Calcul sur les données volumineuses et stockage distribué à grande échelle

Gil Utard

Mémoire

présentée devant

l'Université de Picardie Jules Verne

pour obtenir

le diplôme d'habilitation à diriger des recherches
Spécialité : Informatique

par Gil Utard

Titre :

Calcul sur les données volumineuses et stockage distribué à grande échelle

Rapporteurs	Mr. Pierre Sens	Professeur, LIP6, Sorbonne Université
	Mr. François Taïani	Professeur, IRISA, Université de Rennes
	Mr. Gaël Thomas	Professeur, PDS, Télécom SudParis
Examinateurs	Mr. Gilles Dequen	Professeur, MIS, Université de Picardie Jules Verne
	Mr. Chu Min Li	Professeur, MIS, Université de Picardie Jules Verne

pour les travaux effectués au Laboratoire de Recherche en Informatique d'Amiens (LaRIA), au laboratoire Modélisation, Informations et Système (MIS), et à l'Institut Nationale de Recherche en Informatique et Automatique (INRIA) Rhône Alpes et au Laboratoire de l'Informatique du Parallélisme (LIP) de l'ENS Lyon.

Remerciements

J'exprime ma plus profonde gratitude aux rapporteurs de cette HDR qui m'ont fait l'honneur d'évaluer mon parcours et mon travail.

J'exprime aussi toute ma reconnaissance au Professeur Chu-Min Li, qui a bien voulu se porter garant de mon dossier d'inscription à l'école doctorale.

Mes remerciements au directeur du laboratoire Gilles Dequen, et à la directrice d'équipe Florence Levé, qui par leur constante, mais amicale, pression m'ont convaincu d'aller au bout de cette démarche.

Enfin, je tiens à exprimer ma reconnaissance envers tous mes collègues et étudiants que j'ai côtoyés tout au long de ma carrière, à Amiens, à Lyon, et dans le GDR RSD, pour tous nos échanges fructueux qui ont pu faire avancer nos travaux.

Table des matières

Chapitre 1

Introduction

Ce mémoire est une synthèse des travaux de recherche que j'ai entrepris depuis ma nomination comme Maître de Conférences à l'Université de Picardie Jules Verne en 1998. Ces différents travaux ont été effectués au LaRIA (Laboratoire de Recherche en Informatique d'Amiens), qui est devenu ensuite le MIS (Modélisation, Informatique et Systèmes), après la fusion avec le laboratoire d'automatique de l'Université, au LIP (Laboratoire de l'Informatique du parallélisme à l'ENS Lyon) lors de mon détachement dans le projet ReMaP de l'INRIA Rhône-Alpes de 2001 à 2003, et dans les entreprises que j'ai fondées (UbiStorage, Ugloo), à partir de 2006.

Mon activité peut se découper sur deux périodes. La première est dans le domaine du HPC, et plus particulièrement sur le traitement de données de grande taille. La seconde, qui concerne le problème de stockage et d'archivage distribué de données à grande échelle.

La prochaine partie présente un résumé de mes activités jusqu'à ce jour, avec la liste des doctorants que j'ai encadrés, et la liste de mes publications. Les deux parties qui suivent correspondent aux deux périodes que je distingue. Chacune est composée d'une série d'articles sélectionnés, précédée d'un chapitre qui les replace dans le contexte scientifique. Enfin, le dernier chapitre présente mes perspectives de travaux futurs.

Première partie

Synthèse

Chapitre 2

Résumé des activités de recherche

Comme la plupart d'entre nous, mes premières armes dans la recherche se sont déroulées en 1990, pendant mon stage de DEA, ainsi que mon service en tant que scientifique à l'ETCA et sur le Site Expérimental en Hyper-Parallélisme, où j'étudiais la programmation de la Connection Machine (CM2), et j'effectuais des développements systèmes pour un prototype de machine parallèle embarquée basée sur le processeur i860 et des FPGA Xilink.

J'ai effectué ma thèse de 1992 à 1995 au LIP (ENS Lyon) sous la direction de Luc Bougé en tant qu'allocataire de recherche DGA/DRET. Mon sujet de thèse était l'étude de la sémantique des langages à parallélisme de données (e.g. HPF) en vue de la validation des programmes et des schémas de compilation. En particulier, j'ai défini un système de preuve axiomatique (logique de Hoare) pour un squelette de langage parallèle et j'ai démontré sa complétude. D'autre part, j'ai validé formellement une optimisation de compilation visant à réduire le nombre de synchronisations dans le code généré à partir de programmes C* (une extension data-parallèle du C, utilisée sur la Connection Machine) et j'ai proposé une généralisation.

J'ai aussi étudié l'exécution de programmes data-parallèles sur réseaux de stations de travail. J'ai réalisé un prototype de compilateur intégrant l'équilibrage dynamique de la charge. Ce travail était basé sur le compilateur C* de l'Université du New Hampshire (UNH). Ce travail a été un point de départ d'une collaboration entre le projet ReMAP et l'UNH (travaux de Chritian Perez).

Suite à ma thèse, j'ai été ATER dans le même organisme en 1995, puis ATER au LSV (Laboratoire de Spécification et Validation) à l'ENS Cachan en 1996. En 1997, j'ai été recruté comme MCf au LaRIA (Laboratoire de Recherche en Informatique d'Amiens) de l'UPJV (Université de Picardie Jules Verne), qui est devenu ensuite le MIS (Modélisation, Informatique et Systèmes) après la fusion avec le laboratoire d'automatique de l'Université.

Au cours des premières années à ce poste, j'ai réorienté mon activité de recherche par l'étude des méthodes et outils pour le traitement de grandes masses de données, en particulier sur les grappes de machines, appelées *cluster*. Ces travaux recouvrent des aspects de l'algorithmique numérique, de l'architecture matérielle et réseau, ainsi que des aspects système. Ce travail a bénéficié d'un financement de la région (thèse, investissement et fonctionnement), et a été à l'origine de deux thèses. En septembre 2000, j'ai été promu à la première classe par le CNU, et titulaire par deux fois de la PEDR (1999 et 2003).

En 2002, j'ai été détaché pendant deux ans en tant que CR1 à l'INRIA Rhône-Alpes, dans le projet ReMAP situé au LIP, ENS Lyon. J'ai collaboré avec Jean Yve l'Excellent, alors CR INRIA, sur l'étude de l'impact des accès mémoire des méthodes multifrontales de calcul numérique, qui a donné lieu à un co-encadrement de thèse. Dans le même temps, j'ai initié de nouveaux travaux de recherche sur les systèmes de stockage distribué à grande échelle dans le cadre de

l'ACI CGP2P, qui sera la ligne directrice de mon activité pour la suite.

Lors de mon retour à l'UPJV, j'ai continué les travaux entrepris dans les systèmes de stockage distribué pair à pair avec un objectif de valorisation économique. Ces travaux ont donné lieu à deux encadrements de thèse, quelques publications ainsi que la conception d'un brevet (USA et Europe). En ce qui concerne la valorisation, j'ai défini un projet de création d'entreprise qui a été soutenue par l'Incubateur de Picardie, et qui a été lauréat du concours national d'aide de création d'entreprises de technologie innovante (émergence ee 2004, et création en 2005). J'ai de plus été accompagné dans le cadre de la formation Challenge+ de HEC Paris.

L'entreprise (UbiStorage SA) a été créée en 2006, où j'ai été détaché en tant que PDG jusqu'en 2014. Par la suite, l'activité a été reprise par la société Ugloo SAS avec laquelle un contrat de collaboration de recherche a été conclu, et qui continue à ce jour. Pendant cette période, j'ai maintenu une activité de recherche, notamment dans le cadre d'un projet précompétitif de l'ANR (SPREAD), qui été porté par mon entreprise, où participe le LIP6 (projet Regal), Eurecom, le LACL et l'INRIA Sophia Antipolis (projet MASCOTTE), et dans lequel j'ai co-encadré une thèse CIFRE. Mes travaux de recherche actuelle sont dans la continuité ce cette dernière activité.

2.1 Résumé des travaux effectués et des projets de recherche

Lors de mon arrivé au LaRIA à Amiens en 1997, j'ai ré orienté mon activité de recherche sur le calcul parallèle *out-of-core*, les entrées/sorties parallèles et les grappes de PCs. Je me suis ensuite intéressé à la gestion de données à très grande échelle.

Mes premiers projets de recherche concernent la gestion de grandes masses de données et s'articulent autour de deux axes qui sont :

— le calcul intensif sur des données de grande taille (calcul out-of-core, entrées-sorties à haute performance) ;
— le stockage distribué à grande échelle de type pair à pair.

Le premier axe s'inscrit dans la continuité des travaux que j'ai débutés lors de ma nomination comme Maître de Conférences en 1998. Le second axe a débuté en 2001 dans le cadre d'une ACI GRID sur le pair-à-pair (CGP2P).

Dans ce qui suit, je présente les différents travaux effectués ainsi que des perspectives de recherche pour chacun de ceux-ci.

2.1.1 Calcul intensif sur des données de grande taille

La recherche en parallélisme s'est concentrée avec succès sur les aspects calcul et communication : les nouvelles architectures parallèles sont aujourd'hui capables d'atteindre des puissances de calcul de l'ordre de plusieurs 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.

En 1998, j'ai commencé à étudier les méthodes et outils pour le traitement de grandes masses de données (en particulier sur les grappes de PCs). Cette problématique est aussi appelée le calcul *out-of-core*. Je propose donc de continuer ces travaux de recherche. Ce travail

comporte plusieurs facettes : les aspects système (pagination, systèmes de fichiers), les algorithmes, les outils numériques.

Gestion " intelligente " de la mémoire virtuelle : pagination adaptative

Puisque l'objectif est de pouvoir traiter de grandes données (e.g. des matrices de très grande taille), une idée simple est d'utiliser les gestionnaires de mémoire virtuelle présents sur tous les systèmes. Bien entendu une telle solution est totalement inefficace, les mécanismes standards de pagination sont totalement inadaptés : les politiques de remplacement des pages mémoires de type FIFO ou LRU font l'hypothèse d'une forte localité temporelle dans l'accès aux données par les différentes applications. Cette propriété n'apparaît pas toujours dans le cadre du calcul intensif.

Dans [28, 33], nous avons introduit un nouveau mécanisme système qui permet de reporter la gestion de la mémoire virtuelle au niveau de l'application même : avec la connaissance des accès en mémoire de l'application, les temps d'exécution peuvent être réduits significativement grâce à une pagination adaptée.

Nous avons réalisé un prototype pour Linux se composant d'un module et d'une librairie (MMUM-MMUSSEL). Ils permettent de définir des régions de mémoire dont la pagination est gérée au niveau utilisateur : les défauts de page sont retournés par le système à l'application. Celle-ci peut par exemple évincer une page pour libérer de la mémoire physique, ou alors pré-charger d'autres pages, ceci grâce à un jeu réduit de primitives.

Nous avons validé notre approche par l'intégration de ces mécanismes dans une application réelle irrégulière : le solveur multifrontal MUMPS qui a été développé au CERFACS. Les premiers résultats montrent que l'on peut réduire significativement les Entrées/Sorties par une politique de pagination adaptée [18]. L'avantage de cette approche est qu'elle consiste à annoter le code de l'application, ce qui évite une restructuration de celui-ci, souvent difficile à mettre en œuvre par le coût de développement et de mise au point.

Une suite possible de ce travail serait d'étudier l'emploi de ce type de technique dans les couches de virtualisation et de globalisation des ressources à grande échelle afin d'intégrer, en plus de la gestion des processus et des communications, la gestion des mémoires. Par exemple, proposer un espace d'adressage unique aux applications distribuées dans lequel on combine l'ordonnancement des processus et celui de la mémoire.

Restructuration de code

Dans le cas où le nombre de calculs à effectuer est largement supérieur à la taille des données, on peut espérer un certain taux de réutilisation des données et par conséquent une réduction du volume d'entrées/sorties. Une autre approche, qui peut être complémentaire à la précédente, consiste à restructurer les codes afin d'exhiber suffisamment de *localités temporelles* et *spatiales* des accès aux données et donc minimiser les accès aux disques.

Je me suis intéressé en particulier aux algorithmes et bibliothèques de calcul numérique. En particulier, j'ai étudié les algorithmes parallèles *out-of-cores* de ScaLAPACK. Un modèle de prédiction des performances pour les algorithmes parallèles *out-of-cores* de factorisation matricielle a été défini [25]. Grâce à ce modèle, nous avons démontré un résultat assez remarquable : en choisissant une distribution de la matrice adéquate, et en modifiant l'algorithme initial (introduction d'un schéma de recouvrement du temps d'entrées/sorties par du calcul), le temps d'exécution de l'algorithme *out-of-core* sur une machine disposant d'une mémoire proportionnelle à l'*ordre* de la matrice ($O(n)$) est identique au temps d'exécution de l'algorithme *in-core* disposant d'une mémoire proportionnelle à la *taille* de la matrice ($O(n^2)$) [3]. Par exemple, nous pouvons calculer la factorisation LU d'une matrice d'ordre

100 000 (en double) de 80 Gigaoctets en moins d'une journée avec 16 stations possédant seulement 16×256Mo= 4 Gigaoctets de mémoire au total. L'algorithme parallèle classique (*right-looking*) nécessiterait une mémoire totale de 80 Gigaoctets pour un même temps d'exécution.

Nous avons étendu ce résultat au problème du calcul de l'inverse d'une matrice. Ce calcul est obtenu à partir de la décomposition LU de la matrice. Ici aussi les performances de notre nouvel algorithme *out-of-core* sont identiques à celles de l'algorithme *in-core* [20].

Je me suis intéressé à l'extension de ces techniques *out-of-core* dans le cas creux. La principale difficulté par rapport au cas dense est l'irrégularité des calculs. En collaboration avec d'Abdou Guermouche et Jean-Yves l'Excellent (LIP/ENS-Lyon) nous avons étudier comment intégrer les techniques *out-of-core* dans le solveur creux MUMPS décrit dans la section précédente . C'est un travail qui a aussi fait l'objet d'un contrat avec le CETMEF (Centre d'Étude Maritime et Fluviale, Compiègne). Nous nous sommes intéressés à l'impact de l'irrégularité sur le comportement mémoire, ainsi que l'impact des différentes techniques de renumérotation [2]. Une extension *out-of-core* basée sur les mécanismes de pagination précédemment décrits a été réalisée. La prochaine étape serait d'étudier la parallélisation de cette méthode en concevant de nouvelles politiques d'ordonnancement qui intégreront la gestion de la pagination.

Entrées/sorties Parallèles

Le système d'entrées/sorties est le goulot d'étranglement de toutes architectures, surtout pour les applications qui ne sont pas bornées par les calculs. 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 plusieurs disques. C'est ce qui est proposé par les RAIDs (Redundant Array of Independant Disks).

Pour améliorer les performances dans le cas d'applications parallèles, il faut organiser au mieux les différents accès concurrents au système de fichiers. C'est dans cette optique que la communauté a défini une interface d'entrées-sorties parallèles comme extension de la bibliothèque de communication MPI : MPI-IO. Celle-ci permet de définir différents modes d'accès aux données, à charge pour la bibliothèque d'optimiser les entrées/sorties en fonction du système de fichiers.

Je me suis donc intéressé aux systèmes de fichiers parallèles [30]. En particulier j'ai effectué une adaptation préliminaire de la librairie standard MPI-IO (ROMIO) sur PVFS [27] et étudié d'autres politiques de distribution [24] afin d'améliorer les performances d'entrées/sorties.

Je me suis aussi intéressé à la conception de mécanismes de base dans lesquels la gestion de l'accès aux données dans les grappes est reportée au niveau du réseau. Les cartes réseau modernes sont programmables et possèdent aujourd'hui des processeurs élaborés et une mémoire conséquente. Cette caractéristique peut être utilisée pour gérer les entrées/sorties parallèles, sans passer par le processeur hôte, et ainsi libérer des ressources. Dans le cadre de la thèse d'Olivier Cozette, nous avons ainsi conçu et développé la bibliothèque READ^2 qui projette les disques d'une grappe sur le réseau rapide (en l'occurrence Myrinet) [21, 19]. La suite logique de ce travail serait d'étudier et de définir un système de fichiers parallèles à haute performance pour grappe qui intègre en plus des mécanismes de tolérance aux pannes.

2.1.2 Stockage distribué à grande échelle

En 2001, je me suis intéressé aux nouveaux protocoles de type pair à pair pour la gestion de fichiers sur Internet qui sont apparus. Parmi les plus connus, on peut citer Napster, Freenet, Gnutella où les fichiers sont distribués sur l'ensemble des PCs. Ces systèmes sont des alternatives à l'approche client-serveur du WEB. Alors que Napster est centralisé en ce qui concerne

l'indexation des données, Gnuttella et Freenet sont entièrement distribués. Une conséquence est que la recherche de données particulières est très coûteuse (parcours du réseau) et surtout non exhaustive. En fait ces systèmes sont orientés essentiellement dans la mise en commun et dans la diffusion des données et des informations (surtout les fichiers mp3). Les objectifs avoués étant essentiellement ceux de l'anonymat des sources d'informations (celui-ci est particulièrement poussé sur Freenet où il est impossible de connaître la machine d'origine de l'information), ce qui impose un surcoût non négligeable dans l'accès aux données. Parallèlement des mécanismes de gestion distribuée de fichier pour des approches de type grille où la granularité est plus forte se sont développés (IBP, Oceanstore)

En général, il n'y a aucune garantie en ce qui concerne la fiabilité et la disponibilité des données. L'objectif d'alors était de pouvoir manipuler de grandes masses de données. Les problèmes posés concernent plusieurs points.

— La fiabilité et la disponibilité des données : on doit faire face à la volatilité des ressources de stockage. Les données ne doivent pas disparaître à tous jamais, ou tout au moins avec une probabilité très faible. Cela implique la duplication des données. De même, on ne peut garantir l'accès à certaines données à tout instant, par contre on doit pouvoir la garantir dans une certaine période de temps. Cette période étant donnée par la fréquence de connexions des ressources détenant les données.
— Le contrôle des données : on doit pouvoir contrôler la durée de vie des données, on doit pouvoir les modifier. De par la duplication de celles-ci, plusieurs problèmes fondamentaux surgissent tels que les problèmes de cohérence, de ramasse-miettes.
— La confidentialité et la sécurité des données : les données ne doivent pas être consultées par de tierces personnes, elles ne doivent pas être corrompues. Il faudra employer des mécanismes de cryptage et de certification.
— Le suivi des données. Des mécanismes de migration liés aux différents traitements seront certainement mis en œuvre. Il faudra être capable de localiser les données à tout instant.

Je me suis donc intéressé à la problématique du stockage distribué à grande échelle, ceci dans le cadre de l'ACI Grid CGP2P qui regroupe plusieurs laboratoires (LRI, IMAG, LIFL, LIP, LaRIA). Ce travail s'inscrivait aussi dans l'ACI Masses de Données Grid Explorer. L'un des objectifs étant de proposer une virtualisation de la fonction mémoire jusqu'à aujourd'hui associée à des dispositifs physiques, ou plus précisément de reporter cette fonctionnalité sur le réseau lui-même. C'est un projet qui recouvre plusieurs aspects théoriques comme les systèmes distribués, la distribution et le partitionnement des données, qui doit tenir compte de la dynamicité des systèmes.

En particulier je me suis penché sur la question cruciale de la pérennité des données dans un tel type d'architecture. J'ai donc défini un modèle stochastique qui classifie les différents systèmes de type pair à pair en fonction de la disponibilité et de la volatilité de ses pairs et qui permet de déterminer l'efficacité des différents schémas de redondance (replication simple, Reed Solomon, Shamir, Tornado, ...). Un résultat intéressant de cette étude est que les schémas sophistiqués de redondance, a priori meilleurs (tel que Reed Solomon), s'avèrent moins efficaces que la réplication classique pour les systèmes qui disposent d'un taux de disponibilité moyen [17].

D'autre part j'ai étudié le problème de la distribution des données à grande échelle dans un système de stockage P2P : quand un pair tombe en panne, un processus de reconstruction régénère les données perdues avec l'aide des autres pairs. Dans nos résultats précédents, nous avons observé que pour assurer la pérennité des données, un système de stockage pair à pair doit faire face à un grand nombre de reconstructions.

Pour minimiser le trafic réseau au niveau des pairs lors de la reconstruction, des stratégies

de distribution doivent prendre en compte un nouveau paramètre : le coût de perturbation maximal d'un pair pendant le processus de régénération des données. Le coût de dérangement d'un pair est défini par le nombre de données envoyées par un pair lors de la reconstruction.

Nous avons défini de nouvelles distributions capables de "diluer" au mieux le coût de reconstruction pour chaque pair, afin de minimiser le "dérangement" de ce processus pour chaque utilisateur. Nous avons montré que le coût minimal était obtenu pour des distributions dites "idéales". Cependant nous avons montré que ces distributions idéales correspondent aux solutions du problème des plans projectifs qui reste un problème ouvert des mathématiques : une conséquence est que ces distributions idéales peuvent difficilement être construites. Nous avons donc défini un nouveau type de distribution basée sur les propriétés des nombres premiers et nous avons montré que les distributions obtenues étaient asymptotiquement optimales. Le coût final induit par les reconstructions est alors proportionnel au ratio de la racine carrée du nombre de blocs de données et des pairs. Dans la pratique, les résultats expérimentaux montrent que notre distribution est très proche de l'idéale [16].

C'est ce travail qui a donné lieu par la suite au projet de valorisation et qui continue aujourd'hui.

2.2 Directions de thèse

Eddy Caron (2000) — Calcul numérique sur les données de grande taille.
- — Co-encadrement à 80% avec le Pr. J.-F. Myoupo de l'UPJV.
- — Actuellement, McF à l'ENS Lyon.

Olivier Cozette (2003) — Contributions système pour le traitement de grandes masses de données sur grappes.
- — Co-encadrement à 99% avec le Pr. J.-F. Myoupo de l'UPJV.
- — Actuellement Senior Software Enginneer chez Google, Mountain View, CA, USA.

Abdou Guermouche (2004) — Étude et optimisation du comportement mémoire dans les méthodes parallèles de factorisation de matrices creuses.
- — Co-encadrement à 50% avec J.-Y. Excellent, CR INRIA Rhône-Alpes.
- — Actuellement Maître de Conférences à L'Université de Bordeaux.

Ghislain Secret (2009) — La maintenance des données dans les systèmes de stockage pair à pair.
- — Co-encadrement à 99% avec le Pr. Vincent Villain de l'UPJV.
- — Actuellement gestionnaire de projet chez Orange SA.

Samira Chaou (2013) — Modélisation et analyse de la sécurité dans un système de stockage pair à pair.
- — Co-encadrement à 25% avec le Pr. Franck Delaplace de l'Université d'Evry.
- — Actuellement Ingénieur Sécurité à la RATP.

Bibliographie

— Publications dans des journaux internationaux

[1] Cyril Randriamaro, Olivier Soyez, Gil Utard, and Francis Wlazinski. Data distribution in peer to peer storage system. *J. Grid Computing*, 4 :311–321, 2006.

[2] A. Guermouche, J.Y. L'Excellent and G. Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9) :1191-1218, 2003.

[3] E. Caron and G. Utard. On the Performance of Parallel Factorization of Out-of-Core Matrices. *Parallel Computing*, 30(3) :357-375, February 2004.

[4] Christophe Cérin, Olivier Cozette, Gil Utard, Hazem Fkaier, and Mohamed Jemni. Parallel out-of-core sorting and fast accesses to disks. *Int. J. High Perform. Comput. Netw.*, 3(2/3) :188–202, 2005.

[5] Bernard Jurkowiak, Chu Min Li, and Gil Utard. A parallelization scheme based on work stealing for a class of SAT solvers. *J. Autom. Reason.*, 34(1) :73–101, 2005.

[6] L. Bougé, D. Cachera, Y. Le Guyadec, G. Utard and B. Virot. Formal validation of data-parallel programs : a two-component assertional proof system for a simple language. Theoretical Computer Science (189), p. 71-107. 1997.

[7] D. Cachera and G. Utard. Proving data parallel programs : a unifying approach. Parallel Processing Letters. Décembre 1996. Communiqué à *Int. Work. on Formal Methods for Parallel Programming : Theory and Application (FMPPTA'96)*. Honolulu, Hawaï. Avril 1996.

[8] G. Utard and G. Hains. Deadlock-free absorption of barrier synchronisations. Information Processing Letters. No 56. 1995.

— Edition de numéro spécial

[9] C. Cérin et G. Utard. "Parallélisme et Systèmes Distribués." In *Technique et Science Informatique*, 21(5), Hermès Edition, Juin 2002.

— Chapitre de livre

[10] L. Bougé, D. Cachera, Y. Le Guyadec, G. Utard, and B. Virot. "Formal Validation of Data-Parallel Programs : Introducing the Assertional Approach". In *The Data Parallel Programming Model*, G.-R. Perrin and A. Darte Editors, volume 1132 of *LNCS*, pages 252–281. Springer, June 1996.

— Conférences internationales avec comité de lecture

[11] Gil Utard, Ben Fadhl Mariem. A anonymous data control access protocol in distributed storage systems. In *Proceedings of the Fourth Workshop on Security and Privacy in the Cloud, Beijing, China, may 2018.*

[12] Gil Utard, Hung-Cuong Le, and Trung-Thanh Tran. U-RPC : a protocol for microservices in DHT. In *Proceedings of the Eighth International Symposium on Information and Communication Technology, Nha Trang City, Viet Nam, December 7-8, 2017*, pages 317–324. ACM, 2017.

[13] Samira Chaou, Gil Utard, and Franck Pommereau. Evaluating a peer-to-peer storage system in presence of malicious peers. In Waleed W. Smari and John P. McIntire, editors, *2011 International Conference on High Performance Computing & Simulation, HPCS 2012, Istanbul, Turkey, July 4-8, 2011*, pages 419–426. IEEE, 2011.

[14] Ghislain Secret and Gil Utard. A study of reconstruction process load in P2P storage systems. In Abdelkader Hameurlain, editor, *Data Management in Grid and Peer-to-Peer Systems, First International Conference, Globe 2008, Turin, Italy, September 3, 2008. Proceedings*, volume 5187 of *Lecture Notes in Computer Science*, pages 12–21. Springer, 2008.

[15] Olivier Soyez, Cyril Randriamaro, Gil Utard, and Francis Wlazinski. Dynamic Distribution for Data Storage in a P2P Network In *2nd International Conferance on Advances in grid and Pervaisve Computing (GPC'07), 2-4 May 2007, Paris, France*, pages 555–566..

[16] Olivier Soyez, Cyril Randriamaro, Gil Utard, and Francis Wlazinski. Data distribution for failure correlation management in a peer to peer storage system. In *4th International Symposium on Parallel and Distributed Computing (ISPDC 2005), 4-6 July 2005, Lille, France*, pages 242–249. IEEE Computer Society, 2005.

[17] Gil Utard and Antoine Vernois. Data Durability in Peer-to-Peer Storage Systems. In *Proc. 4th Workshop on Global and Peer to Peer Computing of the IEEE/ACM CCGrid Conference*, Chicago, April 2004.

[18] Olivier Cozette, Abdou Guermouche, and Gil Utard. Adaptative Paging for a Multifrontal Solver. In *Proceedings of the 18th Annual ACM International Conference on Supercomputing : ICS'04*, Saint Malo, juin-juillet 2004, p. 267-276

[19] Olivier Cozette, Cyril Randriamaro, and Gil Utard. READ^2 : Put disks at network level. In *Workshop on Parallel I/O in Cluster Computing and Computational Grids*, Tokyo, Japan, May 2003.

[20] A. Guermouche, J.-Y. L'Excellent, and G. Utard. On the Memory Usage of a Parallel Multifrontal Solver. In *IPDPS'03*, Nice, April 2003.

[21] Olivier Cozette, Cyril Randriamaro, and Gil Utard. Improving Cluster IO Performance with Remote Efficient Access to Distant Device. In *IEEE Workshop on High Speed Local Network (HSLN'02)*, Tampas, Florida, US, November 2002.

[22] E. Caron and G. Utard. Parallel Out-of-Core Matrix Inversion. In *IPDPS'02. The 16th International Parallel and Distributed Processing Symposium*, Fort Lauderdale, April 2002.

[23] A. Guermouche, J.-Y. L'Excellent, and G. Utard. Impact of Sparse Matrix Reordering Techniques on the Memory Usage of a Parallel Multifrontal Solver. In *Proceedings of the 2nd International workshop on Parallel Matrix Algorithms and Applications (PMMA'02), November 9-10*, 2002.

[24] Jonathan Ilroy, Cyril Randriamaro, and Gil Utard. Improving MPI-I/O performance on PVFS. In Rizos Sakellariou, John A. Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par 2001 : Parallel Processing, 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings*, volume 2150 of *Lecture Notes in Computer Science*, pages 911–915. Springer, 2001.

[25] E. Caron, D. Lazure, G. Utard. Performance Prediction and Analysis of Parallel Out-of-Core Matrix Factorization. *In : HiPC'2000, IEEE and ACM Intl. Conf. on High Performance Computing*, Bangalor, India, December 2000.

[26] E. Caron, D. Lazure, G. Utard. Inversion of Huge Matrices on Cluster. *In : Cluster'2000, IEEE Intl. Conf. on Cluster Computing*, Dresde, Germany, December 2000.

[27] Hakan Taki and Gil Utard. MPI-IO on a parallel file system for cluster of workstations. In *International Workshop on Cluster Computing (IWCC '99), 2-3 December 1999, Melbourne, Australia*, pages 150–157. IEEE Computer Society, 1999.

[28] E. Caron, O. Cozette, D. Lazure, G. 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*, Amsterdam, Netherland, 11-13 April 1999. Springer LNCS 1593.

[29] E. Cagniot, M. Cosnard, T. Peugeot and G. Utard. SOPHIE : a Tool for Collecting PHiPAC Metrics of C Code. *In : ICVS'99, International Conference on Vision Systems, Workshop on Performance Characterisation and Benchmarking of Vision Systems*. Las Palmas de Gran Canaria, Canary Island Spain, January 1999.

[30] A. Agahi, R. Russel and G. Utard. A High Performance Modular File System. In *HPCN'98, High Performance Computing and Networking Europe*. Amsterdam, April 1998. Springer LNCS 1401.

[31] G. Utard. Compilation of Data-Parallel Programs into PVM code for Local Area Network of Workstations. *In : EuroPVM'95 – Second Euro PVM users' Group Meeting*. Lyon, september 1995.

— Conférences francophones avec comité de lecture

[32] E. Caron, D. Lazure and G. Utard. Modélisation et optimisation de la factorisation LU out-of-core. In *12iemes Rencontres sur le Parallélisme (RenPar12)*, Besançon, 19-22 juin 2000.

[33] E. Caron, O. Cozette, D. Lazure and G. Utard. Mémoire virtuelle et parallélisme de données. In *11iemes Rencontres sur le Parallélisme (RenPar11)*, Rennes, 9-11 juin 1999.

[34] E. Cagniot, M. Cosnard, T. Peugeot and G. Utard. SOPHIE : un Outil pour la collecte des métriques PHiPAC de Code C. In *10iemes Rencontres sur le Parallélisme (RenPar10)*, Strasbourg, 9-12 juin 1998.

[35] P.J. Hatcher, C. Perez, and G. Utard. Équilibrage de charge de programmes data-parallèles par migration de threads. In *9iemes Rencontres sur le Parallélisme (RenPar9)*, EPFL, Lausanne, Suisse 20-23 mai 1997.

— Conférences internationales sans comité de lecture

[36] J.-F. Collard et G. Utard. Automatic Data Layout and Code Restructuring for Out-of-core Programs. In *Proceeding of the Workshop on Out-of-Core Computation COCA'98*, Cap-Hornu Saint-Valery-sur-Somme, France, September 15-17, 1998.

[37] E. Caron, D. Lazure et G. Utard. Virtual memory organization in data-parallel programs. In *Proceeding of the Workshop on Out-of-Core Computation COCA'98*, Cap-Hornu Saint-Valery-sur-Somme, France, September 15-17, 1998.

— Principaux rapports de recherche, etc.

[38] Gil Utard et Cyril Randriamaro. *Système et procédé de sauvegarde distribuée pérenne.* Brevet Numéro 04/52788, déposé le 26 novembre 2004.

[39] J.-F. Collard et G. Utard. *Proceeding of the Workshop on Out-of-Core Computation COCA'98*, Cap-Hornu Saint-Valery-sur-Somme, France, September 15-17, 1998. Diffusé comme Rapport interne du LaRIA (98-10) et du PRiSM (RR-98/045).

[40] E. Caron, F. Desprez, E .Fleury, D. Lazure et G. Utard. Interface SciLab/ScaLAPACK "*out-of-core*". Document de travail interne de l'action coopérative OURAGAN de l'INRIA.

Deuxième partie

Cluster Computing et traitement de donnée de grande taille

Chapitre 3

Présentation

Mon premier axe de recherche a été l'étude de l'usage du cluster computing pour le traitement de grande masse de données. J'ai eu une approche verticale de la problématique, de l'architecture à l'applicatif, dans laquelle j'ai eu des résultats dans plusieurs thématiques qui vont être présentés dans cette partie.

J'ai obtenu des subventions internes et régionales pour le financement des thèses et l'acquisition d'un cluster d'Alpha basé sur un réseau Myrinet pour les expérimentations. Ce cluster a par la suite été étendu par un financement de l'Université pour qu'il puisse être utilisé par d'autre départements, tels que la chimie biomoléculaire pour le calcul des structures 3D de protéine. Un des outils que nous avions étudiés pendant ces travaux a aussi été utilisé dans le cadre d'un contrat avec le CETMEF (simulation de la houle dans les constructions portuaires, modélisation par éléments finis).

Je présente mes travaux selon trois axes, qui feront chacun l'objet d'un chapitre.

— La gestion des E/S parallèle, dans laquelle nous avons étudié le partage des disques à travers les réseaux à haut débit.
— L'optimisation des algorithmes de calcul numérique direct dans le cadre du calcul out-of-core, dans le cas des systèmes dense et creux.
— Une proposition de mécanisme de gestion de la mémoire virtuel en mode utilisateur, et son application dans le cas du calcul numérique creux.

3.1 $READ^2$, optimisations accès disques

Ce premier axe de travail est issu d'une lecture du papier "The architectural costs of streaming IO : A comparison of workstations, cluster and SMD" de Patterson et al. [Pat98]. Dans cet article les auteurs mettaient en évidence que les principales limitations des performances des applications bornées par les entrées/sorties étaient dues aux bus I/O et aux contentions d'accès sur le bus mémoire. La direction préconisée par les auteurs étant d'augmenter la capacité de ces derniers.

Dans le même temps, j'avais suivi aussi les développements middleware autour des cartes réseau à haut débit tel que myrinet, dans lesquelles on exploitait les possibilités de reprogrammation de celles-ci pour mettre en place des méthodes de communication avec zéro copie, permettant de réduire drastiquement les temps de la latence. Notamment, les travaux de Loic Prylli.

C'est donc tout naturellement que m'est venue l'idée d'étudier comment améliorer les performances d'applications d'entrées/sorties intensives en ayant une approche centrée sur les bus, ceci grâce aux nouvelles capacités proposées par les cartes de réseaux programmables.

Ce travail était en partie l'objet de la thèse d'Olivier Cozette, que j'ai dirigée de 1999 à 2002, actuellement *senior developper* chez Google après un passage cher ARM et Apple.

Les deux articles qui suivent (4,[21];5,[19]) présentent donc les résultats de ces travaux, qui sont passés notamment par le développement de prototype logiciel pour une validation expérimentale sur cluster.

Le premier article présente une première analyse des gains qui pourraient être obtenus par cette méthode, ainsi qu'une première validation expérimentale utilisant des cartes graphiques pour simuler les disques.

Le deuxième article a été réalisé quand nous avons pu acquérir des disques à haut débit (pour l'époque), et pour lesquels le prototype complet à pu être réalisé, notamment par l'intégration des drivers SCSI dans les cartes myrinet. Ce papier présente l'implémentation et les premiers résultats expérimentaux.

3.2 Algorithme Out-of-Core parallèle pour le calcul numérique.

D'un point de vue algorithmique, je me suis dans un premier temps intéressé à la résolution de grands systèmes denses, que l'on trouve dans les modélisations d'électromagnétisme, dont la taille dépasse la mémoire disponible des machines, et qui doit donc faire appel à la technique du calcul out-of-core. Ce travail a fait l'objet de la thèse d'Eddy Caron.

Nous sommes parties de l'étude de ScaLaPACK, une bibliothèque de calcul numérique parallèle. Les performances de ScaLaPACK sont obtenues en grande partie par l'optimisation de la hiérarchie mémoire par le découpage en blocs des problèmes à traiter. Cette bibliothèque était donc un bon point de départ pour atteindre de bonnes performances dans le cadre out-of-core.

Nous avons donc étudié le schéma de calcul out-of-core de ScaLaPACK, qui est une extension de l'algorithme right-looking appelé left-right looking. Nous avons fait une analyse formelle des performances attendues, et nous avons montré quand mettant un schéma de recouvrement du calcul par les entrées/sortie, les temps d'exécution n'étaient pas impactés par ces dernières. Résultat que nous avons implémenté et validé expérimentalement.

Ce travail fait l'objet de l'article qui suit (6,[3]). Il a été étendu au problème de l'inversion de systèmes denses.

Après le cas dense, je me suis intéressé à la problématique de la factorisation dans le cas creux. C'est le travail qui à fait l'objet de la thèse d'Abdou Guermouche en collaboration avec Jean Yves l'Excellent lors de mon détachement INRIA à l'ENS Lyon en 2001-2003.

La plupart des systèmes physiques modélisés passent par la résolution de système creux, les systèmes denses étant en général utilisés pour les problèmes d'électromagnétismes. Bien qu'elles peuvent être de densités faibles, les systèmes étudiés peuvent être limités par la mémoire de par leur taille. Par exemple, j'ai collaboré avec le CETMEF [1] pour la résolution de grands systèmes dans le cadre de la simulation de la houle dans les ports maritimes.

Nous nous sommes donc intéressés à la consommation mémoire dans l'une des méthodes de résolution directe de systèmes creux séquentiels et parallèles, à savoir la méthode multifrontale, et en particulier la solution MUMPS.

Dans un premier temps, nous nous sommes intéressés à l'étude du comportement mémoire de la méthode multifrontale. Jusqu'alors, la première technique pour réduire les calculs, et la consommation mémoire est d'employer des heuristiques de renumérotation, dites aussi de permutation, des matrices afin de limiter le remplissage. Nous avons donc étudié la dynamique de l'utilisation mémoire en fonction des heuristiques. Nous avons proposé une stratégie d'or-

1. Centre d'Etude Technique Maritime et Fluvial

donnancement des calculs qui minimise les pics de consommation mémoire. Le résultat est présenté dans l'article suivant (7,[2]).

3.3 Pagination adaptative

Fort ce cette étude, dans un second temps nous nous intéressé comment optimiser les performances quand la mémoire physique est insuffisante pour contenir l'ensemble des données pendant les calculs et que l'on doit faire appel à la pagination. Cependant, nous avons constaté que la stratégie de pagination de type LRU n'était pas adaptée dans ce cas.

L'idée était de combiner le travail d'Abdou Guermouche avec un autre travail système d'Olivier Cozette. Nous avions développé un mécanisme de gestion de la pagination en mode user, appelé MUMM/MMUSSEL. Nous avons donc conçu un moniteur de pagination spécifique qui grâce aux directives transmises par le solveur, permet de définir différentes stratégies de gestion de la pagination pour chaque type de zone mémoire identifié dans l'analyse de l'accès mémoire du solveur, et ainsi de réduire de manière significative les défauts de pages et les IO, et ainsi accélérer les temps de traitement. Ce travail est présenté dans le dernier article de cette partie (8,[18]).

Improving Cluster IO Performance with Remote Efficient Access to Distant Device*

Olivier Cozette, Cyril Randriamaro
PaLADIN – LaRIA
Université de Picardie Jules Verne
80000 Amiens, France
cozette@laria.u-picardie.fr

Gil Utard INRIA ReMaP Project – LIP
École Normale Supérieure de Lyon
69364 Lyon Cedex 07, France
Gil.Utard@ens-lyon.fr

Abstract

Several Grand Challenge application softwares in biology or physic process large datasets which are stored on disks. These applications require high performance IO systems. Cluster computing is a good approach to build IO intensive platform for low cost: several clusters won many sorting benchmarks (MinuteSort, Datamation)! Much progress has been made in IO components like disk, controller and network: today incredible IO performance can be achieve by cluster. The counterpart of this technological advancement is that much stress is put on the different buses (memory, IO) of each cluster's node. The bandwidth of these buses are fixed. So the cluster IO performance is bounded and cannot be scaled by addition of IO components. In this paper we investigate a technic we called READ^2 *which stand for Remote Efficient Access to Distant Device. The aim of this technique is to reduce the stress which is put on busses of cluster's node during the execution of IO streaming applications using parallel IO. In* READ^2 *any cluster's node directly access to a remote disk of a distant node: the distant processor and the distant memory are removed from the control and data path. With this technique, a cluster can be considered as a* shared disk *architecture instead of a* shared nothing *one, and may inherit works from the SAN community. This paper presents what are the architectural benefit of* READ^2*, i.e. a better use of IO and memory buses which eventually improve the IO scalability of a cluster and the performance of streaming application.*

1 Introduction

Grand challenge applications often process large datasets which require high performance IO systems. For example, the amount of data processed at the European particle accelerator (LHC/CERN), reaches several Petabytes per year [7]. To deal with such datasets, cluster architecture based on commodity components can be designed: disk drives are aggregated to provide a large parallel file system. Hence, San Francisco museum uses 20 PCs with 368 disks to manage a 3,2 Terabytes digitalized picture collection [15]. Thanks to parallel accesses, high performance IO can be achieve: several clusters won many sorting benchmarks (MinuteSort, Datamation) [1]!

Usually, a commodity cluster is considered to be a *shared nothing architecture*. In particular, to share data each cluster's node must behave like a server for other nodes. For example, for distributed parallel file systems like PVFS (Parallel Virtual File System [10]) or PPFS (Portable Parallel File System [8]), each cluster's node is burden to serve local data requested by distant node. Whereas good performance may be obtained for homogeneous collective parallel IO by using adequate placement and redistribution schemes [9], the overhead is non negligible for general access. It can be decomposed in two parts: the first overhead is in the operating system running on each node, the second overhead is in the hardware architecture. In this paper we focus on the hardware overhead.

Thanks to some technological advancement, disk drives are able to achieve up to 90MByte/s of sustained bandwidth, disk controllers can achieve several hundred MByte/s of bandwidth, and network cards

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 IEEE

can achieve several Gigabit/s bandwidth! By adding several IO components to each node, it is possible to get plenty of IO bandwidth for data intensive applications. However, an increase of the IO bandwidth put more pressure on the IO and memory buses of each node. Unfortunatly, these buses cannot be scaled, so the maximun bandwith is bounded. An alternative is to use NAD (Network Attached Devices) where the disk are directly plug in the network. Several works, like GFS (Global File System [13]) or NASD (Network Attached Secure Disk [6]) proven the effectiveness of such technology. Unfortunately, this approach implies deployment of expensive network infrastructure like Fibre Channel.p

We investigate an alternative to NAD which is based on usual network technology we called READ^2 (Remote Efficient Access to Distant Device). In READ^2, we exploit the capability of modern network interface cards to directly drive and access IO device plugged in the same IO bus (usually a PCI bus). For instance, in [16] authors combine two cooperative Myrinet cards on the same IO bus for efficient IP forwarding: data throws directly from one Myrinet card to the second one, the processor is not involved in the data-path. In READ^2, we extend this technique for remote disk access.

In this paper we study what are the benefit of READ^2 access for streaming applications involving parallel IO. In a first part we present our architectural cluster model and we describe how READ^2 accesses are working. In a second part we propose a model for parallel streaming applications where a continuous stream of data are processed. This model is illustrated on two examples Then, we predict what is the benefit of READ^2 accesses in general and for the two examples. Finally we experimentally validate our model.

2 Architecture Model and read^2

In this section we present a cluster architectural model and describe READ^2 accesses.

2.1 Architectural model

A cluster may be considered to be an interconnection of different buses. A node is made up with:

An IO bus: (Input/Output bus) to connect network card, disk controller and IO bridge; it is usually the PCI bus.

A Memory bus: to connect memory to IO bridge.

A Processor bus: to connect processor to IO bridge.

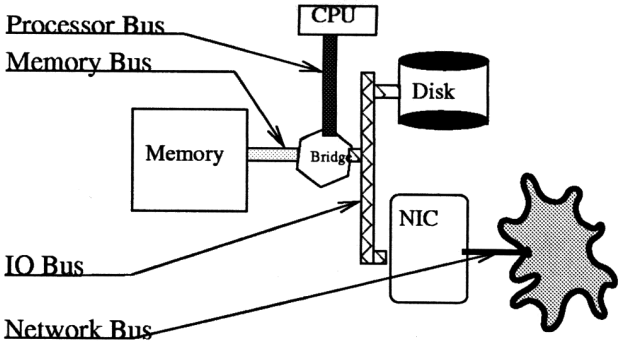


Figure 1. Architecture and variable description

Interconnection is sum up on Figure 1. These buses are characterized by the following constants:

M_d	maximum disk throughput
M_{io}	maximum IO bus throughput
M_m	maximum memory bus throughput
M_c	maximum instruction processing

Usually, disks are attached to the disk controller by another bus like SCSI bus: The global disk IO bandwith is the aggregated bandwith of disks. For the sake of simplicity, we don't consider this class of bus in this paper.

In a cluster nodes are interconnected by network interface card plugged in the IO bus: the network glues IO busses to build a parallel machine. From a logical point of view, the network may be considered to be another bus level. The network is characterized by the constant M_n which is the maximum network throughput.

