Eléments de programmation et soumission de travaux Sur la machine BULL

Ce qui change par rapport au NEC et ce qui ne change pas

R. El Khatib – 08/04/2013

METEO FRANCE
Toujours un temps d'avance

Plan de la présentation

- Architecture
- Programmation
  - Optimisation
  - Parallélisation Open-MP
- Compilation
- Allocation des ressources
  - Cpus
  - Unité de coût
  - Mémoire
- Configuration
  - Namelists pour Arpege/Arome
  - Variables d'environnement
- Profilage élémentaire avec DrHook

Architecture générale

- Les machines NEC SX8R et SX9 sont des machines dites à **architecture vectorielle** *(cf image de l'escalator)* :
  - Caractérisées par des processeurs très puissants mais peu nombreux
  - La puissance de calcul exige un codage soigneux : la **vectorisation**
  - La puissance de calcul exige des tableaux de grande taille pour remplir les 256 **registres vectoriels**
- La machine Bull est une machine dite à architecture scalaire
  - « architecture scalaire » est une appellation impropre : il faudrait plutôt dire « **architecture à mémoire cache** » (« cache-based machine ») :

    cf. L1, L2, L3
  - **Les machines scalaires font aussi de la vectorisation**, mais selon une technologie différente : « Altivec » sur PPC, « SSE » sur Intel hier, « AVX » sur Intel et IBM aujourd'hui, « AVX2 » demain ...
  - Il faudrait plutôt parler de **micro-vectorisation** car la « longueur de vectorisation » équivalente vaut 2 (AVX aujourd'hui) ou 4 (AVX2 demain)

Les disques et le système de fichiers

- **Sur NEC,**
  - Le système de fichiers qui permet de voir n'importe quel fichier depuis n'importe quel nœud (« GFS ») est relativement **lent**
  - => pour obtenir de meilleures performances d'entrées/sorties, on s'efforce d'utiliser les **disques locaux** : il y a 1 disque local par noeud, visible seulement du nœud sur lequel il est branché
  - => synchroniser les disques locaux pour que leur contenus soient identiques d'un nœud à l'autre est possible, mais compliqué.
  - => On détourne aussi une partie de la mémoire vive pour en faire un système de fichiers local encore plus rapide
- **Sur BULL,**
  - Le système de fichiers qui permet de voir n'importe quel fichier depuis n'importe quel nœud (« **Lustre** ») est **conçu pour être rapide et adapté à un grand nombre de nœuds de calcul.**
  - Des disques locaux existent, mais les développeurs n'en aurons pas (ou très peu) besoin.

Le système d'exploitation

- **Sur NEC,**
  le système d'exploitation est un système Unix **propriétaire**
  - Impossible de choisir un autre compilateur
  - Impossible de choisir une autre librairie MPI pour les communications
  - Impossible de choisir un autre logiciel de profilage que *ftrace*
  - => Cela simplifie le choix ;-)
- **Sur BULL,**
  le système d'exploitation **Linux** permet l'installation de logiciels libres
  - Possibilité d'installer les compilateurs gcc/gfortran
    en alternative à icc/ifort d'Intel
  - Choix de librairies MPI :
    BullxMPI, IntelMPI, OpenMPI, MPICH2
  - Choix d'outils de profilage (Intel, « Open-source »)
  - **=> reste à choisir le meilleur pour chaque famille d'application**

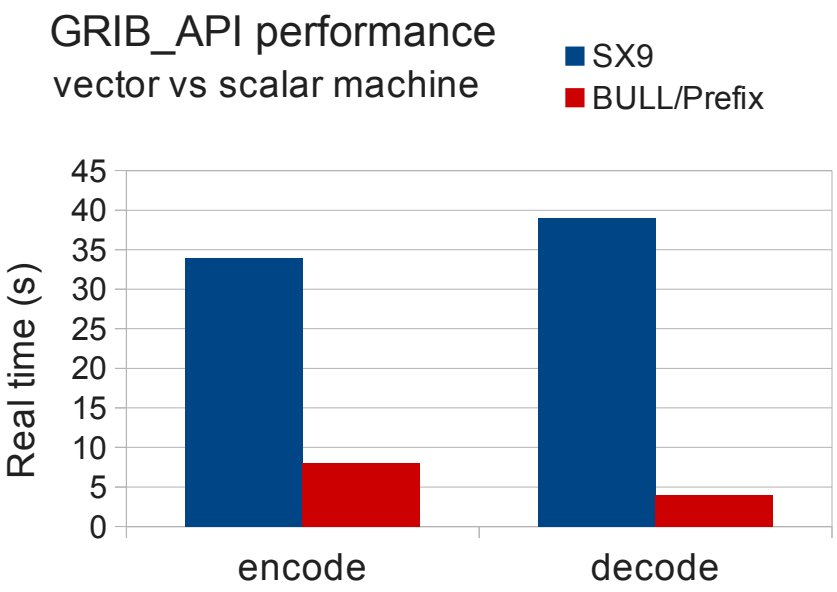
Vocabulaire

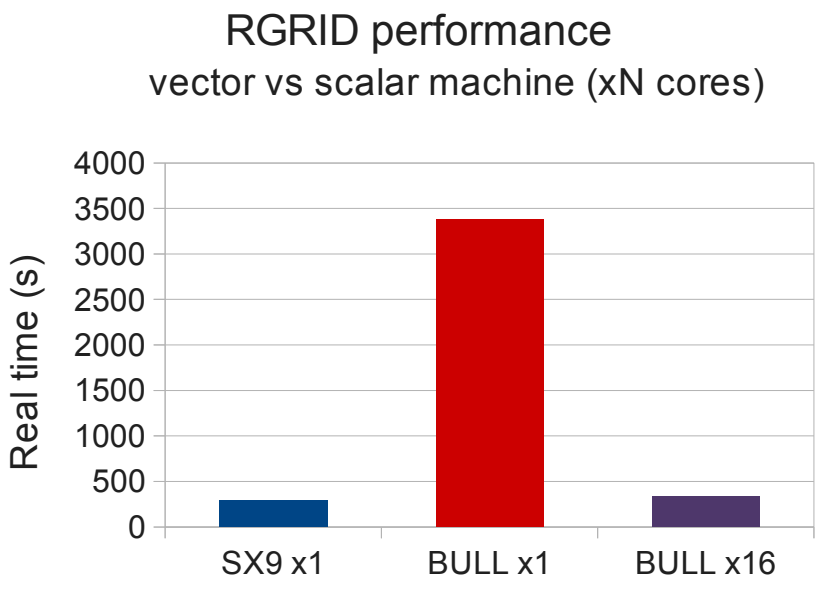
- **Sur NEC,**
  - Un noeud de calcul est constitué de 8 **processeurs** (Tori)
    ou 16 processeurs (Yuki/Kumo)
- **Sur Bull,**
  - Un noeud de calcul est constitué de 2 « sockets »,
    - Chaque « socket » est constitué de 8 « coeurs » (Prefix)
      ou 12 « coeurs » (Beaufix/Prolix)
      - Chaque « coeur » se dédouble
        en un coeur « physique » et un coeur « logique »,
        mais ce dédoublement est d'ordre logiciel

Dans la suite de la présentation, on fera l'amalgame entre « processeur » NEC et « coeur » (sous-entendu physique) Bull
Il y a 8 ou 16 processeurs par noeud sur NEC,
comme il y a 16 ou 24 coeurs par noeud sur Bull.

