter ces problèmes et permettre d'adapter plus aisément la pagination aux applications. Une interface permettant cette gestion est décrite par Keith Krueger [54]. Ce travail n'a jamais eu de suite et il n'existe pas d'implémentation. Une implémentation d'une technique semblable est décrite dans la thèse d'Andrea Haeberlen[42]. Malheureusement elle est effectuée sur un micronoyau *L3* [56] qui ne permet pas de porter aisément ces travaux sur des systèmes d'exploitation courants.

Dans cette partie, nous proposons une définition d'un mécanisme de contrôle de la pagination qui s'inspire des travaux précédents. Ce mécanisme a été implémenté sous *Linux*, et représente 2200 lignes en C.

Cette partie est structurée de la manière suivante, nous décrirons le fonctionnement de la pagination et les limites des techniques usuelles. Nous présenterons notre outil de gestion de la pagination. Nous décrirons l'intérêt de cet outil sur un exemple. Puis nous montrerons que cet outil permet de redéfinir un gestionnaire de pagination général. Nous présenterons l'évolution de cet outil pour améliorer la pagination du solveur *MUMPS*.

Chapitre 1

Mémoire virtuelle

Sommaire

Pour le traitement de grandes masses de données, l'idée première est de laisser le gestionnaire de pagination du système se charger de l'échange des données entre la mémoire principale et le disque. Néanmoins, de nombreuses études montrent les limites du système lorsque la masse de données est grandement supérieure à la mémoire disponible. Par exemple des études telles que celles de Keith Kruger [54] ou Glass [39] montrent des cas où une pagination différente de celle du système d'exploitation donne de meilleures performances.

Dans ce chapitre, nous rappellerons le fonctionnement de la pagination au niveau du processeur puis au niveau du système et nous exposerons quelques exemples pour lesquels la pagination du système est inadaptée.

1.1 Mécanismes de translations d'adresse

La mémoire virtuelle donne l'illusion à l'application que toute la mémoire dont elle a besoin est directement utilisable. En toute rigueur, il faut introduire la notion d'espace d'adressage. Chaque donnée de la mémoire est accédée par son adresse. La taille de ces adresses varie suivant les architectures (32bits sur les processeurs Pentium et 64bits sur les Alpha AXP). La taille de ces adresses détermine le nombre maximum de données pouvant être accédées : c'est la taille maximale de l'espace d'adressage. Ainsi, avec des adresses de 32bits, il est possible d'accéder à au plus 4 milliards d'octets (2^{32} d'octets) et 16.10^{18} octets avec 64bits. Les processeurs modernes ont deux types d'adressages : l'adressage physique pour lequel à chaque adresse correspond un et un seul emplacement dans la mémoire physique, et l'adressage virtuel qui permet de disposer de plusieurs espaces d'adressage par un mécanisme de translations d'adresses.

Avec la translation d'adresses, une adresse ne correspond plus directement à un emplacement dans la mémoire physique. En effet, pour obtenir l'adresse réelle d'une donnée (son adresse physique) le processeur utilise une table qui associe à chaque adresse un emplacement de la mémoire physique ou une erreur signifiant que la donnée n'est associée à aucune mémoire physique. Cette table ne peut contenir une correspondance pour chaque adresse, car elle serait de la même taille que la mémoire. Pour cela, la mémoire virtuelle est divisée en blocs appelés pages (variant de 4Ko à 4Mo suivant les architectures). Chacune de ces pages peut-être associées à une page physique contiguë de même taille en mémoire physique. Les figures. 1.2 et 1.1 présentent l'utilisation de ces tables pour obtenir une adresse physique à partir d'une adresse virtuelle. A chaque page de mémoire virtuelle est associée à divers attributs tels que :

- La présence : indique si la page est en mémoire, c'est-à-dire si une page physique correspond à cette page virtuelle. Dans le cas où une page est marquée non présente, l'accès à cette page par le processeur génère un *défaut de page* qui exécute une fonction du système d'exploitation. Cette fonction se chargera de trouver une page physique libre, d'y charger les données et de l'associer à cette page virtuelle.
- Les droits d'accès : indiquent si les données de la page peuvent être lues, exécutées, écrites ou si seul le système peut y accéder. Ces paramètres permettent par exemple l'isolation et la protection du système d'exploitation par rapport aux processus utilisateurs.

L'ensemble de ces informations est stocké dans une table appelée table de translation. Ainsi, avec une architecture telle que l'Alpha AXP, chaque page virtuelle fait 8Ko et nécessite 8 octets d'information (adresse de la page de mémoire physique associée et attributs). Pour un espace d'adressage de 16.10^{18} octets, il faudrait une table de 16.10^{15} octets. Ceci n'est pas acceptable, la mémoire physique de telles machines ne dépassant pas quelque dizaine de milliards d'octets (Go). Pour réduire cette taille, la table de translation est organisée de manière hiérarchique avec plusieurs niveaux d'indirection, 2 ou 3 sur x86 et 3 sur Alpha AXP. Ainsi, dans le cas extrême où une seule page virtuelle est associée à une page physique, le répertoire de page (racine de l'arbre) ne pointe que sur un répertoire de pages secondaires, qui pointe lui-même sur une table des pages contenant l'adresse de la page physique. Ce qui ne fait que 24Ko pour gérer 16 milliards de milliards d'octets virtuels.

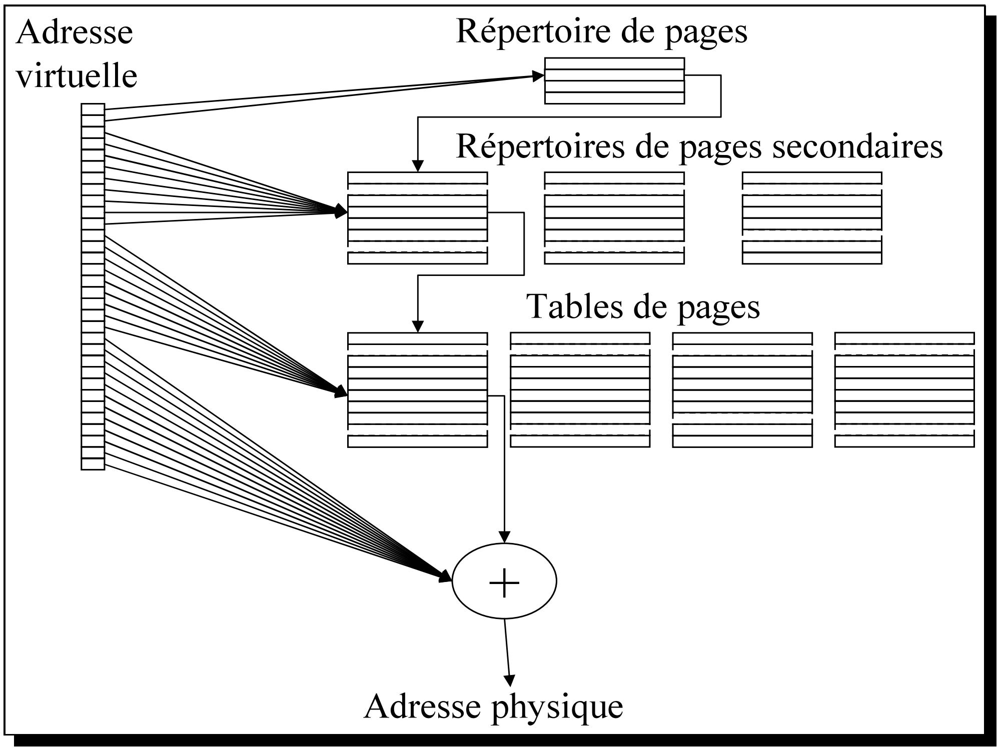


FIG. 1.1 – Translation d'une adresse virtuelle en adresse physique

Penons l'exemple d'un processeur x86 qui dispose de trois niveaux de pages (lorsque l'option PAE[a] est activée.) : chaque page fait 4ko, les structures de données permettant l'accès à l'adresse physique sont :

struct repertoire_de_pages	struct repertoire_de_pages_sec	struct table_des_pages
{	{	{
int present ;	int present ;	int present ;
int attributs ;	int attribut ;	int attribut ;
struct repertoire_de_pages_sec *	struct table_des_pages *	void *
repertoire_de_pages_sec[2048] ;	table_des_pages[2048] ;	adresse
} repertoire_de_pages[4] ;	} ;	} ;

Montrer le mécanisme de translation pour une adresse :

– Soit $0xABCDEF12$ l'adresse virtuelle pour cet exemple.
– Les deux bits de poids fort valent 2, l'adresse du répertoire de page est ainsi stockée dans $repertoire\_de\_page[2]$, comme $repertoire\_de\_page[2].present$ est vrai, ce répertoire est en mémoire.
– Les neufs bits de poid fort suivant valent $0x15E$, l'adresse du répertoire de page secondaire est ainsi stockée dans :
  $repertoire\_de\_page[2]- > repertoire\_de\_page\_sec[0x15E]$

  Comme $repertoire\_de\_page[2]- > repertoire\_de\_page\_sec[0x15E].present$ est vrai, ce répertoire est en mémoire.
– Les neuf bits suivants valent $0x1DE$, le début de la page est ainsi :
  $repertoire\_de\_page[2] - repertoire\_de\_page\_sec[0x15E]- > table\_des\_pages[0x1DE]$.
  Cette page est présente car :
  $repertoire\_de\_page[2]- > repertoire\_de\_page\_sec[0x15E]- > table\_des\_pages[0x1DE].present$ est vrai.
– L'adresse virtuelle est associée à une adresse physique qui est $repertoire\_de\_page[2]- > repertoire\_de\_page\_sec[0x15E]- > table\_des\_pages[0x1DE].adresse + 0xF12$.

[a]PAE : Page Address Extension : permet aux processeurs x86 de gérer 64Go de mémoire physique. Pour cela les entrées dans les tables des pages, les répertoires des pages et le répertoire principal des pages passent de 4 octets à 8 octets par entrée

FIG. 1.2 – Translation d'une adresse sur un processeur x86

1.2 Gestion de la mémoire virtuelle

Quel que soit le système d'exploitation, les techniques de base de gestion de la mémoire sont les mêmes. Ainsi, les systèmes d'exploitation associent une mémoire virtuelle spécifique à chaque processus (l'espace virtuel du processus), ce qui permet de protéger les différents processus entre eux. Chaque processus évolue comme s'il disposait de toute la mémoire virtuelle. Le changement de l'espace virtuel est fait lorsque que le système d'exploitation active un autre processus. Cependant, les processus ont besoin de communiquer entre eux et particulièrement avec le système d'exploitation. Pour cela, le système d'exploitation réserve une partie de l'espace virtuel pour le partage de mémoire entre les processus et une autre partie de l'espace virtuel pour lui (fig. 1.3). Il se réserve ainsi au moins 1Go d'espace virtuel sur les processeurs x86 (que ce soit avec Windows ou Linux) : cela explique pourquoi il est impossible sur x86 de déclarer un tableau de plus de 3Go.

Ce dernier gigaoctet ne peut être accédé que dans le mode système, qui est le mode dans lequel est exécuté le système d'exploitation. Dans ce mode, le processeur peut accéder à toutes les ressources, et en particulier les tables de translations. Ce mode peut être activé soit par des interruptions telle que la frappe d'une touche au clavier ou la réception de données par la carte réseau, soit par des appels systèmes venant du processus. Ces appels systèmes permettent l'appel de fonctions du système d'exploitation en assurant la transition entre le mode utilisateur et le mode du système d'exploitation appelé mode noyau. Ces appels systèmes sont généralement coûteux en terme de temps processeur.

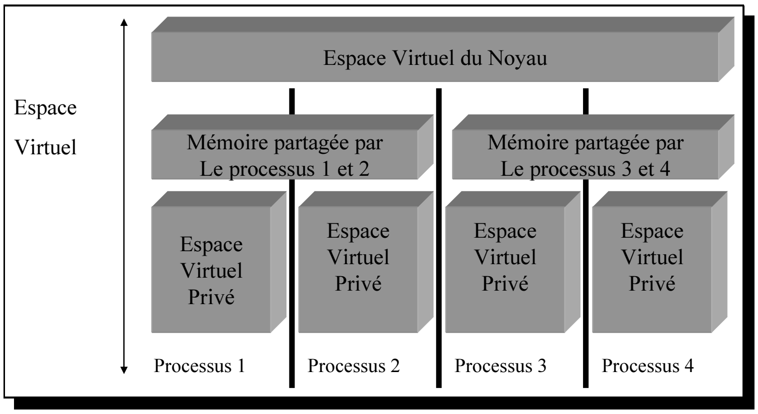


FIG. 1.3 – Division de l'espace virtuel entre le noyau et le processus

1.2.1 Correspondance entre la mémoire virtuelle et secondaire

Les 3Go (ou plus pour des architectures 64bits) attribués à l'application peuvent contenir des objets très divers tels que le fichier contenant le code du processus, la pile, le tas, des librairies partagées, de la mémoire partagée entre processus, ou encore des fichiers projetés en mémoire. Pour gérer ces objets, le système d'exploitation divise, pour ses besoins internes, la mémoire virtuelle de l'application en régions mémoire, chacune correspondant à un objet particulier. Chacune de ces régions peut-être en lecture seule, en écriture, exécutable, associée à un fichier ou partagée.

Les objets fichiers

Lorsqu'une région de mémoire correspond à un fichier ou à un programme, chaque page virtuelle de cette région correspond à un emplacement dans le fichier. Ainsi, si une page de la région n'est associée à aucune page physique (i.e. non présent), le système saura quelles données charger dans une page physique pour rendre présente cette page. De même, lorsque le système d'exploitation voudra libérer une page physique associée à une page virtuelle de cette région, il sauvegardera ses données dans le fichier (si elles ont été modifiées).

Les objets utilisant le *swap*

Une région peut aussi correspondre à des données non associées à des fichiers prédéterminés, mais à la pile, au tas et aux données globales d'un programme. Dans ce cas, il utilise des périphériques d'échanges (*swap device* ou *swap*), qui sont généralement des partitions de disques, ou alors des fichiers spécifiques appelés fichiers de swap. Ces périphériques se caractérisent par de grandes capacités, mais avec des temps d'accès très importants : ils sont environ 100 fois plus lent que la mémoire physique. Lorsque le système veut libérer un page physique, il choisit un emplacement dans le swap, il stocke la page dans cet emplacement, et il marque la page virtuelle associée comme absente et conserve l'adresse de cet emplacement. Lorsque la page virtuelle est réassociée à une page physique, les données sont relues depuis le swap.

Il est important de noter que les demandes d'emplacements se font suivant les besoins du système. Si le système évince (libère les pages physiques associées) les pages 1, 10, 2 et 5 dans cet ordre, la page 1 pourrait avoir l'emplacement 1 dans le périphérique d'échange, la page 2 l'emplacement 3, la 10 le 2 et la 5 le 4. Les autres pages n'auront pas d'emplacement associé, car elles seront en mémoire. Il faut d'ailleurs noter que pour économiser la mémoire, le système ne réserve ni d'emplacement dans un périphérique d'échange ni de page physique dans la mémoire pour les pages virtuelles qui n'ont jamais été accédées.

1.2.2 Politique d'éviction des pages

Nous venons de voir comment le système peut libérer de la mémoire physique ou charger des pages virtuelles. Maintenant, la question est de savoir comment le système décide de charger des pages virtuelles ou d'en évincer. Le but du système est d'avoir le moins d'échanges possible, c'est-à-dire qu'il y ait le moins de *défaut de page*. Pour cela, il convient d'avoir en mémoire physique les pages virtuelles qui vont être utilisées dans un avenir proche. A cette fin, plusieurs techniques sont possibles. Voici les deux politiques les plus couramment utilisées :

La LRU (*Least Recently Used*) : les pages virtuelles à garder en mémoire sont celles qui ont été le plus récemment accédée, les autres peuvent être évincées de la mémoire. Pour être efficace, cette technique suppose une localité temporelle des accès ce qui est vrai pour beaucoup de programme.

La FIFO (*First In First Out*) : les pages virtuelles à garder en mémoire sont celles qui ont été le plus récemment chargées, cette politique est adaptée aux parcours de flux. Une variante, l'algorithme FIFO seconde chance, garde les pages pendant un certain temps dans un cache avant leurs évincement, ce qui permet, si elles sont réutilisées, de redevenir les pages les plus récemment chargées sans accès au disque.

1.2.3 Exemple du gestionnaire de pagination Linux

Nous allons présenter plus précisément la gestion de l'éviction des pages de Linux 2.4. Linux utilise un algorithme pseudo LRU. La différence vient de l'utilisation d'un intervalle de temps pour mesurer la dernière utilisation d'une page et non la date exacte. A cette fin, Linux utilise un processus noyau (*kswapd*), qui est activé suivant la nécessité de libérer de la mémoire. Ce processus observe l'utilisation et la modification des pages afin de les classer dans une des trois listes suivantes (fig. 1.4) :

***active_list* :** cette liste contient les pages qui ont été accédées, c'est-à-dire lues ou écrites récemment.

***inactive_dirty_list* :** cette liste contient les pages qui ne sont plus accessibles et qui doivent être écrite sur le disque pour pouvoir être libérées.

***inactive_clean_list* :** cette liste contient les pages qui ne sont plus accessibles et qui ne nécessite pas ou plus d'être sauvegardées.

Pour faciliter ce classement, ainsi que d'autres opérations comme le besoin de poser des sémaphores sur l'utilisation d'une page, Linux associe à chaque page physique une structure de données qui chaîne les pages dans une des trois listes précédentes et qui donne l'age des derniers accès au moyen de l'attribut *age*. L'attribut *age* est placé à une valeur maximale à chaque passage de *kswapd* si la page a été accédée depuis le dernier passage, sinon l'attribut est décrémenté.

Après chaque défaut de page, l'age de la nouvelle page physique est initialisée à une valeur maximale et il est placé dans la liste *active_list*.

Après plusieurs passages de *kswapd* sans accès à cette page l'attribut *age* arrivera à zéro, la page physique sera déréférencée. Cela signifie que l'accès à la page virtuelle par le processus générera un défaut de page. Néanmoins, la page physique avec les données de la page virtuelle est conservée et référencée dans la liste *inactive_dirty_list* ou *inactive_clean_list* suivant que la page ait été modifiée depuis la dernière écriture sur le disque ou non. Dans le cas où la page n'a jamais été écrite sur le disque, elle est considérée comme à écrire et un emplacement dans le *swap* lui est assigné.

Lorsqu'une page est dans *inactive_dirty_list* ou *inactive_clean_list*, elle peut par un défaut de page être réassociée à la page virtuelle et retourner dans *active_list*. Suivant les besoins en mémoire et les disponibilités des disques, les pages passent de *inactive_dirty_list* à *inactive_clean_list* lorsqu'elles sont

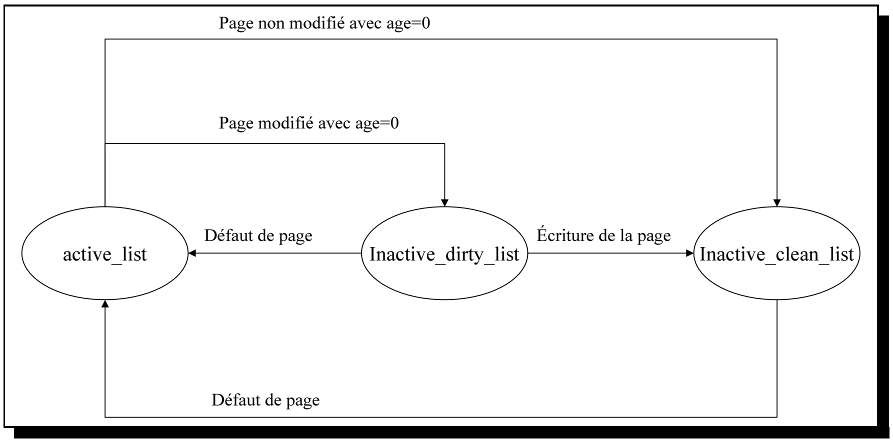


FIG. 1.4 – Gestion des files de pages dans Linux 2.4

écrites sur le disque. Enfin, lorsque le système à besoin de mémoire pour stocker d'autres pages virtuelles, les pages physiques de *inactive_clean_list* sont libérées.

Il faut noter qu'une file *inactive_clean_list* est associée à chaque région mémoire. Ainsi lorsque le système a besoin de mémoire pour une région de mémoire, il cherche en priorité dans la file *inactive_clean_list* associée à cette région.

1.3 Limitation des stratégies usuelles de la pagination

Nous venons d'exposer les mécanismes de la mémoire virtuelle et les stratégies de gestion de celle-ci. Nous allons maintenant exposer deux cas simples qui montrent les limites de ces stratégies, puis nous montrerons que ces problèmes se posent aussi sur des applications d'algèbre linéaire.

1.3.1 Etude de deux cas simples

Les systèmes d'exploitation considèrent que les processus ont un comportement générique basé sur l'hypothèse que les accès mémoire ont une forte localité spatiale et temporelle. C'est-à-dire que les pages utilisées récemment seront utilisées prochainement et qu'à un moment donné, seul un petit sous-ensemble de pages est utilisé. De plus, ils estiment que les prochaines pages à charger sont imprévisibles. C'est pourquoi, les systèmes d'exploitation utilisent, pour la plupart, un gestionnaire basé sur l'algorithme LRU et le chargement des pages à la demande (*demand driven*). D'autre part, il n'y a en général pas de correspondance prédéterminée entre les adresses virtuelles et les adresses dans le fichier de swap, ce qui peut avoir un impact notable sur les performances des entrées/sorties. Ici nous étudierons l'impact de la projection (*mapping*) des pages dans le swap ainsi que de la politique d'éviction des pages sur les performances. Pour cela nous présenterons *l'effet trou* et deux restructurations de code qui peuvent diminuer cet effet, puis nous comparerons ces deux restructurations. Nous étudierons ensuite un exemple où le *mapping* est inadéquat et nous montrerons une restructuration du code qui permet au système de faire un meilleur *mapping*.

Politique LRU et l'effet trou

Prenons maintenant un vecteur X de n pages mémoires et réalisons une multiplication de ce vecteur par une constante k grâce au programme suivant :

```
/* Multiplication d'un vecteur */
while (1==1)

   for (i=1;i<n;i++)
      X[i]=X[i]*k;
```

Supposons que le vecteur *X* contienne une page de plus que la mémoire physique disponible. Comme le montre la figure 1.5, chaque accès à la page suivante (dans l'ordre d'exécution du programme), provoque un défaut de page. Du point de vue de l'algorithme LRU, la page évincée est malheureusement la page qui sera prochainement accédé : c'est *l'effet trou*.

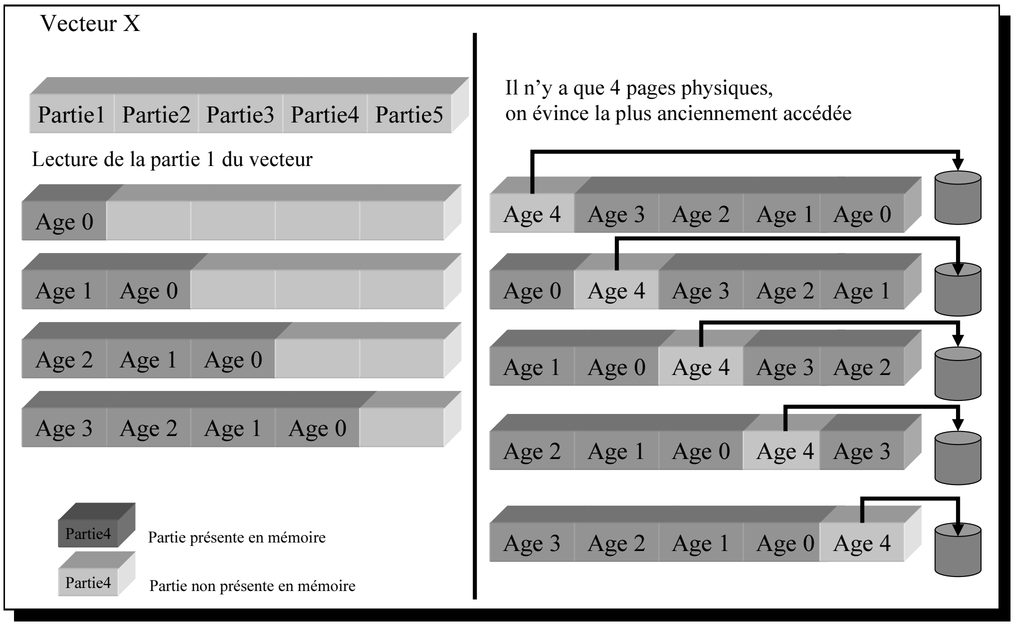


FIG. 1.5 – Suite de multiplications d'un vecteur, la page à évincer pour charger celle qui va être traitée est toujours la prochaine page accédée

Afin d'exposer *l'effet trou*, nous avons exécuté le programme de multiplication d'un vecteur par un scalaire avec différentes quantités de mémoire disponible lors du démarrage de Linux. La figure 1.6 montre que dès que la taille du vecteur dépasse la mémoire de la machine, les performances s'écroulent à cause de l'effet trou. En effet, nous avons vu qu'avec une page de plus que la mémoire, chaque accès à une nouvelle page provoque un défaut de page et l'évincement de la suivante. Dans le cas où il y a n pages en plus, la page évincée est celle qui sera accédée par le programme après les $n-1$ pages suivantes. Ceci place le système dans un cas ou la page suivante n'est jamais disponible : l'effet trou. Nous pouvons ainsi dire que l'effet trou est visible à partir de 20Mo sur la configuration avec 32Mo de mémoire, à partir de 50Mo avec 64Mo de mémoire et à partir de 110Mo avec 128Mo de mémoire. Par définition de l'effet trou, ces valeurs correspondent à la mémoire libre pour le processus.

Nous pouvons également remarquer que les performances sont meilleures avec moins de mémoire. Ce phénomène vient du fait que lorsqu'il y a peu de mémoire, cette mémoire est entièrement prise et le système ne peut pas tenter d'optimiser l'accès au disque. Dans le cas contraire, le système peut réunir les requêtes et pour cela attendre plus de temps avant d'écrire, ce qui se traduit par une attente entre deux demandes d'accès au disque.

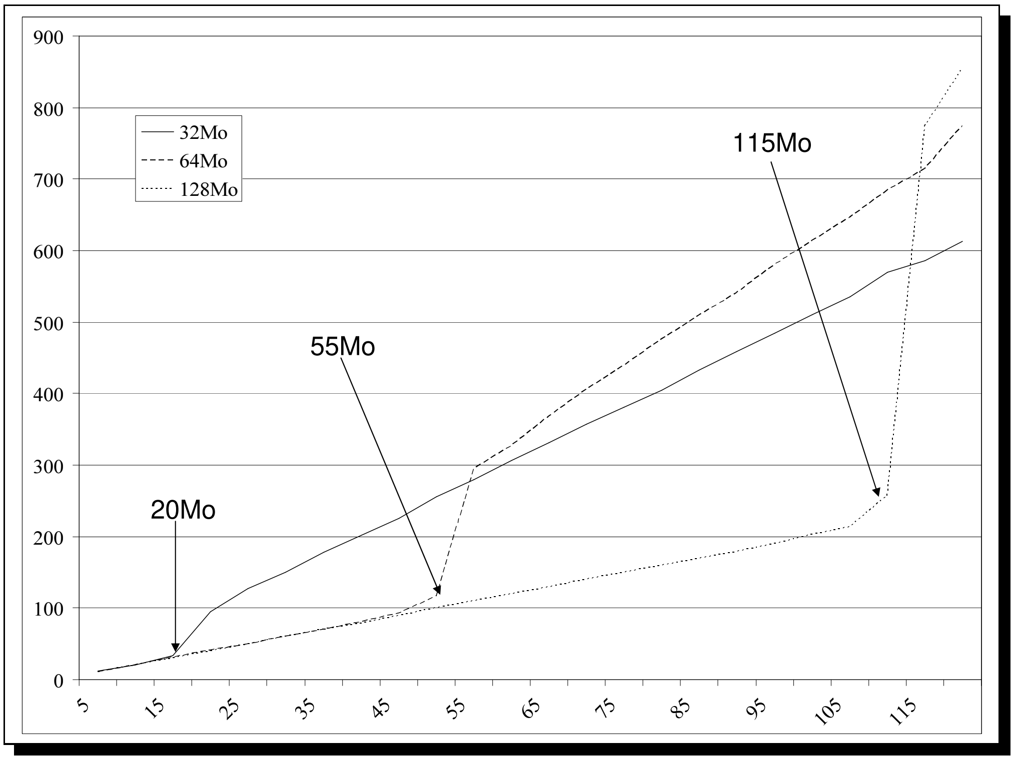


FIG. 1.6 – Lectures successives d'un vecteur et mise en évidence de l'effet trou *Alpha AXP 433Mhz avec disque de 20Mo/s*

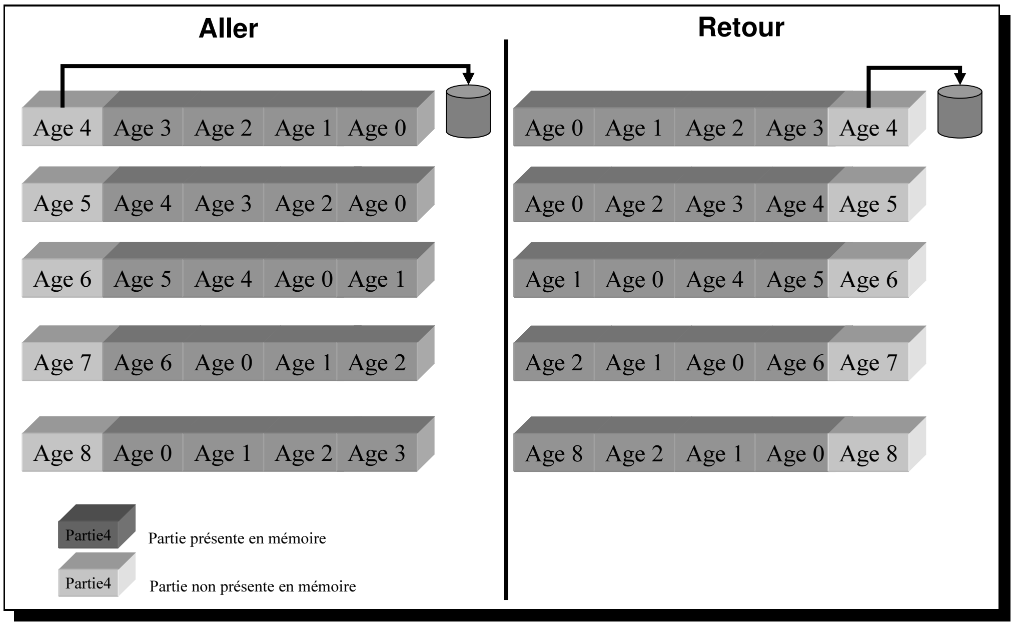


FIG. 1.7 – Suite de multiplications d'un vecteur avec *ZigZag*

Première amélioration : le *ZigZag* Une amélioration consiste à restructurer le code pour que les prochaines pages utilisées soient les plus récemment accédées. Ceci peut être réalisé par la lecture en *ZigZag* du vecteur *X* (fig. 1.7), qui correspond au code suivant :

```
/* Multiplication d'un vecteur */
while (1==1)
   {
   for (i=1;i<n;i++)
      X[i]=(1+X[i]*k)
   for (i=n;i>=0;i--)
      X[i]=(1+X[i]*k)
   }
```

Avec cette technique, il n'y a que deux accès au disque par aller-retour $2n$ dans la méthode originale. De manière plus générale, nous pouvons déduire qu'avec une mémoire de m pages et un vecteur de n pages, il faudra $m - n$ lecture/écriture de pages pour un parcours du vecteur. Nous pouvons ainsi constater sur les figures 1.9, 1.10 et 1.11 que cette dernière technique permet de limiter la perte de performances. L'intérêt principal de cette technique est son indépendance vis-à-vis de la taille de la mémoire, ce qui lui permet d'être facilement implémentée sur des cas simples. Néanmoins, cette technique se comporte moins bien sur les tableaux bien plus grands que la mémoire.

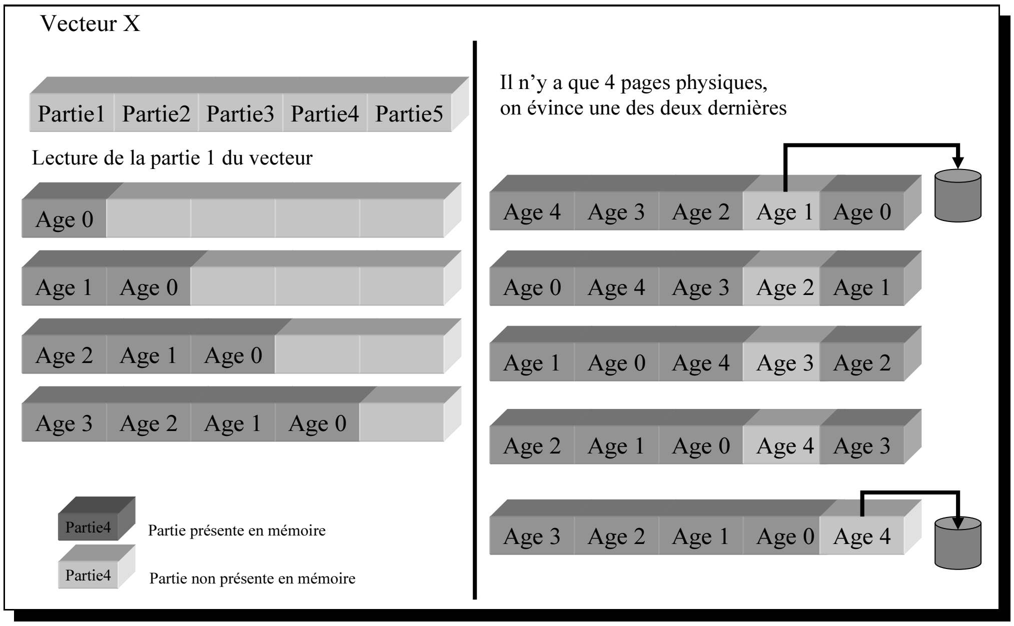


FIG. 1.8 – Suite de multiplications d'un vecteur, il n'y a que deux cas où la page demandée n'est pas présente

Seconde amélioration : le verrouillage des pages physiques Le nombre de lectures/écritures du disque généré par l'effet trou peut pourtant être évité avec un algorithme naïf : les *n-2* premières pages du vecteur restent en mémoire, la dernière page de mémoire libre contient alternativement la page *n-1* ou la page *n* du vecteur (voir la fig. 1.8). Dans ce cas nous n'avons plus que deux lectures et deux écritures sur le disque à chaque parcours du vecteur, ce qui montre encore une fois que l'algorithme LRU n'est pas adapté à ce cas. En effet, lorsqu'il y a *m* pages mémoire et que le vecteur contient *n* pages mémoire on descend de *n* lectures et écritures, à *m-n+1* lectures et écritures.

Nous avons testé l'effet de cette technique, pour influencer le gestionnaire de pagination nous avons verrouillé autant de pages mémoire que nous le permettait le système. Le code restructuré par cette méthode est maintenant :

```
/* Verrouillage du maximum de mémoire possible */
k=n ;while (mlock(X,k)<0) k=k-PAGE_SIZE ;



/* Multiplication d'un vecteur */



while (1==1)
   for (i=1;i<n;i++)
      X[i]=(1+X[i]*k)
```

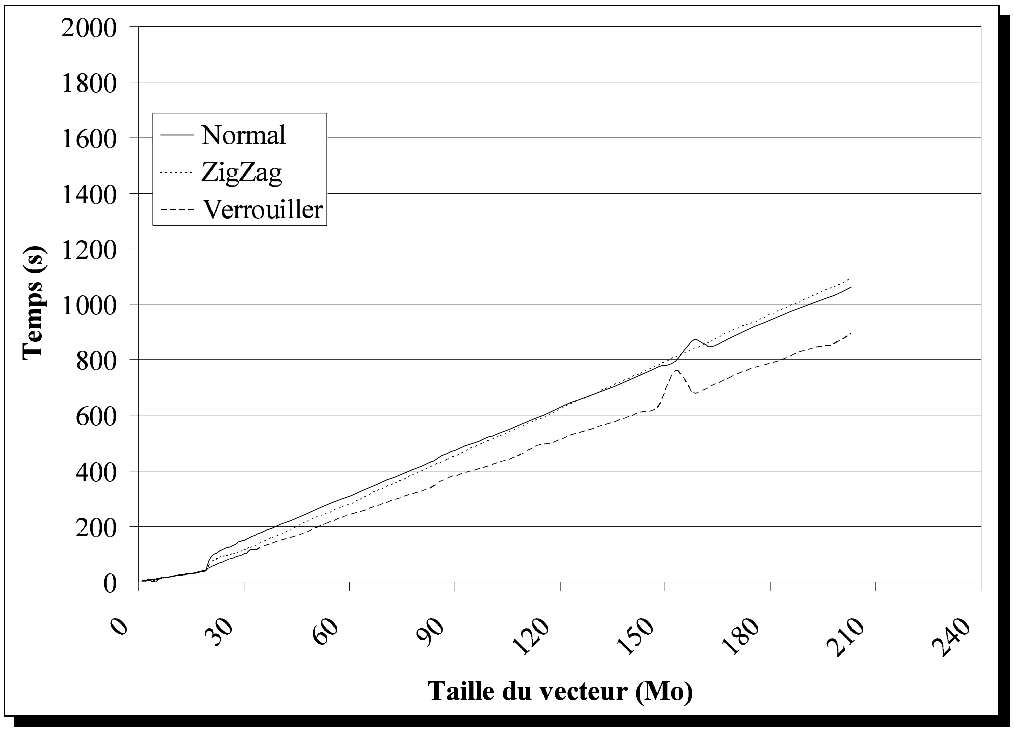


FIG. 1.9 – Lectures successives d'un vecteur avec 32Mo de mémoire

Ces techniques permettent des améliorations de performances qui sont montrées figure 1.9, 1.10 et 1.11. Nous pouvons également remarquer que les paliers correspondants aux moments où le vecteur ne tient plus en mémoire sont grandement diminués. Ceci vient du fait que moins de mémoire est utilisée.

