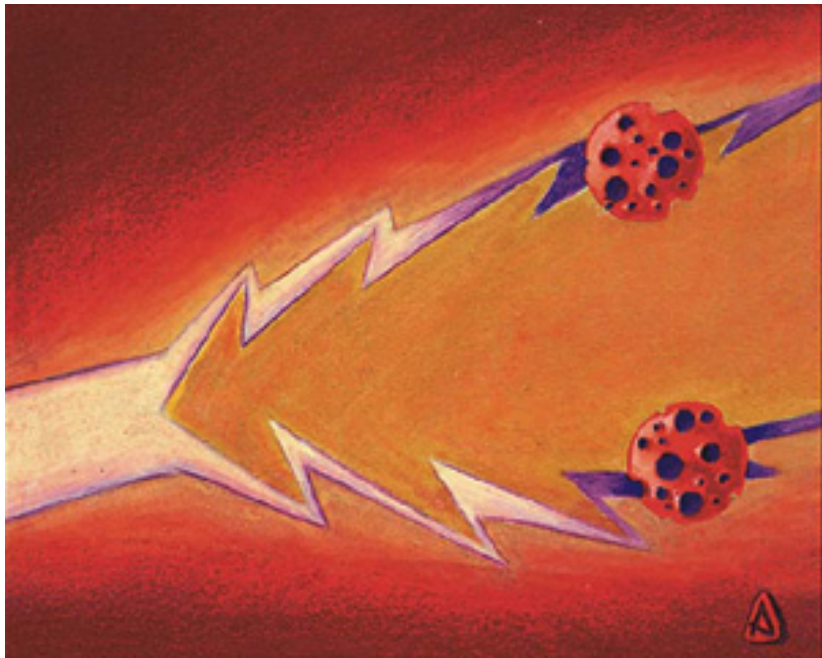


Petit guide de survie à l'attention de l'élève ingénieur plongé dans le parallélisme

Serge Guelton
d'après une œuvre originale de Ronan KERYELL

Télécom Bretagne

11–13 Décembre 2018



Pourquoi parler de performance?

L'énergie est notre avenir, économisons-la

à visionner : *from CppCon 2015: Chandler Carruth "Tuning C++: Benchmarks, and CPUs, and Compilers! Oh My!"*

Devinette #1

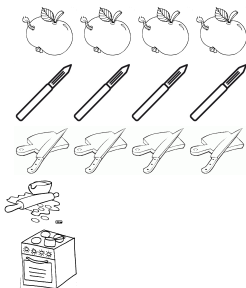
Combien de temps pour faire une tarte aux pommes

4×1 minutes

4×1 minutes

1×5 minutes

1×30 minutes



43 minutes tout seul

xx minutes à 2 ?

Devinette #2

Combien de temps pour faire une tarte aux pommes

4 × 1 minutes



4 × 1 minutes



1 × 5 minutes



1 × 30 minutes



37 minutes à 2

xx minutes à 4 ?

Devinette #3

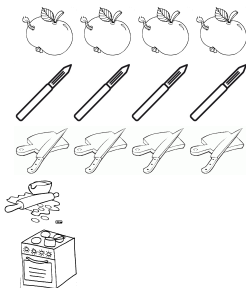
Combien de temps pour faire une tarte aux pommes

4 × 1 minutes

4 × 1 minutes

1 × 5 minutes

1 × 30 minutes



35 minutes à 4

xx minutes à 3 avec un seul couteau et un seul économe ?

Devinette #4

Combien de temps pour faire une tarte aux pommes

4 × 1 minutes



4 × 1 minutes



1 × 5 minutes



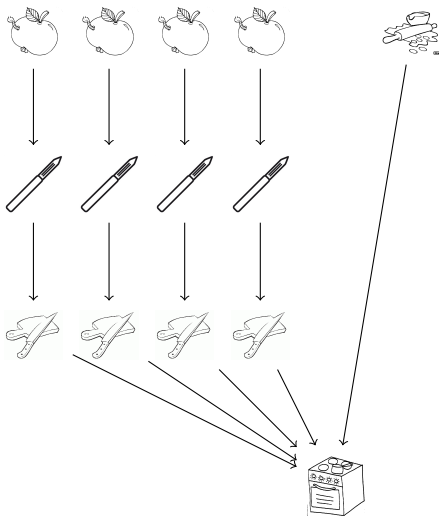
1 × 30 minutes



35 minutes à 3

et si peu de temps pour tout manger...

Formalisation



Chemin critique: $\max(\text{apple} + \text{pen} + \text{knife}, \text{sticks}) + \text{stove}$

Vu par l'informaticien

```
int main() {  
    int apples[] = {0, 1, 2, 3};  
    int segments[sizeof apples];  
    for(int i = 0; i < 4; ++i) {  
        int peeled = peel(apples[i]);  
        segments[i] = cut(peeled);  
    }  
  
    int pastry = make_pastry();  
  
    return cook(segments, pastry);  
}
```

Pourquoi ce cours ? #1

1. Comprendre les différences entre programme parallèle et programme séquentiel ;
2. Se doter de quelques outils pour formaliser un programme et raisonner dessus ;
3. Connaître le matériel et quelques modèles de programmation pour le programmer.

