Programmation des architectures multicoeurs

P. Fortin

Programmation parallèle avancée (PPA)

Master 2 Informatique SAR-STL, UPMC

Plan

- Programmation des architectures multicoeurs homogènes
  - Threads POSIX
  - Cilk
  - OpenMP 3.0
  - Et aussi : Intel Threading Building Blocks, ...
- Extension vers les architectures à mémoire distribuée
  - PGAS
- Parallélisation en mode hybride multiprocessus-multithread

Bibliographie et remerciements

- U.C. Berkeley CS267 (http://www.cs.berkeley.edu/~demmel/cs267_Spr09)
- *Tasking in OpenMP*, A. Duran, 5th International Workshop on OpenMP, 2009
- *Memory Consistency Models and Advanced OpenMP*, V. Sarkar, Rice University (COMP 422, Lecture 8)
- *Using OpenMP*, B. Chapman, G. Jost, R. van der Pas, MIT Press
- Norme OpenMP 3.0

Création des threads : les différentes possibilités (d'après CS267)

- Modèle *cobegin/coend* :

```
cobegin
      job1(a1);
      job2(a2);
coend
```

- Instructions/sections internes exécutables en parallèle
- Plusieurs niveaux d'imbrications possibles
- coend obligatoire

- Modèle *fork/join* :

```
tid1 = fork(job1, a1);
job2(a2);
join tid1;
```

- La procédure issue du fork s'exécute en parallèle
- Attente nécessaire au niveau du join si la procédure issue du fork n'est pas terminée.

- Modèle *future* :

```
v = future(job1(a1));
… = …v…;
```

- Fonction passée à future est/sera évaluée en parallèle
- L'utilisation du résultat devra attendre

→ *cobegin* plus "propre" que *fork*, mais moins général

→ *future* nécessite un compilateur (et un matériel) adapté

Threads POSIX (PThreads)

- Norme POSIX (***P**ortable **O**perating **S**ystem **I**nterface for UNI**X**)*
- Threads POSIX = interface portable (UNIX) pour :
  - Créer des threads
  - Synchroniser des threads entre eux
  - Contexte : mémoire partagée (au sein d'un même processus)
- Thread noyau/utilisateur :
  - *Contention scope* : PTHREAD_SCOPE_SYSTEM / PTHREAD_SCOPE_PROCESS
- Programmation « bas-niveau »
  - Parallélisme explicite
  - Déboguage potentiellement hardu
  - La "base" en programmation multi-thread (GOMP (OpenMP pour gcc), processeur Cell, systèmes embarqués ...)

Threads POSIX (suite)

- Modèle *fork/join* :

```
pthread_create(&thread_id, &attribut, &fonction, &fonction_arg);
pthread_join(thread_id, &valeur_retour);
```

- Gestion des conflits
  - En théorie : toute la mémoire est accessible à tous les threads !
  - En pratique :
    - Variables globales → partagées
    - Variables locales aux fonctions (dans la pile) → « privées »
    - Variables dans le tas et allouées avant création des threads → connues de tous si pointeur connu
    - Variables dans le tas et allouées après création des threads → connues uniquement du thread créateur
  - Interface threads POSIX :
    - Barrières, verrous (mutex), variables de conditions...

Threads POSIX (suite et fin)

