

Débogage et analyse de performance

Laurence Viry et Franck Pérignon

MaiMoSiNE - Collège des Écoles Doctorales.

Grenoble

Janvier 2017

Qu'entend-on par debug et analyse de performance?

Deux étapes importantes du cycle de vie d'un code de calcul :

- ➊ anticiper et/ou corriger les problèmes du code \Rightarrow **deboguer**
- ➋ **analyser** le code et ses résultats, entre autres pour améliorer ses performances.

Performances?

Le code fonctionne et donne de bons résultats, mais ...

- il est trop lent
- ou la taille des problèmes traités est insuffisante \Rightarrow manque de mémoire
- ...

Il faut donc optimiser (ce qui est le sujet d'un cours à part entière!) mais pas n'importe comment \Rightarrow **analyse du code** et de ses performances.

Objectifs de ce cours

- donner quelques **bonnes pratiques** et premiers reflexes à avoir pour déboguer/analyser un code,
- présenter quelques **outils** incontournables/standards.

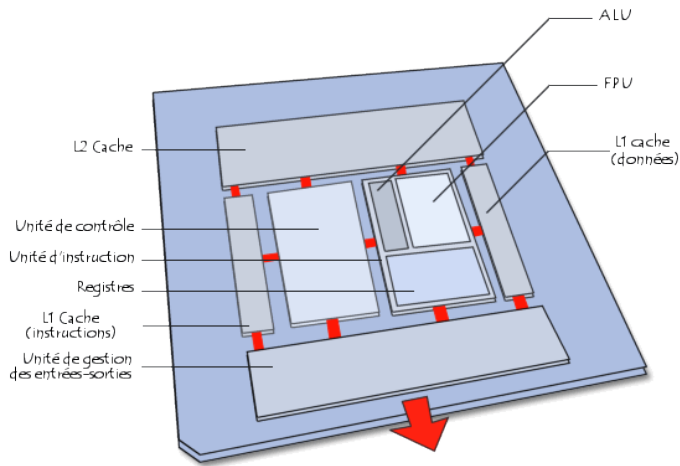
Plan de la séance:

- ① Rappels sur les architecture et la gestion de la mémoire
- ② débogage
- ③ analyse de performances

Rappel : architecture d'un processeur

- l'**Unité Arithmétique et Logique (UAL)** prend en charge les calculs arithmétiques élémentaires et l'**unité de calcul flottant (FPU)** les calculs de nombres flottants,
- l'**unité de contrôle** permet de synchroniser les différents éléments du processeur,
- les **registres** sont des mémoires de petite taille (quelques octets), suffisamment rapides pour que l'UAL puisse manipuler leur contenu à chaque cycle de l'horloge,
- l'**horloge** synchronise toutes les actions de l'unité centrale,
- l'**unité d'entrée-sortie** prend en charge la communication avec la mémoire de l'ordinateur.
- la **mémoire cache** permet d'accélérer les traitements, en diminuant les temps d'accès à la mémoire.
 - Le **cache instructions** reçoit les prochaines instructions,
 - le **cache données** manipule les données (2 ou 3 niveaux).

Rappel : architecture d'un processeur



Rappel : architecture d'un processeur

- La **fréquence d'horloge** détermine la durée d'un cycle.
- Chaque opération utilise un certain nombre de cycles.

⇒ **La fréquence d'horloge n'est pourtant pas le seul critère de performance**

- La **rapidité des accès mémoire** est un facteur important de performance.
- La **fréquence des processeurs** augmente beaucoup plus vite que la **vitesse d'accès à la mémoire**.

Rappel : comment faire des processeurs plus rapides

- Augmenter la fréquence d'horloge (limites techniques, solution coûteuse, coût énergétique)?
- Permettre l'exécution simultanée de plusieurs instructions

⇒ Plus de parallélisme dans les processeurs (pipelining, multicore, ...)

- Améliorer les accès mémoire
 - Mémoire hiérarchique
 - Optimisation des accès

Unité de calcul: CPU versus GPU

Composition d'un noeud de calcul (Avec des proportions différentes)

- Une unité de contrôle
- Des unités d'arithmétique et logique (ALU)
- Mémoire (DRAM, Caches, ...)

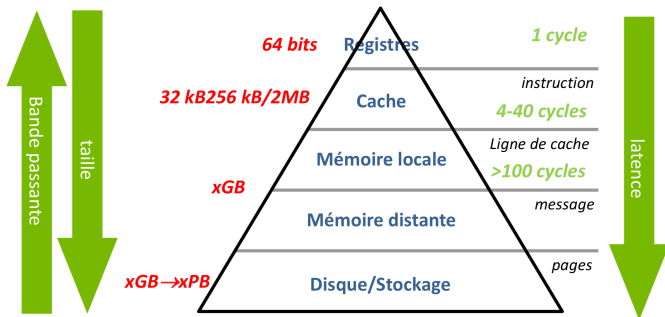


- 👉 Un grand nombre de coeurs capables de traiter rapidement des calculs simples sur des grandes quantités de données (GPU)
- 👉 Les processeurs classiques (CPU) sont moins rapides mais savent résoudre des problèmes plus compliqués
- 👉 Une combinaison de ces deux types de processeurs (accélérateurs)

Mémoire hiérarchique

- Deux types de mémoire à base de semi-conducteur
 - **DRAM (Dynamic Random Access Memory)**: chaque bit est représenté par une charge électrique qui doit être rafraîchie à chaque lecture/écriture.
 - **SRAM (Static Random Access Memory)**: retient ses données aussi longtemps qu'il y a du courant.
- Temps d'un cycle SRAM de 8 à 16 fois plus rapide qu'un cycle de DRAM
- Coût de la mémoire SRAM de 8 à 16 fois plus élevé que la mémoire DRAM
- Solution le plus couramment choisie:
 - De **1 à 3 niveaux** de SRAM en mémoire **cache**
 - La **mémoire principale** en DRAM
 - La **mémoire virtuelle** sur des disques

Hiérarchie mémoire (Séminaire L. Sauge - BULL)



Notions de latence, de bande passante, de distance, de localité spatiale/temporelle des données (**affinité**)

Implications Multi-core

Quelques définitions

- **CPU** (Central Processing Unit): puce ou processeur qui effectue les opérations de base du système.
- **socket**: le socket fournit au CPU les connections au bus système et tous les devices attachés à ce bus (mémoire, adaptateur réseau, I/O, ...).
- **Core**: unité de calcul contenu dans un CPU, ses propres registres et cache L1.

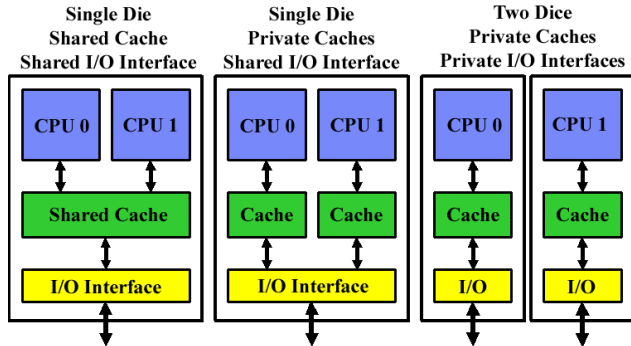
Ressources partagées

- **caches**: les cores d'un CPU peuvent se partager les caches L2 ou L3
- **socket**: Les cores d'un même CPU se partagent le même socket du CPU

Ressources partagées ➡ Potentiel de conflits d'accès aux ressources plus important

A first simple view

Source: © Realworldtech



Et plus de performances si affinités

L'apport de toujours plus de parallélisme dans les calculateurs pose le problème de la gestion des affinités.