Comparaison des deux méthodes Nous constatons que le verrouillage est plus efficace que le *ZigZag*, or nous avions prévu $m-n$ pages lues/écrites par parcours du vecteur pour le *ZigZag* et $m-n+1$ pour le verrouillage. Ce phénomène s'explique par la rareté de la mémoire pour le verrouillage. En effet, le verrouillage laisse peut de mémoire au système, il doit ainsi exécuter immédiatement les écritures car il

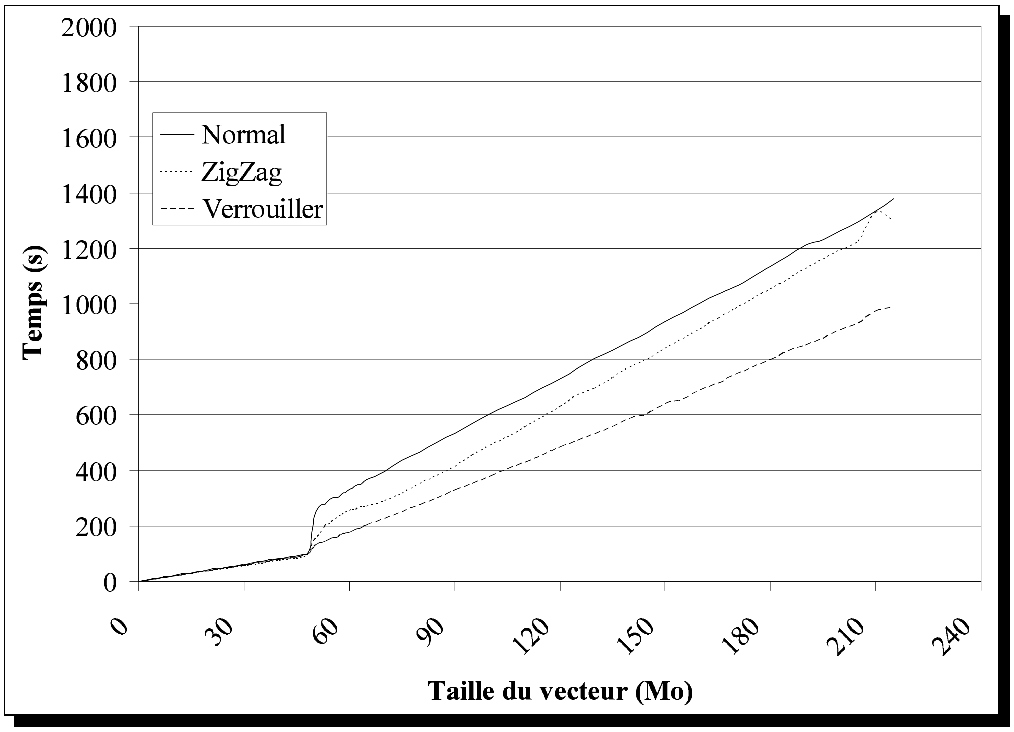


FIG. 1.10 – Lectures successives d'un vecteur avec 64Mo de mémoire

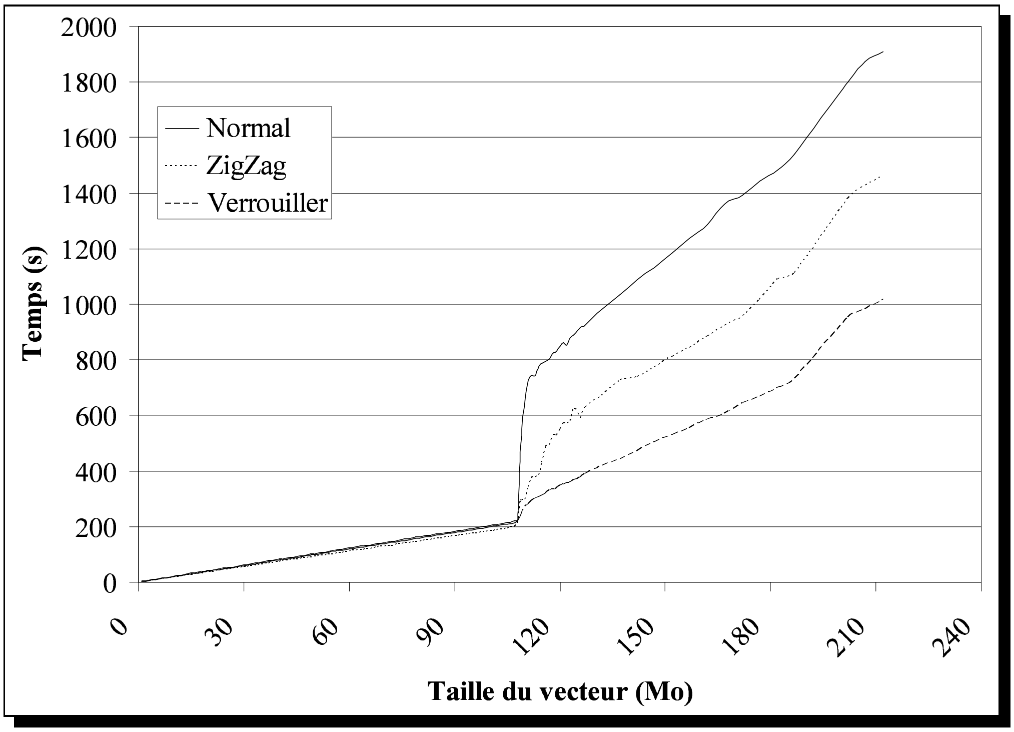


FIG. 1.11 – Lecture successive d'un vecteur avec 128Mo de mémoire

n'a pas assez de mémoire pour en attendre d'autre et ainsi tenter un regroupement. Or, dans le cas du parcours du vecteur, le système ne peut pas espérer de gain en regroupant les requêtes, par contre il perd du temps pendant lequel il aurait pu faire des écritures.

Un effet de ce phénomène survient lorsque la taille du vecteur est importante, les gains venant de la diminution de lectures/écritures de pages deviennent faible, ce qui explique que le *ZigZag* se rapproche de la technique normale. Néanmoins, le verrouillage prend le maximum de la mémoire disponible, il reste ainsi aussi peu de mémoire pour le système qu'on ait 32, 64 ou 128Mo de mémoire au départ. Ceci explique qu'avec de grandes tailles de vecteurs, les temps avec verrouillage sont proches, car leurs stratégies d'écritures des pages sont dictées par la même quantité de mémoire disponible pour le système.

Un problème avec cette technique se produit lorsque l'application n'utilise plus ce vecteur et passe à une autre plage de mémoire. Dans ce cas, le système n'aura que très peu de mémoire pour charger cette nouvelle plage de mémoire. Si cette nouvelle plage mémoire pouvait être contenue dans la mémoire principale, elle pourrait ne plus l'être dégradant ainsi les performances. Pour éviter ce cas où les gains avec ce vecteur sont perdus dans le reste de l'application, il faut qu'à chaque moment, l'application ne verrouille que les pages dont elle a besoin et les déverrouille dès que ces pages ne sont plus importantes. Ceci demande une analyse de l'application en fonction de la mémoire disponible, cette analyse est difficile et revient à faire les entrées/sorties explicite ce qui perd l'intérêt de l'utilisation de la pagination.

Cette technique, comme la précédente, est applicable car le problème est simple et il n'y a pas de dépendance entre les données. Nous verrons par la suite qu'il est bien plus difficile d'appliquer ces techniques avec des cas plus complexes, c'est-à-dire avec des codes correspondants à des applications réelles.

Impact du mapping sur les entrées/sorties

L'emplacement d'une page dans le swap est fixé lors de son premier évincement. Cet emplacement est réservé pour le reste de l'exécution, il est déterminé par l'ordre des demandes d'évincement. Les pages sont évincées dans un ordre indépendant de leurs adresses, il n'y a pas de correspondance directe entre les adresses virtuelles et les emplacements sur le disque, c'est-à-dire que deux adresses virtuelles proches peuvent être éloignées sur le disque. Ce phénomène est amplifié par la pagination d'autres processus qui demande également des emplacements dans le swap.

Considérons une machine avec une unique page mémoire, il sera ainsi indispensable de sauvegarder cette page dans le swap (si elle est modifiée) avant de pouvoir en utiliser une autre. Considérons le produit de convolution de deux vecteurs *X* et *Y* de n éléments, chaque élément remplissant une page. Le programme de convolution est donné à la figure 1.12a.

```
/* Initialisation */
for (i=0;i<n;i++)
    X[i]=0;
for (i=0;i<n;i++)
    Y[i]=1;

/* Calcul de convolution */

d=0;

for (i=0;i<n;i++)
    d=d+X[i]*Y[n-i];
```

a : Programme initial

```
/* Initialisation */

for (i=0;i<n;i++)
    {
    X[i]=0;
    Y[n-i]=1;
    }
```

b : Modification de l'initialisation

FIG. 1.12 – Produit de convolution de deux vecteurs

L'initialisation affectera X[0] et cette valeur sera stockée dans le swap puis X[1] sera affecté et ainsi de suite de même pour Y. Lors de la convolution, les éléments des tableaux sont stockés suivant leurs adresses (voir fig. 1.14). Ainsi, atteindre les deux premiers éléments placés au début et à la fin du fichier de swap, demandera d'aller à deux endroits éloignés du fichier de swap, si nous prenons deux tableaux de 4 pages, nous aurons 7 pages de déplacement. Puis il faudra relire le deuxième élément de X (6 pages de déplacement). Nous avons ainsi $7+6+5+4+3+2+1 = 28$ pages de déplacement pour le produit de convolution. Or, si le fichier de swap est placé linéairement sur le disque, ces déplacements correspondent à des déplacements de la tête du disque, ce qui ralentit fortement l'exécution de l'application (fig. 1.13).

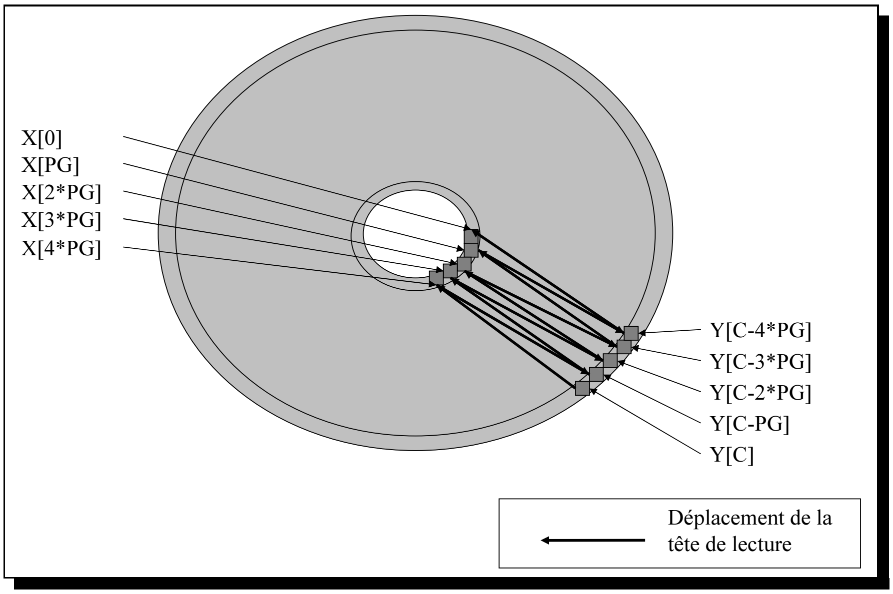


FIG. 1.13 – Déplacement de la tête de lecture

Supposons que nous changions l'ordre d'initialisation des vecteurs de la façon décrite dans la figure 1.12b. Dans ce cas, l'ordre de stockage sera X[0], Y[3] puis X[1] (fig. 1.15). Lors de la convolution, le déplacement dans le fichier de swap ne sera plus de 28, mais de 7.

Nous avons testé ces deux programmes sur un Alpha à 533Mhz doté de 32Mo de mémoire, et d'un disque dur de 8Go et de deux fichiers de swap de 128Mo. Les vecteurs de 40Mo sont suffisamment grands pour ne pas tenir en mémoire, ce qui nous place dans l'effet trou. Nous voyons, figure 1.16, que lorsque la taille des vecteurs augmente, la différence devient plus importante entre les deux initialisations, ce qui met en évidence l'influence de la correspondance sur les performances. L'écart entre les deux courbes évolue avec un palier, qui correspond à l'utilisation du deuxième fichier de swap situé sur un cylindre éloigné du premier fichier de swap. On augmente ainsi les déplacements des têtes de lectures/écritures du disque.

Nous avons vu sur ce cas simple, que la correspondance entre adresse virtuelle et adresse dans le fichier de swap peut influer sur les performances d'une application. Malheureusement, il n'est pas toujours aisé comme sur ce cas simple, de restructurer le code pour améliorer les performances.

1.3.2 Etude de la factorisation LU

Nous venons de voir que la pagination usuelle du système pouvait dégrader les performances d'applications simples. Dans ce paragraphe nous allons montrer que le même type de phénomène se produit sur une opération de calcul scientifique *out-of-core* telle que la factorisation LU d'une matrice.

Considérons le cas d'une factorisation LU sur une matrice d'ordre N. Cette factorisation de matrice

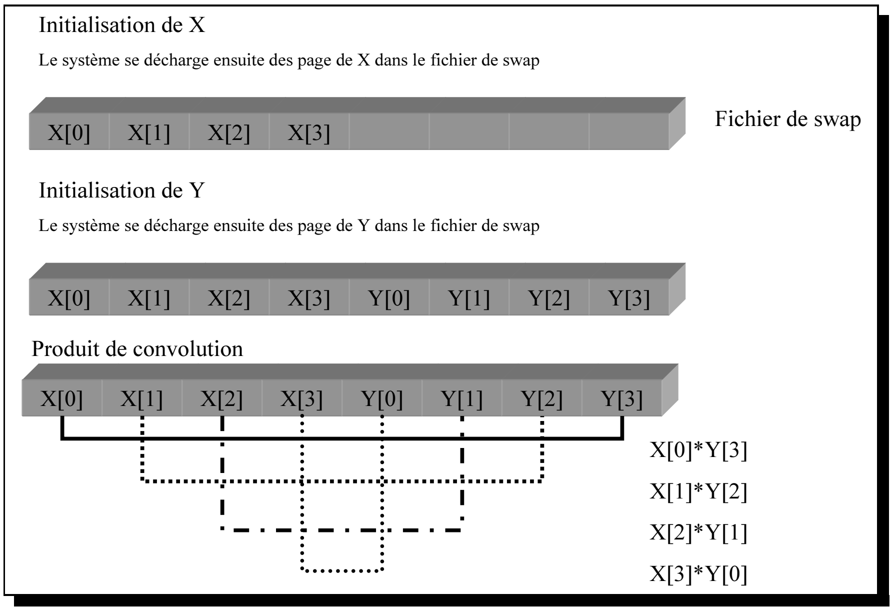


FIG. 1.14 – Produit de convolution avec initialisation usuelle

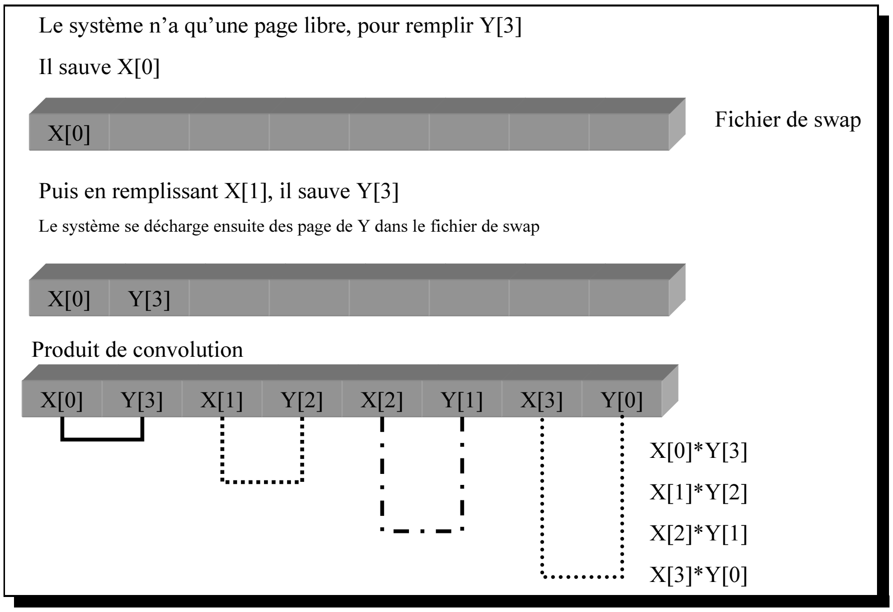


FIG. 1.15 – Produit de convolution avec initialisation modifiée

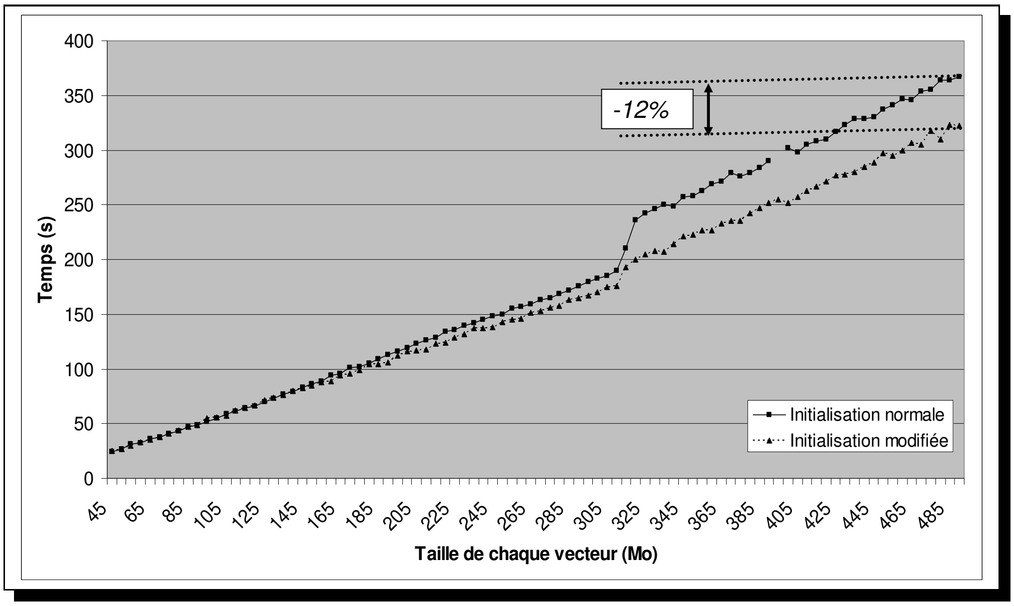


FIG. 1.16 – Produit de convolution avec et sans initialisation modifiée

```
real A(N,N);
for j = 1 to N

   for i = j+1 to N
      for k=1 to j
         A(i,j)=A(i,j)-A(i,k)*A(k,j)
```

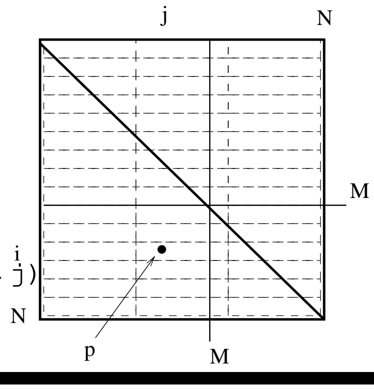


FIG. 1.17 – Factorisation LU. La matrice est stockée par ligne en mémoire. La page p possède les éléments $A(i, j)$. La matrice d'ordre N est plus grande que le nombre M de pages disponibles en mémoire physique.

correspond à la décomposition en deux matrices triangulaire, l'une triangulaire inférieure L et l'autre triangulaire supérieur U. L'algorithme est présenté figure 1.17. Dans ce programme, chaque élément $A(i,j)_{i>j}$ est le produit du vecteur $A(i,1:j)$ par le vecteur $A(1:j,j)$. Estimons le nombre de défauts de page lorsque N est supérieur au nombre de pages physiques en mémoire M. Considérons pour cela l'itération $(i,j)_{i>j}$ et p la page qui contient $A(i,j)$. Puisqu'il y a une dépendance de données entre $A(i,j)$ et $A(i,k)_{k<j}$ et $A(k,j)_{k<j}$, nous montrons que la prochaine utilisation de la page p s'effectuera à l'itération $(i,j+1)$, comme le montre la figure 1.18a. Le nombre de pages accédées avant le prochain accès à la page p, est supérieur ou égal à $N-1 \geq M$. En appliquant la politique LRU, la page p n'est pas disponible suffisamment longtemps en mémoire. Cela se vérifie pour toutes les itérations où $(i,j)_{i>j}$. En fait dans ce cas, on produit un effet trou lors de l'accès à deux éléments contigus sur la même ligne. On a donc au moins $(\frac{N}{2}-1)N$défauts de page.

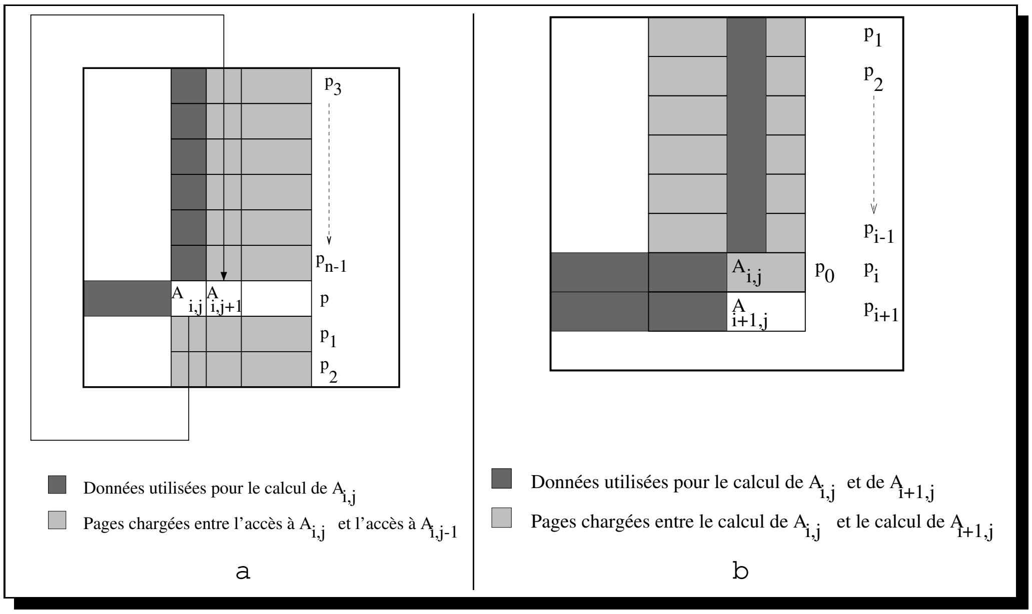


FIG. 1.18 – L'effet trou dans la factorisation LU. Entre deux éléments de deux colonnes consécutives (a) et entre deux éléments consécutifs (b).

A présent considérons les itérations $(i,j)_{i>j}$ dans le cas où $j > M$. On effectue alors $(N-j)$ accès linéaires aux pages j, associés au vecteur $A(1:j,j)$ dans la boucle interne. Le calcul de $A_{(i,j)}$ effectue trop de chargement de pages pour conserver en mémoire les premières pages utilisées pour le calcul de $A_{(i+1,j)}$ (cf figure 1.18b). On se retrouve alors dans *l'effet trou* qui va provoquer j défauts de page. Globalement les deux effets trou impliquent un nombre de défauts de page $p(N,M)$ qui est supérieur à $(\frac{N}{2}-1)N + \sum_{j=M+1}^{j=N} j(N-j)$.

Lorsque M est constant, p(N,M) est en $O(N^3)$. Dans le chapitre suivant nous présentons une nouvelle stratégie de pagination qui permet d'éviter l'écroulement provoqué par *l'effet trou*, c'est-à-dire d'avoir des performances proportionnelles à la quantité de mémoire. A cette fin, nous proposerons, dans le chapitre suivant, une gestion de la pagination adaptée à la factorisation LU et un outil qui permet à l'application d'écrire son propre gestionnaire.

1.4 Conclusion

Nous avons rappelé le fonctionnement de la mémoire. Puis nous avons étudié des exemples nécessitant des données de taille supérieures à la mémoire. Ces exemples ont montrés que la pagination du système Linux fait des choix inadaptés tant au niveau du *mapping* des données sur disques que de la gestion de l'évincement des pages. Ces choix inadaptés provoquent une forte diminution des performances lorsque les données ne tiennent plus en mémoire. Nous pouvons limiter l'effet de ces choix inappropriés par une modification du code dans le cas d'exemples simples. Néanmoins, ces modifications ne peuvent être aisément généralisées à des programmes plus complexes telles que la factorisation LU.

Chapitre 2

Gestion de la mémoire virtuelle au niveau utilisateur

Sommaire

Nous venons de montrer que les politiques de gestion de mémoire virtuelle mises en oeuvre dans les systèmes d'exploitation, ne sont pas adaptées au schéma d'exécution des applications telles que la factorisation LU. Deux approches sont alors envisageables :

La restructuration du code : les performances d'un code *out-of-core* pourront être améliorées si le programmeur peut mettre à profit ses connaissances sur le système de pagination. En effet, le programmeur peut, si cela lui est possible, ordonnancer l'accès aux données afin de diminuer les défauts de page. Une de ces solutions est développée dans la thèse d'Eddy Caron [18]. Une autre solution est de laisser la restructuration à la charge du compilateur comme le décrit Todd Mowry [63].

Modification de la politique de pagination : cette approche est difficile à mettre en oeuvre. En effet, une des rares approches consiste en général à réécrire des parties du noyau du système d'exploitation qui gère la stratégie de gestion mémoire. Dans la suite de ce chapitre, nous présentons un nouvel outil qui permet l'implémentation de gestion de la mémoire virtuelle au niveau utilisateur. A l'aide de cet outil nous pourrons proposer une nouvelle stratégie de gestion de la mémoire virtuelle pour la factorisation LU.

Il y a déjà eu quelques approches de contrôle de la pagination avec des micronoyaux (L3/L4[3], Chorus [1] ou Mach[72]), mais ils ne sont pas aisément transposables sur des systèmes d'exploitation courants tels que *Linux* ou *Windows*.

Nous présentons notre outil *MMUM/MMUSSEL* qui permet cette gestion sous *Linux* en mode utilisateur. Nous décrivons son fonctionnement, puis montrons sur la factorisation LU, décrite dans le chapitre précédent, la mise en œuvre d'une politique permettant la modification de la pagination et ainsi une diminution importante de l'activité de pagination (évincements de page et défauts de page).

Nous montrons que notre outil permet la conception de gestionnaires de pagination complet par la création d'un gestionnaire de pagination destinée aux applications de flux.

2.1 Librairie *MMUM* et le module MMUSSEL

Les politiques usuelles de gestion de mémoire virtuelle ne sont pas toujours adaptées aux besoins. La stratégie de pagination optimale serait de connaître à l'avance l'ordre d'accès des pages et ainsi de calculer à chaque instant quelles pages doivent être évincées et quelles pages doivent être chargées. Néanmoins, cette stratégie est, dans la réalité, irréalisable. Mais en offrant la possibilité à l'utilisateur de pouvoir modifier la stratégie de pagination en fonction de son application, on tend à se rapprocher de la solution optimale puisque spécifique à une application donnée. C'est à partir de ce constat que nous avons développé un nouvel outil de gestion de la mémoire virtuelle au niveau utilisateur.

2.1.1 Description

L'outil de gestion de la mémoire virtuelle est composé d'une librairie appelée MMUM[1] (700 lignes en C), et d'un module noyau appelé MMUSSEL[2][20] (2200 lignes en C). Le module noyau est écrit pour LINUX, mais pourrait être écrit pour n'importe quel Unix utilisant une organisation de la mémoire virtuelle de type MACH[72] (BSD, OSF/1, Hurd,...). *MMUM* est une librairie consacrée au contrôle de la mémoire virtuelle et à l'écriture de gestionnaire de mémoire virtuelle au niveau applicatif. La librairie est combinée avec le module MMUSSEL. Ce module interagit avec le système d'exploitation pour gérer les correspondances entre les pages physiques et les pages virtuelles (fig. 2.3).

Comme nous l'avons dit précédemment, Linux, comme d'autres systèmes d'exploitation, sépare la mémoire en régions auxquelles sont associées des fonctions relatives à la pagination. Lorsque l'on demande à *MMUM/MMUSSEL* de gérer une région, le module Linux *MMUM* détourne la fonction *nopage* qui gère les défauts de page de cette région. Ainsi, lorsqu'un défaut de page se produit sur cette région, *MMUM* le reçoit et le transmet par un signal *SIGSEGV* ou un *callback* à *MMUSSEL* qui avertit le programme. Néanmoins, Linux a besoin d'avoir immédiatement une page pour chaque défaut de page. Pour cela *MMUM* associe une page physique temporaire pour la page virtuelle fautive jusqu'à ce que l'application réagisse ou que *MMUM* reçoive d'autres événements du système.

La première implémentation de *MMUM/MMUSSEL* utilisait un signal SIGSEGV pour communiquer, l'adresse de la page fautive étant mise dans des données statiques. L'inconvénient de cette méthode était la non réentrance : un second défaut de page avant le traitement du premier faisait perdre les informations du premier. Pour limiter ce problème, un module permettant de faire directement (sans passage par des signaux) des callbacks en mode utilisateur a été ajouté par la suite.

Module *callback*

La technique de callback est implémentée dans un module (250 lignes en C). Elle permet d'appeler une fonction en mode utilisateur depuis le noyau. La différence par rapport aux signaux est le passage d'un nombre variable de paramètres et la réentrance : plusieurs appels peuvent être effectués alors que les précédents ne sont pas terminés. Elle utilise la pile et les registres du mode utilisateur, qui sont stockés dans le haut de la pile lorsque l'on est dans le mode noyau. Un callback est effectué par l'appel, en mode noyau, de la fonction *make_callback(handler,param,length)*. Le paramètre *handler* est une fonction réentrante appelée en mode utilisateur, *param* est un pointeur sur les paramètres qui seront envoyés à la fonction utilisateur, le nombre de paramètres est déterminé par *length*.

[1]Memory Management in User Mode.
[2]Memory Management at USer SpacE Level.

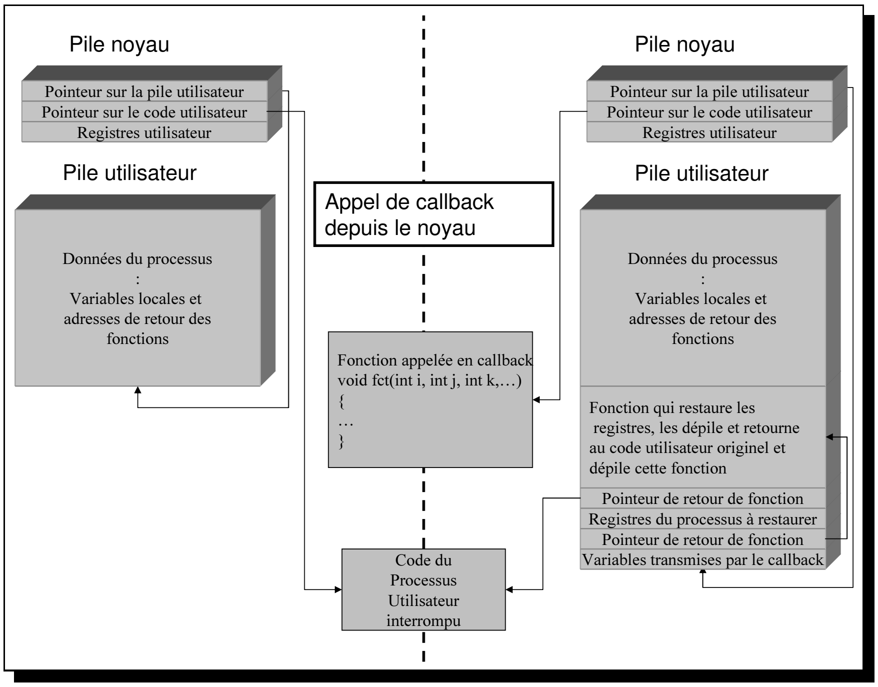


FIG. 2.1 – Callback : modification de la pile système et utilisateur

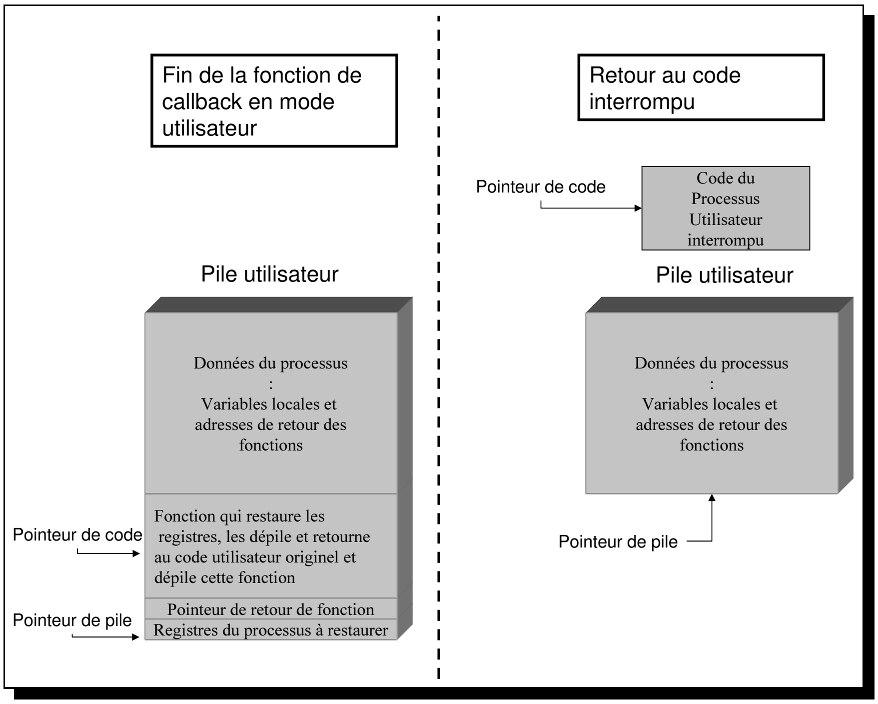


FIG. 2.2 – Callback : retour au code interrompu

La fonction *make_callback()* ajoute à la pile du mode utilisateur le code d'une fonction de retour, les paramètres et une copie des registres utilisateurs (fig. 2.1). Ainsi, lorsque l'on sort du mode noyau, l'exécution continue par la fonction *handler* avec ses paramètres. A la fin de cette fonction, notre fonction est exécutée dans la pile, restaure les registres et redonne le contrôle au code interrompu (fig. 2.2). Cette technique est dépendante de l'*Application Binary Interface* du processeur (*ABI*), mais peut être aisément écrite grâce aux informations des fichiers `arch/TARGET/kernel/signal.c` et la structure `pt_regs` du noyau Linux

Utilisation de *MMUM/MMUSSEL*

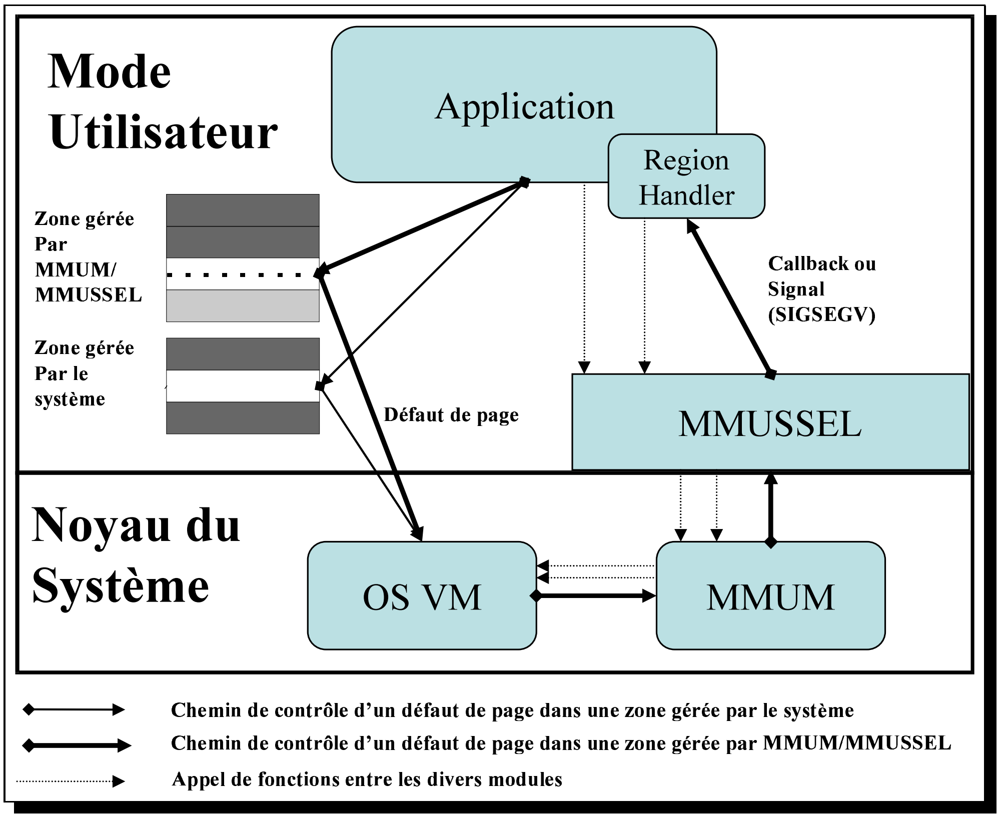


FIG. 2.3 – Schéma général de *MMUM/MMUSSEL*

Nous allons maintenant décrire l'interface de *MMUM/MMUSSEL*. L'application commence par créer une région par l'appel à la fonction `mmum_create()`. La pagination de cette région sera ensuite gérée par l'application.

```
int mmum_create(long size,

        int (*init)(void * data, int région),
        int (*nopage)(MMUSSEL_ZONE région ,long address))
```

Le paramètre `size` est la taille en octets de la nouvelle région. La fonction `init` réalise la création d'une nouvelle région. La fonction `nopage` est appelée à chaque fois qu'un défaut de page est provoqué dans cette région mémoire.

Afin de définir une nouvelle stratégie de gestion de la mémoire virtuelle, il faut dans un premier temps écrire les deux procédures `init` et `nopage`. Plusieurs fonctions de *MMUM* sont conçues pour permettre la gestion de la mémoire physique avec la nouvelle région créée. Nous pouvons parler plus spécifiquement de :

– `mmum_newpage(void *a)` : associe une nouvelle page de la mémoire physique à la page référencée par l'adresse (virtuelle) `a`. Les bits de lecture et d'écriture de la nouvelle page sont positionnés à 0, c'est-à-dire non lue et non modifiée. Une fois la page référencée, elle est verrouillée en mémoire et ne peut pas être évincé par le système.
– `mmum_releasepage(void *a)` : libère la page physique associée à l'adresse (virtuelle) `a`.
– `mmum_xchg(void *a,void *b)` : échange les deux pages physiques associées respectivement aux adresses (virtuelles) `a` et `b` (fig 2.4). Il n'est pas nécessaire d'associer une page physique à l'adresse virtuelle. Si l'adresse `a` est associée à la page physique et qu'il n'y a pas de page associée à l'adresse `b`, après l'appel à `mmum_xchg(a,b)`, l'adresse `b` est associée avec la page précédemment pointée par l'adresse `a`, et aucune page n'est associée à l'adresse `a`.
– `mmum_use_rw(void *a)` : retourne la valeur du bit de lecture/écriture de la page associée à l'adresse (virtuelle) `a` et la remet à zéro.

