

# Notes de Cours INF 8601

Olivier Sirois

2017-10-10

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Taxonomie de Flynn . . . . .	11
1.2	Parallélisme . . . . .	12
1.3	Associativité . . . . .	12
1.4	Fiabilité et Statistiques . . . . .	12
1.5	Mesure de performances . . . . .	13
1.6	Exercices . . . . .	14
1.6.1	1.1 . . . . .	14
1.6.2	1.2 . . . . .	14
1.6.3	1.3 . . . . .	15
1.6.4	1.4 . . . . .	15
1.6.5	1.5 . . . . .	15
1.6.6	1.6 . . . . .	16
1.6.7	1.7 . . . . .	16
1.6.8	1.8 . . . . .	16
1.6.9	1.9 . . . . .	16
1.6.10	1.10 . . . . .	17
<b>2</b>	<b>Pthread et TBB</b>	<b>18</b>
2.1	Fils d'exécution . . . . .	18
2.2	HyperThread . . . . .	18
2.3	Fils d'exécution en mode usager . . . . .	19
2.4	Processus régulier (PThread) . . . . .	19
2.4.1	Commandes . . . . .	19
2.5	Pthread . . . . .	20
2.5.1	Fonctions . . . . .	20
2.5.2	Exemples Pthreads . . . . .	22
2.6	TBB . . . . .	31
2.6.1	for . . . . .	31
2.6.2	réduction . . . . .	35
2.6.3	Scan . . . . .	36
2.6.4	Do . . . . .	37
2.6.5	Pipeline . . . . .	38
2.6.6	Sort . . . . .	39

2.6.7	Tâches . . . . .	39
2.7	Exercice . . . . .	40
2.7.1	2.1 . . . . .	40
2.7.2	2.2 . . . . .	40
2.7.3	2.3 . . . . .	41
2.7.4	2.4 . . . . .	41
<b>3</b>	<b>Cohérence Mémoire</b>	<b>42</b>
3.1	Cohérence de Cache . . . . .	42
3.2	Cohérence par répertoire . . . . .	44
3.3	interface logiciel cache et TLB . . . . .	45
3.4	Optimisation . . . . .	48
3.5	Contraintes d'ordonnancement . . . . .	49
3.6	Course entre deux processeur . . . . .	49
3.7	forcer un réordonnancement . . . . .	49
3.7.1	Instructions . . . . .	50
3.8	Exercice . . . . .	51
3.8.1	3.1 . . . . .	51
3.8.2	3.2 . . . . .	51
3.8.3	3.2 . . . . .	52
3.8.4	3.4 . . . . .	52
<b>4</b>	<b>Modèle de mémoire partagée</b>	<b>53</b>
4.1	Modèle Partagé . . . . .	53
4.1.1	Barrières . . . . .	55
4.2	résumé . . . . .	57
4.3	Exercices . . . . .	58
4.3.1	4.1 . . . . .	58
4.3.2	4.2 . . . . .	58
4.3.3	4.3 . . . . .	59
4.3.4	4.4 . . . . .	59
4.3.5	4.5 . . . . .	59
4.3.6	4.6 . . . . .	60
4.3.7	4.7 . . . . .	60
4.3.8	4.8 . . . . .	60
4.3.9	4.9 . . . . .	61
<b>5</b>	<b>OpenMP</b>	<b>62</b>
5.1	Directives . . . . .	64
5.1.1	parallel . . . . .	64
5.1.2	for . . . . .	64
5.1.3	Sections . . . . .	65
5.1.4	Tasks . . . . .	66
5.1.5	single . . . . .	66
5.1.6	Master . . . . .	67
5.1.7	Critical . . . . .	67

5.1.8	barrier . . . . .	68
5.1.9	taskwait . . . . .	68
5.1.10	atomic . . . . .	69
5.1.11	flush . . . . .	69
5.1.12	ordered . . . . .	70
5.1.13	Synchronisations . . . . .	70
5.2	Clauses . . . . .	70
5.2.1	Shared et Private . . . . .	70
5.2.2	Threadprivate . . . . .	71
5.2.3	Copyin . . . . .	71
5.2.4	réduction . . . . .	71
5.2.5	if . . . . .	71
5.2.6	Nowait . . . . .	71
5.2.7	Collapse . . . . .	72
5.2.8	SIMD . . . . .	72
5.2.9	Safelen . . . . .	72
5.2.10	Aligned . . . . .	72
5.2.11	Uniforme . . . . .	72
5.2.12	Target . . . . .	72
5.3	Fonctions de supports de threads . . . . .	74
5.4	Exercices. . . . .	75
5.4.1	5.1 . . . . .	75
5.4.2	5.2 . . . . .	76
5.4.3	5.3 . . . . .	76
5.4.4	4.4 . . . . .	77
<b>6</b>	<b>Contrôle des Années précédents.</b>	<b>78</b>
6.1	Année 2016 . . . . .	78
6.1.1	1 . . . . .	78
6.1.2	2 . . . . .	78
6.2	Années 2014 . . . . .	87
6.3	Année 2013 . . . . .	97
6.4	Année 2012 . . . . .	105
<b>7</b>	<b>MIPS - Assembleur Vectorielle</b>	<b>114</b>
7.1	Instructions multimédia . . . . .	117
7.2	Processeur Graphique . . . . .	117
7.2.1	Mémoire . . . . .	118
7.2.2	Instructions . . . . .	118
7.3	Exercices . . . . .	119
<b>8</b>	<b>OpenCL</b>	<b>122</b>
8.0.1	Code . . . . .	124
8.0.2	Fonctions Kernels - Sur Dispositif . . . . .	134
8.1	Optimisation . . . . .	137
8.2	Exemple TP2 . . . . .	138

8.3 Exercices . . . . .	141
<b>9 MPI . . . . .</b>	<b>146</b>
9.1 Message et Types . . . . .	147
9.1.1 Communications Globales . . . . .	151
9.2 Groupes de Communications . . . . .	153
9.3 Analyse de performance et optimisation . . . . .	156
9.3.1 Optimisation . . . . .	156
9.4 Conclusion . . . . .	157
9.5 Exercices . . . . .	157
<b>10 Outils de Vérif. et Analyse . . . . .</b>	<b>162</b>
10.1 LTTng . . . . .	162
10.2 gcov . . . . .	163
10.3 gprof . . . . .	163
10.4 Oprofile . . . . .	165
10.5 Valgrind . . . . .	165
10.5.1 Memcheck . . . . .	166
10.5.2 cachegrind . . . . .	167
10.5.3 Address Sanitizer . . . . .	167
10.5.4 Massif . . . . .	168
10.5.5 Helgrind . . . . .	168
10.6 Lockdep . . . . .	169
10.7 Thread Sanitizer . . . . .	169
10.8 CPU profile . . . . .	170
10.9 Heap Profile . . . . .	170
10.10 Traces d'exécution multi-coeurs . . . . .	170
10.10.1 Utilisation de LTTng . . . . .	171
10.10.2 Pourquoi une trace avec un cout minimale? . . . . .	172
10.10.3 Exemple d'optimisation . . . . .	174
10.11 Autres . . . . .	175
10.11.1 synchronisation de traces . . . . .	175
10.11.2 Analyse du chemin critique . . . . .	176
10.12 Vtune . . . . .	177
10.13 Windows Performance Toolkit . . . . .	178
10.14 Vampir . . . . .	179
10.15 PARaver . . . . .	179
10.16 Tuning Analysis Utilities . . . . .	179
10.17 CODEXL . . . . .	180
10.18 APITrace . . . . .	180
10.19 GPUView . . . . .	180
10.20 Promela et SPIN . . . . .	180
10.21 OpenSpeedShop . . . . .	180

<b>11 Virtualisation Haute Performance</b>	<b>182</b>
11.1 Surveillance de système et réseau . . . . .	183
11.2 Matériel . . . . .	185
11.3 Grappes de Calculs . . . . .	185
11.3.1 Infonuagies . . . . .	186
11.3.2 Red Hat MRG . . . . .	186
11.3.3 AMPQ . . . . .	186
11.3.4 Grilles CONDOR . . . . .	186
11.4 Virtualisation . . . . .	188
11.4.1 conteneurs . . . . .	188
11.4.2 Simulateurs d'exécution . . . . .	189
11.4.3 Intel VT, AMD V (virtualisation matériel) . . . . .	189
11.4.4 Paravirtualisation . . . . .	190
11.4.5 Hyperviseur . . . . .	190
11.4.6 Bénéfice de la virtualisation . . . . .	191
11.4.7 Cout de la virtualisation . . . . .	191
11.4.8 Virtualisation du réseau . . . . .	192
11.4.9 Migration . . . . .	192
11.4.10 Calcul parallèle et virtualisation . . . . .	193
11.5 Amazon EC2 . . . . .	194
11.5.1 Stockage . . . . .	194
11.5.2 Utilisation . . . . .	194
11.5.3 Enchère de calculs et adresse . . . . .	195
11.5.4 zones géographiques . . . . .	195
11.5.5 Instance de grappes . . . . .	195
11.5.6 répartiteur de charge . . . . .	195
11.5.7 nuage élastique . . . . .	195
11.5.8 surveillance CloudWatch . . . . .	196
11.5.9 nuage privé. LOL . . . . .	196
11.5.10 Services de bases de données . . . . .	196
11.5.11 Discussion . . . . .	196
11.6 OpenStack . . . . .	196
11.6.1 Nova . . . . .	198
11.6.2 Neutron . . . . .	198
11.6.3 Swift . . . . .	198
11.6.4 Cinder . . . . .	198
11.6.5 Keystone . . . . .	198
11.6.6 Glance . . . . .	199
11.6.7 Horizon . . . . .	199
11.6.8 Ceilometer . . . . .	199
11.6.9 Heat . . . . .	199
11.6.10 Trove . . . . .	200
11.6.11 Marconi . . . . .	200
11.6.12 Savannah . . . . .	200
11.6.13 Discussion . . . . .	200

<b>12 Top 500 + Architecture</b>	<b>201</b>
12.1 Discussion . . . . .	202
<b>13 Exemple d'examen finaux</b>	<b>203</b>
13.1 Examen 2016 . . . . .	203
13.1.1 Question 1a . . . . .	203
13.1.2 Question 1b . . . . .	203
13.1.3 1c . . . . .	203
13.1.4 2a . . . . .	204
13.1.5 2b . . . . .	204
13.1.6 2c . . . . .	204
13.1.7 3a . . . . .	204
13.1.8 3b . . . . .	204
13.1.9 3c . . . . .	204
13.1.10 4a . . . . .	204
13.2 Années 2014 . . . . .	214
13.3 Année 2013 . . . . .	224
13.4 Année 2012 . . . . .	235

# Introduction

- 1940 - Lampes
- 1960 - Transistors
- 1970 - Circuits intégrés
- 1980 - Circuits LSI-VLSI
- 2000 - Ordinateurs de 5-ème génération. parallèle et intelligents

Mémoire disque avait les mêmes améliorations, sauf que c'est 50% au lieu de 60.





**Densité des processeurs**

- Intel 4004, 1971, 10000nm, .74MHz, 2300 transistors
- Intel 8086, 1978, 3000nm, 8MHz, 29000 transistors
- Intel 80386DX, 1985, 1000nm, 33MHz, 275000 transistors
- Pentium Pro, 1995, 600nm, 150MHz, 5.5M transistors
- Pentium 4, 2000, 180nm, 1.7GHz, 42M transistors
- Intel Core 2, 2006, 65nm, 3.0GHz, 291M transistors
- Xeon Westmere, 2012, 32nm, 2600M transistors
- 15 cores Xeon Ivy Bridge EX, 2014, 22nm, 4310M transistors
- 22 cores Xeon Broadwell, 2016, 14nm, 7200M transistors

**Périphériques**

- Interface réseau: 1Gbit/s, 12\$;
- Disque: 6TB, 750MB/s, \$150;
- Mémoire non volatile: 1TB, 750MB/s, \$300;
- Rubans: LTO-5, 3TB, \$2000;
- Carte graphique: GP100 Pascal, 16nm, 15300M transistors, 3584 coeurs;

**Répertoire d'instructions**

- IBM 360
- DEC VAX 11
- Motorola 68000
- Intel 80386
- Sun SPARC
- IBM PowerPC
- ARM
- Intel Itanium
- Intel 64 (AMD64)

**Classes d'ordinateurs** On peut aussi classer les ordinateurs en différentes catégories, en voici un exemple.

- Super-ordinateur, entre 5 et 20M\$, grande pièce dédiée;
- Serveur d'entreprise (mainframe), entre 1 et 5 M\$, coin de pièce;
- Serveur départemental, occupe un coin de pièce, entre 50K\$ et 1 M\$;
- Poste de travail (workstation), dessus de bureau, 4K\$ à 50K\$;
- Micro-ordinateur, dessus de bureau ou creux de la main, 100 à 4K\$;

À présent, la mode est dans le nuage et les grappes d'ordinateurs. c'est flexible, efficace avec virtualisation. Ça l'a un bon rapport performance prix avec une bonne dimension sur la consommation d'énergie. C'est un grand système parallèle en réseau.

Depuis quelques années, on a plafonné en terme d'amélioration de vitesse par processeurs, alors on cherche à avoir quand même des gains de performances, mais au lieu d'améliorer la vitesse on parallélise notre travail sur plusieurs coeur différents.

Cela fait en sorte que la mémoire centrale et les disque n'augmentent pas de vitesse autant que le processeur.

On peut maintenant jouer aussi à des jeux 3D et du vidéo HD. On voit aussi la complexité des systèmes augmenter de manière exponentielle, ce qui à faire naître une panoplie d'outils de monitoring et vulnérabilités.

### Historique des Systèmes parallèles

- Burroughs D825, 1962, 4 processeurs;
- Honeywell Multics System, 1969, 8 processors;
- ILLIAC IV, 1965-1976, Vector machine, 256 units;
- Cray 1, 1976, Vector machine;
- Cray 2, 1985, 8 processors;
- Multi-processeurs (SGI, SUN, IBM);
- Grappes de calcul (Intel, IBM, HP...);

**Programmation des systèmes parallèles.**

- FORTRAN, compilateurs parallélisants, HPF;
- C, Parallel C, UPC, OpenMP (C et FORTRAN);
- PVM, Linda, MPI;
- CUDA, OpenCL

**Hiérarchie de mémoire** La mémoire dans les ordinateurs peuvent être organiser dans une forme de hiérarchie. Typiquement, les mémoires avec beaucoup de places sont beaucoup plus lentes. Versus les mémoires rapide ayant beaucoup moins de place. Dans cette hiérarchie, on considère les mémoires plus rapide en haut allant aux plus lentes (en bas).

- Registres
- Cache L1
- Cache L2
- Cache L3
- Cache  $L_n$
- Mémoire vive NUMA
- \*disque\* SSD
- disque magnétique

**Adresse** Normalement, on dépose une adresse en numéro de bloc et en adresse dans ce bloc. Le taux de succès est proportionnel au nombre d'accès complétés au niveau le plus haut sans interaction avec le niveau inférieur. ?

On peut définir:

$$T_{accesmoyen} = T_{accessucces} + taux_{echec} * penalite_{echec} \quad (1.1)$$

**Mémoire Cache .**

- Blocs de 4 à 128 octets
- Temps d'accès succès 1-4 cycles;
- Pénalité d'échec 8-32 cycles (accès 6-10), (transfert 2-22);
- Taux d'échec 1 à 20 pourcent;

- Dimension 1K à 512K (Intel Core I5, 32K I et 32K D);
- Correspondance directe, associative, par ensemble;
- Remplacement aléatoire ou LRU;
- Ecriture simultanée ou réécriture;

### Mémoire Centrale

- Pages de 4K, 2M ou 4M octets;
- Temps d'accès succès 10-100 cycles;
- Coût de l'échec 500000-6000000 cycles (quelques dizaines de milisecondes);
- Taux d'échec .00001%-.001%;
- Table de pages à 1, 2 ou 3 niveaux, (associatif, remplacement LRU);
- Conversion adresse logique à adresse physique par matériel avec cache (TLB);
- Chargement des pages par le système d'exploitation;
- Une portion du système d'exploitation est toujours en mémoire centrale.

### Système d'exploitation

- Interface avec les périphériques (pilotes);
- Interface avec les processus (appels systèmes);
- Gestion de la mémoire (mémoire des processus, mémoire virtuelle, tampons E/S);
- Systèmes de fichiers;
- Contrôle des accès;
- Ordonnancement;
- Gestion des interruptions.

## 1.1 Taxonomie de Flynn

**SISD** ordinateur conventionnel

**SIMD** ordinateur qui peut faire des opérations sur plusieurs donné. Parfois de manière conditionnel (ordinateur vectoriel, GPGPU)

**MIMD** multi-core, grappes.. etc.

## 1.2 Parallélisme

- La même opération sur des données différentes
- Le même programme sur des données différentes
- Différents programmes sur les mêmes données ou des données différentes

## 1.3 Associativité

La définition d'associativité est:

exemple:

Si on a un bloc de 512 avec une associativité de 4, ça veut dire qu'on a 4 blocs différents. alors 4 bloc de 128.

## 1.4 Fiabilité et Statistiques

on définit:

**MTBF** Mean Time Between Failure. Temps moyens entre les pannes

**MTTF** Mean Time to Failure. Temps jusqu'à une panne

**MTTR** Mean Time to Repaire. Temps moyens pour réparer la panne

**Fiabilité: F**  $F = MTBF = MTTF + MTTR$

**Disponibilité : D**  $D = \frac{MTTF}{MTTF + MTTR}$

**Déterminer taux de pannes, Failure in Time**  $FIT = \frac{1}{MTTF}$  exprimer en # de pannes par milliard d'heures.

Les taux de pannes s'additionnent si la probabilité de panne ne dépend pas de l'âge (accident aléatoire plutôt que l'usure).

**Exemple** MTTF total pour 10 disques. (MTTF = 1Mh),

1 contrôleur, (MTTF = 0.5Mh),

1 alimentation, (MTTF = 0.2Mh),

$FIT = \frac{1}{\frac{10}{1Mh} + \frac{1}{0.5Mh} + \frac{1}{0.2Mh}} = 58823heures$

**Probabilité**

**Probabilité que  $P_1$  et  $P_2$  se produisent (conjonction)**  $P_1 * P_2$

**Probabilité que de cas disjoints se produisent**  $P_1 + P_2$

**Probabilité que m ordinateurs sur n soient fonctionnels** p est la probabilité qu'un ordi soit fonctionnel

$$\frac{n!}{m!(n-m)!} * p^m * (1-p)^{n-m}$$

ou on peut prendre l'inverse, soit = 1-probabilité de tous défectueux =  $1 - (1-p)^n$   
voir exemple des examens précédents pour plus d'explications.

## 1.5 Mesure de performances

On utilise généralement ces mesure pour évaluer la performance des systèmes en parallèle:

- Temps écoulé
- Temps d'Utilisation du CPU, Mémoire, ETC.
- Rendement, résultats/Temps pour ressources.
- Accélération  $A = \frac{P_{amelioration}}{P_{amelioration}}$

**Accélération Partielle, Loi d'Amdahl**  $A_t = \frac{1}{(1-f) + \frac{f}{A_p}}$

ou  $A_t$  est l'accélération partielle, f est la fraction parallélisable de l'élément et  $A_p$  est le nombre de coeur/le facteur d'accélération parfait.

**Facteur d'accélération,  $S(p)$**   $= \frac{T_{sequentiel}}{T_{avec.n.processeur}}$

**Efficacité  $E(p)$**   $= \frac{S(p)}{p}$  Si fraction parallélisable f accélérée de p,

$S(p) = \frac{1}{((1-f) + \frac{f}{p})}$   $S(p)$  tend vers  $\frac{1}{1-f}$  si p très grand.

## 1.6 Exercices

### 1.6.1 1.1

**Question** Une application parallèle prend 4% de son temps pour diviser le travail et 3% pour réconcilier les réponses. Quelle est l'accélération maximale obtenue sur 10, 50, 100 ou 1000 processeurs?

**Réponse** LOI D'AMDAHL

La partie sérielle prend  $.03 + .04 = .07$ . La fraction parallélisable est donc de  $1.0 - .07 = 0.93$ . Le temps d'exécution pour  $n$  processeurs est donc de  $.07 + 0.93 / n$ .

Pour  $n = 10, 50, 100, 1000$ , le temps (accélération) devient: .163 (6.1), .0886 (11.3), .0793 (12.6), .07093 (14.1).

### 1.6.2 1.2

**Question** L'architecture Intel 64 décompose son adresse 64 bits en 16 bits inutilisés,  $4 * 9$  bits pour 4 niveaux de table de pages et 12 bits d'adresse dans la page. La cache L1 fait 32KiO, a une associativité de 4 et contient des blocs de 64 octets. Montrez l'organisation de la hiérarchie de mémoire et décrivez comment se fait l'accès à une donnée dont l'adresse virtuelle est fournie.

**Réponse** L'adresse de 64 bits se décompose en: 16 inutilisés, 9, 9, 9, 9 (index 0, 1, 2, 3), 12 pour octet dans la page. Pour un accès, la cache de prétraduction d'adresse contient 128 entrées de 36 bits d'étiquette vs adresse physique. Dans une bonne proportion des cas on y trouve l'adresse physique correspondant à l'adresse logique cherchée.

Autrement, la table de page de niveau 0 est indexée avec le premier 9 bits, menant vers une page de la table de niveau 1, indexée avec le second champ de 9 bits... jusqu'au niveau 3. Ce dernier niveau contient l'adresse physique et des bits de statut (valide, accessible...).

Une fois l'adresse physique obtenue, il faut vérifier si elle se trouve en cache L1. La cache de donnée de 32KiO contient 512 blocs de 64 octets et donc 128 ensembles de 4 blocs. On retrouve donc pour l'adresse de 64 bits: 16 inutilisés, 35 étiquette, 7 ensemble, 6 octet du bloc. Pour chaque ensemble, on a 4 blocs avec leur étiquette correspondante dans la cache. Le numéro d'ensemble est pris dans l'adresse et l'étiquette vérifiée avec celle des 4 blocs disponibles dans l'ensemble pour voir si le bloc cherché est en cache. Autrement, il faut charger le bloc contenant l'adresse physique recherchée à partir de la mémoire centrale (ou d'une cache L2) vers la mémoire cache L1.

### 1.6.3 1.3

**Question** Un disque contient 4 plateaux de données avec des pistes de 2 à 4.4 cm du centre. Les pistes sont distantes de .05 um et la densité de bits le long de chaque piste est de 400000 bits par cm. Quelle est la capacité de ce disque? Son taux de transfert maximal s'il tourne à 7200RPM?

**Réponse** Le rayon moyen des pistes est de :

$$\frac{4.4+2}{2} = 3.2cm.$$

La piste moyenne contient:

$$2\pi * 3.2cm * 400000b/cm = 8042477bits/piste$$

L'étendue est de:

$$(4.4 - 2) = 2.4cm. \text{ Il y a donc } 2.4cm * 10000um/cm / .05um = 480000pistes.$$

Ceci donne un total de

$$8042477pistes/plateau * 480000bits/piste * 4plateaux/disque / 8bits/octet = 1.93TeraOctets/disque.$$

Le taux de transfert maximal est en piste 0, au rayon = 4.4cm. Nous avons

$$2\pi * 4.4cm * 400000bits/cm * 4plateaux / 8bits/octet = 5529203octets$$

dans ces pistes.

A 7200 RPM ou  $7200/60 = 120$  tours par seconde, cela donne 5529203 octets / tour \* 120 tours/s = 663 MO/s.

### 1.6.4 1.4

**Question** Votre programme effectue un appel à fgets pour lire des données du disque. Décrivez le traitement fait par la librairie et le système d'exploitation?

**Réponse** Premièrement, la librairie vérifie s'il reste des données dans le tampon de lecture pour ce pointeur de fichier. Autrement, un appel système read est effectué. Le système d'exploitation vérifie si le bloc désiré est dans ses tampons d'E/S et sinon envoie une requête au contrôleur de disque et met le processus en attente. Le contrôleur lit le bloc voulu, le copie en mémoire par DMA et envoie une interruption au système d'exploitation. Le système d'exploitation change le statut du processus à prêt et pourrait le relancer immédiatement.

### 1.6.5 1.5

**Question** On vous demande de mesurer la taille des blocs de cache L1 ainsi que la taille de la cache L1. Comment feriez-vous?

**Réponse** Un vecteur peut être accédé en boucle déroulée pour différentes tailles de vecteur. Lorsque la taille dépasse la grosseur de la cache L1 de donnée, le temps d'accès moyen augmente rapidement. Ensuite, on peut lire dans un



vecteur de taille supérieure à L1 des octets consécutifs, 1 sur 2, 1 sur 3... Le temps d'accès moyen augmente jusqu'à ce qu'on atteigne 1 sur taille du bloc.

### 1.6.6 1.6

**Question** Un additionneur 1 bit peut produire une somme sur 1 bit en une unité de temps avec 6 portes logiques. Un additionneur 32 bits peut produire une somme de 32 bits avec 320 portes logiques en 6 unités de temps. Calculez le rendement de chacun (travail / (matériel \* temps))?

**Réponse** Somme 32 bits en 32t avec 6 portes:  $1 / (32 * 6) = .0052$   
 Somme 32 bits en 6t avec 320 portes:  $1 / (6 * 320) = .00052$

### 1.6.7 1.7

**Question** Chaque processeur dans un circuit multi-cœur peut exécuter 2G Instructions/seconde avec un taux de succès en mémoire cache de 99.9%, pour des blocs de 64 octets. Quelle est la bande passante requise vers la mémoire pour chaque processeur pour les instructions? Pour 100 processeurs? Un système multi-cœur contient 100 de ces processeurs dans une grille 10x10 avec 3 canaux à 2GO/s partagés par colonne. Le contrôleur de mémoire centralisé peut supporter 50GO/s. Est-ce que cette organisation peut soutenir le volume d'accès mémoire requis pour les fautes de cache d'instructions des 100 processeurs?

**Réponse** 2G Instructions/seconde avec un taux de succès 99.9%, blocs de 64 octets, 100 processeurs, 3 canaux à 2GO/s partagés par colonne, contrôleur de mémoire 50GB/s. Ceci donne  $2G \text{ lectures/s} * (1-.999) * 64 \text{ octets} = 128MO/s$  ou 1.28GO/s par colonne et 12.8GO/s pour 100 processeurs.

### 1.6.8 1.8

**Question** Donnez un exemple de décomposition pour a) une même opération sur des données différentes, b) un même programme sur des données différentes, c) différents programmes sur les mêmes données ou des données différentes?

**Réponse** Nous pouvons avoir a) un calcul de seuil sur chaque pixel d'une image, b) un calcul de rendu d'image sur différentes scènes d'un film, et c) différents encodages vidéo sur un même film.

### 1.6.9 1.9

**Question** Votre serveur départemental alimente un laboratoire avec 24 ordinateurs (D=.99) pour 20 équipes. Le serveur est en fait une machine virtuelle qui peut rouler sur l'une ou l'autre de deux machines physiques (D=.999) redondantes, connectées chacune à deux SAN (D=.999) partagés en miroir. Chaque SAN est composé de 4 disques (D=.99) en RAID 5. Tout ceci est connecté par

un réseau ( $D=.99999$ ) Quelles sont les chances pour au moins une équipe de ne pouvoir travailler? Quel serait l'effet de la fiabilité du logiciel?

**Réponse** .

Server : avoir deux défectueux,  $0.001 * 0.001 = 1 * 10^{-6}$

avoir  $n/24$  serveur fonctionnels,

$$\frac{24!}{n!(24-n)!} * 0.99^n * 0.01^{24-n} \text{ voir formule}$$

$$\Sigma_{n=20}^{24} = 0.999996373$$

avoir  $n/4$  disque fonctionnels,

$$\Sigma_{n=3}^4 \frac{4!}{n!(4-n)!} * 0.99^n * 0.01^{4-n}$$

$$\text{SAN: deux défectueux, } (1 - (0.999 * 0.9994))^2 = 0.000002558$$

Réseau défectueux, 0.00001

$$\text{Prob que tout fonctionne } (1 - P_{\text{serveur2defectueux}}) * P_{\text{+de20ordi}} * (1 - P_{\text{SAN2defectueux}}) * (1 - P_{\text{reseau}}) = 0.999982815$$

### 1.6.10 1.10

**Question** Un problème de grande taille s'exécute en 24h sur un noeud et en 1h sur un ordinateur parallèle de 16 noeuds identiques au premier. Quelle est la fraction parallélisable du programme? Comment expliquez-vous de tels chiffres?

**Réponse** Le problème décomposé entre plus facilement en mémoire centrale ou en mémoire cache et le tout s'exécute plus efficacement, présentant une accélération plus importante que la réduction du problème. Non, la fraction parallélisable n'est pas négative :-).

## Chapter 2

# Pthread et TBB

### 2.1 Fils d'exécution

Le principe d'avoir des fils d'exécution est de pouvoir continuer le traitement avec un fil lorsqu'un fil est bloqué. Le processus conventionnel est d'avoir une mémoire séparée avec une mémoire partagée optionnelle.

Processeur Intel i5, cycle de 0.4ns, LMBench

Opération	Temps ns	Opération	Temps ns
syscall trivial	65	fork+exit	263000
read	170	fork+execve	269400
write	138	pthread_create	~19000
stat	778	changer de processus	~1600
fstat	205	changer de fil	~1400
open/close	1492	iadd	.2
select 10 fd	515	imul	.18
select 500 fd	7022	idiv	9.52
signal	1348	dadd	1.32
segfault	158	ddiv	12.05
délai de tuyau	22716		

Figure 2.1: Les coûts en termes de temps de processeurs pour certaines opérations de gestion de threads

### 2.2 HyperThread

La technologie hyperthread est en fait une grosse agace. Elle laisse croire qu'elle double la performance de ton processeur en ayant deux \*core\* par coeur physique de ton processeur. Cependant, c'est faux.

La technologie hyperthreading est en fait deux ensemble de registres pour un même processeur. Elle apparaît comme deux processeurs sauf que la performance

n'augmente pas d'un facteur 2, elle augment d'environ 20 à 30 %.

Lorsqu'un des processeur virtuel bloque sur un faute de cache ou quelque chose d'autre. l'autre coeur physique prend la place en un temps très mince et continuer d'exécuter. Cela fait en sorte que sa peut nuire au application parallèle et aux stratégies d'optimisation.

## 2.3 Fils d'exécution en mode usager

La création d'un fils d'exécution comporte les étape suivantes:

- Allouer de la mémoire sur la pile
- inséré une entrée dans la tables de fils d'exécution.
- créer un context qui pointe vers la pile

La fin d'un fil d'exécution :

- Retirer l'entré dans la table des fils d'exécution
- désallouer la pile et le contexte ou les conserver pour recyclage.

Yield: sauver le contexte courant (setjmp), changer pour le contexte d'une autre tâche (longjmp)

## 2.4 Processus régulier (PThread)

Le principe est que chaque processus peut se cloner a l'aide de l'appel système fork. en le clonant, on spécifie ce que le processus doit faire à l'aide d'une fonction qu'on rajoute dans sa pile. Le processus courant continue en même temps que le processus enfant et éventuellement attend le retour de l'enfant avec un waitpid. Le processus enfant démarre sa fonction à l'aide de la commande système exec en allouant des zones de mémoire partagée (si il a besoin de communiquer avec le processus parent). L'enfant à lui-même le contenu initial de mémoire virtuelle avec une copie des descripteurs de fichiers. l'enfant n'hérite pas des signaux, temporisateurs, verrous.. Il doit l'avoir à partir des données qui lui sont passer par son processus parent.

### 2.4.1 Commandes

**Création** pthread\_create(&tid,..);

**Le parent attend souvent après l'enfant:** pthread\_join(tid, ...);

l'enfant partage la mémoire et toutes les ressources avec le parent mais utilise une partie différente de leur mémoire pour sa pile (une pile dans une pile).

**avantages**

- Même espace mémoire et table de page pour les fils d'exécution d'un même processus = plus compact, création et changement de contexte plus rapides
- Plus de coeurs physique = beaucoup plus de performance.

**désavantages**

- Taille moins flexible pour les piles;
- Tout usage concurrent de données partagées doit être protégé par des verrous ou accédé avec des opérations atomiques.
- Réentrance: une interruption peut survenir n'importe quand, l'ordre des données n'est pas nécessairement le même.. gros shitshow.
- Concurrence: On peut avoir deux ou plusieurs thread qui veulent accéder aux mêmes données en même temps.

## 2.5 Pthread

Le principe de pthread, comme le principe générale de la programmation parallèle, est de séparer le travail en plusieurs parties et de se servir des différents coeur de notre ordinateur pour pouvoir accomplir ce travail.

De cette manière, le travail pourra quand même continuer si on a des erreurs de blocage d'E/S (fautes de caches). Pour ce, on devra simplifier le travail de manière asynchrone, avoir un gestionnaire qui alimente un bassin de travailleurs (un thread pool). On divisera le travail de différentes manière. On peut y aller de façon hiérarchique, avec une pipeline, des groupes de pairs, etc.

### 2.5.1 Fonctions

- **pthread\_create(thread,attr,start\_routine,arg)**

Cette fonction créer un thread et l'exécute, elle à comme argument:

thread : c'est le thread, on l'initialise au début du programme quand on fait notre thread pool

attr : l'attribut de cet thread, on peut changer toute sorte de chose comme la grosseur de la pile, certaines caractéristique qui y sont essentiels pour l'ordonnanceur.

start\_routine : la fonction que nous allons exécuter sur ce thread, souvent appeller *\*worker\** ou quelques chose de similaire

arg: les argument que nous allons passer à notre thread. C'est obliger d'être un pointeur à un void. Normalement, on initialise une forme de structure, on la sérialise avec le pointeur en la passant à la thread, et ensuite à l'intérieur de la fonction on applique un cast de la forme de la structure originale pour qu'elle reprenne ses champs et soit accessible.

- **pthread\_exit(status)**

Fonction qui sert à sortir de la thread, on se sert de la variable status pour envoyer un message à notre processus parent si nécessaire.

- **pthread\_cancel(thread)**

On ferme notre thread, on se sert de cette fonction si on à besoin de fermer nos thread sans vraiment avoir besoin de rien en retour.

- **pthread\_attr\_init(attr)**

Cette fonction sert à faire l'initialisation des attributs de nos threads, c'est essentiel à l'ordonnanceur, sauf que sa marche sans mettre de valeurs (les valeurs par défaut sont bonnes)

- **pthread\_join(threadid, status)**

Cette fonction sert à joindre nos threads enfant au thread parents, normalement, elle sont appelés par les thread parents et l'appel de cette fonction fait attendre la thread parent jusqu'à temps que la thread enfant finisse la fonction start\_routine ou que celle-ci fait appel à **pthread\_exit()**.

- **pthread\_detach(threadid)**

On utilise cette fonction si on veut défaire le lien entre un thread paren et un thread enfant, pour des cas très spécifiques.

- **pthread\_attr\_setstacksize(attr, stacksize)**

On utilise cette fonction pour spécifier une grandeur de notre piles.

- **pthread\_mutex\_init(mutex, attr)**

Cette fonction sert à initialiser un mutex.

- **pthread\_mutex\_destroy(mutex)**

Sert à détruire un mutex.

- **mutex\_attr\_..... init ou destroy(attr)**

Fonctions qui sert à initialiser ou à détruire des attribut de mutex.

- **pthread\_mutex\_lock...unlock..trylock(mutex)**

Fonctions qui sert à barrer un mutex, le débarrer, ou essayer de le débarrer. Faut faire attention à l'ordre de verrouillage sinon on peut sa ramasser dans un deadlock.

- **pthread\_cond\_init(condition, attr)**

Initialise une condition pour notre thread.

- **pthread\_cond\_destroy(conditon)**

Détruit..

- **pthread\_condattr\_init...destroy()**

...

- **pthread\_cond\_wait(condition, mutex)**

Attend jusqu'à une certaines conditions pour un mutex ??

- **pthread\_cond\_broadcast(condition)**

..

### 2.5.2 Exemples Pthreads

```
// exemple_pthreads.c
#define REENTRANT
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
void afficher(int n, char lettre){
    int i,j;
    for (j=1; j<n; j++){
```

```

        for (i=1; i < 100000000; i++);
        printf("%c",lettre);
        fflush(stdout);
    }
}

void *threadA(void *inutilise){
    afficher(100, 'A');
    printf("\nFin du thread A\n");
    fflush(stdout);
    pthread_exit(NULL);
}

void *threadC(void *inutilise){
    afficher(150, 'C');
    printf("\nFin du thread C\n");
    fflush(stdout);
    pthread_exit(NULL);
}

void *threadB(void *inutilise){
    pthread_t thC;
    pthread_create(&thC, NULL, threadC, NULL);
    afficher(100, 'B');
    printf("\nLe thread B attend la fin du thread C\n");
    pthread_join(thC, NULL);
    printf("\nFin du thread B\n");
    fflush(stdout);
    pthread_exit(NULL);
}

int main(){
    int i;
    pthread_t thA, thB;
    printf("Creation du thread A");
    pthread_create(&thA, NULL, threadA, NULL);
    pthread_create(&thB, NULL, threadB, NULL);
    sleep(1);
    //attendre la fin des threads
    printf("Le thread principal attend que les autres se terminent\n");
    pthread_join(thA, NULL);
    pthread_join(thB, NULL);
    exit(0);
}

```

Cette exemple donne comme sortie :



**Exemple du TP1** Voici un autre exemple de pThread. Cet exemple est tiré du TP1.

```
#include "dragon.h"
#include "color.h"
#include <sys/types.h>
#include <sys/syscall.h>
#include <unistd.h>
#include "dragon_pthread.h"
int gettid(void);
pthread_mutex_t mutex_stdout;
```

```
void *dragon_draw_worker(void *data){  
    /* on initialise nos variable , code_erreur est evident ,  
       tdat sont nos donnees pour chaque  
           threads , *canvas sont now points de depart pour  
               l'initialisation de la surface . On doit  
                   utiliser des differents points de depart vu que le  
                       range n'est pas le meme que notre start et  
                           end habituel */  
  
    int code_erreur;
```

```

struct draw_data* tdat = (struct draw_data *) data;
int startcanvas =
(tdat->id)*((tdat->dragon_width * tdat->dragon_height)/tdat->nb_thread);
int endcanvas =
(tdat->id + 1) * ((tdat->dragon_width*tdat->dragon_height)/tdat->nb_thread);
int start = (tdat->id)*(tdat->size/tdat->nb_thread);
int end = (tdat->id + 1) * (tdat->size / tdat->nb_thread);

//cette ligne sert a debugger les
// points de depart dans les differentes threads.
//elle indique les points de depart de chaque threads dans le terminal

//printf("start is %d, end is %d, i is %d\n",start, end, tdat->id);

/* 1. Initialiser la surface */
//on initialise avec nos variables dependamment du thread ou l'on se situe

init_canvas(startcanvas, endcanvas, tdat->dragon, -1);
/* 2. Dessiner le dragon */
//on dessine avec nos variables

//printf("nous sommes rendu au wait de la thread %d\n", tdat->id);
code_erreur = pthread_barrier_wait(tdat->barrier);
//printf("nous avons passer la barriere %d\n", tdat->id);

if(code_erreur != -1 && 0){
//error handling de la barriere, cependant selon les man
// pages de barrier_wait, ne devrait jamais se produire
printf("erreur dans la barriere %d, code %d\n",tdat->id, code_erreur);
}

dragon_draw_raw(start, end, tdat->dragon, tdat->dragon_width,
tdat->dragon_height, tdat->limits, tdat->id);

/* 3. Effectuer le rendu final */

/**/

return NULL;
}

int dragon_draw_pthread(char **canvas, struct rgb *image,

```

```

int width, int height, uint64_t size, int nb_thread)
{
    //TODO("dragon_draw_pthread");

    pthread_t *threads = NULL;
    pthread_barrier_t barrier;
    pthread_attr_t Attr;
    limits_t lim;
    struct draw_data info;
    char *dragon = NULL;
    int i;
    int scale_x;
    int scale_y;
    struct draw_data *data = NULL;
    struct palette *palette = NULL;
    int ret = 0;
    void * status;
    int code_erreur;

    pthread_attr_init(&Attr);

    threads = malloc(sizeof(pthread_t)* nb_thread);
    pthread_attr_setstacksize(&Attr, stacksize);
    pthread_attr_setdetachstate(&Attr, PTHREAD_CREATE_JOINABLE);

    data = malloc((sizeof(struct draw_data)) * (nb_thread+1));

    palette = init_palette(nb_thread);
    if (palette == NULL)
        goto err;

    /* 1. Initialiser barrier. */
    pthread_barrier_init(&barrier, NULL, nb_thread +1);

    if (dragon_limits_pthread(&lim, size, nb_thread) < 0)
        goto err;

    info.dragon_width = lim.maximums.x - lim.minimums.x;
    info.dragon_height = lim.maximums.y - lim.minimums.y;

    if ((dragon = (char *) malloc(info.dragon_width * info.dragon_height)) =
        printf("malloc_error_dragon\n"));
        goto err;
}

```

```

    if ((data = malloc(sizeof(struct draw_data) * nb_thread)) == NULL) {
        printf("malloc_error_data\n");
        goto err;
    }

    if ((threads = malloc(sizeof(pthread_t) * nb_thread)) == NULL) {
        printf("malloc_error_threads\n");
        goto err;
    }

    info.image_height = height;
    info.image_width = width;
    scale_x = info.dragon_width / width + 1;
    scale_y = info.dragon_height / height + 1;
    info.scale = (scale_x > scale_y ? scale_x : scale_y);
    info.deltaJ = (info.scale * width - info.dragon_width) / 2;
    info.deltaI = (info.scale * height - info.dragon_height) / 2;
    info.nb_thread = nb_thread;
    info.dragon = dragon;
    info.image = image;
    info.size = size;
    info.limits = lim;
    info.barrier = &barrier;
    info.palette = palette;
    info.dragon = dragon;
    info.image = image;

    for(i = 0; i < nb_thread; i++){
        data[i] = info;
        data[i].id = i;
    }

    \\ 2. Lancement du calcul parallele principal avec draw\_dragon_worker

    for(i = 0; i < nb_thread; i++){
        code_erreur = pthread_create(&threads[i], &Attr, dragon_draw_worker,
        (void *) &data[i]);
        if(code_erreur){
            printf("erreur_dans_la_creation_de_la_thread_%d_dans_draw_dragon
            .....code_%d\n", i, code_erreur);
            goto err;
        }
    }

    /* 3. Attendre la fin du traitement */

```

```

        //printf("nous sommes rendu au wait de la main\n");
        pthread_barrier_wait(&barrier);
        //printf("nous avons passer la barrier de la main\n");
        for(i = 0; i < nb_thread; i++){
            code_erreur = pthread_join(threads[i], &status);
            if(code_erreur){
                printf("erreur dans le join de la thread %d dans draw_dragon_pth
                ..... erreur code %d\n", i, code_erreur);
                goto err;
            }
        }
        scale_dragon(0, height, image, width, height, dragon, info.dragon_width,
        info.dragon_height, palette);
        /* 4. Destruction des variables (a completer). */

        /* la plupart des variable a detruire sont tous dans "done"
        , la seule exception etant status et info.
        info genere une erreur de segmentation si on la detruit alors
        j'assume qu'elle est utiliser ailleurs. */
        free(status);
        goto done;
done:
        FREE(data);
        FREE(threads);
        free_palette(palette);
        *canvas = dragon;
        return ret;

err:
        FREE(dragon);
        ret = -1;
        goto done;
    }

void *dragon_limit_worker(void *data)
{
    struct limit_data *lim = (struct limit_data *) data;

    //cette ligne imprime le TID de la thread
    printf("tid_of_thread is %d\n", (int) getpid());

    piece_limit(lim->start, lim->end, &lim->piece);
    return NULL;
}

/*

```

```

    * Calcule les limites en terme de largeur et de hauteur de
    * la forme du dragon. Requis pour allouer la matrice de dessin.
    */
int dragon_limits_pthread(limits_t *limits, uint64_t size, int nb_thread)
{
    void * status;
    int ret = 0;
    int i;
    pthread_t *threads = NULL;
    pthread_attr_t ptAttr;
    pthread_attr_init(&ptAttr);
    struct limit_data *thread_data;
    piece_t master;
    int code_erreur;
    pthread_attr_setdetachstate(&ptAttr, PTHREAD_CREATE_JOINABLE);

    piece_init(&master);

    /* 1. Allouer de l'espace pour threads et threads_data. */

    //on associe la valeur defini de stack a nos pthreads et on fait un malloc
    // avec le nombre de limit_data par nombre de threads

    pthread_attr_setstacksize(&ptAttr, stacksize);
    threads = malloc(sizeof(pthread_t)*(nb_thread));
    thread_data = malloc((sizeof(struct limit_data)) * (nb_thread));

    /* 2. Lancement du calcul en parallele avec dragon_limit_worker. */

    /*initialisation des donnees pour les
    differentes thread, on divise les valeurs entre
    *start* et *end* par le nombre de threads que nous avons.
    start _____
    Thread1 ———> Thread 2 ———> Thread 3 ———> ..... —————*/

    for(i = 0; i < nb_thread ; i++){
        thread_data[i].id = i;
        thread_data[i].start = (int)(i)*(size/nb_thread);
        thread_data[i].end = (int)(i+1)*(size/nb_thread);
        thread_data[i].piece = master;

        //cette ligne sert a debugger les differentes threads, veuillez enlever
        // le comment pour voir les points de
        //depart de chaque thread dans le terminal

```

```

        printf("start_is %d, end_is %d, id_of_thread_is %d\n", (int)thread_data
            (int)thread_data[i].end, (int)thread_data[i].id);
    }
    // on creer nos threads avec dragon_limit_worker et nos differents point
    for(i = 0; i < nb_thread ; i++){
        code_erreur = pthread_create(&threads[i], &ptAttr, dragon_limit_worker,
            (void *) &thread_data[i]);
        if (code_erreur){
            printf("nous avons une erreur dans la thread %d", i);
            goto err;
        }
    }
    //on join les threads lorsqu'ils ont finient
    //le travaille et on merge les differents morceau.
    for(i = 0; i < nb_thread ; i++){
        code_erreur = pthread_join(threads[i], &status);
        if (code_erreur){
            printf("nous avons une erreur dans le join de la thread %d", i);
            goto err;
        }
        piece_merge(&master, thread_data[i].piece);
    }

    /* 3. Attendre la fin du traitement. */
    goto done;
done:
    FREE(threads);
    FREE(thread_data);
    *limits = master.limits;
    return ret;
err:
    ret = -1;
    goto done;
}

```

## 2.6 TBB

TBB est une librairie qui cherche à accomplir les mêmes chose que pthreads. Par contre, il y a des différences fondamentales.

La librairie travaille à un plus haut niveau, c-à-d. en langage c++. sa nous permet de décrire l'algorithme parallèle plutôt que d'être obligé de gérer manuellement tous les threads. C'est souvent plus efficace étant donné que c'est des experts qui ont fait la librairie..

C'est indépendant de la plate-forme et fonctionne seulement sur les processeurs intel. C'est basé sur des templates et utilise une différente taxonomie (task au lieu de threads).

Le fonction de la librairie, en générale, est qu'on doit faire un overload de l'opérateur () comme méthode dans notre class. ensuite on applique la fonction de tbb sur notre object et il va directement appeller les opérateurs () avec son range à lui.

On sépare le fonctionnement de la librairie en différente \*template\* de parallélisation, soit:

### 2.6.1 for

: parallel\_for

une boucle for quand même normale. Pour le fonctionnement, on définit un overload de notre opérateurs pour itérer à traver la fonction défini dans l'overload.

comme argument, il prend un interval 1d ou 2d et un objet avec un constructeur par copie et un opérateur () pour appliquer l'opération.

exemple :

```
parallel_for(blocked_range<size_t>(0,n), MyClosure(a));
```

ou MyClosure est mon objet.

l'object initial est pris, et il est copier n fois. ou n est le nombre de fois ou TBB juge comme nécessaire de faire. toutes les object sont appliquer sur n intervalles.

La décomposition se fait en arbre jusqu'à ce que l'intervalle soit assez petit. en voici un exemple.

```
struct TrivialIntegerRange {
    int lower;
    int upper;
    bool empty() const { return lower==upper; }
    bool is_divisible() const { return upper>lower+1; }
```



```

        TrivialIntegerRange( TrivialIntegerRange& r, split ) {
            int m = (r.lower+r.upper)/2;
            lower = m;
            upper = r.upper;
            r.upper = m;
        }
};

struct Average {
    const float* input;
    float* output;
    void operator()( const blocked_range<int>& range ) const {
        for( int i=range.begin(); i!=range.end(); ++i )
            output[i] = (input[i-1]+input[i]+input[i+1])*(1/3.f);
    }
};
// Entree [0..n] and Sortie [1..n-1].
void ParallelAverage( float* output, const float* input,
    size_t n
) {
    Average avg;
    avg.input = input;
    avg.output = output;
    parallel_for( blocked_range<int>( 1, n ), avg );
}

```

Exemple du TP1:

```

class DragonDraw
{
    public:
        DragonDraw(struct draw_data* parData)
        {
            data = parData;
        }
        DragonDraw(const DragonDraw& d)
        {
            data = d.data;
        }
        void operator()( const blocked_range<uint64_t>& r ) const
        {
            int indexStart = ((r.begin() * data->nb_thread)/data->size)
            int indexEnd = ((r.end() * data->nb_thread)/ data->size)
            if(indexStart != indexEnd)
            {
                int firstStart = r.begin();

```

```

        int firstEnd = (indexEnd * data->size / data->nb_threads);
        int secondEnd = r.end();
dragon_draw_raw(firstStart, firstEnd, data->dragon,
data->dragon_width, data->dragon_height, data->limits, indexStart);
dragon_draw_raw(firstEnd, secondEnd, data->dragon,
data->dragon_width, data->dragon_height, data->limits, indexEnd);
    }
    else
dragon_draw_raw(r.begin(), r.end(), data->dragon,
data->dragon_width, data->dragon_height, data->limits, indexStart);
    }
    struct draw_data* data;
};

int dragon_draw_tbb(char **canvas, struct rgb *image, int width, int height, uint32_t nb_threads)
{
    //TODO("dragon_draw_tbb");
    struct draw_data data;
    limits_t limits;
    char *dragon = NULL;
    int dragon_width;
    int dragon_height;
    int dragon_surface;
    int scale_x;
    int scale_y;
    int scale;
    int deltaJ;
    int deltaI;

    tid = new TidMap(nb_threads);

    struct palette *palette = init_palette(nb_threads);
    if (palette == NULL)
        return -1;

    task_scheduler_init init(nb_threads);

    /* 1. Calculer les limites du dragon */
    dragon_limits_tbb(&limits, size, nb_threads);
    dragon_width = limits.maximums.x - limits.minimums.x;
    dragon_height = limits.maximums.y - limits.minimums.y;
    dragon_surface = dragon_width * dragon_height;
    scale_x = dragon_width / width + 1;
    scale_y = dragon_height / height + 1;

```

```

scale = (scale_x > scale_y ? scale_x : scale_y);
deltaJ = (scale * width - dragon_width) / 2;
deltaI = (scale * height - dragon_height) / 2;

dragon = (char *) malloc(dragon_surface);
if (dragon == NULL) {
    free_palette(palette);
    *canvas = NULL;
    return -1;
}

data.nb_thread = nb_thread;
data.dragon = dragon;
data.image = image;
data.size = size;
data.image_height = height;
data.image_width = width;
data.dragon_width = dragon_width;
data.dragon_height = dragon_height;
data.limits = limits;
data.scale = scale;
data.deltaI = deltaI;
data.deltaJ = deltaJ;
data.palette = palette;

/* 2. Initialiser la surface : DragonClear */
DragonClear clear(-1, dragon);
parallel_for(blocked_range<int>(0, dragon_surface), clear);

/* 3. Dessiner le dragon : DragonDraw */
DragonDraw draw(&data);
parallel_for(blocked_range<uint64_t>(0, data.size), draw);

/* 4. Effectuer le rendu final */
DragonRender render(&data);
parallel_for(blocked_range<int>(0, height), render);
cout << "Thread_TBB: ";
tid->dump();
cout << "Nombre d'intervalles_TBB: " << nIntervalle << endl;
delete tid;

init.terminate();

free_palette(palette);
*canvas = dragon;

```

```

        return 0;
    }

```

### 2.6.2 réduction

: `parallel_reduce`

On applique une opération de réduction sur notre structure.

Elle doit avoir comme argument un intervalle, un objet avec un constructeur par division, un overload de l'opérateur `()` pour appliquer la méthode de réduction (`join`).

exemple:

```
parallel_reduce( blocked_range<size_t>(0,n), MySum(a) );
```

La décomposition est récursive, c-à-d. jusqu'à temps que l'intervalle soit assez petit.