Performance d'un processeur

- La performance d'un processeur vectoriel ou scalaire dépend du profil de l'application
  - Pour certains codes, il n'y aura rien à faire car ils s'exécuteront plus rapidement que sur une machine vectorielle (« profil scalaire »)
  - pour d'autres, il faudra **optimiser** et **distribuer** ou **paralléliser**





METEO FRANCE
Toujours un temps d'avance

Optimisation, Distribution et Parallélisation

- **Optimiser** = augmenter la performance de l'application pour la même enveloppe de ressources (mémoire, processeurs)
- **Distribuer** = paralléliser les calculs par le biais de la librairie MPI = « Message Passing Interface »
- **Paralléliser** = paralléliser les calculs par le biais de directives « Open-MP » interprétées par le compilateur

Optimisation, Distribution MPI et Parallélisation Open-MP sont complémentaires

- En général,
  - Une application est **distribuée** en tâches MPI au plus haut niveau
    - Chaque tâche MPI est **parallélisée** en processus (threads) Open-MP
      - Chaque processus est **optimisé** pour réduire le temps de calcul

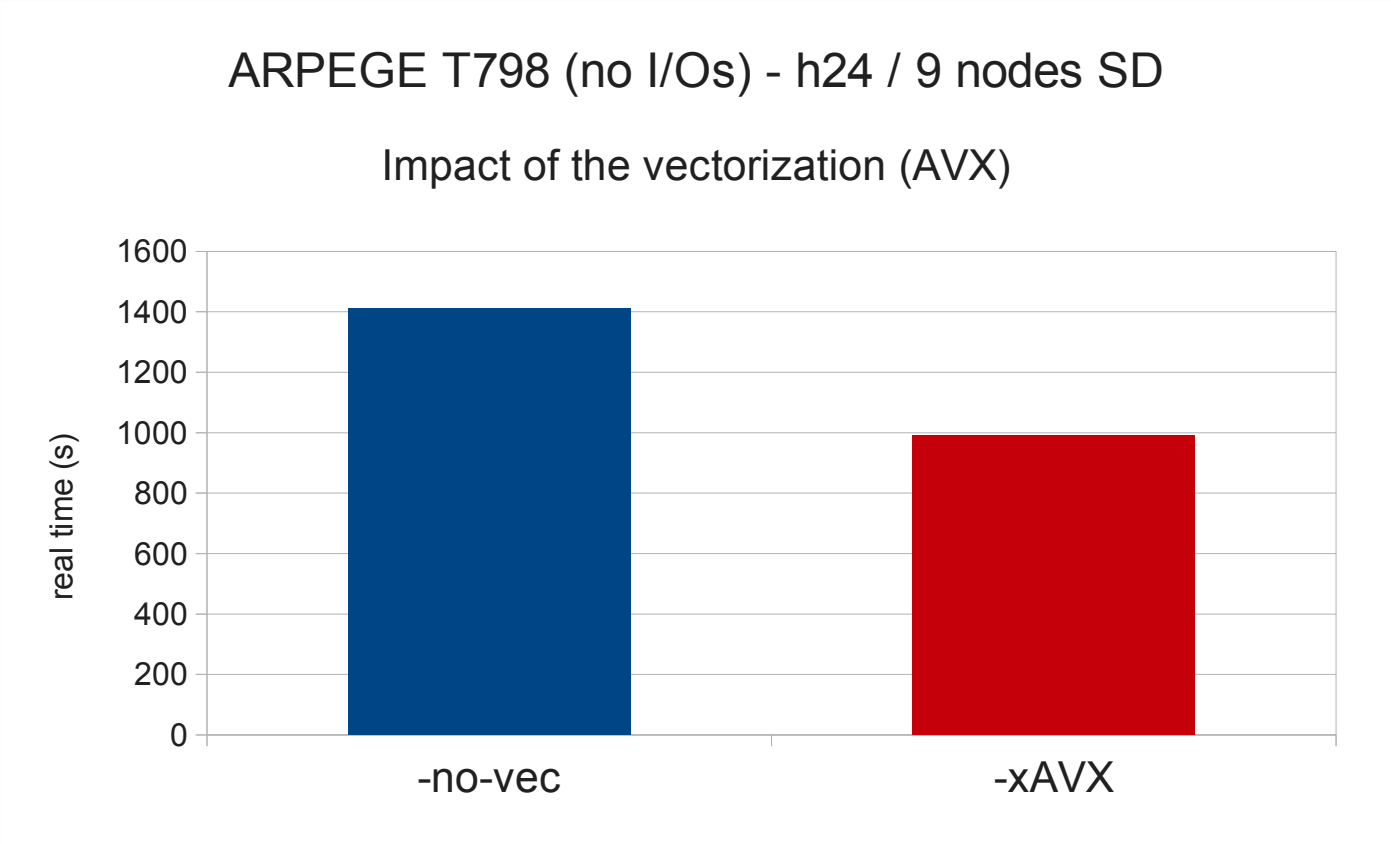
Optimisations (1)

- **Sur NEC,**
  - L'optimisation repose sur la **vectorisation**
  - **Longueur de vectorisation : 256**
  - La vectorisation peut nécessiter l'emploi de **directives** de compilation
  - Des calculs redondants dans une boucle bien vectorisée
    ne sont pas coûteux
- **Sur Bull,**
  - La **vectorisation** existe aussi
  - **Longueur équivalente de vectorisation : aujourd'hui 2** ;
    mais demain 4 ; et après-demain ?
  - Il y a peu de directives de compilation à utiliser
  - L'optimisation repose sur des technique assez basiques
    qu'on observait déjà sur les Cray-1 et Cray-2 (fin des années 80) !

Optimisations (2)