2.2 READ^2: Remote Efficient Access to Distant Device

A Clusters is usually considered to be a *shared nothing* architecture: all nodes are independent and collaborate by message exchanges. The processor, memory and disk of each node are exclusively accessed by the local system. In fact it is a high level point of view of the network (session level). If we consider a lower level point of view (transport level) some components of each node (memory, disk) may be shared by all nodes. For instance the SCI network technology [14] or some Virtual Shared Memory implementation based on remote DMA, allow nodes to share memory.

Usually in parallel file system (e.g. PVFS), to share disk data each node behave like a data server for other

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 IEEE

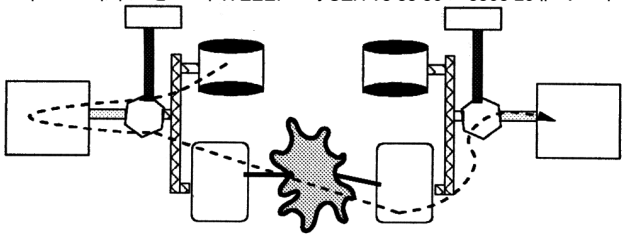


Figure 2. Remote read data path without READ^2

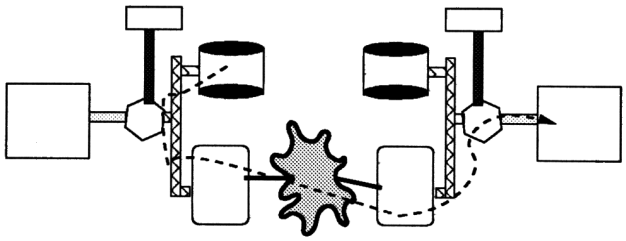


Figure 3. Remote read data path with READ^2

nodes: it is a *peer-to-peer server* approach (P2PS). A consequence is that when a node access data on a remote node, the remote memory bus and the distant IO bus are involved two times in the data path. Patterson et *al* in [2] demonstrated that cluster architectures are penalized by this overuse of the different buses. In particular for streaming applications, the IO bus is usually the bottleneck. Moreover, more stress is put on the memory bus by the file system wich may involve buffer copying policy. Some works like DAFS [4] try to remove this copying penalty by use of remote memory access. But DAFS always involves two traversals of the IO bus for the data path.

In this paper we consider another technique we called READ^2 (Remote Efficient Access to Distant Device). In READ^2 each nodes is able to directly access and control any remote disk: nodes share disks! A first consequence is the memory is not involved in the data path; a second consequence is the IO bus is involved only one time in the data path (Fig. 3, 2). We investigate what is the benefit of such an approach for streaming applications. A work close to our work is OPIOM ([5],[3]). It was designed for the implementation of a distributed Video on Demand server on Myrinet cluster. In OPIOM data transfers go from disk to network through the Myrinet card,

3 Streaming application model

We are considering parallel applications processing contiguous stream of disk data. Moreover the different part of the process (read, compute, communicate and write) are pipelined to achieve maximum overlapping. The local disk stream throughput is represented by the parameter $D = D_r + D_w$, where D_r is the throughput of the read data and D_w is the throughput of the write data.

The application is characterized by the following parameter (see Figure 1):

α: This parameter corresponds to the average cpu memory accesses for each datum of the local disk stream.

β: This parameter corresponds to the average data communicated to/from other nodes for each datum of the local disk stream. We assume that this communication is equidistributed between nodes. This measure is divided in two parts:

- β_d: This parameter represents the average communicated data of the local disk stream which is not involved or produced by the local computation.
- β_l: This parameter represents the average communicated data which are involved or produced by the local computation.

We have $\beta = \beta_d + \beta_l$.

γ: This parameter corresponds to the average cpu instruction for each datum of the local stream.

This parameters are illustrated for two applications: scan and transpose.

3.1 Scan

In this application, each node independently read its local disk and select some records which are wrote back to disk in another file. We assume that each record is composed of a key plus some data. The ratio between the key size and the record size is denoted by r. The fraction of records selected is denoted by f. So, $D_w = f \times D_r$ and $D = (1 + f) \times D_r$.

Because, there is no communication in this application, we have:

$$\beta = 0$$

For each record, the key is accessed by cpu to verify if the record is selected or not. So the average cpu memory accesses is denoted by:

$$\alpha = \frac{r}{1+f}$$

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 IEEE

We assume that the scan select record where key is in a fixed range. So there are two comparison instructions for each record, the average cpu usage is:

$$\gamma = \frac{2r}{1+f}$$

3.2 Transposition

Let n be the number of nodes in a cluster. A $M \times M$ matrix is distributed all over the nodes and stored by block of size $B \times B$, such as $nB = M$ (the memory of one node holds one block). Each block line is assigned to one node and stored on its local disks (cf Fig. 4). When a matrix is transposed, the i^{th} block of the j^{th} processor becomes the j^{th} block of the i^{th} processor. Each node must read $n-1$ blocks from the other nodes

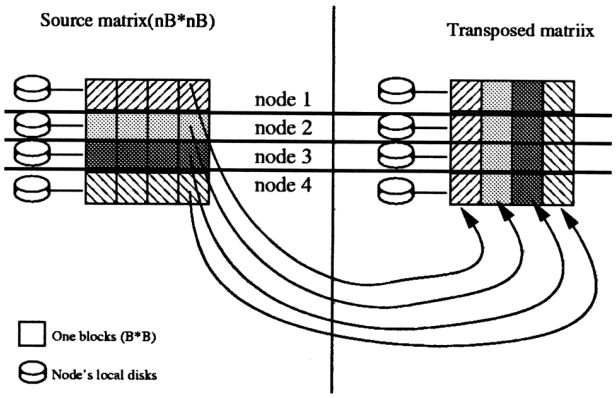


Figure 4. Parallel block transposition

and send them $n-1$ of its own blocks. One block per node remains local.

Let estimate parameters. We have $D_r = D_w$, the same amount of data is read and wrote, then $D = 2D_r = 2D_w$. For communication, each node send $\frac{n-1}{n}$ of the local input disk stream ($D_r = D/2$) and receive the same amount from other nodes for transposition. The transposed data are wrote on the local disk ($D_w = D/2$).

$$\beta_l = \frac{n-1}{n}/2 \qquad \beta_d = \frac{n-1}{n}/2 \qquad \beta = \frac{n-1}{n}$$

Each row must be transposed. So there is two cpu memory accesses for each datum (one load for the read stream and one store for the write stream), so we have

$$\alpha = \gamma = 1$$

4 Prediction of the bus usage

From the previous model, we now derive the bus usages of a streaming parallel application. Let B_{io} be the IO bus usage, B_m the memory bus usage, B_c the cpu usage and B_n the network usage. Let n the number of nodes. For a given streaming application characterized by the triplet (α, β, γ), we determine what is the used throughput of each bus for a streaming throughput $D = D_r + D_w$. We determine the different bus usage when the application use a classical *peer-to-peer* server (P2PS) approach for remote data access and when READ^2 is used. The Table 1 summarize the different bus usage

bus	P2PS	READ^2
B_{io}	$(1+\beta_l+\beta_d)D$	$(1+\beta_l)D$
B_m	$(1+\alpha+\beta_l+\beta_d)D$	$(1+\alpha+\beta_l-\beta_d)D$
B_c	γD	γD
B_n	$n \times \beta D$	$n \times \beta D$

Table 1. The different bus usage of a streaming application (α, β, γ)

This table clearly shows where is the benefit of READ^2.

- The IO bus usage is reduced: the data of local disk stream not involved or produced by the local computation cross the IO bus one time only, the benefit is equal to $\beta_d D$.
- There are less contention on the memory bus: the data of local disk stream not involved or produced by the local computation don't cross the memory bus, the benefit is equal to $2\beta_d D$.

So, this two main gains allow us to increase the maximum bandwidth of disk stream when the IO bus is the bottleneck. At the same time we get more memory traffic for local computation. Note that for applications where $\beta_d = 0$ (e.g. scan) there is no gain.

Bus usage prediction for transposition

We estimate the benefit of READ^2 for the transposition. There are two benefits: The first one is an increase of the peak bandwidth achievable (D), the second one is a freeing of the memory bandwidth.

Peak bandwidth of the transposition

Peak bandwidth is reached when one resource (cpu, bus, network) reaches its limit, that is $M_m = B_m$ or $M_{io} = B_{io}$ or $M_c = B_c$:

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 IEEE

- Without READ^2 ,

$$\left(\frac{M_m}{1+\alpha+\beta_l+\beta_d}, \frac{M_{io}}{1+\beta_l+\beta_d}, \frac{M_c}{\gamma}\right)$$

- With READ^2 ,

$$\left(\frac{M_m}{1+\alpha+\beta_l-\beta_d}, \frac{M_{io}}{1+\beta_l}, \frac{M_c}{\gamma}\right)$$

Hence, transpose maximum disk throughput is:

- Without READ^2,

$$D = min\left(\frac{M_m}{2+\frac{n-1}{n}}, \frac{M_{io}}{1+\frac{n-1}{n}}, M_c\right)$$

- With READ^2,

$$\left(\frac{M_m}{2}, \frac{M_{io}}{1+\frac{n-1}{2n}}, M_c\right)$$

The majority of architectures assume $M_{io} < 2M_m$ and $M_c >> M_m$:

- Without READ^2

$$D = \frac{M_{io}}{1+\frac{n-1}{n}}$$

- With READ^2

$$D = \frac{M_{io}}{1+\frac{n-1}{2n}}$$

Consider common bus bandwidth, ie $M_{io} = 133Mo/s$, $M_m = 800Mo/s$ and $M_c = 2000$ Mips, peak bandwidths achievable are:

- Without READ^2 , $D = 66MB/s$ (achievable with two 40 MB/s disks), $B_m = 190MB/s$ and network rate is $66MB/s$ per link (current Myrinet bandwidth is 200MB/s per link).

- With READ^2 $D = 86MB/s$ (achievable with three 40MB/s disk), $B_m = 172MB/s$ and network rate is $86Mb/s$ per link.

So direct remote IO improves by 25% the transposition rate.

Memory bus contention

The memory bus freeing by READ^2 is determined by β_d. As in the previous section, we assume the IO bus is saturated. With READ^2, although the peak bandwidth of the transposition is increased, and the memory bandwidth is less used than without READ^2:

- Without READ^2, $D = 66Mo/s$ and B_m $190MB/s$.

- With READ^2, $D = 82Mo/s$ and $B_m = 172MB/s$.

The gain of memory bandwidth is about 10%. This gain may be use by another application. This will be illustrated in the next section.

5 Experimental validation

To validate the previous model, the transposition application was implemented with and without READ^2 on a four-node Alpha cluster. Because at the time of this experiment, we didn't have high performance disk subsystem, it was emulated by the DMA of graphic cards. System characteristics are:

- A 340MB/s (M_m) bandwidth memory bus.

- A 130MB/s (M_{io}) bandwidth PCI bus.

- A Myrinet network card with a bandwidth greater than 100MB/s on each node, using the standard GM communication library [12].

- A high speed disks subsystem with a bandwidth greater than 80MB/s (M_r), emulated by a Matrox G200 graphics card.

5.1 READ^2 implementation

Without READ^2, the local graphic card (also called framebuffer) first writes data in the local main memory. Then data are stored in the local Myrinet card memory sent to the remote Myrinet card memory. Then data are stored in the remote main memory. They are finally written in the remote framebuffer after the transposition.

With READ^2, we modified the Myrinet GM standard communication library to let the framebuffer access to the Myrinet memory by DMA. The control flow is performed by the processors using pooling: the DMA was fired when Myrinet buffers was ready. The emulated disk bandwidth is controlled by waiting loops.

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 **IEEE**

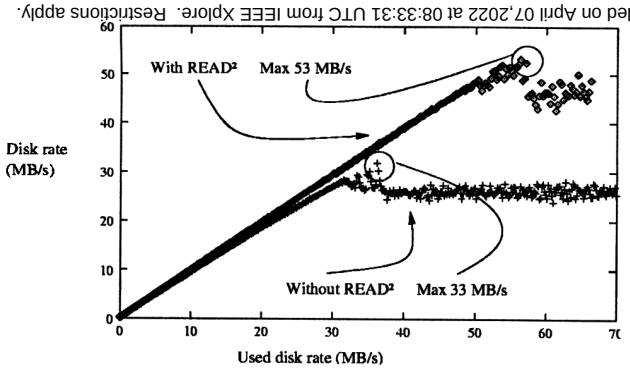


Figure 5. Maximum disk rate achievable with and without READ²

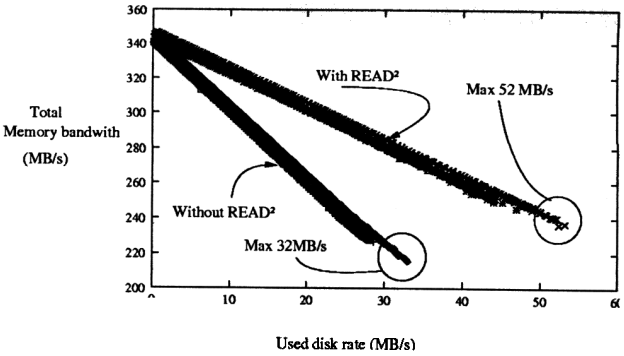


Figure 6. Memory contentions gains with and without READ²

5.2 Experimental Results

We measured the transposition data exchange rate for different disk bandwidths, with and without READ² on four nodes. Figure 5 shows the meazured bandwidth on one node. Raw performance of the architecture is not achieve mainly because of the synchronisation overhead of our implementation. However, we get significant improvement:

- Without READ², the maximum disk rate used is 33 MB/s: it is not possible to improve the performance by the use of a speeder disk.
- With READ², the maximum disk rate used is 53 MB/s by an add of disk.

So, READ² improve the IO scalability, i.e. it is possible to get better bandwidth. rate.

To exhibit memory contention, a second experience introduces the execution of a concurrent application which requires only main memory access and cpu without IO accesses (e.g. filter application on a picture in main memory). Experience results are displayed in Figure 6. Plots show what are the used memory bandwith (so the speed) of the concurrent filter application according to the transposition rate. In both situations, with and without RAID², the concurrent filter process does not modify the transposition peak bandwidth because we gave priority to the transposition application.

- Without READ²: Memory contentions appear when disk rate increase, so memory bandwidth available for filter and transposition decrease.
- With READ²: They are less memory contentions because READ² needs less main memory bandwidth for remote disk accesses.

Thanks to READ², memory bandwith is not wasted and can be use to improve performance of another concurrent application. This means than we can run concurrently a memory intensive and an IO intensive application with a better overall throughput of the system.

6 Conclusion

In this paper we investigated what are the benefits of direct remote access to distant device, called READ², for streaming applications on cluster. We proposed a model for streaming applications to derive bus usages: there are two main gains. The first is an increase of the peak performance for streaming applications. The second is a reduction of bus memory usage and then memory contention. These benefits were experimentally validated on a cluster where high performance disks was emulated by graphic cards.

The next step of our work is a validation of our model with real disks. Currently we have embedded a READ² SCSI driver in the GM firmware of Myrinet cards. We are now able to directly access to any disk from any node in the cluster: a remote disk is *mounted* as a local disk device and accessed as usual by standard IO primitives. The main difference with previous works like OPIOM [5] or Maierhofer thesis [11], is the control path is also embedded in the myrinet firmware. This implies new control flow mechanism to deal both with the communication and disk IO traffic. This implementation will be describe in a next paper.

With READ², a cluster may be considered to be a *shared disks architecture*, where all disks may be accessed by any node like Network Attached Storage Device, without using a specific network. Another work in progress is to use READ² for an implementation of the Global File System [13] on cluster. GFS was im-

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 IEEE

plemented on Fibre Channel technology where disks are directly accessed on network. GFS was ported on cluster architectures with one data server on each node (P2PS approach). We plan to analyse what will be the benefit of our new READ^2 implementation. At the application level, we plan to study the benefit of such an architecture for an external two-pass parallel sorting algorithm.

References

[1] Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David Patterson. High performance sorting on networks of workstations. In *SIGMOD'97*, Tucson, Arizona, May 1997.

[2] 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.

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

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

[5] 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.

[6] G. A. Gibson and R. Van Metter. Network attached storage architecture. *Communication of The ACM*, 43(11), November 2000.

[7] 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.

[8] 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.

[9] Jonhatan Ilroy, Cyril Randriamaro, and Gil Utard. Improving MPI-IO Performance on PVFS. In *EuroPar'2001*, Manchester, UK, August 2001.

[10] 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.

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

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

[13] 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.

[14] Henri E. Bal Raoul A.F. Bhoedjang, Tim Ruhl. *Scalable Coherent Interface.* November 1998.

[15] 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.

[16] S. Walton, A. Hutto, and J. Touch. Efficient high-speed data paths for IP forwarding using host based routers. In *10th IEEE Workshop on Local and Metropolitan Area Networks*, November 1998.

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 **IEEE**

READ^2: Put disks at network level

Olivier Cozette, Cyril Randriamaro and Gil Utard

Université de Picardie Jules Verne, LaRIA
CURI, 6 rue du Moulin Neuf
80000 Amiens, France
and "Pôle de Modélisation de la Région Picardie"
and "Fonds Social Européen"
e-mail: {cozette,crandria}@laria.u-picardie.fr

INRIA ReMaP Project
Laboratoire de l'Informatique du Parallèlisme
École Normale Supérieure de Lyon
69364 Lyon Cedex 07, France
e-mail: Gil.Utard@ens-lyon.fr

Abstract

Grand challenge applications have to process large amounts of data, and then require high performance IO systems. Cluster computing is a good alternative to proprietary system for building cost effective IO intensive platform: some cluster architectures won sorting benchmark (MinuteSort, Datamation)! Recent advances in IO component technologies (disk, controller and network) let us expect higher IO performance for data intensive applications on cluster. The counterpart of this evolution is that much stress is put on the different buses (memory, IO) of each node which cannot be scaled. In this paper we investigate a strategy we called READ^2 *(*Remote Efficient Access to Distant Device*) to reduce this stress. With* READ^2 *any cluster node accesses directly to remote disk: the remote processor and the remote memory are removed from the control and data path: Inputs/Outputs don't interfere with the host processor and the host memory activity. With* READ^2 *strategy, a cluster can be considered as a* shared disk *architecture instead of a* shared nothing *one. This papers describes an implementation of* READ^2 *on Myrinet Networks. First experimental results show IO performance improvement.*

1. Introduction

Grand challenge applications often process large amounts of data which require high performance IO systems. For example, at the European particle accelerator (LHC at CERN [8]), the amount of data to process reaches several Petabytes per year. To deal with such applications, cluster architecture based on commodity components can be designed, where disk drives are aggregated to provide a large parallel file system. For instance, the San Francisco Museum uses 20 PCs with 368 disks to handle a 3.2 Terabytes digital picture collection [15]. At the performance point of view, some cluster architectures won sorting benchmark (MinuteSort, Datamation) [1]!

Usually a commodity cluster is qualified to be a *shared nothing architecture*. A consequence is that to share data, each node must act as a server to other nodes. For instance in distributed parallel file systems like PVFS (Parallel Virtual File System [13]) or PPFS (Portable Parallel File System [9]), each node is burden to serve local data requested by another node. Good performance may be obtained for homogeneous collective parallel IO by using adequate placement and redistribution schemes [11].

Thanks to technological advancement for IO components, some disk drives are able to achieve up to 90MByte/s of sustained bandwidth, disk controllers deliver several hundred MByte/s of bandwidth, and network cards provide several Gigabit/s bandwidth! By adding several IO components in each node, it is possible to get plenty of IO bandwidth for data intensive applications. However, this large IO bandwidth puts pressure on the IO and memory buses of each node, which cannot be scaled. An alternative is to use NAD (Network Attached Devices). Several works, like GFS (Global File System [14]) or NASD (Net-

Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)
0--7695-1919-9/03 $17.00 © 2003 IEEE

work Attached Secure Disk [7]) proven the effectiveness of such technology. Unfortunately, this approach implies the deployment of a specific network infrastructure like Fibre Channel.

We investigate an alternative to NAD which is based on usual network technology and a new method to access remote data called READ^2 (Remote Efficient Access to Distant Device). In READ^2, we exploit the capability of modern network interface cards to directly drive and access IO device plugged on the same IO bus (usually PCI). For instance, in [16] authors combine two cooperative Myrinet cards on the same IO bus for efficient IP forwarding: the data throw directly from one Myrinet card to the second one and the processor is not involved in the data-path, increasing the peak bandwidth of the forwarding scheme. In READ^2, we propose to extend this strategy for remote disk access.

In previous work [4], we studied the bus usage benefits of READ^2 for streaming applications. This papers describes an implementation of READ^2 on Myrinet Networks. After a description of the target architecture, we present the READ^2 model. Then we describe different strategies to implement READ^2. Finally, we show the performance of READ^2 and compare it to PVFS and NBD.

2. READ^2 model

Cluster is usually considered as a *shared nothing* architecture: each node is independent and collaborates with other nodes by message exchanges. The processor, memory and disk of each node are only accessed by the local system. In fact this is a high network level point of view (session level). If we consider a lower level network point of view, i.e. transport level, some components of each node (memory, disk) can be shared by all nodes. For instance the SCI network technology [3] or some Virtual Shared Memory implementations based on remote DMA, allows nodes to share memory.

In fact, a cluster may be considered as an interconnection of different buses. A node is composed of (Figure 1):

An IO bus: (Input/Output bus) to connect network card, disk controller and IO bridge; it is usually a PCI bus.

A Memory bus: to connect memory to IO bridge.

A Processor bus: to connect processor to IO bridge.

Disks are attached to the disk controller by another bus like SCSI bus. We don't consider this bus in this paper, we consider only their aggregated bandwidth. Cluster nodes are interconnected by network interface card plugged on the IO bus: the network glues IO busses to build a parallel machine. From a logical point of view, the network may be considered as another bus level.

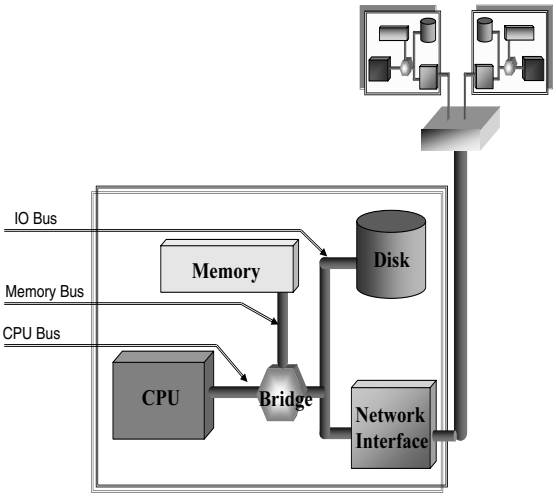


Figure 1. Cluster Architecture

Usually to share disk data in a parallel file system like PVFS, each node must act as a data server for other node: it is a *peer-to-peer server* approach (P2PS). A consequence is when a node accesses data on a remote node; memory bus and IO bus of the remote node are involved two times in the data path. Patterson et *al* in [2] demonstrate that for streaming applications, cluster architectures are penalized by this overuse of the different buses, specially the IO bus which is the bottleneck of current architectures. Moreover, more stress is put on the memory bus by the local file system which may involve buffer copying policy. Some works like DAFS [5] try to remove this copying penalty by direct remote memory access, but always involves two traversals of buses in the data path.

In this paper we investigate another scheme to share disks in a cluster: each node is able to directly access and control any remote disk. A consequence is that for each remote disk access, the memory is not involved in the data path, and the IO bus is involved only one time.

3. Implementation strategies

The common feature of the different implementation schemes we consider in this section is on the data path: when a client node read from a remote disk of the server node, then data go from the disk to the server NIC (Network Interface Card) and then to the client. For a write, data go from the client to the remote NIC and then to the remote disk.

In this section we describe different strategies to implement the control path, which can be characterized by the location of the driver of the remote disk for the request.

Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)
0--7695-1919-9/03 $17.00 © 2003 IEEE

3.1. Server driver (driver in kernel server node)

Usually, the server reads data from its disk to kernel buffers in main memory. Nevertheless, direct access implies to write data directly to the network interface. Hence, a straightforward implementation is to put the kernel buffers in the network interface memory. So, the control of disks stay in the server kernel.

For each client remote read request, the kernel part of the server is noticed of which data must be read. Then the kernel fire its local disk to read the requested data. Because the kernel buffers is put on the memory of the NIC, the disk controller put the read data on it. Once the data is read, the server kernel can fire the send of the data to the client. This is the strategy used by OPIOM [6], a work close to ours which was designed for distributed Video on Demand server on Myrinet cluster.

This strategy is easy to implement, but it main drawback is to requiring a lot of kernel processing time.

3.2. Driver with remote IO port access (driver in kernel client node)

A second strategy is to reuse the standard disk drivers and modify it in such a way it is able to drive the remote controller. This strategy can be achieve by providing remote IO port access.

Usually, a disk driver works in the following way: the drivers builds read or write requests structures in memory, then fire the controller by IO port. The controller reads requests from memory and executes it. Once the requests are completed, the controller put status information in memory. A solution to control the remote disk by the client is:

- put the request structure on the distant node (in NIC main memory) in such a way that the distant controller can accesses it,
- fire the remote controller thanks the remote NIC,
- once the request is completed, states information is send back to the client.

The second point requires for the client an access to distant IO port, which is not always accessible by the NIC. Fortunately the IO ports are usually mapped on memory, so the distant controller may be fired by a remote memory access.

This strategy reuses the Linux driver, so it is easy to adapt it for another disk (controller). But, it must use lot of small messages for the remote access to IO ports. Another main drawback of this strategy is that mutual exclusion mechanism must be introduced to deals with multiples clients. On the one hand,

3.3. NIC Driver (driver in Network Interface server node)

An alternative to the previous strategy is to put the driver in the NIC. Only two messages are sent across the network: the request message and the disk data.

It combines the advantage of the two precedent strategies: distant node is not burden by the client node, concurrent accesses are handle. But this strategy is harder to implement because it relies on specific disk controllers. We choose this last strategy and implemented it with one of the two most common SCSI controllers: the LSI 53c8xx compatible controller (the other is an Adaptec one).

4. READ^2 driver

We implemented this last strategy on the GM Myrinet communication Package: the READ^2 driver glues the GM message firmware and SCSI controller.

4.1. Short description of GM

GM is an efficient message-passing communication package to use Myrinet card designed to provide low latency and high throughput. GM runs in user mode (it does not use system call) and reduce the need of memory copies. GM is used with locked buffers in the main memory. To send and receive messages, the application writes the command in the Myrinet card. Hence, the Myrinet card uses locked uffers without any operating system help.

GM is made up of a Myrinet card firmware, a device driver, a user library. GM provides several ports on each node (like TCP) to enable several applications to share GM on the same machine. GM provides reliable in-order data delivery per port.

4.2. READ^2 and GM firmware

The GM Myrinet card firmware is called MCP (Myrinet Control Program). MCP is made up of four processes as presented in Figure 2: a sending process, a sending DMA process, a receiving process, a receiving DMA process.

Messages received by the Myrinet card are treated by the receiving process to ensure reliability (Fig. 2, ①), then written to main memory by the receive DMA process (②,③) using its target port information.

We reserved a port for READ^2. In-going message for READ^2 are intercepted at the receiving DMA process level (④, ⑤). Then block identifiers to read are sent to the SCSI controller (⑥). When data are ready, the SCSI controller writes it in Myrinet memory (⑦) and so READ^2 sends message with standard GM communication functions (⑧, ⑨).

Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)
0--7695-1919-9/03 $17.00 © 2003 IEEE

Notice that we reuse the standard GM sending and receiving processes; these processes handle the control flow and ensure the reliability of GM messages. So, the READ^2 control flow uses the GM control flow for *write* operation: when a data is accepted by the receiving node, the data will be written. For *read* operation, the control is more complex: when the command is received, the remote node stores the command in a queue command. If the READ^2 queue is full, READ^2 tells it to GM. Then after some retries from GM, the command is cancelled and the client can retry later. When a READ^2 successfully completes a reads, it gives a message to GM to be sent to the client. Nevertheless if the client could not receive the message, GM will drop it after several retries. But the client not know that the read was dropped, so after a timeout, the client sends a message to READ^2 to know all its pending reads in READ^2 queue.

4.3. SCSI controller and Myrinet Interface

Let describe relationship between the SCSI controller and the Myrinet card (Figure 3). Myrinet card can access main memory (or other memory mapped device) with its DMA engine. Also, it possesses local memory and an inboard processor called Lanai. READ^2 uses a part of the Myrinet memory to store the disk buffers, and another part to store the SCSI script. The SCSI script is insructions executed by the SCSI controller processor.

In first implementation, Myrinet card starts a request to SCSI controller by accessing the SCSI controller IO port (Figure 3, ①). This IO port starts the SCSI processor (②). The SCSI processor reads the request from the Myrinet memory and sends it to the disk (③). The SCSI processor receives the result from disk and write it to the Myrinet memory (③,④). Finally, the Lanai checks the IO port register to wait the end of the controller write in the Myrinet memory (⑤).

Using the DMA engine for IO port access has two inconvenient: first there is high latency because DMA engine was designed for large message. Second, to use DMA, regular GM send and receive process must be stopped. To avoid IO port access by DMA engine, we modified the script of the controller which now checks variable in the memory of Myrinet card to wait request (⑥).The Myrinet processor modify this variable to start a new request (⑦), and reads it to wait the request completion.

5. READ^2 usage

We defined a set of new GM functions to access remote device in user mode. Moreover, we develop a new block device for file system. These two interfaces are not interoperable, because the block device uses the Linux block

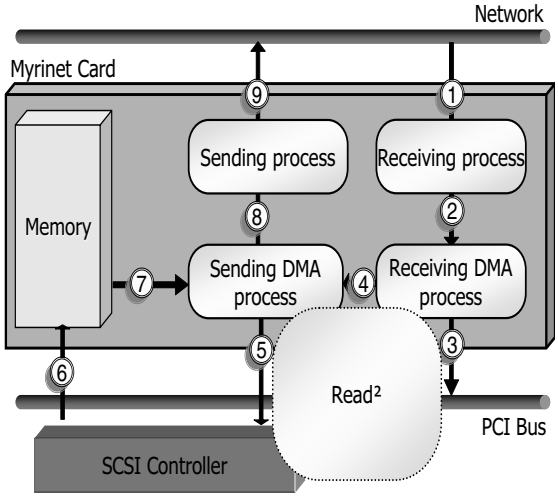


Figure 2. Send command with GM

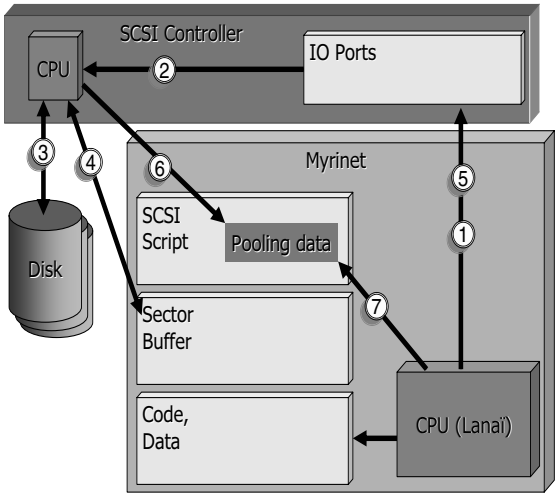


Figure 3. SCSI controller and READ^2

Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)
0--7695-1919-9/03 $17.00 © 2003 IEEE

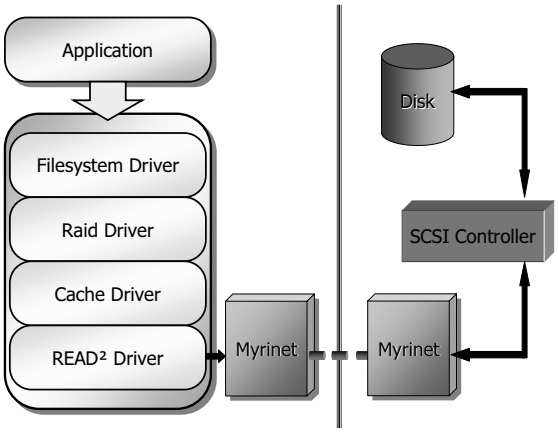


Figure 4. READ^2 driver in Linux kernel

cache whereas the user mode does not take care of cache, there is no coherency.

5.1. User mode functions

The remote disk is accessed as a raw disk: there is no file system. Two functions are provided to access READ^2 driver:

void read2_read$(node, sector, nb, buffer)$**:** read nb sectors of the remote disk $node$ starting at $sector\ node$. This function is asynchronous, `read2_wait` must be used to wait the completion of the request.

int read2_wait$(node, sector, nb, buffer)$**:** wait for a successful read, and give the last finished request.

These functions are based on the GM send and GM receives functions, they create a request message for the remote READ^2 driver and send it to the READ^2 port.

5.2. Block device

The previous interface was designed for fast raw IO, we designed also a block device to be able to deal with file-system: we can mount the standard extended 2 file-system on top of READ^2. When an application read data, the file-system driver request some blocks to the cache driver, if the blocks are not in the cache, then there is request to READ^2 driver, and the driver use the READ^2 user mode function to get the remote data.

6. Experimentation

READ^2 is designed to be the basic component of a high performance parallel file system for clusters. PVFS and GFS are the main available parallel file systems for clusters. A parallel file system is a complex architecture [10]: it must smartly distribute data over disks, it must provide fault tolerance mechanism, it must provide meta-data management, it must ensure coherency for concurrent accesses. However, all of them are based on a primitive mechanism to access remote data. For instance, PVFS uses a specific IO daemon (IOD) on each node, and GFS uses a specialized version of NBD (Network Block Device). NBD is a standard Linux mechanism to access to a remote disk through a local block device, as we do with READ^2.

A trivial way to make a primitive parallel file system is to use a remote disk access mechanism combined with the standard RAID mechanism provided by Linux. The remote disk access mechanism can be NBD or READ^2.

The first experimentation was to compare the performances of those two primitive parallel file systems. There are two measurements: single access performances (a single node accesses to all disks) and concurrent accesses performances (all nodes concurrently access to all disks).

Unfortunately, NBD does not provide concurrent access functionality. Hence, for concurrent accesses, we compared the primitive parallel file system built on READ^2 with PVFS.

NBD and PVFS communications are built over the socket interface. Sockets are built over the TCP/IP stack. The TCP/IP stack calls the GM driver. GM provides an efficient socket interface to bypass the TCP/IP stack: Socket-GM API. Nevertheless, that API is not available in kernel mode, so that it cannot be called by NBD or PVFS.

The following sections present the tests made on a cluster with 8 nodes (bi-Athlon). Nodes are interconnected by Myrinet cards (2 Gb/s). Each node has an Atlas 10KII Quantum SCSI-LVD disk with 40MB/s of measured peak read bandwidth.

6.1. Single access performances

Tests consist in reading by one node a 1GB file stored on PVFS, and on the two primitive parallel file systems based on NBD and READ^2. The file size is chosen big enough to avoid cache effects. We measured the read bandwidth according the number of remote disks.

The Figure 5 shows the results. The disk bandwidth increases with the addition of disks until 3 disks. With more than 3 disks we did not observe gain because of network limitation. Indeed, the message size used by those file systems is less than 10KB. We measured that such a size limits the maximum bandwidth Myrinet network to 110MB.

As expected, we got better performances with READ^2 than with NBD or PVFS, because READ^2 saves system resources. Because PVFS requires more resources, its performances are the worst.

Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)
0--7695-1919-9/03 $17.00 © 2003 IEEE

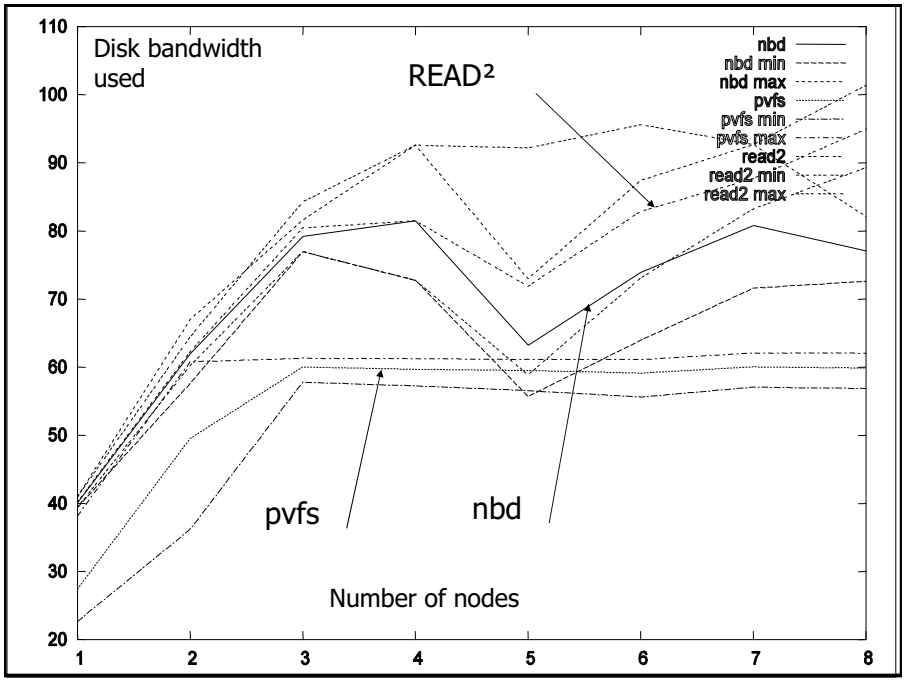


Figure 5. Single access performances

6.2. Concurrent access performances

The test is a concurrent reading by the 8 nodes of a shared 1GB file stored on PVFS, or on the primitive parallel file systems based on READ^2. We measured the aggregated read bandwidth according the number of remote disks.

The Figure 6 shows the results. For similar reasons to the previous experiment, PVFS performances are lesser than READ^2 one. Nevertheless, READ^2 bandwidth is limited by network contentions and SCSI contentions. Using queuing techniques at disk level can reduce SCSI contentions: the disk can reorder several requests to improve throughput.

7. Conclusion

We described the implementation of READ^2 on Myrinet network. We compared its performance with the block device NBD and the file system PVFS. For large accesses to parallel files, we got better performances than PVFS or NBD. We expect more gain with small accesses to parallel files, because READ^2 does not involve the kernel resource at the remote host.

READ^2 provides accesses to remote disks in a similar way to Fibre Channel, iSCSI or Infiniband ([12]) technologies. A next step of our work is to integrate READ^2 in the GFS file system which was devoted to such network disk technology.

References

[1] R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. Patterson. High performance sorting on networks of workstations. In *SIGMOD'97*, Tucson, Arizona, May 1997.

[2] R. H. Arpaci-Dusseau, A. C, Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. 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.

[3] R. A. Bhoedjang, T. Ruhl, and H. E. Bal. Scalable coherent interface. Technical report, Vrije Unniversiteit Amsterdam, November 1998.

[4] O. Cozette, C. Randriamaro, and G. 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.

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

[6] P. 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.

[7] G. A. Gibson and R. Van Metter. Network attached storage architecture. *Communication of The ACM*, 43(11), November 2000.

[8] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger. Data Management in an Internationnal Data

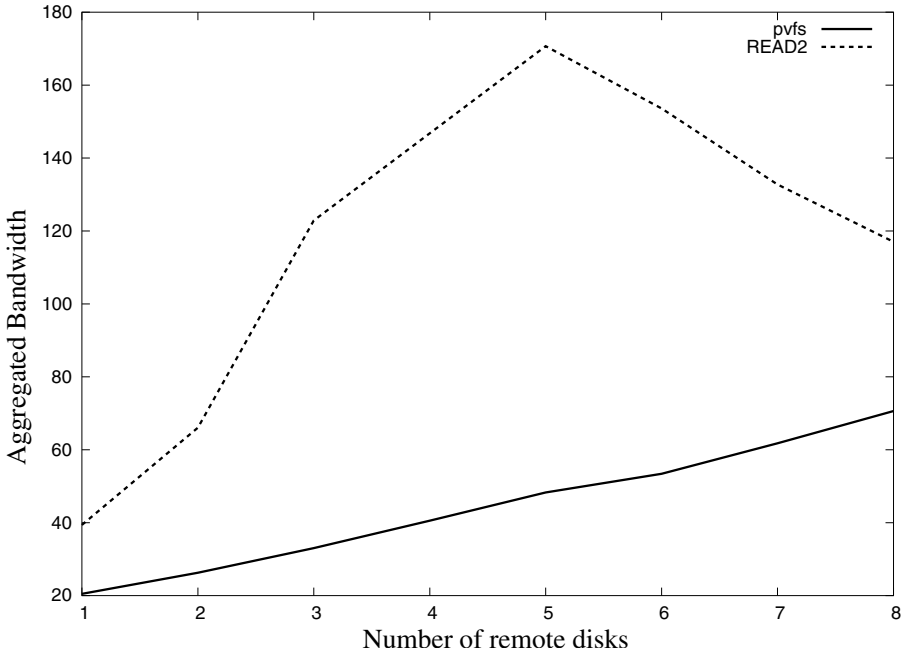


Figure 6. Concurrent accesses performance

Grid Project. In *GRID 2000*, Bangalore (Inde), 17-20 december 2000.

[9] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. 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.

[10] K. Hwang, H. Jin, and R. 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.

[11] J. Ilroy, C. Randriamaro, and G. Utard. Improving MPI-IO Performance on PVFS. In *EuroPar'2001*, Manchester, UK, August 2001.

[12] Infiniband Trade Association. Infiniband ^{TM} Architecture Release 1.0. http://www.infinibandta.org/, 2000.

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

[14] K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigland, and M. 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.

[15] N. Talagala, S. Asami, D. Patterson, B. Futernick, and D. Hart. The Art of Massive Storage: A Web Image Archive. *IEEE Computer Journal*, 33(11), November 2000.

[16] S. Walton, A. Hutto, and J. Touch. Efficient high-speed data paths for IP forwarding using host based routers. In *10th IEEE Workshop on Local and Metropolitan Area Networks*, November 1998.

Available online at www.sciencedirect.com
SCIENCE DIRECT®
Parallel Computing 30 (2004) 357–375
PARALLEL COMPUTING
www.elsevier.com/locate/parco

On the performance of parallel factorization of out-of-core matrices

Eddy Caron *, Gil Utard

GRAAL Project, INRIA Rhône Alpes, LIP Laboratory (UMR CNRS, ENS Lyon, INRIA, Univ. Claude Bernard Lyon 1), 46 Allée d'Italie, 69364 Lyon Cedex 07, France

Received 15 May 2003; received in revised form 13 February 2004; accepted 15 February 2004

Abstract

In this paper, we present an analytical performance model of the parallel left–right looking out-of-core LU factorization algorithm for cluster-like architectures. We show the accuracy of the performance prediction model for the ScaLAPACK library. We analyze the overhead introduced by the out-of-core part of the algorithm and we outline a limitation which was never seen before: for large problems the algorithm has a poor efficiency. This overhead is divided into an IO part and a communication part. We derive an overlapping scheme and minimum memory requirement to avoid the IO overhead. The new scheme is validated by a prototype implementation in ScaLAPACK. We show the impact of the communication overhead on two-dimensional distributions. Then we show that with similar memory requirements a second overlapping scheme may be implemented to avoid the communication overhead. If the size of the physical main memory is proportional to the matrix order ($O(N)$ bytes), then performance of the out-of-core algorithm is similar to that of the in-core algorithm which requires $O(N^2)$ bytes. This paper demonstrates that there is no memory limitation for the factorization of huge matrices.

Keywords: Matrix factorization; Out-of-core; Numerical library (ScaLAPACK)

1. Introduction

Many important computational applications involve solving problems with very large data sets [11]. For example astronomical simulation, crash test simulation,

* Corresponding author.
E-mail addresses: Eddy.caron@ens-lyon.fr (E. Caron), Gil.utard@laria.u-picardie.fr (G. Utard).

0167-8191/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2004.02.002

global climate modeling, and many other scientific and engineering problems can involve data sets that are too large to fit in main memory. Using parallelism can reduce the computation time and increase the amount of memory available, but for challenging applications memory size is always insufficient. Those applications are referred to as "parallel *out-of-core*" applications.

Dense matrix factorization may be used as a direct method to solve linear systems arising from boundary element and electro magnetic scattering problem. Because of the increasing demand of applications dealing with large matrices, it is very important to optimize this routine. To increase the memory size available, a trivial solution is to use the *virtual memory* mechanism of modern operating systems. Unfortunately, in [2] we shown that this solution is inefficient if standard *paging policies* are employed. To get the best performance, the algorithm must generally be *restructured* with explicit IO calls. In this paper, we present a study of such a restructuring for the dense LU factorization problem. More precisely we present an analytical performance model of the parallel left–right looking *out-of-core* algorithm which is used in ScaLAPACK [1]. The aim of this performance prediction model is to optimize the algorithm.

In Sections 2 and 3, we describe the LU factorization and the ScaLAPACK parallel version. In Section 4 we present the *out-of-core* LU factorization and in Section 5 the analytical performance model. In Section 6 we analyze the overhead of the algorithm and we describe a first overlapping scheme for IO overhead. Section 7 analyses the impact of distribution on performance. Section 8 introduces another overlapping scheme to avoid communication overhead and shows that out-of-core performance are similar to in-core ones.

2. LU factorization

The LU factorization of a matrix $A = (a_{ij})_{i \leqslant i,j \leqslant N}$ is the decomposition of A as a product of two matrices $L = (l_{ij})_{1 \leqslant i,j \leqslant N}$ and $U = (u_{ij})_{1 \leqslant i,j \leqslant N}$, such that $A = LU$ where L is *lower triangular* (i.e. $l_{ij} = 0$ for $1 \leqslant j < i \leqslant N$) and U is *upper triangular* (i.e. $u_{ij} = 0$ for $1 \leqslant i < j \leqslant N$).

