Préconisation IRD/DSI Equipe IS

Outils de profilage

gprof, Valgrind

DSI / Informatique scientifique et Appui aux partenaires du Sud

Auteur : Romain Gabriel[1] **Encadrant** : Clive Canape[1]

[1] Equipe « informatique scientifique et Appui aux partenaires du Sud ».

Dernière modif. : 04/05/2010	**Référence :** -	**Version :** 1.0	**Pages:** 20

Diffusion :

Outils présentés :

Valgrind : http://valgrind.org/ (Licence GNU GPL)

GNU gprof : http://sourceware.org/binutils/docs-2.18/gprof/index.html (Licence GNU GPL)

Outils mentionnés :

Oprofile : http://oprofile.sourceforge.net/ (Licence GNU GPL)

Sources :

Wikipédia fr. : http://fr.wikipedia.org/

Wikipédia en. : http://en.wikipedia.org/

Unixgarden : http://www.unixgarden.com/

Sommaire

1. Objectifs :

La programmation informatique intervient dans des domaines variés, incluant ceux de l'IRD (Institut de Recherche pour le Développement). Elle répond à un besoin particulier, pouvant être répétitif et coûteux en temps. **Exemple :** Un calcul de grande envergure, long à mettre en place et à exécuter.

Un programme écrit de façon peu rigoureuse peut fonctionner mais, être jugé trop lent à exécuter, ou trop demandeur en ressources processeur et/ou mémoire.

La mise en évidence de ces défauts par le biais de différents outils se nomme le **profilage de code.** Les objectifs de ce document sont de :

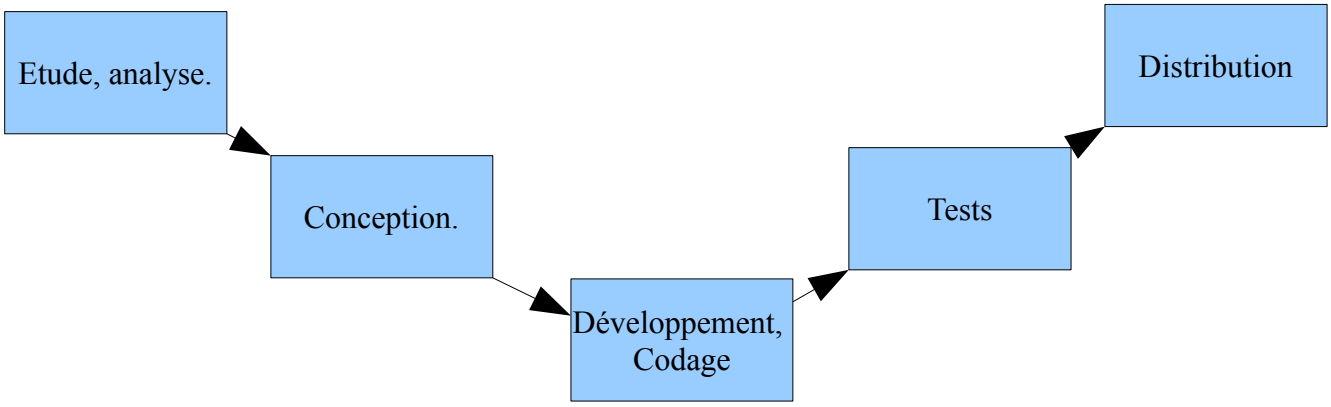
- comprendre brièvement l'origine de ces erreurs et le fonctionnement de ces outils ;
- détailler une démarche typique par le biais de plusieurs exemples simples ;
- aborder plusieurs langages utilisés au sein de l'IRD : C/C++ et Fortran.

Finalement, ces méthodes pourront être appliquées aux programmes écrits par les scientifiques de l'IRD, dont la compétence première n'est pas l'informatique et/ou la programmation.

NOTE : *Il s'agit d'optimisation pour des programmes séquentiels ou en parallèle avec processus léger (threads). Ces méthodes ne s'appliquent pas, avec ces outils, aux autres types de programmation.*

2. Phases de développement :

Avant de comprendre pourquoi et quand optimiser le code d'un programme, il faut s'intéresser au cycle de développement de ce dernier. Ainsi, de façon générale, la programmation d'un logiciel s'établit selon un « cycle en V ». Concrètement, voici une façon simplifiée de schématiser ce cycle particulier :



Nous allons nous intéresser plus particulièrement à la branche de droite. On considère alors la phase de développement (le codage) comme le début du cycle. A la suite de cette phase, comme présenté sur le schéma ci-dessus, viennent la phase de tests (quels qu'ils soient) puis la phase distribution et/ou la mise en service du logiciel concerné.
C'est durant la phase de tests que l'on va détecter les erreurs que comporte le programme, et donc les corriger, en repassant par la phase de codage. Logiquement, c'est entre ces deux étapes que se situent alors le profilage et l'optimisation de code.

3. L'optimisation en résumé :

L'optimisation est un terme assez vague, surtout en ce qui concerne le code d'un programme informatique. Ce qui est présenté dans ce document, c'est une méthode très généraliste visant à obtenir le maximum de résultats pour un minimum d'investissement. Il va de soi qu'il existe bien d'autre techniques, qui peuvent être bien plus complexes, que l'on réservera à des personnes dont l'optimisation est le métier ; établir une méthode minutieuse et complexe n'est ici pas notre but.
Finalement, par le biais de deux outils (gprof et Valgrind) nous voulons, et nous allons détecter et corriger plusieurs types d'erreurs courantes en programmation (permettant entre autre de réduire le temps d'exécution d'une portion de code).

- **Des défauts de mémoire**. Ce sont des problèmes extrêmement fréquents, mais dont la détection de leur cause est généralement relativement facile. De fait, leur correction l'est également.
  L'une des erreurs typiques est l'**erreur de segmentation**, souvent emmenée par un défaut de mémoire, et les **fuites mémoires**. Pour éviter ces défauts, il y a quelques points clés à retenir :
  - **éviter** de baser une condition sur une variable non initialisée (n'ayant pas de valeur) auparavant ;
  - **éviter** de créer des objets sans les détruire par la suite ;
  - **éviter** d'allouer de la mémoire sans la libérer par la suite.