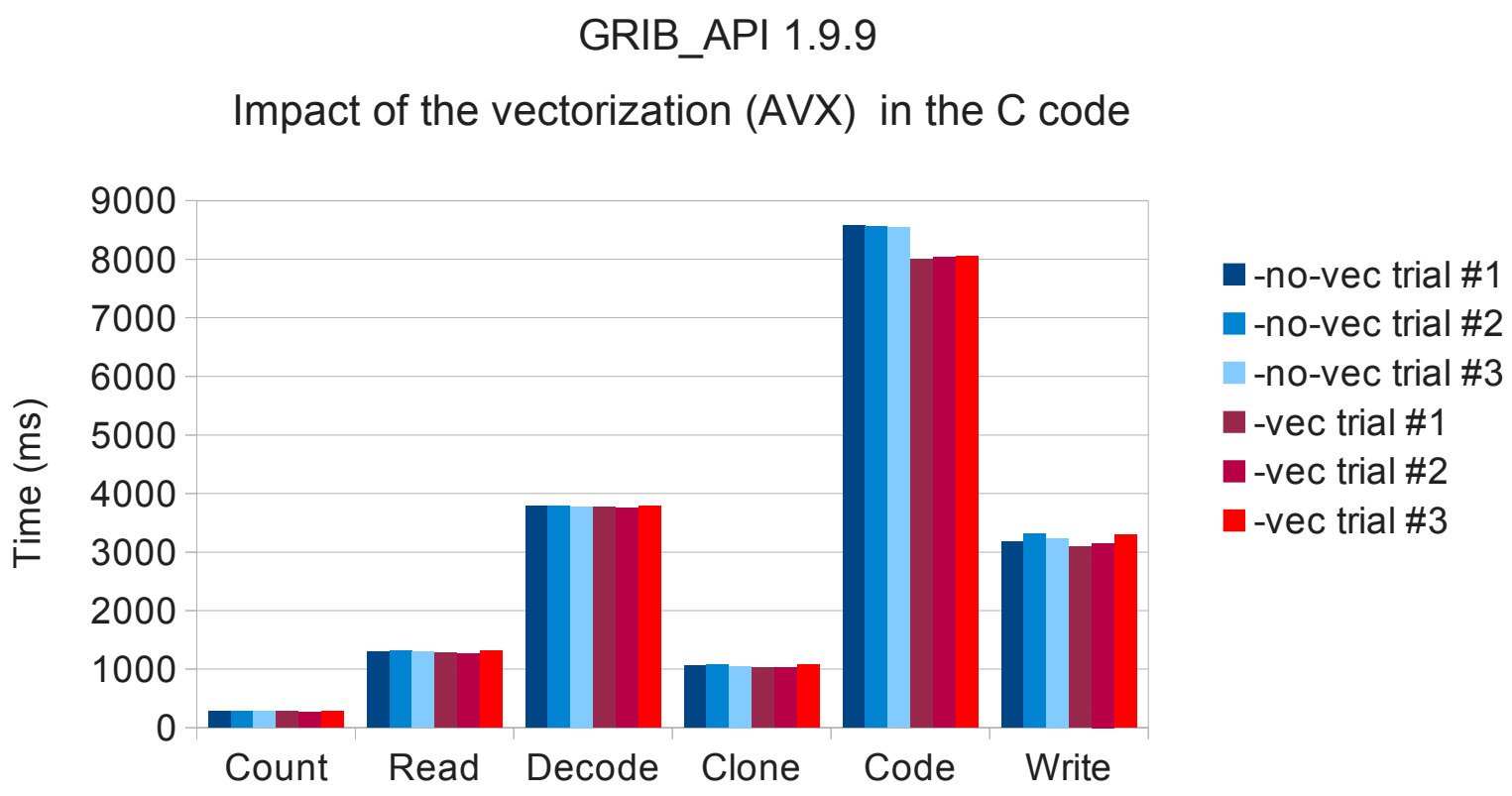
- **Exemples :**
  - C'est l'indice de tableau le plus à gauche qui varie le plus vite (en Fortran)
    => il doit correspondre à la boucle la plus interne
  - Eviter les mises à zéro de tableaux « par précaution »
    ou les copies de tableaux (dans ce cas, un pointeur peut faire l'affaire)
  - Les divisions coûtent beaucoup plus cher que les multiplications
  - Placer les blocs conditionnels (« IF ») à l'extérieur des boucles
  - Utiliser MIN, MAX, SIGN, ... plutôt que des « IF »
  - Aidez le compilateur : optimisez à sa place !
  - Des I/O binaires plutôt que formatés, ou encore mieux : pas d'I/O du tout !
- **A LIRE ABSOLUMENT :**
  http://www.ecmwf.int/services/computing/training/material/hpcf/optimisation.pdf
  pages 44 à 66
- **=> Une bonne pratique : continuez à vectoriser !**

Impact de la vectorisation sur Bull



Impact largement positif (30 %) sur un code au profil vectoriel

Impact de la vectorisation sur Bull



Impact légèrement positif pour l'encodage (5%)
sur un code au profil scalaire

Brève histoire de la parallélisation (1)

- **Années 80-90 : les premiers calculateurs multiprocesseurs étaient des machines à mémoire partagées :**
  - La programmation de la parallélisation était relativement similaire à ce que nous ferions aujourd'hui avec la parallélisation Open-MP
  - Mais l'augmentation du nombre de processeurs engendrait des latences mémoires rédhibitoires

Brève histoire de la parallélisation (2)

- **Début des années 2000 : rupture technologique : les machines à mémoire partagée sont remplacées par des machines à mémoire distribuée :**
  - Le calculateur est divisé en « noeuds » qui sont autant de calculateurs monoprocesseurs
  - Le programme est exécuté sur chaque noeud de calcul, mais pour des sous-ensembles de données :

    on « distribue » les données entre les noeuds
  - La mémoire est privée à chaque noeud de calcul, mais le programme sur un noeud peut échanger des données en mémoire avec le programme sur un autre noeud par le biais d'appels à des sous-programmes de communication :

    Librairie MPI = Message Passing Interface
  - Une mauvaise performance logicielle peut venir d'une distribution déséquilibrée des données, ou de communications trop lentes ou trop fréquentes

Brève histoire de la parallélisation (3)

- **Aux alentours de 2005 : les noeuds de calcul reçoivent plusieurs processeurs**

  => La programmation de la parallélisation devient mixte :
  - Distribution MPI entre les noeuds
  - Parallélisation à mémoire partagée Open-MP à l'intérieur d'un noeud
- **Aux alentours de 2010, pour limiter le coût énergétique :**
  - les processeurs deviennent « multi-coeurs »

    => plus nombreux
  - La fréquence d'horloge stagne ou baisse,
  - Mais la micro-vectorisation se développe
  - La quantité de mémoire par processeur diminue
- **A suivre ...**

Introduction à la parallélisation Open-MP

- Open-MP est un langage de programmation ***standard***
  - basé sur des ***instructions, directives*** ou ***constructions***
  - interprétables par le compilateur (fortran ou C)
  - Nécessite des options spéciales à la compilation et à l'édition de lien
  - Nécessite de définir le nombre de *threads* à l'exécution par la variable d'environnement **OMP_NUM_THREADS** (valeur par défaut : ??)
- La programmation Open-MP est *a priori* facile et non-intrusive ...
- ... mais elle recèle de nombreux pièges !

```
!$ OMP PARALLEL DO PRIVATE(JI)
DO JI=1,N
  ZA(JI)=ZB(JI)+ZC(JI)
ENDDO
!$ OMP END PARALLEL DO
```

*: Les « threads » Open-MP se **partagent** les itérations de la boucle*

=> JI doit être déclarée « privée » à chaque « thread »

Open-MP : Syntaxe

- Toute ligne commençant par «**!$** » est interprétée comme une **instruction exécutable** dès lors que le code est compilé avec Open-MP
- Toute ligne commençant par «**!$OMP** » est interprétée comme une **directive** Open-MP
- On peut écrire des instructions Open-MP sur plusieurs lignes

```
!$ PRINT*, 'CETTE LIGNE APPARAIT CAR LE PROGRAMME &
!$ &EST COMPILE AVEC OPEN-MP' ! Instruction exécutable
!$OMP PARALLEL DO &
!$OMP & PRIVATE(JI)  ! directive
DO JI=1,N
  ZA(JI)=ZB(JI)+ZC(JI)
ENDDO
!$OMP END PARALLEL DO
```