Pourquoi ce cours ? #1

1. Comprendre les différences entre programme parallèle et programme séquentiel ;
2. Se doter de quelques outils pour formaliser un programme et raisonner dessus ;
3. Connaître le matériel et quelques modèles de programmation pour le programmer.

Impossible en trois heures, mais possible de donner les clefs de compréhension

Sujet de Philo (vous avez 3h)

Hardware is like milk, you want it the freshest you can find. Software is like wine, you want it with a bit of age.

Exécution Séquentielle

```
int main() {  
    int apples[] = {0, 1, 2, 3};  
    int segments[sizeof apples];  
    for(int i = 0; i < 4; ++i) {           // (0, i)  
        int peeled = peel(apples[i]);     // (1, i)  
        segments[i] = cut(peeled);        // (2, i)  
    }  
  
    int pastry = make_pastry();            // (3, )  
  
    return cook(segments, pastry);         // (4, )  
}
```

C'est un ordre total

Exécution Parallèle

```
int main() {  
    int apples[] = {0, 1, 2, 3};  
    int segments[sizeof apples];  
    for(int i = 0; i < 4; ++i) {           // (0, 0)  
        int peeled = peel(apples[i]);      // (1, 0)  
        segments[i] = cut(peeled);         // (2, 0)  
    }  
  
    int pastry = make_pastry();             // (0, )  
  
    return cook(segments, pastry);         // (3, )  
}
```

C'est un ordre partiel

Challenges pour le programmeur

Et pas le prog-amateur ☺

Au niveau des concepts

1. Trouver le graphe de dépendance (difficile ! Il y a plein de dépendances cachés : système de fichier, variables statique ou globales, I/O...) ;
2. parmi les ordres possibles, en choisir un sous-ensemble qui soit efficace en plus d'être valide (localité, granularité...) $\simeq$ être bon en algorithmique parallèle

Au niveau technique

1. Choisir les bonnes abstractions pour exprimer ses choix d'ordonnancement ;
2. choisir le bon matériel pour exécuter son programme parallèle ;
3. choisir les bons langages / bibliothèque pour que ces choix puissent évoluer (nouveau matériel / nouveaux algos).

Pourquoi ce cours ? #2

Les 5 commandements

- | | |
|----------------------------|---------------------------------------|
| 1. Machines rapides | 1. potentiellement rapides |
| 2. Haut niveau | 2. ne comprenant que le bas niveau |
| 3. Compilateur intelligent | 3. mais pas plus que vous |
| 4. Modèle séquentiel | 4. sur du matériel parallèle ? |
| 5. A-B-S-T-R-A-C-T-I-O-N | 5. C-L-O-S-E- -T-O- -M-E-T-A-L |

Chemin à parcourir

Concepts de base en parallélisme

Du langage au matériel

Règles d'hygiène dans le monde du parallélisme

OpenMP

Instructions-vectorielles

Accélérateurs matériels via openCL

Autour du parallélisme

Les niveaux de parallélisme

Vus par Michael WOLFE

1. Node level $\rightsquigarrow$ calcul sur grille en mémoire distribuée
2. Socket level $\rightsquigarrow$ plusieurs accélérateurs
3. Core level $\rightsquigarrow$ plusieurs processeurs en mémoire partagée
4. Vector level $\rightsquigarrow$ dans une unité attachée au processeur
5. Pipeline level $\rightsquigarrow$ utilisation du pipeline d'instruction
6. Instruction level $\rightsquigarrow$ VLIW et autres CISC

A long way from home

Concepts de base en parallélisme

Du langage au matériel

Règles d'hygiène dans le monde du parallélisme

OpenMP

Instructions-vectorielles

Accélérateurs matériels via openCL

Autour du parallélisme

Types de parallélisme

Vu du *shell* :

SISD Single Instruction Single Data

```
$> foo 1; bar 2;
```

MIMD Multiple Instruction Multiple Data

```
$> foo 1& BAR 2&
```

SIMD Single Instruction Multiple Data¹

```
$> printf "1 2 3" | xargs -P2 -l1 foo
```

MISD Multiple Instruction Single Data

```
$> foo 1 | bar
```

¹On parle aussi de S Procedure MD

PRAM : Parallel Random-Access Machine

Definition

Machine avec un nombre quelconques de processeurs accédant en temps constant à tout point d'une mémoire globale.