A well known method for the parallelization of the LU factorization is based on the *blocked right-looking* algorithm. This algorithm is based on a *block decomposition* of matrices A, L and U:

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} \\ 0 & U_{11} \end{pmatrix}$$

This block decomposition gives the following equations:

$$A_{00} = L_{00}U_{00} \tag{1}$$

$$A_{01} = L_{00}U_{01} \tag{2}$$

$$A_{10} = L_{10}U_{00} \tag{3}$$

$$A_{11} = L_{10}U_{01} + L_{11}U_{11} \tag{4}$$

These equations lead to the following recursive algorithm:

(1) Compute the factorization $A_{00} = L_{00}U_{00}$ in Eq. (1) (possibly using another method).
(2) Compute L_{01} (respectively U_{10}) from Eq. (2) (respectively (3)). This computation can be done by a triangular solve (L_{00} and U_{00} are triangular).
(3) Compute L_{11} and U_{11} from Eq. (4):
  (a) Compute the new matrix $A' = A_{11} - L_{10}U_{01}$.
  (b) Recursively factorize $A' = L_{11}U_{11}$.

This algorithm is called *right-looking* because once the new matrix A' is computed, the left part (L_{00} and L_{01}) of the matrix is not used in the recursive computation. It is also true for the upper part (U_{00} and U_{10}). Moreover, it is easy to show that this computation can be done *data in place*: only one array is necessary to hold the initial matrix A and the resulting matrices L and U.

For numerical stability, *partial pivoting* (generally row pivoting) is introduced in the computation. Then, the result of the factorization consists of matrices L and U plus the permutation matrix P such that $PA = LU$.

In right-looking algorithm with partial pivoting, the factorization of A_{00} and the computation of L_{01} are merged in the first step. For the sake of presentation, we present an algorithm with partial pivoting (data in place) where row interchanges are applied in two stages.

1a. Compute factorization $P\begin{pmatrix} A_{00} \\ A_{10} \end{pmatrix} = \begin{pmatrix} L_{00} \\ L_{10} \end{pmatrix} U_{00}$ where P is a permutation matrix which represents partial pivoting: the left part of matrix A $\left(\text{i.e. is } \begin{pmatrix} A_{00} \\ A_{10} \end{pmatrix}\right)$ is factorized.
1b. Apply permutation P to the right part of matrix A $\left(\text{i.e. } \begin{pmatrix} A_{01} \\ A_{11} \end{pmatrix}\right)$.
2. Compute U_{01} from Eq. (2).
3a. Compute the new matrix $A' = A_{11} - L_{10}U_{01}$.
3b. Compute L_{11}, U_{11} and P' by a recursive call to factorization $P'A' = L_{11}U_{11}$ (P' is the permutation matrix.)
4. Apply permutation P' to the lower left part of matrix A (i.e. the L_{10} computed in the first step). Finally, return the composition of P and P'.

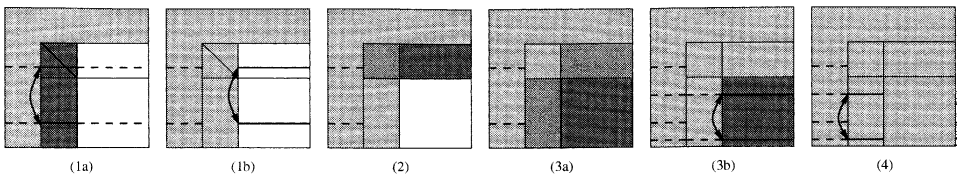


Fig. 1. A recursive call to the right-looking algorithm. Horizontal lines represent pivoting. Dashed lines represent part of rows which are not yet pivoted.

Fig. 1 shows the different steps for the second recursive call of the right-looking factorization.

3. Parallelization

In ScaLAPACK, the parallelization of the previous algorithm is based on a data-parallel approach: the matrix is distributed onto the processors and the computation is distributed according to the *owner compute rule*.

The matrix is decomposed in $k \times k$ blocks. As noticed above, at each recursive application of the right looking algorithm, the left and upper part of the matrix is factorized (modulo a permutation in the lower left part of the matrix). So, for *load balancing*, a cyclic distribution of the data is used.

The matrix uses *block cyclic* distribution on a (virtual) grid of p rows and q columns of processors. The *block decomposition* of the algorithm (shown in Fig. 1) corresponds to the *block distribution* of the matrix. So step 1a of the algorithm is computed by one column of p processors; step 2 is computed by one row of the q processors; step 3a is computed by the whole grid. Pivoting step 1b (respectively 4) is executed concurrently with computation step 1a (respectively 3b).

We now describe more precisely the different steps of the algorithm. Step la is implemented by the ScaLAPACK function `pdgetf2`, which factorizes a block of columns. For each diagonal element of the upper block (i.e. A_{00}) the following operations are applied:

(1) determine the pivot by a *reduce* communication primitive and *exchange* the pivot row with the current row;
(2) *broadcast* the pivot row on columns of processors;
(3) *scale*, i.e. divide, the column under the pivot by the pivot and update the matrix elements on the right of that column.

Step 2 of the algorithm is implemented by the ScaLAPACK function `pdtrsm`: the left-upper block (i.e. A_{00}) is *broadcast* to the processors row and this is followed by a (BLAS) triangular solve.

Step 3a is implemented by the ScaLAPACK function `pdgemm`: the blocks corresponding to U_{01} are broadcast on columns (of processors); the blocks corresponding to L_{10} are broadcast on rows (of processors); then the blocks are multiplied to update A'.

The performance of the parallel algorithm depends on the size of the block and the grid topology. The size of the block determines the degree and granularity of parallelism and also the performance of the BLAS-3 routines used by ScaLAPACK. The topology of the grid determines the cost of communications. In [4], it is shown that best performance are obtained with a grid with few rows: step 1a of the algorithm is fine grained and involves small communications (so a lot of communication latencies) for pivoting and for L_{10} computation.

4. Parallel out-of-core LU factorization

Now, we consider the situation where the matrix A is too large to fit in main memory.

We present the parallel out-of-core left–right looking LU factorization algorithm used by the ScaLAPACK routine `pfdgetrf` for the parallel out-of-core LU factorization [9]. Similar algorithms are also described in [10,13]. In the algorithm the matrix is divided in blocks of columns called *superblocks*. The width of each superblock is determined by the amount of physical memory available.

Similar to the previous parallel algorithm, the matrix is logically distributed on a block cyclic $p \times q$ grid of processors. But only blocks of the current superblock are in main memory, the others are on disk.

The parallel out-of-core algorithm is an extension of the parallel in-core algorithm. It factorizes the matrix from left to right, superblock by superblock. Each time a new superblock of the matrix is fetched into memory (called the *active* superblock), all previous pivoting and updates of a *history of the right-looking algorithm* are applied to the active superblock. To do this update, superblocks lying on the left of the active superblock are read again. Once the update is finished, the right-looking algorithm resumes on the updated superblock, and the factorized active superblock is written on disk. Once the last superblock has been factorized, the matrix is read again in order to apply the remaining row pivoting of the recursive phases (step 4).

The update of each active superblock is summarized in Fig. 2. When a superblock on the left is considered (called the *current* superblock), the update consists in applying row pivoting to the active superblock and:

1′. read the under-diagonal part of the current superblock;
2′. compute the U_{01} part of the active superblock by a triangular solve (function `pdtrsm`);
3′. update A_{11}, i.e. subtract the product of U_{01} part of the active superblock by L_{10} of the current superblock (function `pdgemm`).

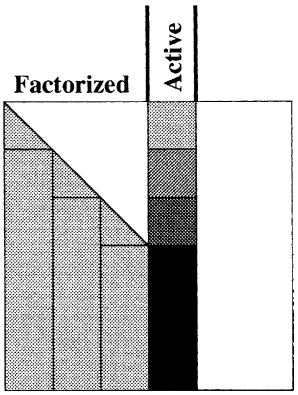


Fig. 2. Superblocks.

5. Performance prediction

In this section we present an architectural model and a prediction for the execution time of the parallel out-of-core left–right looking algorithm.

5.1. Architectural model

The architectural model consists of a distributed memory machine with an interconnection network and one disk on each node. Each node stores its blocks on its own disk. Let us characterize this kind of architecture by some constants representing the computation time, the communication time and the IO time.

Computation time. It is usually based on the time required for the computation of one floating-point operation on one processor and is represented by a constant α. In fact, this time is not constant and depends on the memory hierarchy and on the kind of computation. For instance a matrix multiplication algorithm exhibits good cache reuse whereas the product of a vector by a scalar has poor temporal locality. So we distinguish three time constants for floating point operations: α_g for matrix multiply, α_t for triangular solve, and α_s for scaling of vectors.

Communication time. As usual, the communication time is represented by the $\beta + V\tau$ model, where β is the startup time and τ is the time to transmit one unit of data and V is the volume of data to communicate. We only consider broadcast communication in our model. The constants β and τ are dependent on the topology of the virtual grid: β_p^q is the startup time for a column of p processors to broadcast data [1] on their rows, and $1/\tau_p^q$ represents the throughput. Similarly β_q^p and τ_q^p denote the time for one row of q processors to broadcast data on their columns. These functions depend on the communication network. For instance, on a cluster of workstations with a switch, the broadcast can be implemented by a tree diffusion. Then $\beta_p^q = \log_2 q \times \beta$ and $\tau_p^q = \log_2 q \times \frac{\tau}{p}$ where β is the startup communication time for one node and $1/\tau$ the throughput of the medium. With a hub (i.e. a bus), the model is: $\beta_p^q = p(q-1) \times \beta$ and $\tau_p^q = \tau$ if $q > 1$, $\tau_p^q = 0$ if $q = 1$.

IO time. The IO time is based on the throughput of a disk. Let τ^{io} be the time to read or write one word for one disk, then $\tau_p^{io} = \frac{\tau^{io}}{p}$ is the time to read or write p words in parallel for p independent disks.

5.2. Modeling

To model the algorithm, we estimate the time used by each function. For each function, we distinguish between computation time and communication time, and we distinguish between the intrinsic cost time of the parallel right-looking algorithm and cost introduced by the out-of-core extension.

Let N be the matrix order, K be the column width of superblock, and assume the block size is $k \times k$. The grid of processors is composed of p rows of q columns. We

[1] Data are equi-distributed on processors.

have the following constraints on these constants: N is a multiple of K and K is multiple of k and q. Let $L = \frac{N}{k}$ be the block width of the matrix, $S = \frac{N}{K}$ the number of superblocks, and $B = \frac{K}{k}$ be the block width of a superblock.

Fig. 3 collects costs of the different steps of the algorithm. For the sake of simplicity, we do not consider cost of pivoting cost in our analysis. This cost is mainly the cost of reduction operations for each element of the diagonal and the cost of row interchanges, plus the cost of re-reading/writing the matrix. This time can easily be integrated in the analysis if necessary.

5.2.1. `pdgetf2` cost

Step 1 of the algorithm (ScaLAPACK function `pdgetf2`) is applied on block columns (of width k) under the diagonal (Fig. 4). There are L such blocks. This computation is independent of the superblock size. For the computation cost, we distinguish the computation of blocks on the diagonal (5) and the computation on the blocks under the diagonal (6). The total computation time for `pdgetf2` function for the whole $N \times N$ matrix is (7).

For communications in `pdgetf2`, for each block on the diagonal and for each element on the diagonal, the right part is broadcast to the processor column (8).

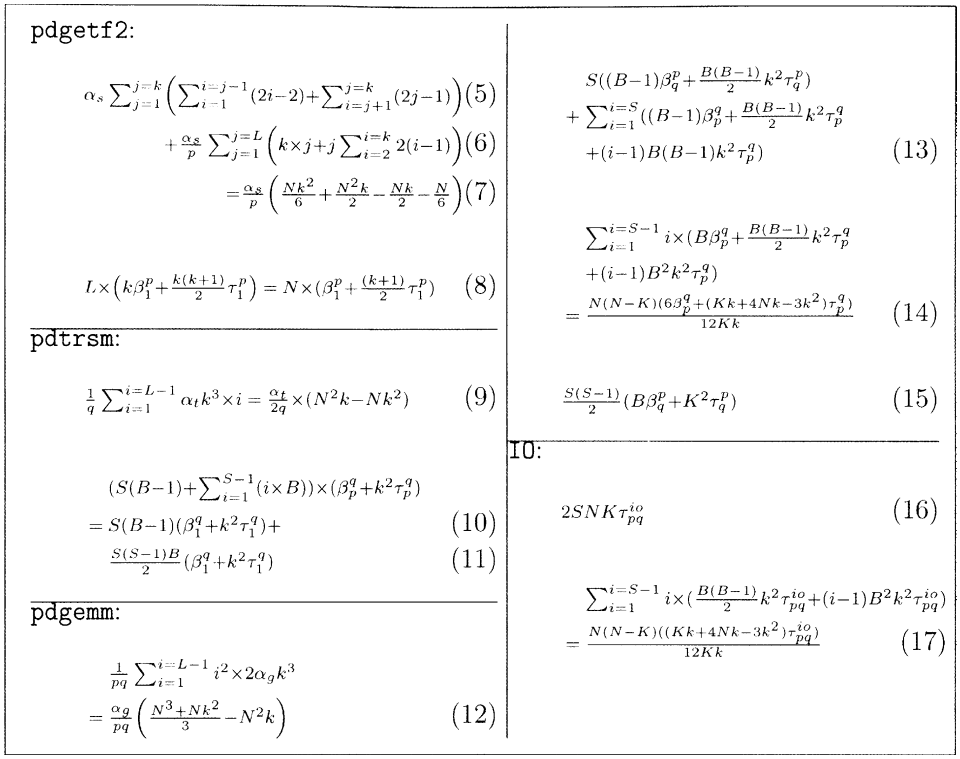


Fig. 3. Costs of the different steps of the left–right looking algorithm for out-of-core LU factorization.

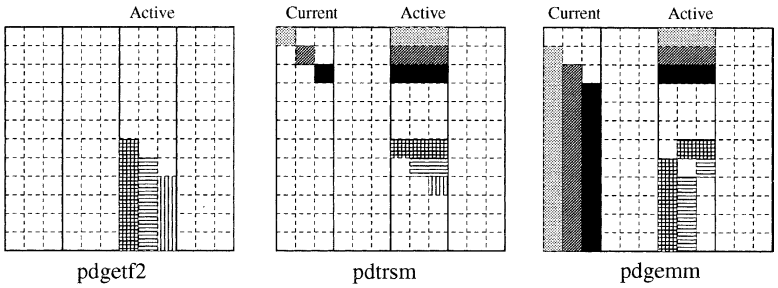


Fig. 4. Blocks involved in the three main functions of the factorization. Blocks from the active superblock and from a current one (there are 4 superblocks) are shown.

5.2.2. `pdtrsm` cost

Step 2 of the algorithm (computation of U_{01}) is applied on each block of row lying on the right of the diagonal: each of them involves a triangular solve (Fig. 4). The computational cost of a triangular solve between two blocks of size $k \times k$ is $\alpha_t k^3$. The total computation cost for every `pdtrsm` performed by the algorithm is (9).

The communication cost for `pdtrsm` is the cost of broadcasting diagonal blocks onto the processor row. One broadcast is done during the factorization of the active superblock (10), and another is needed for the future updates (11).

5.2.3. `pdgemm` cost

Step 3 of the algorithm updates the trailing sub-matrix A' (Fig. 4). The computation is mainly matrix multiplication plus a broadcast. For a trailing sub-matrix of order H, there are $(\frac{H}{k})^2$ block multiplications of size $k \times k$. The cost of such a multiplication is $2\alpha_g k^3$. The total computation cost is (12).

For the communication cost, we distinguish the cost of factorization of the active superblock and the cost of the update of the active superblock. For the factorization of the superblock, the cost corresponds to the broadcast of one row of blocks and the broadcast of one columns of blocks (13). Fig. 2 illustrates the successive updates for an active superblock. Each block of columns under the diagonal in left superblock read are broadcast (14). At the same time symmetric rows of blocks of the current superblock are broadcast (15).

5.2.4. IO cost

The IO cost corresponds to reading/writing the active superblock (16) and reading the superblock on the left (17).

5.3. Experimental validation of the analytical model

To validate our prediction model, we ran the ScaLAPACK out-of-core factorization program on a cluster. We instrumented the ScaLAPACK program for profiling the different parts of the algorithm. Then we compared the measured execution time

with the time predicted by our model. The cluster was made up of 8 PC-Celeron running Linux and interconnected by a Fast-Ethernet switch. Each node had 96 MB of physical memory. The model described in the previous subsections is instanced with the following constants (experimental measurements): $1/\alpha_g = 237$ Mflops, $1/\alpha_t = 123$ Mflops, $1/\alpha_s = 16$ Mflops, $\beta = 1.7$ ms, $1/\tau = 11$ MB/s, $1/\tau_{IO} = 1.8$ MB/s.

Table 1 shows the comparison between the execution time and the predicted time (in *italic*) of the factorization algorithm. M is the matrix order, K the superblock width, p the number of row of the processor grid, q the number of columns, S is the number of superblocks. The size of the matrix in Gigabytes is given in the first column. We measured time for Input/Output, for Computation and we distinguished the Communication time during the factorization of active superblocks and the Communication time during the update of active superblocks. Times are given in hours (h), minutes (m) and seconds (s). The last column shows the real and predicted performance in Mflops (Mflp). For computation and communication, running time was close to the predicted time. There were some differences for IO times. It is mainly due to our rough model of IO: IO performance are more difficult to predict because access file performance depends on the layout of the file on the disk (fragmentation).

6. Out-of-core overhead analysis

In comparison with the standard *in-core* algorithm, the overhead of the *out-of-core* algorithm is the extra IO costs and broadcast (of columns) cost for the update of the active superblock: for each active superblock, left superblocks must be read and broadcast once again!

This overhead cost is represented by Eqs. (11) and (14) for communications and (17) for IO. It is easy to show that if $K = N$ (i.e. $S = 1$) then this cost is equal to zero: it is the *in-core* algorithm execution time.

The first plot in Fig. 6 shows theoretical efficiency for a Fast-Ethernet based cluster. Efficiency is given according to the number of processors (from 8 to 64) and the ratio r of the problem size on the aggregate primary memory size (64 MB/node), thus for different distributions (1, 2, 4 and 8 columns). From this plot, we observe than the overhead has big impact on the performance: the efficiency is very bad for large problems. The overhead cost is $O(N^3)$, and is nonnegligible. In the following, we will show how to reduce this overhead cost. Let $O_C = (11) + (14)$ be the overhead communication cost and $O_{IO} = (17)$ be the overhead IO cost.

6.1. Reducing overhead communication cost

As shown by the model and the experimental results, the topology of the grid of processors has a large influence on the overhead communication cost:

Fact 1. *If the number of columns q is equal to* 1, *then* $O_C = 0$.

Table 1
Comparison of experimental and theoretical (in *italic*) running time

M	K	$p \times q$	IO	Computation	Comm. active	Comm. update	Execution time
12 288 1.2 GB/$S = 4$	3072	1×8	5 m 06 s/*4 m 19 s*	16 m 42 s/*10 m 37 s*	3 m 9 s/*3 m 00 s*	4 m 39 s/*6 m 28 s*	29 m 44 s/*24 m 25 s* 693 Mflp/*844 Mflp*
		2×4	4 m 20 s/*4 m 19 s*	13 m 43 s/*10 m 32 s*	59 s/*1 m 16 s*	1 m 41 s/*2 m 21 s*	20 m 58 s/*18 m 30 s* 983 Mflp/*1114 Mflp*
		4×2	4 m 44 s/*4 m 19 s*	12 m 23 s/*10 m 36 s*	34 s/*55 s*	58 s/*1 m 18 s*	18 m 56 s/*17 m 10 s* 1088 Mflp/*1200 Mflp*
		8×1	4 m 46 s/*4 m 19 s*	12 m 08 s/*10 m 53 s*	1 m 06 s/*1 m 22 s*	1 m 42 s/*2 m 16 s*	20 m 13 s/*18 m 51 s* 1019 Mflp/*1093 Mflp*
20 480 3.3 GB/$S = 10$	2048	1×8	31 m 01 s/*19 m 52 s*	1 h 02 m/*48 m 36 s*	10 m 18 s/*8 m 12 s*	52 m 9 s/*51 m 51 s*	2 h 36 m/*2 h 8 m* 610 Mflp/*742 Mflp*
		2×4	33 m 32 s/*19 m 52 s*	1 h 19 m/*48 m 22 s*	2 m 51 s/*3 m 10 s*	18 m 31 s/*17 m 59 s*	2 h 15 m/*1 h 29 m* 706 Mflp/*1067 Mflp*
		4×2	33 m 10 s/*19 m 52 s*	1 h 04 m/*48 m 34 s*	1 m 07 s/*1 m 40 s*	8 m 14 s/*6 m 55 s*	1 h 49 m/*1 h 17 m* 1208 Mflp/*1238 Mflp*
		8×1	30 m 01 s/*19 m 52 s*	1 h 03 m/*49 m 19 s*	1 m 10 s/*1 m 52 s*	7 m 00 s/*7 m 37 s*	1 h 44 m/*1 h 18 m* 917 Mflp/*1212 Mflp*
27 648 6.1 GB/$S = 27$	1024	1×8	53 m 03 s/*1 h 16 m*	2 h 45 m/*1 h 59 m*	14 m 42 s/*14 m 26 s*	4 h 3 m/*4 h 28 m*	7 h 57 m/*7 h 58 m* 492 Mflp/*490 Mflp*
		2×4	49 m 48 s/*1 h 16 m*	2 h 17 m/*1 h 58 m*	4 m 25 s/*5 m 21 s*	1 h 21 m/*1 h 30 m*	4 h 34 m/*4 h 51 m* 857 Mflp/*805 Mflp*
		4×2	50 m 38 s/*1 h 16 m*	2 h 01 m/*1 h 58 m*	1 m 18 s/*2 m 20 s*	25 m 23 s/*25 m 36 s*	3 h 20 m/*3 h 45 m* 1174 Mflp/*1040 Mflp*
		8×1	51 m 16 s/*1 h 16 m*	1 h 53 m/*2 h 00 m*	52 s/*1 m 58 s*	13 m 30 s/*15 m 00 s*	3 h 02 m/*3 h 34 m* 1290 Mflp/*1097 Mflp*

If there is only one column of processors, there is no broadcast of column during the update. If we consider a communication model where the cost of the broadcast operation is increasing with the number of processors, then the larger the number of columns is, the larger is O_C. Fig. 5 shows the influence of the topology on the performance. In the same figure, there are plots for the predicted performance of the in-core right looking algorithm. We used constants of our small PC-Celeron cluster. With a topology of one column of 16 processors (a ring) there is no extra communication cost. The difference with the *in-core* performance is due to the extra IO.

6.2. Overlapping IO and computations

A trivial way to avoid the IO overhead is to overlap this IO by computation. In the left–right looking algorithm, during updates of the active superblock, the left superblocks are read from left to right. An overlapping scheme consists in reading the next left superblock during the update of the active superblock with the current one: if the time for this update is larger than the time for reading the next superblock, then the overhead due to IO is avoided.

Now, let us consider the resource needed to achieve such a total overlapping. Let M be the amount of memory devoted to superblock in one processor. For a matrix order N the width of a superblock is then $K = \frac{pqM}{N}$. Let O_{IO}^{o} the overhead IO cost not overlapped in this new scheme.

Theorem 2. *If* $pqM \geqslant N\frac{t_{io}}{2\alpha_g}$ *then* $O_{IO}^{o} = 0$.

Proof. Consider the update part of the algorithm (Fig. 2). For the sake of simplicity, we underestimate the update computation time, and we only consider the main cost of this update: computation time of the `pdgemm` part. Let H be the height of the

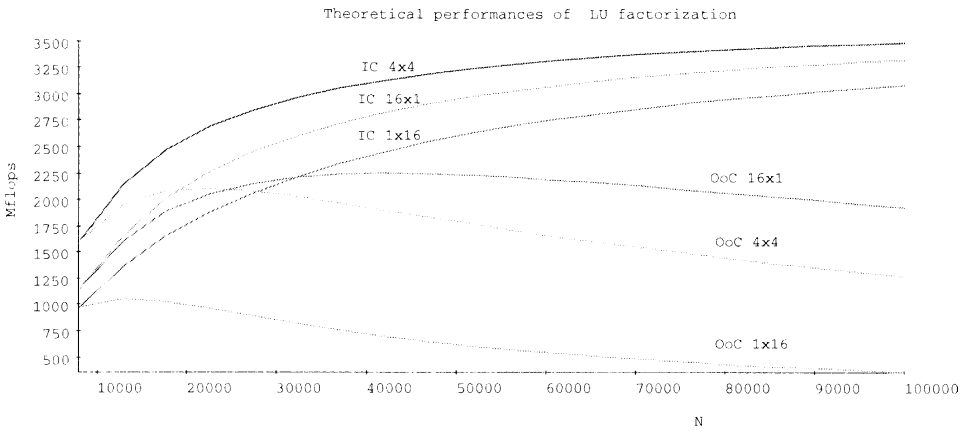


Fig. 5. Theoretical performance of the LU factorization on a cluster of 16 PC-Celeron/Linux (64 MB) interconnected by a Fast-Ethernet switch: comparison of the parallel in core (IC) right-looking algorithm and the parallel out-of-core (OoC) left–right looking algorithm, with three kinds of topologies (1×16, 4×4, and 16×1). N is the matrix order.

current left superblock in the update of the active superblock (in number of $k \times k$ blocks). The computation time for the update of the active superblock with the current one is:

$$\left((H-B)\times B+\frac{B(B-1)}{2}\right)\times\left(2\frac{\alpha_g}{pq}k^3B\right) \tag{18}$$

The time to read the next left superblock is:

$$\left((H-2B)\times B+\frac{B(B-1)}{2}\right)\times(k^2\tau^{io}_{pq}) \tag{19}$$

Now consider the situation where IO is overlapped by computation (i.e. $O^o_{IO}=0$), that is $\frac{(18)}{(19)}\geqslant 1$. We restrict the problem to the following: determinate for which superblock width B

$$\frac{\left((H-B)B+\frac{B(B-1)}{2}\right)}{\left((H-2B)B+\frac{B(B-1)}{2}\right)}\times\frac{\left(2\frac{\alpha_g}{pq}k^3B\right)}{k^2\tau^{io}_{pq}}\geqslant 1$$

The first part of this expression is always greater than 1. We determinate for which superblock width the second part of the expression is greater than 1. By definition $K=k*B$ (K is the width of a superblock in number of columns), and $\tau^{io}_{pq}=\frac{t_{io}}{pq}$. We have

$$\frac{\left(2\frac{\alpha_g}{pq}k^3B\right)}{k^2\tau^{io}_{pq}}\geqslant 1\iff 2\frac{\alpha_g}{t_{io}}K\geqslant 1\iff K\geqslant\frac{t_{io}}{2\alpha_g}$$

Since $K=pqM/N$, if $pqM\geqslant N\frac{t_{io}}{2\alpha_g}$ then $\frac{(18)}{(19)}\geqslant 1$ i.e. $O^o_{IO}=0$.

If we consider a one column distribution and this IO overlapping scheme, we have:

Fact 3. *If the number of columns q is equal to* 1, *then* $O_C=O^o_{IO}=0$.

In this situation, there is no out-of-core overhead and the execution of the out-of-core algorithm is equal to the execution time of the in-core algorithm, assuming there is enough memory to hold the matrix.

6.3. Reducing primary memory size

The previous theorem gives a lower bound of the physical memory size to achieve total overlapping of IO by computation. For instance, for a 16 nodes cluster like the previous one, the primary memory size needed to factorize a 80 GB matrix (100 000 order matrix), we need 26 Megabytes (MB) of memory per superblock (active, current and prefetched) per node, i.e. 78 MB/node. The predicted execution time to factorize the matrix is 4.5 days without overlapping, and 2.5 days otherwise. If we

substitute the Intel Celeron processors of 237 Mflops by Digital Alpha AXP processors of 757 Mflops, then the needed memory size per processor is 252 MB! The predicted computation time is about 36 h without overlapping and about 21 h otherwise (1.7 faster), which is the estimated time for the in-core algorithm with enough memory, i.e. 5 GB/node!

For larger problem, it is possible to reduce the required primary memory. Indeed, in the original algorithm, the width of the active and current superblocks are equal. An idea to reduce the need for physical memory is to specify different width for the active and the current superblock during the update: increase the width of the active superblock (i.e. computation time) and reduce the width of current superblock (i.e. read time).

7. Distribution analysis

In the previous section, we shown that only one dimensional distributions (one column of processor) allow us to avoid the communication overhead of the update part. With IO overlapping, the performance of the out-of-core algorithm is then equal to the performance of the in-core algorithm with the same distribution. Unfortunately, one column distribution is the worst distribution. For instance, reconsider the cluster of AXP processor, the estimated execution time for the in-core algorithm is 18 h instead of 21 h. In a one column distribution we get lower performances because:

- the parallelism is reduced: computation of U_{01} in the LU part (see Section 3) is done by one processor;
- due to partial pivoting the step 1a of LU decomposition (see Section 3) has a small grain with small communications (thus a lot of communication latencies).

In [4], it is shown that best performance for the in-core algorithm are obtained with few rows grid. Considering two dimensional distribution in the out-of-core algorithm reintroduces communication overhead for update: left superblocks read are communicated along processor rows. Fig. 5 suggests that for a fixed number of processor and primary memory size, the update communication is prominent on the gain of two dimensional distribution. In this case, a one-column distribution gives better performance. The second plot in Fig. 6 shows theoretical efficiency according to the number of processor (from 8 to 64) and the ratio r of the problem size on the aggregate primary memory size (64 MB/node), thus for different distributions (1, 2, 4 and 8 columns). For large problems, there is a threshold on N where one-dimensional distribution gives better performance than two-dimensional ones.

We now estimate what the threshold is. For a fixed two dimensional distribution $r \times q = p$ (r rows of q columns of processors, $q \geqslant 2$), we determine the matrix order N where one-column distribution is better than an $r \times q$ distribution. As stated before, one-column distribution is weaker in the computation of the U_{01} (`trsm`) because this computation is done by only one processor. The weakness of

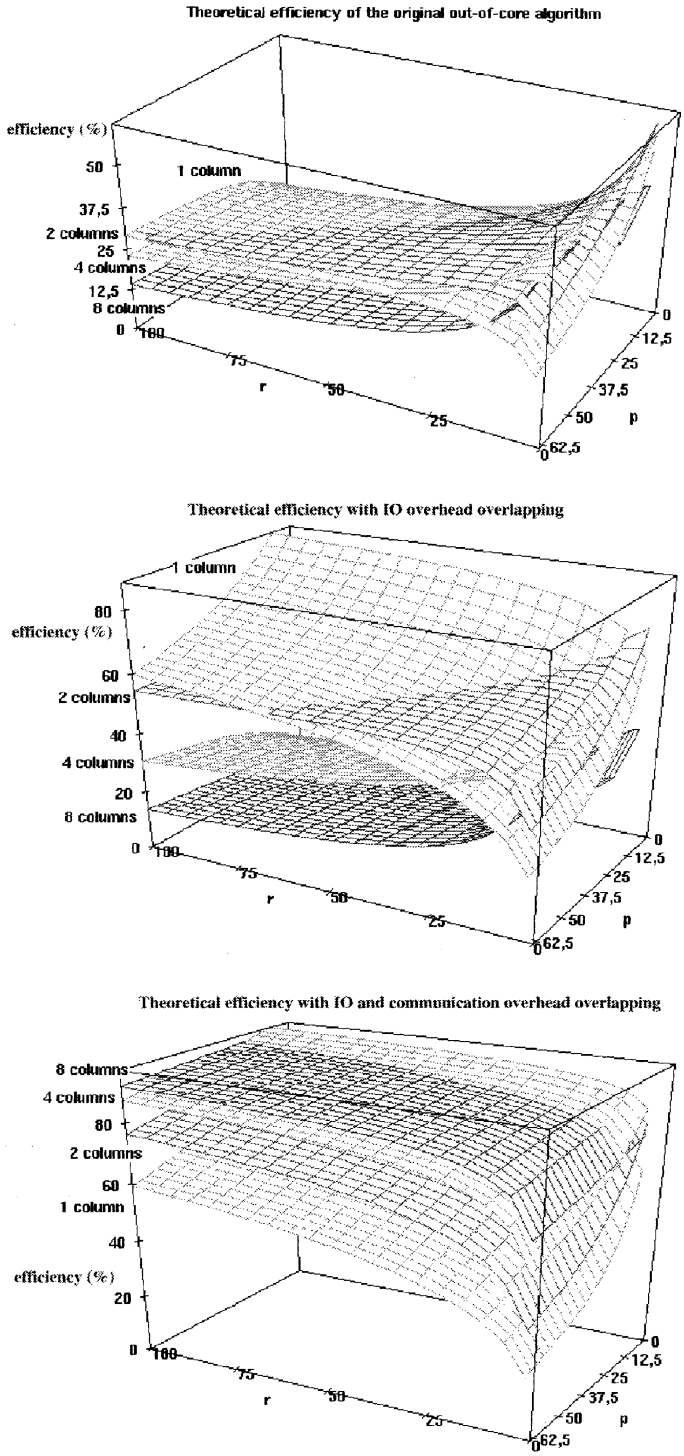


Fig. 6. Theoretical efficiencies of the out-of-core algorithm without overlapping and with overlapping of the IO overhead and with overlapping of the IO and communication overhead. Efficiencies are plotted according the number of processors p (from 8 to 64), the ratio r of the problem size on the aggregate memory size (64 MB/node), thus for four kinds of distribution (grid with 1, 2, 4 and 8 columns).

two-dimensional distribution is the cost for communication update, which is null for one-column distribution. For the sake of simplicity, we consider only this two main cost for the estimation of the threshold.

Let T_1^p (respectively T_q^r)the computation time for the `trsm` part of the factorization in one-column distribution (respectively two-dimensional distribution). From (9) we get:

$$V_t = \frac{N^2k - Nk^2}{2} \tag{20}$$

$$T_1^p = \alpha_t \times V_t \tag{21}$$

$$T_q^r = \frac{\alpha_t}{q} \times V_t \tag{22}$$

Similarly, let U_1^p (respectively U_q^r) estimations for the communication time for the `gemm` update in one-column distribution (respectively two-dimensional distribution). For the sake of simplicity we do not consider communication latency:

$$V_c = \frac{N^3}{3K} - \frac{N^2}{2} + \frac{K \times N}{6} \tag{23}$$

$$U_1^p = 0 \times V_c \tag{24}$$

$$U_q^r = \tau_q^r \times V_c \tag{25}$$

Notice again that the update communication time is $O(N^3)$ whereas `trsm` computation is $O(N^2)$. This suggest that depending on a fixed constant we can find a N where the communication update becomes greater than computation times. So, one-column distribution is better than $r \times q$ distribution when N is such that $T_1^p + U_1^p \leqslant T_q^r + U_q^r$

$$\alpha_t \times V_t \leqslant \frac{\alpha_t}{q} \times V_t + \tau_q^r \times V_c \tag{26}$$

$$\frac{V_t}{V_c} \leqslant \frac{q}{q-1} \times \frac{\tau_q^r}{\alpha_t} \tag{27}$$

Roughly speaking, a one-column distribution is better when the ratio of the volume of computation $(O(N^2))$ on the volume of communication $(O(N^3))$ is less than the number of computation which can be done when one word is communicated.

If we consider an architecture of p processors where τ_q^r increases as q increases (less communication are done in parallel), then increasing the number of columns (and reducing the number of rows) decrease the threshold for which it is preferable to switch to one-column distribution. In fact, in the out-of-core algorithm, a good distribution is a distribution with few columns: it is the opposite of a good distribution for in-core right-looking algorithm.

8. Theoretical benefits of overlapping communication

To achieve better performance, an overlapping scheme for communication update must be introduced. Such an overlapping scheme will allows us to reconsider two dimensional grid distributions.

A possible implementation scheme is to read ahead the second next superblock and communicate the next left superblock during the update of the active superblock with the current one.

To see what gains can be obtained by this new overlapping scheme, let consider the efficiency for different distributions. Fig. 6 shows the theoretical efficiency according to the number of processors (from 8 to 64) and the ratio r of the problem size on the aggregate primary memory size, thus for different distributions (1, 2, 4 and 8 columns). The behaviours of the out-of-core algorithm is similar to the behaviour of the in-core one: a good distribution is a distribution with few rows.

Similarly to the IO overlapping scheme, we consider the resource needed to achieve such a total overlapping. Let M be the amount of memory devoted to one superblock on one processor. For a matrix of order N the width of a superblock is then $K = \frac{pqM}{N}$. Let $O_{\mathrm{C}}^{\mathrm{o}}$ be the communication overhead cost not overlapped in this new scheme.

Theorem 4. *Let r be such that $rk^2\tau_q^p = \beta_q^p$. If $pqM \geqslant N\frac{(r+1)\tau_q^p}{2\alpha_{\mathrm{g}}}$ then $O_{\mathrm{C}}^{\mathrm{o}} = 0$.*

Proof. The proof is similar to the previous one. Consider the update part of the algorithm (Fig. 2). For the sake of simplicity, we underestimate the update computation time, and we consider only the main cost of this update: computation time of the `pdgemm` part. Let H be the height of the current left superblock in the update of the active superblock (in number of $k \times k$ blocks). The computation time for the update of the active superblock with the current one is:

$$\left((H-B)\times B + \frac{B(B-1)}{2}\right) \times \left(2\frac{\alpha_{\mathrm{g}}}{pq}k^3B\right) \tag{28}$$

The communication time of the next left superblock is:

$$\left((H-2B)\times B + \frac{B(B-1)}{2}\right) \times (k^2\tau_q^p + \beta_q^p) \tag{29}$$

Since r is such that $rk^2\tau_q^p = \beta_q^p$:

$$\left((H-2B)\times B + \frac{B(B-1)}{2}\right) \times (r+1)k^2\tau_q^p \tag{30}$$

Now consider the situation where the communication is overlapped by computation (i.e. $O_{\mathrm{IO}}^{\mathrm{o}} = 0$), that is $\frac{(28)}{(30)} \geqslant 1$. We restrict the problem to the following: determinate for which superblock width B satisfies:

$$\frac{\left((H-B)B+\frac{B(B-1)}{2}\right)}{\left((H-2B)B+\frac{B(B-1)}{2}\right)}\times\frac{\left(2\frac{\alpha_g}{pq}k^3B\right)}{(r+1)k^2\tau_q^p}\geqslant 1$$

The first part of this expression is always greater than 1. We determine for which superblock width the second part of the expression is greater than 1. By definition $K = k * B$ (K is the width of superblock in number of columns). We have

$$\frac{\left(2\frac{\alpha_g}{pq}k^3B\right)}{(r+1)k^2\tau_q^p}\geqslant 1 \iff \frac{2\alpha_g}{(r+1)\tau_q^p}K\geqslant 1 \iff K\geqslant \frac{(r+1)\tau_q^p}{2\alpha_g}$$

Since $K = pqM/N$, if $pqM \geqslant N\frac{(r+1)\tau_q^p}{2\alpha_g}$ then $\frac{(28)}{(30)}\geqslant 1$, i.e. $O_{IO}^o = 0$.

Similarly to the IO overhead overlapping scheme, the memory requirement can be reduced by varying the different superblock size, at the price of a more sophisticated implementation.

With the combination of the IO overhead overlapping scheme, this overlapping scheme avoids the overhead of the out-of-core algorithm. The running time is then identical to the in-core algorithm for all distribution.

9. Conclusion

In this paper we presented a performance prediction model of the parallel out-of-core left–right looking LU factorization algorithm which can be found in ScaLAPACK. This algorithm is mainly an extension of the parallel right-looking LU factorization algorithm.

This algorithm was first introduced by [5,10,13], but the performance study was based on some experiments and no limitation was found for it. Thanks to our modeling, we revisited the performance of the algorithm and isolated the overhead introduced by the out-of-core version and outline its limitation: the overhead is $O(N^3)$ for a matrix order N, the order of the computation! The overhead is divided into an IO part and a communication part. We showed that a straightforward scheme to overlap the IO by the computations allows us to reduce the IO overhead of the algorithm. We determined the memory size which is necessary to avoid the IO overhead. The memory size needed is proportional to the square root of the matrix size. We analyzed the impact of communication overhead on two-dimensional distribution of computation: a good distribution strategy is a distribution on few columns of processors and for large problem the best distribution is one-column of processors. It is the opposite strategy for the in-core algorithm. We showed that with a similar memory requirement as for the IO overlapping, it is possible to introduce overlapping for the communication overhead and get freedom on the distribution of computation to achieve better performance.

This paper demonstrates that there is no memory limitation to the factorization of huge matrices. With these two overlapping schemes, the performance of the out-of-core algorithm is similar to the performance of the in-core one (assuming there is

enough memory to hold the matrix). The physical memory requirement if $O(N)$ where N is the matrix order. This memory requirement is determined by the different constants of the architecture and can be reduced by varying the different superblock sizes. The only limitation is in fact the running time (it is a compute bound problem). As running time is similar to the in-core algorithm, the out-of-core algorithm scale as well. We extended this algorithm to the matrix inversion problem in [3].

Thanks to our modeling, we can give hint for dimensioning cluster for large dense matrix factorization. Using overlapping scheme, a good architecture to perform large matrix factorization must be based on fast processors. The memory size is adjusted by the IO and communication performance to achieve total overlapping. As efficiency is the efficiency of the in-core (right-looking) algorithm, optimizations designed for the right looking algorithm may be integrated to improve it. For example another overlapping scheme for intrinsic communication of the in-core algorithm proposed in [8].

At this time, we implemented overlapping of IO overhead but only for Linux Cluster platform. Implementing overlapping of communication overhead needs further development. Unfortunately, there is no standard for asynchronous IO and communication widely implemented today. MPI and MPI-IO are good candidates for asynchronous IO and communication, but BLACS and ScaLAPACK do not integrate these features.

An issue which has not be considered in this paper is fault tolerance. Checkpointing is implicit in this out-of-core algorithm: If the computation failed during the computation (update or factorization) of an active superblock, since data are on disk, we can restart the computation by re-reading the active superblock. To prevent failure during the write phase of the active superblock, a copy can be store in on other file and read again to restart the computation.

Because we consider large matrices, numerical stability of the algorithm is an important concern. Even using partial pivoting, it is known that this kind of blocked algorithm is not stable in general [6]. It was proved this method are conditionally stable for symmetric positive definite or diagonally dominant matrices, and it is unconditionally stable for matrices which are block diagonally dominant by columns. When using such a factorization algorithm to solve linear system, it must be careful about the residual, and try iterative refinement when the residual is too big [7]. Finally for symmetric indefinite matrices, another algorithm may be considered such that diagonal pivoting methods [12].

Acknowledgements

This work is supported by a grant of the "Pôle de Modélisation de la Région Picardie" and by an INRIA grant OURAGAN.

We would like to thank the ID laboratory for granting access to its Cluster Computing Center (http://www-id.imag.fr/grappes.html). This work used the ID/HP i-cluster.

Special thanks to Olivier Cozette (LaRIA) for his help in the development.

References

[1] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK Users' Guide, SIAM, Philadelphia, 1997.
[2] E. Caron, O. Cozette, D. Lazure, G. Utard, Virtual memory management in data parallel applications, in: P. Sloot, M. Bubak, A. Hoekstra, B. Hertzberger (Eds.), Proceedings of the 7th International Conference on High Performance Computing and Networking (HPCN Europe 99), Lecture Notes in Computer Science, vol. 1593, Springer-Verlag, Amsterdam, 1999, pp. 1107–1116.
[3] E. Caron, G. Utard, Parallel out-of-core matrix inversion. In IPDPS'02. The 16th International Parallel and Distributed Processing Symposium, Fort Lauderdale, 2002.
[4] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, LAPACK Working Note: ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers—Design Issues and Performances. Technical Report UT-CS-95, Department of Computer Science, University of Tennessee, 1995.
[5] Ed. F d'Azevedo, J. Dongarra, The design and implementation of the parallel out-of-core scalapack LU, QR and cholesky factorization routines, Concurrency Practice and Experience 12 (15) (2000) 1491–1493.
[6] J.W. Demmel, N.J. Higham, Stability of block algorithm, ACM Transaction on Mathematical Software 18 (3) (1992).
[7] J.W. Demmel, N.J. Higham, R.S. Schreiber, Stability of block LU factorization, Numerical Linear Algebra with Applications 2 (2) (1995) 173–190.
[8] F. Desprez, S. Domas, B. Tourancheau, Optimization of the ScaLAPACK LU factorization routine using communication/computation overlap, in: Europar'96 Parallel Processing, LNCS, vol. 1124, Springer, 1996, pp. 3–10.
[9] J. Dongarra, S. Hammarling, D.W. Walker, Key concepts for parallel out-of-core LU factorization, Parallel Computing 23 (1997).
[10] W.C. Reiley, R.A. van de Geijn, POOCLAPACK : Parallel Out-of-Core Linear Algebra Package, Technical report, Department of Computer Sciences, The University of Texas, Austin, Draft 27 October 1999.
[11] J.M. Del Rosario, A. Choudhary, High performance I/O for massively parallel computers: problems and prospects, IEEE Computer 27 (3) (1994) 59–68.
[12] P.E. Strazdins, The design of scalable out-of-core dense symmetric indefinite factorization algorithms, in: WoPLA'03/ICCS'03 (Workshop on Parallel Lineat Algebra/International Conference on Computational Sciences), Melbourne, Australia, 2003.
[13] S. Toledo, F.G. Gustavson, The design and implementation of SOLAR a portable library for scalable out-of-core linear algebra computations, in: Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems, ACM Press, Philadelphia, 1996, pp. 28–40.

Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Parallel Computing 29 (2003) 1191–1218

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Impact of reordering on the memory of a multifrontal solver

Abdou Guermouche *, Jean-Yves L'Excellent, Gil Utard

INRIA ReMaP Project, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, UMR CNRS-ENS Lyon-INRIA 5668, 46 allée d'Italie, 69364 Lyon Cedex 07, France

Received 15 January 2003; received in revised form 23 May 2003; accepted 30 May 2003

Abstract

This paper is concerned with the memory usage of sparse direct solvers, which depends on the ordering of the unknowns and the scheduling of the computational tasks. We study the influence of state-of-the-art sparse matrix reordering techniques on the memory usage of a multifrontal solver. Concerning the scheduling, the memory usage depends on the tree traversal and how the tasks are assigned to the processors. We analyze the memory scalability when a dynamic scheduling strategy mainly based on the balance of the workload is used. Finally we give hints to improve the parallel memory behaviour.