👉 Les performances en dépendent

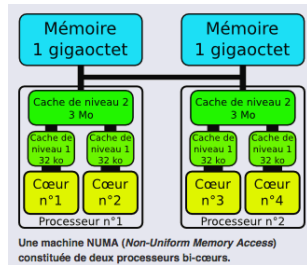
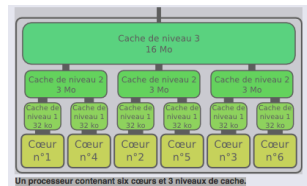
➤ Principe de localité et mémoire cache

👉 optimisation de l'accès aux données.

➤ Caches et affinité entre tâches et données:
le déplacement d'une tâche sur un autre coeur.

➤ Partage de données entre tâches: accès à la mémoire par différents coeurs.

➤ Machine NUMA et affinités pour la mémoire: accès plus rapide aux données proche du processeur qui effectue l'accès.



👉 Nos programmes peuvent s'exécuter sans tenir compte de toutes ces contraintes, mais ils iront moins vite

Transistor counts double every two years. . .

...but how?

Le nombre de transistors continue à augmenter/unité de surface.

- $Puissance \propto \text{frequency}^3$ ➡ fréquence limitée
- Plus de performance/moins de puissance ➡ Adaptation software?

Différents types de variables

- **Variable statique**: l'emplacement en mémoire est définie dès sa déclaration par le compilateur.
- **Variable automatique**: l'emplacement n'est attribuée qu'au lancement de l'unité de programme. Elle est allouée et désallouée à la fin de son exécution.
- **Variable globale**: déclarée avant l'unité de programme "main"
- **Variable locale**: variable dont la portée est restreinte à l'unité de programme où elle est déclarée
 - Automatique par défaut.
 - Rémanente : initialisée à la déclaration (attribut SAVE).
- **Tableau**
 - Automatique: tableaux locaux dont les dimensions dépendent des arguments reçus.
 - Dynamique ou à profil différé: variant d'une exécution à une autre. Il est alloué à la demande explicite du programmeur.

Diverses zones mémoire

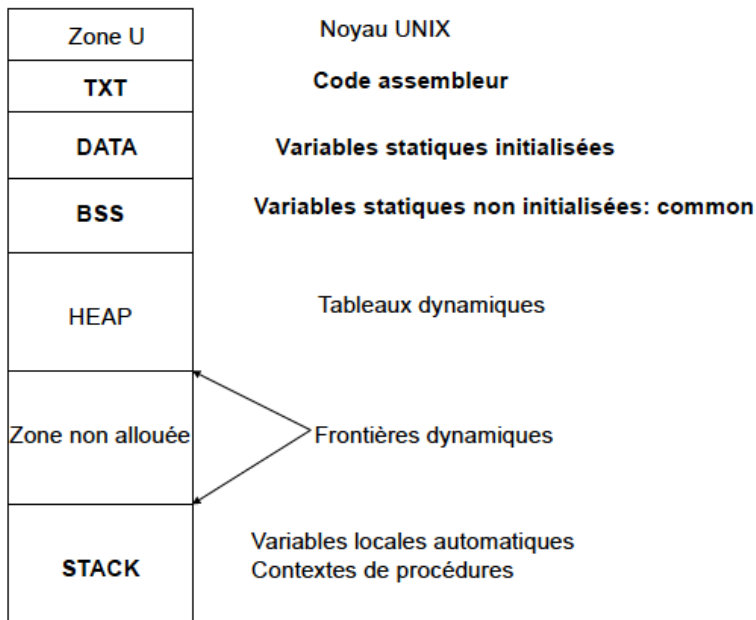
➤ Les zones statiques:

- La zone **U**: zone de communication entre l'application de l'utilisateur et le système.
- La zone **TXT**: zone qui contient l'exécutable.
- La zone **DATA**: zone où sont stockées les données statiques initialisées.
- La zone **BSS** (Block Started Symbol): zone où sont stockées les données statiques non initialisées, en particulier les COMMONs.

➤ La pile (stack) et le tas (heap): zones d'allocations mémoires dynamiques.

➤ Mémoire virtuelle : dernière couche de la hiérarchie mémoire Mémoire requise > mémoire physique : SWAP (disque)

➤ Page : plus petit espace mémoire contigue alloué à un programme.



**Débogage
Vérification**

Validation

Vérification - Validation - Débogage

Trois activités complémentaires pour augmenter la **crédibilité des résultats** de simulation numérique.

◀ **Vérification**: l'implémentation fournit les **solutions des équations du modèle**.

➤ **Vérification de code**:

- Trouver et corriger les erreurs dans les sources du code (**débogage**).
- Améliorer la robustesse des algorithmes numériques.
- Améliorer la fiabilité des logiciels en utilisant les techniques éprouvées de qualité logicielle.

➤ **Vérification de la solution**

- Garantir la pertinence des données d'entrée et de sortie pour le problème d'intérêt.
- Estimation de l'erreur numérique de la solution; erreur due à la méthode de résolution (éléments finis, différences finies,...), à la discrétisation en temps et en espace, à l'implémentation...

◀ **Validation**: l'implémentation fournit les **solutions du problème physique posé**(validité du modèle).

Une première solution : éviter les bugs!

Il est beaucoup plus simple d'**écrire du code propre** que de debuguer un programme ...

Quelques bonnes habitudes:

- code simple, fichiers courts, fonctionnalités bien définies: chaque fonction ne fait qu'une chose et le fait bien,
- lire et appliquer les "best practices" du langage (conventions de nommage ...). Voir par exemple "PEP8, Style Guide" (<https://www.python.org/dev/peps/pep-0008/>) pour Python,
- choisir des noms de variables, fonctions qui ont du sens,
- code lisible et compréhensible par d'autres personnes (pas uniquement le(s) développeur(s)!) ⇒ commentaires dans le code **ET** documentation des fonctionnalités,
- limiter les inter-dépendances, maintenir un code modulaire,
- ...

Ecrire des tests

écrire systématiquement des **tests**!

- Unitaires : vérifier que chaque composante du code fonctionne.
- Non-regression : ce qui fonctionnait avant fonctionne toujours.
- D'intégration : vérifier que les différentes parties sont compatibles, fonctionnent bien ensemble.
- Validation : le code répond au cahier des charges, il résout votre problème correctement.
- Performance, portabilité ...

Et bien sûr exploiter les outils vue à la séance précédente : gestionnaire de version (comparaison avec une version précédente et fonctionnelle du code, identification de la modif qui pose problème, retour en arrière), intégration continue etc.

- Activer, **lire et corriger les warnings** (-Wall).
- Utiliser **différents compilateurs** : ceux-ci sont plus ou moins sévères et les messages de sortie différents.
- Compiler sur différents systèmes/machines.

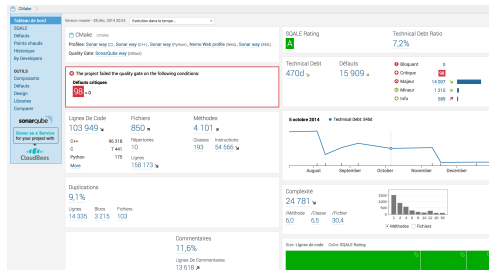
Bref, avoir un code propre, robuste, portable où chaque fonctionnalité a été utilisée/testée au moins une fois (couverture de code).

Analyse statique du code

analyse du comportement d'un programme sans vraiment l'exécuter

- souvent intégré par défaut dans les compilateurs (via les options d'optimisation notamment)
- disponible via des outils dédiés (cppcheck, polyspace, ...)
- sonarQube (<http://www.sonarqube.org>), logiciel permettant d'évaluer la qualité du code, la couverture, ...

Exemple : <http://nemo.sonarqube.org>,



Analyse statique du code (2)

- utiliser un "bon" éditeur et les outils d'analyse associés (pylint ...) : auto-complétion, auto-correction etc.
- Utiliser les outils proposés par le langage : assertions, exceptions ...

Encore une fois : lire correctement tous les messages du compilateurs et/ou de l'éditeur permettra de corriger ou d'éviter un bon nombre de bugs!