- ▶ Très utile pour modéliser la complexité d'un algorithme parallèle !
- ▶ Modèle limité : coût de synchronisation, coût de communications, effets de cache. . .
- ▶ Permet d'estimer la complexité : $\mathcal{O}_{//}(\#op, \#proc)$

Algorithmes séquentiels vs. Algorithmes parallèles

Prenons quelques algos connus de la `libstdc++` et essayons les paralléliser sur une infinité de processeurs, dans le modèle PRAM

- ▶ `std::max_element`
- ▶ `std::reverse`
- ▶ `std::find_if`
- ▶ `std::sort`

sous titre : la masse d'élève s'agite et balbutie quelques propositions

Instant Animation

Sous le regard ahuri de la foule, l'encadrant sort un jeu de carte et propose quelques jeux d'algorithmes parallèles

Métriques

On mesure quoi ? Application | noyau de calcul ?

Accélération (deux implems) $a = \frac{T_A}{T_B}$

Accélération (N proc.) $a = \frac{T_1}{T_N}$

Situations idéale : $N \times T_N = T_1$

E.g.

```
$> for i in data; do foo i & done
```

Complexité de l'algo séquentielle $\mathcal{O}(|data|)$

Complexité de l'algo parallèle $\mathcal{O}_{//}(1, |data|)$

Accélération N

Timers

Utiliser une instrumentation fine ...rdtsc^{bof}, perf ou au pire
gettimeofday. valgrind --tool=callgrind introduit un biais
mais est assez fiable.

Exemple d'algo parallèle $\neq$ algo séquentiel

Calcul de préfixe

a.k.a. sommes partielles d'une séquence

$$\forall i, T_i = \sum_0^i a_i$$

Algo séquentiel

```
T = [ a[0] ] * N
for i in range(1,N):
    T[i] = T[i - 1] * a[i]
```

Complexité : $\mathcal{O}(N)$

Algo parallèle

il existe un algo en $\mathcal{O}_{//}(\log N, \frac{N}{\log N}) \dots$ à voir en TP ☺

Sublinéarité

Quand $\exists N \geq 1, T_N > T_1 \times N$

- ▶ Complexité algorithmique différente ?
- ▶ Transferts mémoires non-amortis ?
- ▶ Comportement vis-à-vis du cache hasardeux ?
- ▶ Contention sur un verrou (*data race*)?
- ▶ famine / mauvais ordonnancement
- ▶ surcout liés à l'infrastructure ?

Conditions de Bernstein

Indépendances entre deux instructions i et j

$$R_i \cap W_j = R_j \cap W_i = W_i \cap W_j = \emptyset$$

Taxonomie rapide des dépendances

$R_i \cap W_j \neq \emptyset =$ write after read

$W_i \cap R_j \neq \emptyset =$ read after write

$W_i \cap W_j \neq \emptyset =$ write after write

$R_i \cap R_j \neq \emptyset =$ read after read

Fausse dépendances !

```
a = b  
print(a)  
a = d
```

[passer en *single static assignment*]

```
for i in l:  
    k += f(i)
```

[paralléliser la réduction]

```
for i in l:  
    k = f(i)  
    a[k] = 1
```

[privatiser k] [supprimer k]

```
for i in range(n):  
    a[i] = 2 * c[i] + b[i]  
    b[i + 1] = cos(c[i])
```

[décaler l'index pour la deuxième
boucle]

Le cas des réductions

Kezako

Une réduction est un opérateur associatif appliqué de façon itérative sur une séquence. E.g. pour calculer la somme des éléments d'un tableau

- ▶ Opération non parallèle mais. . .
- ▶ si associatif, on peut faire des sommes partielles et calculer les sous arbres indépendamment

Exemple de parallélisation de réduction

Cas séquentiel

```
r=0.  
for i in l:  
    r+=i  
print r
```

Découpe en deux blocs indépendants

```
r_0 , r_1 , n=0.,0.,len(l)  
for i in l[0:n/2]  
    r_0+=i  
for i in l[n/2:n-1]  
    r_1+=i  
r=r_0+r_1  
print r
```

~ Dans le cas général, on peut faire une découpe récursive. . .

On parallélise quoi là dedans?

En fonction du matériel...

- ▶ entre instructions (AVX, pipeline d'instructions, ...)
- ▶ entre blocs d'instructions différents (tâches...)
- ▶ entre blocs d'instructions similaires (boucles...)
- ▶ entre programmes (batch...)

Eenie, meenie, minie, moe... oh, why not all of them? — Jaya Ballard, Task Mage

A long way from home

Concepts de base en parallélisme

Du langage au matériel

Règles d'hygiène dans le monde du parallélisme

OpenMP

Instructions-vectorielles

Accélérateurs matériels via openCL

Autour du parallélisme

La crise du logiciel

The software crisis

The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

— Edsger Dijkstra, The Humble Programmer (EWD340),
Communications of the ACM

Au pays des langages

Toujours plus loin...

- ▶ Langages d'assemblages
- ▶ Langages pour machine type VON NEUMANN
- ▶ Langages orientés objet (pour grosses applications)
- ▶ Composants logicielles, modèles, *frameworks* d'applications...

plus vite?

Back me up, MOORE!