  Exemple et explications : Prenons le cas du C/C++, et la déclaration et le remplissage d'un tableau d'entiers de deux façons différentes (*code comportant des erreurs volontaires*).

```
int main(void)
{
        int exemple[10000][10000];
        int i,j;

        for (i=0; i<10000; i++)
                for (j=0; j<10000; j++)
                        exemple[i][j]=0;

        return 0;
}
```

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

#define TAILLE 10000
int main(void)
{
        int i,j;
        int **exemple;

        exemple=malloc(TAILLE*sizeof(int*));
        for(i=0; i<TAILLE; i++)
                exemple[i]=malloc(TAILLE*sizeof(int));

        for(i=0; i<TAILLE; i++)
                for(j=0; j<TAILLE; j++)
                        exemple[i][j]=0;
        return 0;
}
```

Les deux codes vont compiler sans erreurs. En revanche, l'exécutable obtenu avec le code de gauche va entrainer une erreur de segmentation, là ou l'exécutable obtenu avec le code de droite s'exécutera sans problèmes. La différence vient de la façon d'allouer la zone mémoire nécessaire au tableau à deux dimensions.
A gauche, le tableau exemple est une variable dite locale, dont l'espace qu'elle occupe est alloué sur une « pile » évoluant au fur et à mesure du programme. Concrètement, cette pile peut être emmenée à « déborder », et c'est l'erreur que l'on met en évidence ici.
A droite, le tableau exemple est également une variable locale, mais dont l'espace qu'elle occupe est cette fois alloué dynamiquement, sur la mémoire « heap », dont la RAM fait partie. De fait, l'espace disponible est bien plus important, la

mémoire RAM se comptant en Gigaoctets. Il est alors possible d'allouer des tableaux de taille très importante sans causer d'erreur.

- **Des défauts de cache**. Ils sont induits par le processeur sur lequel est lancé le programme. Par conséquent, ces défauts apparaîtrons différemment selon le processeur utilisé. De façon simplifiée, il existe plusieurs types de mémoire dans un ordinateur, chacune ayant ses particularités.
    - La mémoire cache L1 (intégrée au processeur). **Temps de latence** : 1ns.
    - La mémoire cache L2 (intégrée au processeur). **Temps de latence** : 5ns.
    - La mémoire RAM, généralement assez conséquente. **Temps de latence** : 83ns.
    - La mémoire dite de masse : le disque dur de l'ordinateur. Elle peut être sollicitée si la mémoire RAM est saturée. On appelle cela le « swap », ou mémoire virtuelle. **Temps de latence** : 10**ms**.

NOTE **:** *Les temps de latences donnés sont variables selon le matériel utilisé, et constituent donc ici une indication vis à vis de l'ordre de grandeur.*

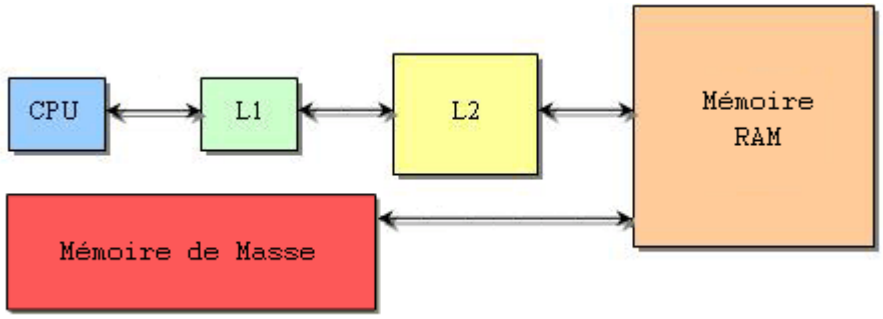


Illustration 1: Source originelle : Wikipédia (modifié)

Lorsque l'on écrit un programme, on a souvent recours à des boucles. **Exemple :** Le parcours d'un tableau de données chiffrées.

Il faut savoir que le traitement de ce genre de boucle est réalisé dans la mémoire cache. Pourquoi ? Tout simplement car les temps d'accès, de lecture et d'écriture dans ce genre de mémoire n'ont absolument rien à voir avec ceux de la mémoire RAM, et encore moins avec ceux du disque dur. En fait, le cache contient des copies de données stockées dans la mémoire centrale (RAM ou de masse). Avant tout accès à la mémoire, le processeur vérifie que les données ne sont pas déjà en cache. Si c'est le cas, un accès à la mémoire est évité, réduisant de fait le temps nécessaire.

On distinguera deux types de défaut de cache : défaut en écriture et **défaut en lecture**. Ce dernier est celui qui va entrainer un ralentissement de l'exécution le plus important, car il faudra charger les données depuis la mémoire RAM ou de masse, bien plus lente que le cache. De plus, si la mémoire cache est saturée, il faudra préalablement libérer une zone mémoire, et copier son contenu dans la mémoire RAM ou de masse. On aura donc affaire à un défaut dit « capacitif », c'est à dire que les informations que l'on traitera seront trop importantes pour être contenues entièrement dans la mémoire cache.

4. Les difficultés de l'optimisation :

Nous avons vu que **le travail d'optimisation est effectué après le développement.** On peut donc le voir comme une étape facultative dans la production d'une application. Dès lors, si l'on décide d'avoir recours à ces méthodes, il faut que cela soit justifié.

1. Investissement en temps :

Ce travail, dans la mesure où il permet, généralement, de perfectionner un programme déjà fonctionnel, demande beaucoup de temps. La question à laquelle il faut répondre est donc : Comment déterminer si le code doit être profilé et optimisé ?
Il y a plusieurs éléments de réponses.

- **premièrement**, si des **erreurs évidentes de stabilité** (fermeture soudaine du programme, interruption des calculs, consommation mémoire inexpliquée...) sont présentes, il peut être bon et « rentable » de localiser la source des erreurs grâce à ces techniques de profilage ;
- **deuxièmement**, si votre programme est fonctionnel mais que vous le **jugez trop long à exécuter**, là encore il peut être intéressant de connaître la partie du code responsable et essayer de l'améliorer ;
- **enfin**, c'est aussi par **curiosité** et conscience professionnelle (avoir un code dit « propre ») qu'il est possible de profiler et d'optimiser son programme.

2. Résultats mitigés :

Les résultats qu'il est possible d'obtenir avec ce genre de procédés sont tout à fait aléatoires, surtout vis à vis de la consommation de ressources processeur et donc du temps d'exécution. Autant il est **possible de diviser le temps d'exécution** d'une fonction, voire du programme par 2, ou 4, autant il est également **possible de n'obtenir que des résultats insignifiants**, ne modifiant absolument pas le comportement général du logiciel. Un raccourci serait de dire que la correction des défauts de mémoire améliorera la stabilité du programme, et la correction des défauts de cache la rapidité de celui-ci. En réalité, les deux sont liés, mais c'est une bonne façon de faire comprendre globalement les choses. Dans ce documents sont présentés que des outils que l'on peut qualifier, dans le domaine de l'optimisation, de « basiques ». Il existe de nombreux autres logiciels qui permettraient une optimisation bien plus importante, mais dont l'utilisation complexe requiert une formation spécifique (Intel Vtune pour n'en citer qu'un seul).

Pour résumer, avant de détailler les étapes d'une procédure type, voici quelques conseils sur ce qui est à proscrire :

- **ne pas** essayer d'optimiser un programme qui fonctionne et s'exécute (subjectivement) en peu de temps. Même si des résultats positifs seraient possibles, ce serait une perte de temps totale ;
- **ne pas** passer trop de temps sur l'optimisation d'un programme. Dépenser des journées de travail pour obtenir un gain de vitesse d'exécution de quelques secondes serait là encore une perte de temps ;
- **ne pas** profiler le code d'un programme complet. Le profilage et l'optimisation doivent être effectués sur des portions de code (fonctions) sans quoi, les résultats obtenus seront illisibles et inexploitables pour les non-avertis.

5. Démarche et outils :

Commençons par présenter un premier code avec lequel nous allons travailler, écrit en C et en Fortran. Ils comportent volontairement les erreurs les plus fréquentes, que l'on va corriger au fur et à mesure de l'avancement.

Code C

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

#define TAILLE 6000

void traitement1 (int **parametre)
{
	int i, j;
	for (i=0; i<TAILLE; i++)
		for (j=0; j<=TAILLE; j++)
			parametre[i][j]=i;
}

void traitement2 (int **parametre)
{
	int i, j;
	for (j=0; j<TAILLE; j++)
		for (i=0; i<TAILLE; i++)
			parametre[i][j]=i;
}

int main(void)
{
	int **tab1, **tab2;
	int nb_exec, condition;
	int i;

	tab1=malloc(sizeof(int*)*TAILLE);
	tab2=malloc(sizeof(int*)*TAILLE);
	for (i=0; i<TAILLE; i++)
	{
		tab1[i]=malloc(sizeof(int)*TAILLE);
		tab2[i]=malloc(sizeof(int)*TAILLE);
	}

	for (nb_exec=0; nb_exec<2; nb_exec++)
	{
		traitement1(tab1);
		traitement2(tab2);

		printf("Tableau 1, case 5999-6000 : %d\n",
tab1[TAILLE-1][TAILLE]);
		printf("Tableau 2, case 5999-5999 : %d\n\n",
tab2[TAILLE-1][TAILLE-1]);
	}

	if (condition>3)
		printf("Traitement de la condition\n");
	else
		printf("Condition non traitée\n");

	return 0;
}
```