À chaque feuille de l'arbre l'opérateur `()` est appliqué sur un range de notre grand range global. La recombinaison (`join`) va ensuite cumuler les résultats obtenus.

```

struct Sum {
    float value;
    Sum() : value(0) {}
    Sum( Sum& s, split ) { value = 0; }
    void operator()( const blocked_range<float*>& r ) {
        float temp = value;
        for( float* a=r.begin(); a!=r.end(); ++a ) {
            temp += *a;
        }
        value = temp;
    }
    void join( Sum& rhs ) { value += rhs.value; }
};

float ParallelSum( float array[], size_t n ) {
    Sum total;
    parallel_reduce( blocked_range<float*>( array, array+n ),
        total );
    return total.value;
}

```

On a aussi l'exemple du TP1...

```

class DragonLimits {
public:

```

```

        DragonLimits ()
        {
            piece_init(&p);
        }
        DragonLimits (const DragonLimits& d, split)
        {
            piece_init(&p);
        }
        void operator() (const blocked_range<int>& r)
        {
            piece_limit(r.begin(), r.end(), &p);
        }
        void join (DragonLimits& j)
        {
            piece_merge(&p, j.p);
        }
        piece_t& getPiece ()
        {
            return p;
        }
        piece_t p;
};

int dragon_limits_tbb(limits_t *limits, uint64_t size, int nb_thread)
{
    //TODO("dragon_limits_tbb");
    DragonLimits limit;
    task_scheduler_init init(nb_thread);
    parallel_reduce(blocked_range<int>(0, size), limit);
    piece_t piece = limit.getPiece();
    *limits = piece.limits;
    init.terminate();
    return 0;
}

```

### 2.6.3 Scan

: parallel\_scan

Une autre méthode qui ressemble beaucoup au `parallel_reduce`, on a comme argument un intervalle, un objet avec constructeur par division, un opérateur () pour scan initial ou final afin de calculer la réduction. On utilise aussi un `reverse_join` pour propager la réduction et assign pour l'initialiser. Prendre note, le nom des méthodes est importants.

exemple:

```
parallel_scan( blocked_range<int>(0,n), MyScan(a, b));
```

Scan fait la même chose que reduce, sauf qu'il ne redonne pas juste la dernière valeur, mais l'ensemble. Pour un vecteur 0,1,2,3,4,5,6,7,8,9; reduce nous donnerait :

45.

tandis que scan:

0, 1, 3, 6, 10, 15, 21, 28, 36, 45.

C'est utile lorsque plusieurs processeurs sont disponible. En voici un exemple.

```
class Body {
    T sum; T* const y; const T* const x;
    Body( T y_[] , const T x_[] ) : sum(0), x(x_), y(y_) {}
    T get_sum() const {return sum;}
    template<typename Tag>
    void operator()( const blocked_range<int>& r, Tag ) {
        T temp = sum;
        for( int i=r.begin(); i<r.end(); ++i ) {
            temp = temp + x[i];
            if( Tag::is_final_scan() ) y[i] = temp;
        }
        sum = temp;
    }
    Body( Body& b, split ) : x(b.x), y(b.y), sum(0) {}
    void reverse_join( Body& a ) { sum = a.sum + sum;}
    void assign( Body& b ) {sum = b.sum;}
};

float DoParallelScan( T y[] , const T x[] , int n ) {
    Body body(y,x);
    parallel_scan( blocked_range<int>(0,n), body );
    return body.get_sum();
}
```

#### 2.6.4 Do

parallel.do

Le parallel do est une instance plus générale de parallélisation. On itère à travers une liste de tâches dans un *\*feeder\**. Notre objet aura besoin d'un constructeur et d'un opérateur () encore une fois.

exemple.

```
parallel_do( first , last , MyClosure );
```

En plus, c'est possible d'ajouter des éléments directement au feeder. Par contre, la liste est parcouru séquentiellement, sa limite la mise à l'échelle.

Exemple.

```
class Body {
    Body() {};
    typedef Cell* argument_type;
    void operator()( Cell* c, tbb::parallel_do_feeder<Cell*>& feeder )
    const {
        c->update();
        c->ref_count = ArityOfOp[c->op];
        for( size_t k=0; k<c->successor.size(); ++k ) {
            Cell* successor = c->successor[k];
            if( 0 == --(successor->ref_count) ) {
                feeder.add( successor );
            }
        }
    }
};

void ParallelPreorderTraversal( const std::vector<Cell*>& root_set ){
    tbb::parallel_do( root_set.begin(), root_set.end(), Body() );
}
```

### 2.6.5 Pipeline

parallel\_pipeline

Le principe de la pipeline c'est de pouvoir mettre une entrée, l'entrée va passer dans un élément de la pipeline, ensuite la sortie du premier élément de la pipeline va aller dans l'entrée de la deuxième élément. ainsi de suite. Utilisé normalement pour faire des filtres.

```
pipeline.add_filter( filter& f );
```

```
pipeline.run( size_t max_number_of_live_token );
```

Exemple.

```
float RootMeanSquare( float* first , float* last ) {
    float sum=0;
    parallel_pipeline(16, make_filter<void, float*>( filter::serial ,
    [&]( flow_control& fc )-> float*{
        if( first<last ) return first++;
        else {
            fc.stop();
        }
        return NULL;
    } );
}
```

```

    }
  }) &
  make_filter<float*,float>(filter::parallel,
    [] (float* p){return (*p)*(*p);}) &
  make_filter<float,void>(filter::serial,
    [&](float x) {sum+=x;})
);
return sqrt(sum);
}

```

### 2.6.6 Sort

parallel\_sort

Très simple, les deux arguments sont des itérateurs avec accès direct avec une fonction de comparaison et d'échange.

Exemple.

```

#include "tbb/parallel_sort.h"
#include <math.h>
using namespace tbb;
const int N = 100000;
float a[N];
float b[N];
void SortExample() {
    for( int i = 0; i < N; i++ ) {
        a[i] = sin((double)i);
        b[i] = cos((double)i);
    }
    parallel_sort(a, a + N);
    parallel_sort(b, b + N, std::greater<float>());
}

```

### 2.6.7 Tâches

On effectue des sections de données en parallèle. On met des task dans une class et TBB va travailler dessus jusqu'à temps qu'il y en ait plus.

Les algorithmes parallèles se divisent les problèmes en tâches qui sont ordonnancés entre les fils/processeur. Il est aussi possible de spécifier directement les tâches à faire exécuter efficacement en parallèle. Chaque fil exécute une tâche à la fois, sans préemption. Lorsqu'il a fini, il prend la prochaine tâche créée par le même fil jusqu'à temps qu'il y en ait plus.. Il est aussi possible d'assigner des priorités.

exemple.



```

#include "tbb/task_group.h"
using namespace tbb;
int Fib(int n) {
    if( n<2 ) {
        return n;
    } else {
        int x, y;
        task_group g;
        g.run([&]{x=Fib(n-1);}); // spawn a task
        g.run([&]{y=Fib(n-2);}); // spawn another task
        g.wait();
        return x+y;
    }
}

```

l'ordonnancement de tâches de manière asynchrone sur plusieurs fils brise la propagation des exceptions normalement fournis par le c++. Les exceptions TBB peuvent être propagés d'un fil à l'autre pour remonter au fil de la tâche parent qui a créé les autres tâches.

On peut faire une structure qui permet l'accès concurrent. Les queues permettent de séparer des tâches parallèles mais un pipeline explicite est souvent mieux optimisé.

```

concurrent_hash_map<Key,T,HashCompare>
concurrent_queue<T>
concurrent_vector<T>

```

On peut ajouter des mutex et autres..

## 2.7 Exercice

### 2.7.1 2.1

**Question** Comment pourriez-vous évaluer ou mesurer la taille requise par un programme pour sa pile?

**Réponse** Statiquement, on peut regarder l'arbre d'appel (non récursif, sans pointeur de fonction) et la taille de pile prise par chaque fonction. Dynamiquement, on peut lire le pointeur de pile de temps en temps. Il est aussi possible de mettre un quota sur la pile et de le diminuer jusqu'à ce qu'il y ait un problème

### 2.7.2 2.2

**Question** Vous avez un vieil ordinateur ayant appartenu à votre grand-père qui ne contient qu'un seul processeur et un seul disque. Vous voulez y effectuer la compilation d'un gros logiciel (e.g. noyau Linux). La compilation de

chaque fichier demande un mélange de calcul et d'accès disque. Combien de fils d'exécution devriez-vous demander à `parallel make`? Combien de fils d'exécution demanderiez-vous pour un serveur Web?

**Réponse** L'idée est d'utiliser le CPU en parallèle avec le disque. L'un ou l'autre sera le goulot d'étranglement et donc au mieux, avec deux fils, si la charge est égale CPU et disque, cela pourrait aller deux fois plus vite. Plus de fils ne feraient qu'encomber l'ordonnanceur et la mémoire cache. Pour un serveur Web, les attentes après le réseau et les clients sont importantes et il est donc intéressant d'avoir de nombreux fils, assez pour éviter qu'ils soient tous pris en attente, non disponibles pour servir des requêtes.

### 2.7.3 2.3

**Question** Deux fils d'exécution communiquent entre eux par un tuyau bi-directionnel. Le premier lit du disque et envoie au second une certaine quantité de donnée puis se met en attente de la réponse et l'écrit sur disque. Le second traite les données au fur et à mesure de leur réception et envoie le résultat aussi au fur et à mesure. Est-ce correct? Peut-on procéder autrement?

**Réponse** Le problème est que le premier ne lit rien avant la fin alors que le second écrit au fur et à mesure. Ceci causera problème à moins que le tampon associé au tuyau soit plus grand que la quantité de données à traiter. Le second bloquera rapidement en écriture et ne pourra plus lire, ce qui bloquera aussi le premier en écriture.

### 2.7.4 2.4

**Question** Un débogueur insère des points d'arrêt en remplaçant l'instruction cible par une instruction d'interruption (INT3). Une fois cet emplacement atteint, le programme est interrompu et le INT3 retiré. Pour poursuivre, le débogueur repart l'exécution en mode une instruction à la fois, remplace le INT3 et repart l'exécution en mode normal. On vous demande de changer le débogueur pour fonctionner en mode non-stop, multi-fils: un point d'arrêt peut ne cibler qu'un seul fil d'exécution. Comment feriez-vous?

**Réponse** La difficulté serait d'enlever et remettre le INT3 alors que d'autres fils continuent leur exécution et doivent ou non s'arrêter sur cette instruction. La solution est de recopier et d'exécuter l'instruction remplacée par le INT3 ailleurs (out of line) et de compenser pour sa position modifiée (e.g. saut relatif au compteur de programme).

## Chapter 3

# Cohérence Mémoire

### 3.1 Cohérence de Cache

Ce chapitre est beaucoup moins axés sur la programmation, et beaucoup plus sur l'architecture des ordinateurs.

Il y a beaucoup d'organisation possible pour représenté la manière dont un ordinateur voit sa mémoire. voici un exemple.

Un aspect important est la Cohérence des caches, ce qui est un des points principale de ce chapitre.

Plusieurs scénario peut être possible. un processeur P1 écrit n dans X, lit X et trouve n. Par contre, un processeur P2 écrit n dans X, P1 lit x et peut trouver n si assez de temps est écoulé. Par contre, on ne peut jamais savoir combien de temps sa peut prendre. Les écriture sont propagées de manière asynchrone pour des fins de perfomances.

Normalement, pour une réécriture, la valeur est écrite en cache puis ensuite copiée ultérieurement en mémoire centrale. Par contre, il averti que le bloc a été modifié pour empêcher toute autre modification simultanée. Par contre, dans le cas d'une écriture immédiate, la valeur est immédiatement écrite en mémoire centrale, alors elle est visible par tous.

- **Cohérence par mise à jour:** une copie de la nouvelle valeur est envoyée à chaque copie en cache.
- **Cohérence par invalidation:** avertir, lorsqu'un bloc est modifié, de détruire les copies maintenant invalides.
- **status d'un bloc en cache:** Invalide, partagé (pour lecture seulement), exclusivité (peut petre écrit et lu).

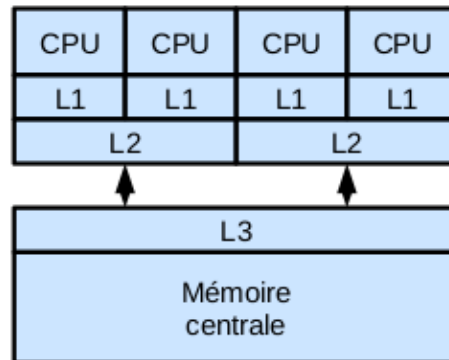


Figure 3.1: organisation de la mémoire cache dans un processeur multi-cœur

Normalement, on a deux mode de cohérence de cache. Soit

- **MOESI:**

- Modified: Cette cache a la copie la plus récente, la seule, la copie en mémoire centrale est invalide.
- Owned: cette cache a la copie la plus récente, d'autres peuvent avoir le status Shared, la copie en mémoire peut être invalide.
- Exclusive: cette cache a la copie la plus récente, la seule, la copie en mémoire est invalide (différence avec Modified)? peut-être que celle-ci peut seulement être modifiée par elle?
- Shared: Cette cache a la copie la plus récente, d'autres aussi, la copie en mémoire peut être invalide si une autre cache a la statut Owned.
- Invalid: ce bloc ne peut être utilisé, son contenu n'est plus à jour.

- **MESI**

- Modified: Cette cache a la copie la plus récente, la seule, la copie en mémoire centrale est invalide.
- Exclusive: cette cache a la copie la plus récente, la seule, la copie en mémoire est invalide (différence avec Modified)? peut-être que celle-ci peut seulement être modifiée par elle?
- Shared: Cette cache a la copie la plus récente, d'autres aussi, la copie en mémoire peut être invalide si une autre cache a la statut Owned.
- Invalid: ce bloc ne peut être utilisé, son contenu n'est plus à jour.

**Cohérence par surveillance du bus** Était très populaire lors de l'âge des processeur avec peu de coeur, souvent avec des écriture simultanée. Le message de mise à jour ou invalidation est envoyé sur le bus, les autre cache incorporent cette information si elle contiennent le bloc visé dans leur cache.

L'accès au bus sérialise les accès.

## 3.2 Cohérence par répertoire

Pour chaque bloc, liste des processeurs qui en ont une copie avec un status modifié.

Lorsqu'un bloc est modifié, s'il était en mode partagé, le répertoire est averti, un message d'invalidation est envoyé aux autres copies et le statut devient modifié.

Pour accéder un bloc modifié, il faut prendre la nouvelle copie et il devient partagé, non modifié. Le répertoire peut être réparti avec une fonction de correspondance directe.

**Surveillance versus répertoire** La surveillance du bus est plus simple et était utilisé pour deux ou 4 processeurs, par contre, les processeurs actuels saturerent trop rapidement le bus. La surveillance par Répertoire est plus compliqué mais requiert moins de communication.

**l'architecture NUMA avec répertoire réparti** Chaque processeur s'occupe d'une petite partie de la mémoire centrale et de son répertoire associé. voir fig 3.2.

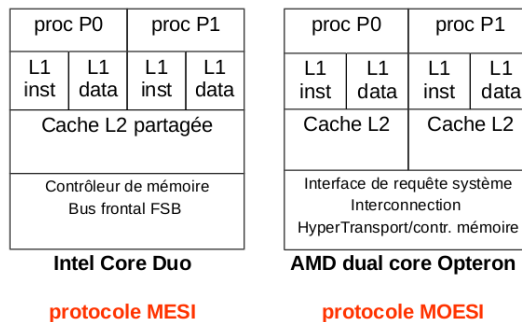


Figure 3.2: architecture NUMA sur les processeur intel et AMD.

Voici la configure de la génération AMD Bulldozer.  
et voici l'architecture d'intel.. Skylake Les i7 d'intels ont une TLB à chaque différent niveau de cache, en plus, ils ont une meilleur de faire des tabs qui leur permettent de garder leur memory mapping pour chaque différent processus.

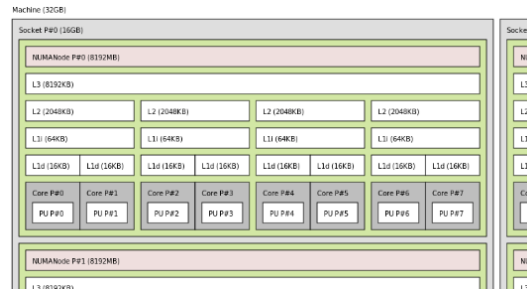


Figure 3.3: AMD Bulldozer

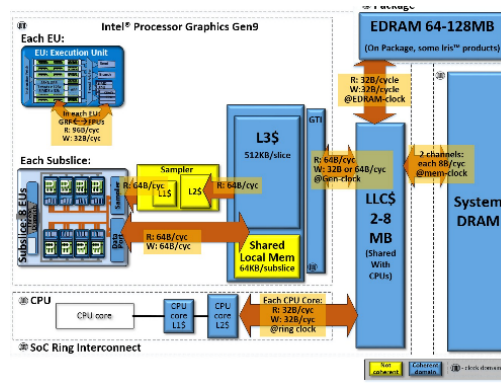


Figure 3.4: configuration de la famille Skylake d'Intel

	TLB instructions	TLB données	TLB niveau 2
Taille	128 entrées	64 entrées	512 entrées
Associativité	4	4	4
Remplacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Latence d'accès	1 cycle	1 cycle	6 cycles
Latence de faute	7 cycles	7 cycles	>> 100 cycles

Figure 3.5: TLB d'i7

### 3.3 interface logiciel cache et TLB

On a accès a un API qui nous permet de gérer le fonction de la TLB et de la cache, en voici des exemples.

#### TLB

- void flush\_tlb\_all(void), quand la table de page du noyau change car elle

	Cache L1	Cache L2	Cache L3
Taille	32Kio I / 32Kio D	256Kio	8Mio (pour 4 coeurs)
Associativité	4 I / 8 D	8	16
Remplacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Latence d'accès	4 cycles	10 cycles	35 cycles
Index	Index virtuel Etiquette physique	Index et étiquette physiques	Index et étiquettes physiques
Adresses	Adresses physiques 36 bits et adresses virtuelles 48 bits (# de page 36 bits et index dans la page 12 bits)		
Cohérence	Cache L3 inclusive avec répertoire de tous les blocs, leur statut et la liste des coeurs qui en ont une copie		

Figure 3.6: Information sur les différent niveau de cache des i7

est reprise par tous les processus.

- `void flush_tlb_mm(struct mm_struct *mm)`, quand tout l'espace d'un processus change, lors d'un fork ou exec.
- `void flush_tlb_range(struct vm_area_struct *vma, unsigned long start, unsigned long end)`, lorsqu'une partie d'espace d'un processus change, lors d'un munmap.
- `void flush_tlb_page(struct vm_area_struct *vma, unsigned long addr)`, lorsque l'entrée pour une page change, suite à une faute de page.
- `void update_mmu_cache(struct vm_area_struct *vma, unsigned long address, pte_t *ptep)`, indique qu'une nouvelle entrée est disponible pour une page virtuelle, par exemple pour précharger le TLB suite à une faute de page.
- 

### Cache

- `void flush_cache_mm(struct mm_struct *mm)`, lorsque tout l'espace d'un processus change, lors d'un exit ou exec.
- `void flush_cache_dup_mm(struct mm_struct *mm)`, lorsque tout l'espace d'un processus change, lors d'un fork.
- `void flush_cache_range(struct vm_area_struct *vma, unsigned long start, unsigned long end)`, lorsqu'une partie de l'espace d'un processus change, lors d'un munmap.
- `void flush_cache_page(struct vm_area_struct *vma, unsigned long addr, unsigned long pfn)`, lorsqu'une page change, lors d'une faute de page.
- `void flush_cache_kmaps(void)`, lors d'un changement à highmem.

- `void flush_cache_vmap(unsigned long start, unsigned long end)`, et `void flush_cache_vunmap(unsigned long start, unsigned long end)`, lorsqu'une partie de l'espace virtuel du noyau change, lors d'un `map_vm_area` ou `unmap_kernel_range`.
- `copy_user_page(...)`, `copy_to_user_page(...)`, `clear_user_page(...)`, `copy_from_user_page(...)`, faire une copie en tenant compte des problèmes d'alias (pour les dcache indexées virtuellement).
- `void flush_dcache_page(struct page *page)`, lorsque le noyau veut écrire ou lire une page qui peut aussi exister dans l'espace virtuel d'un processus et faire un alias.
- `void flush_anon_page(struct vm_area_struct *vma, struct page *page, unsigned long vmaddr)`, avant que le noyau accède une page anonyme qui pourrait être en alias.
- `void flush_kernel_dcache_page(struct page *page)`, après que le noyau ait modifié une page qui peut être en alias, pour enlever la copie de l'espace noyau en cache.
- `void flush_icache_range(unsigned long start, unsigned long end)`, lorsque du code exécutable vient d'être écrit par le noyau car les caches instructions ne s'attendent normalement pas à du code modifiable.
- `void flush_kernel_vmap_range(void *vaddr, int size)`, avant de faire une opération d'E/S sur une région en mémoire, pour propager toute modification en mémoire centrale et enlever de la cache en vue de faire un DMA.
- `void invalidate_kernel_vmap_range(void *vaddr, int size)`, enlever de la cache avant de faire une lecture d'E/S.

Les fautes de caches peuvent arriver pour plusieurs différentes raisons.

**Inévitable** La première accès cause toujours une faute de cache vue que le TLB n'est pas mappé sur le processus

**Conflit** Si deux blocs chauds sont dans le même ensemble.

**Capacité** Si le bloc requis a été évincé faute de place.

**Contention** Si il y beaucoup de demande pour une écriture d'une variable.

**Faux partage** Si jamais il y a deux variables qui sont modifiées par des processeurs différents et qui sont dans le même bloc de cache.



### 3.4 Optimisation

On peut optimiser en utilisant certaines modifications.

**Pipelines superscalaires** Pipeline permettant d'exécuter plusieurs instructions en même temps et dans le désordre, tout en respectant les dépendances. sa se passe au niveau du processeur lui-même.

ex: deux instruction de add sur deux différent registre dans le même coup d'horloge.

**Queues d'écritures**

**Queus de message d'invalidations**

**Bancs de mémoire cache qui peuvent être accédés en parallèle**

**Explication de la pipeline superscalaire** Le compilateur optimiseur peut effectuer tout changement qui ne modifie pas le résultat final. Par exemple, il peut réordonnancer les écriture et lectures (faites dans des variables séparés ne présentant pas de dépendance). Les deux section sont équivalentes.

ld R1, A	ld R1, A
ld R2, B	ld R3, C
ld R3, C	ld R2, B
... calcul ...	... calcul ...
st R4, Resultat1	st R5, Resultat2
st R5, Resultat2	st R4, Resultat1

Les écritures vers la mémoire cache peutventêtre mises en queue sans avoir d'attente synchrone.

Le contenu de la queue est consulté lors des accès en lecture pour assurer une cohérence entre cette écriture récente et une lecture qui suit sur le même CPU. Néanmoins, en mémoire centrale, les accès en écriture sont retardés ( $W \rightarrow R$ ). Il peut y avoir plusieurs queues, en fait, un par banc de mémoire cache. Dans ce cas, l'ordre des écriture peut changer entre deux accès si une queue est plus pleine qu'une autre ( $W \rightarrow W$ ).

### 3.5 Contraintes d'ordonnement

**Ordre séquentiel** : aucun réordonnement. Tout ce qui est dans l'ordre qu'il est écrit. Ne laisse pas place à beaucoup d'optimisation.

**Ordre total des écritures (TSO)** : Les écritures peuvent être retardées par rapport aux lectures en raison des queues d'écriture.

**Ordre partiel des écritures (PSO)** : Les écritures à des variables différentes peuvent être retardées à des degrés divers (queues multiples).

**Ordre faible (WO)** : les lectures peuvent être devancées (mises à jour retardées en queue d'invalidation)

### 3.6 Course entre deux processeurs

La cohérence entre le cache et le TLB est maintenue pour un processeur. Il doit n'y avoir qu'une seule copie maîtresse pour chaque bloc. Les mises à jour vers la copie maîtresse peuvent être retardées (queues d'écriture).

Les mises à jour vers les copies secondaires peuvent être retardées (queues d'invalidations).

Le code suivant espère que p (next, key, data) sera écrit avant qu'il ne devienne accessible (head.next = p):

```

p->next=head.next;
p->key=key;
p->data=data;
head.next = p; p = head.next;

while(p != &head){
  if(p->key==key){
    return(p);
  }
  p = p->next;
}
```

Le compilateur et le pipeline peut réordonner les accès non dépendants.

Les queues d'écriture peut retarder et réordonner les écritures.

La queue d'invalidation pourrait retarder la mise à jour de p et head (exemple d'en haut) même si head.next pointe déjà à p. (valeur non correct).

### 3.7 forcer un réordonnement

On peut forcer un réordonnement avec le API de linux. On utilise des instructions spéciales qui permet au compilateur de réordonner les accès (lecture, écriture ou les deux). On peut aussi le faire en mettant une barrière dans

le pipeline empêchant de réordonnancer les accès (lecture et écriture).

On peut aussi mettre une barrière dans les queues d'écriture pour retenir toutes les nouvelles écriture tant que celles avant la barrière ne sont pas terminées.

Finalement, on peut aussi mettre une barrière qui, lorsqu'une lecture arrive, bloque les accès le temps de traiter toutes les invalidations en queues.

### 3.7.1 Instructions

Noter que cmm n'est requis lorsqu'on est en mode utilisateur.

- `barrier()`: barrière pour le compilateur.
- `cmm_smp_rmb()` en mode usager ou `smp_rmb()` dans le noyau Linux: barrière lecture pour le compilateur et le pipeline, et pour les queues d'invalidation.
- `cmm_smp_read_barrier_depends()`: barrière pour les queues d'invalidation (les valeurs à synchroniser sont dépendantes et ne seront pas touchées par le compilateur et le pipeline).
- `cmm_smp_wmb()`: barrière d'écriture pour le compilateur et le pipeline, et pour les queues d'écriture.
- `cmm_smp_mb()`: barrière pour le compilateur, le pipeline, les queues d'écriture et les queues d'invalidation.

**Explications avec exemple** : Voir exemple en dessous

Une barrière `cmm_smp_wmb()` assure que `p` arrive en mémoire avant d'être accessible (`head.next=p`). Si `p` et `head` sont dans des blocs/bancs de cache différents, leur message d'invalidation peuvent être dans des queues différentes. Si la queue pour `p` est plus pleine, `head` peut arriver avant la mise à jour (invalidation) de `p` et l'ancienne valeur de `p` dans le cache du processeur 1 pourrait être lue, ce qui n'est pas bon du tout.

Une barrière `cmm_smp_rmb()` est trop conservatrice, `cmm_smp_read_barrier_depends()` est mieux.

```
/* Processeur 0 */
p->next = head.next;
p->key = key;
p->data = data;
cmm_smp_wmb();
head.next = p;

/* Processeur 1 */
p = head.next;
while (p != &head) {
    /* ajouter barrière ici! */
    if (p->key == key) {
        return (p);
    }
}
```

```

p = p->next;
}

```

## 3.8 Exercice

### 3.8.1 3.1

**Question** Un multi-processeur à mémoire partagée adressable par mot (de 4 octets) vient avec une mémoire cache associée à chacun des 4 processeurs. Cette mémoire cache est à correspondance directe et contient 4 blocs de deux mots. Montrez le statut de chaque bloc (MESI) après les accès suivants (processeur, P0 à P3, et accès, R ou W suivi de l'adresse et valeur) si les caches sont initialement vides. Discutez des communications dans les cas par mise à jour ou invalidation, et par répertoire ou surveillance de bus.

**Réponse** P0: R 120; P0: W 120, 80; P3: W 120, 80; P1: R 110; P0: W 108, 48; P0: W 130, 78; P3: W 130, 78

Cache directe, 4 blocs de 2 mots, MESI

Adresse 32 bits: étiquette 29 / bloc 2 / mot 1

Adresses décomposées: 120: 15/0/0, 110: 13/3/0, 108: 13/2/0, 130: 16/1/0

Les blocs suivants sont touchés par les accès dans chaque cache. Lorsqu'un bloc présent dans une autre cache (même étiquette) est accédé, il devient S. Si le bloc écrit se retrouve dans une autre cache, il est invalidé dans l'autre cache et modifié localement

P0: R 120, bloc 0 S; P0: W 120, 80, bloc 0 M; P3: W 120, 80, bloc 0 M, P0 bloc 0 I; P1: R 110, bloc 3 S; P0: W 108, 48, bloc 2 M; P0: W 130, 78, bloc 1 M; P3: W 130, 78, bloc 1 M, P0 bloc 1 I.

### 3.8.2 3.2

**Question** Un programme réinitialise un vecteur de 128 entiers de 4 octets à 0. Si la mémoire cache associée au processeur contient des blocs de 64 octets, décrivez les communications nécessaires sur le bus pour une cohérence par surveillance et mise à jour versus par répertoire et invalidation.

**Réponse** Un programme réinitialise un vecteur de 128 entiers de 4 octets à 0, cache avec blocs de 64 octets. Il y a donc 16 éléments entiers du vecteur dans chaque ligne de cache. Avec surveillance et mise à jour, un message est envoyé sur le bus pour chacun des 128 accès. Avec invalidation, une invalidation est envoyée à chaque 16 accès au répertoire; si aucune autre cache ne contient le bloc, cette invalidation n'est pas envoyée aux autres processeurs.

### 3.8.3 3.2

**Question** Deux variables mises à jour par des processeurs différents se retrouvent dans la même ligne de cache. Ces variables servent à calculer une somme dans une boucle et sont donc accédées une fois en lecture et une fois en écriture à chaque fois (4 instructions à 1 cycle/instruction hormis les fautes de cache de 16 cycles de pénalité). Comment peut-on se rendre compte d'un tel problème? Que peut-on faire pour y remédier? Quel facteur d'amélioration pourrait en résulter?

**Réponse** Accès une fois en lecture et une fois en écriture à chaque fois (4 instructions à 1 cycle/instruction hormis les fautes de cache de 16 cycles de pénalité). Le scénario probable est une faute de cache par tour de boucle pour  $4+16=20$  cycles versus 4 cycles. Un tel problème peut être détecté par des outils spécifiques ou avec un bon outil de profilage des fautes de cache. Il suffit d'aligner chaque variable sur une nouvelle ligne de cache pour y remédier.

### 3.8.4 3.4

**Question** Le processeur 0 exécute la fonction foo alors que le processeur 1 exécute la fonction bar sur un ordinateur multi-processeur à mémoire partagée avec cache, queues d'écriture et queues d'invalidation et le protocole de cohérence MOESI. Les variables a et b sont globales et initialement à 0. Est-ce qu'il pourrait arriver que l'assertion  $a==1$  ne soit pas vérifiée? Expliquez?

```
void foo(void){
    a = 1;
    smp_wmb();
    b = 1;
}
void bar(void){
    while (b == 0) continue;
    assert(a == 1);
}
```

**Réponse** La barrière mémoire en écriture assure que  $b=1$  arrivera après  $a=1$  dans la copie maîtresse de ce bloc de mémoire (en mémoire centrale ou en cache avec le statut modified). Il faut toutefois s'assurer que les mises à jour (invalidations) arrivent dans l'ordre sur le processeur 1, avec une barrière mémoire de lecture, `smp_rmb()`, entre les deux énoncés de la fonction bar.

## Chapter 4

# Modèle de mémoire partagée

La programmation à mémoire partagée est un nouveau point de vue. Le compilateur optimiseur génère du code assembleur et peut changer la séquence si sa n'affecte pas le résultat. l'unité centrale de traitement peut changer la séquence des accès aussi en raison de ses tampons d'écriture et de son pipeline.

l'ordre des changements en mémoire vus par les différent processeurs peut différer

Si deux fils d'exécution communiquent par mémoire partagée, il ne peuvent nécessairement se fier sur l'ordre.. Jusqu'à date, ce sont des chose qui ont été mis au claire au chapitre 3.

### 4.1 Modèle Partagé

**Séquentiel** Le processeur, lors d'exécution d'un modèle séquentiel, vois le travail dans l'ordre ou il est écrit. C'est à dire, toute le programme va être exécuter comme il est écrit.

Il attend qu'une écriture soit effective partout avant de continuer ou attend que les écriture de partout soit effective avant de continuer l'exécution du programme.

Cela fait en sorte que le modèle séquentiel est très contraignant, sa réduit donc beaucoup la performance. On peut relâchés différentes contraintes, ce qui nous fera utiliser des instructions de synchronisation pour s'assurer que le programme donne bien le résultats attendu.

Le modèle varie d'une architecture de processeur à un autre. Pour avoir le meilleur des deux mondes, nous devrions utiliser des variables en mémoire partagée et utiliser des primitives de synchronisation pour s'assurer que tout soit cohérent.

Par contre, l'accès mémoire ne sera réordonné avec une écriture que si les deux sont à des cases mémoire différentes. Les accès simples et alignés sont atomiques. Par contre, un accès double mot, ou non aligné pourrait être à moitié complété... Les opérations de synchronisation offertes par les bibliothèques ou le système d'exploitation s'occupent de mettre les instructions requises pour qu'on puisse séquencer les opérations correctement.

Les lectures peuvent avoir lieu avant une écriture à une variable différente qui les précéderait, contrainte  $W \rightarrow R$  relâché (Total Store Order).

Les lectures et écritures à des variables différentes peuvent être réordonnées par rapport aux écritures contenues  $W \rightarrow R$  et  $W \rightarrow W$  relâchés (Partial Store Order).

Tous les accès à des variables différentes peuvent être réordonnés (Weak Ordering).

Architecture	R->R	R->W	W->W	W->R	DR	Pipeline
Alpha	O	O	O	O	O	O
AMD64				O		
IA64	O	O	O	O		O
PA_RISC	O	O	O	O		
POWER	O	O	O	O	O	O
SPARC RMO	O	O	O	O	O	O
SPARC PSO			O	O	O	O
SPARC TSO				O	O	O
x86				O	O	O
x86 OOSTore	O	O	O	O	O	O
zSeries				O	O	O

Figure 4.1: Exemple des différents ordres de stockage

Voici des exemples:

Processeur 1

```
Flag1 = 1
if(Flag2 == 0) {
  *section critique*
}
```

ou

```
Processeur 1
Data = 2000
Head = 1
```

Processeur 2

```
Flag2 = 1
if(Flag1 == 0){
  *section critique*
}
```

ou

```
Processeur 2
while(Head == 0){;}
Value = Data
```

#### 4.1.1 Barrières

On utilise les même barrières que dans la cohérence de cache. Soit:

- Barrière mémoire pleine: `cmm_smp_mb()` en mode usager ou `smp_mb()` dans le noyau Linux. Lectures ou écritures qui précèdent effectuées avant toutes les lectures ou écritures qui suivent.
- Barrière mémoire pour lecture: `cmm_smp_rmb()`. Lectures qui précèdent effectuées avant les lectures qui suivent.
- Barrière mémoire pour écriture: `cmm_smp_wmb()`. Écritures qui précèdent effectuées avant les écriture qui suivent.
- Barrière mémoire pour lectures dépendantes: `cmm_smp_read_barrier_depends()`. Lectures qui précèdent effectuées avant les lectures dépendantes qui suivent.
- Barrière mémoire pour les E/S calquées sur la mémoire: `mmiowb()`. Correspond à un `nop` dans la plupart des cas car jumelé à un spinlock qui vient avec les barrières voulues.

**Exemples:** Processeur 1

```
Data = 2000
cmm_smp_wmb()
Head = 1
```

Processeur 2

```
while(Head == 0) {;}
cmm_smp_rmb()
Value = Data
```



<p>Processeur 1</p> <p>a = 1 b = 2 cmm_smp_wmb() c = 3 d = 4</p>	<p>Processeur 2</p> <p>v = c w = d cmm_smp_rmb() x = a y = b</p>
--	--

<p>Processeur 1</p> <p>Flag1 = 1 cmm_smp_mb()  if(Flag2 == 0) {      /* section critique */ }</p>	<p>Processeur 2</p> <p>Flag2 = 1 cmm_smp_mb()  if(Flag1 == 0) {      /* section critique */ }</p>
---	---

Figure 4.2: exemple 1 barrière mémoire

<p>{ M[0] == 1, M[1] == 2, M[3] = 3, P == 0, Q == 3 }</p>	
<p>Processeur 1</p> <p>M[1] = 4; cmm_smp_wmb() P = 1</p>	<p>Processeur 2</p> <p>Q = P; cmm_smp_read_barrier_depends() D = M[Q];</p>

Figure 4.3: exemple 2 barrière mémoire

**Architecture** On a aussi certaine commande unique par architecture.

- X86: smp\_wmb() est nop, smp\_rmb() et smp\_mb() sont réalisés avec lock,addl.
- AMD64: smp\_rmb() est lfence, smp\_wmb() est sfence, smp\_mb() est mfence.
- PowerPC: smp\_rmb() est lwsync, smp\_wmb() et smp\_mb() sont sync.
- SPARC: smp\_rmb() est membar #LoadLoad, smp\_wmb() est membar #StoreStore, smp\_mb() est membar

**Vérifications** Une barrière mémoire oublié ou un algorithme incomplète peut causer une course mémoire entre les différents processeurs. Il n'y a pas beaucoup de manière de déboguer ce genre de problème (Tracer pendant des jours??). Il faut vraiment modéliser l'algorithme et le matériel pour le valider formellement

en essayant toutes les possibilités (model checking).

On a aussi accès à des mutex, comme:

- spinlock
- mutex
- semaphores
- R/W semaphores
- RCU

On prend et on redonne les mutex avec:

**Lock** Les accès mémoire sont complétés après

**Unlock** les accès mémoire avant seront complétés avant

**Lock.. Unlock** Certaines opérations avant et après peuvent se mélanger à l'intérieur, ce n'est pas une barrière complète..

**Unlock.. Lock** barrière mémoire complète entre avant et après

## 4.2 résumé

Pour résumer la programmation en mémoire partagée, on suit les étapes suivantes:

1. Décomposer le travail en minimisant les interactions (écriture dans des variables partagés entre les fil d'exécutions)
2. Synchroniser les accès aux variables partagées.
3. Lecture sans verrous, écriture atomique.
4. Verrou associé à une ou des variables.
5. Structure mises à jour par RCU.
  - spinlock
  - R/W spinlock
  - mutex
  - semaphores
  - R/W semaphores

- verrous usuels du RCU, par contre c'est très mauvais avec un grand nombre de processeurs
6. Protection des variables
    - Processus avec signal, assurer néanmoins la réentrance pour les variables locales à un thread (CPU)
    - Section critique du noyau modifiant des variables et utilisant le CPUid: désactiver la préemption ou verrouiller.
    - Dans le noyau, désactiver les interruptions et désactiver la préemption si aucun appel n'est fait qui puisse causer un réordonnement.
  7. Attendre que plusieurs fil d'exécution aient atteint un certain point
  8. Join!

#### Méthode de gestion de fils

- Arbr: arbre de décomposition du problème, la racine attend après les enfants récursivement puis envoie le déblocage en sens inverse.
- Papillon: chaque fil communique avec un autre à chaque étape ( $i+1\%n$ ,  $i+2\%n$ ,  $i+4\%n...$ ) Tous reçoivent l'information que tous sont prêts en même temps, pas de phase d'annonce de fin! C'est le plus rapide, par contre il envoie beaucoup plus de message que les autres.

### 4.3 Exercices

#### 4.3.1 4.1

**Question** Expliquez par quel mécanismes possiblement présents sur des architectures modernes, les différentes contraintes W- $\ell$ R, W- $\ell$ W, R- $\ell$ R et R- $\ell$ W pourraient être relâchées afin de gagner en performance?

**Réponse** Les queues d'écriture vont souvent retarder les écritures par rapport aux lectures. Deux blocs de mémoire peuvent opérer en parallèle avec chacun leur queue d'écriture et queue d'invalidation, et causer des réordonnements écriture écriture ou toute autre combinaison.

#### 4.3.2 4.2

**Question** En quoi diffèrent les opérations de synchronisation comme les opérations atomiques sur un mono-processeur versus sur un multi-processeur? Lequel est plus coûteux?

**Réponse** Sur mono-processeur, il suffit de se prémunir contre la préemption, ce qui est en général beaucoup plus court. Sur multi-processeur, un mécanisme est requis pour assurer que la nouvelle valeur soit disponible et modifiée par le processeur qui exécute l'opération atomique, sans interaction possible avec les autres processeurs, un peu comme pour les barrières mémoire.

#### 4.3.3 4.3

**Question** On veut remplacer l'allocateur de mémoire d'un système d'exploitation multi- processeur par un allocateur qui maintient de la mémoire par processeur et la plupart du temps peut servir les allocations localement sur le processeur. Comment peut-on maintenir des structures "par processeur" en mémoire partagée? Comment changent les besoins en synchronisation dans ce cas?

**Réponse** On peut avoir des structures par processeur soit en utilisant le numéro de processeur comme indice dans un vecteur, ou en utilisant un registre. Il faut toutefois se prémunir contre la préemption ou la migration de processeur.

#### 4.3.4 4.4

**Question** Deux processeurs effectuent les opérations suivantes dans un système avec ordonnancement faible. Quels sont les combinaisons de valeurs possibles à la fin?

Processeur 1:

A = 3;

B=4

Processeur 2:

x = A

y = B

**Réponse** permutation de x=1,3 et y = 2,4

#### 4.3.5 4.5

**Question** Les accès et verrous suivants sont exécutés sur deux processeurs, que peut voir ou ne pas voir un troisième processeur?

Processeur 1  
 \*A = a;  
 LOCK M  
 \*B = b;  
 \*C = c;  
 UNLOCK M  
 \*D = d;

Processeur 2  
 \*E = e;  
 LOCK Q  
 \*F = f;  
 \*G = g;  
 UNLOCK Q  
 \*H = h;

**Réponse** Les LOCK empêchent les accès qui suivent de remonter avant. Par contre, \*A = a peut aller après le LOCK. Le UNLOCK empêche les accès qui précèdent de le dépasser mais \*D = d peut remonter avant. Le processeur 3 pourrait donc voir l'affectation de \*D avant celle de \*A.

#### 4.3.6 4.6

**Question** On vous demande de programmer un compteur d'octets, pour les statistiques de transmission par réseau, sur un système d'exploitation multi-processeur très performant où chaque processeur peut recevoir et envoyer des paquets. Cette valeur est mise à jour 1 million de fois par seconde mais lue environ une fois par 5 secondes. Comment vous y prendriez-vous?

**Réponse** On peut avoir un compteur par processeur et il suffit de sommer les compteurs à chaque 5 secondes pour générer les statistiques.

#### 4.3.7 4.7

**Question** Donnez des exemples de structures partagées dans un système d'exploitation multi-processeur?

**Réponse** La liste des tâches, la liste des pages utilisées comme tampon E/S, la liste des périphériques...

#### 4.3.8 4.8

**Question** Deux verrous propres à chaque inode, i.A et i.B, sont requis avant de modifier un inode i dans un système. On vous suggère pour une raison obscure de prendre A avant B pour les inode pairs et B avant A pour les inode impairs. Est-ce que cela peut fonctionner?

**Réponse** Cette manière de procéder n'est pas incorrecte mais confondra les outils de vérification comme LockDep ainsi que les autres programmeurs qui regarderont ce code.

**4.3.9 4.9**

**Question** Quel est le nombre total de messages requis pour un rendez-vous de  $n$  processus avec les méthodes du compteur, en arbre et en papillon? Quel est le délai si les messages peuvent circuler en parallèle sur le réseau mais une unité de temps  $t$  est requise pour l'envoi ou la réception d'un message sur un ordinateur donné?

**Réponse** L'envoi ou la réception de deux messages prend  $2t$ , mais il est possible de recevoir et envoyer un message (non reliés) simultanément en  $1t$ . 2Avec un compteur,  $2(n-1)$  messages sont envoyés mais le délai est de  $2(n-1)$  aussi. En arbre, chaque noeud envoie puis reçoit un message sauf la racine pour  $2(n-1)$  messages et un délai de  $4 \log n$ . En papillon, chaque noeud envoie un message à chaque étape,  $n \log n$ . Le délai est de  $\log n$ .

## Chapter 5

# OpenMP

OpenMP (Open Multi-Processing) est un API en C, C++ et Fortran. Il exist sur Linux, Unix, AIX, Solaris, MacOSX et Microsoft Windows. C'est un consortium sans but lucratif qui inclut AMD, IBM, Intel, Cray, HP, Fujitsu, NVidia, NEC, Microsoft, Texas Instruments, VMware, Oracle Corporation...

OpenMP pour Fortran débuta en 1997. 1998 pour la version C et C++. Version 2.0 en 2000 pour C et C++, Version 2.5 pour C/C++ et Fortran en 2005, version 3.0 en 2008, 3.1 en 2011, 4.0 avec SIMD en 2013, 4.5 avec taskloop et ajustements en 2015.

**caractéristique** OpenMP est un standard complet et portable, c'est disponible gratuitement et peut servir comme manuel de référence (170 pages +).

Le principe d'OpenMP est l'utilisation de pragmas pour déclarer des directive au compilateur. On décrit avec les pragmas les sections qu'on veut paralléliser et comment on veut les paralléliser. C'est ultimement axé sur la performance et sa parallélise à grain fin.

On compte parmi les alternatives...

- Compilateur parallélisant, pas très bon
- Intel Threading Building Blocks, software propriétaire, beurk

OpenMP permet de paralléliser un programme séquentielle graduellement avec une granularité très fine, et même avec la version 4.0, d'utiliser le GPU.

C'est possible de créer un grand nombre de fils d'exécution, plus grand ou égal au nombre de processus (omp parallel) et de diviser le travail entre les fils (avec les directive for, sections, tasks..

Tous les fils d'une région parallèle doivent passer par les même sections de division du travail, même s'ils peuvent être dans des fonctions appelées. Les synchronisation de contrôle découle en bonne partie des pragmas insérés (section parallèles, séquentielle ou barrières). On peut déclarer nos variables comme privées (copie local sur chaque fil) ou partagé.

La synchronisation des données partagées doit se faire par verrous.

**Syntaxes** Un pragma d'OpenMP doit être formulé de la façon suivante:

1. `#pragma omp ...` À tous les cas
2. Directives :
  - `parallel`: Définit notre section comme une section parallèle, obligatoire de définir la section parallèle avant de pouvoir appliquer des méthodes de répartition de travaux
  - `for`: On sépare notre travail sur  $n$  intervalles. On appelle cette fonction avant un `loop for` et `omp` va se charger de le répartir sur les threads.
  - `sections`: on définit une zone où l'on va définir différentes sections de travail. Section qui vont être exécuter en parallèle sur différentes threads.
  - `single` : définit une section qui va être utilisée que par une seule thread.
  - Plusieurs autres..
3. clauses:
  - `shared` : Définit les variables que nous voulons partagés sur les différentes threads.
  - `private` : Définit les variables que nous voulons copier sur chaque threads.
  - `first private`: même chose, sauf que les variables vont prendre la valeur qu'on avait au début de la section parallèle (avant le `pragma parallel`)
  - `last private`: Prend la valeur des variables la plus à jour à chaque fois.
  - `default`: Pas trop sûr..
  - `reduction`: On spécifie que notre boucle `for` fait une opération de réduction. On spécifie la variable que l'on veut utiliser pour la réduction ainsi que l'opération.
  - pleins d'autres..

**Exemples** `#pragma omp parallel for private(x,y) reduction(+:total)`



## 5.1 Directives

### 5.1.1 parallel

Parallel fait la création d'un groupe de fils d'exécution qui s'ajoutent au fil principale au début du bloc.

Le fil principale attend que tous les autres fils aient fini à la fin du bloc. Chaque fil d'exécution fait la même chose.. sa n'optimise rien du tout. On peut apporter des variantes en utilisant sont numéro de fil d'exécution (`omp_get_thread_num()`). Exemple:

```
#include <stdio.h>
#include <omp.h>
int main( int argc , char **argv ){
    int rank , size;
    #pragma omp parallel private(rank)
    {
        rank= omp_get_thread_num();
        size= omp_get_num_threads();
        printf( " Hello_world!_I'm_%d_of_%d\n" ,rank , size );
    }
    return 0;
}
dorsal> export OMP_NUM_THREADS=4
dorsal> gcc -fopenmp -o HelloWorld HelloWorld.c
dorsal> ./HelloWorld
Hello_world!_I'm_1_of_4
Hello_world!_I'm_3_of_4
Hello_world!_I'm_4_of_4
Hello_world!_I'm_2_of_4
```

### 5.1.2 for

Le for doit avoir une forme simple. Le domain de l'itérateur est divisé entre les fils pour tout couvrir. OpenMP utilise différentes stratégies de division du travail. On peut aussi lui en imposer une (statique, dynamique, guidé, auto..)

Chaque fil d'exécution s'occupe d'un intervalle de la boucle. Il est aussi possible d'avoir un combiné. Ex: `#pragma omp parallel for ....`

Exemple:

```
#pragma omp parallel for default(none) private(i,j,sum)
/shared(m,n,a,b,c)
for (i=0; i<m; i++) {
    sum = 0.0;
    for (j=0; j<n; j++) sum += b[i][j]*c[j];
}
```

```

        a[i] = sum;
    }

```

### 5.1.3 Sections

Sa divise ton code en sections, chacune de ses section est exécuter sur un fil d'exécution.

Exemple:

```

#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
        }
        #pragma omp section
        {
        }
    }
    ...
}

//ou bien..
#pragma omp parallel sections
{
    #pragma omp section
    {
        for (int i=0; i<n; i++) {
            input(i);
            signal_input(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<n; i++) {
            wait_input(i);
            process(i);
            signal_process(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<n; i++) {
            wait_process(i);
            output(i);
        }
    }
}

```

```

    }
  }
}

```

#### 5.1.4 Tasks

Très similaire aux sections, par contre, on attend pas que toutes les tâches soient terminées avant de continuer vers la suite. On fait juste rajouter la tâche sur le CPU et on continue le programme.

Exemple:

```

#pragma omp parallel
{
    #pragma omp task
    {
    }
    #pragma omp task
    {
    }
    ...
}

//ou bien

#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        {
            printf(" Hello\n");
        }
        #pragma omp task
        {
            printf(" World");
        }
        printf( ' Bye ');
    }
}

```

#### 5.1.5 single

Cette section n'est exécutée qu'une seule fois.

```
#pragma omp parallel
{
    #pragma omp single
    printf("Beginning _work1.\n");
    work1();
    #pragma omp single
    printf("Finishing _work1.\n");
    #pragma omp single nowait
    printf("Finished _work1_and_beginning _work2.\n");
    work2();
}
```

### 5.1.6 Master

Même chose que single, sauf qu'elle est exécuter par la thread parent de toute.

```
#pragma omp parallel
{
    do{
        #pragma omp for private(i)
        for( i = 1; i < n-1; ++i ){
            xold[i] = x[i];
        }
        #pragma omp single
        {
            toobig = 0;
        }
        #pragma omp for private(i,y,error) reduction(+:toobig)
        for( i = 1; i < n-1; ++i ){
            y = x[i];
            x[i] = average( xold[i-1], x[i], xold[i+1]);
            error = y - x[i];
            if(error > tol || error < -tol) ++toobig;
        }
        #pragma omp master
        {
            ++c;
            printf( "iteration %d, _toobig=%d\n", c, toobig );
        }
        while( toobig > 0 );
    }
}
```

### 5.1.7 Critical

Une section critique, seulement une thread peut exécuter cette section à la fois. C'est comme un verrous partiel.

```

#pragma omp parallel shared(x, y) private(ix_next, iy_next)
{
    #pragma omp critical (xaxis)
    {
        ix_next = dequeue(x);
    }
    work(ix_next, x);
    #pragma omp critical (yaxis)
    {
        iy_next = dequeue(y);
    }
    work(iy_next, y);
}r

```

### 5.1.8 barrier

Lorsque des données peuvent être partagés par des fils d'exécution différents et qu'il y a une dépendance, il faut mettre des barrière pour synchroniser les fils parallèles.

