UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PARREAU, J. LOMBARD, M. MIGEON, J. CORTOIS, A.DUBRULLE

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CHAMLEY Hervé	Géotechnique
M. CONSTANT Eugène	Electronique
M. ESCAIG Bertrand	Physique du solide
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. LABLACHE COMBIER Alain	Chimie
M. LOMBARD Jacques	Sociologie
M. MACKE Bruno	Physique moléculaire et rayonn... ...iosphériques

M. MIGEON Michel	EUDIL
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BRASSELET Jean Paul	Géométrie et topologie
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie Physique
M. BRUYELLE Pierre	Géographie
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. CROSNIER Yves	Electronique
Mme DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBOURSE Jean Pierre	Gestion des entreprises
M. DEBRABANT Pierre	Géologie appliquée
M. DECLERCQ Roger	Sciences de gestion
M. DEGAUQUE Pierre	Electronique
M. DESCHEPPER Joseph	Sciences de gestion
Mme DESSAUX Odile	Spectroscopie de la réactivité chimique
M. DHAINAUT André	Biologie animale
Mme DHAINAUT Nicole	Biologie animale
M. DJAFARI Rouhani	Physique
M. DORMARD Serge	Sciences Economiques
M. DOUKHAN Jean Claude	Physique du solide
M. DUBRULLE Alain	Spectroscopie hertzienne
M. DUPOUY Jean Paul	Biologie
M. DYMENT Arthur	Mécanique
M. FOCT Jacques Jacques	Métallurgie
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. FRONTIER Serge	Ecologie numérique
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériques
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie-Physique
M. GRANELLE Jean Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILBAULT Pierre	Physiologie animale
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. HENRY Jean Pierre	Génie mécanique
M. HERMAN Maurice	Physique spatiale
M. LACOSTE Louis	Biologie Végétale
M. LANGRAND Claude	Probabilités et statistiques

M. LATTEUX Michel	Informatique
M. LAVEINE Jean Pierre	Paléontologie
Mme LECLERCQ Ginette	Catalyse
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique atomique et moléculaire
M. LEROY Jean Marie	Spectrochimie
M. LHENAFF René	Géographie
M. LHOMME Jean	Chimie organique biologique
M. LOUAGE Françis	Electronique
M. LOUCHEUX Claude	Chimie-Physique
M. LUCQUIN Michel	Chimie physique
M. MAILLET Pierre	Sciences Economiques
M. MAROUF Nadir	Sociologie
M. MICHEAU Pierre	Mécanique des fluides
M. PAQUET Jacques	Géologie générale
M. PASZKOWSKI Stéfan	Mathématiques
M. PETIT Francis	Chimie organique
M. PORCHET Maurice	Biologie animale
M. POUZET Pierre	Modélisation - calcul scientifique
M. POVY Lucien	Automatique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. RAMAN Jean Pierre	Sciences de gestion
M. SALMER Georges	Electronique
M. SCHAMPS Joël	Spectroscopie moléculaire
Mme SCHWARZBACH Yvette	Géométrie
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
M. SLIWA Henri	Chimie organique
M. SOMME Jean	Géographie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. THIEBAULT François	Sciences de la Terre
M. THOMAS Jean Claude	Géométrie - Topologie
M. THUMERELLE Pierre	Démographie - Géographie humaine
M. TILLIEU Jacques	Physique théorique
M. TOULOTTE Jean Marc	Automatique
M. TREANTON Jean René	Sociologie du travail
M. TURRELL Georges	Spectrochimie infrarouge et raman
M. VANEECLOO Nicolas	Sciences Economiques
M. VAST Pierre	Chimie inorganique
M. VERBERT André	Biochimie
M. VERNET Philippe	Génétique
M. VIDAL Pierre	Automatique
M. WALLART Françis	Spectrochimie infrarouge et raman
M. WEINSTEIN Olivier	Analyse économique de la recherche et développement
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ABRAHAM Francis	
M. ALLAMANDO Etienne	Composants électroniques
M. ANDRIES Jean Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BALL Steven	Génétique
M. BART André	Biologie animale
M. BASSERY Louis	Génie des procédés et réactions chimiques
Mme BATTIAU Yvonne	Géographie
M. BAUSIERE Robert	Systèmes électroniques
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique atomique et moléculaire
M. BERNAGE Pascal	Physique atomique, moléculaire et du rayonnement
M. BERTHOUD Arnaud	Sciences Economiques
M. BERTRAND Hugues	Sciences Economiques
M. BERZIN Robert	Analyse
M. BISKUPSKI Gérard	Physique de l'état condensé et cristallographie
M. BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie végétale
M. BOHIN Jean Pierre	Biochimie métabolique et cellulaire
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie civil
M. BOIVIN Jean Claude	Spectrochimie
M. BOUCHER Daniel	Physique
M. BOUQUELET Stéphane	Biologie appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BROCARD Jacques	Chimie
Mme BROUSMICHE Claudine	Paléontologie
M. BUISINE Daniel	Mécanique
M. CAPURON Alfred	Biologie animale
M. CARRE François	Géographie humaine
M. CATTEAU Jean Pierre	Chimie organique
M. CAYATTE Jean Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie structurale
M. CHIVE Maurice	Composants électroniques optiques
M. COMYN Gérard	Informatique théorique
Mme CONSTANT Monique	Composants électroniques et optiques
M. COQUERY Jean Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique nucléaire et corpusculaire
M. COUTURIER Daniel	Chimie organique
M. CRAMPON Norbert	Tectolique géodynamique
M. CURGY Jean Jacques	Biologie
M. DANGOISSE Didier	Physique théorique
M. DE PARIS Jean Claude	Analyse
M. DECOSTER Didier	Composants électroniques et optiques
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean Paul	Informatique
M. DELORME Pierre	Physiologie animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
Mme DEMUYNCK Claire	Physique atomique, moléculaire et du rayonnement
M. DENEL Jacques	Informatique
M. DEPREZ Gilbert	Physique du solide - christallographie

M. DERIEUX Jean Claude	Microbiologie
M. DERYCKE Alain	Informatique
M. DESCAMPS Marc	Physique de l'état condensé et cristallographie
M. DEVRAINNE Pierre	Chimie minérale
M. DEWAILLY Jean Michel	Géographie humaine
M. DHAMELINCOURT Paul	Chimie physique
M. DI PERSIO Jean	Physique de l'état condensé et cristallographie
M. DUBAR Claude	Sociologie démographique
M. DUBOIS Henri	Spectroscopie hertzienne
M. DUBOIS Jean Jacques	Géographie
M. DUBUS Jean Paul	Spectrométrie des solides
M. DUPONT Christophe	Vie de la firme
M. DUTHOIT Bruno	Génie civil
Mme DUVAL Anne	Algèbre
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FARVACQUE Jean Louis	Physique de l'état condensé et cristallographie
M. FAUQUEMBERGUE Renaud	Composants électroniques
M. FELIX Yves	Mathématiques
M. FERRIERE Jacky	Tectonique - Géodynamique
M. FISCHER Jean Claude	Chimie organique, minérale et analytique
M. FONTAINE Hubert	Dynamique des cristaux
M. FORSE Michel	Sociologie
M. GADREY Jean	Sciences économiques
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GOBLOT Rémi	Algèbre
M. GOURIEROUX Christian	Probabilités et statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie organique
M. GUELTON Michel	Chimie physique
M. GUICHAOUA André	Sociologie
M. HAIMAN Georges	Modélisation,calcul scientifique, statistiques
M. HOUDART René	Physique atomique
M. HUEBSCHMANN Johannes	Mathématiques
M. HUTTNER Marc	Algèbre
M. ISAERT Noël	Physique de l'état condensé et cristallographie
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et statistiques
M. JEAN Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KOENIG Gérard	Sciences de gestion
M. KOSTRUBIEC Benjamin	Géographie
M. KREMBEL Jean	Biochimie
Mme KRIFA Hadjila	Sciences Economiques
M. LANGEVIN Michel	Algèbre
M. LASSALLE Bernard	Embryologie et biologie de la différenciation
M. LE MEHAUTE Alain	Modélisation,calcul scientifique,statistiques
M. LEBFEVRE Yannic	Physique atomique,moléculaire et du rayonnement
M. LECLERCQ Lucien	Chimie physique
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Marc	Composants électroniques et optiques
M. LEFEVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
M. LEGRAND Michel	Astronomie - Météorologie
M. LEGRAND Pierre	Chimie
Mme LEGRAND Solange	Algèbre
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne

M. LE MAROIS Henri	Vie de la firme
M. LEMOINE Yves	Biologie et physiologie végétales
M. LESCURE François	Algèbre
M. LESENNE Jacques	Systèmes électroniques
M. LOCQUENEUX Robert	Physique théorique
Mme LOPES Maria	Mathématiques
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU François	Sciences économiques
M. MAHIEU Jean Marie	Optique - Physique atomique
M. MAIZIERES Christian	Automatique
M. MANSY Jean Louis	Géologie
M. MAURISSON Patrick	Sciences Economiques
M. MERIAUX Michel	EUDIL
M. MERLIN Jean Claude	Chimie
M. MESMACQUE Gérard	Génie mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MOCHE Raymond	Modélisation,calcul scientifique,statistiques
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie organique
M. MORE Marcel	Physique de l'état condensé et cristallographie
M. MORTREUX André	Chimie organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
M. NIAY Pierre	Physique atomique,moléculaire et du rayonnement
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PALAVIT Gérard	Génie chimique
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. PERTUZON Emile	Physiologie animale
M. PETIT Daniel	Biologie des populations et écosystèmes
M. PLIHON Dominique	Sciences Economiques
M. PONSOLLE Louis	Chimie physique
M. POSTAIRE Jack	Informatique industrielle
M. RAMBOUR Serge	Biologie
M. RENARD Jean Pierre	Géographie humaine
M. RENARD Philippe	Sciences de gestion
M. RICHARD Alain	Biologie animale
M. RIETSCH François	Physique des polymères
M. ROBINET Jean Claude	EUDIL
M. ROGALSKI Marc	Analyse
M. ROLLAND Paul	Composants électroniques et optiques
M. ROLLET Philippe	Sciences Economiques
Mme ROUSSEL Isabelle	Géographie physique
M. ROUSSIGNOL Michel	Modélisation,calcul scientifique,statistiques
M. ROY Jean Claude	Psychophysiologie
M. SALERNO Françis	Sciences de gestion
M. SANCHOLLE Michel	Biologie et physiologie végétales
Mme SANDIG Anna Margarette	
M. SAWERYSYN Jean Pierre	Chimie physique
M. STAROSWIECKI Marcel	Informatique
M. STEEN Jean Pierre	Informatique
Mme STELLMACHER Irène	Astronomie - Météorologie
M. STERBOUL François	Informatique
M. TAILLIEZ Roger	Génie alimentaire
M. TANRE Daniel	Géométrie - Topologie
M. THERY Pierre	Systèmes électroniques
Mme TJOTTA Jacqueline	Mathématiques
M. TOURSEL Bernard	Informatique
M. TREANTON Jean René	Sociologie du travail

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

Remerciements

Je tiens à remercier les membres du jury :

Jean-Marc GEIB pour m'avoir fait l'honneur de présider ce jury.

Paul FEAUTRIER pour avoir accepté de rapporter cette thèse malgré son emploi du temps chargé. Ses nombreuses remarques sur les algorithmes de redistribution et sur le système de fichiers ont permis de clarifier certains points.

Bernard TOURANCHEAU de l'intéret qu'il a manifesté pour mes travaux. Ses nombreuses annotations et remarques bibliographiques ont permis d'enrichir le document final.

Ce travail de thèse constitue un investissement personnel important. Il a été facilité par le soutient de nombreuses personnes. Je remercie particulièrement :

Mon directeur de thèse Jean-Luc Dekeyser qui m'a proposé un sujet et a dirigé mes travaux. Ses remarques objectives parfois incisives mais toujours justifiées ont permis la rédaction de ce document.

Philippe Marquet qui a co-encadré ce travail. Sa parfaite connaissance du système Unix et sa «culture parallèle» m'ont permis de finaliser cette thèse. Je le remercie également pour son exceptionnel `.emacs` et pour les relectures attentives et tatillonne de tous mes travaux.

Je tiens également à remercier tous les membres du bureau 326 : Boris, Christian, Julien, Jean-Luc et Xavier qui ont solutionné une multitude de problèmes techniques. Leur disponibilité permanente et leur bonne humeur habituelle m'ont beaucoup apporté.

Enfin, je tiens à remercier tous les membres du LIFL qui contribuent à entretenir au sein du laboratoire une ambiance chaleureuse et passionnée.

Table des matières

Introduction

De nombreuses applications de physique, de chimie, de biologie mais aussi de médecine nécessitent des puissances de calculs très supérieures à la capacité de traitement d'un processeur. Les besoins de certaines applications sont si importants qu'il faudrait une puissance de calcul de l'ordre du (Tflops) pour les résoudre en des temps raisonnables [55]. Ces applications sont répertoriées comme «Grand Challenge Applications» [1].

Pour répondre aux besoins sans cesse croissants des utilisateurs il a fallu développer des machines capables d'exécuter simultanément plusieurs instructions. Traditionnellement les machines parallèles étaient développées à partir de composants spécifiques très performants (mémoires rapides, processeurs vectoriels...). La tendance actuelle favorise la réutilisation des composants standards. Les machines distribuées comme les fermes de processeurs ALPHA ou la IBM SP2 sont de simples stations de travail reliées à un réseau. Le Téra flops est aujourd'hui à la portée de ces machines distribuées.

Malheureusement les supports logiciels sont très en retard sur les avancées matérielles. La programmation des machines parallèles reste complexe. Pour tirer pleinement profit de sa machine le programmeur doit souvent prendre en compte les détails d'implémentation bas niveau. De ce fait, les codes développés sont peu portables. Pour faciliter le calcul sur machines parallèles des modèles de programmation ont été développés. Le modèle de programmation à parallélisme de données est très intéressant car il conserve un unique flot d'instructions. Ce modèle consiste à appliquer la même instruction à un ensemble homogène de données. C'est un modèle «à grain fin» bien adapté aux algorithmes de calcul matriciel, d'éléments finis, de traitements d'images.... Les langages à parallélisme de données et les propriétés dont ils bénéficient (notamment le déterministe), facilitent la tâche du programmeur en lui permettant d'effectuer des traitements parallèles sur de grands ensembles de données.

Le programmeur reste cependant confronté au problème des entrées/sorties. Ce problème a longtemps été négligé. En conséquence il existe une grande disparité entre la capacité de

calcul et la capacité d'entrées/sorties de la plupart des systèmes. Dans de nombreux cas le temps pour résoudre un problème dépend de la rapidité des entrées/sorties. Or les applications les plus complexes en terme de puissance de calcul sont également celles qui manipulent le plus de données. Un modèle de circulation générale pour l'atmosphère et les océans génère par exemple trente huit Méga octets de données par minute sur une machine Intel Touchstone Delta [21].

Sur les machines distribuées telles que les fermes de processeurs, les entrées/sorties sont en générale très pénalisantes par manque de support logiciel adapté. Le programmeur doit alors acheminer les données des disques aux processeurs. Le développement est complexe et les codes sont peu portables. Pour tirer profit des multiples périphériques généralement disponibles sur les machines distribuées et simplifier la tâche des programmeurs nous proposons un système de fichiers adapté au modèle de programmation à parallélisme de données. Notre environnement intègre notamment la notion de périphérique virtuel distribué que nous appelons DpoDevice. Chaque DpoDevice est défini par une grille de nœuds d'entrées/sorties et par une fonction de distribution semblable à celles utilisées par le langage HPF.

Les DpoDevices sont associés aux répertoires. Tous les DpoFiles d'un même répertoire sont donc distribués sur les même nœuds d'entrées/sorties, avec la même fonction de distribution. Dans notre système de fichier, la fonction de distribution n'est pas associée aux fichiers parallèles. Le programmeur accède à un fichier logique. Il n'a pas à se soucier de la distribution des données sur les nœuds d'entrée/sortie. Les fichiers sont toujours accédés de la même façon, qu'ils soient placés sur un périphérique séquentiel standard ou distribués sur une grille multidimentionelle. Cette souplesse d'utilisation n'a pas été obtenue au détriment des performances. Pour éviter les mouvements de données inutiles, l'utilisateur peut, à l'exécution choisir le répertoire et donc la distribution de données qui convient la mieux à son application.

Dans le premier chapitre nous introduirons une classification des différents modèles de fichiers parallèles existants. Cette classification est basée sur l'expression du parallélisme. Nous montrerons qu'il est particulièrement intéressant dans le cadre des langages à parallélisme de données de conserver la notion d'objet structuré que l'on partitionne sur un ensemble de nœuds. Avec les modèles à objets structurés partitionnés chaque opération d'accès nécessite une redistribution pour projeter les données des nœuds de calculs aux nœuds d'entrées/sorties (et inversement).

Le second chapitre est une étude des algorithmes de redistributions existants. La plupart des algorithmes de redistributions sont conçus pour des compilateurs. Ils redistribuent les

données sur un unique ensemble de processeurs et ont besoin de connaître les paramètres de distribution à la compilation. Les entrées/sorties parallèles font interagir deux ensembles de processeurs, il y a donc deux niveaux de parallélisme potentiel. Par ailleurs les paramètres de redistribution ne sont connus qu'à l'exécution. Pour tirer profit de toutes les ressources disponibles il faut concevoir des algorithmes de redistributions spécifiques.

Le chapitre trois est le point central de cette thèse. De façon générale, une opération de redistribution peut être modélisée à l'aide d'une équation diophantienne à six inconnues. L'étude de redistributions particulières va nous permettre de simplifier ce modèle général puis de construire des algorithmes spécialisés. Ces algorithmes minimisent le volume et le nombre de messages et recouvrent les calculs avec les communications. Pour tirer profit de toute les ressources, il faut répartir au mieux les messages sur l'ensemble des destinataires. Les redistributions qui génèrent de nombreux conflits ont été identifiées. Pour ces cas précis, nous avons calculé un ordonnancement optimal des messages.

Dans le dernier chapitre nous détaillerons le système de fichiers ainsi que la bibliothèque d'entrées/sorties qui ont été développés. Le système de fichiers offre au programmeur et à l'utilisateur un niveau d'abstraction équivalent au système de fichier Unix. Les commandes standards telles que «mv», «cd», ou «cp» s'utilisent avec les paramètres habituels. D'un point de vue logique, rien n'a changé mais les attributs qui sont associés au niveau des DpoDevices sont utilisés par les commandes pour effectuer des traitements supplémentaires. La commande «mv» migre les fichiers d'une machine parallèle à une autre en changeant (si nécessaire) la représentation des données. Elle redistribue les données d'une topologie source à une topologie cible en fonction des fonctions de distributions associées aux périphériques. L'organisation des données sur les topologies d'entrées/sorties est gérée dynamiquement par le système.

Nous résumerons ensuite notre approche et les extensions que nous envisageons dans le futur.

Chapitre 1

Entrées/sorties parallèles

Les avancées dans le domaine des semi-conducteurs ont permis d'obtenir des machines de plus en plus rapides. Les supers ordinateurs sont aujourd'hui capables d'effectuer plusieurs milliards d'opérations virgules flottantes par seconde. Pour tirer parti de la puissance de calcul de ces machines, des modèles de programmation ont été développés. Ceux-ci s'intéressent aux opérations de calculs mais rarement aux opérations d'entrées/sorties. Par manque de modèle adapté aux entrées/sorties parallèles, de nombreuses applications parallèles utilisent le modèle de fichiers Unix qui demeure un des seuls standards (le seul?).

Les fichiers Unix sont constitués de suites linéaires d'octets que l'on accède par l'intermédiaire d'un pointeur scalaire. L'espace d'adressage linéaire que constitue le fichier est très proche de l'espace mémoire qu'utilisent les programmes séquentiels. Cela facilite beaucoup le passage de «l'espace mémoire» à «l'espace fichier». Il est par exemple possible d'accéder à un tableau de taille quelconque par une simple opération de lecture/écriture. L'expressivité des fonctions d'accès Unix repose en grande partie sur les similitudes qui existent entre la mémoire et le fichier.

Sur les machines à mémoires distribuées, les données sont réparties sur les processeurs. La figure (1.1) représente un tableau distribué cycliquement sur trois processeurs et les données correspondantes dans un fichier Unix. On remarque notamment que les données placées sur un processeur ne sont pas contiguës dans le fichier séquentiel. Il n'y a plus de correspondance simple entre l'espace mémoire et l'espace fichier. Pour accéder à ses données chaque processeur doit effectuer une succession de déplacements et de lectures. Le nombre d'appels nécessaires est proportionnel à la taille des données. Cet exemple montre que le modèle de fichiers Unix n'est pas adapté aux machines à mémoires distribuées. Les applications parallèles qui utilisent

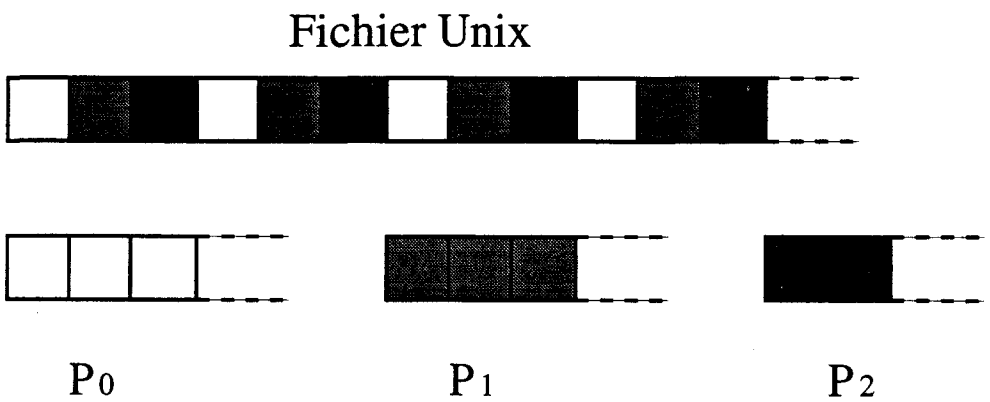


FIG. 1.1 - *Tableau distribué cycliquement sur trois processeurs*

ce modèle sont fortement pénalisées.

Pour éviter le goulot d'étranglement que représente les accès séquentiels il faut proposer au programmeur un modèle d'entrées/sorties adapté à son environnement de programmation. Dans ce chapitre nous proposerons une classification des modèles de fichiers qui ont été proposés. Nous distinguerons notamment les modèles adaptés au parallélisme de données des modèles adaptés au parallélisme de tâches. Comme les performances sont la raison d'être des environnements parallèles nous évoquerons dans une seconde partie les problèmes de mise en œuvre que posent les différents modèles. Dans la dernière partie de ce chapitre nous détaillerons plusieurs projets avec lesquels nous illustrerons la classification précédemment introduite.

1 Modèles adaptés au parallélisme de données

Le parallélisme de données consiste à appliquer une même opération à un ensemble homogène de données. Les données sont regroupées en variables parallèles structurées (ex : tableaux HPF) sur lesquelles on applique des traitements. Le parallélisme est obtenu en partitionnant les données sur les processeurs.

Un modèle de fichier est adapté au parallélisme de données s'il supporte le partitionnement des données sur les processeurs. Le partitionnement peut-être explicité au niveau même du fichier, nous parlerons alors de **fichiers partitionnés**. Il peut également être explicité au niveau des objets qui composent le fichier, nous parlerons alors de **fichiers à objets partitionnés**. Ces deux modèles de fichiers seront qualifiés de **déterministes** car il est toujours

possible de caractériser l'état du fichier indépendamment de l'ordre d'exécution des accès parallèles.

1.1 Modèles à fichiers partitionnés

La plupart des langages à parallélisme de donnée utilisent la notion de **machine virtuelle** de façon à s'affranchir de la taille de la machine parallèle physique. Le programmeur peut ainsi considérer qu'il existe un processeur par élément quelque soit la taille de ses données. Dans sa classification des langages à parallélisme de données, Dominique Lazure distingue les **machines virtuelles d'alignement** et les **machines virtuelles d'instanciation** [34].

Les machines d'alignement ne sont utilisées que pour aligner les données indépendamment de la taille des objets parallèles. La machine virtuelle est alors un référentiel qui n'est utilisé que pour exprimer l'alignement des objets (ex: template HPF). La notion d'objet structuré (en général les tableaux) est alors fondamentale car c'est sur elle que repose le parallélisme.

Les machines d'instanciation sont basées sur la notion de processeur virtuel. Chaque processeur virtuel dispose de ses propres instances de données et l'activité des processeurs conditionne les traitements qui sont effectués. Les variables parallèles déclarées sur les machines virtuelles d'instanciation héritent de la géométrie de la machine sur laquelle elles ont été déclarées. Les machines d'instanciation regroupent donc des objets de mêmes caractéristiques[1].

1.1.1 Un modèle pour les machines virtuelles d'instanciation

Un **fichier partitionné** est déclaré sur une machine virtuelle d'instanciation. C'est une variable parallèle comme les autres qui possède la géométrie de la machine sur laquelle elle a été instanciée.

Chaque processeur virtuel possède sa propre instance de fichier que nous appellerons **fichier séquentiel microscopique.** Les fichiers microscopiques que possède chaque processeur sont disjoints. Une donnée est accessible par exactement un processeur. Ce modèle est donc déterministe. Chaque processeur accède à son propre fichier microscopique à l'aide d'un pointeur de fichier local. Toutes les fonctions d'accès sont exécutées en parallèle sur tous les processeurs virtuels actifs mais la taille des accès varie d'un processeur à l'autre. Comme tous les fichiers microscopiques n'ont pas tous la même taille, la fin de fichier est définie au niveau

1. objets de même géométrie

des fichiers microscopiques et non pas au niveau du fichier global.

Les opérations d'ouverture et de fermeture sont les seules opérations relatives au fichier macroscopique. La gestion de ce fichier global est laissée à la charge du programmeur. C'est par exemple lui qui doit effectuer les réductions pour vérifier qu'une opération d'entrée/sortie s'est déroulée correctement.

1.1.2 Un modèle simple et puissant mais difficile à utiliser

Le modèle à fichier partitionné est très satisfaisant au niveau conceptuel car la notion de processeur virtuel est utilisée à la fois pour les calculs et pour les entrées/sorties.

Ce modèle à fichiers partagés est extrêmement puissant, il permet une gestion très fine du degré de parallélisme (le parallélisme ne dépend que de l'activité des processeurs). Les fichiers microscopiques sont complètement indépendants. Un processeur peut par exemple écrire dans son fichier local toutes les données qu'il possède et qui vérifient un critère particulier. Le programmeur n'a pas à connaître le nombre de données accédées sur chaque processeur virtuel. La gestion des fichiers microscopiques est assurée par le système.

Le modèle à fichier partitionné est conçu pour les machines virtuelles d'instanciations et n'est utilisable qu'avec ce modèle de programmation. Pire, un fichier qui a été créé par une machine virtuelle ne peut-être accédé que par une machine virtuelle identique[2]. Les fichiers partitionnés ont de ce fait un spectre d'application réduit.

Par ailleurs les fichiers partitionnés sont des objets relativement complexes. Il y a un flot de données par processeur virtuel. Un fichier partitionné intègre tous les accès qui ont été effectués par l'ensemble des processeurs de la machine virtuelle. Les fichiers partitionnés ont en plus de la dimension spatiale qui est liée à la géométrie de la machine utilisée, une dimension temporelle qu'ils héritent des fichiers traditionnels. Il est très difficile d'utiliser ce type de fichier sans savoir précisément comment ils ont été générés et ce que représentent les informations disséminées dans les milliers de fichiers microscopiques.

Le modèle conserve au niveau microscopique la sémantique des fichiers séquentiels mais le «passage» au modèle partitionné n'est pas pour autant facile. L'ajout de la dimension temporelle va nécessiter de nouvelles habitudes de programmation. Les fichiers partitionnés seront illustrés dans la seconde partie de ce chapitre par l'étude du modèle Stream* [35, 36,

2. machine qui a la même topologie

6, 28].

1.2 Modèles à objets partitionnés

Dans le paragraphe précédent, nous avons décrit un modèle dans lequel un fichier parallèle est partitionné en un ensemble de fichiers microscopiques. Ce modèle à fichier partitionné est intéressant mais il nécessite de nouvelles habitudes de programmation. Nous allons maintenant nous intéresser à un modèle plus proche du standard Unix dans lequel on ne partitionne plus le fichier mais les objets qui le composent.

Les langages à parallélisme de données utilisent la notion d'objet structuré: ce sont les tableaux HPF ou F90, les shapes HyperC ou les collections C^*. Avec des modèles de fichiers de bas niveau tel que le modèle Unix, le programmeur doit projeter ses objets multidimentionnels partitionnés sur les processeurs dans un espace unidimensionnel. Les objets ont alors deux représentations complètement distinctes et le passage de l'une à l'autre est à la charge du programmeur. Le coût de développement est important et le code peu portable.

Le modèle de fichier à objets structurés partitionnés répond aux besoins de ce type d'environnement. Un fichier à objets structurés partitionnés est une suite linéaire d'objets structurés que l'on accède par l'intermédiaire d'un pointeur scalaire. Ces objets structurés sont partitionnés sur l'espace des processeurs à l'aide d'une fonction de distribution. La fonction de distribution définit la visibilité des processeurs sur le fichier. Un processeur ne peut accéder qu'aux données qui lui sont attribuées par la fonction de distribution. Comme la fonction de distribution définit un partitionnement, chaque donnée est accessible par exactement un processeur. Ce modèle de fichier est donc déterministe.

1.2.1 Facilité d'utilisation

Conceptuellement ces fichiers à objets partagés sont très proches des fichiers Unix. Les objets qui composent les fichiers sont structurés mais pas nécessairement distribués. Il est tout à fait possible de lire ou d'écrire des fichiers à objets structurés à l'aide des bibliothèques d'entrées/sorties séquentielles standard.

1.2.2 Simplicité

Ce modèle permet de conserver au niveau du système de fichier, la notion d'objet parallèle qui est utilisée dans tous les langages à parallélisme de données. Le programmeur n'a pas à linéariser ses objets lors de chaque accès. Les entrées/sorties ne «dénaturent» plus les objets parallèles. Il est plus facile de lire ou d'écriture une image stockée dans une variable parallèle que d'accéder aux $1024 * 1024 * 24$ octets qui la représentent !

2 Modèles adaptés au parallélisme de tâches

Les modèles à fichiers ou à objets partitionnés supportent la notion de variable parallèle (fichier ou objet), ils sont de ce fait bien adaptés aux langages à parallélisme de données. Nous allons maintenant nous intéresser aux modèles adaptés au parallélisme de tâches. À la différence du modèle de programmation à parallélisme de données, le modèle de programmation à parallélisme de tâches utilise plusieurs flots d'instructions. On ne distribue plus les données mais les travaux à exécuter. Les **modèles de fichiers virtuellement partagés** dans lesquels une même donnée peut-être accédée par plusieurs processeurs sont bien adaptés à ce modèle de programmation.

Nous distinguerons les modèles à **référentiels relatifs** dans lesquels chaque tâche définit «sa propre vision du fichier», des modèles à **référentiel global**.

2.1 Modèles à référentiels relatifs

Un fichier virtuellement partagé est une suite séquentielle d'octets que les tâches accèdent par l'intermédiaire d'un pointeur de fichier local. Le fichier constitue un **référentiel global** identique pour toutes les tâches dans lequel chaque objet est identifié par un déplacement absolu (fig. 1.2). En utilisant le référentiel global, chaque tâche peut construire **un référentiel relatif** de façon à définir son propre **espace d'adressage.** Les déplacements ne sont plus spécifiés dans le fichier global mais dans l'espace d'adressage local et en fonction des éléments accessibles (fig. 1.2).

Les espaces d'adressages multiples facilitent l'accès aux données non contiguës. Les données (fig. 1.2) dont les adresses absolues sont 2, 6, 7, et 8 dans le fichier global peuvent par exemple être accédées à l'aide d'une unique opération d'entrée/sortie. L'accès à ces données

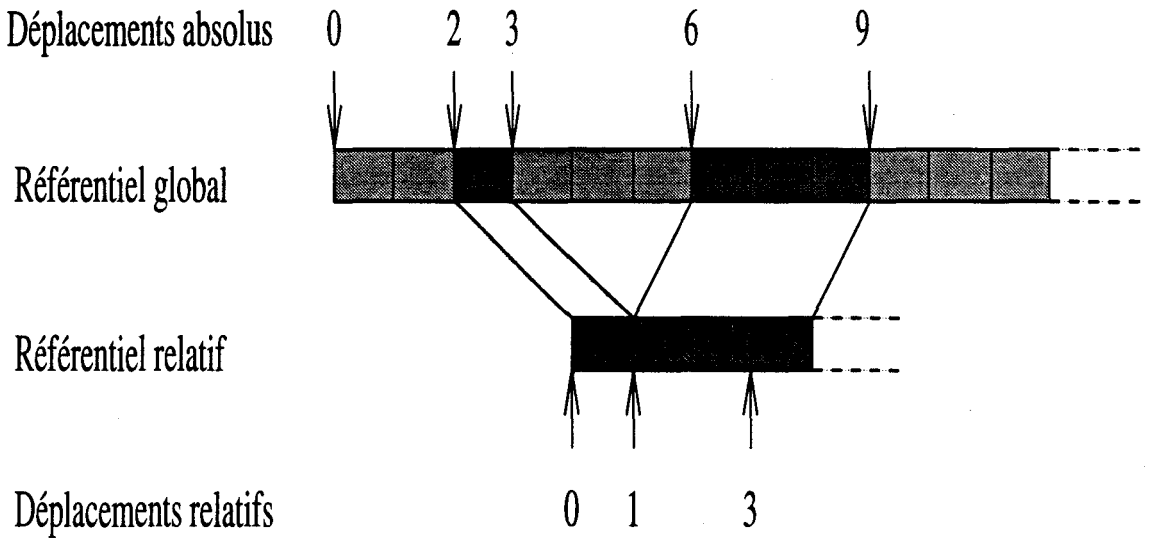


FIG. 1.2 - *Exemple de référentiel*

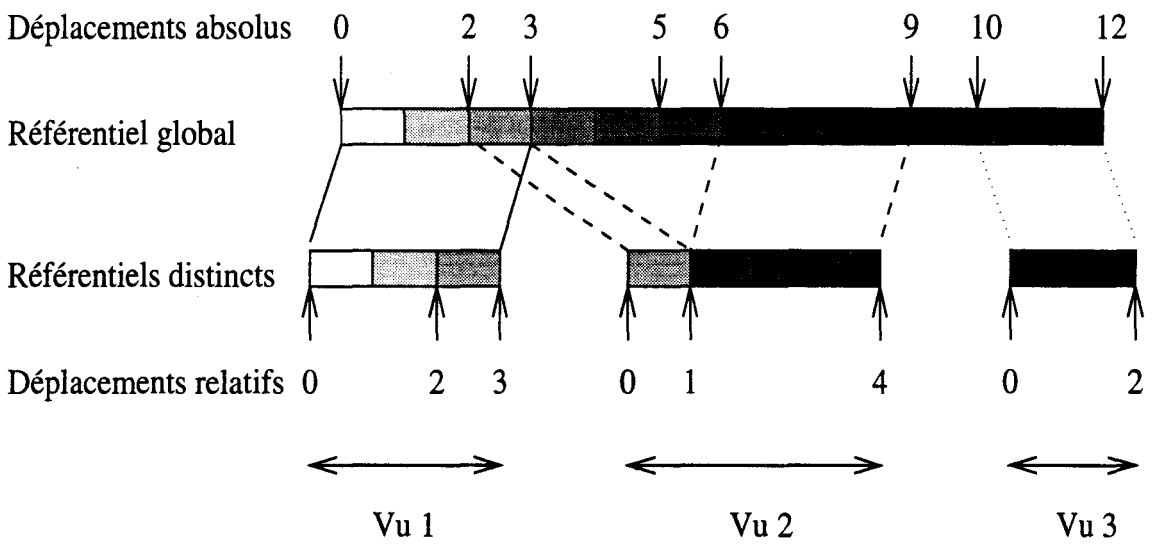


FIG. 1.3 - *Référentiels distincts*

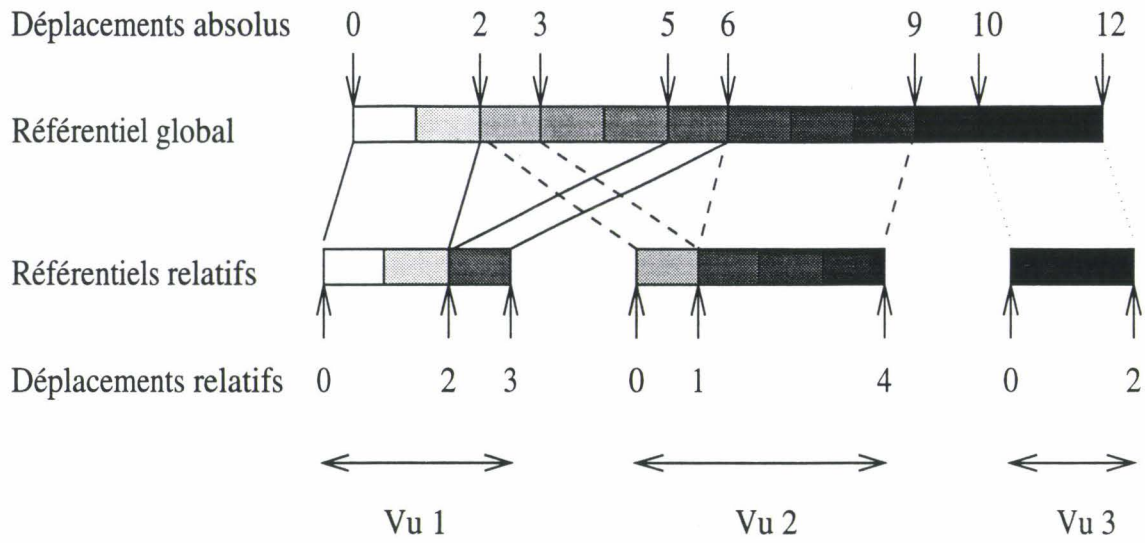


FIG. 1.4 - *Référentiels disjoints*

aurait nécessité plusieurs appels dans un modèle à référentiel unique. La possibilité de définir de multiples espaces d'adressages accroît donc l'expressivité des fonctions d'accès. Nous distinguerons les modèles à référentiels distincts des modèles à référentiels disjoints.

1. **Référentiels distincts (fig. 1.3)**

   Avec ce modèle de fichier une donnée est accessible par un nombre quelconque de tâches. Ce modèle n'est pas déterministe. Il n'est pas possible de caractériser l'état du fichier. Que se passe t-il si deux tâches accèdent en écriture à la même section de fichier?

2. **Référentiels disjoints (fig. 1.4)**

   Ce modèle de fichier est un cas particulier du cas précédent dans lequel les ensembles de données accessibles par chacune des tâches sont disjoints. Cela permet de garantir le déterminisme.

2.2 Remarque

Avec l'approche à «référentiel multiple», le parallélisme est explicité au niveau de l'application. Le programmeur doit définir les ensembles de données accessibles pour chacune de ses tâches. C'est également à lui de gérer les éventuels conflits d'accès. Les fichiers n'ont aucun attribut parallèle[3], ils sont simplement accédés en parallèle.

3. En général ce sont simplement des suites d'octets comme les fichiers Unix

Il est possible de créer des référentiels disjoints dans la plupart des systèmes à espaces d'adressages multiples mais il n'est en général pas possible d'expliciter cette propriété. C'est par exemple le cas avec les bibliothèques PIOUS et MPI-IO qui seront présentées dans la seconde partie de ce chapitre.

2.3 Modèles à référentiel global

Nous allons maintenant nous intéresser aux systèmes qui conservent un unique espace d'adressage et qui explicitent le parallélisme à l'aide des pointeurs de fichiers.

2.3.1 Modèles à pointeurs multiples

Avec les systèmes de fichiers adaptés au modèle à parallélisme de tâches, les entrées/sorties permettent d'accéder en parallèle à des objets scalaires. L'approche à référentiels relatifs permet de définir une «vue locale» du fichier parallèle pour chacune des tâches. Cette approche est intéressante lorsque les tâches accèdent à des sous ensembles du fichier parallèle. Dans de nombreux cas on ne connaît pas *a priori* les ensembles de données qui seront accédés par une tâche ; on est alors contraint de conserver un unique espace d'adressage. Comme les tâches n'accédent pas toutes aux mêmes données en même temps elles utilisent chacune leur propre pointeur de fichier.

Toutes les tâches ont accès à l'ensemble des données du fichier. Du point de vue du système, il n'y a pas d'accès parallèles émanant d'une unique application mais des ensembles d'accès séquentiels réalisés par des tâches indépendantes. Rien ne distingue les accès réalisés par les tâches d'un unique programme parallèle, des accès effectués par des processus indépendants. Le programmeur doit calculer les déplacements dans un espace d'adressage global et gérer les éventuels conflits d'accès.

Ce modèle sera illustré avec la bibliothèque PIOUS [38, 37, 40, 39] mais le représentant le plus illustre de cette catégorie reste le modèle de fichier Unix.

2.3.2 Modèles à pointeur partagé

Avec les modèles à pointeur partagé, les différentes tâches d'une application parallèle peuvent accéder à un unique espace d'adressage à l'aide d'un unique pointeur partagé. Les

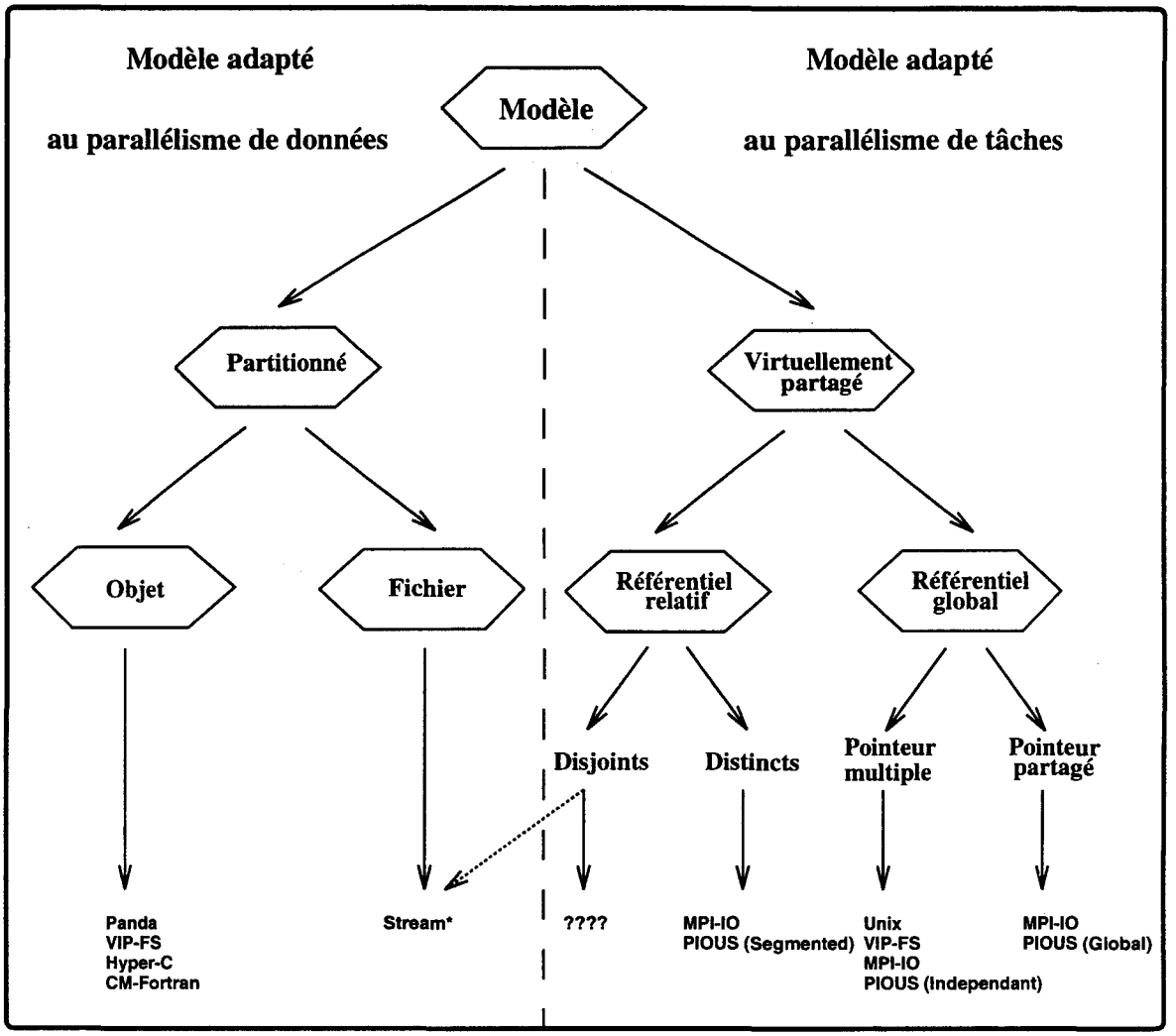


FIG. 1.5 - *Classification des systèmes de fichiers parallèles*

accès sont exclusifs. Deux tâches distinctes ne peuvent accéder en parallèle à un même fichier. Le pointeur partagé est mis à jour après chaque accès et pour toutes les tâches du groupe. Ce groupe est en général spécifié lors de l'ouverture du fichier et reste inchangé jusqu'à sa fermeture. Le système distingue cette fois les accès «groupés» qui émanent d'une application parallèle des accès individuels effectués par un programme séquentiel.

Ce modèle à pointeur partagé est original. Il n'est pas aussi général que les modèles précédents mais il répond aux besoins des applications de type producteur/consommateur. Les accès sont séquentialisés mais ils modifient un pointeur commun. Ce modèle est supporté par les bibliothèques PIOUS et MPI-IO.

3 Classification

La figure (1.5) présente une vue globale des différents modèles présentés. Les modèles de fichiers adaptés au parallélisme de données sont tous déterministes. Dans ces modèles le parallélisme est exprimé au niveau des objets et non pas au niveau des fonctions d'accès. Avec ces modèles le système de fichiers connaît la distribution des données sur les processeurs de calcul. Le programmeur n'a pas à gérer un ensemble d'espaces d'adressages. C'est le système qui projette à chaque accès les données des nœuds de calculs aux nœuds d'entrées/sorties ou des nœuds d'entrées/sorties aux nœuds de calculs.

Dans les modèles adaptés au parallélisme de tâches le parallélisme est explicité au niveau de l'application. Le programmeur peut soit définir les ensembles de données accessibles pour chacune de ses tâches (multiples espaces d'adressages), soit calculer explicitement les déplacements des objets qui seront accédés. Dans tous les cas, le programmeur doit gérer l'organisation physique des données et «faire en sorte» d'éviter les conflits d'accès. Ces modèles ne sont pas déterministes[4]. Le modèle à espaces d'adressages disjoints aurait put être classé avec les modèles partitionnés. Comme le partitionnement reste à la charge de l'utilisateur nous l'avons placé avec les modèles virtuellement partagés.

4 Du modèle à la mise en œuvre

Un modèle de fichier parallèle bien adapté à un modèle de programmation ne sera effectivement utilisé que s'il est possible de concevoir une implantation efficace. Dans la section précédente nous avons étudié puis classé différentes familles de modèles de fichiers parallèles, nous allons maintenant étudier les problèmes de mise en œuvre inhérents à ces modèles.

Les entrées/sorties parallèles font interagir les processeurs de calculs, le réseau de communication et les nœuds d'entrées/sorties (fig. 1.6). Il faut à chaque accès migrer un ensemble de données des processeurs de calcul aux nœuds d'entrées/sorties (ces deux ensembles ne sont pas nécessairement disjoints). Comme la latence du réseau et des disques est généralement très importante par rapport à leur bande passante, les modèles favorisant les accès de grandes tailles seront *a priori* plus efficaces. Par ailleurs, les entrées/sorties ne sont effectivement réalisées en parallèle que si le système dispose d'une fonction de projection avec laquelle il peut répartir équitablement les données sur l'ensemble des nœuds d'entrées/sorties.

4. à l'exception du modèle à espaces d'adressages disjoints

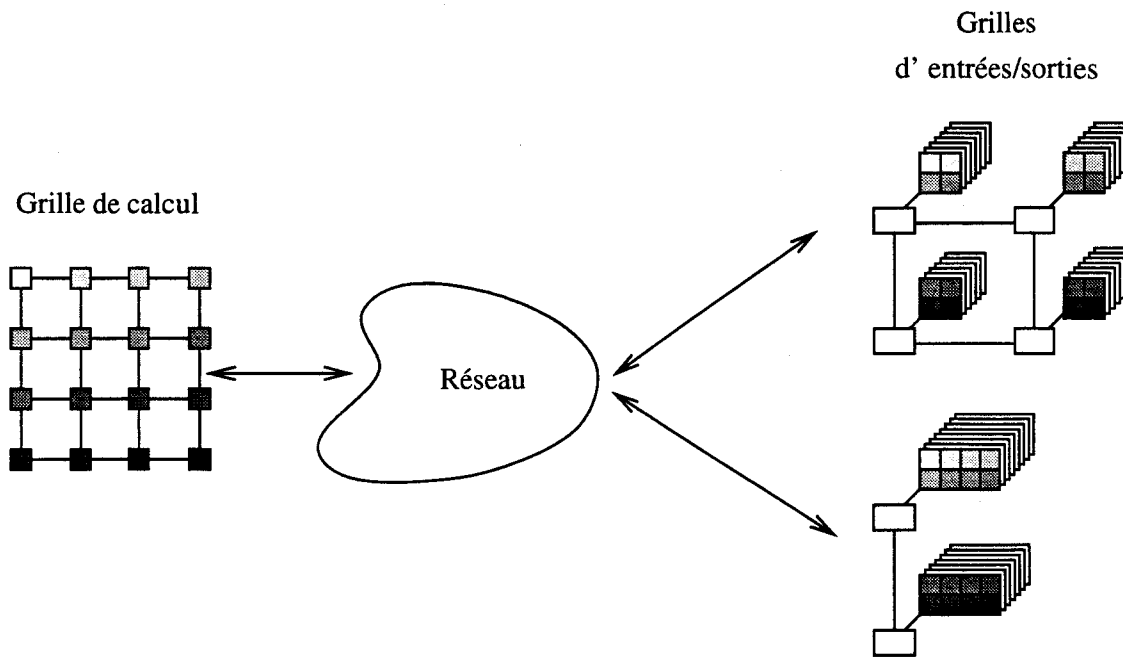


FIG. 1.6 - *Architecture d'un système de fichier parallèle*

4.1 Modèle à fichier partitionné

Avec le modèle à fichier partitionné, chaque processeur virtuel réalise ses propres accès et gère son propre pointeur de fichier. Les volumes de données accédées varient d'un processeur à un autre et le degré de parallélisme évolue constamment[5]. Le schéma d'accès est complètement dynamique. La projection des fichiers microscopiques sur un ensemble de disques est dans ce contexte extrêmement complexe. Il faut équilibrer la charge à la fois dans l'espace (ce qui est déjà un problème compliqué), mais aussi dans le temps !

Par ailleurs les entrées/sorties sont explicitées au niveau des processeurs virtuels. Le modèle à fichier partitionné favorise donc les accès de petites tailles. Ce modèle de fichier semble aussi difficile à programmer qu'à implanter efficacement !

4.2 Modèle à objet partitionné

Avec le modèle à objets partitionnés les accès portent sur des objets de taille importante. Le système dispose d'informations sur la géométrie des objets qui composent le fichier et connaît la fonction de distribution qui est utilisée pour partitionner les données sur les

5. Il ne dépend que de l'activité des processeurs virtuels

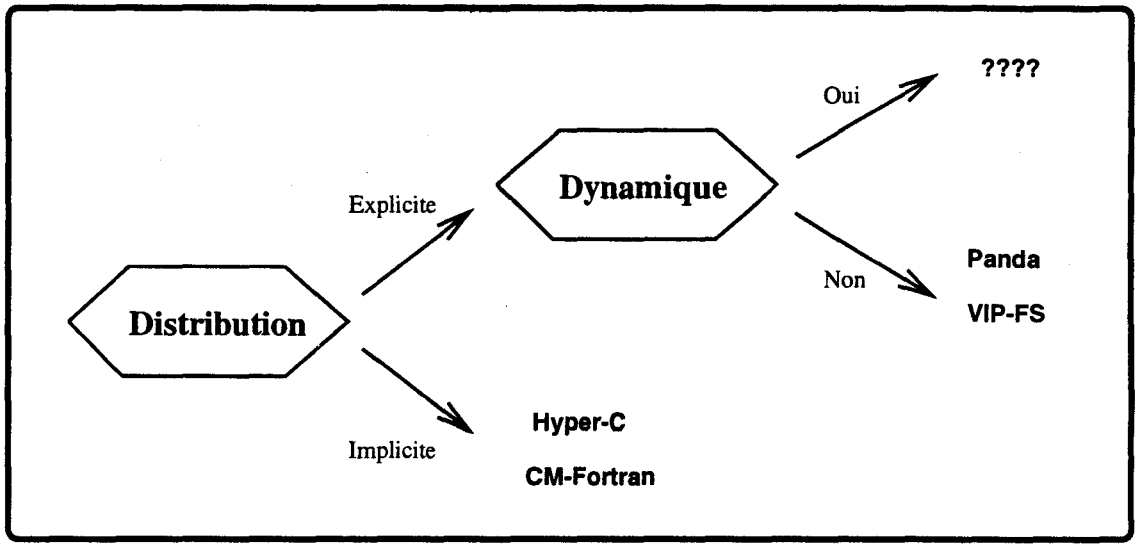


FIG. 1.7 - *Systèmes de fichiers virtuellement partagé avec objets structurés*

processeurs. Ce modèle est beaucoup plus «statique» que le modèle précédent.

Les volumes de données accédées sont à peu près identiques sur tous les processeurs. Cela simplifie énormément le problème de la répartition équitable des données sur les nœuds d'entrées/sorties. En général les objets structurés sont linéarisés puis distribués cycliquement sur un ensemble de disques par blocs de taille identique[6]. C'est ainsi que sont implantés les fichiers de la machine Paragon et tous les systèmes bâtis au dessus de disques RAID.

Pour améliorer la localité des données il peut cependant être intéressant de donner au programmeur ou à l'utilisateur la possibilité de choisir la distribution des données au niveau du système de fichiers. Les différentes interfaces possibles sont présentées figure 1.7. Nous opérons une première distinction entre les **interfaces à distribution implicitement fixée** dans lesquelles la distribution des données est imposée par le système et les **interfaces explicites** qui permettent de choisir la distribution des données dans le système de fichiers. Nous distinguerons ensuite les modèles dans lesquels la distribution est explicitée au niveau du programme[7], des modèles dans lesquels la distribution n'est explicitée qu'au moment de l'exécution. Il convient de noter que ces fonctions de distribution ne modifient pas la sémantique des fonctions d'accès. Le modèle de fichier reste inchangé quelque soit l'interface choisie, seule l'efficacité est affectée.

6. La taille de ces blocs est appelée la «stripping unit»
7. la fonction de distribution est alors disponible à la compilation

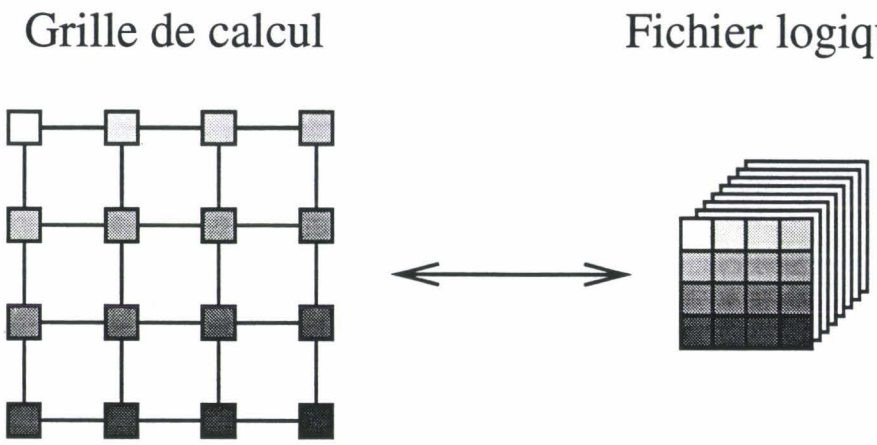


FIG. 1.8 - *Interface implicite : la distribution est cachée*

4.2.1 Interface à distribution implicite fixée

Dans les systèmes de fichiers à interface implicite fixée, l'utilisateur n'a aucun contrôle sur la fonction de distribution qui est utilisée par le système de fichiers. Le programmeur accède à un fichier logique sans savoir où sont placées les données (fig. 1.8). Le système utilise une distribution de données canonique. Les performances obtenues avec ce type d'interface fluctuent énormément car elles dépendent de la distribution qui est utilisée en mémoire : lorsqu'on accède à une ligne d'une matrice qui a été distribuée par colonne, les performances s'effondrent !

4.2.2 Interface à distribution explicite

Il est cette fois possible de choisir la distribution des données au niveau du système de fichiers. La démarche est identique à celle qui est utilisée pour la distribution mémoire dans les langages tels que `HPF`. Les objets structurés sont projetés sur les nœuds d'entrées/sorties comme ils le sont sur les nœuds de calculs. La distribution peut-être explicitée :

À la compilation(1.9) : Le programmeur dispose d'un système de fichiers distribués et doit fixer la distribution pour faire en sorte d'en tirer parti. Le choix d'une fonction de distribution de données «à la compilation» n'est pas toujours facile. Le programmeur doit fixer une distribution même s'il ne dispose pas des informations pertinentes pour effectuer ce choix. Les données qui sont projetées sur un disque ne pourront pas être redirigées dans un «pipe» parallèle. Les données générées sont destinées à un programme

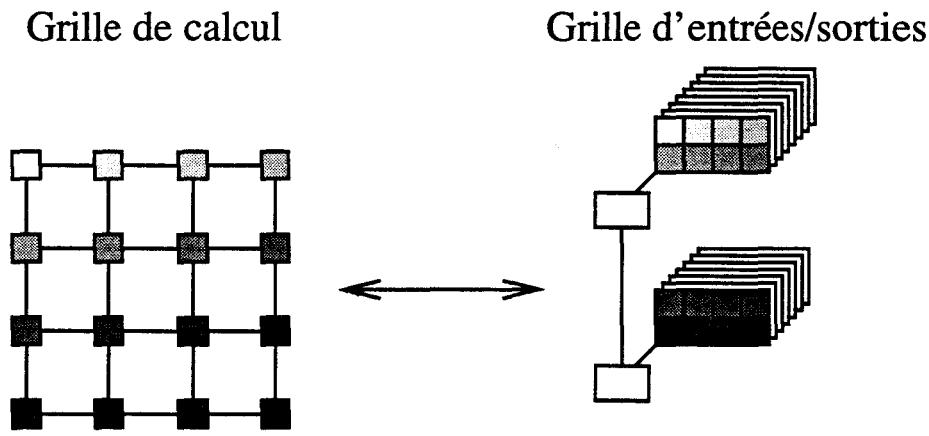


FIG. 1.9 - *Interface explicite : le programmeur impose la distribution*

séquentiel ou à une tâche parallèle. Le «contexte d'exécution» est fixé à la compilation ! Ce modèle limite la réutilisabilité.

Le modèle à objets structurés avec distribution explicite statique sera illustré avec le système PANDA [47, 48]

À l'exécution(1.10) : Avec le système Unix, le programmeur n'a pas à se soucier du périphérique qui sera utilisé lors de l'exécution du programme qu'il développe. Il ne sait pas si les données seront affichées à l'écran, mémorisées sur un disque local ou redirigées dans un «pipe». C'est l'utilisateur qui choisit à l'exécution et en fonction de ses besoins le périphérique qui répond le mieux à ses attentes.

Les systèmes de fichiers structurés dans lesquels la distribution est explicitée à l'exécution conservent ce principe de transparence. Le programmeur accède à un fichier logique. Il ne sait pas comment seront distribuées les données dans le système de fichier. Cette distribution de données est fixée à l'exécution par l'utilisateur. Cette solution est beaucoup plus expressive. Lorsqu'un utilisateur lance un traitement il sait comment seront utilisés les résultats de l'application, il peut choisir une distribution adaptée.

Les systèmes qui utilisent une interface explicite statique sont *a priori* les plus efficaces. Tous les paramètres de distributions sont connus à la compilation. Le compilateur est donc à même d'effectuer des optimisations. Tous les algorithmes de redistribution qui ont été développés dans le cadre du «HPF static subset» sont utilisables.

Malheureusement ce type d'interface limite la réutilisabilité. Le programmeur doit, s'il souhaite migrer des données d'une application à une autre, choisir une distribution canonique.

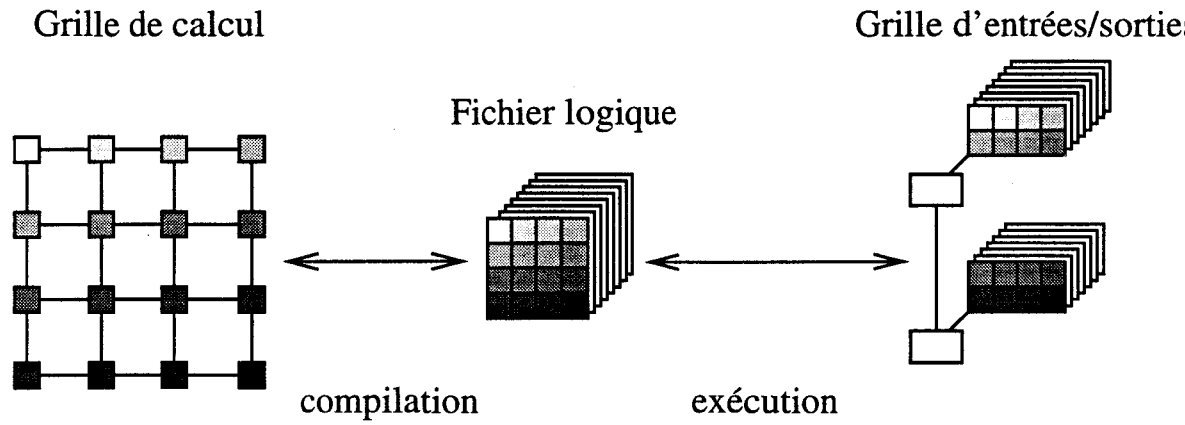


FIG. 1.10 - *Interface explicite à l'exécution : le fichier logique est distribué par l'utilisateur*

On retrouve alors les défauts des interfaces implicites[8] avec en plus une certaine lourdeur syntaxique !

4.3 Modèles à référentiels relatifs

Avec le modèle à «référentiels multiples» le programmeur définit les ensembles de données accessibles pour chacune de ses tâches. Une tâche peut ainsi avec une opération d'entrée/sortie accéder à des données dispersées avec le fichier global. Avec cette approche le système peut accéder les données dans l'ordre qui lui convient le mieux. Il peut par exemple choisir un schéma d'accès qui minimise les déplacements de la tête de lecture [33]. Il peut également choisir de lire un sur-ensemble de données contiguës. C'est ce que les auteurs du projet PASSION appellent le «data sieving» [13].