Code Fortran 90

```
program exemple

double precision, allocatable :: tab1(:,:),tab2(:,:)
integer, parameter :: n=6000
integer :: condition, nb_exec
allocate(tab1(1:n,1:n), tab2(1:n,1:n))

do nb_exec=1, 2
	call traitement (tab1,n)
	call traitement2 (tab2,n)
	write (*,*) "Tableau 1, case 6001-6000 :",tab1(n+1,n)
	write (*,*) "Tableau 2, case 6000-6000 :",tab2(n,n)
enddo

if (condition>3) then
	write (*,*) "Traitement de la condition"
else
	write (*,*) "Condition non traitée"
endif

end program

subroutine traitement (parametre,n)
	double precision, dimension(n,n) :: parametre
	integer :: n

	do i=1,n+1
		do j=1,n
			parametre(i,j)=i
		enddo
	enddo
end subroutine traitement

subroutine traitement2 (parametre,n)
	double precision, dimension(n,n) :: parametre
	integer :: n

	do j=1,n
		do i=1,n
			parametre(i,j)=i
		enddo
	enddo
end subroutine traitement2
```

1. Compilation

Pour pouvoir procéder au profilage de code avec les outils présentés de façon efficace, la compilation du ou des programmes doit être effectuée avec des options particulières. Nous ne présentons ici que les compilateurs GNU. De façon basique, nous compilerons un programme de la sorte :

gcc -pg -g code.c -o executable (pour le langage C)

gfortran -pg -g code.f90 -o executable (pour le langage Fortran 90)

L'option -g permettra de produire les informations symboliques de débogage. On aura ainsi, lors de la détection d'erreur, des informations importantes, telles que les numéros des lignes problématiques.
L'option -pg permettra de générer du code supplémentaire pour écrire des informations de profilage pour des programmes de type « prof », soit dans notre cas, gprof.

Il faut savoir qu'il existe de nombreuses options supplémentaires utilisables lors de la compilation, mais qui doivent être utilisées avec précaution. Ainsi, on peut spécifier, de base, un niveau d'optimisation, ainsi que le processeur utilisé (le code généré sera alors spécifique à ce type de processeur). Par conséquent, le programme ne **s'exécutera que sur ce processeur particulier,** de façon peut-être plus rapide. Ce sont donc des options que l'on conseille uniquement si votre logiciel n'est pas destiné à être portable, et s'exécutera toujours sur le même ordinateur. Ainsi :

L'option -On avec n le niveau d'optimisation. Plus il est élevé, plus la compilation prendra du temps et de la mémoire. De manière générale, il n'est pas conseillé d'utiliser le niveau d'optimisation O3, qui va entrainer un binaire de taille plus importante, et certaines erreurs de compilation et/ou d'exécution. On conseille donc, si vous désirez utiliser cette option, d'utiliser le niveau 2, qui représente le meilleur compromis.

L'option -mcpu=cpu indique le type de processeur pour lequel le code doit être optimisé. Par conséquent, des instructions propres à cette famille de processeur seront utilisées, empêchant ainsi toute portabilité sur un ordinateur possédant un processeur de famille différente (exemple : AMD / Intel)

L'option -march=cpu indique le type de processeur pour lequel le code doit être généré. Cette fois, le code sera spécifique au processeur utilisé (exemple: core2 pour les Core 2 Duo) et ne fonctionnera peut-être même pas sur un processeur de la même famille. L'utilisation de cette option entraine inévitablement l'utilisation de l'option -mcpu.

Le gain que l'on peut obtenir avec ces deux options est extrêmement variable, et dépendra du programme. Par conséquent, à moins de destiner l'utilisation d'un programme à un seul ordinateur spécifique, elles sont à éviter.
En compilant le code fourni (avec la version 4.4 des compilateurs GNU), vous remarquerez qu'aucune erreur n'est indiquée alors que le programme en comporte volontairement. Plus important, même l'exécution du programme n'entraîne aucun avertissement ni erreur. C'est sur ce point que nous insistons : **un programme que l'on veut optimiser est un programme fonctionnel**.
Note : *L'utilisation de l'option* ***-Wall*** *permet d'afficher des avertissements plus précis, et de détecter quelques problèmes.*

2. Analyse et optimisation de premier niveau

L'optimisation doit se faire de façon progressive. Les résultats que l'on peut obtenir ne seront pas proportionnels au temps que l'on y consacrera. Pour commencer, il faut donc analyser brièvement le programme que l'on souhaite optimiser, pour, premièrement, déterminer si quelque chose ne convient pas, et deuxièmement, comprendre comment procéder pour l'améliorer.
Sur les systèmes Unix-like, une commande permettant de chronométrer le temps d'exécution d'un programme existe. C'est la commande **time**. Son utilisation est très simple (time [commande]), et les informations qu'elle donne sont claires. Utilisons la alors avec notre programme en C et en Fortran :

time ./exempleC	time ./exempleF
real 0m4.205s user 0m3.668s sys 0m0.540s	real 0m7.004s user 0m4.816s sys 0m2.180s

Les résultats obtenus sont les suivants :

- **real** : correspond au temps qui s'est écoulé entre le début du programme (exécution de la commande) à la fin (récupération de la main). C'est donc un temps « chronomètre ».
- **user** : correspond au temps passé à exécuter les fonctions (calculs).
- **sys** : représente le temps CPU passé sur des appels systèmes propres au noyau (kernel) du système d'exploitation.

