Ecole « Découverte du Calcul »

Performances de Codes

« *Le grimoire d'un sorcier semble facile à comprendre en comparaison de (plusieurs articles de) nos codes (et de nos coutumiers.)* »
Anatole France (revu et expurgé)

C'est quoi ?
Un « code », une « performance » ?

- Étymologie (*Trésor*, en ligne sur WWW)
  - **Code** : origine latine (Codex, tablette pour écrire)
    - « recueil de **lois** ou de règlements » (1236)
    - « système de **transcription** d'un **message** » (1866)
  - **Performance** : probablement héritage anglo-saxon
    - « **manière** de faire quelque chose, action accomplie » (1869)
    - « **résultat** remarquable, exploit » (1867)
    - « ensemble de **possibilité optimale** d'un appareil » (1929)
- Et nous « choisissons » quoi ?
  - **Code** : les deux :-)
  - **Performance** : les trois :-)

Qui suis-je ?
Un ingénieur à l'interface...

- En physique des particules :
  - Théoriciens : Glashow, Weinberg, ...
  - Expérimentateurs : Charpak, ...
- En informatique : (dans une moindre moindre échelle)
  - Théoriciens : LIP à l'ENS-Lyon
  - Expérimentateurs : Centre Blaise Pascal, PSMN
- CBP : Maison de la simulation
  - Plateaux techniques multi-nœuds,multi- cœurs, GPU, ...
- Émergence de nouveaux concepts/outils/approches :
  - Et leur exploitation !

C'est quoi ?
Un « code », une « performance » ?

- De la cuisine :
  - On a des ingrédients, et on veut faire un plat !
- Dans nos usages :
  - **Simulation** : *au service (discret ?) de la théorie*
  - **Traitement** : *pour expérimentateurs « exigeants »*
  - **Visualisation** : *voir pour percevoir (et partager)*
- Une expérience numérique (**unique**)...
  - Des *recettes* : des « codes » devenant « processus »
  - Des *ustensiles* : librairies, OS, matériel, réseau, ...
  - Des *ingrédients* : modélisation, données**, ...**

C'est comment, un code ?

- Quelles « familles » ?
  - Mon code *à moi que j'ai et dont je suis fier* !
  - Code de mon chef !
    - En fait des générations précédentes avant moi
  - Code « métier »
    - Modèle « Ikea » : à « monter » soi-même
    - Modèle « Crozatier » : à utiliser directement (presque...)
- Dans toute famille, le problème se nomme *héritage*.
  - Quelles dépendances aux :
    - Aux librairies « génériques » : BLAS, Lapack, FFTw
    - Aux librairies propriétaires : Mathworks, Intel, Nvidia, AMD, ...

C'est comment, un code ?

- Quelles « familles »
  - Mon code, « pur », mais à quel point ? (petit `ldd`)
  - Mon code, la partie « sur la ligne de flottaison »
    - Sous la ligne : librairies, compilateurs, OS, matériel, réseau...
  - Mon code « à-moi-que-j'ai-écrit-tout-seul »
    - Suis-je capable de me relire ?
    - Suis-je capable de le partager ?
    - Suis-je capable de juger des fonctionnalités ? Check list ?

La « performance » : comment ? Une question d'observables

Performances sportives

- Pour faire un 100m ?
- Pour faire un marathon ?
- Pour faire du lancer de poids ?
- Pour faire un heptathlon ?

La « performance » : comment ? Une question d'objectifs !

- Placer famille & bagages dans la voiture
- Aspirer des punaises à la sortie de boîte de nuit
- Se déplacer d'un point A à un point B en ville
- Négocier les virages de la montée à *Pikes Peak*

La performance : conditionnée par l'objectif

- La vitesse : durée d'exécution (seulement ?)
- Le travail : immobilisation de ressources
- L'efficacité : utilisation optimale des ressources
- La scalabilité : efficacité du passage à l'échelle
- La portabilité : diffusion à d'autres environnements
- La maintenabilité : temps passé à la maintenance
- L'approche générale
  - Définir un critère mesurable
  - Rechercher le ou les valeurs extrémales (max ou min)

La performance : conditionnée par l'objectif

- La vitesse : durée d'exécution (seulement ?)
- Le travail : immobilisation de ressources
- L'efficacité : utilisation optimale des ressources
- La scalabilité : efficacité du passage à l'échelle
- La portabilité : diffusion à d'autres environnements
- La maintenabilité : temps passé à la maintenance
- L'approche générale
  - Définir un critère mesurable
  - Rechercher le ou les valeurs extrémales (max ou min)

La performance : la « vitesse »

- La vitesse : « *Speed, I am speed... »*
  - Durée d'exécution, mais totale
  - Usage d'un code :
    - Coût d'entrée : apprentissage du logiciel, intégration, …
    - Coût d'exploitation : maintenance, exploitation
    - Coût de sortie : remplacement par code équivalent, adaptation
  - L'optimisation (et son travers) : DD/DE > 1 : pertinent ?
    - DE : Durée d'exécution totale de mon code
    - DD : Durée passée à minimiser cette durée d'exécution
  - Pour la connaître ?
    - Outils système, outils de métrologie de langages/codes, ...
  - Et après moi ? Le déluge (des nouvelles technologies ?)

La performance : le « travail »

- Le travail : « *Time is money* »
  - En physique, le « travail » est une énergie
  - Ressources : CPU, RAM, stockage, réseau
  - En fait, Matriochka :
    - CPU : plusieurs cœurs, CU, ALU, piles, ...
    - RAM : 3 niveaux (voire 4)
    - Stockage : local, lent (NFS), rapide (GlusterFS, Lustre, ...)
    - Réseau : lent (Gigabit), rapide (InfiniBand)
- Job : immobilisation de ressources
  - Nœuds * Temps d'exécution
- Pour un code, « l'empreinte système »
  - Pour la connaître, les outils de profiling, outils système

Performance : « scalabilité »

- Passage à l'échelle :
  - Dans les tâches à réaliser
  - Dans les ressources demandées
  - Le paradoxe de la cuisine...
- Écueils :
  - Effet d'échelle (en fait, plutôt des effets de seuil)
    - Chef d'orchestre ? Quatuor, orchestre symphonique ?
  - Parallélisation indispensable
  - Quoi qu'on exécute, les ressources sont toujours finies...

