Optimisation de logiciels de modélisation sur centre de calcul

Gérald Monard

Pôle de Chimie Théorique
http://www.monard.info/

Introduction

Les ordinateurs sont des appareils électroniques permettant d'effectuer des opérations de base sur les entiers et les réels (flottants). Grosso-modo, ce sont de grosses calculatrices.

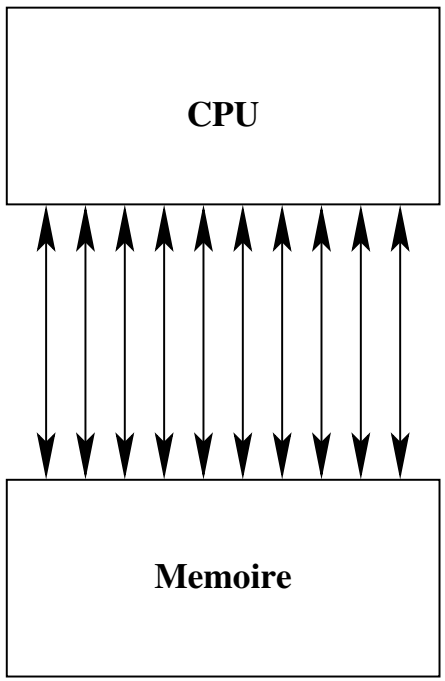
Ces calculatrices sont dorénavant partout dans notre monde: ordinateurs, mais aussi téléphones, "Internet box", cadres photos, voitures, fours, frigos, etc.

☞ tout ce qui contient un CPU (Central Processing Unit)

En Chimie Théorique: on utilise principalement les qualités de calculs des ordinateurs.

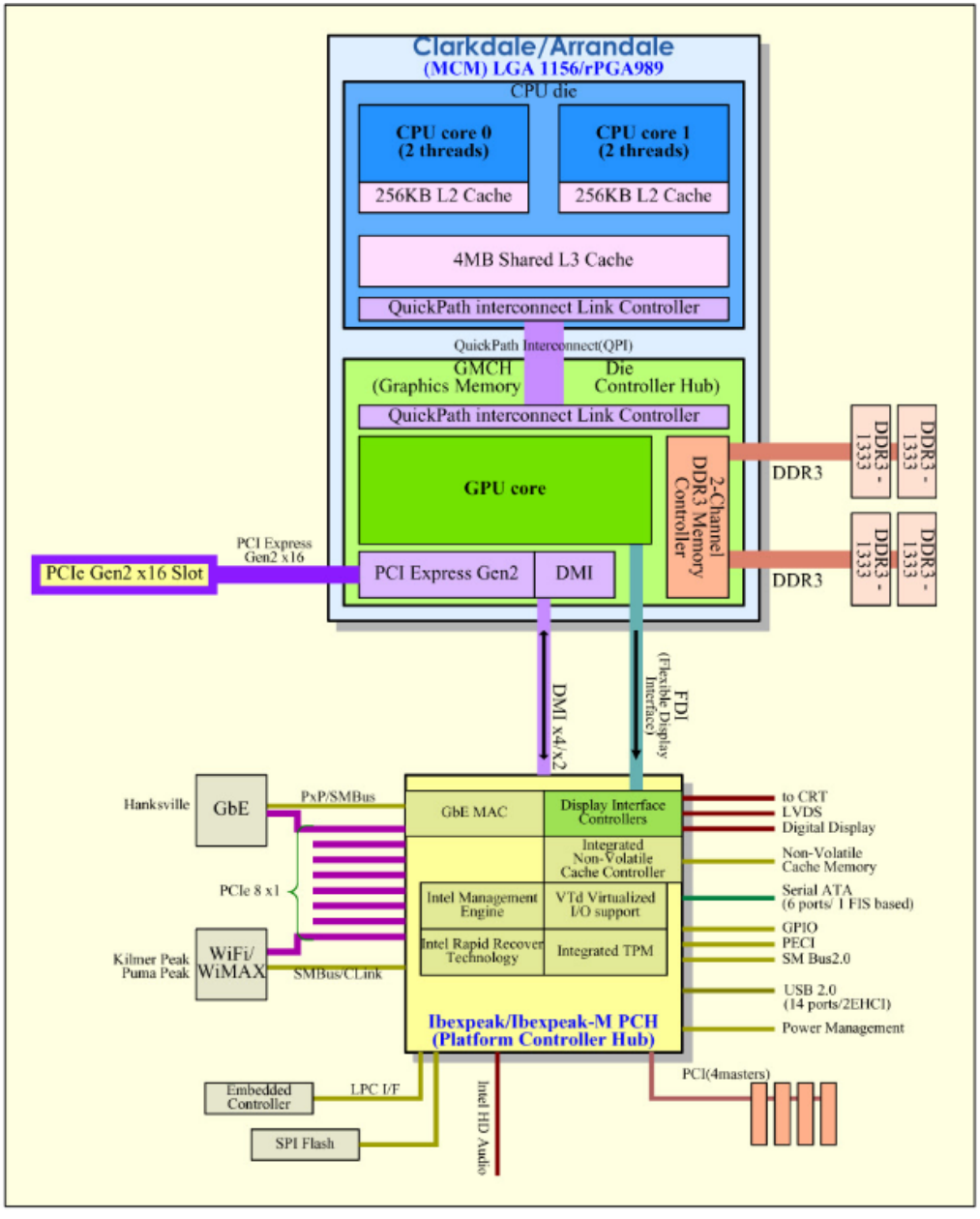
Le but: calculer des propriétés chimiques le plus rapidement possible.