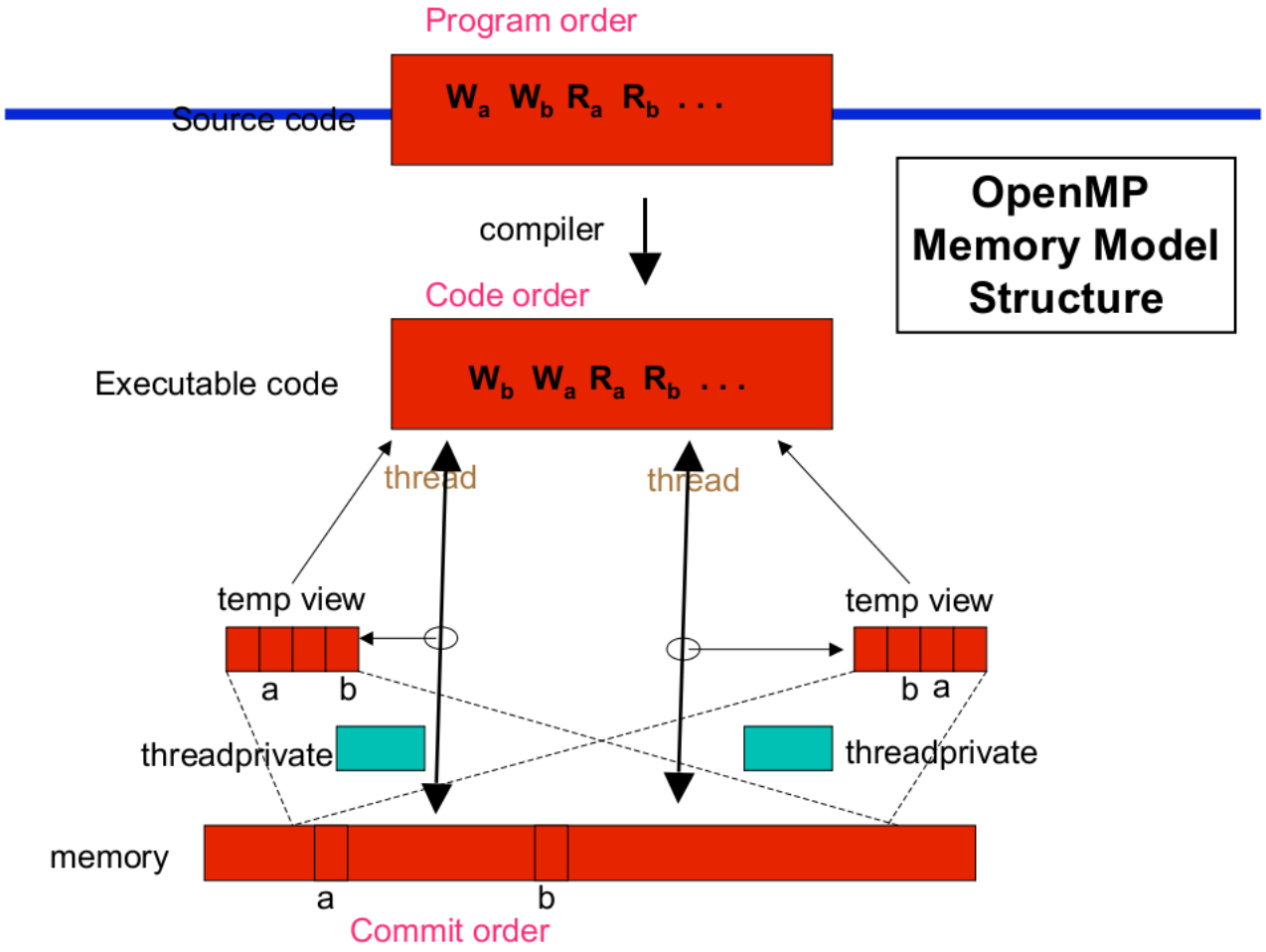
- **Cohérence séquentielle (sequential consistency)** : "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]
- Pour des raisons de performance : généralement **non respectée** sur les architectures multicoeur / multiprocesseur ***cache-coherent → relaxed consistency***
  - Exemple :

Au départ : data=0, flag=0	
Thread 1	Thread 2
data=1	while(flag == 0){}
flag=1	… = data

  - Variables maintenues dans les registres (→ recours à *volatile*)
  - Réordonnancement des instructions par le compilateur / le processeur
  - Réordonnancement des écritures par la hiérarchie mémoire / par le bus
- Utilisation **impérative** des fonctions de l'interface PThreads pour l'accès à toute variable partagée (→ *volatile* seul est insuffisant)

Modèle de cohérence mémoire d'OpenMP

- *Shared memory with relaxed consistency model*



(*Memory Consistency Models and Advanced OpenMP*, V. Sarkar)

Directive OpenMP *flush*

- Utilisation :

  #pragma omp flush [(liste de variables)]

- Si aucune variable n'est mentionnée, toutes les variables partagées et accessibles sont synchronisées avec la mémoire principale.
- *flush* synchronise la *temporary view* du thread avec la mémoire principale
  - ➤ Permet la gestion des architectures non cache-coherent par OpenMP
  - ➤ Empêche le réordonnancement des instructions (concernant les variables listées) « à travers le *flush* »
- Peut permettre de synchroniser des threads, **dans certains cas uniquement → dangereux !**