Performance : « portabilité »

- Capacité de passer d'un environnement à l'autre
  - Environnement matériel :
    - Avant : MIPS, Alpha, PA-RISC, Sparc, PowerPC, x86
      - http://www.irisa.fr/caps/projects/TechnologicalSurvey/micro/PI-957-html/tableofcontents2_1.html
      - Little Endian/Big Endian
      - 32/64 bits (attention aux fausses idées)
      - In-Order/Out-of-Order Execution
    - Maintenant : seulement x86_64 ?
      - Pas seulement : accélérateurs Nvidia, AMD, Intel, ...
    - Après : ARM ?
  - Environnement logiciel (BackOffice) :
    - Librairies, Compilateurs, Systèmes
- Paradigme : portabilité ~ opposé de l'efficacité

Performance : « maintenabilité »

- Capacité d'administrer facilement
  - Compiler Gaussian...
    - Avec Gfortran...
  - Compiler les versions successives de VASP
    - Gfortran versions 4.4 4.6 4.7
    - VASP versions 4.6.36 5.2.12 5.3.3
  - Compiler Code Aster
    - Des composants exigeants des versions différentes de socles
  - Exécuter les vieux binaires
    - Recours à la virtualisation

Observable

- Observable = fonction(code,données,backoffice)
- Qu'est-ce que je maîtrise ?
  - Le code ?
    - Tout le code ?
  - Les données ?
    - L'accès aux données
  - Le back-office
    - Réseau ? Systèmes ?
- Une constante : observer perturbe...
  - Out-of-code : time et al, mais aussi le reste
  - In-Code : commande système avec timer

Time & Time

- 2 versions de « Time »
  - Dans le terminal : *build in* time
    - `time sleep 10`
    - `real  0m10.003s`
    - `user  0m0.000s`
    - `sys  0m0.000s`
  - Commande : /usr/bin/time
    - `/usr/bin/time sleep 10`
    - `0.00user 0.00system 0:10.00elapsed 0%CPU (0avgtext+0avgdata 640maxresident)k`
    - `0inputs+0outputs (0major+207minor)pagefaults 0swaps`
- Exploiter /usr/bin/time !

Le banc de test
Le Pi Monte-Carlo

- Pourquoi ?
  - Stupide (Pi est irrationnel), Simple, Reproductible
  - Aisément parallélisable
- Comment ?
  - Itération
    - Tirage de 2 nombres aléatoires entiers
    - Transformation des nombres en flottants x et y
    - Calcul de $r=\sqrt{x^2+y^2}$
    - Test si $r<1$, incrémentation du compteur
  - Rapport entre compteur et itérations ~ Pi

Le code source, version C

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define znew  ((z=36969*(z&65535)+(z>>16))<<16)
#define wnew  ((w=18000*(w&65535)+(w>>16))&65535)
#define MWC   (znew+wnew)
#define MWCfp MWC * 2.328306435454494e-10f
#define ITERATIONS 1000000000
#ifdef LONG
#define LENGTH unsigned long
#else
#define LENGTH unsigned int
#endif
```

```
LENGTH MainLoopGlobal(LENGTH iterations,unsigned int seed_w,unsigned int seed_z) {
  unsigned int z=seed_z;   unsigned int w=seed_w;
  unsigned  long total=0;
  for (LENGTH i=0;i<iterations;i++) {
    float x=MWCfp ; float y=MWCfp ;
    // Matching test
    if ( sqrt(x*x+y*y) < 1.0f ) {
      total+=1;      }
  }   return(total); }
int main(int argc, char *argv[]) {
  unsigned int seed_w=10,seed_z=10;
  LENGTH iterations=ITERATIONS;
  float pi=(float)MainLoopGlobal(iterations,seed_w,seed_z)/(float)iterations*4;
  printf("\tPi=%.40f\n\twith error %.40f\n\twith %lu iterations\n\n",pi,
        fabs(pi-4*atan(1))/pi,(unsigned long)iterations);
}
```

Time sur PiMC

3 versions : `sqrt, if, =()?:`

- Sur 1 milliard d'itérations :
  - Test avec la racine carrée : `if (sqrt(x*x+y*y)<1.) { in+=1 }`
    - 14.32user 0.00system 0:14.34elapsed 99%CPU (0avgtext+0avgdata 588maxresident)k
    - 0inputs+56outputs (0major+194minor)pagefaults 0swaps
  - Test sans racine carrée : `if ((x*x+y*y)<1.) { in+=1 }`
    - 5.37user 0.00system 0:05.38elapsed 99%CPU (0avgtext+0avgdata 592maxresident)k
    - 0inputs+16outputs (0major+196minor)pagefaults 0swaps
  - Test et affectation : `val=(x*x+y*y)<1.?1:0 ; in+=val`
    - 5.13user 0.00system 0:05.14elapsed 99%CPU (0avgtext+0avgdata 592maxresident)k
    - 0inputs+32outputs (0major+195minor)pagefaults 0swaps

GProf

- Intégré dans toutes les bonnes distributions
- Trois opérations :
  - Compilation : `gcc -o MyEXE MyCode.c -g -pg`
  - Lancement : `./MyEXE` (et génération d'un `gmon.out`)
  - Profiling :
    - Simple, uniquement les fonctions : `gprof MyEXE gmon.out`
    - Étendu avec numéro de ligne : `gprof -l MyEXE gmon.out`

Gprof
Exemple sur Pi par Monte Carlo

gprof Pi_LONG gmon.out

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
101.08     49.41    49.41                             MainLoopGlobal
```