La projection des données sur les nœuds d'entrées/sorties est un problème complexe. Les référentiels relatifs fournissent des informations sur les données qui seront (peut-être) accédées par les processus. On peut imaginer de construire des fonctions de distribution qui prennent en compte ces référentiels. En général le système choisit une distribution par défaut (PIOUS) ou des directives fournies par le programmeur (les «hints» de MPI-IO).

8. les performances sont très dépendantes de la distribution des données sur les processeurs de calculs

4.3.1 Modèle à référentiel unique

Ce modèle n'est pas particulièrement adapté au parallélisme. Il est néanmoins possible de distribuer les données sur plusieurs disques afin d'obtenir des accès physiques parallèles. Cette distribution est alors arbitraire, elle ne dépend ni des objets (qui ne sont pas structurés) ni d'un quelconque découpage du fichier (il n'y a qu'un seul espace d'adressage). Les données sont généralement distribuées cycliquement de façon à bénéficier au maximum des disques disponibles.

5 Conclusion

Nous avons discerné différents modèles de fichiers puis étudié les problèmes que posent la mise en œuvre de ces modèles. Parmi tous les modèles présentés, le modèle à objets structurés est celui qui fournit le plus d'informations au système. Avec ce modèle, une entrée/sortie parallèle est une «simple» opération de redistribution.

Nous allons maintenant illustrer la classification qui a été précédemment introduite ainsi que les problèmes de mise en œuvre que posent les différents modèles avec les projets Stream*, Panda, PASSION, MPI-IO et PIOUS.

6 Stream*

Stream* est une extension du modèle de programmation C* qui supporte la notion de fichiers à parallélisme de données [35, 36, 6, 28]. D'après la classification des modèles de programmation introduite par Dominique Lazure [34] puis étendue par Jean-Luc Dekeyser et Philippe Marquet [19], C* utilise la notion de machine virtuelle d'instanciation. Pour conserver ce modèle de programmation lors des E/S, Stream* associe un flot de données Unix à chacun des processeurs virtuels (VP). Chaque VP gère son propre fichier à l'aide de fonctions d'entrées/sorties semblables aux fonctions Unix. Le programmeur n'a plus à gérer la projection des flots de données logiques sur un unique fichier séquentiel. Les pointeurs de fichiers associés à chacun des VP sont complètement indépendants. Un processeur Virtuel ne peut accéder qu'à son propre fichier. Les accès aux fichiers ne nécessitent aucune synchronisation.

Avec Stream*, le parallélisme est explicité au niveau même du fichier. Stream* est donc d'après notre classification, un modèle à fichiers partitionnés.

6.1 L'interface

Les fonctions d'entrées/sorties sont semblables aux fonctions Unix standards mais elles portent sur des variables parallèles. Quand un fichier parallèle est ouvert, le système alloue une variable de type FILE à chacun des processeurs virtuels de la collection. On n'a plus un pointeur de fichier séquentiel sur un fichier unique mais une collection de pointeurs sur une collection de fichiers. Pour des raisons d'expressivité et d'efficacité, les fonctions de lectures et d'écritures acceptent à la fois les variables parallèles et les variables scalaires. Les fonctions d'écritures sont par exemple les suivantes :

```
int:current fwrite(char:current *buf, int:current size,
                   int:current nitems, FILE:current *fp);

int:current fwrite(char:current *buf, int size,
                   int nitems, FILE:current *fp);
```

L'utilisation des variables scalaires a un impact très important sur les performances. Lorsque les valeurs scalaires sont utilisées, le compilateur est certain que tous les processeurs virtuels effectuent les mêmes entrées/sorties, il peut de ce fait mettre en place de nombreuses optimisations.

6.2 Implémentation

Stream* est construit au dessus d'un système de fichiers qui supporte les accès Unix. Pour obtenir des performances raisonnables en dépit de la souplesse d'utilisation, trois modes ont été définis :

Collective Buffering (CB) : Ce mode est utilisable lorsque tous les VP sont actifs et qu'ils lisent ou écrivent tous des données de même taille.

No Buffering (NB) : Ce mode est le plus efficace. Il est utilisable lorsque les accès vérifient les contraintes précédentes, et lorsque les données accédées sont contiguës en mémoire.

Independent Buffering (IB) : Ce mode d'utilisation est le plus général mais aussi le plus lent. Chaque processeur virtuel gère son propre buffer. La taille des accès varie d'un processeur virtuel à un autre.

Le programmeur n'a pas à spécifier le mode d'utilisation qu'il souhaite utiliser. C'est au compilateur de choisir le mode le plus approprié. L'utilisateur a cependant intérêt à utiliser au maximum les fonctions à paramètres scalaires.

Pour supporter le mode d'utilisation le plus général (IB), les flots de données sont divisés en blocs. Chaque processeur virtuel possède un buffer de la taille d'un bloc dans lequel il effectue ses accès. Quand un buffer est complètement rempli, il est recopié dans un fichier global. Pour ne pas ralentir le système, les blocs sont agrégés en «super bloc». Cette technique peut permettre d'obtenir des performances raisonnables en écriture mais elle ne résout pas les problèmes posés en lecture. Il n'est pas possible d'anticiper les lectures de plusieurs milliers de processeurs virtuels.

6.3 Conclusion

Cette extension du modèle de programmation aux entrées/sorties est très séduisante. Stream* est (il me semble) le seul système qui associe des flots de données aux processeurs virtuels. L'utilisation de ces fichiers répartis pose d'importants problèmes de performances. Les accès «irréguliers» dans lesquels les processeurs vont lire ou écrire des données de taille différente sont particulièrement complexes. La projection efficace de ces ensembles de fichiers hétérogènes sur un périphérique d'entrées/sorties parallèle est à ma connaissance un problème ouvert.

Les trois modes qui ont été définis permettent d'espérer des performances raisonnables pour certaines utilisations particulières. En effet, lorsqu'on se limite aux modes (CB) ou (NB) le modèle à fichier partagé n'est rien de plus qu'un modèle à objets partagés!

7 Le système PANDA

La plupart des applications scientifiques réalise des opérations d'entrées sorties conceptuellement simples telles que la lecture ou l'écriture de tranches de tableaux, de tableaux entiers, ou de listes de tableaux. Malheureusement, les algorithmes nécessaires pour réaliser efficacement ce type d'opérations sur une machine particulière sont généralement complexes et non portables. Pour remédier à ces problèmes, les auteurs proposent une interface de haut niveau, qui encapsule les détails d'implémentations [46, 45, 47, 49, 48]. Cette interface vise en particulier les programmes qui effectuent des sauvegardes régulières «checkpoints», de façon à pouvoir redémarrer l'application en cas de problème. Les écritures sont donc privilégiées.

7.1 Le modèle Panda

Panda permet d'accéder à des tableaux. C'est donc un modèle à objets structurés. Ces tableaux sont distribués à la fois en mémoire et dans le système de fichier. Le programmeur doit spécifier à la création des fichiers la distribution qu'il souhaite au niveau du système de fichier, c'est une interface explicite à la compilation.

7.2 Interface

Panda est conçu pour les programmes SPMD et dispose d'une interface de haut niveau. Les opérations disponibles sont les suivantes :

1. Lecture et écriture d'un tableau ou d'une tranche de tableau ;
2. Lecture et écriture d'une liste de tableaux ;
3. Ajout d'un point de reprise.

Ces opérations doivent être exécutées sur tous les processeurs[9], le résultat est indéfini dans le cas contraire. Elles sont applicables sur tous les tableaux distribués *Collapsed* ou *Block*. L'utilisateur peut, spécifier la distribution de ses données sur une grille de disques logiques précèdement spécifiée ou laisser ce choix à la charge du système. Les paramètres par défauts dépendent de l'architecture utilisée. Plusieurs schémas sont proposés :

Chunking : Cette stratégie est illustré par la figure(1.11). Le cube représente le tableau global. Les sous cubes sont les données placées dans la mémoire des processus du programme parallèle. Panda permet de conserver cette organisation des données sur les disques. C'est ce que les auteurs appellent le «Natural chunking». Tous les éléments d'un même sous cube sont alors contiguës dans le fichier et peuvent être lus ou écrit à l'aide d'un unique accès disque. Cette organisation par défaut est particulièrement efficace lorsque les données sont lues avec la même distribution que celle qui avait été utilisée lors de l'écriture. Dans le cas contraire, une redistribution de données est nécessaire.

9. C'est au compilateur ou à l'utilisateur d'assurer la correction des appels, aucune synchronisation n'est mise en place

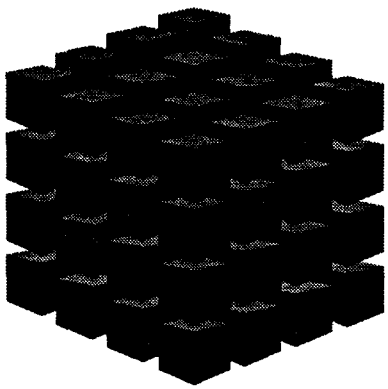


FIG. 1.11 - *Organisation physique des données sur les disques*

subchunking : Cette stratégie est une extension récursive de la stratégie précédente. Les tranches de tableaux sont redécoupées de façon à améliorer l'utilisation des disques (clustering).

Overlaps : Pour réduire les communications de voisinage des zones de recouvrement sont souvent utilisées. Celles-ci peuvent être sauvegardées lors des «checkpoints».

Compression : Panda est également capable de lire ou d'écrire des fichiers compressés à l'aide d'un algorithme de compression classique. Comme Panda manipule des tranches de tableau, il est possible d'accéder aux éléments souhaités sans compresser (décompresser) tout le tableau. Il y a cependant un surcoût pour la gestion d'adresse car la taille d'une tranche de tableau dépend de la «compressibilité» des données qu'elle contient.

7.3 Exemple

La figure 1.12 présente une petite application écrite en C++ qui utilise Panda pour lire et écrire des tableaux distribués «à la HPF».

Les quatre premières instructions permettent de créer trois tableaux d'entiers de taille 512 au cube. Ces tableaux sont distribués par blocs sur une grille tri-dimentionnelle de processeurs. Comme ces tableaux sont toujours utilisés ensembles, le programmeur les a regroupés dans une liste de nom : «simulation». Les données concernant la «simulation» sont placées dans un fichier distribué appelé «simulation.schema». L'organisation des données dans ce fichier ne concerne pas le programmeur. Ces informations sont associées au fichier de données.

Pour effectuer des entrées/sorties le programmeur doit créer un «SchemaFile» et l'associer

à un fichier distribué. Ce «SchémaFile» permet alors de lire et d'écrire des tableaux ou des listes de tableaux. La création d'un nouveau fichier est un peu plus complexe car il faut spécifier la topologie de la grille de disques («disk_rank» et «disk_layout») ainsi que la fonction de distribution qui sera utilisée pour projeter les données sur cette grille («disk_dist»).

Cet exemple illustre les principales fonctionnalitées de PANDA. On remarque notamment :

- Les accès de hauts niveaux qui portent sur des listes d'objets structurés distribués. La distribution des données est à la charge du système.

- L'association des fonctions de distributions et des données dans un même objet : le fichier parallèle. La grille de disques virtuels et les fonctions de distributions sont fixées lors de l'écriture des objets. Ce sont des propriétés du fichier qui ne pourront plus être modifiées.

7.4 Implémentation

Panda a été conçu pour tirer au maximum profit des systèmes de fichiers existants. L'interface de haut niveau a permis de développer des implémentations efficaces sur différentes architectures. Nous allons dans les paragraphes suivants détailler ces architectures de façon à montrer l'indépendance entre le modèle de fichier d'une part et les périphériques d'entrées/sorties disponibles d'autre part.

7.4.1 Implémentation séquentielle

L'implémentation séquentielle (fig. 1.13) est bâtie au dessus du système de fichier Unix. Elle a été développée afin de permettre la visualisation sur machine séquentielle des résultats calculés sur des architectures distribuées. Cette interface doit également permettre de développer de nouveaux codes sur des machines séquentielles avant de les porter sur des machines parallèles. Panda stocke alors les données dans l'ordre Unix traditionnel et utilise un buffer pour réorganiser les données lorsque cela est nécessaire. il minimise ainsi le nombre d'accès disques.

```
status clean_data(){
  status stat                    = STATUS_OK;
  Rank array_rank                = 3;
  Size temperature_array_size[]  = {512, 512, 512};
  ElementSize int_size_in_bytes  = sizeof(int);
  Rank memory_rank               = 3;
  Size memory_layout[]           = {2,4,4};              // 3D 2*4*4 mesh
  Distribution memory_dist[]     = {BLOCK,BLOCK,BLOCK}   // HPF directives
  // Distribute array as four three dimentionnal  subarrays in the file
  Rank disk_rank                 = 3;
  Size disk_layout[]             = {2,2}
  Distribution disk_dist[]       = {BLOCK,BLOCK,NONE}    // HPF directives

  MemoryArrayLayout *memory = new MemoryArrayLayout(memory_rank,
                                  memory_layout, *memory_dist);

  Array *temperature = new Array("temperature", array_rank,
       temperature_array_size, int_size_in_bytes, *memory, *memory_dist);
  Array *pressure = new Array("temperature", array_rank,
       pressure_array_size, int_size_in_bytes, *memory, *memory_dist);
  Array *density = new Array("temperature", array_rank,
       density_array_size, int_size_in_bytes, *memory, *memory_dist);

  ArrayList *simulation = new ArrayList("simulation");

  //insert three arrays in the array list
  simulation->insert(*temperature);
  simulation->insert(*pression);
  simulation->insert(*density);

  // schema file that will be accessed when reading the arrays
  SchemaFile *schema = new SchemaFile("simulation.schema");

  // read all the arrays in the list using info from the schema file
  stat = simulation->read(*simulation);

  // remove the noise from the arry list
  stat = compute_clean_arrays(*simulation);

  // array layout for writting the arrays to disk
  DiskArrayLayout *disk=new DiskArrayLayout(dist_rank, dist_layout,*disk_dist);

  // schema file that will be create or update while writting the array
  schema = new SchemaFile("simulation_clean.schema");

  // write the arrays and create or update the schema file
  return simulation->write(*schema, *disk);
}
```

FIG. 1.12 - *Exemple d'application utilisant PANDA*

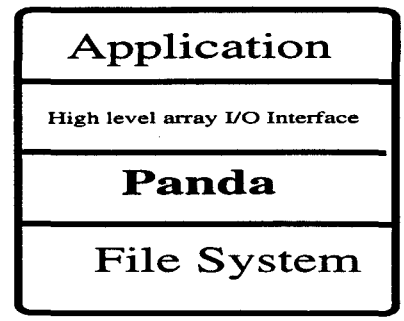


FIG. 1.13 - *Architecture séquentielle*

7.4.2 Implémentation sur système de fichiers parallèles

Cette implémentation (fig. 1.14) est bâtie au dessus d'un système de fichiers parallèle qui supporte le modèle d'entrées/sorties Unix classique : Un fichier est une suite linéaires d'octets. Elle est conçue pour les programmes SPMD. Chaque processeur lit ou écrit directement ses données dans le fichier global. Panda privilégie les écritures en adoptant par défaut le «natural chunking». Les données sont alors placées sur les disques comme elles le sont en mémoire. Chaque processeur peut écrire la tranche de tableau qu'il possède à l'aide d'un unique accès disque. Les auteurs proposent d'utiliser la «two phase access strategy» [20] lorsqu'il est nécessaire de réorganiser les données mais cette optimisation n'est pas mise en place.

7.4.3 Implémentation dirigée par les serveurs

Cette dernière implémentation (fig. 1.15) permet de construire un système de fichiers parallèle à partir d'un ensemble de système de fichiers séquentiels. Les processus PANDA sont alors placés sur les nœuds de calculs (PANDA clients) et sur les nœuds d'entrées/sorties[10] (PANDA serveurs). PANDA transfère alors les données entre les clients et les serveurs. Grâce à l'interface de haut niveau, PANDA a une connaissance globale des opérations d'entrées/sorties. Il connaît la distribution des données sur les disques et celle qui est utilisée en mémoire, il peut de ce fait planifier ses requêtes de façon à minimiser le nombre d'accès disques. Lorsqu'une application souhaite écrire une tranche de tableau, une requête est envoyée aux serveurs. Ceux-ci s'adressent alors aux clients pour obtenir les informations dans l'ordre souhaité, c'est ce que les auteurs appellent la «server-Directed Collective I/O».

La conception et la réalisation de Panda repose sur une interface de haut niveau. Celle-ci

10. Il est nécessaire de pouvoir lancer des processus utilisateurs sur les nœuds d'E/S comme c'est par exemple le cas sur la SP2

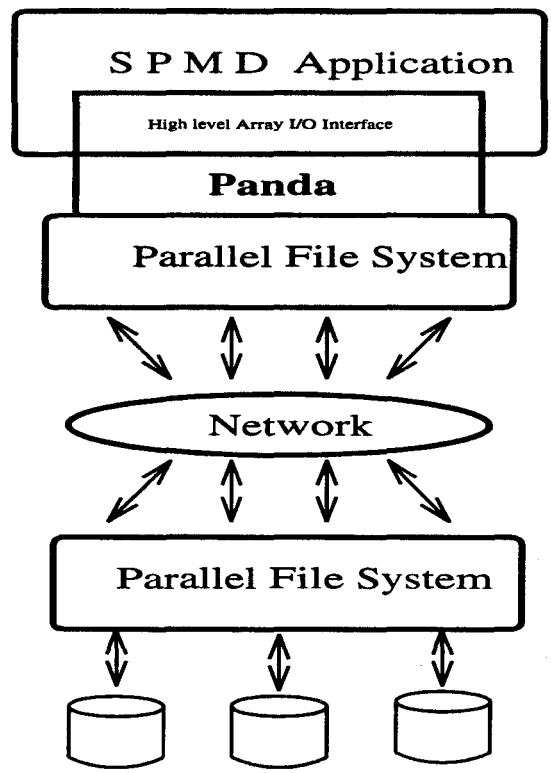


FIG. 1.14 - *Architecture parallèle*

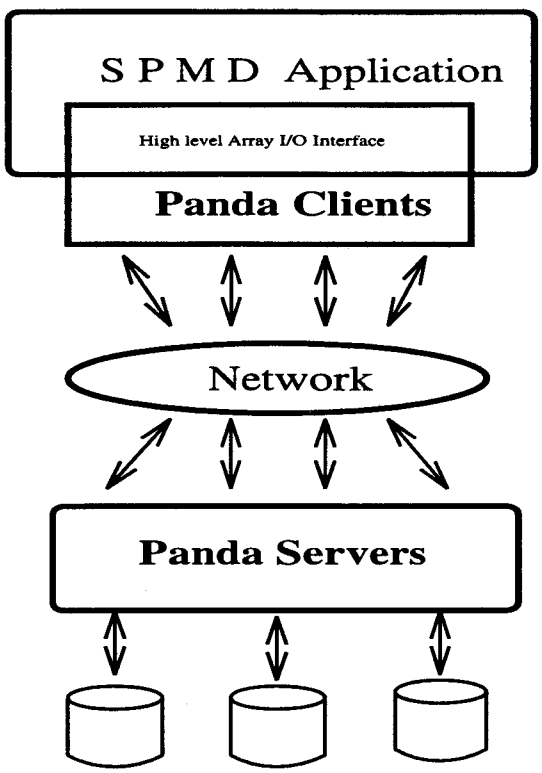


FIG. 1.15 - *Architecture dirigée par les serveurs*

est basée sur les besoins d'applications «réelles» telles que les calculs en dynamique des fluides (CFD). Cette interface abstraite permet d'encapsuler les détails de l'implémentation tout en étant très expressive : une simple requête d'entrée/sortie peut impliquer plusieurs tableaux distribués. Le système a ainsi une vision globale des accès à réaliser et peut choisir librement l'ordonancement de façon à en minimiser les coûts. La stratégie dirigée par les serveurs a été implémentée sur une IBM SP2. Les performances obtenues montrent une excellente utilisation de la bande passante disponible.

7.5 Conclusion

Panda est un système complet qui fonctionne sur différentes plateformes et supporte plusieurs organisations physiques des données. C'est un modèle original qui vise des applications spécifiques (redémarrage d'applications). À la différence de la majorité des systèmes d'entrées/sorties parallèles, PANDA ne distribue pas les fichiers mais les objets qui les composent. Un fichier n'est plus une simple suite d'octets, c'est une suite de tableaux multidimentionnels. On n'accède plus en parallèle à un ensemble d'objets ; on accède à un objet (tableau) distribué en parallèle.

Il est regrettable que le système ne supporte pas les distributions *Cyclic*. L'absence de support pour les systèmes hétérogènes est également regrettable, l'interface de haut niveau aurait permis de traiter simplement ce problème. Il est important de noter l'indépendance du modèle d'entrées/sorties vis à vis des systèmes sous jacents. Un «bon modèle» permet d'obtenir de «bonnes performances» quelque soit le périphérique d'entrée/sortie disponible (encore faut-il que le programmeur soit à même de choisir une distribution appropriée).

8 Le projet PASSION et la bibliothèque VIP-FS

Le projet PASSION[11] regroupe trois axes de recherche :

1. Support pour la manipulation d'objets de très grande taille : «out of core computation». De nouvelles directives HPF sont proposées [9, 10]. Les auteurs introduisent notamment la notion de «FILE TEMPLATE» qui est utilisée pour aligner les données sur les disques. Des techniques de compilations ont été développées puis intégrées dans un compilateur [57, 58].

11. Parallel And Scalable Software for Input/Output

2. Intégration du parallélisme de tâches et du parallélisme de données [4, 5]. Cela est réalisé à l'aide du langage Fortran M qui permet de créer des tubes à parallélisme de données entre deux tâches HPF.

3. Mise au point d'une bibliothèque d'entrées/sorties parallèles [27, 22]. C'est le projet VIP-FS qui fera l'objet des prochains paragraphes.

8.1 Le modèle VIP-FS

Il n'y a pas vraiment de modèle VIP-FS. Cette bibliothèque a été développée pour les programmes à parallélisme de tâches mais aussi pour les programmes à parallélisme de données. Le passage d'un modèle à l'autre s'effectue en spécifiant une distribution de données.

Lorsque le programmeur utilise la distribution par défaut, toutes les tâches partagent le même espace d'adressage (modèle virtuellement partagé à référentiel global). Les tâches accèdent alors aux données indépendamment les unes des autres à l'aide de leur propre pointeur de fichier.

Quand il spécifie une distribution, le programmeur doit utiliser le modèle SPMD[12]. Toutes les tâches effectuent les mêmes accès avec les mêmes paramètres. Ce second mode d'utilisation permet de lire ou d'écrire des tableaux distribués par blocs.

VIP-FS apparaît donc deux fois dans notre classification. Il est présent au niveau des modèles à objets structurés explicitement distribués ainsi qu'au niveau des modèles à pointeurs de fichiers multiples.

8.2 La bibliothèque VIP-FS

VIP-FS (pour VIrtual Parallel File System) est comme son nom l'indique un système de fichiers parallèle virtuel. Le système est construit au dessus des fichiers Unix à l'aide d'une bibliothèque de communication par messages. Il est conçu pour tirer profit des environnements distribués hétérogènes. VIP-FS à été construit pour être portable. Il se compose de trois couches distinctes :

Une interface utilisateur : L'interface utilisateur permet d'accéder aux fichiers parallèles. Cette interface est semblable à l'interface Unix (`pvfs_open`, `pvfs_close`, `pvfs_read`,

12. le résultat est indéfini dans le cas contraire

`pvfs_write`). L'utilisateur peut, s'il le souhaite, spécifier la distribution des données utilisée sur les nœuds de calcul ainsi que celle qui est utilisée sur les disques[13].

Un gestionnaire de fichier parallèle : La bibliothèque VIP-FS émule un système de fichiers parallèles à partir d'un ensemble de systèmes de fichiers séquentiels. Le rôle du gestionnaire de fichier est de fournir une vue globale des fichiers parallèles. Un fichier parallèle est «vu» comme une simple suite d'octets exactement comme un fichier Unix standard.

Des gestionnaires de fichiers séquentiels : Ils jouent le rôle de médiateur entre le gestionnaire de fichiers parallèles d'une part et le système de fichiers sous jacent d'autre part.

Le découpage du système de fichiers parallèles en trois couches bien définies permet de porter facilement les outils d'un système vers un autre. La migration des données d'une couche à une autre est réalisée à l'aide d'une bibliothèque de communication.

8.3 La migration des données

Trois stratégies sont utilisées pour migrer les données des nœuds d'entrées/sorties aux nœuds de calculs :

Direct access : Cette stratégie est utilisée pour les programmes à parallélisme de tâches. Il n'existe pas de notion d'accès global. Les clients possèdent leurs propres pointeurs et effectuent des accès indépendamment les uns des autres.

Two phases access : Cette stratégie et la suivante ne sont utilisables que par les programmes SPMD. La stratégie «en deux temps» consiste à lire ou écrire les données sur les fichiers en fonction de leurs distributions sur les disques de façon à minimiser le nombre d'accès [20]. Les données sont alors rangées dans des buffers puis redistribuées conformément à la distribution souhaitée.

Assumed requests : Cette stratégie est une optimisation de la méthode précédente qui permet de réduire le nombre de messages échangés en partitionnant les processus utilisateurs.

13. Il dispose pour cela de la fonction `pvfs_ioctl`. Seules les distributions par blocs sont possibles

VIP-FS conserve la vision Unix des fichiers mais possède deux modes d'utilisations distincts. Le premier mode «adapté» au parallélisme de données n'apporte rien par rapport à l'interface Unix (même sémantique et mêmes problèmes). Ce premier mode d'utilisation permet de faire des accès séquentiels sur un périphérique d'entrées/sorties parallèle virtuel[14]. Tout se passe «comme si» l'utilisateur avait accès à un fichier Unix placé sur un disque RAID.

En spécifiant la distribution des données en mémoire ainsi que la distribution du fichier sur les disques l'utilisateur a accès à un second mode d'utilisation «adapté» au parallélisme de données. L'interface choisie est discutable car elle ne fait pas apparaître clairement la notion d'objet structuré. Il y a d'une part les accès qui portent sur des tableaux et d'autre part le fichier qui est composé d'une simple suite d'octets. Il y a une dichotomie entre l'espace fichier et l'espace mémoire. Un même objet admet deux représentations et le programmeur doit continuellement passer de l'une à l'autre.

9 MPI-IO

MPI-IO [41, 15, 16, 17] est une extension de la bibliothèque MPI (pour Message Passing Interface). Le but de cette nouvelle interface est de fournir un standard permettant la description d'entrées/sorties parallèle dans les applications MPI. La conception de MPI-IO résulte des constatations suivantes :

- Une entrée/sortie parallèle peut-être modélisée par un envoi de message.
- Les systèmes de fichiers parallèles existants sont très hétérogènes et les applications qui les utilisent ne sont pas portables.
- Le système de fichier Unix n'a pas été conçu pour les accès partagés et n'est donc de ce fait pas adapté aux applications parallèles.

L'interface MPI-IO vise les applications scientifiques et propose des fonctionnalités adaptées aux problèmes concrets. Pour décrire leur bibliothèque, les auteurs de MPI-IO introduisent ce qu'ils appellent : «`The Anatomy of a Parallel I/O Library`». Cette «`anatomy`» détaille les caractéristiques majeures d'une bibliothèque d'entrées/sorties. Dans les prochaines sous sections, nous étudierons les principales caractéristiques de la bibliothèque MPI-IO.

14. Le titre «VIP-FS» est d'ailleurs assez bien choisi

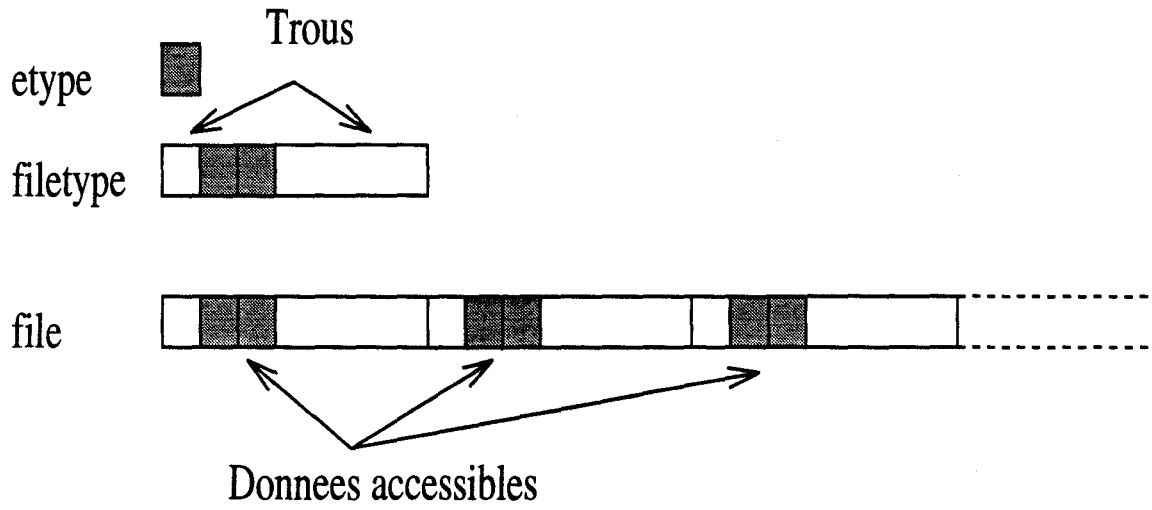


FIG. 1.16 - *Vu local d'un processus*

9.1 Le «partionnement» des données

Pour décrire le «partionnement»[15] des fichiers parallèles les auteurs réutilisent les types de données MPI. À l'aide de ces types, l'utilisateur choisit l'organisation des données dans les buffers des processus utilisateurs appelés **buftype** ainsi que celle qui est utilisée dans le fichier parallèle **filetype** à partir d'un type MPI élémentaire appelé **etype**. Cette construction permet d'assurer la cohérence entre les types buftype et filetype. Elle est également utilisée pour mettre en place les conversions de données nécessaires dans les environnements hétérogènes.

Le fichier global est une suite linéaire d'objets élémentaires etype. Le filetype définit un motif qui est répliqué sur tout le fichier. Il est utilisé pour «partionner» le fichier global. Chaque processus a ainsi une vue locale du fichier parallèle. MPI-IO permet donc de définir des référentiels relatifs. Comme les types MPI sont construits à partir d'un ensemble de champs placés à différents déplacements, il est possible de préserver des trous qui pourront être utilisés par d'autre processus (fig. 1.16) qui utilisent des filetype complémentaires. Les seules données accessibles par le processus, sont celles qui ont été spécifiées dans le filetype. Cette construction est très générale et permet de construire des motifs très variés. Elle permet en particulier de créer des zones de recouvrement. L'organisation des données est à la charge du programmeur. Aucun ordre n'est imposé par le système. Pour aider les programmeurs, MPI-

15. Les auteurs parlent de «data partitionning» mais ce n'est pas un partition au sens mathématique du terme, les sous ensembles ne sont pas nécessairement disjoints et leur réunion n'est pas toujours égale à l'ensemble de départ

IO fournit des constructeurs de filetype pour les modes d'accès les plus courants : broadcast, reduce et distributions HPF. Ces constructeurs devraient couvrir la plupart des besoins.

9.2 Accès aux données

Dans leur anatomie d'un système de fichiers parallèles, les auteurs insistent sur les trois aspects orthogonaux suivants :

1. Le positionnement dans le fichier : Il peut-être implicite ou explicite.
2. La synchronisation : Les accès sont soit bloquants soit non bloquants.
3. La coordination : une opération d'entrée sortie est individuelle ou collective.

MPI-IO fournit des fonctions pour chaque combinaison et propose deux types de pointeurs de fichiers que nous définirons dans les paragraphes suivants.

9.2.1 Positionnement

Lorsqu'on ouvre un fichier MPI-IO, le système crée un pointeur de fichier partagé par tous les processus du groupe ainsi qu'un pointeur de fichier local sur chacun des processus. Ces pointeurs sont complètement indépendants. Les accès réalisés par l'intermédiaire du pointeur local (MPI_read ou MPI_write) ne modifient pas le pointeur partagé (MPI_read_shared ou MPI_write_shared). Le pointeur de fichier global n'a de sens que si tous les processus utilisent le même filetype[16].

Après chaque entrée/sortie, le pointeur de fichier pointe sur le premier élément qui suit la dernière valeur accédée. Ce principe s'applique à tous les pointeurs (globals et partagés) pour tous les déplacements (absolus ou relatifs) et pour tous les accès. Dans tous les cas, la mise à jour du pointeur de fichier est réalisée avant la réalisation effective de l'opération d'entrée/sortie afin de ne pas séquentialiser les appels. À chaque accès l'utilisateur peut :

- utiliser un pointeur de fichier individuel ;
- utiliser le pointeur de fichier partagé;
- expliciter l'adresse du prochain accès.

16. Dans le cas contraire, ils ne peuvent accéder aux mêmes données

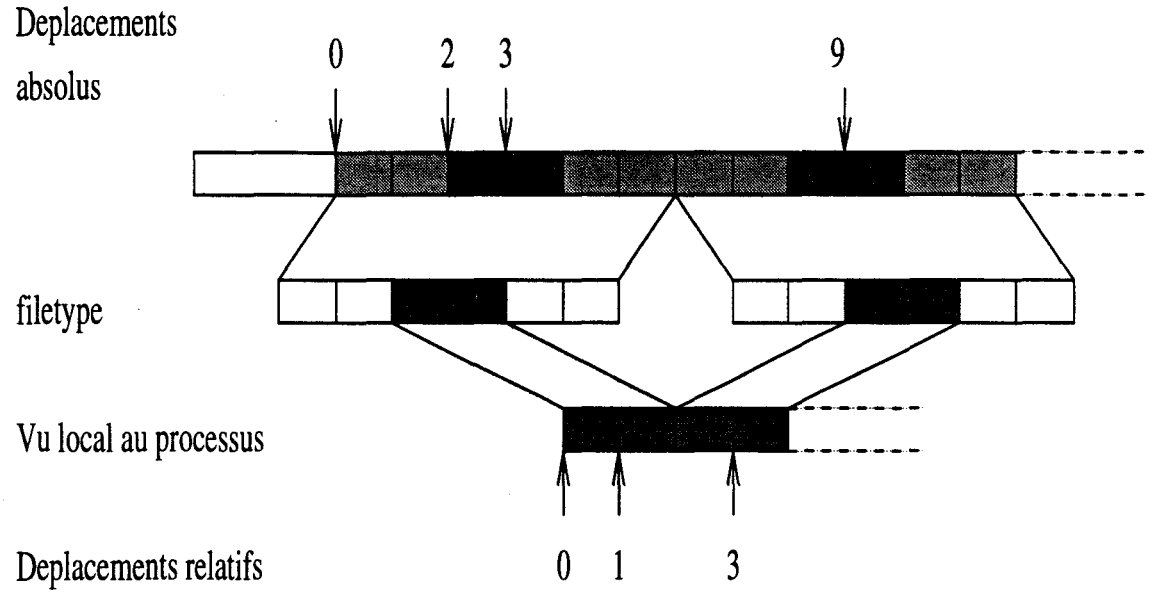


FIG. 1.17 - *Déplacements relatifs et absolus*

Ces trois méthodes sont orthogonales et peuvent être utilisées ensembles dans une même application. Le déplacement est exprimé en nombre de etype. Il peut-être absolu ou relatif au filetype (fig. 1.17). Un pointeur de fichier absolu ignore le partitionnement des données et peut désigner tous les éléments d'un fichier. Lorsque ce pointeur global désigne un «trou», il est automatiquement avancé jusqu'au premier élément accessible.

9.2.2 Le synchronisme

MPI-IO supporte les accès asynchrones. Cela permet de recouvrir les opérations d'entrées/sorties avec les calculs. Les accès asynchrones nécessitent deux appels de fonctions. Une première requête non bloquante permet de demander un accès puis une seconde est utilisée pour attendre la terminaison de l'opération précèdement lancée.

9.2.3 La coordination

Les entrées/sorties parallèles ne sont efficaces que lorsque le système a une vue globale des accès réalisés. Les opérations «collectives» ont été créées dans ce but. Toutes les fonctions sont disponibles en deux versions. Les fonctions indépendantes sont appelées par des processus individuels. Les fonctions collectives sont toujours appelées par tous les processus du groupe.

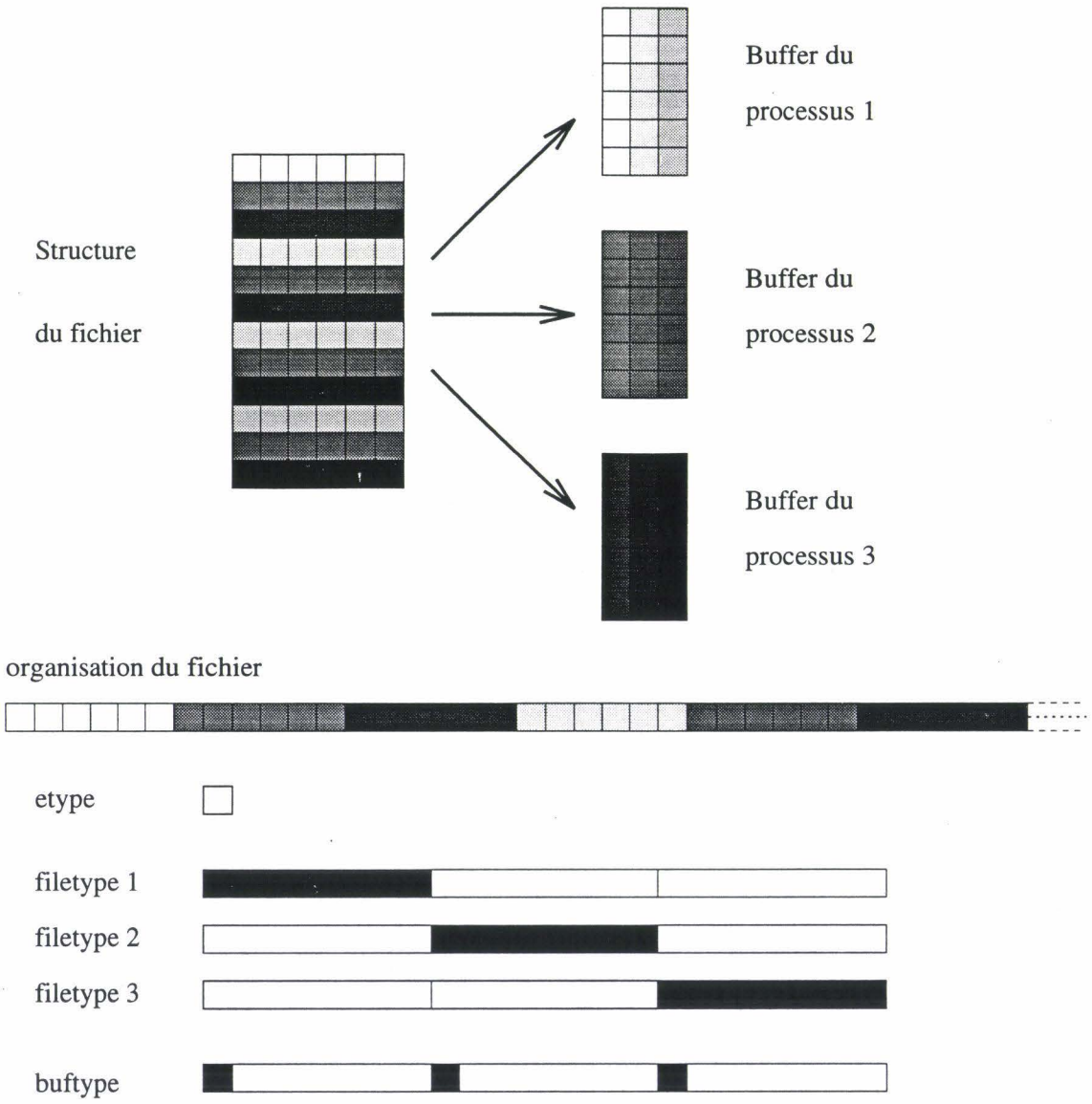


FIG. 1.18 - *Lecture et transposition d'une matrice*

Ces fonctions collectives ne réalisent aucune synchronisation. Du point de vue sémantique il n'y a aucune différence entre les opérations collectives et les autres. Les fonctions collectives sont uniquement proposées pour des raisons d'efficacité car elles autorisent d'importantes optimisations.

9.3 Conclusion

Il est difficile de classer MPI-IO dans un modèle ou dans un autre en raison du large éventail de fonctionnalités qu'offre cette bibliothèque. Dans tous les cas on accède à des objets non structurés placés dans un fichier virtuellement partagé. Nous pouvons donc exclure les modèles adaptés au parallélisme de données. Il semble en revanche difficile d'exclure ne serait ce qu'un unique modèle adapté au parallélisme de tâches.

Avec le mécanisme de filetype, l'utilisateur peut utiliser le même espace d'adressage pour toutes ses tâches ou définir des espaces d'adressages multiples. Aucun mécanisme n'est prévu pour informer le système du non recouvrement des espaces d'adressages mais cela n'a de toute façon pas d'intérêt puisqu'aucune protection n'est jamais mise en place (si deux tâches accèdent aux même instant aux mêmes données le résultat est indéterminé). Les tâches disposent de leur propre pointeur mais aussi d'un pointeur global partagé. MPI-IO est donc à la fois dans les modèles à pointeurs multiples et dans les modèles à pointeur partagé.

MPI-IO est une extension de la bibliothèque de communications MPI. Le rapprochement qui s'amorce entre les bibliothèques d'entrées/sorties d'une part et les bibliothèques de communications d'autre part est très intéressant. Malheureusement il n'est pas encore possible de rediriger les messages dans un fichier. Le système Unix ne distingue pas les communications entre deux processus des opérations d'entrées/sorties.

10 La bibliothèque PIOUS

PIOUS (pour Parallel Input Output Server) est un système de fichiers parallèles construit au dessus de la bibliothèque PVM [38, 37, 40, 39]. L'interface utilisateur supporte les accès parallèles à des fichiers appelés **parafiles** et garantit une sémantique «séquentielle» des accès. Cette sémantique permet d'assurer la cohérence des fichiers. Si P_1 et P_2 sont deux processus distincts qui accèdent au même instant à deux sections s_1 et s_2 non disjointes d'un même parafile, PIOUS garantit comme Unix la séquentialité des accès. Un des deux processus sera bloqué jusqu'à ce que l'autre ait terminé son opération d'E/S (fig. 1.19). Cette sémantique

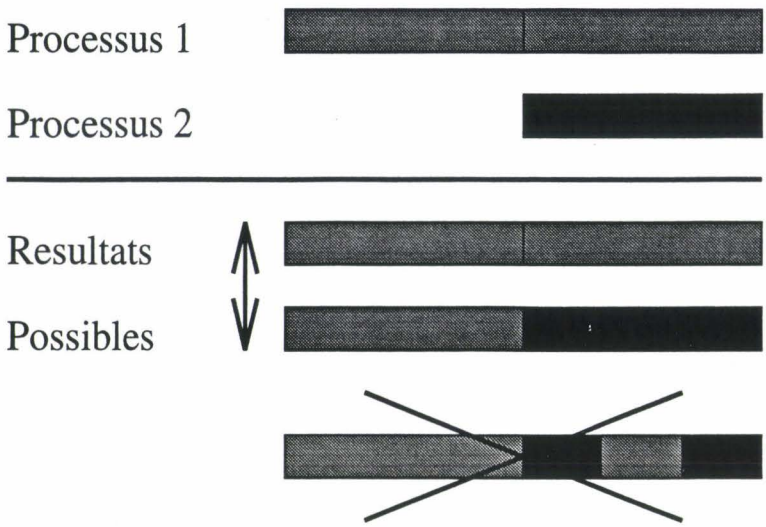


FIG. 1.19 - *Sémantique des accès Unix*

est assurée à l'aide d'un mécanisme de transactions.

10.1 Le modèle PIOUS

Un Parafile est un fichier logique composé d'un ou de plusieurs segments physiques. Ces segments sont des séquences linéaires d'octets et leur nombre reste constant tout au long de l'existence du fichier. Les données de chaque parafile sont distribuées cycliquement sur les segments par blocs de taille constante[17]. L'interface permet à un groupe de processus de manipuler les parafiles avec trois «vues» différentes :

Global : Le fichier parallèle est vu comme une simple suite d'octets, tous les processus du groupe partagent un même pointeur de fichier. Tous les accès sont atomiques.

Independant : Le fichier parallèle est vu comme une simple séquence d'octets, mais les processus du groupe ont leur propre pointeur de fichier. Tous les accès sont atomiques.

Segmented : La structure en segment des parafile est visible, chaque processus accède à un segment avec un pointeur local. Les accès sont toujours atomiques.

La vue utilisée est complètement indépendante de la représentation physique. Il est toujours possible d'ouvrir un parafile avec la vue souhaitée. Tous les processus d'un même groupe

17. L'utilisateur peut s'il le souhaite spécifier la taille de ces blocs.

doivent obligatoirement ouvrir le parafile avec la même vue.

10.2 La bibliothèque utilisateur

La bibliothèque qui permet d'accéder aux fichiers PIOUS, est très proche de la bibliothèque standard «stdio.h». Les fonctions qui permettent d'ouvrir les fichiers sont cependant plus complexes car l'utilisateur doit choisir le mode d'utilisation ainsi que (dans certain cas) des paramètres spécifiques tels que les noms de machines ou la taille des blocs. La sémantique de ces fonctions change en fonction du mode qui a été choisi.

10.3 Conclusion

Le système PIOUS apparaît trois fois dans notre classification car il propose trois modes d'utilisation distincts :

1. Avec le mode **global** les processus partagent un même pointeur avec lequel ils accèdent à un espace d'adressage unique.
2. Le mode **segmented** permet lui, d'utiliser plusieurs espaces d'adressages. Avec ce mode, un processus n'a accès qu'a un sous ensemble du fichier nommé segment. C'est une version limitée du modèle à référentiels relatifs. Les tâches n'ont pas la possibilité de définir leur propre référentiel, elles ont simplement le choix entre plusieurs sous ensembles de données.
3. Le mode **independant** conserve le modèle Unix. Toutes les tâchent partagent le même espace d'adressage. Elles disposent chacune de leur propre pointeur avec lequel elles effectuent des accès indépendemment les unes des autres. PIOUS conserve cependant un intérêt majeur par rapport au système Unix puisqu'il distribue le fichier logique (appelé parafile) sur différents disques.

Dans tous le cas, PIOUS assure l'atomicité des accès à l'aide d'un mécanisme de transaction. Il est regrettable qu'il ne soit pas possible d'éviter ce mécanisme car il existe des contextes dans lesquels celui-ci est complètement inutile ; c'est notamment le cas lorsque les tâches accèdent à des «segments» différents, on a alors un modèle à référentiels disjoints. Si le programmeur avait la possibilité d'expliciter cette propriété, cela lui permettrait de ne pas payer un service inutile. Comme PIOUS ne dispose pas de cette fonctionnalité, nous ne l'avons pas placé au niveau des modèles à référentiels disjoints.

11 Conclusion

La conception d'un modèle d'entrées/sorties parallèle est particulièrement complexe. Les objectifs à atteindre sont multiples et souvent contradictoires. Un «bon modèle» doit être à la fois simple, cohérent avec le modèle de programmation et utilisable dans différents contextes.

Le modèle à objets structurés est bien adapté au parallélisme de données. C'est une extension «naturelle» du modèle Unix suffisamment générale pour supporter les environnements hétérogènes. Avec les interfaces à distribution explicite à l'exécution, l'accès aux données est indépendant de la représentation physique du fichier. Cela simplifie le développement des programmes et permet la réutilisabilité. Comme l'utilisateur a la possibilité d'expliciter la distribution des données qui convient la mieux à son application, les fonctions d'accès qui utilisent ce type d'interfaces sont potentiellement efficaces.

Dans le prochain chapitre nous étudierons les algorithmes de redistributions existants. La plupart d'entre eux redistribuent les données sur une unique machine parallèle et ont besoin de connaître les paramètres de redistributions à la compilation. Pour utiliser en parallèle les nœuds d'entrées/sorties avec les nœuds de calculs et pour recouvrir les calculs avec les communications, nous serons amenés à développer nos propres algorithmes de redistribution.

Chapitre 2

Redistribution : État de l'art

Les machines à mémoire partagée sont bâties autour d'un réseau de communication qui relie les processeurs à une mémoire unique. L'ensemble des processeurs partage un même espace mémoire global auquel ils accèdent par l'intermédiaire du réseau. Le problème de l'accès aux données est reporté au niveau matériel. Cette approche nécessite un réseau de communication extrêmement efficace et une organisation mémoire adaptée de façon à limiter la latence des accès.

Il est possible d'éviter la latence du réseau en utilisant des machines à mémoire distribuée. Avec cette approche, chaque processeur dispose de sa propre mémoire qui n'est accessible que par lui. Ces processeurs sont interconnectés à un réseau de communication qui n'est utilisé que pour communiquer les données d'un processeur à un autre. Un processeur accède directement à sa mémoire locale mais doit effectuer des communications pour accéder aux données placées dans la mémoire des autres processeurs. Cette approche matérielle est efficace lorsque le nombre d'accès local est très supérieur au nombre d'accès distants. Pour maximiser le nombre d'accès local, la plupart des langages à parallélisme de données propose des directives de placement de données. Ces directives ne modifient pas la sémantique des programmes mais elles ont un énorme impact sur les performances [31]. Le choix de la fonction de distribution dépend de l'algorithme à effectuer. Pour conserver une bonne localité de données, il est souvent nécessaire d'effectuer des redistributions entre les différentes phases de calcul.

Dans une première section nous présenterons les fonctions de distributions usuelles. Nous nous intéresserons ensuite aux **algorithmes de redistribution dynamique** capable de redistribuer les données sans connaître les paramètres des distributions à la compilation. Nous distinguerons les **algorithmes généraux** qui effectuent toutes les redistributions des

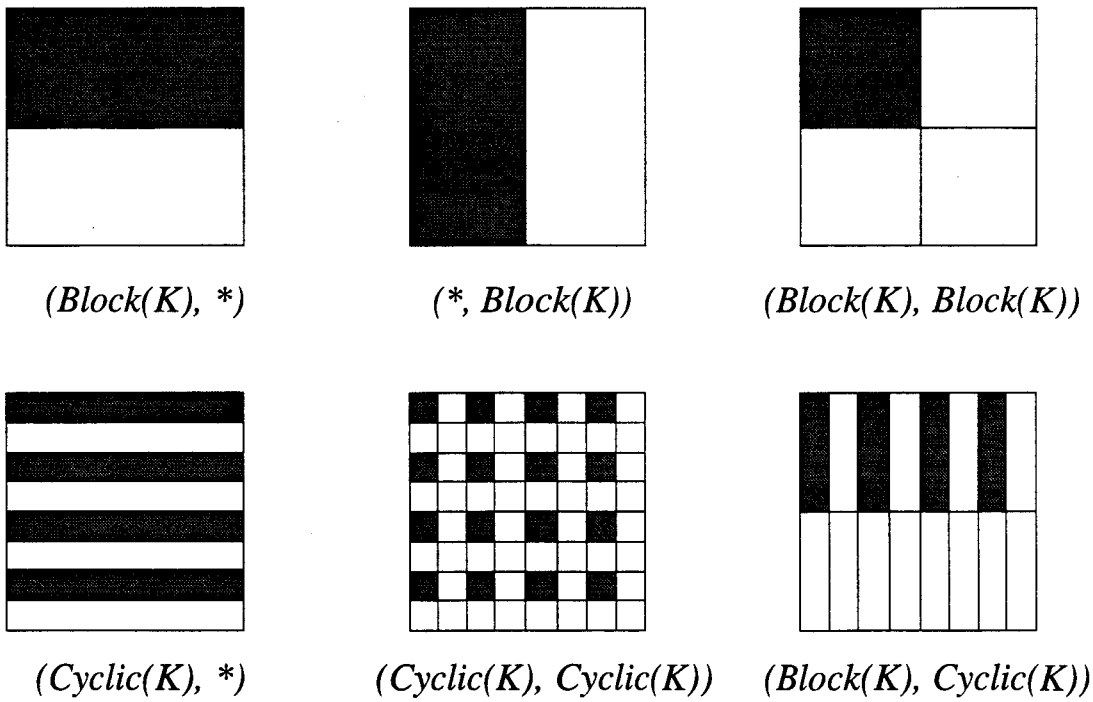


FIG. 2.1 - *Distributions de données usuelles*

algorithmes spécialisés qui sont spécifiques à une redistribution donnée.

Dans une seconde partie nous établirons le lien entre les redistributions et les opérations d'accès qui sont effectuées avec le modèle à objets partitionnés. Avec ce modèle les entrées/sorties peuvent être modélisées par une redistribution HPF. Nous montrerons cependant que le contexte d'utilisation impose la définition de nouveaux algorithmes.

1 Directives de distribution de données

Le développement d'un programme parallèle sur une machine à mémoire distribuée est un processus délicat. Le programmeur doit d'une part distribuer au maximum ces données pour favoriser le parallélisme et d'autre part limiter le nombre de communications qui dégradent les performances. La plupart des langages à parallélisme de données[1] proposent des directives pour distribuer les tableaux par blocs de tailles K sur les processeurs. Lorsque le nombre de blocs est inférieur au nombre de processeurs, chaque nœud de calcul dispose au plus d'un unique bloc de données. On parle alors de distribution « $Block(K)$ ». Dans le cas contraire, les blocs de taille K sont distribués cycliquement sur les processeurs. On parle alors de dis-

1. C'est notamment le cas du langage **High Performance Fortran** [32, 29]

tribution «*Cyclic*(K)». La figure (2.1) présente un tableau bidimentionnel distribué sur une grille carrée de quatre processeurs. Le symbole «*» est utilisé pour représenter les dimensions non distribuées (*Collapsed*). Les parties grisées sont placées sur le processeur de coordonnées $(0, 0)$.

2 Algorithmes de redistribution généraux

La plupart des algorithmes proposés ont besoin de connaître les paramètres de distribution à la compilation. Nous parlerons alors d'algorithme de redistribution **statique**. Les algorithmes statiques sont issues des travaux autour du langage HPF. Le programmeur aligne ses données sur les grilles virtuelles appelées «templates». Une première distribution est d'abord utilisée pour distribuer ces templates sur les processeurs virtuels. Une seconde distribution est ensuite nécessaire pour distribuer ces processeurs virtuels sur les processeurs physiques.

Les techniques de compilations statiques proposées sont basées sur des contraintes linéaires [3, 14], sur les équations diophantiennes [11], ou sur des représentations particulières [44, 43, 43, 50]. Ces algorithmes supportent plusieurs niveaux de distributions et gèrent les alignements de données mais ils ont besoin des paramètres de distribution à la compilation. Dans les prochains paragraphes nous verrons qu'il existe des algorithmes capables de redistribuer dynamiquement les données sans compilation préalable.

Algorithme dynamique général

Loïc Prylli et Bernard Tourancheau proposent des algorithmes qui redistribuent efficacement des «tableaux HPF» d'une grille de processeurs à une autre [24, 42]. Ces algorithmes n'ont pas besoin de connaître les paramètres de distributions à la compilation et sont utilisables avec toutes les distributions HPF. Ces techniques de redistributions ont été implantées dans la bibliothèque de calcul scientifique SCALAPACK [12, 23]. La méthode utilisée consiste à balayer simultanément et pour chaque couple de processeurs (p, p'), les indices des éléments placés sur les processeurs p et p'. Deux compteurs sont utilisés, l'un pour les éléments de p, l'autre pour ceux de p'. Ils sont incrémentés progressivement pour calculer la zone de recouvrement. Les données faisant parties de l'intersection de ces deux ensembles sont placées dans le message destiné à p'.

Pour être efficace de nombreuses optimisations sont proposées. Celles-ci concernent le calcul de la zone de recouvrement mais aussi la stratégie de communication. Pour réduire

le coût du calcul des messages, les auteurs montrent que l'intersection de deux intervalles $Cyclic(K), Cyclic(K')$ sur P et P' processeurs, est périodique de période $PPCM(KP, K'P')$. Il est donc possible de mémoriser cette séquence de façon à éviter le calcul complet. Pour réduire le coût des communications et éviter les problèmes de blocages dûs aux limitations des buffers, trois stratégies sont mises en places :

1. Communications synchrones : Cette stratégie consiste à utiliser deux «tapis roulants» de processeurs. À l'étape i, le processeur p communique avec le processeur cible d'indice $(P - p - i) mod P'$.
2. Communications asynchrones :
   - Calcul et réception asynchrone des messages à recevoir ;
   - Émission pour chaque destinataire des messages qui leurs sont destinés ;
   - Réception et rangement des messages.
3. Pipe-line de communications : Pour recouvrir les calculs et les communications, les messages sont fractionnés. Cette stratégie est implémentée avec les communications synchrones.

Les mesures de performances réalisées corroborent l'étude théorique : les temps de calcul des messages sont négligeables par rapport aux coûts des communications et du rangement des données en mémoire. Cette approche est très intéressante car elle prouve qu'il est possible de redistribuer efficacement les données à l'exécution quels que soient les paramètres de distribution source et destination. Il n'est donc plus nécessaire de se limiter au «HPF static subset». La généralité de la méthode proposée limite cependant les optimisations possibles :

- Il n'est pas possible de déterminer *a priori* les processeurs qui vont interagir. Il y a un calcul de recouvrement pour chaque paire de processeurs source et destination. La complexité de ce calcul est donc proportionnelle au nombre de processeurs source et destination. Dans la plupart des redistributions usuelles, un processeur source ne s'adresse qu'à un sous ensemble des processeurs de la machine d'arrivée. La technique proposée peut devenir coûteuse lorsque le nombre de processeurs utilisés est grand comme c'est le cas sur les machines massivement parallèles.
- Comme on ne connaît pas les couples de processeurs qui vont interagir, on ne peut pas planifier les communications. Il n'est donc pas possible de maximiser l'utilisation des ressources en choisissant un ordonnancement des messages approprié.

L'algorithme proposé est très efficace dans le cas général mais ne tire pas profit de toutes les spécificitées des redistributions particulières. Pour exploiter pleinement les propriétés des redistributions les plus courantes[2], il faut proposer des modélisations spécifiques et construire des algorithmes spécialisés.

3 Algorithmes de redistribution dynamiques spécialisés

Les travaux de Rajeev Thakur et Alok Choudhary [56, 59] et ceux de David W. Walker [60] ont montré que certaines redistributions particulières peuvent être réalisées à l'exécution avec un minimum de calculs.

3.1 Travaux de R. Thakur et A. Choudhary

Rajeev Thakur et Alok Choudhary ont développé des algorithmes spécialisés pour effectuer les redistributions $Block(K)$ vers $Cyclic(1)$, $Cyclic(1)$ vers $Block(K')$, $Cyclic(K)$ vers $Cyclic(TK)$ et $Cyclic(TK')$ vers $Cyclic(K')$[3].Ces algorithmes spécialisés redistribuent à l'exécution, un tableau unidimensionnel, sur une machine parallèle unique. Nous allons maintenant détailler le fonctionnement de ces algorithmes spécialisés.

- $Block(K)$ vers $Cyclic(1)$: En émission, chaque processeur calcule le destinataire du premier élément qu'il possède. Les éléments suivants sont placés dans les différents buffers à l'aide d'un simple calcul de modulo. Les messages sont ensuite reçus dans l'ordre (du processeur 1 au processeur P), ce qui permet de placer les données directement en mémoire sans effectuer de calcul d'adresse.
- $Cyclic(1)$ vers $Block(K')$: Les données à envoyer sont des tranches de tableaux. Pour chaque destinataire, on calcule l'indice du premier et du dernier élément et on place les données correspondantes dans les différents buffers. En réception les données sont placées en mémoire avec un pas égal au nombre de processeurs. Seule l'adresse du premier élément d'un message est calculée. Il est possible de recouvrir le rangement des données reçues avec les communications.
- $Cyclic(TK')$ vers $Cyclic(K')$: Les blocs sources sont divisés en T sous blocs de taille K'. Le destinataire p' de la première donnée est calculé, les T premiers sous blocs sont

2. Les redistributions $Cyclic(K)$ vers $Cyclic(K')$ où K et K' ne sont pas multiples entre eux sont assez rares

3. Les variables suivies du symbole «'» sont relatives à la distribution cible

alors placés dans les buffers à l'aide d'un calcul de modulo. Cette séquence se répète ensuite pour tout les autres blocs et peut donc être stockée. En réception deux cas sont à distinguer :

1. Si $T \leq P$ et P *modulo* $T = 0$; il suffit de calculer l'adresse de la première donnée d'un message puis de placer les données suivantes avec un pas égal à P. On peut donc recouvrir les communications et le rangement en mémoire.

2. Sinon ; il faut réceptionner tous les messages puis lire les données dans les différents buffers à l'aide d'une fonction modulo.

- $Cyclic(K)$ vers $Cyclic(TK)$: Il faut de nouveau distinguer deux cas :

1. Si $T \leq P$ et P *modulo* $T = 0$; il est possible de stocker la séquence des destinataires.

2. Sinon ; il faut calculer la destination de chacun des sous blocs.

En réception, les sous blocs sont placés en mémoire avec un pas égal à T.

La redistribution d'un tableau de dimension Φ, est soit réalisée par Φ redistributions successives (lorsque les redistributions intermédiaires font partie des cas étudiés), soit réalisée avec un algorithme de redistribution général mais peu efficace.

3.2 Travaux de David W. Walker et Steve W. Otto

David W. Walker et Steve W. Otto proposent deux algorithmes pour passer d'une distributions $Cyclic(K)$ à une distribution $Cyclic(TK)$ (et inversement) sur un unique ensemble de P processeurs. Dans ce cas particulier le schéma de communication se répète identique à lui même toutes les PKT données [44] que les auteurs appellent «superblock». Cette propriété est utilisée pour construire les messages. Tous les blocs de taille K ayant même déplacement dans un «superblock» sont placés dans un buffer puis envoyés à leur destinataire. Une redistribution nécessite donc T communications quel que soit la valeur de P. Le nombre de message n'est pas minimal.

3.2.1 Mise en œuvre

Les auteurs se sont particulièrement intéressés à la stratégie de communication. Trois implémentations sont développées et comparées :

Communications avec réception non bloquante – En deux temps (émission puis réception). Les processeurs réceptionnent les messages qui leur sont destinés après avoir posté toutes leurs données. Cette première stratégie permet de minimiser le coût des communications (recouvrement calculs/communications) mais elle nécessite un espace mémoire important (toutes les données sont recopiées dans les buffers).

- En alternant les émissions et les réceptions. À chaque étape, chaque processeur émet puis réceptionne un message. Cette seconde stratégie nécessite moins d'espace mémoire mais elle a un coût (en temps) plus important. Les messages ne sont pas équitablement répartis sur l'ensemble des processeurs. Certains processeurs doivent attendre que les autres aient terminé leur travail avant de recevoir des données.

