Le traitement en parallèle sous Linux

Version française du Parallel Processing HOWTO

Hank Dietz

<hankd CHEZ engr POINT uky POINT edu>

Adaptation française : Dominique van den Broeck

Relecture de la version française : Isabelle Hurbain

Préparation de la publication de la v.f. : Jean-Philippe Guérard

Version : 980105.fr.1.1

4 septembre 2005

Historique des versions		
Version 980105.fr.1.1	2005-09-04	DV
Quelques améliorations mineures de la version française		
Version 980105.fr.1.0	2004-09-13	DV, IH, JPG
Première traduction française		
Version 980105	1998-01-05	HGD
Version originale		

Résumé

Le *traitement en parallèle* (« *Parallel Processing* ») se réfère à l'idée d'accélérer l'exécution d'un programme en divisant celui-ci en plusieurs fragments pouvant être exécutés simultanément, chacun sur son propre processeur, un programme exécuté sur *N* processeurs pouvant alors fonctionner *N* fois plus vite qu'il le ferait en utilisant un seul processeur. Ce document traite des quatre approches de base du traitement en parallèle accessibles aux utilisateurs de Linux : les systèmes Linux SMP, les grappes (« *cluster* ») de systèmes Linux mis en réseau, l'exécution en parallèle avec utilisation des instructions multimédia (ex : MMX), et l'utilisation des processeurs secondaires embarqués dans une machine fonctionnant sous Linux.

Table des matières

1. Introduction

L'*exécution en parallèle* (« *Parallel Processing* ») se rapporte à l'idée d'accélérer l'exécution d'un programme en le divisant en plusieurs fragments pouvant être exécutés simultanément, chacun sur son propre processeur. Un programme exécuté sur *N* processeurs pourrait alors fonctionner *N* fois plus vite qu'il ne le ferait en utilisant un seul processeur.

Traditionnellement, les processeurs multiples sont fournis avec un « ordinateur en parallèle » spécialement conçu. De ce fait, Linux gère désormais les systèmes *SMP* (souvent vendus en tant que « serveurs ») dans lesquels plusieurs processeurs partagent le même bus et la même mémoire au sein d'un même ordinateur. Il est également possible à un groupe d'ordinateurs (par exemple un groupe de PC fonctionnant chacun avec Linux) d'être interconnectés par un réseau pour former un ensemble de traitement en parallèle (« *cluster* »). La troisième alternative pour l'exécution parallèle sous Linux est l'utilisation du jeu d'instructions multimédias étendu (MultiMedia Extend : MMX) pour agir en parallèle sur des vecteurs de données entières. Il est enfin possible d'utiliser un système Linux comme « hôte » hébergeant un moteur de calcul en parallèle *dédié*. Toutes ces approches sont traitées en détails dans ce document.

1.1. Le traitement en parallèle correspond-il à mes besoins ?

Bien que l'emploi de multiples processeurs puisse accélérer nombre d'opérations, la plupart des applications ne peuvent encore tirer profit du traitement en parallèle. A la base, le traitement en parallèle est approprié si :

- Votre application est suffisamment parallélisée pour faire bon usage de multiples processeurs. C'est, en partie, une question d'identification des différentes portions du programme pouvant être exécutées indépendamment et simultanément sur des processeurs séparés, mais vous découvrirez aussi que certaines choses qui *peuvent* être exécutées en parallèle ralentissent le traitement si elles sont exécutées en parallèle sur un système particulier. Par exemple, un programme qui s'exécute en quatre secondes sur une machine unique pourrait être capable de n'occuper qu'une seconde du temps de chacune de quatre machines, mais n'apportera pourtant aucun gain de temps s'il faut trois secondes ou plus à ces machines pour coordonner leurs actions.

- Soit l'application qui vous intéresse en particulier a été *parallélisée*, c'est-à-dire réécrite pour tirer profit du traitement en parallèle, soit vous comptez produire au moins un peu de code original qui le fasse.

- Vous êtes intéressé par la recherche de nouvelles solutions impliquant un traitement en parallèle, ou au moins souhaitez vous familiariser avec. Le traitement en parallèle sous Linux n'est pas forcément difficile, mais ce n'est pas une notion familière à beaucoup d'utilisateurs, et il n'existe pas de livre intitulé « Le *Parallel Processing* pour les nuls », en tout cas pas encore. Ce guide est un bon point de départ, mais il ne contient pas l'intégralité de ce que vous devez connaître.

La bonne nouvelle, c'est que si tout ce qui vient d'être dit est vrai, vous découvrirez cependant que le traitement en parallèle sous Linux peut apporter les performances d'un supercalculateur à des programmes effectuant des opérations complexes ou travaillant sur de très grandes quantités de données. Mais en plus, cela peut être fait en utilisant du matériel peu onéreux et que vous possédez sûrement déjà. Avec çà, il reste aisé d'utiliser un système de traitement en parallèle sous Linux à d'autres choses lorsqu'il n'est pas en train d'accomplir un traitement en parallèle.

Si le traitement en parallèle ne correspond *pas* à vos besoins, mais que vous souhaitez tout de même améliorer sensiblement les performances de votre machine, il reste des choses que vous pouvez faire. Par exemple, vous pouvez améliorer les performances d'un programme séquentiel en remplaçant votre processeur par un plus rapide, en ajoutant de la mémoire, en remplaçant un disque IDE par un « *fast wide SCSI* », et cætera. Si c'est ce qui vous intéresse, sautez directement à la section 6.2, sinon poursuivez votre lecture.

1.2. Terminologie

Bien que le traitement en parallèle ait été utilisé pendant de nombreuses années par de nombreux systèmes, il reste étranger à la plupart des utilisateurs. Aussi, avant de traiter des différentes alternatives, il est important de se familiariser avec une poignée de termes usuels :

SIMD :

> SIMD (« *Single Instruction stream, Multiple Data stream* », ou « Un seul flot d'instruction, plusieurs flots de données »), fait référence à un modèle d'exécution en parallèle dans lequel tous les processeurs traitent la même opération à la fois, mais où chaque processeur est autorisé à agir sur sa propre donnée. Ce modèle convient naturellement au concept où l'on applique le même

traitement sur chaque élément d'un tableau, et est ainsi souvent associé à la manipulation de vecteurs ou de tableaux. Toutes les opérations étant implicitement synchronisées, les interactions entre processeurs SIMD tendent à être facilement et efficacement mises en œuvre.

MIMD :

MIMD (« *Multiple Instruction stream, Multiple Data stream* », ou « Plusieurs flots d'instructions, plusieurs flots de données »), se rapporte au modèle d'exécution en parallèle dans lequel chaque processeur agit essentiellement seul. C'est le modèle qui convient le mieux à la décomposition d'un programme pour une exécution en parallèle sur une base fonctionnelle. Par exemple, une processeur peut mettre à jour une base de données pendant qu'un autre produit l'affichage graphique de la nouvelle entrée. C'est un modèle plus flexible que l'exécution en SIMD, mais qui s'accomplit au risque d'un cauchemar pour le débogueur, les « *race conditions* » ou « accès concurrents », dans lesquels un programme peut planter de façon intermittente à cause des différences de minutage entre les opérations des différents processeurs lorsque celles d'un de ces processeurs sont réorganisées en fonction de celles d'un autre.

SPMD :

SPMD (« *Single Program, Multiple Data* »), ou « Un seul programme, plusieurs données », est une version restreinte du MIMD dans laquelle tous les processeurs exécutent le même programme. Contrairement au SIMD, chaque processeur peut suivre un chemin différent dans le programme.

Bande passante :

La bande passante (« *bandwidth* ») d'un système de communication correspond à la quantité maximum de données que l'on peut transmettre en une unité de temps… une fois que la transmission de données a commencé. La bande passante des connexions série est souvent mesurée en *bauds* ou en *bits par seconde (b/s)*, ce qui correspond en général à huit fois ou dix fois le nombre d'*octets par secondes (O/s ou B/s*, B = « *Byte* » = « Octet »). Par exemple, un modem à 1200 bauds (N.D.T. : comme celui du Minitel) peut transférer environ 120 octets à la seconde (B/s), tandis qu'une connexion réseau ATM à 155 Mb/s est environ 130 000 fois plus rapide, en transférant environ 17 Mo/s. Une bande passante élevée permet un transfert efficace de larges blocs de données entre processeurs.

Latence :

La latence d'un système de communication représente le temps minimum nécessaire pour la transmission d'un objet, en incluant toutes les données « forfaitaires » logicielles pour l'émission et la réception (« *overhead* »). Le temps de latence est très important dans les traitements en parallèle car il détermine la *granularité*, la durée minimum d'exécution d'un segment de code pour gagner en vitesse d'exécution grâce au traitement en parallèle. Concrètement, si un segment de code s'exécute en moins de temps qu'il n'en faut pour transmettre son résultat (ce délai-ci formant la latence), exécuter ce segment en série plutôt qu'en parallèle sera plus rapide puisqu'il n'y aura pas de délai de transmission.

Envoi de messages (« *Message passing* ») :

Les envois de message sont un modèle d'interaction entre les différents processeurs d'un système parallèle. En général, un message est construit logiciellement sur un processeur et envoyé via une interconnexion réseau à un autre processeur. Bien que le surcoût en temps (« *overhead* ») engendré par la gestion de chaque message (ce délai formant la latence) soit élevé, typiquement, il n'y a que peu de restrictions quant à la quantité d'informations que ce message peut contenir. Ainsi, l'envoi de messages peut assurer une bande passante élevée en apportant une méthode efficace pour transmettre de grandes quantités d'informations d'un processeur à un autre. En revanche, afin de réduire les besoins en coûteuses opérations d'envoi de message, les structures de données à l'intérieur d'un programme doivent être réparties à travers tous les processeurs de façon à ce que la plupart des données référencées par chaque processeur se trouve dans sa mémoire locale… Cette tâche porte le nom de « répartition des données » (« *data layout* »).

Mémoire partagée :

La mémoire partagée est elle aussi un modèle d'interaction entre les processeurs d'un système parallèle. Les systèmes comme les machines biprocesseurs Pentium faisant fonctionner Linux partagent *physiquement* la même mémoire entre tous leurs processeurs, si bien qu'une valeur écrite par un processeur est directement accessible par un autre processeur. A l'inverse, la mémoire partagée *logiquement* peut être implémentée sur les systèmes où chaque processeur dispose d'une mémoire qui lui est propre, en convertissant chaque référence à une zone non locale de la mémoire en une communication inter-processeur appropriée. Cette implémentation de la mémoire partagée est généralement considérée comme étant plus facile à utiliser que les files de messages. La mémoire partagée physiquement peut présenter à la fois une bande passante élevée et des temps de latence très bas, mais seulement lorsque les différents processeurs n'essaient pas d'accéder au bus simultanément. Ainsi, le modèle de répartition des données peut avoir une sérieuse influence sur les performances, et les effets de cache et autres peuvent rendre très difficile la détermination du meilleur modèle.

Fonctions d'agrégation (*Aggregate Functions*) :

Dans le modèle des files de messages comme dans celui de la mémoire partagée, une communication est initiée par un processeur seul. Par contraste, une fonction d'agrégation est un modèle implicitement parallèle dans lequel tous les processeurs d'un groupe agissent ensemble. Le cas le plus simple est celui des *barrières de synchronisation*, dans lequel chaque processeur se met en attente jusqu'à ce que le groupe entier ait atteint la barrière. Si chaque processeur émet une donnée en atteignant une barrière, il est possible de demander à l'électronique responsable des communications d'émettre en retour une valeur à chaque processeur, valeur qui pourrait être fonction des données collectées sur tous les processeurs. Par exemple, la valeur de retour pourrait être la réponse à la question « Est-ce qu'un processeur a trouvé la réponse ? » ou pourrait être la somme d'une valeur propre à chaque processeur. Les temps de latence peuvent être bas, mais la bande passante a tendance à être basse elle aussi. Traditionnellement, ce modèle est surtout utilisé pour contrôler l'exécution en parallèle, plutôt que pour distribuer les données.

Communication collective :

C'est un autre nom pour les fonctions d'agrégation, utilisé le plus souvent en référence à celles qui sont construites en utilisant de multiples opérations d'envoi de message.

SMP :

SMP (« *Symmetric Multi-Processor* ») se rapporte au concept d'un groupe de processeurs travaillant ensemble en tant qu'homologues, si bien que chaque partie d'un travail peut être effectuée de la même façon par n'importe quel processeur de ce groupe. Typiquement, le SMP implique la combinaison du MIMD et de la mémoire partagée. Dans l'univers IA32, SMP signifie souvent « compatible MPS » (« *Intel MultiProcessor Specification* »). À l'avenir, cela pourrait signifier « *Slot 2* »…

SWAR :

SWAR (« *SIMD Within A Register* », ou « SIMD à l'intérieur d'un registre ») est un terme générique qui désigne le concept consistant à partitionner un registre en plusieurs champs entiers et à effectuer des opérations sur toute la largeur du registre pour faire du calcul en parallèle SIMD sur tous les champs à la fois. En considérant une machine avec des registres longs de *k* bits (et donc autant pour les chemins de données, et les unités des fonctions), on sait depuis longtemps que les opérations sur les registres ordinaires peuvent fonctionner comme des opérations parallèle SIMD sur *n* champs de *k/n* bits. Bien que ce type de parallélisme puisse être mis en œuvre en utilisant les registres entiers et les instructions ordinaires, plusieurs modèles récents de microprocesseurs intègrent des instructions spécialisées pour améliorer les performances de cette technique pour des tâches orientées multimédia. En plus du *MMX* (« *MultiMedia eXtension* ») d'Intel/AMD/Cyrix, il existe : *MAX* (« *MultimediA eXtension* ») sur l'Alpha de Digital, *MAX* (« *Multimedia Acceleration eXtension* ») sur le PA-RISC de Hewlett-Packard, *MDMX* (« *Digital Media eXtension* », prononcé « Mad Max ») sur MIPS, et *VIS* (« *Visual Instruction Set* ») sur le SPARC V9 de Sun. En dehors des trois constructeurs qui ont adopté le MMX, tous ces jeux d'instructions sont comparables, mais incompatibles entre eux.

Processeur auxiliaires, dédiés. (« *Attached Processors* ») :

Les processeurs auxiliaires sont essentiellement des calculateurs dédiés à une tâche particulière, reliés à un système *hôte* et servant à accélérer certains types de calculs. Par exemple, de nombreuses cartes vidéo et son pour PC embarquent des processeurs dédiés conçus pour accélérer respectivement les opérations graphiques et le *DSP* audio (DSP : « *Digital Signal Processing* », soit « Traitement Numérique du Signal »). Il existe aussi une large variété de *processeurs de tableaux* (« *array processors* »), nommés ainsi car ils sont conçus pour accélérer les opérations arithmétiques sur les tableaux. À dire vrai, un certain nombre de supercalculateurs commerciaux sont en réalité formés de processeurs dédiés rattachés à des stations de travail hôtes.

RAID :

RAID (« *Redundant Array of Inexpensive Disks* », soit « Batterie Redondante de Disques Peu Coûteux ») est une technologie simple servant à améliorer tant la bande passante que la fiabilité des accès disque. Même si le RAID se décline en plusieurs variantes, toutes ont en commun deux concepts-clés : D'abord, chaque bloc de données est *découpé* en segments distribués à un groupe de *n+k* disques de façon à ce que chaque disque n'ait à lire que *1/n*ième de la donnée… offrant ainsi *n* fois la bande passante d'un seul disque. Ensuite, des données redondantes sont écrites pour que les données puissent être recouvrées si un des disques vient à défaillir. C'est important car autrement, si l'un des *n+k* disques tombait en panne, le système de fichiers entier pourrait être perdu. Il existe une bonne présentation du système RAID sur http://www.dpt.com/uraiddoc.html, ainsi que des informations concernant le RAID pour Linux sur http://linas.org/linux/raid.html. Hormis la prise en charge du matériel RAID spécialisé, Linux gère aussi le RAID logiciel 0, 1, 4 et 5 à travers plusieurs disques hébergés sur un système Linux unique. Reportez-vous aux Software RAID mini-HOWTO et Multi-Disk System Tuning mini-HOWTO pour plus de détails. Le RAID au travers de plusieurs disques *sur plusieurs machines en clusters* n'est pas directement pris en charge.

IA32 :

L'IA32 (« *Intel Architecture* », 32 bits) n'a rien à voir avec le traitement en parallèle, mais se réfère à la classe de processeurs dont les instructions sont compatibles avec celles de l'Intel 386. Concrètement, tout processeur Intel x86 après le 286 est compatible avec le modèle de mémoire « à plat[1 [p 71]] » qui caractérise l'IA32. AMD et Cyrix font eux aussi une multitude de processeurs compatibles IA32. Comme Linux a évolué principalement sur des processeurs IA32 et que c'est là qu'est centré le marché de la grande consommation, il est commode d'utiliser le terme IA32 pour distinguer ce type de processeur des PowerPC, Alpha, PA-RISC, MIPS, SPARC, et cætera. La future IA64 (64 bits avec EPIC, « *Explicitly Parallel Instruction Computing* ») va certainement compliquer les choses, mais la production de Merced, le premier processeur IA64, n'est pas envisagée avant 1999.

Produits du commerce :

Depuis la mort de plusieurs fabricants de supercalculateurs en parallèle, les solutions commerciales toutes faites et prêtes à l'emploi (en anglais « *Commercial Off-The-Shelf* » ou *COTS*, pour « disponibles en rayons ») sont couramment considérées comme une nécessité dans le monde des systèmes de calcul en parallèle. En étant totalement puriste, les seuls moyens de traitement en parallèle disponibles sous forme de produits du commerce utilisant des PC sont des choses comme les serveurs Windows NT en SMP et les différentes applications Windows utilisant le MMX. Être aussi puriste ne mène à rien. L'idée fondamentale de l'utilisation de produits du commerce est de réduire les coûts et les temps de développement. Ainsi, une manière plus complète et plus utile de comprendre l'utilisation de ce type de produit serait de dire que la plupart des sous-systèmes tirent profit du marché de masse mais que d'autres technologies sont utilisées là où elles servent vraiment. Le plus souvent, les produits du commerce pour le traitement en parallèle sont utilisés au sein d'un groupe de machines (« *cluster* ») dans lequel les postes sont des PC courants, mais dont l'interface réseau et les logiciels ont été quelque peu *personnalisés*… classiquement, fonctionnant sous Linux avec des applications dont le code source est libre et disponible (par exemple sous *copyleft* ou dans le domaine public), mais pas littéralement des produits du commerce.

1.3. Algorithme d'exemple

Afin de bien comprendre l'usage des différentes approches de programmation en parallèle mises en évidence dans ce guide, il est utile d'étudier cet exemple. Bien qu'un algorithme simple de traitement en parallèle eût suffi, si l'on en choisit un qui a déjà été utilisé pour faire la démonstration d'autres systèmes de programmation parallèle, il devient un peu plus facile de comparer et mettre en évidence les caractéristiques de ces différentes approches. le livre de M. J. Quinn, *Parallel Computing Theory And Practice* (« Théorie et Pratique du Calcul en Parallèle »), seconde édition, édité par McGraw Hill, New York en 1994, utilise un algorithme parallèle qui calcule la valeur de Pi pour présenter différents environnements de programmation sur supercalculateurs parallèles (par exemple, le *message passing* du nCube ou la mémoire partagée des Sequent). Dans ce guide, nous utiliserons le même algorithme.

Cet algorithme calcule la valeur approchée de Pi en faisant la somme de l'aire située sous x au carré. En tant que programme C purement séquentiel, l'algorithme ressemble à :

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

main(int argc, char **argv)
{
  register double largeur, somme;
  register int intervalles, i;
```

```
  /* Lit le nombre d'intervalles désiré */
  intervalles = atoi(argv[1]);
  largeur = 1.0 / intervalles;

  /* fait le calcul */
  somme = 0;
  for (i=0; i<intervalles; ++i) {
    register double x = (i + 0.5) * largeur;
    somme += 4.0 / (1.0 + x * x);
  }
  somme *= largeur;

  printf("Estimation de la valeur de pi: %f\n", somme);