Schéma général d'un ordinateur: le modèle de von Neuman



- Ordinateur = deux parties
  CPU + mémoire
- La mémoire contient les données + les programmes
- programme = liste d'instructions à exécuter en fonction des données
- instruction = donnée codée
  (calcul ou lecture/écriture dans la mémoire)
- CPU: obtient les instructions ou les données de la mémoire; décode les instructions et les exécute séquentiellement

Schéma d'un ordinateur moderne



- Plusieurs CPU
- Mémoires caches sur le processeur (L1/L2) ou entre le processeur et la mémoire centrale (L3)
- La carte graphique peut se transformer en CPU
- Données: en mémoire, sur disque interne/externe, sur le réseau

Objectifs du cours

Mieux appréhender les architectures modernes des ordinateurs séquentiels (1 CPU) ou parallèles (plusieurs CPU):

Plan (2 fois 3 heures de cours)

1. Programmation séquentielle
   a) La compilation/installation de programmes (compilation, automatisation, adaptation)
   b) Le “débuggage” de programmes
   c) Le “profiling” de programmes
   d) L’optimisation séquentielle de programmes
2. Programmation parallèle
   a) Qu’est-ce que le parallèlisme ?
   b) Les modèles de machines parallèles
   c) Les modèles de programmation parallèle
   d) Les lois fondamentales du parallèlisme
   e) Introduction à MPI
   f) Introduction à OpenMP

Bibliographie

- *UNIX pour l'utilisateur - Commandes et Langages de commandes*
  Jean-Louis Nebut (Ed. Technip)
- *Parallel Programming in C with MPI and OpenMP*
  Michael J. Quinn (Ed. McGraw-Hill)
- *High Performance Computing*
  Kevin Dowd, Charles Severance (Ed. O'Reilly)
- *Tutorials from Lawrence Livermore National Laboratory*
  `http://www.llnl.gov/computing/tutorials/`
- *Tutorials from Ohio Supercomputer Center*
  `http://www.osc.edu/hpc/training/`
- *Developping software with GNU*
  `http://www.amath.washington.edu/~lf/tutorials/autoconf/toolsmanual_toc.html`

“Parler” à un ordinateur

Un ordinateur ne reconnait que le langage binaire et les programmes binaires.

L’homme a d’énormes difficultés à parler un langage binaire !

Les langages de programmation permettent d’effectuer le lien entre l’homme et l’ordinateur : un programme est écrit dans un idiome (assez facilement) compréhensif par l’homme puis est traduit (par l’intermédiaire d’un interpréteur ou d’un compilateur) en une séquence d’instructions (binaires) directement reconnaissables par l’ordinateur.

Compilation vs. Interprêtation

Il existe deux sortes de langages : les langages interprétés et les langages compilés (= traduits).

Les langages interprétés sont directement traduits et exécutés. Les instructions sont converties séquentiellement au format binaire par un interpréteur puis exécutées immédiatement.

Les langages compilés utilisent un compilateur qui traduit le programme en langage binaire. Une fois le langage compilé, il est possible de le stocker dans un fichier exécutable. Le compilateur n'intervient qu'au moment de la création du code binaire. Le programme compilé peut s'exécuter sans avoir recours au compilateur.