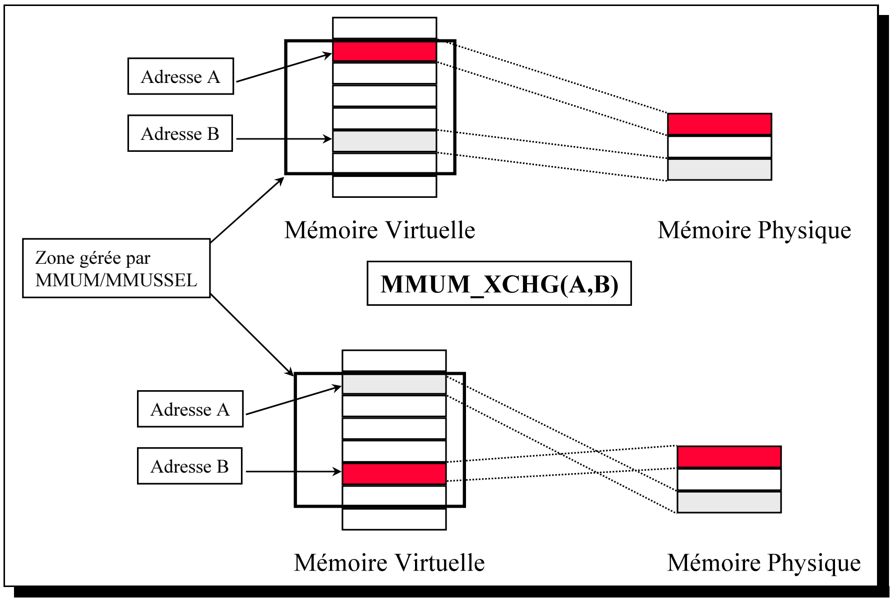


FIG. 2.4 – Echange de deux pages avec mmum_xchg

2.1.2 Projeter un fichier dans la mémoire virtuelle

Pour illustrer les applications potentielles de *MMUM/MMUSSEL*, nous présentons un gestionnaire de mémoire virtuelle qui permet de projeter un fichier dans une nouvelle région mémoire[3]. Le système de gestion de mémoire virtuelle est utilisé par une application qui effectue quelques calculs sur un vecteur de taille N comme le décrit le programme de la figure 2.5. Le contenu du vecteur est en fait le contenu du fichier placé dans une nouvelle région mémoire. La variable `V` est un pointeur sur le début de cette nouvelle région. Les itérations du programme modifient les valeurs du vecteur par la fonction `f`. Les accès aux fichiers sont transparents et sont effectués par le système de gestion de la mémoire virtuelle, c'est-à-dire par la fonction `fm_init` et `fm_nopage`.

[3]Notons que dans les systèmes d'exploitation actuels la fonction `mmap` permet d'effectuer cette opération.

```
void main()  {
double *v;

int i;

   ...
/* Le fichier A est projeté sur le vecteur v */


v = mmum_create(N,fm_init,fm_nopage);

for (i=0;i<N;i++) v[i]=f(v,i);

...

}
```

FIG. 2.5 – Application nécessitant l'accès à un fichier projeté dans une région mémoire.

La figure 2.6 présente le code source des fonctions `fm_init` et `fm_nopage` qui implémente les mécanismes pour projeter le fichier.

Dans cette implémentation, il n'y a qu'une seule page présente en mémoire physique à la fois. Du fait de l'initialisation, cette page est la première page du fichier (fonction `fm_init`). Quand un défaut de page survient, c'est-à-dire qu'une autre page de la région mémoire est accédée, la page courante est écrite sur le disque si elle a été modifiée, et la nouvelle page est chargée en mémoire (fonction `fm_nopage`).

Copie mémoire rapide

Initialement, nous avons créé les fonctions de *MMUM/MMUSSEL* pour contrôler les mécanismes de gestion de la pagination, mais un autre intérêt se trouve dans les fonctions de copies. Ainsi, lors d'un déplacement de données, les données sources ne sont plus utiles et peuvent être remplacées par n'importe quelle autre donnée, cette opération peut ainsi être remplacée par l'échange des pages virtuelles sources et destinations.

Par exemple, si les données de départ ne sont pas réutilisées, `memmove(b,a,PAGE_SIZE)` est équivalent à `mmum_xchg(a,b)` dans le cas ou a et b correspondent au début d'une page ou commencent aux mêmes offsets d'une page. L'intérêt de cette technique réside dans le fait que seuls les deux pointeurs de la table des pages sont modifiés et seuls les quatre pointeurs des répertoires de pages et des répertoires de pages secondaires seront lus. Chacun de ces pointeurs seront chargés dans le cache du processeur. Dans le cas où les lignes de caches ont 64 octets, nous devront lire 64*6 octets et les deux pointeurs modifiés devront être réécrit. Nous obtenons 384 octets lus et 128 écrits, soit 512 octets échangés avec la mémoire. Or, dans le cas d'un processeur x86, une page fait 4096 octets, et la copie correspond au mieux à la lecture de 4096 octets et l'écriture de 4096 octets, soit 8192 octets échangés avec la mémoire. Nous obtenons ainsi un rapport de 16 du point de vue du débit mémoire, ce qui semble intéressant à priori.

Malheureusement, l'utilisation de `mmum_xchg()` requiert le passage en mode noyau suivi de quelques traitements du système avant d'arriver dans MMUM. Ce passage en mode noyau, l'arrivée dans *MMUM* et le retour en mode utilisateur coûtent 300 cycles sur un Pentium II 300Mhz. Le passage en mode noyau est inévitable, mais il peut être accéléré sur les processeurs x86 par l'utilisation de l'instruction `SYSENTER/SYSEXIT` qui limite les vérifications, et réduit à 150 cycles l'appel et le retour à une fonction de *MMUM* (sur le Pentium II 300Mhz). Cette amélioration (400 lignes en assembleur) est accom-

```
void fm_start; / Adresse de la nouvelle région mémoire
*/
void fm_current; / Adresse virtuelle de la page courante */

/* Chargement dans la région */
int fm_size;  /* Taille de la nouvelle région mémoire */
int fm_fd; /* Descripteur de fichier */
void fm_init(void v,int size)
/* La nouvelle région mémoire est initialisée */
{
fm_start=v; /* Enregistre l'adresse de la nouvelle région */
fm_size=size; /* Enregistre la taille de la nouvelle région */

fm_fd=fopen(FILENAME,rw); /* Ouvre le fichier
à projeter dans la région */
/* mémoire  */


mmum_newpage(fm_start); /* Association d'une nouvelle page physique
*/
/* avec le début de la région */
/* Lit la première page de la région */
fread(fm_start,PAGE_SIZE,1,fm_fd);
fm_current=fm_start; /* Enregistre l'adresse virtuelle courante */
/* de la page physique */
}

void fm_nopage(void a) / La page adressée n'est pas présente
en */

/* mémoire physique */
{
/* Vérifie si la page a été modifiée */
/* Si oui, elle est inscrite sur le disque */

if (WRITTEN(mmum_use_rw(fm_current)))
{

fseek(fm_fd,(fm_current-fm_start)/PAGE_SIZE);
fwrite(fm_start,PAGE_SIZE,1,fm_fd);


}

mmum_xchg(fm_current,a);


/* La page physique associée à l'adresse */
/* virtuelle fm_current est à présent */
/* associée à l'adresse virtuelle a */
/* Lit la page correspondant à l'adresse */
/* virtuelle a */
fseek(fm_fd,(a-fm_start)/PAGE_SIZE);
fread(a,PAGE_SIZE,1,fm_fd);
fm_current=a; /* Enregistre l'adresse virtuelle courante */
/* de la page physique  */
}
```

FIG. 2.6 – Fonctions associées à la nouvelle région mémoire pour l'implémentation de la projection de fichier.

Taille de la mémoire	LRU	Optimisée	Efficacité
512	41 194 616	22 761 454	55
768	7 477 734	3 024 878	40
921	1 651 317	281 996	17

TAB. 2.1 – Comparaison entre la factorisation LU version LRU et la version optimisée. Expérimentation sur une matrice d'ordre 1024.

pagné d'une réduction maximale des traitements (moins de 40 instructions assembleur avec la fonction `mmmum_xchg`). Une autre amélioration consiste en la définition de *mmum_xchg_str(a,b,NB_PAGE)* qui permet d'échanger NB_PAGE à la fois, ce qui réduit le nombre de transitions mode noyau/mode utilisateur.

Ces améliorations ayant été implémentées dans *MMUM/MMUSSEL* 2, nous obtenons les résultats expérimentaux suivants pour notre fonction de déplacement de données sur une architecture dont le débit mémoire est limitée à 533Mo/s :

	déplacement de 4Ko	déplacement de 4Mo
memmove	moins de 300Mo/s	moins de 300Mo/s
mmum_xchg_str	600Mo/s	1.2Go/s

2.1.3 Gestionnaire ad-hoc pour la factorisation LU

Nous allons définir une nouvelle stratégie de pagination pour la factorisation LU précédemment introduite dans la section 1.3.2. Nous proposons de modifier dynamiquement la politique de pagination en cours d'exécution. Nous définissons pour cela différentes phases de calcul. La matrice est stockée en ligne, les colonnes sont regroupées par blocs de taille B.

Durant le calcul de tous les éléments d'une colonne de bloc, N pages de la mémoire sont accédées plusieurs fois. Au lieu d'utiliser un algorithme LRU comme politique de gestion de la mémoire virtuelle, nous pouvons utiliser un algorithme LIFO (Last In First Out), avec la page $M-1$ comme première page en mémoire. Ainsi un défaut de page est provoqué pour chaque accès à un élément de $A(i,j)$ lorsque $i = M$, cette stratégie permet d'éviter l'*effet trou* et de réduire le nombre de défauts de page.

À chaque itération i, du fait de la localité spatiale, on a une utilisation étendue des pages associées au vecteur $A(i, 1 : j)$. Lorsque ces pages sont présentes en mémoire physique, c'est-à-dire suite à un défaut de page, le système de gestion de mémoire virtuelle ne les remplace pas. Lorsqu'un bloc de colonnes est atteint, on doit remplacer le bloc de colonnes précédent, et plus précisément ses pages physiques par de nouvelles. Pour cela, le mécanisme de remplacement des pages est changé pour une stratégie LRU. Cela permet d'accéder aux M premiers éléments de la première colonne du nouveau bloc. Une fois ces accès effectués, on reprendra la stratégie LIFO.

Pour implémenter dynamiquement ce changement de stratégie, on réalise un programme de factorisation LU avec plusieurs directives de gestion de la mémoire virtuelle comme le montre la figure 2.7.

Toutes les données dans la sous matrice `A(M :N,1 :N)` sont gérées par une stratégie de remplacement de type LIFO, excepté pour la région définie pour bénéficier de la localité spatiale que nous appellerons *région spatiale*. Toutes les autres données sont accédées suivant la stratégie LRU. L'intérêt de cette solution est de supprimer les défauts de page dans la sous matrice `A(1 :M,1 :N)`. Lorsque l'on expérimente cette stratégie sur un PC sous Linux pour une matrice d'ordre $N = 1024$, les résultats avec différentes valeurs pour M sont présentés dans le tableau 2.1. Plus le rapport M/N est grand, plus l'optimisation proposée est efficace ce qui est conforme à ce que l'on attendait.

- `LRU(A)` : quand un défaut de page est provoqué sur la matrice A, on utilise la stratégie de remplacement des pages LRU.
- `LIFO(A)` : quand un défaut de page est provoqué sur la matrice A, on utilise la stratégie de remplacement des pages LIFO.
- `Spatial(A(i,1 :j))` : les pages qui contiennent le vecteur $A(i, 1 : j)$ ne sont pas remplacées par le système de gestion de la mémoire virtuelle.
- `NoSpatial(A(i,1 :j))` : les pages qui contiennent le vecteur $A(i, 1 : j)$ peuvent être remplacées par le système de gestion de la mémoire virtuelle.

```
real A(N,N);
LRU(A)

for j = 1 to N
  if (j>M) and (j mod B)=0 then
           LRU(A)
    for i = j+1 to N
      Spatial(A(i,1:j))
      for k=1 to j
        if (j\>M) and
           (i=M or k=M) then
                   LIFO(A)
        A(i,j)=A(i,j)-A(i,k)*A(k,j)

     NoSpatial(A(i,1:j))
```

FIG. 2.7 – Factorisation LU avec les directives de pagination.

2.2 Gestionnaire de pagination pour les flux réguliers

Nous venons de décrire l'outil *MMUM/MMUSSEL* qui permet la gestion de la pagination en mode utilisateur et nous l'avons utilisé sur un exemple. Maintenant, nous utilisons cet outil pour concevoir un gestionnaire de pagination générale, c'est-à-dire un gestionnaire indépendant de l'application.

Pour cela, nous nous concentrons sur les applications de flux. Notre gestionnaire cherche à isoler chaque flot de données traité par l'application. Cette approche s'apparente à l'algorithme implémentée par Glass [39], mais cette dernière considère qu'un flot est dans une zone contiguë. Nous prenons en compte les flux qui accèdent à la mémoire par saut, c'est-à-dire avec plusieurs pages situés entre deux pages accédées.

Afin de réaliser ce gestionnaire, nous présentons les algorithmes d'évictions de pages et de préchargement mises en œuvre, puis nous décrivons la mise en œuvre avec *MUMM/MMUSSEL* et enfin, nous présentons les résultats des tests de ce gestionnaire sur des exemples.

2.2.1 Description du gestionnaire

Pour ce gestionnaire de mémoire nous nous plaçons dans le cas du calcul sur des flux. C'est-à-dire la lecture de données régulièrement espacée et non réutilisée. Nous pouvons donc supposer que l'hypothèse de localité temporelle est fausse, c'est-à-dire qu'une page qui vient d'être utilisée ne le sera plus avant longtemps. L'algorithme FIFO pour choisir la page à évincer est donc approprié. Cet algorithme présente aussi les avantages suivants :

- Le temps minimum qui peut différencier l'age de deux pages est proportionnel au nombre de défauts de page, ce qui n'est pas le cas pour l'algorithme LRU[4].
- La fréquence des défauts de page n'influe pas sur l'algorithme, contrairement à l'algorithme LRU [5].

[4] Lorsque l'on parle d'algorithme LRU, on veut dire un algorithme qui sait dans quel intervalle de temps une page a été accédée, et non à quel moment exact elle l'a été.

[5] Si l'on a pas accédé aux pages depuis très longtemps ou si l'on a accédé aux pages entre deux scrutations de l'algorithme, l'algorithme en choisit une à évincer au hasard.

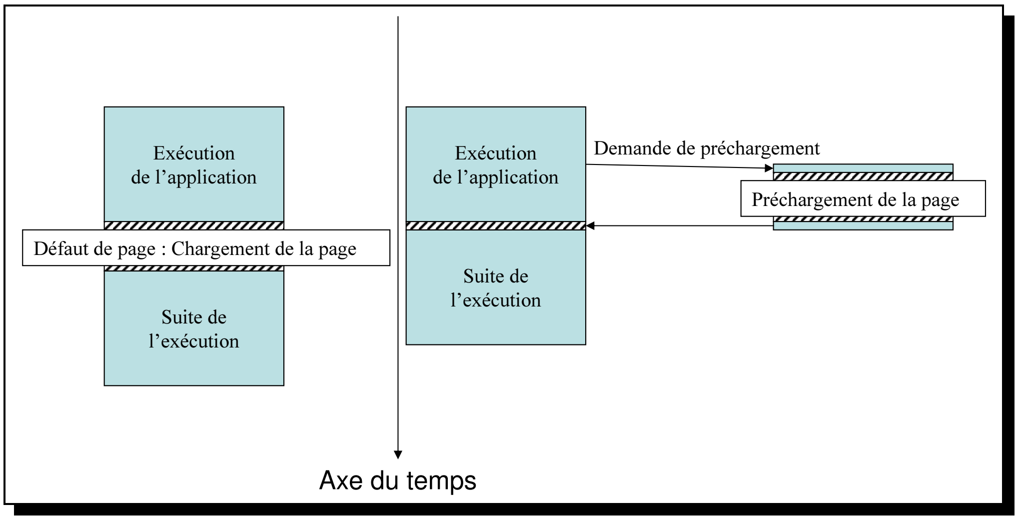


FIG. 2.8 – Recouvrement calcul entrée/sorties par le préchargement

Cette file FIFO correspondra aux pages accessibles par l'application, son *working set* (qui sera noté *WS* dans la suite).

Un moyen d'augmenter l'efficacité de la pagination des flux est le préchargemant des pages qui seront utilisées dans le futur. En effet, comme les données utilisées dans le futur ne sont pas en mémoire, le préchargement peut réduire le temps de chargement lorsque ces données seront nécessaires (fig 2.8). De plus, le préchargement permet potentiellement un réordonnancement des requêtes des entrées/sorties qui peut diminuer leur temps, ainsi une demande des cylindres dans l'ordre 1, 64, 2 et 63 pourra être réordonnancé et devenir 1, 2, 63 et 64. De plus, la prévision des pages qui seront accédées, permet de ne pas retirer de la mémoire une page qui va être accédée.

Ce préchargement implique de connaître à l'avance la liste des pages nécessaires. Nous tenterons d'obtenir cette information par l'analyse des défauts de page.

Description de l'algorithme Le but de cet algorithme est de gérer le préchargement des pages correspondant aux flux identifiés et de collecter les défauts de pages afin d'identifier les flux.

Une application peut traiter plusieurs flux de données avec des vitesses différentes pour chaque flot de données. Dans ce cas, l'algorithme FIFO seul est mal adapté, car le flux le plus rapide engendrera l'évincement des pages des flux moins rapides. Pour résoudre ce problème nous ne retirerons pas tout de suite les pages éjectées de *WS*. Elles seront gardées en mémoire, référencées par une table de double chance (*DC*), mais inaccessibles à l'application.

Les pages préchargées dont nous disposerons seront référencées par une table de prefetch (*PR*), ces pages seront inaccessibles par l'application.

Nous disposons donc de deux tables *DC*, *PR* et d'une file *WS* dont les figures 2.9, 2.10 et 2.11 donnent un exemple.

L'algorithme de prédiction des pages à précharger collectera tous les défauts de page afin d'établir une prédiction. Ces défauts de page viendront des pages non présentes et des pages référencées par *PR* et *DC*. Chacun de ces défauts de page nous permettra d'affiner la prévision des futures pages utilisées :

– Si l'application utilise des pages référencées par *DC*, elles pourront être remises dans *WS* et l'on en gardera une trace. Ceci permettra de distinguer les différent flux de données traitées par l'application, et permettra une prédiction.
– Si l'application utilise des pages référencées par *PR*, elle produira des défauts de page qui permettrons de connaître dans quel ordre ces pages ont été lues, ce qui permettra d'affiner la prédiction

des pages à précharger.
- Si l'application utilise une page absente de la mémoire, on pourra détecter de nouveaux flux de données ou corriger les prédictions.

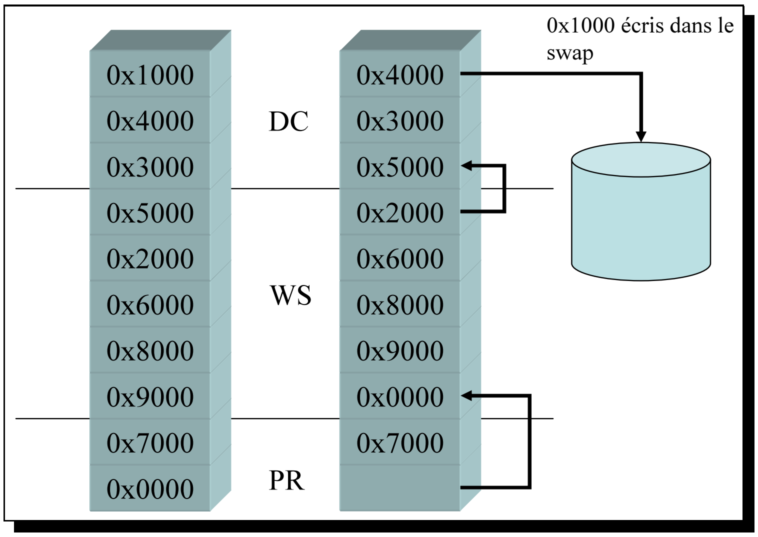


FIG. 2.9 – Pile de l'algorithme : demande de la page 0x000 qui est dans PR

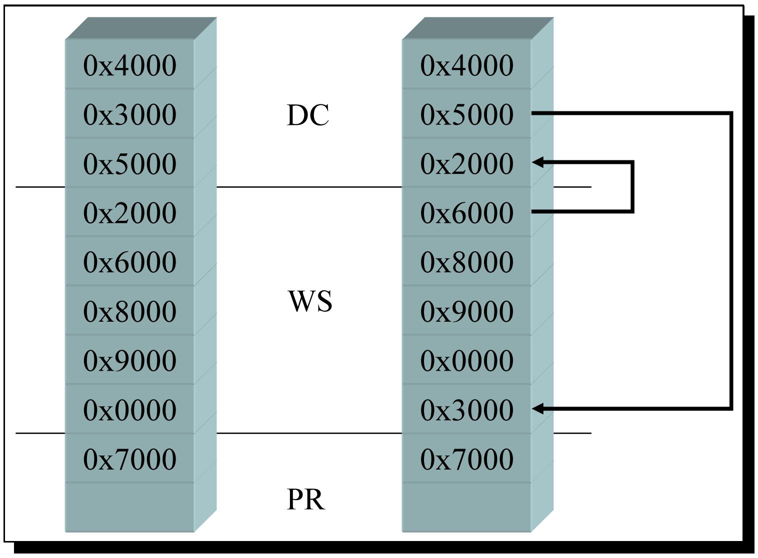


FIG. 2.10 – Demande de la page 0x3000 qui est dans DC

Dans le cas d'un défaut de page :
- Si la page demandée est dans la *PR* alors, le plus ancien élément de *DC* est écrit dans le fichier de swap et est retiré de la mémoire. Les éléments de *WS* descendent d'une place, le dernier va dans *DC*, et la page est insérée dans *WS* (figure 2.9).
- Si la page demandée est dans *DC* alors, cette page est ajoutée à *WS* dont la dernière page va dans *DC* (figure 2.10).
- Si la page n'est pas disponible alors elle est chargée en première place de *WS*, dont la dernière page descend dans *DC*. La plus ancienne page de *DC* est écrite dans le fichier de swap (figure

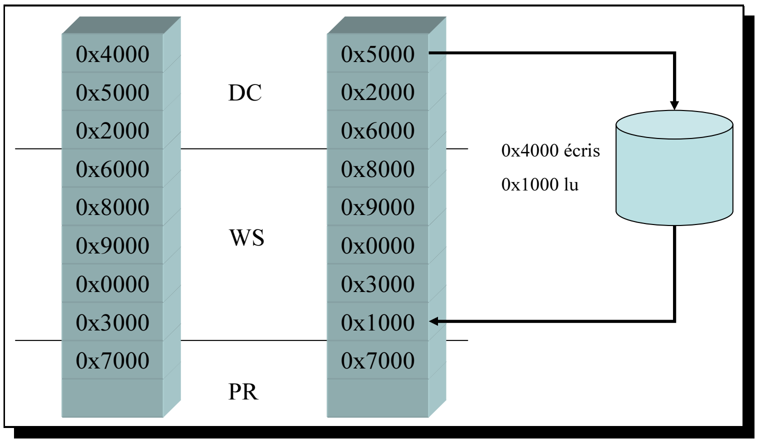


FIG. 2.11 – Demande de la page 0x1000 qui n'est pas en mémoire

2.11).

Afin de réduire l'accès aux disques et de ne pas décharger trop de pages de la mémoire, l'algorithme fait les optimisations suivantes :

- Les pages à évincer sont envoyées dans une file de swap.
- Quand une page est lue par l'application, et qu'elle n'est nulle part en mémoire physique celle-ci est lue sans aucune optimisation possible.
- Quand la file contenant les pages à évincer est pleine :
  - On prévoit les pages à précharger.
  - On lit les pages à précharger et on écrit les pages à évincer (*swapout*) sur le disque dans l'ordre qui permet le plus faible déplacement de la tête de lecture, en regroupant les requêtes au disque.
- Quand une page préchargée n'a pas été utilisée depuis un certain temps on la retire.

Algorithme de préchargement

Pour prévoir les futures pages à charger, on tente d'isoler les différents flux de données traités par l'application. Pour ce faire on extrait des sous-suites de la suite des défauts de page. Chacune de ces sous-suites correspond à un flux de données. En prolongeant ces sous-suites on obtient les pages à précharger, c'est-à-dire les prochaines données des flux.

L'idée est que les défauts de page viennent du traitement de plusieurs flux de données, nous allons donc extraire les sous suites de *trace_pg* qui correspondent à chacun de ces flux de données. Pour cela nous estimons que chacun des flux de données est représentable par le parcours de la mémoire suivant une suite arithmétique. Pour isoler ces flux de données nous allons extraire les sous suites arithmétiques de *trace_pg* qui recouvrent le plus d'éléments de *trace_pg*. Pour connaître le nombre d'éléments couverts par une suite A dans une suite B, on procède ainsi :

1. Commencer avec le dernier élément des vecteurs A et B
2. Répéter
   (a) Si l'élément actuel de A est égal à l'élément actuel de B alors
      i. Marquer cet élément de B comme dans A
      ii. Passer à l'élément suivant de A
   (b) Passer à l'élément suivant de B

Exemple

- 1234 recouvre 4 éléments de 10020300004.

– 3214567 recouvre 3 éléments de 123567 qui sont 567.
– 123456 recouvre 4 éléments de 345263 qui sont 456.
– 123456 recouvre 6 éléments de 1234546.

Pour ce qui est de l'extraction des sous suites on procède ainsi le tableau trace_pg contient l'adresse des derniers défauts de page.

1. Marquer tous les éléments de *trace_pg* inutilisés.
2. Si il existe un élément k de *trace_pg* non encore utilisé alors
   (a) Ml=1
   (b) MP=0
3. Pour tous les éléments l de *trace_pg*
   (a) Prendre comme suite arithmétique celle comprenant *trace_pg(k)* et de raison *trace_pg(k)-trace_pg(l)*
   (b) Soit P le nombre d'éléments couverts par cette suite
   (c) Si P>MP alors MP=P et Ml=l
      i. Si Ml différent de 1 alors
         A. Marquer tous les éléments couverts par la suite arithmétique comprenant *trace_pg(k)* et de raison *trace_pg(k)-trace_pg(Ml)* utilisés.
         B. Ajouter cette suite à la liste des suites extraites

2.2.2 Mise en oeuvre avec *MMUM/MMUSSEL*

L'algorithme de pagination décrit précédemment a été mis en oeuvre sous Linux. Pour cela nous avons utilisé les primitives de *MMUM/MMUSSEL* et nous avons implémenté une bibliothèque qui se substitue à certaines fonction de la `libc` (principalement *malloc()* et *free()*).

Ainsi, pour utiliser notre gestionnaire, il est inutile de recompiler les programmes, il suffit de taper la ligne suivante avant l'exécution du programme : export LD_PRELOAD=/gswap/libalgo/libalgo.so.1.0, dans le cas où cette ligne est dans les scripts de démarrage d'un utilisateur ou du système : tous les programmes utilisent notre gestionnaire. LD_PRELOAD permet de précharger notre librairie avant la libc. Notre librairie dispose d'une fonction malloc() et, de ce fait, ce sera notre fonction malloc qui sera exécutée à chaque appel d'un malloc. Si la mémoire à allouer est supérieure à 1Mo, la mémoire est allouée dans une région gérée par l'algorithme. Dans le cas contraire la mémoire est allouée avec la véritable fonction malloc() de la libc, appelée __libc_malloc(). Il faut remarquer que nous avons choisi de remplacer la fonction malloc() et qu'ainsi, seules les données placées dans le tas seront gérées par notre bibliothèque. Néanmoins, il aurait été possible de gérer l'ensemble de la mémoire du processus : une version plus évoluée de *MMUM/MMUSSEL* le permet de manière plus aisée.

Récapitulatif des structures de données

Pour gérer les tables *DC*, *PR* et la file *WS* associée à la mémoire, il y a trois sortes de tableaux :
– Gestion de la table *DC* et de la file contenant les pages à swapper :
  – *to_swap* : contient les adresses des pages à swapper, ou des pages qui se trouvent dans la région de deuxième chance *DC*. C'est-à-dire qu'elles sont accessibles, mais pas à leur adresse virtuelle. Cette région est située après les données de l'application.
  – *age_toswap* : âge de chacun des éléments de *to_swap*. Si cet âge est trop élevé alors l'élément n'est plus dans *DC* mais dans la liste des pages à swaper[6].

[6]L'âge de tous les éléments est incrémenté à chaque défaut de page.

- Gestion des pages de *WS* : contient les adresses des pages en mémoire qui se trouvent dans la région accessible par l'application. Ce tableau représente une file, c'est-à-dire que le premier élément de ce tableau est l'élément qui va passer dans *to_swap* (double chance) et que le dernier est celui qui vient de provoquer un défaut de page.
- Gestion des pages de *PR* et des pages libres :
  - *free_page* : contient les adresses des pages libres. Lorsque que l'on précharge une page virtuelle, elle est placée dans les pages libres (*free_ page*), avec un âge nul.
  - *free_age* : donne l'âge de dernière modification de chaque page libre.

Exemple de fonctionnement

Supposons (fig. 2.9 2.10 2.11) que la mémoire virtuelle de l'application comprenne quatre pages virtuelles, que WS puisse contenir une seule page qui est la page 0. Supposons que DC est vide, et que PR contient la page 2. Dans ce cas, la page 0 est accessible par son adresse virtuelle, c'est-à-dire qu'un accès à l'adresse virtuelle 0 renverra le contenu de la page 0. La page 2 n'est pas accessible par son adresse virtuelle, mais son contenu est stocké dans le tampon situé dans un autre endroit de l'espace virtuel[7]. Si l'on tente d'accéder à la page virtuelle 2 par son adresse, on déclenche une interruption. Pendant cette interruption, la page virtuelle 2 va être déplacée dans WS, son adresse virtuelle pointera sur son contenu[8]. Puis la page virtuelle 0 sera exclue de WS pour aller dans PR. L'adresse de la page virtuelle 0 ne pointera pas sur son contenu[9]. Le contenu de la page virtuelle 0 est accessible par la page virtuelle 5.

2.2.3 Tests du gestionnaire

Condition de l'expérimentation

L'algorithme a été expérimenté sur un PC Cyrix 6x86 200Mhz doté de 64Mo de mémoire SDRAM, et sur un PC Pentium 200Mhz doté de 48Mo de mémoire EDO. On se place dans le cas suivant :
- notre gestionnaire de pagination utilise un fichier standard, alors que pour le système on utilise une partition de swap ;
- les données sont initialisées, mais ce temps n'est pas pris en compte ;
- les données utilisées ont toujours une taille totale de 120Mo, car ceci correspond à une seule partition de swap de Linux ;
- lors des tests, il n'y a aucune autre application importante qui s'exécute ;
- les tests sont exécutés à partir d'un terminal, X Window n'est pas lancé.

Produit de convolution

Ce test consiste à calculer le produit de convolution de deux vecteurs ayant chacun une taille comprise entre 1Mo et 60Mo chacun. Nous utilisons le programme de la section 1.3.1.0.0, mais, grâce à notre gestionnaire, la correspondance entre adresse virtuelle et adresse dans le fichier de swap est prédéterminée : c'est la même à une translation près.

Sur la figure 2.12, le gain de performance observé est plus important avec moins de mémoire. Ceci vient du fait qu'après avoir initialisé les deux vecteurs, une partie de ces vecteurs est conservée en mémoire par Linux. Or cette partie correspond au début des données qui seront utilisées par le produit de convolution.

[7]En l'occurrence l'adresse virtuelle de la page 4.
[8]Dans la pratique ce changement de pointeur se fera par un xchg(0x400000,0x200000), 0x400000 désignant la page 4 et 0x200000 la page 2.
[9]xchg(0x000000,0x500000).

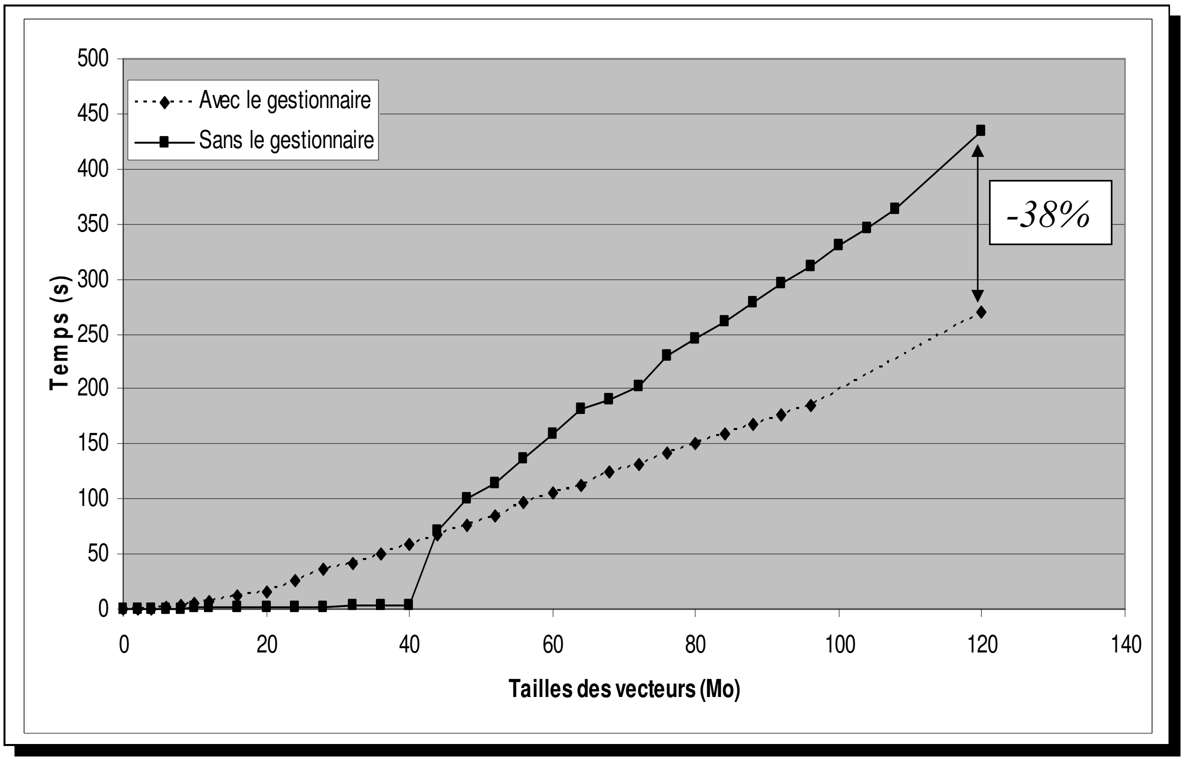


FIG. 2.12 – Produit scalaire de deux vecteurs.

Dans cet exemple, l'algorithme arrive très rapidement à extraire deux suites, une croissante et une décroissante qui correspondent aux balayages des deux vecteurs. L'algorithme précharge les pages futures de ces deux suites en les groupant par 16 pages contiguës à lire.

Filtre

Ce test consiste à adoucir une image. Chaque point prend la valeur de la moyenne de son voisin de gauche et du dessus. Le test a été fait sur une matrice de 10000*10000 entiers courts. Les résultats sont dans le tableau 2.13.

```
void filtre(char [n][n] A);
/* Filtre d'une matrice A */
int x,y;
{
for (x=1;x<=n;x++)
   for (y=1;y<=n;y++)
      A[x][y]=(A[x-1][y]+A[x][y-1]+A[x][y])/3;
}
```

Les gains sont limité par le fait qu'une ligne de la matrice *A* est contenue dans trois pages mémoire. En effet, le calcul d'une nouvelle ligne nécessite les données de la ligne précédente et de la ligne courante, la ligne précédente étant encore en mémoire, il suffit de charger la ligne courante, soit trois pages. Ce qui explique que l'hypothèse que le nombre de données à charger soit limité. Néanmoins, la matrice représente plus que la taille de la mémoire, ce qui nous place dans l'*effet trou* et notre algorithme diminue l'impact de cet effet par un préchargement et un déchargement agressif des données de la matrice.

	Temps pour le filtre d'une image, PC avec 64Mo
Sans gestionnaire	4Min 0s
Avec gestionnaire	2Min 30s

FIG. 2.13 – Utilisation de nôtre gestionnaire avec un Filtre

2.3 Conclusion

Après avoir présenté la mémoire virtuelle et sa gestion dans les systèmes d'exploitation courants, nous avons montré les limites de certaines implémentations. Pour pouvoir dépasser ces limites nous avons créé un module permettant de contrôler la pagination en mode utilisateur. Des techniques semblables avaient déjà été envisagées du point de vue théorique par Keith Krueger [54] et ont été développées sur des systèmes spécifiques tels que Grasshopper [3] où chaque portion de la mémoire et chaque fichier est considéré comme un objet dont les applications peuvent redéfinir les méthodes.

L'apport de MUMM/MMUSSEL est de donner un module qui fonctionne sous un noyau non modifié de Linux. D'autres systèmes tels que *Maîs* [17] ou *RDSMS* [38] permettent de partager virtuellement la mémoire afin de donner l'illusion d'un multiprocesseur à partir d'une grappe de machines tel que le fait *Mosix* [10] [8]. *MMUM/MMUSSEL* peut servir de base pour un tel système et permettre ainsi la conception d'une mémoire virtuellement partagée (*DSM*) sans avoir à modifier le système. Néanmoins, une gestion de la mémoire partagée avec une cohérence séquentielle donne des faibles performances. Ainsi, une perspective serait l'utilisation de *MMUM/MMUSSEL* pour l'implantation de mémoire virtuellement partagée avec une consistance relâchée [38] pour atteindre des performances acceptables ou l'utilisation intensive de préchargement avec une vision globale telle que *PGMS* [79].

Chapitre 3

Moniteur de pagination

Sommaire

Nous venons de voir l'outil *MMUM/MMUSSEL* qui permet de contrôler la pagination en mode utilisateur, la question est maintenant comment utiliser au mieux cet outil pour faire une pagination adaptée à une application. Pour cela nous avons deux possibilités :

- Une première possibilité consiste à décrire une politique de pagination générale, adapté à une classe d'applications et sans communication avec l'application. C'est l'option prise pour le gestionnaire de flux par Glass [39] et dans une version plus large par notre gestionnaire de flux du chapitre précédent.
- Une deuxième possibilité consiste à décrire une technique d'interaction entre l'application et la pagination. Cette option a été prise par Mowry[63], l'interaction avec l'application se faisant par l'indication par le compilateur des données qui vont être utiles et des données qui ne le seront pas ou plus.

Nous choisirons la deuxième solution. Nous l'appliquerons sur l'application réelle qu'est le solveur *MUMPS*. A cette fin, nous analyserons la pagination de cette application avec un outil basé sur une seconde version de *MMUM/MMUSSEL*. Nous en déduirons une première approche du comportement mémoire de *MUMPS*. Nous définirons une méthode pour améliorer la pagination de *MUMPS* et nous validerons cette méthode par une présentation des gains obtenus.

3.1 Contrôleurs de pagination

Nous avons montré l'outil *MMUM/MMUSSEL* dans le chapitre précédent, cet outil à évolué en devenant indépendant du processus qui l'utilise. Nous présentons cette évolution, puis nous montrons son intérêt sur l'exemple de la collecte de l'activité de la pagination sans modification de l'application.