Il y a une barrière implicite à la fin de chaque région.

```

x = 2;
#pragma omp parallel num_threads(2) shared(x)
{
    if (omp_get_thread_num() == 0) {
        x = 5;
    } else {
        /* Print 1: the following read of x has a race */
        printf("1: Thread#%d: x=%d\n", omp_get_thread_num(), x);
    }
    #pragma omp barrier
    if (omp_get_thread_num() == 0) {
        printf("2: Thread#%d: x=%d\n", omp_get_thread_num(), x);
    } else {
        printf("3: Thread#%d: x=%d\n", omp_get_thread_num(), x);
    }
}

```

### 5.1.9 taskwait

Attend après toutes les tâches enfants de la tâche courante.

```

#pragma omp parallel
{
    #pragma omp single

```

```

    {
        #pragma omp task
        {
            printf("Hello\n");
        }
        #pragma omp task
        {
            printf("World\n");
        }
        #pragma omp taskwait
        printf('Bye\n');
    }
}

```

### 5.1.10 atomic

On demande une opération atomique sur la variable accédée.

```

for (i = 0; i < 10000; i++) {
    index[i] = i % 1000;
    y[i]=0.0;
}

for (i = 0; i < 1000; i++) {
    x[i] = 0.0;
}
#pragma omp parallel for shared(x, y, index, n)
for (i=0; i<n; i++) {
    #pragma omp atomic update
    x[index[i]] += work1(i);
    y[i] += work2(i);
}

```

### 5.1.11 flush

Une barrière mémoire complète. La vue temporaire de la mémoire par un fil est synchronisé avec la copie principale. Sa doit être utilisé dans les deux ou plusieurs fil d'exécution qui accèdent des données partagés en parallèle.

```

// Producteur
data = 2000;
#pragma omp flush(data);
flag = 1;
#pragma omp flush(flag);
// Consommateur
#pragma omp flush(flag)

```

```

if(flag == 1) {
    #pragma omp flush(data)
    x = data;
}

```

### 5.1.12 ordered

On exécute la section dans l'ordre d'itération de la boucle. Très contraignant et ne laisse pas beaucoup de place à l'optimisation.

```

#pragma omp parallel for ordered
{
    for(i = 0 ; i < N; i++){
        calcul complexe...
        #pragma omp ordered
        {
            printf 'a[%d]=%d', i, a[i]);
        }
    }
}

```

### 5.1.13 Synchronisations

- omp\_init\_lock
- omp\_destroy\_lock
- omp\_set\_lock
- omp\_unset\_lock
- omp\_test\_lock
- autres..

Pour chaque région parallèle ou division du travail, il faut définir la portée des variables et comment elle sont initialisés et finalisé. On utilise les clauses pour spécifier certaines de ces conditions.

## 5.2 Clauses

### 5.2.1 Shared et Private

Les clauses shared et private doit exister avant la région parallèle visé. l'effet est sur la portée statique et non dynamique de la directive. Il n'y a qu'une seule version des variables shared partagé sur tout nos threads et on a une variable private initialisé par thread. Normalement private va plus vite au cout d'avoir

besoin de plus de mémoire.

First private, les variables privées sont initialisées avec la valeur d'entrée tandis que Last private utilise la dernière valeur que la variable à eu après une itération de boucle. Copyprivate propage une variable d'une région single aux autres fils d'exécution.

### 5.2.2 Threadprivate

Une copie des variables listées est créée pour chaque fils d'exécution. La clause copyin permet d'effectuer de nouveau la copie plus tard. Ces variables copiées continuent d'exister d'une tâche à l'autre et possible d'une section parallèle à l'autre

### 5.2.3 Copyin

Voir Threadprivate

### 5.2.4 réduction

Opérateur de réduction, Variable utilisé pour une réduction avec un opérateur commutatif et associatif entre les fils d'une région parallèle. Voir tableau.

Valid Operators and Initialization Values			
Operation	Fortran	C/C++	Initialization
Addition	+	+	0
Multiplication	*	*	1
Subtraction	-	-	0
Logical AND	.and.	&&	0
Logical OR	.or.		.false. / 0
AND bitwise	iand	&	all bits on / 1
OR bitwise	ior		0
Exclusive OR bitwise	ieor	^	0
Equivalent	.eqv.		.true.
Not Equivalent	.neqv.		.false.
Maximum	max	max	Most negative #
Minimum	min	min	Largest positive #

Figure 5.1: opérateur de réduction

### 5.2.5 if

La région visée n'est effectuée en parallèle que si la condition est vraie. On se sert normalement de sa si la taille du problème est variable et qu'elle est trop petite.

### 5.2.6 Nowait

Lorsque les données sont indépendantes, l'exécution de deux régions peut se chevaucher.  
exemple:



```
#pragma omp parallel shared(n,a,b,c,d,e,f) private(i,scale)
{
    #pragma omp for nowait
    for (i=0; i<n; i++) a[i] = b[i] + c[i];
    #pragma omp for nowait
    for (i=0; i<n; i++) d[i] = e[i] + f[i];
    #pragma omp barrier
    scale = sum(a,0,n) + sum(d,0,n);
}
```

### 5.2.7 Collapse

On se sert du collapse pour indiquer le niveaux d'imbrication de nos boucles for. On peut augmenter le niveau de parallélisme avec sa.

### 5.2.8 SIMD

Directive pour les boucles indiquant qu'on veut déplacer le travail sur un processeur SIMD comme un GPU ou un vecteur. Toute fonction qui peut être appelé par du code SIMD doit être déclarés SIMD (`pragma omp declare SIMD`).

Le compilateur devra assurer que le code visé pourra être compilé pour le processeur SIMD ciblé, par exemple avoir une copie en code natif et une copie en bytecode qui pourra être compilé à l'exécution pour le bon GPU.

### 5.2.9 Safelen

La distance maximale entre deux itérations qui seront exécutés en parallèle sur le dispositif SIMD.

### 5.2.10 Aligned

À combien d'octets sont alignés les variables listés

### 5.2.11 Uniforme

paramètre d'une fonction SIMD qui ne varie pas d'un appel à l'autre de la même exécution SIMD concurrente.

### 5.2.12 Target

plusieurs utilisation:

- Target device: spécifier le dispositif SIMD à utiliser
- Target data: spécifier les données à transférer vers/ du dispositif avec la clause map.

- target update: synchroniser certaines données entre l'hôte et le dispositif SIMD.
- Declare target: définir certaines variables sur le dispositif SIMD.

Exemple:

```
#define N 10000
#define M 1024
#pragma omp declare target
float Q[N][N];
#pragma omp declare simd uniform(i) linear(k) notinbranch
float P(const int i, const int k) { return Q[i][k] * Q[k][i]; }
#pragma omp end declare target
float accum(void) {
    float tmp = 0.0;
    int i, k;
    #pragma omp target
    #pragma omp parallel for reduction(+:tmp)
    for (i=0; i < N; i++) {
        float tmp1 = 0.0;
        #pragma omp simd reduction(+:tmp1)
        for (k=0; k < M; k++) {
            tmp1 += P(i,k);
        }
        tmp += tmp1;
    }
    return tmp;
}

//ou..

#define THRESHOLD1 1000000
#define THRESHOLD2 1000
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(float *p, float *v1, float *v2, int N) {
    int i;
    init(v1, v2, N);
    #pragma omp target if(N>THRESHOLD1) map(to: v1[0:N], v2[:N])\
    map(from: p[0:N])
    #pragma omp parallel for if(N>THRESHOLD2)
    for (i=0; i<N; i++) {
        p[i] = v1[i] * v2[i];
    }
    output(p, N);
}
```

*// ou encore..*

```
extern void init(float *, float *, int);
extern int maybe_init_again(float *, int);
extern void output(float *, int);
void vec_mult(float *p, float *v1, float *v2, int N) {
    int i;
    init(v1, v2, N);
    #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])
    {
        int changed;
        #pragma omp target
        #pragma omp parallel for
        for (i=0; i<N; i++) p[i] = v1[i] * v2[i];
        changed = maybe_init_again(v1, N);

        #pragma omp target update if (changed) to(v1[:N])
        changed = maybe_init_again(v2, N);

        #pragma omp target update if (changed) to(v2[:N])
        #pragma omp target
        #pragma omp parallel for
        for (i=0; i<N; i++) p[i] = p[i] + (v1[i] * v2[i]);
    }
    output(p, N);
}
```

C'est commande sont mieux expliquer dans les documentations d'OMP (v4.0+)

### 5.3 Fonctions de supports de threads

- `omp_set_num_threads`: spécifier le nombre de fils d'exécution.
- `omp_get_num_threads`: nombre de fils d'exécution.
- `omp_get_max_threads`: nombre maximal possible de fils d'exécution.
- `omp_get_thread_num`: numéro du fil courant.
- `omp_get_num_procs`: nombre de processeurs disponibles.
- `omp_in_parallel`: dans une région parallèle?
- `omp_set_dynamic`, `omp_get_dynamic`: ajustement dynamique du nombre de fils d'exécution.
- `omp_set_nested`, `omp_get_nested`: permission d'avoir une région parallèle imbriquée avec possiblement plus de fils d'exécution créés.

- OMP\_STACKSIZE: taille des piles pour les fils d'exécution. Variable ?
- omp\_get\_wtime: temps écoulé en secondes.
- omp\_get\_wtick: résolution de l'horloge utilisée pour omp\_get\_wtime.
- omp\_set\_schedule, omp\_get\_schedule: static, dynamic, guided, auto sont les stratégies possibles pour diviser les itérations de boucles entre les fils d'exécution, avec une granularité spécifiée.
- omp\_get\_thread\_limit: nombre maximal de fils disponibles pour le programme.
- omp\_set\_max\_active\_levels, omp\_get\_max\_active\_levels: profondeur maximale d'imbrication de régions parallèles.
- omp\_get\_level, omp\_get\_active\_level: profondeur courante de régions parallèles (actives) imbriquées.
- omp\_get\_ancestor\_thread\_num: numéro du thread parent.
- omp\_get\_team\_size(level): nombre de thread dans la région parallèle ancêtre.

**Précautions** Il faut faire attention au placement des variables partagés pour éviter le faux partage en cache. Il faut aussi s'assurer que les fonction appelées dans les section parallèle sont prévue pour sa.. (threadsafe).

Il faut accéder les matrices par colonne (dernier indice qui varie le plus vite) et finalement choisir des morceaux assez gros pour minimiser le surcoût de contrôle mais assez petit pour permettre d'équilibrer le travail de chaque fil.

## 5.4 Exercices.

### 5.4.1 5.1

**Question** Un outil de détection de problème de synchronisation vous indique une course dans la section réalisée par votre coéquipier qui affirme que tout est correct. Qui croire?

```
found = 0;
```

```
#pragma omp parallel for
for(i = 0 ; i < n ; i++){
    if(a[i] == key) found = 1;
}
if(found) { ... }
```

**Réponse** Un seul changement peut arriver, toujours le même, `found=1`. Il y a effectivement une course mais le résultat final ne dépend pas de la course.

### 5.4.2 5.2

**Question** Votre co-équipier se plaint que votre programme ne profite pas de la performance disponible en raison de la barrière implicite entre les deux boucles. Il veut les mettre `nowait` ou dans des sections. Est-ce correct? Avez-vous mieux à proposer?

```
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < n; i++){
        a[i] = f1(i);
    }
    #pragma omp for
    for (i = 0; i < n; i++){
        b[i] = a[i] + f2(i);
    }
}
```

**Réponse** Avec un ordonnancement statique, `nowait` serait possible. Mieux encore, on peut fusionner les deux boucles.

```
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < n; i++){
        a[i] = f1(i);
        b[i] = a[i] + f2(i);
    }
}
```

### 5.4.3 5.3

**Question** Votre co-équipier a modifié votre programme en ajoutant des `nowait` et obtient un gain de performance appréciable mais ses résultats seront-ils corrects et fiables?

```
#pragma omp parallel
{
    #pragma omp for schedule(static) nowait
    for (i=0; i<n; i++) c[i] = (a[i] + b[i]) / 2.0f;
    #pragma omp for schedule(static) nowait
```

```
    for (i=0; i<n; i++) z[i] = sqrtf(c[i]);  
    #pragma omp for schedule(static) nowait  
    for (i=1; i<=n; i++) y[i] = z[i-1] + a[i];  
}
```

**Réponse** Entre les deux premières boucles, l’ordonnancement statique règle clairement le problème. C’est aussi le cas pour la troisième boucle car tout est décalé de 1 (indice de boucle et accès de z); le même fil opérera sur la même valeur de z. Le dernier `nowait` est inutile puisque c’est la dernière boucle du `parallel`.

#### 5.4.4 4.4

**Question** Votre programme comporte une boucle dont la longueur de chaque itération varie grandement. Discutez de différents mécanismes qui pourraient être utilisés pour assurer néanmoins une bonne parallélisation.

**Réponse** Avec un ordonnancement de boucle dynamique ou guidé, possiblement avec une granularité assez fine, le nombre d’itérations par fil sera ajusté automatiquement.

## Chapter 6

# Contrôle des Années précédents.

### 6.1 Année 2016

#### 6.1.1 1

a) Question de probabilités. On utilise la formule fourni et on utilise celle dans les notes (chapitre1) pour pouvoir faire le calcul.

$p(t)^{100}$  ou 100 est le nombre de cluster.

b) Callgrind regarde les différentes fonctions utilisé par le programme, et sa nous donne de l'informations sur les fonctions comme le temps de calculs, les appels système, etc. On s'en sert pour voir ou mettre notre effort pour parallélisé notre programme.

c) TBB est plus performant, la différence était quand même grande. il gère lui même la répartition du travail et parallélise à généralement à grain plus fin que pThread.

d) On se sert de la loi d'Amdahl, voir note. 2.95 pour 4 coeur avec 0.88 parallélisable. 7.47 pour 64 coeurs

#### 6.1.2 2

a et b voir notes.

c) Faut le changer,  
quand on change de thread, sa ne dérange pas  
quand on change de processus, il faut que notre TLB puisse se servir d'un étiquette (soit un TLB avancés). sinon, il doit le flusher.

**ECOLE POLYTECHNIQUE DE MONTREAL****Département de génie informatique et génie logiciel****Cours INF8601: Systèmes informatiques parallèles (Automne 2016)****3 crédits (3-1.5-4.5)**

---

**CORRIGÉ DU CONTRÔLE PÉRIODIQUE****DATE: Mercredi le 26 octobre 2016****HEURE: 9h30 à 11h20****DUREE: 1H50****NOTE: Toute documentation permise, calculatrice non programmable permise****Ce questionnaire comprend 4 questions pour 20 points**

---

**Question 1 (5 points)**

- a) Dans une grappe de calcul, la probabilité qu'un ordinateur donné fonctionne sans panne pendant une tâche de durée  $t$  (en heures) est donnée par la fonction  $p(t) = 1/(\cdot 01t + 1)$ . Par exemple, la probabilité qu'une tâche puisse se terminer correctement, sans panne du noeud qui l'exécute, est de 1 pour  $t = 0$ , 0 pour  $t = \infty$  et 0.80645 pour  $t = 24h$ . Si une grappe contient 100 noeuds, quelle est la probabilité que tous les noeuds soient opérationnels pendant une tâche parallèle de 1h? De 2 h? **(2 points)**

*Pour 1h, la probabilité d'exécution correcte est de  $1/1.01$ . Pour 2h elle est de  $1/1.02$ . La probabilité que tous les noeuds soient opérationnels est donc de  $(1/1.01)^{100} = 0.3697$  pour 1h et de  $(1/1.02)^{100} = 0.1380$  pour 2h.*

- b) Lors du premier TP, vous avez utilisé l'outil Callgrind de l'environnement Valgrind. Quelle information est-ce que cet outil fournit? Comment avez-vous utilisé cette information, quelle était son utilité? **(1 point)**

*L'outil Callgrind permet de calculer le nombre de fois que chaque fonction est exécutée et d'estimer le temps passé en exécution dans chaque fonction elle-même (sans les appels imbriqués) et chaque fonction ainsi que celles appelées de manière imbriquée. Cette information aide à décider quelles fonctions paralléliser et permet d'estimer le temps qui peut être sauvé en divisant le temps des fonctions ciblées par le facteur de parallélisme attendu.*



- c) Dans le cadre du premier TP, vous avez effectué le même calcul à l'aide des POSIX Threads et à l'aide de TBB. Quelle version était la plus efficace? Est-ce que la différence était grande? Comment expliquez-vous cette différence? **(1 point)**

*La librairie TBB est en général un peu plus efficace que les POSIX Threads. La différence n'est cependant pas très grande. Un des facteurs de cette performance supérieure est la répartition dynamique du travail entre les threads et l'organisation récursive, en arbre, pour la création des threads, comparé à la plupart des programmes POSIX Threads qui font une boucle dans le thread principal qui crée sériellement tous les threads parallèles.*

- d) Un programme est composé de sections parallélisables et non parallélisables. La fraction parallélisable est de 88%. Ce programme est parallélisé et exécuté sur un ordinateur de 4 coeurs, quel sera le facteur d'accélération? Sur un ordinateur à 64 coeurs? **(1 point)**

*Soit  $t_0$  le temps initial,  $t_1$  le nouveau temps deviendra  $t_1 = (1 - fp)t_0 + fp \times t_0/n$ , l'accélération est donc de  $a = t_0/t_1 = 1/((1 - fp) + fp/n)$ , soit  $1/(.12 + .88/4) = 2.94$  pour 4 coeurs, et  $1/(.12 + .88/64) = 7.47$  pour 64 coeurs.*

## Question 2 (5 points)

- a) Un ordinateur 64 bits, dont la mémoire est adressable à l'octet, utilise des tables de pages à 4 niveaux et des pages de 8Kio. Chaque noeud de la table de pages entre dans une page. Montrez comment l'adresse se décompose en décalage dans la page, index pour chacun des 4 niveaux dans la table, et s'il y a lieu les bits restants. A quoi servent les bits restants s'il y en a? Décomposez l'adresse virtuelle 0x0000ABCDEF012345 en ces différentes composantes (décalage, index3, index2...). **(2 points)**

*Une page de 8Kio ( $2^{13}$ ) requiert 13 bits pour le décalage et peut contenir 1024 entrées de 8 octets (64 bits) lorsqu'utilisée comme noeud dans la table de pages. L'index dans un noeud de la table de pages requiert donc 10 bits pour indexer ces 1024 entrées. Le reste des bits  $64 - 13 - 4 \times 10 = 11$  est inutilisé. Ces bits inutilisés limitent l'espace virtuel adressable par un processus à  $2^{53}$ . L'adresse virtuelle se décompose donc ainsi: 11 bits inutilisés, 10 bits index0, 10 bits index1, 10 bits index2, 10 bits index3, 13 bits décalage. Ceci donne pour l'adresse 0x0000ABCDEF012345: 0b0000 0000 0000 0000 1010 1011 1100 1101 1110 1111 0000 0001 0010 0011 0100 0101 en binaire, ou 0b000 0000 0000|00 0001 0101|01 1110 0110|11 1101 1110|00 0000 1001|0 0011 0100 0101 en séparant les champs. On obtient ainsi champ par champ en hexadécimal: inutilisés 0x0000, index0 0x015, index1 0x1E6, index2 0x3DE, index3 0x009, décalage 0x0345.*

- b) Un ordinateur possède une cache L1 MESI de 64Kio, avec des ensembles de 8 blocs, pour chacun de ses 4 coeurs. Chaque bloc en cache contient 128 octets. Cette cache est initialement vide. Les adresses suivantes sont accédées en séquence, en lecture (R) ou en écriture (W), sur les processeurs spécifiés (P0 à P3). Donnez l'état des cases non vides (M, E, S ou I) après chaque accès. **(2 points)**

P0 R 0x12A; P0 R 0x02A; P1 W 0x12A 2;  
P2 W 0x12A 4; P2 R 0x12A; P1 W 0x02A 8

Pour des blocs de 128 octets, les 7 derniers bits d'adresse représentent le décalage dans le bloc. La cache contient  $2^{16}/2^7 = 2^9 = 512$  blocs de 128 octets soit  $512/8 = 64$  ensembles de 8 blocs. Il faut donc 6 bits pour représenter l'ensemble en cache et le reste constitue l'étiquette du bloc (qui distingue un bloc des autres qui peuvent se retrouver dans le même ensemble). Pour l'adresse 0x12A, cela donne 0b0001 0010 1010 ou 0b100 0010/010 1010, soit étiquette 0, ensemble 0x02, décalage 0x2A. Pour l'adresse 0x02A, cela donne 0b0000 0010 1010 ou 0b100 0000/010 1010, soit étiquette 0, ensemble 0x0, décalage 0x2A. Initialement, tous les blocs en cache ont le statut invalide (I). Les deux premières opérations font charger deux blocs dans la cache de P0 (premier bloc de l'ensemble 2, étiquette 0, statut exclusif (E), premier bloc de l'ensemble 0, étiquette 0, statut E). L'opération suivante invalide le premier bloc de l'ensemble 2 de P0, charge le premier bloc de l'ensemble 2 de P1 et le modifie, ce qui donne le statut modifié (M). La même chose se produit ensuite sur P2 avec la quatrième opération qui invalide le premier bloc de l'ensemble 2 de P1. La cinquième opération ne modifie pas le statut du premier bloc de l'ensemble 2 de P2 qui reste à M. Finalement, la dernière opération invalide le premier bloc de l'ensemble 0 de P0, charge le premier bloc de l'ensemble 0, étiquette 0, de P1 et le modifie, ce qui donne le statut modifié.

- c) La cache de pré-translation d'adresse (TLB) permet de convertir les adresses virtuelles d'un processus en ses adresses physiques, en mémorisant un sous-ensemble (cache) du contenu de la table de pages d'un processus. Sur certaines variantes plus avancées de TLB, une étiquette est ajoutée aux adresses virtuelles dans le TLB pour identifier le processus associé. Doit-on mettre à jour le contenu du TLB lorsque l'espace d'adressage du processus est modifié (e.g., une zone de mémoire partagée est ajoutée par un appel à `mmap()`)? Expliquez. Que doit-on faire avec le contenu du TLB, selon que ce soit un TLB régulier ou une "variante avancée", lorsque l'ordonnanceur change de thread mais tout en restant dans le même processus? Lorsque l'ordonnanceur change de processus? **(1 point)**

Lorsqu'une partie de l'espace d'adressage d'un processus est modifiée, le contenu correspondant dans la table de pages change et les entrées correspondantes dans le TLB doivent être invalidées car elles ne seront plus à jour en fonction de ces changements. Sur certaines architectures, des opérations spécifiques sont disponibles pour cela. Autrement, il faut remettre à zéro le contenu du TLB. Lorsque l'ordonnanceur change de thread mais reste dans le même processus, nous restons dans le même espace virtuel et cela n'affecte pas le TLB. Toutefois, si on change de processus, il faut normalement remettre à zéro le contenu du TLB. Cependant, si le TLB contient des étiquettes qui identifient le processus, chaque pré-translation se fait en tenant compte du processus présent et il n'y a pas de problème. Dans ce cas, il n'est pas nécessaire de changer le contenu du TLB lors d'un tel réordonnancement, même s'il change le processus en exécution. Cela permet possiblement de conserver les entrées de l'autre processus pour quand il reviendra.

### Question 3 (5 points)

- a) Ecrivez un programme, utilisant la librairie TBB, qui offre la fonction `int CheckSorted(float v[], int size)` permettant d'effectuer le calcul suivant. L'argument `v` contient un vecteur de

nombres à virgule flottante alors que l'argument `size` contient la taille de `v`. La fonction `CheckSorted` doit calculer et retourner le nombre de fois que des items dans le vecteur ne sont pas en ordre croissant (i.e., l'item à la position `i` est plus petit que l'item en position `i - 1`, pour `i` entre 1 et `size - 1` inclusivement). **(3 points)**

```
struct Count {
    int value;
    float *v;
    Count() : value(0) {}
    Count( Count& c, split ) { v = c.v; value = 0; }

    void operator()( const blocked_range<int>& r ) {
        int temp_count = value;
        for(int i = r.begin(); i != r.end(); i++ ) {
            if(v[i] < v[i - 1]) temp_count++;
        }
        value = temp_count;
    }
    void join( Count& rhs ) {value += rhs.value;}
};

int CheckSorted(float v[], int size) {
    Count c;
    c.v = v;
    parallel_reduce( blocked_range<int>(1, size), c);
    return c.value;
}
```

- b) Un programme multi-thread s'exécute sur plusieurs processeurs en parallèle. Afin d'aller plus vite, il n'utilise pas de verrou. Un thread veut mettre à jour des valeurs `data_a_1` et `data_a_2` puis `data_b_1` et `data_b_2` et indiquer lorsqu'elles sont valides en mettant à 1 `valid_a` et `valid_b` respectivement. Le programme proposé suit. Doit-on ajouter des barrières mémoire pour avoir un comportement correct (i.e., ne pas lire les données avant qu'elles aient été mises à jour)? Si oui, modifiez le programme en ajoutant les barrières mémoire minimales requises? **(2 points)**

(initialement toutes les variables sont à 0)	
Processeur 0	Processeur 1
<code>data_a_1 = 20;</code>	<code>if(valid_a) {</code>
<code>data_a_2 = 18;</code>	<code>    new_a = data_a_2;</code>
<code>data_b_2 = 8;</code>	<code>}</code>
<code>data_b_1 = 10;</code>	<code>if(valid_b) {</code>
<code>valid_b = 1;</code>	<code>    new_b = data_b_1;</code>
<code>valid_a = 1;</code>	<code>}</code>

*Il faut insérer une barrière d'écriture pour s'assurer que toutes les valeurs ont bien été écrites en mémoire commune avant de ne confirmer le tout par le changement à 1 de valid\_a et valid\_b. Une seule barrière mémoire suffit pour les deux puisque les accès aux data\_a et data\_b sont regroupés. Le pendant est qu'il faut une barrière de lecture après avoir détecté que valid\_a (ou valid\_b) est à 1 afin de s'assurer que les mises à jour des variables data sont bien arrivées au processeur à partir de la mémoire commune. Il n'y a pas de garantie d'ordre d'arrivée entre valid\_a et valid\_b et il se peut que l'un soit à 1 et l'autre à 0, il faut donc une barrière de lecture pour chacune des deux conditions, si on assume que les deux sont indépendantes. En pratique, on pourrait argumenter que chacune des deux peut agir comme validation pour les data\_a et les data\_b, puisque leur affectation est groupée, et réorganiser le code avec if(valid\_a || valid\_b) et n'avoir qu'une seule barrière en lecture.*

```
(initialement toutes les variables sont à 0)
Processeur 0                               Processeur 1

data_a_1 = 20;                               if(valid_a) {
data_a_2 = 18;                               cmm_smp_rmb(); new_a = data_a_2;
data_b_2 = 8;                                }
data_b_1 = 10;                               if(valid_b) {
cmm_smp_wmb();                               cmm_smp_rmb(); new_b = data_b_1;
valid_b = 1;                                }
valid_a = 1;
```

## Question 4 (5 points)

- a) On vous fournit la section de code suivante en OpenMP. Vous devez proposer plusieurs améliorations qui permettront d'obtenir le même résultat mais plus efficacement. Fournissez une nouvelle version de cette section de programme et expliquez en une ou deux lignes de texte chaque amélioration apportée. **(2 points)**

```
#pragma omp parallel for
for(int i = 0; i < nb_cpu; i++) {
    for(int j = 0; j < nb_task; j++) {
        for(int k = 0; k < nb_resource; k++) {
            by_cpu[i] += time[k][i][j];
            #pragma omp atomic
            sum += time[k][i][j];
        } } }
```

```
#pragma omp parallel for
for(int i = 0; i < nb_cpu; i++) {
    for(int j = 0; j < nb_task; j++) {
        for(int k = 0; k < nb_resource; k++) {
```

```

        by_task[j] += time[k][i][j];
    } } }

#pragma omp parallel for
for(int i = 0; i < nb_cpu; i++) {
    for(int j = 0; j < nb_task; j++) {
        for(int k = 0; k < nb_resource; k++) {
            by_resource[k] += time[k][i][j];
        } } }

```

*Plusieurs optimisations sont possibles. Premièrement, on peut fusionner les trois boucles, ce qui évite de recalculer les indices trois fois et améliore la localité de référence. Ensuite, on peut convertir l'opération atomique en réduction, ce qui est plus efficace, changer l'ordre des indices pour varier le plus à droite en dernier dans la matrice time, ce qui améliore la localité de référence en cache, et éviter d'accéder à répétition l'élément de la matrice, calcul inutile que le compilateur optimiseur éviterait possiblement. Par ailleurs, le programme original est problématique car les différents threads pouvaient accéder en parallèle les mêmes entrées de `by_task[j]` et `by_resource[k]`. Il est maintenant possible, avec OpenMP 4.x, de faire des réductions sur des vecteurs et ainsi éliminer cette course dans le programme.*

```

#pragma omp parallel for reduction(+:sum) \
    reduction(+:by_cpu) reduction(+:by_task) \
    reduction(+:by_resource)
for(int k = 0; k < nb_resource; k++) {
    for(int i = 0; i < nb_cpu; i++) {
        for(int j = 0; j < nb_task; j++) {
            int element = time[k][i][j];
            by_cpu[i] += element;
            by_task[j] += element;
            by_resource[k] += element;
            sum += element;
        } } }

```

- b) Le programme OpenMP suivant s'exécute et produit une sortie sur stdout. Donnez une sortie possible produite par l'exécution de ce programme. Est-ce que la sortie peut changer d'une exécution à l'autre? Expliquez. **(2 points)**

```

int main(int argc, char **argv)
{ int nb_thread, thread;
  printf("a)\n");
  omp_set_num_threads(4);
  #pragma omp parallel private(nb_thread, thread)
  { printf("b)\n");
    #pragma omp for schedule(static)

```

```

for(int i = 0; i < 8; i++) {
    nb_thread = omp_get_num_threads();
    thread = omp_get_thread_num();
    printf("c) thread %d / %d, i = %d\n", thread, nb_thread, i);
}
#pragma omp master
printf("d) thread %d / %d\n", thread, nb_thread);
#pragma omp critical
printf("e) thread %d / %d\n", thread, nb_thread);
#pragma omp single
printf("f) thread %d / %d\n", thread, nb_thread);
}
printf("g) thread %d / %d\n", thread, nb_thread);
}

```

*La sortie peut en effet changer car, si l'ordre d'exécution à l'intérieur d'un thread reste le même dans ce cas-ci (pas de données partagées et répartition statique de la boucle), l'ordre entre les threads peut changer. Ainsi, a et g sont hors de la section parallèle et ne changeront pas d'ordre. Les autres printf peuvent changer d'ordre. De plus, la sortie f n'est effectuée que par un seul thread mais celui-ci peut changer d'une fois à l'autre (pragma omp single), ce qui changera le numéro de thread associé à f.*

a)  
 b)  
 c) thread 0 / 4, i = 0  
 c) thread 0 / 4, i = 1  
 b)  
 c) thread 3 / 4, i = 6  
 c) thread 3 / 4, i = 7  
 b)  
 c) thread 2 / 4, i = 4  
 c) thread 2 / 4, i = 5  
 b)  
 c) thread 1 / 4, i = 2  
 c) thread 1 / 4, i = 3  
 d) thread 0 / 4  
 e) thread 1 / 4  
 e) thread 2 / 4  
 e) thread 3 / 4  
 e) thread 0 / 4  
 f) thread 1 / 4  
 g) thread 0 / 0

- c) Peut-on changer la taille de la pile en OpenMP? Comment? Comment sait-on si on doit changer la taille de la pile? (1 point)

*OpenMP permet de spécifier la taille à utiliser pour les piles via une variable d'environnement, OMP\_STACKSIZE. Si la taille est trop petite, le programme essaiera d'accéder des cases mémoire qui dépassent la zone disponible, ce qui peut résulter en une interruption pour cause d'erreur de segmentation, ou même en la corruption de zones mémoires dédiées à d'autres fins dans notre processus.*

Le professeur: Michel Dagenais

## **6.2    Années 2014**



**ÉCOLE POLYTECHNIQUE DE MONTREAL**

**Département de génie informatique et génie logiciel**

**Cours INF8601: Systèmes informatiques parallèles (Automne 2014)**

**3 crédits (3-1.5-4.5)**

---

**CORRIGÉ DU CONTRÔLE PÉRIODIQUE**

**DATE: Lundi le 3 novembre 2014**

**HEURE: 9h30 à 11h20**

**DUREE: 1H50**

**NOTE: Toute documentation permise, calculatrice non programmable permise**

**Ce questionnaire comprend 4 questions pour 20 points**

---

## Question 1 (5 points)

- a) Une entreprise offre un service d'espace disque à distance aux usagers, AieNuage. L'entreprise veut offrir un service fiable mais ne veut pas assumer les coûts de prendre des copies de sécurité sur ruban. Elle utilise donc un système avec un grand nombre de noeuds avec redondance. Chaque noeud est constitué d'une unité de traitement et d'un disque de 1TB, pouvant accueillir les fichiers de 100 usagers. Les fichiers de chaque usager sont stockés en permanence sur 3 noeuds différents, n'importe lequel des 3 pouvant alors servir les requêtes pour cet usager. Si une unité de traitement est en panne, le noeud est indisponible temporairement. Si un disque est en panne, les données de cette copie sont irrémédiablement perdues. La probabilité qu'une unité de traitement soit en panne est de 0.001. La probabilité qu'un disque soit en panne est de .002. Quelle est la probabilité que les fichiers d'un usager à un moment donné soient indisponibles (perdus ou non)? Irrémédiablement perdus? Si le service compte 2 milliards d'usagers, combien ont perdu leurs fichiers irrémédiablement à un instant donné? **(2 points)**

*Les fichiers d'un usager sont sur 3 noeuds spécifiques. Si les 3 disques correspondants sont en panne simultanément, les fichiers sont irrémédiablement perdus, soit une probabilité de  $.002^3 = 8 \times 10^{-9}$ , ou  $10^{-9} \times 2000000000 = 16$  usagers. Pour que les fichiers d'un usager soient disponibles sur un noeud, il faut que l'unité de traitement et le disque soient fonctionnels, une probabilité de  $(1 - .002)(1 - .001) = 0.997002$ . Pour que les fichiers soient non disponibles (temporairement ou irrémédiablement perdus), il faut que les 3 noeuds soient non disponibles  $(1 - 0.997002)^3 = 26.946 \times 10^{-9}$ .*

- b) Sur les processeurs Intel, il est possible de varier la taille des pages en mémoire virtuelle, et d'utiliser soit des petites pages de 4Kio ou des grandes pages de 4Mio. Quel est l'impact de ce choix sur le taux de succès pour la cache de pré-traduction d'adresse (TLB)? Le taux de succès en mémoire cache? L'utilisation efficace de la mémoire physique? **(1 point)**

*Les grandes pages demandent beaucoup moins d'entrées dans le TLB et améliorent donc beaucoup le taux de succès du TLB si elles sont le moins bien utilisées en étant assez pleines. La taille des pages a peu ou pas d'impact sur le taux de succès en mémoire cache puisque les blocs de cache sont beaucoup plus petits que même les petites pages. Les grandes pages peuvent causer une plus grande perte d'espace par fragmentation puisqu'on arrondit chaque segment d'un processus à la frontière de taille de page (4Mio) supérieure, et donc causer une utilisation moins efficace de la mémoire physique.*

- c) Expliquez ce qui peut causer un faux partage de données en cache. Quel en est l'impact? Comment peut-on se débarrasser d'un tel problème? Donnez un exemple d'allocateur en TBB qui permet d'allouer une donnée qui ne présentera pas de tel problème. **(1 point)**

*Lorsque par malchance deux variables différentes se retrouvent dans la même ligne de cache, et que ces deux variables sont utilisées par deux fils d'exécution parallèles qui*

*exécutent sur deux coeurs avec caches séparées, ceci cause un conflit sur cette ligne de cache en raison du faux partage. En utilisant le `cache_aligned_allocator` de TBB, on peut s'assurer que deux objets ne seront pas alloués dans la même ligne de cache, puisque chacun sera aligné au début d'une ligne de cache. Ceci cause un peu de perte d'espace mémoire mais empêche le faux partage.*

- d) Vous venez de paralléliser une fonction  $F$  d'un programme qui était sériel. La nouvelle version de  $F$  peut ainsi utiliser efficacement les 64 processeurs de votre ordinateur. Les autres fonctions sont inchangées. Le programme s'exécute maintenant 10 fois plus vite au total. Quelle fraction du temps prenait initialement la fonction  $F$ ? **(1 point)**

*Soit  $fp$  la fraction parallélisable. Le programme prenait un temps  $t$  et prends maintenant  $t/10 = t \times ((1 - fp) + fp/64)$  donc  $1/10 = 1 - fp \times 63/64$  ou  $0.9 \times 64/63 = fp$  donc  $fp = 0.9143$ .*

## Question 2 (5 points)

- a) Un très grand vecteur  $v$  contient des entiers signés qui représentent les entrées et les sorties de participants pendant *la fête de la plage* à Polytechnique. Par exemple,  $+1$  indique une entrée et  $-2$  la sortie de deux personnes. Sachant que la salle était initialement vide, il faut calculer le vecteur  $t$  qui contient le nombre de participants dans la salle après avoir traité la donnée du vecteur  $v$  à la position correspondante. On veut effectuer ce calcul avec la librairie TBB. Quelle structure (parallel for, parallel reduce, parallel scan ou parallel do) suggérez-vous d'utiliser? Ecrivez le coeur de ce programme, comme dans les exemples utilisés pour les dispositifs du cours expliquant TBB. **(2 points)**

*Il faut avoir calculé la somme de toutes les entrées et sorties afin d'avoir le nombre de participants présents à un instant donné. Ceci peut se faire efficacement avec un parallel scan qui permet de calculer en parallèle la variation pour différentes sections du vecteur  $v$  et ensuite de combiner cette information pour calculer, dans une deuxième passe, de nouveau en parallèle, le nombre total de participants.*

```
class Body {
    T sum; T* const t; const T* const v;

    Body( T t_[], const T v_[] ) : sum(0), v(v_), t(t_) {}

    T get_sum() const {return sum;}

    template<typename Tag>
    void operator()( const blocked_range<int>& r, Tag ) {
        T temp = sum;
        for( int i=r.begin(); i<r.end(); ++i ) {
            temp = temp + v[i];
        }
    }
};
```

```

        if( Tag::is_final_scan() ) t[i] = temp;
    }
    sum = temp;
}

Body( Body& b, split ) : v(b.v), t(b.t), sum(0) {}

void reverse_join( Body& a ) { sum = a.sum + sum;}

void assign( Body& b ) {sum = b.sum;}
};

float DoParallelScan( T t[], const T v[], int n ) {
    Body body(t,v);
    parallel_scan(blocked_range<int>(0,n), body );
    return body.get_sum();
}

```

- b) Dans le premier travail pratique, vous avez examiné et comparé le travail de PThread et TBB. Quels sont les appels système impliqués dans la synchronisation des fils d'exécution de PThread et TBB? **(1 point)**

*Avec PThread, les nouveaux fils d'exécution sont créés avec l'appel système `sys_clone`. Par la suite, la synchronisation des fils d'exécution, par exemple avec des mutex ou `pthread_join`, se fait avec `sys_futex` lorsqu'une attente est nécessaire. La situation avec TBB est assez semblable. La principale différence est que le fil original participe au calcul et que la création de nouveaux fils se fait récursivement, en arbre. Il est amusant de noter que TBB fait de nombreux appels système `sched_yield`. TBB utilise probablement une approche graduelle pour l'acquisition d'un mutex ou d'un sémaphore: quelques essais avec une boucle active, quelques essais avec une boucle de `sched_yield` et finalement un appel aux mutex du système d'exploitation avec `futex`. Sur Linux, il est douteux que les appels à `sched_yield` soient très utiles.*

- c) Un ordinateur 64 bits fonctionne avec une table de pages à 4 niveaux. Seul un bit de son espace adressable n'est pas utilisé, laissant 63 bits pour ses adresses virtuelles. Toutes les pages ont la même taille et sont utilisées autant pour peupler l'espace virtuel des processus que pour contenir les différents niveaux de la table de page. Quelle est la taille d'une page? **(1 point)**

*Une page de  $2^n$  octets prend  $n$  bits pour le décalage dans la page et permet de stocker  $2^n/8$  entrées de 64 bits (8 octets) dans un morceau de la table de pages. Pour choisir parmi ces  $2^n/8$  entrées, il faut  $n - 3$  bits. L'adresse se décompose en 1 bit inutilisé,  $n$  bits pour le décalage et 4 champs de  $n - 3$  bits (un par niveau de la table de pages). Nous avons donc  $64 = 1 + n + 4 \times (n - 3) = 5n - 13$  donc  $64 - 1 + 12 = 5n$  et  $n = 15$ . La taille d'une page doit donc être de  $2^{15}$  octets ou 32Kio.*

- d) Un processeur 64 bits possède une cache L1 de 256Kio avec des blocs de 256 octets. La fonction de correspondance est directe-associative avec un degré d'associativité de 4, ce qui donne des ensembles de 4 blocs chacun. Si on vous donne l'adresse suivante en hexadécimal 0x00000000A0B0C0D. Donnez le numéro d'ensemble en cache, le décalage dans le bloc et l'étiquette associés à cette adresse. **(1 point)**

*Cette adresse peut être exprimée en binaire ainsi: 0000 0000 0000 0000 0000 0000 0000 0000 1010 0000 1011 0000 1100 0000 1101. Le décalage dans le bloc est ce chiffre modulo 256, soit les 8 derniers bits ou 0000 1101 (0x0D). Un ensemble contient  $4 \times 256 = 1024$  octets. La cache contient donc  $256 \times 1024$  octets ou 256 ensembles de 1024 octets. Le numéro d'ensemble en cache est donné par le modulo 256 du reste de l'adresse, soit les 8 bits qui précèdent les 8 derniers ou 0000 1100 (0x0C). L'étiquette est ce qui distingue deux blocs différents qui peuvent se retrouver en cache dans le même ensemble, soit le reste de l'adresse, ou 0x00000000A0B.*

### Question 3 (5 points)

- a) Un des modèles étudiés de cohérence en mémoire partagée est l'ordre partiel des écritures (PSO). Expliquez lesquels réordonnements des accès mémoire (lecture ou écriture versus lecture ou écriture) sont possibles? Donnez un exemple de structure matérielle qui pourrait expliquer de tels réordonnements. **(2 points)**

*Avec PSO, les écritures peuvent non seulement être retardées par rapport aux lectures mais l'ordre des écritures n'est pas respecté. Les écritures sont typiquement retardées par rapport aux lectures en présence de queues d'écritures qui permettent de ne pas bloquer et de laisser les écritures se faire en différé. Lorsqu'on a des bancs de mémoire cache parallèles, on peut utiliser une queue d'écriture par banc et il peut alors arriver qu'une queue soit plus pleine que l'autre. Certaines écritures seraient alors retardées par rapport à d'autres qui ont été faites en même temps ou même un peu avant.*

- b) La section de code suivante s'exécute sur un ordinateur avec ordonnancement faible. Quelles sont les sorties possibles produites par printf? Quelles barrières devrait-on insérer, et où, pour avoir comme seules sorties possibles 1 2 3 ou 0 0 0? **(2 points)**

(initialement valid, a, b et c sont 0)

Processeur 0

Processeur 1

a=1;

if(valid) printf("%d %d %d\n", a, b, c);

b=2;

else printf("0 0 0\n");

c=3;

valid=1;

*Initialement, tout est possible, la mise à jour de chacune des valeurs a, b et c peut ou non avoir atteint le processeur 1. Il est donc possible d'avoir 0 0 0, 0 0 3, 0 2 0, 0 2 3, 1 0 0, 1 0 3, 1 2 0, 1 2 3. En mettant une barrière d'écriture avant valid = 1, nous*

sommes certains que la mémoire centrale verra les écritures de *a*, *b* et *c* avant de voir celle de *valid*. A l'inverse, en mettant une barrière de lecture après la lecture de *valid*, nous sommes certains que toute mise à jour retardée de *a*, *b* et *c* sera prises en compte. A ce moment, soit que *valid* est à 0 et que les valeurs de *a*, *b* et *c* peuvent ou non être en train de se faire mettre à jour, soit que *valid* est à 1 et *a*, *b* et *c* ont été mises à jour et ces mises à jour sont rendues au processeur 1.

Processeur 0

```
a=1;
b=2;
c=3;
smp_wmb();
valid=1;
```

Processeur 1

```
if(valid) {
    smp_rmb();
    printf("%d %d %d\n", a, b, c);
}
else printf("0 0 0\n");
```

- c) Vous devez programmer une application avec plusieurs fils d'exécution de manière à maximiser l'utilisation de toutes les ressources disponibles sur le serveur (processeurs, disques...). Chaque fil traite une requête à la fois qui peut nécessiter du calcul, des accès réseau et des accès disque. Puisque ce logiciel sera installé sur une grande variété de serveurs, vous ne pouvez savoir à l'avance quel sera le nombre de processeurs ou de disques, ni la fraction du temps passée à attendre après les requêtes réseau. Par ailleurs, il faut éviter d'avoir beaucoup trop de fils d'exécution, puisque cela encombre les queues du système d'exploitation, et il ne faut pas non plus continuellement créer et détruire des fils d'exécution, puisqu'il y a un coût non négligeable associé à cela. Proposez une organisation efficace pour décider combien de fils d'exécution créer et pour gérer l'utilisation et l'évolution de ces fils d'exécution. **(1 point)**

*Le plus simple est de commencer avec un nombre de fils d'exécution qui est le double du nombre de processeurs. Ensuite, à chaque nouvelle requête reçue, on vérifie la charge du système. Si le CPU ou les disques sont surchargés, (longue queue d'attente, load average beaucoup supérieur au nombre de processeurs), on peut mettre la requête en queue ou même la refuser puisqu'ajouter des fils d'exécution n'aidera en rien. Si la charge n'est pas très forte, on peut refiler la requête à un fil d'exécution inactif (fil en attente sur la queue des requêtes à traiter) ou on peut créer un nouveau fil, s'il n'en reste plus de libre. Lorsqu'un fil termine le traitement d'une requête, on peut le mettre en attente, ou il peut être détruit si un grand nombre de fils disponibles sont déjà en attente.*

## Question 4 (5 points)

- a) Le programme suivant calcule la distribution des couleurs dans une grande image. Convertissez ce programme sériel en programme OpenMP efficace. **(2 points)**

```
void ColorDistribution(unsigned char image[2048][2048],
```

```

    unsigned int d[256])
{
    for(int i = 0; i < 2048; i++)
        for(int j = 0; j < 2048; j++) d[image[i][j]]++;
}

```

*On serait tenté de tout simplement convertir la première boucle en `parallel for`. Toutefois, l'histogramme stocké dans le vecteur `d` est partagé pour les différentes itérations exécutées en parallèle et il en résulterait de la corruption. Une opération atomique permettrait de le mettre à jour correctement mais serait plutôt inefficace.*

```

void ColorDistribution(unsigned char image[2048][2048],
    unsigned int d[256])
{
    int i, j;

    # pragma omp parallel for private(i,j)
    for(i = 0; i < 2048; i++) {
        for(j = 0; j < 2048; j++) {
            #pragma omp atomic
            (d[image[i][j]])++;
        }
    }
}

```

*Une manière efficace d'obtenir le bon résultat est de faire les sommes dans des copies privées, en ne faisant les sommes dans le vecteur global qu'à la fin du travail. Ici on crée un vecteur local pour chaque fil, et son contenu est copié dans le vecteur global à la fin.*

```

void ColorDistribution(unsigned char image[2048][2048],
    unsigned int d[256])
{
    int i, j;

    # pragma omp parallel private(i,j)
    { unsigned int *d_private = new unsigned int[256];
      for(i = 0; i < 256; i++) d_private[i] = 0;

      # pragma omp for
      for(i = 0; i < 2048; i++) {
          for(j = 0; j < 2048; j++) {
              (d_private[image[i][j]])++;
          }
      }
    }
}

```

```

    }

    #pragma omp critical
    for(i = 0; i < 256; i++) d[i] += d_private[i];
    delete d_private;
  }
}

```

- b) Lorsqu'une boucle `parallel for` est rencontrée en OpenMP, la librairie de support à l'exécution doit répartir les itérations entre les différents fils d'exécution. En supposant que, dans le court programme montré ici, la valeur de `img->width` est de 1024, que le nombre de processeurs disponibles est 8, et que la durée de la fonction `f1` est assez variable, expliquez comment le travail pourrait être réparti efficacement sur plusieurs fils d'exécution par OpenMP, en spécifiant bien quel fil traitera quelles itérations à quel moment. **(2 points)**

```

int encode(struct image *img)
{
    int i, j;

    #pragma omp parallel for private(i,j)
    for (i = 0; i < img->width; i++) {
        for (j = 0; j < img->height; j++) {
            img->data[i][j] = f1(img->data[i][j]);
        }
    }
}

```

*Lorsqu'on ne spécifie rien, OpenMP va typiquement allouer deux fois plus de fils que le nombre de processeurs et utiliser une stratégie guidée pour répartir les itérations. Il pourrait donc partir 16 fils et donner à chacun un morceau assez gros pour commencer, par exemple 32 itérations. Le fil 0 traiterait  $i$  de 0 à 31, le fil 1 de 32 à 63... et le fil 15 de 480 à 511. Au fur et à mesure que les fils terminent, ils peuvent obtenir un second morceau, plus petit, par exemple de 16. A chaque fois, la taille choisie est proportionnelle au nombre d'itérations restantes fois le nombre de fils d'exécution.*

- c) Dans une boucle OpenMP, vous avez besoin de calculer une somme globale. Vous pourriez prendre une variable globale protégée par un verrou (section critique ou mutex), déclarer une variable comme une réduction pour la boucle, ou utiliser une opération d'incrément atomique sur une variable globale. Qu'est-ce qui sera le plus performant? Pourquoi? **(1 point)**

*Une opération atomique sera généralement plus efficace qu'un mutex puisque la prise et le relâchement de mutex sont eux-mêmes réalisés à partir d'une opération atomique.*



*Par contre, la réduction sera beaucoup plus efficace puisque le gros des opérations sera effectué sur une copie locale de la somme.*

Le professeur: Michel Dagenais

### **6.3 Année 2013**

**ECOLE POLYTECHNIQUE DE MONTREAL****Département de génie informatique et génie logiciel****Cours INF8601: Systèmes informatiques parallèles (Automne 2013)****3 crédits (3-1.5-4.5)**

---

**CORRIGÉ DU CONTRÔLE PÉRIODIQUE****DATE: Mardi le 22 octobre 2013****HEURE: 14h45 à 16h35****DUREE: 1H50****NOTE: Toute documentation permise, calculatrice non programmable permise****Ce questionnaire comprend 4 questions pour 20 points**

---

**Question 1 (5 points)**

- a) Une application utilise énormément de mémoire et le système de mémoire virtuelle doit ainsi régulièrement transférer certaines pages sur le disque. Une optimisation suggérée est d'avoir la possibilité de compresser des pages en mémoire. Lorsqu'une page est très utilisée elle est conservée telle quelle, lorsqu'une page est moins utilisée elle est compressée mais conservée en mémoire et peut être décompressée au prochain accès, et lorsqu'une page est très peu utilisée elle est transférée sur disque pour être relue au prochain accès. Cette technique permet de conserver jusqu'à deux fois plus de pages en mémoire lorsqu'elles sont compressées mais demande un certain effort de calcul pour la compression / décompression. En raison de ce compromis, cette technique est présentement marginalement utile. Pour chacun des paramètres suivants, dites s'il rend plus avantageux ou moins avantageux cette technique: i) augmentation de la vitesse du disque en passant à un disque à semi-conducteur (SSD), ii) augmentation de la vitesse du CPU, iii) augmentation du nombre de CPU, iv) augmentation de la taille de la mémoire? Justifiez? **(2 points)**

*Si le disque est plus rapide, le coût de transférer une page vers le disque est moindre et la technique est moins intéressante. Une augmentation de la vitesse du CPU ou du*

*nombre de CPU donne plus de ressources pour la compression / décompression, rend le disque plus lent en terme relatif et accroît l'intérêt de la technique. Si la mémoire est plus grande, les pages à transférer sont moins nombreuses et parmi celles les moins souvent utilisées, la technique est alors moins intéressante.*

- b) Un centre de données contient 10 000 noeuds. Chaque noeud est connecté à une unité externe de disques en RAID contenant 5 disques dont au moins 4 doivent être fonctionnels pour opérer correctement. Chaque noeud en lui-même est en panne en moyenne 8 heures par année. Par ailleurs, chaque unité RAID est en panne lorsque 2 disques ou plus sont défectueux, avec chaque disque qui a une probabilité de panne de 0.001. Quelle est la probabilité de panne pour un noeud (noeud lui-même ou unité de disque en panne) et donc quel est le nombre moyen de noeuds en panne à un instant donné dans le centre de données. **(2 points)**

*Une unité de disque sera fonctionnelle s'il y a 0 ou 1 disque en panne:  $5!/5! \times (1 - .001)^5 + 5!/4! \times .001 \times (1 - .001)^4 = 0.99500999 + .0049800 = 0.99999$ . Un noeud sera fonctionnel  $1 - 8/(365.25 \times 24) = 0.999087$ , soit pour un noeud et ses disques  $0.99999 \times 0.999087 = 0.999077$ . Sur 10 000 noeuds, il y aurait donc à tout moment en moyenne 9990.77 noeuds fonctionnels, soit 9 ou 10 noeuds en panne.*

- c) En une seconde, une application sur un processeur exécute 2 000 000 000 instructions et génère autant d'accès mémoire pour les instructions, en plus de générer 500 000 000 accès pour les données, dans le cas idéal où il n'y a aucune attente après la mémoire (pas de faute de cache). Le taux de succès en cache est de 99.9% pour les instructions et de 99.5% pour les données. La pénalité d'échec est de 80ns. Quelle est la vitesse de cette application sur ce processeur, en nombre d'instructions par seconde, en tenant compte des attentes après la mémoire? **(1 point)**

*Ce qui prendrait idéalement une seconde prendra en plus  $(2\,000\,000\,000 \times (1 - 0.999) + 500\,000\,000 \times (1 - 0.995))80ns = 0.36s$ . Ceci donne donc 2 000 000 000 instructions en 1.36s ou 1470 MIPS.*