Exemple pratique : Python

+++++

- ▶ Code concis
- ▶ Typage dynamique
- ▶ Conteneurs génériques
- ▶ Ramasse miette
- ▶ Orientés objet

- ▶ Machine virtuelle
- ▶ Pas de localité mémoire
- ▶ Compilateur peu efficace
- ▶ Faible support du parallélisme
- ▶ Performances pitoyables

The Good

The Free Lunch Is Over — Herb SUTTER

Plus de loi de MOORE!

On oublie les gentils

- ▶ processeurs cadencé à 3GHz+ (green lantern?)
- ▶ caches à plusieurs niveaux
- ▶ exécution dans le désordre
- ▶ modèle séquentiels

Sans modifications, les programmes tournaient plus vite !

The Bad

Le matériel d'aujourd'hui

- ▶ Multi-cœurs
- ▶ Instructions vectorielles
- ▶ Carte graphiques programmables
- ▶ Différents niveaux mémoires et/ou mémoire séparées

Sans modifications, les programmes tournent moins vite !

The Ugly

L'élève promo 2015

Dans sa boîte à outil, il a

- ▶ des langages séquentiels
- ▶ bases en programmation concurrente

Et de l'enthousiasme !

A long way from home

Concepts de base en parallélisme

Du langage au matériel

Règles d'hygiène dans le monde du parallélisme

OpenMP

Instructions-vectorielles

Accélérateurs matériels via openCL

Autour du parallélisme

Récrire le vieux code

aka algo parallèle != algo séquentiel

- ▶ Utiliser des bibliothèques spécialisées (BLAS, MKL d'Intel...)
- ▶ Utiliser des bibliothèques concurrentes (TBB d'Intel..., MPI, THRUST)
- ▶ Utiliser des langages concurrents (Cilk+ d'Intel, CUDA de Nvidia, OPENCL, GO, suivant les besoins !)
- ▶ Utiliser des frameworks parallèles / patrons de programmation (Hadoop de Apache, Celery...)

Garder le vieux code

- ▶ Question de temps
- ▶ Question de gros sous
- ▶ Ne pas succomber à la *hype* attitude

→ Calquer un modèle parallèle sur un langage séquentiel !

$\simeq$ algo parallèle == algo séquentiel

Ne jamais oublier l'ami AMDAHL

La loi d'AMDAHL

$$T_a = (1 - s) \times T + \frac{s \times T}{Acc}$$

où

T = temps d'exécution avant accélération

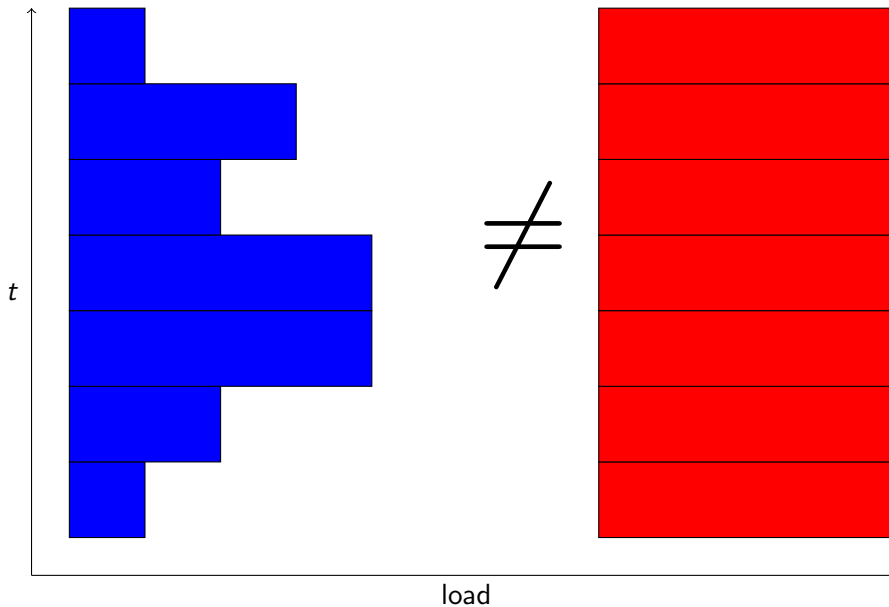
T_a = temps d'exécution après accélération

s = fraction de code concernée par l'accélération

Acc = facteur d'accélération

→ Mieux vaut un processeur à 3Ghz et 10 à 1Ghz, ou 20 processeur à 1Ghz?

En Nimage



The Memory Wall

problème: Les processeurs vont beaucoup plus vite que la mémoire
solutions:

- ▶ L-O-C-A-L-I-T-É
- ▶ masquer les transferts mémoires par du calcul (asynchrone...)
- ▶ faire attention à la taille de ses données (float vs. double ...)

Primitives de synchronisation

partielle

À base de verrous et de sémaphores

totale

À base de barrières mémoires

sans synchro !