3.1.1 MMUM/MMUSSEL 2

Le principal problème de *MMUM/MMUSSEL* est la limitation de l'interaction avec le système d'exploitation. En effet, *MMUM/MMUSSEL* se contente d'intercepter les défauts de page et de verrouiller les pages utilisées en mémoire, le nombre de pages maximum qui peuvent être verrouillées étant déterminés au chargement du module. Ainsi, dans le cas où le système a besoin de plus de mémoire pour d'autres programmes, il ne pourra pas en obtenir de *MMUM/MMUSSEL* ce qui peut ralentir ou bloquer le système. De la même manière, si le système a besoin de moins de mémoire *MMUM/MMUSSEL* ne pourra l'utiliser. La solution adoptée considère que le système évince les pages les moins récemment utilisées et qu'il doit toujours pouvoir évincer une page de la mémoire s'il en a un besoin urgent.

Pour satisfaire cette dernière condition nous utiliserons deux processus, le client (géré par *MMUM/MMUSSEL*) et le moniteur. Ce dernier est un processus standard avec sa pagination gérée par le système. Lorsqu'un défaut de page se produit dans le client, le moniteur reçoit un signal et doit charger la page concernée et la déplacer dans l'espace du client. Lorsque le système veut évincer une page du client, il la déplace dans le moniteur qui doit la sauvegarder sur le disque, cette page sera référencée comme une page récemment utilisée, afin que le système ne l'évince pas du moniteur. Le moniteur influence l'ordre dans lequel seront évincées les pages du client, soit par l'évincement explicite, soit par le changement de l'âge de dernière utilisation des pages (car le système évince en premier les pages les plus âgées). Néanmoins, le moniteur étant un processus standard, si le système a un besoin urgent de mémoire, le moniteur sera paginé avec les pages qui lui viennent du client.

Les nouvelles fonctions de *MMUM/MMUSSEL* sont :

mmum_attach(pid,deb,fin,message()) : cette fonction permet de récupérer la pagination du processus *pid* dans la zone comprise entre *deb* et *fin*. C'est-à-dire qu'à chaque défaut de page dans cette zone, le système appelle la fonction *message()*, qui devra lire la file de message, charger la page et redémarrer le processus. Cette fonction est également appelée lors de la réception d'un *swapout*.

mmum_xchg_monitor(pid1,addr1,pid2,addr2) : cette fonction permet de retirer une page virtuelle de la mémoire virtuelle de pid1 et de la mettre à l'adresse virtuelle addr2 dans pid2, cette fonction se fait par échange de pointeurs entre les tables de translation d'adresse des deux processus. Les utilisations de cette fonction sont l'évincement explicite d'une page d'un client vers le moniteur, l'ajout dans le client d'une page qui vient d'être demandée (défauts de page). Cette fonction est utilisée par le système pour évincer une page d'un client vers un moniteur.

mmum_restart(pid) : cette fonction redémarre un client qui s'est arrêté suite à un défaut de page. Il est nécessaire d'avoir placé la page demandée dans le client avant de le redémarrer.

mmum_userw_monitor(pid,addr) : cette fonction permet de savoir si une page a été modifiée ou accédée depuis le dernier appel à cette fonction.

mmum_touch_monitor(pid,addr) : cette fonction permet de modifier l'âge de dernier accès à une page. Sachant que le système évince les pages les moins récemment utilisées, cette fonction permet de forcer les pages les plus importantes à rester en mémoire (si possible).

mmum_release_monitor(pid,addr) : cette fonction libère la page physique associée à la page virtuelle sans la stocker sur le disque.

Comme avec la version précédente de *MMUM/MMUSSEL*, le moniteur reçoit les informations du système par callback ou par signaux, néanmoins lorsque le moniteur reçoit des pages évincées du client, leur nombre peut être important et ne peut être passé en simple paramètre. Pour cette raison, une région de mémoire du moniteur est verrouillée et est utilisée comme file de réception des messages du système. Ces messages sont de deux types différents :

Défaut_de_page(pid,addr) : signale qu'un client pid est suspendu suite à un défaut de page à l'adresse addr. Dans ce cas, le moniteur doit charger la page demandée, la transférer dans l'espace virtuel du processus et redémarrer le processus.

Swapout(pid,addr,addr_monitor) : signale que le système a évincé la page virtuelle addr du client pid et l'a transférée à l'adresse addr_monitor du moniteur. Le moniteur devra sauvegarder cette page avant de libérer son emplacement par *mmum_release_monitor(getpid(),addr_monitor)*. Il faut noter que si le moniteur ne libère pas suffisamment rapidement les pages, le système peut décider de paginer le moniteur, ce qui fait perdre l'intérêt de *MMUM/MMUSSEL*, mais ne bloque pas le système.

Un premier exemple de l'utilisation de *MMUM/MMUSSEL* 2 est décrit dans la figure 3.2 pour le moniteur et la figure 3.3 pour le client qui lance le moniteur. Dans cet exemple, un programme, que nous nommerons client, lance un moniteur qui prendra en charge la gestion d'un fichier projeté en mémoire. Le moniteur commence par demander le contrôle de la pagination du client par *mmum_attach()*, et attend les messages de pagination du client dans la zone comprenant *a[]*. Lorsque le client accède à une page non présente du tableau, il est arrêté et un message est envoyé au moniteur. Celui-ci charge la page en mémoire et la transfère dans l'espace virtuel du client (avec *mmum_xchg()*). Lorsqu'une page est évincée par le système (swapout), il la transfère dans l'espace virtuel du moniteur, qui l'écrit sur le disque et la retire de son espace virtuel (*mmum_release()*.

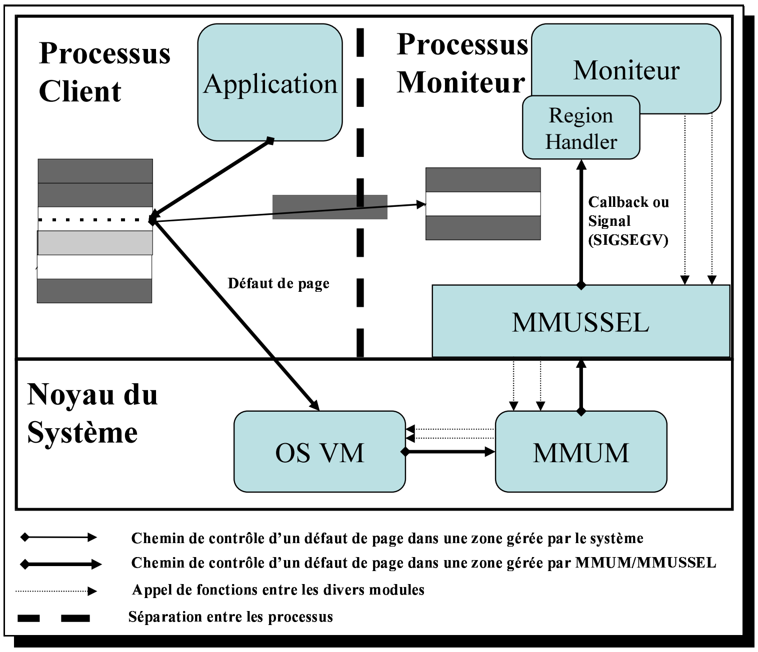


FIG. 3.1 – Interaction entre le moniteur, *MMUM/MMUSSEL* et le client

Cette version de *MMUM/MMUSSEL* implémentée avec un noyau 2.4 de Linux a été plus difficile a implémenter qu'avec le 2.2, car les opérations associées aux zones mémoires ont été réduites depuis les versions 2.2 et 2.0. Il n'y a ainsi plus de fonction swapout associée à une page, ce qui nécessite de créer un périphérique de stockage secondaire pour récupérer les swapout. Pour pouvoir transférer *MMUM* sur un noyau 2.4, nous avons du commencer par l'implémenter sous forme d'une partie du noyau, avant de créer un pilote de périphérique d'échanges qui a permis de le transférer en module.

Nous pouvons noter que nous gardons les fonctionnalités de la première version, ce qui permet de faire fonctionner les anciens programmes. Cette nouvelle implémentation a ensuite été utilisée pour gérer les défauts de page d'une application et ainsi essayer d'améliorer l'application nommée *MUMPS*.

```
#include <stdio.h>
#include <mmussel2.h>

int fd ;
long address_begin;

void message(
  int swno,
  int pid,
  long address,
  long local_address)
{
if (swno==NOPAGE)
  {
  char * t=malloc_aligned(PAGE_SIZE);
// malloc_aligned donne une adresse
// PAGE_SIZE alignée
  llseek(fd,address-address begin,
      SEEK_SET);
  read(fd,t,PAGE_SIZE);
  mmum_xchg(getpid(),t,pid,a);
  free(t);
  }
if (swno==SWAPOUT)
{
// Quand Linux/MMUM évince une page,
// elle est transféré dans cet espace virtue
// à l'adresse local_address
llseek(fd,address-address_begin,SEEK_SET);
write(fd,local_address,PAGE_SIZE);
mmum_release(getpid(),local_address);
}
}

int main(int argc, char ** argv)
{
fopen("/tmp/mappedfile",O_RDWR);
int pid;
if (argc<2)
 printf("Utilisation :\
  ./monitor pid_of_monitored_process\
  address_of_the_mapped_file\n");
fd=fopen("/tmp/mappedfile",O_RDWR);
pid=atoll(argc[1]);
address_begin=atoll(argv[2]);

mmum_attach(pid,address_begin,address_begin

    + 400*PAGE_SIZE,pf,swo);
//Signale que le moniteur est demaré
kill(pid,SIGUSR1);


while (1) sleep(100);
}
```

FIG. 3.2 – Un exemple de projection en mémoire d'un fichier dans un autre processus : le moniteur

```
#include <stdio.h>

main()
{
char a=malloc_aligned(400*PAGE_SIZE);
char pid[10];


long w;
char address[20];
int status;
sprintf(pid,"%d",getpid());

sprintf(address,"%ld",(long)a);
if (fork()==0)

  execlp("./monitor",
     pid,address);


//Attende du démarrage du moniteur

wait(&status) ;



for (w=0;w<400*PAGE_SIZE;w++)
  {
  if (a[w]<10) a[w]++;
// a[] is the mapped file
}
}
```

FIG. 3.3 – Un exemple de projection en mémoire d'un fichier dans un autre processus : le client

3.1.2 Un exemple : un analyseur de pagination

Le premier programme utilisant *MMUM/MMUSSEL* 2 est un outil de récolte et d'analyse de la pagination. Il ne nécessite pas de recompiler ou de modifier l'application.

En effet, l'architecture plus élaborée de *MMUM/MMUSSEL* 2 transfère la gestion de la pagination l'application concernée vers un processus externe, tout en conservant la pagination du système. Ces deux propriétés permettent de concevoir un programme qui se chargera uniquement de collecter les indices des pages chargées et déchargées.

Ce programme de collecte, nommé *anatrace*, utilise peu de mémoire (6Mo), ce qui lui permet de peu influer sur le programme analysé. Nous avons collecté les traces de deux programmes simples qui remplissent un vecteur (voir fig. 3.8). La première représentation de ces traces est le nombre de défauts de page par seconde, ce qui nous permet de déterminer quels sont les moments critiques du programme du point de vue de la pagination et ainsi de connaître les parties du programme qui doivent être modifiées. Une seconde question importante est de savoir quelle serait l'influence de l'augmentation de la mémoire sur le temps d'exécution du programme ainsi que l'espace utilisé à chaque instant (appelé *Working Set observé ou WS*).

Pour cette seconde question, nous avons estimé le temps nécessaire pour lire une page depuis le disque afin d'estimer le temps d'un défauts de page. Nous avons ensuite calculé les pages qui seraient restées en mémoire si cette mémoire avait été plus importante (voir fig 3.4 et 3.5).

LISTE_MEMOIRE : liste contenant les adresses de pages en mémoire lors de l'exécution du programme

TABLEAU_SWAP : tableau contenant les adresses des autres pages évincées, l'adresse la plus faible correspond à la page évincée le plus récemment

TEMPS : temps estimé

TOT_MEM : mémoire totale simulé

1. Faire
   (a) EVENEMENT <- lire_trace // lit l événement suivant du fichier de trace c'est-à-dire l'évincement d'une page ou le défaut de page suivant
   (b) Si EVENEMENT.TYPE=evincer_une_page alors
      i. Retirer(EVENEMENT.ADRESSE,LISTE_MEMOIRE)
      ii. Pour I faire
         A. TABLEAU_ETENDU[I+1]=TABLEAU_ETENDU[I]
      iii. TABLEAU_ETENDUE[0]=EVENEMENT.ADRESSE
      iv. TEMPS=TEMPS+temps_ecriture
   (c) Si EVENEMENT.TYPE=defaut_de_page alors
      i. Si offset(EVENEMENT.ADRESSE,TABLEAU_ETENDU)>TOT_MEM alors
         A. TEMPS=TEMPS+temps_lecture
         B. Retirer(Evénement.adresse,TABLEAU_ETENDU)
   (d) Ajouter(EVENEMENT.ADRESSE,LISTE_MEMOIRE)
2. Tant que il_reste_des_evenements

FIG. 3.4 – Algorithme d'estimation des défauts de page suivant la taille de la mémoire

Les deux exemples étudiés sont une succession de multiplications d'un vecteur de 120Mo par un scalaire et un produit de convolution de deux vecteurs de 100Mo.

Le résultat est présenté sur les figures 3.6 et 3.7. Nous pouvons immédiatement déduire sur ces exemples quelle est la taille de mémoire minimum pour les performances optimales ainsi que l'espace de travail utilisé. Ainsi, la taille optimale pour le produit de convolution (fig 3.6) est légèrement supérieure à 200Mo, ce qui correspond à la taille des deux vecteurs. Nous remarquons également que l'espace de

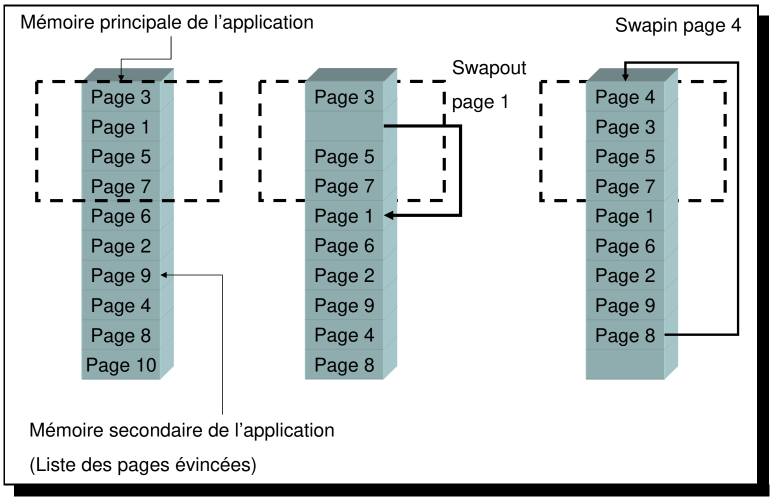


FIG. 3.5 – Exemple de fonctionnement de l'algorithme

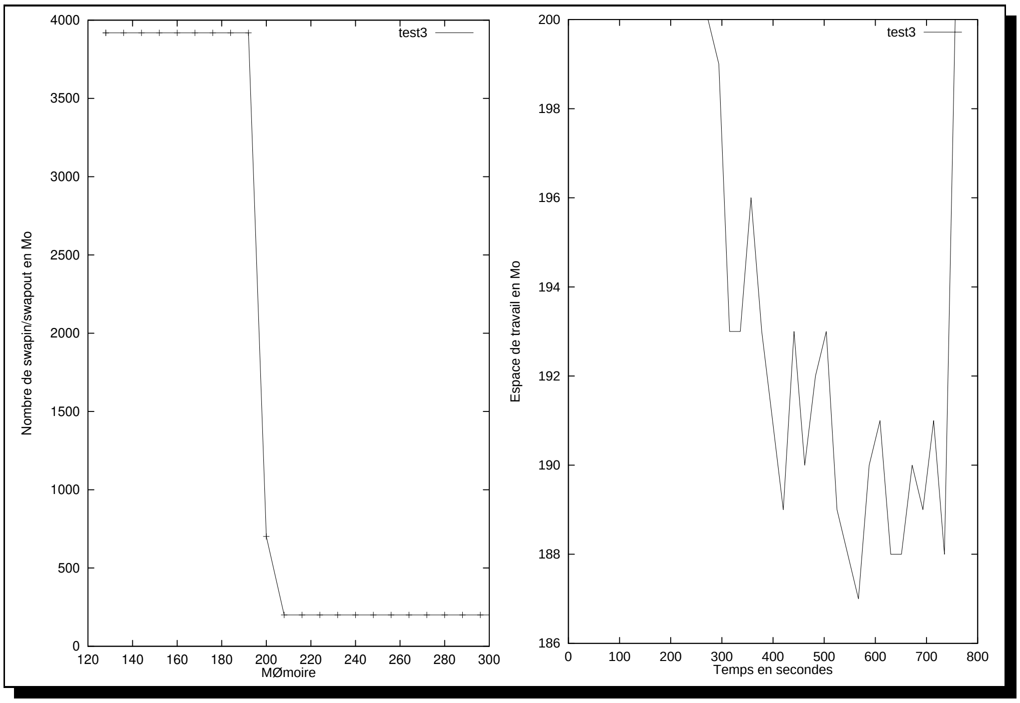


FIG. 3.6 – Taille mémoire optimale et espace de travail d'une convolution de deux vecteurs de 100Mo

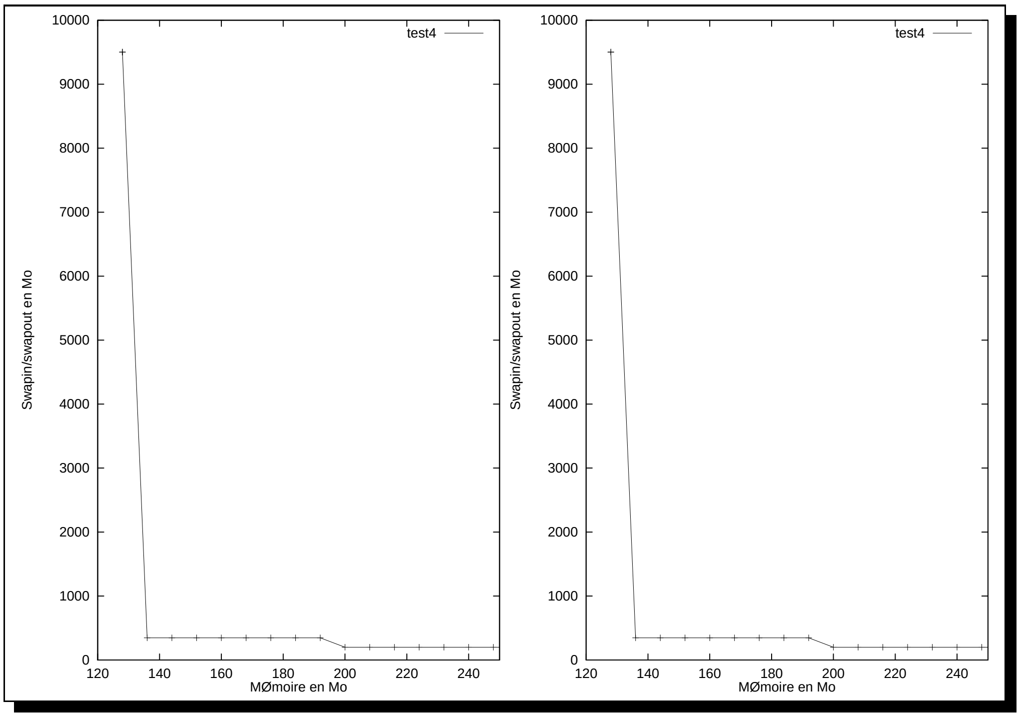


FIG. 3.7 – Taille mémoire optimale et espace de travail d'une succession de multiplications d'un vecteur de 120Mo

```
{
char t[12000000] ;
long j=0;
long k,i;
for (i=0 ;i<120000000;i++)
        t[i]=1;



for (k=0;k<10;k++)

for (i=0 ;i<120000000;i++)
        j=j+t[i]

}
```

```
{
char t[10000000] ;
char m[10000000];
long k,i;
for (i=0 ;i<100000000;i++)
        t[i]=1;
for (i=0 ;i<100000000;i++)
        t[i]=1;

for (k=0;k<10;k++)
for (i=0 ;i<100000000;i++)
        t[i]=t[i]*m[100000000-i];
}
```

FIG. 3.8 – Les deux programmes testés pas *anatrace*

travail fluctue beaucoup, ce qui montre que ce programme ne permet pas d'optimisation portant sur les données locales. Dans le cas du second test (fig 3.7), nous pouvons constater que la taille de mémoire optimale est supérieure à 135Mo, ce qui correspond à la taille du vecteur et nous pouvons constater que l'espace de travail (*Working Set*) est constant, sa taille est celle du vecteur.

Ce programme de collecte de trace, nous permet de détailler la taille optimale de la mémoire ainsi que la taille du *Working Set* à chaque instant, ce qui nous permet de connaître la quantité de mémoire optimale et qui peut nous orienter sur le choix d'une technique pour améliorer les performances.

3.2 Contrôle de la pagination d'un solveur multifrontal

Dans cette section, nous allons analyser le programme *MUMPS* : *mumps* [5, 6] est un solveur pour systèmes linéaires creux. Il est basé sur la méthode multifrontale [33, 34] qui est une méthode directe efficace pour la factorisation de matrices creuses. Cependant, comme les autres méthodes directes, elle est connue pour sa consommation mémoire importante en comparaison avec les méthodes itératives. Ainsi, lorsque la mémoire est limitée, on observe une forte activité de la pagination. Nous avons donc voulu étudier cette pagination pour concevoir un moniteur qui améliore les performances.

A cette fin, nous analysons *MUMPS* avec l'outil d'analyse de trace dont nous venons de parler. Nous validons ensuite cette analyse par une étude du code de l'application. Nous utilisons cette étude pour définir un moyen de communication entre l'application et la pagination. A cette fin, nous écrivons un gestionnaire de pagination implémentant cette interface. Enfin, nous montrons les gains obtenus grâce à cette méthode.

3.2.1 Analyse de la pagination de *MUMPS*

Nous allons commencer par étudier la trace avec notre programme d'analyse de la trace. Pour cela nous avons utilisé une machine avec 32Mo de mémoire afin d'avoir le maximum de défauts de page et donc d'information. Nous avons analysé *MUMPS* avec deux matrices (TWOTONE et SHIP 003 pour les intimes !).

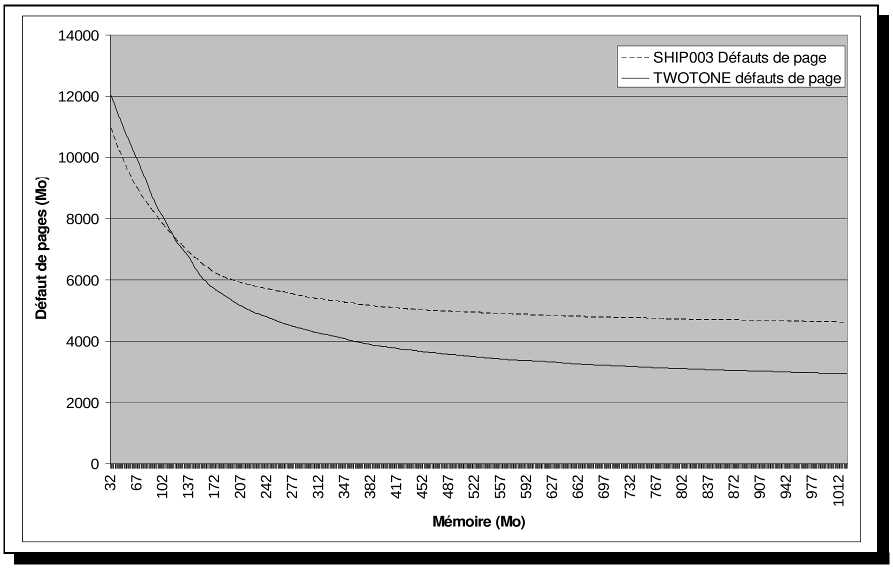


FIG. 3.9 – Taille mémoire optimale pour *MUMPS*

La figure 3.9 donne le nombre estimé de défauts de page en fonction de la mémoire de la machine.

On constate que le nombre de défauts de page est divisé par deux vers 160Mo pour *TWOTONE* et vers 184Mo pour *SHIP 003*. Il est ainsi possible de déduire quelle serait l'augmentation de performances par rapport à la taille de la mémoire.

Cette première étude ne nous renseigne pas sur la possibilité d'améliorer la pagination. Pour cela, nous avons calculé l'espace de travail utilisé à chaque seconde. Nous pouvons voir sur la figure 3.10 que l'espace de travail est fortement variable, néanmoins l'ensemble des abscisses sans point correspond à un espace de travail contenu en mémoire, c'est-à-dire aucun défaut de page. Nous pouvons en déduire que le préchargement de page et le déchargement de pages inutiles pendant ces périodes est non pénalisant, car le disque n'est pas utilisé pour la pagination. Le problème est de connaître quel est l'espace non utilisé pendant ces périodes, et quels sont les pages à précharger et décharger. Pour connaître ces paramètres, nous allons étudier plus en profondeur *MUMPS*.

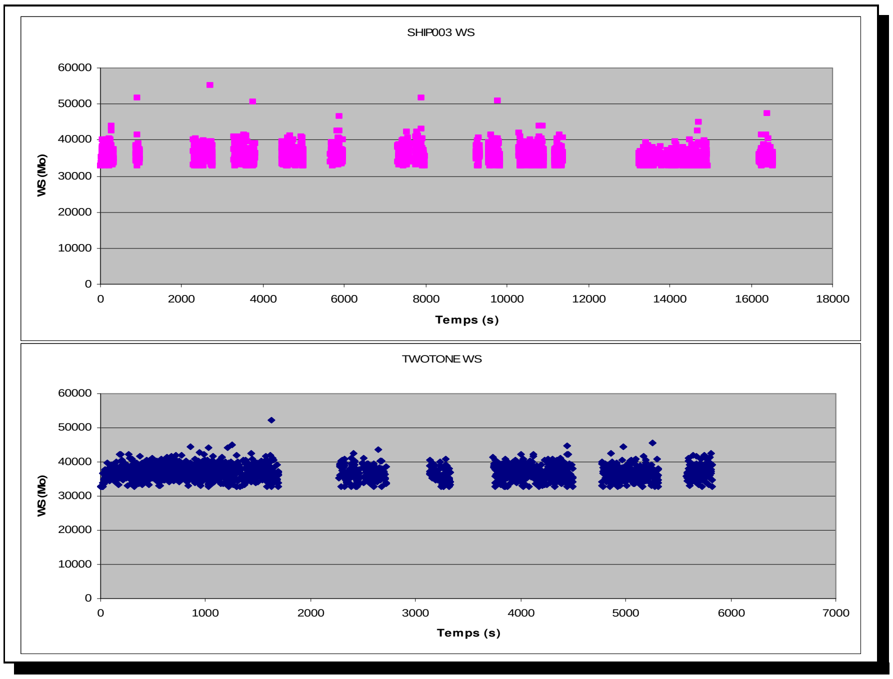


FIG. 3.10 – Espace de travail (*Working Set*) de *MUMPS* au cours du temps pour *MUMPS*

3.2.2 Description des accès mémoire de *MUMPS*

La méthode multifrontale est basée sur un arbre d'élimination [58] qui est une réduction transitive du graphe de dépendance des tâches de factorisation. En pratique, nous utilisons une structure appelée arbre d'assemblage obtenue par fusion de certains nœuds de l'arbre d'élimination.

Un exemple d'arbre d'assemblage est donné dans la figure 3.12. L'arbre est construit à partir de la matrice donnée dans la figure. Les deux feuilles sont indépendantes les unes des autres alors que la racine doit recevoir des contributions des feuilles.

Dans la méthode multifrontale, la factorisation de la matrice est effectuée par une succession de factorisations partielles de petites matrices denses, associées à chaque nœud de l'arbre appelé *matrices frontales*. La structure de la matrice frontale est donnée dans la figure 3.11. La matrice est composée de deux parties. La première correspond au bloc *facteur*, appelé aussi bloc des variables complètement assemblées, qui contient les variables qui ont été éliminées dans le processus de factorisation de la matrice frontale. Le *bloc de contribution* correspond à la deuxième partie de la matrice frontale, il correspond aux variables mises à jour (complément de Schur) pendant le processus de factorisation.

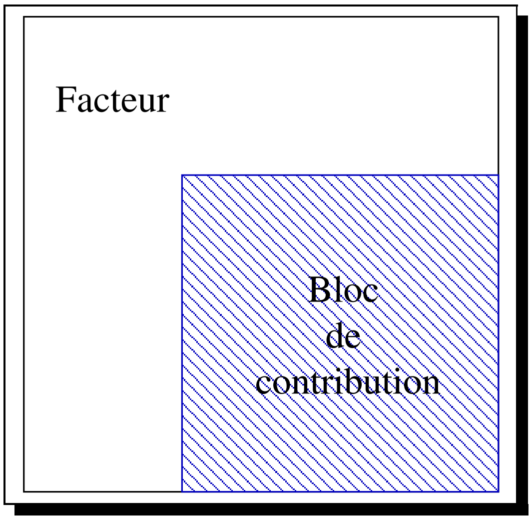


FIG. 3.11 – structure d'une matrice frontale.

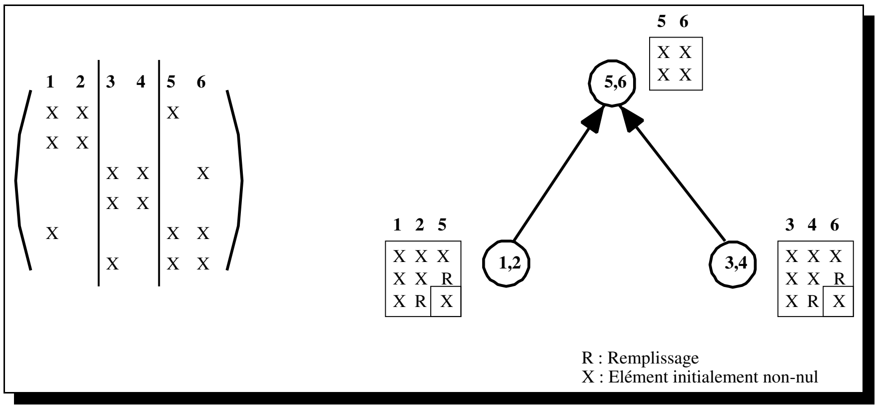


FIG. 3.12 – exemple de matrice et d'arbre d'assemblage associé.

Une fois la factorisation partielle de la matrice frontale effectuée, le bloc de contribution est envoyé au père du nœud auquel elle correspond. Une fois que toutes les contributions des fils sont disponibles au niveau du père, elles peuvent être assemblées (i.e. sommées avec les valeurs contenues dans la matrice du père). L'algorithme d'élimination est un parcours en remontée d'abord de l'arbre d'assemblage où un père est traité dès que ses fils l'ont été. L'algorithme utilise trois emplacements de stockage dans une région mémoire contiguë, une région pour stocker les facteurs, une région gérée comme une pile pour stocker les blocs de contribution, et une dernière région pour stocker la matrice frontale active (courante). Une description détaillée de la gestion mémoire est donnée par P. Amestoy [4].

Pendant le processus de factorisation, l'espace mémoire nécessaire au stockage des facteurs ne fait que croître en taille alors que celui correspondant à la pile (espace contenant les blocs de contribution) varie suivant la nature de l'opération effectuée. En effet, quand la factorisation partielle d'une matrice frontale est effectuée, un bloc de contribution est empilé ce qui augmente la taille de la pile. D'un autre coté, lors de la phase d'assemblage d'une matrice frontale, les blocs de contribution des fils du nœud courant sont consommés ce qui occasionne une diminution de la taille de la pile. La structure de l'espace de travail de *mumps* décrite précédemment est donnée à la figure 3.13.

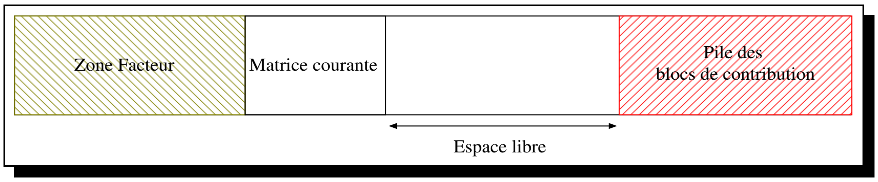


FIG. 3.13 – structure de la mémoire utilisée par *mumps*.

3.2.3 Stratégie d'optimisation de la pagination

Nous venons d'exposer le fonctionnement de *MUMPS*, nous pouvons en déduire certaines caractéristiques intéressantes pour l'amélioration de la pagination :

Temps d'inactivité de la pagination : le temps de calcul d'un front peut-être élevé tout en rentrant en mémoire, on observe peu de pagination pendant ce temps. Cette caractéristique rend ce calcul propice au préchargement des données utiles dans le futur et au déchargement des données inutiles.

Prévisibilité des données utiles : l'arbre d'assemblage est connu avant la factorisation, il est ainsi possible de connaître les données qui seront nécessaires pour les prochains calculs de facteurs, ainsi que les données inutiles. Ces informations permettent un préchargement ou un déchargement des données.

Afin d'utiliser ces caractéristiques, nous avons créé un deuxième processus nommé moniteur. Ce processus utilise *MMUM/MMUSSEL* pour influer sur la pagination de *MUMPS* et ainsi précharger ou décharger des pages. Le mode de communication entre ces deux processus est basé sur le concept de priorité : *MUMPS* donne une priorité différente aux régions de mémoire suivant leur utilité présente et à venir.

Description du moniteur de pagination

Le moniteur est un programme (1800 lignes en C) lancé par *MUMPS* qui utilise *MMUM/MMUSSEL* pour contrôler la pagination (fig. 3.14) reçoit de *MUMPS* des priorités[10] pour des plages d'adresse mémoire par la fonction *monitor_regmem(adr_debut,adr_fin,priorite)*. Ces priorités sont ensuite utilisées

[10] Environ 100 lignes ajoutées à MUMPS pour donner ces prioritées

pour connaître, suivant la mémoire disponible quelles sont les données à conserver en mémoire et quelles sont les données à évincer de la mémoire. L'algorithme cherche quelle est la priorité la plus faible qui permet de stocker en mémoire toutes les pages de priorités identiques ou supérieures. Le but est ensuite de stocker et d'évincer les pages pour obtenir cette configuration dite stable. Le problème du moniteur est qu'il n'a qu'une vue partielle des priorités, car *MUMPS* donne les informations au fur et à mesure du calcul. Le moniteur doit donc recalculer un état stable à chaque fois, ce qui peut engendrer des comportements inadaptés.

Parmi les priorités que peut passer *MUMPS* au moniteur, quelques unes sont spécifiques : la priorité nulle indique qu'une page doit être déchargée de la mémoire car elle ne sera pas utilisée avant longtemps, la priorité maximale indique qu'une page doit absolument être présente en mémoire (à moins qu'il n'y ait pas assez de mémoire). Ces deux priorités sont les seules utiles lorsque *MUMPS* ne fait pas de prédiction sur les prochains accès mémoires.

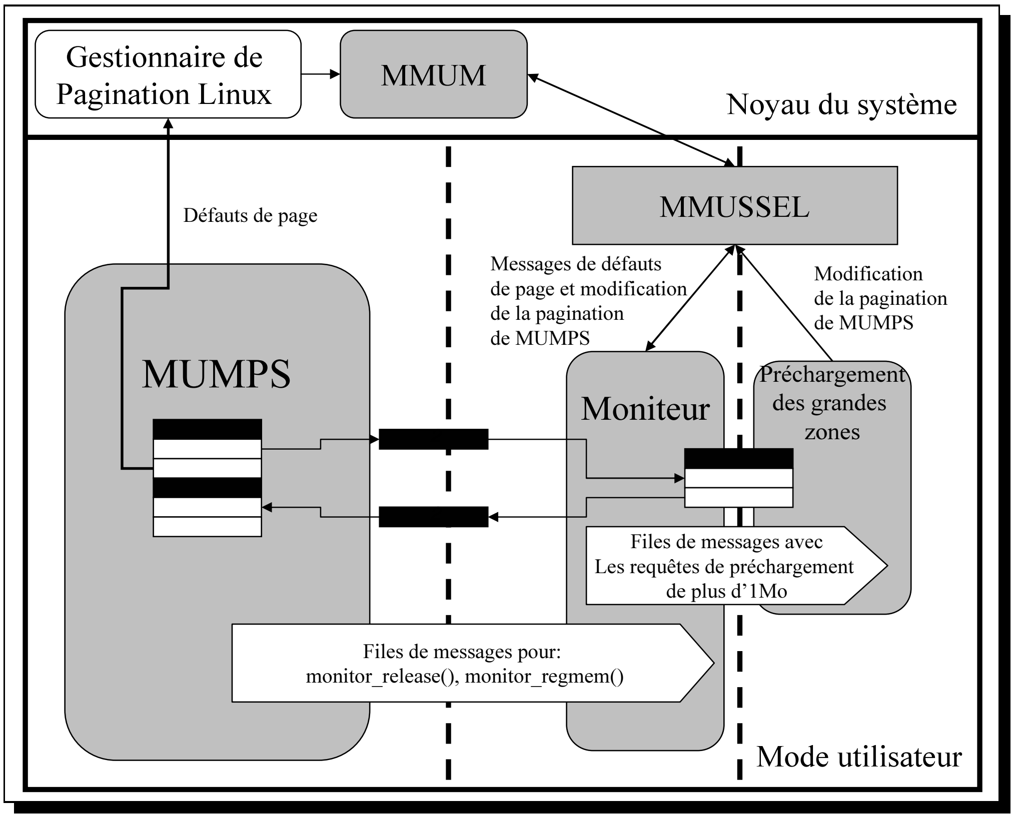


FIG. 3.14 – Relation entre le moniteur et *MUMPS*

Une primitive d'interaction entre *MUMPS* et le moniteur est *monitor_release(addr_debut,addr_fin)*. Cette commande signifie qu'une plage d'adresses contient des données qui ne seront plus jamais utilisées, c'est-à-dire que ces données peuvent être évincées de la mémoire sans être sauvegardées sur le disque. Ceci peut procurer des gains en terme de volume d'entrées/sorties et de latence. Cette primitive est principalement utilisée pour la gestion de la pile dans *MUMPS*. En effet, lorsque la zone mémoire en tête de la pile est libérée au niveau de l'application, les données qu'elle contenait peuvent être détruites. Ainsi, il est possible de la libérer avec *monitor_release*.

Une autre utilisation de cette commande est le déplacement de données. Ainsi, lors d'un déplacement de données, une première page est lue et copiée dans une autre, et ainsi de suite. La page qui vient d'être lue pour être copiée restera en mémoire du fait de l'algorithme LRU. Or, non seulement cette page n'est plus utile en mémoire mais en plus, il est inutile de la sauvegarder. Pour cela nous avons remplacé la fonction `memmove(adresse1,adresse2,taille)` par une fonction qui emploie la primitive *mum_release()* (voir fig. 3.15).