## Question 2 (5 points)

- a) Dans le premier travail pratique, vous avez utilisé l'outil Valgrind / Callgrind. Quelle information en avez-vous tiré? Comment avez-vous pu avec cette information prédire le facteur d'accélération de votre application en multi-processeur? Est-ce que cette prédiction est plutôt une borne inférieure ou supérieure de l'accélération qui sera réellement obtenue? **(1.5 point)**

*Cet outil permet de voir la proportion du temps passé dans chaque fonction. Il peut ainsi montrer les fonctions sur lesquelles il faut se concentrer pour paralléliser ou optimiser le programme. Dans le meilleur des cas, en parallélisant une fonction sur un processeur avec  $n$  coeurs, on obtiendra un facteur d'accélération pour cette fonction de  $n$ . En connaissant la fraction du temps consacré à cette fonction et le facteur d'accélération, il est facile de calculer le gain global résultant avec la loi d'Amdahl.*

*Cette prédiction est une borne supérieure car souvent on ne réussit pas à accélérer par un facteur  $n$ , en raison de temps perdu en contention pour la mémoire cache, en synchronisation, ou en préparation du travail parallèle.*

- b) Toujours pour le premier travail pratique, vous avez utilisé LTTng afin de mieux comprendre le comportement de TBB en comparaison avec la librairie PThread. Combien de fils d'exécution sont utilisés dans chaque cas? Comment se fait la répartition du calcul entre les fils d'exécution? Quelles sont les données écrites dans la trace qui permettent ainsi d'afficher une vue temporelle de l'exécution? **(1.5 point)**

*Dans les deux cas, un fil par processeur logique est démarré pour faire le travail, soit 8. Avec PThread, la décomposition est explicite et chaque thread fait le travail demandé, qui est divisé dès le départ et espère-t-on correspond à environ  $1/8$  du travail, afin de bien équilibrer la charge. Dans le cas de TBB, il découpe le travail récursivement en 2, créant un second fil pour prendre la seconde moitié, jusqu'à avoir un fil par processeur. Rendu là, il prend une fraction du travail à faire afin de compléter l'ensemble en quelques itérations. L'avantage est qu'un fil qui termine tout son travail plus vite peut aller chercher du travail restant auprès des autres fils qui seraient rendus moins loin.*

*La trace contient tous les événements noyaux importants, en particulier tous les ordonnancements (changements de contexte) pour chaque processeur, en plus des appels systèmes (entrée et sortie) pour chaque processus. De là, il est facile de calculer quel est le processus courant sur chaque processeur en fonction du temps et même de montrer pour chaque processus s'il est en mode usager ou système.*

- c) Un serveur est responsable de fournir des images en différents formats et résolutions. Chaque requête pour une image demande en moyenne 150ms de temps de processeur, 50ms de disque et 300ms d'attente après un client lors de la communication réseau. Chaque fil d'exécution traite une requête à la fois, séquentiellement. L'unité centrale de traitement contient 8 processeurs (physiques sans hyperthread). En supposant que les disques et le réseau sont toujours disponibles, combien de fils d'exécution devrait-on avoir au minimum pour toujours occuper les 8 processeurs? Combien de disques devrait-on avoir afin de pouvoir fournir au même débit que les 8 processeurs. **(2 points)**

*Une requête complète prend  $150ms + 50ms + 300ms = 500ms$ , soit une répartition de 0.3 CPU, 0.1 disque et 0.6 réseau. Il faudra avoir des fils d'exécution en proportion avec 8 pour les 8 processeurs et donc  $8 \times 0.1/0.3 = 2.66$  donc 3 pour les disques et  $8 \times 0.6/0.3 = 16$  pour l'attente après les clients, ce qui fait un total de 27 au minimum. Il faut 3 disques puisque chacun des 3 fils réservés aux disques correspond à la charge requise pour utiliser un disque.*

## Question 3 (5 points)

- a) Votre ordinateur contient 8 processeurs. Cependant, vous ne savez pas si ce sont en fait 4 processeurs physiques avec hyperthread ou 8 processeurs physiques. De plus,

vous ne savez pas si certains processeurs partagent ou non leur cache de niveau L1. Cependant, on vous a bien mis en garde contre le fait de mettre deux tâches temps réel sur deux processeurs qui partagent la même cache L1. Sachant que vous pouvez facilement spécifier au système d'exploitation quel fil exécuter sur quel processeur, proposez une méthode (description d'un programme d'essai à exécuter) pour déterminer les processeurs physiques et logiques ainsi que les processeurs physiques qui partagent leur cache L1. **(2 points)**

*Pour déterminer si la technologie hyperthread est utilisée, on peut exécuter un petit programme qui ne fait que des calculs sur des données prenant peu d'espace, il devrait y avoir très peu de fautes de cache. En exécutant un tel programme sur un processeur puis sur deux processeurs en parallèle, la performance de chacun devrait être identique si ce sont deux processeurs physiques différents. Par contre, si les deux processeurs sont en fait virtuels et partagent le même processeur physique, la performance sera environ la moitié pour chacun.*

*Pour déterminer si deux processeurs physiques partagent la même cache L1, différentes stratégies sont possibles. Une stratégie est d'accéder en boucle un vecteur de grande taille (nécessairement plus grande que la taille d'une cache L1). Si le même programme est exécuté sur deux processeurs sans ralentissement, la cache n'est pas partagée. Par contre, si la performance est sensiblement dégradée, l'impact peut venir de la contention sur la cache L1 ou de la contention à un autre niveau du système de mémoire.*

*Une deuxième stratégie plus fiable est de premièrement mesurer la taille de la cache L1. Ceci peut se faire en accédant en boucle un vecteur et en variant la taille de ce vecteur tout en mesurant la performance pour chaque cas. La performance (nombre d'accès par seconde) devrait être assez stable jusqu'à s'approcher de la taille de la cache L1 et par la suite diminuer rapidement. En prenant la plus grande taille avant que la performance ne se dégrade, on utilise au maximum la cache L1. Ensuite, avec cette taille, on peut rouler ce programme sur deux processeurs en parallèle. Si les caches L1 sont séparées, chacun s'exécutera avec la bonne performance. Si les caches L1 sont partagées, chacun se retrouvera avec la moitié de la cache à sa disposition environ et la taille du vecteur sera alors plus grande que l'espace disponible et les deux programmes verront une performance dégradée.*

*Une troisième stratégie est d'avoir deux fils d'exécution parallèles qui lisent et écrivent en boucle une variable partagée. Si la performance est beaucoup meilleure pour une paire de processeurs que pour une autre, cette première paire partage vraisemblablement leur cache L1.*

- b) Un jeu multi-fil s'exécute sur deux processeurs à mémoire partagée. C'est un jeu classique à plusieurs joueurs, chacun ayant le choix entre plusieurs outils, le but du jeu étant de construire quelque chose. Une section de code montrée plus bas effectue la sélection d'outil sur le processeur 0 alors que les informations sur cet outil peuvent être accédées sur le processeur 1. Pour que l'exécution soit cohérente, il faut que les informations sur le nouvel outil choisi soient disponibles avant d'être accédées. Quelles sont les barrières mémoire minimales à insérer (et où) afin d'assurer un comportement correct du programme dans tous les cas? **(2 points)**

```

(initiallement outil = &marteau; marteau.debut = 0)
Processeur 0                               Processeur 1

tournevis.debut = 1;                        outil_courant = outil;
outil = &tournevis;                         debut = outil_courant->debut;

```

Ne sachant le modèle de mémoire de l'ordinateur cible, il faut insérer des barrières mémoire qui effectueront le travail requis selon l'ordinateur sur lequel le programme s'exécute. Sur le processeur 0, il faut que `tournevis.debut` soit disponible avant d'être lu (avant que `outil = &tournevis`), ce qui requiert une barrière mémoire en écriture entre les deux. Sur le processeur 1, il faut que la mise à jour de `tournevis.debut` arrive avant que le nouvel outil ne soit vu. Pour ce faire, il faut donc une barrière de lecture qui assure que si on a lu le nouvel outil, toute autre mise à jour antérieure (`tournevis.debut`) nous soit aussi parvenue. Cependant, le compilateur ou le pipeline super-scalaire ne vont pas causer de réordonnancement entre les lectures de `outil_courant` et `outil_courant->debut` car la deuxième dépend de la première. Cependant, sur de rares processeurs (DEC Alpha), des queues d'invalidation multiples parallèles pourraient causer un problème. La barrière minimale requise est donc une barrière de lecture dépendante. Une barrière de lecture donnerait une exécution cohérente mais serait un peu moins rapide.

```

(initiallement outil = &marteau; marteau.debut = 0)
Processeur 0                               Processeur 1

tournevis.debut = 1;                        outil_courant = outil;
smp_wmb();                                smp_read_barrier_depends();
outil = &tournevis;                         debut = outil_courant->debut;

```

- c) Un système de cache utilise le protocole MOESI et un maintien de la cohérence par répertoire. Expliquez quelle information doit être conservée avec cette organisation pour chaque bloc en mémoire cache? **(1 point)**

Pour chaque bloc en mémoire cache, il faut connaître l'état du bloc (M, O, E, S ou I) ainsi que la liste des copies. L'état pour chaque bloc dans chaque cache peut être représenté sur 3 bits, par exemple (valid, modified, shared). Le répertoire des blocs n'a pas à être recopié dans chaque cache. Par exemple, il peut se trouver dans la cache L3 partagée. Il contient un vecteur de bits (un bit par processeur) pour chaque bloc qui indique pour chaque processeur si ce bloc est en cache ou non, pour savoir si un message d'invalidation doit être envoyé.

## Question 4 (5 points)

- a) La fonction suivante décale la valeur de chaque pixel d'une image. La variable KEY est une constante et l'image reçue en argument n'est pas accédée par d'autres fonctions

en parallèle. Proposez plusieurs améliorations qui devraient permettre d'accélérer significativement cette fonction tout en obtenant le même résultat à la fin. Justifiez. (2 points)

```
int encode(struct image *img)
{
    int i, j, index;
    int checksum = 0;

    #pragma omp parallel for private(i,j,index)
    for (i = 0; i < img->width; i++) {
        for (j = 0; j < img->height; j++) {
            index = i + j * width;
            img->data[index] = img->data[index] + KEY;
            #pragma omp atomic
            checksum += img->data[index];
        }
    }
    return checksum;
}
```

Plusieurs optimisations sont possibles. Premièrement, la matrice de points de l'image doit être accédée par rangées pour augmenter la localité de référence. Ensuite, il est moins efficace de faire une opération atomique que d'utiliser une réduction pour checksum. Par ailleurs, il faut éviter d'accéder à répétition des variables qui ne peuvent être mises dans des registres car elles ne sont pas locales (tous les champs de `img`). Finalement, on peut simplifier légèrement le calcul de l'index. On pourrait aussi fusionner les deux boucles.

```
int encode(struct image *img)
{
    int i, j;
    char *index;
    int checksum = 0;
    int width = img->width;
    int height = img->height;
    char *data = img->data;

    #pragma omp parallel for private(i,j,index) reduction(+:checksum)
    for (i = 0; i < height; i++) {
        index = data + i * width;
        for (j = 0; j < width; j++) {
            *index += KEY;
            checksum += *index;
            index++;
        }
    }
}
```



```
    }  
  }  
  return checksum;  
}
```

- b) Quelle est la différence entre les stratégies *static*, *dynamic*, *guided* et *auto* pour partitionner les itérations des boucles et les ordonnancer sur les fils d'exécution en OpenMP? Expliquez dans quelle situation chacune peut être particulièrement intéressante. **(2 points)**

*Auto choisit automatiquement une stratégie alors que static décompose au départ, de manière prévisible, le travail. Dynamic distribue des blocs d'itérations au fur et à mesure que les fils d'exécution deviennent libres. Les blocs ont tous la même taille (sauf les derniers) mais le nombre de blocs exécutés peut varier d'un fil à l'autre. Avec guided, la taille des blocs varie, celle-ci étant proportionnelle au nombre d'itérations restantes divisé par le nombre de fils d'exécution. Dans certains cas, une partition statique permet d'utiliser l'option *nowait* entre des sections parallèles, ce qui peut améliorer la performance. Static permet aussi de réduire la variabilité, ce qui peut être utile pour le débogage. Dynamic peut être intéressant lorsqu'on veut spécifier une très petite taille de bloc d'itération, lorsque la durée de chaque itération varie énormément. Autrement, guided est généralement une meilleure stratégie. Auto laisse le système choisir, ce qui sera le meilleur choix lorsqu'on a pas de raison particulière d'insister sur le choix de l'une ou l'autre stratégie.*

- c) Quel est l'effet de la clause *nowait* sur le déroulement d'une boucle *for* en OpenMP? Est-ce que ceci peut faire une différence appréciable? Expliquez? **(1 point)**

*La clause *nowait* retire la barrière (rendez-vous) qui serait insérée à la fin de la boucle. Certains fils d'exécution termineront donc possiblement avant d'autres et tous les calculs de la boucle ne seront donc pas encore finis. Si cela n'est pas un problème, par exemple parce que la boucle suivante utilise les mêmes données avec les mêmes fils, le retrait de cette barrière enlève le coût de la barrière elle-même. Plus important encore, ce retrait enlève aussi l'attente de chaque fil jusqu'à ce que le dernier fil ait terminé. L'impact peut donc être significatif.*

Le professeur: Michel Dagenais

## **6.4 Année 2012**

**ÉCOLE POLYTECHNIQUE DE MONTREAL****Département de génie informatique et génie logiciel****Cours INF8601: Systèmes informatiques parallèles (Automne 2012)****3 crédits (3-1.5-4.5)**

---

**CORRIGÉ DU CONTRÔLE PÉRIODIQUE****DATE: Mardi le 30 octobre 2012****HEURE: 9h30 à 11h20****DUREE: 1H50****NOTE: Toute documentation permise, calculatrice non programmable permise****Ce questionnaire comprend 4 questions pour 20 points**

---

**Question 1 (5 points)**

- a) Vous avez accès à une grappe de 128 ordinateurs. Chaque ordinateur est constitué d'une partie électronique avec un taux de disponibilité de 0.999 et de deux disques redondants en miroir. Le taux de disponibilité de chaque disque est de 0.99. Vous avez conçu votre application parallèle pour décomposer le problème en 128 morceaux, faire calculer chaque morceau sur un ordinateur différent, et obtenir le résultat à la fin si tous les ordinateurs ont été opérationnels tout au long du calcul. Quelle est la probabilité que toute la grappe soit disponible à un instant donné? **(2 points)**

*Le miroir de disques sera non disponible si les deux disques sont en panne simultanément,  $(1 - 0.99)^2 = .0001$ , soit une probabilité de disponibilité pour le miroir de 0.9999. La probabilité qu'un ordinateur (disque et électronique) soit disponible est donc de  $0.999 \times 0.9999 = 0.9989$ . La probabilité que les 128 ordinateurs soient disponibles en même temps est conséquemment de  $0.9989^{128} = 0.8686$ .*

- b) Une grappe de 1024 noeuds est utilisée pour une tâche de calcul qui se termine par une réduction (somme des résultats fournis par chacun des 1024 noeuds). Les 1024 noeuds sont connectés à un commutateur performant qui permet à n'importe quelle paire de

noeuds de communiquer en même temps. Toutefois, un noeud donné ne peut envoyer et recevoir qu'un seul message par unité de temps. Proposez une organisation efficace pour effectuer l'opération de réduction. Combien d'unités de temps sont nécessaires pour envoyer les messages requis pour cette opération de réduction? **(2 points)**

*Une propagation en arbre fait le travail avec 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 noeuds, reste 1 noeud... Au premier tour, les 256 feuilles de gauche envoient leur résultat, puis les 256 de droite, ensuite de 256 vers 128, les 128 noeuds de gauche puis les 128 de droite... jusqu'à la racine qui reçoit la valeur de gauche puis celle de droite et les somme, pour un total de 18 unités de temps, auquel il faut ajouter 1 unité pour recevoir la valeur du noeud orphelin. Ceci demande 1023 messages.*

*Une propagation en papillon fonctionne différemment. Au premier cycle, les 512 noeuds impairs envoient leur valeur aux 512 noeuds pairs. Au deuxième cycle, les 256 noeuds pairs non multiples de 4 envoient leur valeur aux 256 noeuds multiples de 4. Ceci se poursuit pour 128, 64, 32, 16, 8, 4, 2 et 1 noeud, pour un total de 10 unités de temps. Cette solution est donc préférable. Ici encore, 1023 messages sont envoyés.*

- c) Vous analysez un programme à paralléliser et trouvez qu'il est constitué d'une phase a) de préparation du problème, une phase b) de calcul matriciel, et une phase c) de regroupement des résultats. En mode sériel, ces phases prennent respectivement a: 10s, b: 4000s c: 20s. En parallélisant le programme sur 128 noeuds, la phase b se retrouve raccourcie par un facteur 128 mais il faut ajouter 20s pour envoyer les données en multi-diffusion aux 128 noeuds et 1s par noeud pour récupérer les résultats. Quel est le temps de calcul sériel? En parallèle? Quel est le facteur d'accélération. **(1 point)**

*Le temps était de  $10s + 4000s + 20s = 4030s$  et devient  $10s + 20s + 4000s / 128 + 20s + 128s = 209.25$ , pour un facteur d'accélération de 19.26.*

## Question 2 (5 points)

- a) Dans votre premier travail pratique, vous avez utilisé la librairie TBB pour paralléliser le problème du dragon, et vous avez étudié comment la librairie TBB découpait le problème en intervalles et assignait les fils d'exécution. Décrivez et expliquez comment la librairie décomposait votre problème du dragon et assignait le travail à plusieurs fils d'exécution. **(2 points)**

*La librairie TBB divise récursivement en 2 le problème en ajoutant des fils d'exécution jusqu'à atteindre 8 fils (nombre de processeurs logiques). Par la suite, une fraction de l'intervalle disponible au fil est traité, de manière à laisser une portion à assigner plus tard de manière plus flexible, permettant ainsi d'équilibrer la charge.*

- b) Vous devez compiler un très gros logiciel constitué de 5000 fichiers. La compilation de chaque fichier prend 0.5s de temps de processeur et 0.4s d'attente après les serveurs de fichiers en réseau. Votre ordinateur contient 8 processeurs. Puisqu'il est facile de paralléliser une telle compilation en répartissant les fichiers entre les processeurs, combien de processus parallèles (ou fils d'exécution) de compilation devrait-on utiliser

au minimum pour accélérer le plus possible la compilation? Est-ce qu'il y a un problème à créer beaucoup plus de processus parallèles que ce minimum requis? **(2 points)**

*Pendant 1 seconde de temps CPU de compilation, il y aura  $0.4s \times 1s / .5s = 0.8s$  de temps d'attente pour le serveur de fichier en réseau. En planifiant 8 fils d'exécution pour les 8 processeurs, il suffirait d'avoir  $8 \times .8s / 1s = 6.4$ , soit 7 fils d'exécution supplémentaires pour bloquer et attendre après les accès aux fichiers. Le problème toutefois avec cela est qu'il y aurait toujours un fil disponible pour occuper un CPU mais ce fil pourrait avoir exécuté la fois précédente sur un autre CPU, forçant ainsi une migration de CPU inutile avec les fautes de cache et de TLB associées. Il serait donc préférable d'avoir  $8 + 8 = 16$  fils d'exécution de manière à tenir tous les CPU occupés et à minimiser la migration de fils d'exécution entre les CPU. Avoir beaucoup plus de fils d'exécution que nécessaire ajoute un peu de coût pour créer et ordonnancer les fils puis pour les changements de contexte. Toutefois, cela est normalement moins coûteux que l'inverse, avoir trop peu de fils et laisser des processeurs inoccupés. Pour cette raison, il est préférable d'avoir un peu trop de fils que pas assez.*

- c) Les ordinateurs du laboratoire bénéficient de la technologie hyperthread. Comment cela fonctionne-t-il et quel en est le bénéfice? Est-ce que cela affecte le choix du nombre de processus parallèles recommandés pour accélérer une tâche parallèle? **(1 point)**

*Les processeurs physiques contiennent un double ensemble de registres et apparaissent chacun comme deux processeurs logiques. Lorsqu'une faute de cache survient sur un processeur logique, l'autre processeur logique devient actif. Ceci permet donc, pour un seul processeur physique, d'accomplir plus de travail car les latences des fautes de pages ne sont pas perdues puisque deux fils (un sur chaque processeur logique) sont disponibles pour utiliser le processeur physique. Le gain de performance est de 10 à 30% environ. Pour une application parallèle, il faut donc s'assurer d'avoir assez de fils d'exécution pour occuper tous les processeurs logiques.*

### Question 3 (5 points)

- a) Le programme suivant s'exécute sur un ordinateur du laboratoire utilisé pour les travaux pratiques du cours INF8601. La variable `sums` est un vecteur d'entiers 32 bits de 64 entrées. Le temps d'exécution de ce programme (temps réel écoulé) pour `nb_iter = 1 000 000 000` est, pour `nb_thread = (1, 2, 4, 8, 16)`, de respectivement (2.57s, 3.41s, 4.60s, 3.67s, 2.85s). Comment expliquez-vous que le temps augmente puis diminue en fonction du nombre croissant de fils d'exécution parallèles? Ces ordinateurs contiennent 4 processeurs et une cache L1 avec des blocs de longueur 64 octets. **(2 points)**

```
omp_set_num_threads(nb_thread);
#pragma omp parallel for shared(sums)
for(i = 0; i < nb_iter; i++) {
    sums[omp_get_thread_num()] += i % 2 + 1;
}
```

Les blocs en cache contiennent 16 mots de 4 octets. Avec un seul processeur, il n'y a pas de faute de cache. Avec deux processeurs, les deux fils d'exécution souffrent du faux partage puisque les entrées `sums[0]` et `sums[1]` tombent dans la même ligne de cache. Le coût associé aux deux processeurs qui se battent pour avoir cette ligne de cache dépasse tout gain qu'il pourrait y avoir avec deux processeurs. A quatre fils d'exécution, le problème est exacerbé. Rendu à 8 fils d'exécution, deux fils partagent chaque processeur physique et peuvent chacun faire un tour de boucle en utilisant le même bloc en cache. Il y a donc autant de contention entre les 4 processeurs mais à chaque fois qu'un processeur a le bloc, deux fils d'exécution peuvent en profiter. Rendu à 16 fils, il y a 4 fils par processeur et donc moins d'échanges pour chaque bloc de cache et la performance s'améliore encore par rapport au pire cas à 4 processeurs.

- b) Les programmes suivants s'exécutent sur deux processeurs à mémoire partagée afin de jouer à Roche, Papier, Ciseaux. Chaque processeur choisit un chiffre entre 0 et 2, attend que le choix de l'autre processeur soit prêt et vérifie s'il a gagné, auquel cas il émet un son pour signifier qu'il a gagné. Est-ce que ce programme donnera le résultat attendu sur un ordinateur à cohérence séquentielle? Avec ordonnancement total des écritures? Avec un ordonnancement partiel des écritures? Avec un ordonnancement faible? Quelles sont les barrières mémoire à insérer (et où) afin d'assurer un comportement correct du programme dans tous les cas? **(2 points)**

(initialement `choix0 = -1; pret0 = 0, choix1 = -1; pret1 = 0`)

Processeur 0

```
choix0 = choisir(0,2);
pret0 = 1;
while(pret1 == 0);
if(gagne(choix0, choix1)
{ jouer_musique();
}
```

Processeur 1

```
choix1 = choisir(0,2);
pret1 = 1;
while(pret0 == 0);
if(gagne(choix1, choix0)
{ jouer_musique();
}
```

Avec un ordinateur à cohérence séquentielle ou même avec un ordonnancement total des écritures, le protocole fonctionnera tel qu'attendu. Il faut toutefois s'assurer que le compilateur ne réordonne pas les écritures, soit en empêchant l'optimisation, en déclarant certaines variables volatiles ou en ajoutant des barrières mémoire de compilateur. Avec un ordonnancement partiel ou faible, il peut arriver que `pret0` (ou `pret1`) soit écrit en mémoire centrale et visible à l'autre processeur avant `choix0` (ou `choix1`). Ceci briserait le protocole et empêcherait le fonctionnement correct. Ceci peut être corrigé avec une barrière d'écriture qui force `pret0` à être propagé en mémoire avant `choix0`, et une barrière de lecture qui force une synchronisation de toutes les lectures, assurant que la mise à jour de `choix0` soit nécessairement lue si `pret0` est lu à 1.

Une alternative à envisager serait de mettre une barrière mémoire complète `smp_mb` plutôt que `smp_wmb` et de laisser tomber `smp_rmb`. Dans ce cas, si la barrière sur le processeur 0 est rencontrée en premier, il transmettrait `choix0` à la mémoire centrale

et viderait sa queue de mise à jour de la cache; choix1 et pret1 pourraient ne pas avoir encore été changés. Si maintenant le processeur 1 progresse et passe sa barrière mémoire, il transmettrait choix1 à la mémoire centrale et obtiendrait la mise à jour pour choix0. Pour la suite, le processeur 1 fonctionnera correctement. Par contre, sur le processeur 0, choix1 bien que rendu en mémoire centrale peut encore être en queue de mise à jour et, par malchance, il pourrait arriver que pret1, bien qu'arrivé en mémoire centrale après choix1, arrive avant choix1 au processeur 0 en raison de queues de mises à jour multiples pouvant causer des réordonnancements. Il serait donc possible selon ce scénario que le processeur 0 obtienne un mauvais résultat, et cette alternative n'est donc pas acceptable.

(initialement choix0 = -1; pret0 = 0, choix1 = -1; pret1 = 0)

Processeur 0

```
choix0 = choisir(0,2);
smp_wmb();
pret0 = 1;
while(pret1 == 0);
smp_rmb();
if(gagne(choix0, choix1)
{ jouer_musique();
}
```

Processeur 1

```
choix1 = choisir(0,2);
smp_wmb();
pret1 = 1;
while(pret0 ==0);
smp_rmb();
if(gagne(choix1, choix0)
{ jouer_musique();
}
```

- c) Dans les systèmes de mémoire cache avec cohérence par répertoire, il faut stocker pour chaque bloc en cache la liste des processeurs qui en possèdent une copie, par exemple pour leur envoyer un message d'invalidation en cas d'écriture. Une manière naïve de réaliser cela serait d'avoir une matrice d'octets avec une rangée pour chaque bloc de mémoire centrale et une colonne pour chaque processeur. Chaque octet permet d'indiquer si un bloc se trouve sur un processeur donné. Est-ce organisé de cette manière dans les processeurs courants? Comment peut-on réaliser ce répertoire de manière à prendre beaucoup moins d'espace? **(1 point)**

*Les processeurs réservent normalement un bit plutôt qu'un octet par processeur pour indiquer la présence ou non d'un bloc. Ensuite, il suffit de stocker cette information pour les blocs en cache et non pas pour tous les blocs de mémoire centrale. C'est exactement ce que fait le i7 de Intel qui ajoute cette information à celle des étiquettes des blocs présents au niveau de sa cache L3.*

## Question 4 (5 points)

- a) Le programme suivant effectue la multiplication de deux matrices contenant des valeurs calculées par les fonctions fn1 et fn2. Le temps requis pour exécuter fn1 et fn2 varie beaucoup selon la valeur de leurs deux arguments. Ce programme ne fonctionne pas à la vitesse espérée et on fait appel à vos services d'expert. Proposez des améliorations

qui permettent d'accélérer le plus possible ce programme parallèle sans changer le résultat final dans la matrice  $c$ . **(2 points)**

```
int i, j, k;
double a[NRA][NCA], b[NCA][NCB], c[NRA][NCB];

#pragma omp parallel shared(a,b,c) private(i,j,k)
{ #pragma omp for
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= fn1(i,j);
  #pragma omp for
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= fn2(i,j);
  #pragma omp for
  for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
      c[i][j]= 0;
  #pragma omp for
  for(j=0; j<NCB; j++)
    for (i=0; i<NRA; i++)
      for (k=0; k<NCA; k++)
        c[i][j] += a[i][k] * b[k][j];
```

*Pour la dernière boucle, l'ordre des indices est mauvais pour la localité de référence. Il faut que  $j$  varie plus vite que  $i$  et  $k$  (pour faire des accès séquentiels dans le même bloc de cache dans  $c$  et  $b$ ), et que  $k$  varie plus vite que  $i$  (pour faire des accès séquentiels dans  $a$ ). Il devient alors possible d'intégrer le calcul de  $a$  à cette boucle car  $a$  est justement de dimension  $[NRA][NCA]$ . Il est facile de mettre `nowait` sur la première boucle restante, permettant en parallèle de finir le calcul de  $b$  et commencer la mise à 0 de  $c$ .*

```
int i, j, k;
double a[NRA][NCA], b[NCA][NCB], c[NRA][NCB];

#pragma omp parallel shared(a,b,c) private(i,j,k)
{ #pragma omp for nowait
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= fn2(i,j);
  #pragma omp for
  for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
```



```

    c[i][j]= 0;
#pragma omp for
for (i=0; i<NRA; i++)
    for (k=0; k<NCA; k++) {
        a[i][k]= fn1(i,k);
        for(j=0; j<NCB; j++)
            c[i][j] += a[i][k] * b[k][j];
    }

```

- b) Pour chacun des 4 outils suivants expliquez leur utilité, leur coût en performance, et un exemple de problème pour lequel il pourrait poser un diagnostic efficacement: profileur de fonction Valgrind/Callgrind, outil de vérification Valgrind/Helgrind, profileur de compteur de performance OProfile, traceur LTTng. **(2 points)**

*Valgrind/Callgrind permet de mesurer la proportion du temps passé dans chaque fonction et donc de voir laquelle a le plus besoin d'optimisation. Chaque entrée et sortie de fonction doit être instrumentée à un coût non négligeable.*

*L'outil Valgrind/Helgrind permet de vérifier l'utilisation des primitives de synchronisation dans un programme, par exemple pour détecter un ordonnancement incohérent de la prise des verrous ou une course sur une variable partagée en raison de l'oubli de protéger l'accès par la prise d'un verrou. Chaque accès en mémoire et chaque action de synchronisation doit être instrumentée impliquant un coût très important.*

*L'outil Oprofile permet d'obtenir pour un programme un profil des divers événements pour lesquels il existe des compteurs matériel. Par exemple, un profil des fautes de cache permet de voir rapidement si une section d'un programme cause beaucoup de fautes en cache par exemple en raison d'un faux partage. Oprofile procède par échantillonnage et ajoute un surcoût minime.*

*Finalement, l'outil LTTng permet de montrer l'état de chaque processus en fonction du temps. Il permet donc facilement de voir tout ce qui retarde un processus, CPU, accès disques, attente après un autre processus ou l'expiration d'un délai. Quelques événements clé du noyau, qui sont déjà instrumentés statiquement, sont typiquement activés pour un surcoût faible.*

- c) Un programme parallèle sur 64 processeurs doit trouver le chemin le plus court pour un voyageur de commerce qui doit visiter un grand nombre de villes. Chaque fil d'exécution, après avoir évalué la longueur d'un chemin, vérifie si ce chemin est plus court que le meilleur chemin trouvé jusqu'à présent. Pour ce faire, le programme peut utiliser des mutex de lecture-écriture ou utiliser la technique RCU afin de lire et éventuellement remplacer la valeur du chemin le plus court jusqu'à présent. Lequel serait le plus performant? Pourquoi? Proposez une troisième solution possiblement plus intéressante? **(1 point)**

*Dans ce cas, la valeur courante de chemin le plus court est lue très souvent mais rarement changée. La technique RCU est très appropriée pour ces cas. Toutefois, il serait aussi possible de simplement avoir une valeur de chemin le plus court par fil*

*d'exécution et de prendre le meilleur choix entre ceux trouvés par chaque fil à la toute fin. Dans ce cas, il n'y aurait plus de contention.*

Le professeur: Michel Dagenais

## Chapter 7

# MIPS - Assembleur Vectorielle

Dans ce chapitre nous allons parler de processeur vectoriel. Ce sont des unités de processing qui ne sont pas nécessairement aussi populaire qu'avant, mais sa n'enlève pas de leur efficacité/puissance. Le principe c'est d'avoir plusieurs registre vectoriels (8 registres de 64 éléments de 64 bits scalaire) et de pacter plusieurs ALU (e.g. FP Add/Subtract, FP Divide, Integer....) ayant chacune sa pipeline (un élément par cycle.

On fait un chargement et rangement à partir de la mémoire, en pipeline (un élément par cycle) avec un décalage de 1 ou plus entre les éléments ou par vecteur d'indirection.

On a aussi un registre de masque (64 bits) pour activer les opérations ou recevoir le résultat d'une comparaison. On a aussi un registre de longueur qui détermine le nombre d'éléments à traiter.

On caractérise les processeur vectoriel par:

- Une réduction du nombre d'instructions exécutées (instruction vectorielle versus boucle).
- Plusieurs instructions vectorielles peuvent s'exécuter presque en même temps par chaînage si elles sont indépendantes ou si le résultat d'une est directement pris par l'autre au cycle suivant avec le matériel adéquat.
- Possibilité de 2 ou 4 voies pour chaque ALU, opérant sur 2 ou 4 éléments en parallèle.
- Attention à nb\_elements / nb\_voies versus profondeur du pipeline.
- Goulot d'étranglement vers la mémoire?

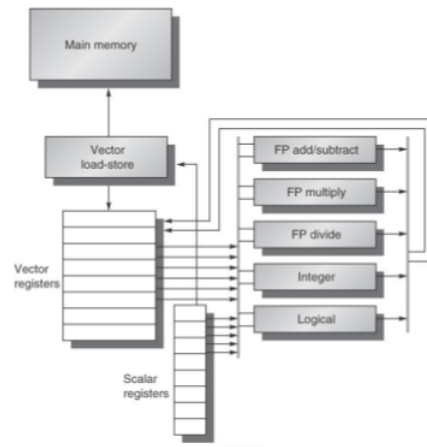


Figure 7.1: Configuration d'un processeur vectoriel

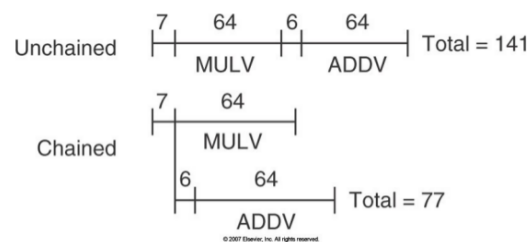


Figure 7.2: Exemple de pipeline avec amélioration sur processeur vectoriel

Voici aussi le répertoire des instructions VMIPS:

Le principe derrière vectoriser une boucle c'est de commencer par découper la boucle en morceaux de 64 éléments + un morceau de moins de 64 éléments. Ensuite on trouve les opérations vectorielles (les adds, les mult, les divide...). On fait ensuite la gestion des registres vectoriels et en remplaçant les opérations conditionnelles par des masques de contrôles des opérations vectorielles. En générale, ça fonctionne bien pour des boucles simples d'algèbres vectorielles.

Voici un exemple:

```
for (i = 0; i < n; i++) Y[i] = a * X[i] + Y[i];
// on cherche à vectoriser cette boucle
```

```
low = 0;
vl = (n % 64)
// on fait une seule itération qui fait moins de 64 et qui reste à se débarrasser
```

Instruction	Operands	Function
ADDV.D	V1, V2, V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1, V2, F0	Add F0 to each element of V2, then put each result in V1.
SUBV.D	V1, V2, V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1, V2, F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1, F0, V2	Subtract elements of V2 from F0, then put each result in V1.
MULV.D	V1, V2, V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1, V2, F0	Multiply each element of V2 by F0, then put each result in V1.
DIVV.D	V1, V2, V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1, V2, F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1, F0, V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1, R1	Load vector register V1 from memory starting at address R1.
SV	R1, V1	Store vector register V1 into memory starting at address R1.
LVWS	V1, (R1, R2)	Load V1 from address at R1 with stride in R2, i.e., $R1 + i \times R2$ .
SVWS	(R1, R2), V1	Store V1 from address at R1 with stride in R2, i.e., $R1 + i \times R2$ .
LVI	V1, (R1+V2)	Load V1 with vector whose elements are at $R1 + V2(i)$ , i.e., V2 is an index.
SVI	(R1+V2), V1	Store V1 to vector whose elements are at $R1 + V2(i)$ , i.e., V2 is an index.
CVI	V1, R1	Create an index vector by storing the values $0, 1 \times R1, 2 \times R1, \dots, 63 \times R1$ into V1.
S--V.D	V1, V2	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
S--VS.D	V1, F0	
POP	R1, VM	Count the 1s in the vector-mask register and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1	VLR, R1	Move contents of R1 to the vector-length register.
MFC1	R1, VLR	Move the contents of the vector-length register to R1.
MVTM	VM, F0	Move contents of F0 to the vector-mask register.
MVFM	F0, VM	Move contents of vector-mask register to F0.

Figure 7.3: Répertoire des instructions MIPS

```
//on itere le nombre initiales de la boucles diviser par 64, car on fait 64
//boucles a la fois. la premiere boucle s'effectue sur vl nombre d'iterations
//car on veut faire le reste en premier. Apres la premiere boucle, on fixe
//vl a 64 pour faire des pleines boucles
```

```
for(j = 0; j <= (n / 64); j++) {
    for(i = low; i < (low + vl); i++) {
        Y[i] = a * X[i] + Y[i];
    }
    //on incremente notre points de depart
    low = low + vl;
    /on place vl a 64 pour faire des pleines iterations
    vl = 64;
}
```

```
// maintenant, nous allons faire un exemple pour la gestion du control flow:
```

```
for(i = 0; i < 64; i++) {
    if(X[i] != 0) X[i] = X[i] - Y[i];
```

```

}

//on peut voir que on utilise un if pour verifier une certaines conditions ,
//pour le faire en assembleur:

//on charge nos valeurs dans nos registres V1 et V2
//on store une valeur a 0 dans F0

LV          V1, Rx
LV          V2, Ry
L.D         F0, #0

// on compare V1 a F0, NE veut dire Not Equal, le S specifie qu'on le fait
// sur un scalaire
SNEVS.D V1, F0

// On soustrait V2 a V1 (V1-V2) et on le place dans V1 (V1 = V1-V2)
SUBVV.D V1, V1, V2

// on store la valeur de V1 dans l'adresse memoire Rx
SV          V1, Rx

```

## 7.1 Instructions multimédia

On a eu un peu de vectorisation opportuniste avec certaines extensions Intel X86. on avait le MMX en 1996, c'était des registre FP64 réutilisé pour 8 opérations de 8 bits ou 4 opérations de 16 bits.

On avait aussi le SSE en 99', 01', 04', 07' qui avait des registres de 128 bits pour 16 opérations de 8 bits, 6 de 16bits, 4 de 32 bits, 2 de 64 bits.. 4 de 32 bits FP, 2 de 64 bits FP.. bref toute sorte de combinaison.

On avait finalement aussi l'AVX en 2010, ou c'était des registres de 256 bits, avec le même principe. On fait de multiple opérations de 64, 32 bits dessus (plus de 8!!).

## 7.2 Processeur Graphique

Dans la même lignée, on a eu aussi l'ère des GPUs. Ce hardware est très puissants et peu dispendieux. C'est comme des super vecteurs à rabais. Le principe est de parallélisé des section de code (grid) décomposé en bloc de fils (thread

blocs).

Chaque bloc est exécuté sur un processeur SIMD du GPU. Un processeur SIMD possède plusieurs (e.g. 32) ALU (thread processor) et gère plusieurs fils simultanément. Le processeur SIMD choisit la prochaine instruction du prochain fil prêt et l'exécute sur ses 32 ALU.

### 7.2.1 Mémoire

Dans les GPU, on a:

- Mémoire privée: pour chaque ALU (résultats intermédiaires).
- Mémoire locale: mémoire locale à chaque processeur SIMD.
- Mémoire GPU: mémoire accessible à tous les processeurs SIMD de même qu'au processeur hôte.
- Un circuit filtre les accès mémoire pour regrouper les accès à des adresses consécutives et les faire en pipeline

### 7.2.2 Instructions

Initialement, on avait les programmes en PTX (Nvidia), qui convertissait du code machine pour la carte spécifique au moment de l'exécution (pas très optimisé??).

Ces registres nous donnait accès à des opérations arithmétique: (s32, u32, f32, s64, u64, f64), transcendentales (sqrt, sin...), logiques, accès mémoire, opérations atomiques, contrôle (branch, call, ret, bar, exit).

Toutes les instructions peuvent être conditionnelles et dépendre d'un bit d'activation dans un registre prédicat. Lorsque certains ALU font une branche IF, les autres sont inactifs et sont réactivés pour le ELSE.

Une instruction vectorielle avec un élément par ALU, plutôt qu'une instruction vectorielle par ALU pendant 64 cycles en pipeline. Les nombreux fils (comme le Hyperthreading) compensent pour la latence d'accès mémoire. Ceci fait en sorte que sa utilise un Modèle de programmation assez contraignant, plus difficile à optimiser. C'est aussi beaucoup plus difficile à analyser la performance ou débogger étant donné que toute le stock se retrouve sur le GPU (pas de std-out/débug conventionel..).

Par contre, sa nous donne accès à du matériel puissant et peu dispendieux! Que nous allons discuter dans la section prochaine : **OPENCL**

## 7.3 Exercices

Voici les exercices pour la section MIPS:



## Chapitre 5 : Vecteurs et GPU

- 5.1 Une opération vectorielle très fréquente est  $Y = a * X + Y$ , appelée DAXPY en double précision. Ecrivez un programme vectoriel (VMIPS) pour effectuer ce calcul.

```
L.D      F0, a
LV       V1, Rx
MULVS.D V2, V1, F0
LV       V3, Ry
ADDVV.D V4, V2, V3
SV       V4, Ry
```

- 5.2 Vous exécutez le programme suivant sur un VMIPS qui prend un cycle par calcul dans un vecteur et peut chaîner les instructions qui n'ont pas de conflit structurel. Ce VMIPS contient une de chacune des unités matérielles suivantes, (avec la profondeur de pipeline associée), ADD-SUB (6), MUL (7), DIV(20), LOAD-STORE (12). Quel est le temps requis?

```
LV       V1, Rx
MULVS.D V2, V1, F0
LV       V3, Ry
ADDVV.D V4, V2, V3
SV       V4, Ry
```

```
LV       V1, Rx           ; 12 cycles de démarrage
MULVS.D V2, V1, F0        ; chaîné, 7 cycles
                               ; vecteur occupé, terminer 64 cycles
LV       V3, Ry           ; 12 cycles de démarrage
ADDVV.D V4, V2, V3        ; chaîné, 6 cycles
                               ; vecteur occupé, terminer 64 cycles
SV       V4, Ry           ; 12 + 64 cycles
```

12+7+64+12+6+64+12+64=241 cycles, FLOP:  $2 * 64 = 128$   
 1.88 cycle par FLOP

- 5.3 La mémoire est constituée de 8 bancs entrelacés avec chacun une latence de 12 cycles avant l'arrivée du premier mot, ou de 6 cycles après la fin du dernier accès non consécutif. Quel est le temps requis pour accéder 64 éléments consécutifs? Espacés de 32?

Les demandes d'accès sont envoyées aux 8 bancs consécutivement et les données commencent à arriver après 12 cycles, une par cycle pendant 64

cycles, un total de 76. Dans le second cas, tous les accès tombent dans le même banc mais non consécutifs à 4 mots de distance ( $32 / 8 = 4$  reste 0). Il faut donc 12 cycles + 1 pour le premier accès et les autres arrivent à 6 cycles d'intervalle:  $63 * 6$ , pour un total de 391 cycles.

5.4 Comment peut-on effectuer la réduction associée à un produit scalaire avec les opérations vectorielles du VMIPS?

On peut faire une simple boucle scalaire. Il est normalement plus rapide de faire quelques opérations vectorielles (32 + 32 éléments, 16 + 16...). Le mieux est souvent une combinaison des deux, faire la réduction vectoriellement de 64 à 32, 16 et 8, et ensuite faire une boucle scalaire.

## Chapter 8

# OpenCL

Appelé le Open Computing Language, OpenCL à été proposé en 2008 par Apple après avoir été retravaillé avec l'aide de AMD, Intel, IBM et Nvidia. C'est un standard développé et maintenant par le consortium Khronos. La première version (1.0) à été released en 2008 avec la version 2.0 en 2013 qui a ajouter du C++. une intégration Vulkan, concept de priorités, minuteriers..

C'est supporté activement par AMD, IBM, Intel et aussi NVidia sous CUDA.

l'objectifs et de permettre d'utiliser les processeurs parallèles auxiliaires, unité vectorielles (MMX), processeurs graphique (GPGPU) et autres processeur de calcule et même les processeur dans les FPGA..

Sa offre un environnement et langage normalisé pour utiliser de manière portable, ces systèmes parallèles hétérogène. C'est un standard ouvert avec implémentation de référence libre.

Le potentiel d'accélération d'OpenCL est très important.. (10 - 100 fois plus rapide !!). On le décrit comme des outils qui permettent aux programmeurs experts de faire des gains de vitesse appréciables pour les applications exigeantes. Le cout de développement est par contre important en temps et en complexité.

C'est aussi un outils de mise au point et d'analyse de performance spécifique, et moins disponible et développés.

Comme par exemple:

- Jeux
- Logiciels de CAO,
- Calculs Scientifique
- craquer des codes de chiffrement (hacker)

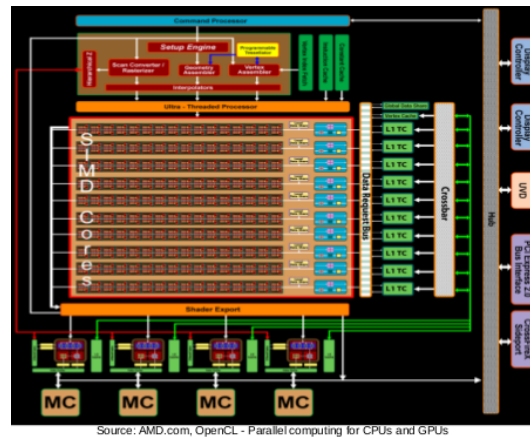


Figure 8.1: Exemple du ATI Radeon HD4870

Voici une configuration typique d'un GPU AMD. le Radeon HD4870 comporte 10 processeur SIMD de 16 processeur de fil de 5 alu, total 800 ALU. Un milliard de transistor (1.2TFlops).

le 5870 lui, à 20 processeur SIMD de 16 processeur de fil de 5 ALU, deux fois plus d'ALU au total avec deux milliards de transistor et 2.7 TFlops de puissance.

Du côté de Nvidia, on à le GTX 480 avec 1.3 TFlops ainsi que le GTX 580 avec 1.6 TFlops..

Certains concepts important reste à retenir...

- Ordinateur hôte (host), exécute le programme principal.
- Dispositif de calcul (device), par exemple une carte graphique.
- Un dispositif est constitué de processeurs SIMD (compute unit) qui peuvent prendre en charge un ou des groupes de travail (work group).
- Un processeur SIMD est constitué de plusieurs ALU (processing element), chacun pouvant prendre un ou plusieurs item de travail (work item).
- Le programme principal définit des objets en mémoire (buffer, image), des fonctions parallèles (kernel) et des événements (events). Il met en queue les fonctions parallèles à exécuter.

Le modèle ressemble à ceci:

Il faut faire certaine distinction entre les différentes mémoire de la carte graphique. La mémoire global (DRAM) est très lente comparé à la mémoire local (SRAM).

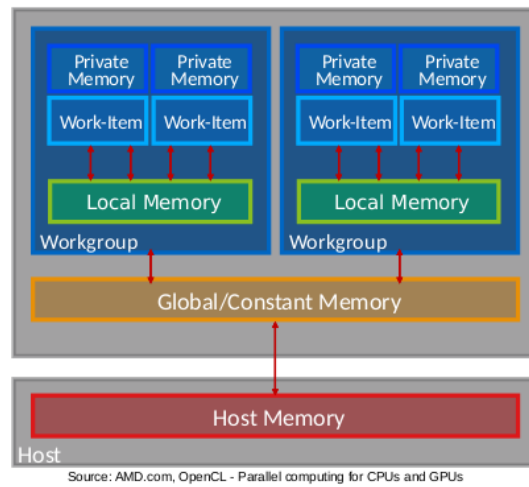


Figure 8.2: Exemple de Modèle de Mémoire OpenCL

Mémoire	Globale	Constante	Locale	Privée
Host	Allocation dynamique Lecture et écriture	Allocation dynamique Lecture et écriture	Allocation dynamique Pas d'accès	Aucun accès ni allocation
Kernel	Ne peut allouer Lecture et écriture	Allocation statique Lecture seulement	Allocation statique Lecture et écriture	Allocation statique Lecture et écriture

Figure 8.3: Modèle de Mémoire OpenCL

Une mémoire cache est insérée entre les deux. Le programme principal peut allouer des tampons (buffer) ou images (2D et 3D).

Il peut aussi copier ou les calquer entre sa mémoire et la mémoire globale du dispositif.

Par contre, Cohérence faible, des barrières mémoire sont requises entre les opérations sur des mêmes éléments de données.

### 8.0.1 Code

Voici certaines exemples de code qui nous des opérations que nous pouvons faire avec les dispositifs/kernels:

**Commande abrégés**

Voici les fonctions primaires d'OpenCL abrégés selon mes notes prise en classe...

- `oclGetPlatformID()`: sa dit la plateformes qu'on a acces
- `clGetDeviceIDs()`: sadonne les ids de devices (du CPU et GPU)
- `clGetDeviceInfo()` : sa donne l'info sur la devices
- `clCreateContext()` : sa crée un contexte, genre une queue de commande est associer a un contexte et a un device
- `clCreateCommandQueue()`: sa créer une queu de tache, associer a un device
- plusieurs commandes comme retain/release et GetInfo qui sont universelle pour la majorité des contextes openCL

\* faut faire beaucoup de gestion d'accès memoire en OpenCL \*

`CL_DEVICE_TYPE_GPU` spécifie qu'on cherche un GPU, n'importe lequel

- `createBuffer()` : pas mal straight forward
- `createSubBuffer()` : créer un sous-buffer d'un buffer
- `clenqueuReadBuffer()` : command pour lire du tampon vers la mémoire hôte, transfere GPU -> CPU
- `clenqueuWriteBuffer()` : contraire..
- image = une forme de tampon, le but est d'inter-opérer avec OpenGL

les memes commands sont disponible pour les images..

le but :

- on créer les vecteurs (notre data)
- on créer notre buffer pour nos variables en spécifiant read/write
- `clCreateProgramWithSource()` : sa lit les fichiers OpenCL pour en faire un programme. on donne la source
- `clCreateProgramWithBinary()`: meme chose, mais on donne les binaire
- une fois compilés:

- `clCreateKernel()` : sa cherche ton programme, tu donne le pointeur vers ta fonction et ensuite sa fait ton kernel
- `clCreateKernelsInProgram()` : sa cherche toute les fonctions d'un programme.
- `oclLoadProgSource()` : sa load la source? un peu bizarre et pas dans l'API

le programme compile l'autre programme, alors il faut que tu check les erreurs

- `clBuildProgram()` : sa batie ton programme avec le kernel et tout.
- `cl_kernel` : sa explique dans les notes
- `clEnqueueNDRangeKernel()` : ??
- `*cl_event` : evenements pour dire qu'on a fini\*
- `clEnqueueNativeKernel()`: soumettre une tache sur notre hote
- `cl.CreateUserEvent()` : on créer un événement, qui peut etre utiliser pour faire dépendre d'autre trucs.
- `cl.SetUserEvent()`: on change le statut
- `cl.SetEventCallback()`: associer une fonction a un callback
- `cl.EnqueueMarker()` : pour mettre un marker dans la queue
- `cl.EnqueueBarrier()`: ajouter une barriere, peu etre important pour mettre une forme de controle d'accès memoire
- `cl.get_profiling_info()`: sa donne l'information sur la queue.
- `cl.flush()`: saflush les commande
- `cl.finish()`: on termine..
- `workgroup = ?` nombre de threads? sa depend de la carte graphique, il est possible de laisser le system determiner par lui-meme le workgroup size

### Accéder au dispositifs

```
/* notre ordinateur est la plate-forme */

cl_int oclGetPlatformID (cl_platform_id *platforms)

/* liste des dispositifs disponibles , usuellement 1 GPU
   on peut specifier le type chercher avec device type */
```

```

cl_int clGetDeviceIDs (cl_platform_id platform,
                      cl_device_type device_type, cl_uint num_entries,
                      cl_device_id *devices, cl_uint *num_devices)

/* Le device info donne l'information sur le nombre de
   workgroup ainsi que le nombre de work item sur le
   dispositif ainsi que le support 64 bits, les tailles
   d'images... */

cl_int clGetDeviceInfo (cl_device_id device, cl_device_info
                      param_name, size_t param_value_size, void *param_value,
                      size_t *param_value_size_ret)

```

### Contexte et Queue

```

/* Le contexte definit tout ce qui est associer a notre
   programmation sur ce dispositif pour un travail. */

cl_context clCreateContext (const cl_context_properties
                          *properties, cl_uint num_devices, const cl_device_id
                          *devices, void (*pfn_notify)(const char *errinfo...),
                          void *user_data, cl_int *errcode_ret)

/* Une queue de commande permet d'envoyer le travail au GPU.
   La queue peut etre demander IN_ORDER ou non. On peut avoir
   plus d'une queue pour des taches independantes. */

cl_command_queue clCreateCommandQueue (cl_context context,
                                       cl_device_id device, cl_command_queue_properties
                                       properties, cl_int *errcode_ret)

```

### Exemple Complet: Device et Contexte

```

/* Initialiser le hardware, contexte et queue de commandes */
cl_int error = 0;
cl_platform_id platform;
cl_context context;
cl_command_queue queue;
cl_device_id device;

error = clGetPlatformID(&platform);
if (error != CL_SUCCESS) { ErrorExit(error); }

error = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1,
                      &device, NULL);
if (err != CL_SUCCESS) { ErrorExit(error); }

```



```
context = clCreateContext(0, 1, &device, NULL, NULL, &error);
if (error != CL_SUCCESS) { ErrorExit(error); }
```

```
queue = clCreateCommandQueue(context, device, 0, &error);
if (error != CL_SUCCESS) { ErrorExit(error); }
```

### Gestion Mémoires

Un tampon peut être créé pour lecture, écriture ou les deux.. (par les fonctions kernels).