Les premiers réflexes ...

Malgré tout, le code plante, que faire? **Analyse dynamique**

Avant d'utiliser un débogueur, quelques outils pour obtenir rapidement et simplement des informations:

- **strace** : surveille les appels système lors de l'exécution du programme.

```
strace -o sortie.txt ./mon_prog
```

Sortie très verbeuse mais permettra notamment de détecter des problèmes d'I/O, de link ...

- **nm, otool, ldd** : vérification des liens avec d'autres librairies, du contenu des binaires ...

Les premiers réflexes ... (2)

- **top, htop**: vue sur l'utilisation de la machine quand le programme tourne.

```
1 [ 0.0%] Tasks: 69, 121 thr; 10 running
2 [||||| 100.0%] Load average: 8.01 8.03 8.05
3 [! 0.7%] Uptime: 70 days, 22:13:26
4 [||||| 100.0%]
5 [||||| 100.0%]
6 [||||| 100.0%]
7 [||||| 100.0%]
8 [||||| 100.0%]
9 [|| 2.6%]
10 [||||| 100.0%]
11 [ 0.0%]
12 [||||| 100.0%]
Mem[||||| 12354/129192MB]
Swp[| 14/993MB]

  PID USER      PRI  NI  VIRT   RES   SHR  S CPU% MEM%   TIME+  Command
12923 mimeau    20    0 1103M  790M 15056 R 100.0  0.6 19h26:48 python TaylorGreen3D_debug.py
12925 mimeau    20    0 1085M  773M 15028 R 99.0  0.6 19h26:51 python TaylorGreen3D_debug.py
12924 mimeau    20    0 1075M  763M 15028 R 99.0  0.6 19h26:54 python TaylorGreen3D_debug.py
12927 mimeau    20    0 1087M  775M 15028 R 99.0  0.6 19h26:49 python TaylorGreen3D_debug.py
12928 mimeau    20    0 1077M  765M 15028 R 99.0  0.6 19h26:53 python TaylorGreen3D_debug.py
12929 mimeau    20    0 1087M  775M 15028 R 99.0  0.6 19h26:49 python TaylorGreen3D_debug.py
12930 mimeau    20    0 1087M  773M 15028 R 99.0  0.6 19h26:48 python TaylorGreen3D_debug.py
12926 mimeau    20    0 1087M  775M 15028 R 99.0  0.6 19h26:50 python TaylorGreen3D_debug.py
13264 perignon  20    0 16712  2624  1280 R  1.0  0.0  0:00.91 htop
      1 root      20    0 10652   712   668 S  0.0  0.0  0:33.53 init [2]
      485 root      20    0 15512  1492   744 S  0.0  0.0  0:00.23 udevd --daemon
```

⇒ détection d'un usage excessif de mémoire, de cpu ...

Déboguer avec des instructions print?

Un méthode couramment rencontrée pour déboguer un code consiste à insérer des instructions `printf()` ou `print*` dans le code.

- demander l'impression des variables qui semblent poser des problèmes,
- compiler + exécuter le code,
- visualiser les sorties des instructions `print`

Inconvénients:

- on doit **recompiler le code** à chaque fois,
- on ne voit que l'état des **variables imprimées**,
- dans le cas d'un **programme multi-process** ou **multi-threads**, les impressions ne respectent pas forcément l'ordre d'exécution.

Utiliser un **débogueur** apporte **simplicité** et **rapidité** dans la phase de mise au point d'un code.

Débogueurs

◀ Il existe un débogueur pour presque tous

- ✧ les langages de programmation (C/C++, Fortran, Java,...) et les compilateurs,
- ✧ les langages de scripts (perl, python, R, ...),
- ✧ les progiciels (Matlab, octave,...).

◀ Deux types d'interface:

- ✧ ligne de commandes,
- ✧ interface graphique (GUI), plus facile d'accès...

◀ Les débogueurs peuvent travailler sur des codes parallèles avec plus ou moins d'efficacité.

Exécuter un programme avec un débogueur

- ◀ **Compiler** le code avec l'option "-g"
⇒ génère une **table des symboles** utilisée par le débogueur.
- ◀ **Quelques commandes de bases**
 - ✧ Afficher le code source avec le numéro des lignes: **list**,
 - ✧ Arrêter le programme à un point donné: **break**, **info break**,
 - ✧ Exécuter le programme ligne par ligne: **step**, **next**,
 - ✧ Examiner et/ou modifier le contenu d'une variable ou d'une expression: **print**, **display**, **undisplay**,
 - ✧ Arrêter le programme lorsque la valeur d'une expression change ou est atteinte: **watch**,
 - ✧ Redémarrer l'exécution : **cont**,
 - ✧ Supprime ou désactiver un point d'arrêt: **delete**, **disable**,
 - ✧ Aide et informations: **help**, **help <commande>**, **info**,
 - ✧ Faire une analyse des **différents espaces mémoire**,
 - ✧ ...

Quelques débogueurs

En ligne de commande:

- **dbx** (Solaris, IRIX, AIX)
- **gdb** (partout)
- **idb** (Intel, Linux IA64 et X86-64) basé sur **gdb** ou **dbx**
- **pdb**: python debugger (docs.python.org/library/pdb.htm)
- **Totalview**: Dynamic source code and memory debugging for C, C++ and Fortran applications (mode commande)
- ...

Des interfaces graphiques

- **ddd** (Data Display Debugger) interfacé avec **gdb**.
- **DDT** (Allinea)
- **Totalview**: interface graphique de Totalview
- **Kdbg**: interface graphique de gdb (<http://www.kdbg.org/>)
- **Eclipse: PTP** (Parallel tool Platform www.eclipse.org/ptp)
- **Débogueur constructeur...**

Débuguer un code avec gdb

① Utiliser gdb

- Compiler le code avec l'option `-g`
- Démarrer gdb et charger le code: `gdb program_name`
- Exécuter le programme à l'intérieur de gdb:
`prompt_gdb: run [arguments] [<input-file] [>output-file]`
- Examiner un fichier core: `gdb program_name core`

② Déterminer où a "planté" le code

- Exécuter le programme dans gdb
- Examiner le fichier core

③ Déterminer pourquoi le code ne fonctionne pas

- Mettre des points d'arrêt et visualiser le contenu des variables
- Exécuter le code pas à pas et analyser l'évolution du contenu des variables et l'enchaînement des instructions.

④ Analyser un problème de gestion de mémoire

- Accès mémoire.
- Fuite de mémoire.

GDB : visualiser le contenu des variables

❶ Visualiser le contenu des variables: **print** [arguments]

Les variables peuvent être de différents types:

- **Type intrinsèque**: réel, entière, complexe, caractère,...
- **Tableau**: entier ou par sous-section.
- **Variable structurée**: type dérivé (Fortran), structure (C/C++),...
- **Objets**: instantiation de classes

❷ Visualiser une variable à chaque modification: **display**.

❸ Visualiser une expression qui utilise les opérateurs du langage:

- **Expression numérique**: **print** `peri_dist + apo_dist`
- **Expression conditionnelle**:

```
do while ( iter <= 150 .and. done == .false.)  
    ...  
enddo
```

❹ Visualiser une variable , **modifier** sa valeur et **relancer** le code.