gprof -l Pi_LONG gmon.out

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 40.49     19.79    19.79                             MainLoopGlobal (Pi.c:53 @ 400905)
 21.59     30.34    10.55                             MainLoopGlobal (Pi.c:40 @ 4008c8)
 15.46     37.90     7.56                             MainLoopGlobal (Pi.c:39 @ 400890)
  9.09     42.34     4.44                             MainLoopGlobal (Pi.c:37 @ 400914)
  3.18     43.90     1.56                             MainLoopGlobal (Pi.c:54 @ 400918)
  2.85     45.29     1.39                             MainLoopGlobal (Pi.c:39 @ 4008c0)
  2.76     46.64     1.35                             MainLoopGlobal (Pi.c:40 @ 4008bd)
  2.65     47.93     1.29                             MainLoopGlobal (Pi.c:53 @ 4008fd)
  2.58     49.20     1.26                             MainLoopGlobal (Pi.c:39 @ 4008f9)
  0.41     49.40     0.20                             MainLoopGlobal (Pi.c:40 @ 400901)
  0.01     49.41     0.01                             MainLoopGlobal (Pi.c:53 @ 4008f6)
```

Gprof
Exemple sur Pi par MC

Pour y voir plus clair dans le code...

```
gprof -l $1 gmon.out | grep "Pi.c:" | while read LINE; do Line=$(echo $LINE | awk -F:
'{ print $2 }' | awk '{ print $1 }') ; Time=$(echo $LINE | awk '{ print $1 }') ; echo
$Time:$(sed -n ${Line},${Line}p Pi.c) ; done
```

Plusieurs lignes apparaissent plusieurs fois :

```
40.49: int inside=((x*x+y*y) < 1.0f) ? 1:0;
21.59: float y=MWCfp ;
15.46: float x=MWCfp ;
9.09: for (LENGTH i=0;i<iterations;i++) {
3.18: total+=inside;
2.85: float x=MWCfp ;
2.76: float y=MWCfp ;
2.65: int inside=((x*x+y*y) < 1.0f) ? 1:0;
2.58: float x=MWCfp ;
0.41: float y=MWCfp ;
0.01: int inside=((x*x+y*y) < 1.0f) ? 1:0;
```

Oprofile

- A compiler et installer soi même...
- Trois opérations :
  - Compilation : `gcc -o MyEXE MyCode.c -g`
  - Lancement : `operf ./MyEXE`
  - Profiling : `opannotate ./MyEXE`

Oprofile
Exemple sur Pi par Monte Carlo

Pour 10e9 itérations : `opannotate --source`

```
 30180 17.1445 :   for (LENGTH i=0;i<iterations;i++) {
 25595 14.5399 :      float x=MWCfp ;
 50380 28.6196 :      float y=MWCfp ;
 69758 39.6278 :      int inside=((x*x+y*y) < 1.0f) ? 1:0;
     1 5.7e-04 :      total+=inside;
```

Pour 10e10 itérations :

```
297647 16.9604 :   for (LENGTH i=0;i<iterations;i++) {
250329 14.2641 :      float x=MWCfp ;
505710 28.8161 :      float y=MWCfp ;
700438 39.9120 :      int inside=((x*x+y*y) < 1.0f) ? 1:0;
    15 8.5e-04 :      total+=inside;
```

GCov

- Outil directement intégré au compilateur
- Utilisation spartiate :
  - Compilation avec options : -fprofile-arcs -ftest-coverage
    - Uniquement a.out comme exécutable
    - Création d'un *.gcno
  - Lancement du a.out :
    - Génération d'un *.gcda
  - Lancement de gcov :
    - Génération d'un *.c.gcov

Gcov
Exemple sur Pi par Monte Carlo

- Effacement des anciens fichiers : `rm *.gc*`
- Compilation : `gcc -O3 -std=c99 -fprofile-arcs -ftest-coverage Pi.c`
- Exécution : `./a.out`
- Profilage : `gcov Pi.c`
- Analyse :

```
        1:   30:LENGTH MainLoopGlobal(LENGTH iterations,unsigned int seed_w,unsigned int seed_z)
1000000001:   37:   for (LENGTH i=0;i<iterations;i++) {
1000000000:   39:      float x=MWCfp ;
1000000000:   40:      float y=MWCfp ;
1000000000:   53:      int inside=((x*x+y*y) < 1.0f) ? 1:0;
1000000000:   54:      total+=inside;
        1:   59:   return(total);
```

Gcov
Exemple sur Pi par Monte Carlo

- Effacement des anciens fichiers : `rm *.gc*`
- Compilation : `gcc -O3 -std=c99 -fprofile-arcs -ftest-coverage Pi.c`
- Exécution : `./a.out`
- Profilage complet (même des blocs) : `gcov -a Pi.c`
- Analyse :

```
1000000001:   37:   for (LENGTH i=0;i<iterations;i++) {
        1:   37-block  0
1000000000:   37-block  1
1000000001:   37-block  2
1000000000:   39:      float x=MWCfp ;
1000000000:   40:      float y=MWCfp ;
1000000000:   53:      int inside=((x*x+y*y) < 1.0f) ? 1:0;
1000000000:   54:      total+=inside;
        1:   59:   return(total);
        1:   59-block  0
```

Et en Python ?
Timeit (avec de la stat' en plus !)

- **Utilisation de ipython, appel d'un programme externe !**

```
ipython
import timeit
%timeit -r 5 ./Pi_LONG
# 5 exécutions...
1 loops, best of 5: 4.91 s per loop
```

- **Utilisation classique dans un programme Python**

```
import timeit
def test():
    L = []
    for i in range(100):
        L.append(i)
if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