Le tampon peut prendre la mémoire hôte spécifiée, allouer de la mémoire hôte ou allouer de la mémoire globale. La mémoire alloué peut être à partir d'une adresse hôte.

On peut définir un sous-tampon dans un tampon (inception ??)

Des commandes permettent de lire ou écrire, bloquant ou non, un tampon, sous-tampon ou section de tampon vers la mémoire hôte. ou vers un autre tampon.

Une commande permet de calquer un tampon sur la mémoire hôte.

Voici des exemples d'initialisation de Buffer

```
/* creer un tampon */

cl_mem clCreateBuffer (cl_context context, cl_mem_flags flags,
                      size_t size, void *host_ptr, cl_int *errcode_ret)

/* definir un sous-tampon */

cl_mem clCreateSubBuffer (cl_mem buffer, cl_mem_flags flags,
                          cl_buffer_create_type buffer_create_type, const void
                          *buffer_create_info, cl_int *errcode_ret)

/* commande pour lire du tampon vers la memoire hote */

cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
                            cl_mem buffer, cl_bool blocking_read, size_t offset,
                            size_t cb, void *ptr, cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list, cl_event *event)

/* commande pour ecrire de la memoire hote vers le tampon */

cl_int clEnqueueWriteBuffer (cl_command_queue command_queue,
                             cl_mem buffer, cl_bool blocking_write, size_t offset,
                             size_t cb, const void *ptr, cl_uint num_events_in_wait_list,
```

```
const cl_event *event_wait_list , cl_event *event)
```

Une image dans le contexte d'OpenCL, peut être défini comme une sorte spécifique de tampons. On peut créer une image 2d ou 3d dans un des formats supportés (clGetSupportedImageFormats) avec une certaine tailles, its par couleur/pixels..

Des fonctions nous permettent d'accéder au contenu de l'image. Il est possible d'envoyer des commandes pour lire, écrire ou copier des images, avec la mémoire hôte, une autre image ou un tampon

Il est aussi possible de calquer une image en mémoire hôte... (calquer??)

### Programme hôte versus dispositif

La plus grande partie du programme est en C ou C++ et est compilé pour l'hôte. Par contre, la parti qui s'exécute sur l'ALU, fonctions avec l'attribut kernel (ce qui va sur le dispositif), doit être compilée pour le bon dispositif à l'exécution. Elle peut être pré-compilé en langage intermédiaire pour réduire le surcout.

Le programme principale doit charger les fichiers OpenCL, les compiler et ensuite mettre en queue de commandes les fonctions kernel désirées.

Voici des exemples de commandes pour faire sa:

```
/* Lire les fichiers OpenCL pour en faire un programme */

cl_program clCreateProgramWithSource (cl_context context ,
                                     cl_uint count, const char **strings ,const size_t *lengths ,
                                     cl_int *errcode_ret)

/* Lire les fichiers OpenCL pre-compiler en IR
   pour en faire un programme */

cl_program clCreateProgramWithBinary (cl_context context ,
                                     cl_uint num_devices , const cl_device_id *device_list ,
                                     const size_t *lengths , const unsigned char **binaries ,
                                     cl_int *binary_status , cl_int *errcode_ret)

/* Compiler le programme */

cl_int clBuildProgram (cl_program program ,cl_uint num_devices ,
                      const cl_device_id *device_list , const char *options ,
                      void (CL_CALLBACK *pfn_notify)(cl_program program ,
```

```

        void *user_data), void *user_data)

/* Obtenir le point d'entry d'une fonction specifier */
cl_kernel clCreateKernel (cl_program program, const char
                        *kernel_name, cl_int *errcode_ret)

/* Obtenir toutes les fonctions */

cl_int clCreateKernelsInProgram (cl_program program,
                                cl_uint num_kernels, cl_kernel *kernels,
                                cl_uint *num_kernels_ret)

/* Specifier les arguments a associer a la fonction en vue
   de mettre en queue l'execution de cette fonction */

cl_int clSetKernelArg (cl_kernel kernel, cl_uint arg_index,
                      size_t arg_size, const void *arg_value)

```

Et maintenant, voici un exemple complet qui match avec l'exemple précédent:

```

size_t size;
const char* src = oclLoadProgSource("/tmp/abc.cl", "", &size);

cl_program pgm = clCreateProgramWithSource(context, 1, &src,
                                           &size, &error);
if(error != CLSUCCESS) { ErrorExit(error); }

error = clBuildProgram(pgm, 1, &device, NULL, NULL, NULL);
if(error != CLSUCCESS) { ErrorExit(error); }

char* bld_info;
clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
                      0, NULL, &size);

bld_info = new char[size+1];
bld_info[size] = '\0';
clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
                      size, bld_info, NULL);

cl_kernel abc_kernel=clCreateKernel(pgm," abc",&error);
if(error != CLSUCCESS) { ErrorExit(error); }

```

### Exécution de fonctions OpenCL

On peut opérer sur des vecteurs pour tirer parti des nombreux ALU, différentes tâches peuvent aller sur les différents processeur et plusieurs tâches peuvent

être assignées au même processeur en hyper-threading.

Exécuter une même fonction sur un grand nombre d'ALU dans plusieurs processeurs. Le travail peut être décrit sur 1, 2 ou 3 dimensions avec pour chaque dimensions un taille global et une taille locale. (eg. 1024X1024 versus 128X128)

Voici des exemples:

```
/* Mettre en queue sur global size par groupe de local size */

cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
                                cl_kernel kernel, cl_uint work_dim, const size_t
                                *global_work_offset, const size_t *global_work_size,
                                const size_t *local_work_size, cl_uint
                                num_events_in_wait_list, const cl_event *event_wait_list,
                                cl_event *event)

/* Mettre en queue pour execution sur un processeur */

cl_int clEnqueueTask (cl_command_queue command_queue,
                      cl_kernel kernel, cl_uint num_events_in_wait_list,
                      const cl_event *event_wait_list, cl_event *event)

/* Mettre en queue sur un processeur de type hôte */

cl_int clEnqueueNativeKernel (cl_command_queue command_queue,
                              void (*user_func)(void *) void *args, size_t cb_args,
                              cl_uint num_mem_objects, const cl_mem *mem_list,
                              const void **args_mem_loc, cl_uint
                              num_events_in_wait_list, const cl_event *event_wait_list,
                              cl_event *event)
```

### Synchronisation par événements

Le programme hôte peut créer des événements et changer leur status pour contrôler quand certaines commande pourront commencer. Il peut par exemple attendre après certains événement, ou il peut demander une fonction de rappel lors du changement d'état d'un événement. La fonction de rappel est très limitée dans ce qu'elle peut appeler.

Encore une fois, voici des exemples:

```
/* Créer un evenement sur lequel faire dependre */

cl_event clCreateUserEvent (cl_context context, cl_int
                           *errcode_ret)
```

```

/* Changer le statut a pret ou a erreur */

cl_int clSetUserEventStatus (cl_event event, cl_int
                             execution_status)

/* Attendre apres des evenements */

cl_int clWaitForEvents (cl_uint num_events, const cl_event
                       *event_list)

/* Associer une fonction de rappel a un evenement */

cl_int clSetEventCallback (cl_event event, cl_int
                           command_exec_callback_type, void (CL_CALLBACK
                       *pfn_event_notify)(cl_event event, cl_int
                           event_command_exec_status, void *user_data),
                           void *user_data)

```

### Barrières

Les commandes dans une queue IN\_ORDER sont sérialisées et les résultats de la commande précédente sont disponible pour la suivante.

Les différentes commandes d'une queue OUT\_OF\_ORDER ou les commandes de queues distincte ne sont pas synchronisé. on peut mettre des marqueur pour savoir lorsque les commande précédentes d'une queue sont terminées.

On ajoute donc des barrières pour assurer que toutes les commande précédentes sont faite avant de commencer les suivantes.

Commande d'attente d'événements peut être mise en queue..

voici des exemples de barrières:

```

/* Indique que toutes les commandes anterieures sont
   terminees */

cl_int clEnqueueMarker (cl_command_queue command_queue,
                       cl_event *event)

/* Assure que toutes les commandes anterieures sont terminees
   avant que les commandes posterieures ne commencent */

cl_int clEnqueueBarrier (cl_command_queue command_queue)

```

```

/* Attend apres les evenements specifiés avant de
   poursuivre */

cl_int clEnqueueWaitForEvents (cl_command_queue command_queue,
                              cl_uint num_events, const cl_event *event_list)

```

### Divers

On a aussi divers autre fonction OpenCL qui peuvent servir à plein d'autre chose..

l'événement de fin d'une commande peut contenir de l'info sur l'exécution: temps de mise en queue, soumission, début et fin.. Ceci permet de comprendre la performance de l'appliaiton.

Lorsqu'une fonction de rappel met une commande en queue, celle-ci n'est pas soumise à ce moment. Il faut faire un flush sur la queue.

On peut attendre pour la fin de l'exécution de toutes les commandes dans une queue (Finish).

Voici des exemples:

```

/* Extraire l'information sur le temps d'execution d'un
   evenement */

cl_int clGetEventProfilingInfo (cl_event event,
                              cl_profiling_info param_name, size_t param_value_size,
                              void *param_value, size_t *param_value_size_ret)

/* Activer la soumission des commandes de la queue */

cl_int clFlush (cl_command_queue command_queue)

/* Attendre que toutes les commandes en queue soient
   terminees */

cl_int clFinish (cl_command_queue command_queue)

```

Voici des exemples complet d'exécutions (qui continue depuis la dernière exemple complet):

```

error = clSetKernelArg(abc_kernel, 0, sizeof(cl_mem),
                      &buffer_a);
error |= clSetKernelArg(abc_kernel, 1, sizeof(cl_mem),
                      &buffer_b);
error |= clSetKernelArg(abc_kernel, 2, sizeof(cl_mem),

```

```

        &buffer_c);
error |= clSetKernelArg(abc_kernel, 3, sizeof(size_t), &size);

if(error != CL_SUCCESS) { ErrorExit(error); }

const size_t wg_size = 512;

const size_t total_size = ((1000000 / 512) + 1) * 512;

error = clEnqueueNDRangeKernel(queue, abc_kernel, 1, NULL,
                                &total_size, &wg_size, 0, NULL, NULL);

if(error != CL_SUCCESS) { ErrorExit(error); }

clEnqueueReadBuffer(queue, buffer_c, CL_TRUE, 0, buffer_size,
                    c, 0, NULL, NULL);

```

Suivit d'un exemple complet de finalisation:

```

delete [] a;
delete [] b;
delete [] c;
clReleaseKernel(abc_kernel);
clReleaseCommandQueue(queue);
clReleaseContext(context);
clReleaseMemObject(buffer_a);
clReleaseMemObject(buffer_b);
clReleaseMemObject(buffer_c);

```

### 8.0.2 Fonctions Kernels - Sur Dispositif

Les kernels OpenCL sont fait avec un sous-ensemble du C99 (très vieux). Avec des extensions spécifiques dans un fichier .cl

On utilise des types scalaires usuels, des bool, des char, short, int, long, signés ou non.. float, half, double, size\_t, ptr\_diff, intptr\_t.

Même types vectoriels (de 2,3,4,6,8,16 éléments)

Par défaut, les composantes d'un vecteur de 4 sont x,y,z et w (quaternion??).

Exemples:

```

float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
pos.xw = (float2)(5.0f, 6.0f);
pos.xyz = (float3)(3.0f, 5.0f, 9.0f);

```

```

a.xyzw = f.s0123;

float4 vf;
float2 low = vf.lo;
float2 high = vf.hi;
float2 even = vf.even;
float2 odd = vf.odd;

uchar4 u;
int4 c = convert_int4(u);

float f;
int i = convert_int(f);

union{ float f; uint u; double d;} u;

```

### Les opérateurs mathématique

Les opérateurs mathématiques sont usuel, relations logiques et comparaisons sur les scalaires, scalaires-vecteurs et vecteurs-vecteurs.

La plupart des fonctions usuelles, incluant les fonction trigo sont ofertes et fonctionnent avec des vecteurs.

Fonctions pour lire ou écrire un vecteur à partir d'un pointeur.

Voici des exemples encore une fois:

```

float4 u, v;
float f;

v = u + f;

u = atan2(v);

float distance(floatn p0, floatn p1)

float length(floatn p)

float8 vload8(size_t offset, const __local float *p)

void vstore8(float8 data, size_t offset, __local float *p)

```

**Attributs** Variables dans 4 espaces distinct: `__global`, `__local` (partagé dans le workgroup), `__constant` (lecture seulement), `__private` (par défaut, pour le



workitem).

Les types `image2d_t` et `image3d_t` sont des objets `__global`.

Arguments `read_only` ou `write_only` pour les fonctions kernels.

Peut spécifier l'alignement, `__attribute__((aligned(8)))`

```
__global float4 *color;
typedef struct {
    float a[3];
    int
    b[2];
} foo_t;

__global foo_t *my_info;
__kernel void my_func(...)
{
    __local float a;
    __local float b[10];
    float c;

    ...
}
```

### Synchronisation

On peut utiliser des barrières pour assurer la synchronisation des différents points dans le processus. on utiliser les méthode suivantes pour insérer différentes barrière dans notre programme:

- `void barrier (cl_mem_fence_flags flags)`: synchroniser tous les workItem du workGroup.
- `void mem_fence (cl_mem_fence_flags flags)`
- `void read_mem_fence (cl_mem_fence_flags flags)`
- `void write_mem_fence (cl_mem_fence_flags flags)`
- Tous les exemple de fence sont des barrière mémoires pour les lectures ou écriture en mémoire du workitem.

On peut faire aussi une copie asynchrone entre la mémoire locale et globale. Un peut comme un événement est associé à chaque opération et peut être utilisé par `wait_group_events`.

Opérations atomique (add, sub, xchg, inc, dec, cmpxchg, min, max, and, or, xor) sur des entiers ou float en mémoire local ou global

Lecture ou écriture d'un pixel d'une image 2d ou 3d.

Différentes modes d'accès (sampler\_t) pour les images.

En voici quelques exemples:

```
/* Simple addition vectorielle c = a + b */

__kernel void abc_kernel (__global const float* buffer_a ,
__global const float* buffer_b , __global float* buffer_c ,
const int nb)
{
    /* Nous sommes en une seule dimension , l'index global nous
       indique l'element sur lequel travailler */

    const int id = get_global_id(0);

    /* Nous avons quelques work_item de plus que la var taille
       il faut donc s'assurer de ne pas dépasser. */

    if (id < nb) buffer_c[id] = buffer_a[id] + buffer_b[id];
}
```

## 8.1 Optimisation

l'optimisation OpenCL se fait en :

- Minimiser les interactions entre l'hôte et le GPU.
- Regrouper les accès mémoire consécutifs en lignes complètes alignées.
- Avoir un grand nombre de fils possibles sur chaque processeur SIMD afin d'avoir toujours quelque chose à faire lors des attentes pour la mémoire et le décodage des instructions.
- Utiliser le matériel au maximum en tenant compte de la capacité (mémoire locale, registres) des processeurs SIMD
- Utiliser les instructions vectorielles (au moins jusqu'à taille 4).

En voici un exemple d'optimisation d'accès mémoire:

```
/* c[m][n] , OpenCL organiser en m*n */
```

```

int x = get_global_id(0);
int y = get_global_id(1);

c[x][y] = a[x] * b[y];

/* OpenCL organiser en n*m */

int x = get_global_id(1);
int y = get_global_id(0);

c[x][y] = a[x] * b[y];

/* get_global_id(0) varie le plus vite, il faut donc le mettre
   pour y, de maniere a faire des acces a des cases memoire
   consecutives en meme temps qui pourront etre regroupes.
   Peut etre plusieurs fois plus rapide! */

```

**Maximisation du nombre d'items** Le but est de déterminer le nombre de registre et la quantité de mémoire local requise par un workitem (par test?).

Selon la quantité de registre et de mémoire disponible sur un SIMD, cela détermine la taille du WorkGroup.

De cette manière, on maximise le nombre de fils possible sur chaque processeur SIMD, en permettant de maintenir le matériel toujours occupé.

On suppose que la taille total divisé par la taille de Work Group est largement supérieur au nombre de SIMD.

**Vectorisation** L'augmentation du nombre de Work Item est plus fiable que l'utilisation d'opérations vectorielles pour augmenter la performance.

Une instruction vectorielle fait des accès mémoire regroupés et demande une seule instruction.

Sur processeur Intel ou sur GPU AMD, les vecteurs float4 sont traités efficacement car certains éléments travaillent sur 4 mots à la fois.

## 8.2 Exemple TP2

À FAIRE...

```

int create_buffer(int width, int height)
{

```

```

/*
 * TODO: initialiser la memoire requise avec clCreateBuffer()
 */

cl_int ret = 0;
output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, width*height*sizeof(unsigned
ERROR_THROW(CL_SUCCESS, ret, "clCreateBuffer failed");
sinoscope = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(sinoscope_t), NULL
ERROR_THROW(CL_SUCCESS, ret, "clCreateBuffer failed");

//goto error;
done:
    return ret;
error:
    ret = -1;
    goto done;
}

int opencl_init(int width, int height)
{
    cl_int err;
    char *code = NULL;
    size_t length = 0;

    get_opencl_queue();
    if (queue == NULL)
        return -1;
    load_kernel_code(&code, &length);
    if (code == NULL)
        return -1;

    /*
     * Initialisation du programme
     */
    prog = clCreateProgramWithSource(context, 1, (const char **) &code, &length,
ERROR_THROW(CL_SUCCESS, err, "clCreateProgramWithSource failed");
    err = clBuildProgram(prog, 0, NULL, "-cl-fast-relaxed-math", NULL, NULL);
ERROR_THROW(CL_SUCCESS, err, "clBuildProgram failed");
    kernel = clCreateKernel(prog, "sinoscope_kernel", &err);
ERROR_THROW(CL_SUCCESS, err, "clCreateKernel failed");
    err = create_buffer(width, height);
ERROR_THROW(CL_SUCCESS, err, "create_buffer failed");

    free(code);
    return 0;
error:

```

```

        return -1;
    }

void opencl_shutdown()
{
    if (queue)    clReleaseCommandQueue(queue);
    if (context)    clReleaseContext(context);

    /*
     * TODO: liberer les ressources allouees
     */

    if (output) clReleaseMemObject(output);
    if (sinoscope) clReleaseMemObject(sinoscope);
    if (kernel) clReleaseKernel(kernel);
    if (prog) clReleaseProgram(prog);
}

int sinoscope_image_opencl(sinoscope_t *ptr)
{
    cl_int ret = 0;
    cl_event ev;
    if (ptr == NULL)
        goto error;

    size_t work_dim[2];
    work_dim[0] = ptr->width;
    work_dim[1] = ptr->height;

    ret = clEnqueueWriteBuffer(queue, sinoscope, CL_TRUE, 0, sizeof(sinoscope_t), ptr,
ERR_THROW(CL_SUCCESS, ret, "clEnqueueWriteBuffer failed");
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), &output);
ERR_THROW(CL_SUCCESS, ret, "clSetKernelArg output failed");
    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), &sinoscope);
ERR_THROW(CL_SUCCESS, ret, "clSetKernelArg sinoscope failed");
    ret = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, work_dim, NULL, 0, NULL, NULL);
ERR_THROW(CL_SUCCESS, ret, "clEnqueueNDRangeKernel failed");
    ret = clFinish(queue);
ERR_THROW(CL_SUCCESS, ret, "clFinish failed");
    ret = clEnqueueReadBuffer(queue, output, CL_TRUE, 0, ptr->buf_size, ptr->buf, 0, NULL,
ERR_THROW(CL_SUCCESS, ret, "clEnqueueReadBuffer failed");

done:
    return ret;
}

```

```
error:
    ret = -1;
    goto done;
}
```

### 8.3 Exercices

Voici la feuille d'exercice:

## Chapitre 5 : OpenCL

- 5.1 Comment faire la transposition des éléments d'une matrice  $b[y][x] = a[x][y]$  tout en évitant les accès mémoire non consécutifs?

On peut copier la matrice de la mémoire globale vers la mémoire locale en variant le second indice en premier, transposer la matrice par une fonction kernel avec les arguments en mémoire locale, et recopier le résultat vers la mémoire globale en variant encore le second indice en premier. Dans certains tests, cette manière de faire est environ 10 fois plus rapide que de transposer à partir de la mémoire globale.

- 5.2 Suggérez différentes optimisations pour la fonction OpenCL suivante qui effectue la multiplication matricielle  $Y=AX$  où  $X$  et  $Y$  sont des vecteurs et  $A$  est une matrice formée uniquement d'éléments non nuls dans certaines diagonales. Le nombre de ces diagonales est `diags` et la position de départ (dans la première rangée) de ces diagonales est fournie dans `offsets`. La position est  $-(n-1)$  pour la diagonale qui ne contient que l'élément de première colonne dernière rangée, 0 pour la diagonale qui va de  $(0,0)$  à  $(n-1,n-1)$ , et  $(n-1)$  pour la diagonale qui ne contient que le dernier élément de la première rangée.

La matrice  $A$  est stockée de manière compacte, une rangée par diagonale non nulle. Pour accéder les éléments de la rangée 0, il faut prendre l'élément 0 de chaque diagonale, donc les éléments de la colonne 0 dans la matrice compacte contenant une diagonale par rangée.

Un work item est le calcul d'un élément dans le vecteur de résultat.

```
__kernel
void dia_spmv(__global float *A, __const int rows,
              __const int diags, __global int *offsets,
              __global float *x, __global float *y) {
    int row = get_global_id(0);
    float accumulator = 0;
    for(int diag = 0; diag < diags; diag++) {
        int col = row + offsets[diag];
        if ((col >= 0) && (col < rows)) {
            float m = A[diag*rows + row];
            float v = x[col];
            accumulator += m * v;
        }
    }
    y[row] = accumulator;
}
```

```
}
```

En premier lieu, le calcul de l'indice de A est simplifié et chaque rangée de A est alignée à l'aide d'espace ajouté à la fin de chaque rangée. Le nouveau paramètre `pitch_A` tient compte de l'espace ajouté. Ceci procure un gain d'environ 10%.

```
__kernel
```

```
void dia_spmv(__global float *A, __const int pitch_A,
              __const int rows,
              __const int diags, __global int *offsets,
              __global float *x, __global float *y) {
    int row = get_global_id(0);
    float accumulator = 0;
    __global float* matrix_offset = A + row;
    for(int diag = 0; diag < diags; diag++) {
        int col = row + offsets[diag];
        if ((col >= 0) && (col < rows)) {
            float m = *matrix_offset;
            float v = x[col];
            accumulator += m * v;
        }
        matrix_offset += pitch_A;
    }
    y[row] = accumulator;
}
```

Les données dans `offsets` sont lues au complet dans chaque work item à partir de la mémoire globale. Il est avantageux de copier ces données en mémoire locale et de les réutiliser par tous les work item dans le même groupe. La performance est augmentée de plus de 50%

```
__kernel
```

```
void dia_spmv(__global float *A, __const int pitch_A,
              __const int rows,
              __const int diags, __global int *offsets,
              __global float *x, __global float *y) {
    int local_id = get_local_id(0);
    int offset_id = local_id;
    while ((offset_id < 256) && (offset_id < diags)) {
        l_offsets[offset_id] = offsets[offset_id];
        offset_id = offset_id + get_local_size(0);
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    int row = (int)get_global_id(0);
```



```

float accumulator = 0;
__global float* matrix_offset = A + row;
for(int diag = 0; diag < diags; diag++) {
    int col = row + l_offsets[diag];
    if ((col >= 0) && (col < rows)) {
        float m = *matrix_offset;
        float v = x[col];
        accumulator += m * v;
    }
    matrix_offset += pitch_A;
}
y[row] = accumulator;
}

```

Finalement, des opérations vectorielles sont utilisées, ce qui procure un gain additionnel d'environ 15%.

```

__kernel
void dia_spmv(__global float *A, __const int pitch_A,
              __const int rows,
              __const int diags, __global int *offsets,
              __global float *x, __global float *y) {
    __local int l_offsets[256];
    int local_id = get_local_id(0);
    int offset_id = local_id;
    while ((offset_id < 256) && (offset_id < diags)) {
        l_offsets[offset_id] = offsets[offset_id];
        offset_id = offset_id + get_local_size(0);
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    int row = get_global_id(0) * 4;
    float4 accumulator = 0;
    __global float* matrix_offset = A + row;
    for(int diag = 0; diag < diags; diag++) {
        int col = row + l_offsets[diag];
        float4 m = vload4(0, matrix_offset);
        float4 v;
        if ((col >= 0) && (col < rows - 4)) {
            v = vload4(0, x + col);
        } else {
            int4 id = col + (int4)(0, 1, 2, 3);
            int4 in_bounds = (id >= 0) && (id < rows);
            v.x = in_bounds.x ? x[id.x] : 0;
            v.y = in_bounds.y ? x[id.y] : 0;
            v.z = in_bounds.z ? x[id.z] : 0;
        }
        accumulator += m * v;
    }
    y[row] = accumulator;
}

```

```
        v.w = in_bounds.w ? x[id.w] : 0;
    }
    accumulator += m * v;
    matrix_offset += pitch_A;
}
vstore4(accumulator, 0, row + y);
}
```

## Chapter 9

# MPI

discuté lors d'un workshop en 1992, MPI est l'approche classique du calcul parallèle distribués. La première version a été émise en 1993, la version finale en 1994. C'est un API de programmation parallèle sur multi-ordinateur (plusieurs noeuds avec plusieurs processeur). C'est une librairie qui sert à gérer les communication entre les différents noeuds afin qu'ils puissent s'envoyer des message et puissent travailler sur un problème de manière efficace. Sa bénéficie de l'expérience avec les systèmes antérieures (PVM, LINDA).

C'est un ensemble cohérent de fonctions permettant une interface simple tout en s'assurant que l'implémentation sous-jacente puisse être efficace.

Sa donne aussi une interopérabilité avec OpenMP et OpenCL. Par contre, la tolérance aux pannes est très faible..

Le public cible de la librairies sont toutes les grandes grappes de calcul, les laboratoires gouvernementaux (modélisation météo/nucléaire) ou industriel (analyse de réseaux de transmission chez l'hydro) et aussi modélisation par éléments finis chez bombardier ou Andritz.

Des version optimisées et outils sont offert par les grands fournisseurs de matériel. (Intel, IBM..)

Exemple d'un programme:

```
#include "mpi.h"
#include <stdio.h>
int main( argc , argv )
int argc;
char **argv;
{
MPI_Init( &argc , &argv );
```

```

printf( "Hello world\n" );
MPI_Finalize();
return 0;
}
mpicc --mpilog --mpitrace -o hello hello.c
mpirun -np 2 hello

```

Le fonctionnement de base est :

1. d'initialiser la librairie
2. Utilisation du communicateur par défaut ou création du communicateur
3. Demander le nombre d'éléments N et sont ran (0 à N-1)
4. avec mpirun, le mee programme s'exécute sur N noeuds mais chaque instance fait un travail différent en se basant sur son rang qui la différencie.

## 9.1 Message et Types

La librairie fonctionne en échangeant des messages selon le rang, comme par exemple des adresse, types d'éléments et nombre, plutôt que adresse et nombre d'octets..

Types sont définis récursivement pour représenter n'importe qu'elle structure de donnée.

Permet de spécifier les données telles qu'elle sont en minimisant la possibilité d'erreur sur la longueur.

Sa permet au système de faire les conversion si requis (e.g. petit ou gros boutien (endian))

voici les types d'MPI:

- Primitifs: `MPI_CHAR`, `MPI_UNSIGNED_CHAR` (and `SHORT`, `INT`, `LONG`), `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG_DOUBLE`..
- Séquences: `MPI_TYPE_CONTIGUOUS`, `MPI_TYPE_VECTOR`, `MPI_TYPE_HVECTOR`, `MPI_TYPE_INDEXED`, `MPI_TYPE_HINDEXED`.
- Agrégation: `MPI_TYPE_CREATE_STRUCT`
- Calcul de la position: `MPI_TYPE_COMMIT`
- Divers: `MPI_TYPE_FREE`, `MPI_GET_ADRESS`, `MPI_TYPE_CREATE_SUBARRAY`, `MPI_TYPE_CREATE_DARRAY`, `MPI_PACK`, `MPI_UNPACK`

et en faisant la description de types:

```

struct Element {
    unsigned temperature;
    double force[3];
    char status;
}
MPI_Datatype ElementType;
MPI_Datatype ElementFieldTypes[3] =
    {MPI_UNSIGNED, MPI_DOUBLE, MPI_CHAR};

int ElementFieldLength[3] = {1, 3, 1};

MPI_Aint ElementFieldPosition[3] =
    {0, sizeof(double), 4 * sizeof(double)};

MPI_Type_create_struct(3, ElementFieldLength,
    ElementFieldPosition, ElementFieldTypes, &ElementType);

MPI_Type_commit(&ElementType);

```

### Message

En ce qui concerne les messages..

on les envoie avec les fonctions suivantes:

- Envoi bloquant:
  - `MPI_SEND(buf, count, datatype, dest, tag, comm)` bloque sur l'envoi d'un message à dest.
  - `MPI_RECV(buf, count, datatype, source, tag, comm, status)` bloque sur l'attente d'un message de source avec tag ou `MPI_ANY_SOURCE`, `MPI_ANY_TAG`.
  - `MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)` aller-retour comme pour un appel de procédure.
  - `MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)` échange de données.
- Envoi non bloquants:
  - `MPI_SEND(buf, count, datatype, dest, tag, comm, request)` l'envoi est demandé, buf ne doit plus être accédé jusqu'à ce que l'envoi soit terminé.

- `MPI_RECV(buf, count, datatype, source, tag, comm, request)` la réception est demandée, `buf` ne peut être accédé avant que la réception ne soit terminée.
- `MPI_WAIT(request, status)`, `MPI_TEST(request, flag, status)` ou `MPI_REQUEST_GET_STATUS(request, flag, status)` et `MPI_REQUEST_FREE(request)` permettent de savoir lorsque la requête est terminée.
- Attente Multiple:
  - `MPI_WAITANY(count, array_of_request, index, status)`, `MPI_TESTANY(count, array_of_request, index, flag, status)` permet d'attendre après une de plusieurs requêtes.
  - `MPI_WAITALL(count, array_of_request, array_of_status)`, `MPI_TESTALL(count, array_of_request, flag, array_of_status)` permet de vérifier toutes les requêtes.
  - `MPI_WAITSOME(incount, array_of_request, outcount, array_of_index, array_of_status)`, `MPI_TESTSOME(count, array_of_request, array_of_index, array_of_status)` permet de vérifier une ou plusieurs requêtes.
- Autres fonctions sur les requêtes:
  - `MPI_PROBE(source, tag, comm, status)`, `MPI_IPROBE(source, tag, comm, flag, status)`, attend ou vérifie si un message est disponible.
  - `MPI_CANCEL(request)` annule une opération, `MPI_TEST_CANCELLED(status, flag)` permet de le vérifier.
- Communications pré-définis:
  - `MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)`, `MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)`, spécifie les arguments pour un envoi/réception à venir, et à répéter.
  - `MPI_START(request)`, `MPI_STARTALL(count, array_of_request)`, démarre la requête.
- Autres types d'envoi:
  - `MPI_BSEND`: buffered. Le contenu est copié avant d'être envoyé.
  - `MPI_SSEND`: synchronous. Ne peut se compléter avant que le receveur ait commencé à recevoir le message.
  - `MPI_RSEND`: ready, le receveur doit déjà être en attente du message. Ceci peut permettre de faire l'envoi en étant assuré qu'un tampon est prêt à l'autre bout.

- MPI\_IBSEND, MPI\_ISSEND, MPI\_IRSEND, MPI\_BSEND\_INIT, MPI\_SSEND\_INIT, MPI\_RSEND\_INIT.

- Gestion manuelle des Tampons:

- MPI\_BUFFER\_ATTACH(buffer, size) spécifier manuellement l'espace à utiliser pour les tampons et conséquemment sa taille.
- MPI\_BUFFER\_DETACH(buffer, size).

Voici un exemple de multiplication de matrice:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define RA 62
#define CA 15
#define CB 7
#define ROOT 0
#define ROOT_A_FEUILLE 1
#define FEUILLE_A_ROOT 2

int main (int argc, char *argv []) {
    int size, rank, src, dest, mtype;
    int rangees, moyenne, extra, offset, i, j, k, rc;
    double a[RA][CA], b[CA][CB], c[RA][CB];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPLCOMM_WORLD,&rank);
    MPI_Comm_size(MPLCOMM_WORLD,&size);

    if (rank == MASTER){
        moyenne = RA / (size - 1); extra = RA % (size - 1);
        offset = 0; mtype = ROOT_A_FEUILLE;
        for (dest=1; dest<=(size - 1); dest++){
            rangees = (dest <= extra) ? moyenne+1 : moyenne;
            MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPLCOMM_WORLD);
            MPI_Send(&rangees, 1, MPI_INT, dest, mtype, MPLCOMM_WORLD);
            MPI_Send(&a[offset][0], rangees*CA, MPLDOUBLE, dest,
            mtype, MPLCOMM_WORLD);

            MPI_Send(&b, CA*CB, MPLDOUBLE, dest, mtype, MPLCOMM_WORLD);
            offset = offset + rangees;
        }
        mtype = FEUILLE_A_ROOT;
```

```

        for (i=1; i<=(size - 1); i++){
            src = i;
MPI_Recv(&offset ,1 ,MPI_INT ,src ,mtype ,MPI_COMM_WORLD,&status );
MPI_Recv(&rangees ,1 ,MPI_INT ,src ,mtype ,MPI_COMM_WORLD,&status );
MPI_Recv(&c[ offset ][0] ,rangees*CB,MPLDOUBLE,src ,mtype ,
MPI_COMM_WORLD,&status );
        }
    }

    if (rank > MASTER){
        mtype = ROOT_A.FEUILLE;
MPI_Recv(&offset ,1 ,MPI_INT ,ROOT,mtype ,MPI_COMM_WORLD,&status );
MPI_Recv(&rangees ,1 ,MPI_INT ,ROOT,mtype ,MPI_COMM_WORLD,&status );
MPI_Recv(&a ,rangees*CA,MPLDOUBLE,ROOT,mtype ,MPI_COMM_WORLD,
&status );
MPI_Recv(&b ,CA*CB,MPLDOUBLE,ROOT,mtype ,MPI_COMM_WORLD,&status );
        for (k=0; k<CB; k++){
            for (i=0; i<rangees; i++){
                c[i][k] = 0.0;
                for (j=0; j<CA; j++)
                    c[i][k]=c[i][k]+a[i][j]*b[j][k];
            }
            mtype = FEUILLE_A.ROOT;
MPI_Send(&offset ,1 ,MPI_INT ,ROOT,mtype ,MPI_COMM_WORLD);
MPI_Send(&rangees ,1 ,MPI_INT ,ROOT,mtype ,MPI_COMM_WORLD);
MPI_Send(&c ,rangees*CB,MPLDOUBLE,MASTER,mtype ,MPI_COMM_WORLD);
        }
        MPI_Finalize ();
    }
}

```

### 9.1.1 Communications Globales

Plusieurs moyens sont disponible pour faire de la communication globales sans avoir à gérer toutes les messages, en voici quelques une, de façon imagier:

voici les différentes fonctions pour pouvoir effectuer les communications globales vu en haut.. les voici:

- Communications globales de base:
  - MPI\_BARRIER(comm) bloque jusqu'à ce que tous les membres du groupe aient appelé cette fonction.
  - MPI\_BCAST(buffer, count, datatype, root, comm) est appelé par tous et le contenu de buffer sur le processus de rang root est copié à



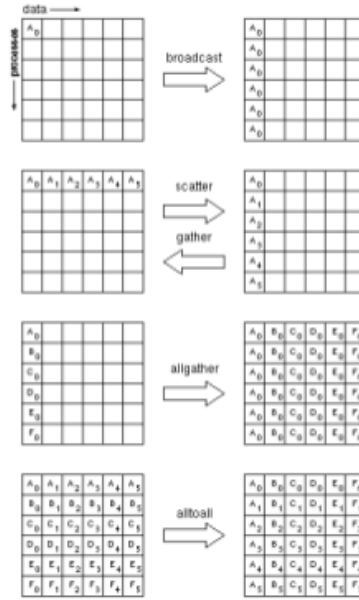


Figure 9.1: communication globales

tous les autres.

- `MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvttype, root, comm)` est équivalent à un send de chaque processus (incluant root) et à n receive de root avec l'adresse de réception calculée  $\text{recvbuf} + i * \text{count}$ .
- `MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvttype, root, comm)` est équivalent à root qui fait n send de sendbuf +  $i * \text{sendcount}$  et chaque processus qui fait un receive.
- `MPI_GATHERV`, `MPI_SCATTERV`, viennent avec un vecteur de `recvcount` ou `sendcount` et un vecteur de positions afin de recevoir ou d'envoyer des messages de taille différente à une position variable.
- méthodes tous à tous:
  - `MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvttype, comm)` est comme un gather excepté que tous les processus, il n'y a pas de root, reçoivent toutes les informations. `MPI_ALLGATHERV` existe aussi
  - `MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvttype, comm)` chaque processus reçoit une partie de ce qui est envoyé par chaque autre. Le bloc j envoyé par le processus i est placé

dans le bloc  $i$  du processus  $j$ . `MPI_ALLTOALLV` et `MPI_ALLTOALLW` existent aussi.

- Réduction:

- `MPI_REDUCE`(sendbuf, recvbuf, count, datatype, op, root, comm)  
les éléments de chaque processus sont combinés ensemble dans l'élément correspondant du root avec l'opération qui peut être MAX, MIN, SUM, PROD, LAND, BAND, LOR, BOR, LXOR, BXOR, MAXLOC, MINLOC; MAXLOC and MINLOC notent le processus qui fournit l'élément maximal ou minimal. Il existe aussi `MPI_ALLREDUCE` qui retourne le résultat à tous.
- `MPI_REDUCE_SCATTER`(sendbuf, recvbuf, recvcounts, datatype, op, comm) distribue les réductions selon le vecteur d'entier `recvcounts`.

- Autres Fonctions:

- `MPI_SCAN`(sendbuf, recvbuf, count, datatype, op, comm)
- `MPI_EXSCAN`, effectuent une réduction avec préfixe inclusive (0 à  $i$ ) ou exclusive (0 à  $i - 1$ ).
- `MPI_OP_CREATE`(function, commute, op) permet de définir une opération en y associant une fonction.
- `MPI_OP_FREE` relâche l'opération créée.

## 9.2 Groupes de Communications

On spécifie dans MPI dans différents groupe de communications. On peut spécifier nous mêmes notre topologies si c'est important. on utilise les fonctions suivantes pour définir nos groupes topologiques:

- Groupes de Communications:

- `MPI_GROUP_SIZE`(group, size), retourne la taille du groupe.
- `MPI_GROUP_RANK`(group, rank), retourne la position dans le groupe.
- `MPI_GROUP_TRANSLATE_RANKS`(group1, n, ranks1, group2, ranks2), pour chacun des  $n$  éléments du groupe 1 spécifiés dans `ranks1`, retourne le rang correspondant de group2 dans `ranks2`, ou `MPI_UNDEFINED` s'il ne s'y trouve pas.
- `MPI_GROUP_COMPARE`(group1, group2, result), compare les groupes pour voir si c'est le même ou s'ils ont les mêmes membres (IDENT, SIMILAR, UNEQUAL).

- Sous-groupes:

- `MPI_COMM_GROUP(comm, group)` retourne le groupe du communicateur.
- `MPI_GROUP_UNION(group1, group2, newgroup)`, `MPI_GROUP_INTERSECTION`, `MPI_GROUP_I` combinent deux groupes pour en faire un nouveau.
- `MPI_GROUP_INCL(group, n, ranks, newgroup)`, `MPI_GROUP_EXCL`, sélectionner des éléments à inclure ou exclure. `RANGE_INCL` et `RANGE_EXCL` existent aussi.

- Autres Fonctions:

- `MPI_GROUP_FREE(group)`, libère le groupe
- `MPI_COMM_COMPARE(comm1, comm2, result)`, vérifie si les communicateurs (listes de membres) sont identiques, similaires ou différents.
- `MPI_COMM_DUP(comm, newcomm)`, copie le communicateur.
- `MPI_COMM_CREATE(comm, group, newcomm)`, crée un communicateur à partir d'un groupe.
- `MPI_COMM_SPLIT(comm, color, key, newcomm)`, crée un communicateur pour chaque groupe de processus avec la même couleur.

- Attributs de communicateurs:

- `MPI_COMM_CREATE_KEYVAL(copy_fn, delete_fn, key, fn_data)` réserve un code, `key`, pour un nouveau type d'attribut. les fonctions `fn_copy` et `fn_delete` sont appelées avec `fn_data` lorsque l'attribut est copié ou relâché.
- `MPI_COMM_FREE_KEYVAL(key)`
- `MPI_COMM_SET_ATTR(comm, key, value)`, `MPI_COMM_GET_ATTR(comm, key, val, flag)`, `MPI_COMM_DELETE_ATTR(comm, key)`, ajoute, accède ou enlève un attribut sur un communicateur.

- Intercommunications:

Communicateur entre deux groupes disjoints, local (inclut le processus courant) et remote (l'autre groupe).

- `MPI_COMM_TEST_INTER(comm, flag)`, `MPI_COMM_SIZE/GROUP/RANK`, `MPI_COMM_REMOTE_SIZE/GROUP`
- `MPI_INTERCOMM_CREATE(local_comm, local_leader, bridge_comm, remote_leader, tag, newintercomm)` doit être appelé collectivement, `bridge_comm` est un groupe englobant dans lequel `remote_leader` est identifié.

- `MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)`
- Topologies Cartésiennes:
  - `MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)`  
crée un nouveau communicateur sur lequel il sera facile de connaître les voisins selon une grille à `ndims` dimensions (périodique, communication entre premier et dernier, ou non).
  - `MPI_DIMS_CREATE` peut aider à choisir la grille.
  - `MPI_CARTDIM_GET`, `MPI_CART_GET`, permettent d'interroger la structure de la grille.
  - `MPI_CART_RANK(coom, coords, rank)`, retourne le rank d'une coordonnée.
  - `MPI_CART_COORDS(comm, rank, maxdims, coords)`, retourne les coordonnées d'un rank.
  - `MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)`, calcule les voisins source et destination pour une communication d'un déplacement et direction (dimension) spécifiés.
  - `MPI_CART_SUB(comm, remain_dims, newcomm)` crée un communicateur sur une topologie sur un sous-ensemble des dimensions.
- Topologies de Graphes:
  - `MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)`, crée un nouveau communicateur pour lequel il est facile de savoir les voisins dans le graphe spécifié. Index dit où le trouve la dernière arête dans `edges` connectée au noeud correspondant.
  - `MPI_GRAPHDIMS_GET`, `MPI_GRAPH_GET`, `MPI_GRAPH_NEIGHBORS_COUNT`, `MPI_GRAPH_NEIGHBORS`, permettent d'interroger la structure du graphe.
  - `MPI_TOPO_TEST(comm, status)`, savoir si cartésien, graphe ou aucun.
- Récupération d'erreur:
  - `MPI_COMM_CREATE_ERRHANDLER(fn, errhandler)`, create a error handler from a pointer to function to call.
  - `MPI_COMM_SET_ERRHANDLER(comm, errhandler)`, associer une fonction de rappel en cas d'erreur lors d'un appel impliquant une communication.
  - `MPI_COMM_GET_ERRHANDLER`

- MPIERRHANDLER\_FREE
- D'autres fonctions utiles ..... :
  - MPI\_PROCESSOR\_NAME(name, resultlen), nom du noeud.
  - MPI\_WTIME() retourne le temps en secondes, double précision.
  - MPI\_WTICK() retourne la résolution du temps en secondes, double précision (e.g. .001s).

## 9.3 Analyse de performance et optimisation

Chaque version de fonction de MPI à une autre version qui peut faire de l'analyse de performance. Comme par exemple, la fonction `MPI_Send -j PMPI_SEND`. Cette nouvelle version fait de la stats et peut nous donner de l'info sur la performance.

### 9.3.1 Optimisation

Le principe pour optimiser les programmes MPI sont les suivants:

- Utiliser tous les noeuds, tout le temps (choisir la granularité, équilibrer la charge).
- Réduire la communication (grouper les messages, recalculer, garder des copies).
- Eviter les copies de données (e.g. `BSend`, données non contiguës).
- `RECV` avant `SEND`, communications asynchrones en parallèle avec le calcul.
- Requêtes réutilisées, éviter la scrutation..

Normalement, pour faire du profilage on trace tous les envois de message et blocages associés. `VampirTRace` est gratuit mais le visualisateur, `Vampir`, est commercial.

`MPE/Jumpshot` (Argonne National Lab), `TAU` (UofOregon, Los Alamos) ainsi que `Paraver` (Barcelona Supercomputing) sont des outils de trace.

Finalement, nous avons l'oracle `Studio Performance Analyzer`.

## 9.4 Conclusion

Pour conclure, MPI est relativement simple (en terme de commandes) et mature. Les grappes de noeuds n'ont pas changé aussi vite que les multi-processeurs. Il n'existe plus beaucoup d'équivalent en terme de compétition..

La mise à l'échelle est cruciale avec les problèmes de tailles monstrueuses avec des grappes de milliers de noeuds. Par contre, pour certaines applications, Hadoop et Spark peuvent être considérer comme des alternatives.

## 9.5 Exercices

Voici les Exercices pour la section MPI

## Chapitre 6 : MPI

- 6.1 Une tâche MPI s'exécute sur deux noeuds. L'idée est d'échanger de l'information entre le noeud 0 et le noeud 1. Cependant, les tâches bloquent et donc ne se terminent jamais. Quel est le problème?

```
switch (rank) {
  case 0:
    MPI_Send(msg0, MSG_SIZE, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Recv(msgr, MSG_SIZE, MPI_INT, 1, 1, MPI_COMM_WORLD, &status);
    break;
  case 1:
    MPI_Send(msg1, MSG_SIZE, MPI_INT, 0, 1, MPI_COMM_WORLD);
    MPI_Recv(msgr, MSG_SIZE, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
    break;
  default:
    break;
}
```

Les deux processus exécutent en même temps le send et bloqueront s'il n'y a pas assez de tampon disponible, espérant que l'autre processus se mette en réception. De manière plus générale, il faut premièrement éviter les blocages, ensuite éviter la sérialisation (chacun sauf le noeud 0 attend la réception du précédent avant d'envoyer au suivant, ce qui crée une longue chaîne). Ensuite, l'envoi avec tampon coûte possiblement une copie et l'envoi asynchrone demande une structure requête, un surcoût faible mais non nul. Les échanges sont un mécanisme intéressant (Send-Receive) et l'alternance noeuds pairs envoient, impairs reçoivent puis l'inverse est un bon moyen de transmettre l'information sans avoir de longues chaînes de dépendances.

- 6.2 Un programme MPI est réparti sur un grand nombre de noeuds, chaque noeud contenant un vecteur qui donne la chaleur pour chaque point en X, le long d'une ligne horizontale qui correspond à un Y donné. Pour simuler la diffusion de chaleur dans une plaque, chaque noeud doit obtenir le vecteur de ses deux voisins, celui avec Y juste en-dessous et celui avec Y juste au-dessus. Chaque noeud a une position, rank, qui va de 0 à size - 1 et qui correspond à la coordonnée en Y. La fonction `itere_chaleur` reçoit en argument `rp` (rangée précédente), le vecteur de chaleur du noeud de rang inférieur (ou NULL pour la première rangée), `rs` (rangée suivante), le vecteur de chaleur du noeud de rang supérieur (ou NULL pour la dernière rangée), et `rc` (rangée courante), le vecteur de chaleur du noeud courant. Cette fonction modifie l'argument `rc` en y plaçant les nouvelles valeurs de chaleur. La boucle de calcul de cette application est fournie. Complétez cette

boucle avec les énoncés (et commentaires appropriés) permettant d'effectuer efficacement ce travail.

```
int rank, size;
MPI_status status;
float rp[2048], rs[2048], rc[2048];
...
for(t = 0 ; t < max_time; t++) {
    /* Obtenir les valeurs des rangées précédentes et suivantes
       et leur fournir les nôtres. Nous sommes la rangée (noeud) rank,
       comprise entre 0 et (size - 1) et size est pair et > 1 */

    /* complétez cette section*/
    ...
    itere_chaleur(rp, rs, rc)
    ...
}
```

Plusieurs stratégies peuvent être utilisées pour réaliser ce travail. Une réalisation relativement simple est d'initier de manière asynchrone toutes les communications (envois de rc et réception de rp et rs). Il suffit ensuite d'attendre que les requêtes soient complétées. Il peut être un peu plus efficace de commander chaque opération sans aller de manière asynchrone, à condition de bien planifier toutes les communications pour éviter les blocages ou les dépendances en chaîne. Par exemple, à chaque envoi sur un noeud doit correspondre l'appel pour la réception sur le noeud de destination.

Une situation à éviter est la suivante:

```
for(t = 0 ; t < max_time; t++) {
    if(rank < (size - 1) {
        MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank+1, 1, rs, 2048, MPI_FLOAT,
                     rank+1, 0, MPI_COMM_WORLD);
    }
    if(rank > 0) {
        MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank-1, 0, rp, 2048, MPI_FLOAT,
                     rank-1, 1, MPI_COMM_WORLD);
    }
    itere_chaleur(rp, rs, rc)
}
```

Dans ce cas, tous les noeuds sauf le dernier commencent par échanger (du noeud courant, rang inférieur) avec le noeud de rang plus élevé. Il n'y a personne pour recevoir, sauf le dernier noeud qui passe directement au deuxième if et peut faire l'échange avec le noeud de rang size - 2. Pendant cet échange, tous les autres noeuds sont bloqués. Une fois que le noeud de rang size - 2 a terminé son échange avec size - 1, il passe au second if et peut compléter l'échange avec le



noeud de rang `size - 3`. Ceci continue ainsi de manière sérielle et tout l'avantage du parallélisme est perdu. Pour éviter ce problème, une stratégie très courante est de séparer les noeuds entre noeuds pairs et impairs. Dans un premier temps, les noeuds pairs échangent `rs` et `rc` avec le noeud impair de rang supérieur, et ensuite ils échangent `rp` et `rc` avec le noeud impair de rang inférieur.