Le temps CPU comme on l'entend, soit le temps passé à exécuter des calculs, c'est la somme des temps **user** et **sys.** Dans le cas d'une application mono-thread, ce temps sera inférieur ou sensiblement égal au temps réel.

Note : *Les temps donnés par la commande time peuvent varier d'une exécution à l'autre. La précision des mesures est également discutable. Ces données ne doivent être utilisées qu'à titre informatif.*

Sachant que notre programme d'exemple s'exécute en 4,205 secondes en C et en 7,004 secondes en Fortran, nous voulons diminuer ce temps, car on le juge trop important pour le peu de calculs qu'il exécute. C'est donc, comme précisé auparavant, un jugement subjectif. Avant même d'essayer de corriger certaines erreurs du code, nous pouvons procéder à une première phase d'optimisation qui – si elle n'est pas déjà faite – permet généralement d'augmenter les performances de façon concrète : il s'agit de la **parallélisation à mémoire partagée**, autrement dit l'utilisation de **threads**. Ceux que l'on préconise sont les **threads openMP**, portables, adaptables à beaucoup de langages, et relativement faciles d'utilisation.
Nous l'adaptons alors à nos programmes d'exemple. Notre **nouveau code est visible à la page suivante**.
Etant donné que l'on fait appel à une librairie externe, il faut le préciser lors de la compilation. Ainsi, les commandes utilisées seront dorénavant celles-ci :

gcc -pg -g code.c -o executable -fopenmp

gfortran -pg -g code.f90 -o executable -fopenmp

Code C

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

#define TAILLE 6000

void traitement1 (int **parametre)
{
	int i, j;
	for (i=0; i<TAILLE; i++)
		for (j=0; j<=TAILLE; j++)
			parametre[i][j]=i;
}

void traitement2 (int **parametre)
{
	int i, j;
	for (j=0; j<TAILLE; j++)
		for (i=0; i<TAILLE; i++)
			parametre[i][j]=i;
}

int main(void)
{
	int **tab1, **tab2;
	int nb_exec, condition;
	int i,nb_thread;

	tab1=malloc(sizeof(int*)*TAILLE);
	tab2=malloc(sizeof(int*)*TAILLE);
	for (i=0; i<TAILLE; i++)
	{
		tab1[i]=malloc(sizeof(int)*TAILLE);
		tab2[i]=malloc(sizeof(int)*TAILLE);
	}


	for (nb_exec=0; nb_exec<2; nb_exec++)
	{
		#pragma omp parallel num_threads(2)
		{
			nb_thread=omp_get_thread_num();
			if (nb_thread==0)
				traitement1(tab1);
			else
				traitement2(tab2);
		}
		printf("Tableau 1, case 5999-6000 : %d\n",
tab1[TAILLE-1][TAILLE]);
		printf("Tableau 2, case 5999-5999 : %d\n",
tab2[TAILLE-1][TAILLE]);
	}

	if (condition>3)
		printf("Traitement de la condition\n");
	else
		printf("Condition non traitée\n");

	return 0;
}
```

Code Fortran 90

```fortran
program exemple
!$ use OMP_LIB
double precision, allocatable :: tab1(:,:),tab2(:,:)
integer, parameter :: n=6000
integer :: condition, nb_exec, nb_thread
allocate(tab1(1:n,1:n), tab2(1:n,1:n))

do nb_exec=1, 2
	!$OMP PARALLEL NUM_THREADS(2)
	nb_thread=OMP_GET_THREAD_NUM()
	if (nb_thread==0) then
		call traitement1(tab1,n)
		write (*,*) "Tableau 1, case 6001-6000 :",tab1(n+1,n)
	else
		call traitement2 (tab2,n)
		write (*,*) "Tableau 2, case 6000-6000 :",tab2(n,n)
	endif
	$OMP END PARALLEL
enddo

if (condition>3) then
	write (*,*) "Traitement de la condition"
else
	write (*,*) "Condition non traitée"
endif

end program

subroutine traitement1 (parametre,n)
	double precision, dimension(n,n) :: parametre
	integer :: n

	do i=1,n+1
		do j=1,n
			parametre(i,j)=i
		enddo
	enddo
end subroutine traitement1

subroutine traitement2 (parametre,n)
	double precision, dimension(n,n) :: parametre
	integer :: n

	do j=1,n
		do i=1,n
			parametre(i,j)=i
		enddo
	enddo
end subroutine traitement2
```

Time ./exempleTC	Time ./exempleTF
real 0m2.111s user 0m3.168s sys 0m0.508s	real 0m3.371s user 0m3.892s sys 0m1.932s

On peut alors immédiatement constater une nette amélioration concernant le temps d'exécution de notre programme, qui est environ divisé par deux. C'est un résultat logique dans notre cas (utilisation de deux threads distincts), mais qui n'est pas toujours vrai (le gain peut être supérieur ou bien inférieur).
On peut en conclure que l'utilisation de la parallélisation constitue une étape qui, selon le cas, peut-être rapide à mettre en place tout en apportant des résultats vraiment significatifs. Cependant, nous jugeons ce temps d'exécution toujours trop long, et voulons continuer notre optimisation un peu plus en profondeur. Pour cela, voici la démarche que nous proposons, à l'aide des outils gprof et Valgrind.

3. Détecter les portions de code qui sont problématiques

Note : *Nous travaillons maintenant sur le code comportant des threads openMP pour suivre un ordre chronologique.*

Si vous avez suivi nos indications de compilation (options -pg -g) et que vous avez déjà exécuté votre programme, un fichier **gmon.out** doit avoir été généré dans le même dossier que votre programme. Si ce n'est pas le cas, placez vous dans le dossier contenant votre programme et exécutez le (./programme). Ce fichier est en fait le rapport d'exécution qu'est capable de lire l'outil gprof.
Deux cas de figure s'offrent à vous :

- vous êtes **familier avec les outils en ligne de texte** :

  En admettant que votre programme se nomme « exemple », utilisez la commande suivvante :

  gprof -b exemple gmon.out (-b pour brief, c'est à dire que les textes d'explication disparaissent)

  La partie des résultats qui nous intéresse, à ce niveau, est la suivante (appelée « Flat Profile ») :

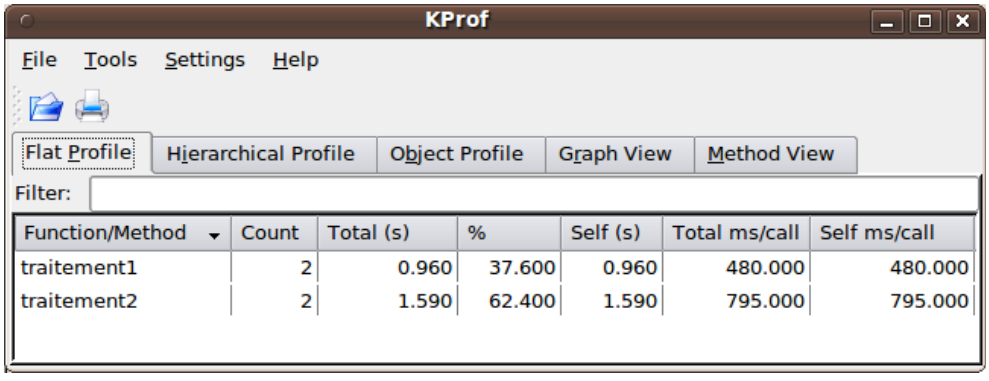
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 62.35      1.59     1.59        2   795.00   795.00  traitement2
 37.65      2.55     0.96        2   480.00   480.00  traitement1
```

- vous préférez l'**utilisation d'interface graphique** :

  En admettant que votre programme se nomme « exemple », utilisez la commande suivante :

  kprof -f exemple

  Une interface similaire à celle-ci devrait alors se lancer :



Pour chaque fonction, on a alors accès à plusieurs informations qui sont :

- calls (Count) : le nombre d'appels total de la fonction ;
- self seconds (Total(s)) : le temps total de l'exécution (ou des exécutions) de la fonction ;
- self ms/call : le temps d'une seule exécution de la fonction ;
- % time (%) : le pourcentage du temps total de l'exécution ou des exécutions de la fonction.