À base d'opérations atomiques

Le C, le C++, et les autres

C : le langage pivot

Les *hardeux* l'aiment : proche de la machine, manipulation directe de la mémoire → interface avec le matériel

Les *softeux* l'aiment : avec le C++, programmation de haut niveau, (orientée objet | générique | procédurale | méta)

Message subliminal

C++14 — C11

Quelques outils de survie

Outils généraux :

- ▶ gdb (pas facile... mais pouissant !)
- ▶ valgrind+(helgrind|callgrind)
- ▶ numctl pour mémoires NUMA
- ▶ tiptop pour cache et IPC (merci Erven !)
- ▶ matière grise

Outils spécialisés (et courants) :

multi-cœurs OpenMP, TBB, OpenCL pthreads

cartes graphiques CUDA, OpenCL, OpenGL

jeux d'instructions vectoriels icc! SSE, AVX

FPGA c2h, catapultC

clusters encore et toujours MPI ...

A long way from home

Concepts de base en parallélisme

Du langage au matériel

Règles d'hygiène dans le monde du parallélisme

OpenMP

Instructions-vectorielles

Accélérateurs matériels via openCL

Autour du parallélisme

Garder le vieux code

La cible plusieurs cœurs de calcul en mémoire partagée

Le principe ajouter des annotations au code pour indiquer les boucles parallèles, les blocs d'instructions indépendants, les données partagées. . .

```
int i , j , n=1000, a [ n ] [ n ] ;  
#pragma omp parallel for private ( j )  
for ( i = 0 ; i < n ; ++ i )  
    for ( j = 0 ; j < i ; ++ j )  
        a [ i ] [ j ] = 1 ;
```

Pro & Con list

J'aime

- ▶ non intrusif
- ▶ garde la sémantique séquentielle
- ▶ incrémental
- ▶ C, C++, Fortran
- ▶ parallélisme de boucle **et** de **tâche**

J'aime pô

- ▶ Garder le code intact est une illusion
- ▶ Pas de dépendances à grain fin

Parallélisme de données

```
#pragma omp parallel for
```

Indique que la boucle peut s'exécuter en //

- ▶ l'indice de boucle et les variables locales sont `private(var)`
- ▶ les autres sont `shared(var)`
- ▶ on peut spécifier un `schedule(static|dynamic|guided,...)`
- ▶ fin bloquante sauf en cas de `nowait`

Quelques clauses de boucles

ordered

sur un bloc : garantit que la section ordonnée s'exécutera dans l'ordre séquentiel

sur une boucle : déclare que la boucle contient une section ordonnée

(first|last)private

fige la valeur à la dernière itération d'une variable privée

```
int k=42, g, r=0, i;  
#pragma omp parallel for firstprivate(k) \  
    lastprivate(i) reduction(|:r)  
for (i=0; i<n; i++) {  
    g=f(i, k);  
    r|=g;  
}  
printf("%d\n", k);
```

Contrôle de la granularité

clause `if(condition)`

exécution en parallèle si *condition* est vérifiée

clause `num_threads(k)`

exécution en parallèle sur *k* fils

Parallélisation de réductions

```
#pragma omp parallel for reduction(op:var)
```

la variable *var* porte une réduction d'opérateur *op*

```
#pragma omp parallel for reduction(+:a)  
for (int i=0;i<n;i++)  
    a+=b[i];
```

Parallélisation de tâches v1

```
#pragma omp parallel section[s]
```

défini un ensemble de sections à exécuter en parallèle

```
#pragma omp parallel sections
```

```
{
```

```
    #pragma omp section
```

```
    f(0);
```

```
    #pragma omp section
```

```
    f(1);
```

```
}
```

Qui fait quoi

Le chef d'abord

`#pragma omp master` définit une section exécuté uniquement par le fil maître

Un à la fois

`#pragma omp single` définit une section exécuté uniquement par un fil, mais n'importe lequel

Synchronisations

Gros grain

`#pragma omp barrier` attends que tous les fils d'exécutions soit terminés

Grain moyen

`#pragma omp flush(var)` attends que tous les fils d'exécutions aient finis d'accéder à *var*

Petit grain

`#pragma omp critical` définit une zone protégée par un verrou

Grain fin

`#pragma omp atomic` définit une opération *thread safe*

Autour d'OpenMP

Variables d'environnement

`OMP_NUM_THREADS` [int] nombre de fils d'exécutions actifs

`OMP_SCHEDULE` [string] ordonnancement utilisé

`OMP_DYNAMIC` [bool] ordonnancement dynamique ?

`OMP_NESTED` [bool] parallélisme imbriqué ?

Préprocesseur

`#if _OPENMP > yyyymm` compilation conditionnelle

Fichier d'en-tête

`#include <omp.h>`, pour les fonctions du *runtime*

`int omp_get_num_threads()`; lit le nombre de *thread* disponibles
dans le contexte actuel

`void omp_set_num_threads()`; change le nombre de *thread*
disponibles dans le context actuel

`int omp_get_max_threads()`; lit le nombre de *thread* disponibles

`int omp_thread_num()`; identifiant du *thread* actif

`int omp_get_num_procs()`; lit le nombre de processeurs
disponibles

`int omp_in_parallel()`; status de la région courante

Plein d'autres fonctions !