Keywords: Sparse direct solvers; Parallel multifrontal method; Reordering; Assembly tree; Memory

1. Introduction

Sparse direct methods and in particular multifrontal methods are robust and efficient techniques to solve large sparse systems of linear equations. However, they are known for their relatively large memory requirements compared to iterative methods so that an in-core execution is not always possible: sometimes, large problems fail to be solved because of a lack of memory on the processors.

In multifrontal solvers two types of memory areas can be distinguished in the process of solving sparse linear systems: a static memory needed to store the final factors

* Corresponding author.
E-mail addresses: abdou.guermouche@ens-lyon.fr (A. Guermouche), jean-yves.l.excellent@ens-lyon.fr (J.-Y. L'Excellent), gil.utard@ens-lyon.fr (G. Utard).

0167-8191/$ - see front matter
doi:10.1016/S0167-8191(03)00099-1

of the sparse matrix; and an additional dynamic memory, also called active memory, needed to store temporary values used by the computation. In the case of multifrontal methods, the latter is handled by a stack mechanism. The size of the active memory can be large, and is sometimes larger than the factors.

The contribution of this paper is an extensive study of the memory usage of parallel multifrontal solvers. We show that the memory depends on both the reordering technique applied and the scheduling of the computational tasks and give hints on how to optimize the memory usage for sequential and parallel cases.

Reordering (i.e., renumbering the unknowns of a sparse linear system) is a well-known technique to reduce the fill in the final factors, and this has a significant impact on the static memory size. In this paper we show that reordering techniques also have a big impact on the active memory size. In the multifrontal method, the active memory size depends on the shape of assembly trees resulting from the reordered matrix. Thus, we present an extensive study of the assembly tree shapes resulting from various combinations of sparse matrices and reorderings.

The active memory size also depends on the way the assembly tree is traversed during the factorization process and how the computation is distributed on the processors. We experimentally study the memory usage of the parallel multifrontal MUMPS.

This paper is organized as follows. In Section 2, we recall some general mechanisms of the multifrontal method. Then we give in Section 3 a description of the reordering techniques and test problems used for our study. In Sections 4 and 5, we study the impact of these reordering techniques on both the shape of the assembly tree and on the evolution of the dynamic memory, respectively. Section 5.2 presents a variant of the algorithm by Liu [17] to modify the traversal of the multifrontal assembly tree in our context. We show how a reduction of the active memory size can be obtained depending on the reordering technique used. After that, we analyze in Section 6 the influence of reordering on the memory balance and consumption for parallel executions and study the main factors that limit the memory scalability. Finally, we draw conclusions.

2. The multifrontal method

Like other direct methods, the multifrontal method [9,10] is based on the elimination tree [18], which is a transitive reduction of the matrix graph and is the smallest data structure representing dependencies between operations. In practice, we use a structure called assembly tree, obtained by merging nodes of the elimination tree whose corresponding columns belong to the same supernode [5]. We recall that a supernode is a contiguous range of columns (in the factor matrix) having the same lower diagonal nonzero structure.

Fig. 1 gives an example of a matrix and its associated assembly tree. From the initial matrix, an assembly tree with three nodes (each corresponding to one supernode) is derived. The two first independent leaf nodes contribute to the computation of the third.

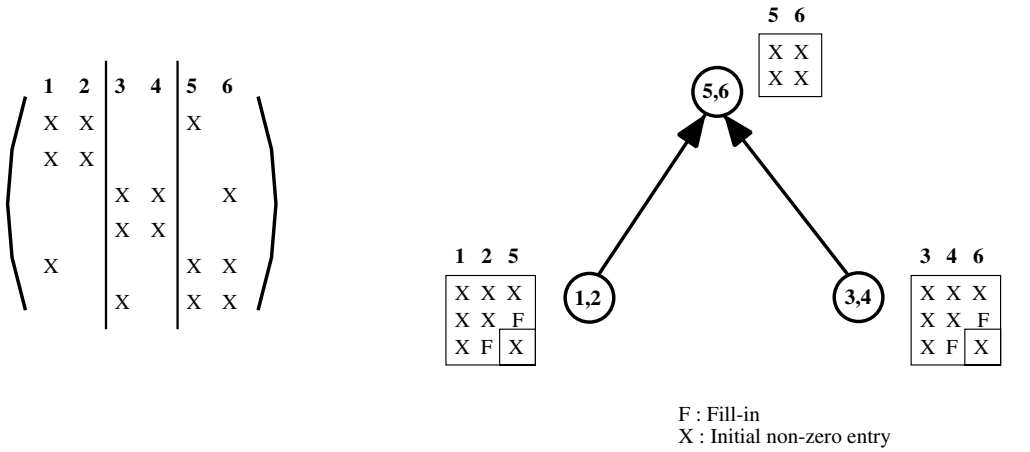
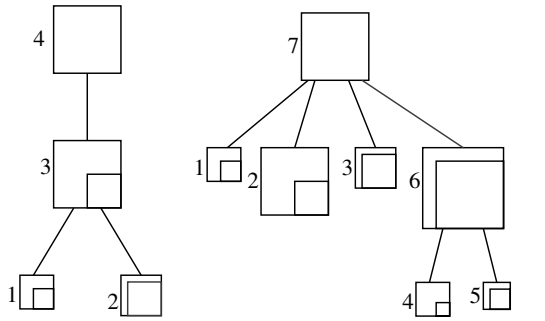


Fig. 1. A matrix with three supernodes ({1,2}, {3,4}, {5,6}) and the associated assembly tree.

In the multifrontal approach, the factorization of a matrix is done by performing a succession of partial factorizations of small dense matrices called *frontal matrices*, which are associated to the nodes of the tree. The order of a frontal matrix is given by the number of nonzeros below the diagonal in the first column of the supernode associated with the tree node. Each frontal matrix is divided into two parts: the *factor* block, also called *fully summed* block, which corresponds to the variables factorized when the elimination algorithm processes the frontal matrix; and the *contribution block* which corresponds to the variables updated when processing the frontal matrix. Once the partial factorization is complete, the contribution block is passed to the parent node. When contributions from all children are available on the parent, they can be assembled (i.e. summed with the values contained in the frontal matrix of the parent). The elimination algorithm is a postorder traversal (we do not process parent nodes before their children) of the assembly tree [22]. It uses three areas of storage in a contiguous memory space, one for the factors, one to stack the contribution blocks, and another one for the current frontal matrix [2]. During the tree traversal, the memory space required by the factors always grows while the stack memory (containing the contribution blocks) varies depending on the operations made: when the partial factorization of a frontal matrix is processed, a contribution block is stacked which increases the size of the stack; on the other hand, when the frontal matrix is formed and assembled, the contribution blocks of the children nodes are popped out of the stack and its size decreases. The stack memory is thus very dependent on the assembly tree topology.

To illustrate our observations, we give in Fig. 2 two examples of assembly trees. The corresponding memory evolution for the factors, the stack and the current frontal matrix is given in Fig. 3. First storage for the current frontal matrix is reserved (see "Allocation of 3" in Fig. 3(a)); then the frontal matrix is assembled using values from the original matrix and contribution blocks from the children nodes, and those can be freed ("Assembly step for 3" in 3(a)); the frontal matrix is factorized ("Factorization step for 3" in 3(a)). Factors are stored in the factor area on the left in our



(a) tree A – deep (b) tree B – wide

Fig. 2. Examples of assembly trees.

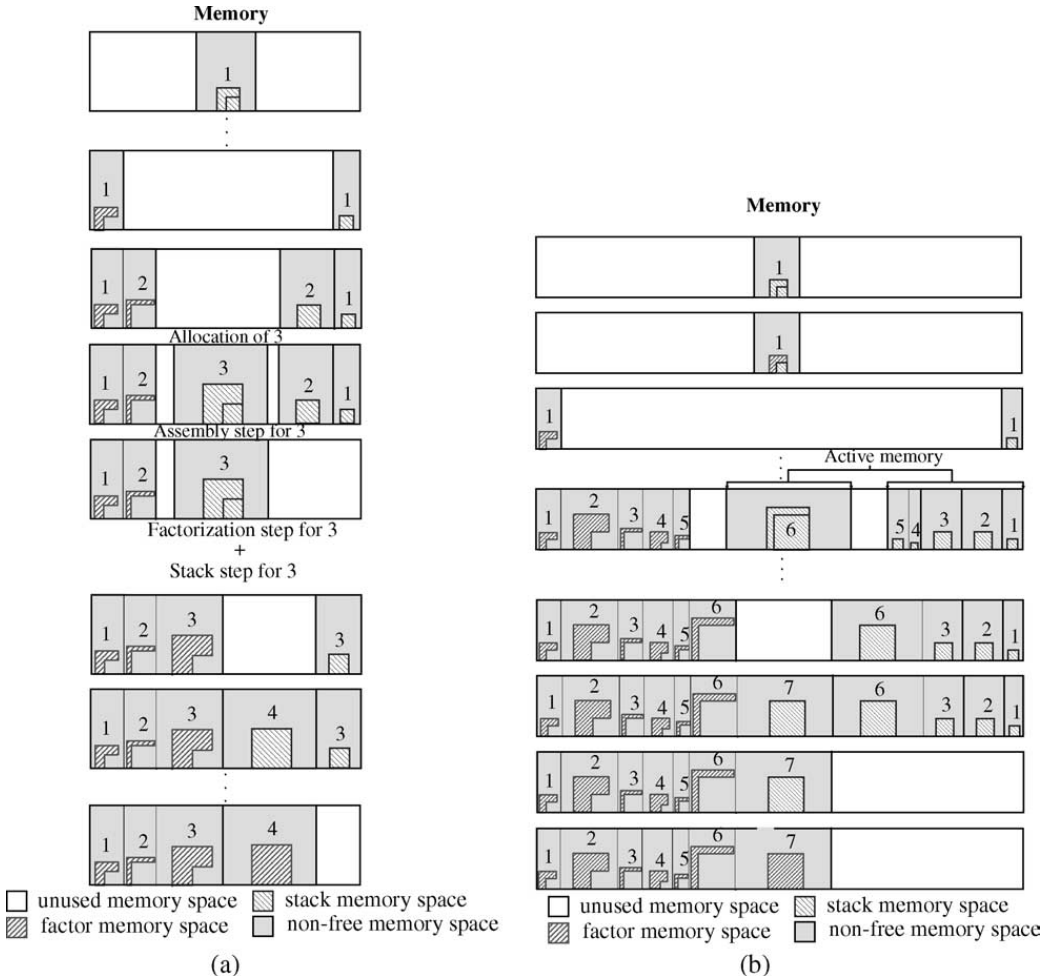


Fig. 3. Memory evolution for the trees given in Fig. 2. (a) Memory usage for tree A. (b) Memory usage for tree B.

figure and the contribution block is stacked (''Stack step for 3''). The process continues until the complete factorization of the root node(s). We can observe the different memory behaviours between the wide tree (Fig. 3(b)) and the deep tree (3(a)): the peak of active memory (see Fig. 3(b)) is larger for the wide tree.

Note that in our description all the contribution blocks for children nodes are assembled at once (in the sequential case). Another approach could be to preallocate the frontal matrix of the parent node and perform an assembly step each time a contribution block is computed. This is generally not done in multifrontal solvers because this strategy implies the use of more complex memory management algorithms and the structure of the frontal matrix of the parent is unpredictable when there is pivoting. Also, except for very wide trees, this is not necessarily a more efficient memory scheme because it implies storing several frontal matrices (each containing all the future contribution blocks of the subtree).

In the rest of the paper we only distinguish between two areas of storage: the factors, and the stack, where the stack includes the storage for the current frontal matrix.

3. Reordering techniques

Reordering the variables of a sparse linear system, i.e. permuting columns and rows (while keeping numerical stability under control), aims at reducing the amount of fill-in. Here we only consider symmetric reordering techniques which can also be applied to an unsymmetric matrix $\mathbf{A}$ by considering the structure of $\mathbf{A} + \mathbf{A}^{\mathrm{T}}$ (after some column permutation for very unsymmetric matrices [8]).

Two popular schemes for symmetric reordering are bottom-up heuristics such as the minimum degree (AMD [1], MMD [16]) or minimum fill (MMF [19,23]) and global or top-down heuristics based on partitioning the graph of the matrix, such as nested dissection [12]. A class of algorithms has also been developed that hybridize top-down nested dissection with bottom-up minimum degree.

Note that although these heuristics mainly focus on the reduction of fill-in, (and thus size of the factors and number of operations), they also have a significant impact on the parallelism (see, e.g. [3]). Here, we are interested in the influence of such techniques on the memory usage and consider the following bottom-up, top-down and hybrid heuristics:

- AMD: the Approximate Minimum Degree [1];
- AMF: the Approximate Minimum Fill, as implemented in `MUMPS`; [1]
- PORD: a tight coupling of bottom-up and top-down sparse reordering methods [24];

[1] Available from http://www.enseeiht.fr/apo/MUMPS.

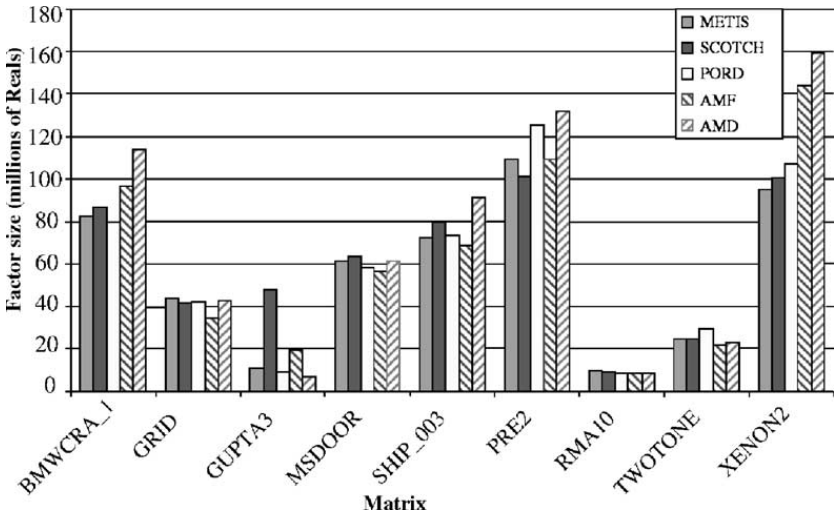


Fig. 4. Size of the factors (millions of reals).

- METIS: we use here the routine `METIS_NODEND` from the METIS package [15] which is an hybrid approach based on multilevel nested dissection and multiple minimum degree;
- SCOTCH: we use a modified version of `SCOTCH` [20] provided by the author that couples nested dissection and (halo) Approximate Minimum Fill (HAMF), in a way very similar to [21]. The switch to HAMF is done when the size of the subgraph obtained is 120.

In the following, we simply use the terms AMD, AMF, METIS, SCOTCH and PORD to refer to these heuristics. We must note that for AMD, AMF, SCOTCH and PORD, the assembly tree is returned directly from the reordering algorithm, while for METIS, only the permutation is returned and `MUMPS` is used to build an assembly tree based on this permutation.

Finally, note that we had initially considered a pure nested dissection algorithm [12], but this one was competitive only in a few cases, and only for extremely regular problems, so that we decided to discard it.

Fig. 5 gives the ratio between the peak of the stack and the final size of the factors in the sequential case. (Note that the final size of factors for every test problem and every reordering technique is given in Fig. 4.) The matrices are from Table 1 and are extracted from either the Rutherford–Boeing collection [7], the collection from University of Florida [2] or the PARASOL collection. [3] We can see that the peak of the stack can be significant compared to the size of factors (the ratio is near to 1).

[2] Available from http://www.cise.ufl.edu/~davis/sparse/.
[3] Available from http://parallab.uib.no/parasol.

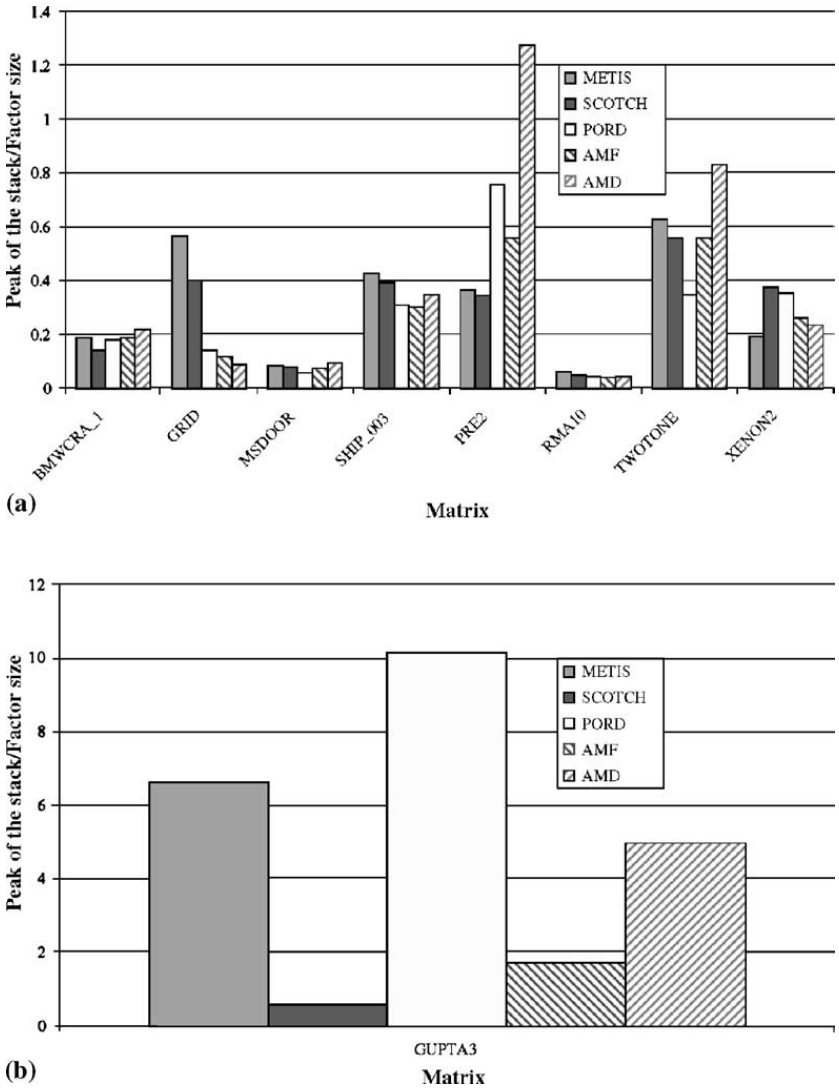


Fig. 5. Ratio between the size of the stack and the size of the factors.

Furthermore, for certain problems like the matrix GUPTA3, the peak of the stack is larger than the size of the factors. This illustrates the fact that the stack memory must be well managed for both sequential and parallel executions.

4. Impact of reordering techniques on the assembly tree

In this section, we study the impact of the reordering technique used on the shape of the corresponding assembly tree. We consider the test problems from Table 1 and the reordering techniques METIS, SCOTCH, PORD, AMF and AMD introduced in Section 3.

Table 1
Description of the test problems

Matrix	Order	NZ	Type	Description
BMWCRA_1	148770	5396386	SYM	Automotive crankshaft model with nearly 150000 TETRA elements (MSC-CRANKSHAFT-150K)
GRID	109744	1174048	SYM	11-point discretization of the Laplacian on a 3D grid (152*38*19)
GUPTA3	16783	4670105	SYM	Linear programming matrix (A*A′), Anshul Gupta
MSDOOR	415863	10328399	SYM	Medium size door
SHIP_003	121728	4103881	SYM	Ship structure from production run
PRE2	659033	5959282	UNS	AT&T,harmonic balance method, large example
RMA10	46835	2374001	UNS	3D CFD model, Charleston harbor. Steve Bova, US Army Eng., WES
TWOTONE	120750	1224224	UNS	AT&T,harmonic balance method, two-tone
XENON2	157464	3866688	UNS	Complex zeolite, sodalite crystals. D Ronis

"SYM" stands for symmetric, "UNS" for unsymmetric.

We use the software package MUMPS (MUltifrontal Massively Parallel Solver) [3,4], which implements parallel multifrontal solvers with threshold partial pivoting for both **LU** and $\mathbf{LDL}^{T}$ factorizations. For our purpose, we first experiment with the sequential version, and the tree is processed using a depth-first search traversal. We have instrumented the code to obtain statistics on both the assembly tree and the memory and be able to understand in better detail the evolution of the memory usage with time. Tests of MUMPS have been made on the IBM SP system of the CINES [4] which is composed of 29 nodes of 16 processors. Each node is equipped with 16 GB of memory shared among its 16 Power3+ (375 MHz) processors. The general shape of the assembly tree (width and depth) was estimated by the number of nodes (Table 2), and the percentage of leaves in the tree (Table 3). Regularity of the shape was estimated by the standard deviation of the depth of the leaves (Table 5), the maximum depth of a leaf (Table 4) and the average number of children (Table 6). Because the stack size is influenced by the size of frontal matrices, we report for each tree the maximum and average sizes of a frontal matrix (Tables 7 and 8). In these tables, the largest value of a row is in bold, while the smallest value in italics.

As previously noticed AMD, AMF, PORD and SCOTCH directly return an assembly tree and METIS only provides a permutation that is used by MUMPS to build

[4] Centre Informatique National de l'Enseignement Supérieur.

Table 2
Number of nodes in the tree

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	8767	*2833*	9268	**9902**	8320
GRID	24 953	*10 218*	24 081	**29 680**	28 224
GUPTA3	413	*26*	1790	1300	**1898**
MSDOOR	31 611	*28 511*	32 843	**33 401**	31 335
SHIP_003	7474	*4294*	7798	**8253**	7634
PRE2	204 359	*169 920*	**215 403**	205 297	195 812
RMA10	4608	*3465*	5109	**5325**	4524
TWOTONE	35 718	*27 904*	41 309	**41 794**	39 460
XENON2	18 990	*13 130*	**20 455**	20 386	19 043

Table 3
Percentage of leaves in the tree

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	39.6	**47.7**	38.1	*33.7*	38.0
GRID	58.8	**67.9**	*49.0*	49.1	51.2
GUPTA3	**95.9**	26.9	23.4	33.8	*21.3*
MSDOOR	55.0	**66.8**	53.9	*51.3*	54.2
SHIP_003	43.8	**59.4**	43.5	*38.7*	43.0
PRE2	69.1	68.9	**74.0**	65.5	*61.0*
RMA10	**43.2**	42.9	42.7	41.8	*39.6*
TWOTONE	68.2	68.8	**72.8**	71.8	*67.5*
XENON2	48.3	**70.4**	49.2	45.3	*42.2*

Table 4
Maximum depth for a node

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	21	*14*	54	**100**	34
GRID	26	*14*	49	**188**	53
GUPTA3	*7*	8	9	**41**	13
MSDOOR	26	*17*	53	**80**	35
SHIP_003	29	*14*	75	**122**	32
PRE2	56	*18*	**115**	99	42
RMA10	*26*	40	43	**208**	165
TWOTONE	55	*15*	77	**193**	47
XENON2	24	*16*	54	**65**	26

an assembly tree. Therefore in the following, remarks concerning to METIS actually apply to the tree obtained by METIS followed by `MUMPS` symbolic factorization.

General shape: We observe in Table 2 that for most test problems, SCOTCH generates the tree with the smallest number of nodes. Then AMD and METIS provide approximately the same number of nodes, and finally, AMF and PORD give trees with a much larger number of nodes compared to SCOTCH. In addition, we observe

Table 5
Variance of the depth of leaves

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	4.37	*1.14*	95.79	**375.00**	20.90
GRID	9.30	*0.95*	98.90	**2852.83**	149.81
GUPTA3	3.15	3.84	*1.02*	**114.48**	5.73
MSDOOR	3.99	*1.49*	**70.44**	53.63	10.90
SHIP_003	19.08	*3.24*	321.38	**508.80**	25.37
PRE2	110.95	*9.16*	**815.96**	265.05	21.52
RMA10	11.13	*7.41*	67.73	**2084.39**	1331.67
TWOTONE	54.31	*5.17*	294.80	**458.47**	80.65
XENON2	7.06	*1.67*	101.60	**144.73**	10.29

Table 6
Average number of children

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	1.66	**1.91**	1.62	*1.51*	1.61
GRID	2.43	**3.11**	*1.96*	1.96	2.05
GUPTA3	**24.24**	1.32	1.31	1.51	*1.27*
MSDOOR	2.22	**3.01**	2.17	*2.05*	2.18
SHIP_003	1.78	**2.46**	1.77	*1.63*	1.75
PRE2	3.23	3.21	**3.85**	2.90	*2.56*
RMA10	**1.76**	1.75	1.75	1.72	*1.65*
TWOTONE	3.14	3.21	**3.68**	3.54	*3.08*
XENON2	1.93	**3.37**	1.96	1.82	*1.73*

Table 7
Maximal frontal matrix order

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	2343	*2040*	2076	2496	**2835**
GRID	**2754**	2343	1721	1536	*1328*
GUPTA3	*827*	**5058**	1643	3028	1030
MSDOOR	1372	**1624**	*1358*	1491	1610
SHIP_003	3456	*3156*	3426	3408	**4038**
PRE2	*4290*	4334	5794	6476	**7502**
RMA10	**466**	422	*378*	439	399
TWOTONE	2382	*2316*	2561	2588	**2684**
XENON2	*2554*	2623	2743	3663	**4501**

from Table 3 that usually, SCOTCH and METIS generate trees with a large percentage of leaves when compared to the trees generated by AMF, AMD or PORD. Effectively, the trees generated by METIS and SCOTCH are rather wide (because of the global partitioning performed at the top), while the trees generated by AMD, AMF and PORD tend to be deeper (see also Table 4).

Regularity: According to Tables 4 and 5, we can see that PORD and AMF generate more unbalanced trees (where depth of leaves varies a lot depending on the

Table 8
Average front order

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	177	**287**	*172*	187	183
GRID	41	**68**	42	*36*	38
GUPTA3	624	**1248**	365	338	*336*
MSDOOR	**74**	*67*	73	72	71
SHIP_003	**170**	153	165	163	*151*
PRE2	**15**	*13*	14	14	14
RMA10	**60**	59	54	*53*	56
TWOTONE	**23**	*18*	20	21	19
XENON2	70	*69*	70	71	**76**

branches) while SCOTCH and METIS generate much better balanced trees. Finally, we can see in Table 6 that PORD, AMD and AMF have trees where the average number of children for a node is smaller than for the METIS and SCOTCH cases; this also illustrates that the tree is not very wide (but deep). These remarks make sense when we know that AMF, AMD and PORD are based on local methods only aiming at minimizing either the degree or the fill.

Front size analysis: According to Tables 7 and 8, we can say that in most cases, SCOTCH and METIS generate trees with frontal matrices that are bigger than those generated by the other reorderings. This observation will help us to explain some results in the next sections. Note that AMD generates trees with big variations of the front size.

Summary: To summarize this section, we have seen that reordering techniques have a strong impact on the shape of the assembly tree. Fig. 6 summarizes the general observations made for the different reorderings on the assembly tree. Concerning the shape of the tree, we have observed that hybrid heuristics like METIS and SCOTCH generate wide well-balanced trees (with a smaller number of nodes for SCOTCH). On the other hand, PORD, AMD and AMF give deep trees; it is interesting to notice that AMD provides better balanced trees than AMF and PORD. In addition, METIS and SCOTCH give trees with bigger frontal matrices than the ones generated by other reorderings.

5. Sequential memory usage of the multifrontal method

Given an assembly tree, an important factor impacting the memory usage is the order in which the nodes of the tree are visited. The only constraint in the traversal of the tree is that parent nodes are processed after their children and in general, for sparse multifrontal solvers the traversal is the depth-first search. In other terms, it's a traversal where we try to process the parent node as soon as it is possible to do so, as this allows to limit the amount of temporary contribution blocks. If we consider the trees of Fig. 7, and assuming that the depth-first search is used with nodes on the left processed first, the best case in terms of memory usage is the tree on the left where we

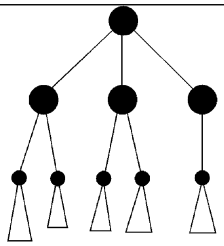
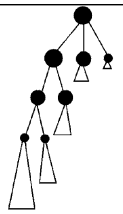
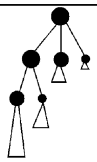
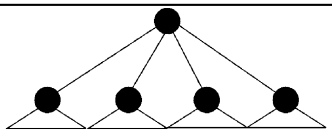
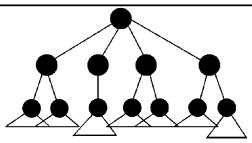
Reordering technique	Shape of the tree	Observations
AMD		• Deep well-balanced tree • Large frontal matrices on top of the tree
AMF		• Very deep unbalanced tree • Small frontal matrices • Very large number of nodes
PORD		• Deep unbalanced tree • Small frontal matrices • Large number of nodes
SCOTCH		• Very wide well-balanced tree • Large frontal matrices • Small number of nodes
METIS		• Wide well-balanced tree • Large number of nodes • Smaller frontal matrices (than SCOTCH)

Fig. 6. Shape of the trees resulting from various reordering techniques.

need to store the contribution blocks of at most two nodes simultaneously. On the other hand, the tree on the right-hand-side corresponds to the worst case because the contribution blocks of all leaves must be stored simultaneously.

Having as purpose to factorize large problems, we are interested in an out-of-core scheme either implicit (relying on system paging) or explicit. In both cases, since factors are not reaccessed once computed, they can be saved to disk. Therefore we focus in the following on the stack memory usage. We begin by studying the impact of reordering techniques on the stack memory. Then, we study how the memory consumption can be improved by using an optimal tree traversal.

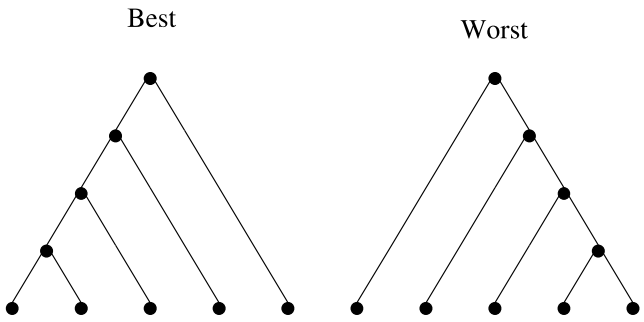


Fig. 7. Importance of the tree traversal.

5.1. Impact of reordering techniques on the memory

After studying the shape of the assembly tree, we now focus on the impact of the reordering techniques on the memory consumption in MUMPS.

Tables 9–11 present the stack memory traffic, the average stack size, and the peak of stack, respectively. For all these quantities, we neglected integer storage, so that the unit used is always the number of real entries. We observed that the storage needed by the integers is small compared to the one needed by the reals. For example, if we consider the matrix SHIP_003 with METIS, the total integer storage (factors + stack) for a sequential execution represents 2.3% of the storage needed by the reals. This ratio can slightly increase for small problems with limited fill-in like RMA10 where it reaches 4.4%.

Memory traffic: Table 9 gives the the sum of the sizes of contribution blocks for all the nodes of the tree. We can observe that the stack memory traffic for SCOTCH is the smallest (in most cases). This is due to the fact that SCOTCH has a smaller number of nodes compared to other reorderings. We also see that PORD and AMF lead to the biggest global stack memory traffic.

Average stack size: Table 10 gives the average size of the stack during execution, defined as the average stack sizes for all variations observed. We see that for PORD the average size of stack memory is smaller than for the other reorderings. This is

Table 9
Total amount of stack memory (millions of reals)

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	432.23	*286.21*	425.46	**709.76**	541.35
GRID	**252.53**	*147.23*	204.81	233.06	201.20
GUPTA3	144.83	*10.36*	**287.47**	231.41	236.74
MSDOOR	230.75	*175.43*	**247.32**	244.51	213.60
SHIP_003	509.66	*249.64*	590.01	**656.21**	496.76
PRE2	1186.45	*280.02*	**1557.68**	1267.98	623.01
RMA10	**20.74**	*13.15*	18.20	19.03	16.63
TWOTONE	305.18	*71.85*	272.93	**437.84**	214.58
XENON2	274.28	*203.04*	348.95	**507.88**	446.99

Table 10
Average size of the stack (millions of reals)

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	3.03	3.38	*2.16*	3.41	**6.10**
GRID	**3.18**	2.56	2.03	1.65	*1.21*
GUPTA3	15.20	*1.44*	**38.72**	7.31	8.41
MSDOOR	1.62	**2.42**	*1.14*	1.64	2.12
SHIP_003	5.76	6.74	*4.32*	7.01	**10.67**
PRE2	16.54	*6.07*	25.04	12.69	**47.31**
RMA10	0.16	0.13	*0.09*	**0.17**	0.14
TWOTONE	4.08	3.79	*2.62*	2.71	**5.62**
XENON2	4.69	4.89	*3.90*	6.11	**11.27**

Table 11
Peak of the stack (millions of reals)

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	10.69	9.53	*8.16*	11.26	**19.32**
GRID	**17.08**	11.91	5.83	4.17	*3.79*
GUPTA3	44.44	27.37	**93.96**	*25.21*	31.72
MSDOOR	4.12	5.22	*3.49*	4.18	**5.82**
SHIP_003	23.42	23.06	20.86	*20.77*	**32.02**
PRE2	*34.95*	36.16	65.60	84.29	**153.57**
RMA10	**0.43**	0.39	*0.28*	0.34	0.33
TWOTONE	13.23	13.54	11.80	*11.63*	**17.59**
XENON2	14.39	15.21	*13.14*	23.82	**37.82**

because PORD has deep trees (as shown in the previous sections) where we do not have to store a lot of contribution blocks at the same time. Compared with the other reorderings that also give deep trees like AMD, its tree often has fewer nodes and smaller frontal matrices which explains the difference in terms of the average size between AMD and PORD. Concerning AMF, we can see that its average is generally greater than the one of PORD. This is because the tree of AMF has larger branches (where there are a lot of memory operations) than the one of PORD. We can also observe that SCOTCH has a good average size because the number of nodes of its tree is smaller than for the other reordering techniques.

Peak stack size: Finally, Table 11 gives the peak of the stack memory observed during the factorization. We can see that the reorderings giving deep trees provide better (i.e., smaller) peaks of stack memory. Indeed, for our test problems, PORD and AMF have the smallest peak. This result is natural since deep trees do not need to store as many contribution blocks simultaneously as the wide trees given by SCOTCH or METIS. We can also observe that the peak of stack memory for AMD (which has a deep tree) tends to be greater than for other reorderings and particularly PORD and AMF (which also have deep trees). The first property that can help us to explain this phenomenon is that we have observed that the nodes on the top of the tree for AMD are larger than the ones for other reorderings. When these

large nodes start to be processed, the stack memory will contain large contribution blocks which will increase the size of the stack when processing the remaining subtrees.

The second property is that we saw that the tree of AMD is usually better balanced than those of AMF and PORD (see Table 5). We also observed that MUMPS chooses to process the largest node first, and the largest node normally tends to have the deepest subtree.

Fig. 8 illustrates the structural difference between AMD's and PORD–AMF's trees. For AMF and PORD, once the deepest subtree is treated, only smaller subtrees still need to be processed, requiring less memory. On the other hand, for AMD, after treating the first node, subtrees that are not far from the first one in terms of size still need to be processed. This will cause an increase of the stack memory because of the storage due to the additional contribution blocks involved. This explains why PORD and AMF behave better in terms of stack size than AMD, although all three have deep trees.

Summary: To summarize this section, we have seen that since reordering techniques have a strong impact on the shape of the assembly tree, they also have a strong impact on the memory usage in the factorization. Table 12 summaries the memory usage according to the reordering techniques. We have seen that PORD and AMF are the reorderings that use the smallest stack size (peak and average). For in-core executions, this should of course be related to the amount of work and the size of the factors, for which the following has been observed (see, for example, [13] as well as Figs. 4 and 5): for small matrices, factors with PORD and AMF are smaller than with SCOTCH and METIS, while for large matrices, METIS, SCOTCH and PORD give the smallest factors.

5.2. Optimal tree traversal order for memory usage

In the previous section, we measured the influence of reordering techniques on the dynamic memory usage. But as noticed the dynamic memory usage also depends on the tree traversal. The simple strategy of MUMPS for tree traversal may not be optimal in terms of dynamic memory usage. In this section we derive an algorithm which

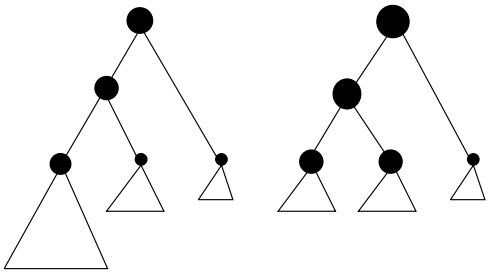


Fig. 8. Structural difference between AMF's tree and AMD's tree.

Table 12
Characteristics of the stack memory for different reordering techniques

	Peak of the stack	Average size of the stack
METIS	+	+
SCOTCH	+	+
PORD	−−	−−
AMF	−	−
AMD	++	++

The symbol "++" means a very big value, "+" means a big value, "−" a small value, and "−−" a very small value.

determines for a given assembly tree what is the optimal tree traversal for minimizing the dynamic memory usage.

We first define some notation which will be used for the description of the algorithm. Let i be a node in the tree and $nb\_children(i)$ the number of children of i. Children of i are denoted as $c_{i,j}$ where j varies between 1 and $nb\_children(i)$. Finally, let cb_i and $factor_i$ be the memory requirement to store the contribution block and the factors (respectively) of the frontal matrix i (as shown in Fig. 9).

In [17], Liu proposes an algorithm that finds the best traversal of the tree in terms of peak stack size for a sequential multifrontal approach such as the code `MA27` (available in the Harwell Subroutine Library). Based on his work, we present here a variant which is more appropriate to a distributed memory multifrontal solver such as `MUMPS`. One specificity of Liu's algorithm is that it assumes that the space for the frontal matrix of a node reuses the space of the contribution block coming from its last child, resulting in a memory gain of the size of this contribution block. In the case of `MUMPS` this optimization is not available because it cannot be implemented simply in a distributed memory parallel context. Thus, the application of Liu's algorithm on a distributed memory code does not always give the best traversal. Indeed, if we consider the tree given in Fig. 10 (with no overlap between factors and contribution blocks), the order given by Liu's algorithm is (a–c–d–b) which gives a peak of the stack of 13 ($=2+1+5+(3+2)$), obtained when b is assembled and before the

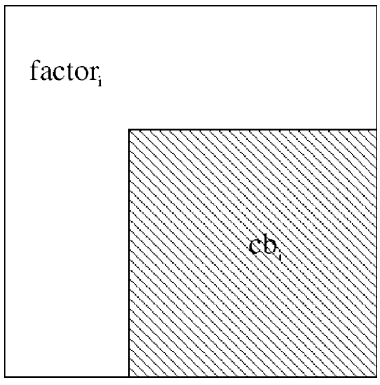


Fig. 9. Structure of a frontal matrix.

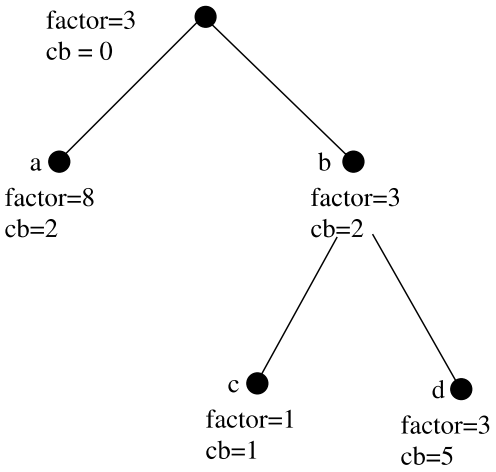


Fig. 10. Example of the application of LIU's algorithm.

contribution blocks of c and d are released. However, using the order (d–c–b–a), a peak of 11 is obtained (also when b is assembled). This explains why we cannot just apply Liu's algorithm in our case. Note also that Liu's initial algorithm is restricted to elimination trees where only one pivot is eliminated at each node. In our case we work on assembly trees (with amalgamated nodes).

Let M_i be the maximum amount of stack memory necessary to process the complete subtree rooted at node i. If i is a leaf then M_i is equal to $store_i = factor_i + cb_i$ real locations. For a parent node i, we must store in memory all contribution blocks of the children $c_{i,j}$ (if any) and the current frontal matrix; thus the assembly step requires a storage:

$$store_i + \sum_{j=1}^{nb\_children(i)} cb_{c_{i,j}}$$

When processing a child node $c_{i,j}$ the stack will contain the first $j-1$ contribution blocks of the brothers of $c_{i,j}$ that have already been processed. The result is that the storage requirement at the time of factorizing the frontal matrix associated with $c_{i,j}$ is:

$$M_{c_{i,j}} + \sum_{k=1}^{j-1} cb_{c_{i,k}}$$

Thus, the storage requirement to process node i is recursively defined as:

$$M_i = \max\left(\max_{j=1,nb\_children(i)}\left(M_{c_{i,j}} + \sum_{k=1}^{j-1} cb_{c_{i,k}}\right), store_i + \sum_{j=1}^{nb\_children(i)} cb_{c_{i,j}}\right) \quad (1)$$

Since we want to minimize the peak of the stack, we should reduce the value of M for the root node(s). Adopting a theorem from [17] which says that the minimum of

```
Tree_Reorder (T):
  Begin
    for all i in the set of root nodes do
      Process_Child(i);
    end for
  End
Process_Child(i):
  Begin
    if i is a leaf then
      M_i=store_i
    else
      for j = 1 to nb_children(i) do
        Process_Child(c_{i,j});
      end for
      Reorder the children c_{i,j} of i in decreasing order of (M_{c_{i,j}} - cb_{c_{i,j}});
      Compute M_i using the formula (1);
    end if
  End
```

Fig. 11. Optimal tree reordering for minimizing stack memory peak.

$\max_j(x_j + \sum_{i=1}^{j-1} y_j)$ is obtained when the sequence (x_i, y_i) is sorted in decreasing order of $x_i - y_i$, we deduce that an optimal child sequence is obtained by rearranging the children nodes in decreasing order of $M_{c_{i,k}} - cb_{c_{i,k}}$.

Considering a tree T, based on this result, the algorithm given in Fig. 11 gives an optimal traversal of the tree in terms of the peak of the stack. This algorithm consists in sorting the children nodes of a node i in descending order of $M_{c_{i,j}} - cb_{c_{i,j}}$ and compute the new value M_i for the parent node i, using (1).

Finally note that we have implemented variants of this algorithm: one for minimizing the global memory (stack + factors) and one for minimizing the average stack size during execution (when the peak is not changed). A complete analysis of all variants is available in [14].

5.3. Experimental results

We performed the experiments of Section 5.1 again, after algorithm given in Fig. 11 has been applied to the tree. In Table 13, we report on the gain in stack memory usage after applying algorithm given in Fig. 11. The gain is computed between the value of peak of stack memory of standard `MUMPS` and `MUMPS` where we postprocess the tree using the algorithm of Fig. 11. We can observe that the algorithm gives good results with METIS and SCOTCH. This can be explained by the fact that these reorderings generate wide trees where the traversal is very important in terms of memory usage. For AMD, we can see that the algorithm does not provide much gain. This is due to the shape of the tree of AMD. Indeed, it is deep and well-balanced. In addition, the brother nodes are not very different in terms of frontal matrix size. This implies that the order of the nodes does not have a strong impact for AMD (relatively well-balanced tree with balanced frontal matrices for brother nodes). Finally, for AMF and PORD we can see that the algorithm does not always give large gains. However, it gives very good results for some matrices like BMWCRA_1. The reason

Table 13
Percentage of reduction of the peak of stack memory observed using Algorithm 11

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	31.3	*22.2*	37.3	**37.4**	22.8
GRID	**31.4**	28.8	1.4	*0*	0
GUPTA3	**37.3**	*0*	1.8	26	8.6
MSDOOR	**20.9**	*0*	0	3.4	0
SHIP_003	24.8	**26**	8.8	*0*	0
PRE2	**0.1**	*0*	0	0	0
RMA10	**31.9**	16.4	24.4	*0*	17.6
TWOTONE	21.8	**23.9**	*0*	6	0
XENON2	21.1	**24.7**	*0*	0	0

for these different results is in the shape of the tree and the order of the brothers already implemented in MUMPS. We recall that MUMPS provides a basic sorting algorithm for the tree that processes the biggest child first. Generally, the biggest child roots the biggest subtree so this should be good for memory. Thus, for cases where the algorithm does not work well like matrix PRE2, or more generally with AMD and PORD, the order of MUMPS is in fact already optimal (thanks to the biggest node being processed before its brothers). To better illustrate the potential of Algorithm of the Fig. 11, we switched off the sorting mechanism of MUMPS; we observed that the gains are in that case much larger. For example, gains for SHIP_003 were 47.2%, 24.7% and 39.7% (instead of 8.8%, 0% and 0%), with PORD, AMF and AMD, respectively. Finally, for matrices like BMWCRA_1, and when the tree is better balanced, the order from MUMPS is not that good, and it is worth using the optimal tree traversal.

6. Memory usage for parallel executions

In this section we mainly focus on the size of the stack memory as a function of the number of processors and of the reordering technique used, when algorithm given in Fig. 11 is first applied to the tree. Because memory evolution depends on the distribution of nodes of the assembly tree onto the processors, we first describe the current scheduling strategy used in MUMPS. Then, we study the memory behaviour for parallel executions for different combinations of matrices and reorderings and analyze the factors that limit memory scalability.