1. Si ADRESSE1 est au milieu d'une page copier la page alors
   (a) B=TAILLE_PAGE – (ADRESSE1 & TAILLE_PAGE)
   (b) ADRESSE1=ADRESSE1+B
   (c) ADRESSE2= ADRESSE 2+B
   (d) TAILLE=TAILLE-B
2. Tant que TAILLE > TAILLE_PAGE faire
   (a) Copier TAILLE_PAGE octets de ADRESSE1 vers ADRESSE2
   (b) ADRESSE1=ADRESSE1+TAILLE_PAGE
   (c) ADRESSE2=ADRESSE2+TAILLE_PAGE
   (d) TAILLE=TAILLE-TAILLE_PAGE
   (e) monitor_release(ADRESSE1,ADRESSE1+TAILLE_PAGE-1)
3. Copier TAILLE octets de ADRESSE1 vers ADRESSE2

FIG. 3.15 – Déplacements de données avec *mmum_release()*

Le moniteur et *MUMPS* utilisent un système de passage de messages pour communiquer. Ce système est basé sur une région de mémoire partagée et des sémaphores. Le moniteur lit les messages et les traite un à un. Néanmoins, pour des raisons de performances, les chargements de grande région de mémoire (plus de 1Mo) sont confiés à un troisième processus.

Certaines commandes demandent une synchronisation, c'est-à-dire qu'elles peuvent évincer certains messages de la file des messages ou attendre la fin de leur traitement. C'est le cas avec les *monitor_releases()* placés après les *monitor_regmem()*. En effet, il n'est plus nécessaire de charger une région si elle a été évincé par un *monitor_release()*. Un cas semblable se produit avec *monitor_regmem()* qui efface les messages précédents concernant une région qui est inclue. Ainsi, lorsque *MUMPS* donne l'information qu'une région à une faible priorité puis que cette région à une forte priorité, le moniteur ne traitera pas la première commande. Un autre cas de synchronisation concerne les *monitor_release()*. Par exemple, si le moniteur ne libère pas immédiatement les pages, *MUMPS* peut réutiliser ces pages avant la prise en compte du *monitor_release()*. Puis le moniteur détruira le contenu de ces pages, les données de cette page auront ainsi été perdues alors qu'elles sont encore utiles. Ce problème est résolu par la synchronisation du moniteur avec *monitor_release_sync()*

Interactions *mumps*-moniteur

Les communications entre *mumps* et le moniteur sont faites à des moments précis du processus de factorisation. Ainsi suivant le type de l'opération effectuée, assemblage ou factorisation partielle, *mumps* émet des requêtes visant à aider la gestion mémoire au niveau du moniteur. Un schéma résumant les différentes phases d'appels des fonctions de communication entre *mumps* et le moniteur est donné à la figure 3.16.

Les interactions entre *mumps* et le moniteur sont faites à chaque étape de la vie de la matrice frontale. Une description des différentes phases du traitement de la matrice frontale est donnée ci-dessous :

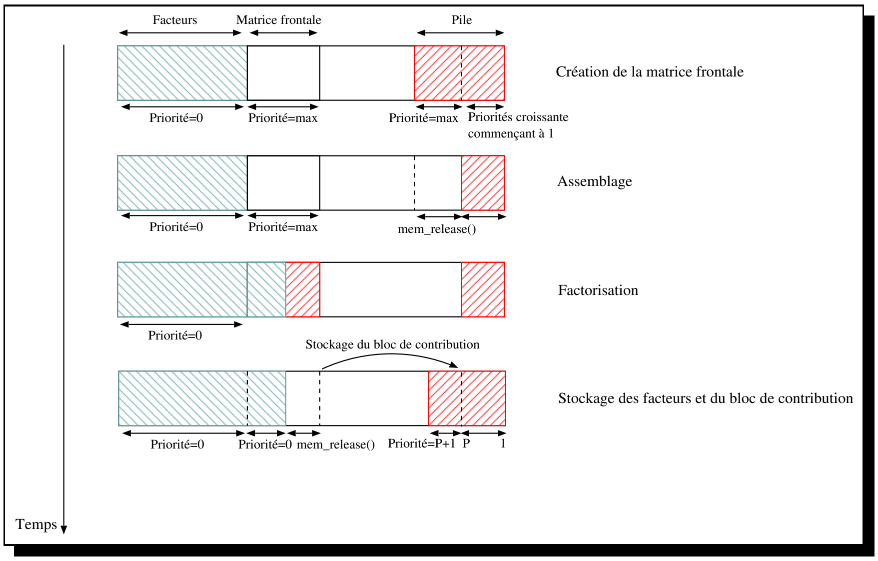


FIG. 3.16 – Schéma d'interaction *mumps*-moniteur.

Création de la matrice frontale. Avant la création de la matrice frontale, la priorité de la région qui doit la contenir est mise à la priorité maximale à l'aide d'un *reg_mem* (création de la matrice frontale à la figure 3.16).

Assemblage. Avant chaque étape d'assemblage, *mumps* émet des requêtes de type *reg_mem* sur les régions qui seront accédées durant cette phase. Ainsi il positionne la priorité de ces régions à la valeur maximale pour assurer leur présence en mémoire physique. Dès que l'assemblage d'un bloc de contribution est terminé, la région mémoire correspondante dans la pile est libérée à l'aide d'un *mem_release*.

Factorisation. La priorité de la région mémoire qui correspond à la matrice frontale étant déjà maximale, la factorisation se déroule sans communication entre *mumps* et le moniteur. Une fois la factorisation finie, le bloc de contribution de la matrice frontale est stocké dans la pile. Il prend alors une priorité égale à la priorité du bloc de contribution qui était en tête de pile plus un (priorité_cb=priorité_tête_pile+1). En ce qui concerne les facteurs de la matrice frontale, leur priorité est mise à zéro puisqu'ils ne seront pas réaccédés pendant la factorisation.

3.2.4 Résultats

Nous avons testé le moniteur et *MUMPS* sur plusieurs matrices symétriques et non symétriques. Sur chacune de ces matrices, nous avons effectué les optimisations que nous venons de décrire.

Nous exposons les gains dans la figure 3.20 et les temps dans la figure 3.19 et 3.21.

Cas de la matrice GUPTA3 Une première remarque que nous pouvons faire sur la matrice GUPTA3 est la diminution de mémoire accélère *MUMPS*. Une explication de ce phénomène as été présenté en section 1.3.1, nous allons la vérifier. La figure. 3.17 montre que vers 100 secondes, il n'y a plus d'évincement de page (swapout) avec 128Mo de mémoire, mais qu'il continue à en avoir avec 32Mo. Cela s'explique par le fait que la mémoire est insuffisante pour 32Mo et qu'ainsi les données utilisées dans la phase de calcul précédente sont libérées. Entre 100 et 150 secondes, un calcul sur de nouvelles données pouvant résider en mémoire est effectué. Ensuite, comme l'indique la figure 3.18, *MUMPS* utilise une autre région de mémoire et certaines données qui étaient en mémoire deviennent inutiles, il faut donc

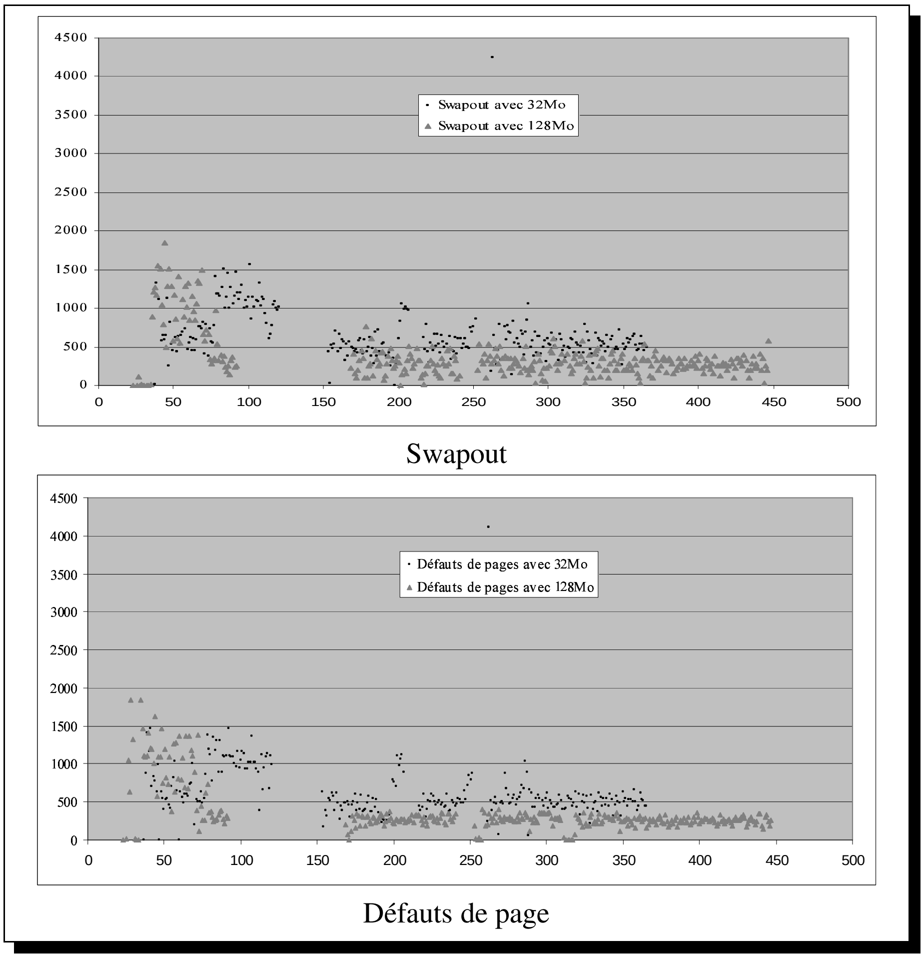


FIG. 3.17 – Gestion de la mémoire pour la matrice gupta3 avec 32Mo et 128Mo de mémoire

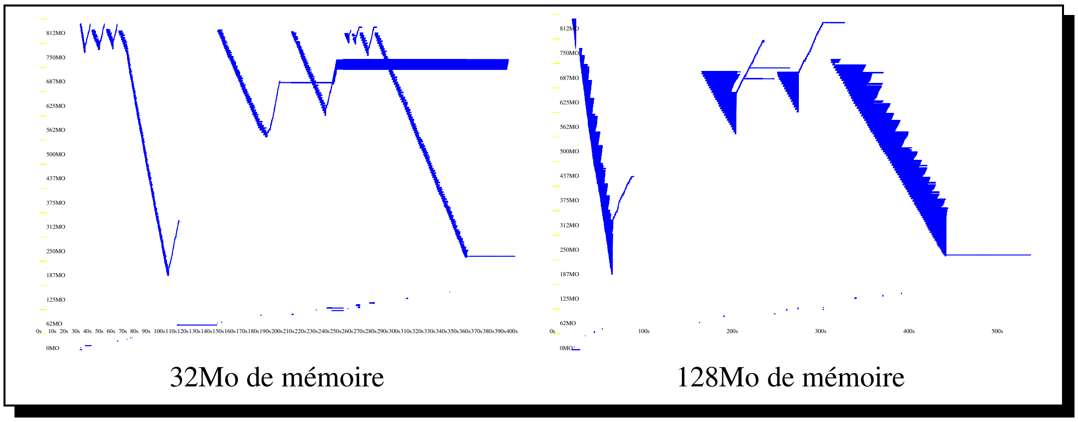


FIG. 3.18 – Utilisation de la mémoire pour la matrice gupta3 avec 32Mo et 128Mo de mémoire

les évincer de la mémoire. Avec 32Mo de mémoire les données utilisées avant le calcul avaient déjà été évincées pendant le calcul, avec 128Mo de mémoire il faut les évincer après le calcul. Le même phénomène se produit par la suite sur de plus petites durées, ce qui explique qu'il y a un taux d'évincements (swapout) inférieur avec 128Mo. Ainsi, ces déchargements en avance compensent celui venant d'un nombre de chargements/évincements des pages moins importants (environ 300 000 avec 128Mo et 400000 avec 32Mo).

Pour résumer, avec 32Mo les données commencent à être évincées plus tôt de la mémoire qu'avec 128Mo, or les données utilisé dans la phase de calcul précédent ne seront pas réutilisées, elles doivent donc être évincées le plus tôt possible de manière à profiter au maximum du débit du disque.

Matrice	Mémoire	Temps (s) avec le moniteur	Temps sans le moniteur
GUPTA3	32Mo	374	424
SHIP 003	32Mo	85168	92164
TWOTONE	32Mo	5626	6067
XENON2	32Mo	5737	6647
GUPTA3	64Mo	408	444
SHIP 003	64Mo	2764	3896
TWOTONE	64Mo	1867	1571
XENON2	64Mo	1876	2118
GUPTA3	128Mo	524	576
SHIP 003	128Mo	1114	1207
TWOTONE	128Mo	429	292
XENON2	128Mo	734	678

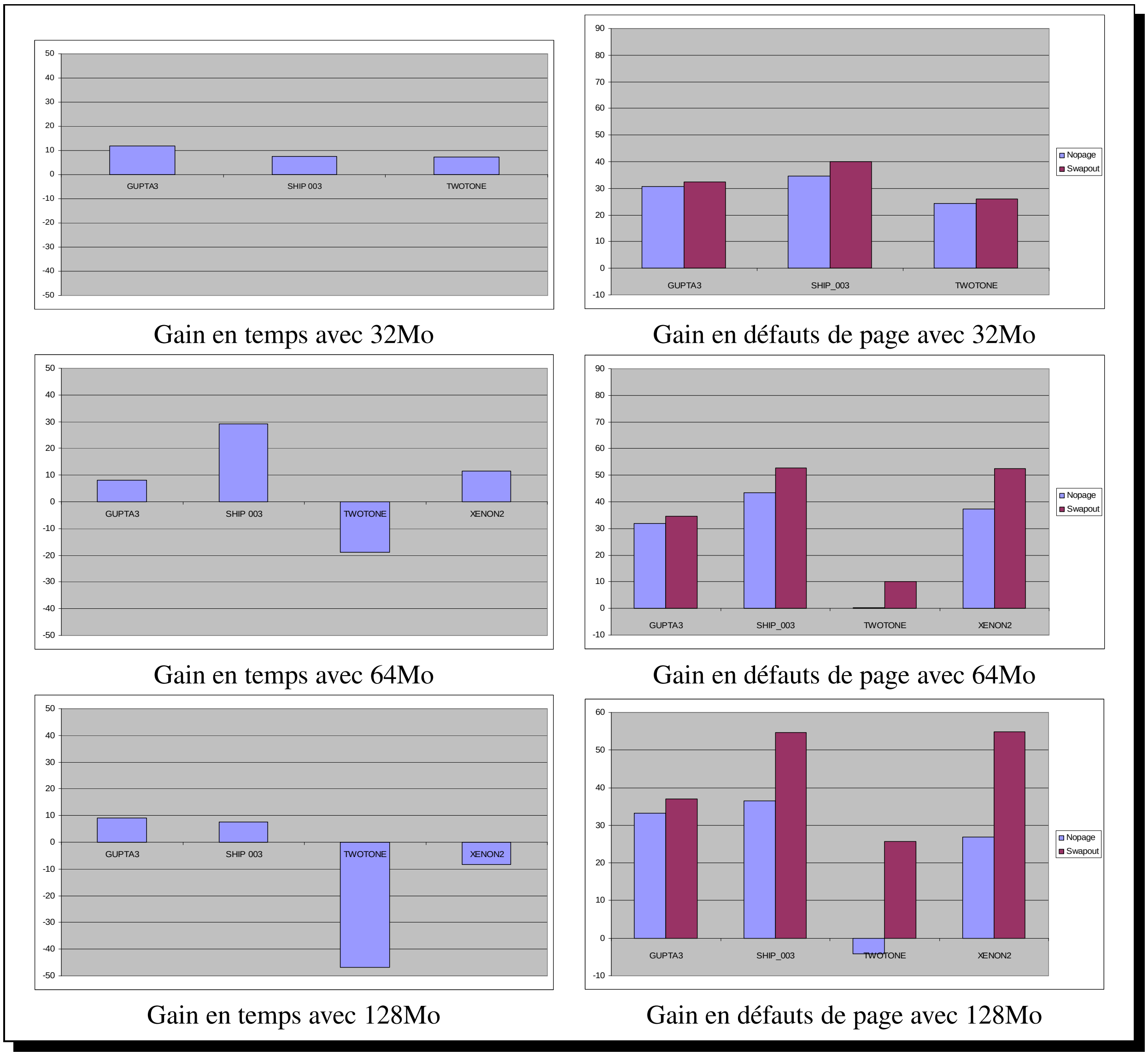
FIG. 3.19 – Temps d'exécution séquentiel de *MUMPS* avec et sans le moniteur sur diverses matrices

MUMPS séquentiel Nous pouvons constater (fig. 3.20) que nous obtenons une réduction des évincements (swapout) dans tout les cas. C'est principalement dû au mécanisme de libération de la mémoire *monitor_release()* décrit précédemment. Concernant le nombre de défauts de page, nous observons un gain dans la plupart des cas à l'exception de la matrice TWOTONE avec 64Mo et 128Mo. En effet, comme le moniteur consomme approximativement 5 Mo de mémoire, les matrices frontales qui étaient contenues en mémoire sans le moniteur, ne peuvent plus l'être. De ce fait, la présence du moniteur augmente considérablement l'activité de pagination pour de telles matrices frontales.

Habituellement, une réduction du nombre de défauts de page diminue le temps d'exécution (fig. 3.19), ce qui n'est pas vérifié sur la matrice XENON2 avec 128Mo. En fait, l'augmentation du temps de factorisation est dû au surcoût du moniteur. En effet, il y a un grand nombre de communication pour des opérations à grains fins (moins de 1Mo). Comme illustration, nous avons mesuré 95 000 requêtes faite au moniteur en 600 secondes d'exécution, alors que dans le cas de SHIP_003 avec 128Mo, nous obtenons 35 000 requêtes pour 1 200 secondes.

Finalement, nous observons de meilleurs gains pour 64Mo que pour 32Mo ou 128Mo. Pour 32Mo le nombre de matrices frontales qui ne peuvent être contenues en mémoire est supérieur à 64Mo. En effet, les factorisations représentent la majeure partie du temps d'exécution et il n'y a aucune optimisation de la pagination pendant la factorisation d'une matrice frontale. Pour 128Mo, le ralentissement dû a la pagination est faible, car il y a peu de défauts de page et leur diminution n'a que peu d'impact sur les temps d'exécution.

Cas parallèle Nous avons également fait des tests avec une version parallèle de *MUMPS*. Les figures 3.22.et 3.21 donnent les temps et les gains en pourcentages d'une exécution sur 4 et 6 nœuds avec 64Mo et 128Mo de mémoire par nœud.



Gain en temps avec 32Mo

Gain en défauts de page avec 32Mo

Gain en temps avec 64Mo

Gain en défauts de page avec 64Mo

Gain en temps avec 128Mo

Gain en défauts de page avec 128Mo

FIG. 3.20 – Gain de *MUMPS* séquentiel (en pourcentage) par l'utilisation du moniteur

Matrice	Mémoire	Processeur	Temps (s) avec le moniteur	Temps sans le moniteur
SHIP 003	64Mo	4	1044	1205
BMWCRA 1	64Mo	4	335	393
XENON2	64Mo	4	762	801
PRE2	64Mo	4	5902	6349
ULTRASOUND3	64Mo	4	6768	8029
SHIP 003	128Mo	4	566	552
BMWCRA 1	128Mo	4	243	241
XENON2	128Mo	4	561	540
PRE2	128Mo	4	3098	3634
ULTRASOUND3	128Mo	4	3657	6179
PRE2	128Mo	6	2779	3172
ULTRASOUND3	128Mo	6	3228	3307

FIG. 3.21 – Temps d'exécution parallèle de *MUMPS* avec et sans le moniteur sur diverses matrices

Nous obtenons de légères pertes pour les petites matrices (XENON2 et BMWCRA_1) avec 4 nœuds et 128Mo. Cet effet provient principalement du moniteur. Nous obtenons des gains intéressants pour les autres problèmes, particulièrement pour le problème ULTRASOUND3 qui expose un gain de 45% avec 4 nœuds de 64Mo. Concernant les résultats sur 6 nœuds, nous constatons des gains atteignant 12% du temps d'exécution.

Il est important de remarquer que le moniteur à un impact important sur la distribution des taches de *MUMPS* sur les nœuds, car elle est dynamique. En effet, l'exécution avec et sans le moniteur donne une distribution différente des taches. Afin d'illustrer ce point, nous avons fait l'expérience suivante : nous avons forcé la stratégie statique de sélection des esclaves, la distribution des taches découlant de cette décision est ainsi fixé. Dans ce cas, nous obtenons un gain de 18% avec ULTRASOUND3 sur 6 nœuds avec 64Mo, alors qu'avec la stratégie fixée nous n'avions aucun gain. Nous pouvons ainsi déduire qu'il est intéressant de prendre en considération l'activité de pagination dans la distribution des taches.

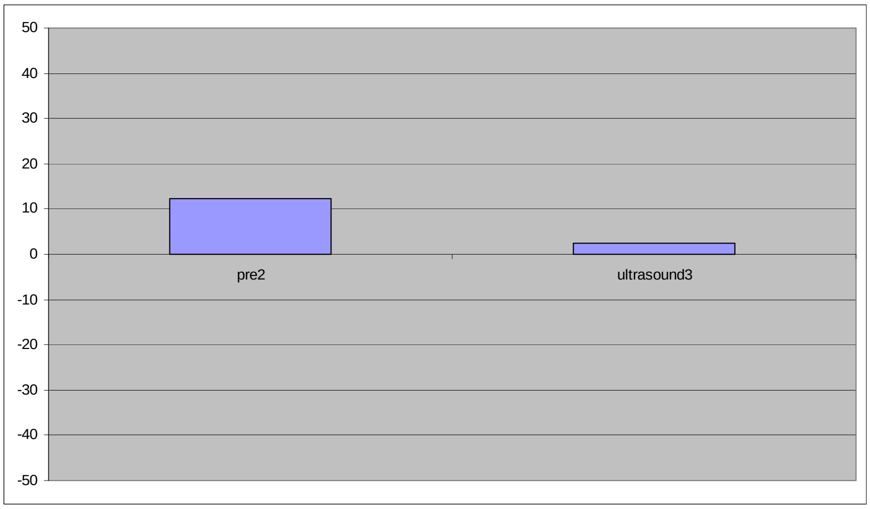
3.3 Conclusion

Nous venons de décrire une amélioration de MUMM/MMUSSEL qui perturbe moins le système que la précédente. L'amélioration repose sur l'utilisation d'un processus extérieur à l'application pour contrôler sa pagination. Nous avons montré l'intérêt du contrôle de la pagination pour augmenter la vitesse des copies et nous avons utilisé *MMUM/MMUSSEL* en coopération avec *MUMPS*.

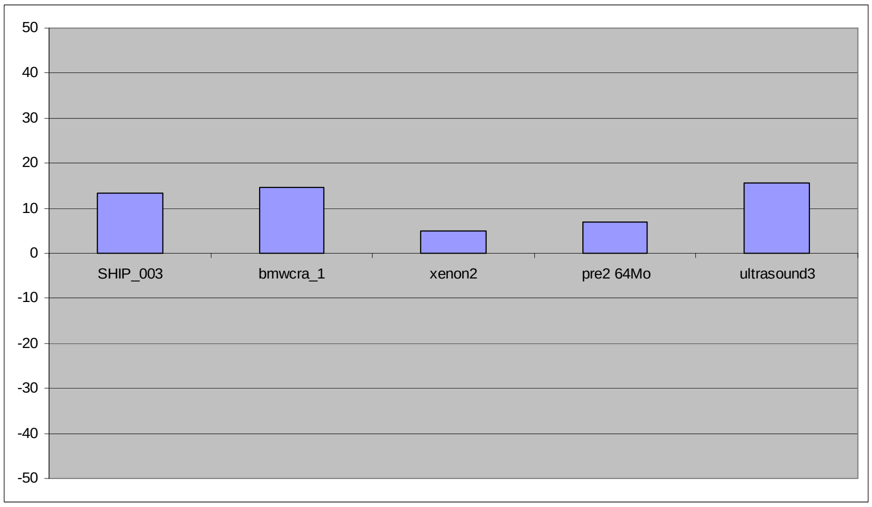
L'interface entre *MMUM/MMUSSEL* et *MUMPS* est standardisée et permet de faire coopérer d'autres programmes avec *MMUM/MMUSSEL*. Ceci avait été réalisé par Todd Mowry [63] [15], mais ce dernier avait une interface moins riche qui ne permettait que d'indiquer les données à précharger ou à décharger, c'est-à-dire l'état de la mémoire voulu à un instant donné. L'avantage de notre approche est de donner plus d'information pour la pagination. Ces informations, permettent de ne pas sauvegarder inutilement des pages qui ne seront plus jamais utilisées et elles permettent d'orienter l'ordre du préchargement par des priorités. Nous avons ainsi une meilleure gestion de la mémoire disponible et des accès au disque.

Un développement futur pourrait être l'amélioration des entrées/sorties, car les gains en défauts de page et évictions ne se traduisent pas directement en gains de temps.

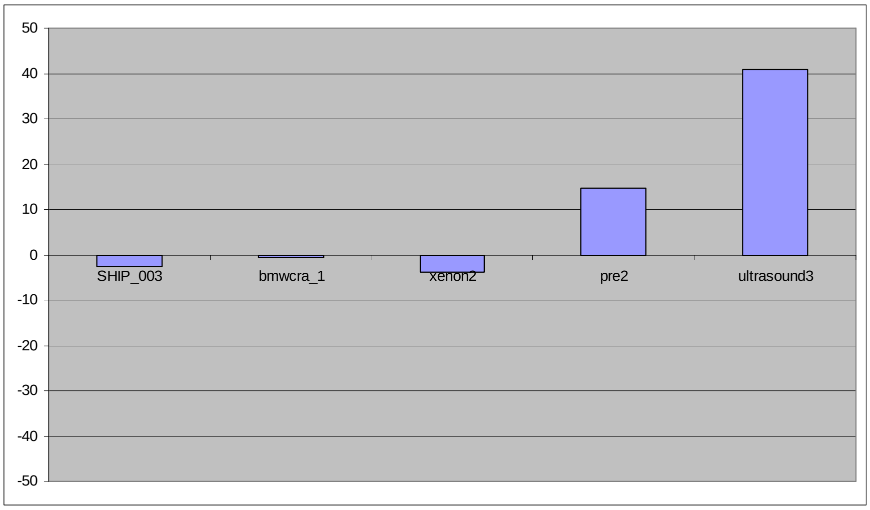
L'interface avec *MUMPS* s'est fait dans le cadre séquentiel et montre des gains intéressants. Les premiers tests sur une version parallèle sont encourageants, sur quatre ou six processeurs nous obtenons des gains significatifs, malgré un ordonnancement qui ne prend pas en compte la charge mémoire. Une perspective future sera le perfectionnement du moniteur et le coordonnancement des taches de *MUMPS*.



128Mo avec 6 processeurs



64MB avec 4 processeurs



128Mo avec 4 processeurs

FIG. 3.22 – Gain de *MUMPS* (en pourcentage) par l'utilisation du moniteur avec une grappe

Deuxième partie

Accès aux disques distants : READ^2

Introduction

Dans la partie précédente, nous avons étudié les limites de la pagination, ces limites sont également liées aux limites des disques. Dans le cas d'une grappe de stations, il est possible de ne pas se limiter aux disques locaux, mais d'utiliser également les disques des autres machines. Nous obtenons ainsi un espace de stockage plus important, nous pouvons espérer une augmentation du débit disque et un partage plus aisé des données.

A cette fin, nous étudions dans cette partie les mécanismes mis en jeux pour l'accès aux disques distants dans une architecture de type grappe de stations et comment améliorer leur efficacité. Ces améliorations auront un impact pour les applications effectuant de nombreuses entrées/sorties.

Les applications telles que le *data mining*, le traitement du signal et la vidéo à la demande (*VOD*) utilisent beaucoup d'espace disque et nécessitent des entrées/sorties performantes. Par exemple, la collection de tableaux numérisés du musée de San Fransisco représente plus de 3 Téraoctets. La masse de données à traiter par l'accélérateur de particules européen (le LHC du CERN[44] est estimée à plusieurs Petaoctets par an. Pour pouvoir traiter de telles applications, on peut employer des grappes de stations à base de composants standard. Par exemple, le musée de San Fransisco [77] utilise 20 PCs avec 368 disques. L'un des principaux attraits d'une telle solution est le prix. De plus, actuellement il apparaît aussi de nouveaux systèmes de fichiers parallèles et distribués (PVFS[57], GFS[68], PPFS[46],...) pour grappes, permettant de tirer partie de l'agrégation des espaces de stockage de chaque nœud.

Pour les applications qui effectuent du calcul intensif sur de grande masses de données (traitement du signal, calcul out-of-core, tri de base de données,...), le système d'entrées/sorties est mis à rude épreuve. De par son architecture où rien n'est partagé (*shared nothing*), chaque nœud d'un cluster agit à la fois comme client et comme serveur pour pouvoir accéder globalement à l'ensemble des données, ceci en plus des calculs qu'il a à effectuer. Il a été montré par D. Patterson [9] que les architectures de type grappe étaient particulièrement pénalisées car le bus d'entrée/sortie pouvait rapidement devenir le goulot d'étranglement. A cette fin de nombreux travaux ont été entrepris pour diminuer l'utilisation des bus et limiter le problème des entrées/sorties.

Les premières améliorations ont été faites pour les serveurs *NFS*. Un serveur *NFS* permet de partager des fichiers locaux avec de nombreux clients. Afin de limiter l'utilisation du processeur du serveur, *DAFS* [28] permet au client de lire et d'écrire directement dans le cache du serveur, le déchargeant pour toute requête portant sur une donnée dans son cache. Cependant, l'inconvénient de *NFS* est de concentrer les requêtes sur un unique nœud, ce noeud est ainsi le point faible du système. L'approche de Lombard et Denneulin[60] consiste à rendre le système de fichiers parallèle *PVFS*[57] accessible à distance par des clients *NFS*, il n'y a plus serveurs mais des serveurs. Néanmoins, le *NFS* nécessite que le client ne voit qu'un serveur, c'est pourquoi les différents nœuds usurpent l'adresse du serveur (*IP spoofing*).

D'autres techniques de plus bas niveau sont apparues pour projeter les disques sur l'ensemble des nœuds du *cluster*. Il s'agit de *NBD* et *RAID-x* [47]. Dernièrement, de nouvelles techniques pour accéder aux données de disques distants sont apparues. Elles sont basées sur les nouvelles capacités des cartes réseaux. Les données du disque transitent directement de ce dernier au client par la carte réseau. Il s'agit d'*ORFA* [41], *OPIOM* [37], *DREAD* et des travaux de Maierhofer [62]. Nous appelons cette technique l'accès distant direct.

Dans le premier chapitre de cette partie nous étudierons précisément le fonctionnement des disques, puis les éléments d'une grappe de stations mise en jeu pour l'utilisation de ces disques.

Avec l'accès distant direct, les données stockées sur des disques distants sont directement accédées, sans utilisation de la mémoire locale du nœud distant. Dans le deuxième chapitre de cette partie, nous étudierons la limite théorique d'extensibilité d'une telle architecture : cette limite est déterminée par la bande passante d'entrées/sorties maximale, qui peut être atteinte lorsque les ressources disques et réseaux sont étendues. Nous considérerons l'amélioration de cette extensibilité par le passage du schéma classique client/serveur à l'accès distant direct. En plus du débit, nous étudierons la réduction de la contention mémoire.

Dans le troisième chapitre, nous étudierons l'ensemble des composants qui entrent en jeu pour l'implémentation d'un accès distant direct et nous présenterons une validation expérimentalement et les gains qui peuvent être obtenus. La bibliothèque que nous avons réalisé se nomme READ^2, elle se compose d'un pilote intégré dans une carte réseau (environ 9000 lignes en C), d'un firmware pour un contrôleur SCSI (environ 300 lignes dans le langage assembleur du contrôleur) et d'un pilote Linux (environ 300 lignes en C).

Chapitre 4

Description des disques et des bus d'entrées/sorties

Sommaire

Nous avons évoqué dans la partie précédente quelques caractéristiques des disques. Afin de mieux appréhender la suite de cette partie qui est centrée sur ce composant, nous allons décrire plus en détail le disque et son lien avec les autres composants. Nous décrirons ainsi l'architecture physique du disque, les divers bus et les contrôleurs qui lui sont associés. Nous décrirons également son utilisation par le système d'exploitation à travers les systèmes de fichiers et l'utilisation de nombreux disques avec le *RAID*.

4.1 Architecture du disque

Un disque dur est composé de plusieurs plateaux (disques) rotatifs, surmontés d'une tête de lecture par plateau (toutes solidaires entre elles). Ces têtes se déplacent perpendiculairement au sens de rotation. Le disque est divisé en cercles concentriques appelés cylindres et chaque cylindre est découpé en secteurs. Lors de la lecture de secteurs contigus, on lit les secteurs qui se suivent sur un cylindre du premier plateau et on passe à la tête suivante qui lit les secteurs du même cylindre sur le deuxième plateau. Une fois que tous les secteurs du cylindre ont été lus, on passe à un autre cylindre. La vitesse de rotation, la vitesse de déplacement des têtes, la densité d'information et la taille des plateaux déterminent les performances du disque dur. Ces performances sont mesurées par le débit et la latence du disque dur :

Le débit : il correspond à la vitesse moyenne obtenue lorsque les données sont lues sur des secteurs contiguës, c'est-à-dire accédées comme décrit précédemment. Pour déterminer ce débit, deux paramètres interviennent. Le premier est la vitesse de rotation, qui est entre 7200 tours par minute pour les disques usuels et jusqu'à 15000 tours par minute pour les meilleurs. Le second est le nombre de secteurs par cylindre ou la densité d'information. Le débit correspond ainsi au produit de ces deux paramètres.

La latence : elle correspond au temps moyen nécessaire pour atteindre un secteur, ceci dépend du temps de déplacement de la tête sur le cylindre cible. Mais même dans le cas où la tête est sur le bon cylindre, il faut attendre que le secteur passe sous la tête, ce qui dépend de la vitesse de rotation.

Les débits varient entre 20Mo/s et 80Mo/s et les latences entre 2ms et 10ms. Si on considère un disque de 40Mo/s avec un temps de latence égal à 2ms avec des secteurs de 512 octets, alors le temps moyen pour accéder à un secteur placé aléatoirement correspond au temps pour lire 80Ko. Cela signifie que pour lire des secteurs non contigus, la latence est un facteur limitant, faisant baisser le débit à 256ko/s (512 octets toutes les 2ms). C'est la latence qui devient alors prépondérante ce qui explique que l'on essaye de rendre les accès contigus.

Les échanges de données avec un disque se font toujours par secteur (couramment 512 octets). Ces données sont demandées au disque qui ne reçoit cette demande que lorsqu'il a envoyé le résultat de la précédente requête. Ce phénomène introduit une latence qui ne dépend pas de la latence de lecture physique. Pour diminuer l'impact de cette latence sur le débit, il est possible de lancer plusieurs requêtes avant la réponse de la première (tag queue). Cette dernière possibilité était, jusqu'à récemment, uniquement disponible avec les disques SCSI.

Du fait de leurs latences spécifiques élevées, les disques sont en général accédés par un bus dédié. Ce bus est contrôlé par des contrôleurs spéciaux, les plus courants étant l'IDE et le SCSI.

Ces contrôleurs induisent une latence supplémentaire, car il y a en plus du temps de chargement des secteurs par le disque, le temps de communication avec le contrôleur. Pour donner un exemple de ce phénomène, prenons une lecture séquentielle d'un disque avec un ancien contrôleur IDE PIO3 limité à 10Mo/s avec un disque permettant un débit du même ordre.

Le disque dispose d'un tampon d'anticipation qui lui permet de précharger la suite d'une piste. Dans ce cas, le préchargement et la lecture linéaire rend négligeable la latence provenant de la lecture physique disque, il ne reste plus que la latence du contrôleur.

Pour modéliser cette latence, nous prenons les notations suivantes :

γ	débit réel
β	temps de latence du contrôleur
n	taille de données
k	nombre de requêtes
ν	débit théorique du disque

On obtient la relation : $\gamma = \frac{1}{n*\frac{1}{\nu}+k*\beta}$, cette formule suggère, que même dans le cas de lectures contiguës, il est préférable de faire une grande requête de lecture plutôt que plusieurs petites.

La figure 4.1 donne les gains. Dans ce test, le temps entre deux demandes de groupe était tel que quel que soit le groupement les secteurs était lus dans les tampons du disque. Ce test a été effectué sur un disque IDE PIO3 par programmation directe du contrôleur, avec la lecture de 60 secteurs, soit 15 pages de 4Ko en une fois, le débit atteint plus de 6500Ko/s, ce qui correspond à 50% de plus que la lecture d'une page de 4Ko à la fois (fig. 4.1).

4.2 Bus associé au disque

Les performances des bus vont influer sur les performances constatées des disques. Ainsi, si les disques atteignent un plus grand débit que ce que peut supporter le bus, ce dernier limitera les performances. Ces bus se caractérisent par un ensemble de commandes et un ensemble de protocoles physiques.

L'IDE inclut un ensemble de commandes permettant de contrôler des périphériques de type blocs, c'est-à-dire des périphériques où chaque bloc de données est accessible aléatoirement, par exemple un disque dur, un lecteur/graveur de CD-ROM, une carte SmartMedia ou une cassette DAT. La norme inclus également l'interface du contrôleur, ce qui permet d'utiliser un pilote unique pour tout les contrôleurs IDE. Cette norme permet deux protocoles physique : le parallèle et le série. Le bus IDE parallèle a un débit maximal qui peut aujourd'hui atteindre entre 130Mo/s et 160Mo/s. Ce bus ne supporte que deux

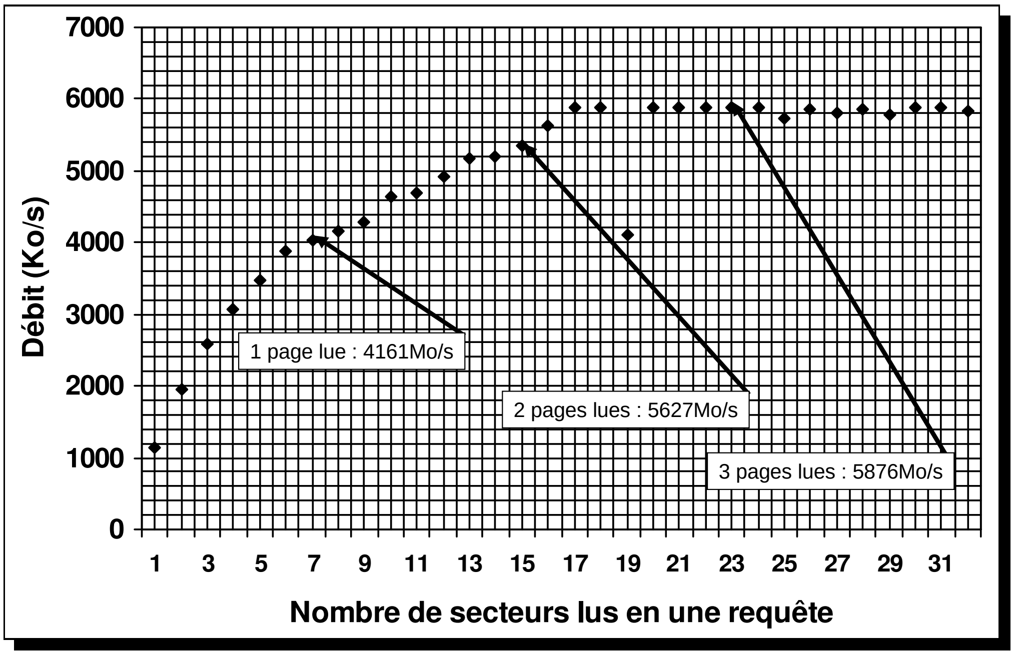


FIG. 4.1 – Vitesse de lecture suivant le nombre de blocs lus en une fois (Test effectué par la programmation directe du contrôleur de disques)

périphériques qui ne peuvent pas communiquer entre eux : ils sont limités à l'envoi ou la réception de données avec le contrôleur. Le bus IDE série ou *Serial ATA* permet de diminuer le nombre de fils nécessaire à la connexion entre le contrôleur et le périphérique, ce qui permet d'augmenter la vitesse maximale (200Mo/s). Néanmoins, ce bus ne permet pas de connecter plus d'un périphérique.