Cette analyse peut être effectuée également sur le programme en Fortran. Les valeurs en pourcentages sont très parlantes, et on remarque immédiatement que :

- programme C : la fonction **traitement2** s'exécute beaucoup plus lentement que la fonction **traitement1** ;
- programme Fortran : la fonction **traitement1** s'exécute beaucoup plus lentement que la fonction **traitement2.**

C'est un résultat normal étant donné qu'en Fortran, les indices de lignes et colonnes sont inversés par rapport au C. Finalement, les résultats sont les mêmes entre les deux langages. Le problème qui se pose est que ces deux fonctions sont sensées effectuer le même traitement : initialiser un tableau (matrice) à deux dimensions.

Explication : Lors du parcours d'un tableau, le cache du processeur est sollicité. Les informations sont stockées en lignes dans le cache. La lecture se fait donc également dans ce sens.

1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18
19	20	...						

Si l'on accède à des données qui sont contigues, le traitement sera beaucoup plus rapide que si l'on accède à des données « éloignées ». En parcourant un tableau dans le sens des colonnes, une nouvelle ligne doit être mise en cache à chaque tour de boucle, ce qui est coûteux en temps, et pas du tout optimisé. Pour palier à ce défaut, il faudra retenir de **toujours parcourir un tableau dans le sens des lignes, sauf si le langage utilisé est le Fortran.**

Nous avons donc détecté un premier défaut d'optimisation dans notre programme, que ce soit en C ou en Fortran. Pour vous convaincre de la différence importante de performance, vous pouvez modifier la fonction **traitement2** (dans le programme en C) et **traitement1** (dans le programme en Fortran) et procéder à une nouvelle analyse avec la commande time et l'outil gprof. Dans un soucis de lisibilité, les codes propres et corrigés ne seront donnés qu'à la fin de cette partie (page 15).

4. Corriger les erreurs mémoires

Avant de commencer, il faut tout de même préciser quelques points importants :

- Valgrind **dégrade énormément** les performances du programme pendant la phase d'optimisation ;
- **ne surtout pas** lancer une analyse Valgrind sur un programme complet dont on sait qu'il est long à exécuter. Préférez l'analyse de fonctions particulières ;
- **ne pas oublier** de compiler le programme ou la portion de programme avec (au moins) l'option -g du compilateur. Les lignes comportant des erreurs seront alors indiquées, facilitant grandement la correction de celles-ci.

L'utilisation de l'outil Valgrind est très simple, et similaire à celle de gprof. Il suffit, si l'on ne veut pas préciser d'options supplémentaires, d'exécuter la commande :

valgrind commande où *commande* est la commande nécessaire au lancement de notre programme.

Aucune interface graphique n'est présente pour l'utilisation de l'outil de fuites de mémoire Valgrind. Cependant, les résultats sont relativement clairs et faciles à comprendre. Si l'on lance une analyse sur notre programme écrit en C (multithreadé), voici ce que l'on obtient :

```
==6164== Invalid write of size 4
==6164==    at 0x8048708: traitement1 (redac_openmp.c:12)
==6164==    by 0x8048924: main.omp_fn.0 (redac_openmp.c:44)
==6164==    by 0x8048842: main (redac_openmp.c:40)
==6164==  Address 0x41c50a8 is 0 bytes after a block of size 24,000 alloc'd
==6164==    at 0x4024C1C: malloc (vg_replace_malloc.c:195)
==6164==    by 0x80487C9: main (redac_openmp.c:33)
==6164==
==6164== Invalid read of size 4
==6164==    at 0x8048870: main (redac_openmp.c:48)
==6164==  Address 0x1602b368 is 0 bytes after a block of size 24,000 alloc'd
==6164==    at 0x4024C1C: malloc (vg_replace_malloc.c:195)
==6164==    by 0x80487C9: main (redac_openmp.c:33)
==6164==
Tableau 1, case 5999-6000 : 5999
Tableau 2, case 5999-5999 : 5999
Tableau 1, case 5999-6000 : 5999
Tableau 2, case 5999-5999 : 5999
==6164== Conditional jump or move depends on uninitialised value(s)
==6164==    at 0x80488BB: main (redac_openmp.c:52)
==6164==
Traitement de la condition
==6164==
==6164== HEAP SUMMARY:
==6164==     in use at exit: 288,051,468 bytes in 12,007 blocks
==6164==   total heap usage: 12,008 allocs, 1 frees, 288,052,876 bytes allocated
==6164==
==6164== LEAK SUMMARY:
==6164==    definitely lost: 48,000 bytes in 2 blocks
==6164==    indirectly lost: 286,152,000 bytes in 11,923 blocks
==6164==      possibly lost: 1,848,144 bytes in 78 blocks
==6164==    still reachable: 3,324 bytes in 4 blocks
==6164==         suppressed: 0 bytes in 0 blocks
==6164== Rerun with --leak-check=full to see details of leaked memory
==6164==
==6164== For counts of detected and suppressed errors, rerun with: -v
==6164== Use --track-origins=yes to see where uninitialised values come from
==6164== ERROR SUMMARY: 12003 errors from 3 contexts (suppressed: 19 from 8)
```

Les erreurs sont signalées en début de rapport tandis que les fuites de mémoire le sont en fin. Pour simplifier la lecture, les erreurs sont ici soulignées d'un trait rouge, et le rapport de fuites mémoire encadré en rouge. Rapidement, on peut détecter 3 erreurs dans notre code ainsi que d'importantes fuites de mémoire :

- Invalid write of size 4
- Invalide read of size 4
- Conditional jump or move depends on uninitialised value(s)

+

- de nombreux octets indiqués comme « definitely lost », « indirectly lost », « possibly lost ».

Ces erreurs sont les plus fréquentes, et nous allons voir comment les corriger, en les étudiant une par une.

A savoir : *Il est possible de ne pas afficher le texte descriptif lors d'une analyse en utilisant l'option* ***-q*** .