6.1. Scheduling strategy used in MUMPS

MUMPS use a combination of static and dynamic mapping with distributed dynamic scheduling of the computational tasks. This is described in detail in [3,4]. The computation is driven by the assembly tree and a certain type of parallelism is assigned to each node. Fig. 12 summarizes the different types of parallelism available in MUMPS:

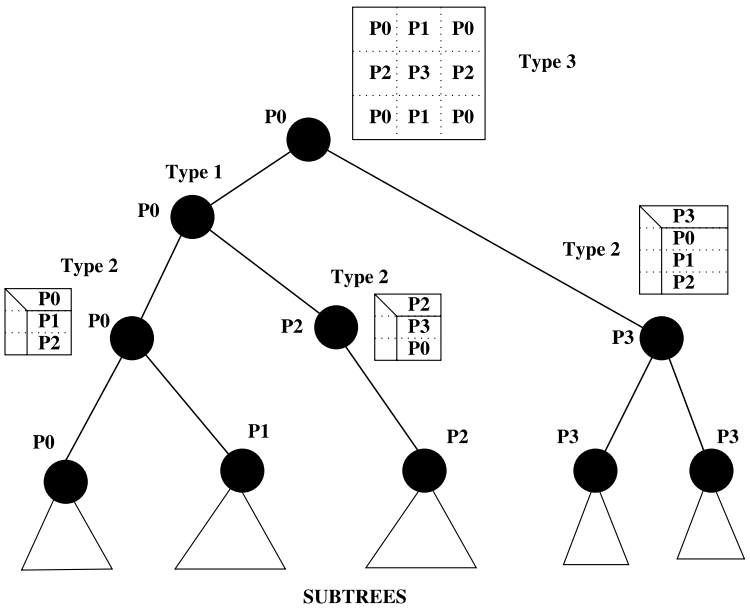


Fig. 12. Example of distribution of a multifrontal assembly tree over four processors.

- The first type uses the intrinsic parallelism induced by the assembly tree: each branch of the tree can be treated in parallel. A type 1 node is statically assigned to one processor which treats it when processors assigned to children nodes have communicated the contribution blocks. Leave subtrees are a set of type 1 nodes all assigned to the same processor. Those are determined using a top-down algorithm [11] and a subtree-to-process mapping is used to balance the computational work of the subtrees onto the processors.
- The second type corresponds to a 1D parallelism of the frontal matrices. For some nodes in the assembly tree, the front is so big that it must be treated in parallel for an adequate granularity. The front is then distributed by blocks of rows. A *master* processor is chosen statically during the symbolic preprocessing step, all the others (*slaves*) are chosen dynamically based on load balance considerations. The *master* processor is responsible for the eliminations of the fully summed pivot block. The master processor dynamically chooses its slave processors according to their workload (rather than memory usage) and assigns them new tasks. The load metric is the number of floating-point operations still to be done, where only the operations corresponding to the elimination process are taken into account (those are an order of magnitude larger than the operations for assembly). Note that the slave selection strategy is different between the symmetric and the unsymmetric cases. Indeed, the granularity is smaller for the symmetric case with more slaves chosen in the symmetric case [4].
- The third type of parallelism, which is a 2D parallelism, concerns the root node, which is processed by all processors using ScaLAPACK [6]: we use a 2D block cyclic distribution.

The choice of the type of parallelism depends on the position in the tree, and on the size of the frontal matrices. For the top of the tree the mapping of type 1 nodes and masters of type 2 nodes is static and only aims at balancing the memory of the corresponding factors. Usually, type 2 nodes are high in the assembly tree (fronts are bigger), and on large numbers of processors, about 80% of the floating-point operations are done in type 2 nodes.

6.2. Parallel results

Tables 14 and 15 (respectively 16 and 17) show the maximum and average stack peak on 16 (respectively 32) processors for different matrices and reorderings.

Balance of the peak across processors: If we consider the difference between the maximum peak and the average peak, we observe that the balance is not perfect and that a better balance is obtained for METIS and SCOTCH. This can be explained by the fact that these reordering techniques generate well-balanced trees where all the subtrees are approximatively of the same size. Concerning AMF, the stack memory is very unbalanced. This is due to the shape of AMF's trees which are also very irregular and unbalanced. For such trees, the subtrees described in the previous section are also irregular. Some processors may for example begin to treat type 2 nodes when other ones are still processing subtrees and this can perturb the memory behaviour. This is also related to the mapping of the nodes of the tree and will be further discussed in Section 6.3.

Note that in the `MUMPS` scheduling strategy, only floating-point operations for the factorizations of frontal matrices are taken into account, the memory of the processors is not considered. Although this leads to a good balance of the workload, the memory load balancing is not perfect with a difference between the maximum peak and the average peak that can be significant.

Scalability of the stack peak: For matrices where the stack size is significant, if we compare the peak of stack memory measured in the sequential execution (Table 11) to the maximum peak of stack on 16 (Table 14) and 32 processors (Table 16), we do not observe a linear improvement of the memory usage: in parallel doubling the

Table 14
Max peak of the stack on 16 processors (millions of reals)

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	6.75	7.71	*5.90*	7.78	**12.63**
GRID	**4.12**	3.97	3.38	*1.93*	3.52
GUPTA3	9.27	4.99	8.84	**16.34**	*4.88*
MSDOOR	**3.03**	2.78	*1.62*	1.80	2.62
SHIP_003	10.02	7.91	5.72	*5.01*	**11.01**
PRE2	9.72	9.96	12.83	*8.67*	**20.46**
RMA10	0.41	0.36	**0.42**	*0.35*	0.36
TWOTONE	**3.80**	3.64	*2.80*	3.58	3.65
XENON2	6.32	5.00	*4.63*	6.29	**9.75**

Table 15
Average peak of the stack on 16 processors (millions of reals)

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	5.30	5.54	*3.90*	5.32	**8.39**
GRID	**3.52**	3.42	2.61	*1.44*	2.85
GUPTA3	**6.12**	3.37	3.05	5.63	*2.92*
MSDOOR	1.55	**1.72**	*1.13*	1.13	1.61
SHIP_003	6.67	6.42	4.29	*3.47*	**8.24**
PRE2	6.90	*6.13*	9.27	6.71	**16.14**
RMA10	**0.25**	0.24	0.21	*0.20*	0.25
TWOTONE	2.36	*1.75*	2.14	2.43	**2.84**
XENON2	3.94	4.11	*3.26*	4.39	**7.66**

Table 16
Max peak of stack on 32 processors (millions of reals)

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	3.71	3.77	*3.44*	4.26	**6.69**
GRID	**2.48**	1.79	2.05	*1.25*	2.29
GUPTA3	7.73	*3.40*	8.85	**16.07**	3.57
MSDOOR	1.41	1.44	1.56	*1.18*	**1.74**
SHIP_003	5.48	4.29	3.15	*2.63*	**6.15**
PRE2	7.08	*5.92*	10.71	6.95	**10.93**
RMA10	**0.40**	0.36	0.36	0.35	*0.31*
TWOTONE	**2.78**	*1.93*	2.77	2.47	2.67
XENON2	3.92	3.52	*3.45*	4.64	**7.97**

Table 17
Average peak of the stack on 32 processors (millions of reals)

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	2.84	2.70	*2.39*	2.92	**5.08**
GRID	1.46	1.34	1.55	*0.84*	**1.86**
GUPTA3	**3.43**	*1.76*	1.93	3.10	2.17
MSDOOR	0.85	0.93	0.72	*0.70*	**1.01**
SHIP_003	3.01	2.87	2.26	*1.92*	**3.58**
PRE2	3.80	*3.25*	5.483	3.88	**8.36**
RMA10	**0.22**	0.20	0.16	*0.16*	0.20
TWOTONE	1.55	*1.17*	1.68	**1.79**	1.62
XENON2	2.10	2.46	*2.00*	2.94	**4.21**

number of nodes, i.e., the memory size, does not mean we are able to treat a problem twice larger.

Figs. 13 and 14 illustrate this point better. The first one gives the ratio between stack memory peak on 1 and 16 processors. We can see that we never reach the bold line that represents a perfect scalability; the best scalability observed is 11 but is in many cases between 2 and 6. This illustrates that the stack memory does not scale

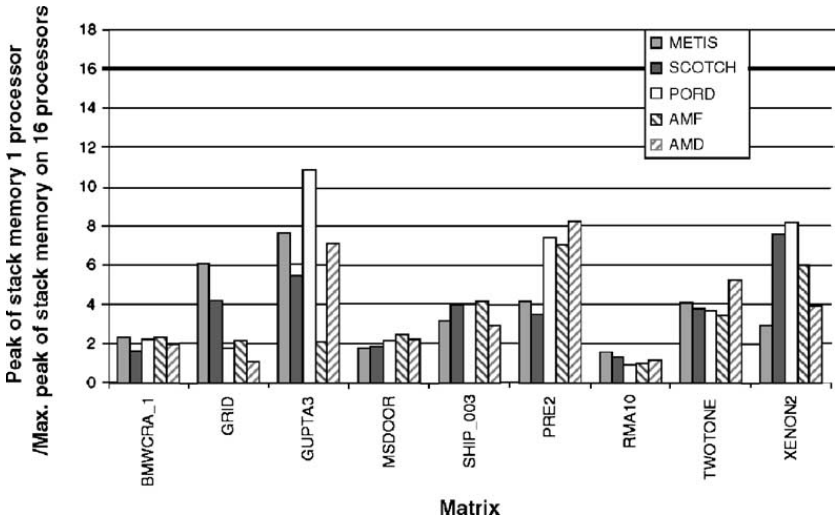


Fig. 13. Ratio between stack memory peak on 1 and 16 processors.

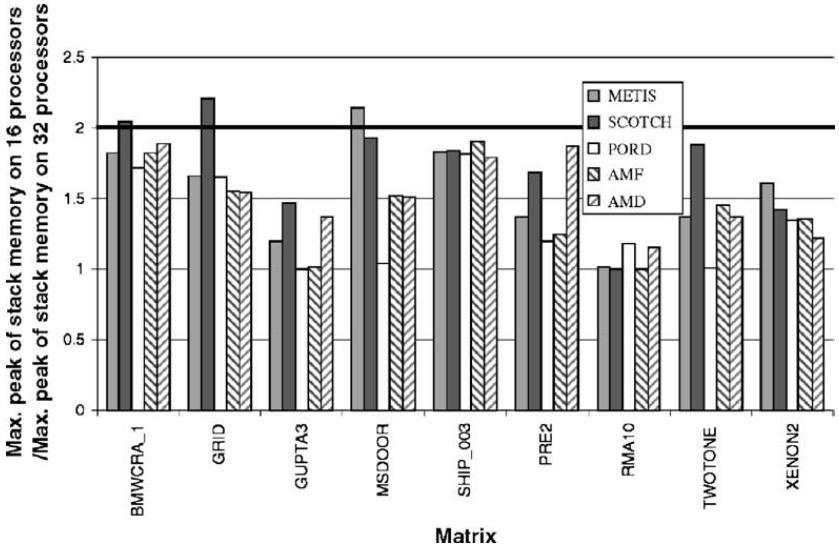


Fig. 14. Ratio between stack memory peak on 16 and 32 processors.

well (except for some cases like GUPTA3 with PORD). Fig. 14 gives a comparison between the peak of the stack on 16 and 32 processors. This time, the stack starts to scale better, although not linearly with the number of processors. The scalability is generally better for the symmetric case because more processors are used for each type 2 node than in the unsymmetric case (see Section 6.1, and [4] for more details).

Scalability of the total memory: Decreasing the stack memory is especially interesting in the case of an out-of-core approach. For an in-core solver like `MUMPS`, one is limited by the total memory (stack and factors). The ratio between the maximum peak of total memory on 16 and 32 processors is given in Fig. 15.

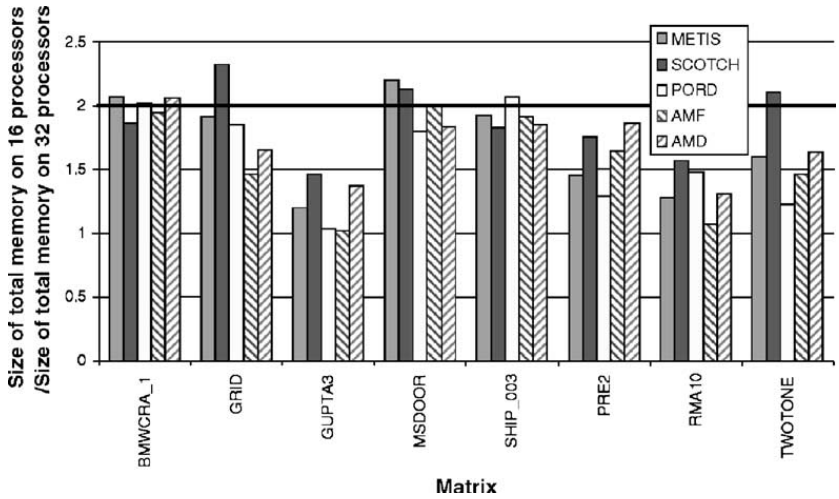


Fig. 15. Comparison of total memory peak for 16 and 32 processors.

We can observe that the scalability of the total memory is significantly better than for the stack memory and is even rather good for symmetric matrices. This is because the slave selection strategy aiming at balancing the flops will tend to provide a good balance of the factors on the processors. Furthermore contrarily to the stack size, the factors have a fixed size, independent of the number of processors. One consequence is that the size of the factors per processor decreases faster than the peak of the stack, and thus, the ratio stack/factors increases with the number of processors. So for problems where the stack is significant compared to the factors and/or for very large numbers of processors, the stack will play an important role even for an in-core parallel solver like MUMPS.

6.3. Factors impacting the memory scalability

In this section we give some remarks about the parallel memory behaviour of MUMPS and aim at finding factors that limit the scalability reported in the previous section. We illustrate this by analyzing some examples of typical situations where the peak of memory is reached and propose approaches that could improve both balance and scalability by avoiding such situations. We will particularly focus on the definition and assignment of subtrees and on dynamic scheduling. Note that in this section, when we say peak of memory, we mean the largest peak of memory across the processors.

• *Small matrices*: We observed in Section 6.2 that for a very small matrix like RMA10 the stack memory scalability is not good. In fact such small problems hardly exhibit any parallelism (no type 2 or type 3 parallelism) and this explains that the memory will not scale; independently of the mapping and of the number of processors, the peak observed is the same and is obtained during the sequential assembly of a node with the contribution blocks of its children.

• *Peak of memory inside a subtree*: We illustrate here the impact of the size of the subtrees on the memory behaviour of the solver. For example, considering the execution on 16 processors of matrix SHIP_003 with METIS, we have observed that the peak of stack memory is reached inside the first subtree treated sequentially by processor 12. In addition, processor 12 has not received any additional task from other processors. This shows that in that case the peak of stack memory is due to the static definition of the subtrees. A possible improvement would consist in splitting critical subtrees and distribute the resulting subtrees among several processors. Since the peak of the stack for a subtree can be determined statically, a strategy to avoid the lack of scalability due to that situation could be to split large subtrees until conditions such as $peak(subtree) < \frac{peak(whole\ tree\ in\ sequential)}{number\ of\ processors}$ and $peak(subtree) < \alpha \times memory\ on\ the\ processor, \alpha < 1$, are satisfied for all subtrees.

Fig. 16 illustrates the subtree splitting. We can see that the subtrees are smaller which is better for memory (but can be worst for performance). This simple example shows that the stack memory must be taken into account in the analysis (static) phase.

• *Slave selection*: An example that shows the importance of taking stack memory into account in the dynamic slave selection strategies of MUMPS is the execution on 32 processors of matrix SHIP_003 with PORD. For this execution the peak of stack memory is reached when processor 4, that has not finished one of its subtrees, is chosen as slave by processor 0. Since priority is given to the work received from other processors (slave work), the amount of memory needed by such tasks add up to the memory of the subtree being treated and increase the peak. We performed the following experiment: we put a synchronization barrier such that all processors wait for all subtrees to be processed. Even if processors store all contribution blocks of subtree roots, we observed a diminution of the maximum stack peak. This shows that this situation can be avoided by changing the slave selection strategy by giving preference to processors not involved in a subtree. This example illustrates that the memory should be taken into account in the selection strategy and a solution (not limited to the subtrees) is to design a general memory-aware dynamic scheduling strategy.

• *Order of subtrees*: A crucial point to obtain good performance and memory behaviour is the order in which the tasks are processed (particularly subtrees). Fig. 17 gives an example that illustrates the impact of the order of the initial pool of subtrees on both the performance and memory behaviour. In MUMPS children are recursively

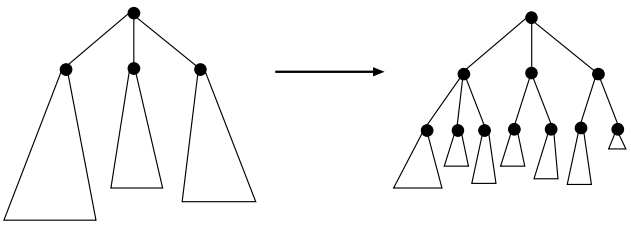


Fig. 16. Static improvement of the memory behaviour.

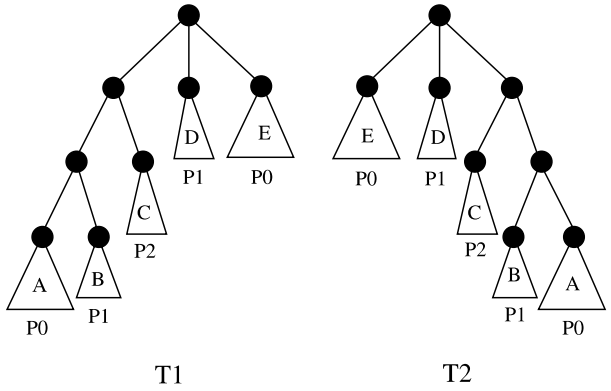


Fig. 17. Impact of the order on which subtrees are treated (processors are labeled P0, P1, P2 and subtrees are labeled A, B, C, D, E).

processed from left to right so that in the tree T1 (on the left), each processor will begin by its deepest (farthest to the root node) subtree. The bottom-up process will be time effective since the processors begin by the deepest parts of the tree. On the contrary, in the tree T2, each processor will begin by the subtree closest to the root. The bottom-up process will not exploit well the parallelism of the tree and be less time effective than T1. Concerning the memory, for tree T1, the first contribution blocks computed by P0 and P1 (corresponding to subtrees A and B) will be consumed quickly. On the other hand, for tree T2, processor P0 (respectively P1) will have to store the contribution blocks of the root node of subtree E (respectively D) until the root node of T2 can be activated. This leads to a larger memory usage for T2 compared to T1.

This simple example shows the great impact of the subtree sequence on each processor. It is important to note that Algorithm of Fig. 11 described in Section 5.2 will help to avoid the situation shown in the example because it tends to begin by the deepest parts of the tree. However, the application of the algorithm is not sufficient to ensure a good memory tree traversal for parallel cases since it does not take the mapping of the upper layers of the tree into account. A more sophisticated strategy to define the order of the subtrees on each processor should be based on both the tree topology and the mapping of the upper layers.

7. Conclusions

Whereas there are a lot of studies on the impact of reordering on fill-in, this paper provides an original study of the memory aspects of parallel multifrontal solvers, and in particular links between the reordering technique and the stack memory usage. We began our study with the impact of reordering techniques on the assembly tree and have observed that reordering techniques like METIS and SCOTCH give wide well-balanced trees while reordering techniques like AMF and PORD (respectively AMD) give very deep unbalanced (respectively balanced) trees with a large number

of nodes. From these results, we showed that deep unbalanced trees are better in terms of memory occupation than wide well-balanced ones. We have also seen how the stack memory evolution not only depends on the shape of the tree but also on the tree traversal during the factorization and have experimented with a variant of the algorithm by Liu to find the best tree traversal (in terms of memory occupation) in a distributed memory multifrontal solver such as MUMPS.

In the parallel case, the stack memory not only depends on the shape of the tree but also on the distribution of the computational tasks onto the processors. Our experiments show that the stack does not scale perfectly with the MUMPS default scheduling strategy based on workload. We analyzed some limitations and presented some ideas that can help improving this behaviour. We believe that optimizing and balancing the stack memory usage for parallel executions requires new scheduling strategies that are memory-aware. Furthermore, the static mapping of the subtrees and the order in which subtrees assigned to the same processor are treated is of great importance for the stack memory. This will be the object of future work.

References

[1] P.R. Amestoy, T.A. Davis, I.S. Duff, An approximate minimum degree ordering algorithm, SIAM Journal on Matrix Analysis and Applications 17 (1996) 886–905.
[2] P.R. Amestoy, I.S. Duff, Memory management issues in sparse multifrontal methods on multiprocessors, International Journal of Supercomputer Applications 7 (1993) 64–82.
[3] P.R. Amestoy, I.S. Duff, J. Koster, J.-Y. L'Excellent, A fully synchronous multifrontal solver using distributed dynamic scheduling, SIAM Journal on Matrix Analysis and Applications 23 (1) (2001) 15–41.
[4] P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, Multifrontal parallel distributed symmetric and unsymmetric solvers, Computer Methods in Applied Mechanics and Engineering 184 (2000) 501–520.
[5] C. Ashcraft, R.G. Grimes, J.G. Lewis, B.W. Peyton, H.D. Simon, Progress in sparse matrix methods for large linear systems on vector computers, International Journal of Supercomputer Applications 1 (4) (1987) 10–30.
[6] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK: A portable linear algebra library for distributed memory computers—design issues and performance, Technical Report LAPACK Working Note 95, CS-95-283, University of Tennessee, 1995.
[7] I.S. Duff, R.G. Grimes, J.G. Lewis, The Rutherford–Boeing sparse matrix collection, Technical Report TR/PA/97/36, CERFACS, Toulouse, France, 1997. Also Technical Report RAL-TR-97-031 from Rutherford Appleton Laboratory and Technical Report ISSTECH-97-017 from Boeing Information & Support Services.
[8] I.S. Duff, J. Koster, On algorithms for permuting large entries to the diagonal of a sparse matrix, SIAM Journal on Matrix Analysis and Applications 22 (4) (2001) 973–996.
[9] I.S. Duff, J.K. Reid, The multifrontal solution of indefinite sparse symmetric linear systems, ACM Transactions on Mathematical Software 9 (1983) 302–325.
[10] I.S. Duff, J.K. Reid, The multifrontal solution of unsymmetric sets of linear systems, SIAM Journal on Scientific and Statistical Computing 5 (1984) 633–641.
[11] A. Geist, E. Ng, Task scheduling for parallel sparse Cholesky factorization, International Journal of Parallel Programming 18 (1989) 291–314.
[12] A. George, J.W.H. Liu, Computer Solution of Large Sparse Positive Definite Systems, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[13] A. Guermouche, J.-Y. L'Excellent, G. Utard, On the memory usage of a parallel multifrontal solver, Technical Report RR-4617, INRIA, 2002. Also LIP Research Report RR(2002)-42.
[14] A. Guermouche, J.-Y. L'Excellent, G. Utard, Analysis and improvements of the memory usage of a multifrontal solver, Technical Report RR-4829, INRIA, 2003. Also LIP Report RR(2003)-08.
[15] G. Karypis, V. Kumar, METIS—a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices—Version 4.0, University of Minnesota, September 1998.
[16] J.W.H. Liu, Modification of the minimum degree algorithm by multiple elimination, ACM Transactions on Mathematical Software 11 (2) (1985) 141–153.
[17] J.W.H. Liu, On the storage requirement in the out-of-core multifrontal method for sparse factorization, ACM Transactions on Mathematical Software 12 (1986) 127–148.
[18] J.W.H. Liu, The role of elimination trees in sparse factorization, SIAM Journal on Matrix Analysis and Applications 11 (1990) 134–172.
[19] E. Ng, P. Raghavan, Performance of greedy heuristics for sparse Cholesky factorization, SIAM Journal on Matrix Analysis and Applications 20 (1999) 902–914.
[20] F. Pellegrini, SCOTCH 3.4 user's guide, Technical Report RR 1264-01, LaBRI, Université Bordeaux I, November 2001.
[21] F. Pellegrini, J. Roman, P.R. Amestoy, Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering, Concurrency: Practice and Experience 12 (2000) 69–84. (Preliminary version published in Proceedings of Irregular'99 LNCS 1586, pp. 986–995.)
[22] E. Rothberg, R. Schreiber, Efficient methods for out-of-core sparse Cholesky factorization, SIAM Journal on Scientific Computing 21 (1) (1999) 129–144.
[23] E. Rothberg, Stanley C. Eisenstat, Node selection strategies for bottom-up sparse matrix ordering, SIAM Journal on Matrix Analysis and Applications 19 (3) (1998) 682–695.
[24] J. Schulze, Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods, BIT 41 (4) (2001) 800–841.

Adaptive Paging for a Multifrontal Solver*

Olivier Cozette
LaRIA - Université de Picardie Jules Verne
Olivier.Cozette@u-picardie.fr

Abdou Guermouche
INRIA GRAAL Project - LIP, ENS-Lyon
Abdou.Guermouche@ens-lyon.fr

Gil Utard
LaRIA - Université de Picardie Jules Verne
Gil.Utard@u-picardie.fr

ABSTRACT

In this paper, we present a new way to improve performance of the factorization of large sparse linear systems which cannot fit in memory. Instead of rewriting a large part of the code to implement an out-of-core algorithm with explicit I/O, we modify the paging mechanisms in such a way that I/O are transparent. This approach will be helpful to study the key points for getting performance with large problems on under sized memory machines with an explicit out-of-core scheme. The modification is done thanks to the MMUM&MMUSSEL software tool which allows the management of the paging activity at the application level. We designed a first paging policy that is well adapted for the parallel multifrontal solver `MUMPS`. We present here a study and we give our preliminary results.

Categories and Subject Descriptors: D.4.2 [Storage Management]: Virtual Memory. G.1.3 [Numerical Linear Algebra]: Sparse, structured, and very large systems (direct and iterative methods).

General Terms: Algorithms, Perfomance.

Keywords: Sparse Numerical Algorithm, Multifrontal Method, Out-of-Core Computation, Virtual Memory Paging.

1. INTRODUCTION

Sparse direct methods and in particular multifrontal methods are robust and efficient techniques to solve large sparse systems of linear equations. However, they are known for their relatively large memory requirements compared to iterative methods so that an in-core execution is not always possible: sometimes, large problems mail fail to be solved because of a lack of memory on the processors [12].
A solution to deal with such large problems is to design a *out of core* solver where the computation is rescheduled with explicit I/O, which is a formidable task [6]. To avoid the reorganization of the existing code, a trivial solution is to use the *virtual memory* mechanism of operating systems where the I/O schedule is transparent. Unfortunately, it is know this solution is generally inefficient if standard *paging policies* are employed (*demand driven* and LRU like page replacement strategy).
In [16], Liu shown Multifrontal method theoretically reduces paging activity comparing to other direct methods, mainly due to a better locality of memory references. However, he shown that for large problems locality gains are avoided by the extra working temporary storage needed by the multifrontal method, which consumes more than the available memory and re-introduces paging activities. So he proposed an hybrid computation method where computation is switched to a conventional column Cholesky algorithm when the the working storage required becomes to big.
In this paper we propose another approach where all the computation is done with the multifrontal method, to conserve locality benefits, and where the extra paging introduced by the working storage is avoided by a better paging policy which is aware of the computation scheme.
In previous work we introduced a new tool, called MMUM&MMUSSEL [5], which allows the management of the paging activity at the application level. Thanks to this tool, we are able to substitute new *paging policies* to the standard one (e.g. introduce prefetching, use other page eviction strategy, ...). Thus, with the knowledge of the memory access pattern of the application, we can design better paging policies which improve the execution time by a better schedule of I/O. Generally, there are very few modifications of the original code of the application. The modification usually consists in instrumenting source code to give memory access information to the new paging scheduler.
In this paper, we present a first instrumentation of `MUMPS`, a parallel multifrontal solver, with MMUM&MMUSSEL in order to optimize the paging activity during the computation. This paper is concerned with the factorization process. First, we give a description of the multifrontal method to solve sparse system, and we focus on the memory access pattern for the sequential case. Then we study the behaviour of the standard paging policy (LRU) with the multifrontal method, and exhibit a better pagination strategy. We present the MMUM&MMUSSEL tool and describe how we implemented our new paging scheduler. Finally, We present first result and conclude.

*This work is supported by a grant of the "Pôle de Modélisation de Picardie" and by a grant of the INRIA OURAGAN ARC.

ICS'04, June 26–July 1, 2004, Saint Malo, France.

2. THE MULTIFRONTAL METHOD

Like other direct methods, the multifrontal method [8, 9] is based on the elimination tree [17], which is a transitive reduction of the graph of the symmetrized filled matrix $(A+A^T)$. The elimination tree is the smallest data structure representing dependencies between operations. In practice, we use a structure called assembly tree, obtained by merging nodes of the elimination tree whose corresponding columns belong to the same supernode [3]. We recall that a supernode is a contiguous range of columns (in the factor matrix) having the same nonzero structure.

Figure 1 gives an example of a matrix and its associated assembly tree. From the initial matrix, an assembly tree with three nodes (each corresponding to one supernode) is derived. The two first independent leaf nodes contribute to the computation of the third.

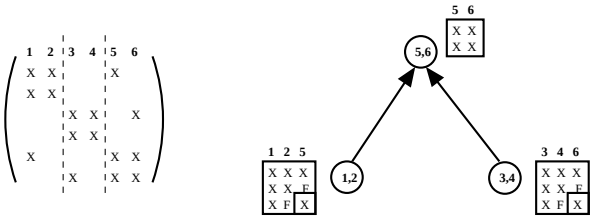


Figure 1: A matrix and the associated assembly tree.

In the multifrontal approach, the factorization of the matrix is done by performing a succession of partial factorizations of small dense matrices called *frontal matrices*, and associated to each node of the tree. The frontal matrix is divided into two parts: the *factor* block, also called *fully summed* block, which corresponds to the variables which are factorized when the elimination algorithm processes the frontal matrix; and the *contribution block* which corresponds to the variables which are updated when processing the frontal matrix. Once the partial factorization is complete, the contribution block is passed to the father node. When contributions from all children are available on the father, they can be assembled (i.e. summed with the values contained in the frontal matrix of the father). The elimination algorithm is a postorder traversal (we do not process father nodes before their children) [18] of the assembly tree. In addition, only one node (task) is treated at a time.

2.1 Memory behaviour of the multifrontal method for the sequential case

The algorithm uses three areas of storage in a contiguous memory space, one for the factors, one to stack the contribution blocks (managed with a stack mechanism), and another one for the current frontal matrix [2]. During the tree traversal, the memory space required by the factors always grows while the stack memory (containing the contribution blocks) varies depending on the operations made: when the partial factorization of a frontal matrix is processed, a contribution block is stacked which increases the size of the stack (see "Factorization" in Figure 2); in opposition, when the frontal matrix is formed and assembled, the contribution blocks of the children nodes are then removed from the stack and its size decreases ("Assembly steps" in Figure 2).

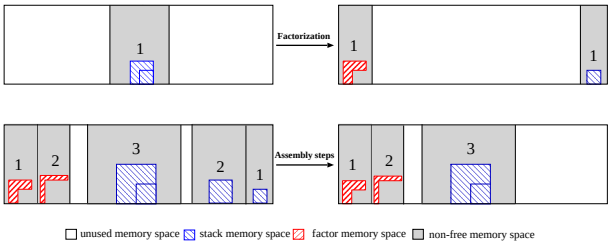


Figure 2: Memory behaviour of the multifrontal method.

3. PAGING SCHEME FOR THE MULTIFRONTAL METHOD

In this section we study the memory access pattern of the multifrontal method. We give the general behaviour of the accesses to the different parts of the memory of the solver. Then we present some optimizations to the accesses that will improve the performances of large applications on under sized memory machines.

As described in Section 2.1, the multifrontal method uses a memory space divided into three parts: Factors memory, stack memory and current frontal matrix memory. We have seen also that the factors are not reaccessed since their computation. Thus they don't have to be present in memory and can be stored on disk as soon as they are computed. Concerning the stack memory behaviour of the multifrontal method, we have seen that a contribution block is used only once during the assembly step. Thus, the data it contains can be destroyed after its treatment. A consequence is that we can imagine to tell to the operating system that this area doesn't have to be stored on disk if it is selected as the victim page for a swapping operation. As a result, the cost of writing this area on disk is suppressed which can be significant with very limited memory machines. We will define this operation as a *memory release*.

3.1 Limitations of the LRU policy for the multifrontal method

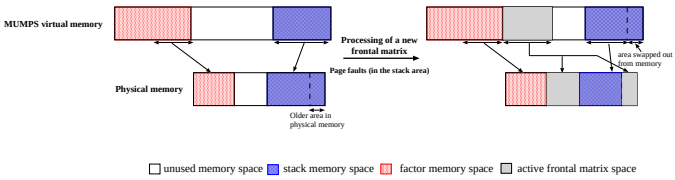


Figure 3: Illustration of system paging strategy.

Paging is a limiting factor to performance when the problem memory size is greather than the available memory. Thus, when system memory swapping occurs, the usual operating system paging policy (LRU : Least Recently Used) can take very bad decisions relative to the selection of the memory areas that will be stored on disk. Indeed if we consider the example given in Figure 3 where we have to process a new frontal matrix, and to assemble the contribution blocks by accessing all the stack memory (we do this assumption to illustrate the behaviour of the LRU policy), we can see that

this frontal matrix cannot be stored in physical memory and thus requires some system paging. Thus system paging will occur and selects, using the LRU policy, the deeper parts of the stack to be swapped out from memory. Furthermore, since the assembly operations have to access to this area (the deeper parts of the stack), some page-faults will occur. This example illustrates that the LRU standard policy of the operating system is not always well-adapted to the memory access pattern of the multifrontal method. The page faults could be avoided if we select the areas corresponding to factors for the swap-out operation. This can be done by "telling" to the operating system that this area does not have to be present in memory.

4. ADAPTIVE PAGING FOR MUMPS

In the previous section we showed that the standard LRU paging policy is not always well suited to deal with the memory access scheme of a multifrontal solver. We presented a better strategy of paging which is based on the knowledge of the computation process. The main idea is to instrument the factorization code with some hints which are used to drive the paging activity. In this section we describe how we implemented the mechanism which allow us to control paging. It is based on a separate process, called the memory monitor, which is running concurrently with the factorization process and manages the virtual space of the solver. We first present how the monitor is implemented, then we will show what are the interactions between the solver and the memory monitor.

4.1 MMUM and MMUSSEL

In this part, we present the memory management tool we used to control paging. There are two main techniques to manage the virtual memory at the application level.
The first category is based on the use of `mlock`, `mmap` and `mprotect` to map and unmap pages to virtual memory. It uses a *SIGSEGV* handler to intercept page faults. This handler uses some shadow system data in the stack to get the page fault address. Thus, it is system dependent. The *libsigsegv* library[1] is the most common example of such a tool and is implemented on lot of systems. The main drawback of this technique is that the memory must be locked and there are no interactions with the system especially if memory is needed for other applications.
The second category of virtual memory tools are based on special kernel like Grasshopper [1], micro-kernel L3/L4 [13]. These kernels allow user mode to control all paging management: page fault and swapout. Therefore, if the system needs more memory, it asks to swapout some pages to the user mode memory manager which chooses the pages to swapout. This category of tools is safer with the system, but are base on uncommon operating system.
Our tool gives the advantage of the second category, but it is working on a more common operating system (Linux). This tool is made up of a dynamically loadable kernel library, called MMUM (Memory Management in User Mode), and a Linux kernel module, called MMUSSEL (Memory Management at USer SpacE Level). The module interacts with the kernel for the management of memory pages.
The memory monitor is a standard process with its own virtual space running concurrently to the monitored process. It then attaches itself to the application by a call to the MMUM function

```
mmum_attach(pid,begin,end,pf,swo)
```

where `pid` is the PID of the monitored process, `pf` and `swo` are two handler functions. All page faults of an attached process in the area beginning at address `begin` and ending at address `end` are then processed by the monitor with the function `pf` which receives the address of the faulting page. Similarly, when the kernel decides to swap out some memory pages of an attached process (because of a lack of memory for example). The monitor is also invoked with the function `swo` which receives the set of memory pages to be swapped out. We have to recall that a swapout is not a direct consequence of a page fault. The goal of swapouts is to maintain a pool of free physical memory pages to satisfy future page faults. Swappers are triggered according a complex heuristic based on time and memory activity.
We present here a subset of functions available for the monitor to manage virtual spaces.

- `mmum_get(pid,a,b)`: remove the physical memory page associated to the virtual address `a` of the attached process PID `pid` and attach it to the virtual address `b` in the memory space of the monitor. When the attached process try to access to the page, a page fault is raised.
- `mmum_put(b,pid,a)`: move the physical memory page associated to the virtual address `b` of the memory space of the monitor to the virtual address `a` of the attached process PID `pid`.
- `mmum_cont(pid)`: restart the monitored process `pid` halted on a page fault.
- `mmum_release(pid,a)`: remove the physical memory page associated to the virtual address `a` of the attached process PID `pid` (data are lost). The virtual page released is considered as a never acceded page: a new physical page will be associated at the next access to this virtual page (no extra I/O).

Thanks to these functions, the monitor is able to manage all the memory space of the attached process. For instance, it can decide to prefetch some parts of data which are needed by the attached process in the future.
To illustrate the function of a memory monitor, we present the source code of a monitor (Figure 4) which maps a file to a memory area of the program presented in Figure 5. This last program creates a virtual memory area and launches the memory monitor which maps the file to it. Each time the monitored process accesses to a new memory page in this area, the function `pf` of the monitor is called to read the corresponding page from the file. The function `swo` is called when the operating system swaps out a page from the area. The page is then put in the memory space of the monitor which write it on the file and release it from physical memory.
Notice than the memory monitor is running concurrently with the monitored process. Moreover the attached process is not aware of the memory monitor: it accesses to its memory as usual. For the monitor, the only knowledge of the memory accesses of the application is given by the succession of page faults and swap-outs. One can imagine

[1] `http:///libsigsegv.sourceforge.net`.

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

int fd ;                                /* File descriptor of the mapped file */
void * start,end;                       /* Start and end addresses of the     */
                                        /* monitored memory area              */
char * buf;                             /* Buffer for reading one page        */


void pf(int pid,void * a)               /* There is a page fault at address a */
                                        /* in the monitored process pid       */
{
    llseek(fd,a-start,SEEK_SET);        /* Read of the page in file           */
    read(fd,buf,PAGE_SIZE);
    mmum_put(buf,pid,a);                /* Put the readed page in the virtual */
                                        /* space of the monitored process     */
    mmum_cont(pid);                     /* The monitored process can continue */
}


void swo(int pid, void * la, void *a)   /* There is a system swapout request  */
                                        /* a is the address of the swapout    */
                                        /* page in the monitored process la   */
                                        /* is the address of the swapout page */
                                        /* in  the monitor                    */
{                                       /* Write the page in the file         */
   llseek(fd,address-start,SEEK_SET);
   write(fd,local_address,PAGE_SIZE);   /* Free the page in  physical memory  */
   mmum_release(getpid(),local_address);
}


int main(int argc, char ** argv)
{
  int pid;
  fd=fopen("MappedFile.dat",O_RDWR);    /* The memory mapped file            */
  pid=atoi(argc[1]);                    /* PID of the monitored process      */
  start=atoll(argv[2]);                 /* Monitored area start address      */
  end=atoll(argv[3]);                   /* Monitored area end address        */
  buf=memalign(PAGE_SIZE,PAGE_SIZE);    /* Allocation of a buffer page       */
  mmum_attach(pid,start,end,pf,swo);    /* Attach monitored area             */
  kill(getppid(),SIGCONT);              /* Raise the monitored process       */
  while (1) sleep(100);                 /* Infinite loop                     */
}
```

Figure 4: The memory monitor code for the memory file mapping.

```
#include <stdio.h>

#define N 400*PAGE_SIZE
main()
{ char *A;
  char Pid[10],Add[20],Size[20];
  int i;

  A=memalign(PAGE_SIZE,N);            /* Allocate a memory area of size N   */
  sprintf(Pid,"%d",getpid());         /* PID of the attached process        */
  sprintf(Add,"%ld",(long)a);         /* Address of the memory area         */
  sprintf(Size,"%ld",(long)N);        /* Size of the memory area            */
  if (fork()==0)
    execlp("./monitor",Pid,Add,Size); /* Start the memory monitor            */
  wait();                             /* Wait the monitor start             */

  for (i=0;i<N;i++) A[i]++;           /* Accesses to the memory mapped file */
}
```

Figure 5: An example to memory file mapping using a memory monitor.

a smart memory monitor which is able to optimize paging activity based on this knowledge. For instance [10, 15] present some memory monitor devoted to specific class of application. However, for application with complex memory access pattern like multifrontal solvers, it seems difficult to derive such smart memory monitor. Another approach is to combine a static compiler analysis with a virtual memory management tool like the Todd Mowry's works [4] where the memory access patern is extracted by the compiler and used by the monitor. However, this technique fails if the memory access patern is only know at runtime like multifrontal solver. For such application, a solution to improve paging is to instrument the monitored process in such a way that it communicates some hints to the monitor describing its memory access scheme evaluated at runtime. In the next Section, we describe this interaction we done between MUMPS and the monitor.

4.2 MUMPS-monitor interaction scheme

The communication scheme between the monitor and `MUMPS` is based on a priority mechanism. `MUMPS` assigns different priorities to memory areas. Thus the monitor decides which area must be written to disk and which area must be read from disk according to their priorities. If an area have a priority equal to zero, it will be written to disk (it is the only area that is systematically written to disk). In the other hand, if the area have a priority equal to the maximal one, it must be present in memory. Finally, the monitor tries to keep in memory the areas having the biggest priorities. The function call used to set the priority to memory area is `reg_mem`.

Concerning the stack memory management, we designed the `mem_release` function which was described in section 3. We recall that this function tells to the monitor that a memory area is free. Thus the monitor can free it without writing of the data it contains to disk.

The communications between `MUMPS` and the monitor are done at special moments of the factorization. Thus, depending on the operation made, assembly or partial factorization, `MUMPS` emits informations aiming at helping the memory management at the monitor side. The communication scheme is given in Figure 6.

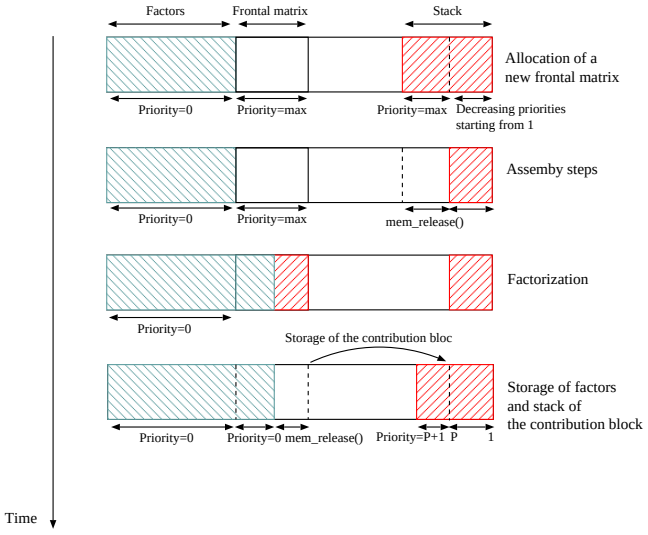


Figure 6: `MUMPS`-monitor Communication scheme.

The interaction between MUMPS and the monitor is done at each step of the life of the frontal matrix. A description of the different steps of the processing of the frontal matrix is given below:

Allocation of the frontal matrix. Before the allocation of a new frontal matrix, the priority of the corresponding memory area is set to the maximal priority with a `reg_mem`. This corresponds to "Allocation of a new frontal matrix" in Figure 6.

Assembly steps. Before each assembly step, MUMPS sets the priority of the areas that will be accessed in this step to the maximal priority by calling `reg_mem` : these areas must be in (physical) memory before the starting of the assembly steps. Furthermore, once the assembly step of a contribution block is done, the corresponding memory area is freed by a call to `mem_release`.

Factorization. Since the priority of the memory area corresponding to the matrix has already been set to the maximal priority, its factorization is done without interactions between MUMPS and the monitor. Once the factorization is done, the contribution block of the frontal matrix is pushed at the top of the stack memory. The priority of the contribution block is then set to the priority of the element that was at the top of the stack plus 1 (priority_cb=priority_top_of_stack+1). Concerning the factors corresponding to the frontal matrix, their priority is set to zero since they will not be reaccessed.

Concerning the factorization of a frontal matrix, it is important to note that once the computation starts, page faults are managed by the standard paging policy (LRU like). That is, if the frontal matrix does not fit in memory, then there will be an increase of the paging activity due to the lack of temporal locality, which can be critical.
In the following section we present some optimizations to the memory access pattern of MUMPS using the monitor.

4.3 Memory access optimizations

We will begin by a description of the stack mechanism used in MUMPS using the monitor. Then we present some improvements to the assembly operation aiming at ensuring better performance on large problems.

Optimizing the stack process

When the factorization of a frontal matrix have been done, the corresponding contribution block must be stacked (pushed at the top of the stack). The stacking process is described in Figure 7. It moves the parts of the contribution block to the top of the stack and it makes the factors contiguous in the memory. Thus, we can see in Figure 7 that the process generates some free space in the memory of MUMPS. These areas (free space) can be written to disk in very limited-memory machines where the memory cannot contain the current frontal matrix and its corresponding contribution block. This can be avoided by *releasing* these areas using the `mem_release` function provided by the monitor. Thus, the cost of writing the free areas to disk can be avoided which can be significant.

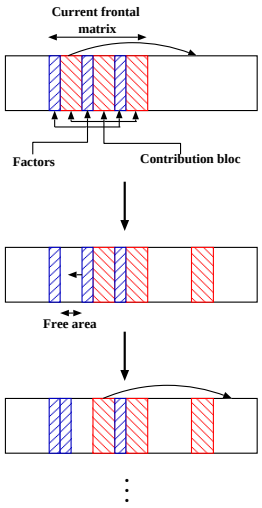


Figure 7: The stacking operation.

Optimizing the assembly operations

The assembly steps of a matrix consists in parsing the values contained in the area corresponding to the contribution blocks of the children nodes and to sum them to some values of the frontal matrix. Thus, we have to access to the memory area at the top of the stack memory (where are stored the contribution blocks of the children nodes), and access at some regions of the current matrix. This operation can be expensive if the memory space needed by the current frontal matrix and the corresponding contribution blocks is bigger than the physical memory size. Indeed, in this configuration, a lot of page faults may occur during the assembly steps which can be very costly. To improve the performance of the assembly steps in such a situation, we free the memory corresponding to the contribution blocks line per line during the traversal of the area using the `mem_release` function like shown in Figure 8. Thus, if a page fault occurs, the system will use the freed areas to load the needed memory pages. This allows to avoid the cost of writing the pages corresponding to the free areas to disk. Note that, liberation operation can be done on blocks of lines to minimize the cost of the calls to `mem_release`. The size of the blocks is determined according to the size of the physical memory.

4.4 Monitor implementation

The Figure 9 describes the implementation of the monitor and its interaction with MUMPS. The monitor is composed of two processes: the first one processes the memory requests from MUMPS (`reg_mem`, `mem_release`) and its page faults, the other one is devoted to the prefetch of large memory areas. This allows the processing of memory requests during big I/O operations.
Each time the monitor process receives a `reg_mem` request, it updates a memory map of the MUMPS memory area (a list of memory areas with their corresponding priority). It selects which area must be swapped in or swapped out in such a way that areas with high priority are kept in the available physical memory. Selected areas are then swapped in or out and large memory areas (greater than 1MB) are

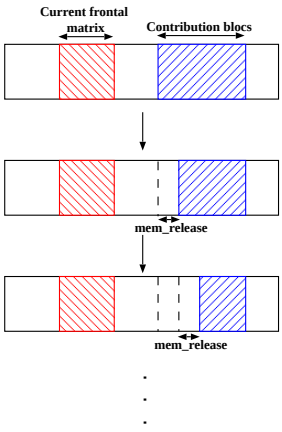


Figure 8: Assembly steps optimization.

prefetched by the prefetch process.
Each time the process receives a `mem_release` request, it frees the physical pages associated to the released memory area. Note that this function is a blocking function at the MUMPS side. Indeed, since the monitor process and MUMPS are executed concurrently, the release function may be executed asynchronously. A consequence is that if `MUMPS` reaccesses the released memory area before the treatment of the corresponding request at the monitor side, a lost of the new data will occur.
The page fault event and swap out event coming from the operating system are processed as usual. For a page fault the incriminated page is read from the swap device. Concerning the swap out, the incriminated pages are written to the swap device. Notice that the goal of the monitor is to keep in memory all the needed pages for the computation and it must also prefetch future acceded page. Thus, the monitor will reduce the number of page faults or swap outs that occur during the computation.

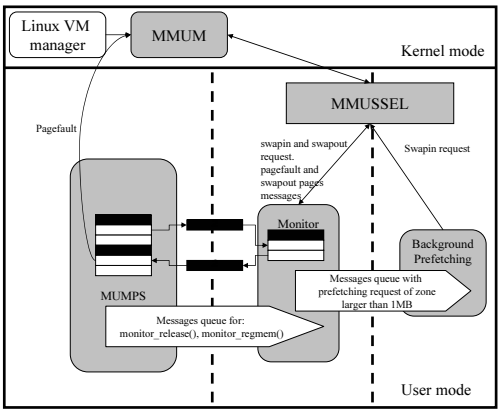


Figure 9: Relationships between MUMPS, MMUM/MMUSSEL and the MONITOR

4.5 Implementation issues

The implementation of the mechanisms described in the previous sections has been done by introducing some function calls to the source code of `MUMPS`. The pseudo-code that corresponds to the different communication mechanisms between `MUMPS` and the monitor is given on Figure 10.