GDB : point d'arrêt

➤ Définir un point d'arrêt :

`break (b) [location][if CONDITION]`

```
break 24
```

```
break 24 if nb-lignes != 3 || nb-colonnes != 2
```

➤ Gérer les points d'arrêt

- Liste des points d'arrêt : `info(i) b`

- Supprimer un point d'arrêt :

`delete(d) <num-point-arret>`

`clear(cl) <ligne-programme>`

- Désactiver un point d'arrêt : `disable`, `enable`

① Affichage à chaque point d'arrêt d'une expression

`display(dis) <expression>`

GDB : Exécuter des commandes à un point d'arrêt

➤ Définir la commande

```
commands <num-point-arret>
command-1
...
command-2
end
```

➤ Example

```
> break 1
> commands 1
> echo valeur de i \n
> print i
> echo valeur du coefficient M[i][j] \n
> print M[i][j]
> continue
> end
```

Séquentiel: Traitement de quelques erreurs classiques

- 1 Visualiser les valeurs d'une variable en cours d'exécution (variablePrinting.c)
- 2 Erreur d'indexe dans un tableau (arrayIndex.c)
- 3 Erreur d'Entrées/Sorties (Fortran, sampleFile.f90)
- 4 Erreur dans le passage des paramètres d'une fonction (dummyArg1.f90,...)
- 5 Erreurs de pointeurs (pointerBugs.c)
- 6 Erreurs d'allocation dynamique (dynamicMemory.f90)
- 7 Division par zéro (trapezoid.f90)

Ces exemples seront fournis aux participants

Allinea DDT

DDT est un outil avec une interface graphique fournit par Allinea qui permet d'analyser et de déboguer des codes en C/C++ et Fortran90 (Il est installé sur froggy).
et avant par:

- Utiliser pour déboguer et analyser des programmes séquentiels ou parallèles.
- Spécialement adaptés aux programmes multi-process (MPI) et/ou multi-Threads ,
- permet de déboguer des programmes CUDA et OpenACC,
- Une partie Memory Debugging : mauvaise utilisation du tas (heap memory), détecte les fuites mémoires (memory leaks),...
- Permet de déboguer sur des machines distantes.
- Récupère et visualise des gros volumes de données.

De la documentation: [Allinea Forge Userguide](#) , [Parallel Debugging with DDT](#)

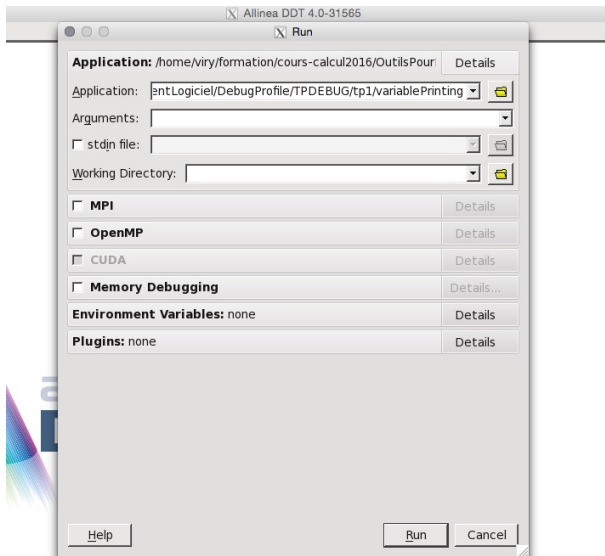
Principales fonctions de DDT (séquentiel)

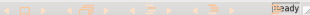
- Visualisation du **source** et du **code assembleur**
- Mise en place de **point d'arrêt**, de **barrière de synchronisation**,
- Contrôle de l'exécution:
 - **Démarrer et arrêter l'exécution** d'un process ou d'un thread, d'un groupe de processus ou de threads,
 - **Exécuter la prochaine instruction** (next, step), jusqu'au **prochain point d'arrêt** ou la **validité d'une expression**,
 - **Suspendre l'exécution** d'un processus ou d'un thread,
 - ...
- **Visualiser** un objet (variables, tableau, pointeur,...), une expression,
- **Visualiser** et **modifier** une variable ou une expression,
- Analyser les **différentes parties de la mémoire** du programme,
- ...

Démarrer un programme sous DDT

- **Compiler** le programme avec l'option **-g** (sans option d'optimisation).
- **Démarrer DDT** de différentes façons:
 - **ddt**: démarrer le débogueur, charger l'exécutable ou l'attacher à un programme en cours d'exécution.
 - **ddt filename**: démarrer le débogueur et charger l'exécutable indiqué par "filename".
 - **ddt filename corefile**: démarrer le débogueur et charger l'exécutable indiqué par "filename" et son fichier core.
 - **ddt filename -a args**: démarrer le débogueur et passer les arguments "args" au programme indiqué par "filename".
 - **ddt filename -remote hostname[:portnumber]**: démarrer le débogueur en local et le serveur DDT sur une machine distante.
- **"ddt"** a des **options** pour contrôler le comportement de l'interface graphique ([dit Reference Guide](#)).

Démarrer un programme sous DDT





Principales fenêtres de DDT

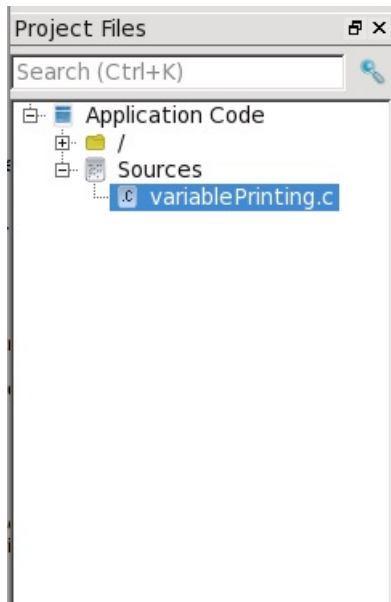
- Process Control:
- Process Group: Information sur un process ou un thread
- Project Files: find sources files
- Source Code
- Variables and stack of current process/thread
- Evaluate Window
- I/O, beakpoints, watchpoints,stack,tracepoint

Process Control - Process Group




- Play/Continue
- Pause
- Add Breakpoint
- Step Into
- Step Over
- Step Out
- Down Stack Frame
- Up Stack Frame

Source Codes



Variables - Stackframe

Locals	Current Line(s)	Current Stack
Current Line(s)		 x
Variable Name	Value	

```

variablePrinting.c x
48 /* Compute difference between elements of array2 and array1 */
49 for (indx = 0; indx < nelem; indx++)
50 {
51     del[indx] = array2[indx] - array1[indx];
52 }
53
54 /* Print the computed differences */
55 printf("The difference in the elements of array2 and array1 are: ");
56 printArray(nelem, del);
57
58 free(array1);
59 free(array2);
60 free(del);
61
62 return 0;
63 }
64
65 void initArray(const int nelem_in_array, int *array)
66 {
67     for (indx = 0; indx < nelem_in_array; indx++)
68     {
69         array[indx] = indx + 1;
70     }
71 }
72
73 int squareArray(const int nelem_in_array, int *array)
74 {
75     int indx;
76
77     for (indx = 0; indx < nelem_in_array; indx++)
78     {
79         array[indx] *= array[indx];
80     }
81     return *array;
82 }
83
84 void printArray(const int nelem_in_array, int *array)
85 {
86     printf("[ ");

```

Current Stack	
Stack Arguments	
#1	main () at /home/viryo/formation/cours-calcul20
#0	initArray (nelem_in_array=10, array=0x601010)

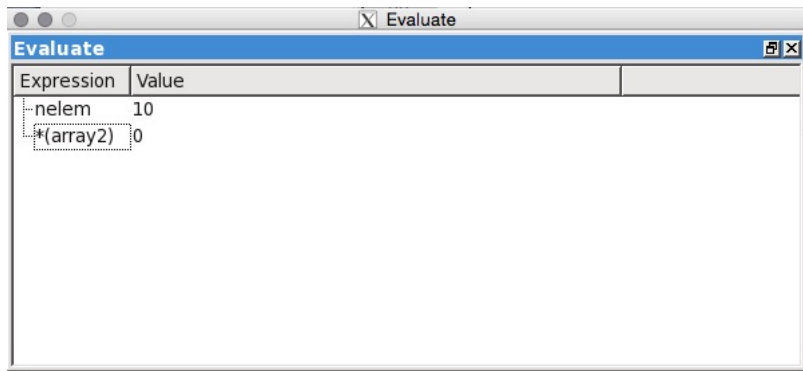
Current Line(s)	
Variable Name	Value
nelem_in_array	10
indx	0