```
for(t = 0 ; t < max_time; t++) {
    if(rank == 0) {
        /* Noeud 0, rangée paire, échanger avec le suivant seulement */
        MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank+1, 1, rs, 2048, MPI_FLOAT,
                     rank+1, 0, MPI_COMM_WORLD);
        itere_chaleur(NULL, rs, rc)
    }
    else if(rank == (size - 1)) {
        MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank-1, 0, rp, 2048, MPI_FLOAT,
                     rank-1, 1, MPI_COMM_WORLD);
        itere_chaleur(rp, NULL, rc)
    } else {
        if(rank % 2) {
            /*rangée impaire, échanger avec précédent puis avec suivant*/
            MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank-1, 0, rp, 2048, MPI_FLOAT,
                         rank-1, 1, MPI_COMM_WORLD);
            MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank+1, 2, rs, 2048, MPI_FLOAT,
                         rank+1, 3, MPI_COMM_WORLD);
        }
        else {
            /* Rangée paire, échanger avec suivant puis avec précédent */
            MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank+1, 1, rs, 2048, MPI_FLOAT,
                         rank+1, 0, MPI_COMM_WORLD);
            MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank-1, 3, rs, 2048, MPI_FLOAT,
                         rank-1, 2, MPI_COMM_WORLD);
        }
        itere_chaleur(rp, rs, rc)
    }
}
```

- 6.3 Chaque tâche MPI contient une matrice diagonale (tous les éléments sur la diagonale sont à `i` pour la tâche `i` et les autres à 0). Chaque tâche MPI doit envoyer le contenu de cette diagonale à la tâche de rang 0 qui place le contenu de la diagonale de la tâche `i` dans la rangée `i` de sa matrice. Le contenu de la matrice est ensuite imprimé par la tâche 0.

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

int main (int argc, char* argv[]) {
    int rank, size;
    MPI_Datatype staircase;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

float m[size][size];
for (int i=0; i < size; i++) {
    for (int j=0; j < size; j++) {
        m[i][j] = (i==j) ? rank : 0;
    }
}

MPI_Type_vector(size, 1, size+1, MPI_FLOAT, &staircase)
MPI_Type_commit(&staircase)
MPI_Gather(m, 1, staircase, m, size, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Type_free(&staircase);

if (rank == 0)
    for (int i=0; i < size; i++) {
        for (int j=0; j < size; j++) {
            printf("%g ", m[i][j]);
            printf("\n");
        }
    }

MPI_Finalize()
return 0;
}

```

## Chapter 10

# Outils de Vérif. et Analyse

### 10.1 LTTng

Sa marche surtout avec des capteurs de données.. genre:

- Énoncés insérés par le programmeurs (TRACE\_event et autres)
- Capteur insérés par le compilateurs (-fprofile-arcs)
- Capteurs insérés dans l'exécutable par décompilation.
- Capteurs insérés dans un programme en exécutions par insertion de point d'arrêt
- La trace est très compacte, alors sa prend pas beaucoup d'espace et sa nous dit toutes les instructions exécutées.

**Prise de Trace** Un printf n'est pas beaucoup efficace pour debugger. c'est long quand tu commence a n'en faire plusieurs par secondes (beaucoup d'overhead). normalement avec optimisation tu peux prendre une trace en 50-200 ms (??). avec le printf pour optimiser, il faut l'écrire en binaire. On décode après quand on a le temps. il faut aussi une structure de données par thread.

On pourrait aussi se passer de verrous, par contre en cas d'interruption on aurait un problème de réentrance. Par contre avec des ops atomique sa marcherait bien.

Au niveau des tampons, sa prendrait un tampons circulaire, un double tampons, ou multiples..

l'écriture sur disque continue perturbe beaucoup l'application étant donné le temps que sa prend écrire.. On peut tout faire en tampons, pi quand on a du temps, on écrit tout le tampons en mémoire. (tout sauver en continue vs. sauver les dernière minutes). Le principe de sauver les derniers instants servent

beaucoup pour faire l'analyse de crash (avoir les derniers message). C'est très couteux alors toujours écrire le tampons en mémoire peut être difficile, on peut séparer sa en deux processus, un qui écrit un qui le met en mémoire.. bref plusieurs différent trucs

## 10.2 gcov

c'est une instrumentation par gcc qui compte chaque entrées dans un bloc de base (une séquence d'instruction sans saut à l'intérieur).

C'est intégré dans l'environnement eclipse, sa serait très couteux, mais certaines énoncés sont exécutés en même temps. le compteur se met dans un compteur de base (15 instructions assembleur) et compte la dedans, le overhead est beaucoup moins grand que de compter à chaque instructions (to % performance).

Pour s'en servir, il faut se servir des bons flags de compilateurs. en exécutant le programme, il faut mettre les énoncés dans notre programme, l'outil va ensuite écrire un fichier avec toute les valeurs de compteurs. Tout le temps que le programme s'exécute, et quand sa fini, il sauve tout dans un fichier.. (je me repète).

on utilise gcov avec le rapport, sa rend les résultats lisible et sa nous explique bien qu'est-ce qui se passe (voir slide GCOV sur les transparents).

```
# Run the program compiled with options -ftest-coverage and -fprofile-arcs
[gzip-1.2.4a]$ gzip </tmp/evlogout >/dev/null
# Files .bb and .bbg are produced at compilation time, files .da at program exit.
# Gcov reads the .bb .bbg .da and .c files and produces .c.gcov
[gzip-1.2.4a]$ gcov -b -f deflate.c
[gzip-1.2.4a]$ less deflate.c.gcov
5      while (lookahead != 0) {
...branch 0 taken = 0%
6933680      INSERT_STRING(strstart, hash_head);
6933680      prev_length = match_length, prev_match = match_start;
6933680      match_length = MIN_MATCH-1;
6933680      if (hash_head != NIL && prev_length < max_lazy_match &&
branch 0 taken = 8%
branch 1 taken = 47%
branch 2 taken = 1%
        strstart - hash_head <= MAX_DIST) {
3367555          match_length = longest_match (hash_head);
3367555          if (match_length > lookahead) match_length = lookahead;
branch 0 taken = 0%
branch 1 taken = 100%...
```

Figure 10.1: résultats typique gcov

Le surcout en mémoire est d'environ 10 %. a peu près la même chose pour les données

## 10.3 gprof

ressemble beaucoup à GCOV. \*l'utilisation d'un profileur peut rendre ton programme 2-3 fois plus vite\* C'est un profileur qui nous permet de compté chaque

entrées dans une fonction encore une fois. Sa nous permet de savoir comment de temps son passer dans chaque fonction. Sa ne fait pas exactement la même chose que GCOV, sa fait une répartition dans le temps d'utilisation de nos fonctions, sa regarde à chaque x temps dans quel fonction nous somme rendus, et sa fait une stats avec sa. sa compte combien de fois une fonctions est appelés et aussi de quel place en mémoire.

Lorsque le programme compiler va s'exécuter, un timer à chaque 1ms va arrêter le programme, prendre la position du programme (dans quel fonction, de quel adresse la fonction est caller). Sa fait un vecteur de la même size du programme, et dans chaque position sa met le nombre de calls de c'est adresse à fait.

par contre, vu que sa s'exécute seulement à chaque 1ms, on peut généralement pas capter la routine d'initialisation.

Sa alloue un très grand espaces mémoire, mais c'est très utiles et sa coute pas beaucoup dépendamment de la fréquence de sampling (1ms vs 1us vs 5ms).

Pour résumer, sa illustre quand même très bien le temps passer dans chaque fonctions. Sa fait un arbre qui montre l'utilisation de chaque fonction (voir TP1).

l'attribution proportionnel en temps d'appel implique que tout les appels sont identique, ce qui n'est pas si vraie que sa.. certaines parties du programme fait des write qui peut varier en temps dépendamment de l'espace ou un écris, gprof ne peut pas vraiment comprendre ces subtilités.

```
# Run the program compiled and linked with option -pg
[gzip-1.2.4a]$ gzip </tmp/evlogout >/dev/null
# File gmon.out is produced when the program exits, used by gprof
[gzip-1.2.4a]$ gprof gzip >gprof.out
[gzip-1.2.4a]$ less gprof.out
Each sample counts as 0.01 seconds.
% cumulative self      self      total
time  seconds seconds  calls  ms/call  ms/call  name
27.13    7.53    7.53    1957     3.85     5.95  fill_window
23.20   13.97    6.44      1  6440.00 19205.66  deflate
14.84   18.09    4.12    1958      2.10      2.10  updcrc
12.16   21.46    3.38
 8.38   23.79    2.33
 3.78   24.84    1.05
...
-----
          0.00  0.00      1/1958      zip [3]
          4.12  0.00    1957/1958    file_read [8]
[7]  15.6    4.12  0.00    1958      updcrc [7]
-----
          0.00  0.00      1/1958      lm_init [27]
          0.00  4.12    1957/1958    fill_window [6]
[8]  15.6    0.00  4.12    1958    file_read [8]
          4.12  0.00    1957/1958    updcrc [7]
```

Figure 10.2: Exemple Gprof

On peut voir que file\_read ne consomme pas beaucoup de cpu, mais qu'il prend quand même beaucoup de temps dans updcrc. Sa permet de voir ou on passe

notre temps

GPROF nous donne le choix avec les options de compilations de faire pas mal ce qu'on veut avec une précision variable.

## 10.4 Oprofile

Outil de profilage qui fait uniquement la partie d'échantillonnage (il prend juste le temps, pas l'arbre d'exécutions). sur Linux, Perf fait pas mal la même chose. l'idée c'est comme gprof, sauf qu'on utilise les compteurs de performances (on trigger à chaque 100k fautes de caches.. etc). C'est très flexible.

On peut demander qu'à chaque fautes il y a une interruption, sa compte, sa nous donne une idées des pertes CPU causés par nos erreurs.

```
Counted CPU_CLK_UNHALTED events, count 50000
vma      samples  %      image name
00000000 450385    48.1828  cc1plus
00000000 11617     1.2428  as
4207d260 7206      0.7709  libc-2.3.2.so (cc1plus)
00000000 7046      0.7538  bash
000c0f70 6966      0.7452  XFree86
00000000 6397      0.6844  vim
00032150 5843      0.6251  libkonsolepart.so (kdeinit)
081ec974 5016      0.5366  lyx
420745e0 4928      0.5272  libc-2.3.2.so (cc1plus)
0804c5a0 4462      0.4774  oprofiled
00009f30 4154      0.4444  libpthread-0.10.so (lyx)
0003b990 4143      0.4432  libkonsolepart.so (kdeinit)
c01163d0 3865      0.4135  vmlinux (cc1plus)
c0155ef0 3800      0.4065  vmlinux (bash)
```

Figure 10.3: Exemple d'Oprofile

on peut voir que cc1plus à été interrompu 50% du temps, sa veut dire qu'elle a causé la moitié des fautes de caches.

on peut même demander d'annoter le code sources en assembleur, en mettant le compte de chaque fonction.

Les dernier outils qu'on a vu date de quand même très longtemps, il existait des variantes mais c'est pas mal les classiques

## 10.5 Valgrind

Valgrind est un peu plus spécialisé. Plusieurs problèmes comme les course, les fautes de caches, et autres erreurs beaucoup spécialisé sont difficile débbugger avec les vieux outils. l'environnement Valgrind décompile le code, et peut ajouter de l'instrumentation a plusieurs endroit, recompiler, et exécuter normalement avec

```

$ opannotate --source --assembly `which oprofiled`
: index = hash->hash_base[odb_do_hash(hash,
: while (index) {
1455 14.2689: test %eax,%eax
13 0.1275: je 804c5ef
: if (index <= 0 || index >= hash->descr...) {
12 0.1177: mov 0x8(%esi),%ecx
2 0.0196: lea 0x0(%esi,1),%esi
19 0.1863: cmp 0x4(%ecx),%eax
78 0.7649: jae 804c638 <odb_insert+0x98>
: char * err_msg;
: asprintf(&err_msg, "invalid %u\n",index);
: odb_set_error(hash, err_msg);
: return EXIT_FAILURE;
: }
: node = &hash->node_base[index];
12 0.1177: lea (%eax,%eax,2),%edx
41 0.4021: mov (%esi),%eax

```

Figure 10.4: Exemple d'Oprofile avec assembleur

des compteurs.

### 10.5.1 Memcheck

Un outil d'instrumentation qui permet de voir chaque écriture et lecture en mémoire, de même que malloc/free sont instrumentés.

Sa aloue un grand vecteurs qui compte pour chaque case mémoire si c'est allouer, utiliser, et autres.. Sa regarde toutes les bitfields (très fins).

évidemment, sa double la mémoire requis, sauf que sa aide beaucoup.

en faisant un malloc, sa déclare la zone accessible (sa change le vecteurs) et sa fait d'autres modif.

En faisant un écriture, on regarde dans le vecteurs si c'est accessible. Sa nous donne une assurance qu'on n'a pas de corruption mémoire. Par compte, sa rend notre programme de 10 à 30 fois plus lent.

sa détecte:

- les accès invalides
- l'utilisation de bits ou octets non initialisés
- les fuites de mémoire
- les free redondants ou incorrects
- les recoupement d'adresse source et destination pour les fonctions de la famille de memcpy.

C'est un outil très efficace pour les programme qui peut rouler 10-30 plus lent, genre pas les trucs real-time / embarqué.

### 10.5.2 cachegrind

Un outil a priori moins intéressant. Sa instrument les accès mémoire. Il va à chaque accès voir où ça se situe en cache (soit spécifier ou découvert automatiquement la longueur de cache). Ça comprend un peu le comportement attendu en cache, il peut ensuite fournir de l'information sur le programme pour dire si il y a des problèmes ou non. Si on a accès à un profileur, ça sert pas à grand chose.. Un point positif, c'est que c'est indépendant de l'architecture, alors on peut tester sur des trucs assez étonnants.

Il y a l'option callgrind, qui est utilisé un peu plus souvent (c'est comme gprof) ça donne l'arbre d'utilisation de chaque fonction ainsi que l'adresse où on l'a appelée. Ça regarde les appelants jusqu'à n niveau (spécifié).

Ça prend environ 50x plus de temps.

Par contre, il est beaucoup plus précis que gprof.

### 10.5.3 Address Sanitizer

Version de google de memcheck, ça instrument à la compilation.. etc. pareil comme memcheck. Ça implique qu'ils ont une version source de toutes leurs applications. (ils roulent linux.. alors c'est cool).

Pour chaque bloc de 8 octets, ils vérifient si la mémoire est allouée au lieu de chaque bit. On perd alors notre accès aux bitfields sauf que ça rend notre programme beaucoup plus performant (ça réduit l'overhead)

On peut changer aussi la résolution (16 bits, 32 bits, 64 bits...) tout se fait par morceaux, alors ça se gère quand même mieux qu'une taille bâtarde.

Pour le reste, ça regarde quand même les choses similaires à memcheck, sauf que ça roule 2 fois plus lent au lieu de 10-30 fois.. c'est mieux pour le real-time.

Un espace libre est alloué entre les espaces, ça fait une zone de quarantaine pour un certain temps et ça nous permet de voir des choses assez intéressantes..

Watchpoint dans le débogueur pour les cas difficiles, en mettant un seul watchpoint, il n'y a pas encore beaucoup de ralentissement. Par contre, en mettant trois watchpoints, ça devient très lent. C'est dû au nombre de registre disponible dans ton CPU. Ça fait des contextes switch à toutes les deux instructions assembleur alors ça fait du MEGA-overhead.

En mettant un watchpoint, ça nous permet d'observer des adresses mémoire où il y a rien, on les met avant nos structures pour voir quand la corruption se passe



### 10.5.4 Massif

c'est comme la même chose que Heap Profile (voir plus bas). Sa regarde l'arbre d'appel des fonctions d'allocations et libération de mémoire, sa fait ensuite un arbre qui montre quel fonction appel malloc et free et regarde la quantité de mémoire utilisé / temps de processeur utilisé

### 10.5.5 Helgrind

Helgrind est encore plus spécialiser, sa nous permet d'instrumenté les programmes en mémoire partager. Sa détecte les course.. Sa détecte les erreur bizarre qui érrore mes données. Sa nous permet de trouver les problème plus difficile sur les applications parallèles. On peut aussi faire la vérification formel(lol)

Chaque variable en mémoire est dans l'état exclusif possédés par le segment de fil qui l'a accédées. Si un autre essaie de l'accéd et que c'est disjoint, le nouveau segment devient propriétaire. Sa analyse le processus de synchronisation des caches (MESI et MOESI) et sa peut nous dire ou sont les problèmes.

Sa fait aussi beaucoup d'analyse sur les verrous. pour chaque verrous il y une variable, il ne sait pas quel variable est accédés a quel verrous, sa regarde les accès mémoire et elle essaie d'associer les verrous aux variable. Quand il pense qu'il a une association, il regarde si l'accès à la mémoire se fait en respectant le verrous, lorsqu'il a un hit, il l'affiche.

Sa ralentit beaucoup le programme (genre 100x)

```
#include <pthread.h>

int var = 0;

void* child_fn ( void* arg ) {
    var++; /*Unprotected vs parent*/
    return NULL;
}

int main ( void ) {
    pthread_t child;
    pthread_create(&child, NULL,
    child_fn, NULL);
    var++; /*Unprotected vs child*/
    pthread_join(child, NULL);
    return 0;
}
```

Thread #1 is the program's root thread

Thread #2 was created  
at 0x511C08E: clone (in libc-2.8.so)  
by 0x4E33A4: do\_clone (in libpthread-2.8.so)  
by 0x4E33A30: pthread\_create (in libpthread-2.8.so)  
by 0x4C299D4: pthread\_create@\*  
(hg\_intercepts.c:214)  
by 0x400605: main (simple\_race.c:12)

Possible data race during read of size 4 at  
0x601038 by thread #1  
at 0x400606: main (simple\_race.c:13)  
This conflicts with a previous write of size 4 by  
thread #2  
at 0x4005DC: child\_fn (simple\_race.c:6)  
by 0x4C29AFF: mythread\_wrapper  
(hg\_intercepts.c:194)  
by 0x4E3403F: start\_thread (in libpthread-2.8.so)  
by 0x511C0CC: clone (in libc-2.8.so)  
Location 0x601038 is 0 bytes inside global var "var"  
declared at simple\_race.c:3

Figure 10.5: Exemple de détection des course d'Helgrind

Helgrind va aller plus loin que l'intersection des verrous. Il peut aussi voir les deadlocks.

Il peut détecter à l'avance les problèmes de deadlocks, il fait une verification par tread/verrous. C'est quand même couteux. Il regarde l'ordre de prise des

verrous et batis un graphe dirigé avec des contraintes. Il dit quel verrous devrait être pris avant quel autre de sorte que le programme ne se met pas dans un deadlocks.

```
Thread #1: lock order "0x7FEFFFA80 before 0x7FEFFFA80" violated
at 0x4C23C91: pthread_mutex_lock (hg_intercepts.c:388)
by 0x40081F: main (tc13_laog1.c:24)
Required order was established by acquisition of lock at 0x7FEFFFA80
at 0x4C23C91: pthread_mutex_lock (hg_intercepts.c:388)
by 0x400748: main (tc13_laog1.c:17)
followed by a later acquisition of lock at 0x7FEFFFA80
at 0x4C23C91: pthread_mutex_lock (hg_intercepts.c:388)
by 0x400773: main (tc13_laog1.c:18)
```

Figure 10.6: Ordre des verrous

## 10.6 Lockdep

C'est un autre outil qui vérifie l'ordre de prise des verrous. C'est un peu plus lent, mais sa fait la même vérif. qu'avec Helgrind. Sa fait un graphe de prise d'ordre et mets des contraintes (par threads). Et sa fait aussi une vérification du niveau et contexte dans lequel est pris une classe de verrous. normal, irq, irq actifs ou non. Un verrou pris en mode normal avec irq actifs et aussi en mode irq peut mener à un blocage.

c'est une convention de règle de priorité, on regarde si on respecte des règles spécifique pour l'utilisation de verrous. C'est très utiles pour le développement des drivers.

## 10.7 Thread Sanitizer

C'est un outil développer par Google. C'est thread safe, comparément à valgrind..

l'app. prend 5 à 10 fois plus de mémoire avec 2 à 20 fois plus de temps.

Sa instrumente à la compilation toute les lectures et ecriture en mémoire, sa regarde si chaque accès respecte la synchronisation.. Sa crée des cases par mots pour stocker l'info (si c'est accessible, si c'est utiliser..) Sa conserve aussi un tampon des derniers appels, et retours sur chaque fil d'exécution de manière qui puisse découvrir les cas de conflits.

## 10.8 CPU profile

Un autre outil de google. qui cherche à faire la même chose que gprof en terme d'échantillonnage, par contre, ils échantillonnent la pile au complet comme valgrind. Ils peuvent rebâtir une pile d'appel parfaite et peuvent recréer presque parfaitement l'exécution basée sur la pile (à chaque 1, 5 ms). ce qui nous manque c'est les fonction d'init, pas contre c'est négligeable.

Normalement, la pile d'appel à a peu près 10-15 pile d'appel. Sa peut varier si on se sert un peu trop de l'orienté objet (genre 50?)

Un programme performance typique à a peu près 15.

Par contre, en terme de couts c'est beaucoup mieux. Au cout du fait qu'il n'y a pas un compte exact comme avec gprof.

## 10.9 Heap Profile

Sa analyse les même chose que le CPU Profile, sauf que c'est spécialiser pour les procédure d'attribution de mémoire (malloc, free, etc.). sa regarde la collection de garbage, ainsi que le temps mis à initialiser nos données. Chaque allocation ou libération en mémoire est prise en compte par l'outil et est échantillonné, ensuite sa analyse l'arbre d'appel pour ces fonctions ainsi que l'espace prise par toutes ces fonctions.

Sa regarde quel fonctions fait quel allocations de mémoire, sa regarde d'où vient les problèmes de leak mémoire et autres. Sa fonction avec C, et avec Go

## 10.10 Traces d'exécution multi-cœurs

Les outils qu'on vient de voir résous des problèmes très spécifiques. Pour les problème d'ordre générale, on se sert d'un débogueur. Pour un outil générique qui n'est pas un débogueur, on utilise un outil de tracage et on analyse la trace a posteriori. Ces outils sont utiles pour des problème qui sont un peu trop subtiles pour le débogueur.

Sur linux:

- LTTng : développer à Poly, dans le lab du prof.
- dTrace : Développer par Oracle.
- Red Hat System Tap : similaire à dTrace, mais pour Red Hat
- Ftrace et Perf : outil de bas niveau qui fait partie des outils par défauts de linux. Sa reprend certains élément des contributions de LTTng au noyau linux.

- GDB tracepoint : nouvelle fonction permettant de placer efficacement des points de trace dans les applications à partir du débogeur. Sa nous permet de débogger les problèmes en temps-réel.
- DynInst: outil pour l'insertion dynamique de points de trace dans des exécutables avant ou pendant leur exécution. sa décompile, ajoute les points de traces, recompile, et sa fait des logs. c'est très couteux/ invasif

l'idée c'est qu'on ne sait jamais le problème d'avance. Alors on insère certains point de traces dans notre programme qui sont stratégique et nous permet de faire beaucoup de déduction. Au moment de l'exécution, on active les points de traces statique/dynamique et quand on spécifie quel outil on appel lorsqu'on frappe un point de trace. **eBPF** nous permet de configurer des règles d'avance et eBPF va gérer les points de traces. Sa convertit en bytecode qui est ensuite converti en exécutables.

Avec sa, lorsqu'on rencontre un point de trace, sa nous permet de faire un peu ce qu'on veut (spécifier dans eBPF).

Normalement, on écrit dans un tampon en mémoire. lorsqu'on a un problème, on prend une copie du tampon et on l'envoie dans un fichier pour analyse. (comme une black box).

On a aussi la possibilité d'ajouter des conditions qui peut convertir du C en bytecode eBPF, sa nous permet d'associer des conditions en points de trace qui est dynamiquement associer aux variables.. Sa nous permet de gérer le surcout/ quantité de données.

l'événement va être écrit dans le tampon avec un timestamp, qui nous permet de recréer un événement.

Sa nous donne vraiment une grande flexibilité sur qu'est-ce qu'on veut avoir et comment, toute en gardant la possibilité de faire des compromis de performances. Ce qui est pas toujours le cas pour les autres outils de tracages.

les gens font souvent une sorte d'équivalent en printf. c'est à dire, lorsqu'on rencontre la condition, on appelle une fonction avec plein de printf qui imprime un peu ce qui se passe présentement. Par contre, c'est dur d'être cohérent.

Un des problèmes est un peu la conformité, ce n'est pas tous les outils qui utilisent le même format.

### 10.10.1 Utilisation de LTTng

En exécutant un programme, on peut spécifier les points de traces qu'on veut regarder. On fait sa en créant une session de tracage qui gère par lui-même ses fichiers d'analyse. En le fermant, on peut quand même avoir accès au fichier

```

TRACEPOINT_EVENT(
    sample_tracepoint,
    message,
    TP_ARGS(char *, text),
    TP_FIELDS(
        ctf_string(message, text)
    )
)

```

Figure 10.7: Définition d'une trace

pour pouvoir analyser.

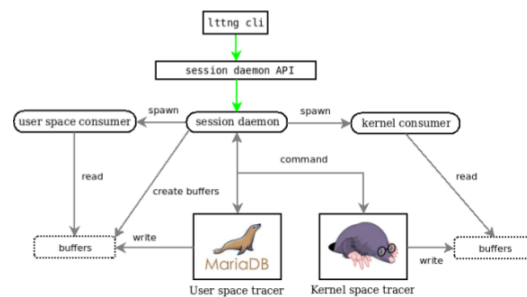


Figure 10.8: Trace Simultané

le client ouvre des pipes avec le daemon..

les images explique visuellement le fonction de l'outil LTTng.

### 10.10.2 Pourquoi une trace avec un cout minimale?

strace trace toutes les appels système, sauf que sa ralenti proportionnellement en fonction du nombre d'appel système. C'est ce qu'on voulait.. pas de bug..

Par contre, lorsqu'on utilise un traceur (ltnng), on peut voir que le bug est encore la (?bug?) le problème a été capté sur la trace. Le problème est du au fait que l'enfant n'a pas eu le temps d'envoyer sigusr 1 avant que l'appel de la mère s'est fait. c'est un exemple typique d'une course.

La performance est important, en utilisant un outil lent, on aurait jamais pu voir qu'un course se produirait

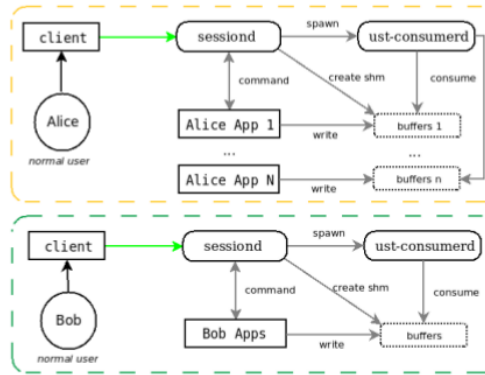


Figure 10.9: Session d'utilisateurs différents

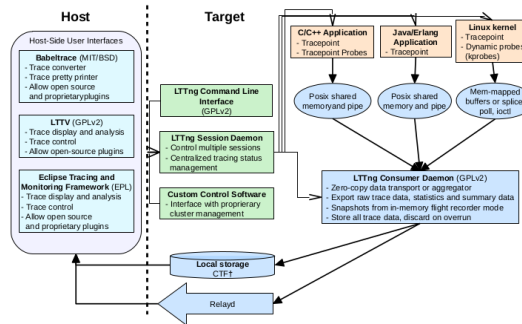


Figure 10.10: Gestion de LTTNG

```

if( sigaction(SIGUSR1, &new_action,NULL) <0) perror("Erreur\n");

if( sigaction(SIGUSR2, &new_action,NULL) <0)perror("Erreur\n");

if((pid = fork()) == 0){
    kill(getppid(), SIGUSR1);
    pause(); // Mise en attente d'un signal
}else {
    kill(pid, SIGUSR2); // Envoyer un signal à l'enfant
    printf("Parent : terminaison du fils\n");
    kill(pid, SIGTERM); // Signal de terminaison à l'enfant
    pid = wait(&etat); // attendre la fin de l'enfant
    printf("Parent: fils a termine %d : %d : %d : %d\n",
        pid, WIFSIGNALED(etat), WTERMSIG(etat), SIGTERM);
}

```

Figure 10.11: bug dans la prise de trace sans cout minimale

```

root@ventoux# ./signal1
Parent : terminaison du fils
Parent: fils a termine 3824 : 1 : 15 : 15
root@ventoux# strace -o trace ./signal1
Signal SIGUSR2 reçu
Signal SIGUSR1 reçu
Parent : terminaison du fils
Parent: fils a termine 3827 : 1 : 15 : 15
root@ventoux#

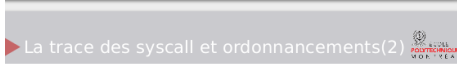
```

Figure 10.12: Il n'y a pas de bug en faisant sa..

```

[04.959234910] sched_process_fork: { 1 }, 3850, { parent_comm =
"signal1", parent_tid = 3850, child_comm = "signal1", child_tid =
3851 }
[04.959237757] sched_migrate_task: { 1 }, 3850, { comm = "signal1",
tid = 3851, prio = 20, orig_cpu = 1, dest_cpu = 3 }
[04.959240377] sched_wakeup_new: { 1 }, 3850, { comm = "signal1",
tid = 3851, prio = 120, success = 1, target_cpu = 3 }
[04.959241790] exit_syscall: { 1 }, 3850, { ret = 3851 }
[04.959267207] sys_kill: { 1 }, 3850, { pid = 3851, sig = 12 }
[04.959271053] exit_syscall: { 1 }, 3850, { ret = 0 }
[04.959271925] sched_stat_wait: { 3 }, 0, { comm = "signal1", tid =
3851, delay = 0 }
[04.959273444] sched_switch: { 3 }, 0, { prev_comm = "kworker/0:1",
prev_tid = 0, prev_prio = 20, prev_state = 0, next_comm =
"signal1", next_tid = 3851, next_prio = 20 }
[04.959280598] exit_syscall: { 3 }, 3851, { ret = 0 }

```



```

[04.959280045] sys_fstat64: { 1 }, 3850, { fd = 1, statbuf = 0x8FE05C14 }
[04.959280870] exit_syscall: { 1 }, 3850, { ret = 0 }
[04.959284802] sys_mmap_pgoff: { 1 }, 3850, { addr = 0x0, len = 4096, prot = 3, flags =
34, fd = 420400295, pgoff = 0 }
[04.959287152] exit_syscall: { 1 }, 3850, { ret = -1216471040 }
[04.959307500] sys_write: { 1 }, 3850, { fd = 1, buf = 0x877E2000, count = 29 }
[04.959320858] exit_syscall: { 1 }, 3850, { ret = 29 }
[04.959322118] sys_kill: { 1 }, 3850, { pid = 3851, sig = 15 }
[04.959324527] exit_syscall: { 1 }, 3850, { ret = 0 }
[04.959324655] sys_wait4: { 1 }, 3850, { upid = -1, stat_addr = 0x8FE03E50, options = 0,
ru = 0x0 }
[04.959330523] sched_process_wait: { 1 }, 3850, { comm = "signal1", tid = 0, prio = 20 }
[04.959336128] sched_switch: { 1 }, 3850, { prev_comm = "signal1", prev_tid = 3850,
prev_prio = 20, prev_state = 1, next_comm = "kworker/1:0", next_tid = 3022, next_prio =
20 }
[04.959348533] sched_switch: { 1 }, 3022, { prev_comm = "kworker/1:0", prev_tid = 3022,
prev_prio = 20, prev_state = 1, next_comm = "kworker/0:0", next_tid = 0, next_prio = 20 }
[04.959366489] sched_process_exit: { 3 }, 3851, { comm = "signal1", tid = 3851, prio = 20 }

```

Figure 10.13: la trace des syscalls

### 10.10.3 Exemple d'optimisation

Certaines optimisation ont été faite en espace usager, sans aucun interaction ave le système d'exploitation. Exemple, en mettant l'activation d'un point de trace sur un if improbable, sa va arriver, mais pas toujours, alors le cout est quand même bas.

En utilisant un format binaire natif. Normalisé en tant que **Multi-core association common trace format**.

en utilisant tampons par CPU avec opérations sans verrous.

technique de synchro entre la lecture des infos de config ainsi que leur modif par la technique Read Copy Update (RCU)

Aucune copie en mémoire directement..

## 10.11 Autres

l'analyse d'un trace se fait en listant les événements de différentes traces en utilisant une référence de temps commune (synchro a posteriori). On construit ensuite un modèle du système en prenant les infos de traces pour fournir de l'info sur ce que le système fait(appels systèmes, lecture/ écriture de fichier.. etc). Avec une navigation efficace sur l'info, sa fait des analyse quand même assez efficace.

### 10.11.1 synchronisation de traces

Chaque noeud à une horloge indépendante, alors même les timestamps ne peuvent pas nécessairement donnée une bonne synchronisation (sur plusieurs CPU). Sa fait une différence de première ordre : valeur initiale, fréquence différentes. c'est possible d'avoir des compteurs logiciels qui sont très forts, sauf que sa prend un peu de performance. La majorité des différences sont causé par la différence de fréquences des cristaux.

On peut voir aussi un effet de second ordre avec la variabilité de l'ordre ainsi que la latence de lecture de l'horloge, vu que c'est dérivé de la fréquence dans le temps (par exemple, la température, l'humidité et la tension des dispositif d'hydro-québec).

Le principes est d'identifier les envois et réceptions corresponants pour les paquets envoyés sur le réseau et utiliser l'information du paquet pour estimer les coefficient de l'horloge A vs. l'horloge B.

C'est un processus qui nous permet de voir le temps d'aller retour entre les deux ordis, ce qui nous aide a les synchroniser.

Avec sa, on peut modéliser l'accès ressources en XML avec les différents événements (type, valeurs des champs) et des changement d'états qu'ils causent. On peut voir quel thread est sur quel coeur, quel device est ouvert.. On peut reconstruire presque toutes les structure de données sur le système d'exploitation. On peut en faire un arbre/ base de données



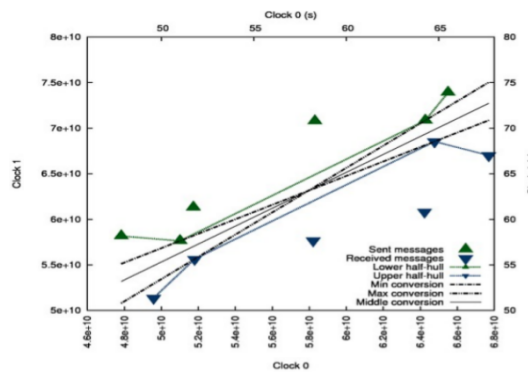


Figure 10.14: Graphe de performance des horloges

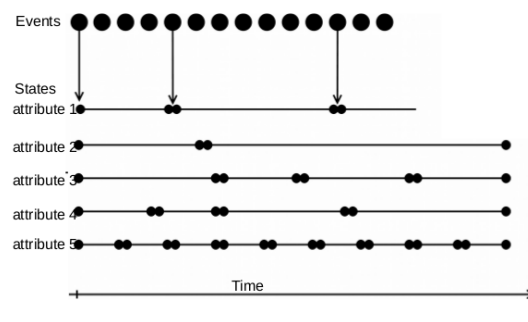


Figure 10.15: Modélisation de l'état

Sa représente l'état courant un moment données. Lors de la modélisation, on peut faire des liens entre les différents états et ainsi en faire nos déductions.

### 10.11.2 Analyse du chemin critique

Le principe est d'analyser, pour chaque processus, quel état est le bottleneck en terme de vitesse. on trouve les points qui nous ralentissent, et ensuite on regarde pourquoi. On peut ensuite faire nos modifications si c'est possible, et essayer de minimiser les pertes causées par notre chemin critique.

on essaierait de minimiser le nombre de requêtes. Au début, on se demandait si apt-get était capable d'être optimisé avec le chemin critique. On a trouvé que apt-get faisait juste attendre et assumait qu'après 0.5 secondes c'était fini (on perd beaucoup de temps quand c'est fini).

On peut aussi regrouper les tâches en métriques, comme par exemple le write

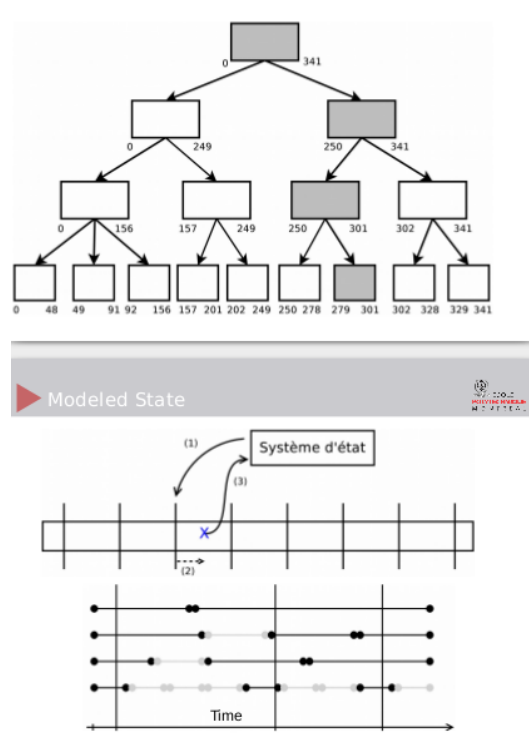


Figure 10.16: Historique et état modélisé

qui peut prendre un temps indéterminer en fonction du matériel sur lequel on écrit

On peut aussi visualiser les différence entre les groupes.

## 10.12 Vtune

Un outil offert sur Linux et Windows. Sa fait du profilage par minuterie ou compteur (Valgrind?, gProf?) sa montre l'état du fil d'exécution en fonction du temps, sa montre le montant d'attente active et passive par verrou ou section critique. Il y a même un API pour pouvoir mettre dans des programme qui vont faire exécuter des événements pour Vtune a fin d'analyse.

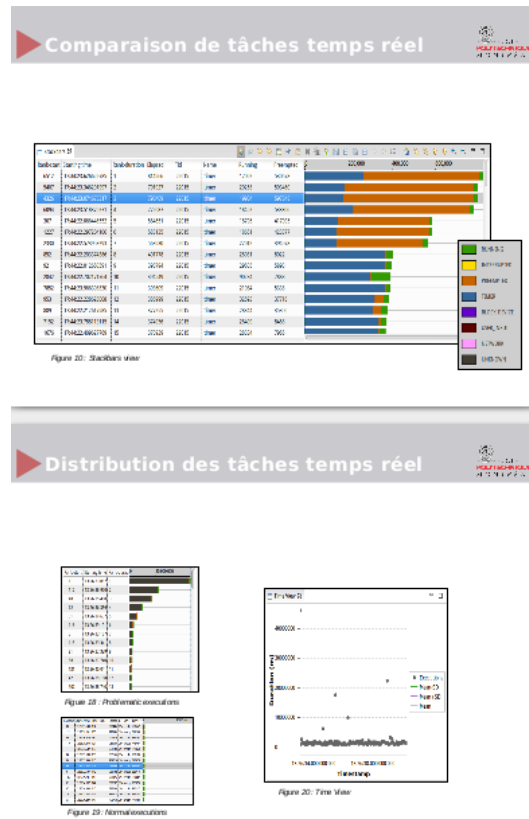


Figure 10.17: résultats des métriques

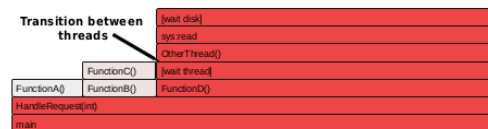


Figure 10.18: différences entre les groupes

## 10.13 Windows Performance Toolkit

Instrumentation du noyau et des bibliothèques système avec Event Tracing for Windows (ETW).

C'est un outil qui offre la possibilité d'ajouter des points de traces dans des applications et de pouvoir les activer dynamiquement. Les traces peuvent ensuite

être analysé avec des outils graphique.. Par contre, la trace doit être analysée sur l'ordinateur dont la trace a été effectuée.

## 10.14 Vampir

\*Les européens sont trop sophistiqués..\*

C'est un outil développé en Allemagne, le but de l'outil est d'analyser la performance des programmes fonctionnant sur plusieurs différents processeurs. On peut voir les événements en fonction du temps/ autres métriques par arbre d'appels (comme gprof). Pour résumer, son analyse des programmes fait avec MPI et autres bibliothèques de parallélisme distribués. On peut définir nos métriques d'évaluations et tout le kit, ça fonctionne avec le Open Trace Format..

## 10.15 PARaver

Un outil de visualisation des événements/performance pour applications parallèles. C'est développé au centre national de supercomputing de Barcelone. Par contre, il faut donner plein d'info pour télécharger malgré le statut open source. C'est un peu louche...

Le but de paraver est très similaire à Vampir, c'est un outil de visualisation pour les systèmes parallèles. C'est entièrement adaptable à différents types de traces. Il est possible d'ajouter des modules supplémentaires et même de comparer des traces. (ça ressemble à motif??)

## 10.16 Tuning Analysis Utilities

C'est américain, projet de l'université d'Oregon. Ça fait de l'instrumentation par interposition de bibliothèque. (LDPreload). Ça teste les différentes versions de bibliothèques. Ça embale les fonctions MPI ordinaire. Analyse statique du code source pour l'insertion automatique d'instrumentation.

On peut faire une activation sélective des points de trace. On a aussi une utilisation de minuterie ou de compteur matériels pour obtenir un profil (Paraprof).

On a aussi jumpshot pour voir l'état des fils d'exécution en fonction du temps.

Interface graphique très primitive aussi...

## 10.17 CODEXL

Outil d'AMD. il y en a un qui sert à développer des jeux, et un pour développer du OpenCL. On peut voir l'arbre d'appel et les transfert de données OpenCL (carte graphique). La plus grande difficulté est qu'on ne peut pas directement faire une trace sur la carte graphique. Le kernel OpenCL doit être loader avec des flags pour qu'il envoie des messages sur la trace CPU. L'outil s'en occupe.

fait notable: CodeXL fait une chaîne d'outil complètement opensource. On peut vraiment voir le fonctionnement

## 10.18 APITrace

Trace les appels OpenGL (graphique). l'idée de l'outil est d'avoir les informations sur la carte graphique frame par frames. Ça nous permet de voir l'arbre d'appel et certains bugs soit dans le programme ou la librairie OpenGL.

## 10.19 GPUView

Trace des événements noyau et en particulier de toutes les commandes du sous-système.

## 10.20 Promela et SPIN

Vérification formelle. l'idée est d'être capable de démontrer que dans tous les cas notre programme va donner le bon résultat (course). Ça regarde le fonctionnement de tous les mutex, barrière et autres.

il faut être capable d'exprimer les différents processus concurrents. Si on a trois threads avec leur propre instruction, on essaye toutes les combinaisons possibles (presque) pour pouvoir s'assurer qu'on respecte toujours les paramètres donnés ou bien pour tout simplement juste voir les résultats possibles.

Le prof utilise le tracage, pour les choses spécifiques = profilage avec perf pour des problèmes de performance, pour la mémoire gprof. Peut-être Valgrind si la performance n'est pas trop importante. Pour des problèmes avec MPI, utiliser TAU

## 10.21 OpenSpeedShop

<https://openspeedshop.org>

Les fonctionnalités d'openspeedshop sont toutes des choses que nous avons vues jusqu'à présent. Ils sont tous intégrés dans un seul outil et comportent même certains outils de performance GPU. Ils fournissent les vidéos pour un tutoriel de leur

```

byte A = 1; bool A_done = 0;
byte B = 2; bool B_done = 0;
byte x = 0; bool x_done = 0;
byte y = 0; bool y_done = 0;

proctype writer()
{
  do
    :: ! A_done -> A = 3; A_done = true;
    :: ! B_done -> B = 4; B_done = true;
    :: A_done && B_done -> break;
  od;
}
proctype reader()
{
  do
    :: ! x_done -> x = A; x_done = true;
    :: ! y_done -> y = B; y_done = true;
    :: x_done && y_done -> break;
  od;
}
Init { atomic {
  run reader(); run writer();
  do :: x_done && y_done -> break; od;
  printf("x = %d, y = %d\n", x, y);
} }

```

x = 3, y = 4  
x = 3, y = 4  
...  
x = 3, y = 2  
x = 3, y = 2  
...  
x = 1, y = 2  
x = 1, y = 2  
...  
x = 1, y = 4  
x = 1, y = 4  
...

Figure 10.19: exemple de Promela

outils, c'est assez complet (environ 100 slides). Ils sont entrain tranquillement d'amasser une communauté assez intéressante. Ils sont mieux financier et on la collaboration de plusieurs groupes.

Ils ont une approche moderne (QT) et sa fonctionne quand même bien. C'est quand même modulaire, alors c'est très customizable (les gens OpenSource aime sa).

Sa va surement continuer à se développer dans les prochaines années.

## Chapter 11

# Virtualisation Haute Performance

au niveau de l'infrastructure physique:

- a l'épreuve des tremblements de terres
- un peu en élévation
- pas beaucoup de fenêtres
- Centre dans des conteneurs, permet le déplacement et amène de la flexibilité
- Bâtiment rudimentaire avec ventilation extérieur pour avoir des couts minimes, sa permet un bon déplacement d'air et sa fait que sa ne vient jamais trop chaud à l'intérieur
- grande puissance d'alimentation électrique
- Refroidissement à l'eau et à l'air. Pays froid (Canada, pays nordique)

Climatisation:

- Chaque Watt consommé utilise 1 W pour le calcule plus 1 W pour climatiser (beaucoup de puissance)
- Air forcé à travers le chassis, watercooling block
- Compromis entre température et usure du matériel
- utiliser l'air extérieur, accepter une température plus haute et économiser pour réacheter du matériel lorsque sa plante..
- Environnement sans humain non éclairé

On fait sa par secteur pour chaque ordinateur. On passe du 110V  $\rightarrow$  12V DC  $\rightarrow$  110V jusque dans nos power bars. On utilise souvent aussi une pile de secours/génératrice.

On regarde aussi beaucoup la performance de flop/watt, c'est plus écologique.

En terme de chassis (les boites) on utilise souvent des boitiers 1U ou 3U (U = 3.7 pouces). On organise sa en rack, on a des standard pour rendre sa compacte:

On met les carte à la verticale, on les met au .75 pouces, on les mets à la verticale pour que ll'air puisse monter et optimiser le transfert de chaleur. On peut facilement se rendre à un grand nombre de carte. On joue aussi beaucoup avec la profondeur des chassis pour optimiser l'espace et le flot d'air dans la structure. Sa prend quand même un moyen de refroidissement.

La compagnie Facebook à voulu faire les choses efficacement sans tout faire sur mesure pour réduire les couts. Ils ont mis une organisation en place appelé OpenHardware qui nous permet de faire du stocke pour facebook, Openhardware met leur plans en ligne et one peut faire leur stocks pour eux en échange d'une compensation. Ultimement, ils cherchent à descendre le cout du matériel en le sourcant directement au producteurs.

Réseautique:

Il faut de la réseautique spécialisé à faible latence, le plus de latence, le plus de temps passer à attendre = le plus de perte de calculs..

On peut se servir aussi de réseaux ordinaire (10G), mais c'est pas trop great.. C'est mieux d'aller chez Infiniband pour avoir la performance. Il faut aussi prendre en considération la connection entre les différents noeuds, on peut avoir toutes les noeuds interconnectés ou bien d'avoir une solution partiellement commutés. On peut choisir d'avoir toute lesnoeuds connectés, mais sa risque de faire un gros clusterf\*\*\*.

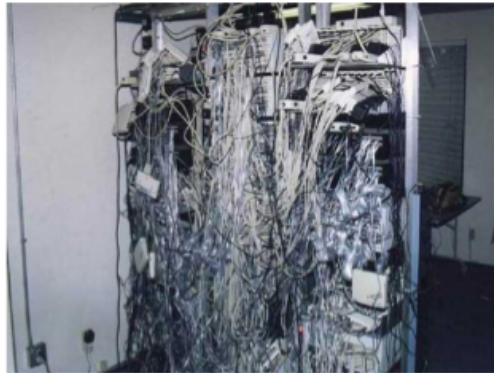
## 11.1 Surveillance de système et réseau

On utilise SNMP pour faire la gestion des noeuds sur le cluster. On peut changer plusieurs paramètre dans le protocole et sa peut changer les attribut de chaque noeuds dans le cluster. Sa nous permet de faire n'importe quoi avec notre systèmes. Sa prend par contre, une console de gestion qui nous permet d'interroger l'état de chaque noeud sur notre cluster.

les console/gestionnaire:

- Nagios





### Le câblage



Figure 11.1: Exemple d'un mauvais et bon câblage..

- OpenNMS
- Ganglia
- IBM Tivoli
- HP Network

Ces console peuvent même faire la découverte automatique du réseau, ça permet de faire une bonne partie du travail. C'est des commandes de l'alimentation et remise à zéro à distance pour les nœuds par KVM-IP ou BIOS-UEFI, on peut aussi faire de la redirection d'IO sur nos différents nœuds. On peut aussi faire un SSH, mais si la machine est gelée, on doit se servir des méthodes précédentes.

Intel a mis un ucontroleur dans la carte réseaux, sa nous permet de faire une communication avec le systèmes gestionnaire et redémarrer la machine en plus de pouvoir changer les paramètres de BIOS. Par contre, il faut que sa soit sécuritaire, sinon n'importe qui peut avoir l'accès. Les nouveaux chips ont même des systèmes d'exploitation sur les chips eux-mêmes.

## 11.2 Matériel

Certaines choses doivent être faites pour garder le systèmes opérationels avec un minimum de downtime:

- Remplacer les disques, blocs d'alimentation ou cartes électroniques assez régulièrement
- Problèmes subtils causé par du matériel en apparence identique, mais pas complètement compatible: interférence, chaleur..
- Contrats de service 4 heur ou 24 heures par jours..

Une analogie peut être fait, cattle vs pet.. Cloud = cattle, ou on fait juste changer les défaillance. tandis que le pet on en prend plus soins..

Il y a aussi une question d'entretien logiciel: On parle de correction de bug, mais dans les systèmes embarqués, il faut quand même prendre en considération que les producteur de logiciels délaisse leur client précédents il s'en foute, alors on peut être pris avec un bug majeur sans nécessairement avoir d'aide du créateur lui-même. Pour les système critique, c'est mieux d'être conservateur sur la version du systèmes d'exploitation. On cherche vraiment la stabilité tandis que certaines applications cherchent vraiment à avoir du bleeding edge.

Tout dépend du contexte..

## 11.3 Grappes de Calculs

Au bout de 5 ans, la valeur résiduelle est près de 0. Il faut vraiment installer les machines le plus vite possible sinon la valeur se perd très rapidement. On finalise l'architecture/processeur à la dernière minute pour vraiment maximiser l'investissement. Il faut aussi considérer l'espace requis (24000 noeuds = grand surface). Il faut aussi s'assurer de pouvoir l'utiliser en tout temps, et de n'importe ou. Normalement, on a une équipe de spécialiste qui sont dédié au bon fonctionnement d'un machine (le plus grand la machine, le plus grand l'équipe).

### 11.3.1 Infonuagies

la virtualisation facilite beaucoup l'implémentation des services sur des plateformes partagés.

### 11.3.2 Red Hat MRG

Red Hat messaging realtime grid. C'était une architecture d'infonuagies initialement fait pour faire du algorithmic trading. c'est très performants..

### 11.3.3 AMPQ

Protocole d'envoi de message pour les applications réparties capable de faible latence. utilisé par Red Hat MRG. pour des applications financière ou militaire avec beaucoup de messages par secondes

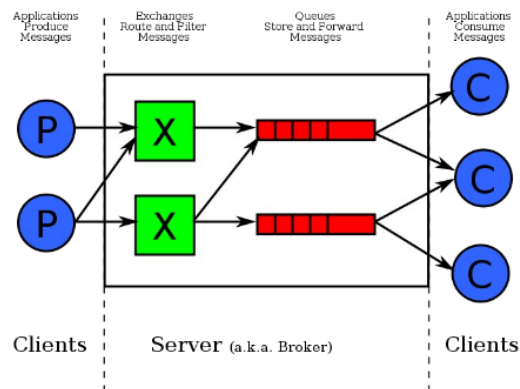


Figure 11.2: Exemple d'AMPQ

C'est fait pour que les messages soit encapsulé et contiennent toute sorte de propriétés comme par exemple l'expiration et tout pleins d'autre choses..

### 11.3.4 Grilles CONDOR

Condor est un systèmes qui fait la gestion de grille, c-a-d la gestion de noeud à grand échelle.

La grille fait en sorte que la load sur la machine est distribué dans le temps, dépendamment des applications. La notion de grilles n'a pas été utilisé pour très longtemps. C'est revenue chez amazon (EC2). Normalement, pour question de config, toute les machine on un daemon qui dit si cette machine peut

accepter des jobs, sont occuper, sont pleins ?

Il faut définir les tâches, les noeuds, certains paramètres de gestions.

```
MyType           = "Machine"
TargetType       = "Job"
Machine          = "froth.example.com"
Arch             = "x86_64"
OpSys            = "LINUX"
Disk             = 35882
Memory           = 128
KeyboardIdle     = 173
LoadAvg          = 0.1000
Requirements=TARGET.Owner=="smith" || LoadAvg<=0.3&&KeyboardIdle>15*60

CPUBusy = $(NonCondorLoadAvg) >= $(HighLoad)
WorkHours = ( (ClockMin >= 480 && ClockMin < 1020) && \
  (ClockDay > 0 && ClockDay < 6) )
AfterHours = ( (ClockMin < 480 || ClockMin >= 1020) || \
  (ClockDay == 0 || ClockDay == 6) )
START = $(AfterHours) && $(CPUIdle) && KeyboardIdle > $(StartIdleTime)
MachineBusy = ( $(WorkHours) || $(CPUBusy) || $(KeyboardBusy) )
WANT_SUSPEND = $(AfterHours)
```

Figure 11.3: Exemple d'une config

il faut spécifier les standard in/out pour voir les résultats.. pleins d'autre choses à faire..

Après avoir tout défini en ce qui concerne les tâches, il faut maintenant définir les caractéristique des noeuds. On peut spécifier quel tâches le noeuds peut prendre, quand peuvent-elles les prendre, qui peuvent-elles les prendre, toute ce fait.