Open-MP : exemple de boucles parallélisées

On parallélise une boucle parce que le profilage indique que cette boucle coûte cher

On ne parallélise pas une boucle parce qu'elle est parallélisable

```
!$OMP PARALLEL DO PRIVATE(JI,JK)
DO JI=1,100
  DO JK=1,2
    ZA(JK,JI)=ZB(JK,JI) + ZC(JK,JI)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```

=> C'est la première boucle sous la directive OMP qui est parallélisée

```
DO JI=1,2
!$OMP PARALLEL DO PRIVATE(JK)
  DO JK=1,100
    ZA(JK,JI)=ZB(JK,JI) + ZC(JK,JI)
  ENDDO
!$OMP END PARALLEL DO
ENDDO
```

=> Paralléliser sur la boucle externe ne permettrait pas d'utiliser plus de 2 threads

Open-MP : portée du parallélisme

- **Ceci était un mauvais exemple :**

```
!$OMP PARALLEL DO PRIVATE(JI)
DO JI=1,N
  ZA(JI)=ZB(JI)+ZC(JI)
ENDDO
!$OMP END PARALLEL DO
```

- En principe on doit d'abord définir une **région parallèle**
- Et à l'intérieur on peut définir des **sections partagées :**

```
INTEGER, EXTERNAL :: OMP_GET_THREAD_NUM ! Récupère le numéro du thread courant
!$OMP PARALLEL PRIVATE(JI)
PRINT*,'CHAMPAGNE POUR LE THREAD ', OMP_GET_THREAD_NUM()
!$OMP DO SCHEDULE(DYNAMIC)
DO JI=1,N
  IF (MOD(N,2)==0) PRINT*,'CAVIAR POUR LE THREAD ', OMP_GET_THREAD_NUM()
ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

OPEN-MP : attributs PRIVATE (1)

- **Par défaut, toutes les variables locales dans le corps du sous-programme courant ont l'attribut « SHARED » :**
  Leurs valeurs sont partagées par tous les threads
- **=> Si une variable locale est en écriture dans la région parallèle, alors elle doivent être déclarée « PRIVATE » si sa valeur dépend du thread :**

```
REAL :: ZX, ZA(N)
!$OMP PARALLEL DO  PRIVATE(JI,ZX)
DO JI=1,N
  ZX=EXP(REAL(JI))
  ZA(JI)=ZX*ZC(JI)
ENDDO
!$OMP END PARALLEL DO
```

=> Analysez toutes les variables qui sont dans les membres de gauche ainsi que toutes les variables en argument de sous-programmes

=> Ne donnez pas l'attribut « PRIVATE » à toutes les variables !

Un attribut erroné, et la parallélisation est fausse !

OPEN-MP : attributs PRIVATE (2)

```
REAL ZA(N)
!$OMP PARALLEL DO  PRIVATE(JI)
DO JI=1,N
  CALL TOTO(JI,ZA)
ENDDO
!$OMP END PARALLEL DO

SUBROUTINE TOTO(K,PA)
USE MACHIN, ONLY : CST, B
INTEGER, INTENT(IN) :: K
REAL, INTENT(OUT)  :: PA(:)
REAL ZX(SIZE(PA))
ZX(K)=EXP(K)
PA(K)=CST*ZX(K)
B=REAL(K)
END SUBROUTINE TOTO

MODULE MACHIN
SAVE
REAL :: CST, B
!$OMP THREADPRIVATE(B)
END MODULE MACHIN
```

- **Par défaut, toutes les variables locales dans un sous-programme appelé ont l'attribut «PRIVATE» :**
- **Les variables qui ont l'attribut Fortran SAVE (variables en module le plus souvent) et qui sont en écriture dans la région parallèle *ne peuvent pas* être déclarées « PRIVATE ».**
- **Elle *peuvent* être « privatisées » à leur déclaration par un attribut «THREADPRIVATE» mais c'est une solution déconseillée : problèmes de maintenance, de performance et incompatibilité avec OOPS.**
- **=> N'utilisez de variables en module qu'en mode « lecture »**
- **=> Passez les variables à écrire en arguments**

OPEN-MP : équilibrage par « scheduling »

```
!$OMP PARALLEL DO SCHEDULE (STATIC) PRIVATE(JI)
DO JI=1,N
  ZA(JI)=ZB(JI)+ZC(JI)
ENDDO
!$OMP END PARALLEL DO
```

=> La répartition des itérations est prédéfinie de façon équilibrée

=> Equilibrage optimal si OMP_NUM_THREADS divise N

=> Efficace si les calculs sont naturellement équilibrés d'une itération à l'autre

```
!$OMP PARALLEL DO SCHEDULE (DYNAMIC,1) PRIVATE(JI)
DO JI=1,N
  IF (ZB(JI) >= ZC(JI)) THEN ; ZA(JI)=ZB(JI)+ZC(JI)
  ELSE ; ZA(JI)=ZB(JI)+2*ZC(JI) ; ENDIF
ENDDO
!$OMP END PARALLEL DO
```

=> Le premier thread disponible traite la première itération non-traitée

=> N'a de sens que si N > OMP_NUM_THREADS

=> Réduit le déséquilibre entre itérations

Pathologies classiques de bogues Open-MP

- **Mon programme Open-MP fonctionne le plus souvent, mais de temps en temps il plante**
  - Problème de variables privées/partagées
  - Variables non-initialisées
- **Mon programme Open-MP fonctionne, mais il ne donne pas deux fois de suite le même résultat**
  - Problème de variables privées/partagées
  - Variables non-initialisées
- **Mon programme Open-MP plante sur Intel en « segmentation violation » même sans optimisation, pourtant il fonctionne très bien avec gfortran**
  - Problème de variables privées/partagées
  - Variables non-initialisées
  - Taille de la pile insuffisante ou tableaux automatiques trop gros
  - Confusion entre région parallèle et section partagée

  (cf tableaux « allocatable »)

OPEN-MP : tableaux « ALLOCATABLE »

Problème :

```
REAL, ALLOCATABLE :: ZA( :)
! ZA : tableau de travail
ALLOCATE(ZA(N2))
!$OMP PARALLEL DO &
!$OMP & PRIVATE(JI,JK,ZA)
DO JI=1,N1
  DO JK=1,N2
    ZA(JK)=...
  ENDDO
ENDDO
!$OMP END PARALLEL DO
DEALLOCATE(ZA)
```

=> A l'entrée dans la zone parallèle, ZA, déjà alloué une fois, est dupliqué en autant de threads ; mais qu'est ce qui est désalloué ??

=> Plantage ou pas dans le DEALLOCATE selon le compilateur

Solution :

```
REAL, ALLOCATABLE :: ZA( :)
! ZA : tableau de travail
!$OMP PARALLEL &
!$OMP & PRIVATE(JI,JK,ZA)
ALLOCATE(ZA(N2))
!$OMP DO
DO JI=1,N1
  DO JK=1,N2
    ZA(JK)=...
  ENDDO
ENDDO
!$OMP END DO
DEALLOCATE(ZA)
!$OMP END PARALLEL
```

Boucles impropres à la vectorisation ou à la parallélisation

Problème :

```
REAL :: ZA(N)
I=0
DO JI=1,N
  ...
  I = I+1
   ... ZA(I) ...
  ...
ENDDO
```

=> La boucle ne vectorise pas réellement car sa longueur de vectorisation vaut 1 !
=> Si Open-MP,
I et ZA doivent-ils être partagés ou privés ??