- ▶ contrôle du type de parallélisme (`omp_set_dynamic,...`)
- ▶ verrous (`omp_init_lock,...`)
- ▶ mesure du temps (`omp_get_wtime`) et précision de la mesure (`omp_get_wtick`)

OpenMP 3

`#pragma omp task`

définition de tâches imbriquées exécutables tout de suite ou en différé

mais aussi `collapse(n)`, `OMP_STACK_SIZE`, ...

Un peu de lecture / sac à lien

- ▶ Les specs :
<http://openmp.org/wp/openmp-specifications>
- ▶ la mini-fiche :
www.plutospin.com/files/OpenMP_reference.pdf

OpenMP 4

RTFM :

<http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>

248 pages de bonheur !

`#pragma omp target`

Support des accélérateurs, avec mouvement de données (map, teams, distribute ...)

`#pragma omp simd`

Boucle parallèle + `omp declare simd`

mais aussi cancelation point, task depend, affinité, réductions étendues ...

Support partiel des compilos !

La suite au prochain numéro...

Concepts de base en parallélisme

Du langage au matériel

Règles d'hygiène dans le monde du parallélisme

OpenMP

Instructions-vectorielles

Accélérateurs matériels via openCL

Autour du parallélisme

A long way from home

Concepts de base en parallélisme

Du langage au matériel

Règles d'hygiène dans le monde du parallélisme

OpenMP

Instructions-vectorielles

Accélérateurs matériels via openCL

Autour du parallélisme

Point de départ

Le matériel un processeur généraliste avec unité d'instruction vectorielle

L'accélération calcul sur n scalaires en un cycle

Les contraintes calcul de type SIMD, mémoire distribuée, jeu d'opération restreint

Comment ? SSE, AVX, NEON, ALTIVEC...

Modèle d'exécution

load Charger des données dans un registre vectoriel (lent !)

work Travailler sur ces données (rapide !)

store Décharger les données depuis les registres vectoriels
(lent !)

Contraintes

- ▶ Seulement pour données contigües en mémoires
- ▶ Pas de chargement partiel
- ▶ Généralement plus rapide quand les données sont alignées
- ▶ attention au nombre de registres disponibles

Contraintes

- ▶ opérations arithmétiques et logiques
- ▶ pas d'appel de fonctions
- ▶ quelques opérations spécialisées (arithmétique saturée. . .)
- ▶ pas de contrôle de flot !

En pratique

pour SSE

Génération automatique :

```
gcc -march=native -ftree-vectorizer-verbose=2
```

- ▶ Fonctions intrinsèques ($\simeq$ fonctions)
- ▶ Types vectoriels ($\simeq$ tableaux)
- ▶ Fichier d'en-tête `xmmintrin.h`, `immintrin.h`

doc officielle : `google://intelc++intrinsicsreference`

Exemple

```
int nb_iters = N / 4;  
float * a = ...;  
for (int i = 0; i < nb_iters; ++i, a += 4)  
  _mm_store_ps(a, _mm_sqrt_ps(_mm_load_ps((__m128*)a
```

a doit être aligné en mémoire, et le nombre d'éléments un multiple de N

A long way from home

Concepts de base en parallélisme

Du langage au matériel

Règles d'hygiène dans le monde du parallélisme

OpenMP

Instructions-vectorielles

Accélérateurs matériels via openCL

Autour du parallélisme

G.P.U.

En 3+3 mots

Graphical Processing Unit

Rapide

Lent

spécialisé

rapide plusieurs centaines de cœurs de calcul

lent transferts de données par port PCI

spécialisé parallélisme de donnée (mais pas que !)

En 4+1 mots

Compute Unified Device Architecture

NVIDIA

- ▶ Un **language** bâtard entre C et C++ pour cartes graphiques N...
- ▶ stable, bien documenté, grosse base de code, bibliothèques
- ▶ environnement de développement (compilo + libs + outils de debug) gratuits
- ▶ Linux, Windows, MacOS

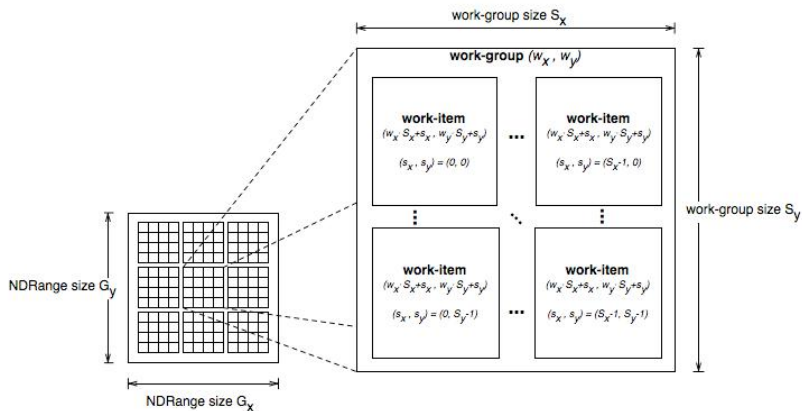
Standard pour cibles parallèles hétérogènes

- ▶ Standardisé en 2008, alternative ouverte à CUDA
- ▶ Langage dérivé du C
- ▶ Ne cible pas uniquement les GPU – plus générique que CUDA
- ▶ Implems chez Intel, AMD, Nvidia, ARM...