  return(0);
}
```

En revanche, cet algorithme séquentiel conduit facilement à une implémentation « parallèle et embarrassante ». L'aire est subdivisée en intervalles, et un nombre quelconque de processeurs peut faire la somme de l'intervalle qui lui est assigné indépendamment des autres, sans nécessité d'interaction entre les processeurs. Une fois que les sommes locales ont toutes été calculées, elles sont additionnées pour former la somme globale. Cette étape requiert un certain niveau de coordination et de communication entre les différents processeurs. Enfin, cette somme globale est renvoyée à l'écran par un seul processeur, en tant que valeur approximative de Pi.

Dans ce guide, les différentes implémentations parallèles de cet algorithme apparaissent là ou les différentes méthodes de programmation sont traitées.

1.4. Structure du document

Le reste de ce document est divisé en cinq parties. Les sections 2, 3, 4 et 5 correspondent aux trois différents types de configuration matérielle pouvant assumer le traitement en parallèle en utilisant Linux.

- La section 2 traite des systèmes Linux sur SMP, lesquels prennent directement en charge l'exécution MIMD en utilisant la mémoire partagée, même si les files de messages sont facilement mises en place, elles aussi. Bien que Linux sache gérer les configurations SMP jusqu'à 16 processeurs, la plupart des ordinateurs SMP de type PC sont dotés soit de deux, soit de quatre processeurs identiques.

- La section 3 traite des batteries d'ordinateurs en réseau (« *clusters* »), chaque machine fonctionnant sous Linux. Un *cluster* peut être utilisé comme un système de traitement en parallèle gérant directement l'exécution en MIMD et l'échange de messages, et peut-être même aussi la mémoire partagée logique. L'exécution SIMD simulée et la communication des fonctions d'agrégation peuvent aussi être prises en charge, selon le réseau exploité. Le nombre de processeurs compris dans un *cluster* peut s'étendre de deux à plusieurs milliers, la principale limitation étant le câblage physique du réseau. Dans certains cas, des machines de différents types peuvent être mélangées au sein d'un *cluster*. Par exemple, un réseau qui combinerait des Alpha DEC et des Pentium sous Linux serait appelé *cluster hétérogène*.

- La section 4 traite du SWAR, le « SIMD dans un registre ». C'est une forme très restrictive d'exécution en parallèle, mais d'un autre coté, c'est une possibilité intégrée aux processeurs ordinaires. Ces derniers temps, les extensions MMX et autres des processeurs modernes ont rendu cette approche encore plus efficace.

- La section 5 traite de l'utilisation de PC sous Linux comme hôtes pour des systèmes de calcul parallèle simples. Sous forme de carte d'extension ou de boîtiers externes, les processeurs auxiliaires peuvent apporter à des systèmes Linux une formidable puissance de traitement pour des applications spécifiques. Par exemple, des cartes ISA disponibles à peu de frais fournissent de multiples processeurs DSP, offrant plusieurs centaines de MégaFLOP aux calculs de grande envergure. En revanche, ces cartes ne sont *que* des processeurs. Elle n'embarquent généralement pas de système d'exploitation, de disque dur, ou de connecteur pour terminal de contrôle, et cætera. Pour rendre ces systèmes exploitables, l'« hôte » Linux doit fournir ces facilités.

La section finale de ce document couvre les aspects d'intérêt général concernant le traitement en parallèle sous Linux, non spécifique à l'une des approches listées ci-dessus.

En lisant ce document, gardez à l'esprit que nous n'avons pas tout testé, et que beaucoup de choses rapportées dans ce document ont toujours « un caractère expérimental » (une jolie manière de dire que cela ne fonctionne pas tout à fait comme espéré ;-)). Cela dit, le traitement en parallèle sous Linux est désormais exploitable, et un groupe incroyablement vaste de personnes travaille à le rendre encore meilleur.

L'auteur de la version originale de ce guide est le Dr (Ph.D) Hank Dietz, actuellement Professeur Associé de l'*Electrical and Computer Engineering* à l'université de Purdue, West Lafayette, IN, 47907-1285. Dietz est propriétaire des droits sur ce document, conformément aux règles du *Linux Documentation Project* (LDP). Bien qu'un effort ait été fait pour assurer l'exactitude de cette présentation, ni Dietz ni l'Université de Purdue ne peuvent être tenus responsables d'éventuels problèmes ou erreurs, et l'Université de Purdue n'endosse la responsabilité d'aucun produit ou travaux traités dans ce document.

1.5. Note du traducteur

Chers lecteurs, avant de poursuivre la lecture de ce guide, il est important de revenir, notament au vu de la date de publication de cette version française, sur plusieurs points :

- Le Professeur Henry G. Dietz (dit « Hank »), après avoir enseigné plusieurs années à l'Université de Purdue et y avoir développé la plupart de ce qui forme ce document, *mène aujourd'hui ses recherches à l'Université du Kentucky*. Son site personnel se trouve désormais ici: http://aggregate.org/hankd/. Cela signifie également que la plupart des références à l'Université de Purdue sont désormais caduques. Toutefois, un certain nombre de ces références ont été conservées en l'état dans ce guide, ce lorsque le contenu référencé était toujours disponible sur le site de l'Université sans avoir été transféré vers le nouveau site. En tout état de cause, dirigez-vous en priorité sur le site de l'Université du Kentucky pour tout contact ou pour obtenir les informations les plus récentes.

- La totalité des termes, notament techniques, employés dans ce documents ont été traduits en français, à quelques exceptions près. C'est par exemple le cas du mot « *cluster* », qui désigne en informatique la mise en parallèle de plusieurs machines individuelles et coordonnées de manière à les faire agir comme un seul super-ordinateur. Le terme français homologue est « grappe ». Toutefois, la fréquence à laquelle ce mot est employé tant dans ce document (un chapitre entier est consacré à ce sujet précis) que dans la communauté du traitement en parallèle en général est telle que le terme original a été conservé dans la présente version française. Dans le même esprit, la notion de « bande passante » se retrouve très fréquement tout au long de ce guide. C'est à la base un abus de langage, mais la popularité de cette formule est également suffisament grande pour la conserver en l'état.

- La version originale de ce document a été écrite en 1998, la version française est parue en 2004. Il va sans dire qu'au cours d'une aussi longue période, le paysage informatique a beaucoup évolué, spécialement en ce qui concerne le développement du noyau Linux. Certaines technologies réseau (telles que ATM, FireWire, ou Fiber Channel) ou de *clustering* (comme MOSIX), recensées comme indisponibles en 1998, ont depuis intégré le noyau, ou sont devenues disponibles. En revanche, il est très peu probable qu'une technologie connue pour fonctionner sous Linux lors de la rédaction de ce document soit devenue inutilisable depuis.

- Plus encore que celui de l'industrie informatique, le paysage du *World Wide Web* s'est transformé de façon à rendre la plupart des liens proposés obsolètes. Un effort a été fait pour assurer leur mise à jour ou leur remplacement, ainsi que la pertinence de leur contenu. En dépit de cela, un certain nombre d'entre eux, en particulier ceux dont les projets étaient hébergés sur les pages personnelles d'étudiants de grandes écoles, n'ont pu être corrigés et ont été retirés du document.

Malgré toutes ces réserves, les techniques couvertes par ce document sont suffisament générales pour rester valables au cours du temps et au travers des différents modèles de machines, et son contenu présente toujours un intérêt à la fois pédagogique et historique, qui restera encore longtemps profitable au lecteur. Tout ceci justifie une publication même tardive.

Enfin, le traducteur s'est efforcé de rendre le présent document aussi correct et fidèle à son original que possible, mais n'est pas infaillible. Tout signalement d'un contresens, d'une erreur technique, ou tout autre défaut de traduction sera apprécié à sa juste valeur à l'adresse suivante : <`dvanden-broeck CHEZ free POINT fr`>.

Bonne lecture !

2. Linux sur SMP

Ce document donne un bref aperçu de la manière dont on utilise le SMP sous Linux pour le traitement en parallèle. L'information la plus à jour concernant le SMP sous Linux est fort probablement disponible via la liste de diffusion du SMP Linux Project (N.D.T. : en anglais). Envoyez un courrier électronique à <`majordomo CHEZ vger POINT rutgers POINT edu`> avec le texte `subscribe linux-smp` pour rejoindre la liste.

Le SMP sous Linux fonctionne-t-il vraiment ? En juin 1996, j'ai fait l'achat d'un bi-Pentium 100MHz flambant neuf. Le système complet et assemblé, comprenant les deux processeurs, la carte-mère Asus, 256 kilo-octets de mémoire cache, 32 méga-octets de RAM, le disque dur d'1.6 giga-octet, le lecteur de CD-ROM 6X, une carte Stealth 64 et un moniteur 15'' Acer m'a coûté 1800 dollars. Cela ne fait que quelques centaines de dollars de plus qu'un système monoprocesseur. Pour faire fonctionner le SMP sous Linux, il a suffi d'installer le Linux monoprocesseur d'origine, de recompiler le noyau en décommentant la ligne `SMP=1` dans le *Makefile* (bien que je trouve le fait de mettre `SMP` à `1` un peu ironique ! ;-)), et d'informer `lilo` de l'existence du nouveau noyau. Ce système présente une stabilité et des performances suffisamment bonnes pour qu'il me serve depuis de station de travail principale. Pour résumer, le SMP sous Linux, ça fonctionne !

La question qui se présente alors est : existe-t-il suffisamment d'API de haut niveau permettant d'écrire et d'exécuter des programmes en parallèle et utilisant la mémoire partagée sous Linux SMP ? Courant 1996, il n'y en avait pas beaucoup. Les choses ont changé. Par exemple, il existe désormais une bibliothèque POSIX de gestion des *threads*[2 [p 71]] très complète.

Bien que les performances soient moins élevées que celles des mécanismes de mémoire partagée natifs, un système Linux sur SMP peut aussi utiliser la plupart des logiciels de traitement en parallèle initialement développés pour des *clusters* de stations de travail en utilisant la communication par *socket*. Les *sockets* (voir section 3.3) fonctionnent à l'intérieur d'une machine en SMP, et même dans un *cluster* de machines SMP reliées en réseau. Cependant, les *sockets* engendrent beaucoup de pertes en temps inutiles pour du SMP. Cela complique le problème car Linux SMP n'autorise en général qu'un seul processeur à la fois à se trouver dans le noyau et le contrôleur d'interruption est réglé de façon à ce que seul le processeur de *boot*[3 [p 71]] puisse traiter les interruptions. En dépit de cela, l'électronique de communication typique des systèmes SMP est tellement meilleure que la plupart des *clusters* en réseau que les logiciels pour *cluster* fonctionneront souvent mieux sur du SMP que sur le *cluster* pour lequel ils ont été conçus.

Le reste de cette section traite de l'électronique contrôlant le SMP, passe en revue les mécanismes Linux de base partageant de la mémoire à travers les différents processus d'un programme en parallèle, fait quelques remarques concernant l'atomicité, la volatilité, les verrous et les lignes de cache, et donne enfin des références vers d'autres ressources de traitement en parallèle à mémoire partagée.

2.1. L'électronique SMP

Bien que les systèmes SMP soit répandus depuis de nombreuses années, jusque très récemment, chaque machine tendait à implémenter les fonctions de base d'une manière suffisamment différente des autres pour que leur gestion par le système d'exploitation ne soit pas portable. Les choses ont changé avec la *Intel's MultiProcessor Specification* (Spécification MultiProcesseurs d'Intel) souvent désignée par *MPS*. La spécification MPS 1.4 est actuellement disponible sous forme de document PDF sur http://www.intel.com/design/intarch/MANUALS/242016.htm[4 [p 71]], mais gardez à l'esprit qu'Intel réorganise souvent son site web. Un large panel de constructeurs fabrique des systèmes conformes à MPS pouvant recevoir jusqu'à quatre processeurs, mais en théorie, MPS admet bien plus de processeurs.

Les seuls systèmes non MPS et non IA32 reconnus par Linux SMP sont les machines SPARC multiprocesseurs de Sun4m. Linux SMP prend aussi en charge la plupart des machines Intel conformes à MPS 1.1 ou 1.4, comptant jusqu'à 16 processeurs 486DX, Pentium, Pentium MMX, Pentium Pro ou Pentium II. Parmi les processeurs IA32 non pris en charge (N.D.T. : par le SMP), on trouve les Intel 386 et 486SX/SLC (l'absence de coprocesseur mathématique interfère sur les mécanismes du SMP) et les processeurs AMD et Cyrix (qui nécessitent des circuits de gestion du SMP différents et qui ne semblent pas être disponibles à l'heure où ce document est écrit).

Il est important de bien comprendre que les performances de différents systèmes conformes à MPS peuvent fortement varier. Comme l'on peut s'y attendre, une des causes de différence de performance est la vitesse du processeur : Une horloge plus rapide tend à rendre les systèmes plus rapides, et un processeur Pentium Pro est plus rapide qu'un Pentium. En revanche, MPS ne spécifie pas vraiment comment le matériel doit mettre en œuvre la mémoire partagée, mais seulement comment cette implémentation doit fonctionner d'un point de vue logiciel. Cela signifie que les performances dépendent aussi de la façon dont l'implémentation de la mémoire partagée interagit avec les caractéristiques de Linux SMP et de vos applications en particulier.

La principale différence entre les systèmes conformes à MPS réside dans la manière dont ils implémentent l'accès à la mémoire physiquement partagée.

2.1.1. Chaque processeur possède-t-il sa propre mémoire cache de niveau 2 (L2) ?

Certains systèmes MPS à base de Pentium, et tous les systèmes MPS Pentium Pro et Pentium II ont des mémoires cache L2 indépendantes (le cache L2 est embarqué dans le module des Pentium Pro et Pentium II). Les mémoires caches L2 dissociées sont généralement réputées augmenter les performances de l'ordinateur, mais les choses ne sont pas si évidentes sous Linux. La principale complication provient du fait que l'ordonnanceur de Linux SMP n'essaie pas de maintenir chaque processus sur le même processeur, concept connu sous le nom d'*affinité processeur*. Cela pourrait bientôt changer. Un débat a récemment eu lieu sur ce sujet dans la communauté des développeurs Linux SMP, sous le titre « *processor bindings* » (« associations de processeurs »). Sans affinité processeur, des caches L2 séparés peuvent introduire des délais non négligeables lorsqu'un processus se voit allouer une tranche de temps d'exécution sur un processeur qui n'est pas le même que celui sur lequel il s'exécutait juste avant.

Plusieurs systèmes relativement bon marché sont organisés de manière à ce que deux processeurs Pentium puissent partager la même mémoire cache L2. La mauvaise nouvelle, c'est que cela crée des conflits à l'utilisation de ce cache, qui dégradent sérieusement les performances lorsque plusieurs programmes séquentiels indépendants s'exécutent simultanément. La bonne nouvelle, c'est que bon nombre de programmes parallèles pourraient tirer profit de la mémoire cache partagée, car si les deux processeurs veulent accéder à la même ligne de mémoire partagée, seul un processeur doit aller la rapatrier dans le cache, et l'on évite des conflits de bus. Le manque d'affinité processeur peut aussi s'avérer moins désastreux avec un cache L2 partagé. Ainsi, pour les programmes parallèles, il n'est pas vraiment certain que partager la mémoire cache L2 soit si préjudiciable que l'on pourrait le penser.

À l'usage, notre bi-Pentium à mémoire cache partagée de 256Ko présente une vaste échelle de performances, dépendantes du niveau d'activité noyau requis. Au pire, le gain en vitesse n'atteint qu'un facteur de 1,2. En revanche, nous avons aussi constaté une accélération de 2,1 fois la vitesse d'origine, ce qui suggère que les calculs intensifs à base de SPMD tirent vraiment profit de l'effet d'« acquisition partagée » (« *shared fetch* »).

2.1.2. Configuration du bus ?

La première chose à dire est que la plupart des systèmes modernes relient le processeur à un ou plusieurs bus PCI qui à leur tour sont « pontés » vers un ou plusieurs bus ISA ou EISA. Ces ponts engendrent des temps de latence, et l'ISA comme l'EISA offrent généralement des bandes passantes plus réduites que le PCI (ISA étant le plus lent). C'est pourquoi les disques, cartes vidéos et autres périphériques de haute performance devraient en principe être connectés sur un bus PCI.

Bien qu'un système MPS puisse apporter un gain en vitesse honorable à plusieurs programmes parallèles de calcul intensif même avec un seul bus PCI, les opérations d'entrées/sorties, elles, ne sont pas meilleures que sur un système monoprocesseur. Elles sont peut-être même un peu moins bonnes à cause des conflits de bus entre les processeurs. Ainsi, si votre objectif est d'accélérer les entrées/sorties, prenez soin de choisir un système MPS comportant plusieurs bus PCI indépendants et plusieurs contrôleurs d'entrées/sorties (par exemple : plusieurs chaînes SCSI). Il vous faudra être prudent, et sûr que Linux reconnaît tout votre matériel. Gardez aussi à l'esprit le fait que Linux n'autorise qu'un seul processeur à la fois à entrer en mode noyau, aussi devrez-vous choisir des contrôleurs qui réduisent au minimum le temps noyau nécessaire à leurs opérations. Pour atteindre des performances vraiment très élevées, il se pourrait même qu'il vous faille envisager d'effectuer vous-même les opérations d'entrée/sortie de bas niveau directement depuis les processus utilisateurs, sans appel système… ce n'est pas forcément aussi difficile que cela en a l'air, et cela permet d'éviter de compromettre la sécurité du système (voir la section 3.3 pour une description des techniques de base).

Il est important de remarquer que la relation entre vitesse du bus et vitesse du processeur est devenue très floue ces dernières années. Bien que la plupart des systèmes utilisent maintenant la même fréquence de bus PCI, il n'est pas rare de trouver un processeur rapide apparié avec un bus lent. L'exemple classique est celui du Pentium 133 qui utilise en général un bus plus rapide que celui du Pentium 150, produisant des résultats étranges sur les logiciels bancs de tests (« *benchmarks* »). Ces effets sont amplifiés sur les systèmes SMP, où il est encore plus important d'utiliser un bus rapide.

2.1.3. Interfoliage de la mémoire et technologie DRAM

L'interfoliage de la mémoire n'a en fait absolument rien à voir avec le MPS, mais vous verrez souvent cette mention accompagner les systèmes MPS car ceux-ci sont typiquement gourmands en bande passante mémoire. Concrètement, l'interfoliage en deux ou en quatre voies organise la RAM de façon à ce que l'accès à un bloc de mémoire se fasse au travers de plusieurs bancs de RAM plutôt qu'un seul. Ceci accélère grandement les accès à la mémoire, particulièrement en ce qui concerne le chargement et l'enregistrement du contenu des lignes de cache.

Il faut toutefois souligner que ce fait n'est pas aussi évident qu'il y parait, car la DRAM EDO et les différentes technologies mémoire tendent à optimiser ce genre d'opérations. Un excellent aperçu des différentes technologies DRAM est disponible sur http://www.pcguide.com/ref/ram/tech.htm.

Ainsi, par exemple, mieux vaut-il avoir de la mémoire DRAM EDO interfoliée à 2 voies, ou de la mémoire SDRAM non interfoliée ? C'est une très bonne question et la réponse n'est pas simple, car la mémoire interfoliée comme les technologies DRAM exotiques ont tendance à être coûteuses. Le même investissement en mémoire plus ordinaire vous apporte en général une mémoire centrale bien plus vaste. Même la plus lente des mémoire DRAM reste autrement plus rapide que la mémoire virtuelle par fichier d'échange…

2.2. Introduction à la programmation en mémoire partagée

Okay, donc vous avez décidé que le traitement en parallèle sur SMP, c'est génial… Par quoi allez-vous commencer ? Eh bien, la première étape consiste à en apprendre un peu plus sur le fonctionnement réel de la communication par mémoire partagée.

A première vue, il suffit qu'un processeur range une valeur en mémoire et qu'un autre la lise. Malheureusement, ce n'est pas aussi simple. Par exemple, les relations entre processus et processeurs sont très floues. En revanche, si nous n'avons pas plus de processus actifs que de processeurs, les termes sont à peu près interchangeables. Le reste de cette section résume brièvement les cas de figure typiques qui peuvent poser de sérieux problèmes, si vous ne les connaissiez pas déjà : les deux différents modèles utilisés pour déterminer ce qui est partagé, les problèmes d'atomicité, le concept de volatilité, les instructions de verrouillage matériel, les effets de la ligne de cache, et les problèmes posés par l'ordonnanceur de Linux.

2.2.1. Partage Intégral contre Partage Partiel

Il existe deux modèles fondamentaux couramment utilisés en programmation en mémoire partagée : le *partage intégral* et le *partage partiel*. Ces modèles permettent tous deux aux processeurs de communiquer en chargeant et rangeant des données depuis et dans la mémoire. La différence réside dans le fait que le partage intégral place toutes les structures en mémoire partagée, quand le partage partiel, lui, distingue les structures qui sont potentiellement partageables et celles qui sont *privées*, propres à un seul processeur (et oblige l'utilisateur à classer explicitement ses structures dans l'une de ces catégories).

Alors quel modèle de partage mémoire faut-il utiliser ? C'est surtout une affaire de chapelle. Beaucoup de gens aiment le partage intégral car ils n'ont pas spécialement besoin d'identifier les structures qui doivent être partagées au moment de leur déclaration. On place simplement des verrous sur les objets auxquels l'accès peut créer des conflits, pour s'assurer qu'un seul processeur (ou processus) y accède à un moment donné. Mais là encore, ce n'est pas aussi simple… aussi beaucoup d'autres gens préfèrent, eux, le modèle relativement sûr du partage partiel.

2.2.1.1. Partage intégral

Le bon coté du partage intégral est que l'on peut aisément reprendre un programme séquentiel existant et le convertir progressivement en programme parallèle en partage intégral. Vous n'avez pas à déterminer au préalable les données qui doivent être accessibles aux autres processeurs.

Posé simplement, le principal problème avec le partage intégral vient du fait qu'une action effectuée par un processeur peut affecter les autres processeurs. Ce problème ressurgit de deux manières :

- Plusieurs bibliothèques utilisent des structures de données qui ne sont tout simplement pas partageables. Par exemple, la convention UNIX stipule que la plupart des fonctions peuvent renvoyer un code d'erreur dans une variable appelée `errno`. Si deux processus en partage intégral font des appels divers, ils vont interférer l'un sur l'autre car ils partagent la même variable `errno`. Bien qu'il existe désormais une bibliothèque qui règle le problème de cette variable, ce problème se présente toujours dans la plupart des bibliothèques comme par exemple X-Window qui, à moins de prendre des précautions très spéciales, ne fonctionnera pas si différents appels sont passés depuis différents processus en partage intégral.

- En temps normal, un programme qui utilise un pointeur ou un index défaillant provoque au pire l'arrêt du processus qui contient le code corrompu. Il peut même générer un fichier `core` vous renseignant sur les conditions dans lesquelles se sont déroulés les événements. En programmation parallèle à partage intégral, il est fort probable que les accès illégaux provoquent la *fin d'un processus qui n'est pas le fautif*, rendant la localisation et la correction de l'erreur quasiment impossibles.

Aucun de ces deux problèmes n'est courant dans le cas du partage partiel, car seules sont partagées les structures explicitement marquées comme telles. De plus, il est trivial que le partage intégral ne peut fonctionner que si les processeurs exécutent exactement la même image en mémoire. On ne peut pas utiliser le partage intégral entre des images de code différentes (c'est-à-dire que vous pourrez travailler en SPMD, mais pas d'une manière générale en MIMD).

Les supports de programmation en partage intégral existent le plus couramment sous la forme de *bibliothèques de* threads. Les *threads* sont essentiellement des processus « allégés » dont l'exécution peut ne pas être planifiée comme celle des processus UNIX normaux et qui, c'est le plus important, partagent tous la même page mémoire. L'adaptation des Pthreads POSIX a fait l'objet de nombreux efforts. La grande question est : ces adaptations parallélisent-elles les *threads* d'un programme en environnement Linux SMP (idéalement, en attribuant un processeur à chaque *thread*) ?. L'API POSIX ne l'impose pas, et certaines versions comme *PCthreads* semblent ne pas implémenter une exécution en parallèle des *threads* : tous les *threads* d'un programme sont conservés à l'intérieur d'un seul processus Linux.

La première bibliothèque de *threads* à avoir pris en charge le parallélisme sous Linux SMP fut la désormais quelque peu obsolète bibliothèque *bb_thread*, une toute petite bibliothèque qui utilisait l'appel Linux `clone()` pour donner naissance à de nouveaux processus Linux, planifiés indépendamment les uns des autres, tous partageant un même espace d'adressage. Les machines Linux SMP peuvent lancer plusieurs de ces « *threads* » car chaque « *thread* » est un processus Linux à part entière. L'inconvénient, c'est que l'on ne peut obtenir l'ordonnancement « poids-plume » apportée par les bibliothèques de *threads* d'autres systèmes d'exploitation. La bibliothèque utilisait un peu de code assembleur intégré dans un code source en langage C pour mettre en place un bloc de mémoire pour la pile de chaque *thread* et fournir des fonctions d'accès atomiques à un tableau de verrous (les *mutex*). Sa documentation se résumait à un fichier `LISEZMOI` et à un court programme d'exemple.

Plus récemment, une version de *threads* POSIX utilisant `clone()` a été développée. Cette bibliothèque, LinuxThreads, est clairement la bibliothèque en partage intégral favorite pour l'utilisation sous Linux SMP. Les *threads* POSIX sont bien documentés, et les documents LinuxThreads README et LinuxThreads FAQ sont vraiment très bien réalisés. A présent, le principal problème est que les *threads* POSIX ont encore beaucoup de détails à régler, et que LinuxThread est toujours un projet en cours d'évolution. D'autre part, les *threads* POSIX ont évolué pendant dans leur phase de standardisation, aussi devrez-vous être prudent pour ne pas développer en suivant une version obsolète du standard.

2.2.1.2. Partage Partiel

Le Partage Partiel consiste réellement à « ne partager que ce qui doit être partagé ». Cette approche est valable pour le MIMD en général (et pas simplement le SPMD) à condition de prendre soin d'allouer les objets partagés aux mêmes endroits dans le plan mémoire de chaque processeur. Plus important encore, le partage partiel facilite l'estimation et l'ajustage des performances, le débogage des sources, et cætera. Les seuls problèmes sont :

- Déterminer à l'avance ce qui doit être partagé peut s'avérer difficile.
- L'allocation d'objets dans la mémoire partagée peut en fait se révéler malaisé, spécialement en ce qui concerne tout ce qui aurait du être déclaré dans la pile. Par exemple, il peut être nécessaire d'allouer explicitement des objets partagés dans des segments de mémoire séparés, nécessitant des routines d'allocation mémoire séparées, et impliquant l'ajout de pointeurs et d'indirections supplémentaires à chaque référence.

Actuellement, il existe deux mécanismes similaires permettant aux groupes de processus sous Linux de posséder des espaces mémoire indépendants, mais de tous partager un unique et relativement étroit segment de mémoire. En supposant que vous n'ayez pas bêtement exclu l'option « System V IPC » lorsque que vous avez configuré votre système Linux (N.D.T. : ici à la recompilation du noyau), Linux gère un mécanisme très portable devenu célèbre sous le nom de « mémoire partagée System V ». L'autre alternative est une fonction de projection en mémoire dont l'implémentation varie grandement selon le système UNIX utilisé : L'appel système `mmap`. Vous pouvez — et devriez — apprendre le fonctionnement de ces primitives au travers des pages du manuel (*man pages*)… mais vous trouverez quand même un rapide aperçu de chacune d'elles dans les sections 2.5 et 2.6 pour vous aider à démarrer.

2.2.2. Atomicité et ordonnancement

Que vous utilisiez l'un ou l'autre des modèles cités ci-dessus, le résultat est à peu près le même : vous obtenez un pointeur sur un bloc de mémoire en lecture/écriture accessible par tous les processus de votre programme en parallèle. Cela signifie-t-il que je peux laisser mes programmes accéder aux objets partagés comme s'ils se trouvaient en mémoire locale ordinaire ? Pas tout à fait…

L'*atomicité* désigne une opération sur un objet effectuée en une séquence indivisible et ininterruptible. Malheureusement, les accès à la mémoire partagée n'impliquent pas que les toutes les opérations sur les données de cette mémoire se fassent de manière atomique. A moins de prendre des précautions spéciales, seules les opérations de lecture ou d'écriture s'accomplissant en une seule transaction sur le bus (c'est-à-dire alignées sur une adresse multiple de 8, 16 ou 32 bits, à l'exclusion des opérations 64 bits ou mal alignées) sont atomiques. Pire encore, les compilateurs « intelligents » comme GCC font souvent des optimisations qui peuvent éliminer les opérations mémoire nécessaires pour s'assurer que les autres processeurs puissent voir ce que le processeur concerné a fait. Heureusement, ces problèmes ont tous deux une solution… en acceptant seulement de ne pas se soucier de la relation entre l'efficacité des accès mémoire et la taille de la ligne de cache.

En revanche, avant de traiter de ces différents cas de figure, il est utile de préciser que tout ceci part du principe que les références à la mémoire pour chaque processeur se produisent dans l'ordre où elles ont été programmées. Le Pentium fonctionne de cette manière, mais les futurs processeurs d'Intel pourraient ne pas le faire. Aussi, quand vous développerez sur les processeurs à venir, gardez à l'esprit qu'il pourrait être nécessaire d'encadrer les accès à la mémoire avec des instructions provoquant l'achèvement de toutes les accès à la mémoire en suspens, provoquant ainsi leur mise en ordre. L'instruction `CPUID` semble provoquer cet effet.

2.2.3. Volatilité

Pour éviter que l'optimiseur du GCC ne conserve les valeurs de la mémoire partagée dans les registres de processeur, tous les objets en mémoire partagée doivent être déclarés avec l'attribut `volatile`. Tous les accès en lecture ou écriture ne nécessitant l'accès qu'à un seul mot se feront alors de manière atomique. Par exemple, en supposant que *p* est un pointeur sur un entier, et que ce pointeur comme l'entier qu'il pointe se trouvent en mémoire partagée, la déclaration en C ANSI ressemblera à :

```
volatile int * volatile p;
```

Dans ce code, le premier `volatile` concerne l'`int` que `p` pointe éventuellement, quand le second `volatile` s'applique au pointeur lui-même. Oui, c'est ennuyeux, mais c'est le prix à payer pour que GCC puisse faire des optimisations vraiment puissantes. En théorie, l'option `-traditional` devrait suffire à produire du code correct au prix de quelques optimisations, car le standard C K&R (N.D.T. : Kernigan & Ritchie) pré norme ANSI établit que toutes les variables sont volatiles si elles ne sont pas explicitement déclarées comme `register`. Ceci étant dit, si vos compilations GCC ressemblent à `cc -O6 …`, vous n'aurez réellement besoin de déclarer les choses comme étant volatiles qu'aux endroits où c'est nécessaire.

Un rumeur a circulé à propos du fait que les verrous écrits en assembleur signalés comme modifiant tous les registres du processeur provoquaient de la part du compilateur GCC l'enregistrement adéquat de toutes les variables en suspens, évitant ainsi le code compilé « inutile » associé aux objets déclarés `volatile`. Cette astuce semble fonctionner pour les variables globales statiques avec la version 2.7.0 de GCC… En revanche, ce comportement n'est *pas* une recommandation du standard C ANSI. Pire encore, d'autres processus n'effectuant que des accès en lecture pourraient conserver éternellement les valeurs dans des registres, et ainsi ne *jamais* s'apercevoir que la vraie valeur stockée en mémoire partagée a en fait changé. En résumé, développez comme vous l'entendez, mais seules les variables déclarées `volatile` offrent un fonctionnement normal *garanti*.

Notez qu'il est possible de provoquer un accès volatile à une variable ordinaire en utilisant un transtypage (« *casting* ») imposant l'attribut `volatile`. Par exemple, un `int i;` ordinaire peut être référencé en tant que volatile par `*((volatile int *) &i);` . Ainsi, vous pouvez forcer la volatilité et les coûts supplémentaires qu'elle engendre seulement aux endroits où elle est critique.

2.2.4. Verrous (*Locks*)

Si vous pensiez que `++i;` aurait toujours incrémenté une variable `i` sans problème, vous allez avoir une mauvaise surprise : même codées en une seule instruction, le chargement et l'enregistrement du résultat sont deux transactions mémoire séparées, et d'autres processeurs peuvent accéder à `i` entre ces deux transactions. Par exemple, deux processus effectuant chacun l'instruction `++i;` pourraient n'incrémenter la variable `i` que d'une unité et non deux. Selon le « Manuel de l'Architecture et de la Programmation » du Pentium d'Intel, le préfixe `LOCK` peut être employé pour s'assurer que chacune des instructions suivantes soit atomique par rapport à l'adresse mémoire à laquelle elles accèdent :

```
BTS, BTR, BTC                     mem, reg/imm
XCHG                              reg, mem
XCHG                              mem, reg
ADD, OR, ADC, SBB, AND, SUB, XOR  mem, reg/imm
NOT, NEG, INC, DEC                mem
CMPXCHG, XADD
```

En revanche, il n'est pas conseillé d'utiliser toutes ces opérations. Par exemple, `XADD` n'existait même pas sur 386, aussi l'employer en programmation peut poser des problèmes de portabilité.

L'instruction `XCHG` engendre *toujours* un verrou, même sans le préfixe `LOCK`, et est ainsi et indiscutablement l'opération atomique favorite pour construire d'autres opérations atomiques de plus haut niveau comme les sémaphores et les files d'attente partagées. Bien sûr, on ne peut pas demander à GCC de générer cette instruction en écrivant simplement du code C. Il vous faudra à la place écrire un peu de code assembleur en ligne[5 [p 71]]. En prenant un objet volatile *obj* et un registre du processeur *reg*, tous deux de type `word` (longs de 16 bits), le code assembleur GCC sera :