Options d'optimisation de compilateurs

- **-On** : Plus n est grand : plus l'optimisation est sophistiquée
  - Plus le temps de compilation est important
  - Plus la taille du code peut devenir grande Options de base :
- **-O0** : aucune optimisation
- **-O1** : optimisations visant à accélérer le code, en particulier quand il ne contient pas beaucoup de boucles
- **-O2** : O1 +
  - Déroulage de boucles (considère notamment un aliasing strict).
  - Augmente sensiblement le temps de compilation.
- **-O3** : O2 + transformation des boucles, des accès mémoire.
- http://calcul.math.cnrs.fr/Documents/Ecoles/LEM2I/Mod2/optim.pdf
- http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Optimisation -On pour Pi par MC

- Avec /usr/bin/time
  - Rien : 102.12s
  - -O0 : 101.23s 1 %
  - -O1 : 49.49s 52%
  - -O2 : 48.69s 52 %
  - -O3 : 48.80s 52 %
- Avec /usr/bin/time : operf en profilage
  - Rien : 228.36s
  - -O0 : 219.97s 4 %
  - -O1 : 107.02s 52 %
  - -O2 : 106.16s 52 %
  - -O3 : 106.14s 52 %

Optimisation -On pour Multiplication Matricielle

- Attention aux optimisations !
  - Si pas d'utilisation (sortie), pas de calcul !
- Pour une 2000x2000 en double :
  - Rien : 84.1s
  - O0 : 84.5s, 0.5 % de plus
  - O1 : 69.0s, 18 % de moins
  - O2 : 69.3s, 18 % de moins
  - O3 : 38.5s, 55 % de moins
- Le O3, c'est pour la vectorisation

Paralléliser ? Oui comme Monsieur Jourdain !

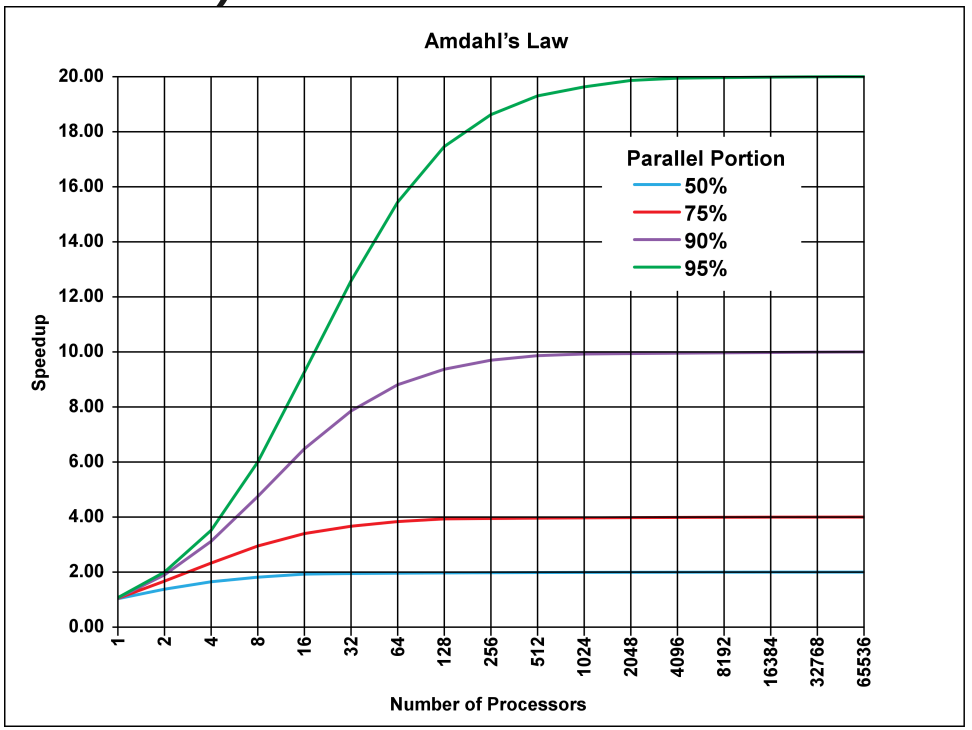
- Grain fin :
  - Exploitation de *pipeline* : plusieurs instructions / cycle
  - Unités vectorielles SSE, …, AVX : plusieurs calculs / cycle
  - Appel des registres XMM (registres vectoriels) :
    - Désassembler le code :

```
x86dis -s intel -e 0x00 -r 0x00 -1 -a 0xEEEE 2>/dev/null | grep "xmm"
```

    - O0 : 15 appels
    - O1 : 17 appels
    - O2 : 15 appels
    - O3 : 58 appels (x4)
- Grossir le grain :
  - Le cœur : OpenMP, OpenCL, Pthreads, TBB, …
  - L'accélérateur (Nvidia, AMD, Intel) : OpenCL, CUDA, la suite Intel
  - Le nœud : MPI

Performance en parallélisme
La loi d'Amdahl

- Loi simple basée sur un rapport :
  - Son exécution dure : T=T1(S+P/N)
    - P % de code //
    - S % de code séquentiel
  - Accélération : 1/(S+P/N)
    - S=0.5, 1.998 pour N=1k
    - S=0.1, 9.91 pour N=1k
    - S=0.01, 91 pour N=1k
    - S=0.001, 500 pour N=1k
  - Et l'immobilisation de ressources ?
    - 500x , 100x, 11x , 2x