Solution à rechercher :

```
DO JI=1,N
  ...
  I = f(JI)
   ... ZA(I) ...
  ...
ENDDO
```

Solution desespérée :

```
I=0
!$OMP PARALLEL DO PRIVATE(JI)
DO JI=1,N
  ...
!$OMP CRITICAL ! Section séquentielle
  I = I+1
   ... ZA(I) ...
!$OMP END CRITICAL
  ...
ENDDO
!$OMP END PARALLEL DO
```

Open-MP : pour aller plus loin

Cours de l'IDRIS (en français) – html ou pdf :

http://www.idris.fr/data/cours/parallel/openmp/

Cours du CEPMMT (en anglais) :

http://www.ecmwf.int/services/computing/training/material/hpcf/intro_OpenMP.pdf

Le compilateur (1)

- Sur **NEC**,

  le compilateur est un ***cross-compilateur*** :
  - On compile sur la frontale (scalaire), on exécute sur les noeuds de calcul (vectoriels)
  - Nécessite des commandes spécifiques : **sx**mpif90, **sx**ar **sx**nm, **sx**ld...
  - Cela rend la compilation de certaines librairies difficiles si la cross-compilation n'est pas supportée (grib_api par exemple)
- Sur **BULL**,

  le compilateur est un **compilateur normal** :
  - Les programmes compilés sur les noeuds de calcul ou les noeuds de login (= frontale) peuvent s'exécuter sur n'importe quel noeud
  - Pas de commandes spécifiques

Le compilateur (2)

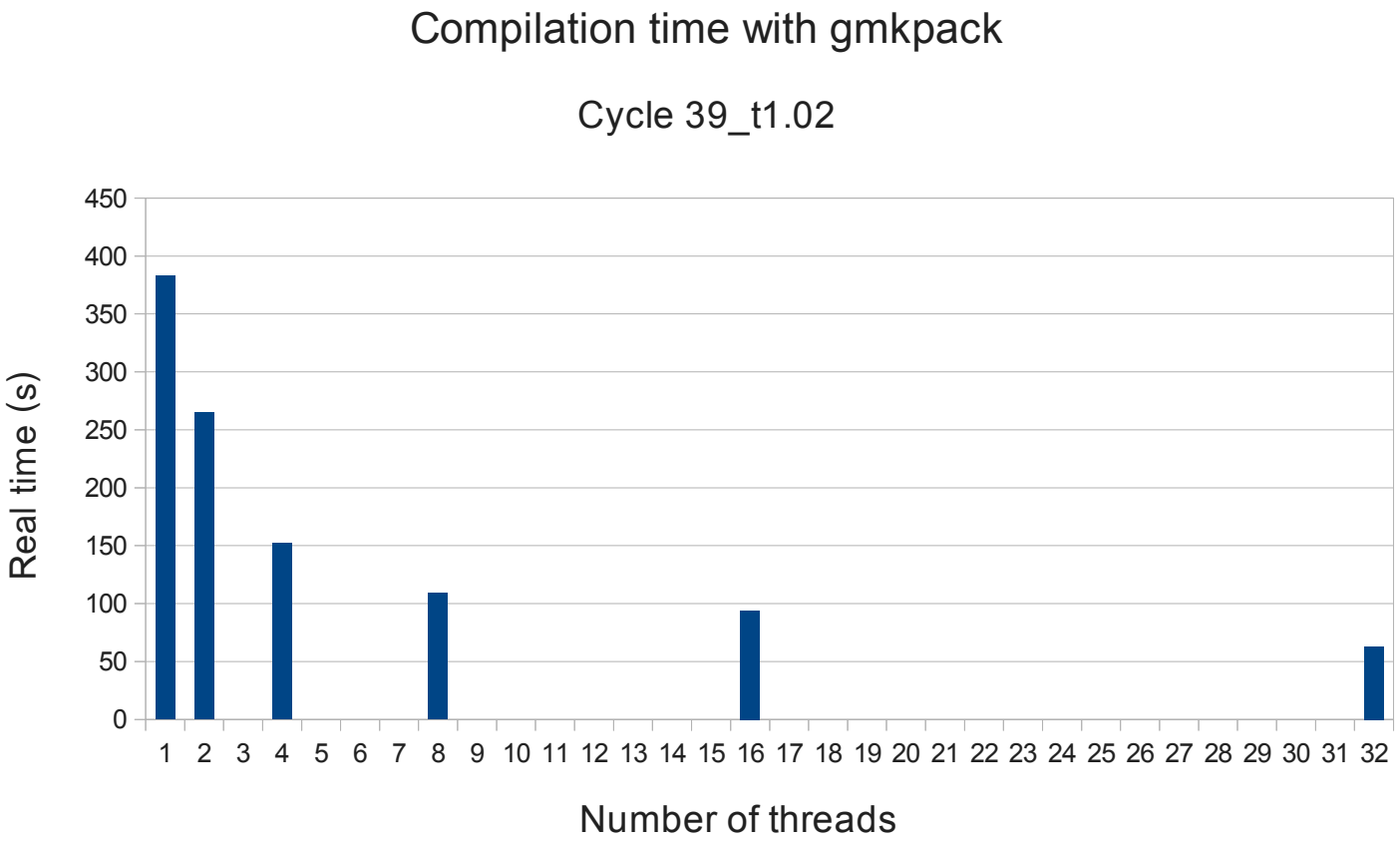
- Sur **NEC**,

  le compilateur (NEC) est un compilateur **propriétaire** :
  - Le nombre d'utilisations simultanées est *illimité*
  - En cas de bug ou de développement nécessaire à faire dans le compilateur, nous sommes en contact direct avec le fabricant
- Sur **BULL**,

  le compilateur (Intel) est un compilateur sous **licence** :
  - Le nombre d'utilisations simultanées est limité par un nombre de licences (25) qui entre dans le prix de la machine
  - => Evitez de lancer simultanément plusieurs jobs de compilation
  - => Mais vous pouvez réduire le temps d'occupation d'une licence en compilant sur plusieurs *threads*

    (dans gmkpack : export GMK_THREADS=...)
  - En cas de bug ou de développement nécessaire à faire dans le compilateur, notre interlocuteur officiel (BULL) n'est pas le fabricant

Ressources pour la compilation



4 *threads* pour un usage courant semble un bon compromis

Les options de compilation du fortran

Options de compilation	NEC	Intel
Pour des performances optimales	-C hopt -Wf,-pvctl,noassume -pi auto line=500	-g -O3 -xAVX -finline-limit=500
Debugging	-gv -EP	-g -O0
Debugging et Débordements de tableaux	-gv -EP -eC	-g -O0 -check bounds
Pré-initialisation à Not-a-Number	-Wf, -init,stack=nan, heap=nan -Wf,-K,a	-fp-stack-check -ftrapuv -fpe0 -fp-speculation=strict -check uninit
Optimisation sans trop de risque	-C vsafe	-g -O2

Comportement du compilateur fortran (*subjectif*)