On parle de code source pour désigner les instructions du programme en format texte compréhensible, et de code executable pour désigner les instructions du programme en langage binaire directement exécutable par l'ordinateur. Tout fichier source mis à jour doit être à nouveau compilé pour recréer le fichier exécutable correspondant.

Langages de programmation

Il existe de (très) nombreux langages de programmation (= de façon de parler à un ordinateur).

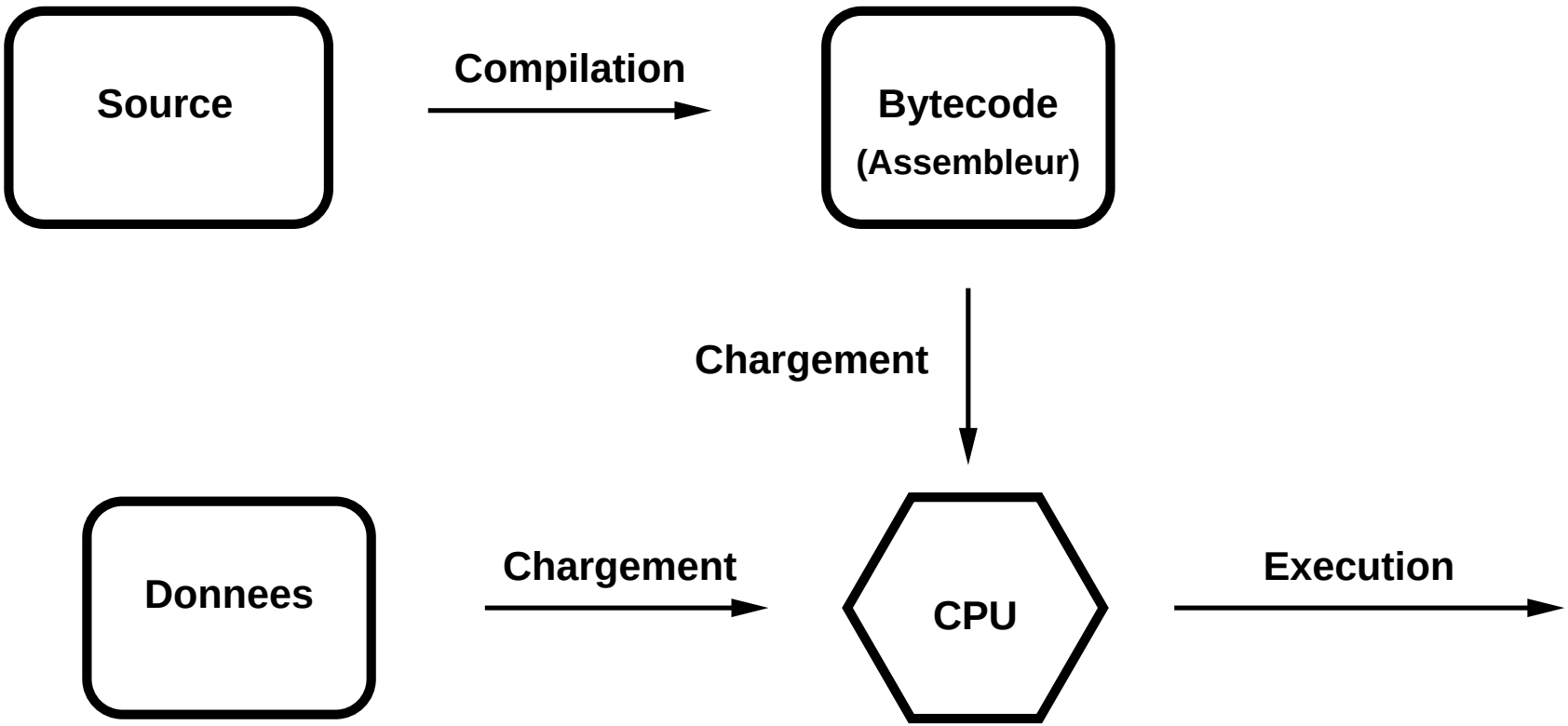
Quelques langages existants ou ayant existés:

Ada	B	C	C++	Python
Perl	Ruby	Fortran	Cobol	Java
Bash	C-Shell	OCaml	Basic	Lisp
Eiffel				

Chaque langage a ses avantages et ses inconvénients. Il n'y a pas de langage de programmation parfait, mais des langages plus ou moins adaptés à certains problèmes (calculs, web, entrée/sorties, etc.).