```
__asm__ __volatile__ ("xchgl %1,%0"
                      :"=r" (reg), "=m" (obj)
                      :"r" (reg), "m" (obj));
```

Quelques exemples de programmes assembleur en ligne utilisant des opérations bit-à-bit pour réaliser des verrous sont disponibles dans le code source de la bibliothèque bb_threads.

Il est toutefois important de se souvenir que faire des transactions mémoire atomiques a un coût. Une opération de verrouillage engendre des délais supplémentaires assez importants et peut retarder l'activité mémoire d'autres processeurs, quand des références ordinaires auraient utilisé le cache local. Les meilleures performances s'obtiennent en utilisant les opérations atomiques aussi *peu* souvent que possible. De plus, ces instructions atomiques IA32 ne sont évidement pas portables vers d'autres systèmes.

Il existe plusieurs alternatives permettant aux instructions ordinaires d'être utilisées pour mettre en œuvre différents types de synchronisation, y compris l'*exclusion mutuelle*, qui garantit qu'au plus un seul processeur met à jour un objet partagé donné à un moment précis. La plupart des manuels des différents systèmes d'exploitation traitent d'au moins une de ces techniques. On trouve un très bon exposé sur le sujet dans la quatrième édition des *Operating System Concepts* (Principes des Systèmes d'Exploitation), par Abraham Silberschatz et Peter B. Galvin, ISBN 0-201-50480-4.

2.2.5. Taille de la ligne de cache

Encore une chose fondamentale concernant l'atomicité et qui peut avoir des conséquence dramatiques sur les performances d'un SMP : la taille de la ligne de cache. Même si le standard MPS impose que les références soient cohérentes quelque soit le cache utilisé, il n'en reste pas moins que lorsque qu'un processeur écrit sur une ligne particulière de la mémoire, chaque copie en cache de l'ancienne ligne doit être invalidée ou mise à jour. Ceci implique que si au moins deux processeurs écrivent chacun sur des portions différentes de la ligne de cache, cela peut provoquer un trafic important sur le bus et le cache, pour au final transférer la ligne depuis le cache vers le cache. Ce problème est connu sous le nom de *faux partage* (« *false sharing* »). La solution consiste uniquement à *organiser les données de telle manière que ce que les objets auxquels on accède en parallèle proviennent globalement de différentes lignes de cache pour chaque processus*.

Vous pourriez penser que le faux partage n'est pas un problème quand on utilise un cache de niveau 2 partagé, mais souvenez-vous qu'il existe toujours des caches de niveau 1 séparés. L'organisation du cache et le nombre de niveaux séparés peut varier, mais la ligne de cache de premier niveau d'un Pentium est longue de 32 octets, et le cache externe typique tourne autour de 256 octets. Supposons que les adresses (physiques ou logiques) de deux objets soient *a* et *b*, et que la taille de la ligne de cache soit *c*, que nous admettrons être une puissance de 2. Pour être très précis, si `((int) a) & ~(c-1)` est égal à `((int) b) & ~(c-1)`, alors les deux références se trouvent dans la même ligne de cache. Une règle plus simple consiste à dire que si deux objets référencés en parallèle sont éloignés d'au moins *c* octets, ils devraient se trouver dans des lignes de cache différentes.

2.2.6. Les problèmes de l'ordonnanceur de Linux

Bien que tout l'intérêt d'utiliser de la mémoire partagée pour les traitements en parallèle consiste à éviter les délais dus au système d'exploitation, ces délais peuvent parfois provenir d'autres choses que les communications en elles-mêmes. Nous avons déjà remarqué que le nombre de processus que l'on devrait créer doit être inférieur ou égal au nombre de processeurs de la machine. Mais comment décide-t-on exactement du nombre de processus à créer ?

Pour obtenir les meilleures performances, *le nombre de processus de votre programme en parallèle doit être égal au nombre de processus qui peuvent être exécutés simultanément, chacun sur son processeur*. Par exemple, si un système SMP à quatre processeurs héberge un processus très actif pour un autre usage (par exemple un serveur *web*), alors votre programme en parallèle ne devra utiliser que trois processus. Vous pouvez vous faire une idée générale du nombre de processus actifs exécutés sur votre système en consultant la « charge système moyenne » (« *load average* ») mise en évidence par la commande `uptime`.

Vous pouvez en outre « pousser » la priorité de vos processus de votre programme parallèle en utilisant, par exemple, la commande `renice` ou l'appel système `nice()`. Vous devez être privilégié [6 [p 71]] pour augmenter la priorité d'un processus. L'idée consiste simplement à éjecter les autres programmes des autres processeurs pour que votre programme puisse être exécuté sur tous les processeurs simultanément. Ceci peut être effectué de manière un peu plus explicite en utilisant la version prototype de Linux SMP disponible sur http://www.fsmlabs.com/products/openrtlinux/ et qui propose un ordonnanceur en temps réel (N.D.T. : il existe désormais un guide consacré à RTLinux, accessible en ligne : RTLinux HOWTO).

Si vous n'êtes pas le seul utilisateur employant votre système SMP comme une machine en parallèle, il se peut que vous entriez en conflit avec les autres programmes en parallèle essayant de s'exécuter simultanément. La solution standard est l'*ordonnancement de groupe* (« *gang scheduling* »), c'est-à-dire la manipulation de la priorité d'ordonnancement de façon à ce que seuls les processus d'un seul programme en parallèle s'exécutent à un moment donné. Il est bon de rappeler, en revanche, que multiplier les parallélismes tend à réduire les retours et que l'activité de l'ordonnanceur introduit des délais supplémentaires. Ainsi, par exemple, il sera sûrement préférable, pour une machine à quatre processeurs, d'exécuter deux programmes contenant chacun deux processus, plutôt que d'ordonnancer en groupe deux programmes de quatre processus chacun.

Il y a encore une chose dont il faut tenir compte. Supposons que vous développiez un programme sur une machine très sollicitée le jour, mais disponible à cent pour cent pendant la nuit pour le traitement en parallèle. Il vous faudra écrire et tester votre code dans les conditions réelles, donc avec tous ses processus lancés, même en sachant que des tests de jour risquent d'être lents. Ils seront en fait *très* lents si certains de vos processus sont en état d'*attente active* (« *busy waiting* »)[7 [p 71]], guettant le changement de certaines valeurs en mémoire partagée, changement censé être provoqué par d'autres processus qui ne sont pas exécutés (sur d'autres processeurs) au même moment. Ce même problème apparaît lorsque l'on développe et que l'on teste un programme sur un système monoprocesseur.

La solution consiste à intégrer des appels système à votre code là où il peut se mettre en boucle en attendant une action d'un autre processeur, pour que Linux puisse donner une chance de s'exécuter à un autre processus. J'utilise pour cela une macro en langage C, appelons-la `IDLE_ME` (N.D.T. : `MetsMoiEnAttente`) : pour faire un simple test, compilez votre programme par « `cc -DIDLE_ME=usleep(1);` ... ». Pour produire un exécutable définitif, utilisez « `cc -DIDLE_ME={}` ... ». L'appel `usleep(1)` réclame une pause d'une microseconde, qui a pour effet de permettre à l'ordonnanceur de Linux de choisir un nouveau processus à exécuter sur ce processeur. Si le nombre de processus dépasse le nombre de processeurs disponibles, il n'est pas rare de voir des programmes s'exécuter dix fois plus rapidement avec `usleep(1)` que sans.

2.3. bb_threads

La bibliothèque bb_threads (*"Bare Bones" threads*) est une bibliothèque remarquablement simple qui fait la démonstration de l'utilisation de l'appel système Linux `clone()`. Le fichier `tar.gz` n'occupe que 7 ko ! Bien que cette bibliothèque ait été rendue pour l'essentiel obsolète par la bibliothèque LinuxThreads, traitée dans la section 2.4, bb_threads reste utilisable, et est suffisamment simple et peu encombrante pour former une bonne introduction à la gestion des *threads* sous Linux. Il est beaucoup moins effrayant de se lancer dans la lecture de ce code source que dans celui de LinuxThreads. En résumé, la bibliothèque bb_threads forme un bon point de départ, mais n'est pas vraiment adaptée à la réalisation de grands projets.

La structure de base des programmes utilisant la bibliothèque bb_threads est la suivante :

1. Lancez le programme en tant que processus unique.

2. Il vous faudra estimer l'espace maximum dans la pile qui sera nécessaire à chaque *thread*. Prévoir large est relativement sage (c'est à çà que sert la mémoire virtuelle ;-), mais souvenez-vous que *toutes* les piles proviennent d'un seul espace d'adressage virtuel, aussi voir trop grand n'est pas une idée formidable. La démo suggère 64Ko. Cette taille est fixée à *b* octets par `bb_threads_stacksize(b)`.

3. L'étape suivante consiste à initialiser tous les verrous dont vous aurez besoin. Le mécanisme de verrouillage intégré à cette bibliothèque numérote les verrous de 0 à `MAX_MUTEXES`, et initialise un verrou *i* par `bb_threads_mutexcreate(i)`.

4. La création d'un nouveau *thread* s'effectue en appelant une routine de la bibliothèque recevant en arguments la fonction que le nouveau *thread* doit exécuter, et les arguments qui doivent lui être transmis. Pour démarrer un nouveau *thread* exécutant la fonction *f* de type `void` et attendant un argument *arg*, l'appel ressemblera à `bb_threads_newthread (f, &arg)`, où *f* devra être déclaré comme suit :

```
void f (void *arg, size_t dummy)
```

Si vous avez besoin de passer plus d'un argument à votre fonction, utilisez un pointeur sur une structure contenant les valeurs à transmettre.

5. Lancement du code en parallèle, en prenant soin d'utiliser `bb_threads_lock(n)` et `bb_threads_unlock(n)` où *n* est un entier indiquant le verrou à utiliser. Notez que les opérations de verrouillage et déverrouillage sont des opérations de blocage[8 [p 71]] très primaires et utilisant des instructions atomiques de verrouillage du bus, lesquelles peuvent causer des interférences d'accès à la mémoire, et qui n'essaient en aucun cas d'agir « proprement ». Le programme de démonstration fourni avec bb_threads n'utilisait pas correctement les verrous pour empêcher `printf()` d'être exécuté depuis les fonctions `fnn` et `main`, et à cause de cela, la démo ne fonctionne pas toujours. Je ne dis pas cela pour démolir la démo, mais plutôt pour bien mettre en évidence le fait que ce travail comporte *beaucoup de pièges*. Ceci dit, utiliser LinuxThreads ne se révèle que légèrement plus facile.

6. Lorsqu'un *thread* exécute `return`, il détruit le processus… mais la pile locale n'est pas automatiquement désallouée. Pour être plus précis, Linux ne gère pas la désallocation, et l'espace mémoire n'est pas automatiquement rendu à la liste d'espace libre de `malloc()`. Aussi, le processus parent doit-il récupérer l'espace mémoire de chaque processus fils mort par `bb_threads_cleanup(wait(NULL))`.

Le programme suivant, écrit en langage C, utilise l'algorithme traité dans la section 1.3 pour calculer la valeur de Pi en utilisant deux *threads* bb_threads.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "bb_threads.h"

volatile double pi = 0.0;
volatile int intervalles;
volatile int pids[2];      /* Numéros de processus Unix des threads */

void
do_pi(void *data, size_t len)
{
  register double largeur, sommelocale;
  register int i;
  register int iproc = (getpid() != pids[0]);

  /* Fixe la largeur des intervalles */
  largeur = 1.0 / intervalles;

  /* Effectue les calculs locaux */
  sommelocale = 0;
  for (i=iproc; i<intervalles; i+=2) {
    register double x = (i + 0.5) * largeur;
    sommelocale += 4.0 / (1.0 + x * x);
  }
  sommelocale *= largeur;

  /* Obtention des permissions, mise à jour de Pi, et déverrouillage */
  bb_threads_lock(0);
  pi += sommelocale;
```

```
  bb_threads_unlock(0);
}

int
main(int argc, char **argv)
{
  /* Récupère le nombre d'intervalles */
  intervalles = atoi(argv[1]);

  /* Fixe la taille de la pile, et crée le verrou */
  bb_threads_stacksize(65536);
  bb_threads_mutexcreate(0);

  /* crée deux threads ... */
  pids[0] = bb_threads_newthread(do_pi, NULL);
  pids[1] = bb_threads_newthread(do_pi, NULL);

  /* nettoie derrière les deux threads */
  /* (forme ainsi une barrière de synchro) */

  bb_threads_cleanup(wait(NULL));
  bb_threads_cleanup(wait(NULL));

  /* Affiche le résultat */
  printf("Estimation de la valeur de Pi: %f\n", pi);

  /* Sortie avec code de SUCCES */
  exit(0);
}
```

2.4. LinuxThreads

LinuxThreads (http://pauillac.inria.fr/~xleroy/linuxthreads/) est une implémentation assez complète et bien construite en accord avec le standard de *threads* POSIX 1003.1c. Contrairement aux autres adaptations d'implémentations de *threads* POSIX, LinuxThreads utilise également l'appel `clone()` du noyau Linux, déjà employé par bb_threads. La compatibilité POSIX implique qu'il est relativement aisé de faire l'adaptation de certaines applications provenant d'autres systèmes, et différents tutoriels et leur support sont disponibles. Bref, c'est incontestablement la bibliothèque à utiliser pour développer des applications *multi-threads* à grande échelle sous Linux.

La structure de base d'un programme utilisant LinuxThreads suit ce modèle :

1. Lancement du programme en tant que processus unique.
2. Initialisation de tous les verrous dont vous aurez besoin. Contrairement aux verrous de bb_threads qui sont identifiés par des numéros, les verrous POSIX sont déclarés comme des variables de type `pthread_mutex_t lock`. Utilisez `pthread_mutex_init(&lock,val)` pour initialiser chacun des verrous que vous utiliserez.
3. Comme avec bb_threads, la création d'un nouveau *thread* se fait par l'appel d'une fonction de la bibliothèque admettant des arguments spécifiant à leur tour la fonction que le nouveau *thread* doit exécuter et les arguments que celle-ci reçoit. Cependant, POSIX impose à l'utilisateur la déclaration d'une variable de type `pthread_t` pour identifier chaque *thread*. Pour créer un *thread* `pthread_t thread` exécutant la fonction `f()`, on appelle `pthread_create(&thread,NULL,f,&arg)`.

4. Lancement de la partie parallèle du programme, en prenant soin d'utiliser `pthread_mutex_lock(&lock)` et `pthread_mutex_unlock(&lock)` comme il se doit.

5. Utilisation de `pthread_join(thread,&retval)` après chaque *thread* pour tout nettoyer.

6. Utilisation de `-D_REENTRANT` à la compilation de votre programme en C.

Voici l'exemple du calcul de Pi en parallèle, s'appuyant sur LinuxThreads. L'algorithme de la section 1.3 est utilisé et, comme pour l'exemple de bb_threads, deux *threads* s'exécutent en parallèle.

```
#include <stdio.h>
#include <stdlib.h>
#include "pthread.h"

volatile double pi = 0.0;    /* Approximation de pi (partagée) */
pthread_mutex_t pi_lock;     /* Verrou de la variable ci-dessous */
volatile double intervalles; /* Combien d'intervalles ? */

void *
process(void *arg)
{
  register double largeur, sommelocale;
  register int i;
  register int iproc = (*((char *) arg) - '0');

  /* Fixe la largeur */
  largeur = 1.0 / intervalles;

  /* Fais les calculs locaux */
  sommelocale = 0;
  for (i=iproc; i<intervalles; i+=2) {
    register double x = (i + 0.5) * largeur;
    sommelocale += 4.0 / (1.0 + x * x);
  }
  sommelocale *= largeur;

  /* Verrouille la variable pi en vue d'une mise à jour,
     effectue la mise à jour, puis déverrouille Pi. */

  pthread_mutex_lock(&pi_lock);
  pi += sommelocale;
  pthread_mutex_unlock(&pi_lock);

  return(NULL);
}

int
main(int argc, char **argv)
{
  pthread_t thread0, thread1;
  void * retval;

  /* Récupère le nombre d'intervalles */
  intervalles = atoi(argv[1]);

  /* Initialise un verrou sur pi */
  pthread_mutex_init(&pi_lock, NULL);

  /* Crée les deux threads */
```

```
  if (pthread_create(&thread0, NULL, process, "0") ||
      pthread_create(&thread1, NULL, process, "1")) {
    fprintf(stderr, "%s: Création des threads impossible\n", argv[0]);
    exit(1);
  }

  /* « Joint » (détruit) les deux threads */
  if (pthread_join(thread0, &retval) ||
      pthread_join(thread1, &retval)) {
    fprintf(stderr, "%s: Erreur à la fusion des threads\n", argv[0]);
    exit(1);
  }

  /* Affiche le résultat */
  printf("Estimation de la valeur de Pi: %f\n", pi);

  /* Sortie */
  exit(0);
}
```

2.5. La mémoire partagée de System V

La gestion des IPC (*Inter-Process Communication*) System V s'effectue au travers d'un certain nombre d'appels système fournissant les mécanismes des files de message, des sémaphores et de la mémoire partagée. Bien sûr, ces mécanismes ont été initialement conçus pour permettre à plusieurs processus de communiquer au sein d'un système monoprocesseur. Cela signifie néanmoins que ces mécanismes devraient aussi fonctionner dans un système Linux SMP, quelque soit le nombre de processeurs.

Avant d'aller plus loin dans l'utilisation de ces appels, il est important de comprendre que même s'il existe des appels IPC System V pour des choses comme les sémaphores et la transmission de messages, vous ne les utiliserez probablement pas. Pourquoi ? Parce ces fonctions sont généralement lentes et sérialisées sous Linux SMP. Inutile de s'étendre.

La marche à suivre standard pour créer un groupe de processus partageant l'accès à un segment de mémoire partagée est la suivante.

1. Lancement du programme en tant que processus unique.

2. En temps normal, chaque instance de votre programme en parallèle devra avoir son propre segment de mémoire partagée, aussi vous faudra-t-il appeler `shmget()` pour créer un nouveau segment de la taille souhaitée. Mais d'autre part, cet appel peut être utilisé pour récupérer l'identifiant d'un segment de mémoire partagée déjà existant. Dans les deux cas, la valeur de retour est soit l'identifiant du segment de mémoire partagée, soit -1 en cas d'erreur. Par exemple, pour créer un segment de mémoire partagée long de *b* octets, on passe un appel ressemblant à `shmid = shmget(IPC_PRIVATE, b, (IPC_CREAT | 0666))`.

3. L'étape suivante consiste à attacher ce segment de mémoire partagée au processus, c'est-à-dire l'ajouter à son plan mémoire. Même si l'appel `shmat()` permet au programmeur de spécifier l'adresse virtuelle à laquelle le segment doit apparaître, cette adresse doit être alignée sur une page (plus précisément être un multiple de la taille d'une page renvoyée par `getpagesize()`, correspondant à 4096 octets), et recouvrera (prendra le pas sur) tout segment de mémoire s'y trouvant déjà. Ainsi est-il plus sage de laisser le système choisir une adresse. Dans les deux cas, la valeur de retour est un pointeur sur l'adresse virtuelle de base du segment fraîchement installé dans le plan mémoire. L'instruction correspondante est la suivante : `shmptr =`

shmat(shmid, 0, 0). Remarquez que vous pouvez allouer toutes vos variables statiques dans ce segment de mémoire partagée en déclarant simplement vos variables partagées comme étant les membres d'une structure de type struct, et en déclarant *shmptr* comme étant un pointeur vers ce type de données. Avec cette technique, une variable partagée *x* serait accessible par *shmptr−>x*.

4. Comme ce segment de mémoire partagée doit être détruit quand le dernier processus à y accéder prend fin ou s'en détache, il nous faut appeler shmctl() pour configurer cette action par défaut. Le code correspondant ressemble à shmctl(shmid, IPC_RMID, 0).

5. Utiliser l'appel Linux fork() [9 [p 71]] pour créer le nombre désiré de processus. Chacun d'eux héritera du segment de mémoire partagée.

6. Lorsqu'un processus a fini d'utiliser un segment de mémoire partagée, il doit s'en détacher. On accomplit cela par un shmdt(shmptr).

Même avec si peu d'appels système, une fois le segment de mémoire partagée établi, tout changement effectué par un processeur sur une valeur se trouvant dans cet espace sera automatiquement visible par les autres processus. Plus important, chaque opération de communication sera exonérée du coût d'un appel système.

Ci-après, un exemple de programme en langage C utilisant les segments de mémoire partagée System V. Il calcule Pi, en utilisant les algorithmes de la section 1.3.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>

volatile struct shared { double pi; int lock; } * partage;

inline extern int xchg(register int reg,
volatile int * volatile obj)
{
  /* Instruction atomique d'échange */
__asm__ __volatile__ ("xchgl %1,%0"
                      :"=r" (reg), "=m" (*obj)
                      :"r" (reg), "m" (*obj));
  return(reg);
}

main(int argc, char **argv)
{
  register double largeur, sommelocale;
  register int intervalles, i;
  register int shmid;
  register int iproc = 0;;

  /* Alloue de la mémoire partagée */
  shmid = shmget(IPC_PRIVATE,
                 sizeof(struct shared),
                 (IPC_CREAT | 0600));
  partage = ((volatile struct shared *) shmat(shmid, 0, 0));
  shmctl(shmid, IPC_RMID, 0);
```

```
  /* Fais les inits ... */
  partage->pi = 0.0;
  partage->lock = 0;

  /* Crée un fils */
  if (!fork()) ++iproc;

  /* Récupère le nombre d'intervalles */
  intervalles = atoi(argv[1]);
  largeur = 1.0 / intervalles;

  /* Fais les calculs locaux */
  sommelocale = 0;
  for (i=iproc; i<intervalles; i+=2) {
    register double x = (i + 0.5) * largeur;
    sommelocale += 4.0 / (1.0 + x * x);
  }
  sommelocale *= largeur;

  /* Verrou d'attente atomique, ajout, et déverrouillage ... */
  while (xchg((iproc + 1), &(shared->lock))) ;
  shared->pi += sommelocale;
  shared->lock = 0;

  /* Fin du processus fils (barrière de synchro) */
  if (iproc == 0) {
    wait(NULL);
    printf("Estimation de pi: %f\n", partage->pi);
  }

  /* Sortie en bonne et due forme */
  return(0);
}
```

Dans cet exemple, j'ai utilisé l'instruction atomique d'échange pour mettre le verrouillage en œuvre. Pour de meilleures performances, préférez-lui une technique de synchronisation évitant les intructions verrouillant le bus.

Pendant les phases de débogage, il est utile de se souvenir que la commande `ipcs` renvoie la liste des facilités des IPC System V en cours d'utilisation.

2.6. Projection mémoire (*Memory Map Call*)

L'utilisation des appels système pour accéder aux fichiers (les entrées/sorties) peut revenir cher. En fait, c'est la raison pour laquelle il existe une bibliothèque de gestion des entrées/sorties gérant un tampon dans l'espace utilisateur (`getchar()`, `fwrite()`, et cætera). Mais les tampons utilisateur ne remplissent pas leur fonction si plusieurs processus accèdent au même fichier ouvert en écriture. La solution Unix BSD à ce problème fut l'ajout d'un appel système permettant à une portion d'un fichier d'être projetée en mémoire utilisateur, en utilisant principalement les mécanismes de la mémoire virtuelle pour provoquer les mises à jour. Le même mécanisme a été utilisé pendant plusieurs années dans les systèmes de Sequent comme base de leur gestion du traitement parallèle en mémoire partagée. En dépit de commentaires très négatifs dans la page de manuel (assez ancienne), Linux semble correctement effectuer au moins quelques unes des fonctions de base, et sait prendre en charge l'usage dérivé de cet appel pour projeter un segment anonyme de mémoire pouvant être partagé par plusieurs processus.

L'implémentation Linux de l'appel `mmap()` est en elle-même une solution intégrée de remplacement des étapes 2, 3 et 4 du schéma classique de mémoire partagée System V, mis en évidence dans la section 2.5. Pour créer un segment de mémoire partagée anonyme :

```
shmptr =
    mmap(0,                        /* Le système choisit l'adresse */
         b,                        /* Taille du segment de mémoire partagée */
         (PROT_READ | PROT_WRITE), /* droits d'accès, peuvent être rwx */
         (MAP_ANON | MAP_SHARED),  /* anonyme, partagé */
         0,                        /* descripteur de fichier (inutilisé) */
         0);                       /* offset fichier (inutilisé) */
```

L'équivalent de l'appel de mémoire partagée System V `shmdt()` est `munmap()` :