Contexte	NEC	Intel
Durée de compilation	**Longue**	**Courte** (sauf exceptions)
in-lining	RAS	(-finline-limit=500) Parfois conflictuel avec la parallélisation open-mp
Vectorisation	Besoin occasionnel de la forcer par directives	Besoin occasionnel d'intervenir (Bug) par directives
Débordements de tableaux	**Laborieux si effectué sur l'intégralité du code**	**Pas de problèmes observés**
Pré-initialisation à Not-a-Number	**Redoutablement efficace**	**Fort peu exploitable** (Variables scalaires sur pile)
Fortran 2003	**Mal supporté**	**Bien supporté**
Erreurs compilateur	**A la compilation**	**A l'exécution**
Reproductibilité numérique	OK	**Spécifier absolument -fp-model precise**

Allocation des ressources

- **Sur NEC,**
  - **l'unité de calcul couramment utilisé est le *processeur :***
    Un noeud de calcul regroupe plusieurs processeurs
    avec beaucoup de mémoire : 128 Go sur SX8 ; 1 To sur SX
  - D'autre part, la parallélisation utilisée est exclusivement MPI
    => Définir le nombre de processeurs = Nombre de tâches MPI
    => Définir la quantité de mémoire allouée (sauf noeuds entiers)
- **Sur Bull,**
  - **l'unité de calcul à considérer est le *noeud :***
    Un noeud de calcul regroupe plusieurs coeurs
    avec peu de mémoire : 32 Go (+ qq noeuds à 128 Go ou 256 Go)
  - D'autre part, la parallélisation utilisée est mixte : MPI x Open-MP
    => Définir le nombre de noeuds alloués
    => Définir le nombre de tâches MPI par noeud
    => Définir le nombre de threads Open-MP par tâche MPI

MPI ou Open-MP : que choisir ?

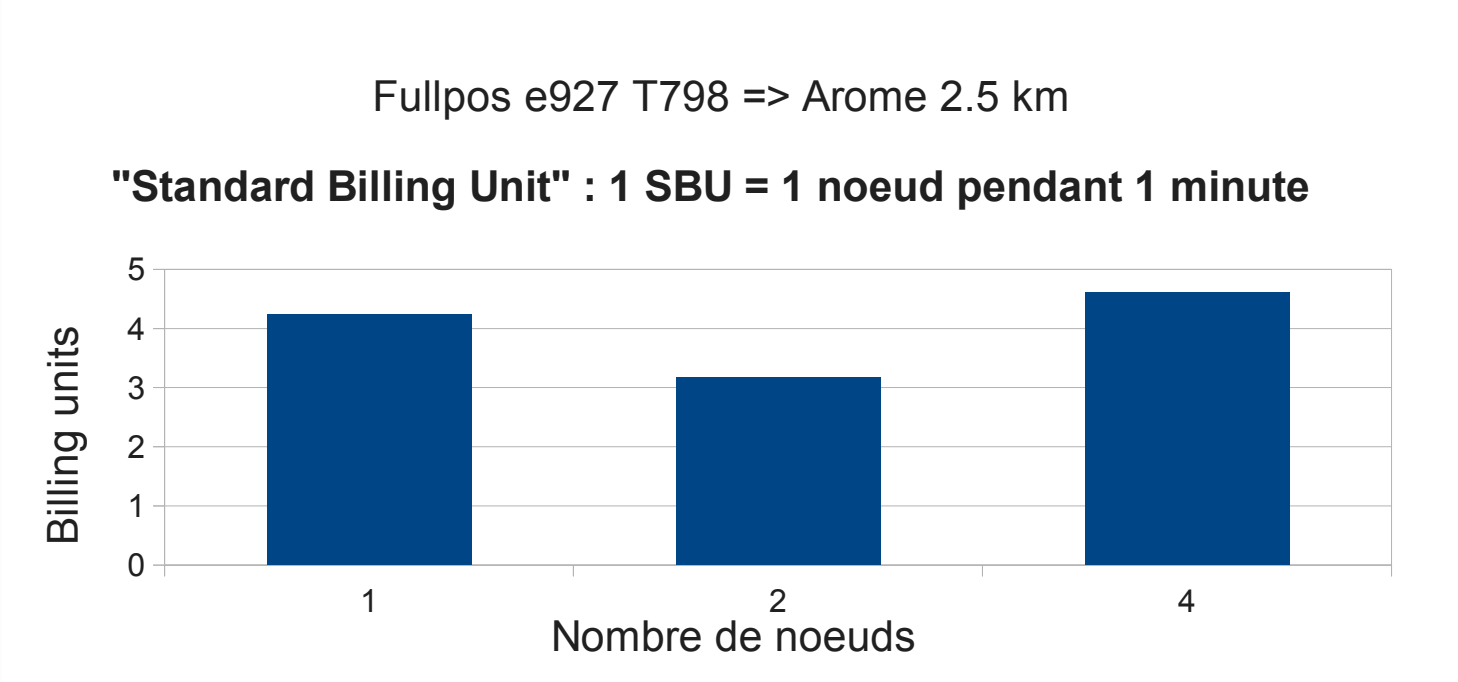
En général :

- **Privilégier les tâches MPI pour réduire le temps de calcul**
- Mais quand le nombre de tâches MPI est élevé :
  - Le job est moins efficace (problème de la scalabilité)
  - Le job coûte plus cher en mémoire

- **Privilégier les threads Open-MP pour réduire la mémoire**
- Mais quand le nombre de threads est élevé :
  - Le job peut devenir moins efficace qu'avec des tâches MPI
  - Le nombre de threads ne doit pas excéder le nombre de coeurs logiques par nœud (la mémoire doit être partagée)

Il faut trouver le ratio optimal « Nombre de threads par tâche MPI »
pour minimiser la facture du job :
Nombre de noeuds utilisés x temps réel d'utilisation
(« Billing units »)

Notion d'unité de coût = « Billing Unit »



Sur 1 noeud (avec 16 threads Open-MP)

- on peut contenir l'application

Mais avec 2 noeuds (16 tâches MPI x 2 threads Open-MP)

- on va plus vite et ça coûte moins cher !

Avec 4 noeuds (64 tâches MPI)

- on va encore plus vite mais la facture est plus élevée

Et l'*hyperthreading* dans tout ça ?

- C'est un moyen de **simuler 2 cœurs de calcul « logiques » avec un seul cœur de calcul physique** : un peu comme si on avait 2 fois plus de processeurs par nœud, mais moins puissants
- Existe aussi sur IBM au CEPMMT, sous le nom « Simultaneous Multithreading » (SMT)
- **Pour un très grand nombre de processeurs et certaines applications, cette technologie permet d'aller un peu plus vite** (environ 4 % de gain sur Arome 2.5 km).
  Mais on ne va sûrement pas 2 fois plus vite !
- **Attention à ne pas comptabiliser les coeurs logiques dans les ressources réelles !!**
- Performance plus complexe à obtenir car il faut **répartir le placement des cœurs logiques par rapport au matériel (cœurs physiques, mémoire)** qui reste bien réel !

Transparent pour les utilisateurs d'Arpege/Aladin/Arome car, comme pour IFS, le placement sera fait dans le code (*Merci Philippe !!*)

Peuplement et Dépeuplement

- *Tant qu'à allouer un noeud de calcul,*
  autant utiliser tous les coeurs qui y sont, n'est-ce pas ?