Une questions délicate : la répartition des ressources. il explique que l'espace est la variable la plus critiques. l'idée est qu'avec une infinité de tâches, comment décider qui à accès à la machine. Sa prend un système sophistiqués qui défini une hiérarchie qui défini la répartition des calculs sur toutes les utilisateurs.

```
GROUP_NAMES = group_physics, group_chemistry, group_physics.lab1,
group_physics.lab2, group_physics.lab3, group_physics.lab3.team1,
group_physics.lab3.team2, group_physics.lab3.team3,
group_chemistry.lab1, group_chemistry.lab2

GROUP_QUOTA_DYNAMIC_group_physics = .4
GROUP_QUOTA_DYNAMIC_group_chemistry = .4
GROUP_QUOTA_DYNAMIC_group_chemistry.lab1 = .4
GROUP_QUOTA_DYNAMIC_group_chemistry.lab2 = .6
GROUP_QUOTA_DYNAMIC_group_physics.lab1 = .2
GROUP_QUOTA_DYNAMIC_group_physics.lab2 = .2
GROUP_QUOTA_DYNAMIC_group_physics.lab3 = .6
GROUP_QUOTA_DYNAMIC_group_physics.lab3.team1 = .2
GROUP_QUOTA_DYNAMIC_group_physics.lab3.team2 = .2
GROUP_QUOTA_DYNAMIC_group_physics.lab3.team3 = .4

GROUP_AUTOREGROUP_group_physics = TRUE
GROUP_AUTOREGROUP_group_physics.lab3 = TRUE
GROUP_AUTOREGROUP_group_physics.lab3.team1 = TRUE
GROUP_AUTOREGROUP_group_chemistry = TRUE
GROUP_AUTOREGROUP_group_chemistry.lab1 = TRUE
GROUP_AUTOREGROUP_group_chemistry.lab2 = TRUE
```

Figure 11.4: Exemple d'un quotas dans une config

On peut aussi spécifier d'autre paramètre subtil comme la consommation d'électricité. On peut décider d'éteindre les machines lorsqu'ils ne sont pas utilisés. On peut aussi spécifier le nombre maximal d'exécution concurrentes qui peuvent s'exécuter. Le monitoring devient un aspect très important dans la gestion du systèmes, on peut avoir un système qui s'assure que tout est respecté, sinon une alarme peut sortir (genre, quand le disque est plein ou qu'on est rendu dans le swap).

Condor est aussi à l'épreuve des pannes, il écrit régulièrement dans un fichier expliquant qu'est-ce qui se passe et lorsqu'il y a une panne, il fait un reboot et ensuite reprend son travail ou il avait arrêté.

### Dagman

dagman offre la possibilité de spécifier des groupes de tâche interreliés dans un graphe. On peut même imbriquer ces graphes pour avoir des groupes de tâches très complexes, et ensuite pouvoir spécifier des variables pour ces groupes de tâches.

Globalement, MRG est un système open source qui offre les mêmes fonctionnalités les uns les autres. Leur but est de faire la gestion de clouds

## 11.4 Virtualisation

On fait la virtualisation parce que ce n'est pas tout le monde qui ont les moyens d'installer des mega systèmes. Ça nous permet de louer des instances de machines offert sur diffère plateforme (amazon) et qui permet la facilité à installer les services sur leur machines.

Wine : librairie qui roule les librairies windows sur linux.. par contre, il y a des fois où ça ne marche pas du tout.. Ça ne marche presque jamais sur les jeux (20%). Il faut être prêt à ajouter des librairies sur Wine, et on peut se casser la tête.

Si on utilise pas Wine, c'est bien de pouvoir virtualiser nos services pour qu'ils puissent rouler sur n'importe quel système d'exploitation.

### 11.4.1 conteneurs

systèmes ou un seul noyau avec des espaces de noms séparés par conteneur pour les PID, IPC, usagers, réseaux, /proc, hostname, fichiers.. etc.

Chaque conteneurs peut rouler une version différentes des librairies applications.. mais il n'y a qu'un seul noyau en exécution. chaque conteneurs est utilisé

par un usagers. Cette solution n'implique aucun cout en performance, sauf la mémoire non partagée par les version différentes (exemple, librairie C), si c'est le cas.

Voici des exemples:

- Linux LXC
- FreeBSD Jails
- Solaris containers

### 11.4.2 Simulateurs d'exécution

Si les conteneurs ne fonctionnent pas, on peut se servir de simulateur d'exécution. C'est une programme qui simule l'exécution une par une de ton programme. On peut pré-traduire les instructions d'une architecture à l'autre, mais en générale, c'est optimiser pour la recompilation dynamique.. sa peut être de 5 à 50 fois plus lent. On est aussi à risque de bug, car il y a quelques différences entre le fonctionnement des systèmes exploitation (exemple: mode protéger génère des interruptions sur windows).

- 2: remplacement de certaines instruction
- 5: recompilation dynamique
- 50: interprétation

On ne veut généralement pas s'en servir, sauf que pour développer des apps cellulaires, sa peut être utile

### 11.4.3 Intel VT, AMD V (virtualisation matériel)

équivalent d'une interruption sauf qu'on associe un contexte à l'environnement. Sa devient problématique lorsqu'on essaie d'avoir un accès au système d'exploitation et aux périphériques. C'est problématique car il faut émuler les accès disques.. c'est très lents, c'est la ou on va être ralenti considérablement.

PCI Passthrough –

on a pratique la même vitesse, jusqu'à peut-être deux fois plus lent.. Si on a un genre de micro-benchmark avec beaucoup d'accès système/IO, sa risque d'être beaucoup plus lent.

le principe est que le OS en virtuel ce fait donnée un plage mémoire (ex, il demande 4G), et quand il parle directement au noyau, il exécute toujours dans

le 4g, cependant, le OS lui fait la traduction du 0- 4g jusqu'à la vraie adresse physique en mémoire. toute le mapping mémoire dans le OS virtuel doit être émulé dans le OS natif, tout ce travail d'émulation de table de pages doit être fait au deux endroit, alors lorsqu'on fait beaucoup d'opérations de gestion de mémoire, le overhead grandit. Sa fait en sorte qu'on a plusieurs différents étages de table de pages. On peut voir des ralentissement d'environ 20% dépendamment de l'utilisation

- 2-3 % : Pas de mise à jour
- 50% : beaucoup de gestion mémoire
- 600% : Micro-benchmark

Les machines virtuels sont rendu efficaces, par contre, il y a quand même des problèmes. La virtualisation se fait bien au premier niveau, par contre, quand on va au deuxième niveau, sa devient vraiment raide.

#### 11.4.4 Paravirtualisation

entre les deux. une collaboration entre une virtualisation matérielle et logiciel. c'est comme un deuxième niveau d'appel systèmes. Il y a un mecanisme qui ressemble au appels systèmes. C'est utilisé par Xen, VMWare le permet aussi en plus d'OpenBox. KVM peut soit utiliser le matériel de virtualisation ou la paravirtualisation si disponible. Le boot d'une machine virtuel sur linux se fait automatiquement (il voit si c'est une machine virtuel) Les pointeurs aux fonctions sont changer aux version virtuelles (paravirtualisation). Il prend en compte si c'est sur une machine native ou virtuelle et optimise en conséquence.

on se retrouve avec un solution, dépendamment de plusieurs chose, qui risque d'être plus performant. par contre, il faut installer une suite de driver spécialiser pour la paravirtualisation.

#### 11.4.5 Hyperviseur

Très populaire. (Xen) amazon roulait Xen jusqu'à tout récemment (maintenant KVM). c'est un système d'exploitation minimale qui gère les interruptions et les accès aux périphériques afin de les répartir entre les systèmes d'exploitation des machines virtuelles. Sa gère les interruption et la mémoire.

Évidemment, on ne voulait pas réimplémenter tout les drivers. On a mit le domaine 0 à linux (qui gère), et les autres domaines seront les autres services qu'on roule sur nos machines.

Certain soucit sont apparus, le controle d'interruption est devenue très compliqué. Si on a plusieurs coeurs, c'était interessant d'avoir des plages mémoire

réservés pour chaque coeur avec des channels accéléra en NUMA. On fait un classement optimal de processus en mémoire. Par contre, cela a faite en sorte que l'hyperviseur en devenue encore plus gros.. éventuellement sa fait en sorte que c'étais quand même très gros et ressemblais a un système d'exploitation lui-même.

Le principe de hot-swap était très bien, par contre, c'était dure de le faire sur l'hyperviseur sans amener des milliers de ligne de code sur Xen. C'est la ou il y a eu l'apparition de KVM..

C'est la ou il y a eu la fin des hyperviseurs.

#### 11.4.6 Bénéfice de la virtualisation

On peut installer ce qu'on veut. On peut avoir un environnement partagé, qui roule plusieurs différents machine virtuelle, ce qui amène une utilisation matériel beaucoup plus performante. On peut isoler les services à fin de sécurités ou de gestion. Plusieurs serveurs virtuels de différents groupes peuvent coexister sur le même serveur physique. Sa amène aussi une modularité des différents services. Cependant, certains de ces avantages peuvent aussi venir avec des conteneurs sans avoir le overhead du système d'exploitation.

Il y a eu un gros mouvement vers la virtualisation, sauf qu'on est revenu en arrière vers les conteneurs lorsqu'on a vue qu'on se sauvait du overhead.

#### 11.4.7 Cout de la virtualisation

Le moindre interruptions bizarre peuvent plantés le systèmes, alors sa peut-être difficile de garder le système en opération. Quand on utilise la paravirtualisation, sa fait l'équivalent d'un appel système sans avoir beaucoup de surcout, on a aussi la virtualisation des registre/hardware et plusieurs autres avantages. Tout sa fait en sorte que le setup peut être quand même difficile, sauf qu'en générale le surcout est faible.

Les changements de contextes était généralement couteux, cependant avec la virtualisation c'est encore plus couteux. Par contre, avec l'avancé des systèmes de pages, sa devient plus performant étant donné les méthodes de changement de pages des processeurs (il les sauves).

Vu qu'on utilise plusieurs système, on se retrouve avec une utilisation mémoire généralement moins efficaces. Dans certaines situations, il faut avoir une copie en mémoire du noyau et des différents exécutables pour que tout les services puissent fonctionner sans avoir trop de problème de concurrence, alors sa peut induire un surcout de mémoire élevé dépendament de quel librairie on doit copier. En plus, c'est difficile de savoir si c'est le même fichier/version. C'est quand même difficile de tenir trace de tout sa malgré les optimisations faite.



Sans les optimisations, on peut se retrouver avec 4+ copies d'une librairie en mémoire, ce qui peut être vraiment coûteux. Les optimisations qui se sont fait au sujet :

- On créer un petit daemon qui fait juste garder en mémoire les positions/-copies/versions des différents librairies identique et les fusionnent en une seule tout en gardant les modifications et subtilités. Généralement sa fonctionne très bien.

C'est crucial d'être performant sur la virtualisation, car si c'est pour induire un cout de 10 %, c'est quoi le but..

#### 11.4.8 Virtualisation du réseau

La virtualisation du réseau est fait à l'intérieur d'un noeud pour connecter les noeuds virtuels; Linux TUN/TAP (network tunnel, network tap). C'est fait pour abstraire une partie d'un réseau d'un plus gros réseaux, sa prend quand même un peu de jus CPU et sa peut pas se faire directement sur la switch.

**VLAN** Réseau local virtuel séparé du reste du réseaux local, c'est des étiquettes ajoutée à chaque paquets Ethernet et des gestions générale sur VLAN

**VPN/VPLS** connexions multi-points virtuelles privés par-dessus le réseaux publique.

Normalement, les réseaux virtuels sont des réseaux dédié (overlay network). On essaie d'optimier la latence, bande passante, qualité de service.. généralement ils chargent pour leurs services.

#### 11.4.9 Migration

Ce qui nous intéresse, par contre, c'est la migration des machines virtuels à travers différentes machines physique. Sa peut être vraiment long et coûteux de migrer une machine, comme par exemple une machine de 20-30gb peut prendre près d'une minute sur un réseaux gigabit..

Le terme migration n'est pas très accurate, car ce qu'on migre c'est pas le système, mais une image. l'essentielle c'est de tout copier les fichier/registre/etc. mais aussi de pouvoir recréer le contexte (adresse IP, autre, etc.) Et en plus, certaines restrictions sont mise (on ne peut pas changer de x86 à PowerPC). Certaines optimisation peuvent être faite pour rendre le processus beaucoup moins coûteux. Comme par exemple on peut commencer par migrer le stoque

beaucoup moins important et ensuite, le stocke important ce fait d'un seul coup.

Pratiquement, on a une sorte de daemon qui commence à copier toutes les fichiers qui ne sont pas primordial au systèmes. Pour déterminer les fichiers, on fait un sampling de tous les fichiers utilisé dans x temps et on regarde les fichiers qui sont le plus utilisé versus les moins utilisé. Ce qu'on espère, c'est que le daemon va converger (copier tous les fichiers) les fichiers les moins utilisés, on recommence ensuite le processus. Le but c'est de le faire dans le moins d'itérations possibles. certains paramètre peuvent être ajuster pour changer les philosophie de migration.

Une autre méthode, serait de faire l'inverse. On copie le coeur du système immédiatement et on commence l'utilisation du nouveau système. Lorsqu'on a une faute de pages, on cherche les données dans l'hôte et lorsqu'on les copie dans le nouveau système, on les met aussi en mémoire. Lorsque le système arrête de toujours avoir des fautes de pages, on commence à migrer les fichiers beaucoup moins important qui ne sont pas toujours utilisé.

La migration sert normalement à équilibrer les charges pour les gros sites/services qui requiert une sorte d'entretien. Une autre application serait pour la redondance, c-a-d, d'avoir des backups.

#### 11.4.10 Calcul parallèle et virtualisation

le cout d'une machine haut performance:

- 100 noeuds
- chassis
- réseau
- aménagement (0.5 M)
- ingénieur
- techiciens
- gestionnaire
- espace
- électricité pendant 5 ans =  $5 * 300k$

Le tout peut couter vraiment chere.. c-à-d 10m

Par contre, en louant des noeuds pendant 5 ans à 1\$ de l'heure, sa revient à 4.3M. Ce qui est moins chere... Le prix peut même descendre mais faut faire attention aux frais de stockage, réseaux et autres

De plus, il y a aussi question de confidentialité, fiabilité et juridiction.. question de sécurité.

## 11.5 Amazon EC2

C'est un service qui est offert depuis 2006. C'est un service de location de noeuds de calculs / machine virtuelles. Il y a plusieurs catégories de noeuds, services de stockages ainsi que plusieurs solutions intégrés de services de base de données, répartition de requêtes réseaux.. etc.

Étant donnée la grande palette de service offert par Amazon, le setup des services EC2 c'est fait de manière très smooth et sont dans le seuls joueurs à avoir ce genre de services et qui reste profitable.

Ils ont une allocations super flexible...

Ils sont basé sur Xen. c'est des images avec un noyau paravirtualisé offerts sur :Ubuntu, Suse, Microsoft server 2008, Solaris.. c'est possible de créer ses propres images linux, en choisissant les bonnes options de paravirtualisation et de les importer sur leur cloud.

en terme d'instance, on a plusieurs différent types:

- Standard; 1 Coeur, 1.7GO, 160GO sur disque
- Large ; 2Coeur, 7.5GO, 850GO sur disque
- High CPU, High Memory, Extra Large..
- Cluster Compute: dual quad core; 23GO, 2TB, réseaux 10GB/s
- Cluster GPU: machine avec deux NVIDIA Tesla M2050.

### 11.5.1 Stockage

Les instances de stockages stocke les données voulus pour le durée d'une instance EC2, avec l'image comme contenue initiale de la partition racine. Par contre, on a des solutions un peu plus permanentes comme EBS. ou tu lous une partition de disque pour stocker permenement des l'espace mémoire qui peut être attaché à une instance à la fois. On a aussi S3, qui est aussi une solution permanente, extensible et accessible de plusieurs d'instance en lecture et écriture simultanée.

### 11.5.2 Utilisation

On peut se servir d'une instance par interface web (aussi shell script), CLI ou même API d'amazon. sa peut être aussi simple que d'aller sur chrome, se connecter au site d'amazon et de se logger sur ton instance et démarrer à distance.

Normalement, on doit sélectionner l'image qu'on veut rouler ainsi que la machine sur laquelle on veut la rouler. On doit normalement aussi définir des règles d'accès pour le groupe de sécurité contenant l'instance.

### 11.5.3 Enchère de calculs et adresse

Il est possible de spécifier un prix de lancement d'instance pour un gros calcul à effectuer à bas prix. On spécifie un seuil, si le prix descend en dessous, notre instant démarre immédiatement et fait les calculs désirés. Par contre, il faut quand même s'assurer que le travail se fait facile tout seul. Les adresses sont internes à EC2, on est en train de ne plus avoir assez d'IPv4, alors on ne peut pas avoir une adresse par machine. on se sert de NAT si on veut que l'adresse soit visible de l'internet, par contre, il y a des frais associés. On peut prendre une instance Elastic IP address, c-à-d, que chaque compte peut avoir 5 add statiques associés au compte/ utilisateurs.

### 11.5.4 zones géographiques

Ils ont commencé au début dans 4-5 régions, (US-EAST, US-WEST, EUROPE, ASIA). La migration d'images peut se faire d'une région à l'autre. Cependant, les lois ne sont pas toujours pareil entre les régions alors question sécurité ça peut être compliqué.

### 11.5.5 Instance de grappes

C'est seulement disponible dans certaines régions. Ça prend des images spéciales qui ont un accès plus direct au matériel. Il faut que ça utilise le elastic block store (EBS). il y a aussi une possibilité de GPU. il va toujours avoir un système de queues sur les images pour vraiment bénéficier du plein potentiel du matériel.

### 11.5.6 répartiteur de charge

Plus rare, c'est un service qui répartit les requêtes, il réfère les requêtes aux différents front-end instanciés sur un network internes (NAT?). Il maintient des statistiques de performance pour mieux répartir la charge des requêtes. Ça fait un travail intelligent de manière automatique pour mieux se servir du matériel. L'utilisateur peut spécifier le fonctionnement du répartiteur pour maximiser l'utilisation. Ça simplifie beaucoup le travail.

### 11.5.7 nuage élastique

L'idée c'est d'utiliser les données du répartiteur de charges pour créer automatiquement de nouvelles images/ machines virtuelles si jamais la demande devient trop élevée.

### 11.5.8 surveillance CloudWatch

C'est une système qui mesure différentes performance du systèmes. Sa peut regarde par exemple le taux d'utilisation du CPU, accès en entrée et en sortie, nombre d'octets.. toute sorte d'affaire

### 11.5.9 nuage privé. LOL

Réseau virtuel, adresse IP choisis par l'utilisateur. on se connecte par réseau SSH, il y a une couche de sécurité supplémentaire, c'est très sécurée.

### 11.5.10 Services de bases de données

C'est un service de hosting de base de données. Il y toute sorte de trucs compliqués dans la gestion d'une base de données (on engage du personnel). Par contre, en utilisant les service d'amazon, il gère toute. Ils offre du MySQL, Oracle, toute les grands joueurs de base de données. Par contre, maintenant les protocole de base de données libres fonctionnent aussi bien qu' Oracle.

### 11.5.11 Discussion

EC2 permet de mettre à pied des systèmes très performants sans mettre les investissement matériel. Sa nous évite normalement un paquet de couts sur divers chose tout en restant dans la simplicité. Par contre, maintenant il y a quand même beaucoup de compétitions. Il faut aussi faire attention aux différents couts qui s'accumulent quand même assez rapidement. C'est plus chère généralement qu'une solution maison si on possède une grappe (LOL) mais sa sauve le hosting du matériel chez toi. C'est une excellent solution pour la capacité excédentaire ou ponctuelle, comme un plan de contingence/ un démarrage rapide..

## 11.6 OpenStack

C'est un projet démarré en 2010 par Rackspace et la NASA auquel se sont joint plus de 200 compagnies comme :

- AT+T
- Ubuntu
- HP
- IBM
- Red Hat
- SUSE
- Cisco

- Dell
- Ericsson
- Hitachi
- Huawei
- Intel
- Juniper
- NEC
- VMWare
- pleins d'autre..

Il y a une nouvelle version deux fois par an. l'API est compatible avec EC2. Malgré la fonctionnalité initialement limité, sa progresse très vite. Openstack offre les services suivants:

- Nova: infrastructure de calcul
- Neutron: infrastructure réseautique
- Swift: stockage de fichier
- Cinder: stockage de blocs
- Keystone: gestion des identités
- Glance: création et partage des images
- Horizon: panneau de commande
- Ceilometer: collecte de métrique
- Heat: configuration par recettes (templates)
- Trove: base de données
- Marconi: service de queue et notification (comme CONDOR)
- Savannah: service Hadoop

Pour résumer, c'est un équivalent libre de tout les outils développer par Amazon. C'est très modulaire alors sa fait qu'une erreur dans un module fait en sorte qu'on a pas besoin de tout ré-écrire. Au début, les première version n'avait pas de facturation/monitoring alors c'était quand même assez primitif. Par contre, maintenant c'est quand même rendu 'big'.

### 11.6.1 Nova

daemon qui roule sur chaque machine qui peut optimiser la gestion des tâches. Sa gère les différent noeuds dans une machine. C'est un serveur NOVA qui commandent les noeuds de calculs. Différentes technologies de virtualisation peuvent être utiliser (KVM, XenServer, VMWARE, LXC(conteneurs), natif (bare-metal), ...). l'architecture est réparti, hautement disponible et asynchrone. Il n'y a pas de notion de queues dans Nova, alors si sa se crée pas à l'instant, il ne se créera jamais.

Le démarrage, réinitialisation, redimensionnement, suspension ou arrêt d'instance se fait à partir du serveur. En plus d'avoir un contrôle d'accès, des quotas et de contrôle du débit. Il utilise une cache local d'images pour avoir un démarrage plus rapide.

INSÉRÉS SLIDE 66

### 11.6.2 Neutron

Sa fait la gestion de réseaux. Sa peut commander un réseau local, des réseaux vlan, un réseau défini par logiciel (SDN) ave OpenFLOW. Sa fait des adresse privées et publiques, statique ou flottante, DHCP.. Aussi un services additionnels comme détection d'insutrition, pare-feu (firewall), VPN, load-balancing..

On peut aussi faire une définition des groupes de sécurités avec des règles pour chacun. c'est très flexible avec beaucoup de paramètres. INSÉRÉ SLIDE 67 (EN BAS) + 68

### 11.6.3 Swift

Sa fait le stockage permanent de fichier. sa se base sur le protocole http, on se sert de curl, get, put, toute sorte d'affaire. Sa se fait avec des fichier et non du binaire.

INSÉRÉ SLIDE 69

### 11.6.4 Cinder

Les fichiers locaux d'un instance sont volatils. Sa fait la gestion des partitions de disque/disque et les associe aux instances appropriés.

INSÉRÉ SLIDE 70

### 11.6.5 Keystone

Répertoire des usager qui peut s'intégrer à LDAP. sa fait une authentification par mot de passe ou par jetons. On implémente aussi une politique d'accès et toute le kit.

### 11.6.6 Glance

Service pour les images. Sa supporte différent type d'images par instances.. on peut utiliser les types: KVM (Raw, qcow2), VirtualBox(VDI), VMWare (VMDK), Hyper-V (VHD).

Il offre une banque d'image déjà construite pour différent système d'exploitation. Ils sont configuré avec la paravirtualisation et tout sorte d'autre flexibilités

### 11.6.7 Horizon

Panneau de commande pour gérer toutes les services d'OpenStack à traver le web. c'est une alternative à l'API et au CLI. sa permetaussi de configure les service/ objets..

### 11.6.8 Ceilometer

C'est un système modulaire et flexible qui collecte et aggrège les donnes d'opération d'OpenStack. c'est utile pour l'optimisation de la performance et autre..

### 11.6.9 Heat

Un autre service qui fournit des recettes pour donner des configurations d'openstack pour pouvoir automatiquement implémenter la configuration sur du nouveau matériel. genre:

créer un serveur web front-end avec 10 instances. On déclare ce qu'on a besoin et l'outil va gérer automatiquement toute les outils pour avoir ce que l'on veut.

```
parameters:
  KeyName:
    type: string
  InstanceType:
    type: string
    description: Instance type for the instance to be created
    default: m1.small
    constraints:
      - allowed_values: [m1.tiny, m1.small, m1.large]
  ImageId:
    type: string
  db_port:
    type: number
    default: 50000

resources:
  my_instance:
    type: AWS::EC2::Instance
    properties:
      KeyName: { get_param: KeyName }
      ImageId: { get_param: ImageId }
      InstanceType: { get_param: InstanceType }
outputs:
  instance_ip:
    description: The IP address of the deployed instance
    value: { get_attr: [my_instance, PublicIp] }
```

Figure 11.5: Exemple de paramètre dans Heat



### 11.6.10 Trove

Un service de base de donnée. Sa fait un accès à une base de donnée par appel de procédure à distance (RPC). Sa fait la création et la gestion de base de données. Sa nous donne accès. Sa gère les backups. sa supporte différentes bases de données relationnelles ou non (MySQL, PostgreSQL..).

### 11.6.11 Marconi

Queues de messages, publication abonnement, notification. AMPQ. C'est bon pour un très grand nombres d'utilisateurs..

### 11.6.12 Savannah

Service Hadoop, Semblable à Amazon Elastic map-reduce. Sa réutilisation les autres services de OpenStack. C'est commandé par Horizon, sa fait des images avec hadoop dans glance. sa prend des noeuds de calculs avec nova. Le stockage se fait sur Swift et finalement l'authentification sur fait avec Keystone.

### 11.6.13 Discussion

Beaucoup de grand joueurs sont impliqués. C'est une infrastructure cloud entièrement open source qui offre des fonctionnalités vraiment semblable à Amazon. Le progrès est très rapide avec de grande différences d'années à l'autre. l'essentiel de la fonctionnalité requise est maintenant disponible.

Le mot clé de l'heure: Infonuagique avec Software, Platform, infrastructure as a service (SaaS, PaaS, IaaS). Il n'est pas efficace d'avoir d'innombrable grappes de petites tailles.. c'est comme si tout le monde produisait son électricité, eau, viande.. Les instances virtuelles offrent toute la flexibilité à très faible coût. Par contre, il faut quand même faire attention à sa facture et aux responsabilités légales.

## Chapter 12

# Top 500 + Architecture

Les chinois sont rendus au top, c'est une source de fierté nationale. Ils n'hésitent pas sur les moyens. Pour les comparés, ont a un benchmarks assez simple, c'est genre une grande boucle. relativement facile a optimiser et représentent bien le potentiel de performance. Dans un contexte comme sa, on montre la puissance brute mais on ne montre pas la performance à résoudre un vraie problème:

Les tops sont rendu au alentours de 100k Teraflops (93 petaflops). Le processeur sur cet ordinateur fais environ 20GFlops. À priori, ce processeur contient environ 10 Millions de coeurs. en assumant un résultats par cycles, on peut faire le calcule.. environ 1.4 GigaFlops par processeur, fois 10 millions = environs 14 Petaflops, le reste de la performance provient du vecteur. Ils ont leur propre systèmes d'exploitation basé sur Linux ainsi que leur propre système Réseaux Sunway qui ressemble grandement à infini-band.

Le deuxième plus rapide (Tianhe-2) est fait de 16000 noeuds avec 2 Xeon 8 coeurs et 3 Xeon 60 coeurs chaqu'un. Sa donne un total de 3120000 coeurs au total. Utilise un Réseau TH Express-2 et produit une performance d'environ 33882 Tera Flops. Utilise Kylin Linux avec un compilateur icc.

Le troisième, Titan, Oak Ridge national lab, avec un système Cray avec 18688 Opteron 2,2 Ghx à 16 coeur qui donne 299008 coeurs. Ils sont monté avec des Nvidia K20X, qui donne une persomance finale de 17590 TeraFlops. Ils utilisent un Réseaux Gemini, réseaux optimiser pour ce genre de système.

Payer pour le réseaux dépend beaucoup du genre de problème qu'on essay d'accomplir, comme par exemple, des simulations MonteCarlo sont pas mal tous indépendant les uns des autres et ne requiert pas vraiment beaucoup d'échange entre les différents noeuds, dans ce cas, une bonne technologie réseaux n'est pas nécessairement utile.

Les ordinateurs cray était monté en cercle pour avoir une topologie régulière

dans la connection des différents noeuds dans le cluster.

Le prochain, Sequoia, contient la technologie BlueGene d'IBM. Il contient 98304 Power BQC 1.6Ghz à 16 coeurs: ce qui donne 1572874 coeurs. Il contient un réseaux sur mesure, roule linux et produit une puissance d'environ 16328 Tera Flops. Autrefois, toutes les top contenait la technologie BlueGene.

Le prochain, RIKEN, contient 88124 SPARC64 VIIIfx 2.0Ghz à 8 coeurs. sa donne 705024 coeurs. Ils ont leur propre réseau TOFU (lol), roule Linux et produit environ 10510 TeraFlops

Mira, Juqueen, pleins d'autre roule blue gene et ont une perforance allant de 0 - 10k teraflops.

## 12.1 Discussion

Fait intéressant, nouvelle entrée dans le top 10, on à la Gyoukou, qui contient 20 millions de coeur allant à 700Mhz. Ils ont la même philosophie que le Sunway Taihulight, qui est d'avoir pleins de petits processeur efficaces et les mettres en parallèle. Chaque processeur contient environ 2048 coeurs. Leur système de refroidissement est fait à partir d'un liquide isolant, ils sont capable d'avoir une super bonne densité.

Ont peut voir plusieurs tendances, la compatibilité intel est une issue. Par contre, ultimement les performances ultime ne viennent pas des processeur standard. Il n'y a rien qui prouve que les processeurs intel sont meilleur que les autres (ou pire) dans ces systèmes. Maintenant on utilise des GPU/coprocesseur pour faire le gros de la programmation. En terme de puissance brute, c'est génial. Par contre, pour des utilisation un peu plus conventionnel, c'est beuacoup plus facile d'utiliser des proceseur ordinaire que des processeurs de ce style.

Aujourd'hui les gens ne regardent pas juste les teraflops, mais aussi les teraflops par watt.

Tilera avait une philosophie de faire plus de coeurs par nodes et de réduire la performance. Par contre, ils sont en faillite...

après la faillite de Tiler, Adapteva s'est mis dans le même mindset malgré leur équipq très petite.

## Chapter 13

# Exemple d'examen finaux

### 13.1 Examen 2016

#### 13.1.1 Question 1a

Ils nous demandent un programme en C qui correspondrait au programme en assembleur. On donne déjà des noms symbolique au lieu d'adresse.. alors c'est bien. On commence par charger des vecteurs. Petit bug avec VLR et 64?? C'est pas mal striaght forward, la forme SGEV.D -i  $\geq$ .. bref c'est sa.

Pour ce qui concerne le loop, mettons qu'on a 70 éléments, on va faire un and avec 63, ce que sa nous donne c'est qu'on prend seulement les bits qui sont plus petites que 63. en shiftant vers la droite pour le nombre de zéro, on se retrouve à diviser par 2 exposant le nombre de bits. ce qui nous donne  $R5 = 6$  et  $R6 = 1$ . le premier tour de boucle va se faire seulement pour 6 éléments... Quand on parle d'uniter regrouper, c'est en fonction des opérations, comme par exemple addition et soustractions sont regrouper ensemble, il y a une uniter de multiplication et un unité de division de plus qu'une uniter de load ou de store. on peut chaîner quand les instructions se trouve dans des unités différents et en ne jouant pas sur les même registres.

#### 13.1.2 Question 1b

La vitesse dépend du chaînage de nos instructions vectoriel. Si deux instruction consecutive sont dans des unités différentes on peut chaîner les événement (enlever le 64). Quand on commence, les chargements et rangement prennent 12 et utilise la meme unité apres alors le premier ne peut pas être chaîner. par contre, le deuxieme peux le chaîner.

#### 13.1.3 1c

Un point = un petit paragraphe.

**13.1.4 2a**

**un secret, programmation opencl et MPI.** On cherche à faire les opérations d'addition et de soustraction de manière local, et seulement un worker du work-group (le master) fait la transposition du résultats dans le global. on utilise la barrière pour s'assurer que tous les worker on fini les calculs avant de les transposé.

le résultats de s est  $(n * (n-1)/2)$

**13.1.5 2b**

Les accès vs la cache sont le principale facteur d'impact sur la performance. on cherche que les work items ont des cases consecutive en memoire. on peut voir que le global id (0) varie le plus vite.

on cherche à maximiser la grosseur des work items pour minimiser le overhead par contre on cherche aussi à maximiser l'utilisation de notre carte graphique.

on multiplie le nombre de coeur SIMD par le nombre de threads (8?)

**13.1.6 2c****13.1.7 3a**

La boucle trie chaque zone, le reduce a comme entrer un int (le min de chaque zone) et utilise la fonction min pour avoir le min de toute les noeuds.

**13.1.8 3b**

all to all i1 vers i2, on modifie, all to all vers i1, voir notes (INSÉRÉ NOTES)

**13.1.9 3c**

MPI\_Send est bloquant. on attend qu'on à reçu avant de continuer, c'est pas très performance mais sa peut nous éviter toute sorte de probleme. Bsend est bloquant aussi, sauf qu'on sauve une copie en buffer alors on peut tout de suite faire autre chose. Lsend envoie quand il peut, c'est pas nécessairement synchroniser sur rien du tout.

**13.1.10 4a**

on commence par faire les feuilles (D et E), étant donné qu'ils n'appellent personne. Par contre, on peut imputer ces valeurs à C et B.  $self + childs = self +$  les imputations des différentes valeurs.

**ÉCOLE POLYTECHNIQUE DE MONTREAL**

**Département de génie informatique et génie logiciel**

**Cours INF8601:** Systèmes informatiques parallèles (Automne 2016)

3 crédits (3-1.5-4.5)

---

**CORRIGÉ DE L'EXAMEN FINAL**

**DATE:** Jeudi le 22 décembre 2016

**HEURE:** 9h30 à 12h00

**DUREE:** 2H30

**NOTE:** Toute documentation permise, calculatrice non programmable permise

**Ce questionnaire comprend 4 questions pour 20 points**

---

## Question 1 (5 points)

- a) Un programme en C a été converti en assembleur Vectoriel MIPS. Le code source en C a été égaré. On vous demande de donner la section de programme en C la plus simple possible à laquelle pourrait correspondre le programme Vectoriel MIPS qui suit. (2 points)

```

LD      R1, N
LD      R2, a
LD      R3, b
LD      R4, c
ANDI    R5, R1, #63
DSRL    R6, R1, #6
ADDI    R6, R6, #1
MTC1    VLR, R5
loop:   LV      V1, R2
        LV      V2, R3
        SLTV.D  V1, V2
        ADDV.D  V3, V1, V2
        CVM
        SGEV.D  V1, V2
        MULV.D  V3, V1, V2
        CVM
        SV      R4, V3
        MTC1    VLR, #64
        ADDI    R2, R2, #64 * 8
        ADDI    R3, R3, #64 * 8
        ADDI    R4, R4, #64 * 8
        ADDI    R6, R6, #-1
        BNEZ    R6, loop

double a[N], b[N], c[N];
int i;

for(i = 0 ; i < N ; i++) {
    if(a[i] < b[i]) c[i] = a[i] + b[i];
    else c[i] = a[i] * b[i];
}
```

- b) Une unité centrale de traitement vectorielle est telle que décrite dans le livre et contient une de chacune des unités suivantes: i) addition et soustraction point flottant, ii) multiplication point flottant, iii) division point flottant, iv) opérations entières, v) opérations logiques, et vi) rangement et chargement. Ces unités en pipeline permettent de produire une nouvelle valeur à chaque cycle. Les instructions scalaires (e.g. LD) prennent un cycle d'exécution chacune. Dans ces

unités en pipeline, les additions et soustractions ont une profondeur de pipeline de 8, les multiplications de 16, les divisions de 24 et les chargements et rangements de 12. Calculez le temps requis pour l'exécution de la séquence d'instructions suivante. **(2 points)**

```

1  LV          V3, R3
2  LV          V4, R4
3  SUBV.D      V5, V2, V1
4  SUBV.D      V6, V3, V4
5  ADDV.D      V1, V5, V6
6  SUBV.D      V3, V5, V6
7  DIVVS.D     V2, V1, F1
8  DIVVS.D     V4, V3, F1
9  SV          V2, R5
10 SV          V4, R6

```

*Les opérations vectorielles peuvent être chaînées lorsque qu'elles sont consécutives et utilisent toutes des unités différentes. C'est le cas des lignes 2 et 3, 6 et 7, et 8 et 9. Le total est de 576 cycles.*

```

1  LV          V3, R3          ; 12 + 64
2  LV          V4, R4          ; 12 +
3  SUBV.D      V5, V2, V1      ; 8 + 64
4  SUBV.D      V6, V3, V4      ; 8 + 64
5  ADDV.D      V1, V5, V6      ; 8 + 64
6  SUBV.D      V3, V5, V6      ; 8 +
7  DIVVS.D     V2, V1, F1      ; 24 + 64
8  DIVVS.D     V4, V3, F1      ; 24 +
9  SV          V2, R5          ; 12 + 64
10 SV          V4, R6          ; 12 + 64

```

- c) Plusieurs nouveaux systèmes avec CPU et GPU intégrés, de compagnies comme AMD et Intel, offrent une mémoire virtuelle partagée entre les CPU et GPU. Quel est l'impact de ce changement? Donnez au moins deux conséquences, nouvelles possibilités ou opérations qui ne sont plus nécessaires. **(1 point)**

*Ceci a un impact important. Les copies ne sont plus requises entre la mémoire centrale et une mémoire séparée sur la carte graphique. La latence pour soumettre un calcul au GPU s'en trouve diminué et il est même possible de partager la mémoire pendant le calcul, par exemple avec des opérations atomiques sur des variables accédées simultanément par le programme sur le GPU et un programme sur le CPU. Autre conséquence, il est maintenant facile d'utiliser des pointeurs dans les structures de données car les adresses virtuelles sont partagées entre le CPU et le GPU.*

## Question 2 (5 points)

- a) La fonction OpenCL suivante est exécutée pour 1023 *work items*. Le vecteur d'entiers  $v$  contient (1, 2, 3, 4, 5..., 1024). Quelle est la valeur des paramètres  $s$  et  $d$  à la fin de l'exécution de cette



## fonction OpenCL. (2 points)

```
__kernel void sumo(__global long *s, __global long *d,
    __global const long *v)
{
    __local long local_s = 0;
    __local long local_d = 0;
    int i = get_global_id(0);
    atomic_add(local_s, v[i]);
    atomic_add(local_d, v[i+1] - v[i]);
    barrier(CLK_LOCAL_MEM_FENCE);
    if(get_local_id(0) == 0) {
        atomic_add(*s, local_s);
        atomic_add(*d, local_d);
    }
}
```

Ce programme calcule dans *s* la somme des éléments de 1 à 1023 dans le vecteur, ce qui donne 1023 éléments dont la valeur moyenne est de  $(1 + 1023) / 2 = 512$ , soit un total de  $512 \times 1023 = 523776$ . Dans *d* on fait la somme des différences entre les éléments consécutifs. La différence est toujours de 1, le total est donc de  $1 \times 1023 = 1023$ .

- b) Une fonction OpenCL fait un traitement simple sur les éléments d'une matrice. On vous propose deux versions. Laquelle sera la plus performante? Pourquoi? On vous suggère de changer pour avoir chaque *work item* qui traite une rangée complète de la matrice. Quel impact cela aurait-il sur le nombre de *work item*? Selon quel critère doit-on choisir d'avoir plus de *work item* plus courts, ou moins de *work item* mais plus longs? (2 points)

```
__kernel void MatrixA(const __global float* input,
    uint matrix_width, uint matrix_height,
    __global float* output)
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    int pos = j * matrix_width + i;
    if(input[pos] < 0) output[pos] = 0;
    else output[pos] = input[pos];
}
```

```
__kernel void MatrixB(const __global float* input,
    uint matrix_width, uint matrix_height,
    __global float* output)
{
    int j = get_global_id(0);
    int i = get_global_id(1);
```

```

int pos = j * matrix_width + i;
if(input[pos] < 0) output[pos] = 0;
else output[pos] = input[pos];
}

```

*Dans la version A, les work item consécutifs (prochaine valeur de `get_global_id(0)`) accèdent des cases mémoires consécutives. Ces accès consécutifs seront automatiquement regroupés et se feront donc beaucoup plus efficacement avec la mémoire globale. La version A sera donc significativement plus rapide. Pour optimiser un calcul, on cherche à avoir assez de work item pour utiliser tous les coeurs de calcul en parallèle (e.g., environ 8 thread par coeur de calcul pour être occupé même en cas de fautes de cache). Une fois ce nombre de work item atteint, il est préférable d'avoir des work item plus longs plutôt que plus nombreux, pour diminuer le temps requis pour transmettre et initialiser les work item (augmenter la fraction du temps utile, par rapport au temps de démarrage). Ici, chaque work item fait très peu de choses et il serait intéressant de lui en faire faire plus, à condition qu'il y ait assez de travail pour occuper tous les coeurs de calcul.*

- c) Sur les nouvelles architectures avec CPU et GPU intégrés, de compagnies comme AMD et Intel, on trouve maintenant des queues de commandes en mode usager (user-level queues) pour donner du travail au GPU. Quel est l'impact de cette nouvelle fonctionnalité sur le coût d'exécution d'une commande, comme l'exécution d'une fonction (kernel en OpenCL)? **(1 point)**

*Pour soumettre une nouvelle commande au GPU, il n'est plus nécessaire de faire un appel système. On sauve ainsi plusieurs centaines de ns. Ceci coupe d'autant le surcoût associé au lancement d'une commande et peut rendre efficace le fait de déléguer plus de travail au GPU, par exemple des courtes commandes qui autrement n'auraient pas été assez longues pour compenser le surcoût associé à leur lancement sur GPU.*

### Question 3 (5 points)

- a) Le programme MPI suivant s'exécute sur 4 noeuds (MPI\_Comm\_size retourne 4). Quelle est la sortie produite par ce programme à l'écran? **(2 points)**

```

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

int main (int argc, char *argv[])
{
    int i, j, rank, size, chunk, start, end, tmp, res;
    int v[] = {11, 9, 7, 5, 3, 1, 10, 8, 6, 4, 2, 0};
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

```

```

MPI_Comm_size(MPI_COMM_WORLD, &size);

chunk = sizeof(v) / sizeof(int) / size;
start = rank * chunk;
end = start + chunk;
for(i = start; i < end; i++) {
    for(j = i; j < end; j++) {
        if(v[j] < v[i]) {
            tmp = v[i]; v[i] = v[j]; v[j] = tmp;
        }
    }
}

printf("Node %d: %d\n", rank, v[start]);
MPI_Reduce(v+start,&res,1,MPI_INT,MPI_MIN,0,MPI_COMM_WORLD);
if(rank == 0) printf("Result: %d\n", res);
MPI_Finalize();
}

```

```

Node 0: 7
Node 1: 1
Node 2: 6
Node 3: 0
Result: 0

```

- b) Le programme suivant s'exécute sur 4 noeuds (MPI\_Comm\_size retourne 4). Donnez le contenu des trois lignes imprimées sur chaque noeud. (2 points)

```

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

int main (int argc, char *argv[])
{
    int size, rank, i, i1[4], i2[4], o1[4];

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    for(i = 0; i < 4; i++) i1[i] = rank * i + i*i;
    MPI_Alltoall(i1,1,MPI_INT,i2,1,MPI_INT,MPI_COMM_WORLD);
    for(i = 0; i < 4; i++) i2[i] = i2[i] * rank;
    MPI_Alltoall(i2,1,MPI_INT,o1,1,MPI_INT,MPI_COMM_WORLD);
    printf("%d: i1={%d,%d,%d,%d}\n",rank,i1[0],i1[1],i1[2],i1[3]);
}

```

```

    printf("%d: i2={%d,%d,%d,%d}\n", rank, i2[0], i2[1], i2[2], i2[3]);
    printf("%d: o1={%d,%d,%d,%d}\n", rank, o1[0], o1[1], o1[2], o1[3]);
    MPI_Finalize();
}

```

```

0: i1={0,1,4,9}
1: i1={0,2,6,12}
1: i2={1,2,3,4}
1: o1={0,2,12,36}
2: i1={0,3,8,15}
2: i2={8,12,16,20}
2: o1={0,3,16,45}
3: i1={0,4,10,18}
3: i2={27,36,45,54}
3: o1={0,4,20,54}
0: i2={0,0,0,0}
0: o1={0,1,8,27}

```

- c) Expliquez la différence entre les fonctions `MPI_Send`, `MPI_Bsend` et `MPI_Isend`. **(1 point)**

*La fonction `Send` bloque jusqu'à ce que la tampon d'envoi redevienne disponible, soit i) parce que le message a été envoyé ou ii) parce qu'il a été copié dans un autre tampon. La fonction `Bsend` prend une copie des données à envoyer et retourne le contrôle au programme, ce qui correspond au cas ii) de `Send`. La fonction `Isend` retourne immédiatement, même si le tampon n'est pas encore disponible (i.e. il est peut-être encore requis pour l'envoi); il faut vérifier l'état de la requête avant de le réutiliser, mais au moins en attendant on peut progresser sur autre chose.*

## Question 4 (5 points)

- a) Vous compilez un programme avec les options de profilage de `gprof` et l'exécutez ensuite. L'information suivante est produite pendant l'exécution. Sur 12 échantillons du compteur de programme récoltés à intervalle régulier d'une seconde, 6 sont dans la fonction C, 3 dans D, 2 dans A et 1 dans B. Le nombre d'appels de chaque fonction (avec la provenance des appels entre parenthèses) est `main: 1` (pas d'appelant), `A:1` (`main:1`), `B:2` (`A:2`), `C:3` (`A:3`), `D:6` (`B:4`, `C:2`), `E:2` (`A:1`, `B:1`). Calculez pour chaque fonction le temps passé dans la fonction elle-même et le temps passé dans la fonction elle-même plus ses enfants, comme le ferait l'outil `gprof`. **(2 points)**

*Le temps passé dans chaque fonction elle-même (soi) est donné par le nombre des échantillons, chacun comptant pour 1s. On a ainsi 6 secondes dans C, 3s dans D, 2s dans A, 1s dans B et 0s dans les autres. Pour le temps passé dans chaque fonction incluant les fonctions appelées (soi + appelés), il faut imputer le temps des fonctions appelées aux fonctions appelantes au prorata des appels. On commence par les fonctions feuilles où le temps "soi + appelés" est le temps de chaque fonction elle-même, soit D: 3s, E: 0s. Ce temps est imputé  $4/6=2s$  à B et  $2/6=1s$  à C. On*

peut donc calculer le temps "soi + appelés" pour B et C et continuer ainsi en remontant jusqu'au programme principal.

Fonction	self	self+childs
main	0s	12s
A	2s	12s (1/1*12s=12s de main)
B	1s	3s (2/2*2s=3s de A)
C	6s	7s (3/3*7s=7s de A)
D	3s	3s (4/6*3s=2s de B et 2/6*3s=1s de C)
E	0s	0s

- b) Pour chacun des 4 cas suivants, suggérez un outil qui serait particulièrement approprié pour découvrir ou cerner le problème. Expliquez comment cet outil permet de détecter ou cerner chaque problème. i) Un programme de calcul à paralléliser prend trop de temps en raison de certaines fonctions qui ne sont pas assez optimisées ou qui pourraient être parallélisées, on veut identifier ces fonctions. ii) Un programme fait par un étudiant de première année est bourré de problèmes comme l'accès à des variables non initialisées, des débordements de vecteurs et des libérations prématurées de mémoire. iii) Un programme s'exécute avec une bonne performance en moyenne et produit un résultat correct mais présente, à de rares occasions, des latences trop élevées pour accomplir certaines tâches. iv) Un programme présente tout probablement des problèmes de faux partage en mémoire cache, et on veut identifier à quel endroit ceci se produit dans le programme. **(2 points)**

Pour avoir une bonne idée du temps passé dans chaque fonction au total i), l'outil gprof ou des profileurs comme Oprofile ou perf fonctionnent très bien. A chaque intervalle de temps, la position courante du compteur de programme est échantillonnée, ce qui donne la fonction en train de s'exécuter. Toute fonction qui consomme beaucoup de temps CPU générera beaucoup d'échantillons et sera facilement identifiée. De plus, le surcoût à l'exécution est faible. Pour trouver les problèmes d'accès mémoire incorrects ii), les outils Memcheck ou Address Sanitizer sont très efficaces, avec Memcheck possiblement plus complet mais plus coûteux en temps à utiliser. Ces outils instrumentent les allocations et libérations de mémoire (pour identifier les zones valides à accéder) ainsi que les lectures et écritures. Pour chaque écriture on note que le contenu devient initialisé et on vérifie que la mémoire est allouée, et pour chaque lecture on vérifie que le contenu a été initialisé et que la mémoire est allouée. Pour des problèmes intermittents comme ceux de latence iii), un traceur comme ftrace ou LTTng, possiblement couplé à un détecteur de latence pour activer la sauvegarde de la trace, est généralement le meilleur choix. On peut ainsi savoir qu'est-ce qui s'est passé à quel moment et trouver les événements qui indiquent où et quand a lieu une latence trop élevée. Par exemple, on peut avoir des événements pour l'entrée et la sortie de fonctions importantes et des appels systèmes, et pour l'ordonnancement des processus. Ceci permet normalement de comprendre ce qui s'est passé pendant la latence problématique. Pour trouver les problèmes de fautes de cache et faux partage iv), les outils de profilage basés sur les compteurs de performance comme Oprofile ou perf sont excellents et peu coûteux en temps. En demandant une interruption à toutes les 100000 fautes de cache, par exemple, et en échantillonnant l'adresse du code et de la variable visés, on peut rapidement voir les sections de code, ou les variables, où surviennent un grand nombre de fautes de cache.