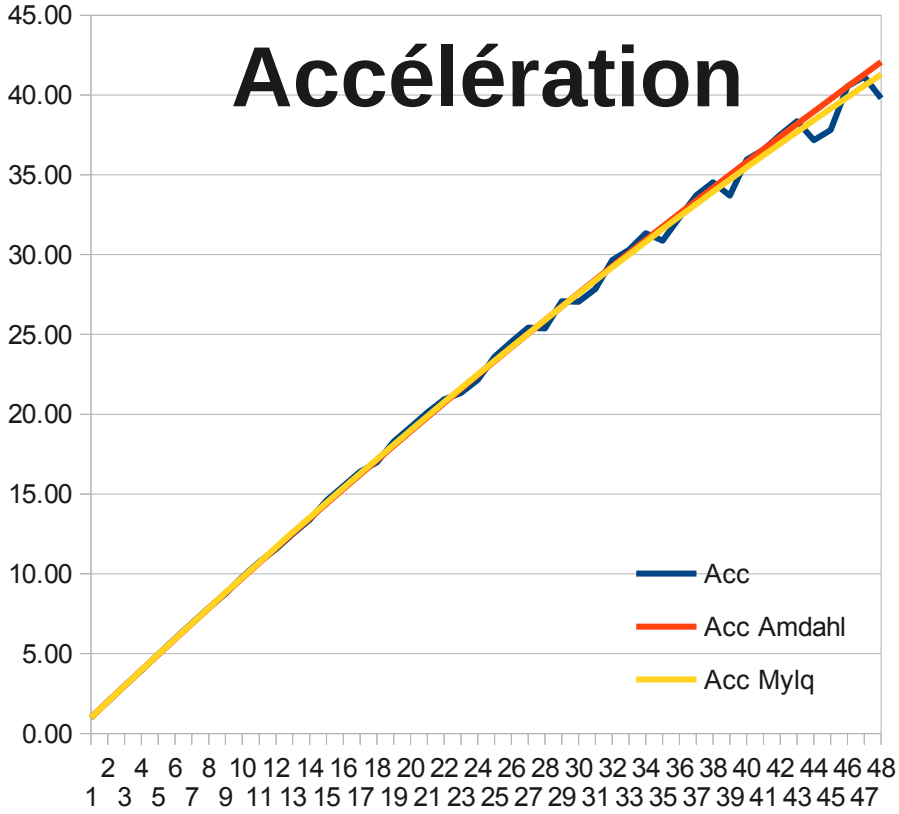


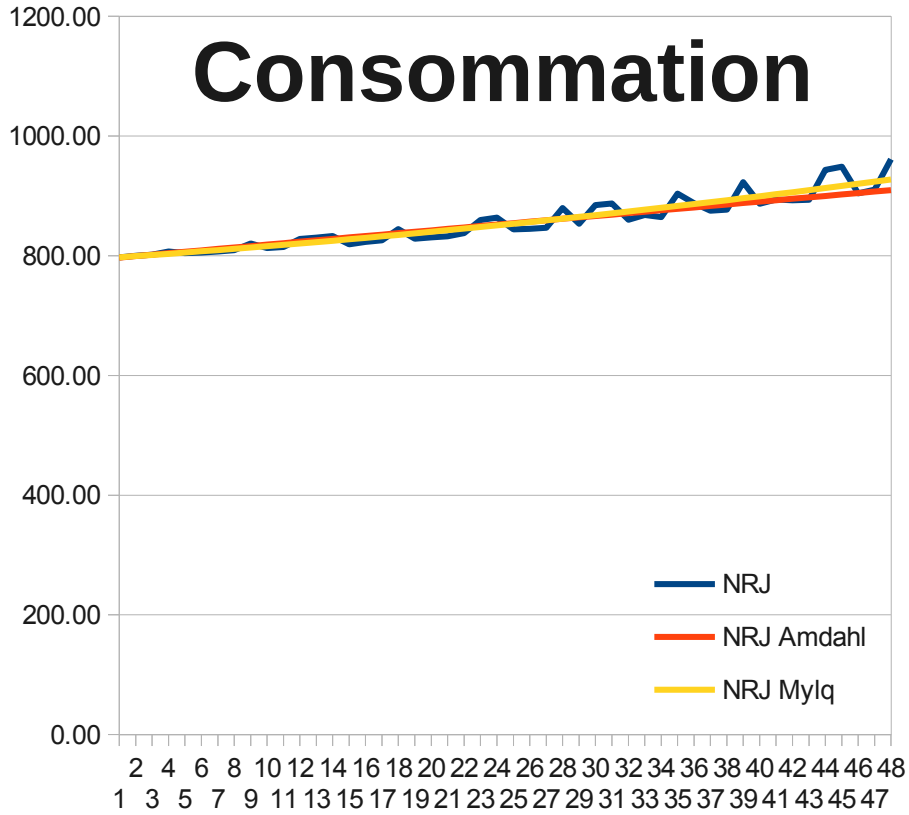
Performance en parallélisme

- Principe du parallélisme : diviser pour calculer
  1) Séparer & distribuer les tâches
  2) Exécuter les tâches sur les unités de traitement
  3) Fusionner les résultats
- Les partages :
  - Dans le temps : *pipelining* (taylorisme)
  - Dans l'espace : du vecteur au passage de message
- Deux performances au banc d'essai :
  - La durée totale du calcul
  - La consommation de ressources

Dans la « vraie vie » Loi d'Amdahl pertinente ?

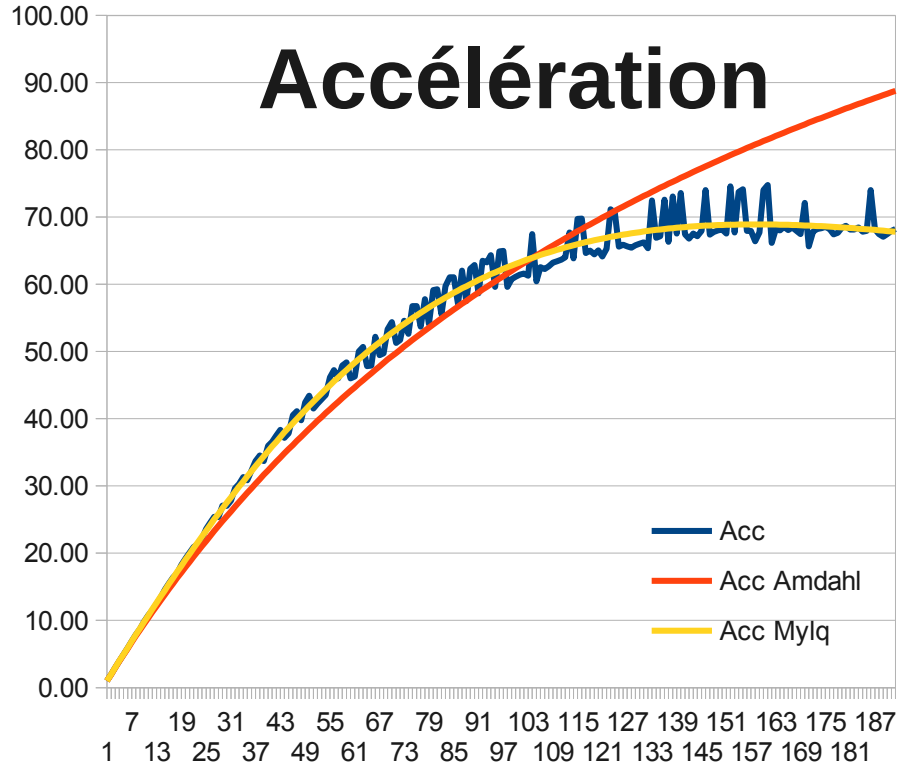
Pi Monte Carlo par MPI, 100Gitérations : de 1 à 48 nœuds, Loi d'Amdahl de 99.7 % de code parallèle