```
munmap(shmptr, b);
```

À mon avis, on ne gagne pas grand chose à utiliser `mmap()` plutôt que les mécanismes de gestion de la mémoire partagée de System V.

3. *Clusters* de systèmes Linux

Cette section tente de donner un aperçu de ce qu'est le traitement parallèle en *cluster* sous Linux. Les *clusters* sont, actuellement, à la fois l'approche la plus populaire et la plus variée, en s'étendant du simple réseau de stations de travail (*Network Of Workstations* : *NOW*) aux machines en parallèle essentiellement construites sur mesure, et dans lesquelles il arrive que l'on trouve comme éléments des PC sous Linux. Il existe également un grand nombre de logiciels pouvant prendre en charge le calcul en parallèle dans des *clusters* de machines Linux.

3.1. Pourquoi un *cluster* ?

Le traitement en parallèle au travers d'un *cluster* offre plusieurs avantages majeurs :

- Chacune des machines d'un *cluster* peut être un système complet, utilisable avec une large gamme d'applications de calcul. Cela conduit beaucoup de gens à suggérer l'idée que ce type de *cluster* pourrait faire usage de tous les « cycles machine perdus » sur les ordinateurs tournant à ne rien faire dans les bureaux. Il n'est pas si facile de récupérer ces cycles, et cela risque de ralentir l'écran de veille de votre collègue, mais cela peut être fait.

- L'explosion actuelle des systèmes en réseau signifie que la plupart du matériel nécessaire à la construction d'un *cluster* se vend déjà en grande quantité et aux prix réduits inhérents à la « grande distribution ». On peut économiser encore un peu plus en partant du principe qu'une seule carte vidéo, un seul moniteur et un seul clavier soient nécessaires pour chaque *cluster* (bien qu'il vous faille tout de même les passer d'une machine à une autre pour faire l'installation initiale de Linux, une fois lancé, un PC sous Linux typique n'a pas besoin de « console »). En comparaison, les systèmes SMP et processeurs auxiliaires représentent des marchés plus réduits, tendant à proposer des prix plus élevés par performances à l'unité.

- Les calculateurs en *cluster* peuvent *évoluer en de très grands systèmes*. Alors qu'il est actuellement difficile de trouver un ordinateur SMP à plus de quatre processeurs qui soit compatible avec Linux, la plupart du matériel réseau disponible permet de bâtir facilement un *cluster* incluant jusqu'à 16 machines. Avec un peu d'huile de coude, on peut mettre en réseau des centaines voire des milliers de machines. En fait, Internet tout entier pourrait être assimilé à un seul immense *cluster*.

- Le fait que remplacer une « mauvaise machine » au sein d'un *cluster* soit une opération triviale comparé à la réparation d'un ordinateur SMP partiellement défaillant garantit une disponibilité bien plus élevée aux configurations de *cluster* soignées. Cela devient important non seulement pour les applications qui ne peuvent tolérer des interruptions de service trop importantes, mais aussi dans l'utilisation en général de systèmes qui contiennent un nombre suffisant de processeurs pour que la panne d'une machine en particulier soit assez courante (par exemple, même si la durée moyenne avant la première panne d'un PC est environ deux ans, dans un *cluster* de 32 machines, la probabilité qu'au moins une machine tombe en panne dans les six premiers mois est assez élevée).

Très bien. Si les *clusters* sont gratuits ou très peu onéreux, peuvent devenir très grands et offrir une haute disponibilité… pourquoi tout le monde n'utilise pas un *cluster* ? Eh bien, parce qu'il y a aussi des problèmes :

- À quelques exceptions près, le matériel réseau n'est pas conçu pour le traitement en parallèle. Typiquement, les temps de latence sont élevés et la bande passante réduite comparés aux systèmes SMP et processeurs auxiliaires. Par exemple, si les temps de latence d'un SMP n'excèdent généralement pas plus de quelques microsecondes, ils atteignent couramment des centaines, voire des milliers de microsecondes dans un *cluster*. La bande passante des communications dans un système SMP dépasse souvent 100 mégaoctets par seconde. Bien que le matériel réseau le plus rapide (c'est-à-dire le « Gigabit Ethernet ») présente une vitesse comparable, la plupart des réseaux sont de 10 à 1000 fois plus lents. La performance d'un matériel réseau est déjà suffisamment médiocre dans un *cluster isolé*. Si le réseau n'est pas isolé du reste du trafic, ce qui est souvent le cas lorsque l'on utilise des « machines qui sont en réseau » plutôt qu'un système conçu pour être un *cluster*, les performances peuvent être bien pires encore.

- La gestion logicielle des *clusters* en tant que système unique est très mince. Par exemple, la commande `ps` ne fait état que des processus s'exécutant sur un seul système Linux, pas des processus s'exécutant à travers le *cluster* Linux entier.

Moralité, les *clusters* ont un très grand potentiel, mais ce potentiel risque d'être très difficile à concrétiser pour la plupart des applications. La bonne nouvelle, c'est qu'il existe un soutien logiciel très développé pour vous aider à obtenir de bonnes performances avec les programmes adaptés à cet environnement, et qu'il existe également des réseaux conçus spécialement pour élargir la palette de programmes pouvant avoir de bonnes performances.

3.2. Le matériel réseau

Les réseaux informatiques sont en pleine explosion… mais vous le savez déjà. Un nombre toujours grandissant de technologies et produits réseau a été développé, et la plupart d'entre eux sont disponibles sous une forme qui peut être utilisée pour faire d'un groupe de machines (des PC fonctionnant chacun sous Linux) un *cluster* de traitement en parallèle.

Malheureusement, aucune technologie réseau ne résout complètement tous les problèmes. À vrai dire, le nombre d'approches différentes, en coût et en performances, est à première vue assez difficile à croire. Par exemple, le coût par machine mise en réseau s'étend de moins de 5 dollars jusqu'à plus de 4000. La bande passante et les temps de latence varient aussi selon quatre ordres de grandeur.

Avant d'en apprendre plus sur les spécificités de certains réseaux, il est important de remarquer que ces choses évoluent très fréquemment (voir http://www.linux.org.uk/NetNews.html pour avoir des nouvelles fraîches concernant les réseaux sous Linux), et qu'il est très difficile d'obtenir des infos précises concernant certains réseaux.

J'ai placé un « *?* » aux endroits incertains. J'ai aussi passé beaucoup de temps à faire des recherches sur ce sujet, mais je reste sûr que ce résumé est plein d'erreurs et d'omissions importantes. Si vous avez des corrections ou des ajouts à y apporter, merci de m'envoyer un courrier électronique en anglais à l'adresse suivante : <`hankd CHEZ engr POINT uky POINT edu`>.

Les résumés comme le LAN Technology Scorecard [10 [p 71]] donnent les caractéristiques de nombreux et différents types de réseaux et de standards de réseaux locaux (LAN). Cependant, l'essentiel de ce guide pratique est centré sur les propriétés les plus indiquées à la construction d'un *cluster* Linux. Chaque section décrivant un type de réseau débute par une courte liste de caractéristiques. Ci dessous, la définition de chaque entrée.

Prise en charge sous Linux :

> Si la réponse est *non*, la signification est claire. Les autres réponses tentent de décrire l'interface de base utilisée pour accéder au réseau. La plupart du matériel réseau est interfacé via un pilote de périphérique du noyau, sachant typiquement gérer les communications TCP/UDP. D'autres réseaux utilisent des interfaces plus directes (par exemple des bibliothèques) pour réduire les temps de latence en évitant d'avoir à passer par le noyau.
>
> Il y a quelques années, il était considéré comme parfaitement acceptable d'accéder au coprocesseur mathématique (« *floating point unit* ») par un appel système mais aujourd'hui, c'est clairement ridicule. À mon avis, nécessiter un appel système pour chaque communication entre les processeurs exécutant un programme en parallèle est peu commode. Le problème est que les ordinateurs n'ont pas encore intégré ces mécanismes de communication, et que les approches non « orientées noyau » tendent à présenter des problèmes de portabilité. Vous allez entendre beaucoup parler de ce sujet dans un avenir proche, principalement sous la forme de la nouvelle *Virtual Interface (VI) Architecture* (http://www.intel.com/intelpress/sum_via.htm) méthode standardisée pour les opérations des interfaces réseau et servant à contourner les couches des systèmes d'exploitation usuels. Le standard VI est appuyé par Compaq, Intel et Microsoft, et aura assurément un impact important sur la conception des SAN (*System Area Network*) dans les prochaines années.

Bande passante maximum :

> C'est le chiffre dont tout le monde se soucie. J'ai généralement repris les débits maximum théoriques. Votre moyenne, elle, *va* varier.

Temps de latence minimum :

> À mon avis, c'est le chiffre dont tout le monde devrait se soucier, plus encore que de la bande passante. Là encore, j'ai utilisé les (irréalistes) valeurs théoriques idéales, mais au moins ces nombres prennent en compte *toutes* les sources de latence, tant logicielles que matérielles. Dans la plupart des cas, les temps de latence réseau ne durent que quelques microsecondes. Des nombres beaucoup plus grands reflètent les couches des matériels et logiciels inefficaces.

Disponibilité :

> Reprise telle quelle, cette ligne décrit la forme sous laquelle vous pouvez acquérir ce type de matériel. Le matériel de la grande distribution est disponible largement et de plusieurs fabricants, le prix étant alors le premier critère de choix. Les choses proposées par « différents fabricants »

sont disponibles chez plus d'un seul concurrent, mais il existe des différences significatives, et des problèmes potentiels d'interopérabilité. Les réseaux produits par un « fabricant exclusif » vous laissent à la merci de ce fournisseur, aussi bienveillant soit-il. « Domaine public » signifie que même si vous ne pouvez trouver quelqu'un pour vous vendre ce type de matériel, vous ou n'importe qui d'autre pouvez acheter les composants et en fabriquer un exemplaire. C'est typiquement le cas des prototypes de recherche. Ils ne sont en général ni prêts à être utilisés par le public, ni disponibles à celui-ci.

Port ou bus utilisé :

La manière dont on relie l'interface réseau à l'ordinateur. Les meilleures performances (et désormais les plus courantes) s'obtiennent avec le bus PCI. Il existe aussi des cartes pour bus EISA, VESA local bus (VLB), et ISA. ISA fut le premier, et il est toujours utilisé par les cartes aux basses performances. On trouve toujours l'EISA comme bus secondaire dans les machines PCI. Ces derniers temps, on ne trouve plus beaucoup de matériel à base de VLB (même si la Video Electronics Standards Association voit les choses autrement).

Bien sûr, toute interface que vous pouvez utiliser sans avoir à ouvrir le boîtier de votre PC est plus qu'attrayante. les interfaces IrDA (N.D.T. : port infrarouge) et USB font leur apparition de plus en plus fréquemment. Le Port Parallèle Standard (SPP) est longtemps resté « ce sur quoi vous branchiez votre imprimante », mais s'est avéré dernièrement être très utile comme extension du bus ISA. Cette nouvelle fonction est améliorée par le standard IEEE 1284, qui spécifie les optimisations EPP et ECP. Il y a aussi le bon vieux port série RS232, lent mais fiable. Je n'ai eu vent d'aucun témoignage concernant l'interconnexion de machines par le biais des connecteurs vidéo (VGA), clavier, souris ou joystick…

Structure du réseau :

Un bus est un fil, un ensemble de fils, ou une fibre (optique). Un *hub* (« concentrateur » ou « plaque tournante ») est une petite boite qui peut recevoir différents types de fils ou de fibres. Les commutateurs (« *switched hubs* ») permettent à plusieurs connexions de transmettre activement et simultanément leurs données.

Coût par machine reliée :

Voici comment lire ces nombres. Supposons que, en dehors des connexions réseau, acheter un PC comme unité de votre *cluster* vous coûte 2000 dollars. L'ajout de Fast Ethernet porte le prix à l'unité à environ 2400 dollars. L'ajout de Myrinet mène ce prix à 3800 dollars environ. Si vous avez 20 000 dollars à dépenser, vous pouvez alors avoir soit 8 machines reliées par du Fast Ethernet, soit 5 machines reliées par du Myrinet. Il est également très raisonnable d'utiliser plusieurs types de réseaux. Par exemple, avec 20 000 dollars, vous pouvez vous offrir 8 machines reliées entre elles par Fast Ethernet et TTL_PAPERS. Choisissez un réseau — ou un ensemble de réseaux — qui permettra à votre *cluster* d'exécuter votre application le plus rapidement possible.

Au moment où vous lirez ces lignes, ces chiffres seront faux… En fait, ils le sont sûrement déjà. Il peut aussi y avoir un grand nombre de réductions, d'offres spéciales, et cætera. Mais en tout cas, les prix cités ici ne seront jamais suffisamment erronés pour vous conduire à faire un choix totalement inapproprié. Nul besoin d'avoir un doctorat (même si c'est mon cas ;-) pour constater que l'utilisation de réseaux onéreux n'a de sens que si votre application a réellement besoin de leurs propriétés ou si les PC utilisés dans votre *cluster* sont eux aussi relativement chers.

Maintenant que vous êtes avertis, place au spectacle…

3.2.1. ArcNet

- Prise en charge par Linux : *pilotes du noyau*
- Bande passante maximum : *2,5 Mbits/s*
- Temps de latence minimum : *1000 microsecondes ?*
- Disponibilité : *différents fabricants*
- Port ou bus utilisé : *ISA*
- Structure du réseau : Hub *ou bus non commutés (anneau logique)*
- Coût par machine reliée : *200 dollars*

ARCNET est un réseau local principalement destiné à être utilisé dans les systèmes de contrôle temps réel embarqués. Comme Ethernet, le réseau est physiquement organisé en prises le long d'un bus d'un ou plusieurs *hubs*. En revanche, et contrairement à Ethernet, il utilise un protocole à base de jetons qui structure de manière logique le réseau comme un anneau. Les entêtes des paquets sont réduites (3 ou 4 octets) et les messages peuvent être très courts (jusqu'à un seul octet). De fait, ARCNET est plus efficace qu'Ethernet. Malheureusement, il est aussi plus lent, et moins populaire ce qui le rend plus cher. Vous trouvez plus d'informations sur le site de l'ARCNET Trade Association.

3.2.2. ATM

- Prise en charge par Linux : *pilotes du noyau, bibliothèques AAL**
- Bande passante maximum : *155 Mbits/s* (bientôt, *1200 Mbits/s*)
- Temps de latence minimum : *120 microsecondes*
- Disponibilité : *différents fabricants*
- Port ou bus utilisé : *PCI*
- Structure du réseau : hubs *commutés*
- Coût par machine reliée : *3000 dollars*

A moins d'avoir été dans le coma ces dernières années, vous avez sûrement beaucoup entendu dire qu'ATM (« *Asynchronous Transfer Mode* ») *est* l'avenir… Eh bien, en quelque sorte. ATM est meilleur marché que HiPPI et plus rapide que Fast Ethernet, et il peut être utilisé sur de très longues distances, ce qui intéresse beaucoup les opérateurs téléphoniques. Le protocole du réseau ATM est également conçu pour fournir une interface logicielle aux temps d'accès réduits et gérant plus efficacement les messages courts et les communications en temps réel (c'est-à-dire les transmissions audio et vidéo numériques). C'est aussi l'un des réseaux aux plus hauts débits que Linux prenne actuellement en charge. La mauvaise nouvelle, c'est qu'ATM n'est pas vraiment bon marché, et qu'il demeure encore des problèmes de compatibilité entre fabricants. Un aperçu du développement ATM sous Linux est disponible sur http://linux-atm.sourceforge.net/.

3.2.3. CAPERS

- Prise en charge par Linux : *bibliothèque AFAPI*
- Bande passante maximum : *1,2 Mbit/s*
- Temps de latence maximum : *3 microsecondes*
- Disponibilité : *grande distribution*
- Port ou bus utilisé : *SPP (port parallèle)*
- Structure du réseau : *câble entre 2 machines*
- Coût par machine reliée : *2 dollars*

CAPERS (« *Cable Adapter for Parallel Execution and Rapid Synchronisation* » : « Adaptateur par Câble pour l'Exécution en Parallèle et la Synchronisation Rapide ») est un produit dérivé du projet PAPERS, de l'*University School of Electrical and Computer Engineering* de Purdue. En substance, ce projet définit un protocole logiciel permettant d'utiliser une simple liaison point-à-point par le port parallèle, type « LapLink », pour implémenter la bibliothèque PAPERS sur deux PC sous Linux. L'idée n'est exploitable à grande échelle, mais le prix est imbattable. Tout comme avec TTL_PAPERS, pour améliorer la sécurité du système, il est recommandé, mais pas nécessaire, d'appliquer un correctif mineur au noyau.

3.2.4. Ethernet

- Prise en charge par Linux : *pilotes du noyau*
- Bande passante maximum : *10 Mbits/s*
- Latence minimum : *100 microsecondes*
- Disponibilité : *grande distribution*
- Port ou bus utilisé : *PCI*
- Structure du réseau : *commutateurs ou concentrateurs, ou même bus simple*
- Coût par machine connectée : *100 dollars* (sans concentrateur, *50 dollars*)

Depuis plusieurs années maintenant, l'Ethernet à 10Mbits/s est le standard des technologies réseau. Une bonne carte Ethernet s'achète pour moins de 50 dollars, et bon nombre de PC en sont aujourd'hui équipés de série, l'interface étant intégrée à la carte-mère. Les réseaux à base Ethernet dont la charge n'est pas très élevée peuvent être organisés comme un long bus à prises multiples sans concentrateur (« *hub* »). Une telle configuration peut servir jusqu'à 200 machines pour un coût minimal, mais n'est pas appropriée au traitement en parallèle. Ajouter un concentrateur non commuté n'améliore pas beaucoup les performances. En revanche, les commutateurs (« *switches* »), qui peuvent offrir une bande passante maximum à plusieurs connexions simultanément ne coûtent qu'aux alentours de 100 dollars par port. Linux prend en charge un nombre impressionnant d'interfaces Ethernet différentes, mais il est important de garder à l'esprit que les variations entre ces matériels peuvent engendrer de grandes différences de performance. Consultez le Guide pratique de la compatibilité matérielle avec Linux (N.D.T. : version française du *Hardware Compatibility HOWTO*) pour plus d'informations concernant les différents équipements fonctionnant sous Linux, et pour avoir une idée de leur qualité.

Le *cluster* Linux à 16 machines réalisé dans le cadre du projet « Beowulf » (initialement développé au CESDIS de la NASA) est une manière intéressante d'augmenter ses performances. C'est à Donald Becker, auteur de plusieurs pilotes de cartes Ethernet, que l'on doit la prise en charge de la répartition du trafic au travers de plusieurs cartes réseau s'éclipsant mutuellement (autrement dit, partageant les mêmes adresses réseau). Cette fonction est intégrée en standard dans Linux, et s'effectue de manière invisible en dessous du niveau des opérations sur les *sockets*. Le coût d'un hub n'étant pas négligeable, relier chaque machine à deux (ou plus) réseaux Ethernet, sans *hubs* ni *switches*, pour améliorer les performances peut s'avérer financièrement très rentable. D'une manière générale, dans les situations où une machine est le goulet d'étranglement d'un réseau, la répartition de charge à travers plusieurs réseaux (« *shadow networks* ») se révèle bien plus efficace que l'usage d'un réseau équipé d'un commutateur seul.

3.2.5. Ethernet (Fast Ethernet)

- Prise en charge par Linux : *pilotes du noyau*
- Bande passante maximum : *100 Mbits/s*
- Temps de latence minimum : *80 microsecondes*
- Disponibilité : *grande distribution*
- Port ou bus utilisé : *PCI*
- Structure du réseau : *concentrateurs ou commutateurs (*hubs *ou* switches*)*
- Coût par machine connectée : *400 dollars (?)*

Bien qu'il existe un certain nombre de technologies différentes nommées « Fast Ethernet », elles se réfèrent souvent à un réseau Ethernet 100 Mbits/s en grande partie compatible avec les anciens câbles et périphériques 10 Mbits/s type « 10 BaseT ». Comme on peut s'y attendre, tout ce qui s'appelle Ethernet bénéficie des prix de la vente de masse, et ces interfaces ne coûtent généralement qu'une fraction du prix des cartes ATM à 155 Mbits/s. Le problème est qu'une collection de machines partageant tous le même « bus » à 100 Mbits/s (à l'aide d'un *hub* non commuté) peut présenter des performances n'atteignant en moyenne même pas celles d'un réseau 10 Mbits/s utilisant un commutateur fournissant à chaque machine une connexion 10 Mbits/s complète.

Les commutateurs pouvant fournir une connexion 100 Mbits à chaque machine simultanément sont chers, mais les prix chutent chaque jour, et ces commutateurs offrent une bande passante autrement plus élevée que de simples *hubs* non commutés. Ce qui rend les commutateurs ATM si onéreux est la nécessité de commuter chaque cellule ATM, cellule relativement courte. Certains commutateurs Ethernet parient sur une fréquence de commutation attendue relativement lente et en tirent profit en utilisant des techniques aux temps de latence réduits à l'intérieur du commutateur, mais nécessitant quelques millisecondes pour changer de voie. Si l'itinéraire de votre trafic réseau change fréquemment, évitez ce type d'équipement.

Notez aussi que, comme pour Ethernet, le projet Beowulf (http://www.beowulf.org) de la NASA développe des pilotes aux performances supérieures car utilisant la répartition de charge entre plusieurs cartes Fast Ethernet.

3.2.6. Ethernet (Gigabit Ethernet)

- Prise en charge par Linux : *pilotes du noyau*
- Bande passante maximum : *1000 Mb/s*
- Temps de latence minimum : *300 microsecondes (?)*
- Disponibilité : *différents fabricants*
- Port ou bus utilisé : *PCI*
- Structure réseau : *commutateurs ou FDR*
- Coût par machine connectée : *2500 dollars (?)*

Je ne suis pas sûr que Gigabit Ethernet ait une raison technologique valable de s'appeler Ethernet… mais son nom inspire le fait que Gigabit Ethernet est conçu pour être une technologie réseau bon marché et de grande distribution, avec une prise en charge native de l'IP. En revanche, les prix actuels reflètent le fait que cela reste un produit difficile à fabriquer.

Contrairement aux autres technologies Ethernet, Gigabit Ethernet apporte un contrôle du flux, ce qui devrait en faire un réseau plus fiable. Les FDR, ou « *Full-Duplex Repeaters* » (« répéteurs bidirectionnels simultanés »), se contentent de multiplexer les lignes, en utilisant des mémoires tampons et des contrôles de flux localisés pour améliorer les performances. La plupart des commutateurs sont construits comme de nouveaux modules pour les différents modèles de commutateurs compatible Gigabit déjà existants. Les commutateurs ou FDR sont distribués ou annoncés par au moins Acacianet, Bay networks, Cabletron (désormais Enterasys. Page française : http://www.enterasys.com/fr/), Networks digital, Extreme networks, Foundry networks, Gigalabs.com, Packet engines, Plaintree systems, Prominet, Sun microsystems, et Xlnt.

Il existe un pilote pour Linux pour les « Yellowfin » G-NIC de Packet Engines [11 [p 71]]. Les premiers essais sous Linux ont fait état d'un taux de transfert environ 2,5 fois supérieur à la plus rapide des cartes Fast Ethernet à 100 Mbits/s. Avec les réseaux Gigabit, une configuration soigneuse du bus PCI est un facteur critique. Il reste toujours un doute quant à la poursuite des améliorations de ce pilote, et du développement des pilotes Linux des autres cartes réseau.

3.2.7. FC (« *Fibre Channel* »)

- Prise en charge par Linux : *non* [12 [p 71]]
- Bande passante maximum : *1062 Mbits/s*
- Temps de latence minimum : *?*
- Disponibilité : *différents fabricants*
- Interface ou bus utilisé : *PCI (?)*
- Structure du réseau : *?*

- Coût par machine connectée : *?*

L'objectif du FC (« *Fibre Channel* ») est de fournir un médium d'entrée/sortie de type bloc aux performances élevées (une trame FC transporte un bloc de données d'une longueur forfaitaire de 2048 octets), particulièrement adapté aux partages de disques et autres périphériques de stockage, qui peuvent alors être reliés directement au réseau FC plutôt qu'à travers un ordinateur. Niveau bande passante, le FC est présenté comme étant relativement rapide, avec un taux s'étendant de 133 à 1062 Mbits/s. Si le FC tend à devenir populaire en tant que solution haut-de-gamme de remplacement du SCSI, il pourrait rapidement devenir une technologie très abordable. Pour le moment, ce n'est pas abordable, et ce n'est pas pris en charge par Linux. La Fibre Channel Association tient une importante collection de références au FC, sur http://www.fibrechannel.org [13 [p 71]].

3.2.8. FireWire (IEEE 1394)

- Prise en charge par Linux : *non* [14 [p 71]]
- Bande passante maximum : *196,608 Mbits/s* (bientôt, *393,216 Mbits/s*)
- Temps de latence minimum : *?*
- Disponibilité : *différents fabricants*
- Interface ou bus utilisé : *PCI*
- Structure du réseau : *aléatoire, sans cycles (auto-configuré)*
- Coût par machine connectée : *600 dollars*

FireWire, ou standard IEEE 1394-1995, est voué à être le réseau numérique à grande vitesse et à prix réduit des appareils électroniques domestiques. Son application-phare est la connexion des caméscopes numériques aux ordinateurs, mais FireWire est destiné à être utilisé dans des domaines s'étendant de l'alternative au SCSI jusqu'à l'interconnexion des différents composants de votre « Home Cinéma ». Il vous permet de relier plus de 64000 périphériques dans une topologie utilisant des bus et des ponts mais ne formant pas de boucle, et détecte automatiquement la configuration des périphériques lorsqu'ils sont ajoutés ou retirés. Les messages courts (les « quadlets », longs de quatre octets) sont également pris en charge, de même que les transmissions isochrones sur le modèle de l'ATM (utilisées pour préserver le synchronisme des messages multimédia). Adaptec propose des produits FireWire permettant de relier jusqu'à 63 périphériques à une seule carte PCI, et tient aussi un bon site d'information générale concernant le FireWire sur http://www.adaptec.com.

Bien que le FireWire ne soit pas le réseau le plus rapide disponible actuellement, le marché de la grande distribution (qui tire les prix vers le bas) et les temps de latence réduits pourraient en faire d'ici l'an prochain la meilleure interface réseau pour le message passing dans un *cluster* de PC sous Linux.

3.2.9. HiPPI et Serial HiPPI

- Prise en charge par Linux : *non* [15 [p 71]]
- Bande passante maximum : *1600 Mbits/s* (*1200 Mb/s* pour Serial HiPPI)

- Temps de latence minimum : *?*
- Disponibilité : *différents fabricants*
- Interface ou bus utilisé : *EISA, PCI*
- Structure du réseau : *commutateurs*
- Coût par machine connectée : *3500 dollars* (*4,500* dollars pour Serial HiPPI)

HiPPI (« *High Performance Parallel Interface* », soit « Interface Parallèle aux Performances Élevées ») était initialement censée fournir un taux de transfert élevé pour l'échange d'immenses blocs de données entre un supercalculateur et une autre machine (un autre supercalculateur, un « *frame buffer* », une batterie de disques, et cætera), et est devenu le standard dominant dans le monde des supercalculateurs. Bien que ce soit un oxymoron, *Serial HiPPI* devient également très populaire en utilisant typiquement de la fibre optique à la place des câbles HiPPI standard de 32 bits de large (donc parallèles). Ces dernières années, les commutateurs HiPPI en croix sont devenus courants et les prix ont sérieusement chuté. Malheureusement, les équipement HiPPI Série, eux, sont encore très onéreux, et sont en général les seuls pris en charge par le bus PCI. Pire, Linux ne gère pas encore HiPPI. Le CERN tient une bonne présentation d'HiPPI sur http://www.cern.ch/HSI/hippi/. Ils tiennent aussi une liste assez longue de distributeurs proposant le HiPPI sur http://www.cern.ch/HSI/hippi/procintf/manufact.htm.

3.2.10. IrDA (« *Infrared Data Association* »)

- Prise en charge par Linux : *non (?)*[16 [p 71]]
- Bande passante maximum : *1,15 Mbits/s* et *4 Mbits/s*
- Temps de latence minimum : *?*
- Disponibilité : *Différents fabricants*
- Interface ou bus utilisé : *IrDA*
- Structure réseau : *Air libre* ;-)
- Coût par machine connectée : *0*

L'IrDA (« *Infrared Data Association* » ou « Association de Données par Infrarouges », sur http://www.irda.org), c'est ce petit appareil à infrarouges sur le coté des ordinateurs portables. Il reste assez difficile, par conception, de relier plus de deux ordinateurs par ce biais, aussi l'IrDA ne se prête-t-il guère à la « clusterisation », ou mise en parallèle massive de nombreuses machines. Don Becker est toutefois l'auteur de quelques travaux préliminaires sur l'IrDA.

3.2.11. Myrinet

- Prise en charge par Linux : *bibliothèques*
- Bande passante maximum : *1280 Mbits/s*

- Temps de latence minimum : *9 microsecondes*
- Disponibilité : *matériel propriétaire.*
- Interface ou bus utilisés : *PCI*
- Structure du réseau : *commutateurs*
- Coût par machine connectée : *1800 dollars*

Myrinet est un réseau local (LAN : « *Local Area Network* ») conçu pour servir également de réseau système [17 [p 71]]. Les versions LAN et SAN utilisent des médias physiques distincts et leur caractéristiques sont sensiblement différentes. La version SAN est généralement utilisée au sein d'un *cluster*.

La structure de Myrinet est très conventionnelle, mais a la réputation d'être très bien implémentée. Les pilotes pour Linux sont connus pour donner de très bons résultats, bien qu'il eût été fait état de frappantes différences de performances d'une implémentation du bus PCI à l'autre.

Actuellement, Myrinet est assurément le réseau favori des responsables de *clusters* n'étant pas trop sévèrement limité au niveau budgétaire. Si, pour vous, un PC Linux typique est un Pentium Pro dernier cri ou un Pentium II avec au moins 256 Mo de mémoire vive, et un disque RAID et SCSI, alors le coût de Myrinet apparaît raisonnable. En revanche, avec des machines plus conventionnelles, il vous faudra probablement choisir entre relier *N* machines avec Myrinet, ou *2N* machines avec plusieurs équipements de type « Fast Ethernet » ou « TTL_PAPERS ». Tout cela dépend réellement de votre budget et du type de calcul qui vous importe le plus.

3.2.12. Parastation

- Prise en charge par Linux : *couches d'abstraction (« HAL ») ou bibliothèques réseau*
- Bande passante maximum : *125 Mbits/s*
- Temps de latence minimum : *2 microsecondes*
- Disponibilité : *fabricant exclusif*
- Interface ou bus utilisé : *PCI*
- Structure du réseau : *maillage, sans concentrateur*
- Coût par machine connectée : *plus de 1000 dollars*

Le projet ParaStation (http://wwwipd.ira.uka.de/parastation) de la section informatique de l'Université de Karlsruhe est en train de mettre sur pieds un réseau « maison » compatible PVM et aux temps de latence réduits. Ils ont d'abord construit un prototype de ParaPC biprocesseur en utilisant une carte EISA conçue sur mesure et des PC fonctionnant sous Unix BSD, puis ont bâti de plus grands *clusters* composés de machines Alpha DEC. Depuis Janvier 1997, Parastation est disponible sous Linux. Les cartes PCI sont produites en coopération avec une société nommée Hitex. Le matériel de Parastation implémente à la fois un système de transmission de messages rapide et fiable, et des barrières de synchronisation simples.

3.2.13. PLIP

- Prise en charge par Linux : *pilotes du noyau*
- Bande passante maximum : *1,2 Mbits/s*
- Temps de latence minimum : *1000 microsecondes ?*
- Disponibilité : *grande distribution*
- Interface ou bus utilisé : *SPP*
- Structure du réseau : *câble entre 2 machines*
- Coût par machine connectée : *2 dollars*

Pour le seul coût d'un câble « LapLink » (N.D.T. : câble parallèle croisé), PLIP (« *Parallel Line Interface Protocol* », soit « Protocole d'Interface par Ligne Parallèle ») permet à deux machines Linux de communiquer par le port parallèle en utilisant les couches logicielles standard basées sur la communication par « sockets ». En termes de bande passante, de temps de latence et d'évolutivité, il ne s'agit pas d'une technologie réseau sérieuse. En revanche, le coût de revient quasi-nul et la compatibilité logicielle s'avèrent être très utiles. Le pilote est partie intégrante du noyau Linux standard.

3.2.14. SCI

- Prise en charge par Linux : *non*
- Bande passante maximum : *4000 Mbit/s*
- Temps de latence minimum : *2,7 microsecondes*
- Disponibilité : *différents fabricants.*
- Interface ou bus utilisé : *PCI et propriétaire*
- Structure réseau : *?*
- Coût par machine connectée : *+ de 1000 dollars*

L'objectif de SCI (Scalable Coherent Interconnect, ANSI/IEEE 1596-1992) consiste essentiellement à fournir un mécanisme de haute performance pouvant assurer des accès cohérents à la mémoire partagée au travers d'un grand nombre de machines. On peut dire sans se mouiller que la bande passante et les temps de latences de SCI sont « très impressionnants » comparés à la plupart des autres technologies réseau. Le problème est que SCI n'est pas très répandu et reste donc assez onéreux, et que ce matériel n'est pas pris en charge par Linux.

SCI est principalement utilisé dans diverses implémentations propriétaires pour des machines à mémoire partagée logiquement et distribuée physiquement, comme le HP/Convex Exemplar SPP et le Sequent NUMA-Q 2000 (voir http://www.sequent.com [18 [p 71]]). Ceci dit, SCI est disponible sous forme de carte PCI et de commutateurs quatre ports de Dolphin (on peut relier ainsi jusqu'à 16 machines en montant ces commutateurs en cascade), http://www.dolphinics.com, sous la série "Clustar". Le CERN tient à jour une bonne collection de liens concernant SCI sur http://www.cern.ch/HSI/sci/sci.html.

3.2.15. SCSI

- Prise en charge par Linux : *pilotes du noyau*
- Bande passante maximum : de *5 Mbits/s* à plus de *20 Mbits/s*
- Temps de latence minimum : *?*
- Disponibilité : *différents fabricants*
- Interface ou bus utilisé : *cartes PCI, EISA ou ISA*
- Structure du réseau : *bus inter-machines partageant des périphériques SCSI*
- Coût par machine connectée : *?*

Le SCSI (Small Computer Systems Interconnect) consiste essentiellement en un bus d'entrée/sortie utilisé par les disques durs, les lecteurs de CD-ROM, les numériseurs (« *scanners* »), et cætera. Il existe trois standards distincts : SCSI-1, SCSI-2 et SCSI-3, en vitesses "Fast" et "Ultra" et en largeur de bus de 8, 16 ou 32 bits (avec compatibilité FireWire annoncée pour SCSI-3). Tout cela est plutôt confus, mais nous savons tous qu'un bon SCSI est bien plus rapide que l'EIDE, et peut gérer plus de périphériques, plus efficacement.

Ce que beaucoup de gens ne réalisent pas, c'est qu'il est très simple de partager le même bus SCSI entre deux ordinateurs. Ce type de configuration est très utile pour partager des disques entre deux machines et mettre en place un système de fail-over, de façon à ce qu'une machine prenne à sa charge les requêtes à une base de données lorsque l'autre machine tombe en panne. C'est actuellement le seul mécanisme reconnu par le *cluster* PC de Microsoft : WolfPack. En revanche, l'incapacité du SCSI à évoluer vers de plus grands systèmes le rend en général inintéressant pour le traitement en parallèle.

3.2.16. ServerNet

- Prise en charge par Linux : *non*
- Maximum bandwidth : *400 Mbits/s*
- Temps de latence minimum : *3 microsecondes*
- Disponibilité : *fabricant exclusif*
- Interface ou bus utilisé : *PCI*
- Structure du réseau : *arbre hexagonal / concentrateurs en mailles tétraédriques*
- Coût par machine connectée : *?*

ServerNet est la solution réseau de haute performance proposée par Tandem (http://www.tandem.com). Dans le monde du traitement des transactions en ligne (« *OnLine Transation Processing* », ou « OLTP ») en particulier, Tandem est réputé être l'un des premiers fabricants de systèmes de haute fiabilité, aussi n'est-il pas surprenant que leurs réseaux ne revendiquent pas simplement la haute performance, mais aussi la « haute fiabilité et intégrité des données ». Une autre facette intéressante de ServerNet : ce matériel serait capable de transférer des données directement de périphérique à périphérique, pas simplement entre processeurs, mais également entre disques durs, et cætera, dans un style unilatéral similaire à ce qui a été suggéré pour les mécanismes d'accès à distance à la mémoire du MPI, décrits dans la section 3.5. Un dernier mot à propos de Servernet : Bien qu'il n'y ait qu'un seul fabricant, celui-ci est suffisamment puissant pour faire établir potentiellement Servernet en tant que standard majeur : Tandem appartient à Compaq[19 [p 71]].

3.2.17. SHRIMP

- Prise en charge par Linux : *interface utilisateur à mémoire mappée*
- Bande passante maximum : *180 Mbits/s*
- Temps de latence minimum : *5 microsecondes*
- Disponibilité : *prototype expérimental*
- Interface ou bus utilisé : *EISA*
- Structure du réseau : *Fond de panier en maille (comme pour le Paragon d'Intel)*
- Coût par machine connectée : *?*

Le projet SHRIMP de la section des sciences des ordinateurs de l'Université de Princeton, met sur pieds un ordinateur parallèle en utilisant dont les éléments de traitement sont des ordinateurs PC sous Linux. Le premier SHRIMP (« *Scalable, High-Performance, Really Inexpensive Multi-Processor* », soit « multiprocesseur évolutif et de hautes performances vraiment bon marché ») était un simple prototype biprocesseur utilisant une mémoire partagée sur une carte EISA développée pour l'occasion. Il existe désormais un prototype pouvant évoluer vers de plus larges configurations en utilisant une interface « maison » pour se connecter à une sorte de concentrateur, essentiellement conçu comme le réseau de routage en mailles utilisé dans le Paragon d'Intel. Des efforts considérables ont été faits pour développer une électronique de « communication mappée en mémoire virtuelle » aux *overheads* réduits, avec sa couche logicielle.

3.2.18. SLIP

- Prise en charge par Linux : *pilotes du noyau*
- Bande passante maximum : *0,1 Mbits/s*
- Temps de latence minimum : *1000 microsecondes ?*
- Disponibilité : *grande distribution*
- Interface ou bus utilisé : *RS232C*
- Structure du réseau : *câble entre deux machines*
- Coût par machine connectée : *2 dollars*

Même si SLIP (« *Serial Line Interface Protocol* ») se situe définitivement au pied de l'échelle des performances, ce protocole (tout comme CSLIP ou PPP) permet à deux machines de communiquer en utilisant les « *sockets* » et ce au travers d'un câble RS232 ordinaire. Les ports RS232 peuvent être reliés à l'aide d'un câble série type NULL-MODEM, ou même au travers d'une ligne téléphonique en utilisant des modems. Dans tous les cas, les temps de latence sont élevés, et la bande passante réduite.

Aussi, SLIP ne devrait être utilisé qu’en dernier recours. En revanche, la plupart des PC sont dotés de deux ports RS232. Il doit donc être possible de relier un groupe de machines sous forme de réseau linéaire ou d’anneau. Il existe même un logiciel de répartition de la charge appelé EQL.

3.2.19. TTL_PAPERS

- Prise en charge par Linux : *bibliothèque AFAPI*
- Bande passante maximum : *1,6 Mbits/s*
- Temps de latence minimum : *3 microsecondes*
- Disponibilité : *conception dans le domaine public, fabricant exclusif*
- Interface ou bus utilisé : *SPP (port parallèle)*
- Structure du réseau : *arbre de concentrateurs*
- Coût par machine connectée : *100 dollars*

Le projet PAPERS (« *Purdue’s Adapter for Parallel Execution and Rapid Synchronization* », soit « Adaptateur pour l’Exécution en Parallèle et la Synchronisation Rapide de l’université de Purdue »), mené par la Purdue University School of Electrical and Computer Engineering (« École Supérieure d’Électricité et d’Ingénierie en Informatique »), développe un ensemble logiciel et matériel évolutif et aux temps de latence réduits pour les communications des fonctions d’agrégation, permettant de mettre sur pieds un supercalculateur en parallèle utilisant comme nœuds des PC d’origine, non modifiés.

Plus d’une douzaine de versions de cartes « PAPERS », reliées au PC à la station de travail via le port parallèle standard (SPP : « *Standard Parallel Port* »), ont été construites, en suivant globalement deux grands axes. Les versions estampillées « PAPERS » visent les hautes performances, quelle que soit la technologie la plus indiquée, la version actuelle utilisant des FPGA (N.D.T. : famille de réseaux logiques programmables), et des modèles d’interfaces PCI à haut débit sont actuellement à l’étude. Par opposition, les versions nommées « TTL_PAPERS » sont conçues pour être facilement reproduites hors de l’université de Purdue, et s’appuient sur des modèles du domaine public remarquablement simples et qui peuvent être mis en place en utilisant de la logique TTL ordinaire. L’une de ces versions est produite commercialement.

Contrairement au matériel sur mesure conçu par d’autres universités, des *clusters* TTL_PAPERS ont été assemblés dans plusieurs écoles depuis les États-Unis jusqu’en Corée du Sud. La bande passante est sévèrement limitée par la connectivité du port parallèle, mais PAPERS met en œuvre des fonctions d’agrégation aux temps de latence très réduits. Même les systèmes orientés messages les plus rapides ne peuvent offrir de telles performances sur ces fonctions d’agrégation. Ainsi, PAPERS est particulièrement performant dans la synchronisation des différents écrans d’un mur vidéo (à débattre dans le *Video-Wall-HOWTO* actuellement en préparation), pour planifier les accès à un réseau à haut débit, pour évaluer les probabilités en recherche génétique, et cætera. Même si des *clusters* PAPERS ont été construits en utilisant AIX d’IBM sur PowerPC, des DEC Alpha OSF/1, ou HP-UX sur HP PA-RISC, le PC sous Linux reste la plate-forme la mieux prise en charge.

Les programmes utilisateur utilisant l’AFAPI de TTL_PAPERS attaquent directement les registres matériels du port parallèle sous Linux, sans effectuer d’appel système à chaque accès. Pour ce faire, l’AFAPI demande d’abord les droits d’accès au port parallèle en utilisant soit `iopl()`, soit `ioperm()`. Le problème est que, l’un comme l’autre, ces appels obligent le programme appelant à être privilégié, ce qui introduit une faille de sécurité potentielle. La solution réside en un correctif optionnel à appliquer au noyau Linux et permettant à un processus privilégié de contrôler les permissions d'accès aux ports d'entrée/sortie pour n'importe quel autre processus.

3.2.20. USB (« *Universal Serial Bus* »)

- Prise en charge par Linux : *pilotes du noyau*
- Bande passante maximum : *12 Mbits/s*
- Temps de latence minimum : *?*
- Disponibilité : *dans le commerce*
- Interface ou bus utilisé : *USB*
- Structure du réseau : *bus*
- Coût par machine connectée : *5 dollars*

USB (« *Universal Serial Bus* », http://www.usb.org) est un bus fonctionnant à la vitesse de l'Ethernet conventionnel, dont les périphériques qui s'y rattachent peuvent être connectés à chaud (« *Hot-Plug* » : sans imposer la mise hors tension préalable du bus) et pouvant accueillir simultanément jusqu'à 127 de ces périphériques pouvant s'étendre du clavier à la caméra de vidéo-conférence. La manière dont on relie plusieurs ordinateurs par le biais de l'USB n'est pas clairement définie. Quoi qu'il en soit, les ports USB sont en train de s'établir très rapidement en standard sur les cartes-mères, au même titre que le port série RS232 ou le port parallèle, aussi ne soyez pas surpris si vous voyez apparaître un ou deux ports USB [20 [p 71]] sur votre prochain PC.

D'une certaine manière, l'USB est pratiquement la version basse performance à prix nul du FireWire que l'on peut se procurer aujourd'hui.

3.2.21. WAPERS

- Prise en charge par Linux : *bibliothèque AFAPI*
- Bande passante maximum : *0,4 Mbits/s*
- Temps de latence : *3 microsecondes*
- Disponibilité : *modèle dans le domaine public*
- Interface ou bus utilisé : *SPP (Port Parallèle Standard)*
- Structure du réseau : *modèle de câblage entre 2 à 64 machines*
- Coût par machine connectée : *5 dollars*

WAPERS (« *Wired-AND Adapter for Parallel Execution and Rapid Synchronization* », soit « Adaptateur par ET Câblé pour l'Exécution en Parallèle et la Synchronisation Rapide ») est une des facettes du projet PAPERS, de la Purdue University School of Electrical and Computer Engineering (« École Supérieure d'Électricité et d'Ingénierie en Informatique »). S'il est construit proprement, le port parallèle possède quatre bits de sortie à collecteur ouvert qui peuvent être câblés entre eux pour former un

ET câblé de 4 bits de large. Ce ET câblé est assez sensible électriquement, et le nombre maximum de machines qui peuvent y être reliées dépend de façon critique des propriétés analogiques des ports (les limites maximum des puits de courant et les valeurs des résistances de *pull-up*). On peut typiquement relier 7 à 8 machines en réseau de cette façon, avec WAPERS. Bien que les coûts et les temps de latences soient très réduits, la bande passante l'est aussi. WAPERS est donc bien plus indiqué comme réseau secondaire dédié aux fonctions d'agrégations que comme unique réseau d'un *cluster*. Comme pour TTL_PAPERS, il existe un correctif noyau visant à améliorer la sécurité, recommandé mais non requis.

3.3. Interface Logicielle Réseau

Avant d'explorer les ressources logicielles existantes en matière de traitement en parallèle, il est utile de couvrir rapidement les bases de l'interface logicielle de bas niveau gérant l'électronique du réseau. Il n'existe en réalité que trois options de base : les *sockets*, les pilotes de périphériques et les bibliothèques utilisateur.

3.3.1. Les *sockets*

Le *socket* est de loin la plus courante des interfaces réseau de bas niveau. Les *sockets* sont partie intégrante d'Unix depuis plus d'une décennie et la plupart des standards dans le domaine du matériel électronique de réseau est conçue pour prendre en charge au moins deux types de protocoles de *socket* : TCP et UDP. Ces deux types de *sockets* vous permettent d'envoyer des blocs de données d'une longueur arbitraire d'une machine à l'autre, mais il existe plusieurs différences importantes. Typiquement, ils engendrent tous deux un temps de latence minimum d'environ 1000 microsecondes, même si les performances peuvent bien pires encore en fonction du trafic.

Ces types de *sockets* constituent l'interface logicielle réseau de base pour la majorité des logiciels de traitement en parallèle portables et de plus haut niveau. Par exemple, PVM utilise une combinaison de l'UDP et du TCP, aussi en connaître les différences vous aidera à affiner les performances de votre système. Ce qui suit n'est qu'un aperçu de TCP et UDP. Référez-vous aux pages du manuel et à un bon livre de programmation pour plus de détails.

3.3.1.1. Le protocole UDP (SOCK_DGRAM)

UDP signifie « *User Datagram Protocol* » ou « Protocole de Datagrammes Utilisateur » mais il est plus facile de se souvenir des propriétés d'UDP en tant que « *Unreliable Datagram Processing* », ou « Traitement des Datagrammes Peu fiable ». En d'autres termes, UDP permet à chaque bloc d'être émis comme un message individuel, mais un message peut être perdu pendant la transmission. De fait, selon l'état du trafic sur le réseau, certains messages UDP peuvent être perdus, arriver plusieurs fois, ou arriver dans un ordre différent de celui dans lequel ils ont été émis. L'expéditeur d'un message UDP ne reçoit pas systématiquement d'accusé de réception, et c'est donc au programme écrit par l'utilisateur qu'il appartient de détecter et compenser ces problèmes. Heureusement, le protocole UDP garantit que si un message arrive, son contenu sera intact (c'est-à-dire que vous ne recevrez jamais un message incomplet).

Le bon coté de l'UDP est qu'il tend à être le plus rapide des protocoles des *socket*. En outre, UDP est « orienté hors connexion » (« *connectionless* »), ce qui signifie que chaque message est essentiellement indépendant des autres. On peut comparer chaque message à une lettre à La Poste. Vous pouvez envoyer plusieurs lettres à la même adresse, mais chacune d'entre elles est indépendante des autres, et vous n'êtes pas limité quant aux nombre de personnes à qui vous pouvez en envoyer.

3.3.1.2. Le protocole TCP (SOCK_STREAM)

Contrairement à l'UDP, le *TCP* est un protocole fiable et orienté connexion. Chaque bloc est considéré non pas comme un message, mais comme un bloc de données appartenant à un flot d'octets voyageant au travers d'une connexion établie entre l'expéditeur et le destinataire. Ce principe est très différent du système de messages de l'UDP car chaque bloc n'est qu'une partie du flot d'octets, et il appartient au programme utilisateur de trouver le moyen de les isoler car il n'y a aucune marque de séparation pour les distinguer. De plus, les connexions sont plus vulnérables aux perturbations du réseau, et seul un nombre limité de connexions simultanées peut exister au sein d'un même processus. Parce qu'il est fiable, le TCP engendre souvent des *overheads* plus importants que l'UDP.

Le TCP réserve en revanche quelques bonnes surprises. Par exemple, si plusieurs messages sont envoyés à travers une connexion, TCP est capable de les rassembler dans une mémoire tampon pour mieux correspondre aux tailles standard des paquets de l'électronique du réseau, ce qui peut donner de meilleurs résultats que l'UDP dans le cas de groupes de messages courts ou de taille inhabituelle. Un autre avantage : Les réseaux bâtis sur des connexions physiques directes entre deux machines peuvent facilement et efficacement être assimilés à des connexions TCP. Ce fut le cas pour la « *Socket Library* » (« bibliothèque de gestion de *Sockets* »), présentant une gestion compatible TCP au niveau de l'utilisateur qui ne différait des appels systèmes TCP standard que par le préfixe `PSS`, que l'on rajoutait au début du nom des fonctions à invoquer.

3.3.2. Les pilotes de périphériques

Lorsque l'on en arrive au stade où il faut effectivement injecter des données sur le réseau ou les y en extraire, l'interface logicielle des Unix standard fait partie du noyau et s'appelle « pilote » (« *driver* »). UDP et TCP ne se contentent pas de transporter des données, ils s'accompagnent également d'importants *overheads* dus à la gestion des *sockets*. Par exemple, il faut que quelque chose s'occupe du fait que plusieurs connexions TCP peuvent partager la même interface réseau physique. Par opposition, un pilote de périphérique dédié à une interface réseau n'a besoin de mettre en œuvre qu'un petit nombre de fonctions de transport élémentaires. Ces pilotes peuvent alors être invoqués à l'aide de l'appel `open()` pour identifier le périphérique adéquat, puis en utilisant par exemple `read()` et `write()` sur le « fichier » ouvert. Ainsi, chaque opération peut transporter un bloc de données en coûtant à peine plus cher qu'un appel système, ce qui permet d'atteindre des délais de l'ordre de quelques dizaines de microsecondes.

Écrire un pilote de périphérique pour Linux n'est pas difficile… pourvu que vous sachiez *parfaitement* comme fonctionne votre périphérique. Si vous n'en êtes pas sûr, n'essayez pas de le deviner. Déboguer un pilote de périphérique n'est pas une chose amusante, et faire des erreurs peut coûter la vie à votre matériel. En revanche, si ces risques ne vous effraient pas, il est possible d'écrire un pilote pour, par exemple, utiliser des cartes Ethernet dédiées comme des connexions machine-vers-machine « bêtes » mais très rapides car exonérées du protocole Ethernet habituel. Pour être exact, c'est pratiquement la technique utilisée par les premiers supercalculateurs Intel. Référez-vous au *Device-Driver-HOWTO* pour plus d'informations.

3.3.3. Bibliothèques utilisateurs

Si vous avez pris des cours de programmation système, on a dù vous y apprendre qu'accéder directement aux registres matériels des périphériques à partir d'un programme utilisateur était l'exemple typique de ce qu'il ne faut pas faire, parce que l'un des principes même d'un système d'exploitation est de contrôler l'accès aux périphériques. Cependant, le simple fait de passer un appel système coûte au minimum quelques dizaines de microsecondes. Dans le cas d'interfaces réseau bâties sur mesure comme TTL_PAPERS, qui peut effectuer des opérations de base sur un réseau en seulement 3 microsecondes, un tel surcoût pour un appel système est intolérable. Le seul moyen d'éviter ce temps d'attente est de faire en sorte que du code s'exécutant au niveau de l'utilisateur, donc une « bibliothèque au niveau de l'utilisateur » [21 [p 72]], puisse accéder directement au matériel, mais sans remettre en cause la souveraineté du système d'exploitation sur la gestion des droits d'accès aux ressources matérielles.

Sur un système typique, les seuls moyens, pour une bibliothèque utilisateur, d'accéder directement aux registres du matériel sont les suivants :

1. Au lancement du programme utilisateur, faire un appel système pour mapper l'espace d'adressage qui contient les registres du périphérique dans le plan mémoire du processus utilisateur. Sur certains systèmes, l'appel système `mmap()` (traité pour la première fois dans la section 2.6) peut être utilisé pour mapper un fichier spécial représentant les adresses de la page de mémoire physique du périphérique. Il est en même temps relativement simple d'écrire un pilote de périphérique effectuant cette opération. De plus, ce pilote peut obtenir l'accès en ne mappant que la ou les pages qui contiennent les registres nécessaires, en maintenant ainsi le contrôle des droits d'accès sous la coupe du système d'exploitation.

2. Accéder ensuite aux registres du périphérique sans passer par un appel système en se contentant de charger ou de ranger des valeurs sur la plage d'adressage mappée. Par exemple, un `*((char *) 0x1234) = 5;` déposera un octet de valeur 5 à l'adresse mémoire 1234 en hexadécimal.

Par bonheur, il se trouve que Linux pour Intel 386 et compatibles offre une solution meilleure encore :

1. En invoquant l'appel système `ioperm()` depuis un processus privilégié, obtenir la permission d'accéder aux ports d'entrée/sortie correspondant précisément aux registres du périphérique. Parallèlement, ces permissions peuvent être gérées par un processus utilisateur privilégié et indépendant (autrement dit : un « méta-système d'exploitation ») en utilisant l'appel système *giveioperm()*, disponible sous la forme d'un correctif à appliquer au noyau Linux.

2. Accéder aux registres du périphérique sans appel système en utilisant les instructions assembleur d'accès aux ports d'entrée/sortie du 386.

Cette seconde solution est préférable car il arrive souvent que les registres de plusieurs périphériques soient réunis sur une même page, auquel cas la première méthode ne pourrait offrir de protection contre l'accès aux autres registres résidant dans la même page que ceux appartenant au périphérique concerné. L'inconvénient est que ces instructions ne peuvent bien sûr pas être écrites en langage C. Il vous faudra à la place utiliser un peu d'assembleur. La fonction utilisant l'assembleur en ligne intégré à GCC (donc utilisable dans les programmes C) permettant de lire un octet depuis un port est la suivante :