Note : *Le code que nous avons analysé est le C, mais les erreurs du code en Fortran sont exactement les mêmes ; seuls les numéros des lignes change.*

- ***Invalid write of size 4.*** Cette erreur est indiquée ligne 12, dans la fonction « traitement1 ». Elle apparaît lorsque l'on tente d'écrire des données dans une zone incorrecte de la mémoire. Il suffit alors d'observer la ligne 12 et celles qui précèdent pour observer qu'il y a une erreur dans la boucle, qui prend une colonne de trop. La correction est simple :

Code C	*Code Fortran 90*

for (i=0; i<TAILLE; i++)
for (j=0; j<=TAILLE; j++)
parametre[i][j]=i;
	do i=1,n+1
	do j=1,n
	parametre(i,j)=i

for (i=0; i<TAILLE; i++)
for (j=0; j<TAILLE; j++)
parametre[i][j]=i;
	do i=1,n
	do j=1,n
	parametre(i,j)=i

- ***Invalid read of size 4 :*** Le problème se trouve ligne 48, dans la fonction « main ». Cette erreur est courante lorsque l'on essaie de lire dans un bloc mémoire invalide. Cela doit être le cas dans le code utilisé :

Code C	*Code Fortran 90*
printf("Tableau 1, case 5999-6000 : %d\n", tab1[TAILLE-1][TAILLE]);	write (*,*) "Tableau 1, case 6001-6000 :",tab1(n+1,n)
printf("Tableau 1, case 5999-5999 : %d\n", tab1[TAILLE-1][TAILLE-1]);	write (*,*) "Tableau 1, case 6000-6000 :",tab1(n,n)

Ayant un tableau de 6000x6000 cases, il est impossible de lire dans la case 5999-6000 en C (le premier indice étant 0) et dans la case 6001-6000 en Fortran 90.

- ***Conditional jump or move depends on uninitialised value(s) :*** L'erreur est ici présente ligne 52. Son nom est explicite : une condition (**exemple :** if) dépend d'une variable non initialisée. En regardant le code, on peut constater notre erreur. La variable condition n'a pas été initialisée, et pourtant une condition est basée sur sa valeur.

Code C	*Code Fortran 90*

if (condition>3)
printf("Traitement de la condition\n");
else
printf("Condition non traitée\n");
	if (condition>3) then
	write (*,*) "Traitement de la condition"
	else
	write (*,*) "Condition non traitée"
	endif

condition=tab1[10][3];
if (condition>3)
printf("Traitement de la condition\n");
else
printf("Condition non traitée\n");
	condition=tab1(10,3)
	if (condition>3) then
	write (*,*) "Traitement de la condition"
	else
	write (*,*) "Condition non traitée"
	endif

Affecter une valeur à la variable « condition » **avant le début de la condition** règle le problème.

- ***Fuites de mémoire :*** Beaucoup de fuites sont détectées dans notre programme. Si le programme est complexe, il peut être bon de relancer l'analyse Valgrind par la commande : « valgrind --leak-check=full commande ». Nous aurons alors plus d'informations sur la provenance des erreurs. Pour notre exemple, le problème est identifiable rapidement : nous allouons de la mémoire sans jamais la libérer. Il suffit de quelques lignes pour y remédier.

Code C	*Code Fortran 90*

for (i=0; i<TAILLE; i++)
{
free(tab1[i]);
free(tab2[i]);
}
free(tab1);
free(tab2);
 	
return 0;
	if(allocated(tab1)) deallocate(tab1)
	if(allocated(tab2)) deallocate(tab2)
	
	end program

Finalement, si vous avez suivi toutes les étapes et corrigé le code au fur et à mesure, voici celui que vous devriez dorénavant avoir :

Code C

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

#define TAILLE 6000
void traitement1 (int **parametre)
{
	int i, j;
	for (i=0; i<TAILLE; i++)
		for (j=0; j<TAILLE; j++)
			parametre[i][j]=i;
}

void traitement2 (int **parametre)
{
	int i, j;
	for (i=0; i<TAILLE; i++)
		for (j=0; j<TAILLE; j++)
			parametre[i][j]=i;
}

int main(void)
{
	int **tab1, **tab2;
	int nb_exec, condition;
	int i,nb_thread;

	tab1=malloc(sizeof(int*)*TAILLE);
	tab2=malloc(sizeof(int*)*TAILLE);
	for (i=0; i<TAILLE; i++)
	{
		tab1[i]=malloc(sizeof(int)*TAILLE);
		tab2[i]=malloc(sizeof(int)*TAILLE);
	}

	for (nb_exec=0; nb_exec<2; nb_exec++)
	{
		#pragma omp parallel num_threads(2)
		{
			nb_thread=omp_get_thread_num();
			if (nb_thread==0)
				traitement1(tab1);
			else
				traitement2(tab2);
		}
		printf("Tableau 1, case 5999-5999 : %d\n",
tab1[TAILLE-1][TAILLE-1]);
		printf("Tableau 2, case 5999-5999 : %d\n",
tab2[TAILLE-1][TAILLE-1]);
	}

	condition=tab1[10][3];
	if (condition>3)
		printf("Traitement de la condition\n");
	else
		printf("Condition non traitée\n");

	for (i=0; i<TAILLE; i++)
	{
		free(tab1[i]);
		free(tab2[i]);
	}
	free(tab1);
	free(tab2);

	return 0;
}
```

Code Fortran 90

```
program exemple
!$ use OMP_LIB
double precision, allocatable :: tab1(:,:),tab2(:,:)
integer, parameter :: n=600 0
integer :: condition, nb_exec, nb_thread
allocate(tab1(1:n,1:n), tab2(1:n,1:n))

do nb_exec=1, 2
	!$OMP PARALLEL NUM_THREADS(2)
	nb_thread=OMP_GET_THREAD_NUM()
	if (nb_thread==0) then
		call traitement (tab1,n)
		write (*,*) "Tableau 1, case 6000-6000 :",tab1(n,n)
	else
		call traitement2 (tab2,n)
		write (*,*) "Tableau 2, case 6000-6000 :",tab2(n,n)
	endif
	!$OMP END PARALLEL
enddo

condition=tab1(10,3)
if (condition>3) then
	write (*,*) "Traitement de la condition"
else
	write (*,*) "Condition non traitée"
endif

if(allocated(tab1)) deallocate(tab1)
if(allocated(tab2)) deallocate(tab2)
end program

subroutine traitement (parametre,n)
	double precision, dimension(n,n) :: parametre
	integer :: n

	do i=1,n
		do j=1,n
			parametre(i,j)=i
		enddo
	enddo
end subroutine traitement

subroutine traitement2 (parametre,n)
	double precision, dimension(n,n) :: parametre
	integer :: n

	do j=1,n
		do i=1,n
			parametre(i,j)=i
		enddo
	enddo
end subroutine traitement2
```

5. Constatation des résultats

Le code corrigé doit être préalablement compilé. Nos fichiers sont nommés « redac_propre.c » et « redac_propre.f90 ». La compilation est alors effectuée par les commandes :

gcc redac_propre.c -o exemplePTC -fopenmp

gfortran redac_propre.f90 -o exemplePTF -fopenmp

Utilisons tout d'abord Valgrind, pour constater que les erreurs que nous avions sont réellement corrigées.
Commande :

valgrind ./exemplePTC

```
==6579== LEAK SUMMARY:
==6579==    definitely lost: 0 bytes in 0 blocks
==6579==    indirectly lost: 0 bytes in 0 blocks
==6579==      possibly lost: 144 bytes in 1 blocks
==6579==    still reachable: 3,436 bytes in 4 blocks
==6579==         suppressed: 0 bytes in 0 blocks
==6579== Rerun with --leak-check=full to see details of leaked memory
==6579==
==6579== For counts of detected and suppressed errors, rerun with: -v
==6579== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 19 from 8)
```

Les fuites mémoires sont réglées, et ERROR SUMMARY n'affiche aucune erreur. Ce nouveau code est donc propre et fonctionnel.

Utilisons alors la commande **time**, pour « chronométrer » l'exécution de notre programme corrigé (en C).