Locals	
Variable Name	Value
array	0x601010
nelem_in_array	10

Stacks Tracepoints Tracepoint Output				Evaluate
				Expression Value
Line	Function	Condition	Start After	
.c 36			0	nelem <No symbol "nelem" in current
.c 42			0	*(array2) <No symbol "array2" in current

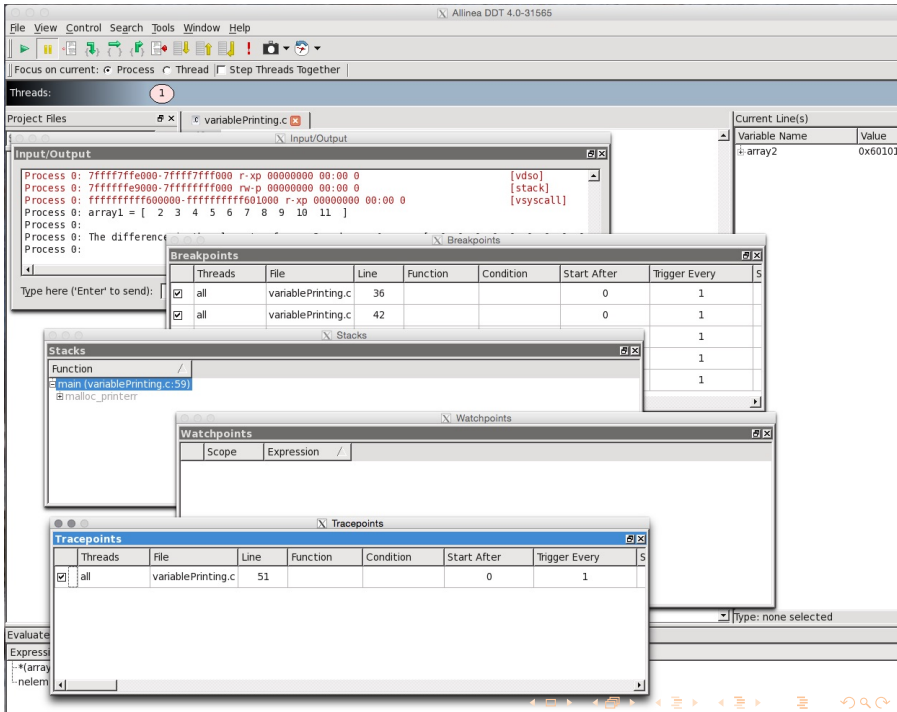
Type: none selected

Evaluate Window



Ecrire comment faire passer une variable ou une expression dans cette fenêtre

- Expression
- Value



Fenêtre visualisation des contrôles

- Input/Output
- Breakpoints
- Watchpoints
- Stacks
- Tracepoints
- Tracepoints Output

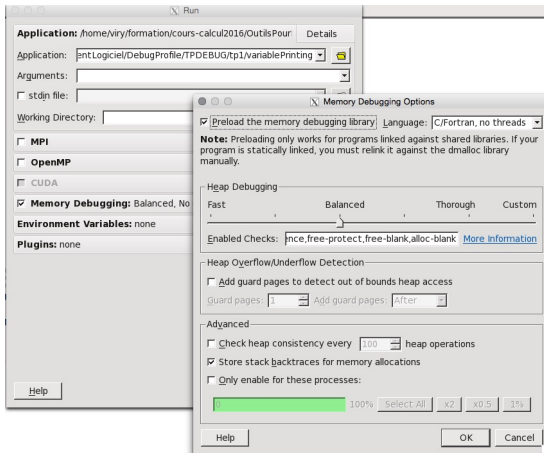
Analyser la mémoire du programme

- Analyse les allocations et désallocations et effectue des contrôles de dépassement des bornes du “heap”
- N'est pas activé par default, positionné au lancement du run.
- Différents niveaux de débouillage mémoire (voir help DDT)

Rappels: différentes zones de la mémoire d'un programme:

- **Mémoire vive**
- **Stack**: partie de la mémoire locale à l'appel d'un sous-programme. On pourra vérifier la cohérence du contenu des variables au passage des arguments.
- **Heap**: partie de la mémoire utilisée lorsqu'un tableau est alloué dynamiquement.
- **Caches**: vérifier les caractéristiques et la façon dont ils sont utilisés.
- **Registres**:

Starting the Memory Debugger



- Tools > Current Memory Usage (graphiques, tableaux)
- Tools > Overall Memory Stats (graphiques, tableaux)

Analyse de performances

Quelles performances?

Le code fonctionne, compile sans warnings, est validé par des tests, mais les performances ne sont pas là, il faut **analyser/optimiser**.

Attention : optimiser peut nécessiter un investissement en temps important pour des résultats pas forcément enthousiasmants ...

Optimiser "au hasard" sera contre-productif. Il est donc impératif d'**analyser les besoins** (veut-on aller plus vite, traiter des problèmes plus gros ...) puis les **performances relativement à ces besoins**.

Points clés:

- **Utilisation de la mémoire** (quelle mémoire et en quelle quantité?).
- Utilisation du (des) cpu(s) : **identifier les parties coûteuses** du code.
- Les entrées/sorties.

⇒ **avoir une idée claire de l'endroit où l'effort d'optimisation devra être fourni.**

Méthodologie conseillée

- Valider le programme à optimiser
- Utiliser des codes déjà optimisés ou s'y ramener
- Utiliser des algorithmes performants
- Analyser le code, et se concentrer sur les sections critiques
- Utiliser les capacités d'optimiser du compilateur
- Effectuer des modifications dans le code pour optimiser les sections critiques

Compteurs Hardware

- Petits registres comptabilisant les occurrences d'événements associés à des fonctions du processeur
 - Instructions load/store
 - Opérations Floating Point
 - Nombre de cycles d'attente d'accès mémoire
 - Cache misses, TLB misses,
 - ...
- Fournies des informations utiles à l'explication d'un manque de performance de certaines parties de codes
- Le suivi de ces événements permet d'affiner l'adaptation du code à l'architecture cible
- Nécessité d'avoir des outils qui exploitent ces compteurs (VTune, PAPI, pfmon, bpmon dans bullxde,...)

Floating-point Instructions

- La majorité des applications scientifiques sont **dominées par des opérations d'arithmétique flottant**.
- Des **fonctions ont été ajoutées au FPU** (Floating-Point Unit) des processeurs modernes pour améliorer les performances de ces opérations (**Multiply/add**,...).
- Comprendre les **implications du standard IEEE 754** utilisé par les **architectures modernes** peut permettre une **optimisation par un plus grand nombre de compilateurs**.
- **FLOPS** : opérations à virgule flottante par seconde.

Le nombre de **FLOPS** est une mesure commune de la vitesse d'un processeur.

MFLOPS mauvais indice de performance d'un code.

S'assurer de la **performances des algorithmes**.

Division

- La division est une opération beaucoup plus coûteuse que l'addition et la multiplication

Pas de pipeline et beaucoup plus de cycles

- Exemple de bonne pratique

```
do i=1,n  
  x(i)=a(i)/b  
end do
```

sera remplacé par

```
rb=1./b  
do i=1,n  
  x(i)=a(i)*rb  
end do
```


Performance, Puissance, FLOPS/sec

- Est ce que seul les GFLOPS/sec comptent?

NON

- Il faut aussi **déplacer les données** et alimenter le processeur.

$$t = T_m + T_c = n_m t_m + n_c t_c = n_c t_c \left(1 + \frac{t_m}{t_c} \frac{1}{q}\right)$$

$$q = \frac{n_c}{n_m} (\text{FLOPS/bytes})$$



Mesures de performance

◀ Mesure du temps

- **Global** : User CPU (wall ou elapsed time), autres statistiques.
- **Partie de code**: appel à une ou plusieurs fonctions (C, Fortran, system...)