Communications synchrones avec ordonnancement Cette stratégie a été développée pour palier aux défauts des deux autres. Pour éviter les déséquilibres de charges constatés avec la méthode précédente, il «suffit» de changer l'ordre d'émission des messages.

- Ordonnancement aléatoire. Il permet d'éviter les cas les plus défavorables. L'écart type diminue avec le nombre des messages.
- Ordonnancement optimal. Avec cette stratégie, chaque processeur réceptionne exactement un message à chacune des étapes. Le calcul de l'ordonnancement est basé sur le fait que le schéma de communication se répète identique à lui même à chaque superblock.

3.2.2 Performances

Des mesures ont été effectuées sur une IBM SP2 et sur une Paragon pour évaluer les différentes stratégies. Les meilleurs résultats sont obtenus avec la méthode asynchrone. L'algorithme synchrone sans ordonnancement est nettement moins performant (environ trois fois plus coûteux). Les performances de cet algorithme varie beaucoup en fonction de T car ce paramètre «fixe» le schéma de communication. Les performances obtenues avec l'ordonnancement aléatoire se rapprochent des résultats obtenus avec l'algorithme asynchrone. L'écart

diminue lorsque T augmente. Plus le nombre de messages est grand, meilleure est la répartition. Les performances obtenues avec l'ordonnancement optimal sont comparables a celles de l'algorithme asynchrone. Les messages sont équitablement répartis mais il n'y a plus de recouvrement calculs/communications.

3.3 Conclusion

Ces travaux sont intéressants car ils montrent que les redistributions n'ont pas toutes la même complexité. Cependant, les auteurs ne traitent pas les redistributions hétérogènes. Les redistributions sont réalisées sur une unique grille de processeurs. Chaque processeur est à la fois émetteur et récepteur. Ces algorithmes ne sont plus valables lorsque le nombre de processeurs sources est différent du nombre de processeurs destinataires.

4 Redistributions et entrées/sorties parallèles

Grâce aux performances des réseaux de communications, il est aujourd'hui possible de faire coopérer plusieurs machines parallèles à mémoire distribuées. Cette utilisation simultanée d'un ensemble de machines parallèles génère de très importants volumes de communications lors des migrations de données d'une machine à une autre. D'un point de vue conceptuel, ces migrations peuvent être visualisées comme des entrées/sorties parallèles. Le programmeur choisit la distribution sur les nœuds de calculs de sa machine et chaque processeur lit ou écrit ses données en fonction de cette distribution. Le système a alors la charge d'acheminer les données en fonction des distributions qui ont été choisies sur les machines sources et destinations.

Si le problème des entrées/sorties parallèles présente de nombreux points communs avec le problème du calcul des redistributions HPF, elles présentent cependant des particularités qui ne sont pas prises en compte par les algorithmes de redistributions «classiques» :

1. Dans le cadre des entrées/sorties parallèles, les paramètres de redistributions sont rarements disponibles à la compilation. Dans ce contexte, tous les calculs de redistributions sont exécutés dynamiquement à l'exécution.

2. Les entrées/sorties parallèles font interagir deux ensembles de processeurs distincts. Il est donc possible d'exploiter deux niveaux de parallélismes en effectuant les calculs sur les processeurs sources et destinations en parallèle.

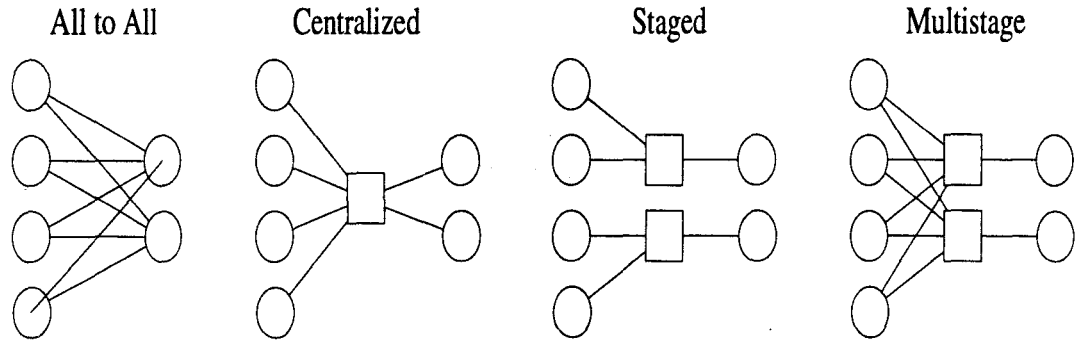


FIG. 2.2 - *Stratégies de communication*

3. Les communications réalisées sur une machine parallèle sont beaucoup moins pénalisantes que celles réalisées entre deux machines hétérogènes. Dans ce contexte, il est essentiel de recouvrir les calculs[4] avec les communications.

Certains travaux récents [2, 49, 48, 5, 58, 26, 10, 52, 53] s'intéressent aux entrées/sorties parallèles et aux redistributions dans les systèmes hétérogènes. La bibliothèque DAREL [30] construite au dessus de MPI permet de redistribuer un tableau multi-dimentionnel, d'un ensemble de processeurs vers un autre. Cette bibliothèque supporte toutes les redistributions car elle est basée sur des algorithmes généraux semblables à celui de la figure (3.3). Ceux ci ne sont hélas pas toujours très performants.

Une bibliothèque de redistribution a également été proposée à l'université de Syracuse [4]. Elle permet de choisir la stratégie utilisée en fonction des propriétés du réseau disponible (fig. 2.2). Les stratégies proposées sont les suivantes :

All to All : C'est une communication totale. Cette stratégie est intéressante lorsque la connectivité du réseau est importante et lorsque sa latence est faible.

Centralized : Les données sont accumulées sur un unique émetteur qui se charge ensuite de les redistribuer sur les récepteurs. Cet algorithme est intéressant lorsqu'on ne dispose que d'un unique lien de communication.

Staged : Les données sont envoyées sur quelques processeurs qui vont ensuite les réémettre à leurs destinataires finaux. Cette stratégie augmente le volume des communications (les données sont communiquées deux fois) mais elle réduit le nombre de messages échangés.

4. Calculs d'adresses, changements de format de donnée, ...

Multi-staged : Cette stratégie est un compromis entre les stratégie `All to All` et `Staged`.

Toutes ces stratégies sont implémentées dans une bibliothèque de communications. Celle-ci est utilisée par le compilateur Fortran M. Le langage Fortran M a été conçu pour exprimer le parallélisme de tâches [25, 26]. Avec ce langage le programmeur peut créer des processus dynamiquement puis les faire communiquer par l'intermédiaire de «`Channels`». En plus du parallélisme de tâches, le langage est conçu pour supporter le parallélisme de données. En particulier le programmeur peut spécifier la distribution de ces données sur des grilles de processeurs.

La figure (fig. 2.3) présente un exemple de code Fortran M. Le programme principal crée deux tâches nommées `sender` et `receiver` ainsi qu'un lien de communication entre ces deux tâches. Ce lien est défini en réunissant un port de sortie nommé «po» avec un port d'entrée appelé «pi». Ce lien de communication va permettre la migration de tableaux de taille 128 par 128. Le programme principal spécifie également les processeurs sur lesquels seront effectuées les tâches.

Les deux tâches sont exécutées sur respectivement seize et huit processeurs. Le producteur émet une séquence de dix tableaux distribués $(Block, *)$ à un consommateur qui les reçoit avec la distribution $(*, Block)$. On voit bien sur cet exemple, qu'une simple communication entre deux tâches à parallélisme de données nécessite des calculs de redistributions. Chacune de ces redistributions est réalisée par le système qui choisit la stratégie la plus adaptée. Une optimisation est mise en place pour les redistributions $(Block, *)$ vers $(*, Block)$. Il est dommage que les auteurs se soient limités à cette redistribution car il aurait été possible dans le cadre des migrations multi-étages de fixer les distributions intermédiaires de façon à traiter efficacement toutes les redistributions.

5 Conclusion

Une entrée/sortie parallèle permet de migrer des données d'une machine à mémoire distribuée à une autre. Ces migrations de données nécessitent des calculs de redistributions dynamiques et font interagir plusieurs ensembles de processeurs. Pour réduire au maximun le coût de ces migrations de données, il faut construire des algorithmes qui tirent profit de l'ensemble des ressources disponibles de façon à recouvrir au maximun les calculs et les communications.

Dans ce chapitre nous nous sommes particulièrement intéressés aux algorithmes de re-

```
      program example
!hpf  processors pr(24)
      INPORT (real x(128,128)) pi
      OUTPUT (real x(128,128)) po
      CHANNEL (in=pi, out=po)
      PROCESSES
         PROCESSCALL sender(po) SUBMACHINE(pr(1:16))
         PROCESSCALL receiver(pi) SUBMACHINE(pr(17:24))
      ENDPROCESSES

      PROCESS sender(po)
!hpf  processors pr(16)
      outport (real x(128,128)) po
      real a(128,128)
!hpf  distribute a(block,*)
      do i = 1,10
        call produce(a)
        SEND(po) a
      enddo
      ENDCHANNEL(po)
      end

      PROCESS receiver(pi)
!hpf  processors pr(8)
      inport (real x(128,128)) pi
      real b(128,128)
!hpf  distribute b(*,block)
      do i = 1,10
        RECEIVE(pi,end=10) b
      call use(b)
      enddo
      continue
      end
```

FIG. 2.3 - *Exemple de code Fortran M*

distributions dynamiques. Nous avons distingué les algorithmes généraux et les algorithmes spécifiques. L'approche proposée par Loïc Prylli et Bernard Tourancheau est intéressante car elle traite toutes les redistributions $Cylic(K)$ vers $Cyclic(K')$ avec un unique algorithme. Cependant en considérant le cas général les auteurs se sont privés des propriétés particulières qui existent pour certaines redistributions. En particulier il n'est pas possible dans le cas général d'énumérer les indices des processeurs qui vont interagir. Un calcul d'intersection est nécessaire pour chaque couple de processeurs source et destination.

Les algorithmes spécifiques qui ont été proposés ne sont pas utilisables lorsque le nombre de processeurs de la machine de départ est différent du nombre de processeurs de la machine d'arrivée et le nombre de cas traités est assez limité. L'étude des stratégies de communication réalisée par David Walker et Steve Otto a montré l'importance de l'ordonnancement des messages. Cette étude a été réalisée sur un unique ensemble de processeurs où il n'est par définition pas possible de recouvrir les calculs nécessaires pour créer les messages avec ceux à effectuer pour ranger les données en mémoire.

Pour mettre à profit toutes les propriétées particulières des redistributions spécifiques nous allons dans le prochain chapitre construire nos propres algorithmes de redistributions adaptés aux environements hétérogènes.

Chapitre 3

Algorithmes de redistributions spécialisés

Avec le modèle de fichier à objets partitionnés les entrées/sorties parallèles nécessitent un calcul de redistribution. Chaque accès aux fichiers fait interagir deux ensembles de processeurs distincts : les nœuds de calculs et les nœuds d'entrées/sorties. Pour tirer profit de toutes les ressources matérielles il faut utiliser en parallèle les nœuds de calcul et les nœuds d'entrées/sorties et recouvrir les calculs avec les communications. Il faut pour cela générer un code distinct pour chaque ensemble de processeurs. Les algorithmes de redistributions «standard» ne sont pas utilisables.

Dans ce chapitre nous proposerons une modélisation des redistributions HPF. Une étude théorique nous permettra d'évaluer le coût d'un algorithme de redistribution énumératif général ainsi que le coût d'un algorithme de redistribution idéal qui n'effectue que des communications. De façon générale, une redistribution est une opération complexe mais la plupart des redistributions usuelles acceptent une modélisation simplifiée. À partir de ces modèles nous construirons des algorithmes de redistributions spécifiques capables d'effectuer dynamiquement une redistribution entre deux ensembles de processeurs distincts en recouvrant les calculs avec les communications.

Dans la partie expérimentale nous montrerons l'intérêt mais aussi les limites des algorithmes proposés. En particulier nous mettrons en évidence des problèmes de répartition de charges sur la machine cible liés à un mauvais ordonnancement des communications. Nous tirerons une nouvelle fois profit de la simplicité des équations de redistribution pour calculer un ordonnancement qui minimise les déséquilibres de charges.

T	tableau global distribué	x	déplacement dans un bloc
N	taille du tableau global	t_{pk}	temps de **packing** d'une donnée
P	nombre de processeurs	t_{upk}	temps de **unpacking** d'une donnée
K	taille des blocs	t_c	temps pour migrer une donnée
p	numéro du processeur	t_{adr}	temps pour calculer une adresse
n	nombre de cycles	φ	nombre de liens de communication

FIG. 3.1 - *Notations et définitions*

1 Modélisation des redistributions hétérogènes

Dans cette section, nous proposerons une modélisation des redistributions hétérogènes. Celle-ci sera ensuite utilisée pour construire les différents algorithmes présentés dans ce chapitre.

1.1 Notations et définitions

Les notations utilisées dans cette thèse sont données en figure (3.1). Les tableaux et les processeurs sont indicés à partir de zéro. Toutes les valeurs constantes pour une redistribution donnée sont notées en majuscules. Les paramètres concernant la machine cible seront toujours suivis du symbole «'». Le symbole «/» est utilisé pour désigner la division entière entre deux nombres positifs. Nous utiliserons également les fonctions suivantes :

$$Div(a,b) = \begin{cases} (a+b-1)/b & \text{si } 0 \leq a \\ -\,(-a/b) & \text{sinon} \end{cases} \tag{3.1}$$

$$L(p) = \begin{cases} (N/PK)K & \text{si } (N/K) mod P < p \\ (1+N/PK)K & \text{sinon} \end{cases} \tag{3.2}$$

$$First(a,b) = \text{b } div(a,b) - a \tag{3.3}$$

La fonction Div calcule la division arrondie à l'entier supérieur. $First$ calcule le plus petit entier positif x, qu'il faut ajouter à l'entier a, pour que $a+x$ soit un multiple de b (où b est un entier positif). La fonction $L(p)$ calcule la taille du tableau local au processeur p.

1.2 Résolution d'une équation diophantienne à deux inconnues

Pour mettre en œuvre nos algorithmes de redistribution, nous serons amenés à résoudre des équations diophantiennes à deux inconnues de la forme suivante :

$$Ax + By = C \tag{3.4}$$

où A, B, C , x et y sont des entiers naturels. Nous utiliserons pour cela la méthode de résolution suivante : Soit α, β, et D tels que $\alpha A + \beta B = D$ avec $D = PGCD(A, B)$. Les valeurs de α, β et D sont obtenues à l'aide de l'algorithme d'Euclide étendu. La complexité de cet algorithme est du même ordre que celle de l'algorithme d'Euclide classique qui est utilisé pour calculer le PGCD de deux entiers naturels. Les solutions de l'équation (3.4) sont obtenues en additionnant la solution particulière

$$(\alpha\tfrac{C}{D} , \beta\tfrac{C}{D})$$

à la solution générale de l'équation

$$Ax + By = 0$$

dont les solutions sont les couples (x, y) suivants:

$$\begin{cases} x = \frac{B}{D}t \\ y = -\frac{A}{D}t \end{cases}$$

avec $t \in \mathbb{Z}$. Les solutions de l'équation (3.4) sont les couples (x, y) qui vérifient :

$$\begin{cases} x = \frac{\alpha C + Bt}{D} \\ y = \frac{\beta c - At}{D} \end{cases} \tag{3.5}$$

1.3 Représentation mémoire et fonctions d'adressage

Lorsqu'on distribue un tableau T de taille N sur P processeurs selon la directive $Cyclic(K)$, chaque processeur possède son propre tableau local Tl constitué d'une suite de

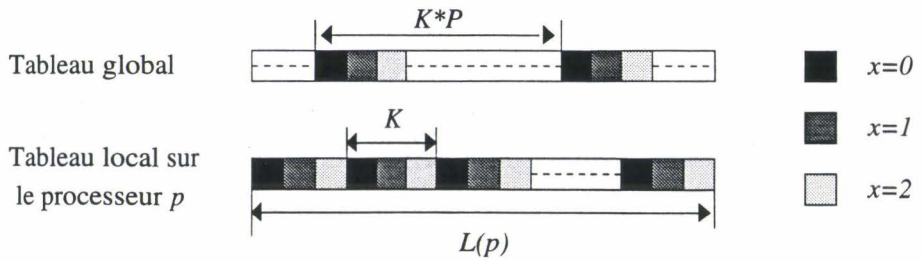


FIG. 3.2 - *Tableau distribué Cyclic*(3)

blocs de taille K (fig. 3.2). À chaque donnée de rang i dans le tableau T, on associe son adresse d'implantation dans un tableau local Tl définie par un n-uplet (p, n, x), où p est le numéro du processeur qui possède la donnée de rang i, n le numéro de bloc dans Tl et x sa position dans ce bloc. Ces valeurs sont calculées de la façon suivante :

$$\begin{cases} p = (i/k) \bmod P \\ n = i/PK \\ x = i \bmod K \end{cases} \Rightarrow T(i) \text{ est sur } p \text{ ssi } \boxed{i = pK + nPK + x} \tag{3.6}$$

$$\text{avec } x \in [0, K[,\ n \in [0, L(p)/K[\text{ et } p \in [0, P[.$$

À l'aide de cette fonction d'adressage, nous allons maintenant proposer une modélisation des redistributions $Cyclic(K)$ vers $Cyclic(K')$. Celle-ci va nous permettre de caractériser l'ensemble des éléments qu'il va falloir migrer entre chaque couple de processeurs.

Remarque Il est important de noter que la distribution $Block(K)$ est un cas particulier de la distribution $Cyclic(K)$ dans lequel chaque processeur ne dispose que d'un unique «bloc» de données[1]. L'équation (3.6) est donc également valable pour les distributions par blocs.

1.4 Équation générale

Une redistribution hétérogène fait interagir deux ensembles de processeurs distincts. Elle permet de passer d'une distribution D, $Cyclic(K)$ sur P processeurs, à une distribution D', $Cyclic(K')$ sur P' processeurs.

1. Il est donc possible de poser n à zéro.

Soit $E_D(p)$ l'ensemble des indices des éléments de T placés sur le processeur p, avec la distribution source D, et $E'_{D'}(p')$, l'ensemble des indices des éléments de T placés sur le processeur p', avec la distribution destination D'. Pour passer de la distribution D à la distribution D', il faut calculer pour chacun des couples (p, p'), l'ensemble $I(p, p')$ des indices des éléments de T, qui seront migrés du processeur p au processeur p'. L'élément d'indice i appartient à $I(p, p')$ s'il est inclus dans les ensembles $E_D(p)$ et $E'_{D'}(p')$. D'après l'équation (3.6), les éléments de $I(p, p')$ vérifient l'équation diophantienne suivante :

$$\boxed{pK + nPK + x \;=\; p'K' + n'P'K' + x'} \;\wedge\; \begin{cases} p \in [0, P[\\ n \in [0, L(p)/K[\\ x \in [0, K[\\ p' \in [0, P'[\\ n' \in [0, L'(p')/K'[\\ x' \in [0, K'[\end{cases} \tag{3.7}$$

1.5 Algorithme énumératif général

Il est possible de calculer les solutions de ce système à l'aide d'un simple algorithme énumératif SPMD. Dans cette approche, chaque processeur source p calcule avec la distribution initiale D, les indices globaux des éléments qu'il possède. Il utilise alors ces indices pour calculer, avec une distribution D', les numéros des processeurs avec lesquels il va devoir communiquer. À chaque couple de processeur (p, p') est associé un tampon dans lequel le processeur p place à l'aide des procédures `setbuf`[2], puis `pk`[3] les valeurs locales destinées au processeur p'. Ces tampons sont envoyés à leurs destinataires lorsque toutes les adresses des éléments de p ont été calculées. Les processeurs cibles réceptionnent leurs P buffers puis effectuent le travail inverse de façon à placer les données reçues dans la mémoire locale.

1.6 Limites de l'algorithme

L'algorithme dynamique[4] présenté figure (3.3) est utilisable pour toutes les redistributions $Cyclic(K)$ vers $Cyclic(K')$ et pour des ensembles de processeurs quelconques ($P \neq P'$). Cet algorithme est malheureusement très coûteux puisqu'il nécessite un calcul d'adresse pour

2. Cette procédure fixe le buffer courant.
3. Les paramètres de cette procédure sont l'offset du premier élément, le nombre d'élément, et le pas.
4. Les paramètres de redistribution n'ont pas à être connu à la compilation.

Algorithme d'émission

```
for (i=0;i<P';i++)
  buf[i]=mkbuf(PvmDataRaw);

N_last=L/K;
K_last=L%K;

for (n=0; n < N_last ;n++){
  for (x=0; x < K; x++){
    i=p*K+n*P*K+x;
    p'=(i/K')%P';
    setsbuf(buf[p']);
    pk(&t[i],1,1);
  }
}

for (x=0; x < K_last; x++){
  i=p*K+n*P*K+x;
  p'=(i/K')%P';
  setsbuf(buf[p']);
  pk(&t[i],1,1);
}

for (p'=0;p'<P';p'++) {
  setsbuf(buf[p']);
  send(p');
}

for (p'=0;p'<P';p'++)
  freebuf(buf[p']);
```

Algorithme de réception

```
for (p=0;p<P;p++) {
  buf[p]=recv(p);
  setrbuf(0);
}

N'_last=L'/K';
K'_last=L'%K';

for (n'=0; n' < N'_last; n'++){
  for (x'=0; x' < K'; x'++){
    i=p'*K'+n'*P'*K'+x';
    p=(i/K)%P;
    setrbuf(buf[p]);
    upk(&t[i],1,1);
  }
}

for (x'=0; x' < K'_last; x'++){
  i=p'*K'+n'*P'*K'+x';
  p=(i/K)%P;
  setrbuf(buf[p]);
  upk(&t[i],1,1);
}

for (p=0;p<P;p++)
  freebuf(buf[p]);
```

FIG. 3.3 - *Algorithme énumératif de redistribution Cyclic(K) vers Cyclic(K')*

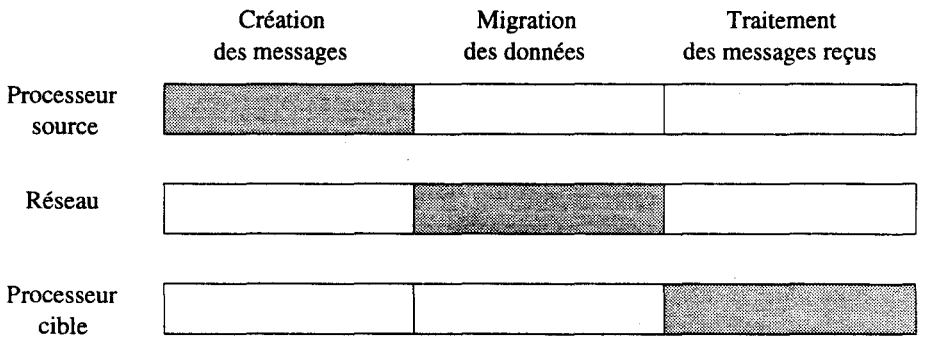


FIG. 3.4 - *redistribution séquentielle*

chaque élément de T. De plus il utilise très mal les ressources disponibles : la redistribution est réalisée en trois étapes successives (fig 3.4). Une première étape est nécessaire pour construire les buffers. Lors de cette première phase, le réseau et les processeurs cibles sont inutilisés. Une seconde étape permet alors de migrer les buffers des processeurs sources aux processeurs destinations. Toutes les données sont envoyées en même temps sur le réseau, et tous les processeurs sont inactifs. Enfin, une dernière étape est alors nécessaire pour ranger les données reçues en mémoire. Ce sont alors les processeurs sources qui restent inactifs. Les communications ne sont pas recouvertes par les calculs et il y a toujours au moins une des deux machines inactive. Dans la section suivante nous proposerons une stratégie de recouvrement adaptée à ce type d'environnement.

2 Stratégie de recouvrement

Sur les machines parallèles comme sur les réseaux locaux, le coût des communications est proportionnel au nombre de messages envoyés. Pour minimiser ce coût lors d'une redistribution, il est nécessaire de bufferiser les messages de façon à n'effectuer qu'une unique communication par couple de processeurs (p, p'). Cette bufferisation, qui est utilisée dans l'algorithme énumératif (fig. 3.3), séquentialise la redistribution. Pour recouvrir les calculs avec les communications, les buffers doivent être calculés indépendamment les uns des autres. Un processeur p peut alors créer puis envoyer l'ensemble $I(p, p')$, des données destinées au processeur p', sans avoir à parcourir toutes les valeurs locales. À chaque étape élémentaire, le processeur p crée un message et l'envoie à son destinataire. Les calculs sont ainsi réalisés en parallèle sur les processeurs sources et destinations. Cela permet de plus de recouvrir les communications par les calculs (fig. 3.5).

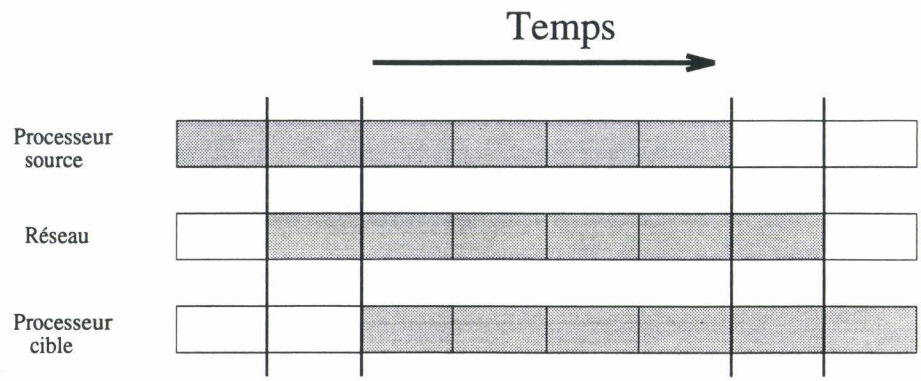


FIG. 3.5 - *Stratégie de recouvrement*

2.1 Mise en œuvre du recouvrement

La méthode de résolution que nous avons utilisée pour construire nos algorithmes, consiste à considérer les cas particuliers dans lesquels une ou plusieurs variables de l'équation (3.7) s'annulent. Dans chacun de ces cas, nous simplifions le système de façon à pouvoir énumérer les solutions de l'équation diophantienne à l'aide d'un ensemble de sections régulières de la forme $(debut : fin : pas)$. Ces sections régulières sont utilisées pour balayer les couples de processeurs qui interagissent et pour construire les ensembles de données à migrer entre chacun de ces couples. En appliquant systématiquement cette technique, nous avons construit des algorithmes qui effectuent des redistributions spécifiques. Tous ces algorithmes bénéficient des propriétés suivantes :

- **Le nombre de message est minimal :** on utilise un unique message par couple de processeurs (p, p') et on ne considère que les couples de processeurs qui interagissent ;
- **Le volume des communications est minimal :** seules les données de T sont communiquées (pour être asynchrone on utilise également l'indice du processeur source) ;
- **Les calculs sont réalisés dynamiquement :** il n'est pas nécessaire de connaître les paramètres de distribution à la compilation ;
- **Le coût du calcul d'adresse ne dépend que des paramètres de distribution :** il ne dépend pas de la taille N du tableau [5] ;
- **Les calculs sont recouverts par les communications :** à chaque étape élémentaire, un message est créé puis envoyé à son destinataire ;

5. Le coût pour placer les données dans le buffer dépend linéairement de la taille des données locales.

Variables nulles	Algorithmes de Redistributions	Pas des sections régulières
n, n'	$Block(K)$ vers $Block(K')$	$\Delta x = \Delta x' = \Delta p' = 1$
n	$Block(K)$ vers $Cyclic(K')$	$\Delta x' = \Delta n' = \Delta p' = 1$ $\Delta x = P'K'$
n'	$Cyclic(K)$ vers $Block(K')$	$\Delta x = \Delta n' = \Delta p' = 1$ $\Delta x' = PK$
x'	$Cyclic(TK')$ vers $Cyclic(K')$	$\Delta n = \mathcal{F}_1(PT, P')$ $\Delta n' = \mathcal{F}_2(PT, P')$ $\Delta p' = \mathcal{F}_3(PT, P')$
x	$Cyclic(K)$ vers $Cyclic(TK)$	$\Delta n = \mathcal{F}_4(P, P'T)$ $\Delta n' = \mathcal{F}_5(P, P'T)$ $\Delta p' = \mathcal{F}_6(P, P'T)$

FIG. 3.6 - *Algorithmes de redistributions développés*

- **On exploite deux niveaux de parallélisme :** les programmes SPMD d'émission et de réception s'exécutent en parallèle.

La figure (3.6) présente une vue globale des différents cas étudiés. Ces redistributions seront détaillées dans la section suivante.

2.2 Étude théorique du coût d'une redistribution

Pour évaluer le gain théorique maximal que l'on peut espérer obtenir en utilisant des algorithmes spécialisés, nous allons dans cette section modéliser le coût d'une redistribution effectuée avec l'algorithme énumératif, ainsi que le coût d'une redistribution idéale qui n'effectue pas de calcul d'adresse, et qui utilise au mieux toutes les ressources disponibles.

2.2.1 Coût de l'algorithme énumératif

Comme nous l'avons vu précédemment, l'algorithme énumératif, réalise la redistribution en trois étapes successives. Le coût global d'une redistribution avec cet algorithme est donc égal à la somme de chacune des étapes : calcul des adresses et placement dans les buffers puis migration des données et enfin rangement des données reçues en mémoire.

$$T_{enumeratif} = \frac{N(t_{adr} + t_{pk})}{P} + \frac{t_c N}{\varphi} + \frac{(t_{adr} + t_{upk})N}{P'} \tag{3.8}$$

Le coût pour un tableau T de taille N est donné par l'équation (3.8). Le temps pour ranger les données dans le buffer est proportionnel à la taille du tableau local sur les processeurs sources. Le coût de la migration dépend de la taille des données et du nombre de liens de communication disponibles (appelé φ). Enfin le coût pour ranger les données en mémoire décroît lorsque la taille des tableaux sur les processeurs cibles diminue. Pour simplifier la prise en compte des communications nous avons approximé le temps de transfert des données sur un lien par la fonction $\mathcal{F} = t_c N$. Cette approximation est satisfaisante lorsque la taille des messages échangés est suffisante pour que l'on puisse négliger le «start-up» (les tests présentés à la fin de ce chapitre ont été réalisés avec $N = 4.10^6$ entiers). Pour obtenir l'équation (3.8), nous avons posé les hypothèses suivantes :

1. Les données sont équitablement réparties sur les processeurs.
2. L'algorithme utilise «au mieux» les liens de communication disponibles.

La première hypothèse se justifie par le fait que nous étudions des distributions régulières avec lesquelles tous les processeurs ont des volumes de données comparables[6]. La seconde hypothèse est beaucoup moins réaliste : on suppose que l'ordre d'émission des messages est tel qu'il minimise le nombre de collisions. Comme on ne connaît pas «à priori» le volume des données qui seront communiquées entre chaque couple de processeurs, il n'est pas possible de calculer le nombre de liens de communication qui seront effectivement utilisés à un instant donné. Nous avons choisi de négliger ces conflits et de ne retenir que le nombre de liens de communication physiques de façon à calculer le temps de redistribution minimal lorsqu'on utilise l'algorithme énumératif.

2.2.2 Coût d'un algorithme idéal

L'algorithme idéal n'effectue pas de calcul d'adresse et utilise en parallèle toutes les ressources disponibles à l'aide d'un «pipeline d'exécution». Le temps global pour réaliser cette redistribution idéale est fonction du temps de traversée du pipeline. Il y a tout d'abord une période de remplissage pendant laquelle un premier message traverse les différents étages. À l'issue de cette initialisation, un message est traité à chaque étape élémentaire. La durée de ces étapes élémentaires dépend de l'étage le plus coûteux. La qualité du recouvrement obtenu est fonction de la capacité du réseau et des performances des machines. Elle est maximale lorsque tous les étages du pipe-line ont des coûts comparables.

6. Nous ne nous intéressons pas aux cas «limites» tels que *Collapsed* ou *Block*(K) avec $PK \gg N$.

$$t_{idéal} = \frac{t_{pk}N}{PP'} + \frac{t_cN}{\varphi P'} + \frac{t_{upk}N}{P'P'} + (P' - 1) * max(\frac{t_{pk}N}{PP'}, \frac{t_cN}{\varphi P'}, \frac{t_{upk}N}{P'P'}) \quad (3.9)$$

Pour obtenir l'équation (3.9) nous supposons que chaque processeur source s'adresse à chacun des processeurs cibles. Ce cas est le plus favorable puisqu'il minimise l'impact du remplissage. Cette hypothèse est discutable car le nombre de correspondants dépend de la redistribution étudiée, mais elle permet d'obtenir une équation simple et générale.

2.2.3 Gain

Nous définissons le gain comme étant égal au temps nécessaire pour effectuer une redistribution avec l'algorithme énumératif, divisé par le temps nécessaire pour réaliser la même redistribution avec l'algorithme spécialisé correspondant.

$$Gain = \frac{t_{énumératif}}{t_{spécialisé}} \quad (3.10)$$

2.3 Courbes théoriques sur la ferme d'ALPHA

Nous disposons d'une ferme de processeurs ALPHA reliés à un bus Ethernet ainsi qu'à un crossbar par seize anneaux FDDI. Pour évaluer les coûts des différents algorithmes ainsi que le gain que nous pouvons espérer obtenir sur cette machine nous avons mesuré avec PVM et en utilisant l'option `PvmDataRaw`[7] les valeurs des constantes qui caractérisent la machine utilisée. Les valeurs obtenues en micro secondes par entier sont les suivantes :

Calcul d'adresse : C'est le temps nécessaire pour calculer une adresse avec l'algorithme énumératif : $t_{adr} = 2.07$.

Gestion des buffers : Avec l'algorithme énumératif, les données sont rangées une à une dans des buffers distincts. Il faut deux appels de fonction pour placer un unique entier[8]. Les temps obtenus avec l'algorithme énumératif sont : $t_{pk} = 6.07$, $t_{upk} = 3.9$. Avec l'algorithme idéal, on suppose que l'on connaît la taille des messages *a priori* et que chaque message est construit à l'aide d'une unique tranche de tableau. Cela n'est effectivement possible que dans le cas de la redistribution $Block(K)$ vers $Block(K')$, dans les autres

7. Lorsqu'on utilise cette option les données sont recopiées dans le buffer sans subir de conversion XDR.
8. Il faut tout d'abord sélectionner le buffer puis ranger la donnée courante

cas les données à envoyer ne sont pas contiguës. Avec l'algorithme idéal, il suffit d'un unique appel de fonction pour placer toutes les données du message courant dans le buffer. Les temps mesurés dans ce cas sont : $T_{pk} = T_{upk} = 0.12$.

Migration des données : Les temps pour communiquer un entier (32 bits) sont respectivement de $t_c = 0.7$ (ce qui correspond à un débit d'environ 45,7 Mbits/s) sur le réseau FDDI, et de $T_c = 3.53$ avec Ethernet (ce qui correspond à un débit de 9.04Mbits/s).

Toutes les courbes théoriques ont été construites à l'aide de ces valeurs et des fonctions (3.8, 3.9, 3.10), en faisant varier le nombre de processeurs sources et destinations de un à huit. Nous comparerons ces courbes théoriques aux résultats réels dans la section (4).

2.3.1 Temps théorique avec le «crossbar» FDDI

Lorsque nous utilisons le «crossbar», le réseau est sans conflit et $\varphi = min(P, P')$. Les figures (3.7) et (3.8) représentent le temps nécessaire pour effectuer la redistribution avec l'algorithme énumératif ainsi que le gain obtenu avec un algorithme idéal.

Les temps d'exécution diminuent linéairement lorsqu'on augmente le nombre de processeurs. Cela est rendu possible par la présence du «crossbar» qui permet de conserver un certain équilibre entre la puissance de calcul de la machine et la capacité de son réseau de communication. Le gain est maximal lorsque $P = P'$ et décroît fortement lorsque le nombre de processeurs sources s'écarte du nombre de processeurs cibles. Cela s'explique par la prépondérance du coût des communications : $t_c = 0.7 >> T_{pk} = T_{upk} = 0.12$. Dans ces conditions le recouvrement des différentes phases n'a que peu d'importance (au maximum 30%). Le temps d'exécution est directement proportionnel au nombre de liens de communication disponibles. Comme $\varphi = min(P, P')$, le temps est minimal lorsque $P = P' = 8$.

2.3.2 Temps théorique avec le bus Ethernet

Lorsqu'on utilise le réseau Ethernet, on ne dispose que d'un unique lien de communication quels que soient les nombres de processeurs sources et destinations. Les figures (3.9) et (3.10) représentent les temps nécessaires pour effectuer la redistribution avec l'algorithme énumératif ainsi que le gain obtenu avec un algorithme idéal lorsqu'on utilise ce réseau.

On constate cette fois que les temps d'exécution ne diminuent que faiblement lorsqu'on augmente les nombres de processeurs. Cela s'explique par la présence d'un goulot d'étran-

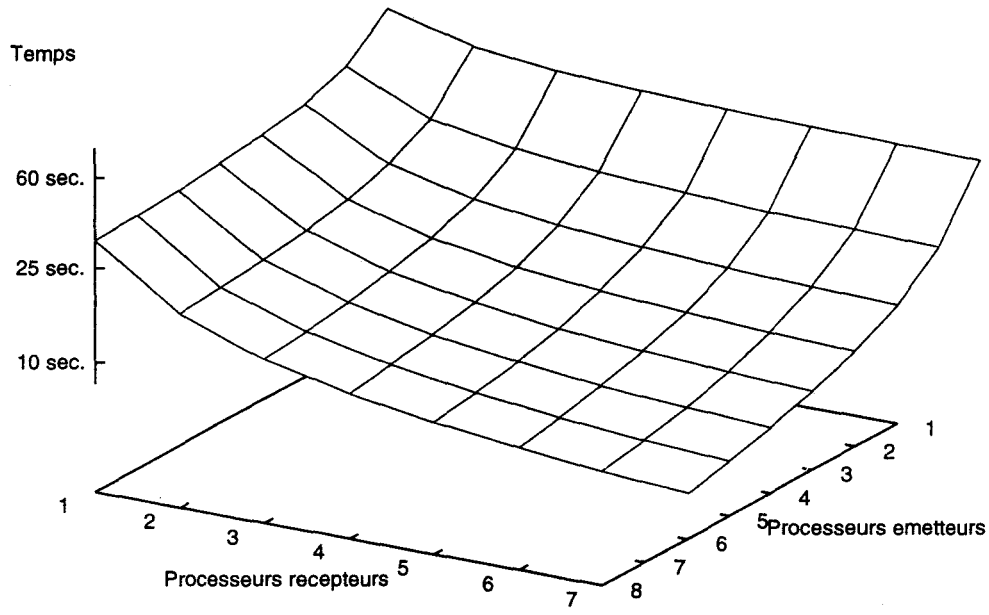


FIG. 3.7 - *Temps théorique avec l'algorithme énumératif et le crossbar FDDI*

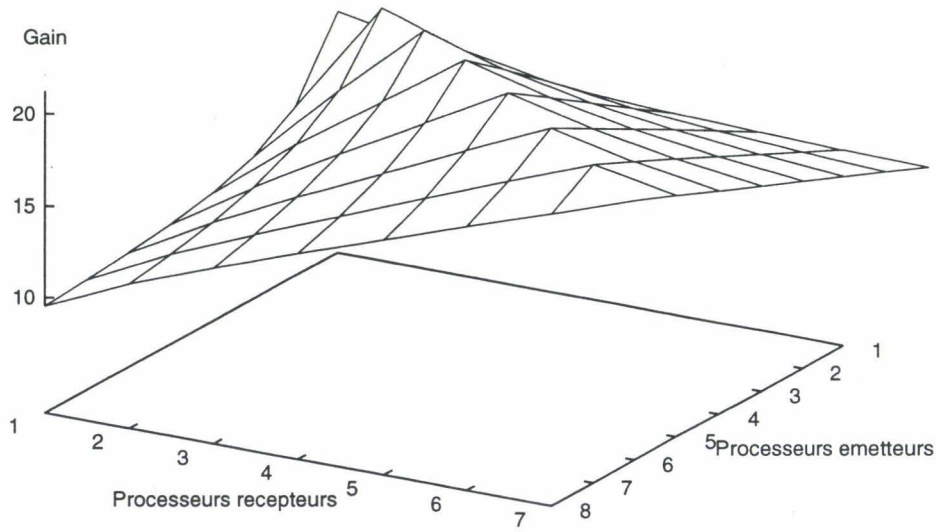


FIG. 3.8 - *Gain théorique avec le crossbar FDDI*

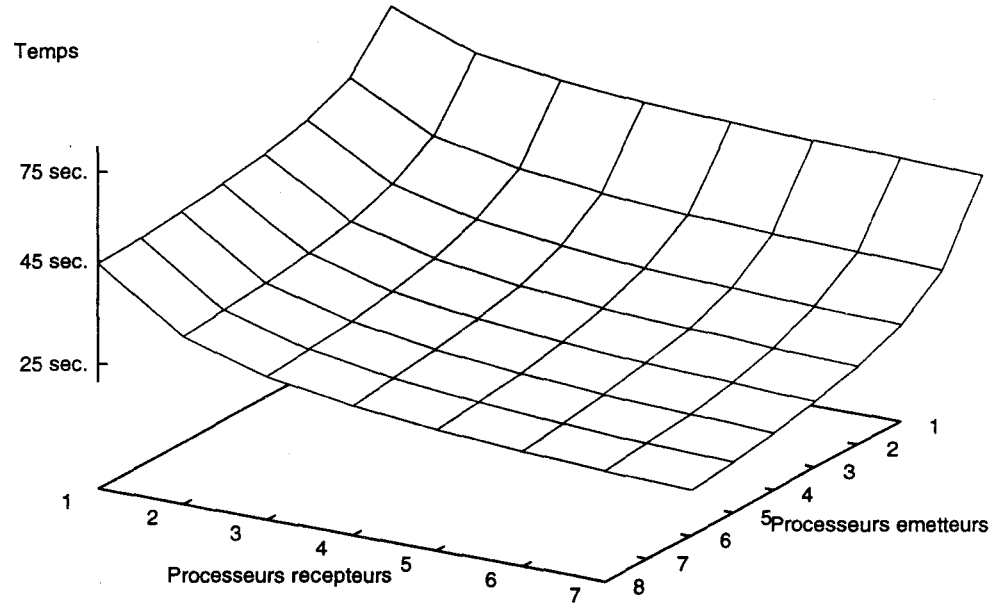


FIG. 3.9 - *Temps théorique avec l'algorithme énumératif sur le bus Ethernet*

glement : le coût des communications est encore plus important que dans le cas précédent et celui-ci reste constant quelque soit le nombre de processeurs. De ce fait, le gain est maximal lorsque le nombre de processeurs est minimal ($P = P' = 1$). On tire alors au maximum profit de l'absence de calculs d'adresses et de la bonne utilisation des buffers. Dans ce cas le gain est d'environ quatre. Le gain tend ensuite vers un lorsque le nombre de processeurs augmente.

2.4 Conclusion

Cette étude théorique a montré qu'il est possible d'obtenir des gains substantiels à l'aide d'algorithmes spécialisés à condition de disposer d'un «bon réseau» de communications. Dans le cas contraire le bénéfice potentiel disparaît lorsque le nombre de processeurs augmente. Par ailleurs, il est important de noter qu'il ne suffit pas de disposer de φ liens de communication pour diminuer d'autant le coût de la migration de données. L'utilisation optimale des liens de communication physiques nécessite le calcul préalable d'un ordonnancement afin de répartir au mieux les messages sur l'ensemble des destinataires.

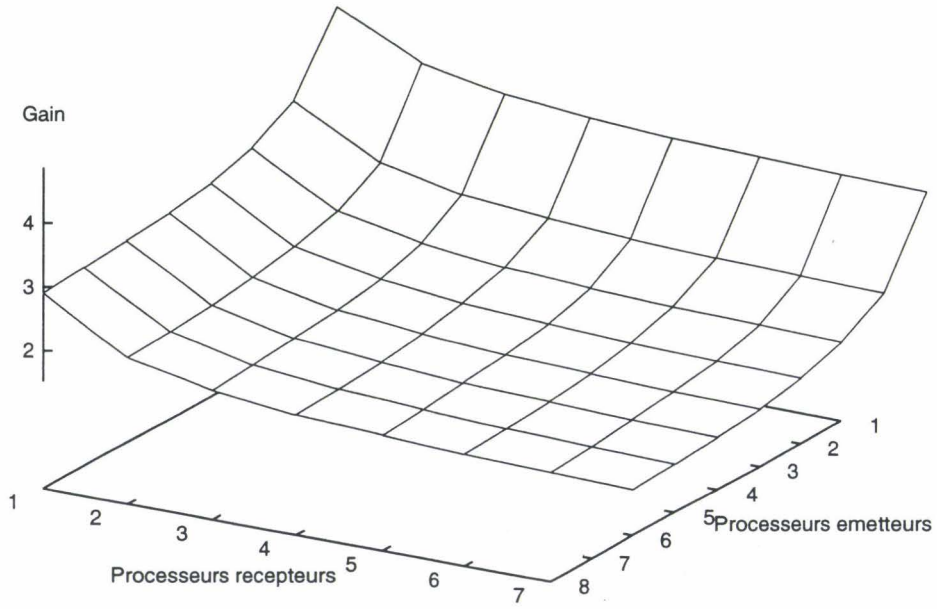


FIG. 3.10 - *Gain théorique avec le bus Ethernet*

3 Redistributions unidimensionnelles

Dans cette section nous allons étudier les cas particuliers dans lesquels une ou plusieurs variables de l'équation (3.7) s'annulent. Dans chacun des cas nous construirons l'algorithme de redistribution spécialisé correspondant.

3.1 Redistribution Block(K) vers Block(K')

Lorsqu'on utilise une distribution par blocs, les variables n, et n' sont nulles. L'équation (3.7) se simplifie et on obtient l'équation suivante :

$$i \in I(p,p') \text{ ssi il existe } x,x' \text{ tels que } \boxed{pK + x = p'K' + x'} \tag{3.11}$$

avec

$$\begin{cases} p \in [0,P[\\ p' \in [0,P'[\end{cases} \text{ et } \begin{cases} x \in [0,K[\\ x' \in [0,K'[\end{cases}$$

3.1.1 Solutions de l'équation

Avec l'équation (3.11) nous allons maintenant calculer les couples de processeurs qui vont interagir ainsi que les ensembles de données à migrer. Les indices des éléments de T placés sur le processeur p avec la distribution source sont dans l'intervalle : $[début, fin]$ avec $début = pK$ et $fin = pK + L(p) - 1$. Ces éléments sont envoyés aux processeurs $p' \in [début/K', fin/K']$. De même, les éléments de T placés sur le processeur p' appartiennent à l'intervalle : $[début', fin']$ avec $début' = p'K'$ et $fin' = p'K' + L'(p') - 1$. Ces éléments sont envoyés par les processeurs d'indices $p \in [début'/K, fin'/K]$. Comme les processeurs ne disposent que d'un unique bloc de données, il suffit pour identifier une donnée, de calculer son déplacement (x ou x') dans le tableau local. Il n'est pas nécessaire de calculer des indices de blocs n ou n'. Ces déplacements vérifient :

- Sur les processeurs sources :

$$\begin{cases} 0 \leq x \leq K-1 \\ p'K' - pK \leq x \leq p'K' - pK + K' - 1 \end{cases}$$

Algorithme d'émission

```
p'_f=p*K/K';
p'_l=min((p*K+L-1)/K',(N-1)/K');

for (p'=p'_f;p'<=p'_l;p'++){
  initsend();
  x_f=max(p'*K'-p*K, 0);
  x_l=min(p'*K'-p*K+K', K)-1;
  nb=x_l-x_f+1;
  pk(&TAB[x_first],nb,1);
  send(p');
}
```

Algorithme de réception

```
p_f=(K'*p')/K;
p_l=min((K'*p'+L'-1)/K,(N-1)/K);
nb_p=p_l-p_f+1;

for (i=0;i<nb_p;i++){
  receive(&p);
  x'_f=max(p*K-p'*K',0);
  x'_l=min(p*K-p'*K'+K-1,L'-1);
  nb=x'_l-x'_f+1;
  upk(&TAB[x'_f],nb,1);
}
```

FIG. 3.11 - *Algorithme de redistribution Block(K) vers Block(K')*

- Sur les processeurs cibles :

$$\begin{cases} 0 & \leq x' & \leq & K'-1 \\ pK-p'K' & \leq x' & \leq & pK-p'K'+K-1 \end{cases}$$

Les solutions de l'équation (3.11) vérifient donc les contraintes suivantes :

$$\begin{cases} p & \in & \mathcal{D}_p & = & [p'K'/K,\ min((p'K'+L'(p')-1)/K, (N-1)/K)] \\ p' & \in & \mathcal{D}_{p'} & = & [pK/K',\ min((pK+L(p)-1)/K', (N-1)/K')] \\ x & \in & \mathcal{D}_x & = & [max(p'K'-pK, 0),\ min(p'K'-pK+K', K)[\\ x' & \in & \mathcal{D}_{x'} & = & [max(pK-p'K', 0),\ min(pK-p'K'+K, K')[\end{cases} \quad (3.12)$$

3.1.2 Algorithme de redistribution *Block(K)* vers *Block(K')*

Les algorithmes (fig. 3.11) d'émission et de réception sont construits à l'aide des contraintes fixées sur les variables x, x', p, et p' (3.12). Un processeur source d'indice p s'adresse à tous les processeurs de l'intervalle $\mathcal{D}_{p'}$[9] pour leur envoyer les données dont les indices sont dans l'intervalle $\mathcal{D}_x$[10]. De même, un processeur cible d'indice p' réceptionne un message en provenance de chacun des processeurs de l'intervalle $\mathcal{D}_p$ dans lesquels sont placées toutes les

9. La valeur de p est alors connue.
10. On connaît alors les valeurs de p et de p'.

données dont les indices sont dans l'intervalle $\mathcal{D}_{x'}$. Les bornes des intervalles $\mathcal{D}_p$ et $\mathcal{D}_{x'}$ sont calculées à l'aide des contraintes (3.12).

En émission une boucle énumère tous les indices des processeurs p' avec lesquels p va communiquer. Pour chaque destinataire p', on calcule les déplacements dans le tableau source, de la première et de la dernière donnée à envoyer. Cela nécessite le calcul des bornes x_f et x_l de l'intervalle $\mathcal{D}_x$. La tranche de tableau $tl[x_f : x_l]$ est alors placée avec la fonction **pk** dans le buffer courant, puis envoyée à son destinataire (fonction **send**). En réception, une boucle est utilisée pour réceptionner les messages puis pour les placer en mémoire (fonction **upk**). Comme le nombre de message est minimal, le nombre de paquets à recevoir est exactement égal à la taille de l'intervalle $\mathcal{D}_p$. Les adresses mémoires cibles sont obtenues à l'aide de la variable x'.

Cet algorithme est très efficace. Le coût du calcul des tranches de tableaux est négligeable par rapport au coût des communications et les données à envoyer sont contiguës. Cela peut permettre une bonne utilisation de la mémoire cache en lecture comme en écriture. À la différence de l'algorithme général, on n'utilise ici qu'un unique buffer de façon à créer un «pipeline» d'exécution semblable à celui de la figure (3.5). Les communications sont recouvertes par les calculs et les algorithmes d'émission et de réception sont exécutés en parallèles.

3.1.3 Remarques

En fixant la variable K à N (resp. K' à N), cet algorithme permet d'effectuer la redistribution *Collapsed* vers $Block(K')$ (resp. $Block(K)$ vers *Collapsed*).

3.2 Redistribution $Block(K)$ vers $Cyclic(K')$

Là encore on utilise l'équation (3.7) mais cette fois, seule la variable n est fixée à zéro. On obtient alors l'équation suivante :

$$i \in I(p,p') \text{ ssi il existe } n', x, x' \text{ tels que } \boxed{n'P'K' = pK - p'K' + x - x'}$$

avec

$$\begin{cases} n' \in [0, L'(p')/K'[\\ x' \in [0, K'[\\ x \in [0, L(p)[\end{cases} \quad et \quad \begin{cases} p \in [0, P'[\\ p' \in [0, P[\end{cases} \tag{3.13}$$

3.2.1 Solutions de l'équation

L'équation (3.13) admet des solutions ssi $(pK - p'K' + x - x')$ est multiple de $P'K'$. Soit $\lambda \in \mathbb{N}$, les solutions de l'équation sont les n-uplets (p, p', x, x', n') qui vérifient les contraintes précédentes et qui s'écrivent :

$$x = First(pK - p'K' - x', P'K') + P'K'\lambda \tag{3.14}$$

$$n' = (pK - p'K' + x - x')/P'K' \tag{3.15}$$

Avec l'équation (3.14), le processeur p peut énumérer à l'aide d'une section régulière de pas $P'K'$ toutes les données qui seront placées sur le processeur p' à l'offset x'. En réception p' énumère les données reçues à l'aide de l'équation (3.15).

3.2.2 Algorithme de redistribution $Block(K)$ vers $Cyclic(K')$

Dans l'algorithme d'émission (fig. 3.13), la boucle externe énumère les nb'_p processeurs avec lesquels p va communiquer. On construit alors le message destiné au processeur p' à l'aide de K' sections régulières que l'on calcule avec l'équation (3.14). Ces sections régulières énumèrent

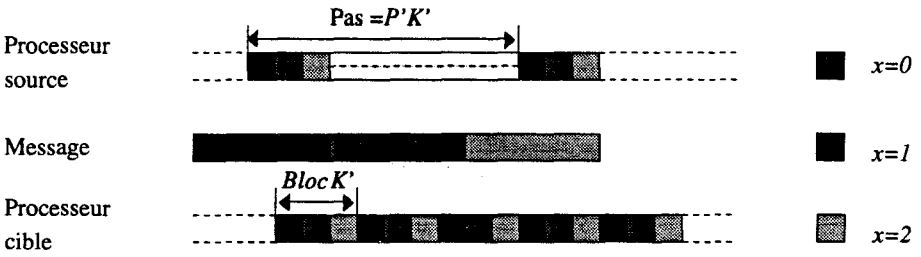


FIG. 3.12 - *Accès mémoire*

Algorithme d'émission

```
if (L == 0) return;
p_first=((p*K)/K')%P';
nb_p'=min(div(L+(p*K)%K',K'),P');
for (i=0; i<nb_p ; i++){
  p'=(p'_first+i)%P';
  initsend();
  for (xp=0;xp<K';x'++){
    x=first(p*K-p'*K'-x',P'*K');
    if (x<L) {
      nb=1+(L-x-1)/(P'*K');
      pk(&TAB[x],nb,P'*K');
    }
  }
  send(p');
}
```

Algorithme de réception

```
nb_recv=0;
while (nb_recv < L'){
  receive(&p);
  for (x'=0;x'<K';x'++){
    x=first(p*K-p'*K'-x',P'*K');
    np=(p*K-p'*K'+x-x')/(P'*K');
    if( K'*n'+x' < L'){
      nb=div( min(K,N-p*K)-x, P'*K');
      upk(&TAB[x'+n'*K'],nb,K');
      nb_recv+=nb;
    }
  }
}
```

FIG. 3.13 - *Algorithme de redistribution Block(K) vers Cyclic(K')*

toutes les données de p qui seront placées sur le processeur p', à un même déplacement dans des blocs contiguës (fig. 3.12). Lorsque toutes les valeurs destinées à p' ont été placées dans le buffer, celui-ci est envoyé à son destinataire. En réception, la boucle externe est utilisée pour recevoir les messages. Pour chacun d'entre eux, une boucle interne énumère les K' valeurs de x'. L'indice n' du premier bloc concerné est obtenu à l'aide de l'équation (3.15). Les données sont ensuite placées en mémoire à l'aide de la fonction **upk**.

3.2.3 Remarque

En fixant la variable K à N, cet algorithme permet d'effectuer la redistribution *Collapsed* vers $Cyclic(K')$.

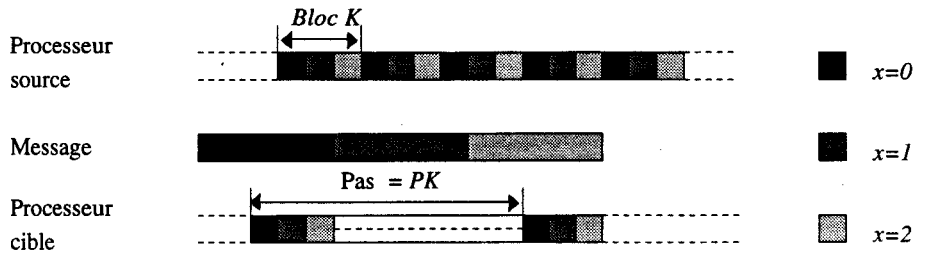


FIG. 3.14 - *Accès mémoire*

3.3 Redistribution $Cyclic(K)$ vers $Block(K')$

Ce cas est le symétrique du précédent. On pose $n' = 0$ et on simplifie l'équation générale. On obtient ainsi le système suivant :

$$i \in I(p,p') \text{ ssi il existe } n,x,x' \text{ tel que } \boxed{nPK = p'K' - pK + x' - x}$$

avec

$$\begin{cases} n & \in & [0, L(p)/K[\\ x' & \in & [0, K'[\\ x & \in & [0, L(p)[\end{cases} \quad \text{et} \quad \begin{cases} p & \in & [0, P'[\\ p' & \in & [0, P[\end{cases} \tag{3.16}$$

3.3.1 Solutions du système

L'équation (3.16) admet des solutions ssi $(p'K' - pK + x' - x)$ est multiple de PK. Soit $\lambda \in \mathbf{N}$, les solutions de l'équation sont les n-uplets (p,p',x,x',n) qui vérifient les contraintes précédentes et qui sont de la forme :

$$x' = First(p'K' - pK - x, PK) + PK\lambda \tag{3.17}$$

$$n = (p'K' - pK + x' - x)/PK \tag{3.18}$$

Algorithme d'émission

```
if (L == 0) return;
p'=(p*P'/P)%P';
envoyee=0;
while (envoyee != L){
  rangee=0;
  initsend();
  for (x=0;x<K;x++){
    xp=first(p'*K'-p*K-x,P*K);
    n=(p'*K'-p*K+x'-x)/(P*K);
    if (n*K+x < L){
      nb=min(div(L-n*K-x,K),
             div(K'-x', P*K));
      pk(&TAB[n*K+x],nb,K);
      rangee+=nb;
    }
  }
  if (rangee != 0)
     send(p');
  envoyee+=rangee;
  p'=(p'+1)%P';
}
```

Algorithme de réception

```
nb_recv=0;
while (nb_recv < L'){
  receive(&p);
  for (x=0; x < K; x++){
    xp=first(p'*K'-p*K-x,P*K);
    if (x' < L'){
      nb=div(L'-x',P*K);
      nb_recv+=nb;
      upk(&TAB[x'],nb,P*K);
    }
  }
}
```

FIG. 3.15 - *Algorithme de redistribution Cyclic(K) vers Block(K')*

3.3.2 Algorithme de redistribution $Cyclic(K)$ vers $Block(K')$

L'algorithme figure (3.15) est proche de celui de la figure(3.13). Dans les deux cas, le coût du calcul des sections régulières est très faible. Les accès en lecture ne sont pas contiguës mais ils le sont en écriture (fig. 3.14).

3.3.3 Remarque

En fixant la variable K' à N, cet algorithme permet d'effectuer la redistribution $Cyclic(K)$ vers $Collapsed$.

3.4 Redistribution $Cyclic(TK')$ vers $Cyclic(K')$

Il est également possible de résoudre efficacement l'équation générale (3.7) lorsque les tailles K et K' des blocs des distributions source et destination sont multiples entre elles. Si $K = TK'$; tous les éléments d'un unique sous bloc d'indice $\mu \in [0, T[$, de taille K' sont envoyés vers un même processeur[11] (fig. 3.16). Nous allons mettre à profit cette propriété pour simplifier l'équation générale de redistribution. Comme tous les éléments d'un même sous bloc sont envoyés vers un unique processeur, nous ne nous intéresserons qu'à la destination du premier élément de chaque sous bloc. Sur les processeurs de l'ensemble de départ, chaque bloc de taille K est constitué de T sous blocs de taille K'. Les indices des premiers éléments de sous blocs vérifient : $x = \mu K'$. Sur les processeurs cibles, les blocs ne sont constitués que d'un unique sous bloc, nous pouvons donc poser $x' = 0$. Nous pouvons ainsi remplacer les deux variables x et x' par une unique variable μ. Après avoir effectué ces substitutions et après simplification du modèle général par K', on obtient l'équation de redistribution spécifique au

11. Car $p' = (pK + nPK + \mu K' + x')/K' \bmod P' = (pK + nPK + \mu K')/K' \bmod P' \quad \forall x' \in [0, K'[$.

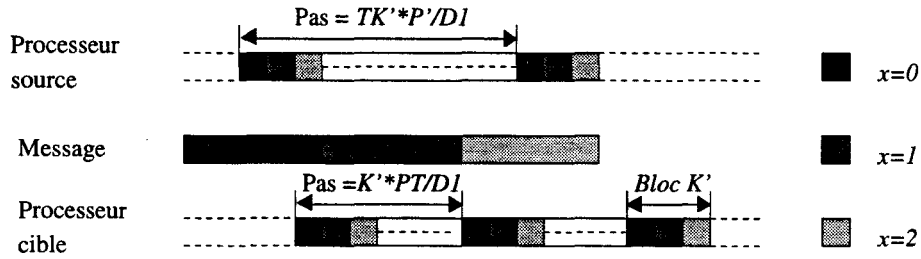


FIG. 3.16 - *Accès mémoire*

cas $Cyclic(TK')$ vers $Cyclic(K')$.

$i \in I(p,p')$ ssi il existe n, n', μ tels que $\boxed{nPT - n'P' = p' - pT - \mu}$

avec

$$\begin{cases} n \in [0, L(p)/K[\\ n' \in [0, L'(p')/K'[\\ \mu \in [0, T[\end{cases} \quad et \quad \begin{cases} p \in [0, P[\\ p' \in [0, P'[\end{cases} \tag{3.19}$$

3.4.1 Solutions de l'équation

Soit $D_1 = PGCD(PT, P')$, et $\alpha_1, \beta_1 \in \mathbb{Z}$ tels que $\alpha_1 PT + \beta_1 P' = D_1$. Les valeurs de D_1, α_1 et β_1 sont obtenues à l'aide de l'algorithme d'Euclide étendu. Soit $q_1 \in \mathbb{N}$, et $r_1 \in [0, D_1[$ tels que $\mu = q_1 D_1 + r_1$. Les valeurs de q_1 et de r_1 sont donc respectivement égales au quotient et au reste de la division entière de μ par D_1. La partie gauche de l'équation diophantienne (3.19) est divisible par D_1, cette équation admet des solutions si et seulement si sa partie droite est également divisible par D_1. La résolution de cette équation sera réalisée en deux étapes successives. Dans une première partie nous calculerons tous les n-uplets (p, p', μ) pour lesquels la valeur de l'expression $p' - pT - \mu$ est multiple de D_1. Ces n-uplets vont caractériser les couples de processeurs qui interagissent ainsi que les indices de sous blocs μ à migrer. Dans une seconde partie, nous calculerons pour un n-uplets (p, p', μ) donné, les indices des blocs n et n' dans lesquels sont placés les sous blocs.