```
real    0m1.323s
user    0m1.932s
sys     0m0.480s
```

Les résultats sont plutôt impressionnants. Nous sommes passé d'un temps d'exécution de 4,2 secondes à un temps d'exécution de 1,3 secondes.

Pour finir, utilisons l'option d'optimisation des compilateurs GNU dont nous avions mentionné l'existence. Nous utiliserons le niveau 2, et les commandes de compilation seront donc les suivantes :

gcc -pg -g -O2 redac_propre.c -o exemplePTC -fopenmp

gfortran -pg -g -O2 redac_propre.f90 -o exemplePTF -fopenmp

Toujours avec **time**, voici ce que nous obtenons pour notre programme en C :

```
real    0m0.711s
user    0m0.692s
sys     0m0.512s
```

Là encore, le temps d'exécution est diminué, ce qui était notre but. Dans ce cas, l'optimisation n'aura pas demandé beaucoup de temps, et les résultats obtenus sont excellents. Il faut noter qu'à plus grande échelle (programme plus complexe et plus lent), les résultats pourraient être encore meilleurs. Aussi, outre un temps d'exécution réduit, nous avons également diminué considérablement l'utilisation mémoire du programme, ce qui est très appréciable si celui-ci est destiné à fonctionner en parallèle d'autres logiciels sur une machine.

6. Etude plus précise :

La partie précédente présentait une démarche relativement simple, pour un code également peu complexe. Cependant, nous n'avons pas abordés des cas plus complexes que vous pourriez rencontrer et auxquels il faut être préparé.

1. Détecter les défauts de cache

Dans le code précédemment étudié, nous détections un défaut de cache facilement, car nous avions deux fonctions effectuant le même traitement de deux façons différentes. Seulement, il est plus proche de la réalité de penser que généralement, seule la fonction problématique sera présente. Dès lors, hormis un temps d'exécution relativement long (et c'est subjectif), celle-ci ne présentera aucun défaut particulier lors d'une analyse avec gprof ou Valgrind par défaut. Pour y remédier, Valgrind comporte un autre outil dédié aux défauts de cache : **cachegrind**.

Important : *Les erreurs de cache sont* ***inévitables****. Il est possible de les diminuer, mais pas de les supprimer en totalité.*

L'utilisation de l'outil passe par l'ajout de plusieurs options à la commande Valgrind basique. Ainsi, on retrouve :

- ***--tool=cachegrind*** *:* **Option indispensable** pour l'utilisation de cachegrind. Spécifie simplement que l'on souhaite utiliser cachegrind à la place de memcheck, qui est l'outil par défaut de Valgrind.

- ***--cachegrind-out-file=filename*** *:* Les fichiers générés à chaque analyse ne sont différenciés que par le PID de la commande lancée. Pour faciliter l'organisation, il est intéressant de spécifier le nom du fichier qui sera généré lors d'une analyse. Cette option le permet, où « filename » est à remplacer par le nom désiré.

- ***--trace-children=yes*** *:* Permet, si le programme que l'on analyse fait appel à des processus fils, de fournir un profiling correct.

- ***--I1=<size>,<associativity>,<line size> :*** Permet d'effectuer des simulations sur la taille du cache L1. Par exemple, --I1=65536,2,64 va spécifier l'utilisation d'un cache de taille 65536 octets.
  Cette option se décline, avec la même syntaxe, pour les caches data (--D1) et L2 (--L2).

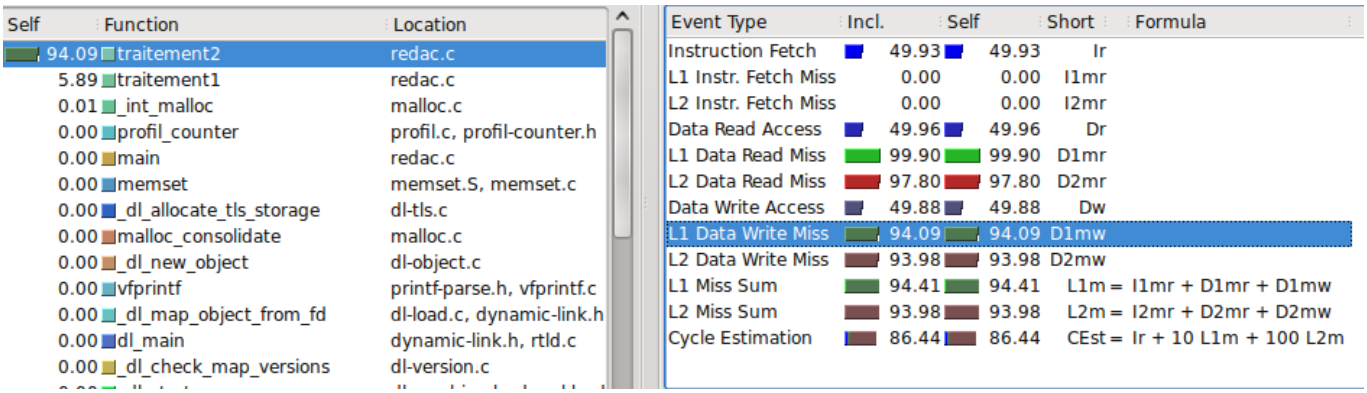
Note : *Il est possible, sur un système Unix-like, de connaître la taille de son cache L1 et L2. La commande* ***lshw*** *donnera la liste et la description des composants de l'ordinateur. Le processeur (et donc la taille des caches) est décrit en début de résultat.* ***Il faudra, pour exécuter cette commande, posséder les droits root.***

Il existe finalement peu d'options pour l'outil cachegrind. La plupart qui sont disponibles concernent l'affichage des résultats lorsque l'analyse est terminée. Dans la mesure où nous allons utiliser un outil graphique, elles sont, pour la plupart, inutiles dans notre cas. Le **manuel** de Valgrind ainsi que le **site officiel** sont très documentés, et il vous sera possible de retrouver toutes les options si vous souhaitez vous passer de l'outil graphique.

Pour analyser un programme de façon rapide, nous n'utiliserons pas d'options spécifiques. La commande utilisée sera donc la suivante :

valgrind --tool=cachegrind commande *où commande représente la commande à entrer pour lancer notre programme.*

Un fichier « cachegrind.out » sera généré. Pour éviter d'interpréter les résultats dans une console, nous conseillons l'utilisation de kcachegrind qui est un outil graphique. Son lancement s'effectue par la commande du même nom, et il suffit ensuite d'ouvrir le fichier cachegrind.out correspondant à notre analyse (ici le programme en C initial, comportant toutes les erreurs).



Self	Function	Location
94.09	traitement2	redac.c
5.89	traitement1	redac.c