  (Nombre de tâches MPI) x (Nombre de threads Open-MP) =
  Nombre total de coeurs disponibles

- *Mais pour utiliser l'hyperthreading, il faudra le spécifier :*
  - Soit en doublant le nombre de tâches MPI dans mpirun/srun
  - Soit en doublant la variable OMP_NUM_THREADS
- *Dépeupler un noeud, c'est ne pas utiliser tous les coeurs qui y sont :*
  - Intéressant pour les applications sensibles à la « bande passante » mémoire (càd : les *accès* à la mémoire, pas la quantité de mémoire)
  - => Faites-le en connaissance de cause uniquement !

Allocation de la mémoire

- **Sur NEC,**
  - La mémoire allouée est définie directement dans les cartes de soumission
  - Il y a beaucoup de mémoire par noeud, donc on a été peu regardant
  - En cas de mémoire insuffisante, plantage avec messages comme :
    - Batch job received signal **SIGXMEM**. (Exceeded memory size limit)
    - cannot **allocate**
- **Sur BULL,**
  - La mémoire est allouée via le nombre de noeuds alloués de façon *exclusive* (sauf classes particulières)
  - En case de mémoire insuffisante, plantage avec messages comme :
    - forrtl: severe (41): insufficient virtual **memory**
    - reg_mr Cannot allocate **memory**
    - exceeded 30720000 KB **memory** limit
    - Exceeded job **memory** limit
  - **=> augmenter le nombre de noeuds (=> +32 Go par noeud supplémentaire)**
  - **=> mieux ? réduire le nombre de tâches MPI au profit du nombre de threads Open-MP (à nombre de threads x nombre de tâches constant)**

Réservation de la mémoire pour la pile (*stack*)

- **Sur NEC,**
  - L'espace mémoire réservée à la pile est définie à l'édition de lien :
    - Exemple pour 32 Go : -Wl,-Z,32G
  - En cas d'insuffisance, plantage avec un message comme :
    - **SIGNMEM: No memory**
- **Sur Bull,**
  - L'espace mémoire réservée à la pile est définie dans l'environnement :
    - Exemple pour 128 Mo :

      export KMP_STACKSIZE=128M

      export KMP_MONITOR_STACKSIZE=128M

      Sans oublier : % ulimit -s unlimited
  - En cas d'insuffisance, plantage avec un **message ambigü** :
    - « forrtl: severe (174): SIGSEGV, *segmentation fault* occurred »
    - « APPLICATION TERMINATED WITH THE EXIT STRING: *Segmentation fault* (signal 11) »

      => Signal que des tableaux automatiques sont trop grands (exemple : NPROMA trop grand) ou trop nombreux

Modifications de namelists (1) : NPROMA etc

- **Sur NEC,**
  - On utilise de grandes valeurs pour NPROMA afin de remplir les 256 registres vecteurs et de s'étaler sur les 4096 banks mémoire :
    NPROMA=NFPROMA=NFPROMA_DEP=-3582
  - On utilise des branches de code dédiées à l'architecture vectorielle : **LOPT_SCALAR=.FALSE.**
  - Pour Arpege, une répartition optimale des points de grilles entre les tâches MPI n'a pas d'impact sur la performance : **LEQ_REGIONS=.FALSE.**
- **Sur BULL,**
  - On utilise de petites valeurs pour NPROMA afin de remplir la mémoire-cache sans la saturer :
    NPROMA=NFPROMA=NFPROMA_DEP=-50
  - On utilise des branches de code dédiées à l'architecture scalaire : **LOPT_SCALAR=.TRUE.**
  - Pour Arpege, une répartition optimale des points de grilles entre les tâches MPI est nécessaire pour une meilleure performance : **LEQ_REGIONS=.TRUE.**

Modifications de namelists (2) : MPI

- **Sur NEC,**
  - La distribution MPI est principalement la distribution appelée historiquement « de niveau A » :
    - Découpage Nord-Sud dans l'espace point de grille :
      NPRGPNS=NPROC ; NPRGPEW=1
    - Découpage sur les ondes *m* dans l'espace spectral :
      NPRTRW=NPROC ; NPRTRV=1
- **Sur BULL,**
  - Le nombre élevé de processeurs nécessaire impose l'utilisation de la distribution MPI appelée historiquement « de niveau B » :
    - Découpage Nord-Sud et Est-Ouest dans l'espace point de grille
      NPRGPNS x NPRGPEW = NPROC
    - Découpage sur les ondes *m*
      et sur les niveaux verticaux (inhomogène !!) dans l'espace spectral :
      NPRTRW x NPRTRV = NPROC

Le plus simple est de supprimer ces 4 paramètres de la namelist et de laisser le programme les déterminer au mieux

Modifications de namelists (3) : MPI

- **Sur NEC,**
  - La boîte aux lettres pour les communications MPI est auto-gérée
- **Sur BULL,**
  - La taille de la boîte aux lettres pour les communications est paramétrée par **namelist** pour le modèle, et par **variable d'environnement** pour ODB
  - Exemple pour 1 Go :
    - MBX_SIZE=1024000000
    - export MPL_MBX_SIZE=1024000000
  - Elle ne peut pas excéder 2 Go (entier codé sur 32 bits)
  - **En cas de dépassement** : abort avec des messages comme :

    Fatal error in MPI_Bsend: Invalid buffer pointer, error stack:
    MPI_Bsend(195).....: **MPI_**Bsend(buf=0x7fff486817e0, count=8352, dtype=0x4c000829, dest=15, tag=11000, comm=0x84000000) failed
    MPIR_Bsend_isend(305): **Insufficient space** in Bsend buffer; requested 66816; total buffer size is 1024000

Modifications de namelists (4) : I/O

- **Sur NEC,**
  - La lecture, le décodage et l'encodage des données sont effectués sur très peu de tâches MPI :
    - NSTRIN << NPROC
    - NSTROUT << NPROC
- **Sur BULL,**
  - La lecture, le décodage et l'encodage des données sont effectués sur sur toutes les tâches *a priori* :
    - NSTRIN=NPROC
    - NSTROUT=NPROC
  - On envisage ultérieurement l'utilisation d'un « serveur d'I/O » en complément : noeud(s) dédié(s) aux I/Os

Variables d'environnement

- **Sur NEC,**
  - Nous n'utilisons pas (ou peu) Open-MP
  - Le debogueur/profileur DrHook n'est guère utilisable :
    export DR_HOOK=0
- **Sur BULL,**
  - Le nombre de threads Open-MP par tâche MPI doit être confirmé par la variable **OMP_NUM_THREADS**
  - Le profileur DrHook est utilisable :
    - **export DR_HOOK=1** ; **export DR_HOOK_OPT=prof**
  - Mais pas le debogueur DrHook à cause du compilateur (Intel) :
    - **export DR_HOOK_IGNORE_SIGNALS=-1**

- Sur toutes les machines : pour faire fonctionner l'abort collectif de MPI :
  export EC_MPI_ATEXIT=0 (sauf pour ODB)

Environnement ODB

- **Sur NEC,** nous utilisons la variable d'environnement

  ODB_IO_METHOD=1

  - Spécifie qu'un « pool » d'observations est formé d'un seul fichier par table
  - Convient bien dans les cas où le nombre de tâches MPI est réduit
  - La fabrications des « pools » d'observations est économe en mémoire
- **Sur BULL,** nous allons devoir utiliser la variable d'environnement

  ODB_IO_METHOD=4

  - Spécifie qu'un « pool » d'observations est formé d'une série de fichiers de taille fixe concaténés
  - Nécessaire quand le nombre de tâche MPI est élevé, pour éviter de devoir manipuler une multitude de trop petits fichiers
  - La fabrications des « pools » d'observations est coûteuse en mémoire
    - => problèmes attendus à faire fonctionner Bator sur un grand nombre de « pools »
    - => solution suggérée (et utilisée dans le benchmark) : utiliser Odbtools comme « shuffle » sur les bases ECMA (possible depuis le cycle 37).