La compilation

L'étape de compilation permet de transformer un code source en un code binaire (bytecode) qui va pouvoir être chargé en mémoire par le CPU puis exécuté. Le résultat différera en fonction des données présentes au moment de l'exécution.



Les différentes phases de la compilation

1. Phase de précompilation (préprocesseur): manipulation textuel simple du programme
   (inclusion de fichiers, remplacement d'occurences, etc.)
2. Analyse lexicale: les instructions sont décomposés en morceaux élémentaires (variables, commentaires, constantes, éléments de langages, etc.)
3. Analyse grammaticale: la syntaxe est vérifiée puis traduit en un langage intermediaire (bytecode non optimisé)
4. Optimisation du code intermédiaire en une ou plusieurs passes
5. Traduction du code intermédiaire optimisé en code assembleur avec prise en compte des spécificité architecturales de l'ordinateur. ☞code objet
6. Edition de liens: transformation du code objet en un programme exécutable.

Exemple: le compilateur gcc

- gcc: compilateur C GNU
- Un programme C simple:

```
1 #include <stdio.h>
2 int main() {
3   printf("Bonjour\n");
4   return(0);
5 }
```

- La commande Unix: `gcc -o hello hello.c`
  `-o hello`: nomme le programme `hello` (a.out par défaut)
- Quatre phases: "preprocess, compile, assemble, link"

  -E Preprocess only; do not compile, assemble or link

  -S Compile only; do not assemble or link

  -c Compile and assemble, but do not link

Bibliothèques (libraries)

- Une bibliothèque est une collection de routines qui peuvent être utilisées dans des programmes.
- Les bibliothèques se distinguent des exécutables dans la mesure où elles ne représentent pas une application. Elles ne sont pas complètes (pas de fonction “main”).
- L’étape d’éditions de liens permet de construire des programmes à partir du code source et des bibliothèques déjà définies.
- Terminologie sous UNIX:

  toto.a bibliothèque statique: la bibliothèque est incluse (complètement) dans le programme à l’édition de liens

  toto.so bibliothèque dynamique: seule le lien vers la bibliothèque est inclus dans le programme. A l’exécution, les parties de codes de la bibliothèque seront exécutées à la demande.

Programme statique vs. programme dynamique

- Si un programme est lié à une bibliothèque dynamique, on parle de programme dynamique
- Sinon on parle de programme statique.
- Programme `ldd`: permet de lister les bibliothèques dynamiques d'un programme.
- Variable d'environnement `LD_LIBRARY_PATH`: permet de définir les répertoires contenant les bibliothèques nécessaires à l'exécution des programmes.

Compilation par morceaux

- Le code source d'un programme peut être très important

  Gaussian G03.B05 = 1,193,237 lignes de fortran (au moins)

  Amber9 = 1,032318 lignes de code (2/3 Fortran, 1/3 C)

- Généralement, le code source va être divisée en morceaux (souvent 1 fichier par routine ou par bloc de routines)
- Compilation par morceaux

```
1 # compilation en une fois
2 gcc -o addition1 addition1.c random_init.c zero_init.c  temps.c
3 # compilation en plusieurs morceaux
4 gcc -c addition1.c
5 gcc -c random_init.c
6 gcc -c zero_init.c
7 gcc -c temps.c
8 gcc -o addition1 addition1.o random_init.o zero_init.o temps.o
```

☞ Avantage: après une modification, on ne recompile que les morceaux qui ont changé

Automatisation de la compilation

- ➢ Il est possible d'inclure l'ensemble des commandes de compilation dans un script
- ☹ A chaque modification, tout le programme est recompilé, même ce qui n'a pas changé
- ☞ la commande Unix `make` permet d'automatiser la compilation en ne compilant que ce qui est nécessaire.
- ➢ Le fichier `Makefile` contient les règles de compilation (= un arbre de dépendance)

Exemple de fichier Makefile

```
1  # les commentaires commencent par #
2
3  # dependance du programme
4  addition1: addition1.o random_init.o zero_init.o temps.o
5      gcc -o addition1 addition1.o random_init.o zero_init.o temps.o
6
7  # ATTENTION: les regles de compilation commencent
8  #            par une TABULATION
9  addition1.o: addition1.c
10     gcc -c addition1.c
11
12 random_init.o: random_init.c
13     gcc -c random_init.c
14
15 zero_init.o: zero_init.c
16     gcc -c zero_init.c
17
18 temps.o: temps.c
19     gcc -c temps.c
```