Nomenclature

- kernel* Une fonction qui s'exécute sur un élément de base du GPU
- thread* Unité logique d'exécution
- warp* Un multi-processeur
- host* Unité de calcul appelant le GPU, généralement un CPU, possède sa mémoire *host memory*
- device* Accélérateur, ici le GPU, avec une mémoire *device memory*

Un modèle d'exécution d'une grande simplicité



De l'influence du matériel sur le modèle de programmation

Le matériel

- ▶ Un GPU est constitué de plusieurs WARPs qui contiennent chacun plusieurs cœurs de calcul
- ▶ Chaque *warp* possède une mémoire locale (la *shared memory*)

Le modèle de programmation

- ▶ Chaque instance de *kernel* est exécutée par un *thread* sur un bloc
- ▶ L'utilisateur choisit le découpage **logique** en blocs et leurs tailles.

De l'influence du matériel sur le modèle de programmation (bis)

Le matériel

- ▶ Chaque *thread* dans le même bloc exécute la même instruction à chaque cycle, mais sur des données différentes

Le modèle de programmation

- ▶ Identification de la donnée accédée par le *kernel* à l'aide d'un jeu de coordonnées
- ▶ Attention aux boucles `while` et autres `if`

Exemple canonique — Somme de deux vecteur

Côté *kernel*

```
__kernel
void vector_add_gpu ( __global const float* src_a ,
                     __global const float* src_b ,
                     __global float* res ,
                     const int num)
{
    /* get_global_id(0) returns the ID of the thread i
    As many threads are launched at the same time, ex
    each one will receive a different ID, and consequ
    const int idx = get_global_id(0);

    /* Now each work-item asks itself: "is my ID insid
    If the answer is YES, the work-item performs the c
    if (idx < num)
        res[idx] = src_a[idx] + src_b[idx];
}
```

Exemple — Somme de deux matrices

Côté *host* L'exemple propre le plus simple fait plus de 100 lignes... un extrait

```
cl_program program = clCreateProgramWithSource
(context, 1, &source, &src_size, &error);
```

```
// Builds the program
```

```
error = clBuildProgram(program, 1, &device,
    NULL, NULL, NULL);
```

$$\begin{bmatrix} \cdot & \cdot & \cdot \end{bmatrix}$$

```
// Enqueuing parameters
```

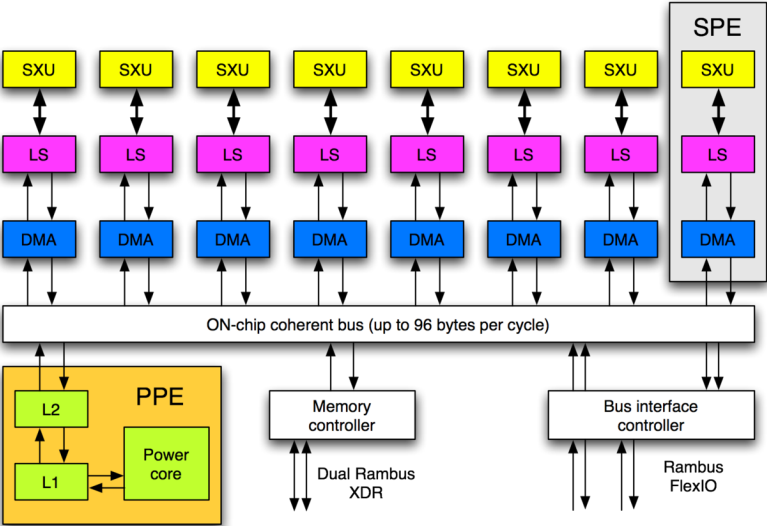
```
error = clSetKernelArg(vector_add_k, 0, sizeof
    (cl_mem), &src_a_d);
```

```
error |= clSetKernelArg(vector_add_k, 1,
    sizeof(cl_mem), &src_b_d);
```

```
error |= clSetKernelArg(vector_add_k, 2,
    sizeof(cl_mem), &res_d);
```

```
error |= clSetKernelArg(vector_add_k, 3,
    sizeof(size_t), &size);
```