◀ Profiling: Analyse détaillée de performance

- **Nombre d'appels des procédures**, temps passé dans chaque procédure, graphe des appels...
- **Utilisation des compteurs hardware**: comptage ou statistique sur des événements

◀ Traçage: analyse et/ou visualisation à posteriori de l'exécution

- Enregistrement des **informations sur des événements** (caches misses, ...) par process/thread pendant l'exécution
- **Reconstruction dynamique** du comportement du programme à partir de ces informations
- Nécessite l'**instrumentation du code** (compilateur, outils,...)

Mesures de performance

- Mesure globale du temps (autres statistiques)

commande **time**

- Mesure du temps d' une partie de code par une fonction explicite du langage (C, Fortran, python,...)
- Analyse complète de performance du code

Outils standard de profiling: **prof**, **gprof**, **Vtune**, **valgrind**,...

Mesure globale du temps - time

◀ time (commande de base UNIX/LINUX)

- ⌞ Temps utilisateur, temps système, elapsed ou wall time
- ⌞ CPU ou Elapsed time = Temps utilisateur + temps système
- ⌞ Sorties
 - **Real** : elapsed time
 - **User** : user time
 - **System** : system time

◀ Propriétés

- ⌞ Résolution de l'ordre de 10ms
- ⌞ Un "system_time" disproportionné peut indiquer un dysfonctionnement
 - Nombre important de "floating-point exceptions "
 - Nombre important de "page faults" ou de "misaligned memory access" ...

◀ Propriétés

CPU time < elapsed time

- Partage de la machine
- Grand nombre d'I/O
- Largeur de bande requise > largeur de bande de la machine
- Pagination et/ou swap important ...

Fonctions explicites de mesure du temps

Appel explicite de sous-programmes à l'intérieur du code , quel que soit le langage.

- Fonction `system_clock()`: elapsed time
- Procédures de la norme Fortran95 (10ms)
 - `cpu_time()` : temps CPU
 - `date_and_time()` : elapsed time
- Dans les langages de scripts python (`timeit`), R (), ...
- Hors standard: `fonctions constructeurs`.
- ...

Profiling

- ◀ Permet de déterminer les **parties de code les plus consommatrices** avant le travail d'optimisation.
- ◀ **Récupère des informations** sur:
 - ✧ Le **temps**, le **nombre d'appels des procédures**, le **graphe des appels**, des statistiques sur des compteurs hardware. . .
 - ✧ **Différentes granularités**, fonction, boucle, bloc de base et même ligne de code
- ◀ **Coût de développement relativement faible**
- ◀ **Plusieurs types de profiling** :
 - ✧ **PC Sampling**: interruption périodique du système ou trap des compteurs hardware. Ne nécessite en général pas de modification du code .
 - ✧ **Basic-Bloc Counting** : l'unité d'analyse est le bloc de base
 - ✧ **Méthode intrusive** en insérant directement des instructions de mesure dans le code (profiler ou manuellement)

Outils de profiling

◀ Méthodes non intrusives

- ✧ prof,gprof
- ✧ Marmot/MUST : MPI correctness checking.
- ✧ TAU (OpenMP, MPI, MPI/OpenMP)
- ✧ INTEL:
 - VTune.
 - TraceAnalyser / TraceCollector :
- ✧ VALGRIND
- ✧ HPCToolkit
- ✧ Scalasca, MAQAO, SCORE-P, Vampir, TAU : large-scale parallel performance analysis

◀ Méthodes intrusives

- ✧ PAPI : bibliothèque standard d'analyse de performances, support de nombreux outils.

Trois étapes pour le profiling `prof`, `gprof`

◀ Instrumentation automatique du programme

➤ Par le compilateur: **prof**, **gprof**

◀ Exécution du programme instrumenté

Crée le fichier de données pour le profiler
(`mon.out`, `gmon.out`, ...)

◀ Utilisation du profiler pour l'extraction et la lecture des résultats: **prof**, **gprof**

prof : CPU_TIME profiling

- **Compilation** : option **-p**
- **Profiling** : **prof** ou **gprof**

Trois étapes

- 1 **f90** <options> **-p** -o prog prog.f
- 2 **prog** # Exécution - création du fichier mon.out (gmon.out)
- 3 **prof prog mon.out > prof.list** # analyse du fichier mon.out

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
99.81	4.82	4.82	11	0.44	0.44	matmul
0.21	4.83	0.01				__intel_memset
0.00	4.83	0.00	11	0.00	0.00	mkl_matmul
0.00	4.83	0.00	2	0.00	0.00	init_matrices
0.00	4.83	0.00	1	0.00	4.82	main

- % time : pourcentage du temps total par cette routine.
- cumulative of second : temps cumulé par cette routine et les précédentes.
- self second : temps cumulé par cette routine.
- calls : nombre d'appels
- self ms/call : temps moyen par appel pour cette fonction.
- Total ms/call : temps moyen par appel pour cette fonction et ses descendants.
- name : nom

Un ensemble d'outils (debug, profile, optimisation, ...) pour le debug et l'analyse de code. <http://valgrind.org>

Entre autres:

- **memcheck** : detection d'erreurs liées à la mémoire,
- **cachegrind** : simulation du comportement des caches
- **callgrind** : profile (graphes d'appel des fonctions ...)
- **massif** : mesure de la quantité de mémoire utilisée dans le tas (heap) et la pile (stack)

Valgrind : fonctionnement général

```
valgrind --tool=XXX --option=valeur ./prog_name
```

XXX == memcheck, callgrind, cachegrind, massif ...

- + : non intrusif
- - : ralentissement (10-50 fois)
- - instrumente tout (y-compris librairies dynamiques liées)
- éventuellement : compiler avec -g et -O0, -Wall (fonction de l'outil utilisé)

memcheck

Détection des fuites mémoire (memory leak) i.e. blocs alloués dynamiquement (heap) et non libérés, des mauvaises initialisations, des accès illégaux ...

```
valgrind --tool=memcheck --show-reachable=yes --leak-check=full ./a.out
```

```
==14381== Memcheck, a memory error detector
==14381== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==14381== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==14381== Command: ./a.out 1
==14381==
==14381== HEAP SUMMARY:
==14381==    in use at exit: 4 bytes in 1 blocks
==14381==    total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==14381==
==14381== 4 bytes in 1 blocks are still reachable in loss record 1 of 1
==14381==    at 0x4A05FDE: malloc (vg_replace_malloc.c:236)
==14381==    by 0x3D5A280E91: strdup (in /lib64/libc-2.12.so)
==14381==    by 0x4005D2: __case_1 (in /home/perignon/ced/test-valgrind/a.out)
==14381==    by 0x4008AC: main (in /home/perignon/ced/test-valgrind/a.out)
==14381==
==14381== LEAK SUMMARY:
==14381==    definitely lost: 0 bytes in 0 blocks
==14381==    indirectly lost: 0 bytes in 0 blocks
==14381==    possibly lost: 0 bytes in 0 blocks
==14381==    still reachable: 4 bytes in 1 blocks
==14381==    suppressed: 0 bytes in 0 blocks
==14381==
==14381== For counts of detected and suppressed errors, rerun with: -v
==14381== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Simulation de l'utilisation du premier et du dernier niveau de cache.