Utilisation de variables prédéfinies

```
1  CC=gcc
2  OBJ=random_init.o zero_init.o temps.o
3
4  prog: addition1
5
6  # variables predefinies:
7  # $@ : la cible
8  # $< : la dependance qui a change
9  # $^ : toutes les dependances
10 #
11 addition1: addition1.o $(OBJ)
12         $(CC) -o $@ $^
13
14 # regle generale:
15 .c.o:
16         $(CC) -c -o $@ $<
```

Makefile: choses diverses (et variées)

- Modification des variables: `make CC=icc`
- Inclusion de fichiers: `include fichier`
- Ne pas afficher l'exécution des règles: `@` avant la règle
- Makefile récursif: `make -C directory` dans une règle ou `cd directory; make`
- `.PHONY`: cible "virtuelle" (pas de fichier créé)
- `echo texte`: affichage de `texte`

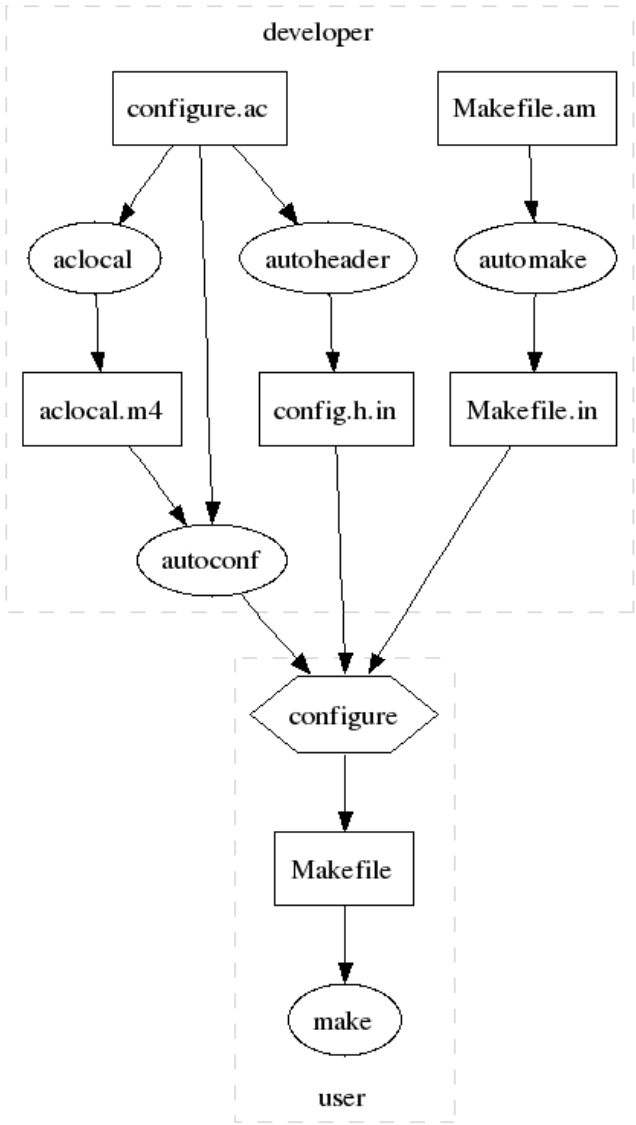
Il existe d'autres logiciels que make

SCons (python), Ant (java), cook (C), etc.

Adaptation de la compilation à l'architecture

- Chaque ordinateur est particulier de par sa configuration (processeur, RAM, disque, compilateur, bibliothèques, etc.)
- Il faut adapter le Makefile au système
  ☞ modification dynamique du Makefile
- Parmi les logiciels qui permettent de faire cela ☞ `autotools`
- `autotools` est une collection de logiciels permettant l'automatisation de la génération de Makefile à partir de règles simples.
- `autotools` comprend `autoconf`, `automake`, et `libtools`

autoconf



autoconf permet de créer le script de configuration: `configure`

automake permet à partir d'un fichier de règles simplifiées (`Makefile.am`) de créer un fichier Makefile simplifié (`Makefile.in`) utilisé par `configure` pour créer le Makefile final

configure est le script exécuté en premier par l'utilisateur. Il génère le Makefile adéquat pour l'architecture courante.

```
1 # commande a executer
2 # par l'utilisateur
3 sh ./configure
4 make
5 # eventuellement
6 make install
```