0.01	_int_malloc	malloc.c
0.00	profil_counter	profil.c, profil-counter.h
0.00	main	redac.c
0.00	memset	memset.S, memset.c
0.00	_dl_allocate_tls_storage	dl-tls.c
0.00	malloc_consolidate	malloc.c
0.00	_dl_new_object	dl-object.c
0.00	vfprintf	printf-parse.h, vfprintf.c
0.00	_dl_map_object_from_fd	dl-load.c, dynamic-link.h
0.00	dl_main	dynamic-link.h, rtld.c
0.00	_dl_check_map_versions	dl-version.c

Event Type	Incl.	Self	Short	Formula
Instruction Fetch	49.93	49.93	Ir	
L1 Instr. Fetch Miss	0.00	0.00	I1mr	
L2 Instr. Fetch Miss	0.00	0.00	I2mr	
Data Read Access	49.96	49.96	Dr	
L1 Data Read Miss	99.90	99.90	D1mr	
L2 Data Read Miss	97.80	97.80	D2mr	
Data Write Access	49.88	49.88	Dw	
L1 Data Write Miss	94.09	94.09	D1mw	
L2 Data Write Miss	93.98	93.98	D2mw	
L1 Miss Sum	94.41	94.41	L1m =	I1mr + D1mr + D1mw
L2 Miss Sum	93.98	93.98	L2m =	I2mr + D2mr + D2mw
Cycle Estimation	86.44	86.44	CEst =	Ir + 10 L1m + 100 L2m

On peut alors naviguer entre les fonctions et les informations, dont les principales sont les suivantes :

- L1 Data Read Miss
- L2 Data Read Miss

  Ce sont les défauts de cache L1 et L2 en **lecture de données (Data).** Ces défauts causent généralement un ralentissement important de l'exécution du code.

- L1 Data Write Miss
- L2 Data Write Miss

  Ce sont les défauts de cache L1 et L2 en **écriture**. Ils provoquent, de façon générale, un ralentissement bien moins important, dans la mesure où l'écriture peut être mise en attente dans une queue.

Le fonctionnement est le suivant : le processeur va tout d'abord chercher à écrire/lire dans le cache L1. Si une erreur survient, c'est à dire qu'il ne trouve pas l'information dans le cache (Miss), il va se rabattre sur le cache L2, qui est plus important en taille.

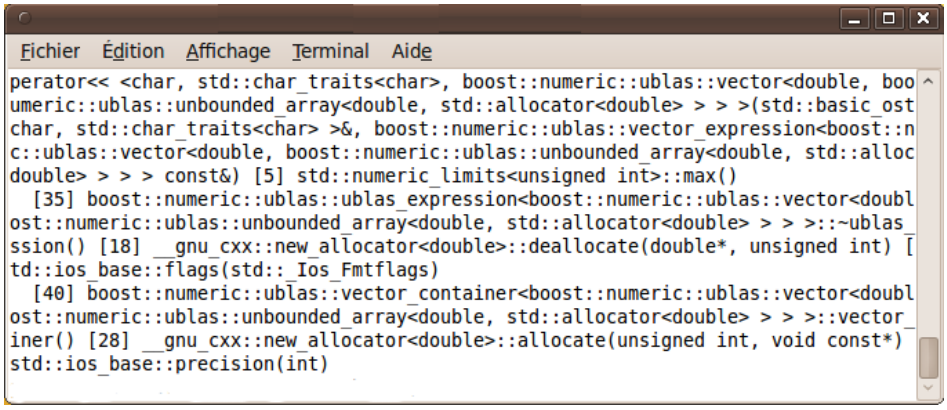
Dans notre exemple (voir capture d'écran plus haut), on aperçoit un important défaut de cache L1 **et** L2 pour la fonction **traitement2,** que ce soit en lecture ou en écriture. Cela s'explique par le fait que la fonction est particulièrement mal optimisée, et également très demandeuse en ressource (tableau de 6000x6000).

En réduisant la taille du tableau sans corriger l'erreur d'optimisation (correction cf page 16), nous pouvons remarquer que les défauts de cache L2 diminuent considérablement, ce qui correspond bien au fonctionnement théorique des accès au cache.

2. Programme complexe

L'exemple que nous avons traité était finalement qu'un programme sans but précis, n'ayant aucune application concrète. De fait, le code était relativement simple, en comparaison d'un programme complet. Dans la pratique, les programmes qu'il faudra optimiser seront beaucoup plus long, et le code beaucoup plus compliqué. Aussi, l'utilisation de types complexes (template, namespace et héritage multiple) pourra être faite, ce qui va, dans la majorité des cas, poser problème quant à la lisibilité des résultats.

Pour illustrer ces propos, prenons l'exemple d'un programme écrit en C++ dont le but est de résoudre des systèmes d'équations différentielles linéaires. La librairie « boost » ainsi que de nombreux alias sont utilisés. En effectuant une analyse de ce logiciel avec gprof, voici ce que nous avons obtenu :



C'est en fait un résultat gprof traditionnel. Cependant, les noms des fonctions utilisées sont tellement longs que l'affichage par ligne est impossible, rendant absolument inexploitables les résultats. A noter que l'utilisation de l'interface graphique kprof résout partiellement ce problème, l'affichage des fonctions n'étant pas affecté.

7. Conclusion :

Nous avons vu pourquoi et comment procéder au profilage de code et à son optimisation. Une démonstration des résultats qu'il est possible d'obtenir a été faite avec succès, ce qui a pu démontrer l'intérêt de ces méthodes. Bien que les outils que nous avons utilisés (gprof et Valgrind) soient considérés comme « basiques » dans le domaine de l'optimisation et du profiling, ils ont l'avantage de s'utiliser de façon simple et de fournir des résultats clairs.

Dans certains cas, et si l'on a du temps à y consacrer, il peut-être intéressant, après avoir suivi et appliqué ce guide, de s'intéresser à des outils plus poussés. De façon générale, ils offrent des résultats bien plus précis. On peut citer, parmi les plus connus et les plus répandus **Intel® Vtune,** son homologue **AMD CodeAnalyst** Performance Analyzer ou encore **PAPI.**

Ce dernier est gratuit, mais nous le considérons comme trop complexe dans le cadre de cette étude.

Pour approfondir l'optimisation tout en restant accessible en terme de complexité, nous préconisons l'utilisation, sur un système Unix-like, de **Oprofile**, qui est un outil gratuit. Si vous êtes intéressé par cet outil, un autre document a été rédigé concernant son utilisation et auquel vous pouvez vous référer.