```
C     mem_release of the area corresponding to the
C     range of indices [BEGIN_INDICE,END_INDICE].
      CALL MEM_RELEASE(A(BEGIN_INDICE),A(END_INDICE)
     *                 ,REQUEST_NUMBER)

C     sets the priority of the area corresponding
C     to the range of indices
C     [BEGIN_INDICE,END_INDICE] to PRIORITY.
      CALL REG_MEM(A(BEGIN_INDICE),A(END_INDICE),
     *             PRIORITY,REQUEST_NUMBER)
```

Figure 10: MUMPS and MMUM&MMUSSEL interactions.

The implementation of the communication between `MUMPS` and the monitor has been done by adding a small number of lines to the source code of `MUMPS` (less than 100 lines while the source code of `MUMPS` is of about 120 000 lines).

5. EXPERIMENTAL RESULTS

To illustrate the gains obtained with the monitor, we experiment our strategies on several problems (see Table 1) extracted from either the Rutherford-Boeing collection [7], the collection from University of Florida[2] or the PARASOL collection[3]. The tests have been performed on a cluster of six nodes equipped with one Alpha EV56 processors at 533 MhZ. Each node has a maximum of 128 MB of memory. We for our tests used METIS [14] as reordering technique.

Matrix	Order	NZ	Type	Description
GUPTA3	16783	4670105	SYM	Linear programming matrix (A*A')
SHIP_003	121728	4103881	SYM	Ship structure
THREAD	29736	2249892	SYM	Threaded connector/contact problem
TWOTONE	120750	1224224	UNS	AT&T,harmonic balance method.
XENON2	157464	3866688	UNS	Complex zeolite,sodalite crystals.

Table 1: Test problems."SYM" stands for symmetric, "UNS" for unsymmetric.

5.1 Sequential execution

We experimented our test problems on one node using 64 MB and 128 MB with and without our memory monitor. This choice is motivated by the fact that the same behaviour we will obtain on such machines can be observed on machines equipped with larger memory for lager problems.

Size	GUPTA3	SHIP_003	TWOTONE	XENON2	THREAD
64 MB	323	4 472	1 496	3 224	1 445
128 MB	350	2 303	598	1907	742

Table 2: Execution Time of `MUMPS` without the monitor (in seconds)

[2] `http://www.cise.ufl.edu/~davis/sparse/`
[3] `http://www.parallab.uib.no/parasol`

Table 2 gives the execution time without monitor. We can observe the execution time increases with the decrease of the size of the physical memory used except for the GUPTA3 matrix. This can be explain by the swap-out mechanism: with less memory, swap-outs are triggered more often and we can observe a better overlap of these swap-outs by the computation (swap-outs are asynchronous to the computation). With more memory, swap-outs are triggered usually later, and the computation process is stopped by the lack of memory pages.

Size	GUPTA3	SHIP_003	TWOTONE	XENON2	THREAD
64 MB	179 K	852 K	226 K	505 K	222 K
128 MB	132 K	373 K	101 K	202 K	99K

Table 3: Number of page faults (K: Kilo, M: Mega)

Table 3 gives the number of page faults without monitor. We can see that the number of pagefaults grows with the reduction of the size of the physical memory.

Memory	Matrix	Monitor User	Monitor System	Read async	Monitor Total	Total Time	Monitor overhead (%)
64	SHIP_003	48	6	262	317	4206	7,56
64	XENON2	38	5	30	74	2691	2,78
128	SHIP_003	25	4	67	97	1925	5,05
128	XENON2	25	4	0,4	30	1912	1,59

Table 4: Monitor overhead (in seconds)

Table 4 gives the overhead of the monitor and shows that the overhead is low for monitor operations. However, the time spent in the function calls corresponding to the *reg_mem* and *mem_release* in the `MUMPS` process can be significant like for the TWOTONE matrix for example. This overhead can be decreased by reducing the number of function calls in `MUMPS` side.

Figure 11 gives the percentage of gain (or loss) for page faults, swap-out using the memory monitor for 64 and 128 MB.

First, we observe that we obtain a reduction of the number of swapouts for all cases. This is first due to good decisions relative to the choice of the pages to swapout using the priorities set with the *reg_mem* function. In addition, the release mechanism helps to decrease the number of swapouts since the released pages have not to be swapout.

Concerning the number of page faults without release, we observe gains for all case, because our priority information allows the monitor to have a better swapout policy. Thus, the needed pages are less often swapout. We can also observe that the number of page faults increases when we activate the release mechanism. This is due to the situation where a page is released and reaccessed immediatly after the release giving a page fault. It is important to note that this kind of page faults are not costly in terms of time since the physical page is available (it has been freed by the release). This kind of page faults occur generaly when the monitor has not yet processed the *reg_mem* request when the application (`MUMPS`) accesses to the corresponding released memory area.

Figure 12 gives the execution (factorization) time decrease with 64 MB and 128 MB using our monitor in comparison with the times measured with the standard paging policy.

Usually, a reduction of the number of page faults leads to a better execution time, with an exception for TWOTONE with 128 MB. For this matrix the increase of the factorization time is due to the overhead of the memory monitor. Indeed, there is a large number of requests sent by the application to the monitor concerning fine grain tasks (less than 10 KB). To better illustrate, we measured 95 000 requests with the monitor for only 600 seconds of computation, whereas with SHIP_003 with 128 MB, we obtain 35 000 requests for 1 200 seconds.

Finally, we observe that gains for execution time are better for 64 MB than for 128 MB. This is due to the fact that paging activity is more important with 64 MB. Thus, reducing the number of page faults with 64 MB of memory will have a bigger impact than with 128 MB (see Table 3). To better illustrate this behaviour, we tested our memory monitor with 32 MB of physical memory. We obtained, a reduction of 37.4% for the GUPTA3 matrix and 27.46% for the XENON2 matrix in comparaison with standard paging policy. It is important to note that for such physical memory size we must have an *out-of-core* factorization scheme for the frontal matrices that does not fit in memory. This explains why we don't have larger gains with a 32 MB physical memory.

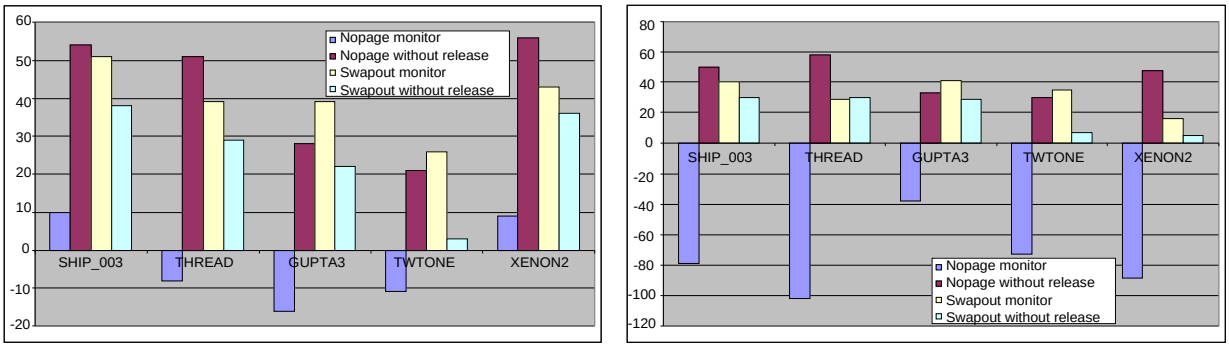
6. CONCLUSION

In this paper, we presented a new way to improve performance of the factorization of large sparse linear systems which cannot fit in memory. Instead of rewriting a large part of the code to implement an out-of-core algorithm with explicit I/O, we modified the paging mechanisms in such a way that I/O are transparent. This is done by the use of a tool, MMUM&MMUSSEL, which allows the management of the paging activity at the application level. We studied the memory access pattern of a parallel multifrontal solver, `MUMPS`, and how to improve paging during the computation. In such a solver, the global factorization is reduced to a set of partial factorizations on smaller matrices (called frontal matrices). The memory of the multifrontal method is made of three parts: factors which are in a write-once area, contribution blocs are in a write-once/read-once area, and a working area containing the current frontal matrices. We described the design and the implementation of a *memory monitor* that provides a first paging policy. This policy tries to keep only useful data in memory and avoids useless I/O for read once data (contribution blocs).

Experiments showed that these first paging policies are able to reduce the number of page faults (I/O) which can lead to a decrease of the execution time. Furthermore, our strategy assumes that there is enough memory for each frontal matrix (the partial factorization is done in core): we focused only on the paging introduced by the extra working storage memory needed by multifrontal methods. So the gains are limited (and can be negative) if this assumption does not hold. Thus we have to improve paging during the partial factorization of a frontal matrix, in a way similar to the one presented in [5].

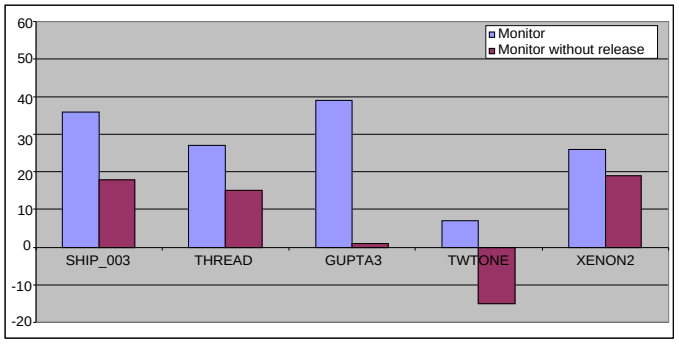
We focussed on the factorization step, the solve step can be also critical in an out-of-core execution scheme. Indeed, the solve step has to access to all the factor area. Furthermore, the cost of the operations made during the solve phase is not very high. Therefore, we must be able to prefetch factor blocks as fast as possible and to do sequential read to ensure a good speed of I/O operations.

It is also important to extend our monitor to the parallel execution scheme. The extension can be done in a natu-

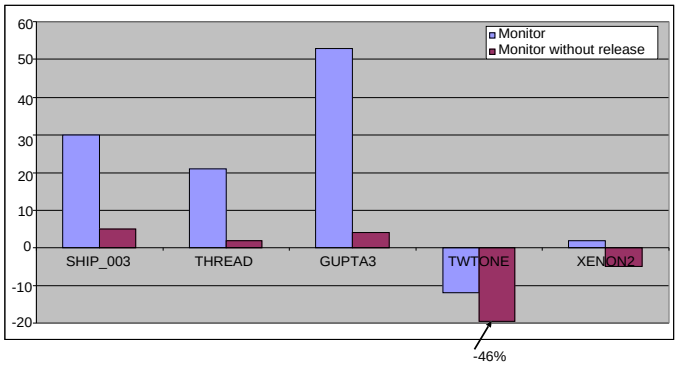


(a) 64 MB (b) 128 MB

Figure 11: Page faults and swap-outs reduction with the monitor with several memory sizes.



(a) 64 MB



(b) 128 MB

Figure 12: Execution time gain with several memory sizes.

ral way except for the stack memory area where the management can be more complex. It would be interesting to test the parallel implementation of `MUMPS` with the monitor with memory-aware scheduling strategies since the default scheduling strategies of `MUMPS` are workload based. Thus, a first step could be to use memory based scheduling strategies [11] which tries to balance memory among processors. A second step is to combine the scheduling with the paging policy by injecting the size of the physical memory of each working processor to the memory-based scheduling strategies.

7. REFERENCES

[1] R. D. B. Alan Dearle, A. Lindström, J. Rosenberg, and F. Vaughan. User-level management of persistent data in the grasshopper operating system. Technical report, University of Sidney, 1994.

[2] 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.

[3] C. Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, and H. D. Simon. Progress in sparse matrix methods for large linear systems on vector computers. *Int. Journal of Supercomputer Applications*, 1(4):10–30, 1987.

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

[5] E. Caron, O. Cozette, D. Lazure, and G. 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.

[6] E. Caron and G. Utard. On the performance of parallel factorization of out-of-core matrices. *Parallel Computing*, 30(3):357–375, 2004.

[7] I. S. Duff, R. G. Grimes, and J. G. Lewis. The Rutherford-Boeing Sparse Matrix Collection. Technical Report TR/PA/97/36, CERFACS, Toulouse, France, 1997. Also Technical Report RAL-TR-97-031 from Rutherford Appleton Laboratory and Technical Report ISSTECH-97-017 from Boeing Information & Support Services.

[8] 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.

[9] 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.

[10] G. W. Glass. Adaptative page replacement. Master's thesis, University of Wisconsin - Madison, 1998.

[11] A. Guermouche and J.-Y. L'Excellent. Memory-based scheduling for a parallel multifrontal solver. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004. To appear.

[12] A. Guermouche, J.-Y. L'Excellent, and G. Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9):1191–1218, 2003.

[13] G. Heiser. Implementation and performance of the mungi single-address-space operating system. Technical report, University of New South Wales CSE, 1997.

[14] G. Karypis and V. Kumar. MeTiS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0.* University of Minnesota, Sept. 1998.

[15] K. Krueger, D. L. A. Vahdat, and T. Anderson. Tools for the development of application-specific virtual memory management. Technical report, University of California, Berkeley, 1993.

[16] J. W. H. Liu. The multifrontal method and paging in sparse Cholesky factorization. *ACM Transactions on Mathematical Software*, 15:310–325, 1989.

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

[18] E. Rothberg and R. Schreiber. Efficient methods for out-of-core sparse cholesky factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, 1999.

Troisième partie

Stockage et archivage distribués à grande échelle

Chapitre 9

Présentation

Mon deuxième axe de recherche a été le stockage distribué et en particulier sur les systèmes pair à pair. Ce travail a débuté en 2001 alors que j'étais en détachement INRIA au LIP/ENS Lyon dans l'équipe Graal dirigé par Frédéric Desprez. Ce travail c'est inscrit tout d'abord dans le cadre de l'ACI Grid CGP2P [1] piloté par Franck Capello. Ce travail a ensuite donné lieu à un projet de valorisation par la création de l'entreprise UbiStorage en 2006, dont j'ai été le dirigeant, qui sera repris par l'entreprise Ugloo en 2014 jusqu'à ce jour.

Au début des années 2000, on a assisté à un fort engouement pour les systèmes de fichiers pair-à-pair. On peut citer Napster, Freenet, Gnutella où les fichiers sont distribués sur l'ensemble des PCs. Ces systèmes sont des alternatives à l'approche client/serveur du WEB.

En fait ces systèmes sont orientés essentiellement dans la mise en commun et dans la diffusion des données et des informations. Les objectifs avoués étant essentiellement ceux de l'anonymat des sources d'informations, l'usage ayant été principalement le piratage de la musique et des films. Le *business model* des principales majors en a été profondément ébranlé, et a donné lieu à l'émergence du streaming rendu possible par l'évolution des capacités des réseaux. Ces systèmes imposaient un surcoût non négligeable dans l'accès aux données. En général, il n'y a aucune garantie en ce qui concerne la fiabilité et la disponibilité des données.

Dans le même temps, d'autres travaux plus confidentiels, car non dédiés exclusivement à la diffusion de fichiers musicaux ou vidéo, ont été développés. On peut citer InterMemory, PAST ou Oceanstore. Ils se présentent comme des systèmes de partage d'espaces de stockage. À la différence des systèmes précédemment cités, ces nouveaux systèmes intègrent des mécanismes qui assurent la confidentialité et la pérennité des données.

Le principe général de fonctionnement des systèmes pair-à-pair repose sur des bases communes. Dans les réseaux pair-à-pair, il n'y a pas de distinction claire et nette entre clients et serveurs comme on le trouve dans de nombreuses architectures (Web, FTP, ...). Chaque pair qui compose le réseau est à la fois client et serveur. Les systèmes sont conçus de tels sortes qu'aucun des nœuds ne soit réellement indispensable pour le fonctionnement général : si un ou plusieurs nœuds sont défaillants, cela ne paralyse pas le système.

Ces systèmes pair-à-pair se caractérisent par des sur-réseaux ad hoc basés sur des connexions point à point formant un graphe faiblement connexe dont l'objectif est de fournir un mécanisme tolérant aux pannes de routage entre les nœuds. Ce mécanisme peut avoir de plus une fonction de localisation des données.

1. Calcul Global Pair à Pair

9.1 Système P2P dédiés au stockage

L'un des précurseurs est InterMemory [Che99, Gol98]. Il se caractérise par un mécanisme de protection des données extrêmement poussées, basée sur leur redondance, leur fragmentation et leur dispersion.

PAST [Dru01] est un dispositif de stockage de données développé conjointement par l'Université de Rice et Microsoft. Dans PAST, les nœuds et les fichiers possèdent des identifiants uniques. Les fichiers sont stockés sur les nœuds dont l'identifiant est le plus proche de l'identifiant du fichier. Le mécanisme de routage tolérant aux pannes est PASTRY. La pérennité des données est assurée par la réplication.

Enfin OceanStore [Kub00] est un vaste projet de stockage pérenne de données. C'est probablement le projet le plus abouti concernant le stockage de données pair-à-pair. Les données sont stockées sur des serveurs dédiés qui possèdent une forte connectivité et une large bande passante, situés, par exemple, chez les fournisseurs d'accès à Internet. Le système de routage tolérant aux pannes d'OceanStore est Tapestry [Zha01, She01]. Ces serveurs collaborent afin de fournir un service de stockage ayant la propriété d'ubiquité, c'est-à-dire que l'accès aux données se fait de façon transparente pour l'utilisateur de n'importe quels points d'accès.

Les données sont organisées suivant deux niveaux de stockage. Le premier niveau est celui qui va assurer la disponibilité des données même lors de la défaillance d'un ou plusieurs serveurs de stockage. Pour cela un mécanisme de redondance [Wea02a] semblable à celui utilisé dans InterMemory et d'autosurveillance des serveurs collaborant ensemble est mis en place. Le deuxième niveau de stockage est une réplication qui n'a pas pour but d'assurer la pérennité des données, mais la proximité de celle-ci. Il s'agit en fait de répliqua placés sur des serveurs proches de l'utilisateur pour lui garantir les meilleurs temps d'accès à ses données.

Les techniques de routage des requêtes et méthodes de redondance sont deux éléments qui caractérisent les systèmes de stockage pair-à-pair.

Pour ce qui est du routage des requêtes, une forme générale se dégage : les requêtes sont dirigées suivant un identifiant unique, ou considéré comme tel, qui caractérise un fichier. Chaque nœud est spécialisé pour un certain identifiant de fichier. Dans PAST et OceanStore, on considère que les nœuds ont une connexion stable et ont un identifiant unique. Dans PAST, le fichier est stocké sur le nœud d'identifiant le plus proche tandis que, dans OceanStore, ce n'est qu'un lien vers le nœud de stockage. Ces deux systèmes garantissent l'aboutissement de la recherche.

Le système statique de PAST complique l'équilibrage de charge, car les fichiers doivent être stockés sur le nœud dont l'identifiant est le plus proche de celui du fichier. Lorsqu'un nœud n'a plus de place, les auteurs de PAST proposent, dans [Row01b], plusieurs solutions pour y remédier : dans la première, le nœud qui n'a plus de place demande à ses voisins (des nœuds dont l'identifiant est proche du sien) de stocker les fichiers pour lui, le nœud d'origine conservant un lien vers les nœuds qui stockent réellement le fichier, une seconde méthode, plus radicale, consiste à changer l'identifiant du fichier.

Dans PAST, la pérennité est assurée par un système de réplication simple. Pour InterMemory et OceanStore, les garanties sont basées sur des systèmes de redondance plus sophistiqués à base de fragmentation que nous présentons plus en détail dans la suite.

Pérennisation des données : Redondance

La redondance est la clé de la pérennité des données dans un système de stockage distribué. En effet, stocker une donnée sur un seul nœud est relativement risqué, puisque ce nœud peut disparaître à tout moment sans prévenir. Pour cela, il faut que les informations soient redondantes pour que, même si des nœuds sont défaillants, il soit toujours possible de

récupérer l'information. L'exemple le plus simple de redondance est celui de la réplication des données, c'est-à-dire que l'on se contente de copier les données telles quelles sur plusieurs machines comme dans PAST ou Freenet. On notera r le facteur de réplication. Pour $r = 1$, un seul répliqua, la tolérance est la défaillance d'un nœud, pour $r = 10$, elle sera de 10 nœuds.

Dans un mécanisme de redondance avec fragmentation, les blocs de données de taille B sont découpés en s fragments de taille $\frac{B}{s}$. À partir de ces fragments, on détermine r fragments de redondance, eux aussi de taille $\frac{B}{s}$. Les $s + r$ fragments sont tels qu'à partir de s fragments quelconques il est possible de reconstituer les données initiales. L'ensemble des fragments sont disséminés sur des nœuds distincts. Le système tolère donc r défaillances. La réplication n'est qu'un cas particulier où $s = 1$ et r est le nombre de réplications.

Le ratio $\frac{s}{s+r}$ détermine l'espace utile, c'est-à-dire le rapport entre la taille de la donnée et l'espace de stockage. Par exemple, dans le cas d'une seule réplication ($s = 1$, $r = 1$), l'espace utile est de 50% = $\frac{1}{1+1}$. Pour 9 répliqua ($s = 1$, $r = 9$), il n'est plus que de 10%. Dans le cas où $s = 9$ avec le même facteur de tolérance $r = 9$, l'espace utile est alors de 50%. La fragmentation des blocs permet, à tolérance équivalente avec la réplication, de gagner en espace utile et donc de stocker plus d'informations dans le système. La méthode usuelle pour déterminer les fragments redondants est basée sur le codage de Reed Solomon.

Bien qu'avec un facteur de redondance adapté, on peut résister à un grand nombre de défaillances, un mécanisme statique ne permet pas d'assurer la pérennité des données. En effet, si on considère que la durée de vie des nœuds suit une loi classique de probabilité telle que la loi exponentielle, alors on a une forte probabilité d'avoir perdu plus de la moitié des nœuds au bout du temps moyen de la durée de vie d'un nœud. Pour assurer la pérennité, il est donc nécessaire d'introduire un mécanisme de réparation qui détecte et reconstruit les fragments de données perdus.

D'autre part, la plupart des études sur l'efficacité des mécanismes de redondance font généralement l'hypothèse que les pairs sont non corrélés en ce qui concerne les pannes. Des études ont montré que cette hypothèse n'est en général pas vérifiée. Des mécanismes pour détecter des corrélations entre les pairs ont été développés. Ils se basent sur différentes hypothèses sur la structure du réseau et des observations en temps réels. Il faut alors distribuer les données de manière à éviter les corrélations trop fortes de pannes [Wea02b].

Structure générale des systèmes P2P : les DHT

Les premiers systèmes avaient une architecture monolithique, i.e. où les différentes fonctionnalités du système étaient interdépendantes. La tendance actuelle pour la conception de nouveaux systèmes de fichiers pair-à-pair est de se baser sur une architecture type à trois niveaux . Cette architecture a été introduite par CFS (Chord File System [Dab01]). Le premier niveau propose un mécanisme de routage entre les pairs par un sur-réseau (*overlay*) tolérant aux pannes (Chord [Sto01], Pastry [Row01a], Tapestry [Zha01], ...). Le deuxième niveau implémente un dictionnaire distribué et redondant sur ce sur réseau (DHT, *Distributed Hash Table*), chaque entrée de ce dictionnaire étant composée d'une clef et d'un objet associé (e.g. le fichier). L'objet est inséré dans la DHT qui le réplique afin d'assurer un certain niveau de tolérance aux pannes. Enfin le dernier niveau s'appuie sur les deux précédents et est responsable de l'organisation logique des données.

En général un identificateur unique appartenant à un grand espace de noms est affecté à chaque pair. Les identificateurs sont choisis de telle manière que ceux-ci soient le plus dispersés dans l'espace de noms. Pour router les messages, chaque pair maintient une table de routage avec les identificateurs d'autres pairs et leur adresse IP.

En ce qui concerne la DHT (le dictionnaire distribué), il existe une fonction qui projette

l'espace des clefs du dictionnaire dans l'espace de noms des pairs, par exemple l'identité si les deux espaces de noms sont identiques. Lors de l'insertion d'un nouvel objet dans le dictionnaire, le pair qui le stockera est celui qui sera atteint par l'algorithme de routage en fonction de la clef projetée de l'objet. En général l'algorithme de routage désigne le pair qui à l'identifiant le plus proche de la projection de la clef.

L'objet inséré est répliqué par la DHT pour faire face à la disparition de pairs. Typiquement, chaque pair maintient une liste des pairs qui ont un identifiant "voisin" (au sens de l'algorithme de routage) dans l'espace de noms, l'objet est alors dupliqué sur ces pairs.

9.2 Description de mes travaux

Mes travaux dans ce domaine se sont focalisés sur deux axes. Le premier, plus académique, sur l'étude de la pérennité dans de tels systèmes de stockage pair à pair. Le second, sur la conception d'un tel système pour le stockage de données immutables (notamment pour les backups et l'archivage), qui a donné lieu à un projet de valorisation et à la commercialisation d'un tel système par la société Ugloo.

Quand on vise une grande pérennité, la redondance en soi ne suffit pas. Il faut lui adjoindre un mécanisme de régénération qui permet de maintenir cette dernière à un niveau suffisant pour la conservation des données dans le temps. Les travaux présentés dans la suite de ce document concerne donc cette problématique.

L'article suivant (**??**,[17]) est une étude quantitative du comportement, par une modélisation stochastique, des systèmes de stockage P2P en ce qui concerne la pérennité. L'originitalité de cette approche est qu'elle s'appuie, en plus de la durée de vie des pairs, sur le degré de disponibilité de ces derniers, ce qui permet de caractériser les systèmes P2P. Il en ressort que les mécanismes de redondances sophistiqués, par exemple de type Reed Solomon, sont moins adaptés que des mécanismes classiques de réplications quand la disponibilité fait défaut.

Ce dernier travail avait aussi mis le doigt sur le coût du maintien du niveau redondance, appelé coût de reconstruction. En effet, lorsque l'on vise de longues périodes de stockage, les données devront être régénérées de nombreuses fois pour faire face aux défaillances des nœuds de stockage. Cette problématique a donc été abordée dans le cadre de la thèse de Ghislain Secret. Dans ce travail un simulateur d'un système de stockage générique, tel que décrit dans le précédent travail, avait été développé. À partir de ce simulateur, nous avions étudié l'impact des différents paramètres (degré de dispersion, niveau de redondance, seuil de déclenchement) sur le coût généré par les reconstructions. Un autre résultat la définition d'une stratégie de construction proactive qui permettez de lisser le coût de reconstruction dans le temps. Le deuxième article (10,[14]) est une publication de ce travail.

Le troisième article (11,[1]) présenté par la suite est un complément au précédent. L'objectif est de déterminer une distribution des données qui minimise l'impact des reconstructions pour tous les pairs. L'idée originale étant d'utiliser la géométrie affine où les points représentent les pairs et les droites les blocs de données. Les espaces mathématiques qui peuvent être définis dans cette géométrique permettant de construire des distributions qui bornent le coût de reconstruction pour chaque pair. En particulier par l'utilisation de plan et de projection affine où les droites on qu'une seule intersection. La publication décrit justement la construction d'une telle distribution.

Data Durability in Peer to Peer Storage Systems*

Gil Utard
LaRIA
Université de Picardie Jules Verne
80000 Amiens,France
Gil.Utard@laria.u-picardie.fr

Antoine Vernois
Graal INRIA Project
LIP- École Normale Supérieure de Lyon
69364 Lyon Cedex 07, France
avernois@ens-lyon.fr

Abstract

In this paper we present a quantitative study of data survival in peer to peer storage systems. We first recall two main redundancy mechanisms: replication and erasure codes, which are used by most peer to peer storage systems like OceanStore, PAST or CFS, to guarantee data durability. Second we characterize peer to peer systems according to a volatility factor (a peer is free to leave the system at anytime) and to an availability factor (a peer is not permanently connected to the system). Third we model the behavior of a system as a Markov Chain and analyse the average life time of data (MTTF) according to the volatility and availability factors. We also present the cost of the repair process based on these redundancy schemes to recover failed peers. The conclusion of this study is that when there is no high availability of peers, simple replication scheme may be more efficient than sophisticated erasure codes.

1. Introduction

Today, Peer to Peer systems (P2P) are widely used mechanisms to share resources on Internet. Very popular systems was designed to share CPU (Seti@home, XtremWeb, Entropia) or to publish files (Napster, Gnutella, Kazaa, ...). In the same time, some systems was designed to share disk space (OceanStore, Intermemory, PAST, Farsite). The primary goal of such systems is to provide a transparent distributed storage service. These systems share common issues with CPU or files sharing systems: resource discovery, localisation mechanisms, dynamic point to point network infrastructure... But for sharing disk systems data lifetime is the primary concern. P2P CPU or file publishing systems can deal with node failures: the computation can be restarted anywhere or the published files resubmitted to the system. For disk sharing systems, node failure is a critical event: the stored data are definitively lost. So data redundancy and data recovery mechanisms are crucial for such systems.

In this paper we present a quantitative study of the usual data redundancy and repair schemes found in existing systems: replication and erasure resilient codes. After a presentation of some P2P systems and redundancy mechanisms, we introduce a characterisation of P2P storage system according to the volatility and availability of peers. We propose a stochastic model of such systems using Markov Theory which allows the study the data lifetime. We also consider the cost to recover lost data.

2. Peer to Peer system

Among peer to peer data systems, we distinguish two categories: P2P systems devoted to document publishing, and P2P systems devoted to storage and which integrate data survival mechanisms.

P2P systems are mainly characterised by an "over-network" or virtual backbone based on point to point connections leading to a loosely coupled graph. The aim of the over-network is mainly to furnish routing and localisation mechanisms.

Since the precursor Napster, lot of P2P file publishing systems have appeared. Gnutella [11] is one of them and is defined as a protocol specification to share documents. File localisation is done by a breath-search in the connection graph. Freenet [4, 3] is also a file publishing project where one of the primary goals is to insure anonymity of users (data producer or data consumer). It integrates cryptography of documents, auto adaptable routing, and a primitive replication mechanism to insure data survival of popular files.

Concerning peer to peer storage systems, one precursor is InterMemory [2, 6]. It is characterised by a complex redundancy scheme for data survival. PAST [5] is a joint project of the Rice University and Microsoft. In

*This Project (http://www.ustorage.net) is supported by the CNRS-INRIA-MENRT ACI GRID CGP2P grant.

PAST, nodes and files have a unique identifier in a common name space. Files are stored on node which have the closest identifier to the file identifier. Tolerant routing mechanism is PASTRY. Data survival is done by file replications. OceanStore [8, 1, 12] is a large project where data are stored in a set of collaborative server (long time survival, high speed connection) which are untrusted.

In this paper, we focus on this second class of peer to peer systems where data are disseminated on peers, and we study the efficiency of mechanisms used to insure data durability.

3. Redundancy mechanism

In distributed storage systems, redundancy is the base mechanism to protect data from node vanishing. The first redundancy scheme is replication: several copies of data are disseminated on different nodes. Replication is the mechanism used by PAST or Freenet. Let r be the replication factor: for $r = 1$ there is only one replica and the system tolerates only one node failure per data, if $r = 10$, the system tolerates 10 node failures per data.

A more sophisticated redundancy scheme is erasure code. In such a system a block of data of size B is fragmented into s smaller blocks of size $\frac{B}{s}$. In addition, r fragments of same size are generated and contain redundancy information about the initial data. These fragments are disseminated over the peer to peer network. Redundancy is such that any fragment can be recover from any combination of s fragments from the $s+r$ fragments. So this coding allows a tolerance factor equal to r. Replication is a particular case of erasure code where s is equal to 1. Erasure codes are used by the OceanStore project to maintain data-survival.

Let (s, r) be a redundancy scheme where block of data are divided into s fragments and where there are r fragments of redundancy. Ratio $\frac{s}{s+r}$ represents the useful space, i.e. ratio between size of the initial data and space used to store the data. For a fixed useful space, erasure code with a fragmentation factor s greater than 1 is *a priori* more efficient than simple replication because it allows a greater tolerance factor r. For instance, compare the $(1, 3)$ redundancy scheme with the $(4, 12)$.

An usual method to build erasure codes is the Reed Solomon encoding scheme [10]. Let B the initial data block to be encoded, B is considered as a vector $B = (b_j)_{j\in[1..s]}$ of dimension s where each b_j is a fragment of the initial data. The erasure code is the vector $E = (e_i)_{i\in[1..(s+r)]}$ of dimension $s+r$. The vector E is derived from B by the way of a matrix A of dimension $s \times (s+r)$: $E = AB$. Let A be decomposed by rows: $A = (A_i)_{i\in[1..(s+r)]}$. The matrix A is such that for each subset of s rows $(A_{i_j})_{j\in[1..s]}$, the rows are linearly independents. Thus, for any subset of s fragments $E' = (e_{i_j})_{j\in[1..s]}$ the initial data $B = (b_j)_{j\in[1..s]}$ is reconstructed in this following way: let $A' = (A_{i_j})_{j\in[1..s]}$ be the matrix build with the rows corresponding to the subset of fragments of E, the initial vector B is then equal to $A'^{-1}E'$.

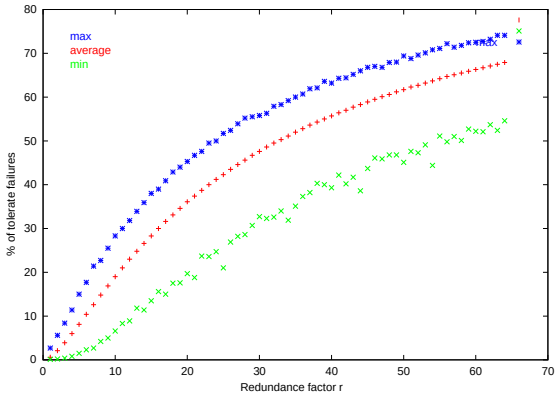


Figure 1. Percentage of peer failures before the loose of the first block for erasure code according to the redundancy factor r. The number s of initial fragments of the data block is equal to 16. There are 250000 blocks stored on 10000 peers.

Figure 1 shows the percentage of peers which can fail without loose of data according to the redundancy factor r. This result was obtained by the simulation of the distribution of 250000 blocks on 10000 peers where each block is divided in $s = 16$ fragments. For instance, with $r = 30$, we are able to tolerate up to 50% peer failures.

Whereas a big redundancy factor allows us to face to a big number of peer failures, this static mechanism does not provide good data survival. Consider for instance that peer lifetime follows an exponential deviate of parameter λ, more than half nodes probably fail after the average life time of one node.

To guarantee data survival, it is necessary to introduce a self repair mechanism which detects and rebuilds lost data. A repair mechanism is divided into two parts: The first part is the detection of node failure and the second is the reconstruction of the nodes' data. The reconstruction uses the redundancy mechanism previously described: dispersed redundancy fragments are collected to recover lost fragments on other peer(s).

4. A stochastic model for P2P storage systems

In this section, we present a stochastic model of the behaviour of peer to peer storage systems. The system is com-

posed of independent nodes which share their disk space where the fragments are stored. Nodes are free to leave the system at anytime. We assume that population in the system is stable: the rate of nodes leaving the system is equal to the rate of nodes entering the system.

4.1. Stochastic model for a peer

We assume also a homogeneous behaviour of peers. This behaviour is described by the stochastic automate depicted on Figure 2. In this automate, there are three states for

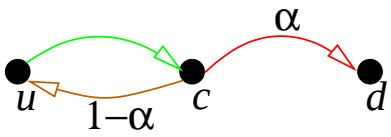


Figure 2. Node behaviour.

a peer node: connected (state c), temporarily unavailable or disconnected (state u), permanently unavailable or dead (state d). A node is not permanently connected to the system and may be disconnected for some time. The duration of a connection (resp. a disconnection) is given by a continuous random variable which follows an exponential deviate with an average lifetime equal to λ. (resp. μ). The average duration of a cycle connected/disconnected is given by $\lambda + \mu$. For the sake of presentation we set the unit of time equal to the cycle duration, i.e. $\lambda + \mu = 1$. The parameter λ is the *availability* factor of a node during its lifetime in the system. After a connection, a node can leave the system with a probability equal to α or stay in the system with a probability equal to $1 - \alpha$. The parameter α is the *volatility* factor of a node in the system.

The average lifetime denoted by τ of a node in the system, i.e. when it is in state c or u, is the average time of the first connection plus the product of average number of cycles connection/disconnection by the average duration of a cycle. So

$$\tau = \lambda + \sum_{k=0}^{k=\infty} (1-\alpha)^k \alpha(\lambda+\mu) = \lambda + \frac{1-\alpha}{\alpha}$$

(we fixed $\lambda + \mu = 1$). Since duration of states c and u follows exponential deviates and number of cycles follows a geometrical distribution, the lifetime of a node in the system is memoryless and follows a exponential deviate with an average duration equal to τ.

The parameters λ (availability) and α (volatility) allow us to characterize peer to peer systems (see Figure 3). For instance when λ is close to 1 and α is close to 0 we can consider the system to be a peer to peer server network like

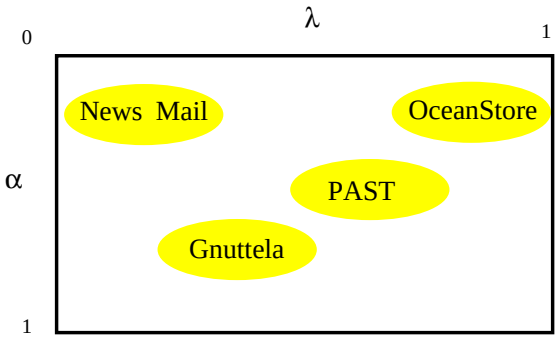


Figure 3. Characterisation of P2P systems according to their availability and their volatility.

OceanStore. When λ is smaller and α is close to 0 we can consider to be in presence of a system like a message service, i.e. peers are faithful but make short connection to the system. The worst case is when α is close to 1 and λ is small.

4.2. Block automate