Lorsqu'un processeur de l'ensemble de départ calcule les indices des processeurs avec lesquels il va communiquer, la valeur de p est connue. Il doit donc chercher tous les couples

(p', μ) tels que la valeur de l'expression $p' - pT - \mu$ soit multiple de D_1, ces couples vérifient :

$$| p' - pT - \mu | \, mod D_1 = 0 \Longleftrightarrow | p' - pT - r_1 | \, mod D_1 = 0$$

$$\Rightarrow \; p' = (pT + r_1) \, mod D_1 + \lambda_2 D_1 \tag{3.20}$$

avec $\lambda_2 \in \mathbb{N}$. Chaque valeur de r_1 génère une nouvelle section régulière de pas D_1. Comme ces sections régulières sont toutes disjointes [12], on ne construit qu'un unique message par couple de processeurs (p, p'). Les indices μ des sous blocs de ce message sont de la forme $\mu = q_1 D_1 + r_1$. La division entière de μ par D_1 nous permet d'énumérer les indices de sous blocs à envoyer à un processeur cible, sans augmenter le nombre de messages.

Avec l'équation (3.20) nous pouvons énumérer les indices p' des processeurs avec lesquels p va communiquer ainsi que les indices de sous blocs μ qu'il faudra envoyer. Il nous faut maintenant calculer les indices de blocs n et n' qui «abritent» ces sous blocs. Nous allons pour cela résoudre l'équation (3.19) en utilisant la méthode de résolution exposée section (1.2). Soit $\lambda_1 \in \mathbb{N}$. Les couples (n, n') solution de l'équation vérifient :

$$n = \alpha_1 \frac{p'-pT-\mu}{D_1} + \lambda_1 \frac{P'}{D_1} \tag{3.21}$$

$$n' = \beta_1 \frac{p'-pT-\mu}{D_1} + \lambda_1 \frac{PT}{D_1} \tag{3.22}$$

Pour chaque n-uplets (p, p', μ), on énumère les indices de blocs n et n' à l'aide de deux sections régulières de pas P'/D_1 et PT/D_1 (fig. 3.16).

3.4.2 Algorithme de redistribution $Cyclic(TK')$ vers $Cyclic(K')$

Dans l'algorithme de redistribution $Cyclic(TK')$ vers $Cyclic(K')$ (fig. 3.17), les blocs de taille K placés sur les processeurs sources sont découpés en T sous blocs de taille K'. Ces sous blocs sont indicés de 0 à $T - 1$ avec la variable μ. Comme nous l'avons vu précédemment, toutes les valeurs d'un même sous bloc, sont envoyées vers un unique processeur cible.

12. Car $r_1 \in [0, D1[$.

Algorithme d'émission

```
GCD(P*T,P',&D1,&ALPHA1,&BETA1);
for (r1=0; r1 < min(D1,T); r1++){
  p'_f=(M*T+r1)%D1;
  for (p'=p'_f;p'<P';p'+=D1){
    initsend();
    nb_send=0;
    for (mu=r1; mu<T; mu+=D1){
      nb_send+=remplir();
    }
    if (nb_send > 0)
      send(tp');
  }
}

int remplir(){

 rangee=0;
 C1=p'-p*T-mu;
 lambda1=div(-ALPHA1*C1/D1,P'/D1);
 n=ALPHA1*C1/D1+lambda1*P'/D1;
 for (x=0; x<K'; x++){
   offset=n*T*K'+mu*K'+x;
   if (offset < L){
     nb=div(L-offset, STRIDE);
     pk(&TAB[offset],nb,STRIDE);
     rangee+=nb;
   }
 }
 return rangee;
}
```

Algorithme de réception

```
GCD(P*T,P',&D1,&ALPHA1,&BETA1);
while (nb_recv < L'){
  receive(&p);
  mu_f=first(-p'+p*T,D1);
  for (mu=mu_f; mu<T;mu+=D1){
    nb_recv+=vider();
  }
}

int vider(){

 videe=0;
 C1=p'-p*T-mu;
 LAMBDA1=div(BETA1*C1/D1,P*T/D1);
 n'=-BETA1*C1/D1+LAMBDA1*P*T/D1;
 for (x'=0; x' < K'; x'++){
   offset=n'*K'+x';
   if (offset < L'){
     nb=div(L'-offset,STRIDE);
     videe+=nb;
     upk(&TAB[offset],nb,STRIDE);
   }
 }
 return videe;
}
```

FIG. 3.17 - *Cyclic(TK') vers Cyclic(K')*

Chaque processeur source énumère les indices des processeurs cibles à l'aide de deux sections régulières. Celles-ci sont calculées à l'aide de l'équation (3.20). Pour chaque destinataire p', une dernière section régulière est utilisée afin de calculer les indices μ des sous blocs à envoyer. Ces indices sont de la forme : $\mu = q_1 D1 + r_1$. Ces trois boucles énumèrent tous les n-uplets (p, p', μ) pour lesquels l'équation (3.19) est susceptible d'admettre des solutions. Il ne reste plus alors qu'à ranger les données correspondantes dans le buffer courant. C'est le travail de la fonction `Remplir` (fig. 3.17). Elle est basée sur l'équation (3.21). Elle place tous les sous blocs d'indice μ destiné au processeur p' dans le buffer (fig. 3.16).

La fonction qui réceptionne les données est plus simple. Contrairement à la fonction précédente, elle ne calcule pas l'adresse de ses correspondants. Elle réceptionne simplement les messages puis énumère les indices de sous blocs. La fonction `vider` (fig. 3.17) est basée sur l'équation (3.22), elle range en mémoire tous les sous blocs d'indice μ qui étaient dans le buffer.

3.5 Redistribution Cyclic(K) vers Cyclic(TK)

Ce cas est le symétrique du cas précédent. Lorsque $K' = TK$ tous les éléments d'un même sous bloc de taille K sont envoyés à un même destinataire[13]. Pour tirer profit de cette propriété nous posons $x = 0$ et $x' = \mu K$ de façon à simplifier l'équation générale par K, K', x et x'. On obtient ainsi l'équation suivante :

$$i \in I(p,p') \text{ ssi il existe } n, n', \mu \text{ tel que } \boxed{nP - n'P'T = p'T - p + \mu}$$

avec

$$\begin{cases} n \in [0, L(p)/K[\\ n' \in [0, L'(p')/K'[\\ \mu \in [0, T[\end{cases} \quad et \quad \begin{cases} p \in [0, P[\\ p' \in [0, P'[\end{cases} \tag{3.23}$$

13. Car $p' = (pK+nPK+x)/TKmodP' = ((p+nP)K+x)/TKmodP' = ((p+nP)K)/TKmodP' \forall x \in [0, K[$.

3.5.1 Solutions du système

Soit $D_1 = PGCD(P, P'T)$, $D_2 = PGCD(T, D_1)$ et α_1, α_2, β_1, $\beta_2 \in \mathbb{Z}$ tels que $\alpha_1 P + \beta_1 P'T = D_1$ et $\alpha_2 T + \beta_2 D_1 = D_2$. Soit q_1, $q_2 \in \mathbb{N}$, $r_1 \in [0, D_1[$ et $r_2 \in [0, D_2[$ tels que $\mu = q_1 D_1 + r_1$ et $r_1 = q_2 D_2 + r_2$. La partie gauche de l'équation diophantienne (3.23) est divisible par D_1, cette équation admet des solutions si et seulement si sa partie droite est également divisible par D_1. La résolution de cette équation est plus complexe que dans le cas précédent car la variable p' dont la valeur n'est pas connue est ici multipliée par la constante T. Pour résoudre cette équation nous allons utiliser à deux reprises la technique de résolution exposée section (1.2). Une première résolution va nous permette de calculer tous les n-uplets (p, p', μ) qui vérifient $|p'T - p + \mu| \; mod \; D_1 = 0$. Une seconde résolution sera ensuite nécessaire pour calculer les indices des blocs n et n' concernés.

Tout d'abord, nous allons calculer les couples de processeurs qui interagissent et les indices de sous blocs à échanger. Pour cela, nous calculons les n-uplets (p, p', μ) pour lesquels la partie droite de l'équation (3.23) est divisible par D_1. Soit $\gamma_1 \in \mathbb{Z}$ tels que :

$$\begin{array}{rcl} | \, p'T - p + \mu \, | \; mod D_1 & = & 0 \\ \iff \quad p'T - p + \mu & = & \gamma_1 D_1 \\ \iff \quad p'T + (q_1 - \gamma_1) D_1 & = & p - r_1 \end{array}$$

Cette équation admet des solutions si et seulement si $p - r_1$ est multiple de D_2 c'est à dire si $\boxed{r_2 = p \; mod \; D_2}$. D'après (3.5) les solutions sont de la forme :

$$p' = \frac{\alpha_2 (p - r_1)}{D_2} + \frac{\lambda_2 D_1}{D_2} \tag{3.24}$$

Toutes les données envoyées par le processeur d'indice p seront placées dans des sous blocs d'indices $\mu = q_1 D_1 + q_2 D_2 + p \; mod D_2$. Le nombre de messages que le processeur p va construire avec l'équation (3.24) est minimal s'il existe un unique couple (λ_2, r_1), pour chaque correspondant p'. Démontrons par l'absurde l'unicité de ce couple. Supposons qu'il existe deux couples (λ_2, r_1) et (λ'_2, r'_1) qui génère le même processeur p'.

$$\text{On a } r_1, r'_1 \in [0, D_1[, \; q_2, q'_2 \in \mathbb{N} \, , \lambda_2, \lambda'_2 \in \mathbb{Z} \text{ et } \left\{ \begin{array}{rcl} r_1 & = & q_2 D_2 + r_2 \\ r'_1 & = & q'_2 D_2 + r_2 \\ p' & = & \frac{\alpha_2 (p - r_1)}{D_2} + \frac{\lambda_2 D_1}{D_2} \\ p' & = & \frac{\alpha_2 (p - r'_1)}{D_2} + \frac{\lambda'_2 D_1}{D_2} \end{array} \right.$$

$$\text{donc} \quad \frac{\alpha_2(p-r_1)}{D_2} + \frac{\lambda_2 D_1}{D_2} = \frac{\alpha_2(p-r'_1)}{D_2} + \frac{\lambda'_2 D_1}{D_2}$$

$$\text{d'où} \quad D_1(\lambda_2 - \lambda'_2) = \alpha_2(r_1 - r'_1)$$

$$\text{finalement on obtient :} \quad \frac{D_1}{D_2}(\lambda_2 - \lambda'_2) = \alpha_2(q_2 - q'_2) \tag{3.25}$$

Pour démontrer qu'il n'existe qu'un unique couple (λ_2, r_1) qui génère la valeur p' avec l'équation (3.24), il nous faut maintenant montrer que les seules solutions de l'équation (3.25) sont $\lambda_2 = \lambda'_2$ et $q_2 = q'_2$. De par la définition des variables nous avons : $\alpha_2 T + \beta_2 D_1 = D_2$ et $D_2 = PGCD(T, D_1)$. Les entiers $\frac{T}{D_2}$ et $\frac{D_1}{D_2}$ sont donc premiers entre eux.

- Si $D_1 = D_2$:

  q_2 et q'_2 sont respectivement les quotients des divisions de r_1 et r'_1 par D_2. Comme r_1 et r'_1 sont dans l'intervalle $[0, D_1[$, on a $q_2 = q'_2 = 0$ et $\lambda_2 = \lambda'_2$. Toutes les données envoyées par le processeur d'indice p sont placées dans des sous blocs d'indices $\mu = q_1 D_1 + p \, mod D_2$.

- Si $D_1 \neq D_2$:

  La partie gauche de l'équation (3.25) est divisible par D_1/D_2 et α_2 ne l'est pas[14]. $(q_2 - q'_2)$ doit donc nécessairement être multiple de D_1/D_2. D'après les définitions de r_1 et r'_1, nous avons $D_2(q_2 - q'_2) = r_1 - r'_1$, et donc $|q_2 - q'_2| < \frac{D_1}{D_2}$. La partie droite de l'équation (3.25) n'est divisible par que si $q_2 = q'_2$. Cela implique $\lambda_2 = \lambda'_2$.

Dans tous les cas $q_2 = q'_2$ et $\lambda_2 = \lambda'_2$. Il existe un unique couple (λ_2, r_1) qui donne la valeur de p' avec l'équation (3.24). Cette équation sera utilisée d'une part énumérer les indices des processeurs destinataires avec une première section régulière de pas D_1, et d'autre part pour calculer les indices de sous blocs à envoyer avec une seconde section régulière de pas D_1/D_2).

Nous avons calculé tous les n-uplet(p, p', μ) pour lesquels l'équation (3.23) est susceptible d'admettre des solutions ; nous allons maintenant calculer pour chacun de ces n-uplet, les indices de blocs (n, n') dans lesquels sont placés les données. D'après la méthode de résolution exposée section (1.2), ces couples vérifient :

$$n = \alpha_1 \frac{p'T - p + \mu}{D_1} + \lambda_1 \frac{P'T}{D_1} \tag{3.26}$$

14. $\alpha_2 T/D_2 + \beta_2 D_1/D_2 = 1$.

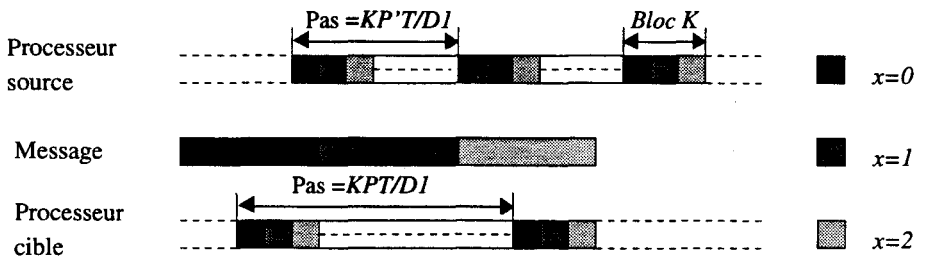


FIG. 3.18 - *Accès mémoire*

$$n' = \beta_1 \frac{p'T-p+\mu}{D_1} + \lambda_1 \frac{P}{D_1} \tag{3.27}$$

Pour chaque n-uplets (p, p', μ), on énumère les indices de blocs n et n' à l'aide de deux sections régulières de pas $P'T/D_1$ et P/D_1 (fig. 3.18).

3.5.2 Algorithme de redistribution $Cyclic(K)$ vers $Cyclic(TK)$

Les algorithmes d'émission et de réception (fig. 3.17) sont très proches des algorithmes définis pour la redistribution $Cyclic(TK')$ vers $Cyclic(K')$. Les variables q_2 et r_2 qui ont été introduites lors de la résolution n'apparaissent pas. Elles ont été «consommées» au niveau de la première boucle. La variable r_1 prend successivement toutes les valeurs qui vérifient $r_1 = q_2 D_2 + r_2$, avec $r_1 \in [0, min(D_1, T)[$ et $r_2 = p \, mod D_2$.

Les valeurs de départ et les pas ne sont plus les mêmes mais le nombre de sections régulières et la complexité de l'algorithme restent inchangés.

4 Résultats expérimentaux

Toutes les mesures présentées dans cette section ont été réalisées sur une ferme de seize processeurs ALPHA [15]. Ces processeurs sont reliés à un crossbar à l'aide de seize anneaux de type FDDI ainsi qu'à un bus Ethernet. Dans chacun des cas, nous avons mesuré le temps nécessaire pour migrer avec la bibliothèque PVM, un tableau T de quatre millions d'entiers, d'un ensemble E_1, de P processeurs, à un ensemble E_2, de P' processeurs, en conservant toujours $E_1 \cap E_2 = \emptyset$. Toutes les courbes présentées dans cette section ont été calculées en effectuant la moyenne de cinq séries de mesures. La taille du tableau à été choisie pour simplifier l'étude des résultats obtenus. Le coût d'une migration de données est en général

15. Une machine étant en panne, la *quasi* totalité des mesures ont été réalisées sur 15 processeurs.

Algorithme d'émission

```
GCD(P,P'*T,&D1,&ALPHA1,&BETA1);
GCD(T,D1,&D2,&ALPHA2,&BETA2);
D=D1/D2;
for (r1=p%D2;r1<min(D1,T);r1+=D2){
  lambda2=div(-ALPHA2*(p-r1)/D2,D);
  p'_min=ALPHA2*(p-r1)/D2+lambda2*D;
  for (p'=p'_min; p'<P'; p'+=D){
    initsend();
    placee=0;
    for (mu=r1; mu<T; mu+=D1){
      placee+=remplir2();
    }
    if (placee > 0) send(p');
  }
}

int remplir2(){

 rangee=0;
 C1=p'*T-p+mu;
 lambda1=div(-ALPHA1*C1/D1,P'*T/D1);
 n=ALPHA1*C1/D1+lambda1*P'*T/D1;
 for (x=0; x<K; x++){
   offset=n*K+x;
   if (offset < L){
     nb=div(L-offset,STRIDE);
     pk(&TAB[offset],nb,STRIDE);
     rangee+=nb;
   }
 }
 return rangee;
}
```

Algorithme de réception

```
GCD(P,P'*T,&D1,&ALPHA1,&BETA1);
while (nb_recv < L'){
  receive(&p);
  mu_f=first(p'*T-p,D1);
  for (mu=mu_f;mu<T;mu+=D1){
    nb_recv+=vider2();
  }
}

int vider2(){

 videe=0;
 C1=p'*T-p+mu;
 LAMBDA1=div(BETA1*C1/D1,P/D1);
 n'=-BETA1*C1/D1+LAMBDA1*P/D1;
 for (x=0; x < K; x++){
   offset=n'*K*T+mu*K+x;
   if (offset < L'){
    nb=div(L'-offset,STRIDE);
    videe+=nb;
    upk(&TAB[offset],nb,STRIDE);
   }
 }
 return videe;
}
```

FIG. 3.19 - *Cyclic(K) vers Cyclic(TK)*

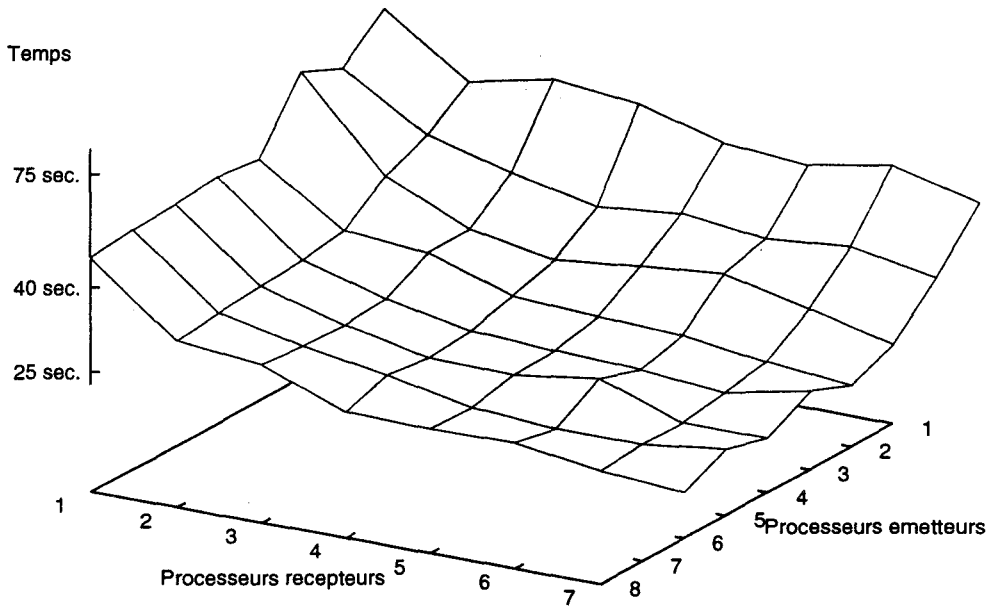


FIG. 3.20 - *Redistribution Block(K) vers Block(K') avec l'algorithme énumératif sur Ethernet*

approximé par la fonction $\mathcal{F}(N) = \alpha N + \beta$. Les tests effectués montrent qu'avec $N = 4000000$, nous pouvons à la fois négliger β et éviter les défauts de pages qui se produisent lorsque la taille des données est trop importante. Pour les distributions par blocs, nous avons choisi de prendre : $K = div(N, P)$ et $K' = div(N, P')$, de façon à répartir au mieux la charge sur l'ensemble des processeurs.

Dans chacun des cas, nous comparerons les temps obtenus avec l'algorithme énumératif à ceux obtenus avec l'algorithme de redistribution spécialisé correspondant. Le calcul des gains permettra d'évaluer les qualités mais aussi les limites des algorithmes précédemment introduits.

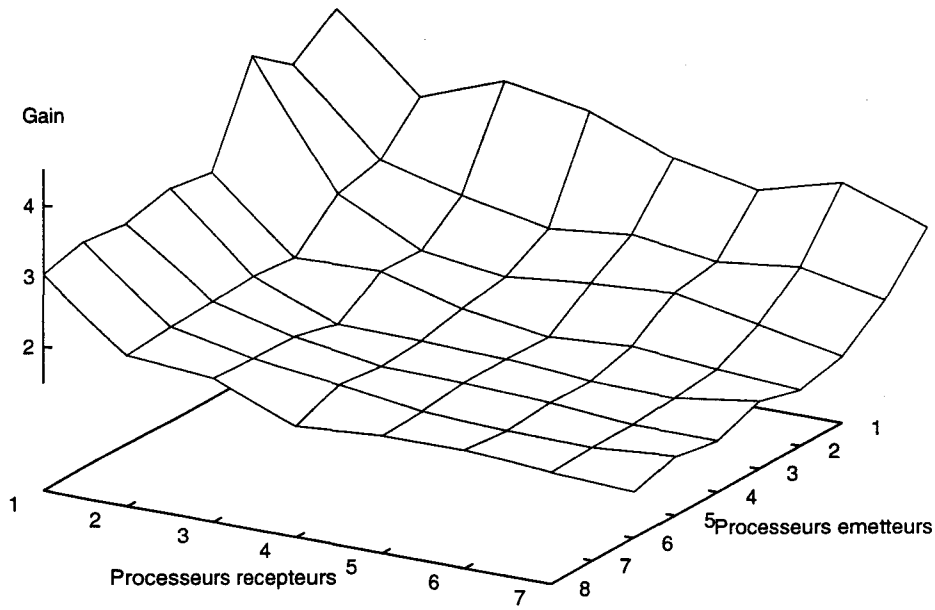


FIG. 3.21 - *Gain avec l'algorithme spécialisé Block(K) vers Block(K') sur Ethernet*

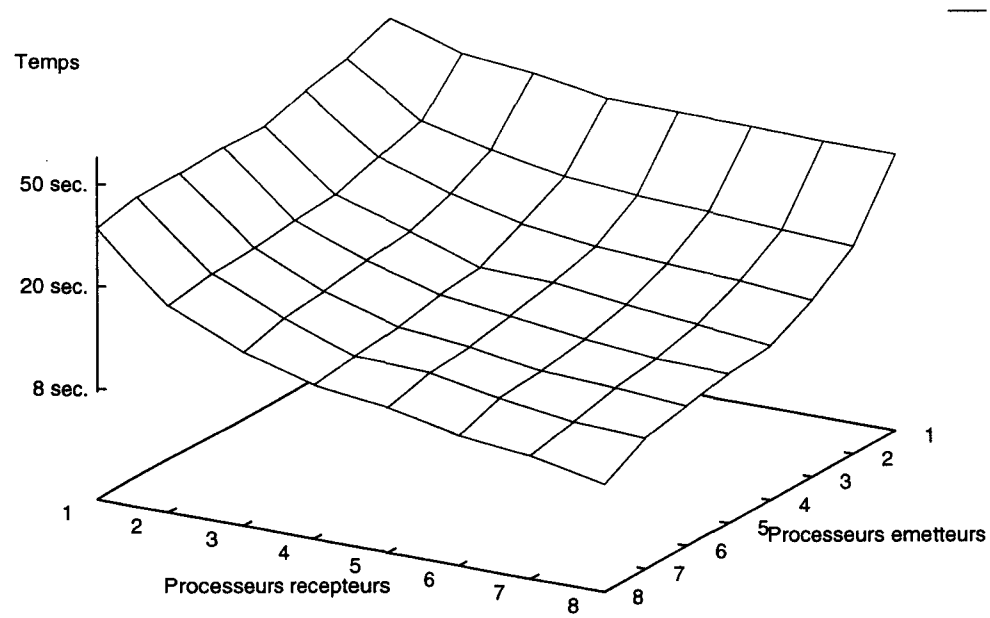


FIG. 3.22 - *Redistribution Block(K) vers Block(K') avec l'algorithme énumératif sur FDDI*

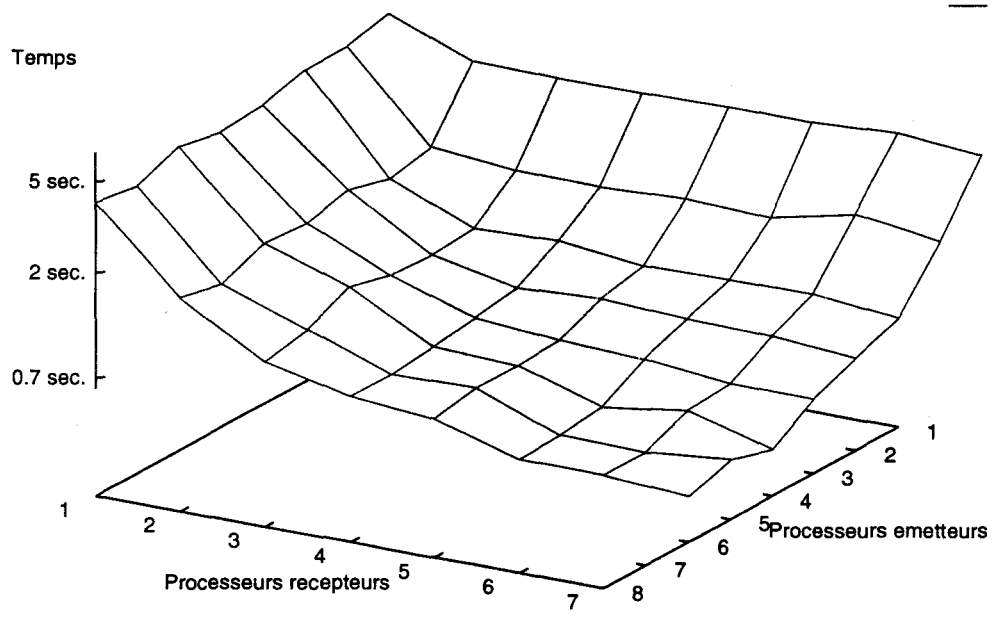


FIG. 3.23 - *Redistribution Block(K) vers Block(K') avec l'algorithme spécialisé sur FDDI*

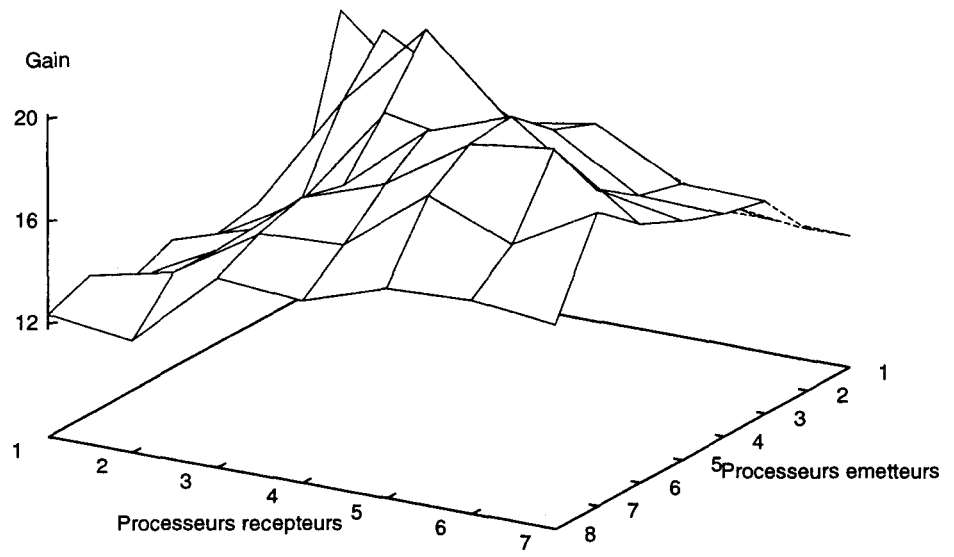


FIG. 3.24 - *Gain avec l'algorithme spécialisé Block(K) vers Block(K') sur FDDI*

4.1 Redistribution $Block(K)$ vers $Block(K')$

4.1.1 Avec l'algorithme énumératif

La figure (3.20) présente les résultats obtenus avec l'algorithme énumératif sur le bus Ethernet. La figure (3.22) a été obtenue avec le même algorithme sur le crossbar FDDI.

Avec le bus Ethernet Les temps ne diminuent que faiblement lorsqu'on augmente le nombre de processeurs. La courbe est également symétrique par rapport au plan $P = P'$.

Avec le crossbar FDDI Les temps diminuent de façon à peu près linéaire avec le nombre de processeurs. La courbe est symétrique par rapport au plan $P = P'$.

Ces courbes expérimentales corroborent assez bien l'étude théorique. Les temps expérimentaux sont un peu supérieurs aux temps théoriques. Les performances de l'algorithme énumératif général vont nous servir de référence pour mesurer l'efficacité des algorithmes spécifiques précédemment proposés. Comme le réseau Ethernet ne permet pas d'obtenir de résultats intéressants en raison du goulot d'étranglement qu'il représente, nous ne nous intéresserons, pour les autres algorithmes spécialisés, qu'aux performances obtenues avec le crossbar FDDI.

4.1.2 Gain avec l'algorithme spécialisé $Block(K)$ vers $Block(K')$

Sur le bus Ethernet Le gain est maximal ($\simeq 4$) lorsque $P = P' = 1$ puis décroît rapidement lorsque le nombre de processeurs augmente. Il tend ensuite vers un lorsque celui-ci devient grand (1,5 pour $P = P' = 8$). Cette courbe est conforme à celle prévue par l'étude théorique (paragraphe 2.2). Le bus Ethernet constitue un goulot d'étranglement. Il n'est pas possible dans ces conditions d'obtenir des performances satisfaisantes.

Avec le crossbar FDDI Le gain est important quelles que soient les valeurs de P et P'. Il est maximal lorsque $P = P'$ et croît avec la fonction $min(P, P')$. Si l'allure de la courbe est globalement conforme à celle prévue par l'étude théorique, la corrélation est moins bonne que dans le cas précédent. En particulier on constate que le gain diminue moins rapidement lorsqu'on s'écarte de l'égalité $P = P'$. On mesure par exemple pour $P = 1$ et $P' = 7$, un gain d'environ treize alors que le gain théorique est de dix. Inversement, la valeur théorique maximale (pour $P = P'$) est plus élevée que la valeur expérimentale. Le gain théorique pour $P = P' = 7$ est égal à 21, alors que le gain mesuré est égal à 19.

Nous allons voir que ces écarts s'expliquent par des simplifications du modèle théorique d'une part, et par des problèmes expérimentaux d'autre part[16].

Lors de notre étude théorique, nous avons modélisé le coût d'un algorithme idéal à l'aide de la fonction (3.9). Pour construire cette fonction, nous avons supposé que le nombre de messages émis par chacun des processeurs sources, était égal au nombre de processeurs cibles. Or, lorsque $P = P'$, la taille des blocs sur la machine de départ est égal à la taille des blocs de la machine d'arrivée et chaque processeur source ne s'adresse qu'à un unique destinataire. Il n'y a donc pas de recouvrement et cela explique les moins bonnes performances.

La décroissance moins rapide des performances lorsque la valeur de P s'écarte de la valeur de P' s'explique elle, par des problèmes pratiques. Nous disposons d'une machine MIMD, et il n'existe pas sur ce type de machine de moyen matériel pour synchroniser les processeurs comme on peut en trouver sur les machines SIMD. Nous ne disposons que de synchronisations logiques réalisées par envoi de messages. Dans ce contexte, tous les processeurs ne passent pas la barrière de synchronisation au même instant. Certains vont parvenir à émettre alors que d'autres seront encore bloqués.

4.2 Redistribution $Block(K)$ vers $Cyclic(K')$

4.2.1 Avec l'algorithme énumératif

Les temps obtenus avec l'algorithme énumératif sont présentés figure (3.25). La courbe est quasiment identique à celle obtenue pour la redistribution $Block(K)$ vers $Block(K')$ fig. (3.22), elle est simplement légèrement translatée vers le haut[17]. Ce «surcoût» est nécessairement lié aux communications car l'algorithme énumératif exécute toujours les mêmes calculs quelle que soit la redistribution effectuée.

Lors de l'étude théorique nous avons modélisé le coût d'une redistribution avec l'algorithme énumératif à l'aide de l'équation (3.8) suivante :

$$T_{enumeratif} = \frac{N(t_{adr}+t_{pk})}{P} + \frac{t_c N}{\varphi} + \frac{(t_{adr}+t_{upk})N}{P'}$$

Comme le volume de données à traiter est indépendant de la redistribution effectuée, le temps théorique ne dépend que des valeurs de P et P' ($\varphi = min(P, P')$). Cette équation

16. Ces problèmes existaient déjà dans le cas précédent mais ils étaient occultés par les très mauvaises performances du réseau.

17. La «bosse» pour $P = 4$ et $P' = 7$ n'est pas significative, elle résulte d'une perturbation extérieure.

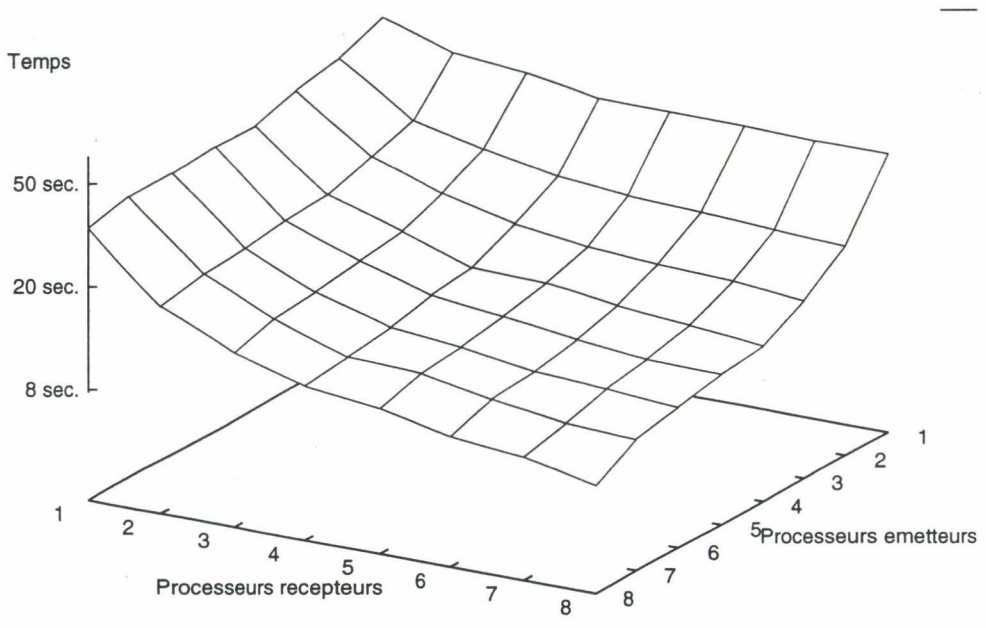


FIG. 3.25 - *Redistribution Block(K) vers Cyclic(8) avec l'algorithme énumératif*

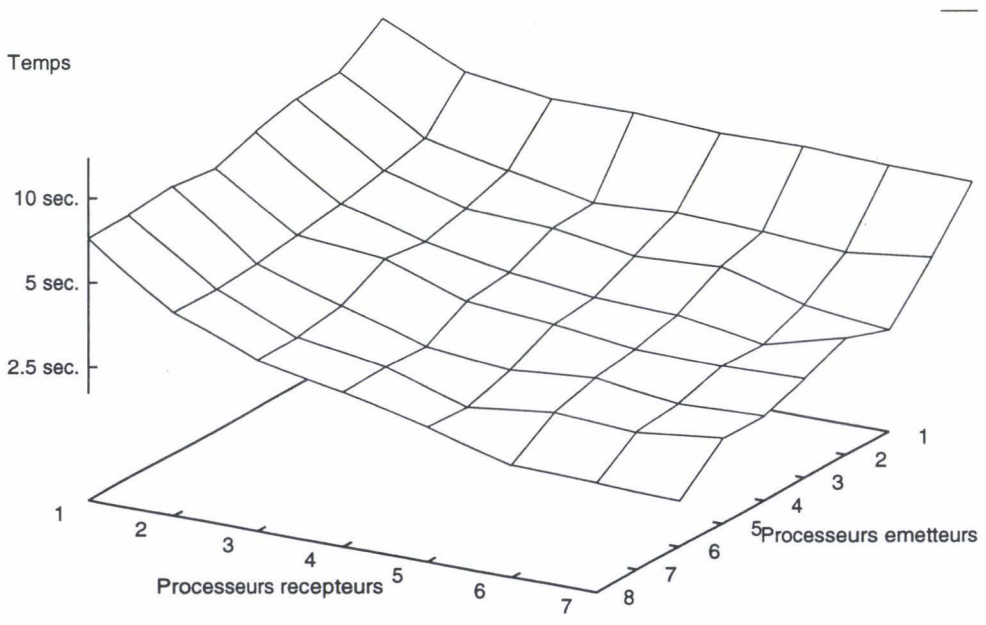


FIG. 3.26 - *Redistribution Block(K) vers Cyclic(8) avec l'algorithme spécialisé*

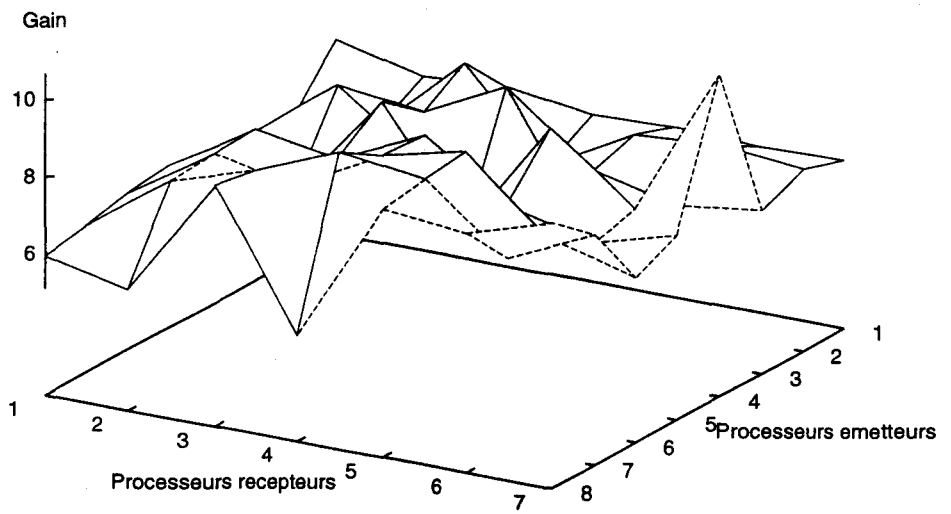


FIG. 3.27 - *Gain avec l'algorithme spécialisé Block(K) vers Cyclic(K')*

prend en compte le nombre de liens de communications ainsi que la bande passante de ces liens, mais elle suppose une utilisation effective de toutes les ressources disponibles.

L'algorithme énumératif utilise toujours le même schéma de communication. Les processeurs sources s'adressent d'abord au processeur cible d'indice 0, puis 1, ..., puis $P' - 1$. Cet ordonnancement n'est pas toujours idéal. Avec la redistribution *Bloc* vers *Bloc*, un processeur source ne s'adresse qu'à un sous ensemble des processeurs de la machine d'arrivée (les autres messages sont «vides»). Dans ce contexte le nombre de conflits est limité et la bande passante est bien utilisée. Avec la redistribution *Bloc* vers *Cyclic*(8) les volumes des messages sont identiques pour tous les processeurs. Tous les processeurs sources envoient au même moment un message de taille ($\simeq K/(P'P)$) à un unique destinataire. Le réseau est donc très mal utilisé. L'écart constaté est cependant assez faible en raison du coût très important des fonctions de calcul d'adresses ($t_c = 0.7$, $t_{adr} = 2.07$, $t_{pk} = 6.07$, $t_{upk} = 3.9$).

4.2.2 Avec l'algorithme spécialisé

Les résultats obtenus (fig. 3.26) avec l'algorithme spécialisé sont inégaux. On constate des «irrégularités» pour certaines valeurs de (P, P'). Ces variations résultent de la présence ou de l'absence de conflits.

L'impact des communications qui était limité avec l'algorithme général est cette fois prépondérant. Avec l'algorithme énumératif les communications sont «noyées» dans les calculs, les conflits ont un impact limité sur les performances globales. Avec les algorithmes spécialisés les communications sont prépondérantes car il n'y a pas de calcul d'adresse, les performances dépendent directement du nombre de conflits.

4.2.3 Gain avec la redistribution $Block(K)$ vers $Cyclic(K')$

Les gains obtenus sont irréguliers car les deux algorithmes n'utilisent pas les mêmes schémas de communication. On ne rencontre donc pas les mêmes conflits avec les mêmes paramètres (valeurs de P et P'). Avec l'algorithme spécialisé, un processeur source calcule l'indice du processeur avec lequel il va communiquer en fonction de l'indice de la première données qu'il possède. La présence ou l'absence de conflits dépend alors des valeurs de P, P' et N (fig. 3.26).

Étudions par exemple les conflits qui se produisent lorsque $N = 2^{22}$, $P' = 2$ et $K' = 8$. Si $P = 2^x$ alors $K = div(N, P) = 2^y$ et $p' = (pK) \; mod \; P' = (2^y p) \; mod \; 2 = 0$. Cela signifie que tous les processeurs sources s'adressent en même temps au même destinataire lorsque la valeur de P est une puissance de deux. Comme tous les processeurs énumèrent les indices des processeurs destinataires avec la même section régulière, on conserve le même schéma de communication pendant toute la redistribution. Cela explique parfaitement les minimums obtenus (fig. 3.27) pour $P' = 2$ et $P \in \{2, 4, 8\}$.

4.3 Redistribution $Cyclic(K)$ vers $Block(K')$

Cette redistribution présente de nombreuses similitudes avec la redistribution $Block(K)$ vers $Cyclic(K')$ nous devrions donc obtenir des résultats comparables.

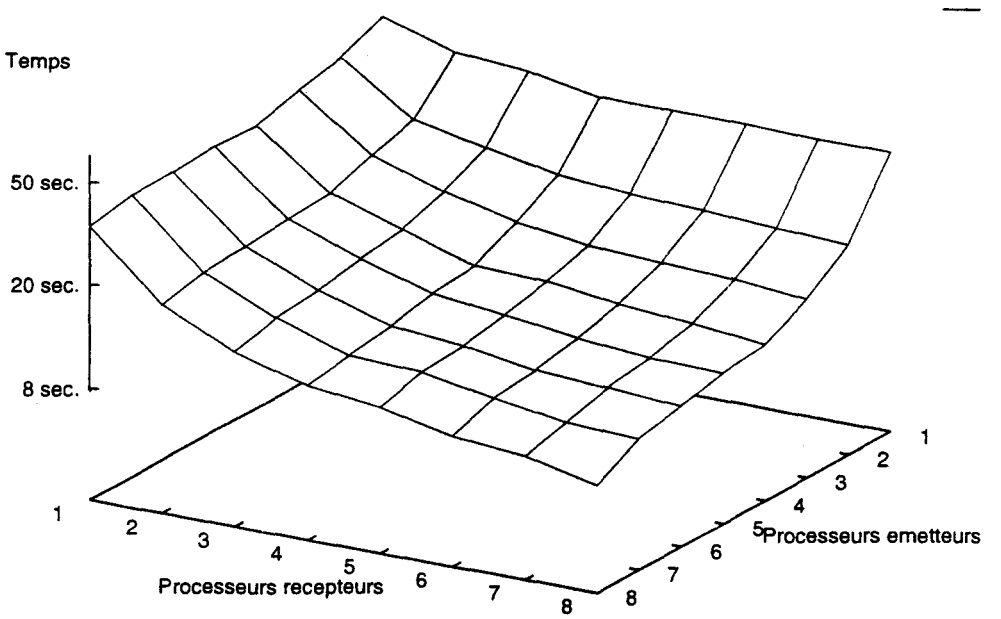


FIG. 3.28 - *Redistribution Cyclic(8) vers Block(K') avec l'algorithme énumératif*

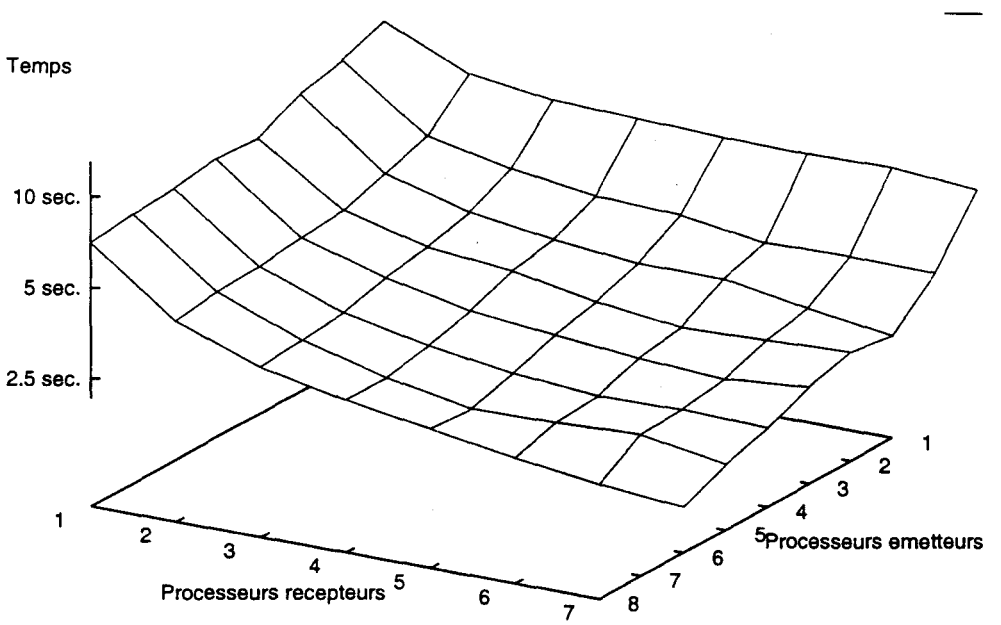


FIG. 3.29 - *Redistribution Cyclic(8) vers Block(K') avec l'algorithme spécialisé*

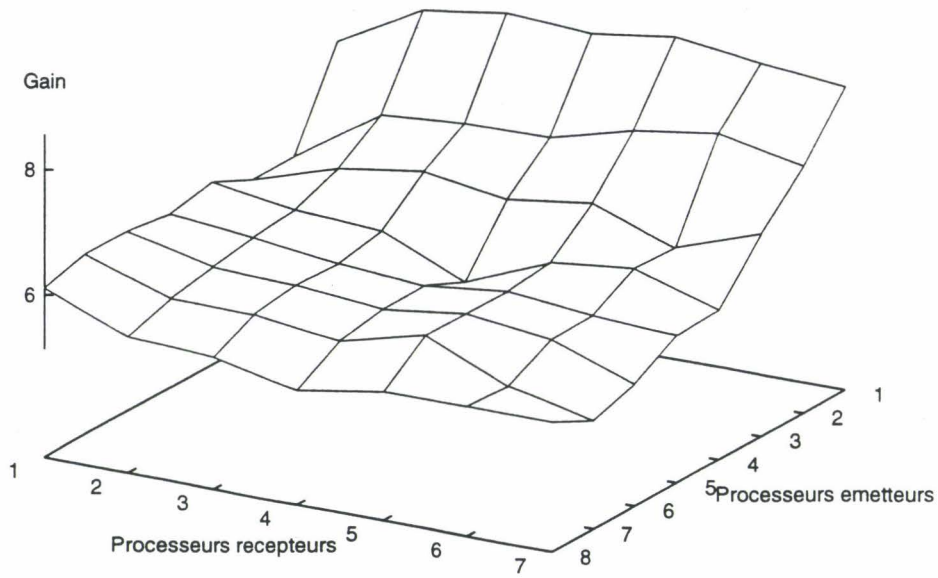


FIG. 3.30 - *Gain avec l'algorithme spécialisé $Cyclic(K)$ vers $Block(K')$*

4.3.1 Avec l'algorithme énumératif

Les résultats (fig. 3.28) sont conformes à nos attentes. La courbe est à peu près identique à celle obtenue lors de la redistribution $Block(K)$ vers $Cyclic(K')$.

4.3.2 Avec l'algorithme spécialisé

La courbe (fig. 3.29) est homogène quelque soient les valeurs de P et P'. À la différence de la redistribution $Block(K)$ vers $Cyclic(K')$, on n'observe plus de variations brutales de performances. Par ailleurs les temps obtenus sont beaucoup plus importants et l'écart augmente avec le nombre de processeurs. Comment expliquer ces différences?

Les algorithmes de redistribution $Block(K)$ vers $Cyclic(K')$ et $Cyclic(K)$ vers $Block(K')$ sont très proches. Ils effectuent à peu près les mêmes calculs. Dans les deux cas tous les processeurs sources s'adressent à chacun des processeurs destinataires. Les volumes des messages sont à peu près égaux à (N/PP'). Seul l'ordre d'émission des messages est modifié.

Dans les deux cas, les processeurs sources calculent l'indice du premier processeur avec

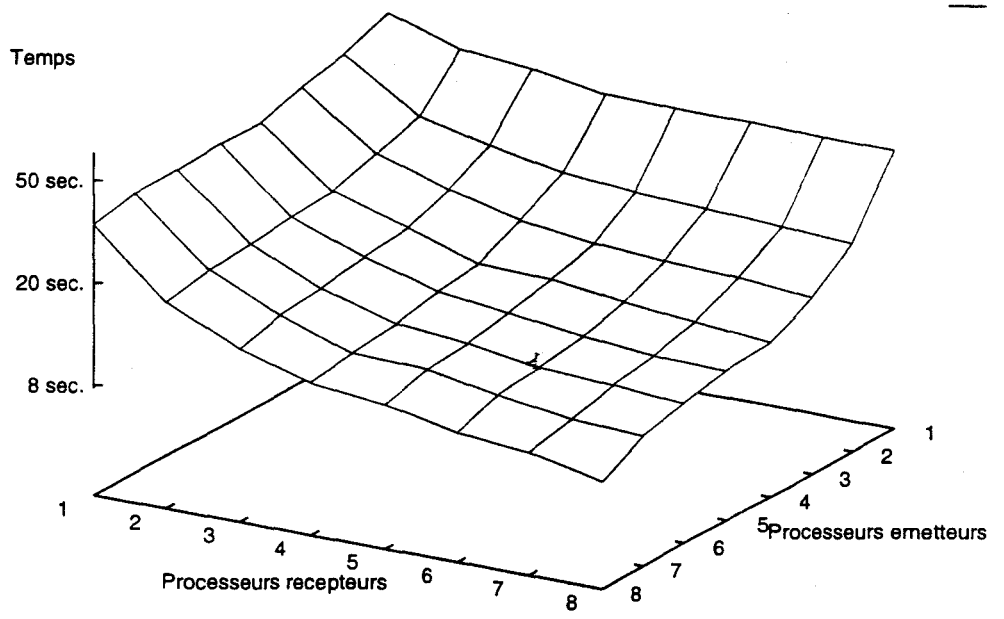


FIG. 3.31 - *Redistribution Cylic(8) vers Cylic(1) avec l'algorithme énumératif*

lequel ils vont communiquer en fonction de l'indice de la première donnée qu'ils possèdent. Avec la distribution $Cyclic(K)$, l'indice de la première donnée placée sur le processeur p est pK. Dans notre exemple k est égal à huit, et pK appartient à l'intervalle $[0, div(N, P')[$ (indices des données placées sur le processeur $p' = 0$) quelle que soit la valeur de p.

Le cas présenté est le plus défavorable: tous les processeurs de la machine source s'adressent systématiquement au même processeur cible. La machine d'arrivée est donc mal utilisée.

4.3.3 Gain avec la redistribution $Cyclic(K)$ vers $Block(K')$

Le gain (fig. 3.30) reste à peu près constant (environ 5) quelles que soient les valeurs de P et P'. Le gain est un peu supérieur lorsque P est petit. Ce résultat n'est pas surprenant car le nombre de conflit est proportionnel à P.

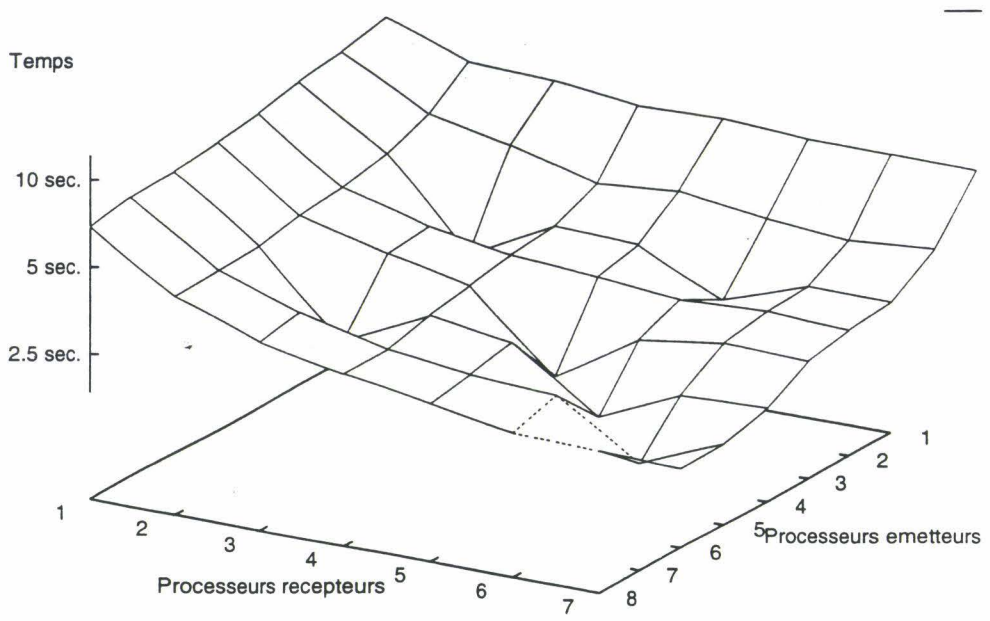


FIG. 3.32 - *Redistribution Cylic(8) vers Cylic(1) avec l'algorithme spécialisé*

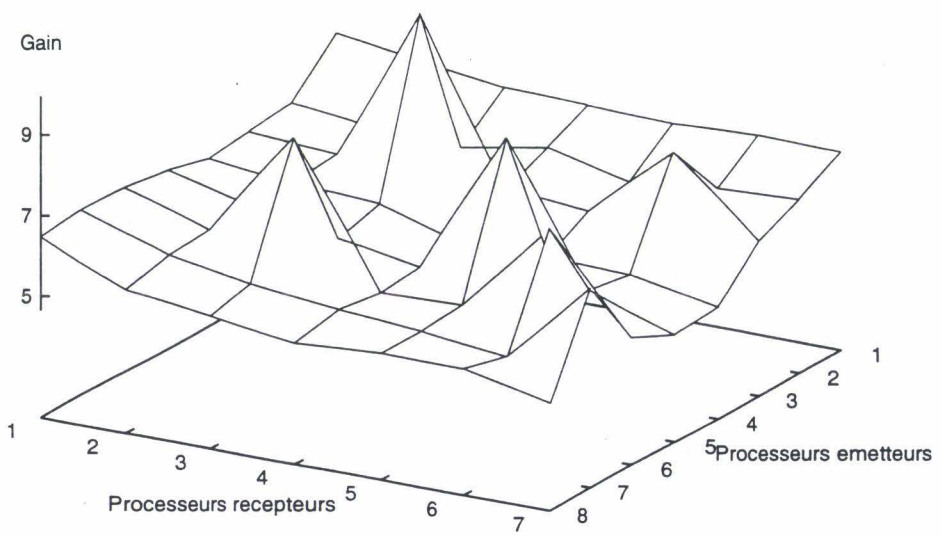


FIG. 3.33 - *Gain avec l'algorithme spécialisé Cylic(8) vers Cylic(1)*

4.4 Redistribution $Cyclic(TK')$ vers $Cyclic(K')$

4.4.1 Avec l'algorithme énumératif

Les temps obtenus avec l'algorithme général (fig. 3.31) sont très proches de ceux obtenus dans le paragraphe précédent. Le coût d'une redistribution avec cet algorithme ne dépend pratiquement pas des valeurs de K et K'. Les résultats seraient différents avec un réseau de communication moins performant (ou avec des processeurs plus rapides).

4.4.2 Avec l'algorithme spécialisé

Les temps pour effectuer les redistributions diminuent faiblement lorsqu'on augmente le nombre de processeurs (fig. 3.32). On observe cependant des pics de performances pour certaines valeurs de (P, P').

4.4.3 Gain avec la redistribution $Cyclic(TK')$ vers $Cyclic(K')$

Les gains obtenus figure (3.33) avec l'algorithme $Cyclic(TK)$ vers $Cyclic(K)$ sont inégaux. Il s'agit là encore d'un problème de répartition de la charge sur la machine cible. La répartition des calculs sur les processeurs cibles dépend ici des valeurs de P et P'. Dans les cas les plus favorables, les messages sont envoyés à l'ensemble des processeurs cibles, toutes les ressources sont utilisées, le recouvrement des calculs et des communications est bon et le gain est égal à dix. Dans les autres cas l'ensemble des processeurs sources s'adresse à un sous ensemble des processeurs de la machine d'arrivée et le gain ne dépasse pas sept.

4.5 Redistribution $Cyclic(K)$ vers $Cyclic(TK)$

Les résultats obtenus figure (3.35) et figure (3.36) pour la redistribution $Cyclic(K)$ vers $Cyclic(TK)$ sont extrêmement proches de ceux obtenus avec la redistribution $Cyclic(TK')$ vers $Cyclic(K')$. C'est assez étonnant car les algorithmes de redistribution utilisés sont assez différents. Dans les prochains paragraphes (5.4 et 5.5) nous montrerons que le nombre de liens de communication effectivement utilisés dépend des valeurs de P et P' et qu'il est identique pour ces deux algorithmes.

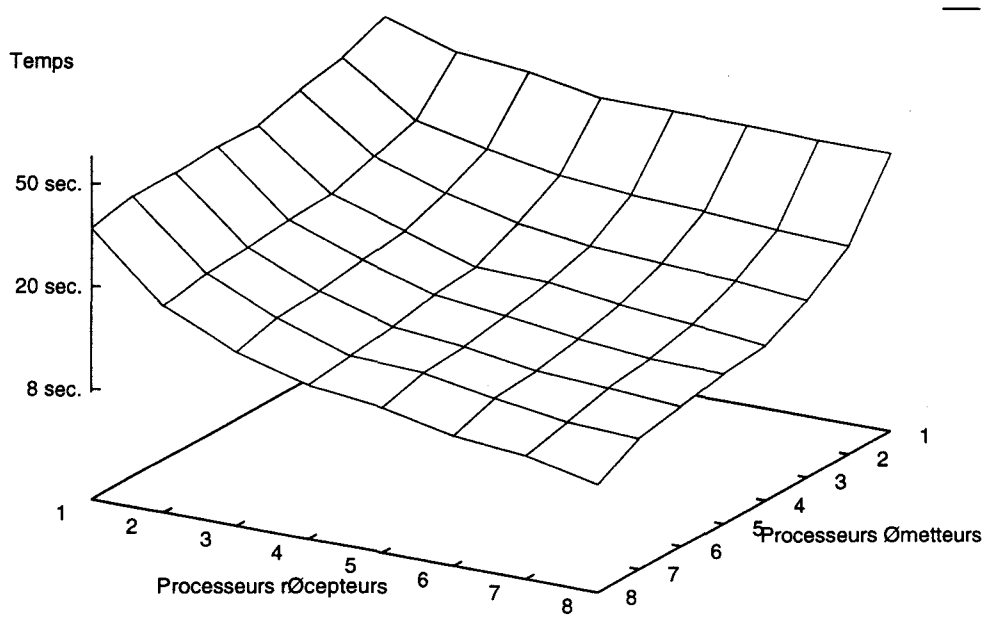


FIG. 3.34 - *Redistribution Cyclic(1) vers Cylic(8) avec l'algorithme énumératif*

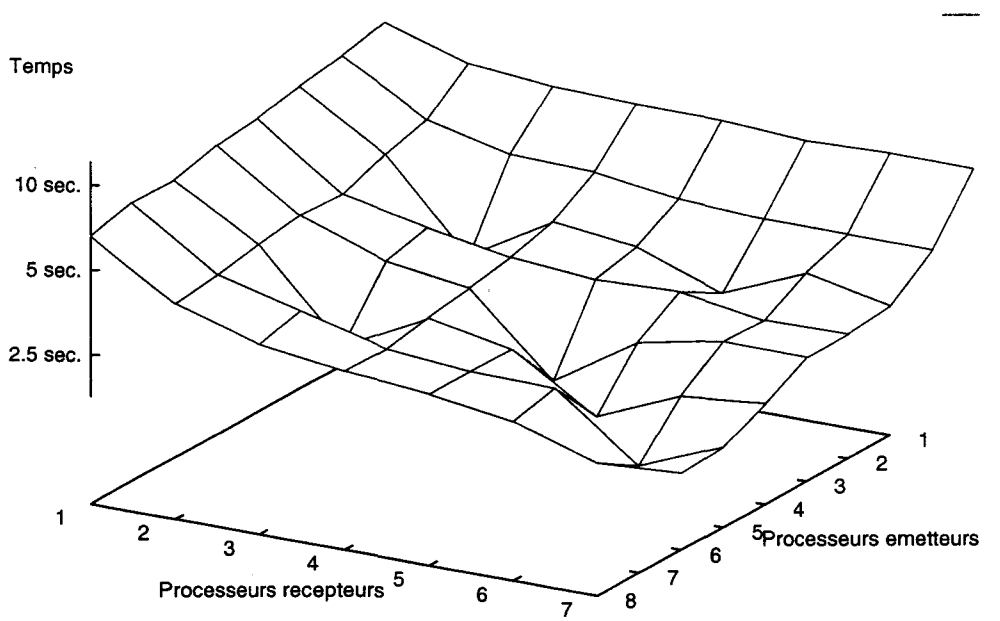


FIG. 3.35 - *Redistribution Cyclic(1) vers Cylic(8) avec l'algorithme spécialisé*

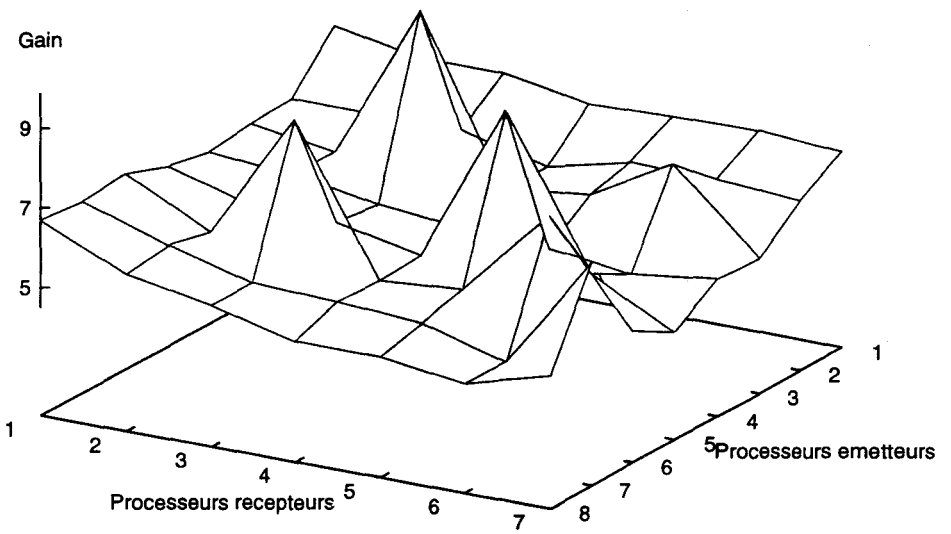


FIG. 3.36 - *Gain avec l'algorithme spécialisé Cyclic(K) vers Cylic(TK)*

4.6 Conclusions

Ces résultats montrent le potentiel important des algorithmes spécialisés tout en soulignant leurs limites : Pour tirer au maximum profit du recouvrement des calculs et des communications, il faut répartir au mieux la charge sur les processeurs de la machine cible. Dans la section suivante, nous allons pour chacune des redistributions étudiées, calculer l'ordre d'émission des messages qui minimise les déséquilibres de charge.