```
extern inline unsigned char
inb(unsigned short port)
{
    unsigned char _v;
__asm__ __volatile__ ("inb %w1,%b0"
                      :"=a" (_v)
                      :"d" (port), "0" (0));
    return _v;
}
```

La fonction symétrique permettant l'émission d'un octet est :

```
extern inline void
outb(unsigned char value,
unsigned short port)
{
__asm__ __volatile__ ("outb %b0,%w1"
                      :/* pas de valeur retournée */
                      :"a" (value), "d" (port));
}
```

3.4. PVM (« *Parallel Virtual Machine* »)

PVM (pour « *Parallel Virtual Machine* », soit « Machine Virtuelle en Parallèle ») est une bibliothèque de *message-passing* portable et disponible gratuitement, s'appuyant généralement directement sur les *sockets*. Cette bibliothèque s'est incontestablement établie comme le standard *de facto* dans le domaine du traitement en parallèle à l'aide de *clusters* à transmission de messages.

PVM prend en charge les machines Linux monoprocesseur et SMP, comme les *clusters* de machines Linux reliées entre elles à l'aide par des réseaux reconnaissant les *sockets* (donc SLIP, PLIP, Ethernet, ATM). En fait, PVM fonctionnera même à travers un groupe de machines utilisant différents types de processeurs, de configurations et de réseaux physiques — Un *cluster hétérogène* — même si l'envergure de ce *cluster* est de l'ordre de la mise en parallèle de machines en utilisant Internet pour les relier entre elles. PVM offre même des facilités de contrôles de tâches (« *jobs* ») en parallèle au travers d'un *cluster*. Cerise sur le gâteau, PVM est disponible gratuitement et depuis longtemps (actuellement sur http://www.epm.ornl.gov/pvm/pvm_home.html), ce qui a conduit bon nombre de langages de programmation, de compilateurs, et d'outils de débogage ou autres à l'adopter comme leur « bibliothèque cible portable de *message-passing* ». Il existe également un groupe de discussion : news:comp.parallel.pvm.

Il est important de remarquer, en revanche, que les appels PVM ajoutent généralement aux opérations *socket* un *overhead* non négligeable alors que les temps de latence de celles-ci sont déjà importants. En outre, les appels eux-mêmes ne sont pas aisés à manipuler.

Appliquée au même exemple de calcul de Pi décrit en section 1.3, la version PVM du programme en langage C est la suivante :

```
#include <stdlib.h>
#include <stdio.h>
#include <pvm3.h>

#define NPROC   4

main(int argc, char **argv)
{
  register double sommelocale, largeur;
  double somme;
  register int intervalles, i;
  int mytid, iproc, msgtag = 4;
  int tids[NPROC];  /* Tableau des numéros des tâches */

  /* Début du traitement avec PVM */
  mytid = pvm_mytid();

  /* « Je rejoins le groupe et, si je suis la première instance,
     iproc=0, je crée plusieurs copies de moi-même. »
```