- c) Pour le deuxième travail pratique, vous avez repris un algorithme sériel afin de le paralléliser avec OpenMP. Après avoir simplement ajouté des directives OpenMP `parallel for`, il y avait des sources de bruits dans la sortie générée. Quelle en était la cause? Est-ce que ces problèmes affectaient aussi la performance? **(1 point)**

*Une variable partagée causait des problèmes lorsque plusieurs fils d'exécution se mettaient à l'accéder en parallèle avec OpenMP. Ceci causait à la fois un problème de corruption non déterministe (bruit dans les données) et un problème de performance en raison du ping pong d'invalidations et de fautes entre les cache de plusieurs processeurs, dont les fils accédaient en parallèle la même variable.*

Le professeur: Michel Dagenais

## **13.2    Années 2014**

Le final de l'année 2014

**ÉCOLE POLYTECHNIQUE DE MONTREAL**

**Département de génie informatique et génie logiciel**

**Cours INF8601:** Systèmes informatiques parallèles (Automne 2014)

3 crédits (3-1.5-4.5)

---

**CORRIGÉ DE L'EXAMEN FINAL**

**DATE:** Mercredi le 17 décembre 2014

**HEURE:** 13h30 à 16h00

**DUREE:** 2H30

**NOTE:** Toute documentation permise, calculatrice non programmable permise

**Ce questionnaire comprend 4 questions pour 20 points**

---



## Question 1 (5 points)

- a) Convertissez le programme suivant en code efficace écrit en assembleur Vectoriel MIPS. (3 points)

```
double a[N], b[N], c[N], d[N];
int i;

for(i = 0 ; i < N ; i++)
    if((a[i] * b[i]) != 0) d[i] = (a[i] + b[i]) * c[i];

; charger les valeurs et adresses dans des registres
    LD      R1, N
    LD      R2, a
    LD      R3, b
    LD      R4, c
    LD      R5, d
    L.D     F0, #0
; nombre d'opérations vectorielles de 64 éléments
    DSRL    R6, R1, #6
; nombre d'opérations restant à la fin
    ANDI    R7, R1, #63
    BEQZ    R6, end ; moins de 64 éléments
loop:  LV     V1, R2
      LV     V2, R3
      LV     V3, R4
; d est chargé car l'ancienne valeur reste si a[i]*b[i] == 0
      LV     V4, R5
      MULVV.D V5, V1, V2
      SNEVS   V5, F0
      ADDVV.D V5, V1, V2
      MULVV.D V4, V3, V5
      CVM
      SV      V4, R5
      ADDI    R2, R2, #64 * 8
      ADDI    R3, R3, #64 * 8
      ADDI    R4, R4, #64 * 8
      ADDI    R5, R5, #64 * 8
      ADDI    R6, R6, #-1
      BNEZ    R6, loop
end:   MTC1   VLR, R7
      LV     V1, R2
      LV     V2, R3
      LV     V3, R4
```

```

LV          V4, R5
MULVV.D V5, V1, V2
SNEVS      V5, F0
ADDVV.D V5, V1, V2
MULVV.D V4, V3, V5
CVM
SV          V4, R5

```

- b) Une unité centrale de traitement vectorielle est telle que décrite dans le livre et contient une de chacune des unités suivantes: i) addition et soustraction point flottant, ii) multiplication point flottant, iii) division point flottant, iv) opérations entières, v) opérations logiques, et vi) rangement et chargement. Ces unités en pipeline permettent de produire une nouvelle valeur à chaque cycle. Les instructions scalaires (e.g. L.D) prennent un cycle d'exécution chacune. Dans ces unités en pipeline, les additions et soustractions ont une profondeur de pipeline de 7, les multiplications de 12, les divisions de 18 et les chargements et rangements de 10. Calculez le temps requis pour l'exécution de la séquence d'instructions suivante. **(1 point)**

```

L.D          F0, Ra
L.D          F1, Rb
LV           V1, Rc
MULVS.D V1, V1, F0
LV           V2, Rd
MULVS.D V2, V2, F1
ADDVV       V3, V1, V2
MULVS.D V3, V3, F0
SV           V3, Re

```

*Les deux premières instructions prennent 1 cycle chacune. Les deux instructions suivantes peuvent se chaîner en  $10 + 12 + 64 = 86$  cycles. On ne peut chaîner le LV qui suit car l'unité de load / store est déjà occupée. Les 3 instructions suivantes peuvent aussi se chaîner en  $10 + 12 + 7 + 64 = 93$  cycles. Il faut arrêter là car l'instruction MULVS.D qui suit requiert l'unité de multiplication déjà occupée par le MULVS.D précédent. Les deux dernières instructions font une dernière chaîne et prennent  $12 + 10 + 64 = 86$  cycles. Le total est donc  $2 + 86 + 93 + 86 = 267$  cycles.*

- c) Les GPU sont constitués de plusieurs processeurs SIMD, chacun étant constitué de plusieurs éléments simples de calcul (thread processor). Est-ce que différents processeurs SIMD peuvent travailler sur différents programmes ou fonctions? Est-ce que différents éléments simples de calcul d'un même processeur SIMD peuvent travailler sur différents programmes ou fonctions? Expliquez. **(1 point)**

*Différents processeurs SIMD peuvent effectivement travailler sur différents programmes ou fonctions. Par contre, et c'est la définition même de l'acronyme SIMD, différents éléments de calcul d'un processeur SIMD travaillent tous sur la même instruction en même temps. En cas d'exécution conditionnelle (if) chaque élément de calcul passera par les deux branches de la condition mais ses opérations seront désactivées lorsqu'il sera dans la branche pour laquelle son élément de donnée ne satisfera pas la condition.*

## Question 2 (5 points)

- a) Le programme suivant calcule PI par la méthode de Monte-Carlo. La fonction `srand48` initialise le générateur de nombres aléatoires alors que la fonction `drand48` fournit un nombre point flottant (double) entre 0 et 1. Ainsi, ce programme génère un point dans le carré -1 à 1 (longueur de 2), en `x` et `y`, et détermine si ce point est à l'intérieur du cercle de rayon 1, ce qui devrait être le cas dans une proportion de PI (aire du cercle) / 4 (aire totale du carré). En faisant la somme du nombre de fois où c'est à l'intérieur, divisé par le nombre d'essais, on doit approximer PI / 4 et donc PI en multipliant ce résultat par 4. On vous demande de convertir ce programme en fonction kernel OpenCL qui fait un travail équivalent. Vous pouvez supposer que les fonctions `srand48` et `drand48` et les points flottants *double* sont disponibles en OpenCL. Vous pouvez laisser certaines opérations simples être faites par le programme qui appelle cette fonction, par exemple le calcul final de pi. Expliquez quels sont les paramètres d'entrée et de sortie de votre fonction kernel et comment vous suggérez de grouper le travail en *work item* et en groupes. (3 points)

```
int i, inside, nb = 100000000;
double x, y, pi;
long sum_pi = 0;

srand48(0);
for(i = 0; i < nb; i++) {
    x = drand48() + drand48() - 1.0;
    y = drand48() + drand48() - 1.0;
    inside = sqrt(x*x + y*y) <= 1 ? 1 : 0;
    sum_pi += inside;
}
pi = (sum_pi * (double)4.0) / nb;
```

*Il faut faire 100M calculs. Dans ce cas-ci, il n'y a pas de dépendance entre les différents calculs, il faut simplement faire une réduction à la fin pour grouper tous les résultats. On a donc intérêt à faire beaucoup de travail dans chaque work item, ce qui est plus efficace, tout en s'assurant d'avoir assez de work item pour bien occuper tous les éléments de calcul. Souvent, on recommande d'avoir plusieurs items par élément de calcul de sorte que, lorsqu'un item est bloqué par une faute de cache, un autre peut progresser. Une carte typique contient 1000 ou 2000 éléments de calcul. Avec 10000 work item, chacun sera bien occupé. On peut donc demander à chaque work item de faire 10000 itérations. Ainsi, le kernel ne requiert pas de donnée en entrée mais doit accumuler son résultat en sortie. Chacun des 10000 work item fait un `atomic_add` dans la variable `local_sum_pi` partagée par le groupe. Ensuite, le premier élément du groupe (`get_local_id(0) == 0`) ajoute cette valeur avec un `atomic_add` à la somme globale. Il ne restera plus au programme principal qu'à multiplier la somme globale par 4 et la diviser par 100 000 000.*

```
/* A la toute fin du calcul, sum_pi devra être divisé par le total et m
```

```

__kernel void calcule_pi(__global long *sum_pi)
{
    int i, inside, nb = 10000;
    double x, y;
    long tmp_sum_pi = 0;
    __local local_sum_pi = 0;

    /* chaque thread doit avoir un seed différent sinon le calcul sera id
    srand48(get_global_id(0));
    for(i = 0; i < nb; i++) {
        x = drand48() + drand48() - 1.0;
        y = drand48() + drand48() - 1.0;
        inside = sqrt(x*x + y*y) <= 1 ? 1 : 0;
        tmp_sum_pi += inside;
    }
    atomic_add(local_sum_pi, tmp_sum_pi);
    barrier(CLK_LOCAL_MEM_FENCE);
    if(get_local_id(0) == 0) atomic_add(*sum_pi, local_sum_pi);
}

```

- b) Dans la fonction OpenCL qui suit, on veut faire une copie locale du vecteur `a`, `local_a`, afin d'accélérer la suite. Cette copie est répartie entre les *work item* du groupe, de sorte que chaque item copie sa part du vecteur ( $1024 / \text{local\_size}$  éléments). Une fois cette copie terminée, le gros des calculs, chaque calcul pouvant utiliser tous les éléments du vecteur `local_a`, doit commencer. Le chargé de laboratoire vous suggère qu'il faut ajouter un énoncé de synchronisation après la copie locale et avant les vrais calculs. Est-ce vrai? Quel énoncé devriez-vous mettre? Quel est son effet? Que serait le problème qui pourrait se produire sans cet énoncé de synchronisation? **(1 point)**

```

__kernel void compute(__global float *a, __global float *b) {
    int local_id = get_local_id(0);
    int local_size = get_local_size(0);
    __local float local_a[1024];

    /* Copie locale: chaque work item copie localement
       1024 / local_size éléments */
    for(i = local_id; i <= 1024; i += local_size) {
        local_a[i] = a[i];
    }

    /* Vrais calculs: une fois la copie de a complète,
       le calcul commence */
    ...
}

```

*Puisque chaque élément de calcul copie une partie différente du vecteur, et que chacun peut avoir à accéder tous les éléments, il faut s'assurer que la copie est terminée dans tous les*

*éléments avant de continuer. Ceci peut être fait avec une barrière locale, (pour tous les éléments de calcul du même processeur SIMD), `barrier(CLK_LOCAL_MEM_FENCE)`. Cette barrière s'assure aussi que la mémoire cache entre les différents éléments de calcul devienne cohérente en propageant les modifications. A défaut de mettre une telle barrière, certains éléments de calcul pourraient commencer leur calcul en lisant des valeurs de `local_a` qui n'ont pas encore été copiées ou pour lesquelles la copie n'a pas été propagée à l'élément de calcul.*

- c) Dans le travail pratique 2, vous avez comparé la performance de OpenMP et de OpenCL pour différents problèmes (images de différentes tailles). Pour quel genre de problème utiliseriez-vous plutôt OpenMP ou plutôt OpenCL? Donnez un exemple. **(1 point)**

*OpenMP donne accès à un nombre plus limité de processeurs mais requiert relativement peu de temps de mise en marche ou de transfert de données. Habituellement, les thread ont déjà été créés et sont en attente de travail. Sur un GPU avec OpenCL, il faut copier les données vers la carte, démarrer le calcul, et récupérer les données. Le surcoût est donc plus important. Par contre, le GPU donne accès à des milliers d'éléments de calcul qui peuvent opérer en parallèle. Pour de grandes images, le gain en parallélisme d'utiliser le GPU sera donc particulièrement avantageux. En règle générale, plus le problème est facile à paralléliser à grande échelle et demande relativement peu de communication, plus le GPU avec OpenCL sera avantageux.*

## Question 3 (5 points)

- a) La fonction suivante fait l'intégration numérique par la méthode trapézoïdale d'une fonction  $f$ . Ecrivez un programme MPI qui appelle cette fonction afin de réaliser cette intégration en parallèle sur un grand intervalle (qui sera séparé en petits intervalles à traiter sur chaque noeud). Le début et la fin de l'intervalle à traiter ainsi que l'incrément (step) vous seront fournis dans des variables globales auxquelles votre programme peut faire référence: `begin`, `end`, `step`. Votre programme doit imprimer le résultat final sur le noeud 0. **(2 points)**

```
float integre_trapeze(float begin, float end, float step) {
    float resultat = 0;
    float x;
    int i;

    resultat = (f(begin) + f(end)) / 2.0;
    x = begin;
    for (x = begin + step; x < end; x += step) {
        resultat += f(x);
    }
    resultat *= h;
    return resultat;
}
```

*Il suffit de diviser l'intervalle en autant de sous-intervalles que nous avons de noeuds, et ensuite chaque noeud s'occupe de son sous-intervalle. Cette information peut être recalculée par chaque noeud et n'a pas à être communiquée. Finalement, il faut faire une réduction pour accumuler les intégrales calculées pour chaque sous-intervalle.*

```
float begin = 0.0, end = 100.0, step = .00001;

int main (int argc, char *argv[])
{
    int size, rank, nb_interval;
    float interval_size, start_interval, result, global_result;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    interval_size = (end - begin) / size;
    start_interval = begin + rank * interval_size;

    result = integre_trapeze(start_interval, start_interval + interval_si

    MPI_Reduce(&result, &global_result, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COM

    if (rank == 0) { printf("Result = %f\n", global_result);
}
```

- b) Le programme suivant s'exécute sur 4 noeuds (MPI\_Comm\_size retourne 4). Donnez le contenu des deux lignes (in et ol) imprimées sur chaque noeud? **(2 points)**

```
int main (int argc, char *argv[])
{
    int size, rank, i, in[4], ol[4];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    for(i = 0; i < 4; i++) {
        in[i] = i*i + 2 * rank * i + rank * rank; ol[i] = 0;
    }
    MPI_Allgather(in + rank, 1, MPI_INT, ol, 1, MPI_INT,
        MPI_COMM_WORLD);
    printf("%d: in={%d, %d, %d, %d};\n", rank, in[0], in[1],
        in[2], in[3]);
    printf("%d: ol={%d, %d, %d, %d};\n", rank, ol[0], ol[1],
```

```

        o1[2], o1[3]);
MPI_Finalize();
}

0: in={0, 1, 4, 9};
0: o1={0, 4, 16, 36};
1: in={1, 4, 9, 16};
1: o1={0, 4, 16, 36};
2: in={4, 9, 16, 25};
2: o1={0, 4, 16, 36};
3: in={9, 16, 25, 36};
3: o1={0, 4, 16, 36};

```

- c) Dans le cadre du travail pratique 3, à la question 3.4, vous avez tracé un graphique de l'accélération selon le nombre de processeurs, avec une courbe pour chaque taille d'image. Décrivez l'allure de ces différentes courbes. Est-ce que l'accélération en fonction du nombre de processeurs croît plus vite pour les petites ou les grandes images? Comment expliquez-vous cela? **(1 point)**

*Etant donné le coût initial de démarrage, et de communication pour envoyer les données et recevoir les résultats, il est beaucoup plus efficace de traiter des grosses images. Pour de petites images, le facteur d'accélération sature assez rapidement lorsque le nombre de processeurs et noeuds augmente (e.g., 30 processeurs). Par contre, avec des plus grosses images, un facteur d'accélération intéressant est obtenu jusqu'à un peu plus d'une dizaine de noeuds (e.g., 128 processeurs).*

## Question 4 (5 points)

- a) L'outil Helgrind de Valgrind permet de détecter les courses, un des problèmes les plus importants dans les programmes parallèles multithread. Donnez un exemple d'accès à une variable partagée par plus d'un thread en décrivant le cas où il y a une course et le cas où des primitives de synchronisation empêchent une course. Expliquez comment Helgrind peut distinguer les deux et détecter le cas où la course est présente. **(2 points)**

*Lorsque plusieurs thread accèdent une même variable sans synchronisation, une trace des accès mémoire et des verrous montrera des accès à cette même case mémoire par deux thread sans qu'un verrou associé n'ait changé de propriétaire (relâchement par le premier thread et prise par le suivant qui veut accéder la variable). Supposons le cas d'une structure de donnée qui décrit les paramètres d'une imprimante. La fonction qui modifie cette structure de donnée prend un verrou, modifie quelques paramètres, et relâche le verrou. Si un autre thread accède à ces paramètres sans prendre de verrou et tombe en plein pendant que cette structure se fait modifier, il pourrait lire divers paramètres dont certains ont été modifiés et d'autres pas et se retrouver avec des valeurs incohérentes entre elles. S'il utilise un verrou pour ces accès, ceci ne pourra pas se produire. Dans le cas sans verrou, même si la lecture*

survient à un moment où les valeurs ne sont pas en train d'être modifiées, la trace montrera néanmoins que la même case mémoire est accédée par un second thread sans qu'un verrou n'ait été pris. La détection peut donc se faire sans avoir à tomber juste au moment rare où la course cause une erreur. Helgrind vérifie aussi pour chaque variable partagée quels sont les verrous possédés par le thread qui accède la variable. Il prend l'intersection entre les verrous possédés au moment de l'accès par les différents threads. A la fin de l'exécution, l'intersection contient le verrou associé à cet élément de donnée. Si l'intersection est vide, un des thread a fait l'accès sans détenir le bon verrou.

- b) L'outil gcov peut indiquer le nombre de fois que chaque bloc de code est exécuté. L'outil gprof fournit un profil du temps d'exécution pour chaque section du programme. Les deux peuvent donc donner une bonne indication des points chauds d'un programme. Quels sont les avantages de gprof par rapport à gcov pour évaluer le temps passé dans chaque section d'un programme? **(1 point)**

*Pour étudier le temps passé dans chaque section d'un programme, gprof présente deux avantages. Premièrement, gprof impose un surcoût moins important puisqu'il demande un échantillon à chaque milliseconde, ce qui est presque négligeable, de même qu'une valeur à chaque entrée dans une fonction, ce qui est tout de même beaucoup moins coûteux qu'à chaque entrée dans un bloc linéaire comme gcov. Deuxièmement, l'échantillonnage est basé sur le temps consommé et est donc plus précis puisqu'il tient compte du coût relatif de chaque instruction, contrairement à gcov qui ne fait que tenir un décompte des instructions exécutées.*

- c) Les outils OProfile ou perf peuvent, tout comme gprof, fournir un profil du temps d'exécution d'un programme. Quels sont les avantages de ces programmes par rapport à gprof? En quoi sont-ils plus versatiles? **(1 point)**

*Les outils comme OProfile et perf font de l'échantillonnage tout comme gprof mais sont plus versatiles car le temps (compteur de cycle) n'est qu'une des nombreuses métriques qu'ils peuvent employer. Ils peuvent ainsi produire un profil des fautes en cache, des blocages dans le pipeline et de nombreux autres paramètres.*

- d) La virtualisation permet, entre autres, de partager une machine physique entre plusieurs machines virtuelles. Ce partage est-il utile pour le calcul parallèle? En quoi les machines virtuelles peuvent-elles être utiles pour le calcul parallèle? **(1 point)**

*Le calcul parallèle est généralement constitué de processus qui font un usage intensif du CPU. Il y a donc peu d'intérêt à consolider quelques machines sur une seule machine physique par le biais de la virtualisation; il n'y a pas de temps mort et donc peu d'économie à faire en récupérant du temps mort comme avec des serveurs souvent peu occupés. Par contre, la virtualisation permet aussi de découpler le logiciel du matériel et donc aide la portabilité. Avec la virtualisation, l'utilisateur peut fournir une image configurée selon ses besoins (version du système d'exploitation et des bibliothèques). Ceci peut présenter un avantage suffisant pour compenser la perte de performance associée à la couche de virtualisation.*

Le professeur: Michel Dagenais



### **13.3 Année 2013**

Le finale de l'année 2013

**ECOLE POLYTECHNIQUE DE MONTREAL**

**Département de génie informatique et génie logiciel**

**Cours INF8601: Systèmes informatiques parallèles (Hiver 2014)**

**3 crédits (3-1.5-4.5)**

---

**CORRIGÉ DE L'EXAMEN FINAL**

**DATE: Mercredi le 18 décembre 2013**

**HEURE: 13h30 à 16h00**

**DUREE: 2H30**

**NOTE: Toute documentation permise, calculatrice non programmable permise**

**Ce questionnaire comprend 4 questions pour 20 points**

---

## Question 1 (5 points)

- a) Un programme reçoit dans un vecteur *in* des informations lues de capteurs. Il doit effectuer une conversion et ajouter une calibration *cal* spécifique à chaque capteur. Convertissez ce programme en code efficace écrit en assembleur Vectoriel MIPS. (3 points)

```
double in[N], out[N], cal[N];
int i;

for(i = 0 ; i < N ; i++) out[i] = ((in[i] + cal[i]) * 1.8 + 32);

; charger les valeurs et adresses dans des registres
        LD      R1, n
        LD      R2, in
        LD      R3, cal
        LD      R4, out
        L.D     F0, #1.8
        L.D     F1, #32
; nombre d'opérations vectorielles de 64 éléments
        DSRL    R5, R1, #6
; nombre d'opérations restant à la fin
        ANDI    R6, R1, #63
        BEQZ    R5, end ; moins de 64 éléments
loop:    LV      V1, R2
        LV      V2, R3
        ADDV.D  V3, V1, V2
        MULVS.D V3, V3, F0
        ADDVS.D V3, V3, F1
        SV      V3, R4
        ADDI    R2, R2, #64 * 8
        ADDI    R3, R3, #64 * 8
        ADDI    R4, R4, #64 * 8
        ADDI    R5, R5, #-1
        BNEZ    R5, loop
end:     MTC1    VLR, R6
loop:    LV      V1, R2
        LV      V2, R3
        ADDV.D  V3, V1, V2
        MULVS.D V3, V3, F0
        ADDVS.D V3, V3, F1
        SV      V3, R4
```

- b) Le Xeon Phi peut traiter 512 bits en parallèle, sous la forme de 16 valeurs point flottant de 32 bits, ou 8 de 64 bits. Son pipeline d'exécution comporte 12 étages mais permet

de produire un résultat de 512 bits à chaque cycle. Comment cela se compare-t-il avec l'architecture vectorielle MIPS étudiée, en termes de matériel requis, de performance et de bande passante requise venant de la mémoire? **(1 point)**

*Le Xeon Phi requiert beaucoup plus de matériel, avec l'équivalent de 8 ALU point flottant de 64 bits pour chaque processeur, contre seulement 1 pour le VMIPS. Par contre, il peut produire 8 calculs point flottant de 64 bits à chaque cycle, contre seulement 1 pour le VMIPS. Le coût de remplir le pipeline au début d'un calcul est un facteur à ne pas négliger pour le nombre réel de calculs point flottant par cycle, mais il nous manque d'information pour comparer les deux. Il est probable que le Xeon Phi puisse plus facilement enchaîner les opérations point flottant sans avoir à bloquer et attendre plusieurs cycles de latence pour remplir le pipeline. Sur la plupart des architectures vectorielles, la bande passante vers la mémoire demeure ultimement le goulot d'étranglement. Pour tirer parti de son matériel de calcul plus performant que le VMIPS, le Xeon Phi devrait avoir une bande passante vers la mémoire accrue en proportion.*

- c) Vous désirez mettre sur votre liste de cadeaux un ordinateur pour le calcul parallèle. Vous hésitez entre un co-processeur de type GPU (e.g. AMD ou NVidia) ou un Intel Xeon Phi. Quels sont les principales différences architecturales et avantages de chacun? **(1 point)**

*Les GPU sont composés de nombreux processeurs SIMD (e.g. 16), chacun venant avec plusieurs éléments de calcul (e.g. 16). Ces processeurs SIMD ont une architecture particulière, optimisée pour le traitement graphique, et leur répertoire d'instruction n'est pas visible par le programmeur et peut changer d'une génération à l'autre. Les GPU se vendent en grande quantité, étant donné leur attrait pour l'affichage des jeux. Leur prix est donc très compétitif. Cependant, ils sont découplés du processeur principal et utilisent un modèle de programmation différent. Il est donc plus difficile de les mettre en oeuvre, de les programmer, d'étudier leur performance et de les déboguer. Le Xeon Phi offre un matériel comparable en terme de puissance de calcul, avec 60 processeurs contenant chacun l'équivalent de 16 ALU 32 bits, pour un total de 960 unités de calcul. Néanmoins, cela demeure une extension naturelle à l'architecture usuelle des processeurs centraux des ordinateurs. Il s'agit d'une unité centrale de traitement de 60 coeurs qui accepte le répertoire d'instruction X86\_64 avec des instructions vectorielles ajoutées. Il est donc plus facile de le mettre en oeuvre en réutilisant les outils existants.*

## Question 2 (5 points)

- a) On veut générer un histogramme pour la valeur des pixels dans une grande image. Le code en C pour effectuer ce travail est fourni. Proposez une implémentation efficace en OpenCL qui effectue ce travail. Fournissez le code pour la fonction de type kernel correspondante. Expliquez par ailleurs les arguments fournis pour l'appel de cette fonction. **(3 points)**

```

void Histogram(unsigned char image[4096][4096],
               unsigned int histogram[256])
{
    int i, j;

    for(i = 0; i < 4096; i++) {
        for(j = 0; j < 4096; j++) {
            histogram[image[i][j]]++;
        }
    }
}

```

*Le calcul à effectuer sur cette grande quantité de données est relativement simple mais le résultat final combiné de tous ces calculs constitue un point de contention. Heureusement, il est possible de calculer un histogramme sur un sous-ensemble de l'image et ensuite de combiner ces histogrammes à l'aide d'une réduction. Une solution assez simple est d'avoir comme unité de travail le calcul d'un histogramme sur un grand nombre de points, par exemple une rangée de 4096 pixels. Ensuite, par le biais d'opérations atomiques locales, chaque unité de travail termine en combinant son histogramme avec un seul appartenant au groupe de travail. Finalement, cet histogramme est reporté sur celui global en donnant une section à transférer pour chaque unité de travail. Cette fonction de type kernel est appelée avec une seule dimension (le nombre de rangées) et une taille de 4096, chaque unité de travail prenant en compte une rangée complète. La taille du groupe de travail (nombre de rangées traitées dans un même groupe) est laissée à 0, pour être choisie par OpenCL à l'exécution.*

```

__kernel void Histogram(__global unsigned char image[4096][4096],
                       __global unsigned int histogram[256])
{
    int i;
    int row = get_global_id(0), group_size = get_local_size(0);
    int chunk_size = 256 / group_size + 1;
    int start_chunk = get_local_id(0) * chunk_size;
    int end_chunk = min(256, start_chunk + chunk_size);
    unsigned int tmp_histogram[256];
    __local unsigned int local_histogram[256];

    for(i = 0; i < 256; i++) tmp_histogram[i] = 0;
    for(i = 0; i < 4096 ; i++) tmp_histogram[image[row][i]]++;
    for(i = start_chunk; i < end_chunk; i++) local_histogram[i] = 0;
    barrier(CLK_LOCAL_MEM_FENCE);
    for(i = 0; i < 256; i++)
        atomic_add(local_histogram[i], tmp_histogram[i]);
    barrier(CLK_LOCAL_MEM_FENCE);
    for(i = start_chunk; i < end_chunk; i++)

```

```
    atomic_add(histogram[i], local_histogram[i]);
}
```

- b) Que fait la fonction `barrier(CLK_LOCAL_MEM_FENCE)`? Donnez un exemple de cas où son utilisation serait requise? **(1 point)**

*Cette fonction effectue un rendez-vous entre tous les fils d'exécution d'un groupe de travail. De plus, elle constitue une barrière mémoire garantissant que tout accès effectué après verra le résultat de tout accès effectué avant. Cette fonction est requise lorsque plusieurs éléments de calcul travaillent sur différents morceaux d'un résultat et qu'ensuite ils doivent consulter les autres morceaux produits par les autres éléments de calcul. Ceci assure en effet que chaque élément a terminé ses calculs et que toutes les modifications ont été propagées vers la mémoire locale (partagée par le groupe de travail).*

- c) Dans le travail pratique 2, vous deviez calculer une image avec le programme sinoscope en utilisant d'abord OpenMP et ensuite OpenCL. À partir de quelle superficie d'image valait-il mieux utiliser OpenCL et la carte graphique plutôt que le processeur avec OpenMP à 8 fils? Expliquez pourquoi. **(1 point)**

*L'utilisation de OpenCL requiert un certain temps de démarrage pour initialiser le contexte et la carte GPU, compiler le code OpenCL, transmettre le programme et les données à la carte et récupérer le résultat. Si le temps de calcul avec OpenMP est du même ordre de grandeur que ce temps de démarrage pour le GPU, ce n'est pas intéressant d'utiliser OpenCL. Par contre, lorsque l'image est plus grande, dans les milliers ou dizaines de milliers de pixels, le temps de calcul augmente rapidement sur les 8 processeurs avec OpenMP. La vitesse de calcul plus grande du GPU, étant donné ses centaines d'ALU, rend alors l'utilisation d'OpenCL avantageuse.*

### Question 3 (5 points)

- a) Le programme suivant doit s'exécuter en 21 processus sur autant de noeuds. Il décompose un vecteur en morceaux de taille semblable et demande à chaque processus d'ajouter un nombre à chaque élément dans son morceau, en plus de faire la somme de ses éléments. Pour chacun des 21 processus, dites combien d'éléments font partie du morceau de vecteur à traiter (`mysize` tel qu'imprimé sur chaque noeud)? Le programme bloque avant la fin et reste en attente. Quel est le problème? **(2 points)**

```
#define SIZE 100000000
float v[SIZE];

int main (int argc, char *argv[])
{
    int i, j, rank, size, offset, chunksize, mysize, extra;
    double mysum = 0.0, sum = 0.0;
```

```
MPI_Status s;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
chunksize = SIZE / size;
extra = SIZE % size;
if(rank == 0) {
    for(i=0; i < SIZE; i++) {v[i] = i; sum += v[i]; }
    offset = chunksize;
    for (i = 1; i < size; i++) {
        mysize = i < (size - extra) ? chunksize : chunksize + 1;
        MPI_Send(&offset, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
        MPI_Send(&mysize, 1, MPI_INT, i, 2, MPI_COMM_WORLD);
        MPI_Send(&v[offset], mysize, MPI_FLOAT, i, 3, MPI_COMM_WORLD);
        offset = offset + mysize;
    }
    offset = 0; mysize = chunksize;
}
else {
    MPI_Recv(&offset, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &s);
    MPI_Recv(&mysize, 1, MPI_INT, 0, 2, MPI_COMM_WORLD, &s);
    MPI_Recv(&v[offset], mysize, MPI_FLOAT, 0, 3, MPI_COMM_WORLD, &s);
    printf("%d: mysize = %d\n", rank, mysize);
}

for(i = offset; i < offset + mysize; i++) {
    v[i] = v[i] + i; mysum = mysum + v[i];
}
MPI_Reduce(&mysum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if(rank == 0) {
    for(i = 1; i < size; i++) {
        MPI_Recv(&offset, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &s);
        MPI_Recv(&mysize, 1, MPI_INT, i, 2, MPI_COMM_WORLD, &s);
        MPI_Recv(&v[offset], mysize, MPI_FLOAT, i, 2, MPI_COMM_WORLD, &s);
    }
    printf("Sum = %f\n", sum);
} else {
    MPI_Send(&offset, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    MPI_Send(&mysize, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Send(&v[offset], mysize, MPI_FLOAT, 0, 3, MPI_COMM_WORLD);
}
MPI_Finalize();
}
```

La taille de 100 000 000 divisée par 21 donne 4761904, reste 16. Les 5 noeuds de 0 à 4 inclusivement traiteront 4761904 éléments, alors que les 16 derniers, de 5 à 20 inclusivement, en traiteront un de plus, 4761905.

Au moment de retourner le résultat, le troisième élément, le contenu du vecteur *v*, est envoyé avec une valeur de *tag* de 3, alors que le processus 0 fait un *MPI\_Recv* avec un *tag* de 2. Il attend donc un message avec cette valeur 2 qui n'arrivera jamais et bloque ainsi.

- b) Le programme suivant s'exécute sur 4 noeuds (*MPI\_Comm\_size* retourne 4). Donnez le contenu des deux lignes (*o1* et *o2*) imprimées sur chaque noeud? **(2 points)**

```
int main (int argc, char *argv[])
{
    int size, rank, i, in[4], o1[4], o2[4];

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    for(i = 0; i < 4; i++) { in[i] = i + rank * rank; o1[i] = o2[i] = 0; }
    MPI_Allgather(in, 1, MPI_INT, o1, 1, MPI_INT, MPI_COMM_WORLD);
    MPI_Alltoall(in, 1, MPI_INT, o2, 1, MPI_INT, MPI_COMM_WORLD);
    printf("%d: o1={%d, %d, %d, %d};\n", rank, o1[0], o1[1], o1[2], o1[3]);
    printf("%d: o2={%d, %d, %d, %d};\n", rank, o2[0], o2[1], o2[2], o2[3]);
    MPI_Finalize();
}
```

Les processus 0 à 3 ont respectivement dans *in*: { 0, 1, 2, 3 }, { 1, 2, 3, 4 }, { 4, 5, 6, 7 } et { 9, 10, 11, 12 }. Conséquemment, *Allgather* placera dans *o1* les premières valeurs de chaque *in*. Le contenu de *o1*, { 0, 1, 4, 9 }, sera le même pour tous les processus. *Alltoall* placera dans *o2* le premier élément de chaque *in* sur le processus 0, les second éléments sur le processus 1... ce qui donnera pour les processus 0 à 3 respectivement dans *o2*: { 0, 1, 4, 9 }, { 1, 2, 5, 10 }, { 2, 3, 6, 11 } et { 3, 4, 7, 12 }.

- c) Dans le cadre du travail pratique 3, vous avez mesuré la performance des fonctions de communication de type bloquant (blocking), avec tampon (buffered) et asynchrone (async). Expliquez comment fonctionne chaque type, et décrivez et expliquez les résultats obtenus pour ces types de communication dans le cadre du travail pratique 3? **(1 point)**

Les appels bloquants ne retournent que lorsque les données sont parties et que le tampon d'envoi passé en paramètre peut être réutilisé. Ce temps, bloqué en attente, aurait pu être utilisé pour continuer des calculs pendant que le DMA copie les données de ce tampon vers la carte réseau. Les appels avec tampon (buffered) requièrent une copie supplémentaire vers un tampon temporaire, un coût additionnel en temps et en mémoire; il n'y a pas d'attente mais un surcoût léger en temps d'exécution actif



*et en mémoire. Les appels asynchrones sont normalement les plus efficaces. Il n'y a pas d'attente ni de copie mais seulement le coût d'un appel pour vérifier lorsque la communication est terminée et le tampon peut être réutilisé. La programmation avec les appels asynchrones peut cependant être légèrement plus complexe.*

*Dans le travail pratique, l'application utilisait une chaîne d'appels bloquants qui attendaient les uns après les autres. C'était donc très lent à cause de la sérialisation qui en résultait. Les appels avec tampon donnaient un meilleur résultat et les appels asynchrones, évitant une copie, donnaient le meilleur résultat.*

## Question 4 (5 points)

- a) Vous voulez évaluer le temps d'exécution pris par les différentes parties d'un programme afin de l'optimiser. Trois options s'offrent à vous: l'outil de profilage **gprof** qui instrumente les entrées de fonction et échantillonne le compteur de programme à chaque milliseconde, l'outil de trace **lttng** avec l'entrée et la sortie de chaque fonction qui ont été instrumentées, et l'outil **Perf** **tools CPU profile** qui échantillonne le contenu de la pile d'exécution à chaque milliseconde. Comparez ces trois options en termes de surcoût ajouté au programme et d'informations obtenues (complétude, fiabilité, précision...). **(2 points)**

*L'outil gprof ajoute environ 5% de temps pour compter le nombre de fois que chaque fonction est appelée et par qui, et .5% de temps pour accumuler les échantillons de compteur de programme à chaque milliseconde. Un grand tampon est requis pour l'histogramme des échantillons mais il est relativement peu rempli et plusieurs pages ne sont probablement jamais allouées. Les temps passés dans chaque fonction sont assez précis, même si une erreur associée à l'échantillonnage est toujours possible. Les temps imputés aux parents (temps passé dans une fonction et celles appelées) ne sont pas fiables. En effet, ils reposent sur l'hypothèse que chaque appel, indépendamment de sa provenance, a la même durée en moyenne. Il est très possible que les appels de différentes origines pour une même fonction soient en moyenne assez différents. Le nombre d'appels de chaque fonction, obtenu par instrumentation, est exact.*

*L'outil lttng, qui instrumente chaque entrée et sortie de fonction, fournira une information très précise et détaillée. Chaque appel pourra être analysé séparément et il demeure possible de créer toutes sortes de statistiques (temps moyen par appel, selon la provenance, temps passé dans chaque fonction ou chaque fonction et celles appelées...). Par contre, le surcoût en temps d'exécution sera considérable puisque les entrées et sorties de fonctions sont très nombreuses, un facteur de ralentissement de 2 ou plus, selon la longueur moyenne des fonctions. Un grand espace sera aussi requis pour écrire la trace sur disque.*

*L'outil CPU profile prend un peu plus de temps que .5%, puisqu'il échantillonne toute la pile plutôt que simplement l'adresse courante, mais moins que les 5% requis pour instrumenter toutes les entrées de fonction. De plus, il est facile de jouer avec ce pourcentage en ajustant la fréquence d'échantillonnage. Cette technique ne permet*

*pas de voir tous les appels effectués et ne fournit donc pas un décompte exact des appels, ou même un graphe complet des appels. Par contre, les valeurs de temps passé dans une fonction et celles appelées est plus fiable, autant que celles pour le temps passé dans chaque fonction elle-même. Cet outil représente un rapport surcoût versus information extraite particulièrement avantageux.*

- b) Comment l'outil lockdep peut-il détecter la possibilité d'un interblocage, même si celui-ci ne s'est pas produit pendant l'exécution sous analyse? **(1 point)**

*Les interblocages interviennent lorsqu'il y a un cycle dans le graphe de dépendance entre les fils d'exécution et les mutex. Une excellente stratégie pour éviter toute possibilité de tel cycle est d'imposer un ordre entre les verrous pour leur acquisition par chaque fil. L'outil lockdep note tout au long de l'exécution d'un programme l'ordre dans lequel les verrous sont acquis par chaque fil d'exécution. Chaque fois qu'un verrou est pris par un fil, un lien dirigé est ajouté entre ce verrou et le précédent acquis et encore détenu par ce fil. Ces liens s'ajoutent et un graphe est formé reliant les différents verrous. Si, à la fin de l'exécution, il existe un cycle dans le graphe, il n'y a pas d'ordre qui ait été toujours respecté pour l'acquisition des verrous. En conséquence, même si un interblocage ne s'est pas produit, ceci donne une indication qu'un interblocage aurait pu se produire avec un ordonnancement différent des actions effectuées par les différents fils d'exécution.*

- c) Quel outil permet efficacement d'estimer les fautes de cache, causées par les différentes sections d'un programme en exécution, avec un surcoût ajouté au programme extrêmement faible? Expliquez? **(1 point)**

*Les compteurs de performance peuvent générer une interruption à chaque  $n$  fautes de cache. Il est alors possible de noter l'adresse qui a causé la faute de cache. Ce coût est faible et peut être ajusté en variant la valeur de  $n$ . Les outils comme Oprofile et Perf utilisent les compteurs de performance de cette manière.*

- d) En calcul parallèle, l'efficacité du traitement est un facteur souvent primordial. Néanmoins, l'utilisation de virtualisation, par exemple avec OpenStack, est de plus en plus populaire pour le calcul parallèle, en dépit de son surcoût qui peut facilement atteindre 10%. Quel en est l'intérêt qui compense pour ce surcoût? **(1 point)**

*Pour les organisations où il existe de nombreux utilisateurs potentiels et occasionnels, le temps requis pour adapter le logiciel à l'ordinateur parallèle est un frein à l'utilisation d'une infrastructure de calcul partagée. De la même manière, si un utilisateur peut avoir à utiliser différents calculateurs parallèles, (pour le prototype, pour la recherche, pour la production, pour des calculs de plus grande dimension), la portabilité de son environnement de travail devient une considération importante. En encapsulant sa configuration dans une image OpenStack, plus un fichier de configuration OpenStack, l'utilisateur peut obtenir une grande portabilité. Ainsi, un service informatique pour le calcul parallèle peut, avec OpenStack, perdre 10% de sa puissance de calcul avec la virtualisation mais gagner en taux d'occupation de ses infrastructures, attirant plus de clients. De plus, le service informatique et ses clients sauvent beaucoup de temps en effort d'adaptation de leurs configurations logicielles.*

*Un système comme OpenStack permet de consolider plusieurs machines virtuelles peu occupées sur une seule machine physique. Cependant, ceci est peu utile pour le calcul parallèle, où chaque noeud est normalement occupé à presque 100%.*

Le professeur: Michel Dagenais

## **13.4 Année 2012**

Le final de l'année 2012

**ECOLE POLYTECHNIQUE DE MONTREAL**

**Département de génie informatique et génie logiciel**

**Cours INF8601: Systèmes informatiques parallèles (Hiver 2013)**

**3 crédits (3-1.5-4.5)**

---

**CORRIGÉ DE L'EXAMEN FINAL**

**DATE: Mercredi le 5 décembre 2012**

**HEURE: 13h30 à 16h00**

**DUREE: 2H30**

**NOTE: Toute documentation permise, calculatrice non programmable permise**

**Ce questionnaire comprend 4 questions pour 20 points**

---

## Question 1 (5 points)

- a) Un programme effectue pour chaque point la moyenne avec ses deux voisins. Convertissez ce programme en code efficace écrit en assembleur Vectoriel MIPS. Tous les éléments de  $a$  ( $a[0]$  à  $a[n + 1]$ ) sont accédés mais seulement les éléments  $b[1]$  à  $b[n]$  sont écrits. **(3 points)**

```
double a[n + 2], b[n + 2];

for(i = 1 ; i <= n ; i++) b[i] = (a[i] + a[i + 1] + a[i - 1]) / 3;

; charger les valeurs et adresses dans des registres
    LD      R1, n
    LD      R2, a      ; a[0] (i - 1)
    LD      R3, b
    ADDI    R4, R2, #8 ; a[1] (i)
    ADDI    R5, R2, #16; a[2] (i + 1)
    ADDI    R3, R3, #8 ; b[1] (i)
    L.D     F0, #3
; nombre d'opérations vectorielles de 64 éléments
    DSRL    R6, R1, #6
; nombre d'opérations restant à la fin
    ANDI    R7, R6, #63
    BEQZ    R6, end ; moins de 64 éléments
loop:  LV     V1, R2
      LV     V2, R4
      LV     V3, R5
      ADDV.D V4, V1, V2
      ADDV.D V4, V4, V3
      DIVVS.D V4, V4, F0
      SV     V4, R3
      ADDI    R3, R3, #64 * 8
      ADDI    R4, R4, #64 * 8
      ADDI    R5, R5, #64 * 8
      ADDI    R6, R6, #-1
      BNEZ    R6, loop
end:   MTC1   VLR, R7
      LV     V1, R2
      LV     V2, R4
      LV     V3, R5
      ADDV.D V4, V1, V2
      ADDV.D V4, V4, V3
      DIVVS.D V4, V4, F0
      SV     V4, R3
```

- b) Une unité centrale de traitement vectorielle est telle que décrite dans le livre et contient une de chacune des unités suivantes: addition et soustraction point flottant, multiplication point flottant, division point flottant, opérations entières, opérations logiques, et rangement et chargement. Ces unités en pipeline permettent de produire une nouvelle valeur à chaque cycle. Les instructions scalaires (e.g. L.D) prennent un cycle d'exécution chacune. Dans ces unités en pipeline, les additions et soustractions ont une profondeur de pipeline de 8, les multiplications de 10, les divisions de 20 et les chargements et rangements de 12. L'instruction de comparaison (SGTV) est traitée par l'unité d'addition et soustraction point flottant, et prend le même nombre de cycles qu'une addition ou soustraction. Calculez le temps requis pour l'exécution de chacune des deux séquences d'instructions suivantes. **(2 points)**

```
/* Version inconditionnelle */
L.D      F1, #10
L.D      F2, #2
LV       V1, R1
MULVS.D  V1, V1, F2
SV       V1, R1
```

```
/* Version conditionnelle */
L.D      F1, #10
L.D      F2, #2
LV       V1, R1
SGTVS.D  V1, F1
MULVS.D  V1, V1, F2
CVM
SV       V1, R1
```

*Les deux premières instructions prennent 1 cycle chacune, la troisième et la quatrième peuvent être chaînées en  $12 + 10 + 64$  cycles, la cinquième est seule (unité de rangement/chargement occupée dans la chaîne précédente) et prend  $12 + 64$  cycles. Le total est donc:  $2 + 12 + 10 + 64 + 12 + 64 = 164$  cycles. Dans la version conditionnelle, le temps requis est le même, à la différence que SGVT s'ajoute à la chaîne LV/MULTV, ce qui ajoute 8 cycles, et CVM ajoute 1 cycle, pour un total de 173 cycles.*

## Question 2 (5 points)

- a) Il faut calculer une version réduite d'une image. Pour ce faire, la moyenne d'une région de  $16 \times 16$  pixels est calculée et devient le pixel correspondant de l'image réduite. Le code en C pour effectuer ce travail est fourni. Proposez une implémentation efficace en OpenCL qui effectue ce travail. Fournissez le code pour la fonction de type kernel correspondante. Expliquez par ailleurs les arguments fournis pour l'appel de cette fonction et donnez l'énoncé `clEnqueueNDRangeKernel` avec ses arguments qui serait appelé pour exécuter votre fonction. **(3 points)**

```

int i, j;
float image_in[1024][1024];
float image_out[64][64]

for(i = 0; i < 1024; i++) {
    for(j = 0; j < 1024; j++) {
        image_out[i/16][j/16] += image_in[i][j] / 256;
    }
}

```

Chaque item (work item) est constitué d'une portion 16x16 de l'image qui produit un pixel en sortie. En deux dimensions, la taille du problème est de 64x64. La fonction `clEnqueueNDRangeKernel` est appelée avec deux dimensions, une grandeur totale de 64x64 et sans taille de groupe spécifiée, laissant le système décider des valeurs optimales.

```

__kernel void Thumb_Pixel(__global float image_in[1024][1024],
    __global float* image_out[64][64])
{
    int i, j;
    int k = get_global_id(1);
    int l = get_global_id(0);
    int start_i = k * 16;
    int end_i = start_i + 16;
    int start_j = l * 16;
    int end_j = start_j + 16;
    for(i = start_i ; i < end_i ; i++)
        for(j = start_j; j < end_j ; j++)
            sum += image_in[i][j];
    image_out[k][l] = sum / 256;
}

```

- b) Dans la mesure où les processeurs sur les GPGPU sont de type SIMD, qu'arrive-t-il lorsque le code OpenCL d'une fonction de type kernel, exécutée sur ces processeurs, contient des conditions (if then else) de sorte que le traitement requis peut différer d'un élément à l'autre du groupe de traitement (work group). Comment cela est-il exécuté par le processeur SIMD? Le temps de traitement est-il le plus court entre la branche if et else, le plus long, ou la somme des deux? **(1 point)**

*Le processeur SIMD doit effectuer chaque opération, qu'elle soit dans le if ou le else. Il va seulement désactiver ces opérations selon le cas pour chaque élément de donnée, afin d'obtenir le résultat voulu (if ou else). En conséquence, le temps requis est déterminé par la somme des opérations dans les deux branches.*



- c) En OpenCL, est-il possible d'appeler la fonction malloc (ou new en C++) à l'intérieur d'une fonction de type kernel? Pourquoi? **(1 point)**

*Les fonctions de type kernel ne peuvent allouer de mémoire autrement que statiquement. En effet, le nombre de registres disponible sur chaque processeur SIMD est limité et doit être connu à l'avance pour bien répartir les différents fils d'exécution.*

### Question 3 (5 points)

- a) Dans un cours de sécurité informatique, le professeur a donné un énoncé de devoir encrypté. Heureusement, il a fourni la fonction de déryption (sans la clé) et spécifié que la clé de déryption est un entier de 64 bits dont seulement les 6 octets les moins significatifs sont utilisés. Il faut donc essayer toutes les valeurs qu'il est possible de représenter avec 6 octets. Voici le programme sériel pour ce faire mais qui prendrait plus de temps que l'échéance pour la remise du devoir. Convertissez ce programme en programme MPI efficace. Les ordinateurs utilisés ont des entiers (int) de 64 bits et la taille du message chiffré est connue statiquement (LEN). Pour simplifier le programme MPI, il suffit d'imprimer le résultat trouvé sur le noeud qui trouve la bonne clé, comme dans le programme sériel. **(3 points)**

```
int key, end;
char message_chiffre[LEN];
char *message_clair;

end = 1 << (6 * 8);
for(key = 0 ; key < end; key++) {
    decrypt(message_chiffre, LEN, key, message_clair);
    if(strncmp("Devoir 1", message_clair, 8) == 0)
        printf("Devoir decode:\n%s", message_clair);
}
```

*Le noeud 0 est le coordonnateur, il divise le travail entre tous les noeuds, lui inclus.*

```
char message_chiffre[LEN];
char *message_clair;
int size, rank, end, nb_keys, ks, ke;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
if(rank == 0) read_message(message_chiffre, LEN);
MPI_Bcast(message_chiffre, LEN, MPI_CHAR, 0, MPI_COMM_WORLD);

end = 1 << (6 * 8);
```

```

nb_keys = (end / size);
ks = nb_keys * rank;
if(rank == (size - 1)) ke = end;
else ke = ks + nb_keys

for(; ks < ke; ks++) {
    decrypt(message_chiffre, LEN, ks, message_clair);
    if(strncmp("Devoir 1", message_clair, 8)
        printf("Devoir decode:\n%s", message_clair);
}

MPI_Finalize();

```

- b) Dans le cadre du travail pratique 3, le nombre des instances du programme MPI était augmenté de quelques instances jusqu'au nombre total de coeurs par ordinateur multiplié par le nombre d'ordinateurs dans la grappe. Est-ce que le délai de communication varie beaucoup entre le cas de deux instances sur le même ordinateur versus deux instances sur deux ordinateurs différents? Le délai de communication signifie-t-il nécessairement un délai équivalent perdu pour le programme parallèle? **(1 point)**

*Sur un même noeud, les messages peuvent s'échanger par mémoire partagée, ce qui est passablement plus rapide que par réseau. La bande passante de la mémoire est de plusieurs centaines de MO par seconde, alors que la bande passante du réseau est de 10MO/s ou 100MO/s selon que les prises sont à 100Mbit/s ou 1Gbit/s. Heureusement, avec des communications asynchrones, il est souvent possible de continuer à effectuer du traitement utile pendant les délais de réseau, en structurant bien son application.*

- c) La fonction suivante, rencontrée dans le cadre du travail pratique 2 contient un problème de performance sérieux. Lequel? Suggérez un correctif. **(1 point)**

```

int encode_slow_c(struct chunk *chunk)
{ int i, j, checksum = 0;
  int width = chunk->width, height = chunk->height;
  int key = chunk->key;
  char *data = chunk->data;

  #pragma omp parallel for private(i,j) reduction(+:checksum)
  for (i = 0; i < width; i++) {
    for (j = 0; j < height; j++) {
      int index = i + j * width;
      data[index] = data[index] + key;
      checksum += data[index];
    }
  }
  chunk->checksum = checksum;
}

```

```
    return 0;
}
```

*Les accès au vecteur data dans la boucle la plus interne sont à des positions non consécutives, ce qui donne une mauvaise localité de référence et cause beaucoup plus de fautes de cache. Il suffit de remplacer les trois lignes après le pragma par:*

```
for (i = 0; i < height; i++) {
    for (j = 0; j < width; j++) {
        int index = i * width + j;
```

## Question 4 (5 points)

- a) Les plus récents processeurs Intel bénéficient du support pour les tables de pages étendues (EPT) afin de mieux supporter la virtualisation. Comment cela fonctionne-t-il? Expliquez quelles opérations sur une machine virtuelle seront plus rapides ou plus lentes avec EPT? **(2 points)**

*Les tables de pages étendues permettent de supporter par matériel la double traduction de 1) adresse logique machine virtuelle à adresse physique de machine virtuelle (= adresse logique de machine physique), 2) adresse logique de machine physique à adresse physique. Ainsi, lorsque la machine virtuelle met à jour ses tables de pages, elles sont automatiquement prises en compte par le matériel. Autrement, sans EPT, l'environnement de virtualisation doit protéger la mémoire prise par ces tables de pages des machines virtuelles afin qu'une interruption soit générée lors de leur mise à jour, permettant de reporter le changement dans les tables de la machine physique et d'émuler le bon comportement. Ainsi, avec EPT, les mises à jour de tables de pages dans les machines virtuelles sont beaucoup plus rapides. Par contre, les accès en mémoire à partir des machines virtuelles, en raison de la double traduction, sont un peu plus lents. Ceci ne touche toutefois que les accès pour lesquels le TLB ne contient pas déjà la pré-traduction.*

- b) Un GPGPU récent de NVidia, le GTX580, contient 16 processeurs SIMD de 32 éléments de traitement (avec un ALU) chacun, pour un total de 512 éléments de traitement. Par comparaison, le Intel Xeon Phi qui vient d'être annoncé contient 60 processeurs avec vecteur. Sur les forums, certains amateurs prétendent qu'avec 60 processeurs plutôt que 512, le Xeon Phi n'est pas compétitif du tout. Qu'en est-il? Comment peut-on comparer leur performance pour des tâches de calcul scientifique parallèle?

**(1 point)**

*Le vecteur du Xeon Phi peut traiter en parallèle 16 mots de 32 bits. Avec 60 processeurs contenant un vecteur de 16 mots, ceci fait un total équivalent à 960 éléments de traitement, ce qui se compare avantageusement aux 512 du GPGPU. En outre, les processeurs du Xeon Phi sont cadencés à une plus haute fréquence et ne souffrent*

*pas des mêmes délais requis sur les GPU pour communiquer entre la mémoire hôte et la mémoire du GPU. Ainsi, le Xeon Phi a le potentiel d'être très compétitif avec le GTX580 et même avec les modèles améliorés de GPU qui sortiront en même temps.*

- c) L'entreprise pour laquelle vous travaillez aimerait pouvoir tirer profit des cycles inutilisés de tous les ordinateurs de bureau des employés, particulièrement lorsque ceux-ci sont absents, en réunion ou le soir. Quel logiciel pourrait faire ce travail? Est-ce que ceci constitue une grille ou un nuage? **(1 point)**

*Ceci est le scénario typique d'une grille. Le logiciel MRG de Red Hat, en particulier la portion grille basée sur Condor, est un bon logiciel pour offrir un tel service. Un nuage est typiquement utilisé pour permettre un accès flexible à une grappe d'ordinateurs dédiés à être partagés.*

- d) Dans quelles circonstances utiliseriez-vous OpenMP ou OpenCL? Donnez un exemple. **(1 point)**

*OpenMP ne peut être utilisé pour accéder à la puissance de calcul du GPGPU, OpenCL est donc le seul choix pour un GPGPU. OpenCL, bien qu'il puisse être utilisé pour un CPU multi-cœur conventionnel, est inutilement contraignant comme environnement de programmation comparé à OpenMP. Si les deux type de ressources sont disponibles, un GPGPU avec OpenCL sera intéressant s'il est facile de décomposer le problème en un très grand nombre d'unité de travail (work item) qui peuvent être traitées en parallèle de manière plus ou moins indépendante. Par exemple, sur un ordinateur sans GPU, OpenMP est préférable car plus simple d'utilisation. Pour un problème de traitement d'image, avec un grand nombre d'opérations parallèles, OpenCL sera le meilleur choix si la performance est critique et mérite un investissement plus important en programmation, et si un GPGPU est disponible.*

Le professeur: Michel Dagenais