5 Répartition de la charge

Dans la section précédente nous avons mis en évidence des problèmes liés à un mauvais ordonnancement des calculs. En particulier nous avons observé des cas dans lesquels tous les messages étaient envoyés à un sous ensemble des processeurs cibles. Pour éviter ces cas défavorables nous allons maintenant mettre à profit la simplicité des équations de redistribution spécialisées pour modifier nos algorithmes de façon à répartir au mieux la charge de travail sur l'ensemble des processeurs cibles.

Dans la section (3) nous avons construit des algorithmes spécialisés. Dans chacun des cas :

- les processeurs sources énumèrent les indices des processeurs cibles à l'aide d'une même section régulière (au premier indice près) ;
- le volume des messages est à peu près identique sur tous les processeurs puisqu'ils sont construits avec les mêmes sections régulières (au première indice près).

Le temps pour constituer puis traiter ces messages est donc à peu près le même sur tous les processeurs. **On conserve donc le même schéma de communication pendant toute la durée de la redistribution.** Si deux processeurs sources adressent leur premier message à un même destinataire ils entreront systématiquement en conflit à chaque envoi de messages. Les indices des processeurs destinataires sont simplement translatés d'une étape à une autre. Cette propriété est essentielle : Si nous parvenons à calculer un décalage dans le parcours des sections régulières qui répartit au mieux les messages sur l'ensemble des processeurs cibles, cette propriété sera conservée pendant toute la durée de la redistribution.

Définitions À chaque étape de calcul chacun des P processeurs de la machine source, envoie un message à un processeur de la machine cible. Soit $\mathcal{F}_i(p)$ la fonction qui calcule le destinataire p' du message envoyé par le processeur p à l'étape i. Nous définissons la charge $\mathcal{C}_i(p')$, du processeur p', à l'étape i, comme étant égale au nombre de messages qui lui sont destinés à cette étape. On a donc $\mathcal{C}_i(p') = Cardinal(\mathcal{F}_i^{-1}(p'))$.

La répartition est qualifiée d'optimale lorsque la différence de charge entre deux processeurs quelconques de $\mathcal{M}_d$ est inférieur ou égale à un.

5.1 Redistribution $Block(K)$ vers $Block(K')$

Dans cette section nous allons montrer que la répartition de charge obtenue avec l'algorithme de redistribution $Block(K)$ vers $Block(K')$ (fig. 3.11) est optimale. Pour cela nous allons montrer que la différence de charge entre deux processeurs cibles est au plus égale à un.

Nous allons calculer le nombre de processeurs p qui s'adressent à p' lors de la première étape. Un processeur source calcule l'adresse de son premier correspondant en fonction de l'indice de la première donnée qu'il possède. Ces indices sont par définition, les multiples de K. La charge d'un processeur $p' \in [0, N/K']$ est donc $\mathcal{C}_0(p') = (K' - First(p'K', K) + 1)/K$ $(K'$

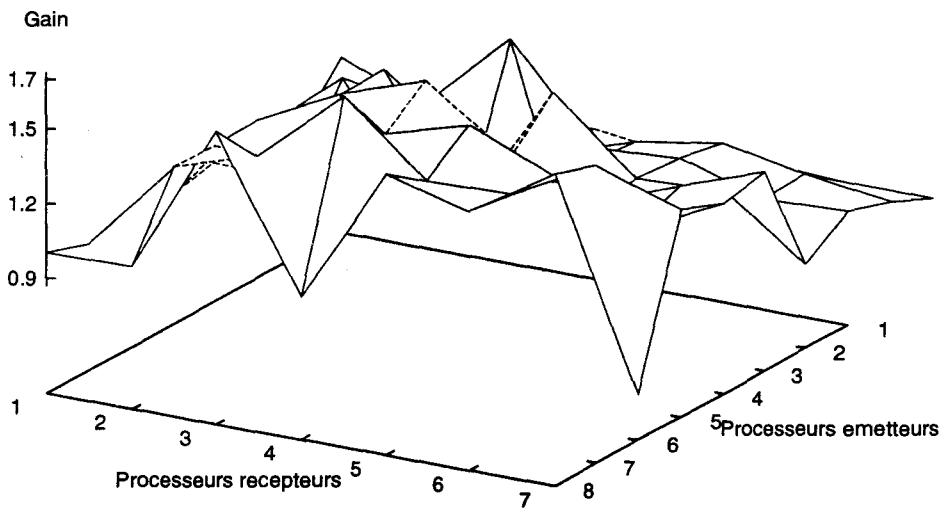


FIG. 3.37 - *Gain obtenu avec le décalage (redistribution Block(K) vers Cyclic(8))*

est la taille du bloc de données que possède P'. $First(p'K', K)$ est l'adresse dans ce bloc, du premier élément, dont l'adresse globale est divisible par K). Comme $First(p'K', K) \in [0, K[$, la différence de charge entre deux processeurs cibles de l'intervalle $[0, N/K'[$[18] est au pire égale à un.

5.2 Redistribution $Block(K)$ vers $Cyclic(K')$

Il est possible de distinguer deux cas :

Si $K \geq P'K'$: Chaque processeur de la machine de départ s'adresse à l'ensemble des processeurs de la machine d'arrivée[19]. Pour répartir au mieux les conflits, il suffit d'introduire un décalage dans le parcours des correspondants en fonction de l'indice du processeur émetteur. À l'étape i, le processeur d'indice p, s'adresse au processeur d'indice $p' = (p + i) \bmod P$. La fonction modulo est utilisée pour répartir les conflits cycliquement sur l'ensemble des processeurs de la machine d'arrivée. L'algorithme (fig. 3.13)

18. Les processeurs d'indices supérieurs n'ont pas le même nombre de données.
19. On ne considère que les processeurs qui ont effectivement des données.

reste valable, il suffit simplement d'ajouter le décalage au niveau du parcours des destinataires. Les gains obtenus en introduisant ce décalage sont présentés figure (3.37). On constate une amélioration sensible par rapport aux résultats précédents.

Si $K < P'K'$: Chaque processeur de la machine source s'adresse à un sous ensemble des processeurs de la machine d'arrivée. Comme les tailles des blocs K et K' ne sont pas nécessairement multiples, les ensembles de destinataires varient d'un processeur source à un autre. N'étant pas parvenu à identifier des classes de processeurs ayant même destinataires, nous ne proposerons pas d'algorithme de minimisation des conflits dans ce cas.

Remarque Le cas non traité ($K < P'K'$) n'est pas fréquent. Lorsqu'on utilise une distribution cyclique le nombre de cycles est généralement important (dans le cas contraire il est préférable d'utiliser une distribution par blocs beaucoup moins coûteuse). En général K est très grand devant $K'P'$

5.3 Redistribution $Cyclic(K)$ vers $Block(K')$

Cette redistribution est très proche de la précédente. Nous avons rencontré les mêmes problèmes et apporté les mêmes solutions.

Si $K' \geq PK$: Tout processeur de la machine de départ s'adresse à l'ensemble des processeurs de la machine d'arrivée. Comme pour le cas précédent nous allons introduire un décalage en fonction de l'indice du processeur émetteur. À l'étape i, le processeur d'indice p s'adresse au processeur d'indice $p' = (p+i) \, mod \, P$. L'algorithme (fig. 3.15) reste valable, il suffit simplement d'ajouter le décalage au niveau du parcours des destinataires. Les gains obtenus avec ce décalage sont présentés (fig. 3.38).

Si $K' < PK$: Un processeur de la machine source ne s'adresse qu'à un sous ensemble des processeurs de la machine d'arrivée. Comme les tailles des blocs K et K' ne sont pas nécessairement multiples, les ensembles de destinataires varient d'un processeur source à un autre.

Remarque Là encore le cas non traité correspond à une redistribution «dégénérée».

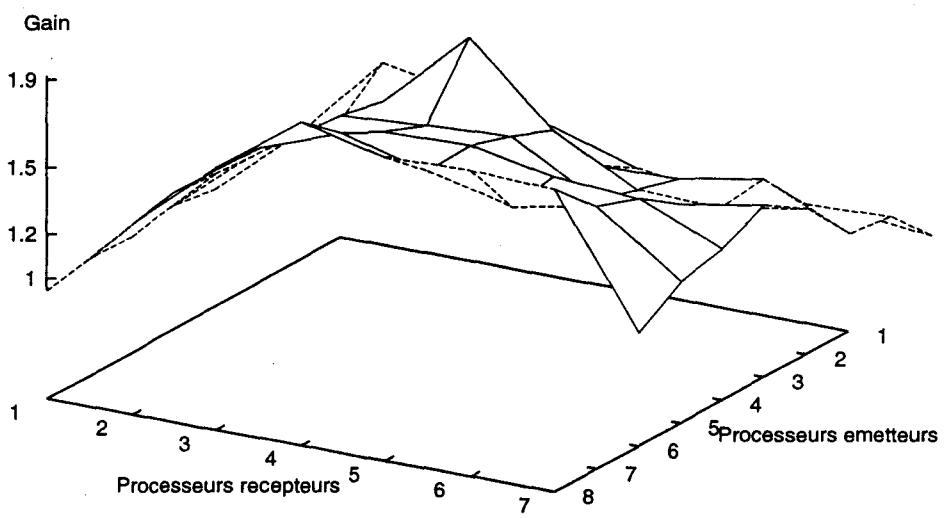


FIG. 3.38 - *Gain obtenu avec le décalage (Redistribution Cyclic(8) vers Block(K'))*

5.4 Redistribution $Cyclic(TK')$ vers $Cyclic(K')$

Pour minimiser les conflits nous allons tout d'abord caractériser les sous ensembles de processeurs sources qui s'adressent lors d'une même étape à un même processeur cible. Nous chercherons ensuite un ordonnancement qui minimise ces conflits.

D'après l'équation (3.19), deux processeurs sources p_1 et p_2 s'adressent à un même destinataire s'il existe deux entiers relatifs γ_1 et γ_2 et deux entiers naturels μ_1, μ_2 dans l'intervalle $[0, T[$ tel que :

$$\begin{cases} nPT - n'P' &= p' - p_1T - \mu_1 &= \gamma_1 D_1 \\ nPT - n'P' &= p' - p_2T - \mu_2 &= \gamma_2 D_1 \end{cases}$$

À l'aide de ces deux égalités nous allons calculer la distance $\Delta p = p_2 - p_1$ qui sépare deux processeurs qui entrent en conflit. Pour cela nous allons tout d'abord soustraire la seconde égalité de la première.

$$(p_2 - p_1)T + (\mu_2 - \mu_1) = (\gamma_1 - \gamma_2)D_1$$

On effectue ensuite la division de μ par q_1 de façon à regrouper les différents termes.

$$\begin{aligned} & \Delta pT + \Delta\mu_1 && = -\Delta\gamma_1 D_1 \\ \Leftrightarrow\ & \Delta pT + \Delta q_1 D_1 + \Delta r_1 && = -\Delta\gamma_1 D_1 \\ \Leftrightarrow\ & \Delta pT + \Delta r_1 && = D_1(-\Delta\gamma - \Delta q_1) \end{aligned}$$

Pour minimiser le nombre de conflit, il faut minimiser le nombre de solutions possibles. Comme le nombre de solutions augmente avec le nombre de variables (dimension du problème), nous allons utiliser la même valeur de q_1 sur tous les processeurs de façon à poser $\Delta q_1 = 0$. On obtient ainsi :

$$\Delta pT + \Delta\gamma D_1 = -\Delta r_1$$

Soit $D_2 = PGCD(D_1, T)$ et $q_2, r_2 \in \mathbf{N}$ tel que $r_2 \in [0, D_2[$ et $r_1 = q_2 D_2 + r_2$. L'équation précédente admet des solutions si et seulement si Δr_1 est divisible par D_2. Pour mieux expliciter les valeurs de r_1 puis de μ qui génèrent des conflits on effectue finalement la division entière de r_1 par D_2.

$$\Delta pT + \Delta\gamma D_1 = -\Delta q_2 D_2 - \Delta r_2$$

Cette dernière équation admet des solutions si et seulement si la valeur $(\Delta q_2 D_2 + \Delta r_2)$ est divisible par D_2. Il y a donc des conflits potentiels entre deux processeurs sources si :

$$\Delta p = -\Delta q_2 + \frac{\lambda D_1}{D_2} \quad \text{et} \quad \Delta r_2 = 0 \tag{3.28}$$

Avec cette équation nous pouvons calculer la distance qui sépare deux processeurs susceptible de s'adresser au même moment à un même processeur cible. Nous allons maintenant calculer un ordonnancement des messages qui minimise les conflits. D'après l'équation (3.20), chaque processeur source d'indice p s'adresse aux processeurs cibles d'indice :

$$p' = (pT + r_1)\ mod D_1 + \lambda_2 D_1$$

Chaque processeur parcours les indices des processeurs avec lesquels il va communiquer à l'aide de deux sections régulières qui énumèrent les valeurs de r_1 et de λ_2. Comme $r_1 =$

$q_2 D_2 + r_2$, nous disposons de trois degrés de liberté. Nous pouvons parcourir les valeurs de r_2, q_2 et λ_2 dans un ordre quelconque.

Pour minimiser le nombre de conflits potentiels nous allons choisir la même valeur de q_2 sur tous les processeurs de façon à poser Δq_2 dans l'équation (3.28). Nous obtenons ainsi :

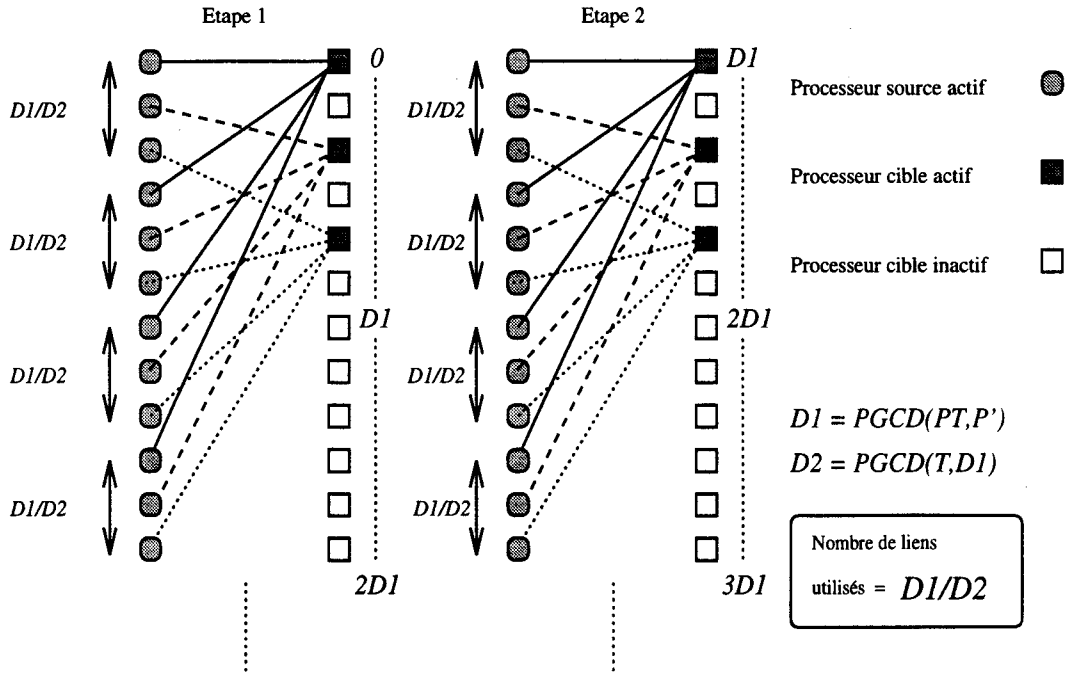
$$\Delta p' = 0 \Longleftrightarrow \Delta p = \frac{\lambda D_1}{D_2} \text{ et } \Delta r_2 = 0 \tag{3.29}$$

Deux processeurs entrent en conflit, si la distance qui les sépare est multiple de D_1/D_2(fig. 3.39(a)). Nous pouvons donc donner la même valeur de r_2 à D_1/D_2 processeurs consécutifs sans que cela ne génère de conflit. Il suffit ensuite d'incrémenter cette variable tous les D_1/D_2 processeurs pour éviter les conflits. Comme il y a exactement D_2 valeurs de r_2 distinctes et que P' est divisible par D_1 (car $D_1 = PGCD(PT, P')$), nous pouvons ainsi attribuer des destinataires différents aux D_1 premiers processeurs sources.

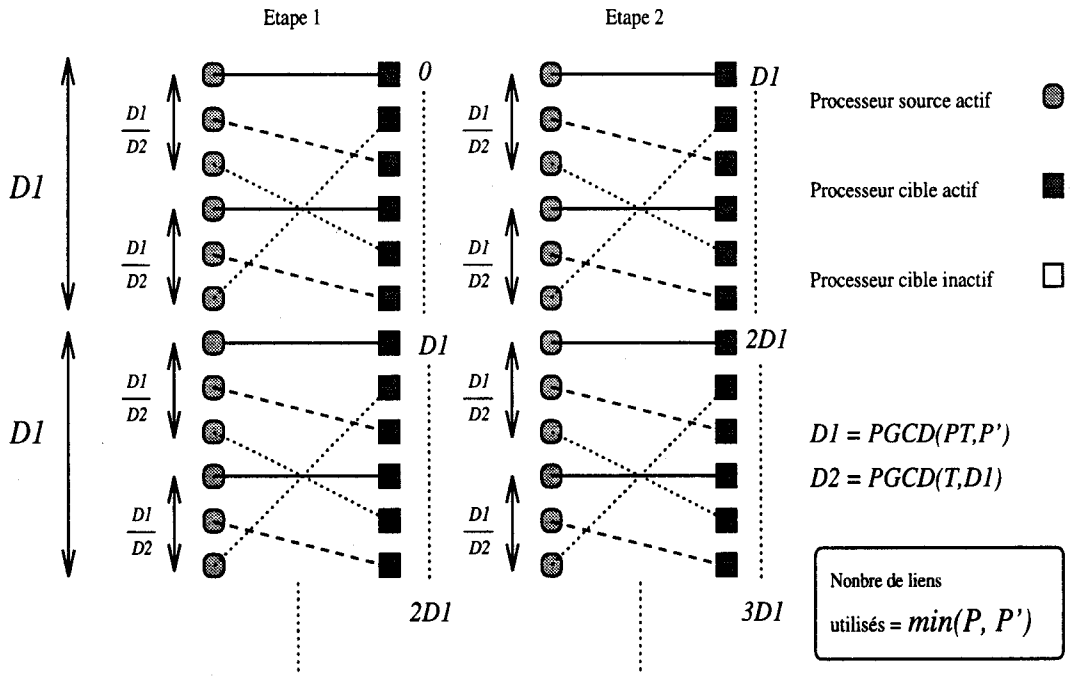
Pour minimiser les conflits résiduels nous allons maintenant mettre à profit le second degré de liberté (valeur de λ_2) disponible avec l'équation (3.20) de façon à projeter les processeurs d'indices $[zD_1, (z+1)D_1 - 1]$ sur les processeurs cibles d'indices $[(zD_1) mod\, P', ((z+1)D_1 - 1) mod\, P']$. Pour cela il suffit d'incrémenter la valeur de λ_2 tous les D_1 processeurs. Comme il y a exactement P'/D_1 valeurs de λ_2 possibles, la charge des processeurs sur la machine d'arrivée est égale à :

$$div(P, D_1)/(P'/D_1) \leq C_0(p') \leq div(div(P, D_1), (P'/D_1)) \tag{3.30}$$

La différence de charge entre deux processeurs cibles est donc au pire égale à un. En décalant le parcours des sections régulières qui énumèrent les valeurs de r_1 et λ_2, nous sommes parvenus à minimiser les conflits. La figure (3.39(a)) représente les conflits qui sont générés avec l'algorithme initial. On remarque que deux processeurs sources s'adressent à un même destinataire si la distance qui les sépare est un multiple de D_1/D_2 ; il y a exactement D_1/D_2 processeurs cibles actifs pendant toute la durée de la redistribution (les conflits sont simplement translatés d'une étape à une autre). La figure (3.39(b)) représente le schéma de communications que l'on peut obtenir en introduisant des décalages. On voit bien sur celui-ci qu'il n'y a pas de conflit si le nombre de processeurs cibles est suffisant ($P \leq P'$), dans le cas contraire, les conflits sont répartis équitablement.



(a) Sans répartition de charge (algorithme spécialisé fig. 3.17)



(b) Avec répartition de charge (algorithme spécialisé fig. 3.41)

FIG. 3.39 - *Répartition de charge pour la redistribution Cyclic(TK') vers Cyclic(K')*

5.4.1 Illustration

Nous allons maintenant vérifier la validité de nos hypothèses. Peut-on retrouver par le calcul les maxima observés figure (3.40(a))?

D'après l'équation (3.28), deux processeurs sources s'adressent à un même destinataire si la «distance»[20] qui les sépare est multiple de D_1/D_2. Cela signifie qu'il y a exactement D_1/D_2 processeurs cibles actifs (fig. 3.39(a)). Nous allons calculer la valeur de D_1/D_2 pour $P \in \{6, 7, 8\}$.

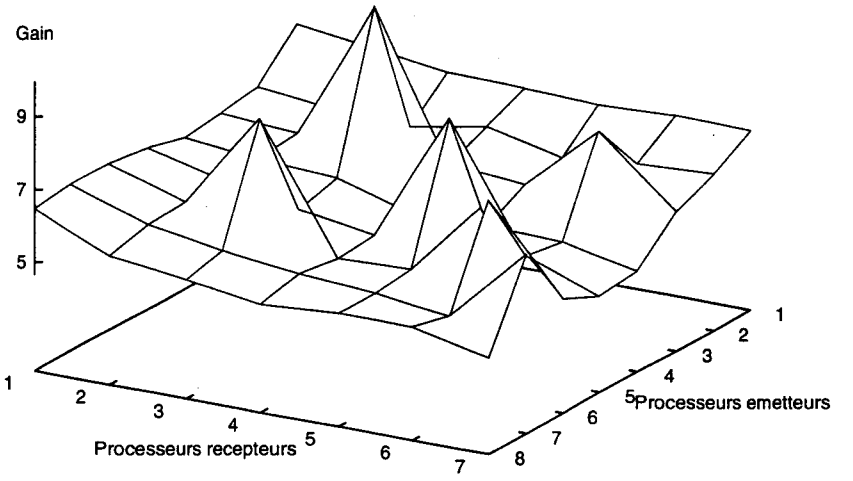
- Nombre de processeurs cibles actifs pour $P = 8$
  $D_1 = GCD(PT, P') = GCD(8 * 8, P') = 2^\gamma$
  $D_2 = GCD(D_1, T) = GCD(2^\gamma, 8) = 2^\gamma$
  On a $D_1/D_2 = 2^\gamma/2^\gamma = 1$ quelle que soit la valeur de P'.
  Le nombre de processeur cible actif est donc toujours égal à un. Cela explique les mauvais résultats pour $P = 8$

- Nombre de processeurs cibles actifs pour $P = 7$
  $D_1 = GCD(PT, P') = GCD(7 * 8, P') = 2^\gamma 7^\beta$
  $D_2 = GCD(D_1, T) = GCD(2^\gamma 7^\beta, 8) = 2^\gamma$
  On a $D_1/D_2 = 2^\gamma 7^\beta/2^\gamma = 7^\beta$
  Le nombre de processeurs cibles actifs est donc toujours égal à un sauf pour $P' = 7$. Cela est parfaitement en accord avec les résultats obtenus.

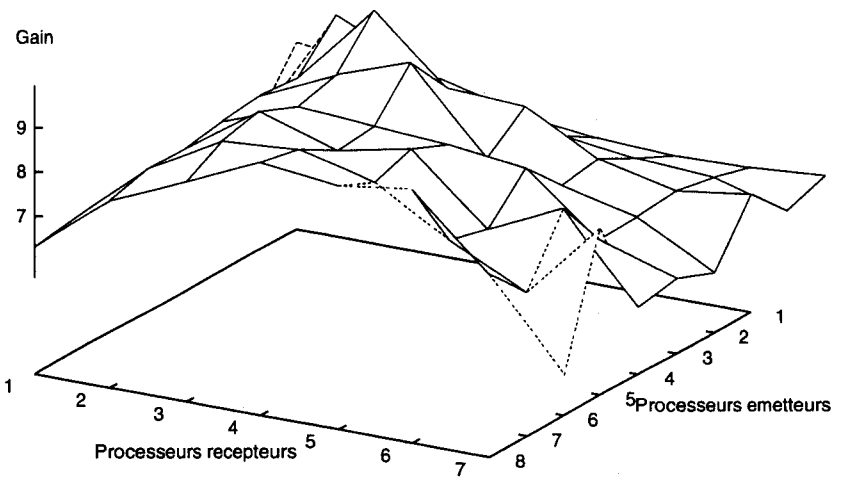
- Nombre de processeurs cibles actifs pour $P = 6$
  $D_1 = GCD(PT, P') = GCD(6 * 8, P') = 2^\gamma 3^\beta$
  $D_2 = GCD(D_1, T) = GCD(2^\gamma 3^\beta, 8) = 2^\gamma$
  On a $D_1/D_2 = 2^\gamma 3^\beta/2^\gamma = 3^\beta$
  Là encore, les résultats expérimentaux corroborent les calculs puisque l'on observe deux pics de performances pour $P = 3$ et $P' = 6$.

Les irrégularités constatées sont donc bien liées à la présence ou à l'absence de conflits. Pour éviter les cas défavorables nous avons mis en place un nouvel algorithme, celui-ci est décrit dans le paragraphe suivant.

20. Différence entre leurs indices.



(a) Sans répartition de charge(algorithme spécialisé fig. 3.17)



(b) Avec répartition de charge(algorithme spécialisé fig. 3.41)

FIG. 3.40 - *Gain avec la redistribution $Cyclic(TK')$ vers $Cyclic(K')$*

```
GCD(P*T,P',&D1,&ALPHA1,&BETA1);
GCD(T,D1,&D2,&ALPHA2,&BETA2);

/*  on découpe l'espace des processeurs sources en sous espaces */
/*  de taille D1 et chaque processeur calcul un deplacement en  */
/*  fonction du sous espace dans lequel il est place.           */

delta_lambda1=p/D1;

/*  on redécoupe ensuite chaque sous espaces de façon à obtenir */
/*  un second niveau de décalage.                               */

delta_r2=(p%D1)/(D1/D2);

/*  Enfin, on calcule le nombre de valeurs possible pour chaque */
/*  variable de façon à repartir les conflits à l'aide d'une    */
/*  fonction modulo.                                            */

nb_r1=min(T,D1);
nb_lambda1=div(P',D1);


for (i=0; i<nb_r1; i++){
  r1=(i+delta_r2)%nb_r1;
  p'_min=(p*T+r1)%D1;
  for (j=delta_lambda1; j<delta_lambda1+nb_lambda1; j++){
    p'=p'_min+(j%nb_lambda1)*D1;
    initsend();
    nb_send=0;
    for (mu=r1; mu<T; mu+=D1){
      nb_send+=remplir();
    }
    if (nb_send > 0) send(p');
  }
}
```

FIG. 3.41 - *Redistribution Cyclic(TK') vers Cyclic(K') avec répartition de charge*

5.4.2 Algorithme de redistribution $Cyclic(TK')$ vers $Cyclic(K')$

Pour construire un algorithme de redistribution $Cyclic(TK')$ vers $Cyclic(K')$ qui répartit au mieux les messages sur les processeurs cibles, nous avons modifié l'algorithme proposé figure (3.17), de façon à intégrer les décalages calculés dans le paragraphe précédent. Le nouvel algorithme est présenté figure (3.41). Les modifications ne portent que sur les deux boucles chargées d'énumérer les valeurs des variables λ_1 et r_1. Les fonctions `Remplir` et `Vider` ne sont pas modifiées. L'algorithme de réception reste également valable.

5.4.3 Résultats expérimentaux

Les résultats obtenus avec ce nouvel algorithme sont présentés figure (3.39b). Les performances sont beaucoup plus homogènes et toujours supérieures à celles obtenues avec l'algorithme précédent. Comme l'algorithme utilise au mieux tous les liens de communications, le gain est maximal lorsque $P = P'$.

5.5 Redistribution $Cyclic(K)$ vers $Cyclic(TK)$

Dans cette sous section nous allons chercher à calculer les décalages qui minimisent les conflits lorsqu'on passe d'une distribution $Cyclic(K)$ à une distribution $Cyclic(TK)$. Comme dans le cas précédent, nous allons tout d'abord caractériser les processeurs sources qui s'adressent à un même destinataire. Pour cela nous utilisons l'équation (3.23) :

$$\begin{cases} nP - n'P'T &= p'_1T - p_1 + \mu_1 &= \gamma_1 D_1 \\ nP - n'P'T &= p'_2T - p_2 + \mu_2 &= \gamma_2 D_1 \end{cases}$$

À l'aide de ces deux égalités nous calculons la distance $\Delta p = p_2 - p_1$ qui sépare deux processeurs qui entrent en conflit. Tout d'abord on soustrait la seconde égalité de la première.

$$-(p_2 - p_1) + (\mu_2 - \mu_1) = (\gamma_2 - \gamma_1)D_1$$

On effectue ensuite la division de μ par q_1 [21], puis la division de r_1 par D_2 [22] de façon à regrouper les différents termes.

$$-\Delta p + \Delta r_1 = D_1(\Delta\gamma - \Delta q_1)$$

$$\Delta p - \Delta r_2 = D_1(\Delta q_1 - \Delta\gamma) + D_2\Delta q_2$$

Pour minimiser le nombre de conflit, il faut minimiser le nombre de solutions possibles. Pour cela on utilise la même valeur de q_2 sur tous les processeurs de façon à poser $\Delta q_2 = 0$. On obtient ainsi :

$$\Delta p = \Delta r_2 + \lambda D_1 \Longleftrightarrow |\Delta p| \; mod \; D_1 = \Delta r_2 \tag{3.31}$$

Cette équation montre que deux processeurs sources p_1 et p_2 sont susceptibles de s'adresser au même destinataire si la différence entre leurs indices est inférieure à D_2 (à un modulo près).

Dans les paragraphes suivants nous allons partitionner l'espace des processeurs sources en D_1/D_2 classes disjointes. Nous montrerons que deux processeurs appartenant à deux classes différentes ne peuvent entrer en conflit. Nous proposerons ensuite une fonction pour minimiser les conflits qui opposent les processeurs d'une même classe.

5.5.1 Définition

La classe du processeur d'indice p est : $Classe(p) = (p \; mod \; D_1)/D_2$

5.5.2 Propositions

1. Il y a exactement D_1/D_2 classes distinctes.
2. Le nombre de processeurs est identique dans chacune des classes.
3. Tous les processeurs d'une même classe s'adressent aux mêmes destinataires.
4. Deux processeurs de classes distinctes ne s'adressent jamais au même destinataire.

21. $\mu = q_1 D_1 + r_1$.
22. $r_1 = q_2 D_2 + r_2$.

5.5.3 Preuves

Nous allons maintenant démontrer les propositions précédentes.

Proposition 1 C'est immédiat de par la définition des classes.

Proposition 2 Le nombre de processeurs est identique dans chacune des classes car $D_1 = PGCD(P, P'T)$ et $D_2 = PGCD(D_1, T)$ donc D_1 divise P et D_2 divise D_1.

Proposition 3 Démontrons que deux processeurs p_1, p_2 appartenant à la même classe i s'adressent aux mêmes destinataires. D'après la définition d'une classe, on a:

$$\begin{cases} (p_1 \ mod \ D_1)/D_2 = i \\ (p_2 \ mod \ D_1)/D_2 = i \end{cases} \iff \begin{cases} (p_1 \ mod \ D_1) - iD_2 = r_2 & \text{avec } r_2 = p_1 \ mod D_2 \\ (p_2 \ mod \ D_1) - iD_2 = r'_2 & \text{avec } r'_2 = p_2 \ mod D_2 \end{cases}$$

$$\iff \begin{cases} p_1 - \gamma D_1 - iD_2 = r_2 & \gamma \in \mathbb{Z} \\ p_2 - \gamma' D_1 - iD_2 = r'_2 & \gamma' \in \mathbb{Z} \end{cases}$$

Finalement on a $\Delta p = \lambda D_1 + \Delta r_2$. Tous les processeurs d'une même classe sont donc susceptibles d'entrer en conflit (Ils s'adressent tous aux mêmes destinataires mais pas nécessairement dans le même ordre).

Proposition 4 Nous allons maintenant démontrer que deux processeurs appartenant à deux classes distinctes ne s'adressent jamais au même destinataire.

$$\text{On a} \begin{cases} p_1 \in Classe_i \\ p_2 \in Classe_j \\ i, j \in [0, \frac{D_1}{D_2}[\\ i \neq j \end{cases} \text{ d'où } \begin{cases} p_1 = iD_2 + \gamma D_1 + r_2 \\ p_2 = jD_2 + \gamma' D_1 + r'_2 \end{cases}$$

d'où $\Delta p = (j-i)D_2 + \Delta\gamma D_1 + \Delta r_2 \neq \lambda D_1 + \Delta r_2$ car $(j-i) \in]-\frac{D_1}{D_2}, 0[\cup]0, \frac{D_1}{D_2}[$.

Deux processeurs appartenant à deux classes différentes ne peuvent donc pas entrer en conflit.

5.5.4 Minimisation des conflits dans une classe

Nous avons montré dans les paragraphes précédents que les conflits sont confinés dans les classes. Dans ce paragraphe nous allons calculer un ordonnancement des messages qui minimise les conflits qui opposent les processeurs d'une même classe. D'après l'équation (3.24), le processeur p communique avec les processeurs p' dont les indices vérifient :

$$p' = \frac{\alpha_2(p-r_1)}{D_2} + \frac{\lambda_2 D_1}{D_2}$$

Nous avons choisi d'utiliser la même valeur de q_2 sur tous les processeurs pour réduire le nombre de conflits potentiels. La valeur de r_1 est donc fixée (car $r_1 = q_2 D_2 + p \; mod \; D_2$). Il nous reste un unique degré de liberté : le choix de la valeur de λ_2.

Pour minimiser les conflits, il faut répartir «au mieux» les différentes valeurs de λ_2. Pour cela, nous allons associer à chaque processeur une classe et un indice de classe, puis nous distribuerons cycliquement toutes les valeurs de λ_2 en fonction de l'indice de classe de chaque processeur.

Définition L'indice de classe d'un processeur est : $\mathcal{I}_c(p) = p \; mod \; D_2 \; + \; (P/D_1) * D_2$. À chaque processeur p on associe un unique couple $(Classe(p), \mathcal{I}_c(p))$.

La relation est bijective

$$\begin{aligned} &Classe(p) * D_2 + \frac{\mathcal{I}_c(p)}{D_2} * D_1 + \mathcal{I}_c(p) mod D_2 \\ &= (pmodD_1 - pmodD_2) + (p - pmodD_1) + (pmodD_2) = p \end{aligned}$$

Calcul des décalages Démontrons qu'il existe exactement $P'/(D_1/D_2)$ valeurs de λ_2 solutions de l'équation de redistribution quelle que soit la valeur de p. Cette propriété sera ensuite utilisée pour calculer les décalages $\Delta\lambda_2$ puis pour calculer la charge d'un processeur cible.

$$\text{On a} \begin{cases} D_1 = GCD(P, P'T) \\ D_2 = GCD(T, D1) \end{cases} \quad \text{d'où} \begin{cases} P'T = \gamma_1 D_1 \\ T = \gamma_2 D_2 \\ D_1 = \gamma_3 D_2 \\ \alpha_2 \frac{T}{D_2} + \beta_2 \frac{D_1}{D_2} = 1 \end{cases}$$

D'après le théorème de Bezout, T/D_2 et D_1/D_2 sont premiers entre eux. Or $P'(T/D_2)$ est multiple de D_1/D_2. P' est donc nécessairement multiple de D_1/D_2. Nous venons de démontrer qu'il existe exactement $N_{\lambda_2} = P'/(D_1/D_2)$ valeurs de λ_2 possibles quelle que soit la valeur de p. Cela signifie que chacun des processeurs sources s'adresse à exactement $P'/(D_1/D_2)$ processeurs cibles. Nous allons distribuer cycliquement les valeurs de λ_2 dans chacune des classes à l'aide de la fonction suivante :

$$\Delta\lambda_2 = \mathcal{I}_c(p) \; mod \; N_\lambda \tag{3.32}$$

Grâce au décalage calculé en (3.32), les N_{λ_2} premiers processeurs d'une classe donnée, parcourent la même section régulière sans entrer en conflit. On répète ensuite ce schéma de communication de façon cyclique. La charge d'un processeur cible p' est alors :

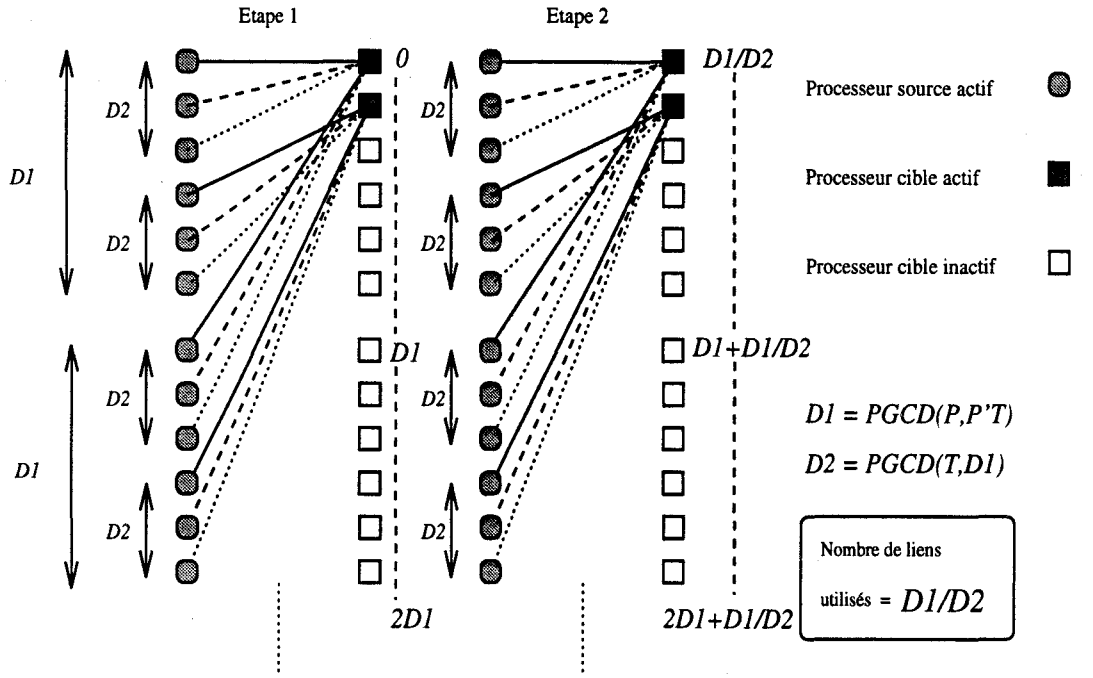
$$\frac{P/N_{classe}}{N_{\lambda_2}} \leq \mathcal{C}_0(p') \leq div(P/N_{classe}, N_{\lambda_2}) \Longleftrightarrow \frac{P}{P'} \leq \mathcal{C}_0(p') \leq div(P', P) \tag{3.33}$$

La différence de charge entre deux processeurs cibles est donc au pire égale à un. Nous sommes parvenus à minimiser les conflits en décalant le parcours de la section régulière qui énumère les valeurs de λ_2. La figure (3.42a) représente les conflits qui sont générés avec l'algorithme initial. La figure (3.42b) représente le schéma de communication qu'on obtient en introduisant les décalages précédemment calculés.

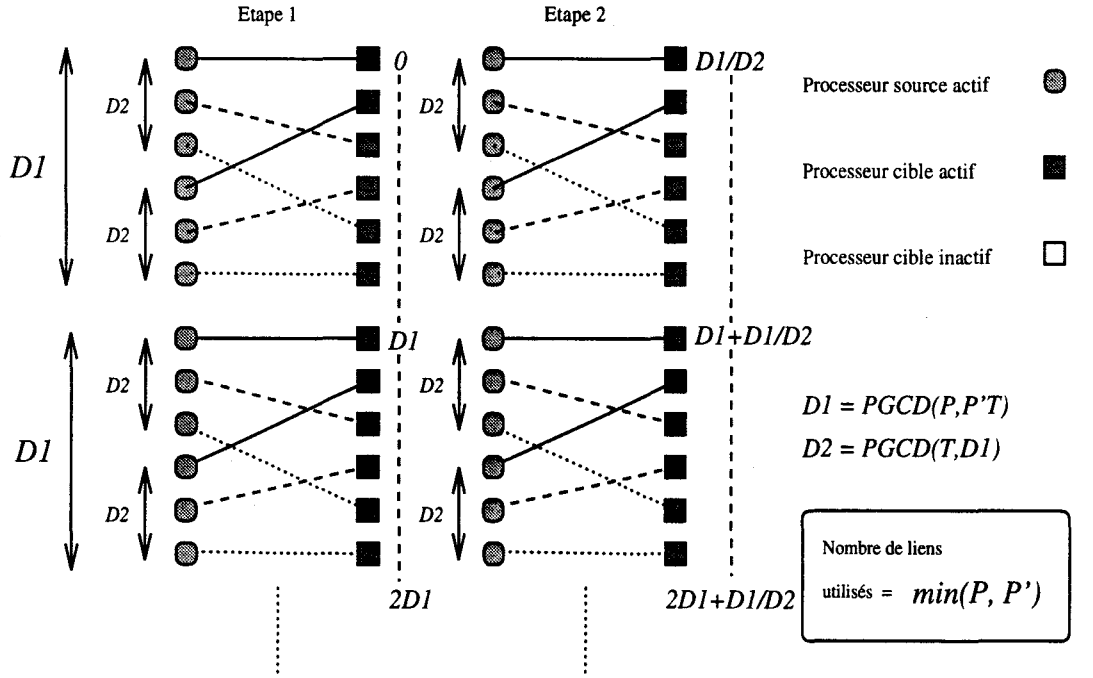
5.5.5 Illustration

Expliquons les maximums locaux observés figure (3.43a). D'après l'équation (3.31), deux processeurs sources s'adressent à un même destinataire si la «distance» qui les sépare est inférieur à D_2 à un modulo D_1 près. Cela signifie qu'il y a exactement D_1/D_2 processeurs cibles actifs (fig. 3.42a). Nous allons calculer la valeur de D_1/D_2 pour $P \in \{6, 7, 8\}$.

- Nombre de processeurs cibles actifs pour $P = 8$
  $D_1 = GCD(P, P'T) = GCD(8, 8P') = 8$
  $D_2 = GCD(D_1, T) = GCD(8, 8) = 8$
  On a $D_1/D_2 = 8/8 = 1$ quelle que soit la valeur de P'.
  Le nombre de processeurs cibles actifs est donc toujours égal à un. Cela explique les mauvais résultats pour $P = 8$

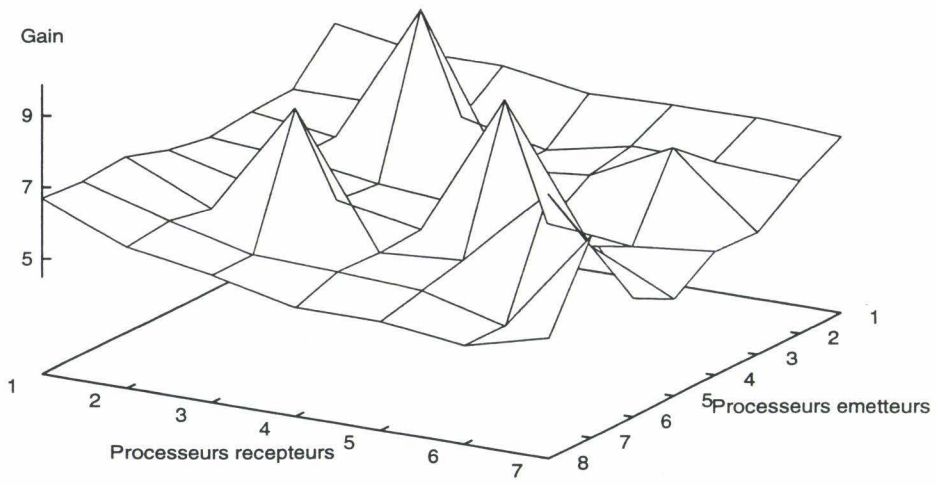


(a) sans répartition de charge (algorithme spécialisé fig. 3.19)

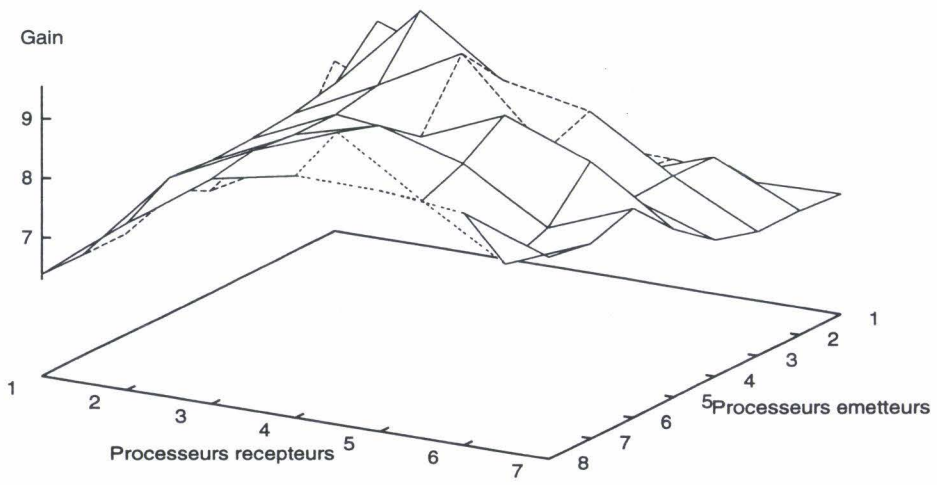


(b) Avec répartition de charge (algorithme spécialisé fig. 3.44)

FIG. 3.42 - *Répartition de charge pour la redistribution $Cyclic(K)$ vers $Cyclic(TK)$*



(a) sans répartition de charge (algorithme spécialisé fig. 3.19)



(b) avec répartition de charge (algorithme spécialisé fig. 3.44)

FIG. 3.43 - *Gain avec la redistribution Cyclic(K) vers Cyclic(TK)*

- Nombre de processeurs cibles actifs pour $P = 7$
  $D_1 = GCD(P, P'T) = GCD(7, 8P') = 7^\beta$
  $D_2 = GCD(D_1, T) = GCD(7^\beta, 8) = 1$
  On a $D_1/D_2 = 7^\beta/1 = 7^\beta$
  Le nombre de processeurs cibles actifs est donc toujours égal à un sauf pour $P' = 7$. Cela est parfaitement en accord avec les résultats obtenus.

- Nombre de processeurs cibles actifs pour $P = 6$
  $D_1 = GCD(P, P'T) = GCD(6, 8P') = 2 * 3^\beta$
  $D_2 = GCD(D_1, T) = GCD(2 * 3^\beta, 8) = 2$
  On a $D_1/D_2 = 2 * 3^\beta/2 = 3^\beta$
  Là encore, les résultats expérimentaux corroborent les calculs puisque l'on observe deux pics de performances pour $P = 3$ et $P' = 6$.

Les irrégularités constatées semblent donc bien liées à la présence de conflits.

5.5.6 Algorithme de redistribution $Cyclic(K)$ vers $Cyclic(TK)$

Pour construire un algorithme de redistribution $Cyclic(K)$ vers $Cyclic(TK)$ qui répartisse au mieux les messages sur les processeurs cibles, nous avons modifié l'algorithme présenté figure (3.19), de façon à intégrer le décalage précédemment calculé. Ces modifications ne portent que sur la boucle chargée d'énumérer les valeurs de λ_2. Cet algorithme modifié est présenté figure (3.44).

5.5.7 Résultats expérimentaux

Les résultats obtenus avec ce nouvel algorithme sont présentés figure (3.43b). Les performances sont comparables à celles obtenues avec l'algorithme de redistribution $Cyclic(TK')$ vers $Cyclic(K')$ optimisé. Dans les deux cas, le coût du calcul des décalages est négligeable par rapport au gain qui résulte de la bonne utilisation du réseau.

6 Redistributions multi-dimensionnelles

Dans cette section nous montrerons qu'il est possible d'étendre la méthodologie précédemment développée de façon à migrer un tableau T de dimension Φ d'une grille de processeurs

```
GCD(P,P'*T,&D1,&ALPHA1,&BETA1);
GCD(T,D1,&D2,&ALPHA2,&BETA2);

/*        Il n'y a qu'un seul niveau de décalage                */
/*        L'espace des processeurs sources est découpé en sous  */
/*        espaces de taille D2. Chaque processeur calcul son    */
/*        propre décalage en fonction du sous espace dans       */
/*        lequel il se trouve.                                  */

nb_lambda2=P'/(D1/D2);
delta_lambda2=p%D2+(p/D1)*D2;

for (r1=p%D2; r1 < min(D1,T); r1+=D2){
  lambda2=div(-ALPHA2*(p-r1)/D2,D1/D2);
  p'_min=ALPHA2*(p-r1)/D2+lambda2*D1/D2;
  for (j=delta_lambda2; j<delta_lambda2+nb_lambda2; j++){
    p'=p'_min+(j%nb_lambda2)*(D1/D2);
    initsend();
    placee=0;
    for (mu=r1; mu<T; mu+=D1){
      placee+=remplir2();
    }
    if (placee > 0) send(p);
  }
}
```

FIG. 3.44 - *Redistribution Cyclic(K) vers Cyclic(TK) avec répartition de charge*

à une autre.

6.1 Fonctions d'adressages multi-dimensionnelles

Soit $T(N_0, \ldots, N_{\Phi-1})$, un tableau de dimension Φ, distribué $Cyclic(K_0, \ldots, K_{\Phi-1})$ sur une grille $G(P_0, \ldots, P_{\rho-1})$ de processeurs. Pour simplifier notre prendrons $\rho = \Phi$. Cela est toujours possible quelle que soit la dimension de la grille. Il suffit simplement de renuméroter les processeurs où les éléments de tableaux.

Si $\Phi > \rho$

On modifie le vecteur d'indice $(p_0, p_1, \ldots, p_{\rho-1})$ de chaque processeur de la façon suivante :

$$(p_0, p_1, \ldots, p_{\rho-1}, \underbrace{0, 0, \ldots, 0}_{\Phi-\rho+1})$$

Si $\Phi < \rho$

On ne considère que les processeurs placés sur les Φ premières dimensions de la grille.

Nous pouvons ainsi identifier chaque élément de T par un vecteur d'indices $\vec{\imath}$ de dimension Φ auquel on associe les vecteurs $\vec{p}$, $\vec{n}$, $\vec{x}$ suivants :

$$\left\{ \begin{array}{lcll} p_\varphi & = & (i_\varphi / k_\varphi) \bmod P_\varphi & \\ n_\varphi & = & i_\varphi / P_\varphi K_\varphi & \Rightarrow t(\vec{\imath}) \text{ est sur } \vec{p} \text{ ssi } \; i_\varphi = p_\varphi K_\varphi + n_\varphi P_\varphi K_\varphi + x_\varphi \\ x_\varphi & = & i_\varphi \bmod K_\varphi & \end{array} \right. \tag{3.34}$$

$$\forall\ \varphi \in [0, \Phi[\text{ avec } x_\varphi \in [0, K_\varphi[,\ n_\varphi \in [0, L(p_\varphi)/K_\varphi[, \text{ et } p_\varphi \in [0, P_\varphi[.$$

En utilisant ces vecteurs de n-uplets, nous allons maintenant calculer les ensembles de données qu'il va falloir migrer entre chaque couple de processeurs.

6.2 Modélisation des redistributions multi-dimensionnelles

On cherche à caractériser l'ensemble $I(V_m, V'_{m'})$ des éléments à migrer entre les couples de processeurs $(V_m, V'_{m'})$ pour passer d'une distribution source D, $Cyclic(K_1, \ldots, K_\Phi)$ sur une grille $G(P_1, \ldots, P_\Phi)$, à une distribution D', $Cyclic(K_1, \ldots, K_\Phi)$, sur une grille $G'(P'_1, \ldots, P'_\Phi)$. Cette ensemble est obtenu par résolution du système suivant :

$$\forall \varphi \in [0, \Phi[,\ i_\varphi \in I(p_\varphi, p'_\varphi) \text{ ssi } i_\varphi \in E_D(p_\varphi) \cap E_{D'}(m'_\varphi)$$

On a donc d'après (3.34) (3.35)

$$p_\varphi K_\varphi + n_\varphi P_\varphi K_\varphi + x_\varphi \;=\; p'_\varphi K'_\varphi + n'_\varphi P'_\varphi K'_\varphi + x'_\varphi$$

On remarque que ces Φ équations diophantiennes sont complètement indépendantes. On peut donc réutiliser les algorithmes précédemment proposés.

7 Conclusion

Dans ce chapitre, nous avons proposé une modélisation des redistributions hétérogènes $Cyclic(K)$ vers $Cyclic(K')$. À l'aide de cette représentation, nous avons construit des algorithmes spécialisés capables de réaliser des redistributions spécifiques à l'exécution. Ces algorithmes bénéficient des propriétés suivantes :

le nombre de message est minimal ;

le volume des communications est minimal ;

les calculs sont réalisés dynamiquement ;

le coût du calcul d'adresse ne dépend que des paramètres des redistributions ;

les calculs sont recouverts par les communications ;

les algorithmes SPMD d'émission et de réception s'exécutent en parallèle.

En utilisant les équations spécifiques à chacune des redistributions nous avons de plus réussi à calculer un ordonnancement des calculs qui minimise les déséquilibres de charges. Les mesures effectuées sur une ferme de processeurs ALPHA montrent que le coût des algorithmes proposés dépend essentiellement des performances de la bibliothèque de communication et du réseau qu'elle utilise.

Dans le prochain chapitre, nous utiliserons les algorithmes de redistribution qui ont été développés pour mettre en œuvre notre système de fichiers à parallélisme de données. Avec ces algorithmes dynamiques efficaces nous montrerons qu'il est possible de concevoir une bibliothèque d'entrées/sorties parallèles à la fois portable et performante. Avec cette bibliothèque le programmeur n'a pas à connaître la distribution des données au niveau du système de fichiers. Les redistributions sont effectuées dynamiquement et de façon transparente pour l'utilisateur.

Chapitre 4

Système d'entrées/sorties parallèles

D'après le petit Robert, l'informatique est «La science de l'information» ; c'est «L'ensemble des techniques de la collecte, du tri, de la mise en mémoire, de la transmission et de l'utilisation des informations traitées automatiquement». Cette définition fait bien apparaître l'importance des entrées/sorties. Tous les programmes effectuent des traitements sur des données. Cela suppose des mécanismes pour accéder les valeurs à traiter puis pour conserver une trace des résultats obtenus de façon à pouvoir les exploiter ultérieurement. Les périphériques d'entrées/sorties sont très nombreux. Les données peuvent provenir d'un capteur de température, d'un clavier ou d'un disque. Les résultats peuvent être affichés à l'écran, sauvegardés sur une bande ou envoyés à un dispositif de guidage.

Le système Unix fournit une abstraction de tous ces supports de données physiques. Grâce à cette abstraction, le programmeur peut lire ou écrire des données indépendamment de la nature du support. Il n'a pas à se soucier de la provenance ou du devenir des données qu'il traite. Les données sont toujours accédées de la même façon à l'aide du système de fichier. La possibilité de lire ou d'écrire des données sur des fichiers distants (par exemple avec NFS[1]) permet de plus l'échange de données d'un ordinateur à un autre sous réserve que ces deux machines utilisent la même représentation de données. Le système de fichier Unix est donc bien adapté aux environnements séquentiels homogènes dans lesquels toutes les machines utilisent le même format de données

Nous proposons un environnement comparable mais adapté aux machines parallèles hétérogènes et au modèle de programmation à parallélisme de données. Notre environnement permet notamment la construction de périphériques virtuels distribués. Chaque périphérique

1. Pour Network File System

est défini par une grille de nœuds d'entrées/sorties semblable aux templates HPF et par une fonction de distribution (*Block* ou *Cyclic*) pour distribuer les fichiers sur ces nœuds. Chaque périphérique est associé à un répertoire. De cette façon la représentation d'un fichier ne dépend que du répertoire dans lequel il est placé. Tous les fichiers d'un même répertoire sont placés sur la même machine avec la même distribution. L'utilisateur peut dynamiquement migrer ces fichiers d'une machine parallèle à une autre à l'aide d'une simple commande shell. Le système se charge alors d'effectuer les redistributions et les changements de formats de données nécessaires.

Dans une première partie, nous définirons un modèle de fichier à objets structurés partitionnés. Nous introduirons ensuite le système de fichiers. Celui-ci intègre la notion de fichier à objets partitionnés et supporte les périphériques distribués. Enfin nous nous intéresserons à la bibliothèque d'entrées/sorties parallèles. Cette bibliothèque est conçue pour le modèle de programmation SPMD. Nous montrerons en particulier comment l'association de la distribution aux répertoires nous a permis de développer une interface à distribution explicite à l'exécution. Grâce à cette interface le programmeur fait des entrées/sorties dans un fichier logique. C'est l'utilisateur qui choisit à l'exécution la distribution qu'il souhaite. Il lui suffit pour cela de choisir le répertoire approprié ou d'en créer un avec une distribution adaptée à ses besoins. Cette bibliothèque sera illustrée par des exemples.

1 Le modèle de fichier

Les tableaux multi-dimensionnels sont les objets de bases du calcul scientifique et sont utilisés de façon intensives dans les «Grand Challenge Applications» [7]. En règle générale, ces objets sont persistants. Les données créées par une application sont sauvegardées pour être ensuite modifiées ou utilisées par d'autre programmes. De façon habituelle le programmeur utilise le modèle de fichier Unix. Il doit alors linéariser ses objets. Tous les calculs de distributions sont à sa charge. Le coût de développement est important et le code produit n'est pas portable. Pour éviter ces problèmes, des modèles de fichiers adaptés ont été développés. Parmi ces modèles, deux sont particulièrement bien adaptés au parallélisme de données. Il s'agit :

1. du modèle à fichier partitionné

2. du modèle à objets partitionnés

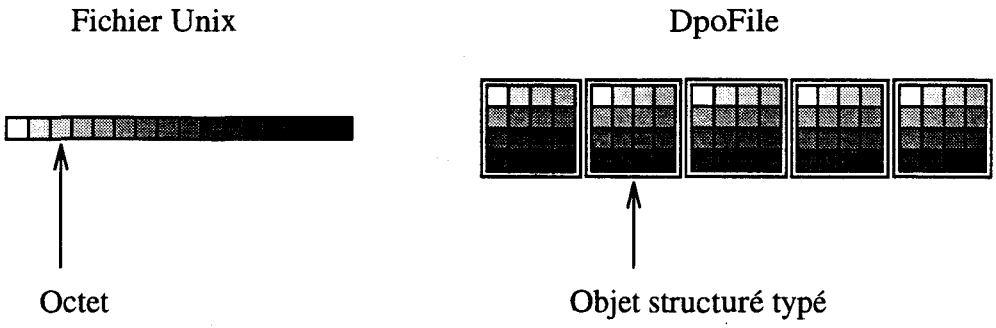


FIG. 4.1 - *Un DpoFile est une suite d'objets structurés*

Le premier modèle n'est pas suffisamment général pour être utilisé dans les environnements hétérogènes car il est basé sur la notion de machine virtuelle d'instanciation. Celle-ci n'est utilisée que par un sous-ensemble des langages à parallélisme de données. Le second modèle présente trois intérêts majeurs :

1. Il est «assez proche» du modèle de fichier Unix. Il conserve notamment un référentiel unique et un pointeur de fichier scalaire. Ce modèle ne nécessite pas de nouvelles habitudes de programmation.

2. Il est bien adapté au parallélisme de données. Ce sont les données qui sont distribuées pas l'espace de travail.

3. Il peut être mis en œuvre efficacement, ce qui est toujours appréciable.

Le modèle à objets partitionnés supporte la plupart des modèles à parallélisme de données (machines non virtuelles, machines virtuelles d'instanciation, machines virtuelles d'alignement...), et permet de passer facilement d'un environnement parallèle à un environnement séquentiel, il est donc bien adapté aux environnements hétérogènes. Comme les tableaux sont les objets de base du calcul scientifique, nous allons dans les prochains paragraphes définir un modèle de fichier à tableaux partitionnés.