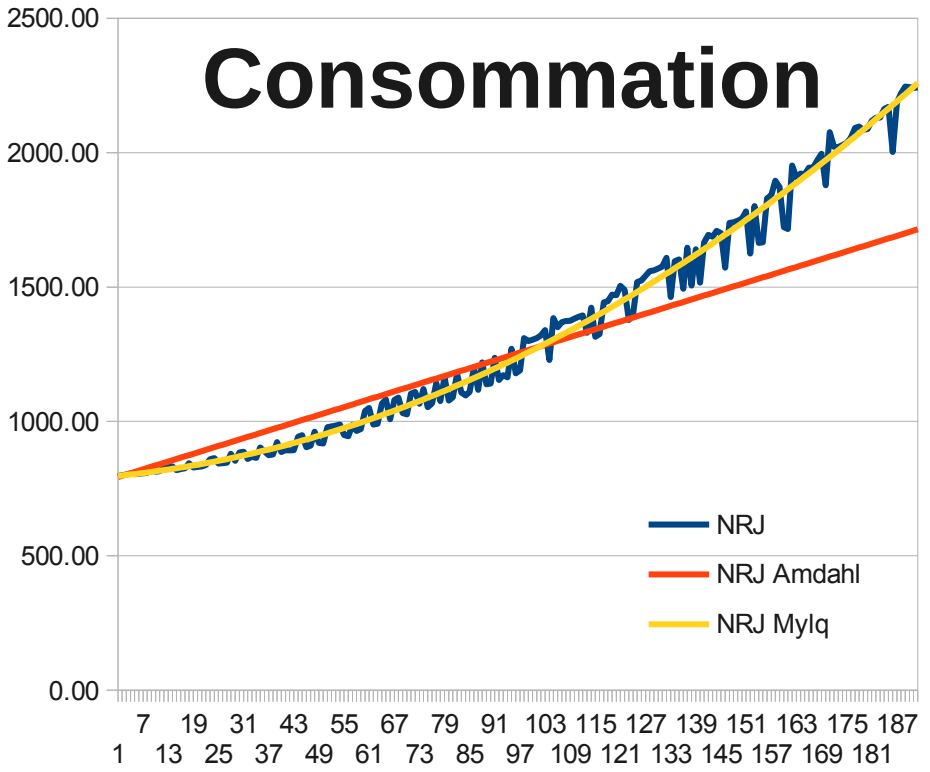




Avec encore plus de noeuds Loi d'Amdahl pertinente ?

Pi Monte Carlo par MPI, 100Gitérations : de 1 à 192 nœuds, Loi d'Amdahl de 99.3 % de code parallèle





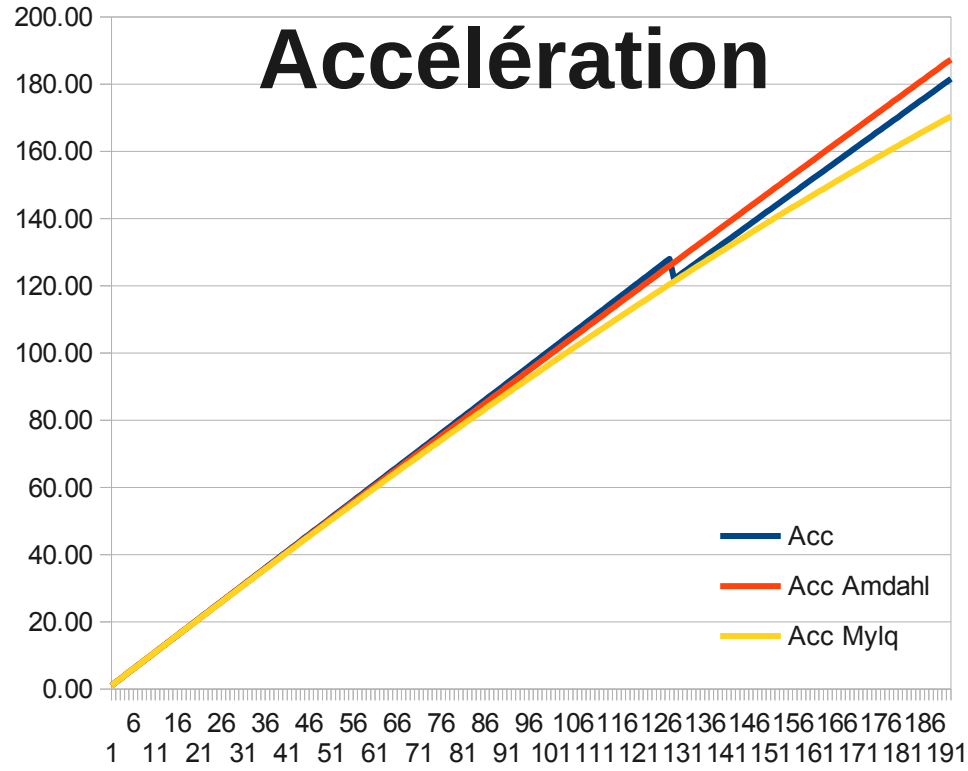
Mais quelle est donc cette loi de Mylq qui s'adapte mieux ?