La norme SCSI inclut, comme l'IDE, un ensemble de commandes pour périphérique de blocs. Ces commandes impliquent directement la possibilité de communiquer entre deux périphériques. Plusieurs protocoles permettent d'utiliser les commandes SCSI, certains sont basés sur une technologie série : *Firewire*, *SSA*, *SAS*, *FibreChannel* [32] ou *HIPPI* et permettent de créer un réseau contenant plusieurs centaines de périphérique sur plusieurs kilomètres. Néanmoins, le support physique le plus courant est le bus SCSI parallèle (*SPI [66] : SCSI Parallel Interface*).

Ce bus permet d'avoir jusqu'à 16 périphériques par bus. Pour utiliser avantageusement ce bus, le protocole permet aux périphériques de communiquer entre eux. Ainsi, un disque dur peut envoyer des données à un autre sans passage par le contrôleur SCSI du nœud. Afin de gérer ces multiples périphériques, le protocole utilise 16 identificateurs pour les périphériques, l'identificateur le plus petit étant toujours prioritaire pendant l'arbitrage. Cet arbitrage se produit lors d'une déconnexion d'un périphérique ou d'une erreur, il se caractérise par l'inutilisation du bus pendant un court instant (*bus free*).

Ce protocole contient deux modes, le mode commande pour envoyer ou recevoir une requête qui est à 5Mo/s et le mode données qui est à la vitesse maximale du bus (320Mo/s pour l'ultra3 wide). Le mode de commande permet de négocier quelles seront les vitesses maximales du mode données ainsi que l'envoi des requêtes et des erreurs. Ces deux modes permettent de lire ou d'écrire des données sur un périphérique tel que le disque.

Néanmoins, l'attente entre la demande de lecture et le début de l'envoi des données est grande et le bus serait monopolisé pendant un temps important. Afin de limiter ce problème, le protocole permet de libérer le bus (fig. 4.2), c'est la déconnexion pendant le traitement de la requête par le disque et il y a reconnexion du disque lorsque les données sont prêtes. Nous utiliserons cette technique pour pouvoir utiliser deux disques en même temps.

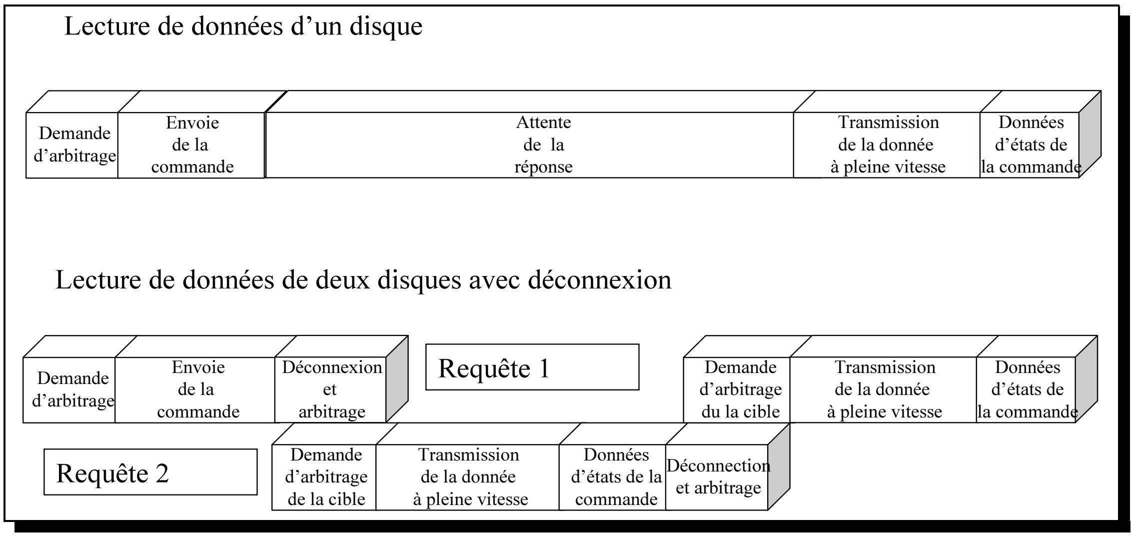


FIG. 4.2 – Utilisation du bus SCSI avec ou sans déconnexion

4.3 Contrôleur de disques

Le contrôleur est un pont entre le disque est les autres composants du nœud. Il fait transiter les données, les commandes et les erreurs, entre le bus du disque et le bus permettant d'accéder à la mémoire centrale et au processeur : le bus d'entrées/sorties qui est souvent le bus PCI. Les opérations sur le bus SCSI sont nombreuses et les transferts de données sur le bus d'entrées/sorties sont pénalisantes pour le processeur central. Afin de limiter ces problèmes, la majorité des contrôleurs SCSI disposent de fonctions d'accès direct à la mémoire *DMA* et d'un microcontrôleur assurant les opérations de contrôle du bus SCSI. Un exemple de ce contrôleur implémentant ces fonctionnalités est le *LSI Logic* SYM53C8xx dont nous parlerons dans la section 6.2.3.

4.4 Du disque aux fichiers : les partitions et le système de fichiers

Les systèmes d'exploitation divisent le disque en plusieurs unités logiques nommées partitions. Ces unités permettent de stocker des données de différentes natures, comme par exemples plusieurs systèmes d'exploitation. Ces unités logiques sont accédées par l'intermédiaire de systèmes de fichiers, qui gèrent la manière dont sont stockés les fichiers sur les disques. Les principaux systèmes de fichiers sont FAT (MS-DOS), Extended 2/3 (Linux), NTFS (Windows) et HFS+ (Max OS X). Les systèmes de fichiers ont deux fonctions :

Allocation et gestion de l'espace : le système de fichiers fait la correspondance entre chaque fragment d'un fichier et l'emplacement de ce fragment sur le disque. Il se charge également de trouver des emplacements libres pour la création ou l'agrandissement de fichiers existants. Il faut noter que la correspondance n'est pas directe. C'est-à-dire qu'un fichier de 8 secteurs n'est pas obligatoirement placé sur 8 secteurs consécutifs. Néanmoins, le système de fichiers essaye de mettre les fichiers sur des secteurs consécutifs de façon à y accéder plus rapidement.

Gestion des méta données : le système de fichiers permet d'accéder aux fichiers par l'intermédiaire de répertoires qui contiennent les données associées aux fichiers. Parmi ces données se trouve l'endroit où commence le fichier, sa taille, ses autorisations d'accès ou sa date de modification. Il faut noter que les répertoires sont gérés pour l'allocation et la gestion de l'espace comme de simples fichiers.

4.5 Techniques RAID

Certaines applications nécessitent de nombreux disques pour stocker leurs données et des semaines de calcul. Dans ces cas, la fiabilité des disques est importante car la panne d'un disque devient fortement probable et pénalisante. Afin de tolérer ces pannes un certain nombre de stratégies ont été élaborées, elles sont regroupées sous le nom de *RAID*. Les niveaux *RAID* [71] : Redondant Array of Independant Disks correspondent chacun à une gestion spécifique de la redondance :

RAID0 : cette technique consiste à mettre le premier secteur de la partition sur le premier disque, le second sur le deuxième et ainsi le secteur k sur disque $k\ modulo\ n + 1$. Lorsqu'une lecture se produit, par exemple pour les secteurs 1 à $n - 1$, la lecture est effectuée sur tous les disques en même temps. Le temps de lecture est donc celui du disque le plus lent, par contre le débit est plus élevé car les lectures sont effectuées en parallèle. Cette méthode s'appelle l'entrelacement (*stripping*) ou *RAID0* (fig. 4.3).

RAID1 : un disque sur deux est une copie : c'est du *mirroring* ou *RAID1*. Dans ce cas le secteur k est sur le disques $2 * (k\ modulo\ \frac{n}{2}) + 1$ et sur le disque $2 * (k\ modulo \frac{n}{2}) + 2$. La lecture du secteur 1 et 2 est effectuée respectivement par le disque 1 et le disque 2

RAID4 : le *RAID1* permet une tolérance à la panne d'un disque sur deux, cependant il est possible d'avoir moins de disques consacrés à la redondance. Avec le *RAID4*, un disque contient une somme de contrôle des autres disques[11]. Lors de la perte d'un disque, il est possible de retrouver la donnée manquante en soustrayant au secteur du dernier disque la somme des autres.

RADI5 : le *RAID4* place les sommes de contrôles sur un unique disque qui est utilisé pour chaque modification d'un secteur des autres disques (pour chaque écriture), le *RAID5* réparti les données de contrôle sur chaque disque.

4.6 Conclusion

Nous venons de voir les divers composants architecturaux mis en œuvre dans les accès aux disques ainsi que les méthodes utilisées par le système pour stocker les fichiers sur ces disques. Nous pouvons donc modéliser sa latence et son débit, néanmoins nous nous intéresserons particulièrement à son débit dans la suite.

[11]C'est-à-dire que le bit l du disque n est égal à un ou exclusif des bits l des autres disques.

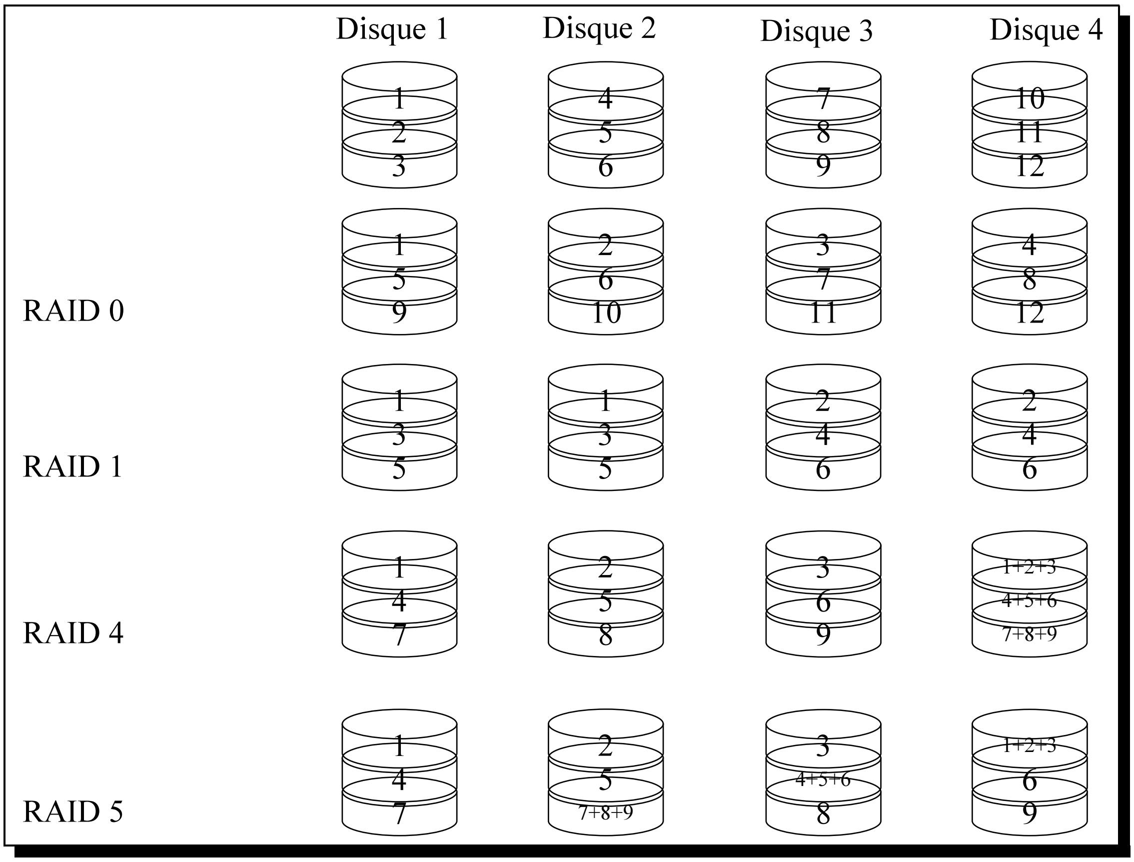


FIG. 4.3 – Les différents modes *RAID*

Chapitre 5

Saturation des bus dans les grappes

Sommaire

Dans la section précédente, nous avons étudié le composant disque et son fonctionnement. Nous allons maintenant intégrer ce disque dans une grappe de machines. Pour cela nous décrivons les bus de chaque machine qui transportent les données en provenance ou à destination du disque, c'est-à-dire du contrôleur de disques. Nous décrivons plus en détail un de ces bus, le bus PCI qui joue le rôle de bus d'entrées/sorties.

Nous en déduisons que nous pouvons considérer, pour certaines applications, ces bus uniquement du point de vue de leurs débits. Pour cela nous nous basons sur une étude faite par Patterson [9]. Nous étudions plusieurs applications et nous montrons leurs limitations dues au bus d'entrées/sorties. Puis, nous montrons que grâce aux possibilités des cartes réseaux récentes nous pouvons réduire cette limitation d'un facteur de 50%.

5.1 Description des grappes de stations courantes et du bus PCI

Les grappes de stations sont construites en assemblant des composants reliés par des connexions que l'on nomme bus : ces bus permettent le transfert de données d'un composant à un autre. Chaque nœud est relié aux autre par le réseau. Le nœud lui-même est constitué d'éléments tels que la mémoire, le processeur, le disque, le pont d'entrées/sorties ou une carte réseau. Ces éléments sont connectés par divers bus. Dans les architectures telles que le PC, ces interconnexions sont réalisées par trois bus (fig. 5.1) :

- Le bus d'entrée/sorties (E/S) qui relie les disques, le pont, les cartes réseaux et les autres composants d'extensions comme certaines cartes graphiques (les cartes PCI). Ce bus est généralement le bus PCI, qui dans sa version de base (32bits, 33Mhz) est limité a 132Mo/s et qui dans sa version avancée (PCI-X 64bits,133Mhz) est limité a 1056Mo/s.
- Le bus mémoire qui relie le pont et la mémoire. Ce bus a un débit entre quelques centaines de mégaoctets par seconde et plus de 5Go par seconde pour les dernier bus de Pentium4.

– Le bus processeur qui relis le pont et le processeur. Ce bus a un débit qui est égal à celui du bus mémoire.
– Le bus disque : IDE ou SCSI. Ce bus se distingue des autres par sa latence (fig. 5.2).

Débit concerné	Exemples
Débit du bus d'entrées/sorties	PCI 32bits 33Mhz : 132Mo/s PCI-X 133Mhz : 1Go/s AGP 8x : 4Go/s
Débit du bus mémoire	SDRAM 133 :1Go DDR SDRAM 400 : 3,2Go/s DDR SDRAM 400 bicanal 6,4Go/s
Débit du réseau	Ethernet 100 :12Mo/s Myrinet 2000 :200Mo/s Infiniband[49] plus de 800Mo/s

FIG. 5.1 – Quelques débits de bus

Parmi les diffèrent éléments du nœud, le pont est le seul à être connecté à tous les bus, car il est responsable d'acheminer les données qui viennent d'un bus et qui vont vers un autre. Par exemple, pour qu'une carte réseau accède à la mémoire, elle envoie ses données sur le bus E/S : ces données passent par le pont et sont envoyées sur le bus mémoire avant d'être reçues par la mémoire.

Bus	SCSI Ultra3	PCI 66Mhz 64bits
Taille minimum d'une requête de lecture/écriture	512 octets	1 octets
Taille maximum d'une lecture/requete	32ko/256 octets	256octets/80bits
débit	160Mo/s	560Mo/s
Temps d'arbitrage	100us (bus free)	<15ns (2 cycles de bus)
Nombre de transactions simultanées maxima	non bornées	2 (pipelining)
Latence	Non borné	<300ns (4 cycles à 32 cycles)
Nombre de périphérique	16	4 (+sous bus)

FIG. 5.2 – Comparaison du bus SCSI et du bus PCI

Une caractéristique des bus est qu'une seule donnée peut passer à un moment donné. Ainsi, dans l'exemple précèdent, le processeur ne pourra pas accéder au disque, car le bus d'E/S est utilisé par la carte réseau. Néanmoins, pour permettre qu'un composant ne le monopolise pas complètement, il existe un mécanisme de partage des bus. En effet, pour chaque paquet (64 octets pour le bus PCI), il y a un arbitrage qui garantit l'équité.

Une description générale des bus du point de vue logique et électronique a été faite par Hsu et Peir[45], ainsi qu'une description du bus PCI par le *PCI SIG*[74]. Nous nous contenterons ici d'une description rapide du bus PCI afin de mieux comprendre le transfert de données sur ce bus (voir fig. 5.3 pour plus d'informations). Un transfert de données entre deux périphériques PCI est appelé transaction. Une transaction est constitué de plusieurs phases : l'arbitrage, l'envoi de l'adresse, la sélection du périphérique et le transfert de données. Ces transactions se comportent différemment suivant le mode qui peut être : *pipelining* et *burst*, l'un des deux, ou aucun des deux. Le *pipelining* correspondant à une transaction qui commence avant la fin de la précédente et le mode *burst* correspond au transfert de plusieurs données consécutive en une seule transaction.

Voici une description plus approfondie des phases de ce bus :

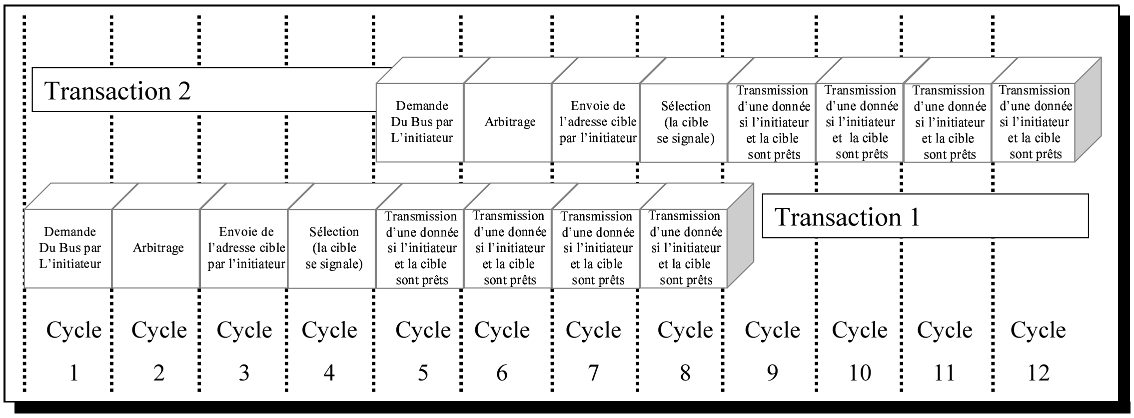


FIG. 5.3 – Transferts de données sur le bus PCI.
Ce schéma ne représente pas exactement les cycles PCI, mais montre que l'on peut envoyer une donnée par cycle

Arbitrage : la transaction précédente n'est pas achevée ou aucune n'est en court. Pendant cette phase, chaque périphérique qui veut initier un transfert se signale. Un composant central détermine qui obtient le contrôle du bus et transmet l'acceptation au périphérique qui sera nommé initiateur.

Envoi de l'adresse : la transaction précédente n'est pas achevée si le mode *pipelining* est activé (ce qui est presque toujours le cas pour les régions de mémoire physique), sinon il faut attendre la fin de la transaction précédente. Pendant cette phase, l'adresse que le périphérique veut atteindre est envoyée ainsi que l'opération à effectuer. Ces opérations sont de trois natures :

- **Lecture/Ecriture mémoire :** le périphérique veut écrire des données à cette adresse mémoire ou il veut lire des données de cette adresse mémoire.
- **Lecture/Ecriture de port E/S :** le périphérique veut accéder à des registres spéciaux d'un autre périphérique. L'adresse indique le périphérique et le registre. Les opérations sur les ports d'E/S sont conçues pour programmer des périphériques comme le lancement de requête par le contrôleur disque, l'émission d'un son par le haut parleur interne ou le lancement d'un envoi de données par l'interface réseau. A cette fin, l'accès au port d'E/S ne permet ni le *pipelining*, ni le mode *burst* (décrit ci-dessous), ni de garder les données dans le cache du processeur. Ceci permet de ne pas avoir de problème de cohérence, mais ralentit ces accès.
- **Lecture/Ecriture espace de configuration :** Le périphérique veut accéder aux données de configuration d'un autre périphérique. Ces données concernent l'activité d'un périphérique, ses capacités et les ressources qu'il utilise. Comme pour les ports d'E/S, le *pipelining*, le mode *burst* et la duplication des données dans le cache du processeur est interdite.

Sélection du périphérique : la transaction précédente n'est pas achevée si le mode *pipelining* est activé (ce qui est presque toujours le cas pour les régions de mémoire physique), sinon il faut attendre la fin de la transaction précédente. Le périphérique concerné par l'adresse se signale à tous les autres et se prépare à recevoir ou à émettre des données : ce périphérique est appelé la cible.

Transfert des données : une donnée est transférée, le bus PCI est de 32 ou 64 bits : une donnée d'au plus 64 bits est transférée. Si le mode *burst* est activé, ce qui est le cas pour la mémoire, la donnée destinée à l'adresse suivante peut immédiatement être transférée, sans repasser par les étapes précédentes. Néanmoins, certains périphériques tels que la carte graphique que nous verrons en 5.5.2.0 ne peuvent supporter un transfert de données à chaque cycle. Il est ainsi nécessaire pour transférer une nouvelle donnée que le périphérique initiateur et le périphérique cible aient signalé qu'ils sont prêts. Il faut noter que toutes les autres étapes de la prochaine transaction, à part le transfert de données peuvent se dérouler en même temps que ce transfert, c'est le *pipelining*.

Chacune des opérations précédentes dure un cycle de bus, et un cycle de bus par donnée pour le transfert. Cette dernière opération étant effectuée en même temps que les autres, chaque cycle du bus permet de transférer une donnée. Ainsi, dans les cas usuels, le bus PCI 64bits 66Mhz permet de transférer $64bits * 66000000 = 533Mo/s$. Nous pouvons ainsi considérer les bus comme de simples tuyaux qui sont caractérisés par leurs débits.

5.2 Description de l'architecture

Nous venons de décrire le fonctionnement du bus d'entrées/sorties PCI d'une machine et nous avons vu qu'il est semblable aux autres bus. Nous utilisons cette description pour une modélisation de plus haut niveau dans laquelle nous ne prendrons en compte que le débit de ce bus.

Pour cela, nous considérons une grappe de stations de *n* nœuds pour laquelle les interfaces réseaux et les disques sont suffisamment nombreux pour maximiser les performances globales des entrées/sorties. Chaque nœud peut être considéré comme un bus d'entrées/sorties (Bus E/S), un bus mémoire, un bus processeur et un réseau, l'ensemble étant interconnecté par un pont d'entrées/sorties.

Dans ces divers bus, le meilleur débit est en général celui du bus processeur. Ce qui explique que ce n'est pas un goulot d'étranglement, c'est pour cela qu'il ne sera pas considéré dans notre modèle. Pour une application donnée, les variables suivantes correspondent aux débits utilisés pour chaque bus.

Débit concerné	maximum	utilisé
Débit disque	M_r	D_r
Débit du bus d'entrées/sorties	M_{io}	D_{io}
Débit du bus mémoire	M_{mem}	D_{mem}
Débit du réseau	M_{res}	D_{res}
Instructions exécutées par seconde	M_{cpu}	D_{cpu}

L'élément central d'une grappe de stations pour le flux d'entrées/sorties est le bus d'entrées/sorties : il est connecté aux disques locaux, au pont d'entrées/sorties et à l'interface réseau. Néanmoins, c'est couramment le bus le plus lent.

En effet, le débit réseau peut être accru par l'addition d'interfaces réseau et le débit disques peut être accru par l'addition d'autres disques, alors que le débit du bus d'entrées/sorties ne peut pas être étendu. Comme exposé dans un article de Patterson[9] et un article de Jacob et al[51], ce bus est le goulot d'étranglement d'une grappe de stations pour les flux d'entrées/sorties. Pour maximiser les performances, le réseau et les disques peuvent être étendus jusqu'à saturer le bus d'entrées/sorties.

En considérant le niveau de technologie actuel, une telle architecture est viable. Par exemple, prenons une grappe de stations qui comprend 10 PC. Chaque nœud est composé d'un bus mémoire de débit 1Go/s (SDRAM 133Mhz) et d'un bus d'entrées/sorties a 500Mo/s (PCI 64bits 66Mhz). Les interfaces réseaux et les disques sont connectés au bus d'entrées/sorties, qui les limite. Comme ce bus à un débit de 500Mo/s, alors un débit de 500Mo/s pour les disques (25 disques SCSI et deux contrôleurs SCSI) et de 500Mo/s pour le réseau (5 cartes Gigaoctets) est suffisant.

Dans l'analyse, les applications sont supposées traiter un flot continu de données. Les données sont stockées sur tous les disques dont le débit agrégé sera un des facteurs qui déterminent l'accélération des applications. Pour l'étude de leurs performances, le processeur est supposé suffisamment puissant. Pour atteindre l'optimum, l'accès aux disques et aux interfaces réseaux est fait en parallèle. De tels accès sont possibles par un *pipelinage* de traitements de données. Le bus d'entrées/sorties ne peut être augmenté, il devient un goulot d'étranglement. Les performances maximales sont ainsi atteintes lors de la saturation de ce bus.

5.3 Maximisation du débit des entrées/sorties

Patterson[9] compare les stations de travail, les grappes de stations et les multiprocesseurs du point de vue de l'extensibilité du débit des entrées/sorties. Il considère plusieurs applications avec différents besoins de calculs (tri, lecture d'une base de données, transpositions de matrices). Leurs analyses montrent que pour la grappe de stations usuelle, le bus d'entrées/sorties est un goulot d'étranglement à cause de l'agrégation du trafic de données des disques et des interfaces réseaux. C'est particulièrement le cas pour l'application de transposition de matrices, c'est pourquoi nous restreignons notre étude à ce cas. Les performances sont modélisées en considérant une application qui fait de nombreuses transpositions de matrices.

Soit n le nombre de nœuds dans la grappe. Chaque transposition consiste à transposer une matrice de taille $M * M$ distribuée sur l'ensemble des nœuds. La matrice est découpée en blocs de taille $B * B$, tels que $nB = m$ (un bloc est contenu dans la mémoire d'un nœud). Chaque ligne de blocs est affectée à un nœud et stockée sur ses disques (fig. 5.4). Pour chaque processeur i, son $j^{ème}$ bloc de la matrice transposée sera le $i^{ème}$ du $j^{ème}$ processeur. La figure 5.4 décrit l'algorithme correspondant. Chaque

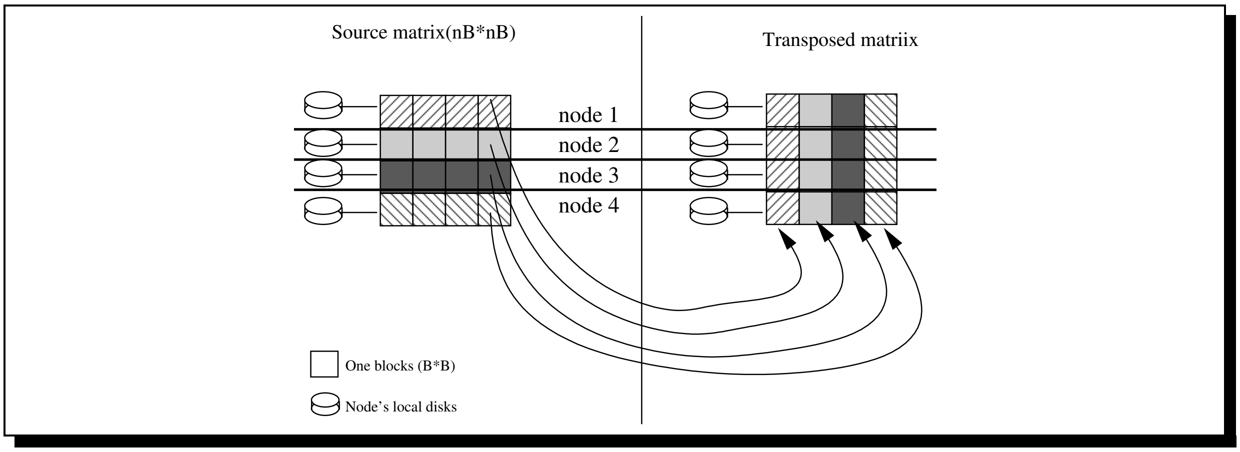


FIG. 5.4 – Organisation des données sur chaque nœud et redistribution de données lors d'une transposition

nœud doit lire $n-1$ blocs des autres nœuds, 1 en local et en envoyer $n-1$ aux autres.

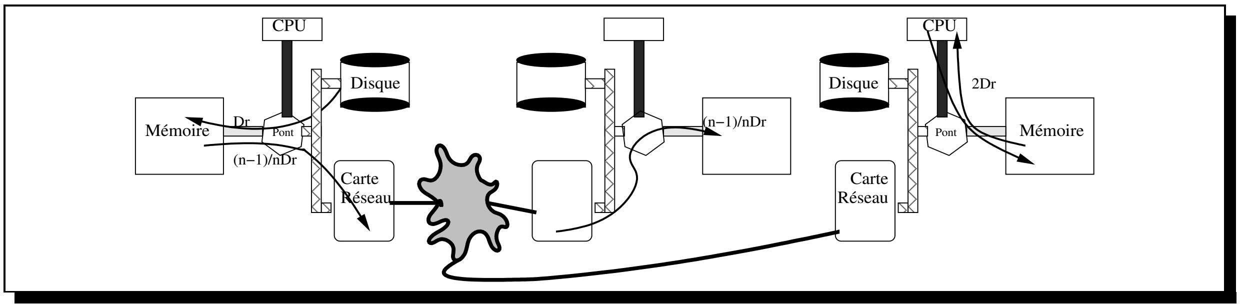


FIG. 5.5 – Utilisation des bus pour les lectures/écritures disque

Si D_r est le débit des disques en lecture sur chaque nœud, alors le débit du bus d'E/S correspond à (fig. 5.5) :

- la lecture des blocs du disque vers la mémoire D_r ;
- l'envoi des blocs du disque local (en mémoire) utilisés par les autres nœuds $\frac{n-1}{n}D_r$;
- la réception des blocs venant des autres nœuds par la carte réseau $\frac{n-1}{n}D_r$;

– l'écriture des résultats D_r.

L'application étant constituée d'une succession de telles transpositions, les opérations sont *pipelinées* : pendant que les blocs d'une matrice sont lus, d'autres sont transposés en mémoire et d'autres sont écrits. Ainsi, le débit utilisé du bus d'E/S (noté $D_{E/S}$) est la somme de chacune de ces actions élémentaires :

$$D_{E/S} = D_r + \frac{n-1}{n}D_r + \frac{n-1}{n}D_r + D_r = 4D_r - \frac{2}{n}D_r \quad (5.1)$$

Toutes ces transactions passent par la mémoire et sont comptabilisées dans l'utilisation du bus mémoire. Mais chaque bloc doit aussi être transposé. Pour cela le processeur doit lire chaque donnée en mémoire et la réécrire en mémoire, le débit utilisé pour cette opération est donc $2D_r$. Le débit total utilisé en mémoire, noté D_{mem} est :

$$D_{mem} = D_r + \frac{n-1}{n}D_r + \frac{n-1}{n}D_r + D_r + 2D_r = 6D_r - \frac{2}{n}D_r \quad (5.2)$$

Le processeur étant supposé suffisamment véloce, quand cette application s'exécute seule, elle n'est limitée que par le bus d'E/S ou par le bus mémoire, les autres ressources étant supposées suffisantes. On en déduit que le débit (D_r) maximal de cette application est atteint lorsque l'un des deux bus est saturé ($M_{mem} = D_{mem}$ ou $M_{E/S} = D_{E/S}$), c'est-à-dire

$$si\ M_{E/S} = 4D_r - \frac{2}{n}D_r\ ou\ M_{mem} = 6D_r - \frac{2}{n}D_r$$

Ainsi, si l'on ajoute des disques au nœud, la bande passante de la transposition ne pourra excéder $\frac{n}{4n-2}M_{IO}$.Le bus d'entrées/sorties est le goulot d'étranglement et limite l'extensibilité de la grappe de stations.

5.4 Améliorer l'extensibilité avec l'accès distant direct

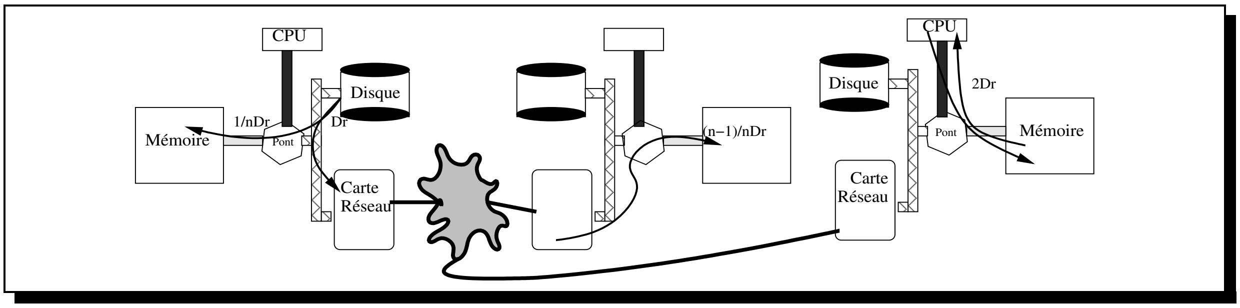


FIG. 5.6 – Utilisation des bus avec l'accès distant direct

Dans les cas précédents, les blocs lus sur le disque local du nœud sont stockés en mémoire avant d'être envoyés à leurs destinataires. L'accès distant direct consistera en l'envoi des blocs directement aux cartes réseaux (fig. 5.6). L'utilisation du bus mémoire et du bus d'E/S s'en trouve diminué. Les actions et les débits associés sont :

– la lecture des blocs du disque vers la mémoire $\frac{1}{n}D_r$;
– l'envoi direct des blocs du disque local utilisé par les autres nœuds $\frac{n-1}{n}D_r$;

– la réception des blocs venant des autres nœuds par la carte réseau $\frac{n-1}{n}D_r$;
– l'écriture des résultats D_r.

L'application sature les ressources lorsque

$$M_{E/S} = 3D_r - \frac{2}{nD_r} \; ou \; M_{mem} = 4D_r - \frac{2}{nD_r}$$

Ce qui donne, avec les mêmes hypothèses que précédemment, un débit maximum de $D_r = \frac{n}{3n-2}M_{E/S}$. Soit un débit de 25% supérieur au cas précédent.

5.5 Contention

Dans cette section, l'étude sera consacrée au bénéfice de l'accès distant direct pour la maximisation des performances d'une application. Cette étude expose un autre bénéfice de l'accès distant direct, qui est la diminution de la contention pour l'accès mémoire.

Afin d'étudier l'impact de la contention, nous modéliserons et expérimenterons les gains apportés à deux applications qui s'exécutent de manière concurrente. La première application sera un filtre d'image, qui utilise intensivement l'accès à la mémoire. La deuxième est une redistribution de données sur disque, qui utilise activement l'accès aux entrées/sorties.

5.5.1 Modélisation

Afin d'exposer le problème de la contention, nous allons modéliser l'utilisation des bus de deux application : un Filtre qui correspond à une opération sur une image et une Redistribution.

Les performances du Filtre sont représentées par le nombre d'instructions du Filtre traité par le processeur (D_{cpu}) et le débit du disque (D_r) représente les performances de la Redistribution. Les performances seront modélisées lorsque le Filtre et la Redistribution sont exécutés de manière indépendante et lorsque qu'ils sont exécutés de manière concurrente.

Performances isolées

Filtre. Le Filtre utilise uniquement la mémoire et le processeur. Le calcul utilise un certain nombre d'instructions par accès mémoire, noté α_{cpu}. Ce processus est constitué de deux phases, la lecture de N octets et leurs traitements. La durées de chaque phase est ainsi : $\frac{N}{D_{mem}}$ (read)+ $\frac{N\alpha_{cpu}}{M_{cpu}}$ (calcul).

$$D_{mem} = \frac{N}{\frac{N}{M_{mem}} + \frac{N\alpha_{cpu}}{M_{cpu}}} \tag{5.3}$$

$$D_{mem} = min\left\{ \; M_{mem}; \frac{M_{mem}M_{cpu}}{M_{cpu}\alpha_{cpu}+M_{mem}} \; \right\} \tag{5.4}$$

Redistribution. Pour la Redistribution seule, on considère que le calcul est négligeable, les ressources utilisées sont la mémoire, les bus d'entrée/sortie et le réseau. Chaque donnée lue ou écrite pour un autre nœud, est écrit en mémoire puis envoyée au réseau, elle passe deux fois par la mémoire et ainsi deux fois par le bus d'entrées/sorties : $D_{mem} = 2D_r$,$D_{io} = 2D_r$, $D_{res} = 2D_r$.

Comme toute la puissance est disponible au moins une de ces ressources est saturée :

$$D_r = min\left\{ \; M_r; \frac{M_{mem}}{2}; D_{res}; \frac{D_{io}}{2} \; \right\} \tag{5.5}$$

Redistribution avec l'accès distant direct. Les données sont redistribuées directement entre les disques, sans transiter par la mémoire. Il y a un bénéfice de près de 100% :

$$D_r = min\left\{ \ M_r; D_{res} \ \right\} \tag{5.6}$$

Performances

Sans accès distant direct. Dans ce cas, la bande passante mémoire est partagée par le Filtre et la Redistribution.

$$D_r = min\left\{ \ M_r; \frac{1}{2}\left(M_{mem} - \frac{M_{mem}M_{cpu}}{M_{cpu}\alpha_{cpu}+M_{mem}}\right); M_{res}; \frac{M_{io}}{2} \ \right\} \tag{5.7}$$

Dans la majorité des architectures, la bande passante du bus d'entrées/sorties est très inférieure à celle du bus mémoire. De ce fait, les entrées/sorties sont prioritaires par rapport au processeur pour l'accès au bus mémoire. On a ainsi $M_{io} << M_{mem}$.

$$D_r = min\left\{ \ M_r; M_{res}; \frac{M_{io}}{2} \ \right\} \tag{5.8}$$

La formule montre que le débit de la Redistribution n'est pas affecté par la présence du Filtre. C'est le débit de ce dernier qui montre le recouvrement de ces deux applications.

$$D_{cpu} = min\left\{ \ M_{cpu}; \frac{M_{mem}-2D_r}{\alpha_{cpu}} \ \right\} \tag{5.9}$$

Le débit du Filtre avec Redistribution est inférieur à celui sans. Ceci montre que le recouvrement de ces deux applications n'est que partiel, car elles utilisent toutes deux la mémoire.