```
  */
  iproc = pvm_joingroup("pi");

  if (iproc == 0) {
    tids[0] = pvm_mytid();
    pvm_spawn("pvm_pi", &argv[1], 0, NULL, NPROC-1, &tids[1]);
  }
  /* On s'assure que tous les processus sont prêts  */
  pvm_barrier("pi", NPROC);

  /* Récupère le nombre d'intervalles */
  intervalles = atoi(argv[1]);
  largeur = 1.0 / intervalles;

  sommelocale = 0.0;
  for (i = iproc; i<intervalles; i+=NPROC) {
    register double x = (i + 0.5) * largeur;
    sommelocale += 4.0 / (1.0 + x * x);
  }

  /* On ajuste les résultats locaux en fonction de la largeur */
  somme = sommlocale * largeur;
  pvm_reduce(PvmSum, &sum, 1, PVM_DOUBLE, msgtag, "pi", 0);

  /* Seul le processus rattaché à la console renvoie le résultat */
  if (iproc == 0) {
    printf("Estimation de la valeur de pi: %f\n", somme);
  }

  /* On attend que le programme soit terminé,
     on quitte le groupe et
         on sort de PVM. */
  pvm_barrier("pi", NPROC);
  pvm_lvgroup("pi");
  pvm_exit();
  return(0);
}
```

3.5. MPI (« *Message Passing Interface* »)

Bien que PVM soit le standard de fait en matière de bibliothèque de *message-passing*, MPI (« *Message Passing Interface* ») tend à devenir le nouveau standard officiel. Le site du standard MPI se trouve sur http://www.mcs.anl.gov:80/mpi/ et le groupe de discussion correspondant sur news:comp.parallel.mpi.

En revanche, avant d'explorer MPI, je me sens obligé de parler rapidement de la guerre de religion qui oppose PVM à MPI et qui dure depuis quelques années. Je ne penche ni pour l'un ni pour l'autre. Voici un résumé aussi impartial que possible des différences entre les deux interfaces :

Environnement de contrôle de l'exécution

> Pour faire simple, PVM en a un, et MPI ne précise pas s'il existe, ni comment il doit être implémenté. Cela signifie que certaines choses comme lancer l'exécution d'un programme PVM se fait de la même manière partout, alors que pour MPI, cela dépend de l'implémentation utilisée.

Prise en charge des *clusters* hétérogènes.

PVM a grandi dans le monde de la collecte des cycles machines inutilisés sur les stations de travail, et sait donc gérer donc directement les mélanges hétérogènes de machines et de systèmes d'exploitation. A contrario, MPI part du principe général que la cible est un MPP (« *Massively Parallel Processor* », soit « Processeur Massivement Parallèle ») ou un *cluster* dédié de stations de travail pratiquement toutes identiques.

Syndrome de l'évier.

PVM se révèle être conçu pour une catégorie d'utilisation bien définie, ce que MPI 2.0 ne fait pas. Le nouveau standard MPI 2.0 inclut une variété de fonctionnalités qui s'étendent bien au delà du simple modèle de *message-passing*, comme le RMA (« *Remote Memory Access* », soit « Accès Mémoire à Distance ») ou les opérations d'entrée/sortie en parallèle sur les fichiers. Toutes ces choses sont-elles bien utiles ? Assurément… mais assimiler MPI 2.0 est comparable à réapprendre depuis zéro un langage de programmation totalement nouveau.

Conception de l'interface utilisateur.

MPI a été conçu après PVM, et en a incontestablement tiré les leçons. MPI offre une gestion des tampons plus simple et plus efficace et une couche d'abstraction de haut-niveau permettant de transmettre des données définies par l'utilisateur comme des messages.

Force de loi.

Pour ce que j'ai pu en voir, il existe toujours plus d'applications conçues autour de PVM qu'autour de MPI. Néanmoins, porter celles-ci vers MPI est chose facile, et le fait que MPI soit soutenu par un standard formel très répandu signifie que MPI est, pour un certain nombre d'institutions, une question de vision des choses.

Conclusion ? Disons qu'il existe au moins trois versions de MPI développées de façon indépendante et disponibles gratuitement pouvant fonctionner sur des *clusters* de machines Linux (et j'ai écrit l'un d'eux) :

- LAM (« *Local Area Multicomputer* », soit « MultiOrdinateur Local ») est une mise en œuvre complète du standard 1.1. Il permet aux programmes MPI de s'exécuter sur un système Linux individuel ou au travers d'un *cluster* de systèmes Linux communiquant par le biais de *sockets* TCP/UDP. Le système inclut des facilités de base de contrôle de l'exécution, ainsi que toute une gamme d'outils de développement et de débogage de programmes.

- MPICH (« *MPI CHameleon* ») est conçu pour être une implémentation complète et hautement portable du standard MPI 1.1. Tout comme LAM, il permet aux programmes MPI d'être exécutés sur des systèmes Linux individuels ou en *clusters* via une communication par *socket* TCP/UDP. En revanche, l'accent est porté sur la promotion de MPI en fournissant une implémentation efficace et facilement repositionnable. Pour porter cette implémentation, il faut réimplémenter soit les cinq fonctions de la « channel interface », soit, pour de meilleures performances, la totalité de l'ADI (« *Abstract Device Interface* », soit « Interface Périphérique Abstraite »). MPICH, et beaucoup d'informations concernant ce sujet et la façon de le porter, sont disponibles sur http://www.mcs.anl.gov/mpi/mpich/.

- AFMPI (« *Aggregate Function MPI* ») est une sous-implémentation du standard MPI 2.0. C'est celle que j'ai écrite. S'appuyant sur AFAPI, elle est conçue pour être la vitrine des RMA et des fonctions de communications collectives, et n'offre donc qu'un soutien minimal des types MPI, de ses systèmes de communication, et cætera. Elle permet à des programmes C utilisant MPI d’être exécutés sur un système Linux individuel ou au travers d’un *cluster* mis en réseau par du matériel pouvant prendre en charge l’AFAPI.

Quelque soit l’implémentation MPI utilisée, il est toujours très simple d’effectuer la plupart des types de communication.

En revanche, MPI 2.0 incorpore plusieurs paradigmes de communication suffisamment différents entre eux fondamentalement pour qu’un programmeur utilisant l’un d’entre eux puisse ne même pas reconnaître les autres comme étant des styles de programmation MPI. Aussi, plutôt que d’explorer un seul exemple de programme, il est utile de passer en revue un exemple de chacun des (différents) paradigmes de communication de MPI. Tous les programmes qui suivent emploient le même algorithme (celui de la section 1.3) utilisé pour calculer Pi.

Le premier programme MPI utilise les appels de *message-passing* MPI sur chaque processeur pour que celui-ci renvoie son résultat partiel au processeur 0, qui fait la somme de tous ces résultats et la renvoie à l’écran :

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
  register double largeur;
  double somme, sommelocale;
  register int intervalles, i;
  int nproc, iproc;
  MPI_Status status;

  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
  intervalles = atoi(argv[1]);
  largeur = 1.0 / intervalles;
  sommelocale = 0;
  for (i=iproc; i<intervalles; i+=nproc) {
    register double x = (i + 0.5) * largeur;
    sommelocale += 4.0 / (1.0 + x * x);
  }
  sommelocale *= largeur;
  if (iproc != 0) {
    MPI_Send(&lbuf, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  } else {
    somme = sommelocale;
    for (i=1; i<nproc; ++i) {
      MPI_Recv(&lbuf, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
               MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      somme += sommelocale;
    }
    printf("Estimation de la valeur de pi: %f\n", somme);
  }
  MPI_Finalize();
  return(0);
}
```

Le second programme MPI utilise les communications collectives (qui, pour ce cas précis, sont incontestablement les plus appropriées) :

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
  register double largeur;
  double somme, sommelocale;
  register int intervalles, i;
  int nproc, iproc;

  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
  intervalles = atoi(argv[1]);
  largeur = 1.0 / intervalles;
  sommelocale = 0;
  for (i=iproc; i<intervalles; i+=nproc) {
    register double x = (i + 0.5) * largeur;
    sommelocale += 4.0 / (1.0 + x * x);
  }
  sommelocale *= largeur;
  MPI_Reduce(&sommelocale, &somme, 1, MPI_DOUBLE,
             MPI_SUM, 0, MPI_COMM_WORLD);
  if (iproc == 0) {
    printf("Estimation de la valeur de pi: %f\n", somme);
  }
  MPI_Finalize();
  return(0);
}
```

La troisième version MPI utilise le mécanisme RMA de MPI 2.0 sur chaque processeur pour ajouter la valeur locale de la variable `sommelocale` de ce dernier à la variable `somme` du processeur 0 :

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
  register double largeur;
  double somme = 0, sommelocale;
  register int intervalles, i;
  int nproc, iproc;
  MPI_Win somme_fen;

  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
  MPI_Win_create(&somme, sizeof(somme), sizeof(somme),
                 0, MPI_COMM_WORLD, &somme_fen);
  MPI_Win_fence(0, somme_fen);
  intervalles = atoi(argv[1]);
  largeur = 1.0 / intervalles;
  sommelocale = 0;
  for (i=iproc; i<intervalles; i+=nproc) {
    register double x = (i + 0.5) * largeur;
    sommelocale += 4.0 / (1.0 + x * x);
  }
  sommelocale *= largeur;
  MPI_Accumulate(&sommelocale, 1, MPI_DOUBLE, 0, 0,
```

```
                1, MPI_DOUBLE, MPI_SUM, somme_fen);
  MPI_Win_fence(0, somme_fen);
  if (iproc == 0) {
    printf("Estimation de la valeur de pi: %f\n", somme);
  }
  MPI_Finalize();
  return(0);
}
```

Il est utile de préciser que le mécanisme RMA de MPI 2.0 prévient de façon remarquable tout problème de structures de données se trouvant à des adresses mémoires différentes selon les processeurs, en se référant à une "fenêtre" incluant l'adresse de base, une protection contre les accès mémoire hors de portée, et même le rééchelonnement d'adresse. À une implémentation efficace, s'ajoute le fait qu'un traitement RMA peut-être reporté jusqu'à la prochaine `MPI__Win_fence`. Pour faire simple, le mécanisme RMA est un étrange croisement entre mémoire partagée distribuée et *message passing*, mais reste une interface très propre pouvant générer des communications très efficaces.

3.6. AFAPI (« *Aggregate Function API* »)

Contrairement à PVM, MPI, et cætera, l'interface AFAPI (« *Aggregate Function API* », ou « Interface à Fonctions d'Agrégation ») n'a pas débuté sa vie en tant que couche d'abstraction portable s'appuyant sur un réseau matériel ou logiciel existant. AFAPI était plutôt la bibliothèque de gestion bas niveau d'un matériel spécifique pour PAPERS (« *Purdue's Adapter for Parallel Execution and Rapid Synchronization* », soit « Adaptateur pour l'Exécution en Parallèle et la Synchronisation Rapide de l'université de Purdue »).

PAPERS a été rapidement présenté dans la section 3.2. Il s'agit d'un réseau à fonction d'agrégations conçu sur mesure dont le modèle est dans domaine public et qui présente des temps de latence inférieurs à quelques microsecondes. Mais surtout, il s'agit de la tentative de construction d'un supercalculateur formant une meilleure cible pour la technologie des compilateurs que les supercalculateurs déjà existants. Il se distingue en qualité de la plupart des efforts en matière de *clusters* Linux et de PVM/MPI, qui s'attachent généralement à essayer d'exploiter les réseaux standard au profit des rares applications en parallèle présentant une granularité suffisante. Le fait que les éléments de PAPERS soient des machines PC sous Linux ne sert qu'à permettre l'implémentation de prototypes à des coûts les plus avantageux possibles.

La nécessité d'avoir une interface logicielle de bas niveau commune à plus d'une douzaine d'implémentations différentes d'un prototype a conduit la bibliothèque PAPERS à être standardisée sous le nom d'AFAPI. Mais le modèle utilisé par AFAPI est simple en lui-même et bien plus adapté aux interactions à la granularité plus fine, typiquement du code compilé par des compilateurs parallélisés, ou écrit pour des architectures SIMD. Non seulement la simplicité du modèle rend les machines PAPERS aisées à construire, mais elle apporte également une efficacité surprenante aux ports d'AFAPI sur différents types de système, tels que les SMP.

AFAPI fonctionne actuellement sur des *clusters* Linux utilisant TTL_PAPERS, CAPERS ou WAPERS. Elle fonctionne également (sans appel système ni même instruction de verrouillage de bus, voir la section 2.2) sur les machines SMP utilisant une bibliothèque de gestion de mémoire partagée type System V (« *System V Shared Memory* ») appelée SHMAPERS. Une version fonctionnant sur des *clusters* Linux utilisant la diffusion UDP sur des réseaux conventionnels (Ex : Ethernet) est en cours de développement. Toutes les versions d'AFAPI sont écrites pour être appelées à partir des langages C ou C++.

L'exemple suivant est la version AFAPI du programme de calcul de Pi décrit dans la section 1.3.

```
#include <stdlib.h>
#include <stdio.h>
#include "afapi.h"

main(int argc, char **argv)
{
  register double largeur, somme;
  register int intervalles, i;

  if (p_init()) exit(1);

  intervalles = atoi(argv[1]);
  largeur = 1.0 / intervalles;

  sum = 0;
  for (i=IPROC; i<intervalles; i+=NPROC) {
    register double x = (i + 0.5) * largeur;
    somme += 4.0 / (1.0 + x * x);
  }

  somme = p_reduceAdd64f(somme) * largeur;

  if (IPROC == CPROC) {
    printf("Estimation de la valeur de pi: %f\n", somme);
  }

  p_exit();
  return(0);
}
```

3.7. Autres bibliothèques de gestion de *clusters*

Outre PVM, MPI et AFAPI, les bibliothèques suivantes proposent des services qui peuvent s'avérer utiles au travers de grappes de machines Linux. Ces systèmes sont traités ici de manière moins approfondie simplement parce que, contrairement à PVM, MPI et AFAPI, je n'ai que peu, voire aucune expérience pratique de l'utilisation de ceux-ci sur des *clusters* Linux. Si l'une de ces bibliothèques (ou même d'autres) vous est particulièrement utile, merci de m'envoyer un courrier électronique en anglais à <hankd CHEZ engr POINT uky POINT edu> en me détaillant vos découvertes. J'envisagerai alors d'ajouter une section plus complète à son sujet.

3.7.1. Condor (migration de processus)

Condor est un système de gestion de ressources distribuées qui peut diriger de vastes *clusters* de stations de travail hétérogènes. Sa conception a été motivée par les besoins des utilisateurs souhaitant utiliser la puissance inexploitée de tels *clusters* au profit de leurs tâches aux temps d'exécution prolongés et aux calculs intensifs. Condor reproduit dans une large mesure l'environnement de la machine initiale sur celle qui exécute le processus, même si ces deux machines ne partagent pas un système de fichier ou un mécanisme de mot de passe communs. Les tâches sous Condor qui se résument à un processus unique sont automatiquement interceptées et déplacées entre les différentes stations en fonctions des besoins pour les mener à terme.

Condor est disponible sur http://www.cs.wisc.edu/condor/. Une version Linux existe également. Contactez l'administrateur du site, <condor TIRET admin CHEZ cs POINT wisc POINT edu>, pour plus de détails.

3.7.2. DFN-RPC (Réseau Allemand de la Recherche — « *Remote Procedure Call* »)

Le DFN-RPC (un outil du Réseau Allemand de la Recherche — « *Remote Procedure Call* ») a été développé pour distribuer et paralléliser des applications d'intéret scientifique ou technique entre une station de travail et un serveur de calcul ou un *cluster*. L'interface est optimisée pour les applications écrites en Fortran, mais le DFN-RPC peut aussi être utilisé dans un environnement de langage C. Une version Linux a été écrite. Plus d'information sur ftp://ftp.uni-stuttgart.de/pub/rus/dfn_rpc/README_dfnrpc.html.

3.7.3. DQS (« *Distributed Queueing System* »)

Pas vraiment une bibliothèque, DQS 3.0 (« Distributed Queueing System », soit « Système de Files d'attente Distribuées ») est un système de mise en file d'attente des tâches qui a été développé et testé sous Linux. Ce système a été conçu pour permettre à la fois l'utilisation et l'administration d'un *cluster* de machines hétérogènes comme une seule entité. Disponible sur http://www.scri.fsu.edu/~pasko/dqs.html.

Il existe aussi une version commerciale nommée CODINE 4.1.1 (« *COmputing in DIstributed Network Environments* », soit « CAlcul en Environnement Réseau Distribué »).

3.8. Références générales aux *clusters*

Les *clusters* peuvent être construits et utilisés de tellement de manières différentes que certains groupes ont apporté des contributions particulièrement intéressantes. Ce qui suit fait référence aux différents projets liés à la mise en place de *clusters* pouvant avoir un intérêt d'ordre général. Ceci inclut un mélange de références à des *clusters* spécifiques à Linux et à des *clusters* génériques. Cette liste est présentée dans l'ordre alphabétique.

3.8.1. Beowulf

Le projet Beowulf, se focalise sur la production de logiciels pour une utilisation de stations de travail immédiatement disponibles basée sur du matériel PC de grande distribution, un réseau à haut débit interne au *cluster*, et le système d'exploitation Linux.

Thomas Sterling a été le principal acteur de Beowulf, et continue d'être un promoteur franc et éloquent de l'utilisation de *clusters* Linux dans le domaine du calcul scientifique en général. À vrai dire, plusieurs groupes parlent à présent de leur *cluster* comme de système de « classe Beowulf », et ce même si la conception de ce *cluster* s'éloigne du modèle Beowulf officiel.

Don Becker, apportant son appui au projet Beowulf, a produit nombre des pilotes réseau utilisés par Linux en général. Plusieurs de ces pilotes ont même été adaptés pour être utilisés sous BSD. C'est également à Don que l'on doit la possibilité, pour certains pilotes, de répartir le trafic réseau à travers plusieurs connexions parallèles pour augmenter les taux de transfert sans utiliser d'onéreux commutateurs. Ce type de répartition de la charge réseau était le principal atout des *clusters* Beowulf.

3.8.2. Linux/AP+

Le projet Linux/AP+ ne concerne pas exactement le *clustering* sous Linux, mais s'attache à faire fonctionner Linux sur l'AP1000+ de Fujitsu, et à y apporter les améliorations appropriées en matière de traitement en parallèle. L'AP1000+ est une machine en parallèle à base de SPARC et disponible dans le commerce, utilisant un réseau spécifique avec une topologie en tore, un taux de transfert de 25Mo/s et un temps de latence de 10 microsecondes… Pour faire court, cela ressemble beaucoup à un *cluster* Linux SPARC.

3.8.3. Locust

Le projet Locust est en train de mettre au point un système de mémoire partagée virtuelle qui utilise les informations obtenues à la compilation pour masquer les temps de latence des messages et réduire le trafic réseau lors de l'exécution. « Pupa » forme la base du système de communication de Locust, et est implémenté à l'aide d'un réseau Ethernet reliant des machines PC 486 sous FreeBSD. Et Linux ?

3.8.4. Midway DSM (« *Distributed Shared Memory* »)

Midway est une DSM (« *Distributed Shared Memory* », soit « Mémoire Partagée Distribuée ») logicielle, similaire à TreadMarks. Le bon coté réside en l'utilisation d'indications à la compilation plutôt que de relativement lents mécanismes d'erreur de page, et en sa gratuité. Le mauvais coté est cela ne fonctionne pas sur des *clusters* Linux.

3.8.5. Mosix

MOSIX apporte des modifications au système d'exploitation BSD « BSDI » pour proposer une répartition de charge réseau (« *load balancing* ») dynamique et une migration de processus préemptive au travers d'un groupe de PC mis en réseau. C'est un système très utile non seulement pour le traitement en parallèle, mais d'une manière générale pour utiliser un *cluster* comme une machine SMP évolutive. Y aura-t-il une version Linux ? Voyez http://www.mosix.org pour plus d'informations [22 [p 72]].

3.8.6. NOW (« *Network Of Workstations* »)

Le projet NOW (« *Network Of Workstations* », ou « Réseau de Stations de Travail ») de l'université de Berkeley (http://now.cs.berkeley.edu) a conduit dans une large mesure l'effort pour le calcul en parallèle en utilisant des réseaux de stations de travail. Bon nombre de travaux sont menés là-bas, tous tournés vers la « démonstration en pratique d'un système à 100 processeurs dans les prochaines années ». Hélas, ils n'utilisent pas Linux.

3.8.7. Traitement en parallèle avec Linux

Le site web du « Traitement en parallèle avec Linux » (« *Parallel processing using Linux* »), sur http://yara.ecn.purdue.edu/~pplinux>, est le site officiel de ce guide pratique et de plusieurs documents en rapport avec ce thème, y compris des présentations en ligne. Parallèlement aux travaux du projet PAPERS, l'École Supérieure d'Électricité et d'Informatique de Purdue (« Purdue University School of Electrical and Computer Engineering ») reste un leader en matière de traitement en parallèle. Ce site a été mis en place pour aider les autres à utiliser des PC sous Linux pour faire du traitement en parallèle.

Depuis l'assemblage du premier *cluster* de PC Linux en février 1994, bien d'autres furent également assemblés à Purdue, dont plusieurs équipés de murs vidéos. Bien que ces *clusters* s'appuyaient sur des machines à base de microprocesseurs 386, 486 ou Pentium (mais pas de Pentium Pro), Intel a récemment accordé à Purdue une donation qui lui permettra de construire plusieurs grands *clusters* de systèmes à Pentium II (avec pas moins de 165 machines par *cluster*). Même si tous ces *clusters* sont ou seront équipés de réseaux PAPERS, la plupart sont également dotés de réseaux conventionnels.

3.8.8. Pentium Pro Cluster Workshop

Les 10 et 11 avril 1997, le laboratoire AMES a tenu à Des Moines, dans l'état de l'Iowa aux États-Unis, le « *Pentium Pro Cluster Workshop* » (« Atelier de *clusters* Pentium Pro »). Le site web de cet atelier, http://www.scl.ameslab.gov, renferme une mine d'informations concernant les *clusters* PC, glanées auprès de tous les participants.

3.8.9. TreadMarks DSM (« *Distributed Shared Memory* »)

La DSM (« *Distributed Shared Memory* », ou « Mémoire Partagée Distribuée ») est une technique avec laquelle un système de *message-passing* peut se présenter et agir comme un SMP. Il existe quelques systèmes de ce genre, la plupart utilisant les mécanismes d'erreur de page du système d'exploitation pour déclencher la transmission des messages. TreadMarks est l'un des plus efficaces, et fonctionne sur les *clusters* Linux. La mauvaise nouvelle est que « TreadMarks est distribué à un coût réduit aux universités et organisations à but non lucratif ». Pour plus d'informations concernant le logiciel, prenez contact (en anglais) avec <`tmk CHEZ cs POINT rice POINT edu`>.

3.8.10. U-Net (« *User-level NETwork interface architecture* »)

Le projet U-Net (« *User-level NETwork interface architecture* », ou « Architecture d'interface Réseau au Niveau Utilisateur »), accessible sur http://www.eecs.harvard.edu/~mdw/proj/old/unet, tente d'apporter temps de latence réduits et taux de transfert élevés sur du matériel réseau du commerce en virtualisant les interfaces réseau de manière à ce que les applications puissent envoyer et recevoir des messages sans passer par un appel système. U-Net fonctionne sur des PC Linux en utilisant du matériel Fast Ethernet basé sur une puce DEC DC21140, ou une carte ATM Fore Systems PCA-200 (mais pas PCA-200E).

3.8.11. WWT (« *Wisconsin Wind Tunnel* »)

On trouve bon nombre de projets relatifs à l'utilisation de *clusters* dans le Wisconsin. Le projet WWT (« *Wisconsin Wind Tunnel* », ou « Soufflerie du Wisconsin »), sur http://www.cs.wisc.edu/~wwt/, mène toutes sortes de travaux orientés vers le développement d'une interface « standard » entre les compilateurs et le matériel réseau sur lequel ils s'appuient. Il existe le « *Wisconsin COW* » (pour « *Cluster Of Workstation* »), « *Cooperative Shared Memory* » et « *Tempest* », le « *Paradyn Parallel Performance Tools* », et cætera. Malheureusement, il n'y a pas grand chose concernant Linux.

4. SIMD *Within A Register* : SWAR (Ex : utilisation de MMX)

Le SIMD (« *Single Instruction stream, Multiple Data stream* » ou « Un seul flux d'instruction, plusieurs flux de données ») à l'Intérieur d'Un Registre (ou « SIMD *Within A Register* » : SWAR) n'est pas un concept récent. En considérant une machine dotée de registres, bus de données et unités de fonctions de *k* bits, il est connu depuis longtemps que les opérations sur les registres ordinaires peuvent se comporter comme des opérations parallèles SIMD sur *n* champs de *k*/*n* bits chacun. Ce n'est en revanche qu'avec les efforts récents en matière de multimédia que l'accélération d'un facteur deux à huit apportée par les techniques SWAR a commencé à concerner l'informatique générale. Les versions de 1997 de la plupart des microprocesseurs incorporent une prise en charge matérielle du SWAR[23 [p 72]] :

- AMD K6 MMX (« *MultiMedia eXtensions* »)
- Sun SPARC V9 VIS (« *Visual Instruction Set* »)

Il existe quelques lacunes dans la prise en charge matérielle apportée par les nouveaux microprocesseurs, des caprices comme par exemple la prise en charge d'un nombre limité d'instructions pour certaines tailles de champ. Il est toutefois important de garder à l'esprit que bon nombre d'opérations SWAR peuvent se passer d'accélération matérielle. Par exemple, les opérations au niveau du bit sont insensibles au partitionnement logique d'un registre.

4.1. Quels usages pour le SWAR ?

Bien que *tout* processeur moderne soit capable d'exécuter un programme en effectuant un minimum de parallélisme SWAR, il est de fait que même le plus optimisé des jeux d'instructions SWAR ne peut gérer un parallélisme d'intérêt vraiment général. A dire vrai, de nombreuses personnes ont remarqué que les différences de performance entre un Pentium ordinaire et un Pentium « avec technologie MMX » étaient surtout dûes à certains détails, comme le fait que l'augmentation de la taille du cache L1 coïncide avec l'apparition du MMX. Alors, concrètement, à quoi le SWAR (ou le MMX) est-il utile ?

- Dans le traitement des entiers, les plus courts étant les meilleurs. Deux valeurs 32 bits tiennent dans un registre MMX 64 bits, mais forment aussi une chaîne de huit caractères, ou encore permettent de représenter un échiquier complet, à raison d'un bit par case. Note : il *existera une version « virgule flottante » du MMX*, bien que très peu de choses aient été dites à ce sujet. Cyrix a déposé une présentation, ftp://ftp.cyrix.com/developr/mpf97rm.pdf, qui contient quelques commentaires concernant *MMFP*. Apparemment, MMFP pourra prendre en charge le chargement de nombres 32 bits en virgule flottante dans des registres MMX 64 bits. Ceci combiné à deux pipelines MMFP fournirait quatre FLOP [24 [p 72]] en simple précision par cycle d'horloge.
- Pour le SIMD, ou parallélisme vectorisé. La même opération s'applique à tous les champs, simultanément. Il existe des moyens d'annihiler ses effets sur des champs sélectionnés (l'équivalent du « *SIMD enable masking* », ou « activation du masquage »), mais ils sont compliqués à mettre en œuvre et grèvent les performances.
- Pour les cas où la référence à la mémoire se fait de manière localisée et régulière (et de préférence regroupée). Le SWAR en général, et le MMX en particulier se révèlent désastreux sur les accès aléatoires. Rassembler le contenu d'un vecteur `x[y]` (où `y` est l'index d'un tableau) est extrêmement coûteux.