Directive OpenMP *flush* (2)

- *flush* implicites (placés automatiquement par OpenMP) :
  - au niveau d'une *barrier*,
  - à l'entrée et à la sortie des directives *critical, ordered, parallel, parallel for, parallel sections*
  - à la sortie de *for, sections, single*
- Pas de *flush* implicite si :
  - la clause *nowait* est présente,
  - à l'entrée et à la sortie de *master*
  - à l'entrée de *for, sections, single*
- Hors de ces points de synchronisation implicites, *flush* peut être utilisé pour rendre certaines modifications de variables ``visibles'' entre les threads (mais reste « à éviter »...)

Exemple d'utilisation de *flush*

- Exemple A.21.1c (OpenMP 3.0 spec)

```
int sync[NB_THREAD];
float work[NB_THREAD];

#pragma omp parallel \
   private(moi, voisin) \
   shared(work,sync)
{
  moi=omp_get_thread_num();
  sync[moi]=0;
  #pragma omp barrier

  # 1er calcul f() :
   work[moi] = f(...);
```

```
  #pragma omp flush(work,synch)
  synch[moi] = 1;
  #pragma omp flush(synch)

  voisin=(moi > 0 ? moi -1 :
    omp_get_num_thread()-1);
  while (synch[voisin] == 0) {
     #pragma omp flush(synch)
  }
  #pragma omp flush(work,synch)

  /* 2ieme calcul g() (en utilisant la
valeur du voisin) :  */
     … = g(work[voisin]);
```

Partage de variables et performance

- True sharing :
  - Ok pour accès en lecture seule ou pour écritures rares
  - Ecritures fréquentes → goulot d'étranglement
  - Solution possible : chaque processeur travaille sur une copie locale (puis éventuellement réduction) → si l'algorithme le permet
- False sharing :
  - Dû au fait que les caches travaillent par bloc (*ligne* de cache)
  - Exemple : tableau d'entiers, chaque entier écrit fréquemment par 1 processeur différent → plusieurs entiers dans une ligne de cache
  - Solution possible : allouer les données de chaque processeur de façon contiguë / éviter l'entrelacement des données en mémoire

Cilk

- Modèle *future*
- Cilk-5.3 → vise multithreading sur SMP (http://supertech.csail.mit.edu/cilk/)
- Exemple : Fibonacci récursif

```
cilk int fib (int n){
        if (n < 2) return n;
        else { int x, y;
           x = spawn fib (n-1);
           y = spawn fib (n-2);
           sync;
           return (x+y); }}
```

- Exécution parallèle : vol de tâche avec prise en compte de la localité des données (*cache oblivious*)

OpenMP 3.0

- Version 3.0 : mai 2008
- gcc-4.4, Intel C/C++ compiler (icc) …
- Nouveautés :
  - Loop collapsing : regroupement des itérations de plusieurs (« petites ») boucles imbriquées
  - Garanties sur « *static schedule* » de 2 boucles consécutives
  - Nouvelle clause *schedule* : **auto**
  - ...
  - Et surtout : nouvelle directive « ***task*** »
    → modèle *future*

    → parallélisation d'une plus grande gamme d'applications : boucles non bornées, algorithmes récursifs, schéma producteur/consommateur...
    (Modèle *cobegin* d'OpenMP 2.5 surtout adapté aux nids de boucles)
- team = ensemble des threads exécutant une région parallèle OpenMP
  #pragma omp parallel
  (thread 0 = *master* = thread original)

OpenMP tasks

(d'après A. Duran, *Tasking in OpenMP*)

- Pourquoi des tâches ?
  - ➤ Exemple : parcours de liste en parallèle avec OpenMP 2.5

```
void parcours_liste(liste l){
    element e = l → first;
    #pragma omp parallel firstprivate (e)
    while (e != NULL){
        #pragma omp single nowait
        traiter (e) ;
        e = e → next;
    }
}
```

- Peu naturel
- Mauvaises performances
- Pas composable

OpenMP tasks

- Pourquoi des tâches ? (suite)
  - ➤ Autre exemple : parcours d'arbre en parallèle avec OpenMP 2.5