Avec accès distant direct. La Redistribution n'utilise plus la mémoire. Il n'y a plus de concurrence d'accès aux ressources. Le Filtre n'est plus ralenti et le recouvrement est complet. Les performances pour deux applications sont identiques aux performances dans le cas isolé.

5.5.2 Expérimentations

Les expérimentations ont été effectuées sur deux nœuds Alpha interconnectés par des cartes Myrinet. Comme nous n'avions pas assez de disques pour saturer le bus d'entrées/sorties, nous les avons simulés par une carte graphique. Cette carte peut lire et écrire depuis le bus d'entrées/sorties comme le ferait un disque. Le système est architecturé autour d'un bus mémoire de débit d'au plus 340Mo/s (M_{mem}) et d'un bus d'entrées/sorties (PCI) d'au plus 130Mo/s (M_{io}). Le réseau est constitué de cartes Myrinet de débit de 100Mo/s (M_{res}). La carte graphique est une Matrox G200 de débit d'au plus 80Mo/s.

Implémentation de l'accès distant direct

Pour implémenter l'accès distant direct, nous avons modifié la librairie de communication de Myrinet GM([64]) pour pouvoir utiliser la mémoire des cartes graphiques (*framebuffer*). Avec l'accès distant direct, les données passent directement de la carte graphique à la carte Myrinet, qui les envoie à l'autre carte Myrinet. Le contrôle de flot est fait par scrutation du processeur, cette action étant peut coûteuse en cycles processeur.

Sans l'accès distant direct, les données passent de la carte graphique à la mémoire, puis sont copiées vers la carte Myrinet qui les transmet à l'autres.

Résultats expérimentaux

Le Filtre et la Redistribution ont été testés de manière concurrente avec les deux stratégies : avec et sans l'accès distant direct. Nous avons observé, dans le cas sans accès distant direct, que la bande passante du bus d'entrées/sorties attribuée à la Redistribution est fixe, qu'elle soit exécutée avec le Filtre ou seul. Ce résultat s'explique par l'arbitrage du pont d'entrées/sorties, qui donne un accès prioritaire aux requêtes venant du bus d'entrées/sorties sur celles venant du processeur.

Dans la première expérience nous faisons varier le nombre d'opérations pour chaque donnée accédée par le Filtre, c'est-à-dire le besoin de bande passante mémoire du Filtre et nous mesurons la bande passante mémoire effective utilisée par le Filtre. Les mesures correspondent aux prédictions du modèle : pour une forte demande de bande passante mémoire, l'utilisation de l'accès distant direct fait gagner 46Mo/s au Filtre (libéré par la Redistribution).

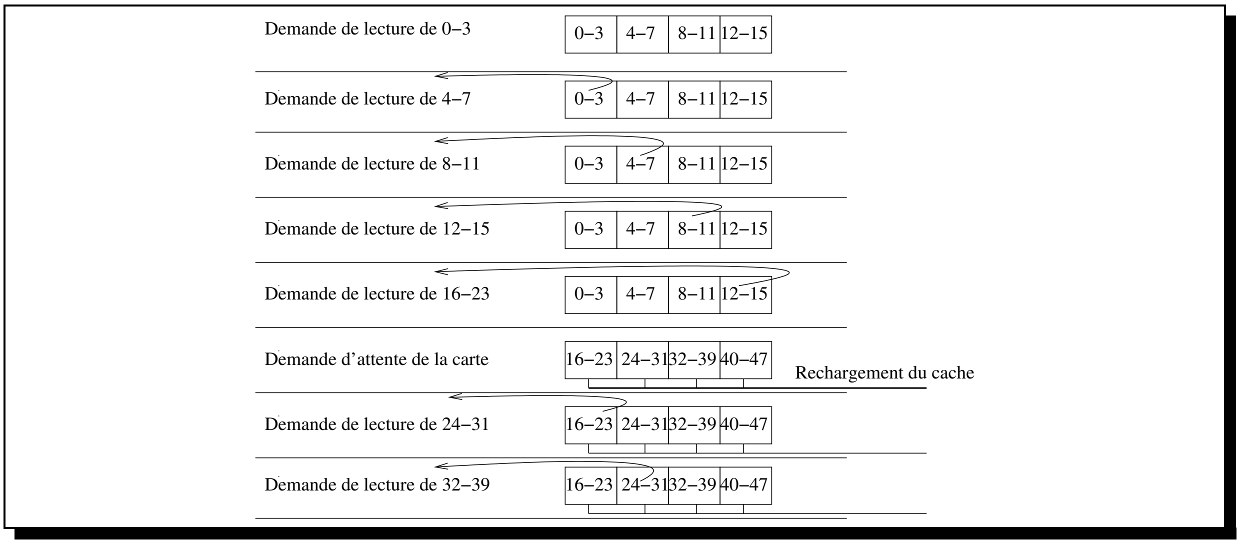


FIG. 5.7 – Le cache de la carte graphique pour l'accès au framebuffer

Cette utilisation correspond à une utilisation de 81Mo/s de bande passante du bus PCI, alors que sa bande passante maximale théorique est de 130Mo/s. L'explication est la présence d'un cache sur la carte graphique. En effet, lorsque qu'une lecture ou une écriture du framebuffer est demandée (cf [48]), la carte sauvegarde le cache précédent et charge un cache de 4 éléments de 4 octets consécutifs commencant à l'adresse demandée. Cette opération prend deux cycles de bus de plus, les 3 éléments suivants étant accédés en un cycle (voir fig. 5.7). Il faut ainsi 6 cycles pour lire 4 éléments, il y a ainsi 33% du débit du bus qui est perdu, soit un maximum de 82Mo/s.

Dans une seconde expérience, nous faisons varier la bande passante du disque (Figure 5.8) et nous observons la bande passante mémoire du Filtre. Nous observons que l'utilisation de l'accès distant direct rend le Filtre et la Redistribution indépendants et que la bande passante de la Redistribution passe d'au plus 46Mo/s à au plus 81Mo/s. Nous notons que la bande passante mémoire disponible pour le filtre néanmoins légerement réduite lors de l'accés distant direct, cela vient du fait que la resistribution demande de faire une lecture distante (avec l'accés distant), puis de sauver les données sur le disque local, ce qui utilise la mémoire.

Conclusion

Nous avons présenté les intérêts théoriques de la technique d'accès distant aux disques sans passage par la mémoire distante. Puis nous avons validé ces résultats sur une simulation. Nous proposons

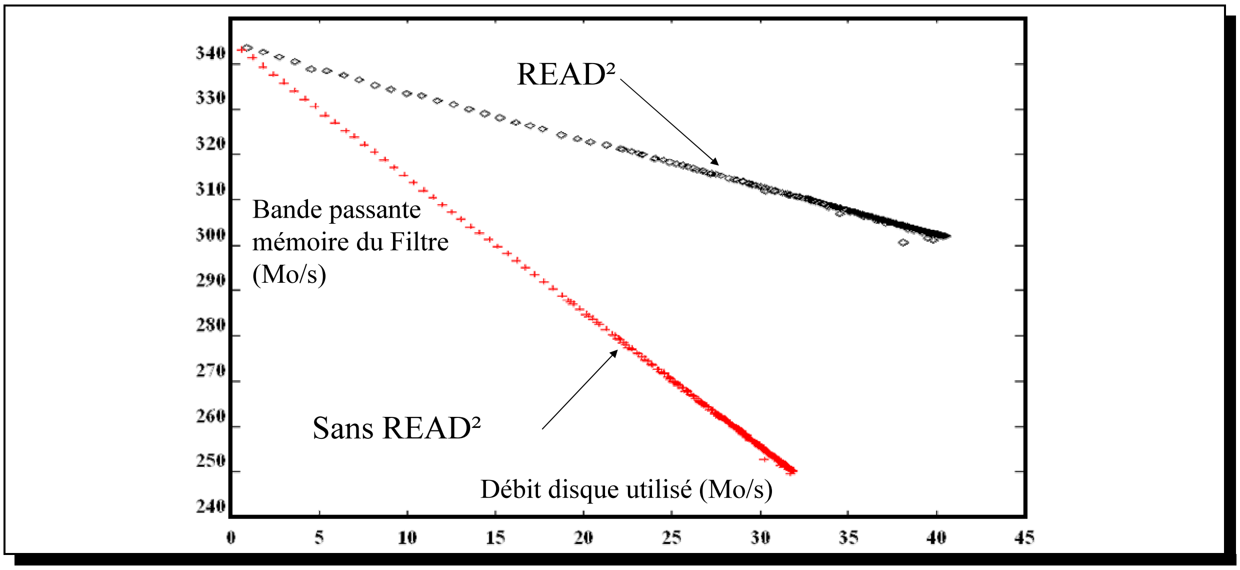


FIG. 5.8 – Interaction de la bande passante d'entrées/sorties et de la bande passante mémoire

maintenant de raffiner cette technique pour décharger le processeur du disque distant et de faire une implémentation accessible aux applications usuelles.

Chapitre 6

READ^2 : une implémentation de l'accès rapide aux disques distants

Sommaire

Dans les chapitres précédents nous avons présenté les composants mis en œuvre pour le transfert de données d'un disque distant, puis nous avons proposé une méthode pour améliorer les performances usuelles. Dans ce chapitre nous proposons une implémentation effective et performante de cette technique, appelés READ^2.

Quelques tentatives ont été faites pour implémenter l'accès direct aux disques distants tels que OPIOM [13], Maierhofer [62] ou DREAD [35]. READ^2 se distingue de ces précédentes approches par le chemin de contrôle des requêtes d'entrées/sorties et la localisation du pilote de disques distant. Notre stratégie qui embarque le pilote dans la carte de communication est inhabituelle, mais permet des gains de performance parce qu'elle décharge entièrement le processeur hôte du disque de toute intervention.

6.1 Stratégies d'implémentation

Les caractéristiques communes des différentes stratégies d'implantation considérées dans cette section concernent le chemin des données : lorsqu'un client lit des données du disque d'un serveur distant, les données vont du disque à la carte réseau du serveur et sont transférées au client. Pour une écriture, les données vont du client à la carte réseau du serveur et de cette carte au disque.

6.1.1 Pilote serveur (pilote dans le noyau du nœud serveur)

Habituellement, le serveur lit les données du disque vers les tampons du noyau du système d'exploitation, ces tampons se trouvent en mémoire centrale. Néanmoins, l'accès direct implique d'écrire les données directement dans la carte réseau.

Une première implémentation consiste à utiliser des tampons, qui ne sont pas dans la mémoire centrale mais directement dans la mémoire de la carte réseau. Ainsi, le contrôle des disques reste au niveau du noyau du serveur.

Pour chaque requête de lecture distante d'un client, le noyau du serveur est informé qu'une donnée doit être lue. Il programme le disque local pour lire les données requises.

Comme les tampons du noyau sont dans la carte réseau, le contrôleur du disque écrit les données directement dans la carte réseau. Une fois ces données écrites dans la carte réseau, le système demande l'envoi de ses données au client.

Cette stratégie est celle utilisée dans OPIOM [37], un travail proche du nôtre qui est conçu pour la vidéo à la demande (VOD) sur grappe à base de Myrinet. Cette stratégie est relativement facile à implémenter, mais le principal problème est l'utilisation importante des ressources du processeur du serveur.

6.1.2 Pilote avec accès distant aux ports d'E/S (pilote dans le noyau du client)

Une seconde stratégie consiste en la réutilisation du pilote de disques et la modification de manière à accéder au contrôleur distant. Cette stratégie peut être réalisée grâce à l'ajout de primitives d'accès distant aux ports d'entrées/sorties. La carte réseau est ainsi transformée en pont entre les bus PCI des machines (comme le pont sebring [78] ou SCI [11]).

Le travail du pilote de disques consiste à construire une structure de requête de lecture ou d'écriture en mémoire et à programmer le contrôleur par les ports d'E/S. Le contrôleur lit alors la requête et l'exécute. Une fois la requête achevée, le contrôleur écrit en mémoire son état et les données lues si c'était une lecture. Une solution pour contrôler le disque distant par le client est de :

- stocker la structure de requête dans le nœud distant (dans la mémoire de sa carte réseau) pour que le contrôleur puisse y accéder ;
- programmer et démarrer le contrôleur distant grâce à la carte réseau distante ;
- dès que la requête est achevée, les informations d'état et les éventuelles données sont écrites dans la mémoire de la carte réseau qui les renvoie au client.

Le second point requiert que le client accède des ports d'E/S distant (cf. 5.1), qui ne sont pas accessibles par la carte réseau (elle ne peut accéder qu'à la mémoire). Heureusement, les ports d'E/S sont souvent projetés en mémoire, le contrôleur distant peut ainsi être programmé par des accès mémoires distants. Cette stratégie réutilise le pilote Linux, elle est donc simple à adapter pour divers contrôleurs de disques, c'est la stratégie qui a été implémenté dans le pilote de disques distant *DREAD*[35]. Cette implémentation utilisant les cartes SCI qui projette la mémoire distante dans l'espace d'adressage local. Néanmoins, cette stratégie nécessite de nombreux petits messages pour l'accès aux ports d'E/S. Dans le cas d'une carte Myrinet, cette stratégie nécessite l'ajout de primitives de lectures/écritures distantes et l'utilisation de ces primitives dans le pilote Linux. Un autre problème est le mécanisme d'exclusion mutuelle introduit pour supporter plusieurs clients. Ce mécanisme n'est pas implémenté dans *DREAD* qui ne permet qu'un seul client par contrôleur.

6.1.3 Pilote dans la carte réseau du nœud serveur

Une alternative aux stratégies précédentes consiste à placer le pilote dans la carte réseau du serveur. Ainsi, il n'y plus que deux messages qui traversent le réseau : la requête et les données du disque.

Cette alternative a l'avantage combiné des deux précédentes stratégies : le processeur du nœud distant (et donc son noyau) n'est pas utilisé par le client, les accès concurrents sont traités. Le problème est que cette stratégie est difficile à implémenter car elle dépend d'un contrôleur spécifique et d'une carte réseau spécifique. Nous avons choisi une des deux plus courantes classes de contrôleurs SCSI : les contrôleurs compatibles LSI 53c8xx (l'autre classe étant celle d'Adaptec).

6.2 Pilote READ^2

Nous avons implémenté cette dernière stratégie avec GM, le logiciel de communication des cartes Myrinet : le pilote READ^2 fait interagir le firmware de GM et le contrôleur SCSI.

6.2.1 Courte description de GM

GM est une librairie efficace de communication par passage de messages qui utilise les cartes Myrinet pour obtenir une faible latence et une grande bande passante. GM s'exécute en mode utilisateur (aucun appel système) et permet de faire peu de copies mémoire. GM reprend les optimisations de *BIP* [70] en lui ajoutant la fiabilité du transport des messages. GM utilise des tampons verrouillés en mémoire principale (non paginable). Les demandes d'envois de messages ou de libération de tampon se font par l'écriture par l'application de commande dans la mémoire de la carte Myrinet (fig. 6.1). De plus, une fois les tampons en mémoire centrale verrouillé, la carte Myrinet n'a plus besoin du système pour l'envoi et la réception des messages. GM est constitué d'un firmware pour carte Myrinet, d'un pilote de périphérique et d'une librairie en mode utilisateur. GM donne plusieurs ports pour chaque nœud (comme TCP) pour permettre à plusieurs applications de partager GM sur la même machine. GM permet un acheminement fiable et ordonné pour chaque port.

6.2.2 READ^2 et le firmware GM

Le firmware des cartes Myrinet est appelé MCP (Myrinet Control Program), il est constitué de quatre processus présentés dans la figure 6.2 : un processus d'envoi, un processus d'envoi DMA, un processus de réception et un processus de réception DMA.

Les message reçu par la carte Myrinet sont traités par le processus de réception qui s'assure de leur intégrité (fig. 6.2,①), il sont alors écrit en mémoire par le processus de réception DMA (②,③) en utilisant l'information du port destination.

Nous avons réservé un port pour READ^2. Les messages entrant à destination de ce port sont interceptés au niveau du processus de réception DMA (④, ⑤). Les blocs à lire (ou écrire) qui sont ensuite envoyés au contrôleur SCSI (⑥). Lorsque les données sont prêtes (pour la lecture), le contrôleur SCSI les écrit en mémoire (⑦) et READ^2 les envoie par des fonctions de communication standard de GM (⑧, ⑨).

Il faut remarquer que nous réutilisons le processus d'envoi et de réception standard de GM ; ces processus régulent le contrôle de flot et s'assure de l'intégrité des données. Ainsi, le contrôle de flot de *read*^{2} pour l'écriture est directement hérité de celui de GM, dès que le message comprenant les données et la commande d'écriture est accepté, l'écriture sera effectuée. Dans le cas ou les tampons sont pleins, READ^2 demande à GM de réessayer. Après quelques essais infructueux, GM abandonne et en informe l'expéditeur. Pour les opérations de lecture, le contrôle est plus complexe : lorsque READ^2 a lu des données avec succès il tente de les envoyer au client, mais comme dans le cas précédent, le message peut ne pas arriver, dans ce cas le client n'est pas averti. Pour que le client n'attende pas indéfiniment ces données, nous avons ajouté une primitive qui permet l'abandon d'une lecture. Ainsi, si le client ne reçoit pas une donnée avant un certain temps, il peut envoyer une commande d'abandons de toutes les lectures en attentes.

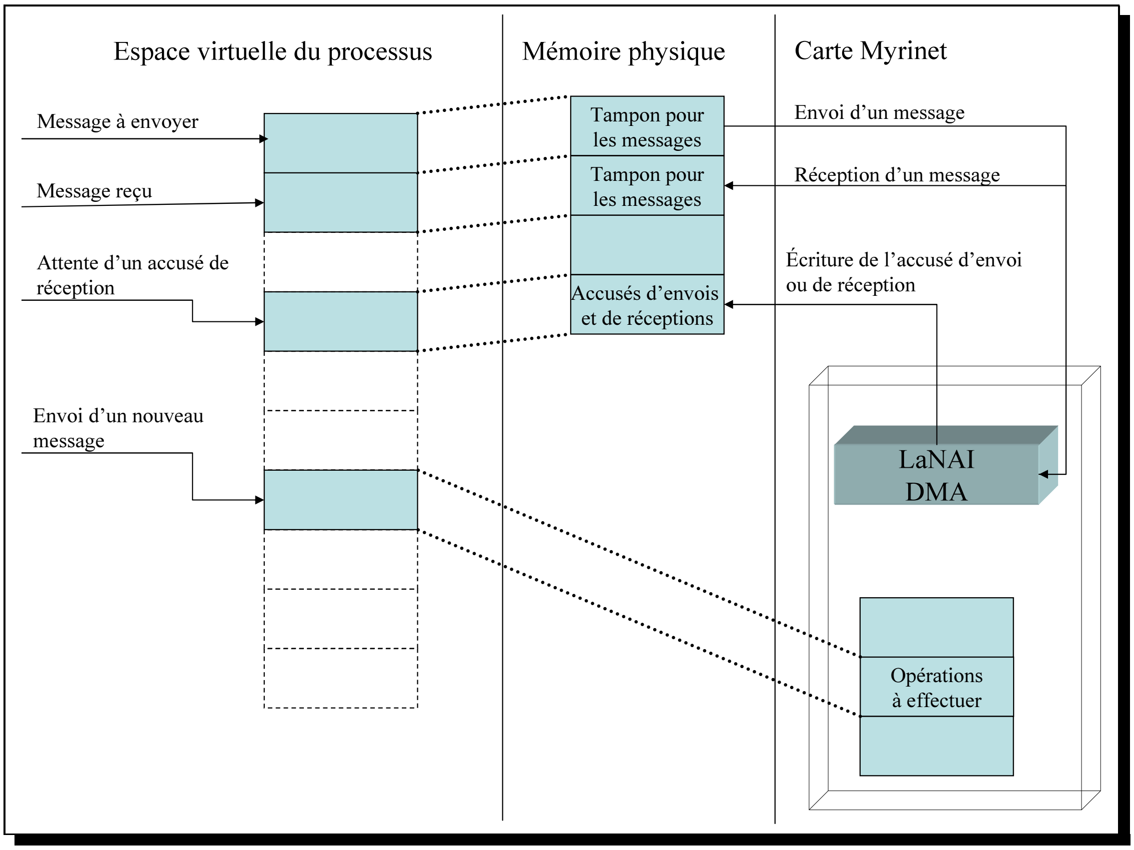


FIG. 6.1 – Envoi et réception de messages avec GM

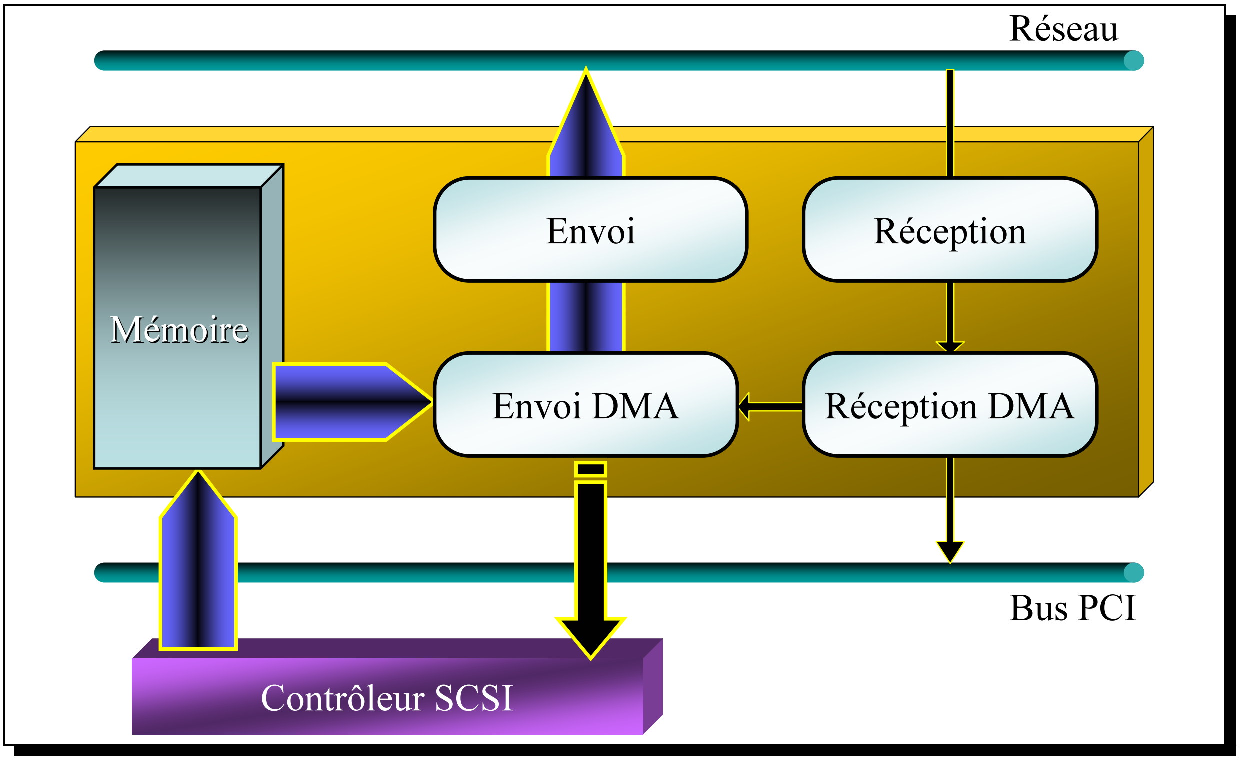


FIG. 6.2 – Envoyer une commande avec GM

6.2.3 Contrôleur SCSI et la carte réseau Myrinet

Décrivons maintenant, la relation entre le contrôleur SCSI [61] et la carte Myrinet (Figure 6.3). La carte Myrinet peut accéder à la mémoire principale (et a toute autre mémoire projeté par un périphérique) avec sa DMA. De plus, elle possède une mémoire embarquée, ainsi qu'un processeur du nom de Lanaï. READ^2 utilise une partie de la mémoire de la carte Myrinet pour stocker ses tampons de disques, et une autre partie pour stocker le script SCSI. Le script SCSI contient les instructions qui seront exécutées par le processeur du contrôleur SCSI.

Dans la première implémentation, la carte Myrinet commence l'exécution d'une requête par l'accès aux ports d'E/S du contrôleur SCSI (figure 6.3, ①). Ces ports d'E/S démarrent le processeur SCSI (②). Puis il lit la requête l'envoi au disque (③). Il reçoit ensuite la réponse du disque et l'écrit dans la mémoire de la carte Myrinet (③,④). Finalement, le Lanai scrute les ports d'E/S pour attendre la fin de l'écriture de la réponse dans la mémoire de la carte Myrinet (⑤.

Néanmoins, l'utilisation du DMA pour l'accès aux ports d'E/S a deux inconvénients : le DMA de la carte Myrinet a une grande latence car elle est conçu pour de long messages et l'usage du DMA interrompt la réception et l'envoi de messages GM standard. Pour supprimer l'accès aux ports d'E/S par la DMA, nous avons modifié le script SCSI du contrôleur pour qu'il scrute une variable (sémaphore) dans la mémoire de la carte Myrinet afin d'attendre une requête (⑥). La processeur de la carte Myrinet modifie cette variable pour envoyer une nouvelle requête (⑦), et la scrute pour attendre l'achèvement d'une requête.

6.3 Utilisation de READ^2

Notre travail a consisté à définir un ensemble de nouvelles fonctions basées sur GM pour l'accès au disque distant en mode utilisateur. De plus, nous avons développé un nouveau périphérique de blocs

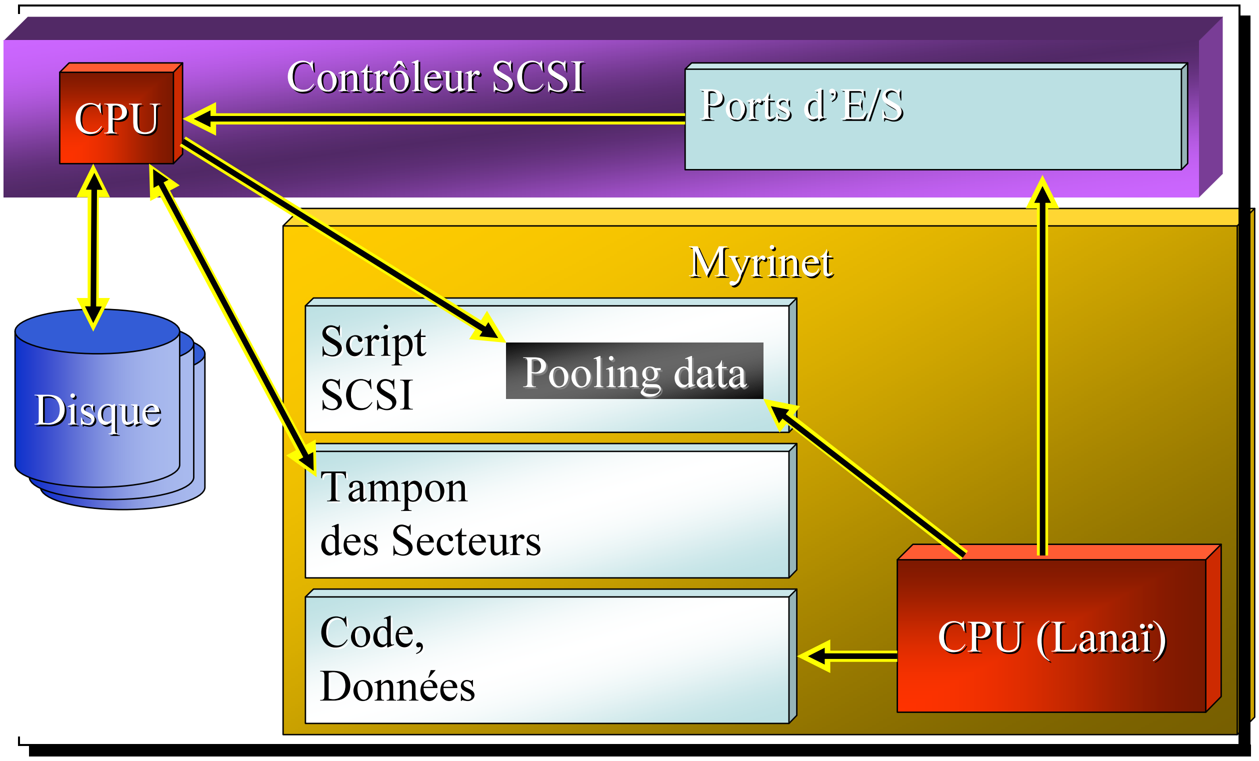


FIG. 6.3 – READ² et le contrôleur SCSI

pour les systèmes de fichiers. Ces deux interfaces ne sont pas interopérables, parce que le périphérique de blocs utilise les caches de Linux alors que le mode utilisateur n'en est pas informé, il n'y a donc pas de cohérence.

6.3.1 Fonctions en mode utilisateur

Le disque distant est accédé comme un unique fichier : il n'y a pas de système de fichier. Deux fonctions principales permettent l'accès à READ² :

void read2_read$(node, sector, nb, buffer)$ **:** lit nb secteurs du disque distant $node$ à partir du secteur $sector$. Cette fonction est asynchrone, `read2_wait` doit être utilise pour attendre l'achèvement de la requête.

int read2_wait$(node, sector, nb, buffer)$ **:** attend la fin d'une lecture et donne la dernière requête finie.

Ces fonctions sont basées sur les fonctions d'envoi et de réception de messages de GM, elles créent un message contenant une requête pour READ² et l'envoi au destinateur sur le port de READ².

6.3.2 Périphérique de blocs (Block device)

L'interface précédente est conçue pour de rapides entrées/sorties brutes, nous avons aussi conçu un périphérique de blocs pour supporter les systèmes de fichiers : nous pouvons ainsi monter le système de fichiers standard ext2 sur READ².

Ainsi, lorsque l'application lit des données, le système de fichiers recherche les blocs correspondant dans son cache ; s'il n'y sont pas, une requête est transmise au périphérique de blocs READ² qui utilise les fonctions utilisateur pour transmettre la demande au disque distant.

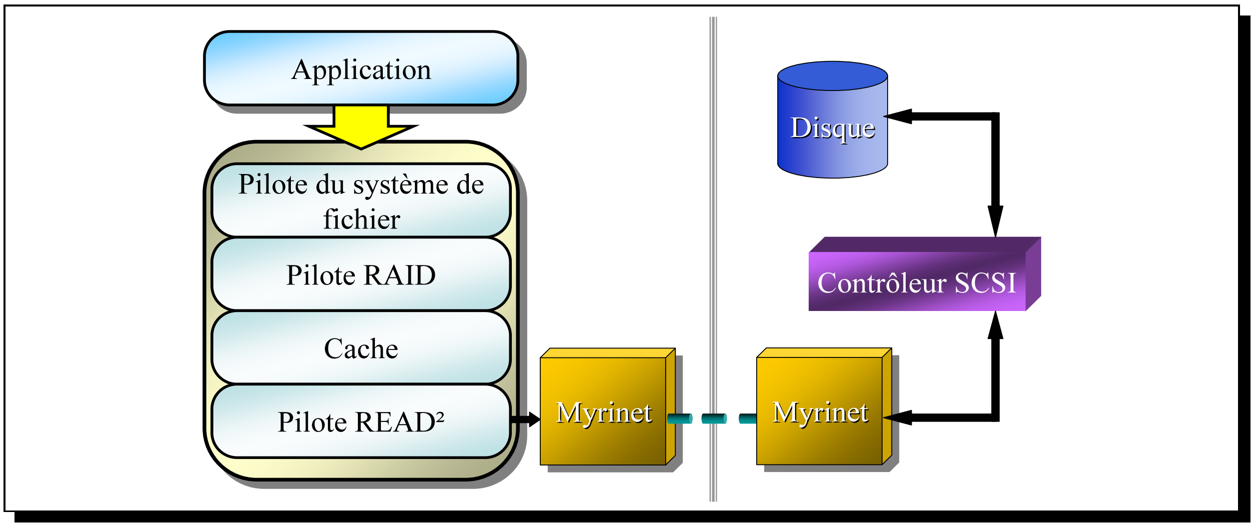


FIG. 6.4 – Le pilote READ^2 dans le noyau de Linux

6.4 Expérimentation

READ^2 est conçu pour être le composant de base d'un système de fichiers haute performance pour grappes de stations. PVFS et GFS sont les principaux systèmes de fichiers pour grappes. Un système de fichiers parallèles avec une architecture plus orienté système est *RAID-x*[47] : il doit distribuer les données entre les disques, assurer la tolérance aux pannes, la gestion des méta données et la cohérence des accès concurrents. Pourtant, toutes ces fonctionnalités sont basées sur un mécanisme primitif d'accès aux données distantes. Par exemple, PVFS utilise un service d'E/S spécifique (IOD) sur chaque nœud et GFS utilise une version spécialisée de NBD (network block device). NBD est un mécanisme standard de Linux pour accéder au disque distant à travers un périphérique de blocs local, comme READ^2.

Une méthode triviale pour constituer un système primitif de fichiers parallèles consiste en l'utilisation combinée du pilote RAID0 de Linux et du système de fichiers ext2. Le mécanisme d'accès au disque distant pouvant être NBD ou READ^2. La première expérimentation est de comparer les performances de ces deux systèmes primitifs de fichiers parallèles. Il y a deux mesures : l'accès simple (un nœud accède aux disques de tous les autres) et l'accès concurrent (tous les nœuds accèdent à tous les disques). Malheureusement, NBD ne supporte pas les accès concurrents, nous avons dû choisir PVFS pour cette comparaison. NBD et PVFS communiquent par l'interface socket. Les sockets utilisent la pile TCP/IP qui utilise le pilote GM. GM fournie une interface socket efficace qui outrepasse la pile TCP/IP, néanmoins cette interface n'est disponible qu'en mode noyau, elle n'a donc pas pu être utilisée avec NBD et PVFS. Les paragraphes suivants présentent les tests pratiqués avec une grappe de 8 nœuds (bi-Athlon). Les nœuds sont interconnectés par des cartes Myrinet (2 Gb/s). Chaque nœud dispose d'un disque Atlas 10KII Quantum SCSI-LVD avec un débit crête de 40Mo/s.

6.4.1 Lecture d'un fichier réparti sur plusieurs nœuds

Les tests consistent en la lecture par un nœud de fichiers de 1Go stockés avec PVFS ou l'un de nos deux systèmes primitifs de fichiers parallèles. La taille du fichier est choisie suffisamment grande pour ne pas avoir les effets de cache. Nous avons mesuré le débit de lecture par rapport au nombre de disques distants.

La figure 6.5 montre les résultats. Le débit disque augmente avec l'ajout de disques jusqu'à trois disques, avec plus de 3 disques il n'y a plus de gain à cause des limitations du réseau. En effet, la taille des messages des cartes Myrinet est de 9Ko pour les paquets Ethernet (ainsi que pour READ^2), or les

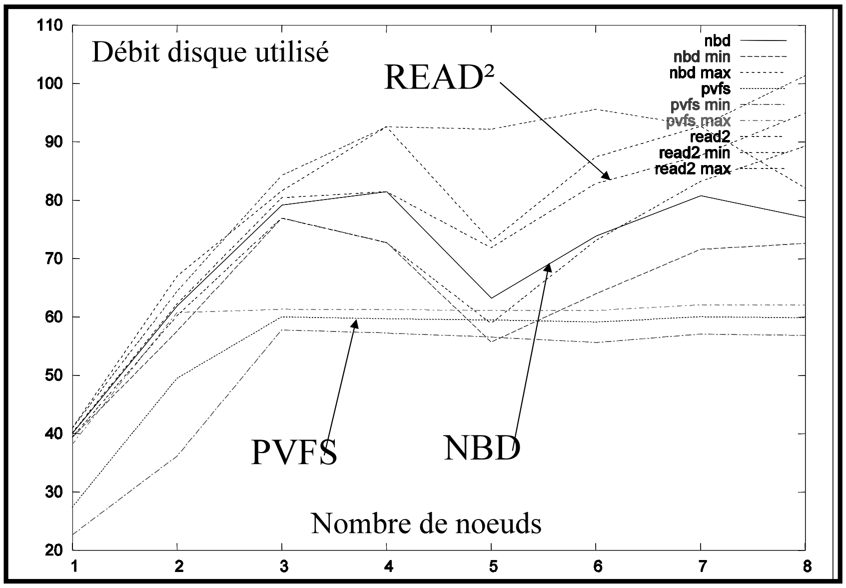


FIG. 6.5 – Lecture par un nœud d'un fichier répartie sur plusieurs nœuds

cartes Myrinet ne peuvent dépasser 110Mo/s pour une telle taille de messages. Comme attendu, READ^2 a de meilleures performances que PVFS ou NBD, parce qu'il utilise moins de ressources système. PVFS étant celui qui utilise le plus de ressource système, ses performances sont les plus mauvaises.

6.4.2 Accès concurrent

Ce test consiste en la lecture concurrente de n fichiers de 1Go par n nœuds, avec soit PVFS, soit le système primitif de fichiers parallèles basé sur READ^2. Nous avons mesuré le débit agrégé suivant le nombre de disques. La figure 6.6 montre les résultats, pour des raisons similaires à l'expérience précédente, PVFS a de moins bonne performance que READ^2. Néanmoins, le débit de READ^2 est limité par les contentions réseau, du bus SCSI et la taille des tampons réduits de READ^2. L'utilisation des techniques SCSI dites de QUEUEING permettra d'envoyer une nouvelle requête avant la fin de la précédente, le disque pourra réordonner les requêtes et réduira les temps mort entre la fin d'une requête et le début de la suivante.

6.5 Implémentation avec GFS

Nous venons d'utiliser un système de fichiers sans écriture cohérente. Dans ce paragraphe nous allons décrire l'implémentation de READ^2 avec le système de fichiers parallèle GFS[68].

Ce système de fichiers a été conçu pour la gestion de disques partagés utilisant des disques *Fibrechannel* (évolution du SCSI avec 128 périphériques). Ces disques sont tous connectés au même « bus », et sont accessibles par tous les nœuds sans distinction. Les problèmes principaux sont liés à la cohérence des caches. En effet, plusieurs nœuds peuvent lire le même secteur, le modifier et l'écrire. Si par exemple, deux nœuds ajoutent un fichier dans un répertoire, lors de l'écriture des caches, seul la dernière écriture sera visible, et un fichier sera perdu.

Afin de résoudre ce problème, GFS introduit la notion de verrouillage, c'est-à-dire qu'à un moment donné seul un nœud est propriétaire d'un secteur. Pour cela, GFS utilise une commande SCSI [69] qui permet d'utiliser des sémaphores stockées dans une région mémoire de chaque disque.