For a given block of data, we describe its availability by the automate shows on Figure 4. The block is coded by $s+r$ fragments which are stored on $s+r$ distinct peers. We consider only these peers. A block is available when more than s peers are connected. A block of data is dead when more than r peers failed. A state is the couple (number of connected peer nodes, number of failed peer nodes). Arrows

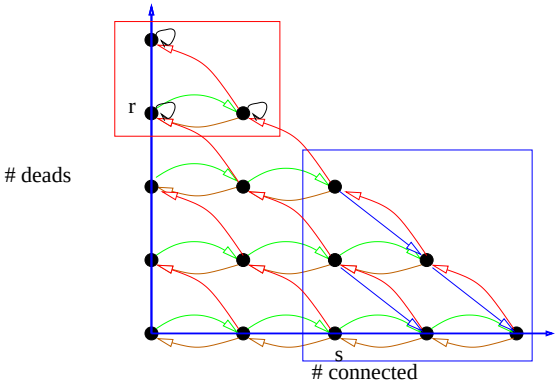


Figure 4. State transition for the availability of a block of data with a redundancy scheme $(s, r) = (2, 2)$.

correspond to the different transitions between states: connection of a peer (right arrows), disconnection of a peer

(left arrows) and failure of a peer (up arrows). States in the right box correspond to states where the data block is available (there are enough connected peers to rebuild the initial data), states in the upper box correspond to states where the data block is lost (more than r fragments are definitively lost and thus the initial data cannot be rebuild).

The repair process is represented by down arrows: the repair is done by a new peer which grabs s fragments of the block from connected peers and rebuild the lost fragment using the erasure code. So a repair can be done only when at least s fragments, i.e. peers, are connected. In our model, we assume that if a node fail, then there is enough space in the system to store the lost data, i.e. there is a peer not yet involved in the storage of the considered block which stores the recovered fragment.

4.3. Stochastic behavior of a block

If we considerer probabilities of state transitions, this automate describes a non recurrent Markov Chain with one absorbing state [7]. Let describe more formally the automate previously introduced. Let a_i^j be the state where i nodes are dead ($0 \leq i \leq s+r$) and j nodes are connected ($0 \leq j \leq s+r-i$). Probability for each state transition is determined in the following way.

Connecting: One peer is reconnecting the system, it is the transition from state a_i^j to state a_i^{j+1} ($0 \leq j < s+r-i$). Because all peers are independents the transition probability follows an exponential deviate with an average lifetime equals to $\frac{\lambda}{(s+r-j)}$.

Disconnecting: One peer is disconnecting the system, it is the transition from state a_i^j to state a_i^{j-1} ($0 < j \leq s+r-i$). Similarly to the reconnection, the transition probability follows an exponential deviate with an average lifetime equals to $\frac{\mu}{j}$.

Failure: One peer is leaving the system, it is the transition from state a_i^j to state a_{i+1}^{j-1} ($0 \leq i < s+r$, $0 < j \leq s+r-i$). The transition probability follows an exponential deviate with an average lifetime equals to $\frac{\tau}{(s+r-i)}$.

4.4. Dynamic repair transition

We employ the Markov Embedded Chain technique in order to estimate the transition probability of a repair transition. We consider that failed peers are repair in the order of their failure. The repair transition probability from state a_i^j to state a_{i-1}^{j+1} ($0 \leq i \leq s, s \leq j \leq s+r-i$) is the probability that when the oldest fail node is repaired, there are less than i failed nodes in the system.

The main problem is to determinate time to repair a node. We fix the *effective* time to rebuild a node (a.k.o. "cpu time") equal to t_e. However, the repair process is only possible when at least s nodes remain connected during the reparation process. We consider that when less than s nodes are connected, then the repair process is suspended. It restarts when s nodes are connected. So we have to evaluate what is the *real* time t_r to repair a node (a.k.o. "elapsed time").

We approximate t_r with the average fraction of time the repair process can be run (i.e. there are at least s nodes connected). Let f_r be this fraction, f_r is estimated in the following way: let d_i be the average lifetime to be in states a_i^j for all j such that $0 \leq j \leq s+r-i$ (there are i failed nodes) before one more node fails: $d_i = \frac{\tau}{s+r-i}$. Let c_i be the probability than at least s nodes are connected when there are i nodes failed. Connecting/reconnecting is a pure birth/death process, so c_i can be obtained by a simple product formulation. The fraction f_r is then estimated by:

$$f_r = \frac{\sum_i c_i d_i}{\sum_i d_i}$$

The real time t_r of the repair process is $\frac{t_e}{f_r}$.

Let $X(t)$ be the random variable representing the number of failed node remaining when the oldest failed node is repair at time t (pure death process). We estimate the probability of a repair transition from state a_i^j to state a_{i-1}^{j+1} ($s \leq j \leq s+r-i, i < s$) by $P(X(t_r) < i)$.

5. MTTF of data blocks

In the following we present a quantitative study of the modeled P2P storage system with automatic repair process. This quantitative study allow us to estimate what is the MTTF (*mean time to failure*) of data block in such a system, i.e. the average lifetime of a data block in the system.

Thanks to our Markov model, the MTTF can be estimated classically by a discretisation of time and considering the resulting transition probability matrix P where the absorbing state is removed. Then we compute the vector $E = (I-P)^{-1} \times U$ where I is the identity matrix and U the unit vector. The elements E_i gives the average lifetime starting from state i.

Figure 5 shows the estimated MTTF according to the availability rate of peers (λ), thus for three redundancy schemes with the same useful space ((7,21), (4,12) and (1,3)[1]) and for two volatility parameters: $\alpha = 0.1$ and $\alpha = 0.3$ (for high volatility factors, the MTTF is too low). The time fixed for the repair process is $t_e = 1$, i.e. the time for one period of connection/disconnection (we observed a

[1]The redundancy scheme (1,3) corresponds to the replication scheme.

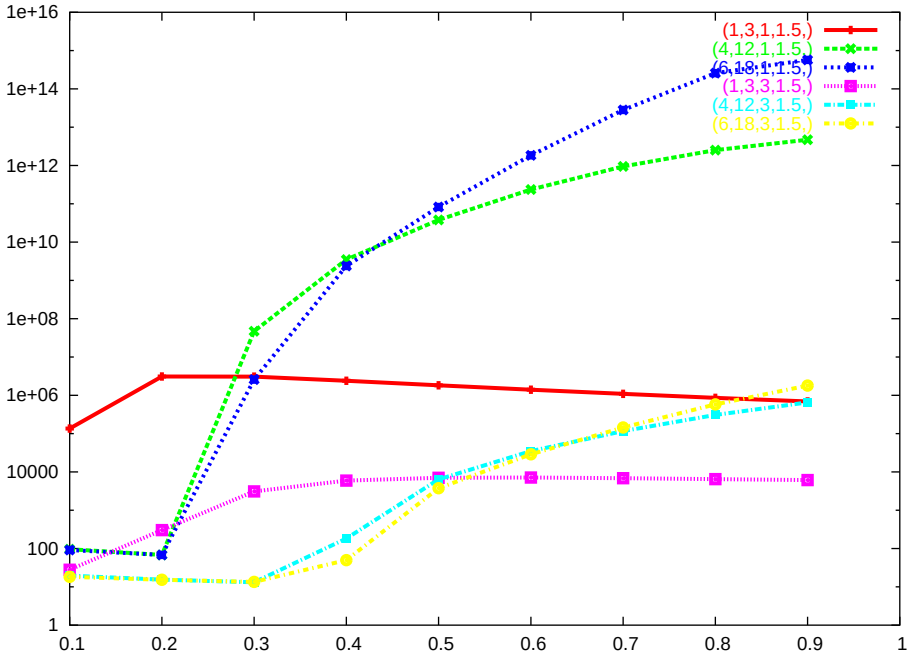


Figure 5. MTTF according to the availability of peers (λ).

similar behaviour for other repair process time). For a system with high availability (λ close to 1), MTTF is better with s big as expected. But for a system with low availability, we observe a better MTTF for the replication scheme.

This suggests that for a peer to peer system with lower availability of peers, the replication scheme is better than erasure code. Figure 6 shows the MTTF according to the fragmentation parameter s for a fixed useful space ratio (25%), and for different volatility and availability parameters.

This phenomena may be explain by considering the frequency of each state. Figure 7 shows the relative frequency of each state for a (4,12) erasure code which is obtain using Markov Theory. Darker state are the more frequent state during the life of a block. These frequency was plot for $\alpha = 0.3$ and different availability parameters (0.4, 0.5 and 0.7). The vertical line is the limit number of connected peers which are necessary to rebuild a failed peer. When the availability is reduced, we see that more frequent states are close to this limit. This means that the state of the block is more often in a state where the repair process is suspended, so the real time to repair is greater, probability of not recovering the block of data (which determines MTTF) increases.

6. Data recover costs

In the previous part, we estimated the MTTF of a peer to peer system characterized by (α,λ) based on a redundancy scheme (s, r) which integrates a failure repair mechanism. In this part, we will discuss about costs generated by this repair mechanism.

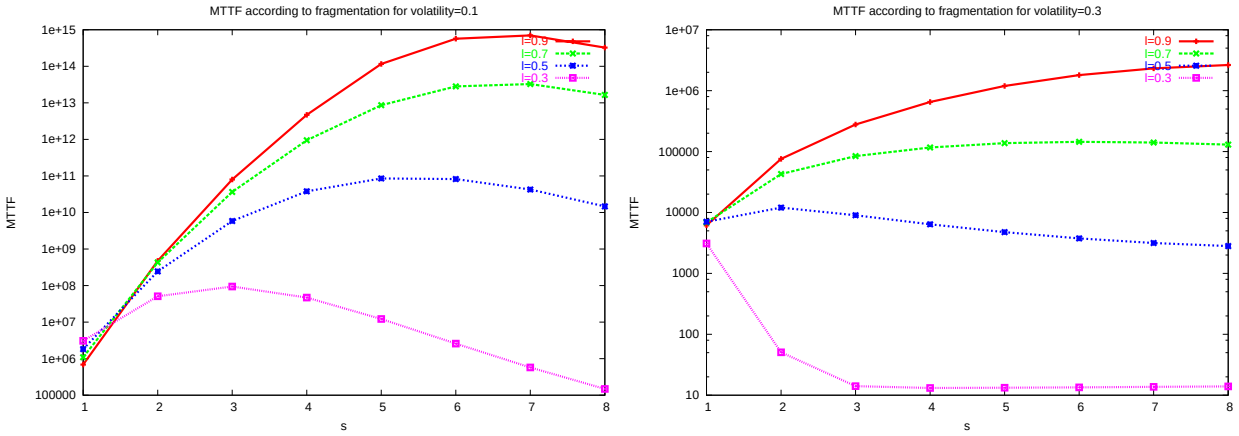
One of our hypothesis is that the repair mechanism is able to rebuild data of a failed node in a fixed time t_e. To rebuild one fragment in a (s, r) redundancy erasure code, s fragments must be retrieved from alive nodes. Lets say that each node hosts 10000 fragments of 4KB (i.e. 40MB per peer), and let s be equal to 8, then $10000 * 4 * 8 =$ 320MB have to transit on the network to repair one failed peer. The peer to peer system must deal with simultaneous failed peers, so the repair process is limited by the capacity of the peer to peer network.

Let $T_x = \frac{1}{\tau}$ be the average rate of node failure where τ is the average lifetime of a node. In our model

$$T_x = \frac{1}{\lambda + \frac{1-\alpha}{\alpha}}$$

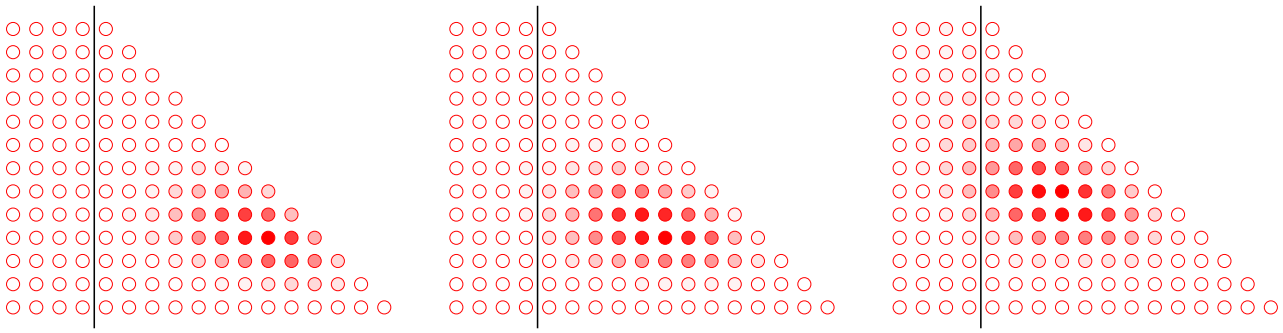
T_x increases with α, and then the number of simultaneous failed peers to repair increases in the same order.

Figure 8 is the average peer lifetime according to the volatility α. We can see these plots as the average number of time period ($\lambda + \mu$) a node stays in the system before leaving it. This means that $\frac{1}{Tx}$ peer must be repair per cycle on average. The amount of data to communicate in the network is proportional to $\frac{1}{Tx}$. Using an erasure code with a fragmentation factor equals to s implies that the amount of data communicated in the network is multiplied by s in a



(a) $\alpha = 0.1$ (b) $\alpha = 0.3$

Figure 6. MTTF according to the fragmentation factor s for different λ (l) and α.



(a) $\lambda = 0.7$ (b) $\lambda = 0.5$ (c) $\lambda = 0.4$

Figure 7. State frequency according of a $(4, 12)$ redundancy scheme for different availability factors λ ($\alpha = 0.3$).

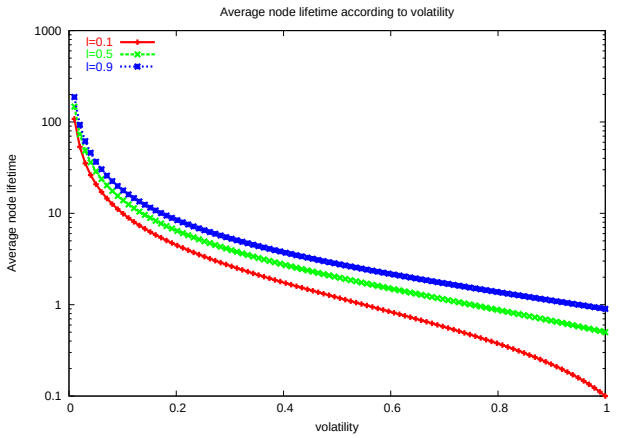


Figure 8. Average lifetime of node according to α for different λ.

cycle. For instance is $Tx = 8$ and $s = 8$, then the amount of data communicated in the network in a cycle is equal to the volume of data stored!

6.1. Conclusion

In this paper, we presented a quantitative study of data survival in Peer to Peer Storage Systems. We characterized Peer to Peer systems according to a volatility factor (peer are free to leave the system at anytime) and to an availability factor (peer are not permanently connected to the system).

We compared the two main redundancy schemes, replication where data are duplicated on different peers, and erasure codes where data are first divided into s fragments and where r fragments of redundancy are added to tolerate failure. We estimated the data lifetime of these redundancy scheme by a Markov Chain model. Whereas the erasure coding scheme is *a priori* more efficient than simple replication, a result of this study is that simple replication is better when there is no good availability of peers: erasure codes require high availability of peers to be efficient.

For repair cost point of view, we also shown that erasure codes increase the amount of communication to recover failed peer: s times the amount of data stored in the failed peer must be accessed to rebuild the lost data. A large part of the network capacity must be used to maintain data integrity.

In fact, contrarily to conventional wisdom, erasure codes like Reed Solomon are not the *cure-all* to insure long data lifetime in peer to peer storage systems: it is efficient only for peer to peer system with highly available peers like OceanStore, where peers are servers hosted by some Internet Providers. Our future work is to design and analyse other redundancy scheme mechanisms which may be better than simple replication for data survival and erasure code for the cost of the repair mechanism. We can imagine new redundancy scheme which mixes replication and erasure code to combine the low cost of replication and to increase the efficiency of erasure code because of a better availability of fragments. We plan also to study the Tornado [9] method which employs a sparse hierarchical redundancy scheme which may reduce the needs for highly available peers.

References

[1] David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Christopher Wells, Ben Zhao, and John Kubiatowicz. Oceanstore: An extremely wide-area storage system. Technical Report Technical Report UCB CSD-00-1102, U.C. Berkeley, May 1999.

[2] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, 1999.

[3] Ian Clarke. A distributed decentralised information storage and retrieval system, 1999.

[4] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.

[5] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Procedings of HOTOS*, pages 75–80, 2001.

[6] A. Goldberg and Peter N. Yianilos. Towards an archival intermemory. In *Proceedings of IEEE Advances in Digital Libraries, ADL 98*, pages 147–156, Santa Barbara, CA, 1998. IEEE Computer Society.

[7] Leonard Kleinrock. *Queueing system*, volume Volume I: theory. Wiley-Interscience Publication, 1975.

[8] John Kubiatowicz, Divid Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[9] M. Luby, M. Mitzenmacher, A. Shokrollay, and D. Spielman. Efficient Erasure Correction Codes. *IEEE Trans. on Information Theory*, 47(2), February 2001.

[10] James S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software Practice and Experience*, 27(9):995–1012, 1997.

[11] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings of International Conference on Peer-to-peer Computing*, August 2001.

[12] Chris Wells. The oceanstore archive: Goals, structures, and self-repair. Master's thesis, University of California, Berkeley, May 2001.

A Study of Reconstruction Process Load in P2P Storage Systems*

Ghislain Secret[1] and Gil Utard[2]

[1] LaRIA-MIS, Université de Picardie Jules Verne,
Amiens, France
[2] UbiStorage, Pôle Jules Verne,
Amiens, France

Abstract. In this paper we present a study of the load generated by the reconstruction process of P2P storage system. This reconstruction process maintains redundancy of data for durability to face peer failures found in P2P architectures. We will show that the cost induced is not negligible and we will show which parameters of the underlying P2P system can reduce it. To our best knowledge it is the first study of this topic.

1 Introduction

Today, Peer to Peer systems (P2P) are widely used mechanisms to share resources on Internet. Very popular systems were designed to share CPU (`Seti@home`, XtremWeb, Entropia) or to publish files (Napster, Gnutella, Kazaa). In the same time, some systems was designed to share disk space (OceanStore [4,9], Intermemory [2], PAST [3]. The primary goal of such systems is to provide a transparent distributed storage service. These systems share common issues with CPU or files sharing systems: resource discovery, localisation mechanisms, dynamic point to point network infrastructure... But for sharing disk systems data lifetime is the primary concern. P2P CPU or file publishing systems can deal with node failures: the computation can be restarted anywhere or the published files resubmitted to the system.

For disk sharing systems, node failure is a critical event: the stored data are definitively lost. So introducing data redundancy, such as the well known Rabin dispersal technique [5], and data recovery mechanisms is crucial for such systems.

Some previous works focused on the feasibility of such system, mainly for the data durability question: is the data redundancy scheme and data reconstruction mechanism sufficient to insure no data lost? Whereas the answer is yes for some parameters([1,10]), in [8], the authors outline that reconstruction processes introduce a new load in the P2P system, mainly the communication cost to maintain redundancy.

In this paper we present a first study of this load and the impact of the P2P system parameters on cost. This study is done by simulation. To our best

* This work is supported by the ANR Spreads project (www.spreads.fr).

A. Hameurlain (Ed.): Globe 2008, LNCS 5187, pp. 12–21, 2008.

knowledge it is the first work on this topic. After a presentation of usual redundancy schemes and reconstruction mechanisms used in P2P storage systems, we present our simulation process. We present the impact of system parameters on load induced by the reconstruction mechanism. Then we conclude.

2 Redundancy Scheme and Reconstruction Process

A peer-to-peer storage system is characterised by peers volatility. Peers connect and disconnect randomly. But in storage systems, the main issue is data durability. To cope with peers volatility, data redundancy is introduced.

The most simple method is data replication on different peers. To deal with r failures, data is replicated r times. However, replication requires a space r times larger than original data size. This ratio between the size of the data and actual space used to store the data is called *usable space*. Usable space is defined as the ratio between the original data size and the storage space. e.g.for a given 3 times replicated data, fault tolerance is 3 and useful space is $\frac{1}{1+r}$, i.e. $\frac{1}{4}$.

Storage systems that use replication as redundancy mechanism suffer from a low usable space. Other redundancy techniques that maximise usable space have been developed, like IDA schemes [5]. The mechanism is to fragment a data block in s fragments. Then, from these fragments, r redundancy fragments are computed. The $s + r$ fragments of the data block are distributed on different peers. Any combination of s fragments allows to rebuilt the raw data. Therefore the system tolerates r failures.

In the case of redundancy with fragmentation, usable space is expressed by the ratio $\frac{s}{s+r}$. e.g. for a given original data cut into 5 pieces, plus 3 redundancy fragments, fault tolerance is 3. Usable space is $\frac{s}{s+r}$, in our exemple: $\frac{5}{8}$. Blocks fragmentation allows usable space gain, for equivalent fault tolerance to replication. Therefore it allows to store more information in the system.

Note that replication is a special case of this redundancy scheme, where $s = 1$ and r is the number of replicates.

2.1 Reconstruction Process

In addition to redundancy scheme, a reconstruction mechanism of lost fragments is introduced to ensure data durability. A reconstruction threshold k is defined ($k \leq r$). For each data block, when $r - k$ fragments are lost due to peer failures, a fragment regeneration process is triggered: $r - k$ alive peers will receive regenerated fragments. Note that it will be necessary to communicate the equivalent of the original data size (s fragments) through the system to conduct a reconstruction.

2.2 Reactivity Threshold

For a given block, k is the minimum redundancy fragments remaining in system before block reconstruction process is triggered. When redundancy fragments of a block is less than or equal than k, block rebuild process is launched to recover

it. So, if $k = r - 1$, block reconstruction is initiated from the first fragment lost. We call this strategy *"anxious"*. At the opposite, if $k = 1$, block reconstruction is delayed until $r - 1$ fragments are lost, i.e. only one redundancy fragment remains in the network. This reconstruction strategy is called *"zen"*. Therefore k determines reactivity of the reconstruction process in the system.

These mechanisms allow data durability in network by judiciously setting system parameters. We refer to studies [6,8,7] to maintain data durability in P2P storage systems.

However, impact of system parameters on network load is not neutral. For each reconstruction, the block is communicated through the network. Hence, too often regenerate the redundancy of the system leads to a heavy load on the network. In this document, induced traffic (network load) in a P2P system is studied. It is the traffic generated by the permanent reconstructions, necessary to maintain data durability in system.

3 Peer to Peer Storage System Simulation

We consider a peer to peer storage system consisting of N independent peers who share their storage space. Peers are free to leave the system at any time. Peers who disappear are considered dead. It is assumed that the average population is constant, thus the average rate number of peers leaving the system is equal to the average rate number of peers who join it.

For the sake of simplicity, and without loss of generality, we consider that the data are blocks of uniform size.

A peer to peer storage system is determined by the parameters s, r and k, previously defined. We assume that the detection delay of a failed peer is constant. The regenerated fragments are randomly redistributed on living peers (which do not already have a fragment of the same block).

Peers lifetime is governed by a deviate law. Failure rate is determined by the number of peers in the network and their lifetime. For instance, in a 1,000 peers network, where the mean lifetime of each peer is 12 months, the average time between failures is 8 hours and 45 minutes. In this example there is about 3 failed peer per day.

Our study focuses on the data recovery costs in a peer to peer storage system, with the features set out above and without any data loss. We have developed a simulator to study the behaviour of the peer to peer storage system. The simulator evaluates the amount of data communicated between peers induced by the reconstruction process. The network load at time t is represented by the number of data blocks in a reconstruction state at time t.

Figure 1 shows an example of over time traffic load, in a 100 peers network, each storing 5 GB of data, and with a lifespan of 12 months. The block size is 10 MB. Each block is divided into 8 equally sized fragments, to which, using erasure codes, 8 redundancy fragments are added: $(s, r) = (8, 8)$. We assume that each peer is connected to the network through broadband connection (1 Mb download/256 Kb upload). The detection delay of peer failure is 1 day. The reconstruction threshold k is 5 remaining redundancy fragments.

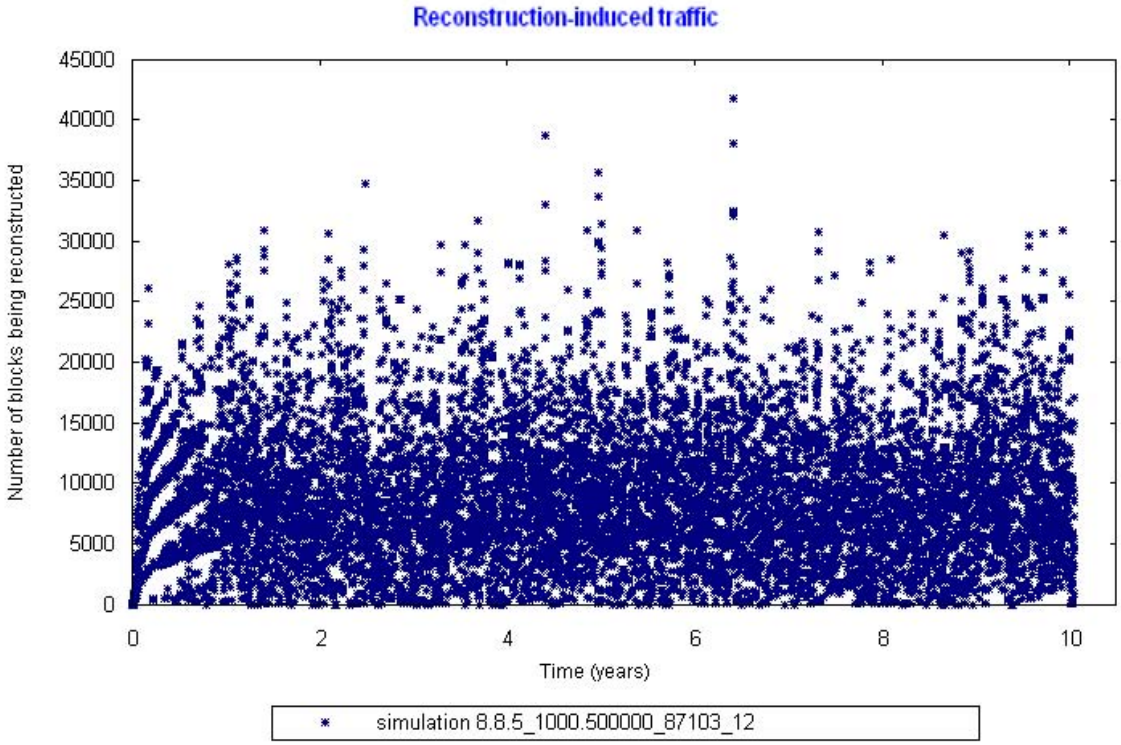


Fig. 1. Traffic load example

Figure 1 shows that reconstructions-induced traffic is not negligible. The average number of blocks being rebuilt is measured at each peer failure. For the parameters set in the simulation above, average blocks being rebuilt is 9216 blocks of a total of 500,000. The important standard deviation (5704) shows us significant variations in traffic load.

In this example, each peer hosts an average of 8,000 fragments, from various blocks. On average, 3 failed peers per day. This means that 24,000 fragments on average are lost every day. Among them, about 9000 blocks are critical (compared to k) and require a reconstruction.

The data volume exchanged between peers to rebuild the blocks is over than 100 GB a day, or more than 100 MB per peer and per day. Consequently, on average, each peer spends more than one hour per day for the reconstruction of the data. The volume of data exchanged to maintain the redundancy of data is important. It may reduce the bandwidth used by peers for data storage.

In the following, we will explore the factors which can reduce this traffic.

4 Parameters Effects

In this part, we will study influence of various system parameters on the reconstructions-induced load.

For all simulations, the following parameters are fixed. Each peer stores 5 GB of data. Data is divided into equally sized blocks. Each peer bandwidth is 256 Kb/s up and 1 Mb/s down. The simulation covers 10 years. The peer average lifetime is 12 months. The detection delay of a peer failure is one day.

The varying parameters are the number of peers in the system; the block size; the blocks fragmentation s; the number of redundancy fragments r; the reconstruction threshold k.

4.1 Size of the System, Blocks Dilution

To illustrate influence of number of peers in the network on traffic load, two configurations were compared. Only the number of peers (1,000 and 5,000 peers) differs in the two simulations exposed. The total number of blocks in the system is 50,000 in the first case and 250,000 in the second case.

In Figure 2, results show that network load is not proportional to number of peers. For instance in Figure 2, there is a factor 3 on the network load for a factor 5 on the number of peers.

Consider a block. If we increase the number of peers, the probability that a fragment of this block is located on a faulty peer is lower if there is more peers in network. This probability decreases faster than the increase of the number of blocks. Since each block is always connected to the same number of peers ($s + r$ peers), the increase in the number of peers is tantamount to diluting the blocks in the system.

Therefore, erosion data blocks will be slower when the number of peers in the system is greater. The need to rebuild appear later and, as a result, the network load on the reconstructions is reduced. Consequently large networks have a better performance in terms of network load on the storage volume available to the users.

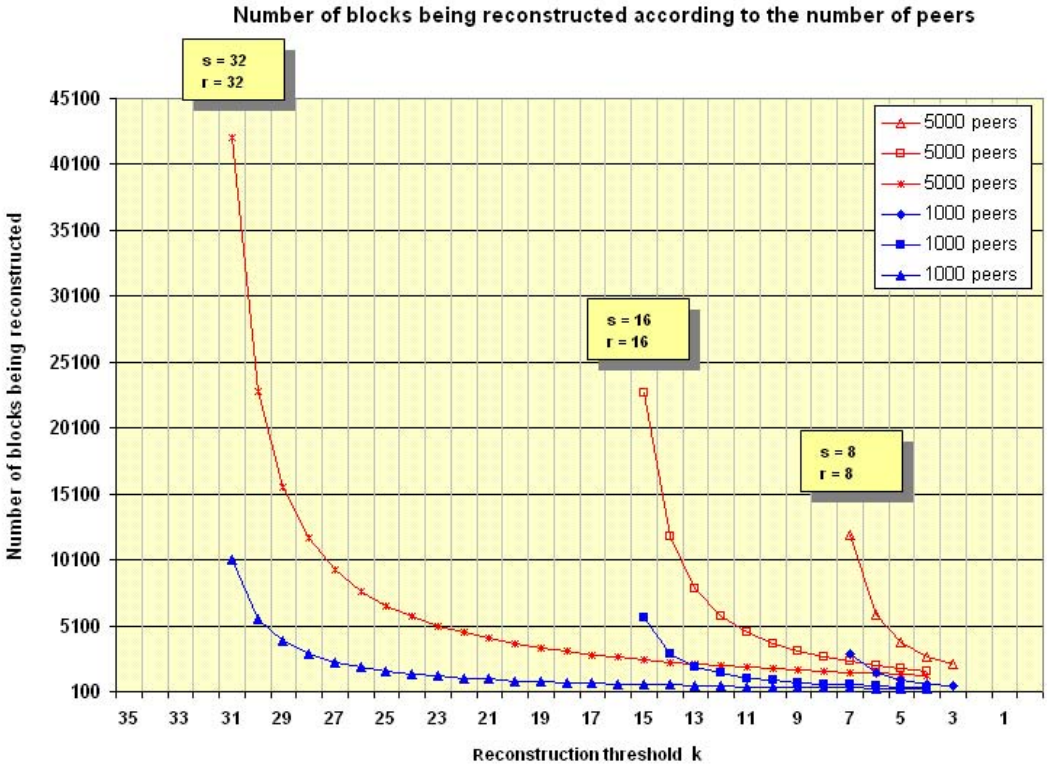


Fig. 2. Number of peers

4.2 System Reactivity

According to threshold k, network load evolution for different simulations is summarised in the Figure 3:

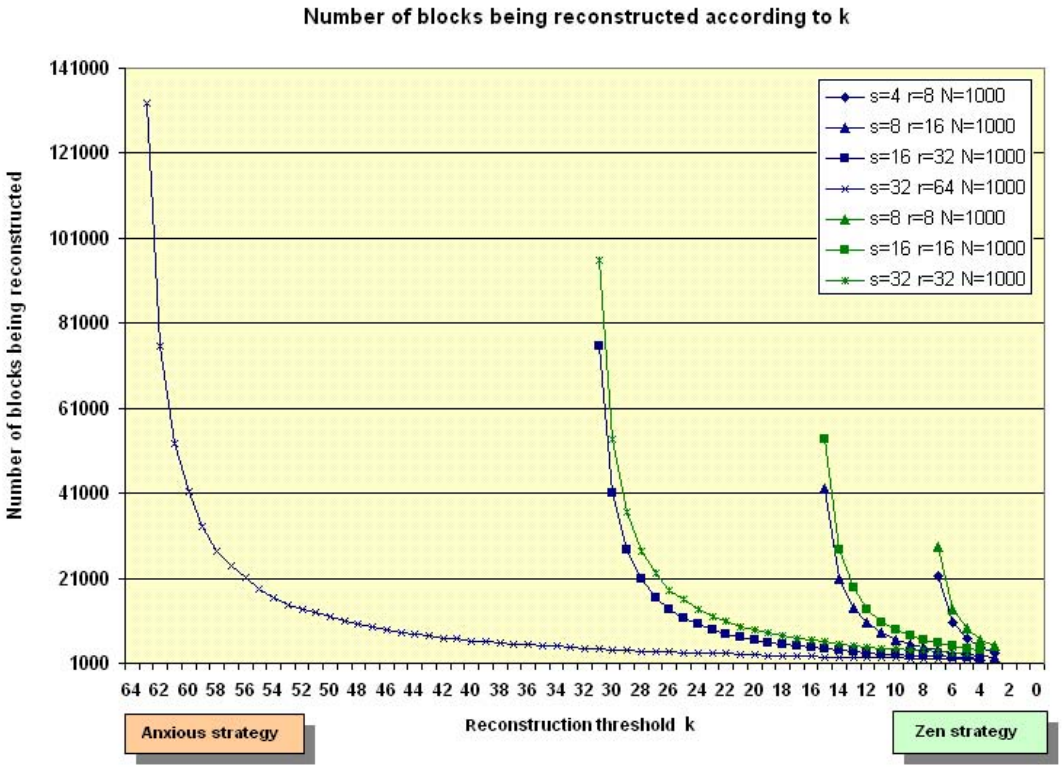


Fig. 3. Reactivity

Figure 3 shows that the gain in network load drop when reactivity k decreases by some units from the anxious strategy.

Therefore it is better to be not too anxious to reduce the network load of the system. Indeed, the gain on network load is on the first units of the reduction in the system reactivity. In fact, it is quite pointless to try to be as zen as possible. The gain on network load will be negligible. In addition, a too zen strategy might not allow the system to maintain data durability.

4.3 Data Dispersion

Data dispersion is defined by the ratio between number of blocks in network and number of peers. Note, with data volume per peer maintained constant, to increase the number of blocks in the system is to decrease the size of the blocks. In this case, peers host fragments of a larger number of blocks.

Two configurations were analysed, both in a 1,000 peers network: 50,000 blocks of size 100 MB each and 500,000 blocks of size 10 MB.

Increasing the block size from 10 to 100 MB leads to reduce the network load significantly (see Figure 4: an average of 8.5 times in this example), whatever the strategy implemented (regardless of the system reactivity k).

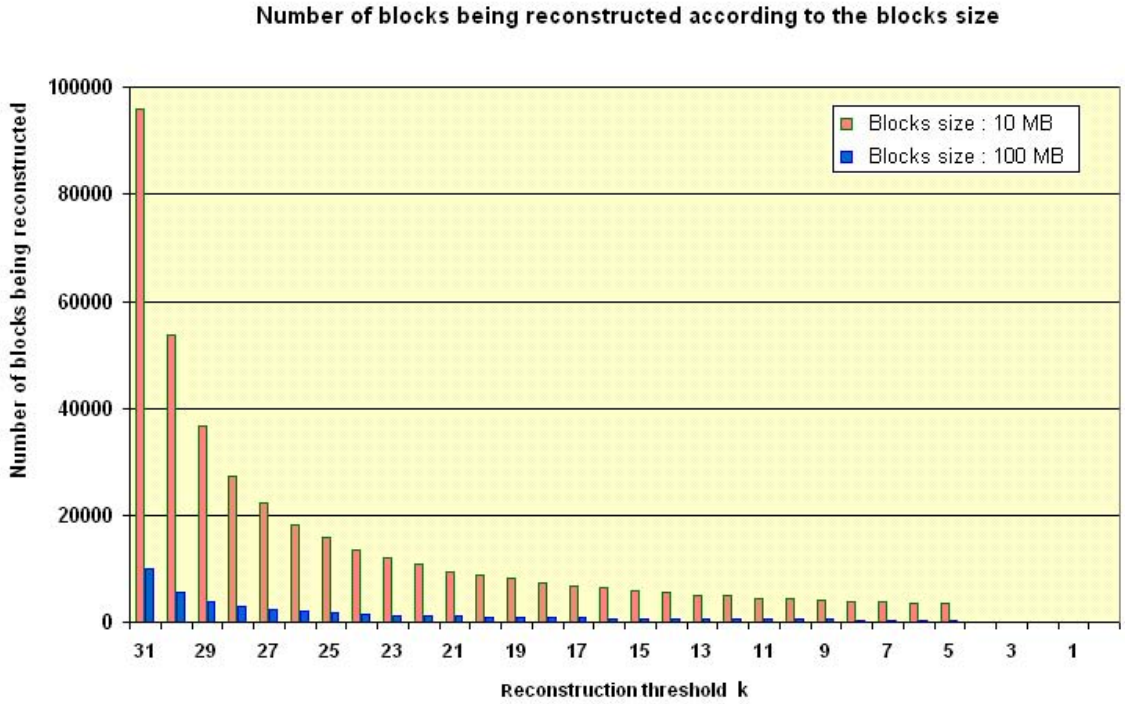


Fig. 4. Blocks size effects

In fact, in the second case (500,000 blocks of size 10 MB) each peer hosts an average of 10 times more fragments than in the first case. Therefore, when a peer disappears, 10 times as many blocks are affected. But, blocks fragmentation s and number of redundancy fragments r are the same in both cases. Probability that a given block is hit is greater, and as a result, it is necessary to rebuild more often.

When data volume is constant, increasing block size reduces the number of reconstructions. A larger block size allows to concentrate blocks on fewer peers.

However, when the block size increases, the fragments are also bigger. In our case, data volume exchanged in the reconstruction of each block is 10 times greater. During reconstruction, time required to transfer the fragments will be more important. However, this increase in time of reconstruction is negligible compared to time detection (of the order of one day).

4.4 Usable Space

Usable space is defined by the relationship between s and $s + r$. We will observe behaviour of the network load when parameters s and r vary. Two sets of simulations are conducted. In the first s is kept constant and the network load is measured with a value of r, then the double of that value. In the second, r is kept constant and the network load is measured with a value of s, then the double of that value.

When r is doubled, we see in the Figure 5 that the number of blocks being rebuilt is lower, regardless of the value of k.

For example, for $k = 20$, with $s = 32$ and $r = 32$, the average number of blocks being rebuilt is close to 9000. For the same value of $k = 20$ but with

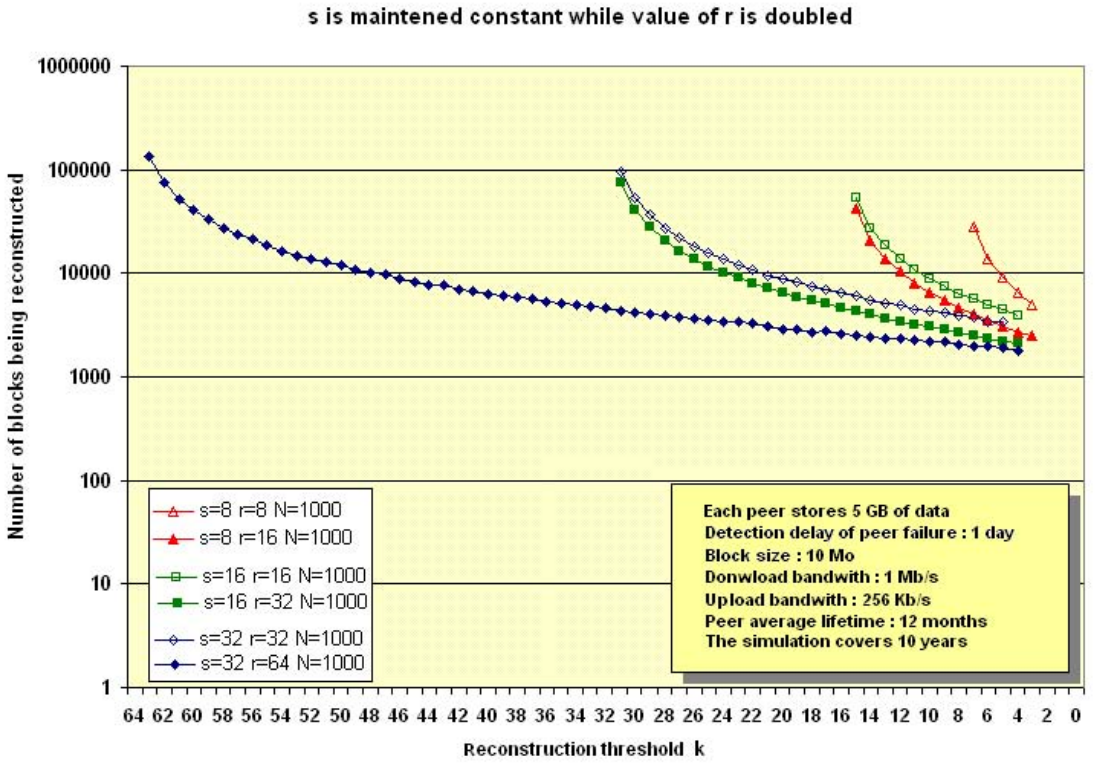


Fig. 5. Redundancy factor effects

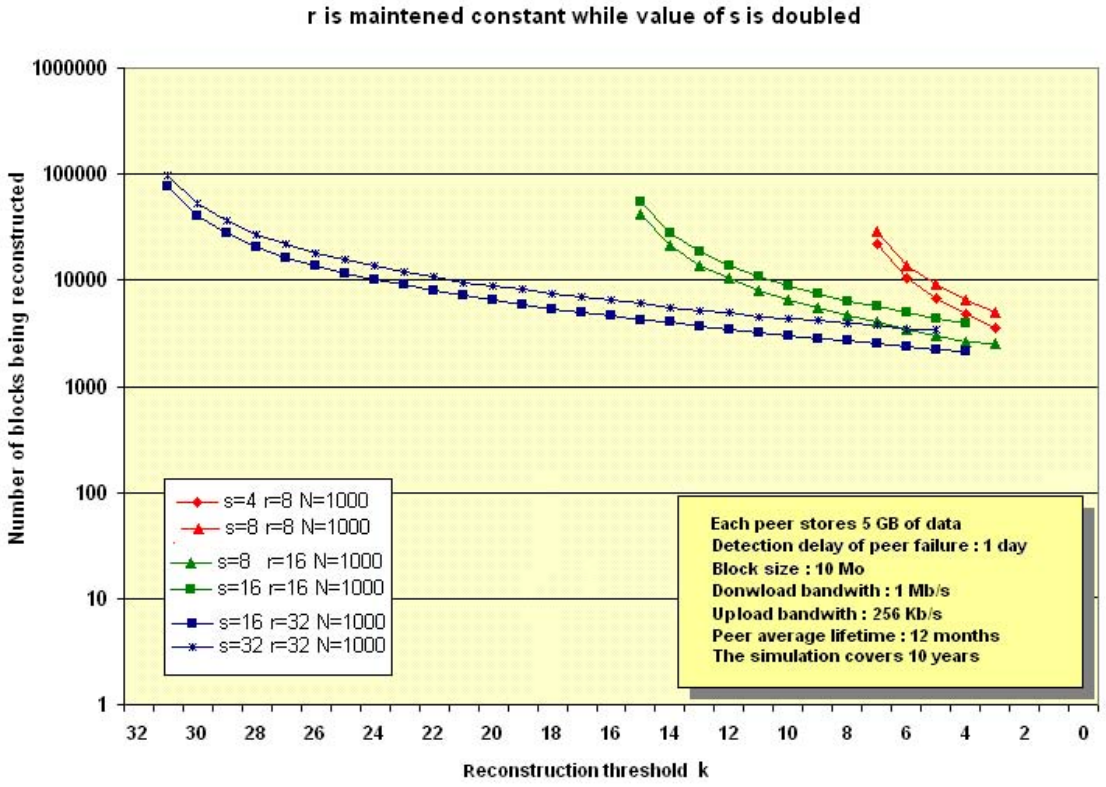


Fig. 6. Blocks segmentation effects

$s = 32$ and $r = 64$, the average number of blocks being rebuilt is close to 3000, i.e. three time less.

A good strategy would be to choose r great. However, a high value of r leads to a loss of usable space. For instance, usable space is 50% for $k = 20$, $s = 32$ and $r = 32$. For $k = 20$, $s = 32$ and $r = 64$ it drops to 33%. We operate less efficiently user storage space.

To offset the loss of usable space, blocks fragmentation can be increased, i.e. $s = 64$, $r = 64$, where usable space is 50%.

Unfortunately, the Figure 6 shows that the network load increases when s increases. Indeed, increasing blocks fragmentation in network is to expose more the blocks. The likelihood of losing a fragment is greater. That increase, therefore, leads to an increase in the number of reconstruction necessary to maintain data durability in the system.

Note that this increase in the network load is not proportional to the increase of s. For instance, it varies from +53% for $k = 5$ to +27% for $k = 31$.

Note also that the value k we choose to compare the network load has a different meaning depending on the number of redundancy fragments r. For instance, if the value of k is 15 and the value of r is 16, then we are in anxious strategy, whereas if the value of r is =32 we are much more zen.

Let δ be *anxiety level*, $\delta = \frac{k}{r-1}$. We can compare the number of blocks being rebuilt at a constant level of anxiety and constant usable space by setting $s = r$.

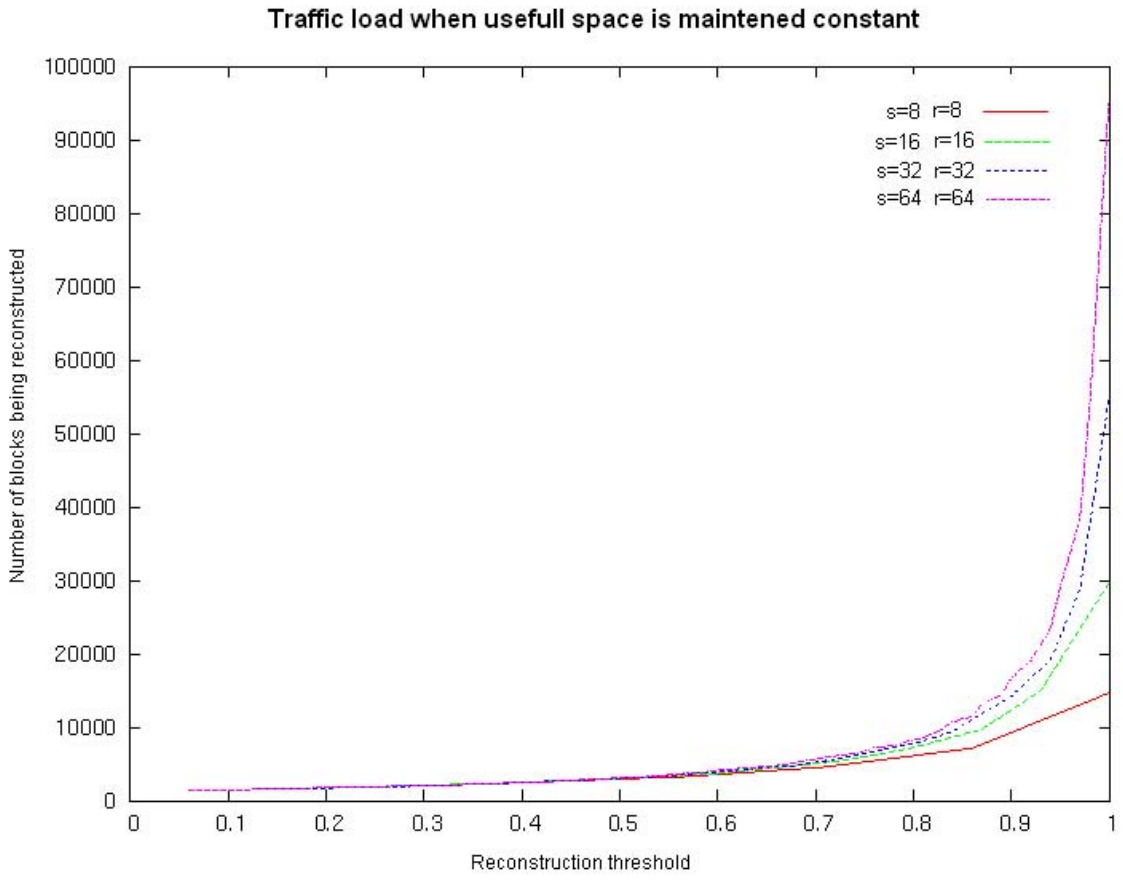


Fig. 7. s and r variations

Figure 7 shows the variation in the number of blocks being rebuilt under these conditions.

5 Conclusion

In this paper, we presented a study of the reconstruction process load in P2P storage system. Study was done by simulation of a generic P2P storage system. We shown this load is not negligible, so it is important to detect which parameters can reduce it. We shown that this load is proportional to the system size. We shown also that a too reactive system (say "anxious") generates huge load which can be significantly reduce by slightly diminishing reactivity of the reconstruction process, thus without compromising data durability of the system. We observed also that reducing spreading of data in the system reduce linearly the number of reconstruction but not the volume of data exchanged.

In this paper, we studied load for *reactive* system, that is regeneration of redundancy is done with a fixed threshold. As a future work, we plan to investigate the load in *pro-active* systems, when regeneration of redundancy is done preventively (using for instance information on age of peer).

References

1. Alouf, S., Dandoush, A., Nain, P.: Performance analysys of P2P storage systems. In: Mason, L.G., Drwiega, T., Yan, J. (eds.) ITC 2007. LNCS, vol. 4516. Springer, Heidelberg (2007)
2. Chen, Y., Edler, J., Goldberg, A., Gottlieb, A., Sobti, S., Yianilos, P.: A prototype implementation of archival intermemory. In: Proceedings of the Fourth ACM International Conference on Digital Libraries (1999)
3. Druschel, P., Rowstron, A.: PAST: A large-scale, persistent peer-to-peer storage utility. In: Procedings of HOTOS, pp. 75–80 (2001)
4. Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.: Oceanstore: An architecture for global-scale persistent storage. In: Proceedings of ACM ASPLOS. ACM, New York (2000)
5. Rabin, M.O.: Efficient dispersal of information for security, load balancing, and fault tolerance. J. ACM 36(2), 335–348 (1989)
6. Sit, E., Haeberlen, A., Dabek, F., Chun, B., Weatherspoon, H., Morris, R., Kaashoek, M., Kubiatowicz, J.: Proactive replication for data durability. In: 5th International Workshop on Peer-to-Peer Systems (IPTPS 2006). (2006)
7. Utard, G.: Perennite dans les systemes de stockage pair a pair. In: Ecole GRID 2002 (2002)
8. Utard, G., Vernois, A.: Data Durability in Peer-to-Peer Storage Systems. In: Proc. 4th Workshop on Global and Peer to Peer Computing. IEEE/ACM CCGrid Conference, Chicago (April 2004)
9. Wells, C.: The oceanstore archive: Goals, structures, and self-repair. Master's thesis. University of California, Berkeley (May 2001)
10. Williams, C., Huibonhoa, P., Holliday, J., Hospodor, A., Scwarz, T.: Redundancy management for p2p storage. In: Seventh IEEE International Symposium on Cluster Computing and The Grid (CCGrid 2007). IEEE, Los Alamitos (2007)

Data distribution for failure correlation management in a Peer to Peer storage system*

Cyril Randriamaro, Olivier Soyez, Gil Utard, Francis Wlazinski
LaRIA, 5 rue du moulin neuf, 80000 Amiens, FRANCE
{cyril.randriamaro,olivier.soyez,gil.utard,francis.wlazinski}@u-picardie.fr

Abstract

This article presents a dynamic data distribution method for data storage in a P2P system. In our system (named Us), peers are arranged in groups called Metapeers to deal with account failure correlation. To minimize end user traffic according to the reconstruction process, distribution must take into account a new measure: the maximum disturbance cost of a peer. In a previous vork, we defined a static distribution scheme which minimizes this reconstruction cost derivated from affine plan theory. In this paper we extend this distribution scheme to deal with the dynamic behaviour of peer to peer systems.

1 Introduction

Today, Peer to Peer systems (P2P) are widely used mechanisms to share resources on Internet. Very popular systems was designed to share CPU (`Seti@home`, XtremWeb, Entropia) or to publish files (Napster, Gnutella, Kazaa). In the same time, some systems was designed to share disk space (OceanStore [1, 2], Intermemory [3], PAST [4], Farsite [5]). The primary goal of such systems is to provide a transparent distributed storage service. These systems share common issues with CPU or files sharing systems: resource discovery, localisation mechanisms, dynamic point to point network infrastructure... But, for sharing disk systems, data lifetime is the primary concern. P2P CPU or file publishing systems can deal with node failures: the computation can be restarted anywhere or the published files resubmitted to the system. For disk sharing systems, node failure is a critical event: the stored data are definitively lost. So data redundancy and data recovery mechanisms are crucial.

*This Project (http://www.ustorage.net) is supported by the ACI GRID CGP2P and the ACI MD GDX.

1.1 Peer to Peer storage systems

Among peer to peer data systems, we distinguish two categories: P2P systems devoted to document publishing, and P2P systems devoted to storage which usually integrate data survival mechanisms.

P2P systems are mainly characterised by an over-network or virtual backbone. Such systems are based on point to point connections leading to a loosely coupled graph. The aim of the over-network is mainly to furnish routing and localisation mechanisms.

Since Napster, lot of P2P file publishing systems have appeared. Gnutella [6] is one of them and is defined as a protocol specification to share documents. File localisation is done by a breath-search in the connection graph. Freenet [7] is also a file publishing project where one of the primary goals is to insure anonymity of users (data producer or data consumer). It integrates cryptography of documents, auto adaptable routing, and a primitive replication mechanism to insure data survival of popular files.

Concerning peer to peer storage systems, one precursor is InterMemory [3]. It is characterised by a complex redundancy scheme for data survival. PAST [4] is a joint project of the Rice University and Microsoft. In PAST, nodes and files have a unique identifier in a common name space. Tolerant routing mechanism is PASTRY. Data survival is done by file replications. OceanStore [1, 2] is a large project. Data are stored in a set of collaborative untrusted servers with long time survival and high speed connection.

1.2 Us system

In this paper, we focus on the second category of peer to peer systems we have defined in 1.1. We present dynamic data distribution strategies for a peer to peer storage system we are designing called Us (Ubiquitous storage) [8, 9]. For scalability, data are distributed on thin peers using the well known Rabin

Proceedings of the 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)
0-7695-2434-6/05 $20.00 © 2005 IEEE

dispersal technique [10]. Contrarily to other systems like OceanStore, where data are distributed on server peers, in Us, data are distributed on end user peers: each Us peer is both storage space consumer and storage space provider. The main goal of Us is to provide a virtual storage device to each user which insures data durability.

Us shares common features with the OceanStore project: to insure durability, a data dissemination with a data redundancy mechanism is used. The advantage of such method is scalability. The inconvenient is that we have to face a higher failure rate of peer because the number of peers is a several order of magnitude greater than the number of peers in OceanStore and peers are less robust than OceanStore servers.

The main mechanism used to insure data durability is redundancy based on erasure code. Such code is the mechanism used by OceanStore and Us to maintain data-survival.

In [11], we have studied the MTTF (Mean Time To Failure) of peer to peer storage systems and we have shown that we have to face a continuous stream of data in the peer to peer network to insure data reconstruction process. We have shown that peer volatility generates a huge amount of data communication. So we have to distribute fragments of data on peers to guarantee a good load balance of the commmunication during the reconstruction process. It follows that Us should not be intrusive for thin client during the continuous repairing process: the fraction of the thin client bandwidth used for reconstruction must be as low as possible. Our objective is to minimise the maximum number of fragments that any alive peer must send during reconstruction processes. Moreover, the reconstruction must be diluted among peers by a good data distribution and in addition the failure correlation must be managed.

1.3 Redondancy mechanism