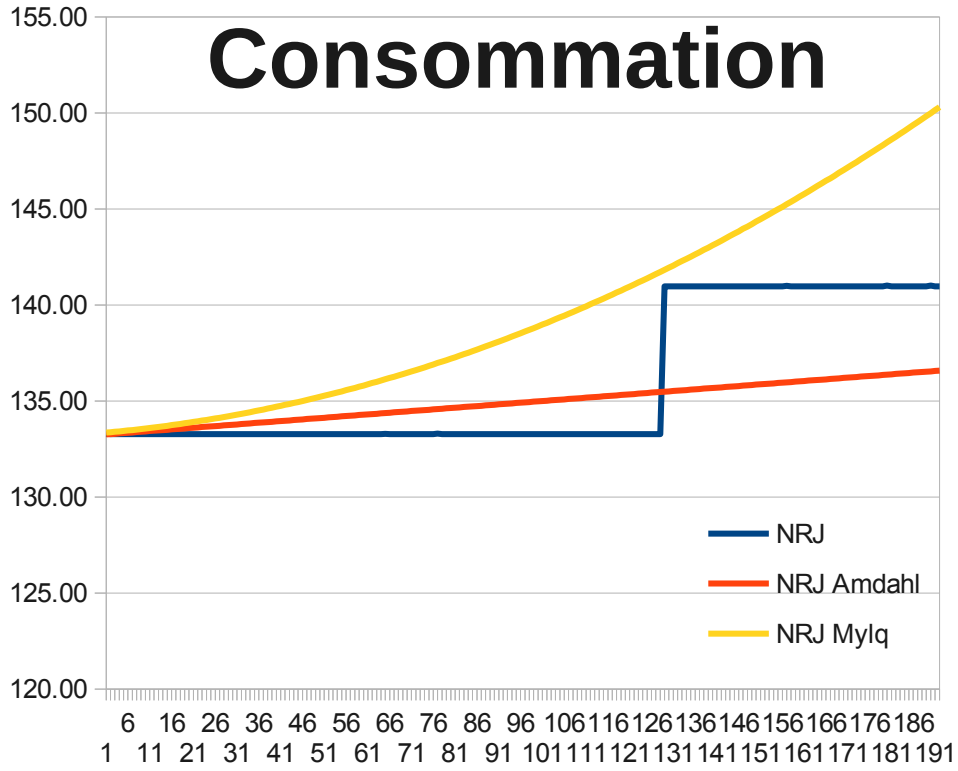
Amdahl est mort, Vive Mylq ?

- Amdahl : `T=Ts+Tp/N`
- Mylq : `T=Ts+Tp/N+Tm*N`
  - Juste une loi linéaire du nombre de cœurs
- Quelle valeur pour `Tm` ?
  - Dépendante du parallélisme
  - Déterminée par l'expérience :
    - 100Gitérations, 192 nœuds, `0.031788`
    - 1Titérations, 192 nœuds, `0.025835`
- Dans ce cas :

Et pour un autre parallélisme, sur un GPU par exemple...

Pi Monte Carlo par OpenCL, 1Gitérations : de 1 à 192 Blocks, Loi d'Amdahl de 99.98 % de code parallèle