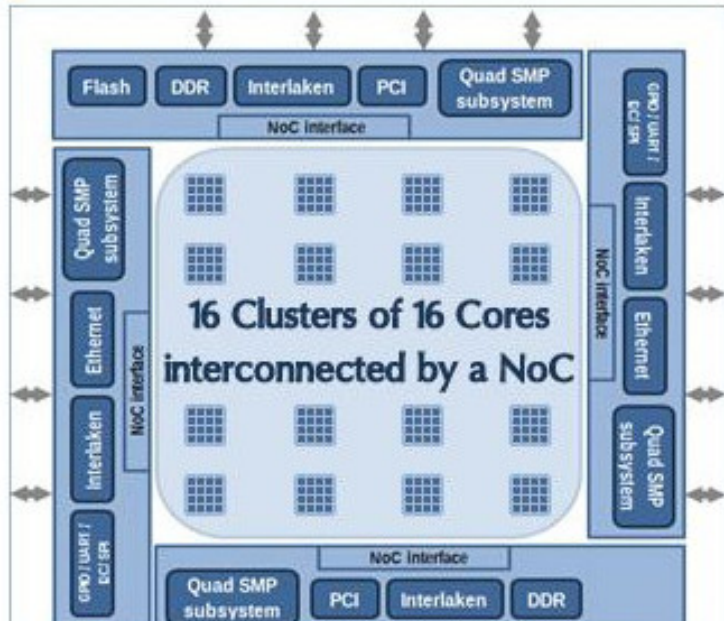

Musée des horreurs — Cell



Musée des horreurs — Bulldozer



Musée des horreurs — Kalray/MPPA



A long way from home

Concepts de base en parallélisme

Du langage au matériel

OpenMP

Instructions-vectorielles

Accélérateurs matériels via openCL

Autour du parallélisme

Parlons Optimisation

Hoare (puis Knuth)

We should forget about small efficiencies, say about 97% of the time:

Premature optimization is the root of all evil

Parlons Optimisation

Hoare (puis Knuth)

We should forget about small efficiencies, say about 97% of the time:

Premature optimization is the root of all evil

Soyons optimistes

une ligne sur 33 peut être optimisée !

Piqûre de rappel

Optimisez à la fin !

1. Code séquentiel
2. Code parallèle (OPENMP ?)
3. Code avec mémoire global uniquement
4. Mesures de performances
5. Optimisation spécifique au matériel
6. `while(not_enough) goto 4`

À la rencontre de Joe et Stephanie

99% de Joe

Qui codent avec une vision superficielle du matériel et du parallélisme

1% de Stephanie

Qui maîtrisent les difficiles concepts liés au matériel et la parallélisation, et produisent le (lo—inter)giciel pour les Joe

Python

- ▶ numpy <http://numpy.scipy.org>
- ▶ pyCUDA <http://mathematician.de/software/pycuda>
- ▶ pyOpenCL
<http://mathematician.de/software/pyopencl>
- ▶ copperhead <http://code.google.com/p/copperhead>
- ▶ ...

C++

- ▶ Thrust <http://code.google.com/p/thrust/>
- ▶ VC
<http://code.compeng.uni-frankfurt.de/projects/vc>
- ▶ NT2 <https://github.com/MetaScale/nt2>

Autres Langages

Scientifiques

- ▶ GPUMat : CUDA pour Matlab
- ▶ R+GPU : CUDA pour R
- ▶ ...

ou

Dédiés

- ▶ GPU : CUDA, OPENCL, BROOK
- ▶ SSE : CILK+
- ▶ ...

Compilateurs

`gcc`

`-ftree-parallelize-loops, -ftree-vectorize`

`icc`

`-parallel, -vec`

Dans les deux cas, utiliser les options de diagnostic...

Compilateurs + Humains

pcc

Jeu de directives haut niveau à la OPENMP pour programmer sur accélérateur matériel

hmpp

Jeu de directives bas niveau à la OPENMP pour programmer sur accélérateur matériel

p4a

Génération automatique de code parallèle à partir d'algorithmes parallèles

Chemin parcouru

Concepts de base en parallélisme

Du langage au matériel

Règles d'hygiène dans le monde du parallélisme

OpenMP

Instructions-vectorielles

Accélérateurs matériels via openCL

Autour du parallélisme

Conclusions

Parallélisme et gros sous

- ▶ L'industrie a déjà prit le tournant
- ▶ Gros effort scientifique (passé, présent, à venir)
- ▶ Inertie forte des codes existants

Parallélisme et éducation

- ▶ Sujet à la pointe à TB il y a 20 ans !
- ▶ Sujet tolérant mal la médiocrité (ErKatm)
- ▶ Sujet très mouvant, guidé par le matériel

Pour expliquer son activité à sa grand mère

pipeline une bretonne pour 3 *bilig*

parallélisme de tâche la recette de la tarte aux pommes revisitée !

parallélisme de données le travail à la chaîne ... pour les
processeurs :-)

transferts asynchrone au supermarché : l'un fait la queue, l'autre
fait les courses.