1.1 Représentation du fichier structuré

Pour construire notre modèle à objets partitionnés nous avons simplement remplacé les caractères des fichiers Unix par des tableaux multi-dimensionnels que nous appelons **DPO** (pour Data Parallel Object).

Un **DpoFile** est une suite linéaire de DPO (fig. 4.1). Tous les DPO d'un même DpoFile ont la même géométrie[2] et le même type. Cela simplifie le modèle et autorise une mise en œuvre efficace.

1.2 Partionnement des données

Le partitionnement des DPO sur les processeurs de calcul, est défini à l'aide d'une fonction de distribution HPF. Toutes les distributions usuelles ($Collapsed$, $Block(K)$, $Cyclic(K)$) sont reconnues. Les propriétés de ces distributions nous assurent que les domaines accessibles par chacun des processeurs sont disjoints[3]. Les accès sont donc déterministes. La fonction de distribution détermine la visibilité d'un processeur de calcul sur le DpoFile. Seules les données distribuées sur le processeur p sont accessibles par celui-ci. La fonction de distribution est définie lors de l'ouverture du DpoFile et reste inchangée jusqu'à sa fermeture.

Il est important de noter que cette fonction de distribution est fixée à l'ouverture et non pas à la création du DpoFile. Cette distribution est celle qui est utilisée par l'application sur les nœuds de calculs, elle est complètement indépendante de celle utilisée par le système de fichiers. Il est de ce fait possible d'ouvrir successivement un même DpoFile avec des fonctions de distributions distinctes. Cette propriété est particulièrement intéressante pour les environnements hétérogènes: deux tâches à parallélisme de données peuvent s'échanger des DPO sans avoir à connaître leurs distributions respectives ni passer par une distribution canonique.

1.3 Accès aux données

Nous venons de définir les ensembles de données accessibles par chacun des processeurs, nous allons maintenant nous intéresser à la sémantique des fonctions d'accès.

Un DpoFile est une simple suite de DPO. Tous les DPO d'un DpoFile sont accessibles par l'ensemble des processeurs. L'espace d'adressage est unique. L'accès aux objets est réalisé par l'intermédiaire d'un pointeur scalaire qui spécifie le déplacement par rapport au début du fichier. Sous Unix ce déplacement est exprimé en octet. Dans notre système l'objet élémentaire est le DPO. Tous les accès sont atomiques. Un programme ne peut accéder à une «tranche» de DPO ni recouvrir une partie d'un DPO par un autre. Un DPO ne peut pas être partiellement défini. Le déplacement est donc exprimé en DPO.

2. même nombre de dimension et taille identique sur chaque dimension.
3. C'est bien un partitionnement au sens mathématique du terme!

Nous montrerons dans les prochaines sections que ce modèle répond effectivement à un besoin et qu'il peut être mis en œuvre efficacement.

2 Système de fichiers à objets partitionnés

Dans les sections précédentes nous avons défini un modèle de fichier à objets partitionnés. Nous allons maintenant construire un système de fichiers qui supporte ce modèle.

2.1 Distribution des données

Un système de fichiers parallèle doit avant tout être performant (c'est sa raison d'être). De nombreuses études montrent que le coût des entrées/sorties est fortement lié à la taille des données accédées [18, 13]. Pour augmenter la taille de ces accès il faut «rapprocher» l'organisation des données utilisée au niveau des fichiers de celle qui est employée pour effectuer les calculs [21].

Pour obtenir des performances satisfaisantes nous avons réutilisé les techniques qui ont été développées pour le calcul parallèle. Les périphériques parallèles virtuels sont construits à partir d'un ensemble de périphériques séquentiels, et les fonctions de distributions *Bloc* et *Cyclic*, sont celles qui ont été définies dans le langage HPF. L'utilisation des mêmes fonctions de distributions en mémoire et sur les disques, facilite le travail du système. En particulier, lorsque la distribution des données sur les disques est identique à celle utilisée en mémoire, tous les accès sont locaux!

2.2 Système de fichiers et périphériques virtuels

La plupart des «devices» parallèles physiques conservent la notion de système de fichier mais ne gèrent que des flots séquentiels. À l'inverse, les bibliothèques d'entrées/sorties parallèles telles que VIP-FS [27], ou PANDA [45], supportent les fonctions de distributions mais n'offrent pas un niveau d'abstraction satisfaisant. Ces bibliothèques associent la distribution des données aux fichiers et ne fournissent pas le concept de périphérique virtuel.

Le système de fichiers Unix fournit au programmeur une abstraction des périphériques physiques. Cette abstraction est un concept fondamental pour assurer la réutilisabilité des programmes. Un système de fichiers parallèle doit fournir un niveau d'abstraction équivalent.

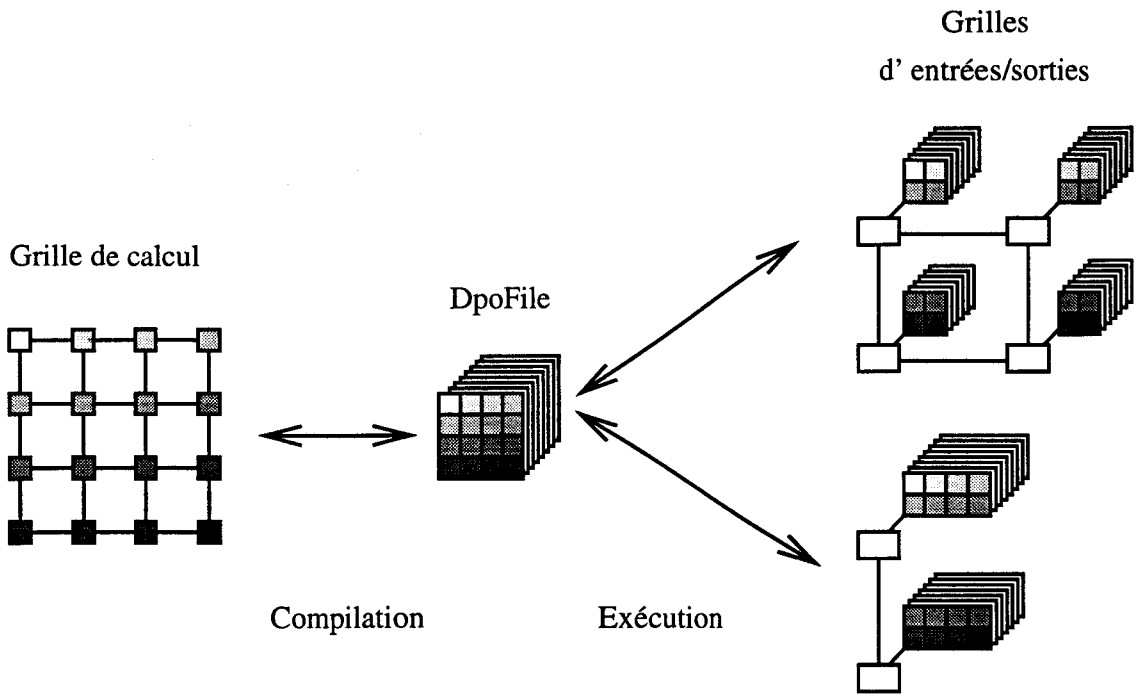


FIG. 4.2 - *Système de fichier multi-dimensionnel*

2.3 Mise en œuvre du système de fichiers

Pour adapter le système de fichiers Unix au modèle de programmation à parallélisme de données nous allons étendre les notions de «device» et celle de fichier de façon à prendre en compte la structure des objets parallèles et permettre le choix d'une fonction de distribution. Nous allons ainsi concilier

- **L'efficacité :**
  - Utilisation en parallèle de plusieurs périphériques séquentiels.
  - Choix à l'exécution d'une fonction de distribution adaptée aux besoins.
- **et la portabilité :**
  - On offre la notion de périphérique virtuel.
  - Le programmeur accède à un fichier logique (fig. 4.2). Les codes sont réutilisables.

Le système de fichiers distribués introduit trois nouveaux objets :

- **Les DpoDevices** sont des périphériques d'entrée/sorties virtuels. Chaque DpoDevice est défini par une grille de nœuds d'entrées/sorties à laquelle est associée une fonction de distribution ainsi qu'un format de donnée. À chaque nœud d'entrées/sorties est associé un répertoire dans lequel seront placées les données locales. Ce répertoire n'a pas à être visible par les nœuds de calculs qui accèdent au DpoDevice. Le système se charge de projeter les données sur les périphériques séquentiels qui constituent le DpoDevice.

  Traditionnellement, les périphériques sont montés au niveau d'un répertoire par un «super utilisateur». Les fichiers placés dans les répertoires créés sous un point de montage sont projetés sur le device qui a été monté (fig. 4.3). Cette approche est mal adaptée aux environnements distribués :

  - Seul l'utilisateur est à même de choisir la distribution des données qui convient à son application. Comme il existe une infinité de distributions possibles, chaque utilisateur doit avoir la possibilité de créer sa topologie d'entrées/sorties sur laquelle il pourra projeter ses données avec la distribution souhaitée.
  - Le choix d'un DpoDevice a un énorme impact sur les performances. La bande passante d'une grille de 128 disques reliée à un cross-bar FDDI est très différente de celle obtenue avec un montage NFS. Il est important de fixer pour chaque répertoire le «device» que l'on souhaite utiliser.

  Les DpoDevices ne seront pas «montés» mais simplement associés à certains répertoires. L'association d'un répertoire à un DpoDevice ne fixe que la distribution des DPO du répertoire associé. L'association est une opération locale qui n'est pas héritée par les répertoires fils (fig. 4.4).

- **Les DpoFiles** sont des fichiers dans lesquels sont placés des DPO homogènes (mêmes géométries et mêmes objets élémentaires). Un DpoFile n'est plus une simple suite d'octets, c'est une suite d'objets structurés typés. La structure des objets est utilisée pour distribuer les données sur les topologies d'entrées/sorties. Le typage des fichiers est particulièrement utile dans les environnements hétérogènes. Grâce à lui le système va pouvoir passer d'une représentation de données à une autre.

- **Les répertoires** sont les répertoires Unix «classiques», dans lesquels sont placés tous les objets Unix usuels (répertoires, fichiers, liens, ...) ainsi que des DpoFiles. Un répertoire peut être associé à un DpoDevice. Dans ce cas les DPO qui se trouvent dans

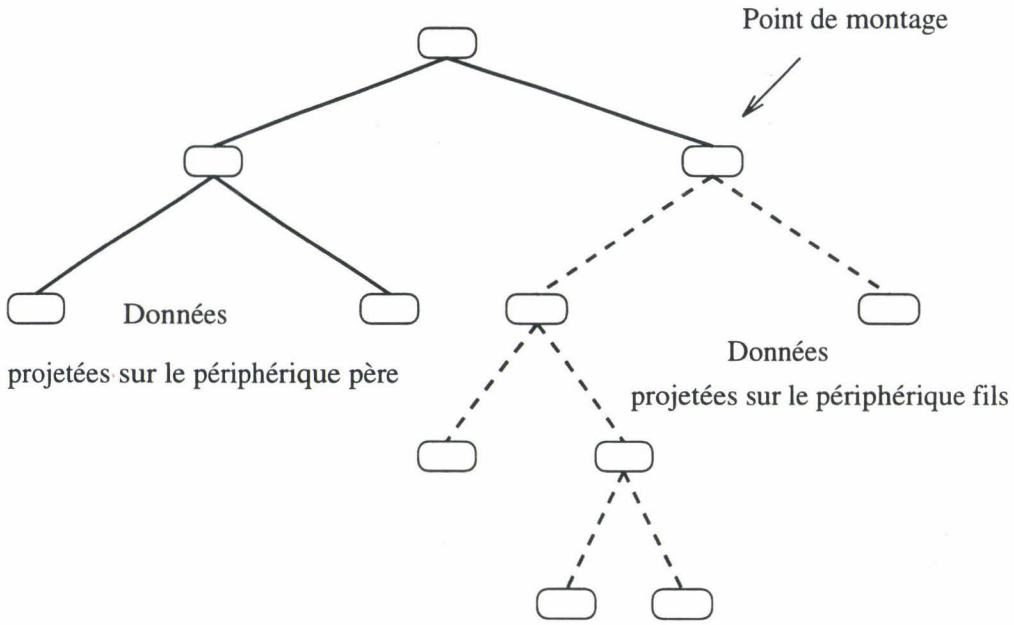


FIG. 4.3 - *Montage Unix*

le répertoire sont distribués sur les nœuds du DpoDevice. Ces répertoires spéciaux sont appelés **DpoBox**.

Nous avons ajouté aux fichiers, aux répertoires et aux devices des attributs pour prendre en compte la structure des objets parallèles. Ces trois «briques de base» du système de fichiers possèdent ainsi une «dimension parallèle».

3 Entrées/Sorties parallèles et redistributions dynamiques

Dans la section précédente nous avons défini un système de fichiers distribués, nous allons maintenant montrer que ce système s'utilise aussi simplement que le système de fichiers traditionnel.

Le figure (4.5) représente le répertoire de travail d'un utilisateur. Celui-ci a créé deux DpoBox de nom `Grille_1D` et `Grille_2D`[4], ainsi qu'un répertoire Unix. La DpoBox `Grille_1D` est associée à une grille uni-dimensionelle de nœuds d'entrées/sorties. Le DpoFile de nom `vecteur` est distribué par *Bloc* sur ces nœuds conformément à la fonction de distribution qui a été spécifiée lors de la création de la DpoBox. La DpoBox `Grille_2D` est associée à une grille

4. La création d'une DpoBox est réalisée à l'aide de la commande `mkdpobox` présentée dans l'annexe B

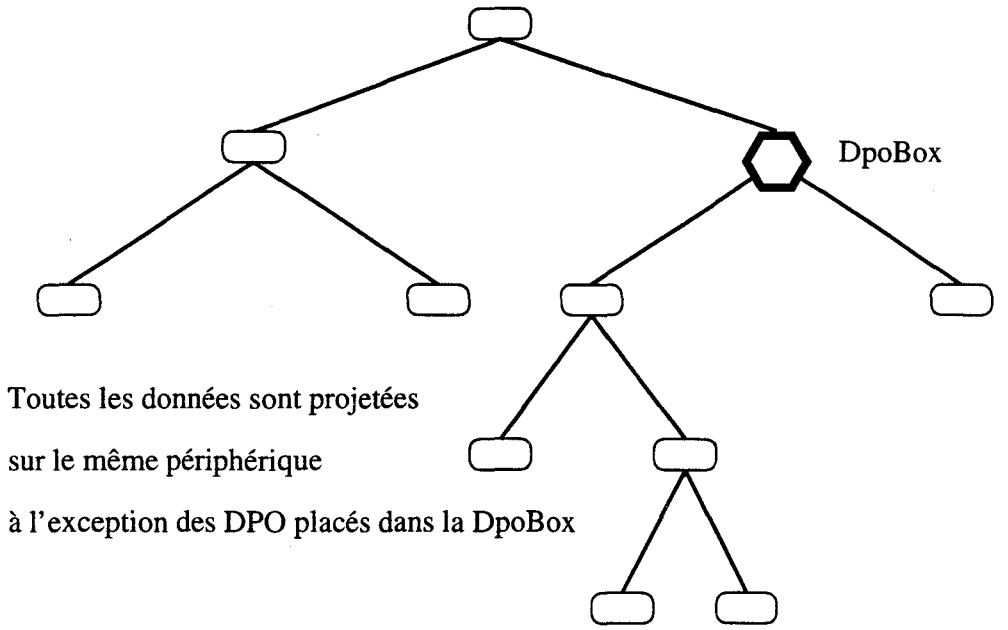


FIG. 4.4 - *Association d'un répertoire à une DpoBox*

2*2 sur laquelle le DpoFile de nom **Matrice** est distribué $Cyclic(1), Bloc$. Enfin le DpoFile de nom **Matrice2** est placé dans un simple répertoire Unix. Ce fichier n'est donc pas distribué.

Les trois DpoFiles ne sont pas sur la même machine physique et n'ont pas le même format de données (fig. 4.6). **Vecteur** est distribué sur des stations de travail **Sun**, **Matrice** est distribué sur une ferme de processeurs **Alpha** et **Matrice2** est placé sur un simple **Sparc**. Ces objets apparaissent tous au niveau du système de fichiers. L'environnement traditionnel est préservé. Les DpoFiles sont des fichiers et les DpoBox des répertoires. Toutes les commandes de base telles que **cd, cp, mv, rm, etc.** sont conservées[5]. Elles fournissent les services habituels : **cd** change le répertoire courant, **cp** copie les objets... L'utilisateur peut ainsi déplacer le DpoFile de la DpoBox **Grille_2D** à la DpoBox **Grille_1D** :

```
mv matrice ../Grille_1D
```

ou de la DpoBox **Grille_2D** au répertoire **Répertoire_Unix** :

```
mv matrice ../Grille_1D
```

Les commandes s'utilisent comme dans l'environnement séquentiel traditionnel. D'un point

5. Ces commandes sont détaillées dans l'annexe B

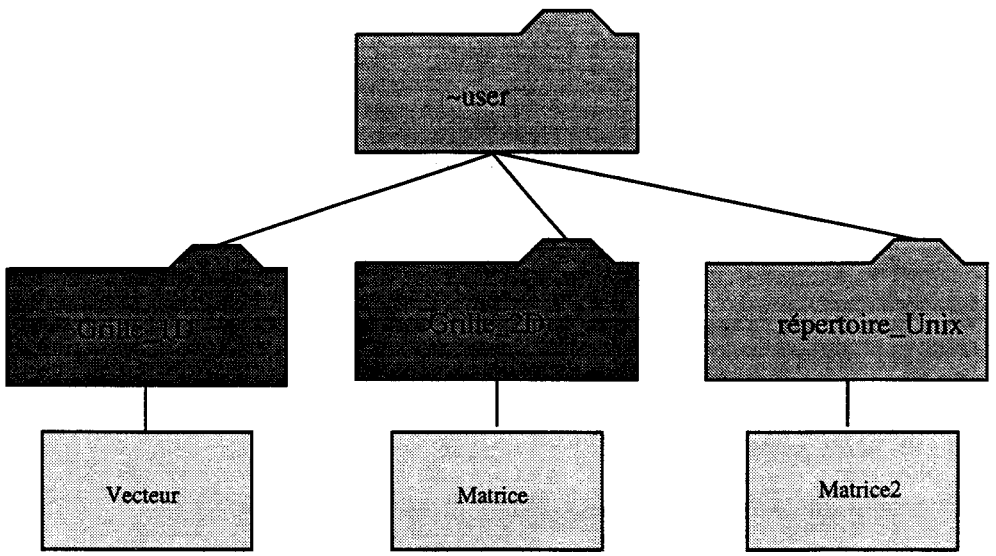


FIG. 4.5 - *Arborescence d'un utilisateur*

de vue logique, rien n'a changé mais les attributs qui ont été ajoutés au niveau de la DpoBox sont utilisés par les commandes pour effectuer des traitements supplémentaires. La commande `mv` migre les DpoFiles d'une machine parallèle à une autre en changeant (si nécessaire) la représentation des données. Elle redistribue les données d'une topologie source à une topologie cible en fonction des fonctions de distributions associées aux répertoires. L'organisation des données sur les topologies d'entrées/sorties est gérée dynamiquement par le système.

4 Bibliothèque d'entrées/sorties parallèles

La différence majeure entre le modèle de fichiers Unix et le modèle à objets structurés est liée à la notion de distribution. Les objets structurés sont distribués sur les nœuds de calculs mais aussi sur les nœuds d'entrées/sorties. Ces deux niveaux de distributions compliquent énormément le modèle à objets partagés et limitent son intérêt. Les accès sont relatifs à une topologie et à une distribution d'entrées/sorties.

Le choix d'une distribution est fonction des traitements que l'on souhaite effectuer. On peut par exemple projeter une image sur un simple nœud d'entrée/sortie pour la visualiser avec un utilitaire standard ou au contraire la distribuer sur une grille pour effectuer un traitement parallèle. En fixant la distribution des objets sur les nœuds d'entrées/sorties le programmeur restreint le spectre d'application de ses programmes. Il est extrêmement difficile dans ces conditions de créer des programmes réutilisables.

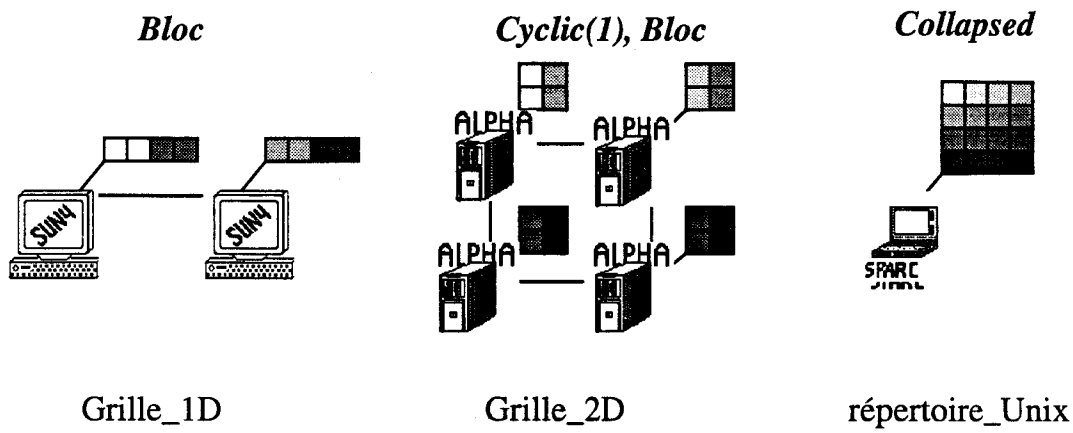


FIG. 4.6 - *Distribution des données sur les nœuds d'entrées/sorties*

Pour isoler le programmeur de la représentation des données sur les nœuds d'entrées/sorties nous avons introduit la notion de périphérique virtuel. Dans notre système de fichiers la distribution des données n'est pas une propriété des fichiers parallèles. Elle ne dépend que du répertoire utilisé. Cela nous a permis de développer une interface à distribution explicite à l'exécution.

4.1 Fonctions d'accès

La bibliothèque que nous avons développée est conçue pour être utilisée sur des machines à mémoires distribuées avec le modèle de programmation SPMD. Elle est conçue au dessus des fonctions d'entrées/sorties standards et de la bibliothèque PVM. Les principales fonctions d'accès sont présentées (fig. 4.7). La fonction `DPopen` est la plus complexe. Elle accepte six paramètres:

1. le chemin du DpoFile accédé: `path`
2. le mode d'accès: `mode`
3. la grille de processeurs de calcul ainsi que la fonction de distribution de données associée à cette grille : `lgrid`.
4. la géométrie des DPO : `shape`
5. le type des objets élémentaires qui composent le DPO : `type`
6. et enfin une variables pour mémoriser les problèmes : `error`.

La fonction `DPopen` retourne un descripteur de DpoFile sur chacun des processeurs. Celui-ci sera utilisé lors de chaque fonction d'accès. Ce descripteur est nul lorsque les paramètres sont incorrects. La fonction `DPread` accepte quatre paramètres:

- un descripteur de DpoFile: `stream`
- un buffer dans lequel seront placées les données locales: `local_data`. Elles sont rangées dans l'ordre canonique du langage C. Ce buffer local est alloué par l'utilisateur. Cela peut être réalisé à l'aide de la fonction `DPmalloc` qui accepte deux paramètres: le descripteur de fichier et le nombre de DPO à allouer. L'utilisateur peut également utiliser la fonction `DP_local_nb_item` pour obtenir le nombre de données locales et utiliser la commande `malloc` standard.
- un entier qui représente le nombre de DPO à lire: `nb`.
- Et enfin la variable `error` pour mémoriser les erreurs.

Cette fonction retourne le nombre de DPO effectivement accédés. La fonction `DPwrite` accepte les mêmes paramètres et effectue le travail inverse. Enfin la fonction `DPclose` permet de fermer un fichier. Elle libère les ressources qui étaient utilisées. Ces fonctions d'accès parallèles sont des «extensions naturelles» des fonctions d'entrées/sorties standards (présentées figure 4.8). Elles sont exécutées de manière asynchrone sur tous les processeurs. Une description plus détaillée est fournie en annexe. Nous allons maintenant illustrer ces fonctions sur un exemple.

4.2 Illustration

La plupart des utilitaires Unix travaillent sur des flots d'entrées/sorties (grep, cat,...). Il est possible de construire des utilitaires parallèles qui fonctionnent sur le même schéma. On peut par exemple concevoir des outils pour inverser des matrices, faire des transformées de Fourier, ...à partir d'un DpoFile. Ces utilitaires travaillent tous sur un flot de DPO et ont tous la même «architecture».

Fonction	Paramètres	Description
DPopen	CST char *path, CST char *mode, CST LOCAL_GRIDS *lgrid, SHAPES *shape, TYPES *type, ERRNOS *error	Ouvre le DpoFile et retourne un descripteur de DpoFile
DPclose	DPSTREAMS *stream, ERRNOS *error	Ferme le DpoFile
DPwrite	DPSTREAMS *stream, CST char *local_data, CST NATS nb, ERRNOS *error	Écriture de nb DPO dans le DpoFile
DPread	DPSTREAMS *stream, char *local_data, CST NATS nb, ERRNOS *error	Lecture de nb DPO depuis le DpoFile
DPseek	DPSTREAMS *stream, CST int offset, CST int whence, ERRNOS *error	Déplace le pointeur de fichier

FIG. 4.7 - *Fonctions d'accès parallèles*

Fonction	Paramètres	Description
open	char *path, int flag , mode_t mode	Ouvre le Fichier et retourne un descripteur de fichier
close	int filedes	Ferme le fichier
write	int filedes , const void *buffer, size_t nbytes	Écriture de nb octets dans le fichier
read	int filedes , const void *buffer, size_t nbytes	Lecture de nb octets depuis le fichier
lseek	int filedes, off_t offset, int whence	Déplace le pointeur

FIG. 4.8 - *Fonctions d'accès Unix*

```
Do_the_job(int argc, char *argv[]){
  DPSTREAMS *f1,*f2;

  /*          File opening and error checking                  */

  f1=DPopen(argv[1],"r",&local_grid,&array_shape,&type,&error_r);
  f2=DPopen(argv[2],"w",&local_grid,&array_shape,&type,&error_w);

  if (f1 == NULL) {DPperror(&error_r, "Source DPfile"); exit(2);}
  if (f2 == NULL) {DPperror(&error_w, "Target DPfile"); exit(2);}
  buffer=DPmalloc(f1, 1);  /*         memory allocation        */

  /*    arrays are treated each in turn until the last one     */

  while (DPread(f1,buffer,1) == 1){
    Do_the_job(buffer);
    DPwrite(f2,buffer,1);
  }
  free(buffer);  DPclose(f1);  DPclose(f2);
}
```

FIG. 4.9 - *Exemple d'utilisation*

La figure (4.9) détaille les opérations d'accès à effectuer pour réaliser de tels utilitaires. Ce programme est de type SPMD. Il est exécuté sur tous les processeurs. Le DpoFile «f1» est utilisé en entrée pour accéder aux DPO sur lesquels sont effectués les calculs. Le DpoFile «f2» est utilisé pour stocker les résultats.

La distribution des DPO au niveau du système de fichiers n'est pas explicitée. Cette distribution ne dépend que du répertoire dans lequel sont placés les DpoFiles. Dans l'exemple (4.9), les noms des DpoFiles sont des paramètres du programme. Cela permet à l'utilisateur de spécifier à l'exécution l'organisation des données sur les nœuds d'entrées/sorties.

La bibliothèque d'entrée/sortie fournit une interface de haut niveau qui isole le programmeur des détails d'implémentations et permet de construire des codes réutilisables. Les accès aux données sont indépendants de la topologie, de la distribution et de la représentation binaire qui est utilisée. Ces informations sont fournies implicitement à l'aide du nom du DpoFile.

```
fonction(){
  ....

  f1=DPopen("danger","w",grid_bloc,&shape,&type,&error);
  f2=DPopen("danger","r",grid_cyclic,&shape,&type,&error);

  DPwrite(f1,buffer,1,&error);
  .....
  DPread(f2,buffer,1,&error)

  .....
}
```

FIG. 4.10 - *La sémantique du modèle n'est pas garantie par la bibliothèque*

4.3 Synchronisme

Les synchronisations sont extrêmement coûteuses sur les machines MIMD. Il est essentiel, pour ne pas dégrader l'efficacité des programmes, d'éviter toute synchronisation inutile.

Pour assurer la sémantique du modèle de programmation à parallélisme de données il «suffit» de précéder chaque opération de communication par une synchronisation, encore faut-il être capable de reconnaître les instructions qui vont induire ces communications.

Avec notre modèle de fichiers parallèles, la visibilité des processeurs est définie lors de l'ouverture du DpoFile et reste inchangée jusqu'à sa fermeture. Par ailleurs, les ensembles de données accessibles par les processeurs sont disjoints. Pour faire communiquer deux processeurs par l'intermédiaire d'un DpoFile, il faut ouvrir successivement un même fichier en lecture et en écriture avec des distributions de calculs distinctes.

Les figures (4.10) et (4.11) mettent en évidence ces communications. Dans l'exemple (4.10), le DpoFile `danger` est ouvert à la fois en lecture avec une distribution par *Block* et en écriture avec une distribution *Cyclic*. Un processeur peut ainsi accéder avec le pointeur `f2` à des données qui n'étaient pas accessibles avec le pointeur `f1`. Pour assurer la sémantique du modèle de programmation à parallélisme de données il faut que tous les processeurs aient exécuté l'instruction `DPwrite` avant que l'un d'entre eux n'effectue l'instruction `DPread`. Il faut donc mettre en place une synchronisation avant l'instruction de lecture[6].

6. Cette synchronisation existe peut-être déjà dans le code placé entre ces deux instructions (par exemple à cause d'une communication)

```
fonction(){
  ....

  f1=DPopen("danger","w",grid_bloc,&shape,&type,&error);
  DPwrite(f1,buffer,1,&error);
  .....
  DPclose(f1);
  f1=DPopen("danger","r",grid_cyclic,&shape,&type,&error);
  DPread(f2,buffer,1,&error)
  .....
}
```

FIG. 4.11 - *Les fonctions d'ouvertures et de fermetures sont elles aussi non bloquantes*

L'exemple (fig. 4.11) met en évidence le même problème avec un unique pointeur de fichier. Là encore il faut introduire une synchronisation entre la dernière opération d'écriture et la première opération de lecture. Pour décider de la nécessité d'une synchronisation il faut connaître précisément le contexte d'exécution. Une bibliothèque ne dispose pas des informations nécessaires à ce choix. Il y a alors deux politiques possibles :

1. On précède chaque opération d'accès par une barrière de synchronisation.
2. On laisse à l'utilisateur la charge d'introduire les synchronisations nécessaires.

La première solution est coûteuse car les synchronisations sont souvent inutiles. Les fonctions d'accès de notre bibliothèque n'effectuent aucune synchronisation. C'est au compilateur ou au programmeur qu'il appartient d'assurer la sémantique du modèle de programmation.

5 Mise en œuvre efficace

Dans les sections précédentes nous avons défini le système de fichiers et la bibliothèque de fonctions d'accès parallèle que nous avons conçus. Cet environnement s'utilise aussi simplement que l'environnement séquentiel standard et permet la portabilité. Nous allons montrer que cette facilité d'utilisation n'a pas été obtenue au détriment des performances.

Il est difficile de mettre en place un système d'entrées/sorties performant en raison de la complexité de l'environnement d'exécution. Chaque accès aux fichiers fait interagir les nœuds d'entrée/sortie avec le réseau de communication et les nœuds de calculs (fig. 4.12).

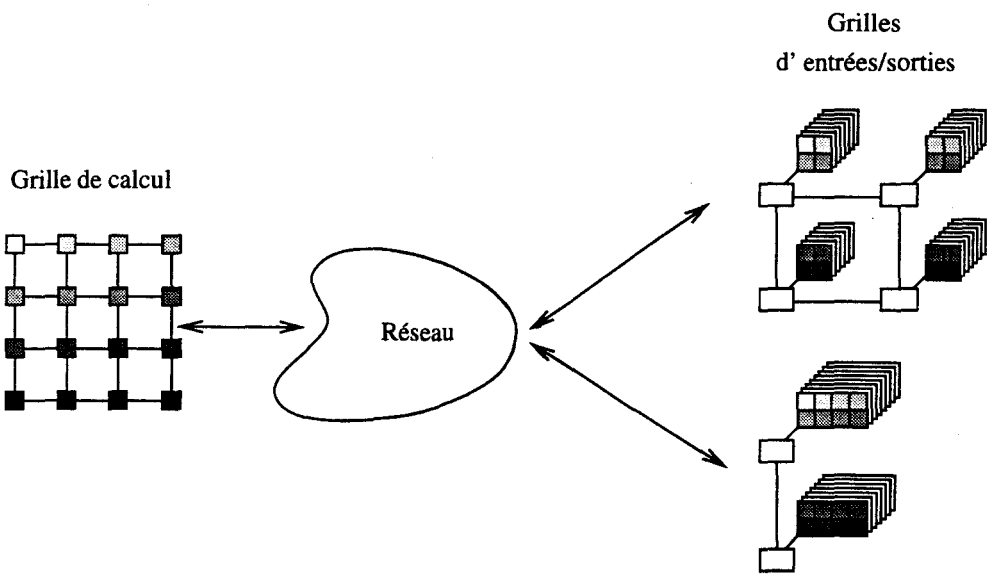


FIG. 4.12 - *Architecture du système de fichier*

Pour obtenir des performances élevées nous avons pris en compte l'ensemble des opérations à effectuer pour réaliser une entrée/sortie parallèle de façon à éviter les goulots d'étranglement.

5.1 Nombre d'accès disques minimal

Les études montrent que les performances des entrées/sorties sont en grande partie liées au nombre d'accès disque effectués. Plus celui-ci est faible et meilleur sont les performances [18, 49, 48, 46]. L'organisation des données en mémoire sur les machines parallèles est généralement très différente de celle qui est utilisée par le système de fichier. De ce fait les processeurs effectuent de très nombreux accès disques pour lire ou écrire leurs données sur les disques. Ces accès sont très pénalisants. Il faut déplacer la tête de lecture et mettre à jour les différents niveaux de cache qui sont très souvent invalidés.

Pour limiter au maximum le coût des accès disques, Juan Miguel Rosario, Rajesh Bordawekar et Alok Choudhary proposent la «two-phase access strategy» [20] qui consiste à effectuer les accès en deux étapes successives. Les données sont tout d'abord lues conformément à leurs distribution sur les disques. Cela ne nécessite qu'un unique accès sur chacun des nœuds d'entrée/sortie. Une seconde étape est ensuite nécessaire pour redistribuer les données sur les processeurs de calcul.

L'interface de haut niveau a facilité la mise en œuvre de cette stratégie. Les accès portent

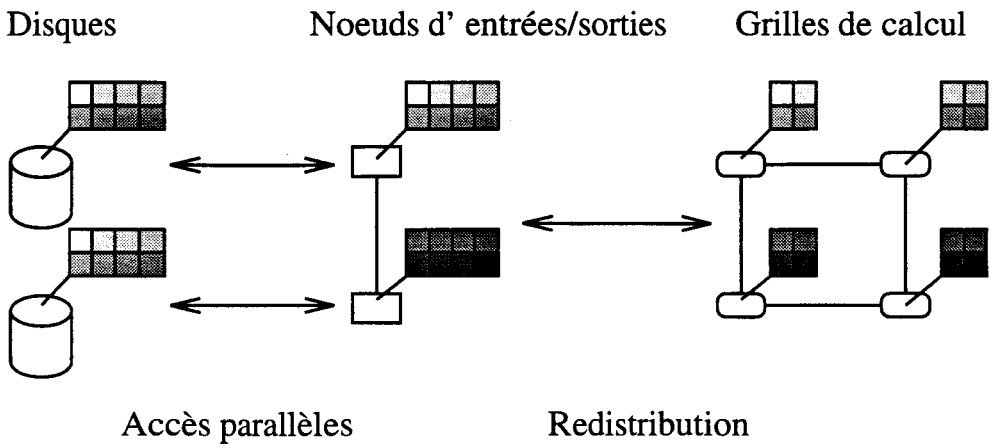


FIG. 4.13 - *Stratégie d'accès en deux phases successives*

sur des objets atomiques structurés. Le système a donc une vision globale des mouvements de données. Il suffit pour définir un accès, de spécifier les distributions source et destination ainsi que le déplacement en nombre de DPO par rapport au début de fichier. Lors de la première étape, chaque nœud d'entrée/sortie effectue un accès disque pour recopier les données en mémoire. Une seconde étape est ensuite nécessaire pour redistribuer les données sur les nœuds de calculs (fig. 4.13).

5.2 Bonne utilisation du réseau de communication

Pour limiter au maximum le coût des accès réseau il suffit de ne pas l'utiliser ! Grâce aux périphériques virtuels, l'utilisateur peut construire une topologie d'entrées/sorties identique à la topologie de calculs et utiliser les mêmes fonctions de distributions en mémoire et sur les disques. Chaque nœud effectue alors des calculs et des entrées/sorties. Tous les accès sont locaux.

Cependant une distribution adaptée à une phase de calcul ne l'est pas forcement pour la suivante. Il est souvent nécessaire de redistribuer les données entre deux traitements successifs. Ces redistributions sont effectuées à l'aide des algorithmes de redistribution présentés dans le chapitre précédent. Ces algorithmes minimisent le volume et le nombre de messages. Ils recouvrent les calculs avec les communications et minimisent les déséquilibres de charges.

5.3 Préchargement des données

Le préchargement des données est très fréquemment utilisé pour accélérer l'accès aux données. Cette optimisation est mise en place au niveau matériel (avec les caches disques) ; elle est également utilisée par le système et par certaines applications.

Comme nous utilisons la «two phase access strategie», la représentation des DPO sur les disques est identique à celle qui est utilisée en mémoire sur les nœuds d'entrées/sorties (fig. 4.13). Les données accédées sont toujours contiguës. Le système tire donc au maximum profit des multiples couches chargées d'anticiper les accès.

Comme le partitionnement des données est connu dès l'ouverture du fichier et qu'il n'évolue pas jusqu'à sa fermeture, le système connaît exactement les ensembles de données qui seront projetés sur chaque processeur. Les propriétés des distributions utilisées nous assure de plus que ces ensembles de données sont disjoints. Il est donc possible de mettre en place un niveau d'anticipation supplémentaire sans avoir à gérer des problèmes de cohérences de cache. Le préchargement des DPO permet de s'affranchir de la latence du réseaux et des disques.

6 Illustration

Nous allons maintenant illustrer les qualités de notre système de fichiers sur un exemple complet. Il s'agit de la résolution d'un système linéaire $AX = B$. Cette résolution est réalisée à l'aide de la bibliothèque ScaLAPACK[7]. Les entrées/sorties ainsi que la création de la grille de calcul sont effectuées par notre environnement.

La bibliothèque ScaLAPAK [8] est une bibliothèque d'algèbre linéaire conçue pour les machines à mémoires distribuées de type MIMD. Avant d'appeler les fonctions de calculs, le programmeur doit préalablement distribuer ses données sur l'ensemble de ses processus. Aucun mécanisme n'est prévu pour lire et écrire les données sur lesquelles portent les traitements. La résolution des systèmes de type $AX = B$ est traité en exemple dans le «ScaLAPACK User's Guide». Pour mieux montrer l'intérêt de notre environnement cette exemple est reproduit dans l'annexe C.

7. Scalable Linear Algebra PACKage

```
void main(CST int argc, CST char **argv){

  process_enroll();
  if (get_parent_tid() == NoParent){  /* processus pere */
    ERRNOS          errors;
    MAPPINGS        mapping;
    HOST_INFOS      hosts;
    GRIDS           grid;
    NATS            i;

    if (argc != 5) usage("solve DpoBox_name A X B");
    if (DPdirinfo(argv[1], &mapping, &hosts, &errors) == Problem)
      DPperror(&errors, "DPdirinfo");
    else {
      if (grid_spawn(&mapping,hosts.physical,&grid,"solve",argv) != Ok){
        fprintf(stderr,"can' t start grid\n");
        exit(2);
      }
    }
  }
  else {                              /* processus fils */
    LOCAL_GRIDS     lgrid;

    grid_init(&lgrid);
    Do_the_job(argv, &lgrid);
  }
  process_exit();
}
```

FIG. 4.14 - *Programme principal*

6.1 Gestion des processus

La figure (4.14) présente le programme principal. Ce programme accepte quatre paramètres.

- le nom de la grille sur laquelle seront effectués les calculs ;
- les noms des DpoFiles dans lesquels sont stockées les matrices A et B ;
- le nom du DpoFile dans lequel sera stocké le résultat.

Ce programme est exécuté à la fois par le processus maître et par l'ensemble des processus esclaves. La fonction `get_parent_tid` retourne la constante `No_Parent` au processus maître. Celui-ci commence par vérifier le nombre de paramètres. Il obtient ensuite les propriétés de la grille de calcul (taille de la grille, fonction de distribution et noms des processeurs) à l'aide de la fonction `DPdir_info`. Il lance ensuite l'exécutable «solve» sur tous les nœuds de cette grille.

Les processus esclaves appellent la fonction `grid_init` pour obtenir leurs coordonnées puis exécutent la fonction `Do_the_job`. Tous les processus (le maître et les esclaves) terminent leur exécution en appelant la fonction `process_exit`.

Les fonctions de gestion de la grille de calculs ne sont pas indispensables pour effectuer des entrées/sorties parallèles. Nous les fournissons car elles simplifient l'utilisation de notre bibliothèque et incitent le programmeur à utiliser les mêmes topologies pour ses calculs et ses entrées/sorties. Dans le programme d'exemple fourni avec le guide d'utilisateur de ScaLAPACK, la gestion des processus est effectuée par les programmes `PDSCAEX` et `PDSCAEXINFO` cela nécessite un centaine de lignes de code pour un résultat analogue.

6.2 Les entrées/sorties parallèles

La figure (4.15) présente les opérations à effectuer pour réaliser les entrées/sorties parallèles. On retrouve l'intérêt principal de notre bibliothèque : le programmeur n'a pas à connaître la représentation des données dans le système de fichiers. Les fonctions d'accès se chargent de redistribuer les données conformément à la fonction de distribution choisie par le programmeur. L'allocation des buffers sur chaque processus est effectuée à l'aide de la fonction `DPmalloc`. Cette fonction prend en compte la taille du type élémentaire mais aussi la géométrie des DPO, la topologie de calcul et la fonction de distribution qui ont été choisies. La taille des buffers varie donc d'un processus à un autre. Les opérations d'accès sont effectuées à l'aide des fonctions `DPread` et `DPwrite` qui ont déjà été présentées.

Dans l'exemple ScaLAPACK, ces opérations d'entrées/sorties sont effectuées par les fonctions `PDLAREAD` et `PDLAWRITE`. Ces fonctions représentent chacune une centaine de lignes de code. Les accès sont séquentiels. Le programmeur doit acheminer lui même les données du nœud d'entrée/sortie aux nœuds de calculs ou des nœuds de calculs au nœud d'entrée/sortie.

6.3 Appel à ScaLAPACK

La figure (4.16) présente le travail de transposition mémoire à réaliser pour utiliser notre bibliothèque d'entrées/sorties avec le langage fortran. Ce travail est nécessaire car il n'est pas possible avec le système de fichiers actuel de choisir entre un rangement «column major» et «row major». Cette possibilité devrait être ajoutée au niveau des propriétés des DpoBox.

Toutes les informations relatives à la grille de calcul et les données sur lesquelles portent les traitements sont passées en paramètre à la fonction fortran `fortran_interface`. Le résultat est ensuite transposé afin de passer du rangement par colonne du langage fortran, au rangement par ligne du langage C.

La figure (4.17) représente la fonction `FORTRAN_INTERFACE`. Les fonctions `SETPVMTIDS`, `BLACS_GET`, `BLACS_GRIDMAP` et `BLACS_GRIDINFO` sont utilisées pour construire une grille de calcul au format ScaLAPACK à partir de la grille précédemment créée. La fonction `DESCINIT` est appelée deux fois. Elle définit un descripteur (taille du tableau global, et taille des blocs dans chaque dimension) pour le tableau A puis pour le tableau B. Lorsque ce travail préalable a été effectué, la fonction `PDGESV` est appelée. C'est cette fonction qui effectue la résolution du système linéaire.

7 Conclusion

Nous proposons d'intégrer les périphériques séquentiels qui existent sur les machines distribuées au sein d'une entitée logique appelée «DpoDevice». Un DpoDevice est défini par une topologie de nœuds d'entrées/sorties à laquelle est associée une fonction de distribution. Chaque utilisateur a la possibilité de définir ses propres DpoDevices. Ces périphériques parallèles virtuels sont plongés dans le système de fichiers de façon à offrir un niveau d'abstraction équivalent au système de fichier Unix. Les commandes Shell standards (mv, cp, rm...) sont surchargées pour supporter les fichiers parallèles et permettre la redistribution interactive d'une topologie d'entrées/sorties à une autre.

Pour accéder aux DpoDevice nous avons construit une bibliothèque d'entrées/sorties parallèles adaptée au modèle de programmation à parallélisme de données. Cette bibliothèque à objets structurés partitionnés dispose d'une interface à distribution explicite à l'exécution. Le programmeur n'a pas à connaître la représentation des données dans le système de fichiers. Cette représentation est choisie à l'exécution par l'utilisateur.

```
void Do_the_job(char *argv[], LOCAL_GRIDS *compute_grid){
  SHAPES shape_A, shape_B;
  ERRNOS err;
  DPSTREAMS *A,*B,*X;
  TYPES type;
  char *data_A, *data_B;

  A = Dpo_open(argv[3],"r",compute_grid,&shape_A,&type, &err);
  if (A == NULL) {DPperror(&error, "DPopen A"); return;}
  else {
    B = Dpo_open(argv[4],"r",compute_grid,&shape_B,&type, &err);
    if(B == NULL) {
      DPperror(&error, "DPopen B"); DPclose(A, &err); return;}
    else {
      X = Dpo_open(argv[5],"w",compute_grid,&shape_B,&type, &err);
      if(X==NULL){
        DPperror(&error,"DPopen X");
        DPclose(A, &err); DPclose(B, &err);return;
      }
    }
  }
  if (shape_A.size[0] != shape_A.size[1] || shape_A.nb_dim != 2)
    usage("Problem with A shape");
  if (shape_A.size[1] != shape_B.size[0] || shape_B.nb_dim > 2)
    usage("Problem with B shape");

  data_A=DPmalloc(A, 1);
  data_B=DPmalloc(B, 1);
  DPread(A, data_A, 1, &err); DPclose(A, &err);
  DPread(B, data_B, 1, &err); DPclose(B, &err);
  solve(compute_grid, shape_A, data_A, shape_B, data_B);
  DPwrite(X, data_B, 1, &err);
  DPclose(X, &err);
}
```

FIG. 4.15 - *Entrées/sorties parallèles*

```
void transpose_int(int *source, int *cible, int N0, int N1){
  int i,j,NB=N0*N1;

  for (i=0; i<NB; i++) cible[(i%N1)*N0+i/N1]=source[i];
}

void transpose_double(double *source, double *cible, int N0, int N1){
  int i,j,NB=N0*N1;

  for (i=0; i<NB; i++) cible[(i%N1)*N0+i/N1]=source[i];
}

void solve(LOCAL_GRIDS *compute_grid, SHAPES shape_A,
           char *data_A, SHAPES shape_B, char *data_B){

  int K[MAX_DIM], L_A[MAX_DIM], L_B[MAX_DIM];
  int tids[MAX_NODE], i;
  int P0=compute_grid.mapping.nb_node[0];
  int P1=compute_grid.mapping.nb_node[1];
  double *TA= (double *)data_A;
  double *TB= (double *)data_B;

  dim_bloc(&compute_grid.mapping, &shape_A, K, L_A);
  dim_bloc(&compute_grid.mapping, &shape_B, K, L_B);
  {
    double *A=(double *)malloc(sizeof(double)*L_A[0]*L_A[1]);
    double *B=(double *)malloc(sizeof(double)*L_B[0]*L_B[1]);

    transpose_int(compute_grid.tids,tids,P0,P1);
    transpose_double(TA, A, L_A[0], L_A[1]);
    transpose_double(TB, B, L_B[0], L_B[1]);

    fortran_interface_(tids, &P0, &P1,
                       &shape_A.size[0], &shape_B.size[1],
                       &K[0],&K[1], &L_A[0], &L_A[1],
                       &L_B[0], &L_B[1], A, B);

    transpose_double(B, TB, L_B[1], L_B[0]);
  }
}
```

FIG. 4.16 - *Passage du format C au format Fortran*

```
      SUBROUTINE FORTRAN_INTERFACE(TIDS, P0, P1, N, NRHS,
     $     K0 , K1, L0_A, L1_A, L0_B, L1_B,
     $     A, B)

      INTEGER P0, P1, TIDS(P0, P1), N , NRHS,
     $     K0 , K1, L0_A, L1_A, L0_B, L1_B

      DOUBLE PRECISION A(L0_A, L1_A), B(L0_B, L1_B)
      INTEGER D_LEN, NB_PROC, MY_LIG, MY_COL
      PARAMETER (D_LEN = 9)
      INTEGER communicateur, DESC_A(D_LEN), DESC_B(D_LEN),
     $     info, i, j
      DOUBLE PRECISION PIVOT(L0_A + K0)


      CALL SETPVMTIDS(P0*P1,TIDS)
      CALL BLACS_GET(-1, 0, communicateur)
      cALL BLACS_GRIDMAP(communicateur, TIDS, P0, P0, P1)
      CALL BLACS_GRIDINFO(communicateur, P0, P1,
     $     MY_LIG, MY_COL)
      CALL DESCINIT(DESC_A, N, N, K0, K1, 0, 0,
     $     communicateur, L0_A, info)
      CALL DESCINIT(DESC_B, N, NRHS, K0, K1, 0, 0,
     $     communicateur, L0_B, info)

      CALL PDGESV(N, NRHS, A, 1, 1, DESC_A, PIVOT, B, 1, 1,
     $     DESC_B, info)

      CALL BLACS_GRIDEXIT(communicateur)
      CALL BLACS_EXIT(1)

      END
```

FIG. 4.17 - *Appel de la fonction ScaLAPAK*

Conclusion

Résumé des travaux

Nous proposons un environnement complet qui intègre la notion d'objet structuré au niveau de la bibliothèque de fonctions d'accès mais aussi au niveau du système de fichiers et des périphériques d'entrées/sorties [51, 54]. Le concept d'objet parallèle est utilisé à la fois pour les opérations de calculs et pour les opérations d'accès. Il n'est pas plus difficile de migrer un objet parallèle d'une topologie d'entrées/sorties à une topologie de calculs que de passer d'une topologie de calculs à une autre. Ces deux opérations sont conceptuellement identiques. Le programmeur n'a plus à linéariser ses données pour les projeter dans un système de fichiers séquentiel.

La notion de périphérique virtuel a permis de développer un système hétérogène à la fois simple et performant. C'est notamment sur cette notion de périphérique virtuel que repose l'interface explicite à l'exécution. Avec cette interface le programmeur accède à un fichier logique. Il n'a pas à connaître l'organisation des données au niveau du système de fichiers. La distribution des données dans le système de fichiers est spécifiée à l'exécution par l'utilisateur. On concilie ainsi les performances (la distribution des données est bien adaptée aux besoins) et la portabilité (comme la distribution n'est pas fixée à la compilation, un même programme peut être utilisé dans différents contextes).

La notion de périphérique virtuel a également permis d'étendre les fonctionnalités des principales commandes Shell. Avec notre environnement l'utilisateur peut interactivement migrer ses fichiers d'une machine parallèle à une autre. Les redistributions et les changements de format nécessaires sont réalisés dynamiquement par le système. Un premier prototype a été réalisé. Il fonctionne sur processeurs ALPHA et sur PC. L'intérêt du système de fichiers et de la bibliothèque d'entrées/sorties parallèle ont été illustrés avec un exemple concret : la résolution d'un système linéaire avec la bibliothèque ScaLAPACK.

Pour tirer profit de toutes les ressources disponibles nous avons développé des algorithmes de redistribution dynamiques capables de recouvrir les calculs avec les communications [52, 53]. Plutôt que de construire un algorithme général qui effectue toutes les redistributions HPF, nous avons préféré développer un algorithme par type de redistribution. Cela nous a permis de simplifier le modèle général et de bénéficier des «bonnes propriétés» spécifiques à chaque redistribution. Dans chacun des cas étudiés nous avons montré qu'il est possible d'énumérer les couples de processeurs qui interagissent et de construire les messages à échanger à l'aide d'un ensemble de sections régulières. Les performances obtenues sur une ferme de processeurs ALPHA montrent l'intérêt mais aussi les limites de ces algorithmes. Pour profiter au maximum du recouvrement des calculs avec les communications il faut calculer un ordonnancement qui minimise les déséquilibres de charges sur la machine cible. À l'aide des équations de redistributions spécifiques nous avons réussi à répartir la charge sur l'ensemble des processeurs cibles. Les résultats obtenus avec les nouveaux algorithmes ont confirmé l'intérêt du nouvel ordonnancement des messages.

Perspectives

7.1 Support de communication parallèle

Les systèmes d'entrées/sorties sont généralement utilisés pour accéder aux données persistantes placées dans les fichiers mais ils peuvent également permettre de faire communiquer deux programmes. Sous Unix cela est réalisé à l'aide des «pipes» et des «pipes nommés». Ce mécanisme est très puissant puisqu'il permet de faire coopérer deux programmes en redirigeant simplement les flots d'entrées/sorties.

Avec la bibliothèque que nous avons définie le programmeur accède à un fichier logique et ne sait pas comment sont représentées les données. Il est donc possible d'introduire une notion de «pipe» parallèle nommé sans modifier l'interface de la bibliothèque. Un programme qui utilise des DpoFiles pourrait sans aucune modification être réutilisé avec des «DpoPipes».

Avec ces DpoPipes l'utilisateur pourrait faire coopérer des tâches à parallélisme de données et construire interactivement des applications hétérogènes utilisant plusieurs machines parallèles. La mise en œuvre des DpoPipes ne présente pas de difficultés particulières mais cela nécessite du temps ! Il faut introduire une nouvelle commande Shell pour créer le DpoPipe et modifier le code de la fonction `Dpo_open`.

7.2 Vers une version «domaine public»

Le système de fichiers que nous avons développé pourrait intéresser les utilisateurs de machines MIMD à mémoires distribuées. Pour cela il faut passer de la maquette (qui fonctionne) à un système robuste et bien documenté disponible sur FTP.

DPFS User's Guide and Reference Manual

Dominique Sueur, Philippe Marquet and Jean-Luc Dekeyser
Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille
e-mail: {sueur, marquet, Dekeyser}@lifl.fr

March 3, 1998

Abstract

Most parallel I/O systems allow parallel accesses to sequential files. The **DPFS** is a multi-dimensional file system. Each file is a list of arrays distributed onto a grid of I/O processors as `template` onto **HPF** processor grids.

I/O node topologies and distribution functions are placed at the device level. This means that file distribution only depend on their directory location. By using distributed parallel devices we keep a Unix-like environment. Files and directories are managed with usual `shell` commands.

However, code portability and user friendly are not parallel environment main properties; they are above all built for efficiency. With **DPFS**, parallel I/O are completely asynchronous and the number of disk accesses is minimal. Moreover, when the I/O distribution matches the computing distribution all accesses are local.

Table of Contents

1 Introduction

DPFS is a the Data Parallel File System. It extends the Unix file notion to handle multi-dimensional arrays instead of characters. These data-parallel files are distributed onto grids of I/O processors as `template` onto **HPF** processor grids.

With **DPFS** the I/O topology and the distribution function are directory properties. By this way files can be dynamically and automatically redistributed from one I/O node topology to another one or from an I/O grid to a computing processor grid. These parallel I/O consist of data redistributions.

First we will present the parallel file system. The main components will be illustrated with examples. We will then describe the parallel file model and the SPMD library. Implementation details will be finally presented.

2 Array file for array languages

Multi-dimensional arrays are the fundamental unit of data-parallel programs [7, 6]. Most data-parallel languages like **HPF, F90...** are based on arrays.

With Unix files, data are stored on disk using traditional array ordering which is a simple list of bytes. This data ordering is well suited for sequential environments because array items are stored on the same order in the memory and on the disk. In this context, a multi-dimensional array can be access with a unique read or write call. That is not the case when data is distributed across processor grids. Data items of each processor are scattered across the Unix file. There is no way to address them as a whole.

To minimize the number of disk accesses the user has to choose an I/O node, to perform all the file accesses on this node and then to redistribute the data according to his need (Figure 1). He has to write and to maintain one code for the I/O and an other one for the computation. Moreover all accesses are performed on a single disk. The Unix file model is not dedicated to parallel programming [4, 9]. With this file model, distributed accesses are both error prone and inefficient.

In today's workstation based environment, each Unix machine has its own devices. The **DPFS** allows the user to distribute it arrays on these devices as they are distributed on memory. Data redistributions are automatically performed by the file system when needed (Figure 2).

3 DPFS: a data-parallel file system

3.1 Devices for parallel I/O

With Unix devices the programmer does not need to know where data come from or how these data will be used. Data can come from a satellite receiver, a keyboard or a disk. At the programming level, the data location abstraction is a main concept of the Unix system to ensure code portability. Results can be displayed on a screen, stored on a disk or sent to an other computer (by using Network File System). An equivalent abstraction seems essential for parallel file systems.

On parallel machines, each Unix node owns its devices. To exchange data between nodes a parallel device is built from a set of distributed Unix devices. A DPdevice is a grid of Unix I/O nodes associated with a distribution function. It is similar to a virtual shared memory built from distributed memories. Most of the parallel I/O libraries like VIPS-FS [8] or PANDA [5, 6] associates

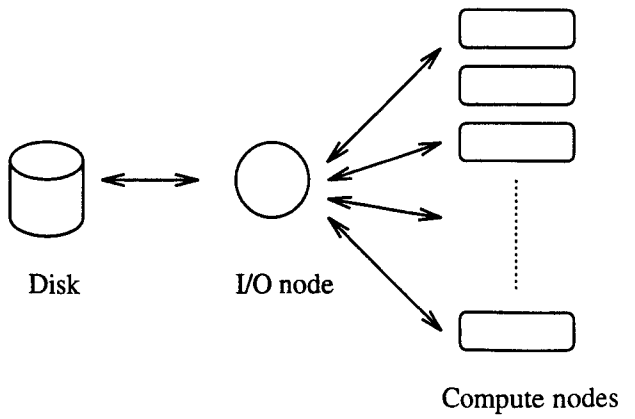


Figure 1: Sequential file system

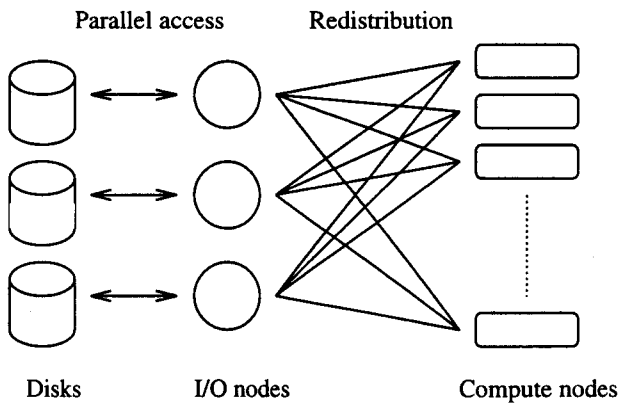


Figure 2: Parallel file system

the distribution to the file, not to the device. They do not provide a virtual parallel device. The program has to manage the file distribution explicitly.

3.2 DPFS Architecture

With the Unix file system, *files* are stored on a *device* according to their *directory* location. To keep the same construction on a parallel file system we introduce three types of objects.

- **DPdevices** are virtual parallel devices. A DPdevice is defined by a template of I/O nodes and a data distribution function. The same concept of distribution is used on I/O nodes and on compute nodes. An array access is performed by a data redistribution between these two topologies. When the compute nodes match the I/O nodes all the accesses are local.

  An example of DPdevice specification is given in Figure 3. The device topology is a bidimensionnal grid of 3×2 nodes. The *nodeij* are virtual nodes. Each virtual node must refer to a physical node with a file path where the data chunks will be stored. Node architecture is

```
TOPOLOGY   ((node00 node01) (node10 node11) (node20 node21))

NODE

  node00        farm1-giga.lifl.fr      ALPHA   /tmp/node00
  node01        farm2-giga.lifl.fr      ALPHA   /tmp/node01
  node10        farm3-giga.lifl.fr      ALPHA   /tmp/node10
  node11        farm4-giga.lifl.fr      ALPHA   /tmp/node11
  node20        farm5-giga.lifl.fr      ALPHA   /tmp/node20
  node21        farm6-giga.lifl.fr      ALPHA   /tmp/node21

DISTRIBUTION     CYCLIC 1 BLOCK
```

Figure 3: DPdevice specification

also given in order to perform data conversion when needed. Finally, a distribution function is given for each dimension of the grid.

Note that device topologies and array ranks are not necessary the same. When a one dimensional array is written on the specified device, it is distributed $Cyclic(1)$ on nodes 00, 10, and 20. Outer dimensions of the I/O grid are not taken into account. When the array rank is higher than the grid rank, the higher array ranks are collapsed onto the grid.

- **DPfiles** are files of homogeneous arrays. All arrays of a same DPfile have the same shape and type. The type, the rank and the sizes of a DPfile are fixed at creation time and cannot be changed.

- **DPdirectories** are Unix directories associated at creation time with a DPdevice. Each DPfile of a DPdirectory is distributed onto the same I/O nodes in respect of the distribution of the associated DPdevice. A DPfile can also be put on an Unix directory. In that case all the data is store on the same I/O node.

  The DPdevice association is only used for DPfiles, it is not taken into account for ordinary Unix objects.

The Figure 4 shows a user working directory. This directory illustrates the main property of the file system: the data distribution function is associated to the DPdevices, not to the DPfiles. A same DPfile can be either on a simple disk or distributed over a bidimensionnal grid. The user can dynamically move his arrays from the Pc to the ALPHA farm. He has only to take the parallel file with the mouse and to put it on the target parallel directory. Data redistribution and conversion (to support heterogeneity) are automatically performed by the file system.

With the DPfile manager[1], the user can navigate through an array in order to visualize the data distribution. He can also select an I/O node to highlight all array chunks stored on a given Unix device.

[1]The graphical interface was done by Christian Lefebvre and Jeremy Allays.