To insure data durability Us use usual redundancy mechanism based on erasure code techniques: peers (physical computers) send data blocks to be stored on other peers. Each block is split into f fragments including redundancy informations. For perennity reasons, each fragment is stored on a different peer. When a peer fails, all fragments it stored must be rebuild and redistributed to other peers. To rebuild each fragment, $f-1$ fragments must be grabbed from some other peers.

1.4 Failure correlation

Depending on geography, a peer failure may be correlated with other peers failures, like electrical damage, flooding. An another example due to peers very close geographically, implys peers physically under the same network, if this network is shutdown, all peers under this networks is down. The notion of failure correlation, introduced in [12], is an important factor for fault tolerance technique. Peers selected for dissemination of fragments of a data block must avoid correlated failures, otherwise correlated failures may catch the redundancy mechanism out.

1.5 Metapeers

In Us, peers are arranged in groups called *Metapeer* according to their correlated failure. Each peer belongs to exactly one Metapeer. A couple of peers which exhibits a high probability of correlated failure belong to the same Metapeer. So, a couple of peers coming from two different Metapeer must exhibit low probability of correlation failure. When a block of data is disseminated in Us, peers choosen to store fragments are selected from different *Metapeer*. Two fragments of the same block cannot be stored on peers of the same Metapeer. Due to the data redondancy information, all of the peers of the same Metapeer can be down without data losses. How the Metapeers are constructed is not the topic of this paper, interested reader can consult the Weatherspoon et al paper [12] which present a framework for online discovery of such Metapeers.

2 Definitions

2.1 Notations

Let P be the set of peers, and B be the set of stored blocks. For a peer p, α_p is the number of fragments stored by p, and B_p is the set of blocks such that p stores a fragment of, i.e. the set of the blocks to rebuild for peer p failure. Finally, let N be the total number of peers, $f(\leq N)$ be the fragments number of a block, and NB be the number of blocks.

2.2 Data distribution definition

A data distribution maps fragments from blocks over the peers. A distribution is restricted by the condition that the f fragments of one data block are stored on f distinct peers. We consider $f \leq N$.

Proceedings of the 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)
0-7695-2434-6/05 $20.00 © 2005 IEEE

P	The peer set [1–N]
f	Fragments number of a block, $f \leqslant N$
B	The blocks set [1–NB]
b	A block, a set of f peers
p	A peer
B_p	Block set of peer p
NB	Total number of blocks, NB=$\mid B \mid$
N	Total number of peers, N=$\mid P \mid$
α_p	Number of fragments stored by peer p

Table 1. This table summarises notations used in this paper.

Each block b can be represented by the list of those f peers. The fragments of a peer p belong to distinct blocks. A data distribution D can be defined by:

$$D : B \mapsto P^f$$
$$\forall b \in B, p_1, p_2, ..., p_f \in P, p_1 \neq p_2 \neq ... \neq p_f,$$
$$b \mapsto \{p_1, p_2, ..., p_f\}$$

For any data distribution, and for any number of blocks stored, we have :

$$NB = \frac{1}{f} * \sum_{i=1}^{N} \alpha_p \quad (1)$$

Now let introduce the notion of communication cost for peers during the reconstruction process.

2.3 Local communication cost of a peer

The disturbance cost is indicated by the number of data communications which are requested from a single peer for rebuilding lost data. The local communication cost $C_{loc(p,q)}$ is the number of fragments that a peer p sends to rebuild fragments of a peer q:

$$\forall p, q \in P, C_{loc(p,q)} = |B_p \bigcap B_q|$$

And the total number of fragments needed by the reconstruction is equal to the sum of all local cost peers, except the dead peer q. So we have :

$$\forall q \in P, \alpha_q * (f-1) = \sum_{p=1, p\neq q}^{N} C_{loc(p,q)} \quad (2)$$

In Figure 1 example, if peer 4 fails, local costs are : $C_{loc(1,4)}$=$C_{loc(2,4)}$=$C_{loc(3,4)}$=$C_{loc(6,4)}$=$C_{loc(7,4)}$= $C_{loc(8,4)}$=1, $C_{loc(5,4)}$=2 and $C_{loc(9,4)}$=0.

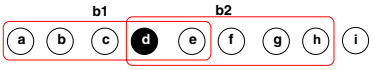


Figure 1. Two blocks storage, $f = 5$, $N = 9$.

2.4 Global communication cost of a peer

While two peers can send packets simultaneously, the global communication cost is defined by the most sending peer, i.e. the global communication cost is the maximum of all local communication cost. Let $C_{glob(q)}$ be the global cost to rebuild peer q fragments:

$$\forall q \in P, C_{glob(q)} = \max C_{loc(1,q)}, C_{loc(2,q)}, ...,$$
$$C_{loc(q-1,q)}, C_{loc(q+1,q)}, ..., C_{loc(N,q)}$$

Back to the storage example of Figure 1, peer 4 global communication cost is $C_{glob(4)} = 2$.

2.5 Maximal communication cost

Considering a fragment distribution peers and that any peer can fail, the maximal communication cost is the maximum of the global communication costs:

$$C_{max} = \max_{q \in P} C_{glob(q)}$$

Back to the storage example of Figure 1, maximal communication cost is $C_{max} = 2$.

2.6 Problem formulation

Let us define the notion of optimal distribution.

Definition 2.1 $\forall f, \forall N, \forall NB$, *an* **optimal distribution** D *is a data distribution that minimizes the maximal communication cost* C_{max} *with the given number of stored blocks* NB*, so let* D' *be an another data distribution:*

$$C_{max}(D) \leq C_{max}(D')$$

Let N and f be fixed parameters. Our goal is to provide an optimal distribution for a given value of NB. By definition, this is equivalent to providing an optimal distribution for a given value of C_{max}.

3 Distributions

We want to extend the problem to find a dynamic distribution taken into account the failure correlation.

Proceedings of the 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)
0-7695-2434-6/05 $20.00 © 2005 IEEE

So in a first time, we present three static data distribution. In 3.1.1 and 3.1.2, we present existing distributions coming from mathematical theory: finite affine plane distribution for $N = f^2$ and finite projective plane distribution for $N = f^2-f+1$. But these distributions are too restrictive for our problem. In 3.1.3, we give a new method of distribution which respect all conditions of our problem in a static way: a general case distribution for all values of N. And in a second time, we present dynamic data distribution. In 3.2.1, we present the random distribution that will be compared with our distributions in 3.3. In 3.2.2, we give a method based on Metapeer which is more adapted to dynamic problems: a Metapeers distribution.

3.1 Static data distributions

3.1.1 Finite affine plane distribution

Let take an optimal distribution based on the construction of finite affine planes of order f, when this construction is possible. The order of an affine plane is the number n, $n \geq 2$, such that :

1. The total number of points is n^2 and the total number of lines is $n(n+1)$.
2. All the lines share n points and all points share $n+1$ lines.
3. The intersection of two lines is no more that one.

So, the analogy with our problem is :

1. The order n corresponds to the number f of peers in a blocks.
2. The points of the finite affine plane of order n are peers, so $N = f^2$.
3. The lines of the finite affine plane of order n are blocks, so $NB = f^2 + f$.
4. The intersection of two blocks is no more that one, this imply a $C_{max} = 1$.

We have proved in [13], that this distribution is an optimal one. Figure 3 represents a finite affine plane of order 3. It has 9 points and 12 lines.

This distribution requires N to be equal to f^2, it is a high restriction. In addition, for some values of f, finding a construction of an affine plane of order f is still an open problem. But this distribution gives a good structuration of the network.

Figure 3 is an example of such distribution. Let P={1,2,3,4,5,6,7,8,9} be the point set, then the lines set is: L={{1,2,3},{4,5,6},{7,8,9},{1,4,7},{2,5,8},{3,6,9},{1,5,9}, {2,6,7},{3,4,8},{3,5,7},{2,4,9},{1,6,8}}

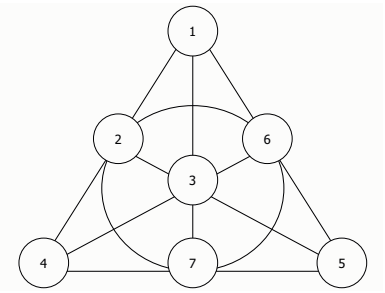


Figure 2. The Fano plane or finite projective plane of order 2

3.1.2 Finite projective plane distribution

In this case, a distribution can be defined by the construction of finite projective planes of order $(f-1)$, when this construction is possible. The order of the projective plane is n, such that the number of points is n^2+n+1 and the number of lines is n^2+n+1, all the lines share $n+1$ points and all points share $n+1$ lines. The intersection of two lines is one.

Figure 2 is an example of a finite projective plane of order 2, called the Fano plane. It is composed of 7 points and 7 lines. Each line contains 3 points. If $P = \{1,2,3,4,5,6,7\}$ is the points set, then the lines set is L={{1,2,4},{2,3,5},{3,4,6},{4,5,7},{1,5,6},{2,6,7},{1,3,7}}. Line {2,6,7} in figure 2 is represented by a circle.

The analogy with our problem is that the order n may correspond to the number $f-1$ where f is the number of peers in a block and the points of the finite projective plane of order n may correspond to peers. It follows that the total number of peers is $N = f^2-f+1$. The lines of the finite projective plane of order n are blocks. So, we get $NB = f^2-f+1$ and the intersection of two blocks is 1.

Like the distribution based on finite affine plane: this distribution is optimal, but requires N to be equal to f^2-f+1. For some values of f, finding a construction of a projective plane of order $f-1$ is still an open problem. But this distribution gives a good structuration of the network.

3.1.3 General case distribution

This distribution is designed for f a prime number and all N, such that $f^2 \leq N$. First, we construct M_i matrices that are used to build the distribution.

Let r, s be two integers such that $r^2 \leq s$. Assume there exists a greatest prime integer p such that $p \times r \leq s$ and $r \leq p$. For instance, such an inte-

Proceedings of the 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)
0-7695-2434-6/05 $20.00 © 2005 IEEE

ger p trivially exists when r is prime and $r^2 = s$: in this case $p = r$. We consider the p matrices M_1, M_2, ..., M_p with p lines and r columns defined by $M_k = (a^k_{ij})_{1\leq i\leq p; 1\leq j\leq r}$ where $a^k_{i1} = k$ and $a^k_{ij} = 1 + (j-1) \times p + ([i - 1 + (k-1) \times (j-2)] \mathrm{mod} p)$ $\forall 1 \leq i \leq p$ and $\forall 2 \leq j \leq r$.

For example, when $r = 3$ and $s = 9$, we get $p = 3$, $M_1 = \begin{pmatrix} 1 & 4 & 7 \\ 1 & 5 & 8 \\ 1 & 6 & 9 \end{pmatrix}$, $M_2 = \begin{pmatrix} 2 & 4 & 8 \\ 2 & 5 & 9 \\ 2 & 6 & 7 \end{pmatrix}$ and $M_3 = \begin{pmatrix} 3 & 4 & 9 \\ 3 & 5 & 7 \\ 3 & 6 & 8 \end{pmatrix}$.

Let us first remark that, for any integer q such that $0 \leq q \leq r-1$, the integers of the interval $[1 + p \times q; p \times (q+1)]$ only appear in the $(q+1)^{\text{th}}$ column of the matrices M_1, ..., M_k. Moreover, two different lines of the matrices M_1, ..., M_p have at most one common element.

Let us recall that f is the number of fragments and N the cardinal of the peers set. Let NB_P be the number of blocks, we will obtain with our construction.

We assume that $f^2 \leq N$ and f is prime. We define two integers p_1 and p_2 in the following way. The integer p_1 is the greatest prime integer such that $p_1 \times f \leq N$ and $f \leq p_1$. The integer p_2 is the greatest integer such that $p_2 \times f \leq p_1$ and which verifies either $p_2 < f$ or p_2 is prime.

We can build a distribution such that $NB_P = p_1^2 + f \times p_2$ when $p_2 < f$ and $NB_P = p_1^2 + f \times p_2^2$ when $p_2 \geq f$.

To manage failure correlation, this distribution is quasi-optimal and gives a good structure that can be used to implemented Metapeers over peers. The main inconvenient is this distribution is not flexible about the value N. When N moves, it is not reasonable to redistribute always all data. At this moment, no algorithm is able to take into account this condition.

So, we use a random distribution coupled with this distribution to obtain a dynamic data distribution that managed failure correlation.

Let us consider a structured optimal distribution , like in Figure 3, that optimizes the fragment sends. When a peer fails, the rebuilt fragments must be stored on other peers. On the one hand, several peers can store the new fragments. Then the structure is blown. On the other hand, new fragments can be stored on the same new peer to guaranty that the damaged structure is rebuilt. Then, the sending parallelization is avoid by the reception.

Satisfying both conditions can be performed by the use of a peer group instead of a single peer for each structure node. Such groups are called Metapeers.

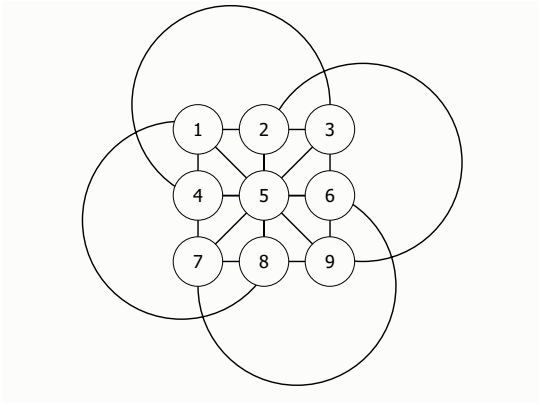


Figure 3. Finite affine plane of order 3

Due to failure correlation, this is the distribution chosen to organize Metapeers over the peers.

3.2 Dynamic data distribution

3.2.1 Random distribution

The random distribution stores the f fragments of each data block on f distinct peers chosen randomly among all the peers. Due to statistics, this distribution must be efficient for a large number of peers. Indeed the probability to obtain equal lists of f peers or with a big number of common peers is weak. Nevertheless, this distribution needs a global knowledge of the full network, which is difficult to implement in a peer to peer network. The storage system PAST [4] is an example of such a distribution use: each peer and all resources have an unique identifier, associated with a dynamic routing system depending on these identifiers. A file is stored on the peer the identifier is the closest to the identifier file. The peer volatility implies that a new peer with a closer identifier can appear after the storage. Then, additional communications must be generated to find the file. Random data distribution is usually a good non optimal data distribution to minimize the reconstruction cost such defined. But unfortunately this distribution does not permit to exploit the physical network topology to avoid failure corellation. To do so, structured distribution strategies must be applied.

3.2.2 Metapeer distribution

The optimal distribution is not well adapted to the dynamic behavior of peer to peer systems. So, we propose a distribution which mixes optimal distribution

Proceedings of the 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)
0-7695-2434-6/05 $20.00 © 2005 IEEE

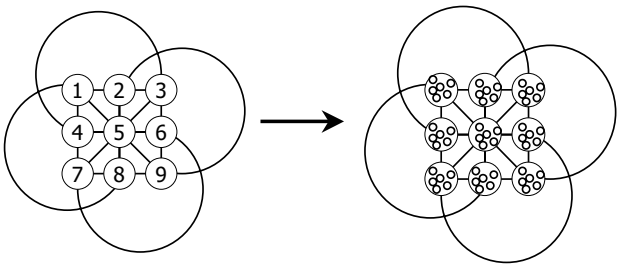


Figure 4. From optimal distribution to optimal dynamic distribution

with a random one. In this distribution, the set of peers is partitioned in groups called Metapeers and the set of Metapeers is structured by the optimal distribution . Figure 4 is an example of such distribution. The number of Metapeers is selected in such a way that we will be able to achieve an optimal distribution over their.

Let us define a dynamic distribution, we use the General Case distribution and replace peers by Metapeer : node i from the distribution is replaced by Metapeer i. A fragment stored in node i will be stored in one of the peers of Metapeer i. Consequently the number of blocks is proportionnal to the number of nodes in the resulting structure.

Our simulation showed that this distribution is able to achieve better performance than the random distribution, but unfortunatly we can not retrieve the optimal performance of an optimal distribution. In the next part, we explain how the routing can be made into the Metapeers and we explain the reconstrution process.

With this structured distribution, we are able to define a mecanism for the management of the dynamic behavior of peer to peer storage systems. For instance, when a new peer arrives, it first selects the Metapeer it will integrate. To improve data lifetime, the Metapeer is selected in such a way that the new peer is geographically far from peers of other Metapeers (w.r.t. some balancing criterions). The new peer also selects peers of other Metapeers which have good communication bandwidth with it. Then, when a reconstruction must be achieved, it sends fragments to those peers.

In the same way that peers of the old distributions, fragments of a block are distributed over Metapeers of the structure. For each selected Metapeer, a random function selects the storing peer. In order to balance the storage, the random function is modified to tend to select the peer that stores the less. Afterwards, the function will take into account the network topology.

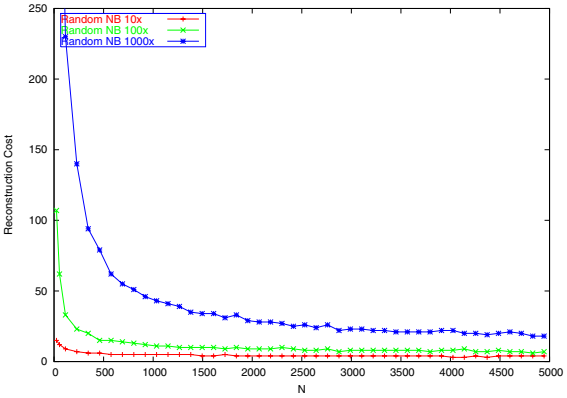


Figure 5. Recontruction cost depending on N. Impact of NB on the random distribution with $f = 5$.

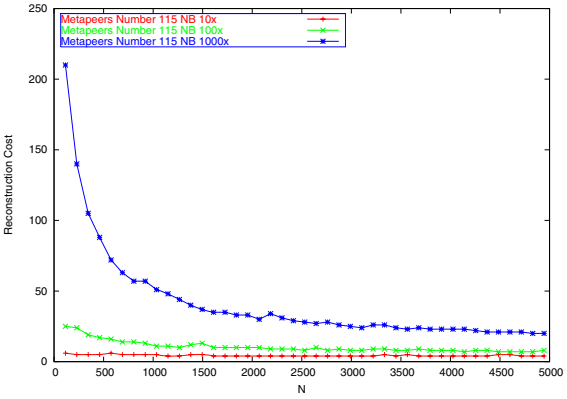


Figure 6. Recontruction cost depending on N. Impact of NB on the Metapeer distribution with $f = 5$.

3.3 Analysis

The first two distributions are the finite affine plane distribution seen in section 3.1.1 and finite projective plane distribution seen in section 3.1.2. They are kinds of the General Case distribution. These distributions were analyzed in paper [13]. We showed that the General Case distribution cost is close to or equal to

Proceedings of the 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)
0-7695-2434-6/05 $20.00 © 2005 IEEE

the theoretical bound. To realize the efficiency of the Metapeer distribution, we compare the General Case distribution, that is an ideal distribution, with the random distribution. The Metapeer distribution and random distribution are dynamic distributions. Even if we know that the General Case distribution is not dynamic, we know that no distribution can be better than the lower bound given by the General Case distribution. So the General Case distribution will be our reference. Our goal is to evaluate how much it costs to take into account the failure correlation and the structure.

For our experimentations, we use a simulator that computes the reconstruction cost depending on the value N, f, and a given distribution.

Figure 5, 6, show the impact of NB depending on N on the reconstruction cost. The value NB is always a factor of N, from 10 to 1000. Figure 5 shows for the random distribution and Figure 5 represents the Metapeer distribution.

Figure 7, 8, show the impact on the reconstruction cost of f depending on N. Figure 7 is with f equals to 5. Figure 7 is with f equals to 29.

We always consider that the Metapeer size is the same for all Metapeers. Consequently, Figure 7, 8, 6 show the reconstruction cost depending on the Metapeer size, while the depend on N. For exemple, the first point given by a Metapeer distribution is obtained with a Metapeer size of one, i.e one peer per Metapeer, and consequently with a value of N equals to the total number of Metapeers.

Figure 5 and Figure 6 show that with random distribution and the Metapeer distribution, when you increase the number of blocks stored by each peer, we have the same behavior in both case. This is due to the fact that in the Metapeer distribution, we use a random selection inside Mmetapeers. This is the reason why the behavior is similar.

Figure 7 shows that for small values of the Metapeer size, the random distribution cost is worth than the Metapeer one. It confirms the advantage to compute an optimal distribution versus a random distribution. Another observation, see Figure 8, is about the Metapeer sizes: the Metapeer distribution cost is close to the random distribution cost, when the Metapeer size is bigger than two. So we do not need to choose a great number of Metapeer. Hence we dont need to have a big structure to manage the failure correlation.

Figure 8 shows that even if the Metapeer sizes grows, the Metapeer distribution cost is always very close to the random distribution cost. We can conclude, that the cost to manage the failure correlation and to have a dynamic distribution is not so high, but we expect to reduce this cost.

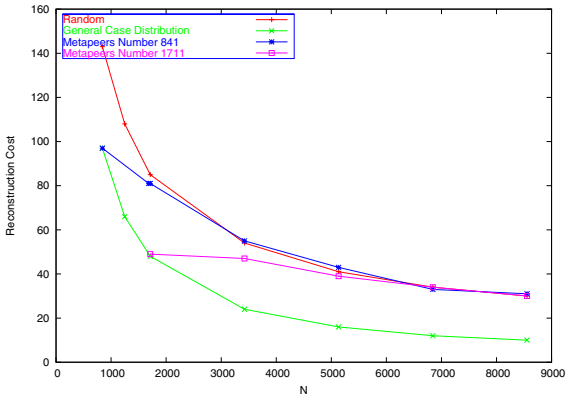


Figure 7. Recontruction cost depending on N. Each peer stores around 100 blocks and $f = 29$.

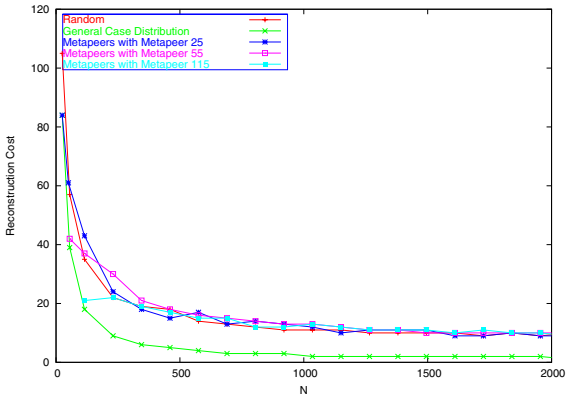


Figure 8. Recontruction cost depending on N. Each peer stores around 100 blocks and $f = 5$.

4 Conclusion

In this paper, we analyzed the reconstruction cost in a peer to peer storage system where data are distributed using a dispersal redundant information scheme. A good distribution of data is a distribution which minimizes the data sent by a peer to rebuild data lost by a peer failure. The random data distribution is usually a good distribution to minimize the

Proceedings of the 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)
0-7695-2434-6/05 $20.00 © 2005 IEEE

reconstruction cost for big value of N. But unfortunately this distribution does not permit to exploit the physical network topology to take into account of the peer failure correlations. On the other hand, an optimal distribution is too strict and it is not well adapted to the dynamic behavior of peer to peer systems. We proposed a distribution which mixes the static General Case distribution with a random one.

Simulations show that the number of Metapeers are selected in such a way we are able to achieve a good distribution taken into account peer failure correlation. We can potentially improve the Metapeer distribution to have better performance than others. We think that this can be done by the modification of the peer selecting fonction in Metapeer. This is a future feature.

We want to thank to Loïc Crampon for his participation in simulator development.

References

[1] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of ACM ASPLOS*. ACM, November 2000.

[2] C. Wells, "The oceanstore archive: Goals, structures, and self-repair," Master's thesis, University of California, Berkeley, May 2001.

[3] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos, "A prototype implementation of archival intermemory," in *Proceedings of the Fourth ACM International Conference on Digital Libraries*, 1999.

[4] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility," in *Procedings of HOTOS*, 2001, pp. 75–80.

[5] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," 2002. [Online]. Available: citeseer.ist.psu.edu/adya02farsite.html

[6] M. Ripeanu, "Peer-to-peer architecture case study: Gnutella network," in *Proceedings of International Conference on Peer-to-peer Computing*, August 2001.

[7] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Workshop on Design Issues in Anonymity and Unobservability*, 2000, pp. 46–66.

[8] C. Randriamaro, O. Soyez, and G. Utard, "Us : Prototype de stockage pair à pair," Laboratoire de Recherche en Informatique d'Amiens," Technical Report LaRIA 2003-09, Sept. 2003.

[9] O. Soyez, "Us : Prototype de stockage pair à pair," in *RENPAR 2003, la Colle sur Loup, France*, October 2003, pp. 214–218.

[10] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *Journal of ACM*, vol. 38, pp. 335–348, 1989.

[11] G. Utard and A. Vernois, "Data durability in peer to peer storage systems," in *4th IEEE Workshop on Global and Peer to Peer Computing*, Chicago, April 2004.

[12] H. Weatherspoon, T. Moscovitz, and J. Kubiatowicz, "Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems," in *Proceedings of International Workshop on Reliable Peer-to-Peer Distributed Systems*, 2002.

[13] C. Randriamaro, O. Soyez, G. Utard, and F. Wlazinski, "Data distribution in a peer to peer storage system," in *GP2PC05 2005, UK, Cardiff*, May 2005.

Proceedings of the 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)
0-7695-2434-6/05 $20.00 © 2005 IEEE

Quatrième partie

Perspectives

Chapitre 12

Travaux futurs

Mes dernières activités de recherche se concentrent autour du stockage distribué à grande échelle qui fait suite au travail de valorisation précédent. Ce travail c'est concrétisé par un brevet sur un système pérenne de stockage distribué à grande échelle [38], et la réalisation d'un logiciel, appelé DeepTorrent, basé sur une extension du protocole BitTorrent, qui est aujourd'hui exploité par la société Ugloo [1], avec qui j'ai un contrat de collaboration. Autour de ce travail de développement logiciel, d'autres aspects ont été abordés, tels que le contrôle d'accès anonyme aux données [11], ou l'intégration d'un mécanisme de RPC dans les DHT [12].

En manière de perspective de recherche, toujours dans le cadre de cette collaboration avec la société Ugloo, je compte continuer les travaux autour du stockage pérenne à très grande échelle, notamment sur les codes de redondance et les mécanismes de reconstruction.

A plus long terme, je voudrais aborder les méthodes et techniques qui permettraient d'établir la confiance dans un réseau ouvert d'opérateur de stockage, qui s'appuient sur l'utilisation des blockchains.

12.1 Maintien de l'intégrité des données à grande échelle

Cette dernière décennie, l'étude de nouveaux codes correcteurs ainsi que des mécanismes de régénération a connu un vif regain d'intérêt, notamment par les plus gros opérateurs de stockage que sont Google, Amazon et Microsoft Azure qui soumettent actuellement de nombreux brevets sur ce domaine (voire par exemple US9244761B2, US8386841B1) . Ces opérateurs ont été aussi confrontés au coût de la réparation. Cependant la plupart de ces travaux s'adressent essentiellement à des architectures de type data-center où il y a une grande homogénéité du matériel et des réseaux spécifiques, ainsi qu'un taux de volatilité (churn) contrôlé par l'environnement (personnel dédié à la maintenance).

Cela ne correspond pas aux environnements que nous souhaitons adresser qui se caractérisent par une très grande hétérogénéité du matériel et du réseau sous-jacent, ainsi qu'une volatilité beaucoup plus aléatoire. L'irrégularité du réseau implique que le placement des données et de leur redondance a beaucoup d'impact sur les performances et la pérennité. De précédents travaux théoriques ont montré que certaines parties de ce problème étaient NP-difficile, notamment quand un des objectifs est de minimiser l'impact des reconstructions sur chaque nœud. Des travaux plus récents abordant cette problématique [Bar17], ont proposé des simulations dans un cas d'usage particulier (implémentation d'un tuple space) en utilisant le code clas-

1. Vous trouverez une description de la solution dans le livre blanc mis en annexe

sique de Reed Solomon. Les auteurs ont évalué quelques stratégies de placement sans en identifier de particulièrement prometteuses. D'autre part, cette étude ne considère pas des taux de churn élevés, tels que celles que l'on rencontrerait dans un environnement plus volatile. Il y a donc tout un nouveau champ d'études et d'expérimentations à mener qui devront tenir compte des nouveaux codes de redondance, des techniques de network coding, d'équilibrage de charge dynamique. C'est en partie ce nouveau domaine qu'abordera mon projet de recherche.

Une approche complémentaire est d'étudier les nouveaux codes correcteurs. En effet, pour réduire le coût de la régénération des données, les nouveaux codes se divisent en deux grandes catégories, les codes locaux (local code ou LC) [Sat13], qui au prix d'un taux de redondances plus élevé réduisent la taille des informations nécessaires à une reconstruction, notamment avec des redondances hiérarchiques, et les codes régénérateurs (Regenerator code ou RG) [Dim10], qui sont capable de régénérer les données avec moins de lecture grâces aux techniques de Network Coding. Enfin, une approche utilisant de très larges codes, où la fragmentation des données est poussée à l'extrême pour fournir un très haut niveau de redondances sans surcoût à fait son apparition ces dernières années et se base sur les codes fontaines, comme par exemple Raptor [Lub17]. Cette étude nécessitera des travaux d'analyse quantitative pour estimer la pertinence de ces derniers à notre système. Cela nécessitera la conception d'un modèle stochastique de notre système.

De la même manière, une grande partie de ce travail préliminaire consistera au développement de simulateur de notre solution sur un Framework tel que Omnet++, qui permettra, d'une part d'analyser les performances de notre système sur divers types de réseaux à grande échelle, et d'autre part évaluer l'efficacité des nouveaux mécanismes de redondance. Cette simulation nous permettra aussi d'étudier de quelle manière nous pourrons implémenter ces nouveaux codes.

Nous estimons que le choix de nouveaux schémas de redondance, ainsi que de nouveaux mécanismes de reconstruction, auront un impact non négligeable sur la structure du code actuel qui a s'appuie sur des schémas réguliers de reconstruction. L'architecture du code sera probablement impactée par les nouveaux choix, notamment en ce qui concerne le placement des données. Il faudra développer de nouveaux prototypes et les tester sur des plateformes de test dédiées.

Enfin, le choix de nouveaux codes peut impliquer de nouvelles primitives d'encodage et de décodage. Certaines primitives peuvent être coûteuses en calcul. Actuellement, j'ai fait l'acquisition d'un nouveau type d'architecture, les PIM (Processeurs In Memory) de la société UpMem. C'est une architecture massivement parallèle qui intègre des processeurs au niveau de RAM. Un travail connexe sera donc d'étudier l'efficacité de ce type d'architecture pour de nouveaux noyaux de calcul.

12.2 Construction d'un réseau ouvert fiable d'agents de stockage

Avec l'introduction de son offre S3 en 2007, Amazon a créé un nouveau marché du stockage des données. Amazon propose de l'hébergement de données dans ses centres de données, dans lesquels les clients peuvent déposer et retirer leurs données, tout en bénéficiant de l'ubiquité intrinsèque qu'offre Internet.

Aujourd'hui, Amazon segmente son marché en déclinant de nouvelles offres de stockage avec différents niveaux de qualités de service pour adresser les différents cycles de vie des données. Les autres majors, que ce soit Google, Apple ou Microsoft, se sont aussi engouffrés sur ce nouveau marché et ont leurs propres offres, respectivement Google-Drive, Apple iCloud ou Microsoft-Drive. D'autres acteurs de taille plus modeste, tels que OVH, proposent aussi leur solution de stockage Cloud.

Depuis plus de dix ans, les efforts de standardisation des interfaces de stockage, par exemple S3 ou OpenStack, font que le marché est passé du B2C au B2B. Aujourd'hui, des acteurs tels que DropBox, ou des éditeurs de logiciels de stockage, tels que EMC, Rubrick, Veam... utilisent les offres de stockage Cloud existantes pour proposer leur service à leurs clients. Le secteur se structure, avec d'un côté des opérateurs qui proposent du stockage pur, et de l'autre des acteurs qui proposent des services ajoutés sur les données stockées.

Le succès est tel que prochainement l'offre sera insuffisante. Les tarifs commencent dès aujourd'hui à augmenter (voir Amazon ces dernières années), et les majors sont contraint de déployés de nouveau data centers pour faire face à la demande. Il est très difficile pour de nouveau acteurs d'entrer dans ce marché de par les investissements nécessaires pour avoir la masse critique minimale, et de par la position de domination qui a été prise par les majors.

La seule solution pour ouvrir le marché à de nouveaux acteurs est de définir un protocole qui permette de s'intégrer dans un réseau global, où chaque agent est rétribué en fonction de sa contribution. C'est ce que proposent les projets SIA, Storj ou Filecoin en se basant sur le pouvoir de libéralisation des marchés par l'utilisation de blockchains [Ben20].

Dans les solutions qui sont proposées, une des problématiques les plus importantes, et de vérifier l'intégrité dans le temps des données stockées par des tiers, à savoir comment vérifier que chaque opérateur respect le contrat de conservation des données dans la durée. C'est ce travail que j'aimerai aborder par la suite.

Ce travail abordera principalement l'étude des mécanismes de contrôles de l'intégrité des données stockées chez les tiers, et en particulier un mécanisme de contrôle collégial. Ce mécanisme ce basera en premier lieu sur les techniques de preuve de possession (PDP et POR), qui génèrent des challenges que doivent valider les tiers pour prouver qu'ils possèdent bien les données stockées. Elle se basera en second lieu sur l'étude des mécanismes d'incitation, de punitions et de réputations basés sur l'utilisation de la blockchain, qui permettent d'asseoir la confiance dans de tels systèmes.

Bibliographie

[Bar17] R. Barbi, V. Buravlev, C. A. Mezzina, et V. Schiavoni. *Block Placement Strategies for Fault-Resilient Distributed Tuple Spaces : An Experimental Study.* in Distributed Applications and Interoperable Systems, vol. 10320, L. Y. Chen et H. P. Reiser, Éd. Cham : Springer International Publishing, 2017, p. 67-82.

[Ben20] Nazanin Zahed Benisi, Mehdi Aminian, Bahman Javadi. Blockchain-based decentralized storage networks : A survey. In *Journal of Network and Computer Applications*, Volume 162, July 2020. https ://doi.org/10.1016/j.jnca.2020.102656.

[Che99] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, 1999.

[Dab01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Symposium on Operating Systems Principles*, pages 202–215, 2001.

[Dru01] P. Druschel and A. Rowstron. PAST : A large-scale, persistent peer-to-peer storage utility. In *Procedings of HOTOS*, pages 75–80, 2001.

[Dim10] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, et K. Ramchandran. *Network Coding for Distributed Storage Systems.* http ://arxiv.org/abs/0803.0632.

[Gol98] A. Goldberg and Peter N. Yianilos. Towards an archival intermemory. In *Proceedings of IEEE Advances in Digital Libraries, ADL 98*, pages 147–156, Santa Barbara, CA, 1998. IEEE Computer Society.

[Kub00] John Kubiatowicz, Divid Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore : An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[Lub17] M. G. Luby, R. Padovani, T. J. Richardson, L. Minder, et P. Aggarwal. *Liquid Cloud Storage.* http ://arxiv.org/abs/1705.07983.

[Pat98] 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.

[Row01a] Antony Rowstron and Peter Druschel. Pastry : Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[Row01b] Antony I. T. Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001.

[Sat13] M. Sathiamoorthy et al. *XORing Elephants : Novel Erasure Codes for Big Data.* http ://arxiv.org/abs/1301.3791.

[She01] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John Kubiatowicz. Bayeux : An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video*, June 2001.

[Sto01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[Wea02a] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication : A quantitative comparison. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, March 2002.

[Wea02b] Hakim Weatherspoon, Tal Moscovitz, and John Kubiatowicz. Introspective failure analysis : Avoiding correlated failures in peer-to-peer systems. 2002.

[Zha01] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry : An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

Cinquième partie

Annexes

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) **EP 1 815 359 B1**

(12) **FASCICULE DE BREVET EUROPEEN**

(45) Date de publication et mention de la délivrance du brevet:
12.05.2021 Bulletin 2021/19

(21) Numéro de dépôt: **05818291.6**

(22) Date de dépôt: **18.11.2005**

(51) Int Cl.:
G06F 11/10 *(2006.01)* ***G06F 11/14*** *(2006.01)*
G06F 16/182 *(2019.01)*

(86) Numéro de dépôt international:
PCT/FR2005/002876

(87) Numéro de publication internationale:
WO 2006/056681 (01.06.2006 Gazette 2006/22)

(54) SYSTÈME ET PROCÉDÉ DE SAUVEGARDE DISTRIBUÉE PÉRENNE DES DONNÉES

VORRICHTUNG UND VERFAHREN ZUR VERTEILTEN DAUERHAFTEN DATENSICHERUNG

SYSTEM AND METHOD FOR PERENNIAL DISTRIBUTED DATA BACK UP

(84) Etats contractants désignés:
AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HU IE IS IT LI LT LU LV MC NL PL PT RO SE SI SK TR

(30) Priorité: **26.11.2004 FR 0452788**

(43) Date de publication de la demande:
08.08.2007 Bulletin 2007/32

(73) Titulaire: **UGLOO**
75002 Paris (FR)

(72) Inventeurs:
- **UTARD, Gil**
  F-80440 Boves (FR)
- **RANDRIAMARO, Cyril**
  F-80310 Picquigny (FR)

(74) Mandataire: **Novagraaf Technologies**
Bâtiment O2
2, rue Sarah Bernhardt
CS90017
92665 Asnières-sur-Seine Cedex (FR)

(56) Documents cités:
US-A1- 2004 049 700 US-A1- 2004 064 693

- **BOLOSKY W J; DOUCEUR J R; ELY D; THEIMER M: "Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs" PERFORMANCE EVALUATION REVIEW, vol. 28, no. 1, juin 2000 (2000-06), pages 34-43, XP002344594 USA**
- **ADYA A; BOLOSKY W J; CASTRO M; CERMAK G; CHAIKEN R; DOUCEUR J R; HOWELL J; LORCH J R; THEIMER M; WATTENHOFER R P: "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment" PROCEEDINGS OF THE FIFTH SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 2002, pages 1-14, XP002344595 Berkeley, CA, USA**
- **MUTHITACHAROEN A; MORRIS R; GIL T M; CHEN B: "Ivy: a read/write peer-to-peer file system" PROCEEDINGS OF THE FIFTH SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 2002, pages 31-44, XP002344596 Berkeley, CA, USA**
- **KUBIATOWICZ J ET AL: "OceanStore: An Architecture for Global-Scale Persistent Storage", ASPLOS. PROCEEDINGS. INTERNATIONAL CONFERENCE ON ARCHITECTURALSUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, NEW YORK, NY, US, 1 January 2000 (2000-01-01), pages 1-12, XP002993765,**
- **WEATHERSPOON H ET AL: "Silverback: A global-scale archival system", ACM SOSP. PROCEEDINGS OF THE ACM SYMPOSIUM ON OPERATING SYSTEMSPRINCIPLES, ACM, US, 1 March 2001 (2001-03-01), pages 1-15, XP002302722,**
- **ZHAO B Y ET AL: "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing", REPORT UCB/CSD-01-1141, XX, XX, 1 April 2001 (2001-04-01), pages 1-27, XP002995768,**

EP 1 815 359 B1

Le stockage de données indestructible

Persistance de données au meilleur coût

Livre blanc - Août 2020

Ugloo