Il existe donc d'importantes limitations, mais ce type de parallélisme intervient dans un certain nombre d'algorithmes, et pas seulement dans les applications multimédia. Lorsque l'algorithme est adapté, le SWAR est plus efficace que le SMP ou le parallélisme en *clusters*… et son utilisation ne coûte rien !

4.2. Introduction à la programmation SWAR

Le concept de base du SWAR (« *SIMD Within A Register* », ou « SIMD à l'intérieur d'un registre ») réside dans le fait que l'on peut utiliser des opérations sur des registres de la taille d'un mot pour accélérer les calculs en effectuant des opérations parallèles SIMD sur n champs de k/n bits. En revanche, employer les techniques du SWAR peut parfois s'avérer maladroit, et certaines de leurs opérations peuvent au final être plus coûteuses que leurs homologues en série car elles nécessitent des instructions supplémentaires pour forcer le partitionnement des champs.

Pour illustrer ce point, prenons l'exemple d'un mécanisme SWAR grandement simplifié gérant quatre champs de 8 bits dans chaque registre de 32 bits. Les valeurs de ces deux registres pourraient être représentées comme suit :

```
        PE3     PE2     PE1     PE0
      +-------+-------+-------+-------+
Reg0  | D 7:0 | C 7:0 | B 7:0 | A 7:0 |
      +-------+-------+-------+-------+
Reg1  | H 7:0 | G 7:0 | F 7:0 | E 7:0 |
      +-------+-------+-------+-------+
```

Ceci indique simplement que chaque registre est vu essentiellement comme un vecteur de quatre valeurs entières 8 bits indépendantes. Alternativement, les valeurs `A` et `E` peuvent être vues comme les valeurs, dans Reg0 et Reg1, de l'élément de traitement 0 (ou « PE0 » pour « *Processing Element 0* »), `B` et `F` comme celles de l'élément 1, et ainsi de suite.

Le reste de ce document passe rapidement en revue les classes de base des opérations parallèles SIMD sur ces vecteurs d'entiers et la façon dont ces fonctions peuvent être implémentées.

4.2.1. Opérations polymorphiques

Certaines opérations SWAR peuvent être effectuées de façon triviale en utilisant des opérations ordinaires sur des entiers 32 bits, sans avoir à se demander si l'opération est réellement faite pour agir en parallèle et indépendemment sur ces champs de 8 bits. On qualifie ce genre d'opération SWAR de *polymorphique*, parce que la fonction est insensible aux types des champs (et à leur taille).

Tester si un champ quelconque est non-nul est une opération polymorphique, tout comme les opérations logiques au niveau du bit. Par exemple, un « ET » logique bit-à-bit ordinaire (l'opérateur « `&` » du registre C) effectue son calcul bit-à-bit, quelque soit la taille des champs. Un « ET » logique simple des registres ci-dessus donnerait :

```
         PE3       PE2       PE1       PE0
      +---------+---------+---------+---------+
Reg2  | D&H 7:0 | C&G 7:0 | B&F 7:0 | A&E 7:0 |
      +---------+---------+---------+---------+
```

Puisque le bit de résultat *k* de l'opération « ET » logique n'est affecté que par les valeurs des bits d'opérande *k*, toutes les tailles de champs peuvent être prises en charge par une même instruction.

4.2.2. Opérations partitionnées

Malheureusement, de nombreuses et importantes opérations SWAR ne sont pas polymorphiques. Les opérations arithmétiques comme l'addition, la soustraction, la multiplication et la division sont sujettes aux interactions de la « retenue » entre les champs. Ces opérations sont dites *partitionnées* car chacune d'elles doit cloisonner effectivement les opérandes et le résultat pour éviter les interactions entre champs. Il existe toutefois trois méthodes différentes pouvant être employées pour parvenir à ces fins :

4.2.2.1. Instructions partitionnées

L'approche la plus évidente pour implémenter les opérations partitionnées consiste peut-être à proposer une prise en charge matérielle des « instructions parallèles partitionnées » coupant le système de retenue entre les champs. Cette approche peut offrir les performances les plus élevées, mais elle nécessite la modification du jeu d'instruction du processeur et implique en général des limitations sur la taille des champs (Ex : ces instructions pourraient prendre en charge des champs larges de 8 bits, mais pas de 12).

Le MMX des processeurs AMD, Cyrix et Intel, Le MAX de Digital, celui d'HP, et le VIS de Sun implémentent tous des versions restreintes des instructions partitionnées. Malheureusement, ces différents jeux d'instructions posent des restrictions assez différentes entre elles, ce qui rend les algorithmes difficilement portables. Étudions à titre d'exemple l'échantillon d'opérations partitionnées suivant :

Instruction	AMD/Cyrix/Intel MMX	DEC MAX	HP MAX	Sun VIS
Différence Absolue		8		8
Maximum		8, 16		
Comparaison	8, 16, 32			16, 32
Multiplication	16			8x16
Addition	8, 16, 32		16	16, 32

Dans cette table, les chiffres indiquent les tailles de champ reconnues par chaque opération. Même si la table exclut un certain nombre d'instructions dont les plus exotiques, il est clair qu'il y a des différences. Cela a pour effet direct de rendre inefficaces les modèles de programmation en langages de haut niveau, et de sévèrement restreindre la portabilité.

4.2.2.2. Opérations non partitionnées avec code de correction

Implémenter des opérations partitionnées en utilisant des instructions partitionnées est certainement très efficace, mais comment faire lorsque l'opération dont vous avez besoin n'est pas prise en charge par le matériel ? Réponse : utiliser une série d'instructions ordinaires pour effectuer l'opération malgré les effets de la retenue entre les champs, puis effectuer la correction de ces effets indésirés.

C'est une approche purement logicielle, et les corrections apportent bien entendu un surcoût en temps, mais elle fonctionne pleinement avec le partitionnement en général. Cette approche est également « générale » dans le sens où elle peut être utilisée soit pour combler les lacunes du matériel dans le domaine des instructions partitionnées, soit pour apporter un soutien logiciel complet aux machines qui ne sont pas du tout dotées d'une prise en charge matérielle. À dire vrai, en exprimant ces séquences de code dans un langage comme le C, on permet au SWAR d'être totalement portable.

Ceci soulève immédiatement une question : quelle est précisément l'inefficacité des opérations SWAR partitionnées simulées à l'aide d'opérations non partitionnées ? Eh bien c'est très certainement la question à 65536 dollars, mais certaines opérations ne sont pas aussi difficiles à mettre en œuvre que l'on pourrait le croire.

Prenons en exemple le cas de l'implémentation de l'addition d'un vecteur de quatre éléments 8 bits contenant des valeurs entières, soit `x+y`, en utilisant des opérations 32 bits ordinaires.

Une addition 32 bits ordinaire pourrait en fait rendre un résultat correct, mais pas si la retenue d'un champ de 8 bits se reporte sur le champ suivant. Aussi, notre but consiste simplement à faire en sorte qu'un tel report ne se produise pas. Comme l'on est sûr qu'additionner deux champs de k bits ne peut générer qu'un résultat large d'au plus k+1 bits, on peut garantir qu'aucun report ne va se produire en « masquant » simplement le bit de poids fort de chaque champs. On fait cela en appliquant un « ET » logique à chaque opérande avec la valeur `0x7f7f7f7f`, puis en effectuant un addition 32 bits habituelle.

```
t = ((x & 0x7f7f7f7f) + (y & 0x7f7f7f7f));
```

Ce résultat est correct… sauf pour le bit de poids fort de chacun des champs. Il n'est question, pour calculer la valeur correcte, que d'effectuer deux additions 1-bit partitionnées, depuis `x` et `y` vers le résultat à 7 bits calculé pour `t`. Heureusement, une addition partitionnée sur 1 bit peut être implémentée à l'aide d'une opération « OU Exclusif » logique ordinaire. Ainsi, le résultat est tout simplement :

```
(t ^ ((x ^ y) & 0x80808080))
```

D'accord. Peut-être n'est-ce pas si simple, en fin de compte. Après tout, cela fait six opérations pour seulement quatre additions. En revanche, on remarquera que ce nombre d'opérations n'est pas fonction du nombre de champs. Donc, avec plus de champs, on gagne en rapidité. À dire vrai, il se peut que l'on gagne quand même en vitesse simplement parce que les champs sont chargés et redéposés en une seule opération (vectorisée sur des entiers), que la disponibilité des registres est optimisée, et parce qu'il y a moins de code dynamique engendrant des dépendances (parce que l'on évite les références à des mots incomplets).

4.2.2.3. Contrôler les valeurs des champs

Alors que les deux autres approches de l'implémentation des opérations partitionnées se centrent sur l'exploitation du maximum d'espace possible dans les registres, il peut être, au niveau du calcul, plus efficace de contrôler les valeurs des champs de façon à ce que l'interaction de la retenue entre ceux-ci ne se produise jamais. Par exemple, si l'on sait que toutes les valeurs de champs additionnées sont telles qu'aucun dépassement ne peut avoir lieu, une addition partitionnée peut être implémentée à l'aide d'une instruction ordinaire. Cette contrainte posée, une addition ordinaire pourrait en fin de compte apparaître polymorphique, et être utilisable avec n'importe quelle taille de champ sans programme de correction. Ce qui nous amène ainsi à la question suivante : Comment s'assurer que les valeurs des champs ne vont pas provoquer d'événements dus à la retenue ?

Une manière de faire cela consiste à implémenter des instructions partitionnées capables de restreindre la portée des valeurs des champs. Les instructions vectorisées de maximum et de minimum du MAX de Digital peuvent être assimilées à une prise en charge matérielle de l'écrêtage des valeurs des champs pour éviter les interactions de la retenue entre les champs.

En revanche, si l'on ne dispose pas d'instructions partitionnées à même de restreindre efficacement la portée des valeurs des champs, existe-t-il une condition qui puisse être imposée à un coût raisonnable et qui soit suffisante pour garantir le fait que les effets de la retenue n'iront pas perturber les champs adjacents ? La réponse se trouve dans l'analyse des propriétés arithmétiques. Additionner deux nombres de *k* bits renvoie un résultat large d'au plus *k*+1 bits. Ainsi, un champ de *k*+1 bits peut recevoir en toute sécurité le résultat d'une telle opération, même s'il est produit par une instruction ordinaire.

Supposons donc que les champs de 8 bits de nos précédents exemples soient désormais des champs de 7 bits avec « espaces de séparation pour retenue » d'un bit chacun :

```
            PE3            PE2            PE1            PE0
      +----+-------+----+-------+----+-------+----+-------+
Reg0  | D' | D 6:0 | C' | C 6:0 | B' | B 6:0 | A' | A 6:0 |
      +----+-------+----+-------+----+-------+----+-------+
```

Mettre en place un vecteur d'additions 7 bits se fait de la manière suivante : On part du principe qu'avant l'exécution de toute instruction partitionnée, les bits des séparateurs de retenue (`A'`, `B'`, `C'`, et `D'`) sont nuls. En effectuant simplement une addition ordinaire, tous les champs reçoivent une valeur correcte sur 7 bits. En revanche, certains bits de séparation peuvent passer à 1. On peut remédier à cela en appliquant une seule opération supplémentaire conventionnelle et masquant les bits de séparation. Notre vecteur d'additions entières sur 7 bits, `x+y`, devient ainsi :

```
((x + y) & 0x7f7f7f7f)
```

Ceci nécessite seulement deux opérations pour effectuer quatre additions. Le gain en vitesse est évident.

Les lecteurs avertis auront remarqué que forcer les bits de séparation à zéro ne convient pas pour les soustractions. La solution est toutefois remarquablement simple : Pour calculer `x-y`, on garantira simplement la condition initiale, qui veut que tous les bits de séparation de `x` valent 1 et que tous ceux de `y` soient nuls. Dans le pire des cas, nous obtiendrons :

```
(((x | 0x80808080) - y) & 0x7f7f7f7f)
```

On peut toutefois souvent se passer du « OU » logique en s'assurant que l'opération qui génère la valeur de `x` utilise `| 0x80808080` plutôt que `& 0x7f7f7f7f` en dernière étape.

Laquelle de ces méthodes doit-on appliquer aux opérations partitionnées du SWAR ? La réponse est simple : « celle qui offre le gain en rapidité le plus important ». Ce qui est intéressant, c'est que la méthode idéale peut être différente selon chaque taille de champ, et ce à l'intérieur d'un même programme, s'exécutant sur une même machine.

4.2.3. Opérations de communication et de conversion de type

Bien que certains calculs en parallèle, incluant les opérations sur les pixels d'une image, ont pour propriété le fait que la *n*ième valeur d'un vecteur soit une fonction des valeurs se trouvant à la *n*ième position des vecteurs des opérandes, ce n'est généralement pas le cas. Par exemple, même les opérations sur les pixels telles que l'adoucissement ou le flou réclament en opérande les valeurs des pixels adjacents, et les transformations comme la transformée de Fourrier (FFT) nécessitent des schémas de communications plus complexes (et moins localisés).

Il est relativement facile de mettre efficacement en place un système de communication à une dimension entre les valeurs du voisinage immédiat pour effectuer du SWAR, en utilisant des opérations de décalage non partitionnées. Par exemple, pour déplacer une valeur depuis `PE` vers `PE`(*n*+1), un simple décalage logique suffit. Si les champs sont larges de 8 bits, on utilisera :

```
(x << 8)
```

Pourtant, ce n'est pas toujours aussi simple. Par exemple, pour déplacer une valeur depuis `PE`*n* vers `PE`(*n*-1), un simple décalage vers la droite devrait suffire… mais le langage C ne précise pas si le décalage vers la droite conserve le bit de signe, et certaines machines ne proposent, vers la droite, qu'un décalage signé. Aussi, d'une manière générale, devons-nous mettre explicitement à zéro les bits de signe pouvant être répliqués.

```
((x >> 8) & 0x00ffffff)
```

L'ajout de connexions « à enroulement » (« *wrap-around* ») à l'aide de décalages non partitionnés [25 [p 72]] est aussi raisonnablement efficace. Par exemple, pour déplacer une valeur depuis `PE`*i* vers `PE`(*i*+1) avec enroulement :

```
((x << 8) | ((x >> 24) & 0x000000ff))
```

Les problèmes sérieux apparaissent lorsque des schémas de communications plus généraux doivent être mis en œuvre. Seul le jeu d'instructions du MAX de HP permet le réarrangement arbitraire des champs en une seule instruction, nommée `Permute`. Cette instruction `Permute` porte vraiment mal son nom : Non seulement elle effectue une permutation arbitraire des champs, mais elle autorise également les répétitions. En bref, elle met en place une opération arbitraire de `x[y]`.

Il reste malheureusement très difficile d'implémenter `x[y]` sans le concours d'une telle instruction. La séquence de code est généralement à la fois longue et inefficace, parce qu'en fait, il s'agit d'un programme séquentiel. Ceci est très décevant. La vitesse relativement élevée des opérations de type `x[y]` sur les les supercalculateurs SIMD MasPar MP1/MP2 et Thinking Machines CM1/CM2/CM200 était une des clés majeures de leur succès. Toutefois, un `x[y]` reste plus lent qu'une communication de proximité, même sur ces supercalculateurs. Beaucoup d'algorithmes ont donc été conçus pour réduire ces besoins en opérations de type `x[y]`. Pour simplifier, disons que sans soutien matériel, le meilleur est encore très certainement de développer des algorithmes SWAR en considérant `x[y]` comme étant interdit, ou au moins très coûteux.

4.2.4. Opérations récurrentes (réductions, balayages, et cætera)

Une récurrence est un calcul dans lequel il existe une relation séquentielle apparente entre les valeurs à traiter. Si toutefois des opérations associatives sont impliquées dans ces récurrences, il peut être possible de recoder ces calculs en utilisant un algorithme parallèle à structure organisée en arbre.

Le type de récurrence parallélisable le plus courant est probablement la classe connue sous le nom de réduction associative. Par exemple, pour calculer la somme des valeurs d'un vecteur, on écrit du code C purement séquentiel, comme :

```
t = 0;
for (i=0; i<MAX; ++i) t += x[i];
```

Cependant, l'ordre des additions est rarement important. Les opérations en virgule flottante et les saturations peuvent rendre différents résultats si l'ordre des additions est modifié, mais les additions ordinaires sur les entiers (et qui reviennent au début après avoir atteint la valeur maximum) renverront exactement les mêmes résultats, indépendemment de l'ordre dans lequel elles sont effectuées. Nous pouvons ainsi réécrire cette séquence sous la forme d'une somme parallèle structurée en arbre, dans laquelle nous additionnons d'abord des paires de valeurs, puis des paires de ces sous-totaux, et ainsi de suite jusqu'à l'unique somme finale. Pour un vecteur de quatre valeurs 8 bits, seulement deux additions sont nécessaires. La première effectue deux additions de 8 bits et retourne en résultat deux champs de 16 bits, contenant chacun un résultat sur 9 bits :

```
t = ((x & 0x00ff00ff) + ((x >> 8) & 0x00ff00ff));
```

La deuxième fait la somme de ces deux valeurs de 9 bits dans un champs de 16 bits, et renvoie un résultat sur 10 bits :

```
((t + (t >> 16)) & 0x000003ff)
```

En réalité, la seconde opération effectue l'addition de deux champs de 16 bits… mais les 16 bits de poids fort n'ont pas de signification. C'est pourquoi le résultat est masqué en une valeur de 10 bits.

Les balayages (« *scans* »), aussi connus sous le nom d'opérations « à préfixe parallèle » sont un peu plus difficile à mettre en œuvre efficacement. Ceci est dû au fait que contrairement aux réductions, les balayages peuvent produire des résultats partitionnées.

4.3. SWAR MMX sous Linux

Pour Linux, nous nous soucierons principalement des processeurs IA32. La bonne nouvelle, c'est qu'AMD, Cyrix et Intel implémentent tous le même jeu d'instructions MMX. En revanche, les performances de ces différents MMX sont variables. Le K6, par exemple, n'est doté que d'un seul *pipeline* là où le Pentium MMX en a deux. La seule nouvelle vraiment mauvaise, c'est qu'Intel diffuse toujours ces stupides films publicitaires sur le MMX… ;-)

Il existe trois approches sérieuses du SWAR par le MMX :

1. L'utilisation des fonctions d'une bibliothèque dédiée au MMX. Intel, en particulier, a développé plusieurs « bibliothèques de performances », offrant toute une gamme de fonctions optimisées à la main et destinées aux tâches multimédia courantes. Avec un petit effort, bon nombre d'algorithmes non-multimédia peuvent être retravaillés pour permettre à quelques unes des zones de calcul intensif de s'appuyer sur une ou plusieurs de ces bibliothèques. Ces bibliothèques ne sont pas disponibles sous Linux, mais pourraient être adaptées pour le devenir.

2. L'utilisation directe des instructions MMX. C'est assez compliqué, pour deux raisons : D'abord, le MMX peut ne pas être disponible sur le processeur concerné, ce qui oblige à fournir une alternative. Ensuite, l'assembleur IA32 utilisé en général sous Linux ne reconnaît actuellement pas les instructions MMX.

3. L'utilisation d'un langage de haut niveau ou d'un module du compilateur qui puisse directement générer des instructions MMX appropriées. Quelques outils de ce type sont actuellement en cours de développement, mais aucun n'est encore pleinement fonctionnel sous Linux. Par exemple, à l'Université de Purdue (http://dynamo.ecn.purdue.edu/~hankd/SWAR/), nous développons actuellement un compilateur qui admettra des fonctions écrites dans un « dialecte » explicite parallèle au langage C et qui générera des modules SWAR accessibles comme des fonctions C ordinaires, et qui feront usage de tous les supports SWAR disponibles, y compris le MMX. Les premiers prototypes de compilateurs à modules ont été générés à Fall, en 1996. Amener cette technologie vers un état réellement utilisable prend cependant beaucoup plus de temps que prévu initialement.

En résumé, le SWAR par MMX est toujours d'un usage malaisé. Toutefois, avec quelques efforts supplémentaires, la seconde approche décrite ci-dessus peut être utilisée dès à présent. En voici les bases :

1. Vous ne pourrez pas utiliser le MMX si votre processeur ne le prend pas en charge. Le code GCC suivant vous permettra de savoir si votre processeur est équipé de l'extension MMX. Si c'est le cas, la valeur renvoyée sera différente de zéro, sinon nulle.

```
inline extern
int mmx_init(void)
{
        int mmx_disponible;

        __asm__ __volatile__ (
                /* Récupère la version du CPU */
                "movl $1, %%eax\n\t"
                "cpuid\n\t"
                "andl $0x800000, %%edx\n\t"
                "movl %%edx, %0"
                : "=q" (mmx_disponible)
                : /* pas d'entrée */
        );
        return mmx_disponible;
}
```

2. Un registre MMX contient essentiellement ce que GCC appellerait un `unsigned long long`. Ainsi, ce sont des variables de ce type et résidant en mémoire qui vont former le mécanisme de communication entre les modules MMX et le programme C qui les appelle. Vous pouvez aussi déclarer vos données MMX comme étant des structures de données alignées sur des adresses multiples de 64 bits (il convient de garantir l'alignement sur 64 bits en déclarant votre type de données comme étant membre d'une `union` comportant un champ `unsigned long long`).

3. Si le MMX est disponible, vous pouvez écrire votre code MMX en utilisant la directive assemble `.byte` pour coder chaque instruction. C'est un travail pénible s'il est abattu à la main, mais pour un compilateur, les générer n'est pas très difficile. Par exemple, l'instruction MMX `PADDB MM0,MM1` pourrait être encodée par la ligne d'assembleur in-line GCC suivante :

   ```
   __asm__ __volatile__ (".byte 0x0f, 0xfc, 0xc1\n\t");
   ```

   Souvenez-vous que le MMX utilise une partie du matériel destiné aux opérations en virgule flottante, et donc que le code ordinaire mélangé au MMX ne doit pas invoquer ces dernières. La pile en virgule flottante doit également être vide avant l'exécution de tout code MMX. Cette pile est normalement vide à l'entrée d'une fonction C ne faisant pas usage de la virgule flottante.

4. Clôturez votre code MMX par l'exécution de l'instruction `EMMS`, qui peut être encodée par :

   ```
   __asm__ __volatile__ (".byte 0x0f, 0x77\n\t");
   ```

Tout ce qui précède paraît rébarbatif, et l'est. Ceci dit, le MMX est encore assez jeune… de futures versions de ce document proposeront de meilleures techniques pour programmer le SWAR MMX.

5. Processeurs auxiliaires des machines Linux

Même si cette approche n'est plus à la mode depuis quelques temps, il reste virtuellement impossible aux autres méthodes de maintenir à la fois prix bas et performances élevées, chose habituellement rendue possible par l'utilisation d'un système Linux, lorsqu'il s'agit d'héberger dans la machine une unité de calcul dédiée. Le problème est que la prise en charge logicielle de ces outils est très limitée. Nous serons plus ou moins livrés à nous mêmes.

5.1. Un PC Linux est une bonne station d'accueil

En général, les processeurs secondaires tendent à se spécialiser dans l'exécution de fonctions bien spécifiques.

Avant de se sentir découragés parce que livrés à nous-mêmes, il est utile de comprendre que, bien qu'il puisse être difficile de le préparer à recevoir un système particulier, un PC sous Linux reste l'une des rares plate-formes qui se prêtent convenablement à cet usage.

Un PC forme un bon système d'accueil pour deux raisons : La première est sa capacité à évoluer aisément et de façon peu onéreuse. Les ressources telles que la mémoire, les disques, le réseau et autres peuvent être ajoutées vraiment très facilement à un PC. La seconde est sa facilité d'interfaçage. Non seulement les prototypes de cartes ISA et PCI sont largement répandus et disponibles, mais le port parallèle fournit également des performances raisonnables dans une interface discrète. L'adressage séparé des entrées / sorties de l'architecture IA32 facilite également l'interfaçage en offrant une protection de ces adresses au niveau du port individuel.

Linux est également un bon système d'exploitation pour l'hébergement de systèmes dédiés. La libre et complète disponibilité du code source, et les nombreux manuels de programmation avancée constituent évidement une aide très précieuse. Mais Linux apporte également une gestion des tâches pratiquement temps-réel. Il existe même une version fonctionnant en vrai temps-réel. Mais une chose peut-être plus importante encore est le fait que, même dans un environnement Unix complet, Linux sait prendre en charge des outils de développement écrits pour fonctionner sous MS-DOS ou Windows. Les programmes DOS peuvent être exécutés dans un processus Unix en utilisant `dosemu`, qui dresse une machine virtuelle protégée qui peut littéralement exécuter du code MS-DOS. La prise en charge des programmes Windows 3.xx sous Linux est encore plus directe : certains logiciels libres comme `wine` simulent Windows 3.11 [26 [p 72]] suffisamment bien pour permettre à la plupart des programmes Windows d'être exécutés correctement sur une machine Unix équipée de X-Window.

Les deux sections suivantes donnent des exemples de systèmes parallèles auxiliaires que j'aimerais voir être exploités sous Linux :

5.2. Avez-vous essayé le DSP ?

Un marché prospère s'est développé autour des puces DSP (« *Digital Signal Processing* », ou « Traitement Numérique du Signal ») à hautes performances. Bien qu'elles soient en général conçues pour être embarquées dans des systèmes dédiés à des usages bien spécifiques, elles peuvent aussi être utilisées comme de très bons ordinateurs parallèles annexes. Voici pourquoi :

- Plusieurs d'entre elles, comme le TMS320 de Texas Instruments et la famille SHARC d'Analog Devices sont conçues pour permettre l'assemblage de machines parallèles en n'utilisant que très peu, voire aucun circuit logique intermédiaire.
- Ces puces sont bon marché, spécialement au niveau du coût par MIP ou par MFLOP. Prix des circuits de gestion logique de base compris, on peut trouver un processeur DSP pour le dixième du prix du processeur d'un PC, à performances comparables.
- Elle ne nécessitent que peu de puissance et ne dissipent pas beaucoup de chaleur. Cela signifie qu'il est possible d'alimenter toute une poignée de processeurs de ce type avec une simple alimentation PC conventionnelle, et de les enfermer dans le boîtier d'un PC sans transformer celui-ci en haut-fourneau.

- La plupart des jeux d'instructions des DSP contiennent des choses assez exotiques et que les langages de haut niveau (c'est-à-dire : comme le C) ne sont pas à même d'utiliser correctement. Par exemple, l'« adressage de bit inversé ». En utilisant des systèmes parallèles inclus dans une machine, il est possible de compiler et d'exécuter directement la plupart du code sur cette machine, tout en confiant l'exécution des quelques algorithmes consommant la plupart du temps système aux DSP, par l'entremise d'un code particulièrement soigné et optimisé à la main.

- Ces DSP ne sont pas vraiment conçus pour faire fonctionner un système d'exploitation UNIX ou assimilé, et ne sont généralement pas non plus adaptés à une utilisation autonome comme processeur général d'un ordinateur. Le système de gestion de la mémoire, par exemple, est insuffisant pour beaucoup d'entre eux. En d'autres mots, ils sont bien plus efficaces lorsqu'ils sont intégrés au sein d'une machine hôte plus polyvalente … telle qu'un PC sous Linux.