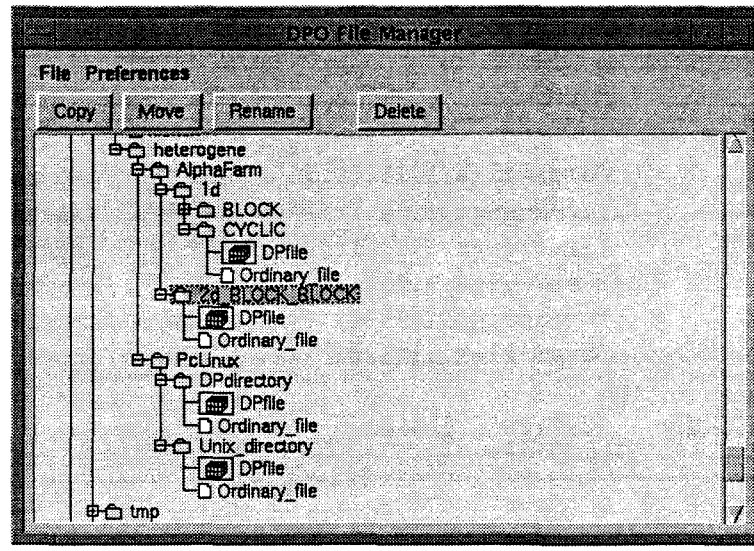


Figure 4: Logical view of a user directory

3.3 Parallel accesses

An array access is processed with two successive steps: each I/O node read his local array chunk and then redistributes it towards the compute grid (fig 5). There are two special cases:

- if the I/O topology and the compute topology are exactly the same, no data distribution is needed. A parallel access is just a set of remote accesses;
- if nodes are both I/O and compute nodes, a parallel access only requires local accesses.

3.4 Shell Commands

Usual shell commands like `cd`, `ls`, `rm`, `mv` are overloaded to manage DPfiles and DPdirectories. New functionalities are added, for example:

- Options have been added to the `ls` command to get parallel object properties (file type and array shape).
- Parallel directories are created with the new `mkdpdir` command.

Consider the following example. An application runs on a 3×2 grid of processors with a (*Cyclic*(1), *Block*) distribution. This application has tremendous I/O requirements but only the first two processors have a local disk. Let us define a DPdevice for it. We create a grid of virtual I/O nodes which exactly matches the compute node topology (`TOPOLOGY` entry on the Figure 3). We map the virtual I/O nodes on the first two processors (`NODE` entry). The I/O node distribution is then copied from the processor distribution. The defined I/O topology insures remote parallel accesses without data distribution.

The `mkdpdir` command creates a DPdirectory on this DPdevice. All files created in the DPdirectory inherit the data distribution. In particular the result of a `cp` or `mv` command in the DPdirectory is redistributed to keep this property.

For `mv` and `cp` commands the following rules are applied :

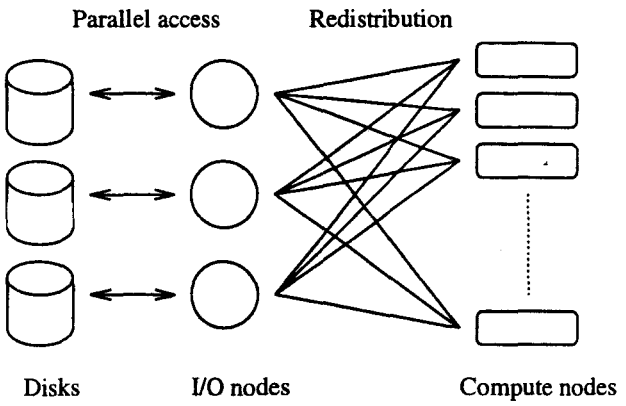


Figure 5: Parallel accesses

Target \ Source	File	Directory	DPfile	DPdirectory
File	std	not allowed	not allowed	not allowed
Directory	std	std	1	4
DPfile	not allowed	not allowed	2	not allowed
DPdirectory	std	std	3	4

1. The DPfile is stored on the Unix device with the XDR[2] data representation.
2. Target DPfile is replaced by the source DPfile. Data representation only depends on the target directory (parallel or not).
3. The DPfile is distributed on the DPdevice associated to the target directory.
4. The DPdevice association is kept. There is no data communication.

Parallel attributes (array rank and type) are associated to the files. A parallel file can be stored on a sequential directory but it never loses its parallel property. It is possible to create a parallel file from a sequential one with the `mkdpfile` command. To do that, give the array shape and type. The reverse operation can also be done with the **DPFS** overloaded `cat` command.

4 The data-parallel file model

A data-parallel file is a sequential stream of homogeneous multi-dimensional arrays. All the arrays of the same file have the same rank and the same sizes. A DPfile can be seen as a Unix file where each byte has been replaced by an array.

4.1 Data partitioning

To balance the tremendous computational bandwidth of parallel machines, parallel devices seem essential. The I/O distribution and the processing distribution are not necessarily similar. These two distinct distributions in the same program complicate the programmer task.

[2] eXternal Data Representation.

Our data-parallel file model hides the file distribution to the programming level. The file distribution only depends on the directory used. The programmer does not need to be aware of it, he only has to choose a distribution on computing nodes. *Block*(K) and *Cyclic*(K) **HPF** distributions are both supported.

The computing distribution is given at opening time and cannot be changed until the DPfile is closed. This distribution fixes the processes visibility over the file. Each data item is exactly owned by one processor. A given processor can only access data it owns according to the specified processing distribution. Note that this distribution is only used for the application and not for data storage. A same file can be successively opened with two distinct processing distributions.

4.2 Data access

Read, write and seek functions are similar to Unix functions but they concern data-parallel objects. The file pointer remains sequential. It gives an offset from the beginning of the file. In Unix this displacement is measured in bytes. In our system the elementary unit is the array. An array is an atomic object: a program cannot access an array chunk or overlap an array with an other. The displacement is expressed in arrays.

5 DPFS access library

To access array files we have built a parallel library for the SPMD model. A same program is executed by a grid of processes working on distributed arrays. The main four functions of the library are illustrated Figure 6. This SPMD program works on an array stream. It may be a `FFT` program or your favorite data-parallel algorithm. We have only kept the access function skeleton.

First the input and output DPfiles are open. The `DPopen` function is a little bit complex with its six parameters. The first two parameters are the file name and the access mode (read, write or append). The third one encloses the computation grid and distribution as well as the process coordinates on this grid. This parameter defines the process view over the file. Fourth and fifth parameters are the array shape and type. The remaining variable is only for error checking. Note that there is no information about the file distribution. The physical representation of the file only depends of the directory used, it is implicitly given with the file name. The `DPopen` function returns a pointer to the file. Like the Unix `fopen` function, the pointer is `NULL` if an error occurs. In that case the `DPperror` function is called to know what has happened.

With the `DPread` function each processor reads its own data set according to the processing distribution specified at open time. `DPread` returns the number of arrays actually accessed. This result is used to loop until the end of the file. Arrays are treated one by one and the result is stored in the output file. Finally the function `DPclose` is called twice to close the parallel files. Thanks to the DPdevice motion, this program can be used whatever the source and the target file distribution. The data location is only given at run-time according to the user needs, the programmer has not to care about it.

In order to avoid the cost of unnecessary synchronizations, read, write, open and close functions are completely asynchronous. Each process owns a local file pointer. That means that two processes can access distinct arrays at the same time. On the example the source and the target files are opened on the same local grid. The same data partitioning is kept for both of them. Processors cannot communicate through the file even if the output result overwrite the input data. There is no need for synchronization. In fact synchronizations are only necessary when a same physical file

```
main(int argc, char *argv[]){
  ....
  DP_STREAMS *f1,*f2;

  /*          File opening and error checking              */

  f1=DPopen(argv[1],"r",&local_grid,&shape,&type,&error_r);
  f2=DPopen(argv[2],"w",&local_grid,&shape,&type,&error_w);

  if (f1==NULL) {DPperror(&error_r, "Source DPfile"); exit(2);}
  if (f2==NULL) {DPperror(&error_w, "Target DPfile"); exit(2);}

  buffer=DPmalloc(f1, 1);  /*          memory allocation       */

  /*    arrays are treated each in turn until the last one    */

  while (DPread(f1,buffer,1) == 1){
    Do_the_job(buffer);
    DPwrite(f2,buffer,1);
  }
  free(buffer);
  DPclose(f1);
  DPclose(f2);
}
```

Figure 6: Example of the **DPFS** access library

is opened both in read and write mode with two distinct distribution functions. We believe that such file usage are not very common.

6 Performance issues

We have defined a high level interface which insulates programmers from the physical storage implementation, we will see that this interface not only eases the programmer task but also allows performances.

Experiments reported in Figure 7 have been done on a farm of ALPHA processors interconnected by a cross-bar supporting FDDI bandwidth (about 50Mb/s per link with **PVM**[3]). Each processor has a local disk; peak disk bandwidth is about 200Mb/s. Reported values are average results of 10 runs. In each case, we record the time necessary to read a *Block* distributed array of 500.000 integers (20 Mb) from a first set of P I/O nodes, toward another set of P' computing nodes with the same distribution function. In each case, nodes with number lower than $min(P, P')$ were both I/O nodes and compute nodes.

The results show the impact of data locality. The same I/O distribution for I/O nodes and

[3]Parallel Virtual Machine

Disk \ Node	1	2	3	4	5	6	7	8	9	10
1	190	19	21	21	20	18	19	19	19	17
2	21	390	37	26	25	24	23	23	23	23
3	25	34	590	47	45	41	39	33	32	32
4	63	36	51	770	56	57	52	50	51	45
5	43	41	63	65	930	65	62	55	54	53
6	56	48	71	65	70	1100	69	68	59	59
7	65	43	72	72	74	73	1200	73	69	66
8	50	55	69	72	76	66	75	1300	74	69
9	69	47	74	74	75	80	83	78	1700	71
10	67	61	72	80	83	78	79	73	71	1700

Figure 7: Performance results in Mb/s

compute nodes insures local accesses without any communication. In that case the speedup is linear. The available bandwidth with ten disks is about 1.7 Gb/s. We have a very efficient virtual distributed device. Unfortunately the figures are not the same with distinct topologies. In that case, arrays have to be redistributed so we have to pay for the address computation and for the data communication. With distinct topologies the available bandwidth mostly depends on:

- the array size. The network latency is very high for small arrays.
- the number of I/O nodes. The more they are, the better the performance.

The disk access cost is the same whatever the source and target distributions are.

With low level interface, performances can greatly vary as a function of the data distribution. Performances are good when arrays in memory correspond to contiguous storage locations on disk. The file system performs poorly if the elements are scattered across the file [9]. To avoid this we used the "two-phase access strategy" developed by Juan Miguel Rosario, Rajesh Bordawekar and Alok Choudhary [3]. This strategy involves a division of the parallel I/O functions into two separate phases. First, data accesses are performed on the I/O nodes in respect of the data distribution over the disks. In a second phase, the data are redistributed to match the distribution required by the application.

With our high level interface, the file system has a global view of the data transfers. The two-phase strategy have been easily implemented. Arrays are first read with only one disk access on each I/O node and then redistributed towards the computing grid. Array migrations from one grid to another leads us to define efficient algorithms for runtime redistributions. We proposed specialized dynamic redistribution algorithms [1, 2]. In particular we have shown that we can enumerate processor indexes and create messages by only using a set of regular sections $(first : last : stride)$. In each case, the communication volume and the number of message sent are kept minimal.

7 Creating DPFS applications

This section details how to install and then use **DPFS**. It is presumed that **PVM** is already installed for each machine that is to host the data-parallel file system.

```
TOPOLOGY (node0 node1)

NODE

node0    farm1-giga      alpha   /tmp/node0
node1    farm2-giga      alpha   /tmp/node1

DISTRIBUTION CYCLIC 1
```

Figure 8: Data-parallel device specification

```
Drwxr-xr-x   2 sueur    west          512 Feb 20 10:08 ./
drwxr-xr-x   4 sueur    west          512 Feb 20 10:05 ../
-rw-r--r--   1 sueur    west         1325 Feb 20 11:18 Manual.tex
Frw-r--r--   1 sueur    west        16000 Feb 20 10:08 VA
Frw-r--r--   1 sueur    west        16000 Feb 20 10:08 VB
```

Figure 9: Logical view of the `/home/sueur/C1` data-parallel directory

7.1 DPFS compilation

DPFS only needs **PVM**, a Unix interface and an **AINSI C** compiler. You can edit the file `Makefile.path` to change the target directory.

7.2 Starting DPFS

Before to start a **DPFS** application you have to spawn the `pvmgs` (the **PVM** group server) and the **DPFS** `manager` process in your current Parallel Virtual Machine.

Any SPMD application can use the virtual file system by using the **DPFS** library. **DPFS** functions will not interfere with your **PVM** application if it does not used any of the set of consecutive tags reserved for **DPFS**. In case of trouble you can translate **DPFS** tags by editing the `TAG_START` definition in the `DPFS/source/message.h` file.

To compile a **DPFS** application, include the require header file of the `DPFS/include` directory and link with `libpvm3.a, libgpvm3.a, libdpfs.a` in that order.

8 Implementation details

DPFS is built on top of the Unix file system with the **PVM** library. Parallel objects (files and directories) are distributed on several I/O nodes. In this section will see that a single logical file entry may be represented by several Unix files. This files are managed by **DPFS** and remain hidden at the application level.

```
drwxr-xr-x   2 sueur     west          512 Feb 20 10:08 ./
drwxr-xr-x   4 sueur     west          512 Feb 20 10:05 ../
-rw-r--r--   1 sueur     west          141 Feb 20 10:05 .DPdir
-rw-r--r--   1 sueur     west           32 Feb 20 10:08 .DPfile_VA
-rw-r--r--   1 sueur     west           32 Feb 20 10:08 .DPfile_VB
-rw-r--r--   1 sueur     west         1325 Feb 20 11:18 Manual.tex
-rw-r--r--   1 sueur     west            0 Feb 20 10:08 VA
-rw-r--r--   1 sueur     west            0 Feb 20 10:08 VB
```

Figure 10: Physical representation of the `/home/sueur/C1` directory (manager view)

```
drwxr-xr-x   2 sueur     west          512 Feb 20 10:00 ./
drwxr-xr-x   3 sueur     west          512 Feb 20 09:57 ../
-rw-r--r--   1 sueur     west         8000 Feb 20 10:08 .DPdata_VA
-rw-r--r--   1 sueur     west         8000 Feb 20 10:08 .DPdata_VB
```

Figure 11: Physical representation of the `/tmp/node0/home/sueur/C1` directory (I/O node view)

8.1 Physical representation

In the figure 8, we defined a one dimensional topology of two I/O nodes. `node0` and `node1` are virtual I/O nodes. They refer to the physical processors `farm1-giga` and `farm2-giga`. Data-parallel files written on this DPdevice will be distributed *Cyclic*(1) on the two I/O nodes, and they will be stored under the `/tmp/node0`, `/tmp/node1` regular directories.

Data-parallel directories The `/tmp/home/sueur/C1` data-parallel directory is associated to the DPdevice we have previously defined. The figure 10 is the output of the **DPFS** `ls -l` shell command on this directory. Data-parallel directories and regular directories look like very much the same. To distinguish parallel entries from regular one, new flags have been added to the `ls` command (**D** for data-parallel directories, **F** for data-parallel files). On this example

- `/home/sueur` is a regular directory;
- `/home/sueur/C1` is a data-parallel directory;
- `/home/sueur/Manual.tex` is a regular file;
- `/home/sueur/VA` and `/home/sueur/VB` are data-parallel file. Each of them contain an array of 4.000 integers (the integer size is 4 bytes).

DPFS `ls` command command give the logical view of the file system. Figure 10 is the output of the `/bin/ls` usual Unix command on the local machine. It give an insight of the physical data representation. Each data-parallel directory is represented by:

- a Unix directory (i.e. `/home/sueur/C1`);
- a description file which defined the I/O nodes topology and distribution (i.e. `/home/sueur/.DPdir`);
- a Unix directory on each I/O node (for example `/tmp/node0/home/sueur/C1` Figure 11). The logical directory name is prefixed by the path specified at the DPdevice level.

Data-parallel files Each data-parallel file is represented by at least three Unix files:

- an empty file (i.e. `/home/sueur/C1/VA`);
- a description file (i.e. `/home/sueur/C1/.DPfile_VA`) to store the array shape and type;
- one data file on each I/O node (i.e. `/tmp/node0/home/sueur/C1/.DPdata_VA`). When a DPfile is stored on a Unix regular directory, the data file and the description file are located on the same Unix directory.

8.2 DPFS processes

Distributed file access and management are performed using three distinct type of processes:

The manager process gives the logical view of the file system. It must have an access to the description files. It is spawn by the user and remain alive until the end of the PVM session.

Local manager processes are used on each I/O node to manage the local file systems. There are for example used to change the DPfile access rights. These processes only manage regular files (they don't know anything about parallel objects). They are spawn by the manager and remain alive until the end of the PVM session. These processes do not concern user's application.

Data server are created by `DPopen` functions on each I/O nodes. They perform Unix file access, data redistribution and conversion when necessary. They are associated with a `DPSTREAMS` and remain alive until the associated `DPSTREAMS` is closed.

Clients processes can access all the parallel files seen by the manager even if they have no local file system.

8.3 Memory mapping

The data distribution is specified by the programmer at open time. In Figure 13, the same array T, is distributed over two distinct grids.

- The first is a one dimensional grid of three processes. It is associated with a $Cyclic(1)$ distribution. Since the array is a bi-dimensional array, the last dimension is collapsed into process memories. The `DPmalloc` allocates only the require memory (10 integers for the first two processes, only 5 for the last one). Data items and mapped on memory like they are with the **C** language (Figure 14).
- The second grid has two dimensions. The associated distribution is $Cyclic(2)$, *Block*. The first distribution function ($Cyclic(2)$) is apply to the first dimension of the array. The *Block* distribution is apply to the second one (distribution functions are apply, dimension by dimension).

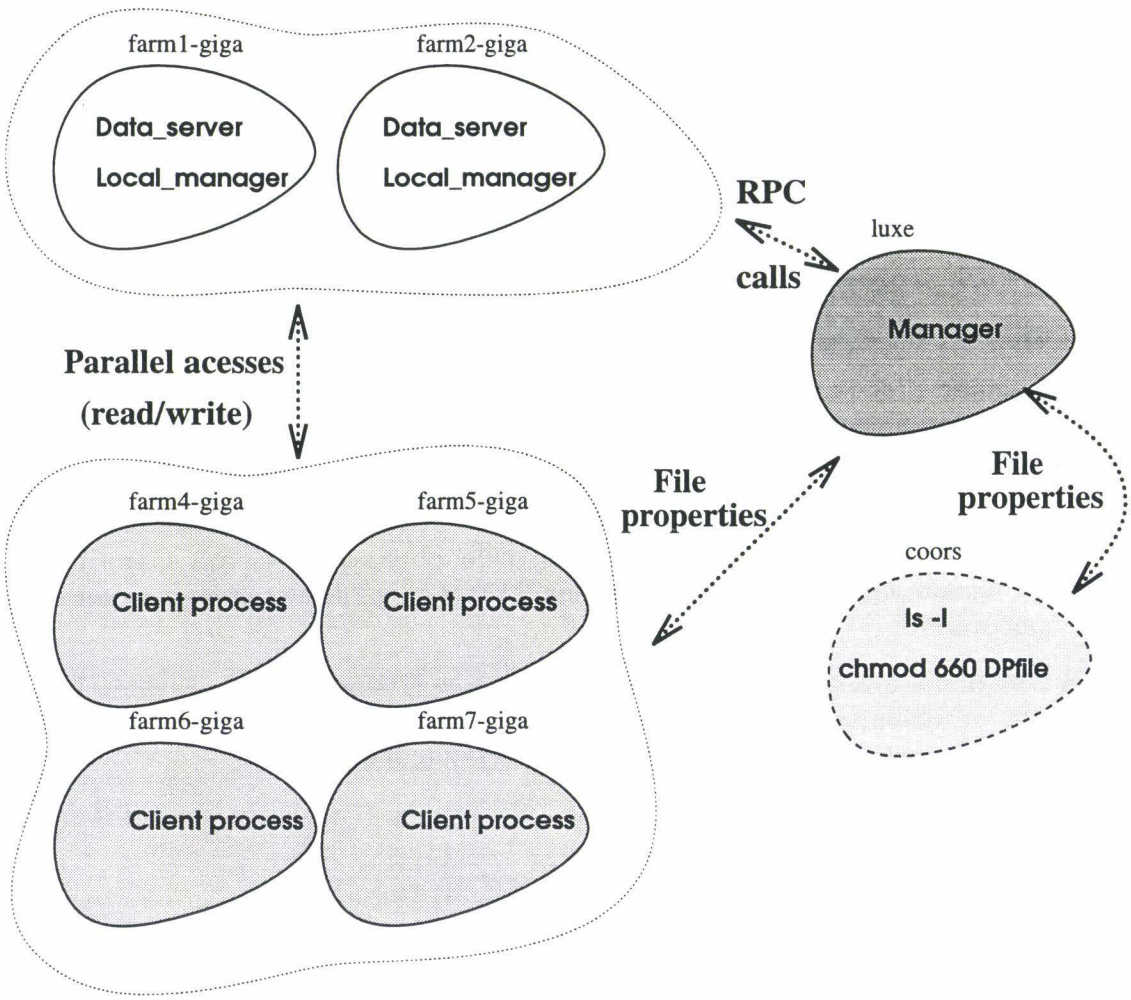


Figure 12: Parallel file system architecture

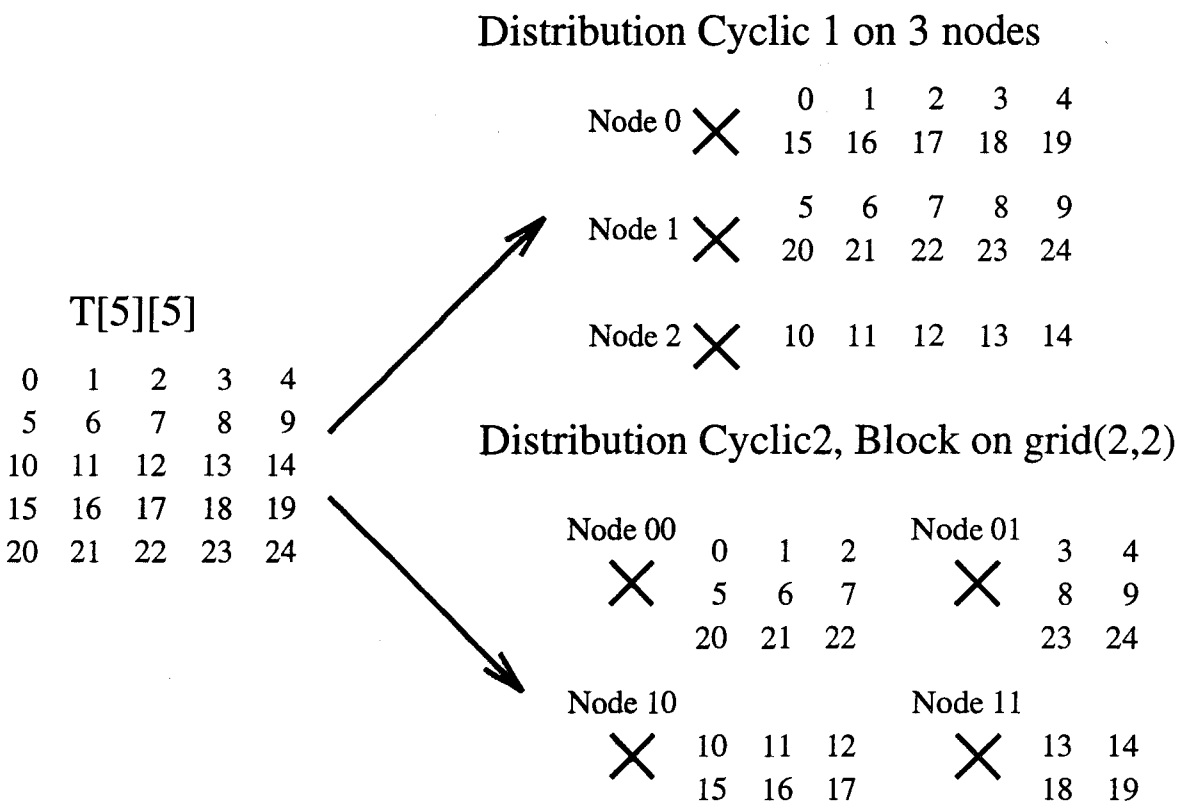


Figure 13: Data distribution

Local buffers

Node 00	0 1 2 5 6 7 20 21 22	Node 0	0 1 2 3 4 15 16 17 18 19
Node 01	3 4 8 9 23 24	Node 1	5 6 7 8 9 20 21 22 23 24
Node 10	10 11 12 15 16 17	Node 2	10 11 12 13 14
Node 11	13 14 18 19		

Figure 14: Memory mapping

8.4 Shell commands

To understand how **DPFS** works we are going to detail what happen when the user changes the access mode of a data-parallel file with the `chmod` shell command.

DPFS `chmod` command is a modified version of the **GNU** `chmod`, we have just added severals tests to support parallel entries. First, `chmod` asks the manager for the file entry type. This is done with the `DPfiletype` function. It requires two communications between the manager and the chmod process. For parallel entries, the `chmod` Unix call is replaced by a `DPchmod` **DPFS** call. In that case the `manager` change the mode of the local files and ask the `local_managers` to do the same on there `data_file` (Figure 12).

For example, to change the access mode of the `/home/sueur/C1/VA` DPfile (Figure 9), the `manager` first change the access mode of `/home/sueur/C1/VA`, `/home/sueur/C1/.DPfile_VA` files (Figure 10). In case of trouble, a scalar error is reported, otherwise an RPC is sent to the `local_managers` of `farm1-giga` and `farm2-giga` to change the data file access mode (Figure 11).

8.5 Parallel accesses

To initiate parallel accesses, the `DPopen` function ask for file properties. A `Data_server` is then spawned an each I/O nodes[4]. Parallel access are then performed with a remote access or a data redistribution between the data servers and the application processes (Figure 12). Note that the process manager gives only the file properties. It does not represent a system bottleneck.

9 Program example

Dot product

Here we show a simple `SPMD C` program that computes dot products. It works on two parallel files containing one dimensional arrays. To perform the dot product in parallel we equally distribute the two arrays on a set of slave processes. Each slave performs a local dot product on his local array chunk and sends the local result to a master process. The master receives all the local dot products and sum them into a global result.

First, each process enrolls itself in the **PVM** machine with the `process_enroll` function. It then calls the `get_parent_tid`. If the process wasn't spawn by another **PVM** process, `get_parent_tid` return `NoParent`. In that case the current process is the master, it must check the arguments and spawn copies of the current program.

The master process checks the parameters. The two DPfiles must have the same numbers of arrays. Arrays must have the same shape and they must be arrays of integers. If every thing OK, the master spawn a grid of slave processes. In order to avoid unnecessary communication, it spawns one process on each I/O node of the first parallel directory.

To perform the final reduction, the master process waits after all the local dot products, sums then and prints the global result.

The slave processes first open the two DPfiles. Since the application and the first DPdevice have the same distribution, the first `DPopen` function is just a call to the Unix `fopen` function. We don't know anything about the second DPfile. It may be on a remote machine. It that case, the second `DPopen` function spawns a `data_server` process on each I/O node.

[4] local accesses are done with usual Unix functions

With the DPSTREAMS descriptors, two buffers are allocated to store the local array chuncks. The processes then loop until the end of file (the two files have the same number of arrays). They read an array on each DPfile, compute a local result and send it to the master. At the end of the loop, they release the memory and close the DPfiles.

Dot product example

```
#include <stdio.h>
#include <type.h>
#include <constant.h>
#include <DPIO.h>
#include <DPstat.h>
#include "../source/message.h"
#include "../source/comm.h"
#include "../source/process.h"

#define NB_ARRAY 1
#define RESULT   2
void main(int argc, char *argv[]){
  process_enroll();
  if (get_parent_tid() == NoParent){

    /* master process, it spawns the others */

    ERRNOS          errors;
    MAPPINGS        mapping;
    HOST_INFOS      hosts;
    GRIDS           grid;
    SHAPES          shape1, shape2;
    TYPES           type1, type2;
    ERRNOS          error1, error2;
    NATS            i, j, nb_array1, nb_array2;
    RTS             result1, result2;
    char            directory_name[MAX_PATH];
    int             global_result, local_result;

    /* first we check the parameters and we get DPdirectory properties */
    /* To avoid data migration, the compute topology is the topology   */
    /* of the first DPdirectory.                                       */

    if (argc != 3) usage();
    sprintf(directory_name, "%s/..", argv[1]);
    if (DPdirinfo(directory_name, &mapping, &hosts, &errors) == Problem) {
      DPperror(&errors, "DPdirinfo");
      exit(2);
    }
    result1=DPfileinfo(argv[1], &shape1, &type1, &nb_array1, &error1);
```

```
    result2=DPfileinfo(argv[2], &shape2, &type2, &nb_array2, &error2);
    if (result1 == Problem) {
      DPperror(&error1, "DPfileinfo argv1"); that_all_folks();}
    if (result2 == Problem) {
      DPperror(&error2, "DPfileinfo argv2"); that_all_folks();}
    if (!same_shape(&shape1, &shape2) || (type1 != type2)) {
      fprintf(stderr, "Not the same kind of arrays !\n");
      that_all_folks();
    }
    if (nb_array1 != nb_array2){
      fprintf(stderr, "what should I do with extra arrays ?\n");
      that_all_folks();
    }
    /* It's Ok, we can start the slave processes            */

    if (grid_spawn(&mapping,hosts.physical,&grid,
                   argv[0],(CST char **)argv) == Problem){
      fprintf(stderr,"can' t start grid\n");
      exit(2);
    }
    /* the final reductions are done by me (what a shame !) */

    printf("The dot products are:\n");
    for (j=0; j<nb_array1; j++){
      global_result=0;
      for (i=0; i<grid.mapping.nb_total; i++){
        recv(grid.tids[i], RESULT);
        upk_int(&local_result);
        global_result+=local_result;
      }
      printf("%d ", global_result);
    }
  }
  else {
    /* I'm a poor little slave => I have to do the job ! */

    LOCAL_GRIDS     lgrid;

    grid_init(&lgrid);
    dot_product(argv, &lgrid);
  }
  /* master or slave, every thing come to an end ! */
  process_exit();
}

void usage(void){
  fprintf(stderr, "Usage: dot_product DPfile DPfile\n");
```

```
  exit(2);
}

void dot_product(char *argv[], LOCAL_GRIDS *lgrid){
  DPSTREAMS    *f1, *f2;
  SHAPES       shape1, shape2;
  TYPES        type1, type2;
  ERRNOS       error1, error2;
  int          *buffer1, *buffer2;
  SIZES        size;
  NATS         i=strlen(argv[2]), nb_item;
  char         DPfile[MAX_PATH];
  char         target[MAX_PATH];
  int          result, parent_tid=get_parent_tid();

  f1=DPopen(argv[2], "r", lgrid, &shape1, &type1, &error1);
  f2=DPopen(argv[3], "r", lgrid, &shape2, &type2, &error2);

  /* We allocate two buffers to store the local array chuncks */

  buffer1=(int *)DPmalloc(f1, NB_ARRAY);
  buffer2=(int *)DPmalloc(f1, NB_ARRAY);
  nb_item=DPlocal_nb_item(f1);

  while ((DPread(f1, (char *)buffer1, NB_ARRAY, &error1) == 1 )&&
         (DPread(f2, (char *)buffer2, NB_ARRAY, &error2) == 1 )) {

    /* first we perform the local dot products                */
    result=0;
    for (i=0; i<nb_item; i++) result+=buffer1[i]*buffer2[i];

    /* the global reduction is then done by the master process */
    mk_default();
    pk_int(result);
    send(parent_tid, RESULT);
  }
  /* buffers are no longer needed */
  free(buffer1);
  free(buffer2);
  DPclose(f1, &error1);
  DPclose(f2, &error2);
}

void that_all_folks(void){
  process_exit();
  exit(2);
}
```

10 Conclusion

Getting good I/O performances from parallel programs is a critical problem for many application domains. Few workstations have parallel I/O capabilities but a lot of them have several Unix devices.

The **DPFS** extends the Unix file notion to handle multi-dimensional arrays instead of characters. These data-parallel files are distributed onto Unix devices as `template` onto **HPF** processors grids. To relieve the programmer from the physical data representation, the distribution function is fixed at the DPdevice level. By this way arrays can be automatically redistributed from one I/O node topology to another one or from one I/O grid to a computing processor grid.

A file system is above all a permanent repository of informations between applications. The parallel file system *raison d'être* is performance. With **DPFS** the user has not to choose between these two qualities. A parallel file can always be accessed by an application whatever its distribution is.

References

[1] Dominique Sueur, *Algorithmes de redistribution de données, application aux systèmes de fichiers parallèles distribués*, Thèse de doctorat, Université des Sciences et Technologies de Lille, June 1997.

[2] Dominique Sueur and Jean Luc Dekeyser, *Dynamic Redistribution on Heterogeneous Parallel computers*, Euro-par'96, École Normale supérieure de Lyon, August 1996.

[3] Juan Miguel del Rosario and Rajesh Bordawekar and Alok Choudhary, *Improved Parallel I/O via a Two-Phase Run-time Access Strategy*, IPPS'93 Workshop on Input/Output in Parallel Computer Systems, April 1993.

[4] Juan Miguel del Rosario and Alok Choudhary, *High Performance I/O for Parallel Computers: Problems and Prospects*, IEEE computer, 27(3):59-68, March 1993.

[5] K. E. Seamons and Y. Chen and M. Winslett and Y. Cho and S. Kuo and M. Subramaniam, *Persistent Array Access Using Server-Directed I/O*, 8th International Working Conference on Scientific and Statistical Database Management, Stockholm, Sweden, june 1996.

[6] Kent Eldon Seamons, *PANDA: fast access to persistent arrays using high level interfaces and server directed input/output*, University of Illinois at Urbana-Champaign, 1996.

[7] Jean Luc Bell and George Patterson, *Data Organization in Large Numerical Computations*, The Journal of Supercomputing, 1 (1) 1987.

[8] Michael Harry and Juan Miguel del Rosario and Alok Choudhary, *VIP-FS: A VIrtual, Parallel File System for High Performance Parallel and Distributed Computing*, Ninth International Parallel Processing Symposium, April 1995.

[9] Thomas H. Cormen and David Kotz, *Integrating Theory and Practice in Parallel File Systems*, Proceedings of the 1993 DAGS/PC Symposium, pp. 64-74, Hanover, NH, June 1994.

NAME

DPchmod – change permissions of a parallel file entry

SYNOPSIS

#include <sys/types.h>
#include <sys/time.h>
#include <constant.h>
#include <type.h>
#include <DPerrno.h>
#include <DPstat.h>

RTS DPchmod (CST char ****path*, **mode_t *mode*, **ERRNOS ****error);*

DESCRIPTION

The mode of the file given by *path* is changed. *path* must be visible by the manager process.

Modes are specified by *or'ing* the following:

S_ISUID	04000 set user ID on execution
S_ISGID	02000 set group ID on execution
S_ISVTX	01000 sticky bit
S_IRUSR (S_IREAD)	00400 read by owner
S_IWUSR (S_IWRITE)	00200 write by owner
S_IXUSR (S_IEXEC)	00100 execute/search by owner
S_IRGRP	00040 read by group
S_IWGRP	00020 write by group
S_IXGRP	00010 execute/search by group
S_IROTH	00004 read by others
S_IWOTH	00002 write by others
S_IXOTH	00001 execute/search by others

The effective UID of the process must be zero or must match the owner of the file.

The effective UID or GID must be appropriate for setting execution bits.

Depending on the file system, set user ID and set group ID execution bits may be turned off if a file is written. On some file systems, only the super-user can set the sticky bit, which may have a special meaning (i.e., for directories, a file can only be deleted by the owner or the super-user).

RETURN VALUE

On success, **Ok** is returned. On error, **Problem** is returned, and *error* is set appropriately.

ERRORS

Depending on the file system, other errors can be returned. The more general errors for **DPchmod** are listed below:

EPERM The effective UID does not match the owner of the file, and is not zero.

EROFS The named file resides on a read-only file system.

EFAULT *path* points outside your accessible address space.

ENAMETOOLONG
path is too long.

ENOENT The file does not exist.

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component of the path prefix is not a directory.

EACCES Search permission is denied on a component of the path prefix.

ELOOP *path* contains a circular reference (i.e. via a symbolic link)

EFCORRUPT
File corrupted.

EHOSTS Can't contact host.

SEE ALSO

DPchown(2), **DPutime**(2), **chmod**(2)

AUTHORS AND CONTRIBUTORS

Based on GNU chmod function.

NAME

DPchown – change ownership of a DPentry

SYNOPSIS

#include <unistd.h>
#include <sys/types.h>
#include <constant.h>
#include <type.h>
#include <DPerrno.h>
#include <DPstat.h>

RTS DPchown (CST char ****path*, CST uid_t** *owner*,
CST gid_t *group*, **ERRNOS ****error*);

DESCRIPTION

The owner of the file specified by *path* is changed. Only the super-user may change the owner of a file. The owner of a file may change the group of the file to any group of which that owner is a member. The super-user may change the group arbitrarily. *path* must be visible by the manager process.

If the *owner* or *group* is specified as −1, then that ID is not changed.

RETURN VALUE

On success, **Ok** is returned. On error, **Problem** is returned, and *error* is set appropriately.

ERRORS

Depending on the file system, other errors can be returned. The more general errors for **DPchown** are listed below:

EPERM The effective UID does not match the owner of the file, and is not zero; or the *owner* or *group* were specified incorrectly.

EROFS The named file resides on a read-only file system.

EFAULT *path* points outside your accessible address space.

ENAMETOOLONG
path is too long.

ENOENT The file does not exist.

ENOMEM Insufficient kernel memory was available.

ENOTDIR A component of the path prefix is not a directory.

EACCES Search permission is denied on a component of the path prefix.

ELOOP *path* contains a circular reference (i.e., via a symbolic link)

EFCORRUPT
File corrupted.

EHOSTS Can't contact host.

NOTES

DPchown does not follow symbolic links.

SEE ALSO

DPchmod(2), **DPutime**(2), **chown**(2)

AUTHORS AND CONTRIBUTORS

Based on GNU chown function.

NAME

DPclose – close a data parallel stream

SYNOPSIS

#include <constant.h>
#include <type.h>
#include <DPIO.h>

**int DPclose(DPSTREAMS ****stream*, **ERRNOS ****error*);

DESCRIPTION

The **DPclose** function dissociates the *DPstream* from its underlying file.

RETURN VALUES

Upon successful completion 0 is returned. Otherwise, **EOF** is returned and the parameter *error* is set to indicate the error. In either case no further access to the stream is possible.

ERRORS

ENODPSTREAM
The argument *DPstream* is not an open DPstream.

ERGRID
Remote I/O grid not responding.

EHOSTS
Can't contact host.

SEE ALSO

DPopen(3), **DPread**(3), **DPwrite**(3), **fclose**(3)

NAME

DPclosedir – close a directory stream (parallel or not)

SYNOPSIS

#include <constant.h>
#include <type.h>
#include <dirent.h>
#include <DPdirent.h>

int DPclosedir (DIR *dir);

DESCRIPTION

The **DPclosedir** function closes the directory stream associated with *dir*. The directory stream descriptor *dir* is not available after this call. **DPopendir**, **DPreaddir**, **DPrewinddir**, and **DPclosedir**, are provided to skip DPFS system's files.

RETURN VALUE

The **DPclosedir** function returns 0 on success or −1 on failure.

ERRORS

EBADF
Invalid directory stream descriptor *dir*.

SEE ALSO

DPopendir(3), **DPreaddir**(3), **DPrewinddir**(3), **closedir**(3)

NAME

DPdirinfo – get DPdirectory properties

SYNOPSIS

#include <unistd.h>
#include <sys/types.h>
#include <constant.h>
#include <type.h>
#include <DPerrno.h>
#include <DPstat.h>

RTS DPdirinfo(CST char ****path*, MAPPINGS ****mapping***,**
HOST_INFOS ****hosts*, ERRNOS ****errors***);**

DESCRIPTION

DPdirinfo returns informations on the *mapping* and *hosts* parameters about the data-parallel directory whose name is the string pointed to by *path*. *path* must be visible by the manager process. A Unix directory is a special data-parallel directory with only one node (i.e topology rank = nb_node = 1, distribution = COLLAPSED, host = manager process host, path = NULL, ARCHITECTURE = XDR).

MAPPINGS structure is declared in type.h as follows:

```
typedef struct {
  NATS nb_dim;
  NATS nb_node[MAX_DIM];
  DISTRIBUTIONS d[MAX_DIM];
  NATS nb_total;
} MAPPINGS;
```

HOST_INFOS structure is declared in type.h as follows:

```
typedef struct {
  VIRTUAL_HOSTS   virtual[MAX_NODE];
  PHYSICAL_HOSTS  physical[MAX_NODE];
  PATH_HOSTS .    path[MAX_NODE];
  ARCHITECTURES   architecture[MAX_NODE];
} HOST_INFOS;
```

RETURN VALUE

On success. **Ok** is returned. On error, **Problem** is returned, and *error* is set appropriately.

ERRORS

EDCORRUPT
DPdevice corrupted.

ENOENT
File does not exist.

EHOSTS
Can't contact host.

SEE ALSO

DPfilesize(2), **DPfileinfo**(2), **DPfiletype**(2)

NAME

DPfileinfo – get DPfile properties

SYNOPSIS

#include <unistd.h>
#include <sys/types.h>
#include <constant.h>
#include <type.h>
#include <DPerrno.h>
#include <DPstat.h>

RTS DPfileinfo(CST char ****path*, **SHAPES *****shape*, **TYPES *****type,
**NATS *****nb_array*, **ERRNOS *****errors*);

DESCRIPTION

DPfileinfo returns informations on the *shape*, *type* and *nb_array* parameters about the data-parallel file whose name is the string pointed to by *path*. *path* must be visible by the manager process.

SHAPES structure is declared in type.h as follows:

```
typedef struct {
  NATS nb_dim;
  NATS size[MAX_DIM];
  NATS nb_item;
} SHAPES;
```

The array *type* is one of the following: **Int**, **Long**, **Float**, **Double**, **Char**, **Boolean**.

RETURN VALUE

On success, **Ok** is returned. On error, **Problem** is returned, and *error* is set appropriately.

ERRORS

ENODPFILE
Not a data-parallel file.

EFCORRUPT
File corrupted.

ENOENT
File does not exist.

EHOSTS
Can't contact host.

SEE ALSO

DPfilesize(2), **DPdirinfo**(2), **DPfiletype**(2)

NAME

DPfilesize – give the file size in byte

SYNOPSIS

#include <constant.h>
#include <type.h>
#include <DPerrno.h>
#include <DPstat.h>

RTS DPfilesize (CST char ****path*, NATS ****file_size***, ERRNOS ****errors***);**

DESCRIPTION

DPfiletype function returns the size of the file *path* in the *file_size* parameter. You do not need any access rights to the file to get this information but you need search rights to all directories named in the path leading to the file. *path* must be visible by the manager process.

RETURN VALUE

On success, **Ok** is returned. On error, **Problem** is returned, and *error* is set appropriately.

ERRORS

EFCORRUPT
File corrupted.

ENOENT
File does not exist.

EHOSTS
Can't contact host.

SEE ALSO

DPfiletype(2), **DPfileinfo**(2), **DPdirinfo**(2)

NAME

DPfiletype – give the file entry type

SYNOPSIS

#include <constant.h>
#include <type.h>
#include <DPerrno.h>
#include <DPstat.h>

RTS DPfiletype (CST char * *path*, **FILE_TYPES *** *file_type*, **ERRNOS *** *error);*

DESCRIPTION

DPfiletype function return information in the *file_type* parameter about the specified file: *path*. You do not need any access rights to the file to get this information but you need search rights to all directories named in the path leading to the file. *path* must be visible by the manager process.

file_type value is one of the following:

Islink (a Unix link)

Isreg (a regular file)

Isdir (a regular directory)

Ischr (a character device)

Isblk (a block device)

Ifsock (a socket)

Isdpfile
(a data-parallel file)

Isdpdir
(a data-parallel directory)

RETURN VALUE

On success, **Ok** is returned. On error, **Problem** is returned, and *error* is set appropriately.

ERRORS

EFCORRUPT
File corrupted.

ENOENT
File does not exist.

EHOSTS
Can't contact host.

SEE ALSO

DPfilesize(2), **DPfileinfo**(2), **DPdirinfo**(2)

NAME

DPmalloc – Allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>
#include <constant.h>
#include <type.h>
#include <DPIO.h>

char *DPmalloc(CST DPSTREAMS * *stream*, **CST NATS** *nb*);

DESCRIPTION

DPmalloc allocates memory on the local process to stored *nb* distributed arrays according to the distribution associated to the *stream*.

The data parallel *stream* refers to the local process grid, the local distribution function, the array shape and type. This informations are used by the **DPmalloc** function to allocate the local memory.

RETURN VALUES

The **DPmalloc** return value is a pointer to the allocated memory or **NULL** if the request fails.

SEE ALSO

malloc(3), **free**(3)

NAME

DPmkdpdir – create a data-parallel directory

SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>
#include <constant.h>
#include <type.h>
#include <DPerrno.h>
#include <DPstat.h>

RTS DPmkdpdir(CST MAPPINGS *mapping, CST HOST_INFOS hosts,
        CST char *path, CST mode_t mode, ERRNOS *error);
```

DESCRIPTION

DPmkdpdir attempts to create a data-parallel directory named *path*.

MAPPINGS and *HOST_INFOS* structures are declared in type.h as follows:

```
typedef struct {
  NATS nb_dim;
  NATS nb_node[MAX_DIM];
  DISTRIBUTIONS d[MAX_DIM];
  NATS nb_total;
} MAPPINGS;

typedef struct {
  VIRTUAL_HOSTS   virtual[MAX_NODE];
  PHYSICAL_HOSTS  physical[MAX_NODE];
  PATH_HOSTS      path[MAX_NODE];
  ARCHITECTURES   architecture[MAX_NODE];
} HOST_INFOS;
```

mapping defines the logical I/O grid topologie and distribution. *hosts* refers to the physical nodes and path that will be used at each access. Each data-parallel files that will be store on the directory will be distributed across the I/O nodes according to the specified distribution.

mode specifies the permissions to use. It is modified by the process's **umask** in the usual way: the permissions of the created file are **(mode & ~umask)**.

The newly created directory will be owned by the effective uid of the process. If the parent directory has the set group id bit set then so will the newly created directory.

RETURN VALUE

On success, **Ok** is returned. On error, **Problem** is returned, and *error* is set appropriately.

ERRORS

EEXIST

path already exists (not necessarily as a directory).

EFAULT

path points outside your accessible address space.

EACCES

The parent directory does not allow write permission to the process, or one of the directories in *path* did not allow search (execute) permission.

ENAMETOOLONG
path was too long.

ENOENT
A directory component in *path* does not exist or is a dangling symbolic link.

ENOTDIR
A component used as a directory in *path* is not, in fact, a directory.

ENOMEM
Insufficient kernel memory was available.

EROFS
path refers to a file on a read-only filesystem and write access was requested.

ELOOP
path contains a reference to a circular symbolic link, ie a symbolic link whose expansion contains a reference to itself.

ENOSPC
The device containing *path* has no room for the new directory. **ENOSPC** The new directory cannot be created because the user's disk quota is exhausted.

EHOSTS
Can't contact host.

SEE ALSO

DPrmdpdir(2), **DPdirinfo**(2), **DPfiletype**(2), **DPopendir**(3)

NAME

DPopen – Data Parallel stream open functions

SYNOPSIS

#include <constant.h>
#include <type.h>
#include <DPIO.h>

DPSTREAMS *DPopen (CST char *_path_, CST char *_mode_,
CST LOCAL_GRIDS *_lgrid_, SHAPES *_shape_,
TYPES *_type_, ERRNOS *_error_);

DESCRIPTION

The **DPopen** function opens the DPfile whose name is the string pointed to by *path* and associates a DPstream with it. *path* must be visible by the manager process (it does not be to be visible by the application processes). DPopen must be call be all the processes of the *lgrid* with the same parameters.

The argument *mode* points to a string beginning with one of the following sequences:

r Open text file for reading. The stream is positioned at the beginning of the file.

w Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

a Open for writing. The file is created if it does not exist. The stream is positioned at the end of the file.

The argument *lgrid* defined the compute topology and distribution. It fixed the file partionning. A process can only access data chunck that it owns according to the specified compute distribution. This distribution fixes the processes visibility over the file. Each data item is exactly owned by one processor. The distribution specified with the *lgrid* parameter is only used for the application and not for data storage. A same file can be successively opened with two distinct processing distributions. The data location onto the disks only depend on the directory used.

The argument *shape* refer to the array rank and sizes. It's an output parameter when the file is open in reading mode, otherwise it must be given by the application.

type refer to the file array type.

Any created files will have mode **S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH** (0666). as modified by the process' umask value (see **umask**(2)).

RETURN VALUES

Upon successful completion **DPopen** returns a **DPFILE** pointer. Otherwise, **NULL** is returned and the variable *error* is set to indicate the error.

ERRORS

ENODPFILE
The *path* provided to **DPopen**. is invalid.

ETYPE
The *type* provided to **DPopen**. is invalid.

ESHAPE
The *type* provided to **DPopen**. is invalid.

ERGRID
Remote I/O grid not responding.

EFCORRUPT
File corrupted.

EDCORRUPT
DPdevice corrupted.

EHOSTS
Can't contact host. **DPopen** function may also fail and set *error* for any of the errors specified for the routine **fopen**(3).

SEE ALSO

DPclose(3), **DPread**(3), **DPwrite**(3), **DPmalloc**(3), **DPperror**(3), **DPseek**(3)

NAME

DPopendir – open a directory (parallel or not)

SYNOPSIS

#include <constant.h>
#include <type.h>
#include <dirent.h>
#include <DPdirent.h>

DIR *DPopendir(CST char * *dir_name);*

DESCRIPTION

DPreaddir opens a directory stream corresponding to the directory *dir_name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory. **DPopendir**, **DPreaddir**, and **DPclosedir** are provided in order to skip the **DPFS** system files.

RETURN VALUE

DPopendir function returns a pointer to the directory stream or NULL if an error occurred.

ERRORS

EACESS
Permission denied.

EMFILE
Too many file descriptors in use by process.

ENFILE
Too many files are currently open in the system.

ENOENT
Directory does not exist, or *dir_name* is an empty string.

ENOMEM
Insufficient memory to complete the operation.

ENOTDIR
dir_name is not a directory.

SEE ALSO

DPreaddir(3), **DPclosedir**(3), **DPrewinddir**(3)

NAME

DPperror – print a system error message

SYNOPSIS

#include <constant.h>
#include <type.h>
#include <DPerrno.h>

void DPperror (CST ERRNOS * *error*, **CST char *** *msg);*

DESCRIPTION

The routine **perror** produces a message on the standard error output, describing the *error* parameter.

ERRNOS structure is declared as follows:

```
typedef struct {
  VIRTUAL_HOSTS  name[MAX_NODE];
  int            t_errno[MAX_NODE];
  ERROR_TYPES    e_type;
  NATS           nb;
} ERRNOS;
```

When *error->e_type* is **None** or **Scalar**, the argument string *msg* is printed first, then a colon and a blank, then the message and a new-line. When *error->e_type* is **Plural**, the argument string *msg* is printed first, then a colon and a new line. Each error message is then preceded by the name of the virtual I/O node, a colon and a blank.

To be of most use, the argument string should include the name of the function that incurred the error.

NAME

DPread, DPwrite – distributed array stream input/output

SYNOPSIS

#include <constant.h>
#include <type.h>
#include <DPIO.h>

int DPread(DPSTREAMS * *stream*, **char *** *ptr*,
CST NATS *nb*, **ERRNOS *** *error);*

int DPwrite(DPSTREAMS * *stream*, **CST char *** *ptr*,
CST NATS *nb*, **ERRNOS *** *error);*

DESCRIPTION

With the **DPread** and the **DPwrite** functions, each processor access his local data set according to the data distribution specified at opening time. These functions are completly asynchronous but they must be call be all the processes of the local process grid with the same parameters.

The function **DPread** read *nb* arrays from the data parallel *stream*, storing them on the location given by *ptr*.

The function **DPwrite** writes *nb* arrays to the data parallel stream pointed to by *stream*, obtaining them from the location given by *ptr*.

RETURN VALUES

DPread and **DPwrite** return the number of arrays successfully read or written (i.e. not the number of array items). It's only a local result. If an error occurs the local result may be different form one process to an other.

ERRORS

ERGRID
Remote I/O grid not responding.

EHOSTS
Can't contact host.

DPread/DPwrite
function may also fail and set *error* for any of the errors specified for the routine **fread/fwrite(3).**

SEE ALSO

DPseek(3), **DPopen**(3), **DPclose**(3), **DPperror**(3)

NAME

DPopendir – read a directory (parallel or not)

SYNOPSIS

#include <constant.h>
#include <type.h>
#include <dirent.h>
#include <DPdirent.h>

struct dirent *DPreaddir (DIR * *dir_pointer);*

DESCRIPTION

The **DPreaddir** function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to be *dir*. It returns NULL on reaching the end-of-file or if an error occurred.

The data returned by **DPreaddir** is overwritten by subsequent calls to **DPreaddir** for the same directory stream.

According to POSIX, the *dirent* structure contains a field *char d_name[]* of unspecified size, with at most **NAME_MAX** characters preceding the terminating null character. Use of other fields will harm the portability of your programs.

RETURN VALUE

The **DPreaddir** function returns a pointer to a dirent structure, or NULL if an error occurs or end-of-file is reached.

SEE ALSO

DPopendir (3), **DPclosedir** (3), **DPrewinddir** (3)

NAME

DPrewinddir – reset a directory stream (parallel or not)

SYNOPSIS

#include <constant.h>
#include <type.h>
#include <dirent.h>
#include <DPdirent.h>

void DPrewinddir (DIR ***dir_pointer);**

DESCRIPTION

The **DPrewinddir()** function resets the position of the directory stream *DIR* to the beginning of the directory.

RETURN VALUE

The **DPreaddir()** function returns no value.

SEE ALSO

DPopendir(3), **DPreaddir**(3), **DPclosedir**(3)

NAME

DPrmdpdir – delete a directory

SYNOPSIS

#include <unistd.h>
#include <sys/types.h>
#include <constant.h>
#include <type.h>
#include <DPerrno.h>
#include <DPstat.h>

RTS DPrmdpdir(CST char ****path*, ERRNOS ****error***);**

DPrmdpdir deletes a data-parallel directory, which must be empty.

RETURN VALUE

On success, **Ok** is returned. On error, **Problem** is returned, and *errno* is set appropriately.

ERRORS

EPERM The filesystem containing *path* does not support the removal of directories.

EFAULT *path* points outside your accessible address space.

EACCES Write access to the directory containing *path* was not allowed for the process's effective uid, or one of the directories in *path* did not allow search (execute) permission.

EPERM The directory containing *path* has the sticky-bit (**S_ISVTX**) set and the process's effective uid is neither the uid of the file to be deleted nor that of the directory containing it.

ENAMETOOLONG
path was too long.

ENOENT A directory component in *path* does not exist or is a dangling symbolic link.

ENOTDIR
path, or a component used as a directory in *path*, is not, in fact, a directory.

ENOTEMPTY
path contains entries other than . and .. .

EBUSY *path* is the current working directory or root directory of some process.

ENOMEM
Insufficient kernel memory was available.

EROFS *path* refers to a file on a read-only filesystem.

ELOOP *path* contains a reference to a circular symbolic link, ie a symbolic link containing a reference to itself.

EHOSTS Can't contact host.

ENODPDIRECTORY
Not a data-parallel directory

SEE ALSO

DPmkdpdir(2), **DPdirinfo**(2), **DPfiletype**(2), **DPopendir**(3)

NAME

DPrmdpfile – delete a data-parallel file

SYNOPSIS

#include <unistd.h>
#include <sys/types.h>
#include <constant.h>
#include <type.h>
#include <DPerrno.h>
#include <DPstat.h>

RTS DPrmdpfile(CST char **path*, ERRNOS **errors);

DESCRIPTION

DPrmdpfile deletes a data-parallel file.it.

RETURN VALUE

On success, **Ok** is returned. On error, **Problem** is returned, and *errno* is set appropriately.

ERRORS

EFAULT *path* points outside your accessible address space.

EACCES Write access to the directory containing *path* is not allowed for the process's effective uid, or one of the directories in *path* did not allow search (execute) permission.

EPERM The directory containing *path* has the sticky-bit (**S_ISVTX**) set and the process's effective uid is neither the uid of the file to be deleted nor that of the directory containing it.

ENAMETOOLONG
path was too long.

ENOENT A directory component in *path* does not exist or is a dangling symbolic link.

ENOTDIR
A component used as a directory in *path* is not, in fact, a directory.

EISDIR *path* refers to a directory.

ENOMEM
Insufficient kernel memory was available.

EROFS *path* refers to a file on a read-only filesystem. TP **ENODPFILE** *path* is not a data-parallel file.

EFCORRUPT
Data-parallel file corrupted.

EHOSTS Can't contact host.

SEE ALSO

DPrmdpdir(2), **DPdirinfo**(2), **DPfiletype**(2), **DPopendir**(3)

NAME

DPseek – reposition a data parallel stream

SYNOPSIS

#include <constant.h>
#include <type.h>
#include <DPIO.h>

DPseek(DPSTREAMS ****stream*, **CST int *offset*,
CST int *whence*, **ERRNOS ****error*);

DESCRIPTION

The **DPseek** function sets the file position indicator for the stream pointed to by *stream*. The new position, measured in arrays, is obtained by adding *offset* bytes to the position specified by *whence*. If *whence* is set to **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively.

RETURN VALUE

The **DPseek()** function returns no value.

SEE ALSO

DPopen(3), **DPread**(3), **DPclose**(3), seek **(3)"**

NAME

DPutime – change access and/or modification times of DPfile entry

SYNOPSIS

#include <utime.h>
#include <constant.h>
#include <type.h>
#include <DPerrno.h>
#include <DPstat.h>

RTS DPutime (CST char ****path*, CST struct utimbuf ****buf***, ERRNOS ****error***);**

DESCRIPTION

DPutime changes the access and modification times of the inode specified by *path* to the *actime* and *modtime* fields of *buf* respectively. If *buf* is **NULL**, then the access and modification times of the file are set to the current time. The *utimbuf* structure is:

```
struct utimbuf {
        time_t actime;    /* access time */
        time_t modtime;   /* modification time */
};
```

utimes is just a wrapper for **utime**: *tvp*[0].*tv_sec* is *actime*, and *tvp*[1].*tv_sec* is *modtime*. The *timeval* structure is:

```
struct timeval {
        long    tv_sec:             /* seconds */
        long    tv_usec:    /* microseconds */
};
```

RETURN VALUE

On success, **Ok** is returned. On error, **Problem** is returned, and *error* is set appropriately.

ERRORS

EACCESS
Permission to write the file is denied.

ENOENT *filename* does not exist.

ERGRID Remote I/O grid not responding.

EFCORRUPT
File corrupted.

EHOSTS Can't contact host.

SEE ALSO

DPchown(2)

AUTHORS AND CONTRIBUTORS

Based on GNU utime function.

NAME

chgrp – change the group ownership of files

SYNOPSIS

chgrp [–Rcfv] [––recursive] [––changes] [––silent] [––quiet] [––verbose] [––help] [––version] group file...

DESCRIPTION

chgrp changes the group ownership of each given file to the named group, which can be either a group name or a numeric group ID.

OPTIONS

–c, ––changes
: Verbosely describe only files whose ownership actually changes.

–f, ––silent, ––quiet
: Do not print error messages about files whose ownership cannot be changed.

–v, ––verbose
: Verbosely describe ownership changes.

–R, ––recursive
: Recursively change ownership of directories and their contents.

––help
: Print a usage message on standard output and exit successfully.

––version
: Print version information on standard output then exit successfully.

AUTHORS AND CONTRIBUTORS

Based on GNU chgrp command.

NAME

chmod – change the access permissions of files

SYNOPSIS

chmod [–Rcfv] [––recursive] [––changes] [––silent] [––quiet] [––verbose] [––help] [––version] mode file...

DESCRIPTION

chmod changes the permissions of each given file according to *mode*, which can be either a symbolic representation of changes to make, or an octal number representing the bit pattern for the new permissions.

The format of a symbolic mode is '[ugoa...][[+-=][rwxXstugo...]...][....]'. Multiple symbolic operations can be given, separated by commas.

A combination of the letters 'ugoa' controls which users' access to the file will be changed: the user who owns it (u), other users in the file's group (g), other users not in the file's group (o), or all users (a). If none of these are given, the effect is as if 'a' were given, but bits that are set in the umask are not affected.

The operator '+' causes the permissions selected to be added to the existing permissions of each file; '-' causes them to be removed; and '=' causes them to be the only permissions that the file has.

The letters 'rwxXstugo' select the new permissions for the affected users: read (r), write (w), execute (or access for directories) (x), execute only if the file is a directory or already has execute permission for some user (X), set user or group ID on execution (s), save program text on swap device (t), the permissions that the user who owns the file currently has for it (u), the permissions that other users in the file's group have for it (g), and the permissions that other users not in the file's group have for it (o).

A numeric mode is from one to four octal digits (0-7), derived by adding up the bits with values 4, 2, and 1. Any omitted digits are assumed to be leading zeros. The first digit selects the set user ID (4) and set group ID (2) and save text image (1) attributes. The second digit selects permissions for the user who owns the file: read (4), write (2), and execute (1); the third selects permissions for other users in the file's group, with the same values; and the fourth for other users not in the file's group, with the same values.

chmod never changes the permissions of symbolic links; the **chmod** system call cannot change their permissions. This is not a problem since the permissions of symbolic links are never used. However, for each symbolic link listed on the command line, **chmod** changes the permissions of the pointed-to file. In contrast, **chmod** ignores symbolic links encountered during recursive directory traversals.

OPTIONS

–c, ––changes
: Verbosely describe only files whose permissions actually change.

–f, ––silent, ––quiet
: Do not print error messages about files whose permissions cannot be changed.

–v, ––verbose
: Verbosely describe changed permissions.

–R, ––recursive
: Recursively change permissions of directories and their contents.

––help
: Print a usage message on standard output and exit successfully.

––version
: Print version information on standard output then exit successfully.

AUTHORS AND CONTRIBUTORS

Based on GNU chmod command.

NAME

chown – change the user and group ownership of files

SYNOPSIS

chown [–Rcfv] [––recursive] [––changes] [––help] [––version] [––silent] [––quiet] [––verbose] [user][:.][group] file...

DESCRIPTION

chown changes the user and/or group ownership of each given file, according to its first non-option argument, which is interpreted as follows. If only a user name (or numeric user ID) is given, that user is made the owner of each given file, and the files' group is not changed. If the user name is followed by a colon or dot and a group name (or numeric group ID), with no spaces between them, the group ownership of the files is changed as well. If a colon or dot but no group name follows the user name, that user is made the owner of the files and the group of the files is changed to that user's login group. If the colon or dot and group are given, but the user name is omitted, only the group of the files is changed; in this case, **chown** performs the same function as **chgrp**.

OPTIONS

–c, ––changes
: Verbosely describe only files whose ownership actually changes.

–f, ––silent, ––quiet
: Do not print error messages about files whose ownership cannot be changed.

–v, ––verbose
: Verbosely describe ownership changes.

–R, ––recursive
: Recursively change ownership of directories and their contents.

––help
: Print a usage message on standard output and exit successfully.

––version
: Print version information on standard output then exit successfully.

AUTHORS AND CONTRIBUTORS

Based on GNU chown command.

NAME

cp – copy files (parallel or not)

SYNOPSIS

cp [options] source dest
cp [options] source {source} directory
Options:
[–abdfilprsuvxPR] [–S backup-suffix] [–V {numbered,existing,simple}] [––backup] [––no-dereference] [––force] [––interactive] [––one-file-system] [––preserve] [––recursive] [––update] [––verbose] [––suffix=backup-suffix] [––version-control={numbered,existing,simple}] [––archive] [––parents] [––link] [––symbolic-link] [––help] [––version]

DESCRIPTION

If only two files are given, **cp** copies the first onto the second. A file entry can only be overwitten by a file entry of the same type. A directory cannot overwrite a file. A file cannot overwrite a DPfile... If the last argument names an existing directory, **cp** copies each other given file into a file with the same name in that directory. An error is produced if the last argument is not a directory and more than two files are given. By default, cp does not copy directories. New created DPfiles are distributed onto the DPdevice associated to the target DPdirectory.

OPTIONS

–a, ––archive
Preserve as much as possible of the structure and attributes of the original files in the copy. The same as *–dpR*.

–b, ––backup
Make backups of files that are about to be overwritten or removed. Only works with regular entries.

–d, ––no-dereference
Copy symbolic links as symbolic links rather than copying the files that they point to, and preserve hard link relationships between source files in the copies. This option produce an error message on DPfiles and DPdirectories.

–f, ––force
Remove existing destination files.

–i, ––interactive
Prompt whether to overwrite existing regular destination files.

–l, ––link
Make hard links instead of copies of non-directories. Link are not supported on parallel entries.

–P, ––parents
Form the name of each destination file by appending to the target directory a slash and the specified name of the source file. The last argument given to **cp** must be the name of an existing directory. For example, the command `cp ––parents a/b/c existing_dir` copies the file *a/b/c* to *existing_dir/a/b/c*, creating any missing intermediate directories. Source and target DPdirectories are associated to the same DPdevices.

–p, ––preserve
Preserve the original files' owner, group, permissions, and timestamps.

–r Copy directories recursively. Source and target DPdirectories are associated to the same DPdevices. Source and target DPfiles are distributed on the same I/O nodes with the same distribution.

–s, ––symbolic-link
Make symbolic links instead of copies of non-directories. All source filenames must be absolute (starting with '/') unless the destination files are in the current directory. This option produces an error message on DPentries.

-u, --update
Do not copy a nondirectory that has an existing destination with the same or newer modification time.

-v, --verbose
Print the name of each file before copying it.

-x, --one-file-system
Skip subdirectories that are on different filesystems from the one that the copy started on.

-R, --recursive
Copy directories recursively.

--help
Print a usage message on standard output and exit successfully.

--version
Print version information on standard output then exit successfully.

-S, --suffix backup-suffix
The suffix used for making simple backup files can be set with the **SIMPLE_BACKUP_SUFFIX** environment variable, which can be overridden by this option. If neither of those is given, the default is "~", as it is in Emacs. Not supported for parallel entries.

AUTHORS AND CONTRIBUTORS
Based on GNU cp command.

NAME

ls

SYNOPSIS

ls [−abcdfgiklmnpqrstuxABCFGLNQRSUX1] [−w cols] [−T cols] [−I pattern] [−−all] [−−escape] [−−directory] [−−inode] [−−kilobytes] [−−numeric-uid-gid] [--no-group] [−−hide-control-chars] [−−reverse] [−−size] [−−width=cols] [−−tabsize=cols] [−−almost-all] [−−ignore-backups] [−−classify] [−−file-type] [−−full-time] [−−ignore=pattern] [−−dereference] [−−literal] [−−quote-name] [−−recursive] [−−sort={none,time,size,extension}] [−−format={long,verbose,commas,across,vertical,single-column}] [−−time={atime,access,use,ctime,status}] [−−help] [−−version] [−−color[={yes,no,tty}]] [−−colour[={yes,no,tty}]] [name...]

DESCRIPTION

ls list each given file or directory name. Directory contents are sorted alphabetically. Files are by default listed in columns, sorted vertically, if the standard output is a terminal; otherwise they are listed one per line. For

OPTIONS

−a, −−all

List all files in directories, including all files that start with '.'.

−b, −−escape

Quote nongraphic characters in file names using alphabetic and octal backslash sequences like those used in C.

−c, −−time=ctime, −−time=status

Sort directory contents according to the files' status change time instead of the modification time. If the long listing format is being used, print the status change time instead of the modification time.

−d, −−directory

List directories like other files, rather than listing their contents.

−f Do not sort directory contents; list them in whatever order they are stored on the disk. The same as enabling *−a* and *−U* and disabling *−l, −s,* and *−t.*

−−full-time

List times in full, rather than using the standard abbreviation heuristics.

−g Ignored; for Unix compatibility.

−i, −−inode

Print the index number of each file to the left of the file name.

−k, −−kilobytes

If file sizes are being listed, print them in kilobytes. This overrides the environment variable POSIXLY_CORRECT.

−l, −−format=long, −−format=verbose

In addition to the name of each file, print the file type, permissions, number of hard links, owner name, group name, size in bytes, and timestamp (the modification time unless other times are selected). For files with a time that is more than 6 months old or more than 1 hour into the future, the timestamp contains the year instead of the time of day. The mode displayed with the -l flag is interpreted by the first character, as follows:

b Block special file

c Character special file

d Directory

D Data-parallel directory

l Symbolic link

p First-In-First-Out (FIFO) special file

s Local socket

– Ordinary file

F Data-parallel file

The next nine characters are divided into three sets of three characters each. The first three characters show the owner's permission. The next set of three characters show the permission of the other users in the group. The last set of three characters show the permission of everyone else. The three characters in each set show read, write and execute permission of the file. Execute permission of a directory lets you search a directory for a specified file.

−m, −−format=commas
List files horizontally, with as many as will fit on each line, separated by commas.

−n, −−numeric-uid-gid
List the numeric UID and GID instead of the names.

−p Append a character to each file name indicating the file type.

−q, −−hide-control-chars
Print question marks instead of nongraphic characters in file names.

−r, −−reverse
Sort directory contents in reverse order.

−s, −−size
Print the size of each file in 1K blocks to the left of the file name. If the environment variable POSIXLY_CORRECT is set, 512-byte blocks are used instead.

−t, −−sort=time
Sort directory contents by timestamp instead of alphabetically, with the newest files listed first.

−u, −−time=atime, −−time=access, −−time=use
Sort directory contents according to the files' last access time instead of the modification time. If the long listing format is being used, print the last access time instead of the modification time.

−x, −−format=across, −−format=horizontal
List the files in columns, sorted horizontally.

−A, −−almost-all
List all files in directories, except for '.' and '..'.

−B, −−ignore-backups
Do not list files that end with '~', unless they are given on the command line.

−C, −−format=vertical
List files in columns, sorted vertically.

−F, −−classify
Append a character to each file name indicating the file type. For regular files that are executable, append a '*'. The file type indicators are '/' for directories, '@' for symbolic links, '|' for FIFOs, '=' for sockets, and nothing for regular files.

−G, −−no−group
Inhibit display of group information in a long format directory listing.

−L, −−dereference
List the files linked to by symbolic links instead of listing the contents of the links.

−N, −−literal
Do not quote file names.

−Q, −−quote-name
Enclose file names in double quotes and quote nongraphic characters as in C.

−R, −−recursive
List the contents of all directories recursively.

−S, −−sort=size
Sort directory contents by file size instead of alphabetically, with the largest files listed first.

−U, −−sort=none
Do not sort directory contents; list them in whatever order they are stored on the disk. This option is not called *−f* because the Unix **ls** *−f* option also enables *−a* and disables *−l*, *−s*, and *−t*. It seems useless and ugly to group those unrelated things together in one option. Since this option doesn't do that, it has a different name.

−X, −−sort=extension
Sort directory contents alphabetically by file extension (characters after the last '.'); files with no extension are sorted first.

−1, −−format=single-column
List one file per line.

−w, −−width cols
Assume the screen is *cols* columns wide. The default is taken from the terminal driver if possible; otherwise the environment variable **COLUMNS** is used if it is set; otherwise the default is 80.

−T, −−tabsize cols
Assume that each tabstop is *cols* columns wide. The default is 8.

−I, −−ignore pattern
Do not list files whose names match the shell pattern *pattern* unless they are given on the command line. As in the shell, an initial '.' in a filename does not match a wildcard at the start of *pattern*.

−−color, −−colour, −−color=yes, −−colour=yes
Colorize the names of files depending on the type of file. See **DISPLAY COLORIZATION** below.

−−color=tty, −−colour=tty
Same as *−−color* but only if standard output is a terminal. This is very useful for shell scripts and command aliases, especially if your favorite pager does not support color control codes.

−−color=no, −−colour=no
Disables colorization. This is the default. Provided to override a previous color option.

−−help
Print a usage message on standard output and exit successfully.

−−version
Print version information on standard output then exit successfully.

DISPLAY COLORIZATION

When using the *--color* option, this version of **ls** will colorize the file names printed according to the name and type of file. By default, this colorization is by type only, and the codes used are ISO 6429 (ANSI) compliant.

You can override the default colors by defining the environment variable **LS_COLORS** (or **LS_COLOURS**). The format of this variable is reminicent of the termcap(5) file format: a colon-separated list of expressions of the form "xx=string", where "xx" is a two-character variable name. The variables with their associated defaults are:

no	0	Normal (non-filename) text
fi	0	Regular file
di	32	Directory
ln	36	Symbolic link
pi	31	Named pipe (FIFO)
so	33	Socket
bd	44;37	Block device
cd	44;37	Character device
ex	35	Executable file
mi	(none)	Missing file (defaults to fi)
or	(none)	Orphanned symbolic link (defaults to ln)
lc	\e[	Left code
rc	m	Right code
ec	(none)	End code (replaces lc+no+rc)

You only need to include the variables you want to change from the default.

File names can also be colorized based on filename extension. This is specified in the LS_COLORS variable using the syntax "*ext=string". For example, using ISO 6429 codes, to color all C-language source files blue you would specify "*.c=34". This would color all files ending in .c in blue (34) color.

Control characters can be written either in C-style \-escaped notation, or in **stty**-like ^-notation. The C-style notation adds \e for Escape, _ for a normal space characer, and \? for Delete. In addition, the \ escape character can be used to override the default interpretation of \, ^, : and =.

Each file will be written as <lc> <color code> <rc> <filename> <ec>. If the <ec> code is undefined, the sequence <lc> <no> <rc> will be used instead. This is generally more convenient to use, but less general. The left, right and end codes are provided so you don't have to type common parts over and over again and to support weird terminals; you will generally not need to change them at all unless your terminal does not use ISO 6429 color sequences but a different system.

If your terminal does use ISO 6429 color codes, you can compose the type codes (i.e. all except the **lc**, **rc**, and **ec** codes) from numerical commands separated by semicolons. The most common commands are:

0	to restore default color
1	for brighter colors
4	for underlined text
5	for flashing text
30	for black foreground
31	for red foreground
32	for green foreground
33	for yellow (or brown) foreground
34	for blue foreground
35	for purple foreground
36	for cyan foreground

37 for white (or gray) foreground
40 for black background
41 for red background
42 for green background
43 for yellow (or brown) background
44 for blue background
45 for purple background
46 for cyan background
47 for white (or gray) background

Not all commands will work on all systems or display devices.

A few terminal programs do not recognize the default end code properly. If all text gets colorized after you do a directory listing, try changing the **no** and **fi** codes from 0 to the numerical codes for your standard fore- and background colors.

BUGS

On BSD systems, the *−s* option reports sizes that are half the correct values for files that are NFS-mounted from HP-UX systems. On HP-UX systems, it reports sizes that are twice the correct values for files that are NFS-mounted from BSD systems. This is due to a flaw in HP-UX; it also affects the HP-UX **ls** program.

If there was a single standard for the English language it would not be necessary to support redundant spellings.

AUTHORS AND CONTRIBUTORS

Based on GNU cp command.

NAME

mkdpdir – make data-parallel directories

SYNOPSIS

mkdpdir dpdevice_file dpdir_name

DESCRIPTION

This manual page documents the **mkdpdir** command. **mkdpdir** creates a data-parallel directory with the given name. By default, the mode of created directories is 0777 minus the bits set in the umask.

dpdevice specification example:

we declare a 3*2 virtual I/O node topology

TOPOLOGY

(

((node000 node001) (node010 node011))
((node100 node101) (node110 node111))
((node200 node201) (node210 node211))

)

The virtual I/O nodes must refer to physical nodes

NODE

node000	farm1-giga	ALPHA	/tmp/node000
node001	farm1-giga	ALPHA	/tmp/node001
node010	farm2-giga	ALPHA	/tmp/node010
node011	farm2-giga	ALPHA	/tmp/node011
node100	farm3-giga	ALPHA	/tmp/node100
node101	farm3-giga	ALPHA	/tmp/node101
node110	farm4-giga	ALPHA	/tmp/node110
node111	farm4-giga	ALPHA	/tmp/node111
node200	farm5-giga	ALPHA	/tmp/node200
node201	farm5-giga	ALPHA	/tmp/node201
node210	farm6-giga	ALPHA	/tmp/node210
node211	farm6-giga	ALPHA	/tmp/node211

and finally, the distribution for each dimension.

DISTRIBUTION

BLOCK CYCLIC 16 BLOCK

AUTHORS AND CONTRIBUTORS

Based on GNU ls command.

NAME

mv – move files from one directory (parallel or not) to an other

SYNOPSIS

mv [options] source dest
mv [options] source {source} directory
Options:
[–bfiuv] [–S backup-suffix] [–V {numbered,existing,simple}] [––backup] [––force] [––interactive] [––update] [––verbose] [––suffix=backup-suffix] [––version-control={numbered,existing,simple}] [––help] [––version]

DESCRIPTION

This manual page documents the **mv** command. If the last argument names an existing directory, **mv** moves each other given file into a file with the same name in that directory. Otherwise, if only two files are given, it moves the first onto the second. An error is produce if the last argument is not a directory and more than two files are given.

mv command is able to move a DPfile from one parallel computer to an other. Arrays are redistributed from the source topology to the target topology according to the DPdirectories distributions. Binary representation is also modified to match the binary format of the target architecture.

Regular file entries cannot overwrite or being overwritten by data-parallel entries.

If a destination file is unwritable, the standard input is a tty, and the *–f* or *––force* option is not given, **mv** prompts the user for whether to overwrite the file. If the response does not begin with 'y' or 'Y', the file is skipped.

OPTIONS

–b. ––backup
Make backups of files that are about to be removed.

–f. ––force
Remove existing destination files and never prompt the user.

–i. ––interactive
Prompt whether to overwrite each destination file that already exists. If the response does not begin with 'y' or 'Y', the file is skipped.

–u. ––update
Do not move a nondirectory that has an existing destination with the same or newer modification time.

–v. ––verbose
Print the name of each file before moving it.

––help
Print a usage message on standard output and exit successfully.

––version
Print version information on standard output then exit successfully.

–S. ––suffix backup-suffix
The suffix used for making simple backup files can be set with the **SIMPLE_BACKUP_SUFFIX** environment variable, which can be overridden by this option. If neither of those is given, the default is '~', as it is in Emacs.

–V. ––version-control {numbered.existing.simple}
The type of backups made can be set with the **VERSION_CONTROL** environment variable, which can be overridden by this option. If **VERSION_CONTROL** is not set and this option is not given, the default backup type is 'existing'. The value of the **VERSION_CONTROL** environment variable and the argument to this option are like the

GNU Emacs 'version-control' variable; they also recognize synonyms that are more descriptive. The valid values are (unique abbreviations are accepted):

't' or 'numbered'
: Always make numbered backups.

'nil' or 'existing'
: Make numbered backups of files that already have them, simple backups of the others.

'never' or 'simple'
: Always make simple backups.

AUTHORS AND CONTRIBUTORS

Based on GNU mv command.

NAME

rm – remove files (parallel or not)

SYNOPSIS

rm [–dfirvR] [––directory] [––force] [––interactive] [––recursive] [––help] [––version] [––verbose] name...

DESCRIPTION

rm removes each specified file. By default, it does not remove directories.

If a file is unwritable, the standard input is a tty, and the *–f* or *––force* option is not given, **rm** prompts the user for whether to remove the file. If the response does not begin with 'y' or 'Y', the file is skipped.

OPTIONS

–d. ––directory

Remove directories with 'unlink' instead of 'rmdir', and don't require a directory to be empty before trying to unlink it. Only works for the super-user. Because unlinking a directory causes any files in the deleted directory to become unreferenced. it is wise to **fsck** the filesystem after doing this. This option is not avalaible for DPdirectories.

–f. ––force

Ignore nonexistent files and never prompt the user.

–i. ––interactive

Prompt whether to remove each file. If the response does not begin with 'y' or 'Y', the file is skipped.

–r. –R. ––recursive

Remove the contents of directories recursively.

–v. ––verbose

Print the name of each file before removing it.

––help

Print a usage message on standard output and exit successfully.

––version

Print version information on standard output then exit successfully.

AUTHORS AND CONTRIBUTORS

Based on GNU rm command.

NAME

rmdir – remove empty directories (parallel or not)

SYNOPSIS

rmdir [–p] [––parents] [––help] [––version] dir...

DESCRIPTION

rmdir removes each given empty directory. If any non-option argument does not refer to an existing empty directory, it is an error.

OPTIONS

–p, ––parents

Remove any parent directories that are explicitly mentioned in an argument, if they become empty after the argument file is removed.

––help

Print a usage message on standard output and exit successfully.

––version

Print version information on standard output then exit successfully.

AUTHORS AND CONTRIBUTORS

Based on GNU rmdir command.

Annexe C

```
      PROGRAM PDSCAEX
*
*  -- ScaLAPACK example code --
*     University of Tennessee, Knoxville, Oak Ridge National Laboratory,
*     and University of California, Berkeley.
*
*     Written by Antoine Petitet, August 1995 (petitet@cs.utk.edu)
*
*     This program solves a linear system by calling the ScaLAPACK
*     routine PDGESV. The input matrix and right-and-sides are
*     read from a file. The solution is written to a file.
*
*     .. Parameters ..
      INTEGER            DBLESZ, INTGSZ, MEMSIZ, TOTMEM
      PARAMETER          ( DBLESZ = 8, INTGSZ = 4, TOTMEM = 20000,
     $                     MEMSIZ = TOTMEM / DBLESZ )
      INTEGER            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DT_,
     $                   LLD_, MB_, M_, NB_, N_, RSRC_
      PARAMETER          ( BLOCK_CYCLIC_2D = 1, DLEN_ = 9, DT_ = 1,
     $                     CTXT_ = 2, M_ = 3, N_ = 4, MB_ = 5, NB_ = 6,
     $                     RSRC_ = 7, CSRC_ = 8, LLD_ = 9 )
      DOUBLE PRECISION   ONE
      PARAMETER          ( ONE = 1.0D+0 )
*     ..
*     .. Local Scalars ..
      CHARACTER*80       OUTFILE
      INTEGER            IAM, ICTXT, INFO, IPA, IPACPY, IPB, IPPIV, IPX,
     $                   IPW, LIPIV, MYCOL, MYROW, N, NB, NOUT, NPCOL,
     $                   NPROCS, NPROW, NP, NQ, NQRHS, NRHS, WORKSIZ
      DOUBLE PRECISION   ANORM, BNORM, EPS, XNORM, RESID
*     ..
*     .. Local Arrays ..
      INTEGER            DESCA( DLEN_ ), DESCB( DLEN_ ), DESCX( DLEN_ )
      DOUBLE PRECISION   MEM( MEMSIZ )
*     ..
```

```
*     .. External Subroutines ..
      EXTERNAL           BLACS_EXIT, BLACS_GET, BLACS_GRIDEXIT,
     $                   BLACS_GRIDINFO, BLACS_GRIDINIT, BLACS_PINFO,
     $                   DESCINIT, IGSUM2D, PDSCAEXINFO, PDGESV,
     $                   PDGEMM, PDLACPY, PDLAPRNT, PDLAREAD, PDLAWRITE
*     ..
*     .. External Functions ..
      INTEGER            ICEIL, NUMROC
      DOUBLE PRECISION   PDLAMCH, PDLANGE
      EXTERNAL           ICEIL, NUMROC, PDLAMCH, PDLANGE
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          DBLE, MAX
*     ..
*     .. Executable Statements ..
*
*     Get starting information
*
      CALL BLACS_PINFO( IAM, NPROCS )
      CALL PDSCAEXINFO( OUTFILE, NOUT, N, NRHS, NB, NPROW, NPCOL, MEM,
     $                  IAM, NPROCS )
*
*     Define process grid
*
      CALL BLACS_GET( -1, 0, ICTXT )
      CALL BLACS_GRIDINIT( ICTXT, 'Row-major', NPROW, NPCOL )
      CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )

      WRITE( NOUT, FMT = * )
     $         'BLACS init est termine '


*
*     Go to bottom of process grid loop if this case doesn't use my
*     process
*
      IF( MYROW.GE.NPROW .OR. MYCOL.GE.NPCOL )
     $   GO TO 20
*
      NP    = NUMROC( N, NB, MYROW, 0, NPROW )
      NQ    = NUMROC( N, NB, MYCOL, 0, NPCOL )
      NQRHS = NUMROC( NRHS, NB, MYCOL, 0, NPCOL )
*
*     Initialize the array descriptor for the matrix A and B
*
      CALL DESCINIT( DESCA, N, N, NB, NB, 0, 0, ICTXT, MAX( 1, NP ),
     $               INFO )
```

```
      CALL DESCINIT( DESCB, N, NRHS, NB, NB, 0, 0, ICTXT, MAX( 1, NP ),
     $               INFO )
      CALL DESCINIT( DESCX, N, NRHS, NB, NB, 0, 0, ICTXT, MAX( 1, NP ),
     $               INFO )
*
*     Assign pointers into MEM for SCALAPACK arrays, A is
*     allocated starting at position MEM( 1 )
*
      WRITE( NOUT, FMT = * )
     $         'Attention allocation de pointeur '


      IPA = 1
      IPACPY = IPA + DESCA( LLD_ )*NQ
      IPB = IPACPY + DESCA( LLD_ )*NQ
      IPX = IPB + DESCB( LLD_ )*NQRHS
      IPPIV = IPX + DESCB( LLD_ )*NQRHS
      LIPIV = ICEIL( INTGSZ*( NP+NB ), DBLESZ )
      IPW = IPPIV + MAX( NP, LIPIV )
*
      WORKSIZ = NB
*
*     Check for adequate memory for problem size
*
      INFO = 0
      IF( IPW+WORKSIZ.GT.MEMSIZ ) THEN
         IF( IAM.EQ.0 )
     $      WRITE( NOUT, FMT = 9998 ) 'test', ( IPW+WORKSIZ )*DBLESZ
         INFO = 1
      END IF
*
*     Check all processes for an error
*
      CALL IGSUM2D( ICTXT, 'All', ' ', 1, 1, INFO, 1, -1, 0 )
      IF( INFO.GT.0 ) THEN
         IF( IAM.EQ.0 )
     $      WRITE( NOUT, FMT = 9999 ) 'MEMORY'
         GO TO 10
      END IF
*
*     Read from file and distribute matrices A and B
*

      WRITE( NOUT, FMT = * )
     $         'lecture des matrices'
```

```
      CALL PDLAREAD( 'SCAEXMAT.dat', MEM( IPA ), DESCA, 0, 0,
     $               MEM( IPW ) )
      CALL PDLAREAD( 'SCAEXRHS.dat', MEM( IPB ), DESCB, 0, 0,
     $               MEM( IPW ) )
*
*     Make a copy of A and the rhs for checking purposes
*
      WRITE( NOUT, FMT = * )
     $         'recopie des matrices'


      CALL PDLACPY( 'All', N, N, MEM( IPA ), 1, 1, DESCA,
     $              MEM( IPACPY ), 1, 1, DESCA )
      CALL PDLACPY( 'All', N, NRHS, MEM( IPB ), 1, 1, DESCB,
     $              MEM( IPX ), 1, 1, DESCX )



      WRITE( NOUT, FMT = * )
     $         'On commence le traitement'


*
*     Call ScaLAPACK PDGESV routine
*
      IF( IAM.EQ.0 ) THEN
         WRITE( NOUT, FMT = * )
         WRITE( NOUT, FMT = * )
     $         '*********************************************'
         WRITE( NOUT, FMT = * )
     $         'Example of ScaLAPACK routine call: (PDGESV)'
         WRITE( NOUT, FMT = * )
     $         '*********************************************'
         WRITE( NOUT, FMT = * )
         WRITE( NOUT, FMT = * ) 'A * X = B, Matrix A:'
         WRITE( NOUT, FMT = * )
      END IF
      CALL PDLAPRNT( N, N, MEM( IPA ), 1, 1, DESCA, 0, 0,
     $               'A', NOUT, MEM( IPW ) )
      IF( IAM.EQ.0 ) THEN
         WRITE( NOUT, FMT = * )
         WRITE( NOUT, FMT = * ) 'Matrix B:'
         WRITE( NOUT, FMT = * )
      END IF
```

```
      CALL PDLAPRNT( N, NRHS, MEM( IPB ), 1, 1, DESCB, 0, 0,
     $               'B', NOUT, MEM( IPW ) )
*
      CALL PDGESV( N, NRHS, MEM( IPA ), 1, 1, DESCA, MEM( IPPIV ),
     $             MEM( IPB ), 1, 1, DESCB, INFO )
*
      IF( MYROW.EQ.0 .AND. MYCOL.EQ.0 ) THEN
         WRITE( NOUT, FMT = * )
         WRITE( NOUT, FMT = * ) 'INFO code returned by PDGESV = ', INFO
         WRITE( NOUT, FMT = * )
         WRITE( NOUT, FMT = * ) 'Matrix X = A^{-1} * B'
         WRITE( NOUT, FMT = * )
      END IF
      CALL PDLAPRNT( N, NRHS, MEM( IPB ), 1, 1, DESCB, 0, 0, 'X', NOUT,
     $               MEM( IPW ) )
      CALL PDLAWRITE( 'SCAEXSOL.dat', N, NRHS, MEM( IPB ), 1, 1, DESCB,
     $                0, 0, MEM( IPW ) )
*
*     Compute residual ||A * X  - B|| / ( ||X|| * ||A|| * eps * N )
*
      EPS = PDLAMCH( ICTXT, 'Epsilon' )
      ANORM = PDLANGE( 'I', N, N, MEM( IPA ), 1, 1, DESCA, MEM( IPW ) )
      BNORM = PDLANGE( 'I', N, NRHS, MEM( IPB ), 1, 1, DESCB,
     $                 MEM( IPW ) )
      CALL PDGEMM( 'No transpose', 'No transpose', N, NRHS, N, ONE,
     $             MEM( IPACPY ), 1, 1, DESCA, MEM( IPB ), 1, 1, DESCB,
     $             -ONE, MEM( IPX ), 1, 1, DESCX )
      XNORM = PDLANGE( 'I', N, NRHS, MEM( IPX ), 1, 1, DESCX,
     $                 MEM( IPW ) )
      RESID = XNORM / ( ANORM * BNORM * EPS * DBLE( N ) )
*
      IF( MYROW.EQ.0 .AND. MYCOL.EQ.0 ) THEN
         WRITE( NOUT, FMT = * )
         WRITE( NOUT, FMT = * )
     $      '||A * X  - B|| / ( ||X|| * ||A|| * eps * N ) = ', RESID
         WRITE( NOUT, FMT = * )
         IF( RESID.LT.10.0D+0 ) THEN
            WRITE( NOUT, FMT = * ) 'The answer is correct.'
         ELSE
            WRITE( NOUT, FMT = * ) 'The answer is suspicious.'
         END IF
      END IF
*
   10 CONTINUE
*
      CALL BLACS_GRIDEXIT( ICTXT )
*
```

```
   20 CONTINUE
*
*     Print ending messages and close output file
*
      IF( IAM.EQ.0 ) THEN
         WRITE( NOUT, FMT = * )
         WRITE( NOUT, FMT = * )
         WRITE( NOUT, FMT = 9997 )
         WRITE( NOUT, FMT = * )
         IF( NOUT.NE.6 .AND. NOUT.NE.0 )
     $      CLOSE ( NOUT )
      END IF
*
      CALL BLACS_EXIT( 0 )
*
 9999 FORMAT( 'Bad ', A6, ' parameters: going on to next test case.' )
 9998 FORMAT( 'Unable to perform ', A, ': need TOTMEM of at least',
     $        I11 )
 9997 FORMAT( 'END OF TESTS.' )
*
      STOP
*
*     End of PDSCAEX
*
      END



      SUBROUTINE PDSCAEXINFO( SUMMRY, NOUT, N, NRHS, NB, NPROW, NPCOL,
     $                        WORK, IAM, NPROCS )
*
*  -- ScaLAPACK example code --
*     University of Tennessee, Knoxville, Oak Ridge National Laboratory,
*     and University of California, Berkeley.
*
*     Written by Antoine Petitet, August 1995  (petitet@cs.utk.edu)
*
*     This program solves a linear system by calling the ScaLAPACK
*     routine PDGESV. The input matrix and right-and-sides are
*     read from a file. The solution is written to a file.
*
*     .. Scalar Arguments ..
      CHARACTER*( * )    SUMMRY
      INTEGER            IAM, N, NRHS, NB, NOUT, NPCOL, NPROCS, NPROW
*     ..
*     .. Array Arguments ..
```

```
      INTEGER            WORK( * )
*     ..
*
* =====================================================================
*
*     .. Parameters ..
      INTEGER            NIN
      PARAMETER          ( NIN = 11 )
*     ..
*     .. Local Scalars ..
      CHARACTER*79       USRINFO
      INTEGER            ICTXT
*     ..
*     .. External Subroutines ..
      EXTERNAL           BLACS_ABORT, BLACS_GET, BLACS_GRIDEXIT,
     $                   BLACS_GRIDINIT, BLACS_SETUP, IGEBR2D, IGEBS2D
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          MAX, MIN
*     ..
*     .. Executable Statements ..
*
*     Process 0 reads the input data, broadcasts to other processes and
*     writes needed information to NOUT
*
      IF( IAM.EQ.0 ) THEN
*
*        Open file and skip data file header
*
         OPEN( NIN, FILE='SCAEX.dat', STATUS='OLD' )
         READ( NIN, FMT = * ) SUMMRY
         SUMMRY = ' '
*
*        Read in user-supplied info about machine type, compiler, etc.
*
         READ( NIN, FMT = 9999 ) USRINFO
*
*        Read name and unit number for summary output file
*
         READ( NIN, FMT = * ) SUMMRY
         READ( NIN, FMT = * ) NOUT
         IF( NOUT.NE.0 .AND. NOUT.NE.6 )
     $      OPEN( NOUT, FILE = SUMMRY, STATUS = 'UNKNOWN' )
*
*        Read and check the parameter values for the tests.
*
*        Get matrix dimensions
```

```
*
      READ( NIN, FMT = * ) N
      READ( NIN, FMT = * ) NRHS
*
*     Get value of NB
*
      READ( NIN, FMT = * ) NB
*
*     Get grid shape
*
      READ( NIN, FMT = * ) NPROW
      READ( NIN, FMT = * ) NPCOL
*
*     Close input file
*
      CLOSE( NIN )
*
*     If underlying system needs additional set up, do it now
*
      IF( NPROCS.LT.1 ) THEN
         NPROCS = NPROW * NPCOL
         CALL BLACS_SETUP( IAM, NPROCS )
      END IF
*
*     Temporarily define blacs grid to include all processes so
*     information can be broadcast to all processes
*
      CALL BLACS_GET( -1, 0, ICTXT )
      CALL BLACS_GRIDINIT( ICTXT, 'Row-major', 1, NPROCS )
*
*     Pack information arrays and broadcast
*
      WORK( 1 ) = N
      WORK( 2 ) = NRHS
      WORK( 3 ) = NB
      WORK( 4 ) = NPROW
      WORK( 5 ) = NPCOL
      CALL IGEBS2D( ICTXT, 'All', ' ', 5, 1, WORK, 5 )
*
*     regurgitate input
*
      WRITE( NOUT, FMT = 9999 )
     $             'SCALAPACK example driver.'
      WRITE( NOUT, FMT = 9999 ) USRINFO
      WRITE( NOUT, FMT = * )
      WRITE( NOUT, FMT = 9999 )
     $             'The matrices A and B are read from '//
```

```
     $              'a file.'
            WRITE( NOUT, FMT = * )
            WRITE( NOUT, FMT = 9999 )
     $              'An explanation of the input/output '//
     $              'parameters follows:'
*
            WRITE( NOUT, FMT = 9999 )
     $              'N       : The order of the matrix A.'
            WRITE( NOUT, FMT = 9999 )
     $              'NRHS    : The number of right and sides.'
            WRITE( NOUT, FMT = 9999 )
     $              'NB      : The size of the square blocks the'//
     $              ' matrices A and B are split into.'
            WRITE( NOUT, FMT = 9999 )
     $              'P       : The number of process rows.'
            WRITE( NOUT, FMT = 9999 )
     $              'Q       : The number of process columns.'
            WRITE( NOUT, FMT = * )
            WRITE( NOUT, FMT = 9999 )
     $              'The following parameter values will be used:'
            WRITE( NOUT, FMT = 9998 ) 'N    ', N
            WRITE( NOUT, FMT = 9998 ) 'NRHS ', NRHS
            WRITE( NOUT, FMT = 9998 ) 'NB   ', NB
            WRITE( NOUT, FMT = 9998 ) 'P    ', NPROW
            WRITE( NOUT, FMT = 9998 ) 'Q    ', NPCOL
            WRITE( NOUT, FMT = * )
*
         ELSE
*
*           If underlying system needs additional set up, do it now
*
            IF( NPROCS.LT.1 )
     $         CALL BLACS_SETUP( IAM, NPROCS )
*
*           Temporarily define blacs grid to include all processes so
*           information can be broadcast to all processes
*
            CALL BLACS_GET( -1, 0, ICTXT )
            CALL BLACS_GRIDINIT( ICTXT, 'Row-major', 1, NPROCS )
*
            CALL IGEBR2D( ICTXT, 'All', ' ', 5, 1, WORK, 5, 0, 0 )
            N     = WORK( 1 )
            NRHS  = WORK( 2 )
            NB    = WORK( 3 )
            NPROW = WORK( 4 )
            NPCOL = WORK( 5 )
*
```

```
      END IF
*
      CALL BLACS_GRIDEXIT( ICTXT )
*
      RETURN
*
   20 WRITE( NOUT, FMT = 9997 )
      CLOSE( NIN )
      IF( NOUT.NE.6 .AND. NOUT.NE.0 )
     $   CLOSE( NOUT )
      CALL BLACS_ABORT( ICTXT, 1 )
*
      STOP
*
 9999 FORMAT( A )
 9998 FORMAT( 2X, A5, '   :        ', I6 )
 9997 FORMAT( ' Illegal input in file ',40A,'.  Aborting run.' )
*
*     End of PDSCAEXINFO
*
      END


      SUBROUTINE PDLAREAD( FILNAM, A, DESCA, IRREAD, ICREAD, WORK )
*
*  -- ScaLAPACK example --
*     University of Tennessee, Knoxville, Oak Ridge National Laboratory,
*     and University of California, Berkeley.
*
*     written by Antoine Petitet, August 1995 (petitet@cs.utk.edu)
*
*     .. Scalar Arguments ..
      INTEGER            ICREAD, IRREAD
*     ..
*     .. Array Arguments ..
      CHARACTER*(*)      FILNAM
      INTEGER            DESCA( * )
      DOUBLE PRECISION   A( * ), WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  PDLAREAD reads from a file named FILNAM a matrix and distribute
*  it to the process grid.
*
*  Only the process of coordinates {IRREAD, ICREAD} read the file.
```

```
*
*  WORK must be of size >= MB_ = DESCA( MB_ ).
*
*  =====================================================================
*
*     .. Parameters ..
      INTEGER            NIN
      PARAMETER          ( NIN = 11 )
      INTEGER            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DT_,
     $                   LLD_, MB_, M_, NB_, N_, RSRC_
      PARAMETER          ( BLOCK_CYCLIC_2D = 1, DLEN_ = 9, DT_ = 1,
     $                     CTXT_ = 2, M_ = 3, N_ = 4, MB_ = 5, NB_ = 6,
     $                     RSRC_ = 7, CSRC_ = 8, LLD_ = 9 )
*     ..
*     .. Local Scalars ..
      INTEGER            H, I, IB, ICTXT, ICURCOL, ICURROW, II, J, JB,
     $                   JJ, K, LDA, M, MYCOL, MYROW, N, NPCOL, NPROW
*     ..
*     .. Local Arrays ..
      INTEGER            IWORK( 2 )
*     ..
*     .. External Subroutines ..
      EXTERNAL           BLACS_GRIDINFO, INFOG2L, DGERV2D, DGESD2D,
     $                   IGEBS2D, IGEBR2D
*     ..
*     .. External Functions ..
      INTEGER            ICEIL
      EXTERNAL           ICEIL
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          MIN
*     ..
*     .. Executable Statements ..
*
*     Get grid parameters
*
      ICTXT = DESCA( CTXT_ )
      CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )
*
      IF( MYROW.EQ.IRREAD .AND. MYCOL.EQ.ICREAD ) THEN
         OPEN( NIN, FILE=FILNAM, STATUS='OLD' )
         READ( NIN, FMT = * ) ( IWORK( I ), I = 1, 2 )
         CALL IGEBS2D( ICTXT, 'All', ' ', 2, 1, IWORK, 2 )
      ELSE
         CALL IGEBR2D( ICTXT, 'All', ' ', 2, 1, IWORK, 2, IRREAD,
     $                 ICREAD )
      END IF
```

```
      M = IWORK( 1 )
      N = IWORK( 2 )
*
      IF( M.LE.0 .OR. N.LE.0 )
     $   RETURN
*
      IF( M.GT.DESCA( M_ ).OR. N.GT.DESCA( N_ ) ) THEN
         IF( MYROW.EQ.0 .AND. MYCOL.EQ.0 ) THEN
            WRITE( *, FMT = * ) 'PDLAREAD: Matrix too big to fit in'
            WRITE( *, FMT = * ) 'Abort ...'
         END IF
         CALL BLACS_ABORT( ICTXT, 0 )
      END IF
*
      II = 1
      JJ = 1
      ICURROW = DESCA( RSRC_ )
      ICURCOL = DESCA( CSRC_ )
      LDA = DESCA( LLD_ )
*
*     Loop over column blocks
*
      DO 50 J = 1, N, DESCA( NB_ )
         JB = MIN(  DESCA( NB_ ), N-J+1 )
         DO 40 H = 0, JB-1
*
*           Loop over block of rows
*
            DO 30 I = 1, M, DESCA( MB_ )
               IB = MIN( DESCA( MB_ ), M-I+1 )
               IF( ICURROW.EQ.IRREAD .AND. ICURCOL.EQ.ICREAD ) THEN
                  IF( MYROW.EQ.IRREAD .AND. MYCOL.EQ.ICREAD ) THEN
                     DO 10 K = 0, IB-1
                        READ( NIN, FMT = * ) A( II+K+(JJ+H-1)*LDA )
   10                CONTINUE
                  END IF
               ELSE
                  IF( MYROW.EQ.ICURROW .AND. MYCOL.EQ.ICURCOL ) THEN
                     CALL DGERV2D( ICTXT, IB, 1, A( II+(JJ+H-1)*LDA ),
     $                              LDA, IRREAD, ICREAD )
                   ELSE IF( MYROW.EQ.IRREAD .AND. MYCOL.EQ.ICREAD ) THEN
                     DO 20 K = 1, IB
                        READ( NIN, FMT = * ) WORK( K )
   20                CONTINUE
                     CALL DGESD2D( ICTXT, IB, 1, WORK, DESCA( MB_ ),
     $                              ICURROW, ICURCOL )
                  END IF
```

```
               END IF
               IF( MYROW.EQ.ICURROW )
     $            II = II + IB
               ICURROW = MOD( ICURROW+1, NPROW )
   30       CONTINUE
*
            II = 1
            ICURROW = DESCA( RSRC_ )
   40    CONTINUE
*
         IF( MYCOL.EQ.ICURCOL )
     $      JJ = JJ + JB
         ICURCOL = MOD( ICURCOL+1, NPCOL )
*
   50 CONTINUE
*
      IF( MYROW.EQ.IRREAD .AND. MYCOL.EQ.ICREAD ) THEN
         CLOSE( NIN )
      END IF
*
      RETURN
*
*     End of PDLAREAD
*
      END



      SUBROUTINE PDLAWRITE( FILNAM, M, N, A, IA, JA, DESCA, IRWRIT,
     $                      ICWRIT, WORK )
*
*  -- ScaLAPACK example --
*     University of Tennessee, Knoxville, Oak Ridge National Laboratory,
*     and University of California, Berkeley.
*
*     written by Antoine Petitet, August 1995 (petitet@cs.utk.edu)
*
*     .. Scalar Arguments ..
      INTEGER            IA, ICWRIT, IRWRIT, JA, M, N
*     ..
*     .. Array Arguments ..
      CHARACTER*(*)      FILNAM
      INTEGER            DESCA( * )
      DOUBLE PRECISION   A( * ), WORK( * )
*     ..
*
*  Purpose
```

```
*  =======
*
*  PDLAWRITE writes to a file named FILNAMa distributed matrix sub( A )
*  denoting A(IA:IA+M-1,JA:JA+N-1). The local pieces are sent to and
*  written by the process of coordinates (IRWWRITE, ICWRIT).
*
*  WORK must be of size >= MB_ = DESCA( MB_ ).
*
*  ======================================================================
*
*     .. Parameters ..
      INTEGER            NOUT
      PARAMETER          ( NOUT = 13 )
      INTEGER            BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DT_,
     $                   LLD_, MB_, M_, NB_, N_, RSRC_
      PARAMETER          ( BLOCK_CYCLIC_2D = 1, DLEN_ = 9, DT_ = 1,
     $                     CTXT_ = 2, M_ = 3, N_ = 4, MB_ = 5, NB_ = 6,
     $                     RSRC_ = 7, CSRC_ = 8, LLD_ = 9 )
*     ..
*     .. Local Scalars ..
      INTEGER            H, I, IACOL, IAROW, IB, ICTXT, ICURCOL,
     $                   ICURROW, II, IIA, IN, J, JB, JJ, JJA, JN, K,
     $                   LDA, MYCOL, MYROW, NPCOL, NPROW
*     ..
*     .. External Subroutines ..
      EXTERNAL           BLACS_BARRIER, BLACS_GRIDINFO, INFOG2L,
     $                   DGERV2D, DGESD2D
*     ..
*     .. External Functions ..
      INTEGER            ICEIL
      EXTERNAL           ICEIL
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          MIN
*     ..
*     .. Executable Statements ..
*
*     Get grid parameters
*
      ICTXT = DESCA( CTXT_ )
      CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )
*
      IF( MYROW.EQ.IRWRIT .AND. MYCOL.EQ.ICWRIT ) THEN
         OPEN( NOUT, FILE=FILNAM, STATUS='UNKNOWN' )
         WRITE( NOUT, FMT = * ) M, N
      END IF
*
```

```
      CALL INFOG2L( IA, JA, DESCA, NPROW, NPCOL, MYROW, MYCOL,
     $              IIA, JJA, IAROW, IACOL )
      ICURROW = IAROW
      ICURCOL = IACOL
      II = IIA
      JJ = JJA
      LDA = DESCA( LLD_ )
*
*     Handle the first block of column separately
*
      JN = MIN( ICEIL( JA, DESCA( NB_ ) ) * DESCA( NB_ ), JA+N-1 )
      JB = JN-JA+1
      DO 60 H = 0, JB-1
         IN = MIN( ICEIL( IA, DESCA( MB_ ) ) * DESCA( MB_ ), IA+M-1 )
         IB = IN-IA+1
         IF( ICURROW.EQ.IRWRIT .AND. ICURCOL.EQ.ICWRIT ) THEN
            IF( MYROW.EQ.IRWRIT .AND. MYCOL.EQ.ICWRIT ) THEN
               DO 10 K = 0, IB-1
                  WRITE( NOUT, FMT = 9999 ) A( II+K+(JJ+H-1)*LDA )
   10          CONTINUE
            END IF
         ELSE
            IF( MYROW.EQ.ICURROW .AND. MYCOL.EQ.ICURCOL ) THEN
               CALL DGESD2D( ICTXT, IB, 1, A( II+(JJ+H-1)*LDA ), LDA,
     $                       IRWRIT, ICWRIT )
            ELSE IF( MYROW.EQ.IRWRIT .AND. MYCOL.EQ.ICWRIT ) THEN
               CALL DGERV2D( ICTXT, IB, 1, WORK, DESCA( MB_ ),
     $                       ICURROW, ICURCOL )
               DO 20 K = 1, IB
                  WRITE( NOUT, FMT = 9999 ) WORK( K )
   20          CONTINUE
            END IF
         END IF
         IF( MYROW.EQ.ICURROW )
     $      II = II + IB
         ICURROW = MOD( ICURROW+1, NPROW )
         CALL BLACS_BARRIER( ICTXT, 'All' )
*
*        Loop over remaining block of rows
*
         DO 50 I = IN+1, IA+M-1, DESCA( MB_ )
            IB = MIN( DESCA( MB_ ), IA+M-I )
            IF( ICURROW.EQ.IRWRIT .AND. ICURCOL.EQ.ICWRIT ) THEN
               IF( MYROW.EQ.IRWRIT .AND. MYCOL.EQ.ICWRIT ) THEN
                  DO 30 K = 0, IB-1
                     WRITE( NOUT, FMT = 9999 ) A( II+K+(JJ+H-1)*LDA )
   30             CONTINUE
```

```
            END IF
         ELSE
            IF( MYROW.EQ.ICURROW .AND. MYCOL.EQ.ICURCOL ) THEN
               CALL DGESD2D( ICTXT, IB, 1, A( II+(JJ+H-1)*LDA ),
     $                       LDA, IRWRIT, ICWRIT )
            ELSE IF( MYROW.EQ.IRWRIT .AND. MYCOL.EQ.ICWRIT ) THEN
               CALL DGERV2D( ICTXT, IB, 1, WORK, DESCA( MB_ ),
     $                       ICURROW, ICURCOL )
               DO 40 K = 1, IB
                  WRITE( NOUT, FMT = 9999 ) WORK( K )
   40          CONTINUE
            END IF
         END IF
         IF( MYROW.EQ.ICURROW )
     $      II = II + IB
         ICURROW = MOD( ICURROW+1, NPROW )
         CALL BLACS_BARRIER( ICTXT, 'All' )
   50 CONTINUE
*
      II = IIA
      ICURROW = IAROW
   60 CONTINUE
*
      IF( MYCOL.EQ.ICURCOL )
     $   JJ = JJ + JB
      ICURCOL = MOD( ICURCOL+1, NPCOL )
      CALL BLACS_BARRIER( ICTXT, 'All' )
*
*     Loop over remaining column blocks
*
      DO 130 J = JN+1, JA+N-1, DESCA( NB_ )
         JB = MIN(  DESCA( NB_ ), JA+N-J )
         DO 120 H = 0, JB-1
            IN = MIN( ICEIL( IA, DESCA( MB_ ) ) * DESCA( MB_ ), IA+M-1 )
            IB = IN-IA+1
            IF( ICURROW.EQ.IRWRIT .AND. ICURCOL.EQ.ICWRIT ) THEN
               IF( MYROW.EQ.IRWRIT .AND. MYCOL.EQ.ICWRIT ) THEN
                  DO 70 K = 0, IB-1
                     WRITE( NOUT, FMT = 9999 ) A( II+K+(JJ+H-1)*LDA )
   70             CONTINUE
               END IF
            ELSE
               IF( MYROW.EQ.ICURROW .AND. MYCOL.EQ.ICURCOL ) THEN
                  CALL DGESD2D( ICTXT, IB, 1, A( II+(JJ+H-1)*LDA ),
     $                          LDA, IRWRIT, ICWRIT )
               ELSE IF( MYROW.EQ.IRWRIT .AND. MYCOL.EQ.ICWRIT ) THEN
                  CALL DGERV2D( ICTXT, IB, 1, WORK, DESCA( MB_ ),
```

```
     $                          ICURROW, ICURCOL )
                  DO 80 K = 1, IB
                     WRITE( NOUT, FMT = 9999 ) WORK( K )
   80             CONTINUE
               END IF
            END IF
            IF( MYROW.EQ.ICURROW )
     $         II = II + IB
            ICURROW = MOD( ICURROW+1, NPROW )
            CALL BLACS_BARRIER( ICTXT, 'All' )
*
*           Loop over remaining block of rows
*
            DO 110 I = IN+1, IA+M-1, DESCA( MB_ )
               IB = MIN( DESCA( MB_ ), IA+M-I )
               IF( ICURROW.EQ.IRWRIT .AND. ICURCOL.EQ.ICWRIT ) THEN
                  IF( MYROW.EQ.IRWRIT .AND. MYCOL.EQ.ICWRIT ) THEN
                     DO 90 K = 0, IB-1
                        WRITE( NOUT, FMT = 9999 ) A( II+K+(JJ+H-1)*LDA )
   90                CONTINUE
                  END IF
               ELSE
                  IF( MYROW.EQ.ICURROW .AND. MYCOL.EQ.ICURCOL ) THEN
                     CALL DGESD2D( ICTXT, IB, 1, A( II+(JJ+H-1)*LDA ),
     $                             LDA, IRWRIT, ICWRIT )
                   ELSE IF( MYROW.EQ.IRWRIT .AND. MYCOL.EQ.ICWRIT ) THEN
                     CALL DGERV2D( ICTXT, IB, 1, WORK, DESCA( MB_ ),
     $                             ICURROW, ICURCOL )
                     DO 100 K = 1, IB
                        WRITE( NOUT, FMT = 9999 ) WORK( K )
  100                CONTINUE
                  END IF
               END IF
               IF( MYROW.EQ.ICURROW )
     $            II = II + IB
               ICURROW = MOD( ICURROW+1, NPROW )
               CALL BLACS_BARRIER( ICTXT, 'All' )
  110       CONTINUE
*
            II = IIA
            ICURROW = IAROW
  120    CONTINUE
*
         IF( MYCOL.EQ.ICURCOL )
     $      JJ = JJ + JB
         ICURCOL = MOD( ICURCOL+1, NPCOL )
         CALL BLACS_BARRIER( ICTXT, 'All' )
```

```
*
  130 CONTINUE
*
      IF( MYROW.EQ.IRWRIT .AND. MYCOL.EQ.ICWRIT ) THEN
         CLOSE( NOUT )
      END IF
*
 9999 FORMAT( D30.18 )
*
      RETURN
*
*     End of PDLAWRITE
*
      END
```

Bibliographie

[1] *Survey of Principal Investigators of Grand Challenge Applications*, volume Workshop on Grand Challenge Applications and Software Technology, Pittsburgh, May 1993.

[2] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993.

[3] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. In *CPC'93*, November 1993.

[4] B. Avalani, A. Choudhary, I. Foster, R. Krihnaiyer, and M. Xu. A data transfer library for communicating data-parallel tasks. Technical report, Syracuse University, Argone Natinal Laboratory, NY 13244, 1994.

[5] B. Avalani, A. Choudhary, I. Foster, and R. Krishnaiyer. Integrating task and data parallelism using parallel I/O techniques. In *International Workshop on Parallel Processing*, December 1994.

[6] S. Batra, P. J. Hatcher, and R. D. Russell. The design and implementation of data-parallel files. In *Workshop on Modeling and Specification of I/O*, San Antonio, Texas, October 1995.

[7] J. L. Bell and G. Patterson. Data organization in large numerical computations. *The Journal of Supercomputing*, 1(1), 1987.

[8] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, and J. Dongarra. ScaLAPACK: a linear algebra library for message-passing computers. In *SIAM Conference on Parallel Processing*, March 1997.

[9] R. Bordawekar and A. Choudhary. HPF with parallel I/O extensions. Technical Report SCCS-613, NPAC, Syracuse University, 1993.

[10] R. Bordawekar and A. Choudhary. Extending I/O capabilities of High Performance Fortran: Initial experiences. Technical Report CACR-115, Scalable I/O Initiative, Center of Advanced Computing Research, California Insititute of Technology, December 1995.

[11] S. Chatterjee, J. R.Gilbert, F. J. Long, R. Schreiber, and S.-H. Ten. Generating local addresses and communication sets for data-parallel program. In *Symposium on Principles and Practice of Parallel Programming*, San Diego, May 1993.

[12] J. Choi, J. Dongarra, and D. Walker. The design of scalable software libraries for distributed memory concurrent computers. In *Environments and tools For Parallel Scientific Computing*, pages 3–15, 1992.

[13] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, and R. Ponnusamy. PASSION : Parallel and scalable software for input-output. Technical Report SCCS-636, Syracuse University, Syracuse, NY 13244, September 1994.

[14] F. Coelho and C. Ancourt. Optimal compilation of HPF remappings. Technical report, Centre de Recherche en Informatique, July 1995.

[15] Corbett, Feitelson, and Fineberg. Overview of the MPI-IO parallel I/O interface. In *Third Workshop on I/O in Parallel and Distributed Systems, IPPS '95*, pages 1–15, Santa Barbara, CA, April 1995.

[16] P. Corbett and D. Feitelson. Overview of the MPI-IO parallel I/O interface. In *Third Workshop on I/O in Parallel and Distributed Systems, IPPS '95, Santa Barbara, CA*, pages 1–15, April 1995.

[17] P. Corbett, D. Feitelson, S. Fineberg, and Y. Hsu. Overview of the MPI-IO parallel I/O interface. In *IPPS'95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, April 1995.

[18] T. H. Cormen and D. Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[19] J.-L. Dekeyser and P. Marquet. Supporting irregular and dynamic computations in data-parallel languages. In *Spring School on Data Parallelism*, pages 197–219, Les Ménuires, France, Mar. 1996. Lectures Notes in Computer Science, Tutorials Series.

[20] J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS'93 Workshop on Input/Output in Parallel Computer Systems*, April 1993.

[21] J. M. del Rosario and A. Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE computer, 27(3):59-68*, March 1993.

[22] J. M. del Rosario, M. Harry, and A. Choudhary. The design of VIP-FS a virtual, parallel file system for high performance parallel and distributed computing. Technical Report SCCS-628, Syracuse University, NY 13244-4100, 1994.

[23] J. Dongarra, R. V. D. Geijn, and R. Waley. Two dimensional basic linear algebra communication subprograms. In *Environments and tools For Parallel Scientific Computing*, pages 17–29, 1992.

[24] J. Dongarra, C. Randriamaro, L. Prylli, and B. Tourancheau. Array redistribution in scalapack using PVM. In *EuroPVM users' group*, 1995.

[25] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *J. Parallel and Distributed Computing*, 25(1), 1995.

[26] I. Foster, M. Xu, B. Avalani, and A. Choudhary. A compilation system that integrates high performance Fortran and Fortran M. In *Proceedings of the Scalable High-Performance Computing Conference (Knoxville Tennessee)*, pages 293–300. IEEE Computer Society, May 1994.

[27] M. Harry, J. M. del Rosario, and A. Choudhary. VIP-FS : a virtual, parallel file system for high performance parallel and distributed computing. In *Ninth International Parallel Processing Symposium*, 1995.

[28] P. J. Hatcher, R. D. Russell, S. Kumaran, and M. J. Quinn. Implementing data-parallel programs on commodity clusters. In *Spring School on Data Parallelism*, Les Menuires, France, March 1996. Lectures Notes in Computer Science, Tutorials Series.

[29] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[30] E. T. Kalns and L. M. Ni. Darel: A portable data redistribution library for distributed-memory machines. In *Scalable Parallel Libraries Conference*, East Lansing, MI 48824-1027, October 1994.

[31] K. Knobe, J. Lukas, and J. Steele. Data optimization : Allocation of arrays to reduce communication on SIMD machines. *Parallel and Distributed Computing*, 8(2):102–118, 1990.

[32] C. Koelbel, D. loveman, R. Schreiber, G. steele, and M. Zosel. *High perfromance Fortran Handbook.* MIT press, 1994.

[33] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, November 1994.

[34] D. Lazure. *Programmation Géométrique à Parallélisme de Données — Modèle, Langage et Compilation.* Thèse de doctorat (PhD Thesis), Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, January 1995.

[35] J. A. Moore, P. J. Hatcher, and M. J. Quinn. Stream*: Fast, flexible, data-parallel I/O. In *Parco*, September 1995.

[36] J. A. Moore, P. J. Hatcher, and M. J. Quinn. Efficient data-parallel files via automatic mode detection. In *Fourth Annual Workshop on I/O in Parallel and Distributed Systems*, Philadelphia, Pennsylvania, May 1996.

[37] S. A. Moyer and V. Sunderam. A parallel I/O system for high-performance distributed computing. In *IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems*, 1994.

[38] S. A. Moyer and V. Sunderam. PIOUS : A scalable parallel I/O system for distributed computing environments. In *High Performance Computing Comference 94*, Atlanta, GA 30322, U.S.A, 1994.

[39] S. A. Moyer and V. S. Sunderam. *PIOUS for PVM, Version 1.1, User's Guide and Reference Manual.* Emory University, Atlanta, GA 30322, U.S.A, December 1994.

[40] S. A. Moyer and V. S. Sunderam. Parallel I/O as a parallel application. In *International Journal of Supercomputer Applications*, volume 9, pages 95–107, 1995.

[41] C. Peter and Feitselson. MPI-IO: A parallel file I/O interface for MPI. Technical Report 19841 (87784), IBM T.J. Watson Research Center, Novemeber 1994.

[42] L. Prylli and B. Tourancheau. Efficient bloc cyclic data redistribution. In *Euro-par'96.* Ecole Normale supérieure de Lyon, August 1996.

[43] S. Ramaswamy. *Simultaneous exploitation of task and data parallelism in irregular scientific applications.* PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[44] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for memory multicomputers. In *Frontiers'95*, February 1995.

[45] K. E. Seamons. *PANDA : fast access to persistent arrays using high level interfaces and server directed input/output.* PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[46] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in panda. In *Proceedings of Supercomputing*, San Diego, December 1995. ACM SIGPLAN.

[47] K. E. Seamons, Y. Chen, M. Winslett, Y. Cho, S. Kuo, and M. Subramaniam. Persistent array access using server-directed I/O. In *8th International Working Conference on Scientific and Statistical Database Management*, Stockholm, Sweden, June 1996.

[48] K. E. Seamons and M. Winslett. An efficient abstract interface for multidimensional array I/O. In *Proceedings of Supercomputing*, pages 650–659, Washington, November 1994.

[49] K. E. Seamons and M. Winslett. Physical schemas for large multidimensional arrays in scientific computing applications. In *7 th international working conference on Scientific and Statistical Database Management*, pages 218–227, September 1994.

[50] J. M. Stichnoth, D. O'Hallaron, and T. R. Gross. Generatimg communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.

[51] D. Sueur. Shell hétérogène à parallélisme de données. In *Renpar'7, Actes des 7 Rencontres Francophones du parallélisme*, pages 58–61, PIP-FPMs Mons, Belgique, June 1995.

[52] D. Sueur. Redistributions dynamiques sur machines parallèles hétérogènes. In *Renpar'8, Actes des 8 Rencontres Francophones du parallélisme*, pages 41–44, Bordeaux, May 1996.

[53] D. Sueur and J.-L. Dekeyser. Dynamic redistribution on heterogeneous parallel computers. In *Euro-par'96.* Ecole Normale supérieure de Lyon, August 1996.

[54] D. Sueur and J.-L. Dekeyser. Dpshell: A data parallel file system. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, Minnesota, March 1997.

[55] R. Thakur. *Runtime Support for In-Core and Out-of-Core Data-Parallel Programs.* PhD thesis, Dept. of Electrical and Computer Eng., Syracuse University, May 1995.

[56] R. Thakur, alok Choudhary, and Fox. Runtime array redistribution in HPF programs. In *SHPCC'94, pp 309-316.* IEEE, May 1994.

[57] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and runtime support for out-of-core HPF programs. In *8th ACM Int. Conf. on Supercomputing*, pages 382–391, July 1994.

[58] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. Passion runtime library for parallel I/O. In *Scalable Parallel Libraries Conference*, October 1994.

[59] R. Thakur, A. Choudhary, and J. Ramanujam. Efficient algorithms for array redistribution. *to appear in IEEE Trans. on Parallel and Dist. Systems*, 1996.

[60] D. Walker and S. Otto. Redistribution of block-cyclic data distributions using MPI. *Concurrency: Practice and Experience*, 8(9):707–728, 1996.

Table des figures

THESES

1997

06 Janvier — **Philippe MESEURE**
Modélisation de corps déformables pour la simulation d'actes chirurgicaux.

07 Janvier — **Olivier ROUSSEL**
L'achèvement des bases de connaissances en calcul propositionnel et en calcul des prédicats.

08 Janvier — **Nouredine MELAB**
Gestion de la granularité et régulation de charge dans le modèle P^3 d'évaluation parallèle des langages fonctionnels

09 Janvier — **Cédric DUMOULIN**
Dream : une mémoire partagée répartie à cohérence programmable.

13 Janvier — **Raymond NAMYST**
PM^2 : Un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.

14 Janvier — **Philippe MERLE**
CorbaScript - Corba Web : propositions pour l'accès à des objets et services distribués.

27 Janvier — **David GALINEC**
Exécution asynchrone de programmes synchrones par transformations automatiques : application au traitement d'images temps-réel.

14 Mars — **Côme RACZY**
Commandes optimales en temps pour des systèmes différentiellement plats. Application aux commandes de satellites et de grues.

27 Mars — **Jean Jacques VANDEWALLE**
Projet OSMOSE : Mosélisation et implémentation pour l'interopérabilité de services carte à microprocesseur par l'approche orientée objet.

10 Juin — **David SIMPLOT**
Langages de mots de figures, monoïdes inversifs et langages de mots à deux dimensions

20 Juin — **Dominique SUEUR**
Algorithmes de redistribution de données. Application aux systèmes de fichiers parallèles distribués.

25 Juin

Olivier DELGRANGE

Un algorithme rapide pour une compression modulaire optimale. Application à l'analyse de séquences génétiques.

THESES

1998

09 Janvier — **David CARLIER**
Représentation permanente, coordonnée par une carte à microprocesseur, d'un utilisateur mobile.

16 Janvier — **Yves DENNEULIN**
Conception et ordonnancement des applications hautement irrégulières dans un contexte de parallélisme à grain fin.

23 Janvier — **Jaafar GABER**
Plongements et manipulations d'arbres dans les architectures distribuées.

30 Janvier — **Grégory SAUGIS**
Interfaces 3D pour le travail coopératif synchrone, une proposition.