Ainsi, lorsqu'un disque veut lire un secteur, il demande le sémaphore correspondant à la région, et lit le secteur si la région est disponible. Dans le cas de succès, il peut modifier le secteur et le réécrire en

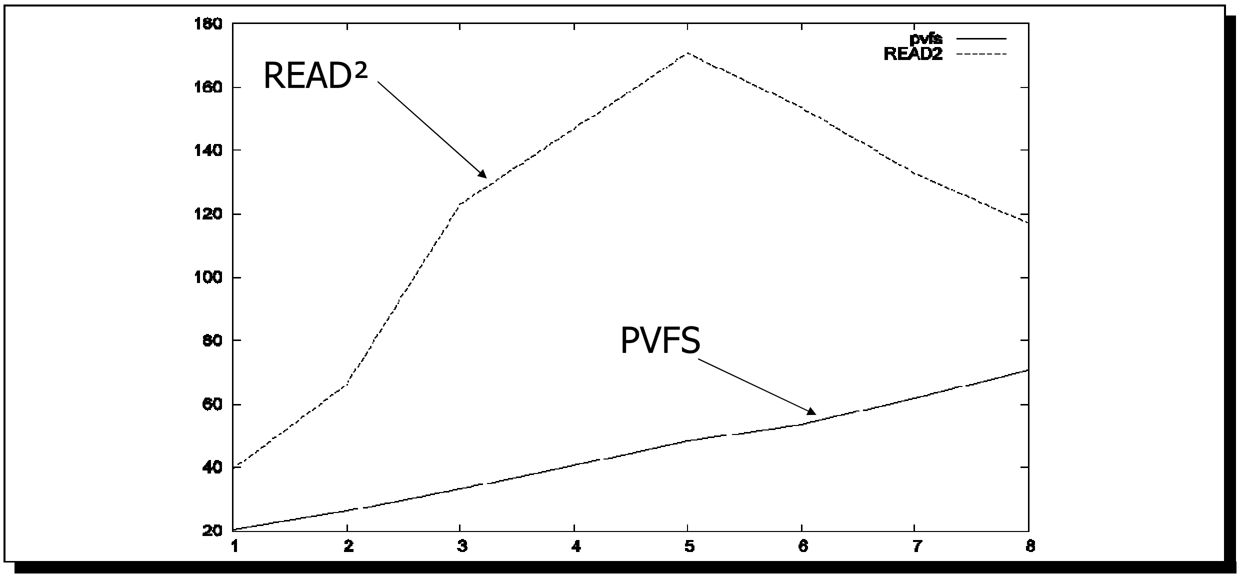


FIG. 6.6 – Accès concurrent aux fichiers

lâchant le sémaphore. Dans le cas d'échec, le secteur peut être lu et stocké dans les tampons du nœud, mais ces tampons seront protégés en écriture : pour une écriture sur ces tampons, le système devra redemander et obtenir le sémaphore, puis relire les secteurs. Les tampons protégés en lectures seront régulièrement réactualisés, afin de contenir des données valides.

GFS fonctionne originellement avec des disques *FibreChannel*. Néanmoins, il ne dépend que d'une commande permettant à tous les nœuds d'accéder à un certain nombre de sémaphores. Ainsi, il dispose de modules lui permettant d'accéder a d'autres types de disques *NAD* tel que le iSCSI. Un module particulier permet d'utiliser les disques des autres nœuds : *gnbd* qui est une version de NBD avec le support des sémaphores.

Nous avons utilisé le module GNBD, que nous avons modifié pour lire et écrire avec READ^2 et nous avons utilisé des sémaphores que nous avons mis dans les cartes Myrinet.

Nous avons utilisé GFS/GNBD/IP/GM et GFS/READ^2 pour la lecture concurrente de fichiers en parallèle sur 4 nœuds ainsi que d'un fichier par un nœuds. Nous obtenons des résultats sensiblement égaux entre GNBD et READ^2. Néanmoins, le grand intérêt de READ^2 est la non interférence avec le système. Nous avons donc fait tourner une application de calcul qui utilise intensivement la mémoire et le processeur.

Nous obtenons les mêmes performances que le disque du nœud soit lu à distance ou non avec *read*^{2} (2393 secondes sans READ^2 et 2360 secondes avec). En effet, dans le cas de GNBD, nous avons une dégradation de performance, nous passons de 2393 secondes sans GNBD à 2039.

6.6 Conclusion

Nous venons de décrire une implémentation de READ^2 sur une carte Myrinet et présenté les premiers résultats. Cette implémentation est dépendante du contrôleur, mais une extension pourrait être l'utilisation des contrôleurs IDE. En effet, actuellement les contrôleurs IDE ne sont programmables que par des ports d'E/S, qui ne sont pas accessible par la carte Myrinet à cause de la limitation de son DMA. Néanmoins, la norme des contrôleurs IDE (*ATA/ATAPI Host Adapter Standard*) spécifie une nouvelle technique d'accès nommée *ADMA*[76]. Cette technique permet d'accéder à l'ensemble des registres du contrôleur par accès mémoire. Elle inclut une liste de commandes à exécuter, qui peut résider dans la mémoire de la carte Myrinet. De même le standard IDE inclut désormais le *queueing* [75], ce qui permet d’obtenir des performances similaire entre l’IDE ou le SCSI [80]. Nous pourrons ainsi implémenter READ^2 sur les disques les plus courants.

Conclusion

Les contributions de cette thèse portent sur la pagination et les entrées/sorties distantes hautes performances pour les applications traitant de grandes masses de données.

La pagination

Notre contribution est d'avoir défini un outil de gestion de la pagination en mode utilisateur, qui permet d'adapter celle-ci aux particularités des applications. A cette fin, nous avons présenté le fonctionnement de la pagination du point de vue système, puis nous avons présenté les politiques les plus courantes nécessaires à sa gestion par les systèmes d'exploitation. Nous avons ensuite montré l'insuffisance de ces techniques pour certaines applications.

Les systèmes d'exploitation tels que Linux ne permettant pas aux applications de contrôler leur pagination, nous avons conçu un outil proposant cette fonctionnalité sous Linux. Cet outil a nécessité l'écriture d'un module Linux nommé *MMUSSEL* et d'une librairie nommée *MMUM*. Une première application a été la définition d'une politique de pagination adaptée au problème de la factorisation LU. Nous avons ainsi exposé, pour cette application, une diminution de l'activité de pagination (défauts de page et évincements) allant jusqu'à 55%. Nous avons voulu aller plus loin et créer un gestionnaire de pagination complet adapté aux applications travaillant sur des flux. Ce gestionnaire a permis des gains en temps d'exécution de plus de 30% sur une application spécifique.

Malheureusement, notre outil ne permettait pas d'interagir harmonieusement avec le système d'exploitation. Nous l'avons donc amélioré par la déportation de la gestion de la pagination sur un processus séparé. Cette amélioration conduit à la gestion de la pagination des applications sans modifier leur code.

Grâce à cette nouvelle méthode de contrôle de la pagination, nous avons pu analyser les performances d'une application réelle : le solveur de matrice creuse *MUMPS*. Cette méthode a permis des gains de performance pouvant atteindre 30% en temps et un gain toujours supérieur à 30% en activité de pagination (défauts de page et évincements). Le contrôle de la pagination que nous avons définie fonctionne aussi pour la version parallèle de *MUMPS* et permet d'obtenir des gains de performances significatifs (jusqu'à 40%).

Le contrôle de la pagination pourra être utilisé pour l'ordonnancement parallèle (*gang-scheduling*). Il pourrait ainsi être possible d'utiliser notre gestionnaire de pagination avec des applications parallèles afin d'orienter l'ordonnancement parallèle des tâches pour prendre en compte la pagination de chaque nœuds, d'ailleurs nous travaillons actuellement sur l'intégration de ce *coscheduling* dans *MUMPS*.

Un développement futur pourra être la conception, à partir d'une analyse de la pagination des applications cibles, de gestionnaires spécifiques à certaines classes d'applications, comme par exemple le contrôleur de pagination pour les applications de gestion de flux au chapitre 2.2. Nous pourrons également utiliser l'outil *MMUM/MMUSSEL* pour développer un gestionnaire de pagination à distance, ou un gestionnaire permettant de limiter les entrées/sorties par la compression de la mémoire. Dans ce cas, seule les données de la page en cours d'accès serait décompressée, les autres données serait compressées et ce serait au gestionnaire de faire la compression et la décompression des données.

Ces futurs gestionnaires permettront d'affiner la connaissance des informations et des contrôles nécessaires sur la pagination tels que l'état global de la mémoire et des autres processus. Nous pourrons ainsi ajouter ces fonctionnalités à MUMM/MMUSSEL.

Les entrées/sorties distantes

Notre contribution est d'avoir écrit une bibliothèque d'accès efficace aux entrées/sorties distantes. Cette idée vient du constat que les entrées/sorties des grappes sont leurs facteurs le plus limitatifs pour le traitement de grandes masses de données. En effet, notre modélisation du comportement architectural des grappes de stations, dans le cas d'entrées/sorties intensives, nous a permis d'estimer ces limitations. Nous avons ensuite étudié l'impact sur ce modèle du passage direct des données du disque à la carte réseau. Cette technique présentant des gains théorique allant jusqu'à 50% de débit en plus avec une contention mémoire réduite d'autant, nous avons testé cette technique avec l'émulation d'un disque rapide qui nous a donné un résultat proche de l'estimation (44%).

Après les résultats probants de cette étude préliminaire, nous avons fait une implémentation de la technique de l'accès direct distant sous la forme d'un outil nommé READ^2. Pour cette implémentation, nous avons choisi d'embarquer le pilote du disque dans la carte réseau (Myrinet), déchargeant par la même occasion le processeur du nœud du disque. Afin de pouvoir dialoguer efficacement avec la carte réseau, ce choix a nécessité la modification du *firmware* du contrôleur de disques. Cette implémentation permet le contrôle du pilote de disques par des messages de la bibliothèque *GM*[64] sans la perturber. Ceci permet l'interopérabilité de READ^2 avec *GM*.

Afin de valider READ^2, nous avons créé un pilote de blocs Linux. Ce pilote a permis, avec le pilote *RAID* et *ext2* de Linux, de créer un système de fichiers primaire sans gestion de la cohérence de cache. Ce système de fichiers a été comparé avec un autre basé sur le pilote de disques distant Linux *NBD*. Il a ensuite été comparé au système de fichiers parallèle *PVFS*[57], et nous avons obtenu des gains de performances de l'ordre 50%.

Nous avons ensuite implémenté READ^2 avec le système de fichiers parallèle *GFS* [68] avec READ^2. Nous avons ainsi constaté que READ^2 n'utilise pas les ressources processeur et mémoire du nœud du disque contrôlé.

Ce dernier système de fichiers gère l'utilisation de multiples disques. Néanmoins, l'implémentation de READ^2 au sein même de la carte Myrinet, peut permettre de déporter certaines fonctions du système de fichiers parallèle. Ainsi, le *RAID* pourrait être géré par READ^2. De manière générale nous pourrions inclure des fonctionnalités plus spécifiques permettant d'améliorer des applications travaillant sur des données placées sur disque telles que le tri.

Une autre utilisation de READ^2, peut être l'utilisation avec l'outil *MMUM/MMUSSEL*. En effet, lorsque l'on veux envoyer des données dans une page qui a été évincée sur disque, on pourrait utiliser READ^2 pour directement écrire les données sur disque, sans avoir besoin de recharger la page en mémoire pour la réception des données. Or, les protocoles de passage de messages sans copie verrouillent des pages car ils ne savent pas gérer ce dernier cas. Nous pouvons ainsi imaginer des protocoles de passage de messages sans copie mémoire, sans page verrouillée.

Annexe A

Un travail connexe à la mémoire virtuelle : le cache du processeur

Sommaire

La mémoire virtuelle est un élément de la hiérarchie mémoire. Un autre élément est le cache du processeur. Ce cache souvent constitué de deux ou trois niveaux permet de stocker temporairement les données de la mémoire centrale. L'intérêt de ce cache est la différence de débit et de latence par rapport à la mémoire principale. En effet le débit de la mémoire ne dépasse pas les 6Go/s pour les dernières architectures (Pentium4 FSB 800Mhz), alors que les processeurs ont besoin de 10 fois plus (plus de 50Go/s pour les derniers processeurs). De plus, la latence est bien plus importante (plus de dix fois) ce qui ralentit les instructions qui utilisent la mémoire. Pour cela le cache permet, comme la mémoire virtuelle pour le stockage secondaire, de stocker une partie des données de la mémoire centrale.

Pour les mêmes raisons que pour la mémoire virtuelle, le cache est séparé en pages appelées lignes de cache, faisant chacune plusieurs octets (64 octets pour le Pentium). Lorsque l'on veut accéder à un des octets de la ligne de cache, il faut que la ligne entière soit chargée. Ainsi, pour écrire un octet dans une ligne de cache, il faut la charger entièrement. Si une application utilise peu le cache, elle se trouvera dans une position d'accès direct à la mémoire et sera ralentie par ce goulot d'étranglement.

Pour réduire ce problème, des algorithmes ont été conçus pour travailler sur des blocs de données de taille suffisamment faible pour entrer dans le cache et ainsi ne pas nécessiter d'accès mémoire pour traiter le bloc [30], mais seulement pour le charger et sauvegarder le résultat. C'est le cas de nombreuses implémentations de BLACS (élément de base pour le calcul matriciel).

Néanmoins, ces techniques se basent sur l'idée que les données que l'on utilise restent dans le cache et comme ces algorithmes ne communiquent pas directement avec le cache, ils ne peuvent être certains que les données utilisées restent dans le cache. De plus les applications sur des flux de données ou celles ne pouvant travailler sur des blocs contenus dans le cache peuvent être mal gérées par le cache, c'est-à-dire que leurs données seront conservées ou évincées de manière inappropriée.

A cette fin, de nombreuses applications se sont orientées vers le préchargement des données (dans le filtrage [36] ou dans le cas général [29]), de façon à limiter leurs temps de chargement depuis la mémoire. Ces applications se trouvent confrontés à la pollution du cache, c'est-à-dire a la présence de données inutiles en lieu et place de données utiles. Afin de limiter ces effets, nous proposons d'influencer la gestion du cache.

A.1 Contrôler le cache

Afin de mieux comprendre le fonctionnement du cache et son influence sur certaines applications, nous allons examiner sa politique de remplacement des données, c'est-à-dire comment le cache choisit les données à évincer. Comme pour la mémoire virtuelle, l'algorithme de remplacement est généralement la LRU, mais avec la différence qu'il est géré matériellement et non logiciellement. Néanmoins, les concepteurs de microprocesseurs ont donné la possibilité de choisir quelques autres stratégies. Ainsi, pour le Pentium, nous disposons d'un attribut dans la table de translation d'adresse qui permet d'associer une politique de gestion du cache (L1 et L2) différente pour chaque page mémoire. Il y a quatre politiques disponibles :

- Le Write Back : les données sont stockées dans le cache et écrites (si elles ont été modifiées) en mémoire lorsqu'elles doivent être évincées du cache. Toute lecture ou écriture d'une donnée non présente dans le cache se traduit par l'allocation d'un emplacement dans le cache et la lecture de la donnée depuis la mémoire centrale vers le cache avant l'opération de lecture ou écriture. La politique d'évincement des pages est la LRU. C'est la technique par défaut et c'est en général la plus rapide.
- Le Write Throught : il se distingue du Write Back par le fait que toute donnée écrite est directement écrite en mémoire et une donnée à écrire n'est pas mise dans le cache si elle n'a pas été lue auparavant : c'est un cache en lecture. Comme les données sont directement écrites en mémoire, ces opérations sont plus lentes. Ceci explique pourquoi cette technique est en général plus lente que le Write Back.
- Le non caché : les données ne sont pas stockées dans le cache. Toutes les écritures et les lectures vont directement vers la mémoire. Cette technique conduit généralement aux performances les plus faible.
- Le Write Combinning : Lors d'une écriture, quelques tampons sont réservés. Lorsqu'ils sont remplis les données sont transférées en mémoire et les tampons libérés. Ceci permet de ne pas avoir à lire préalablement les données depuis la mémoire, et permet de ne pas conserver les données dans le cache si elles ne sont pas utiles. Cette technique est plus efficace pour les opérations sur les flux tels que la vidéo. Elle est néanmoins à part car elle n'assure pas la cohérence avec les autres caches et la mémoire principale du système.

A.2 Implémentation

Nous avons implémenté sur Pentium une API qui permet depuis le mode utilisateur de changer la politique de gestion de cache associé à une page. Cette implémentation utilise un module qui permet de changer les attributs PCD, PWT et PAT des pages qui contrôlent la gestion de leur données dans le cache.

```
setcache(void * add,size_t size,int prot)
```

- add : adresse de début de la région
- size : nombre d'octets de la région
- prot : politique de gestion du cache qui sera affectée à cette région

A.3 Résultat

Nous avons fait plusieurs tests sur un processeur avec 16Ko de cache pour valider cette API et les gains théoriques. Pour le premier test, nous utilisons deux tableaux et nous changerons la politique de gestion du cache du premier (tab1), le second restera en write back.

```
For(l=0;l<1024*128*4;l++)

tab1[l]=tab1[l]+tab2[l &4095];
```

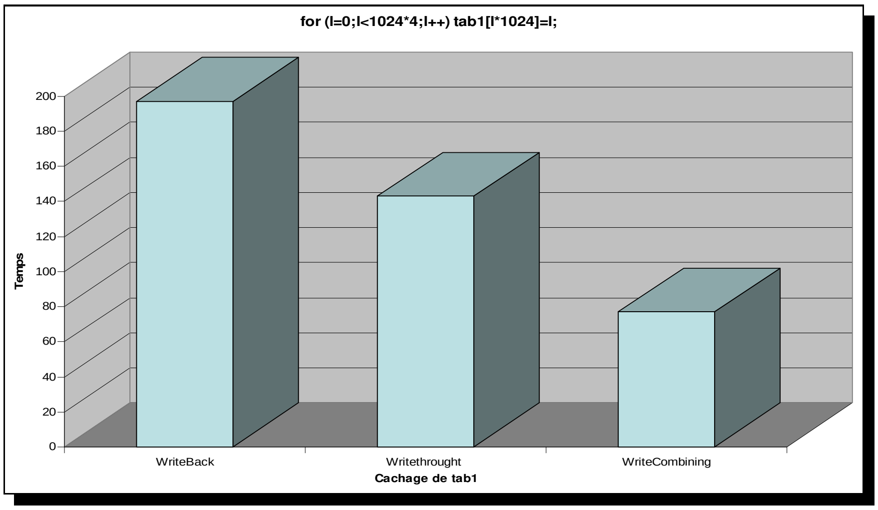


FIG. A.1 – Test sur un flux et un tableau contenu dans le cache

Dans ce test, avec le write back et le write throught, le tableau tab2 restera dans le cache car il est constamment utilisé et il est plus petit que le cache. Le tableau tab1 ne restera que temporairement dans le cache, car il n'est utilisé qu'une fois. Ainsi, le gain du write throught par rapport au write back vient du fait que les données sont immédiatement écrites alors que le write back attend que le cache soit plein pour commencer les écritures. Le Write Combinning a les mêmes gains que le Write Throught, mais il profite d'une technique d'écriture des données optimisée.

Pour le second test, nous remplirons un tableau avec le contenu d'un plus petit qui est de taille inférieure à celle du cache et qui sera en Write Back.

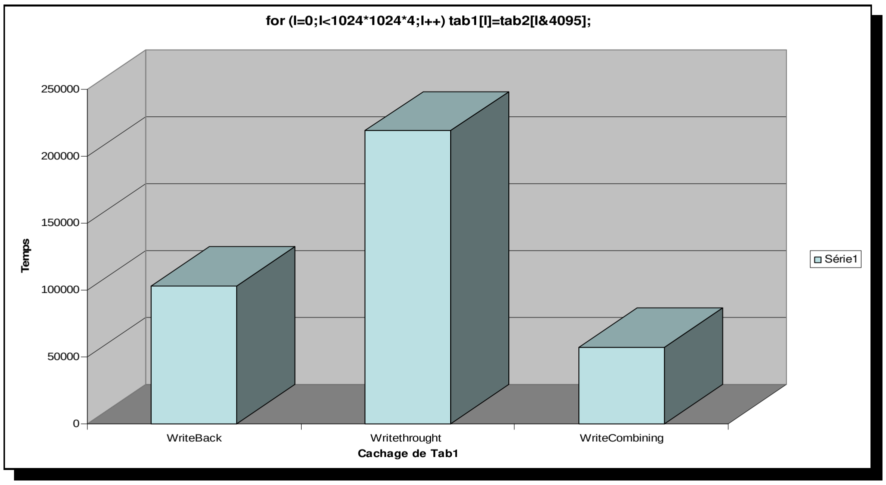


FIG. A.2 – Test de l'initialisation d'un tableau

```
For(l=0;l<1024*128*4;l++) tab1[l]=tab2[l &4095];
```

Le write back et le write throught chargent tab1 dans le cache, puis le modifient et l'écrivent dans la mémoire. Le write combining est ainsi plus rapide car les données à écrire ne sont pas chargées depuis la mémoire. Le write throught effectu chaque écriture après qu'elle ait été demandée par le processeur, alors que le write back effectue un bloc d'écriture et un bloc de lecture, ce qui est plus optimisé pour le processeur.

Le troisième test correspond à une opération courante : l'initialisation d'un tableau.

```
For(l=0;l<1024*4;l++) tab1[l*1024]=l;
```

Le Write Back lit le tableau depuis la mémoire avant de l'écrire et attend que le cache soit rempli pour commencer l'écriture. Ces deux ralentissements n'ont pas lieu avec le write throught, mais les écritures ne sont pas optimisées. Le write combining permet de ne plus avoir aucun des problèmes précédents car les écritures sont optimisées.

A.4 Conclusion

La pagination est semblable à la gestion du cache, car ce sont deux gestions d'une antémémoire, qui ne font des suppositions sur les données accédées dans le futur, sans dialogue avec l'application. Pour cette raison toutes deux utilise une politique d'évictions semblable : la LRU. Ce dernier point explique que les mêmes problèmes d'inadaptation de la politique d'évincement peuvent se produirent.

Ces quelques tests montrent que malgré un nombre plus limité de possibilités que pour la pagination, le changement de la politique de gestion du cache pour une politique adaptée aux applications peut permettre des gains de performance.

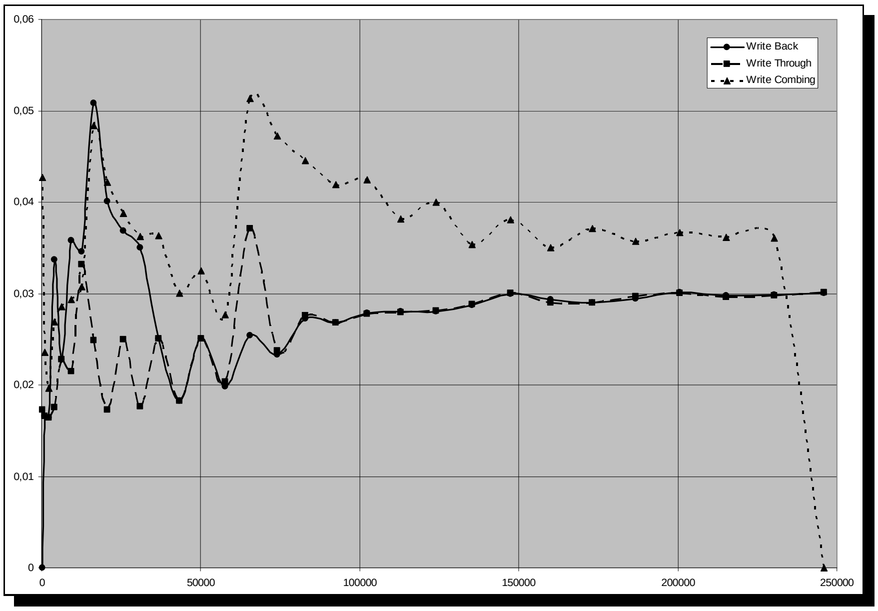


FIG. A.3 – Calcul d'un produit de matrice

Table des figures

Bibliographie

[1] V. Abrossimov, M. Rozier, and M. Shapiro. Generic virtual memory management for operating system kernels. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, volume 23, pages 123–136, 1989.

[2] Alexis Agahi, Robert D. Russel, and Gil Utard. A High Performance Modular File System. In *Proc. of High Performance Computing and Networking Europe'98*, volume 1401 of *LNCS*, Amsterdam, April 1998. Springer.

[3] Rexe Di Bona Alan Dearle, Anders Lindström, John Rosenberg, and Francis Vaughan. User-level management of persistent data in the grasshopper operating system. Technical report, University of Sidney, 1994.

[4] P. R. Amestoy and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. of Supercomputer Applics.*, 7 :64–82, 1993.

[5] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1) :15–41, 2001.

[6] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184 :501–520, 2000.

[7] D. Anderson, J. Chase, S. Gadde, A. Gallatin, K. Yocum, and M. Feeley. Cheating the i/o bottleneck : Network storage with trapeze/myrinet, 1998.

[8] Françoise André, Christine Morin, and Maria-Teresa Segarra. Mechanism for global processor and memory management on a NoW. In *HPCN EUROPE*, 1998.

[9] Remzi H. Arpaci-Dusseau, Andrea C, Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. The Architectural Costs of Streaming I/O : A comparaison of Workstations, Clusters and SMPs. In *Proceeding of the 4th IEEE International Symposium on High-Performance Computer Architecture (HPCA-4)*, Las Vegas, February 1998.

[10] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5) :361–372, 1998.

[11] Raoul A.F. Bhoedjang, Tim Ruhl, and Henri E. Bal. Scalable coherent interface. Technical report, Vrije Unniversiteit Amsterdam, November 1998.

[12] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 142–153, 1994.

[13] Alice Bonhomme and Patrick Geoffray. High Performance Video Server using Myrinet. In *1st Myrinet User Group (MUG)*, Lyon, 28-29 September 2000.

[14] Rajesh Bordawekar, Alok Choudhary, and Juan Miguel Del Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 367–376. ACM Press, 1993.

[15] Angela Demke Brown, Todd C. Mowry, and Orran Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, 19(2) :111–170, 2001.

[16] P. Cadinot, N. Dorta, B. Folliot, and P. Sens. Mécanisme de va-et-vient réparti en mémoire dans un environnement réseau haut débit. In *MPR'96- Journée de Recherche sur la Mémoire Répartie Partagée, 143-150*, May 1996.

[17] Philippe Cadinot, Neilze Dorta, and Bertil Folliot. MAÎS : un système de pagination en mémoire distante dans un enironnement réseau à haut débit. In *RenPar'9*. EPF Lausanne, 20-23 may 1997.

[18] Eddy Caron. *Calcul numérique sur données de grande taille*. Thèse, Université de Picardie Jules Verne, December 2000.

[19] Eddy Caron, Olivier Cozette, Dominique Lazure, and Gil Utard. Mémoire virtuelle et parallélisme de données. In *11es Rencontres sur le Parallélisme (RenPar11)*, Rennes, 9-11 juin 1999.

[20] Eddy Caron, Olivier Cozette, Dominique Lazure, and Gil Utard. Virtual memory management in data parallel applications. In *HPCN'99, High Performance Computing and Networking Europe – Workshop on High Performance Computation on Very Large Data Sets*, volume 1593 of *LNCS*. Springer, 11-13 April 1999.

[21] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. Implementation of a parallel file system : CCFS a case of study. Technical Report FIM/84.1/DATSI/94, Universidad Politecnic Madrid, Madrid, Spain, 1994.

[22] Giuseppe Ciaccio. Optimal communication performance on fast ethernet with GAMMA. In *IPPS/SPDP Workshops*, pages 534–548, 1998.

[23] MPI-IO Committee. MPI-IO : A Parallel File I/O Interface for MPI, 1996. Available from `http ://parallel.nas/nasa.gov/MPI-IO/mpi-io.html`.

[24] Olivier Cozette. Maximisation du débit d'entrées/sorties des grappes de stations par l'utilisation de l'accés distant direct. In *ACTES RenPar'2002, Hamamet, Tunisie*, April 2002.

[25] Olivier Cozette. Read^2 : Partager les disques au niveau réseau. In *RenPar'15 15ème Rencontres Francophones du Parallélisme*, La Colle sur Loup, 2003.

[26] Olivier Cozette, Cyril Randriamaro, and Gil Utard. Improving Cluster IO Performance with Remote Efficient Access to Distant Devices. In *Workshop on High Speed Local Network (HSLN'02)*, Tampa, USA, November 2002.

[27] Olivier Cozette, Cyril Randriamaro, and Gil Utard. Read2 : Put disks at network level. In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 698. IEEE Computer Society, 2003.

[28] DAFS collaborative. Direct Access File System version 1.0. http ://www.dafscollaborative.org/, 2001.

[29] Steven P. Van der Wiel and et al. When caches are not enough : A review of data prefetching techniques.

[30] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–241, 1999.

[31] Bernd Dreier, Markus Zahn, and Theo Ungerer. The rthreads distributed shared memory system. In *Proc. of the 3rd Int'l Conference on Massively Parallel Computing Systems (MPCS'98)*, 1998.

[32] David H. C. Du, Tai-Sheng Chang, Jenwei Hsieh, Yuewei Wang, and Sangyup Shim. Interface comparaisons : Ssa versus fc-al, April-June 1998.

[33] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9 :302–325, 1983.

[34] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5 :633–641, 1984.

[35] Mads Bondo Dydensborg. Direct remote access to devices. Master's thesis, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen OE, Denmark, 2000., 2000.

[36] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2) :115–126, 2001.

[37] Patrick Geoffray. OPIOM : Off-Processor IO with Myrinet. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGRID'01)*, Brisbane, Australia, 15-18 May 2001.

[38] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA : Retrospectives and Reprints*, pages 376–387, 1998.

[39] Gideon W. Glass. Adaptative page replacement, master thesis. Master's thesis, University of Wisconsin - Madison, 1998.

[40] Olivier Gluck, Amal Zerrouki, Jeanlou Desbarbieux, Alexandre Fenyo, Alain Greiner, Frack Wajsburt, Cyril Spasevski, and Fabricio Silva. Protocol and Performance Analysis of the MPC Parallel Computer. *Proceedings of 15th International Parallel & Distributed Processing Symposium (IPDPS'2001)*, page 52, San Francisco, CA, USA, April 2001.

[41] Brice Goglin and Loc Prylli. Design and implementation of orfa. Technical Report TR2003-01, LIP, ENS Lyon, Lyon, France, September 2003.

[42] Andreas Haeberlen. *Managing Kernel Memory Resources from User Level*. PhD thesis, Universitat Karlsruhe (TH). Insitut fur Betrirbs und Dialogsystem Lehrstuhl Systemarchitectur, April 2003.

[43] G. Heiser. Implementation and performance of the mungi single-address-space operating system, 1997.

[44] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an Internationnal Data Grid Project. In *GRID 2000*, Bangalore (Inde), 17-20 december 2000.

[45] Hsu and Peir. Buses. In *Allen B. Tucker, Jr. (Editor-in-Chief), The Computer Science and Engineering Handbook*. CRC Press, in cooperation with ACM, 1997, 1997.

[46] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS : A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995. ACM Press.

[47] Kai Hwang, Hai Jin, and Roy Ho. RAID-x : A new distributed disk array for I/O-centric cluster computing. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, pages 279–287, Pittsburgh, PA, 2000. IEEE Computer Society Press.

[48] Matrox Graphics Inc. *Matrox MGA-200 Specification. Document Number 10582-MS-0301*, November 1998.

[49] Infiniband Trade Association. *Infiniband ^{TM} Architecture Release 1.0*, 2000.

[50] Y. Ishikawa, H. Tezuka, A. Hori, S. Sumimoto, T. Takahashi, F. O'Carroll, and H. Harada. Rwc pc cluster ii and score cluster system software – high performance linux cluster, 1999.

[51] Bruce L. Jacob, Peter M. Chen, Seth R. Silverman, and Trevor N. Mudge. An analytical model for designing memory hierarchies. *IEEE transaction on computer*, pages 1180–1185, October 1996.

[52] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks : Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[53] Orran Krieger and Michael Stumm. HFS : A performance-oriented flexible file system based on building-block compositions. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 95–108, Philadelphia, May 1996. ACM Press.

[54] Keith Krueger, David Loftesness Amin Vahdat, and Thomas Anderson. Tools for the development of application-specific virtual memory management. Technical report, University of California, Berkeley, 1993.

[55] K. Kubota, Y. Tanaka, M. Matsuda, M. Sato, H. Tezuka, A. Hori, and Y. Ishikawa. Communication performance of gigabit LAN workstation cluster RWC/WSC. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and U. Trottenberg, editors, *Parallel Computing : Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo'97, 19-22 September 1997, Bonn, Germany*, volume 12, pages 441–444, Amsterdam, 1998. Elsevier, North-Holland.

[56] Jochen Liedtke. Toward real microkernels : The inefficient, inflexible first generation inspired development of the vastly improved second generation, which may yet support a variety of operating systems. *Communications of the ACM*, 39(9) :70–77, 1996.

[57] W.B. Ligon III and R. B. Ross. An Overview of the Parallel Virtual File System. In *Proc. of the 1999 Extreme Linux Workshop*, June 1999.

[58] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11 :134–172, 1990.

[59] Bernard Tourancheau Loic Prylli. Bip messages user manual. Technical report, INRIA, September 1997.

[60] Pierre Lombard and Yves Denneulin. Serveur nfs distribué pour grappes de pcs. In *ACTES RenPar'2002, Hamamet, Tunisie*, April 2002.

[61] LSILOGIC. *LSI53C1010-33 PCI to Dual Channel Ultra160 SCSI Multifunction Controller*, February 2001.

[62] Martin Maierhofer. *Efficient Arbitration and Bridging Techniques for High-Performance Conventional Multimedia Servers*. PhD thesis, University of Teesside, 2000.

[63] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 3–17. USENIX Association, October 1996.

[64] Myricom. *Myricom GM myrinet software and documentation*, 2000. http ://www.myri.com.

[65] Jean-François Méhaut and Raymond Namyst. PM2 :Parallel Multithreaded Machine. A computing environment for distributed architecture. In *Parallel Computing*, pages 279–285. Elsevier Science Publisher, September 1995.

[66] T10 American National. Scsi parallel interface-4 project 1365d revision 10, August 2002.

[67] Yoonho Park, Ridgway Scott, and Stuart Sechrest. Virtual memory versus file interfaces for large, memory-intensive scientific applications. In *Proceedings of Supercomputing '96*. ACM Press and IEEE Computer Society Press, November 1996. Also available as UH Department of Computer Science Research Report UH-CH-96-7.

[68] Kenneth W. Preslan, Adrew P. Barry, Jonatha E. Brassow, Grant M. Erickson, Erling Nygaard, Christopher J. Sabol, Steeven R. Soltis, David C. Teigland, and Matthew T. O'Keefe. A 64-bit, Shared Disk File System for Linux. In *16th Mass Storage Systems Symposium*, San Diego, March 15-18 1999. IEEE.

[69] Kenneth W. Preslan, Steven R. Soltis, Christopher J. Sabol, Mathew T. O'Keefe, Gerry Houlder, and Jim Coomes. Device locks : Mutual exlusion for storage area networks. Technical report, Parallel Computer Systems Laboratory University of Minnesota, Deagate Technology, 1999.

[70] L. Prylli and B. Tourancheau. BIP : A new protocol designed for high performance networking on myrinet. *Lecture Notes in Computer Science*, 1388 :472– ? ?, 1998.

[71] The RAIDBook : A source book for RAID technology. The RAID Advisory Board, Lino Lakes, MN, June 9 1993. First Edition.

[72] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron dan David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for uniprocessor and multiprocessor architectures. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–9, 1987.

[73] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 297–306, San Jose, California, 1994.

[74] PCI SIG. *Conventional PCI 2.3*, 2003. http ://www.pcisig.com.

[75] NCITS T13. Working t13 draft 1410d at attachment with packet interface - 6 (ata/atapi-6), August 2001.

[76] NCITS T13. Working t13 draft 1510d ata host adapter standards, August 2001.

[77] Nisha Talagala, Satoshi Asami, David Patterson, Bob Futernick, and Dakin Hart. The Art of Massive Storage : A Web Image Archive. *IEEE Computer Journal*, 33(11), November 2000.

[78] PLX Technologie. *PLX Technologie*, 2000. http ://www.plxtech.com.

[79] Geoffrey M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 33–43. ACM Press, 1998.

[80] B. White, W. Ng, and B. Hillyer. Performance comparison of ide and scsi disks, 2001.

Résumé

Cette thèse a pour objectif d'étudier et de définir des concepts et des outils nécessaire au traitement de grandes masses de données. A cette fin, elle se concentre sur l'aspect entrées/sorties et plus particulièrement sur la pagination et l'accès aux données distantes.

Dans la première partie, cette thèse présente un outil *MMUM/MMUSSEL* de gestion de la pagination en mode utilisateur, qui permet d'adapter celle ci aux particularités des applications. Cet outil a été utilisé pour améliorer la factorisation LU [19, 20]. Puis *MMUM/MMUSEL* a permis de créer un gestionnaire de pagination pour les applications travaillant sur des flux. Cet outil a été étendu afin d'interagir avec le système d'exploitation. Cette extension se concrétise par la déportation de la gestion de la pagination sur un processus séparé, nommé moniteur. La première application est un logiciel d'analyse de la pagination des programmes. Ce programme a permis l'étude de la pagination du solveur de matrice creuse *MUMPS*. Cette étude a permis la création d'un moniteur et d'une interface avec la pagination permettant l'amélioration de la pagination.

Dans la seconde partie, cette thèse décrit une bibliothèque d'accès efficace aux entrées/sorties distante. Pour ce faire, le comportement architectural des grappes de stations dans le cas d'entrées/sorties intensif est modélisé. Cette modélisation a permis l'étude des gains engendrés par le passage direct des données du disque à la carte réseau. Cette étude a été validée par l'émulation des disques a l'aide d'une carte graphique [24, 26]. Puis l'outil READ^2 a été développé. Il se compose d'un pilote de périphérique et d'une bibliothèque permettant d'accéder aux disques distant à l'aide d'un pilote embarqué dans la carte réseau[25, 27]. Cet outil est interopérable avec la bibliothèque de passage de messages *GM*. La première application de READ^2 a été la création d'un système de fichiers primaire afin de comparer les performances de READ^2 avec le pilote *NBD*. Puis READ^2 a été implémenté avec *GFS*. Ce système de fichiers a été comparé avec *GFS* sur IP. Cette comparaison a permis de constater que READ^2 n'utilise pas les ressources processeur et mémoire du nœud du disque contrôlé.

Abstract

The goal of this thesis is to define some concepts and tools to process huge data set. So, it focuses on the paging and remote data access to address the input/output aspect.

In the first part of the thesis, we define the user paging tool *MMUM/MMUSSEL*. This tool has been used to improve the factorisation LU paging performance[19, 20]. *MMUM/MMUSSEL* has been used to create a generic paging system for streaming application. This tool has been improved to collaborate with the system. This new release migrate the paging to a new process named monitor. It is used to make a paging analysis program. This program was used to analyse the paging of an multifrontal solver *MUMPS*. This study aid to create a new interface to exchange data between programs and paging system.

In the second part, we describe an efficient scheme to access remote disks. We first modelize the bus of cluster of Workstations. This study leads to study the interest of read remote disk without crossing remote memory. This study have been validated with disks simulated by graphics cards [24, 26]. So, we have implemented READ^2 tool. This tool was broken in two parts : an kernel device driver, a driver embedding in the network card [25, 27]. This tool don't broke the *GM* message system. The first application was a primitive file system build on top of READ^2. We compare this file system with *NBD* and *PVFS*. We used READ^2 with the distributed file system *GFS*. We compared *GFS* over READ^2 with *GFS* over GM/IP. We show that READ^2 uses neither main processor power nor memory bandwidth.