Exemple d'utilisation d'autoconf

1. Creation d'un fichier `Makefile.am`
2. Creation d'un fichier `configure.ac`
3. `aclocal`
4. `autoconf`
5. `automake -a`

`Makefile.am`

```
1 # liste des programmes
2 bin_PROGRAMS=addition1
3 # liste des sources necessaires par programme
4 addition1_SOURCES=addition1.c zero_init.c random_init.c temps.c
```

configure.ac

```
1  # initialisation
2  AC_INIT(addition1.c)
3
4  # creation du Makefile.in a partir de Makefile.am
5  # variable: nom du package, numero de version
6  AM_INIT_AUTOMAKE(addition1, 1.0)
7
8  # verifie la presence du compilateur C
9  AC_PROG_CC
10
11 # peut-on installer des programmes ?
12 AC_PROG_INSTALL
13
14 # fin: creation du Makefile
15 AC_OUTPUT(Makefile)
```

Utilisation

```
1  ./configure
2  make
```

Avantages de l'utilisation de `autotools`

Pour le programmeur:

- Génération automatique des Makefiles
- Gestion de la récursivité (dans `Makefile.am`)
- Gestion des packages (`make dist`)

Pour l'utilisateur:

- Simple à utiliser

```
1 ./configure CC=ifc --bindir=/usr/local
2 make
3 make check
4 make install
```

- et une aide existe ☺:

```
1 ./configure --help
```

A la chasse aux bugs !

- Associés aux compilateurs, il existe des outils d'aide au développement qui permet d'ausculter le comportement d'un programme
- `gdb`: GNU debugger (autre nom: `idb`, `pgdb`, etc.)
- Principe:

  compilation utilisation de l'option `-g` qui rajoute du code spécifique au debuggage

  debuggage soit post-mortem (lecture d'un fichier `core`), soit en temps réel ☞ utilisation de gdb

Exemple d'utilation de gdb

```
1 sh ./configure CC="gcc" CFLAGS="-g"
2 make clean
3 make
4 gdb addition1
5 (gdb) run
6 (gdb0 quit
```

Interfaces graphiques

- kdbg
- Netbeans
- eclipse
- kdevelop
- etc.

Temps d'exécution d'un programme

Commande UNIX `time` ☞ 3 nombres

- temps utilisateur (User Time)
- temps système (System Time)
- temps réel (Elapsed Time ou Wall Clock Time)

```
1 $ time ./addition1
2 real    0m0.002s
3 user    0m0.000s
4 sys     0m0.000s
```

Profil d'une œuvre

- Le "profiling" permet d'obtenir une idée précise du temps qui est écoulé dans chaque routine d'un programme
- Compilation: utilisation de l'option de compilation `-p` (ou `-pg`)
- Execution: normal (!)
- Post-traitement: utilisation de `prof` (ou `gprof`)

```
1 sh ./configure CFLAGS="-pg"
2 make clean
3 make
4 ./addition1
5 gprof addition1 > addition1.gprof
```

Accélération d'un programme

Il existe différentes manières (complémentaires) d'accélérer un programme:

- ➢ Utiliser de bonnes options de compilation
- ☞ de `-O0` jusque `-O3` et au-delà
  Attention `-O` = conservatif / `-O3` = non conservatif
- ➢ Utiliser un compilateur rapide (`gcc` vs. `icc` vs. `pgcc`)
- ➢ Utiliser les spécificités architecturales de la machines (SSE1/2/3/4, etc.)
- ☞ voir les options spécifiques du compilateur
- ➢ Bien écrire son code = éviter les branchements (`if`) dans les boucles; choisir le bon ordre des boucles
  (voir exemple sur multiplication de matrices)
- ➢ Utiliser des bibliothèques numériques
- ☞ BLAS, LAPACK (`http://www.netlib.org`)