Profilage élémentaire avec DrHook (1)

- Variables d'environnement :
  - export DR_HOOK=1
  - export DR_HOOK_IGNORE_SIGNALS=-1
  - export DR_HOOK_OPT=prof
- Génération d'1 fichier par tâche MPI :
  - drhook.prof.1 drhook.prof.2 ... drhook.prof.$NPROC
- Pour obtenir un profilage moyen :
  - Utiliser **drhook_merge_walltime_max** (sous ~khatib/public/bin)
  - Usage : cat drhook.prof.* | perl -w drhook_merge_walltime_max
  - **Tri des routines par temps maximal**
- Les routines parallélisées par Open-MP ne sont visibles que dans les fichiers drhook.prof.* :
  - identifiées par un prefix « * » et un suffixe « @*n* » où n est le numéro du thread Open-MP
  - => Si « @1 » seulement, la routine n'est pas parallélisée !

Profilage élémentaire avec DrHook (2)

Name of the executable : ./MASTERODB
Number of MPI-tasks : 144
Number of OpenMP-threads : 1
Wall-times over all MPI-tasks (secs) : Min=984.020, Max=984.630, Avg=984.397, StDev=0.128
Routines whose total time (i.e. sum) > 1.000 secs will be included in the listing

Avg-%	Avg.time	Min.time	Max.time	St.dev	Imbal-%	# of calls : Name of the routine
13.53%	133.210	120.727	138.384	3.554	12.76%	2106164 : SGEMMX
3.09%	30.394	4.146	83.588	16.231	95.04%	21024 : SLCOMM
5.44%	53.517	44.497	71.569	5.081	37.83%	27982944 : LAITRI
2.81%	27.707	3.158	68.292	18.717	95.38%	21024 : SLCOMM2A
2.08%	20.481	4.215	42.535	8.826	90.09%	23472 : TRGTOL
2.25%	22.109	11.475	39.906	5.073	71.24%	20880 : TRLTOM
2.52%	24.804	11.069	37.973	6.401	70.85%	21600 : TRLTOG
3.42%	33.665	33.014	34.616	0.281	4.63%	2543904 : ADVPRCS
3.17%	31.201	29.273	34.151	1.494	14.28%	10175616 : LARCHE
3.10%	30.558	29.659	31.249	0.346	5.09%	2543904 : ACCVIMP
2.12%	20.880	15.767	30.930	2.696	49.02%	21600 : TRMTOL
0.69%	6.779	0.010	28.470	6.348	99.96%	1614379 : CONVECT_CLOSURE_SHAL
1.99%	19.629	15.373	26.772	2.459	42.58%	2543904 : ACBL89
2.53%	24.888	23.845	26.667	0.593	10.58%	2543904 : ACTURB

Le déséquilibrage (Imbal-%) est source de perte de « scalabilité »

Profilage élémentaire avec DrHook (3)

Profiling information for program='./MASTERODB', proc#7:

No. of instrumented routines called : 891
Instrumentation started : 20130219 171235
Instrumentation ended : 20130219 171238
Instrumentation overhead: 0.29%
Memory usage : 996 MBytes (heap), 282 MBytes (rss), 0 MBytes (stack), 0 (paging)
Wall-time is 3.21 sec on proc#7 (16 procs, 2 threads)
Thread#1: 3.20 sec (99.72%)
Thread#2: 0.67 sec (20.82%)

#	% Time (self)	Cumul (sec)	Self (sec)	Total (sec)	# of calls (Size; Size/sec; Size/call; MinSize; MaxSize)	Self ms/call	Total ms/call	Routine@<thread-id>
1	13.16	0.423	0.423	0.423	38	11.12	11.12	SLCOMM@1
2	11.21	0.783	0.360	0.360	1	360.12	360.12	FAIXLA_MT@1
3	5.53	0.960	0.177	0.178	1	177.49	177.51	SETUP_TRANS@1
4	4.71	1.111	0.151	0.151	4	37.81	37.81	FAIFLA_MT@1
5	4.15	1.245	0.133	0.134	15	8.89	8.92	DIWRGRID_MOD:DIWRGRID_RECVX@1
6	3.54	1.359	0.114	0.114	59	1.93	1.93	TRGTOL@1
7	3.35	1.466	0.107	0.107	38	2.83	2.83	SLCOMM2A@1
8	3.26	1.571	0.105	0.105	50	2.09	2.09	TRLTOG@1
9	2.63	1.655	0.084	0.084	5	16.89	16.89	GRIB_API:IGRIB_NEW_FROM_TEMPLATE@1
10	2.08	1.722	0.067	0.067	41	1.63	1.63	TRLTOM@1
11	1.60	1.773	0.051	0.051	93	0.55	0.55	DRESDDH@1
12	1.56	1.823	0.050	0.050	1	50.11	50.11	SLRSET@1
13	1.45	1.870	0.047	0.047	50	0.93	0.93	TRMTOL@1
14	1.41	1.915	0.045	0.045	1	45.17	45.17	KPP_WSCALE_MOD:KPP_WSCALE@1
15	1.20	1.953	0.038	0.409	148	0.26	2.77	*APLPAR@1
16	1.11	1.989	0.036	0.058	148	0.24	0.39	*ACCVIMP@1
17	1.11	2.025	0.036	0.036	1650	0.02	0.02	*LAITRI@1
18	1.07	2.025	0.034	0.364	118	0.29	3.09	APLPAR@2
19	1.03	2.058	0.033	0.035	36	0.92	0.97	TRMTOS@1
20	1.02	2.058	0.033	0.033	1276	0.03	0.03	LAITRI@2

Ceci n'est pas la dernière diapositive de ma présentation
mais le dernier « slide » de mon « talk »

Ne dites pas « communiqué de presse »,
dites « press release »
« Checkez » plutôt que de vérifier ce que vous faites
« Shiftez » vos réunions au lieu de les décaler
« Killez » vos jobs, et ne tuez pas vos travaux !
N 'étudiez pas la tarification, faites le « pricing »
Faites des « updates » de vos « softs »,
pas des mises à jour de vos logiciels
Un « call » avec vos partenaires vaut mieux qu'une conférence téléphonique
« Shipez » des « samples » de votre marchandise au lieu d'en envoyer des exemplaires par bateau

... et si tout va bien, c'est « Blue sky » !