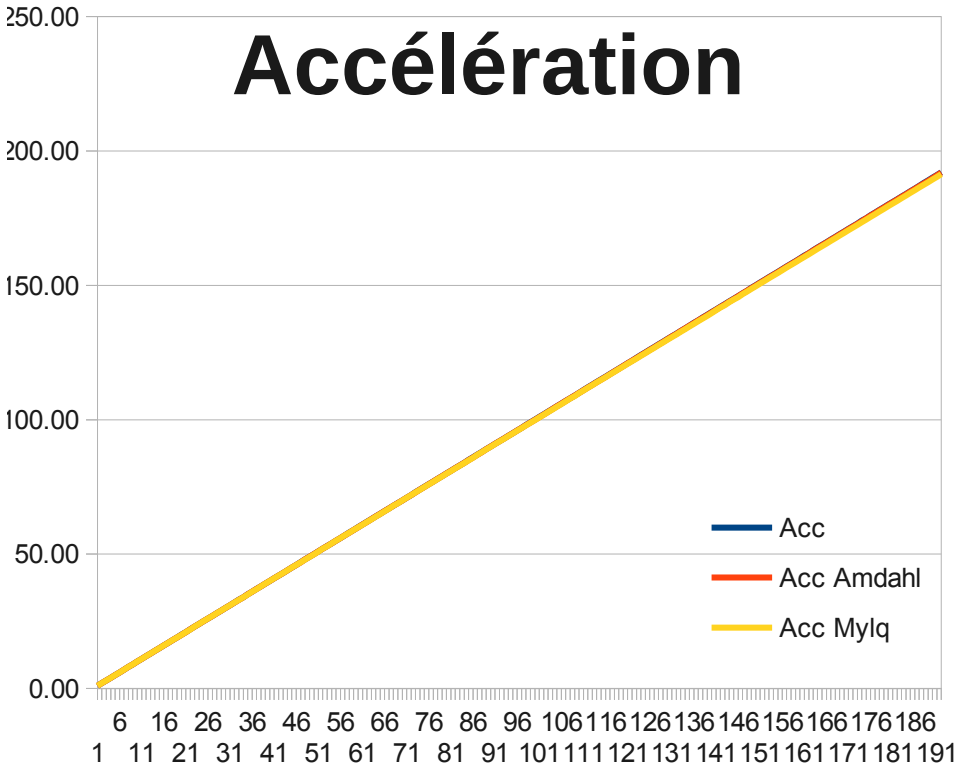


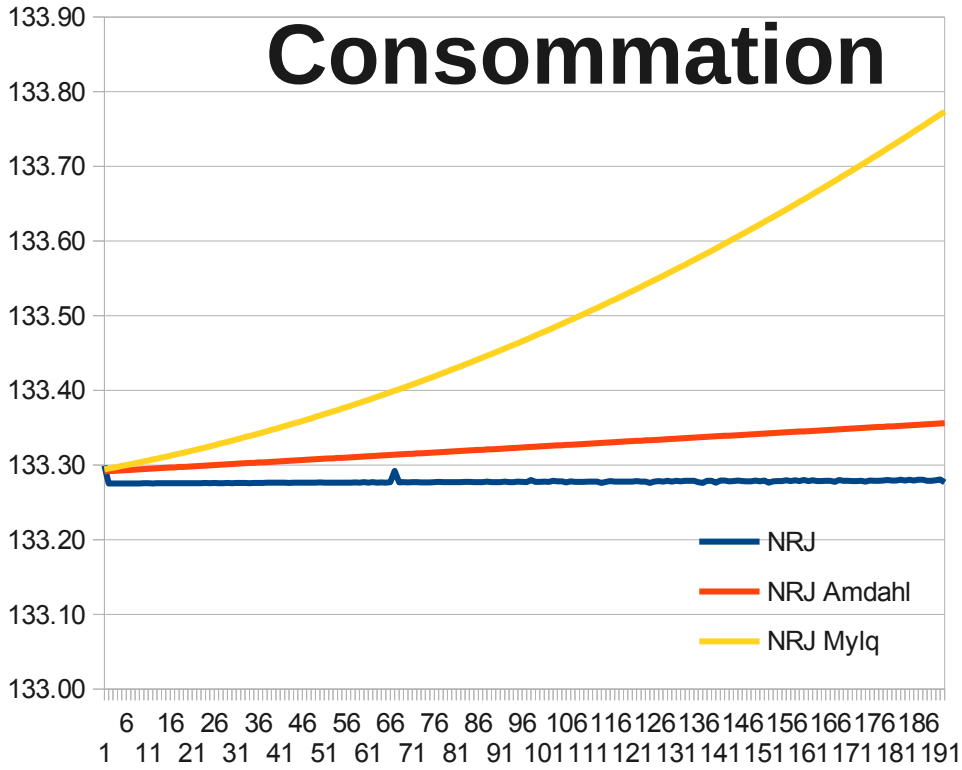


C'est quoi ce « glitch » à 128 Blocks ?

Toujours PiMC en GPU, mais avec un mix Blocks/Threads

Pi Monte Carlo par OpenCL, 1Gitérations : de 1 à 192 Threads, Loi d'Amdahl de 100.003 % de code parallèle





Quasi-linéaire, mais, en y regardant de plus près...

Comparaison MPI/CPU & OpenCL/GPU

- Parallélisme « gros grain » MPI
  - 1 nœud : 128 Mitérations/s
  - 192 nœuds : 20.4 Gitérations/s
- Parallélisme « grain fin » OpenCL sur GPU
  - 1 GPU cœur : 7.50 Mitérations/s
  - 192 GPU cœurs : 1.44 Gitérations/s
- Conclusion :
  - 1 cœur GPU : 17x plus lent qu'un nœud
  - 192 cœurs GPU : 14x plus lent que 192 nœuds
  - Mais : 192 cœurs GPU : 11 nœuds...

Optimisation en Python

- Une approche progressive
  - Premier code : démonstrateur fonctionnel
    - Ça marche !!!
    - Exploration des parties coûteuses
    - Analyse du grain de parallélisation possible
  - Second code :
    - Sans parallélisation : langage plus bas niveau
      - Cython pour commencer
      - Recodage manuel des parties coûteuses
    - Avec parallélisation : dépendance du grain
      - Gros grain : MPI pour du multi-nœuds
      - Grain moyen : librairie multiprocessing pour du multi-cœurs
      - Grain fin : passage sur OpenCL ou CUDA

Performance & Reproductibilité

- Opérations flottantes : non distributives
- Depuis années 1990, processeurs *Out-of-order*
- Sur une machine, plus de 100 processus
  - Nœud : tout le contrôle des jobs, du réseau, …
  - Station de travail : environnement graphique
- Sur les compilateurs :
  - Ordre des opérations : option fp-model de icc
  - http://www.nccs.nasa.gov/images/FloatingPoint_consistency.pdf
  - Respect des « normes » IEEE
- Finalement, entre processeurs numériques & analogiques
  - L'essentiel reste de savoir ce que nous faisons...

Conclusion

- En informatique comme ailleurs
  - Tout est expérience, toute expérience est unique
  - Toute observable perturbe l'expérience
- Une bonne performance, c'est :
  - Un code fonctionnel sur des données pertinentes
  - L'exploration par des observables multiples
- Une bonne optimisation, c'est :
  - Rarement de l'accélération pure (énergie, scalabilité, ...)
  - Toujours du parallélisme avec différentes techniques
  - Un esprit critique secondé d'une trousse à outil fournie

Iconographie

- http://en.wikipedia.org/wiki/Hanna_Melnychenko
- http://www.letsrun.com/photos/2010/newyorkcitymarathon/imagepages/image5.php
- http://fr.wikipedia.org/wiki/Shelly-Ann_Fraser-Pryce
- http://www.girlswithmuscle.com/272260/
- http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg
- http://www.motorauthority.com/image/100428458_peugeot-208-t16-pikes-peak-race-car
- http://69hereweare.files.wordpress.com/2011/01/velov-profil1.jpg
- http://fr.123rf.com/photo_12365511_voiture-familiale-pleine-de-bagages-prets-pour-les-vacances.html
- http://perlbal.hi-pi.com/blog-images/15199/gd/1174220336/voiture-tuning.jpg