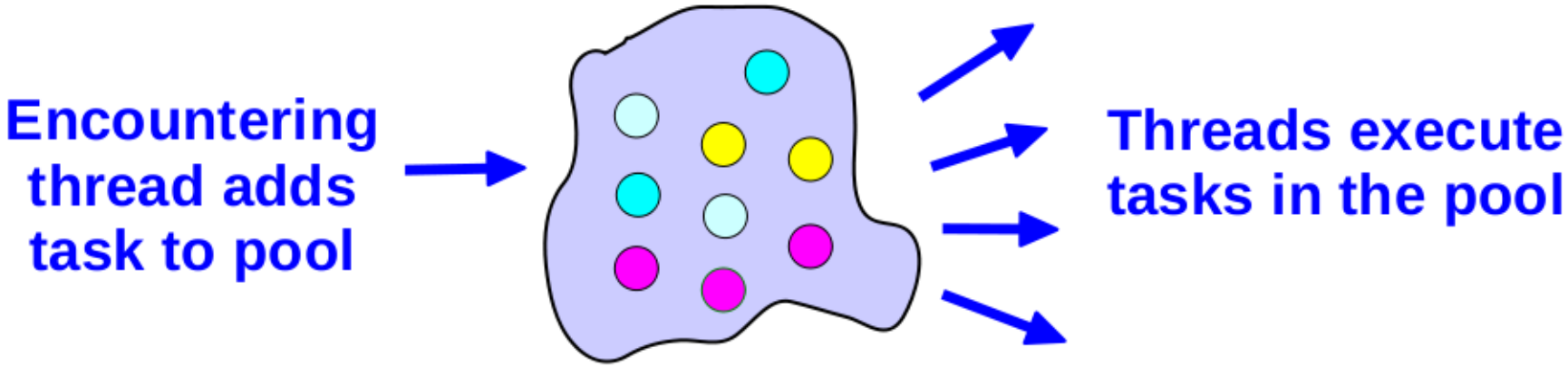
```
void parcours_arbre (arbre  *a){
#pragma omp parallel sections
  {
    #pragma omp section
    if (a → gauche) parcours_arbre ( a → gauche) ;
    #pragma omp section
    if ( a → droit) parcours_arbre ( a → droit) ;
  }
   traiter (a) ;
}
```

- Trop de régions parallèles : surcoûts, synchronisations supplémentaires, pas toujours bien supporté par l'implémentation ...

OpenMP tasks

- Directive ***task*** :
  #pragma omp task [clauses]
  structured block
- Task = unité de travail = code à exécuter + données (environnement) + ICV (internal control variables)
- Chaque thread qui « rencontre » la directive ***task*** crée une tâche/task → **exécution immédiate ou différée** par 1 des threads de la *team*
- Hautement composable : imbrication possible dans
  - des régions parallèles
  - d'autres tâches
  - des directives « omp for », « omp sections », « omp single »

« OpenMP tasks » en image



Developer specifies tasks in application
Run-time system executes tasks

(An Overview of OpenMP 3.0, R. van der Pas, IWOMP2009)

OpenMP tasks : portée des variables

- Variables partagées/privées :
  - Variables globales partagées par défaut
  - Attribut *shared* conservé
  - Le plus utile avec les tâches : ***firstprivate***
    - Attention aux pointeurs avec *firstprivate* : seul le pointeur est concerné, pas la zone pointée
- Attention aux variables (non *firstprivate*) dans la pile :
  - Le tâche mère ne doit pas les rendre invalides avant que les tâches filles ne s'en servent...
  - Solutions possibles :
    - Utiliser *firstprivate*
    - Déplacer les variables dans le tas (mais quand les « désallouer » ?)
    - Introduire des synchronisations

OpenMP tasks : synchronisation

- Barrières
  - ➤ Implicites : à la fin d'une région parallèle
  - ➤ Explicites : **#pragma omp barrier**
- Garantie : toutes les tâches créées par un thread de la team courante sont terminées à la sortie de la barrière
- De plus : barrière de tâche

  #pragma omp taskwait

  → La tâche courante attend alors la complétion de toutes ses tâches *filles* (seulement « filles directes », pas descendants).