Bien que de nombreux modems et cartes son contiennent des processeurs DSP accessibles par des pilotes Linux, les grands profits arrivent avec l'utilisation de systèmes parallèles intégrés comprenant au moins quatre processeurs DSP.

Même si la série TMS320 de Texas Instruments, sur http://dspvillage.ti.com, est populaire depuis un bon moment, les systèmes de ce type étant disponibles sont encore peu nombreux. Il existe à la fois une version uniquement entière et une version virgule flottante du TMS320. Les anciens modèles utilisaient un format de virgule flottante en simple précision assez inhabituel, mais les nouveaux gèrent à présent les formats IEEE. Les anciens TMS320C4x (ou simplement « C4x »), accomplissaient jusqu'à 80 MFLOP en utilisant le format de nombre en virgule flottante simple précision spécifique à TI. En comparaison, un seul « C67x » apportera jusqu'à 1 GFLOP en simple précision et 420 MFLOP en double précision au format IEEE, en utilisant une architecture à base de VLIW nommée VelociTI. Non seulement il est aisé de configurer un groupe de circuits de ce type pour en faire un multiprocesseur, mais un de ces circuits, le multiprocesseur « 'C8x », formera à lui seul un processeur principal à technologie RISC accusant 100 MFLOP au format IEEE, et doté soit de deux, soit de quatre DSP esclaves intégrés.

L'autre famille de processeurs DSP qui ne soit pas simplement utilisée ces temps-ci par quelques rares systèmes en parallèle est SHARC (ou « ADSP-2106x »), d'Analog Devices. Ces circuits peuvent être configurés en un multiprocesseur à mémoire partagée formé par 6 processeurs sans nécessiter de logique externe pour les associer, et peuvent aussi former de plus grands systèmes grâce à des circuits de liaison de 4 bits. Les systèmes au-delà de celui-ci sont essentiellement destinés aux applications militaires, et reviennent assez cher. Toutefois, Integrated Computing Engines, Inc. produit un petit jeu de deux cartes PCI assez intéressant, nommé GreenICE. Cette unité contient un réseau de 16 processeurs SHARC, et affiche une vitesse de pointe d'1.9 GFLOP au format IEEE simple précision. GreenICE coûte moins de 5000 dollars.

À mon avis, les circuits DSP parallèles intégrés mériteraient une plus grande attention de la part du petit monde du calcul en parallèle sous Linux…

5.3. Calcul à l'aide des FPGA et circuits logiques reconfigurables

Si l'objectif final du traitement en parallèle reste l'atteinte des plus hautes vitesses possibles, et bien pourquoi ne pas fabriquer du matériel sur mesure ? Nous connaissons tous la réponse : Trop cher, trop long à développer, le matériel ainsi conçu devient inutile lorsque l'algorithme change même de façon minime, et cætera. Cependant, les récentes avancées faites dans le domaine des FPGA (« *Field Programmable Gate Array* », ou « Réseaux Logiques Programmables à effet de Champ ») ont réduit à néant la plupart de ces objections. Aujourd'hui, la densité des portes logiques est suffisamment élevée pour qu'un processeur entier puisse être intégré dans un seul FPGA, et le temps de reconfiguration (ou de reprogrammation) d'un de ces FPGA a également chuté à un niveau où l'on peut raisonnablement reconfigurer le circuit même entre deux phases d'un même algorithme.

Il ne faut pas avoir froid aux yeux pour utiliser ces techniques : Il faudra travailler avec des langages de description matérielle tels que le VHDL pour configurer les FPGA, tout comme écrire du code de bas niveau pour programmer les interfaces avec le système d'accueil Linux. En revanche, le coût des FPGA est assez bas, spécialement pour les algorithmes opérant sur des données entières et de précision normale (ce qui ne représente en fait qu'un surensemble restreint de tout ce à quoi le SWAR s'applique bien), ces FPGA peuvent effectuer des opérations complexes à peu près aussi vite qu'on peut les leur transmettre. Par exemple, de simples systèmes à base de FPGA ont accusé des performances supérieures à celles des supercalculateurs lors de recherches sur des bases de données génétiques.

Il existe plusieurs compagnies produisant du matériel approprié à base de FPGA, mais les deux suivantes en sont un bon exemple :

Virtual Computer Company propose toute une gamme de produits utilisant des FPGA Xilinx à base de SRAM et reconfigurables dynamiquement. Leur « Virtual ISA Proto Board » 8/16 bits coûte moins de 2000 dollars.

L'ARC-PCI d'Altera (« *Altera Reconfigurable Computer* », sur bus PCI), sur http://www.altera.com/html/new/pressrel/pr_arc-pci.html, est un type de carte similaire, mais utilisant les FPGA d'Altera et un bus PCI comme interface plutôt qu'un bus ISA.

Bon nombre de ces outils de conception, langages de description matérielle, compilateurs, routeurs, et cætera, sont livrés sous forme d'un code objet qui ne fonctionne que sous DOS ou Windows. Nous pourrions simplement conserver une partition DOS/Windows sur notre PC d'accueil et redémarrer lorsque nous en avons besoin. Toutefois, la plupart de ces outils peuvent probablement fonctionner sous Linux en utilisant `dosemu` ou des émulateurs Windows tels que `wine`.

6. D'intérêt général

Les sujets couverts dans cette section s'appliquent aux quatre modèles de traitement en parallèle sous Linux.

6.1. Compilateurs et langages de programmation

Je suis principalement un « chercheur en compilateurs ». Je devrais donc être capable de dire s'il y a beaucoup de compilateurs vraiment performants, générant du code parallèle efficace pour les systèmes Linux. Malheureusement, et pour dire la vérité, il est difficile de battre les performances obtenues en exprimant votre programme en parallèle avec des communications et autres opérations parallèles, le tout dans un code en langage C, compilé par GCC.

Les projets de compilateurs ou de langages suivants représentent une partie des meilleurs efforts produits pour la génération d'un code raisonnablement efficace depuis les langages de haut niveau. Généralement, chacun d'eux est raisonnablement efficace pour les tâches qu'il vise, mais aucun ne correspond aux langages et aux compilateurs puissants et tout-terrain qui vous feront abandonner définitivement l'écriture de programmes C compilés par GCC… ce qui est très bien comme çà. Utilisez ces langages et compilateurs à ce pour quoi ils ont été conçus, et vous serez récompensés par des durées de développement plus courtes, des opérations de maintenance et de débogages réduites, et cætera.

Il existe un grand nombre de langages et de compilateurs en dehors de ceux listés ici (dans l'ordre alphabétique). Une liste de compilateurs librement disponibles (la plupart n'ayant rien à voir avec le traitement en parallèle sous Linux) est accessible ici : http://www.idiom.com/free-compilers.

6.1.1. Fortran 66/77/PCF/90/HPF/95

Au moins dans la communauté de l'informatique scientifique, le Fortran sera toujours présent. Bien sûr, le Fortran d'aujourd'hui n'a plus la même signification que celle définie par le standard ANSI de 1966. En quelques mots, le Fortran 66 était très limité. Le Fortran 77 a ajouté des nuances dans ses fonctions, la plus intéressante étant la prise en charge améliorée des données de type caractère et la modification de la sémantique des boucles `DO`. Le Fortran PCF (« *Parallel Computing Forum* ») a tenté d'ajouter diverses fonctions de gestion de traitement en parallèle au Fortran 77. Le Fortran 90 est un langage moderne et pleinement fonctionnel, qui apporte essentiellement des facilités de programmation orientée objet ressemblant au C++ et une syntaxe de tableaux en parallèle au langage du Fortran 77. HPF (« *High-Performance Fortran* », lui-même proposé en deux versions (HPF-1 et HPF-2), est essentiellement la version standardisée et améliorée de ce que beaucoup d'entre nous ont connu sous les noms de CM Fortran, MasPar Fortran, ou Fortran D. Il étend le Fortran 90 avec diverses améliorations dédiées au traitement en parallèle, très centrées sur la spécification d'agencements de données. Citons enfin le Fortran 95, version quelque peu améliorée et raffinée du Fortran 90.

Ce qui fonctionne avec le C peut en général fonctionner aussi avec `f2c`, `g77`, ou les produits Fortran 90/95 commerciaux de NAG. Ceci est dû au fait que tous ces compilateurs peuvent au final produire le même code que celui utilisé en arrière-boutique par GCC.

Les paralléliseurs Fortran commerciaux pouvant générer du code pour SMP sont disponibles sur http://www.kai.com et http://www.crescentbaysoftware.com. Il n'est pas clairement indiqué si ces compilateurs sont utilisables pour des machines SMP Linux, mais ils devraient l'être étant donné que les *threads* POSIX standards (soit les LinuxThreads) fonctionnent sur des systèmes SMP sous Linux.

Le Portlan Group propose des compilateurs Fortran HPF (et C/C++) parallèles commerciaux générant du code Linux SMP. Il existe aussi une version ciblant les *clusters* s'appuyant sur MPI ou PVM. Les produits de http://www.apri.com pourraient aussi être utiles sur des *clusters* ou des machines SMP.

Parmi les Fortran parallélisés disponibles librement et qui pourraient fonctionner sur des systèmes Linux en parallèle, citons :

- ADAPTOR (« *Automatic DAta Parallelism TranslaTOR* », http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html, qui peut traduire du Fortran HPF en Fortran 77/90 avec appels MPI ou PVM, mais qui ne mentionne pas Linux.
- Fx, de l'Université Carnegie Mellon (N.D.T. : Pittsburgh), vise certains *clusters* de stations de travail, mais Linux ?
- HPFC (un prototype de Compilateur HPF) génère du code Fortran 77 avec appels PVM. Est-il utilisable sur un *cluster* Linux ?
- PARADIGM (« *PARAllelizing compiler for DIstributed-memory General-purpose Multicomputers* », http://www.crhc.uiuc.edu/Paradigm) peut-il lui aussi être utilisé avec Linux ?

- Le compilateur Polaris génère du code Fortran pour multiprocesseurs à mémoire partagée, et pourrait bientôt être reciblé vers les *clusters* Linux PAPERS.

- PREPARE, http://www.irisa.fr/EXTERNE/projet/pampa/PREPARE/prepare.html, vise les *clusters* MPI… On ne sait pas vraiment s'il sait générer du code pour processeurs IA32.

- Combinant ADAPT et ADLIB, « shpf » (« *Subset High Performance Fortran compilation system* ») est dans le domaine public et génère du code Fortran 90 avec appels MPI. Donc, si vous avez un compilateur Fortran 90 sous Linux…

- SUIF (« *Stanford University Intermediate Form* », voir http://suif.stanford.edu) propose des compilateurs parallélisés pour le C et le Fortran. C'est aussi l'objectif du *National Compiler Infrastructure Project*… « Quelqu'un s'occupe-t-il des systèmes parallèles sous Linux ? »

Je suis sûr d'avoir omis plusieurs compilateurs potentiellement utiles dans les différents dialectes du Fortran, mais ils sont si nombreux qu'il est difficile d'en garder la trace. A l'avenir, je préférerais ne lister que ceux qui sont réputés fonctionner sous Linux. Merci de m'envoyer commentaires et corrections en anglais par courrier électronique à <`hankd CHEZ engr POINT uky POINT edu`>.

6.1.2. GLU (« *Granular Lucid* »)

GLU (« *Granular Lucid* ») est un système de programmation de très haut niveau basé sur un modèle hybride combinant les modèles intentionnel (« *Lucid* ») et impératif. Il reconnaît à la fois les *sockets* PVM et TCP. Fonctionne-t-il sous Linux ? (Le site du « *Computer Science Laboratory* » se trouve sur http://www.csl.sri.com.

6.1.3. Jade et SAM

Jade est un langage de programmation en parallèle sous forme d'extension au langage C et exploitant les concurrences de la granularité des programmes séquentiels et impératifs. Il s'appuie sur un modèle de mémoire partagée distribuée, modèle implémenté par SAM dans les *clusters* de stations de travail utilisant PVM. Plus d'informations sur http://suif.stanford.edu/~scales/sam.html.

6.1.4. Mentat et Legion

Mentat est un système de traitement en parallèle orienté objet qui fonctionne avec des *clusters* de stations de travail et qui a été porté vers Linux. Le « *Mentat Programming Language* » (MPL) est un langage de programmation orienté objet basé sur le C++. Le système temps-réel de Mentat utilise quelque chose qui ressemble vaguement à des appels de procédures à distance (« *remote procédure calls* » : RPC) non bloquantes. Plus d'informations sur http://www.cs.virginia.edu/~mentat.

Legion http://www.cs.virginia.edu/~legion est construit au dessus de Mentat, et simule une machine virtuelle unique au travers des machines d'un réseau large zone.

6.1.5. MPL (« *MasPar Programming Language* »)

À ne pas confondre avec le MPL de Mentat, ce langage était initialement développé pour être le dialecte C parallèle natif des supercalculateurs SIMD MasPar. MasPar ne se situe plus vraiment sur ce segment (ils s'appellent maintenant NeoVista Solutions et se spécialisent dans le *data mining*[27 [p 72]], mais leur compilateur MPL a été construit en utilisant GCC. Il est donc librement disponible. Dans un effort concerté entre les université de l'Alabama, à Huntsville, et celle de Purdue, le MPL MasPar a été reciblé pour générer du code C avec appels AFAPI (voir la section 3.6), et fonctionne ainsi à la fois sur des machines Linux de type SMP et sur des *clusters*. Le compilateur est, néanmoins, quelque peu bogué… voir http://www.math.luc.edu/~laufer/mspls/papers/cohen.ps.

6.1.6. PAMS (« *Parallel Application Management System* »)

Myrias est une compagnie qui vend un logiciel nommé PAMS (« *Parallel Application Management System* »). PAMS fournit des directives très simples destinées au traitement en parallèle utilisant de la mémoire partagée virtuelle. Les réseaux de machines Linux ne sont pas encore pris en charge.

6.1.7. Parallaxis-III

Parallaxis-III est un langage de programmation structurée qui reprend Modula-2, en ajoutant des « processeurs et connexions virtuels » pour le parallélisme des données (un modèle SIMD). La suite Parallaxis comprend des compilateurs pour ordinateurs séquentiels et parallèles, un débogueur (extension des débogueurs `gdb` et `xgdb`), et une vaste gamme d'algorithmes d'exemple de différents domaines, en particulier dans le traitement des images. Il fonctionne sur les systèmes Linux séquentiels. Une ancienne version prenait déjà en charge diverses plate-formes parallèles, et une prochaine version le fera à nouveau (comprendre : ciblera les *clusters* PVM). Plus d'informations sur http://www.informatik.uni-stuttgart.de/ipvr/bv/p3/p3.html.

6.1.8. pC++/Sage++

pC++/Sage++ est une extension au langage C autorisant les opérations de type « données en parallèle », en utilisant des « collections d'objets » formées à partir de classes « éléments » de base. Il s'agit d'un préprocesseur générant du code C++ pouvant fonctionner en environnement PVM. Est-ce que cela fonctionne sous Linux ? Plus d'informations ici : http://www.extreme.indiana.edu/sage.

6.1.9. SR (« *Synchronizing Resources* »)

SR (« *Synchronizing Resources* », ou « ressources synchronisées ») est un langage de programmation de situations concurrentes et dans lequel les ressources encapsulent les processus et les variables qu'ils partagent. Les « opérations » forment le principal mécanisme d'interactions entre processus. SR propose une intégration novatrice des mécanismes utilisées pour invoquer et traiter les opérations. En conséquence, les appels de procédures locaux ou à distance, les rendez-vous, le *message passing*, la création dynamique de processus, la multi-diffusion et les sémaphores sont tous pris en charge. SR gère également les variables et opérations globales partagées.

Il existe une adaptation Linux, mais le type de parallélisme que SR peut y faire fonctionner n'est pas clairement défini. Plus d'informations sur http://www.cs.arizona.edu/sr/www/index.html.

6.1.10. ZPL et IronMan

ZPL est un langage de programmation par matrices conçu pour gérer les applications scientifiques et d'ingénierie. Il génère des appels à une interface de *message-passing* assez simple nommée IronMan, et les quelques fonctions qui constituent cette interface peuvent facilement être implémentées à l'aide de pratiquement n'importe quel système de *message-passing*. Il est toutefois principalement tourné vers PVM et MPI sur des *clusters* de stations de travail, et peut fonctionner sous Linux. Plus d'informations sur http://www.cs.washington.edu/research/zpl/home/index.html.

6.2. Question de performance

Beaucoup de gens passent beaucoup de temps à faire des mesures de performances « au banc d'essai » de cartes-mères particulières, de carte réseau, et cætera, pour déterminer laquelle d'entre elles est la meilleure. Le problème de cette approche est que, le temps d'arriver à des résultats probants, le matériel testé peut ne plus être le meilleur. Il peut même être retiré du marché et remplacé par un modèle révisé et aux propriétés radicalement différentes.

Acheter du matériel pour PC, c'est un peu comme acheter du jus d'orange. Il est généralement fabriqué à partir de produits de bonne qualité, et ce quelque soit le nom de la compagnie qui le distribue. Peu de gens connaissent ou se soucient de la provenance de ces composant (ou du concentré de jus d'orange). Ceci dit, il existe quelques différences auxquelles on devrait être attentif. Mon conseil est simple : Soyez simplement conscients de ce que vous pouvez attendre de votre matériel lorsqu'il fonctionne sous Linux, puis portez votre attention sur la rapidité de livraison, un prix raisonnable et une garantie convenable.

Il existe un excellent aperçu des différents processeurs pour PC sur http://www.pcguide.com/ref/cpu/fam/; En fait, le site PC Guide entier est rempli de bonnes présentations techniques de l'électronique d'un PC. Il est également utile d'en savoir un peu sur les performances et la configuration d'un matériel spécifique, et le Linux Benchmarking HOWTO est un bon document pour commencer.

Les processeurs Intel IA32 sont dotés de plusieurs registres spéciaux qui peuvent être utilisés pour mesurer les performances d'un système en fonction dans ses moindres détails. Le VTune d'Intel, sur http://developer.intel.com/design/perftool/vtune, fait un usage poussé de ces registres de performances, dans un système d'optimisation du code vraiment très complet… qui malheureusement ne fonctionne pas sous Linux. Un pilote sous forme de module chargeable et une bibliothèque de fonctions servant à accéder aux registres de performances du Pentium sont disponibles et ont été écrits par Cuneyt Akinlar. Souvenez-vous que ces registres de performance sont différents selon les processeurs IA32. Ce code ne fonctionnera donc que sur un Pentium, pas sur un 486, ni sur un Pentium Pro, un Pentium II, un K6, et cætera.

Il y a encore un commentaire à faire à propos des performances, destiné spécialement à ceux qui veulent monter de grands *clusters* dans de petits espaces. Les processeurs modernes incorporent pour certains des capteurs de température, et utilisent des circuits ralentissant la cadence du processeur lorsque cette température dépasse un certain seuil (tentative de réduction des émissions de chaleur et d'amélioration de la fiabilité). Je ne suis pas en train de dire que tout le monde devrait sortir s'acheter un module à effet Peltier (ou « pompe à chaleur ») pour refroidir chaque CPU, mais que nous devrions être conscients du fait qu'une température trop élevé ne se contente pas de raccourcir la durée de vie des composants, mais agit aussi directement sur les performances d'un système. Ne placez vos ordinateurs dans des configurations pouvant bloquer le flux d'air, piégeant la chaleur dans des zones confinées, et cætera.

Enfin, les performances ne sont pas seulement une question de vitesse, mais également de fiabilité et de disponibilité. Une haute fiabilité signifie que votre système ne plantera pratiquement jamais, même si un composant vient à tomber en panne… ce qui nécessite généralement un matériel spécial comme une alimentation redondante et une carte-mère autorisant le remplacement « à chaud » des équipements qui y sont connectés. Tout cela est en général assez cher. Une haute disponibilité signifie que votre système sera prêt à être utilisé pratiquement tout le temps. Le système peut planter si l'un des composants tombe en panne, mais il pourra être réparé et redémarré très rapidement. Le High-Availability HOWTO traite plusieurs des cas de base de la haute disponibilité. En revanche, dans le cas des *clusters*, une haute disponibilité peut être assurée en prévoyant simplement quelques pièces de rechange. Éliminer le matériel défectueux et le remplacer par une de ces pièces de rechange peut apporter une plus grande disponibilité pour un coût d'entretien moins élevé que celui d'un contrat de maintenance.

6.3. Conclusion — C'est fini !

Alors, y a-t-il quelqu'un dans l'assemblée qui fasse du traitement en parallèle sous Linux ? Oui !

Il y a encore peu de temps, beaucoup de gens se demandaient si la mort de plusieurs compagnies fabricant des supercalculateurs parallèles n'annonçait pas la fin du traitement en parallèle. Ce n'était alors pas mon opinion (voir http://dynamo.ecn.purdue.edu/~hankd/Opinions/pardead.html pour un aperçu assez amusant de ce qui, à mon avis, s'est vraiment passé), et il semble assez clair qu'aujourd'hui le traitement en parallèle est à nouveau sur une pente ascendante. Même Intel, qui n'a cessé que récemment de produire des supercalculateurs parallèles, est fier du soutien apporté au traitement en parallèle par des choses comme le MMX, et l'EPIC (« *Explicitly Parallel Instruction Computer* », ou « Ordinateur à jeu d'Instructions Parallèles Explicites ») de l'IA64.

Si vous faites une recherche sur les mots « Linux » et « Parallèle » dans votre moteur de recherche favori, vous trouverez un certain nombre d'endroits impliqués dans le traitement en parallèle sous Linux. Les *clusters* de PC sous Linux, en particulier, font leur apparition un peu partout. Le fait que Linux se prête particulièrement bien à ce genre de tâche combiné aux coûts réduits et aux hautes performances du matériel pour PC ont fait du traitement en parallèle sous Linux une approche populaire, tant pour les groupes au budget restreint que pour les vastes laboratoires de recherche nationaux confortablement subventionnés.

Certains projets énumérés dans ce document tiennent une liste de sites de recherche « apparentés » utilisant des configurations Linux parallèles similaires. Vous trouvez cependant sur http://yara.ecn.purdue.edu/~pplinux/Sites un document hypertexte dont le but est d'apporter photographies, descriptions et contacts vers les sites de tous les projets utilisant Linux pour effectuer des traitement en parallèle. Si vous voulez voir votre propre site y figurer :

- Vous devez avoir un site « permanent » présentant le fonctionnement d'un système Linux en parallèle : Une machine SMP, un système SWAR, ou un PC équipé de processeurs dédiés, configuré pour permettre aux utilisateurs d'*exécuter des programmes parallèles sous Linux*. Un environnement logiciel basé sur Linux et prenant directement en charge la gestion du traitement en parallèle (tel que PVM, MPI ou AFAPI) doit être installé sur le système. En revanche, le matériel ne doit pas nécessairement être dédié au traitement en parallèle sous Linux et peut être employé à des tâches complètement différentes lorsque les programmes parallèles ne sont pas en cours d'exécution.

- Vous devez formuler une demande pour que votre site apparaisse sur cette liste. Envoyez vos informations en anglais à <`hankd CHEZ engr POINT uky POINT edu`>. Merci de respecter le format utilisé par les autres entrées pour les informations concernant votre site. *Aucun site ne sera répertorié sans demande explicite de la part du responsable de ce site.*

Quatorze *clusters* sont actuellement répertoriés dans cette liste, mais nous avons été informés de l'existence de plusieurs douzaines de *clusters* Linux à travers le monde. Bien sûr, cette inscription n'implique aucun engagement. Notre objectif est simplement de favoriser le développement des connaissances, de la recherche et de la collaboration impliquant le traitement en parallèle sous Linux.

[1 [p 7]] N.D.T. : « *flat memory model* », dans lequel toute la mémoire se trouve dans le même plan mémoire, par opposition aux segments, mémoire paginée, et tout autre modèle composé.

[2 [p 11]] N.D.T. : décomposition d'un programme en plusieurs processus distincts, mais travaillant simultanément et de concert.

[3 [p 11]] N.D.T. : qui a démarré la machine avant de passer en mode SMP.

[4 [p 11]] N.D.T. : le lien vers l'ancienne version 1.1, lui, n'existe plus. La documentation la plus récente se trouve à ce jour sur http://www.intel.com/design/Pentium4/documentation.htm.

[5 [p 17]] N.D.T. : inséré au sein du code source, ici en C.

[6 [p 18]] N.D.T. : soit, sous Unix, être sous le compte `root`.

[7 [p 19]] N.D.T. : temporisations introduites au sein d'un programme en utilisant par exemple des boucles et en consommant ainsi tout le temps machine alloué au processus plutôt qu'en rendant la main au système.

[8 [p 20]] N.D.T. : « *spin locks* » : mise en état d'attente jusqu'à ce qu'une condition soit remplie.

[9 [p 24]] N.D.T. : Il s'agit en fait d'un appel Unix standard.

[10 [p 28]] N.D.T. : il semble que ce panel ne soit plus mis à jour depuis un certain temps. Le site suggéré proposait à la date de rédaction de la version française le bulletin le plus récent, mais n'est pas officiel.

[11 [p 33]] N.D.T. : désormais intégré aux sources du noyau.

[12 [p 33]] N.D.T. : les pilotes FC pour le noyau Linux ont en fait été écrits en 1999.

[13 [p 34]] N.D.T. : La *Fibre Channel Association* (FCA) et la *Fibre Channel Loop Community* (FCLC) ont fusionné en 2000 pour former la *Fibre Channel Industry Association*.

[14 [p 34]] N.D.T. : les premiers pilotes FireWire pour Linux ont été écrits en 1999 mais, bien que disponibles en standard, sont toujours considérés comme expérimentaux.

[15 [p 34]] N.D.T. : HiPPI est aujourd'hui pris en charge par Linux, mais de façon restreinte et expérimentale.

[16 [p 35]] N.D.T. : l'IrDA est pris en charge par le noyau depuis 2000.

[17 [p 36]] SAN : « *System Area Network* ».

N.D.T. : à ne pas confondre avec « *Storage Area Network* », qui partage le même acronyme.

[18 [p 37]] N.D.T. : Sequent a été absorbé par IBM en septembre 1999.

[19 [p 39]] N.D.T. : et donc désormais à Hewlett-Packard.

[20 [p 41]] N.D.T. : les ports et le standard USB sont aujourd'hui parfaitement reconnus par Linux.

[21 [p 44]] N.D.T. : par opposition à un « appel système », donc sans franchir la barrière du passage en mode noyau.

[22 [p 53]] N.D.T. : MOSIX fonctionne aujourd'hui sous Linux.

[23 [p 54]] N.D.T. : initialement, huit liens étaient proposés à cet endroit, devenus pour la plupart obsolètes.

[24 [p 55]] N.D.T. : FLOP = « *FLoating point OPeration* », ou « OPération en Virgule Flottante ».

[25 [p 60]] N.D.T. : il s'agit en fait de simuler la ré-entrée par le coté opposé des données éjectées par le décalage. L'exemple qui suit permet d'implémenter ce cas en langage C, mais la plupart des microprocesseurs sont équipés d'instructions de rotation effectuant cela en une seule opération.

[26 [p 63]] N.D.T. : WINE a évolué avec le temps et reconnaît naturellement les versions plus récentes de ce système d'exploitation.

[27 [p 67]] N.D.T. : le *data mining* consiste à étudier et mettre au point des techniques efficaces de recherche d'une information au travers d'une immense quantité de données.