```
valgrind --tool=cachegrind ./a.out
```

- affichage de l'utilisation des caches
- génère un fichier cachegrind.out.PID, PID=process id du job lancé avec valgrind

Pour post-traiter le fichier (avec les infos fonction par fonction)

```
cg_annotate cachegrind.out.PID [file.c]
```

```
==20572== I   refs:      1,791,459
==20572== I1  misses:      1,400
==20572== LLi misses:      1,338
==20572== I1  miss rate:    0.07%
==20572== LLi miss rate:    0.07%
==20572==
==20572== D   refs:      665,052 (472,117 rd  + 192,935 wr)
==20572== D1  misses:      11,731 ( 10,069 rd  +   1,662 wr)
==20572== LLD misses:       7,208 (   6,012 rd  +   1,196 wr)
==20572== D1  miss rate:    1.7% (   2.1%   +   0.8% )
==20572== LLD miss rate:    1.0% (   1.2%   +   0.6% )
==20572==
==20572== LL refs:      13,131 ( 11,469 rd  +   1,662 wr)
==20572== LL misses:       8,546 (   7,350 rd  +   1,196 wr)
==20572== LL miss rate:    0.3% (   0.3%   +   0.6% )
```

Profiling et graph d'appel des fonctions,

```
valgrind --tool=callgrind ./a.out
```

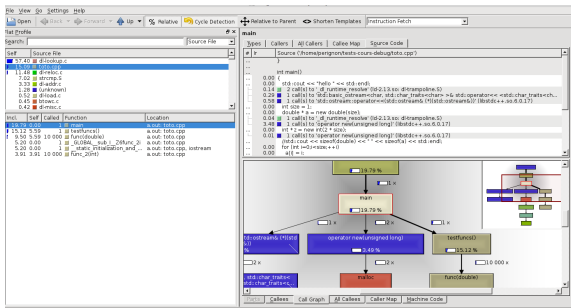
- affichage du nombre d'instructions exécutées,
- génère un fichier callgrind.out.PID, PID=process id du job lancé avec valgrind

Pour post-traiter le fichier (avec les infos fonction par fonction)

```
callgrind_annotate callgrind.out.PID [file.c]
```


Interface graphique à callgrind et cachegrind. Très bien mais pas toujours disponible sur les clusters de calcul.

Equivalent sur macos : qcachegrind.



Mesure de la mémoire utilisée dans le tas (heap) ou la pile (stack)

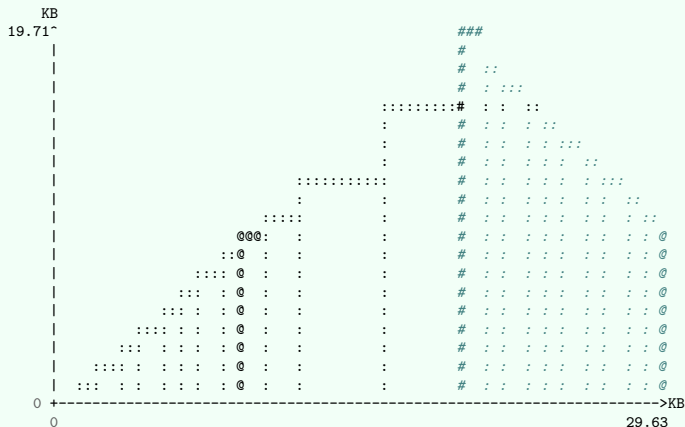
```
valgrind --tool=massif ./a.out
```

Post-traitement:

```
ms_print massif.out.PID
```

massif I

```
-----  
Command:          ./a.out  
Massif arguments:  --time-unit=B  
ms_print arguments: massif.out.9259  
-----
```



Number of snapshots: 25
Detailed snapshots: [9, 14 (peak), 24]

massif II

n	time(B)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
0	0	0	0	0	0
1	1,016	1,016	1,000	16	0
2	2,032	2,032	2,000	32	0
3	3,048	3,048	3,000	48	0
4	4,064	4,064	4,000	64	0
5	5,080	5,080	5,000	80	0
6	6,096	6,096	6,000	96	0
7	7,112	7,112	7,000	112	0
8	8,128	8,128	8,000	128	0
9	9,144	9,144	9,000	144	0

98.43% (9,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

->98.43% (9,000B) 0x400626: main (mass.cpp:20)

Valgrind - Conclusion

Valgrind est un outil complet, facilement disponible et relativement simple à utiliser

⇒ un réflexe indispensable: lancer valgrind (au moins memcheck) systématiquement sur son code.

Pourquoi PAPI

PAPI est un standard

- Les compteurs Hardware existent sur la plupart des processeurs.
- Les mesures de performances associées à ces compteurs varient suivant les processeurs.
- Il y a quelques API permettant de les utiliser, pas toujours simples, pas toujours opérationnelles...

But de PAPI

- Fournir une API portable, simple à utiliser pour accéder aux compteurs hardware sur la majorité des plates-formes HPC
- Définir des métriques de performances communes à un grand nombre de plates-formes fournissant des informations utiles à l'optimisation de codes.
- Encourager les vendeurs à standardiser les interfaces et la sémantiques des compteurs hardware.
- De nombreux outils d'analyse de performance sont construits sur PAPI.

MMPerftools

[MMPerftools](#) is a collection of performance tools for HPC environments.

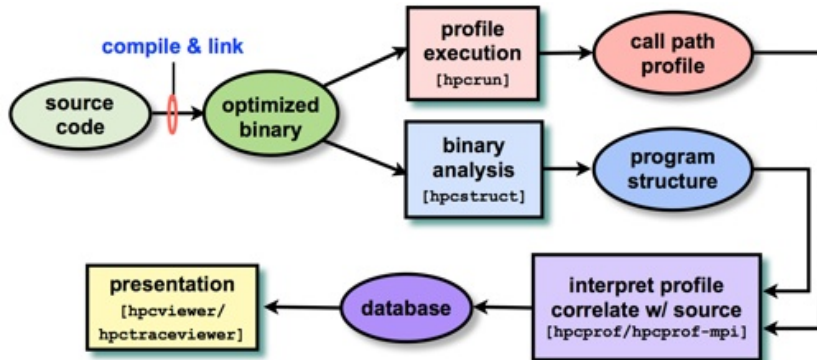
The tools allow low-overhead measurement of CPU metrics, memory usage, I/O and MPI.

- [papiex](#) : provides a simple and uniform interface to collect detailed job, process and thread-level statistics using PAPI.
- [mpiex](#) : to perform high-speed, low-overhead, MPI profiling using mpiP
- [hpcex](#) : to perform, detailed, call-stack sampling and tracing based on hardware performance metrics using HPCToolkit

See [sample output for a 4 task MPI program](#)

HPCToolkit (BULLX)

HPCToolkit est un ensemble d'outils permettant de mesurer et d'analyser les performances d'un programme (OpenSource)



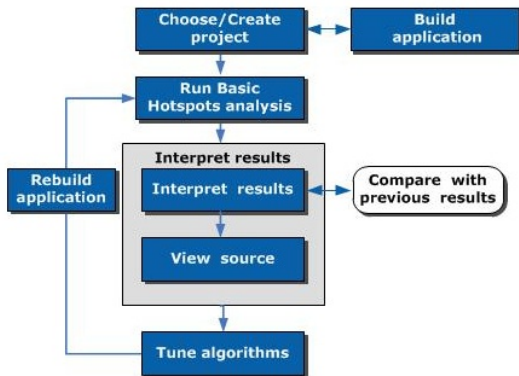
HPCToolkit (suite)

- **hpcrun** : mesure des indicateurs de performances contextuels pendant l'exécution.
- **hpcstruct** : récupère la structure statique du programme.
- **hpcprof, hpcprof-mpi** : mise en relation des indicateurs de performances et le source de l'application.
- **hpcviewer, hpctraceviewer** : deux interfaces graphiques permettant d'analyser de façon interactive les résultats des analyses.

Pour en savoir plus sur HPCToolkit

Evaluer les performances d'un code: VTUNE Amplifier

- Applications séquentielles ou parallèles (OpenMP, MPI, CUDA)
- Interface graphique (**amplxe-gui**) ou en mode commande (**amplxe-cl**)
- Finding Hotspots



VTune - Méthodologie

- Build the target with full optimizations, which is recommended for performance analysis.
- Create a Performance Baseline
- Create a new project in Vtune
- Run hotspots analysis
- Interpret Result Data
- Analyse Code
- Compare with previous result

Intel® VTune™ Amplifier XE

Analysis Types (based on technology)



Software Collector Any x86 processor, any virtual, no driver	Hardware Collector Higher res., lower overhead, system wide
Basic Hotspots Which functions use the most time?	Advanced Hotspots Which functions use the most time? Where to inline? – Statistical call counts
Concurrency Tune parallelism. Colors show number of cores used.	General Exploration Where is the biggest opportunity? Cache misses? Branch mispredictions?
Locks and Waits Tune the #1 cause of slow threaded performance – waiting with idle cores.	Advanced Analysis Dig deep to tune bandwidth, cache misses, access contention, etc.



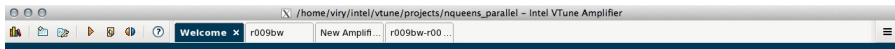
Software & Services Group, Developer Products Division

Copyright © 2014, Intel Corporation. All rights reserved.

*Other brands and names are the property of their respective owners.

Optimization
Notice

VTune (INTEL)



[Getting Started](#)

Welcome to Intel VTune Amplifier XE 2013

Current project: nqueens_parallel

- ▶ [Bandwidth Analysis](#)
- ▶ [Locks and Waits Analysis](#)
- ▶ [Cycles and uOps - Sandy Bridge / Ivy Bridge Analysis](#)
- ▶ [General Exploration - Knights Corner Platform Analysis](#)
- ▶ [New Analysis...](#)

- ▶ [New Project...](#)
- ▶ [Open Project...](#)

Recent Projects:

- > [mytest](#)
- > [BT-MZ](#)
- > [nqueens_parallel](#)

Recent Results:

- > [r009bw \[nqueens_parallel\]](#)
- > [r008bw \[nqueens_parallel\]](#)
- > [r007lw \[nqueens_parallel\]](#)
- > [r003lw \[mytest\]](#)
- > [r002cuop \[mytest\]](#)

VTune - Type d'analyse

Choose Analysis Type

Analysis Type

- Algorithm Analysis
 - Basic Hotspots
 - Advanced Hotspots
 - Concurrency
 - Locks and Waits
- Microarchitecture Analysis
 - General Exploration
 - Bandwidth**
- CPU Specific Analysis
 - Intel Core 2 Processor Analysis
 - Nehalem / Westmere Analysis
 - Sandy Bridge Analysis
 - Access Contention
 - Branch Analysis
 - Client Analysis
 - Core Port Saturation
 - Cycles and uOps
 - Memory Access
 - Port Saturation
 - Haswell Analysis
 - Knights Corner Platform Analysis
 - Custom Analysis
 - Cycles and uOps - Sandy Bridge / Ivy Bridge 0

Bandwidth Copy

Measure the data read and written to DRAM via the processor's integrated memory controller and determine whether the code is saturating available bandwidth. This analysis type is based on the hardware event-based sampling collection. Press F1 for more details.

Details

Events configured for CPU: Intel(R) Xeon(R) E5 processor

NOTE: For analysis purposes, Intel VTune Amplifier XE 2013 may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the Project Properties dialog.

Event Name	Sample After	LBR Filter	
CPU_CLK_UNHALTED.REF_TSC	2000003		Ref
CPU_CLK_UNHALTED.THREAD	2000003		Cor
INST_RETIRED.ANY	2000003		Inst
MEM_LOAD_UOPS_RETIRED.LLC_MISS	100007	None	Mis

☐ Collect stacks

☐ Estimate call counts

Chipset events to collect:

☒ Analyze memory bandwidth

☐ Analyze user tasks

Event mode:

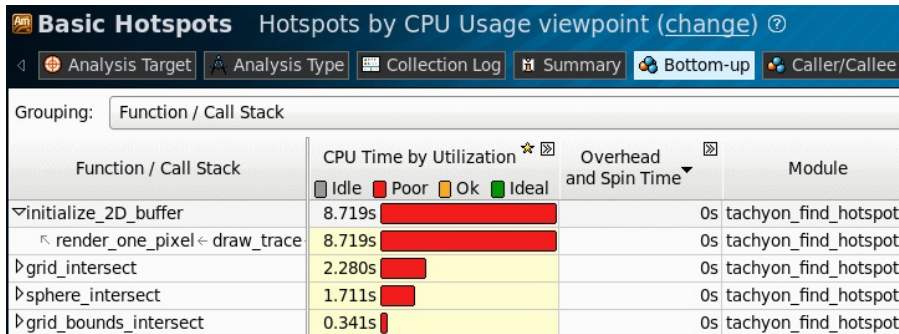
Start

Start Paused

Project Properties

Command Line...

VTune - Basics hotspots



VTune - Analyse du Code

Source		Assembly grouping: Address				
Source Line	Source	CPU Time: Total	Address	Source Line	Assembly	CPU Time: Total ...
		Idle Poor				Idle Poor O
116			0x40c0f4	118	movl %edx, (%rdi,%	0.923s
117	do {	0.901s	0x40c0f7	121	mov %r8d, %eax	3.008s
118	mem_array [j*mem_array_i_max+i] = *	2.000s	0x40c0fa	121	imul %r10d	
119			0x40c0fd	121	mov %r8d, %eax	1.050s
120	// Code to give the array accesses		0x40c100	121	sar \$0x1f, %eax	
121	if ((iteration_count % 3) == 0) j=j	5.388s	0x40c103	121	sub %eax, %edx	0.180s
122	else j=j+2;		0x40c105	121	lea 0x3(%rcx), %eax	
123	iteration_count++;	0.290s	0x40c108	121	add \$0x2, %ecx	0.970s
124	} while (j < mem_array_j_max);		0x40c10b	121	lea (%rdx,%rdx,2),	
125	}		0x40c10e	121	cmp %edx, %r8d	0.180s
126	/*****		0x40c111	121	cmovz %eax, %ecx	
127	/*****		0x40c114	117	cmp \$0xb3, %ecx	0.901s

TraceAnalyser –TraceCollector (INTEL - ITAC)

- Outil d'analyse de performances de programme séquentiel ou parallèle.
- Deux composants :
 - **Trace Collector** : permet la génération automatique de la trace de l'exécution d'un programme.
 - **Trace Analyser** : programme de visualisation graphique de la trace générée.
- Implémentée sur la plupart des plateformes.
- Documentation:

Objectifs :

Utilisation de gprof et valgrind

Description du TP :

Le TP peut se faire sur froggy ou (partiellement) sur votre machine locale.

- Pour récupérer les sources du TP:

```
git clone git@gitlab.com:CED-2017/Demos.git  
cd TPProfil
```

Dans le répertoire multmat, vous trouverez un fichier `tp.cxx` (programme principal) qui fait appel à des fonctions définies dans `matmul.cxx` avec un calcul 'naïf' de multiplication de matrices et un calcul via la librairie mkl.

- 1 Compilez `tp.cxx` et `matmul.cxx` avec les compilateurs intel (icpc). Il faudra utiliser l'option `"-mkl"`.
- 2 Utilisez `gprof` pour établir des diagnostics sur ces calculs de produits de matrice.

TP - valgrind

Utilisez les programmes du répertoire valgrind, avec les compilateurs gnu.

- 1 Testez valgrind/memcheck avec le programme valgrind-memcheck.c
- 2 Testez valgrind/massif avec le programme massif.cpp.
- 3 Utilisez valgrind callgrind/cachegrind sur le calcul de produit de matrices (tp.cxx + matmul.cxx)

Quelques remarques:

- Kcachegrind n'étant pas disponible sur froggy, vous pourrez l'utiliser sur votre machine locale avec les programmes matmul.cxx et tp.cxx qui sont dans le répertoire valgrind, en version non-mkl.
- Pour les calculs sur Froggy un script oar est également fourni dans les TP. Utilisation: Modifier la ligne correspondant au nom de l'exécutable, ajouter éventuellement des "module load" puis

```
oarsub -l ./oarscript.sh
```