Représentation matricielles

exemple:

$$A = \begin{bmatrix} 1. & 2. & 3. \\ 4. & 5. & 6. \\ 7. & 8. & 9. \\ 10. & 11. & 12. \end{bmatrix}$$

traduction en C:

```
1     double A[4][3];
2
3
4     A[0][0] = 1.0d0;
5     A[0][1] = 2.0d0;
6     A[0][2] = 3.0d0;
7     A[1][0] = 4.0d0;
8     A[1][1] = 5.0d0;
9  ...
10    A[3][0] = 10.0d0;
11    A[3][1] = 11.0d0;
12    A[3][2] = 12.0d0;
```

traduction en Fortran:

```
1        double precision A
2        dimension A(4,3)
3
4        A(1,1) = 1.0d0
5        A(1,2) = 2.0d0
6        A(1,3) = 3.0d0
7        A(2,1) = 4.0d0
8        A(2,2) = 5.0d0
9     ...
10       A(4,1) = 10.0d0
11       A(4,2) = 11.0d0
12       A(4,3) = 12.0d0
```

en mémoire (C):

1. 2. 3.	ligne 1
4. 5. 6.	ligne 2
7. 8. 9.	ligne 3
10. 11. 12.	ligne 4

en mémoire (Fortran):

1. 4. 7. 10.	colonne 1
2. 5. 8. 11.	colonne 2
3. 9. 6. 12.	colonne 3

Multiplication de deux matrices

Attention à la mémoire cache !

Version C:

```
1  double A[n][p], B[p][m], C[n][m];
2  int i, j, k;
3
4  for (i = 0; i < n; i++)
5  {
6      for (k = 0; k < p; k++)
7      {
8          for (j = 0; j < m; j++)
9          C[i][j] = C[i][j] + A[i][k] * B[k][j];
10     }
11 }
```

Multiplication de deux matrices

Version Fortran

```
1  double precision A, B, C
2  dimension A(n,p), B(p,m), C(n,m)
3  integer i, j, k
4
5  do j = 1, m
6     do k = 1, p
7        do i = 1, n
8           C(i,j) = C(i,j) + A(i,k) * B(k,j)
9        end do
10    end do
11 end do
```

Bibliothèques numériques

- `http://www.netlib.org`
- BLAS: Basic Linear Algebra Subprograms
- Ce sont des routines pour effectuer des opérations de bases sur les vecteurs et les matrices (Fortran 77).
- BLAS niveau 1 : opérations vecteurs-vecteurs
- BLAS niveau 2 : opérations matrices-vecteurs
- BLAS niveau 3 : opérations matrices-matrices
- Avantage de BLAS : efficace, portable, facilement accessible.

  + $\exists$ des versions machine-spécifique $\longrightarrow$ très efficace car utilise au mieux les possibilités des machines.
- ATLAS : BLAS qui s'optimise pour une architecture donnée.

BLAS est Limité aux opérations élémentaires.

exemple avec BLAS-3 :

$$
\begin{aligned}
C &\leftarrow \alpha AB + \beta C \\
C &\leftarrow \alpha A^{\mathrm{T}} B + \beta C \\
C &\leftarrow \alpha AB^{\mathrm{T}} + \beta C \\
C &\leftarrow \alpha A^{\mathrm{T}} B^{\mathrm{T}} + \beta C
\end{aligned}
$$

Si T est triangulaire

$$
\begin{aligned}
B &\leftarrow \alpha TB \\
B &\leftarrow \alpha T^{\mathrm{T}} B \\
B &\leftarrow \alpha BT \\
B &\leftarrow \alpha BT^{\mathrm{T}}
\end{aligned}
$$

...

Convention d'appel :

Caractère 1 : type donnée dans la matrice

S : REAL, D : DOUBLE PRECISION, C : COMPLEX, Z : COMPLEX*16 ou DOUBLE COMPLEX

Caractère 2 et 3 : type de matrice

GE : matrice rectangulaire (général), HE : l'une des matrices est hermitienne ($A = A^{\mathrm{T}^*}$), SY : l'une des matrices est symétrique, TR : l'une des matrices est triangulaire

Caractère 4 et 5 : type d'opérations

MM : produit de matrices, SM : résolution d'une systèmes d'équations linéaires, . . .

Exemple: _GEMM

LAPACK

- Bibliothèques de routines portables (Fortran 77) permettant de résoudre la plupart des problèmes numériques en algèbre linéaire.
- Ex.: systèmes d'équations linéaires, valeurs propres, décomposition en valeurs singulières, (LU, Cholesky, SVD, QR, ...)
- Haute performance, notamment sur les architectures parallèles, vectorielles, à mémoires partagées, ...
- LAPACK fait appel a BLAS $\longrightarrow$ très efficace et portable.
- Version Parallèle : PBLAS et ScaLAPACK.