Parcours de liste en OpenMP 3.0

```
void parcours_liste(liste l){
   element e = l → first;
   while (e != NULL){
      #pragma omp task firstprivate(e)
      traiter (e) ;
      e = e → next;
  }
#pragma omp taskwait // garantie ici la terminaison du parcours
}
```

```
liste l;
#pragma omp parallel
  parcours_liste(l);
```

```
liste l;
#pragma omp parallel
#pragma omp single
  parcours_liste(l);
```

```
liste l[N];
#pragma omp parallel
#pragma omp for
for (i=0;i<N;i++)
  parcours_liste(l[i]);
```

Optimisation : clause *untied*

- Par défaut : les tâches sont ***tied*** (liées)
  - Une tâche liée est toujours exécutée par le même thread
- Conséquence : performance/ordonnancement parfois non optimale
  - Exemple : parcours récursifs → déséquilibrage de charge
- Clause ***untied*** → la tâche n'est plus liée à 1 thread donné, mais alors
  - Éviter les variables *threadprivate*
  - Éviter l'utilisation de l'indice du thread dans les calculs de la tâche
  - Faire très attention aux verrous et sections critiques → interblocages possibles...

Optimisation : grain des tâches

- Facteur clé pour la performance :
  - ➤ Regrouper des tâches « fines » :
    - Clause ***if***
      #pragma omp task if (prof < PROF_MAX)
    - Instruction ***if*** :

```
void f(..., int prof){
…
#pragma omp task
 {
    ...
    if (prof < PROF_MAX)
                f(..., prof+1)
    else
                f_sequentielle(...)
 }
}
```

Modèles de programmation parallèle PGAS

- PGAS : Partitioned Global Address Space
  - *Global Address Space*: chaque thread peut accéder directement aux données distantes (lecture et écriture)
    → masque la distinction mémoire partagée/distribuée
    → programmation plus aisée
  - *Partitioned*: les données sont explicitement désignées comme locales ou globales/distantes
    → distribution/localité des données cruciale pour performance et passage à l'échelle
- Programmation SPMD (comme programmes MPI)
- Exemples de langages PGAS :
  - Co-Array Fortran (inclus dans la norme Fortran 2008)
  - Unified Parallel C (UPC)
  - Titanium, Fortress, Chapel, X10...

Caractéristiques des langages PGAS (d'après CS267 K. Yelick)

- Structures de données distribuées :
  - tableaux distribués, pointeurs/références locaux et globaux
- *One-sided* "communication" de type "mémoire partagée" :
  - Simples affectations : **x[i] = y[i];** ou **t = *p;**
  - Opérations non élémentaires (copie mémoire, copie de tableaux)
  - Optionnel : appel à distance de fonctions (remote function invocation)
- Maîtrise de la distribution des données :
  - PGAS différent de "*(cache-coherent) distributed shared memory*"
  - Les données distantes restent distantes.
- Synchronisation
  - Barrières globales, verrous, barrières mémoires
- Communications collectives, opérations (collectives) d'Entrées/Sorties, etc.

Exemple de langage PGAS : Co-Array Fortran

- Extensions au Fortran 95, inclus dans Fortran 2008
- Chaque processus = *image*
- Les *co-arrays* → notation [ ]
  REAL, DIMENSION(N)[*] :: X,Y

  X(:) = Y(:)[Q]
  → copie du tableau Y de l'image Q dans la copie locale du tableau X (*get*)

  Y(:)[Q] = X(:) ! *put*
- Fonctions intrinsèques :
  - NUM_IMAGES(), THIS_IMAGE(), SYNC_ALL(), START_CRITICAL , END_CRITICAL
- Support explicite des entrées/sorties parallèles

Parallélisation en mode hybride multiprocessus-multithread (MPI-thread)

- Avantages **possibles (suivant l'application)** par rapport au mode multiprocessus (« MPI pur ») :
  - Réduction du volume ou du nombre de communication(s)
  - Réduction de la consommation mémoire / noeud
    → *memory scalability*
  - Possibilité de mettre en place un équilibrage de charge **dynamique** entre les threads d'un même noeud, alors que l'équilibrage de charge est **statique** entre les noeuds
- Exemples : convolution en imagerie, diffusion de chaleur sur une grille, méthodes hiérarchiques pour le problème à N-corps...
- Remarque : grappe de noeuds SMP pour MPI pur
  → communications MPI intra-noeud par segments de mémoire partagée

Autre exemple : méthode du gradient conjugué

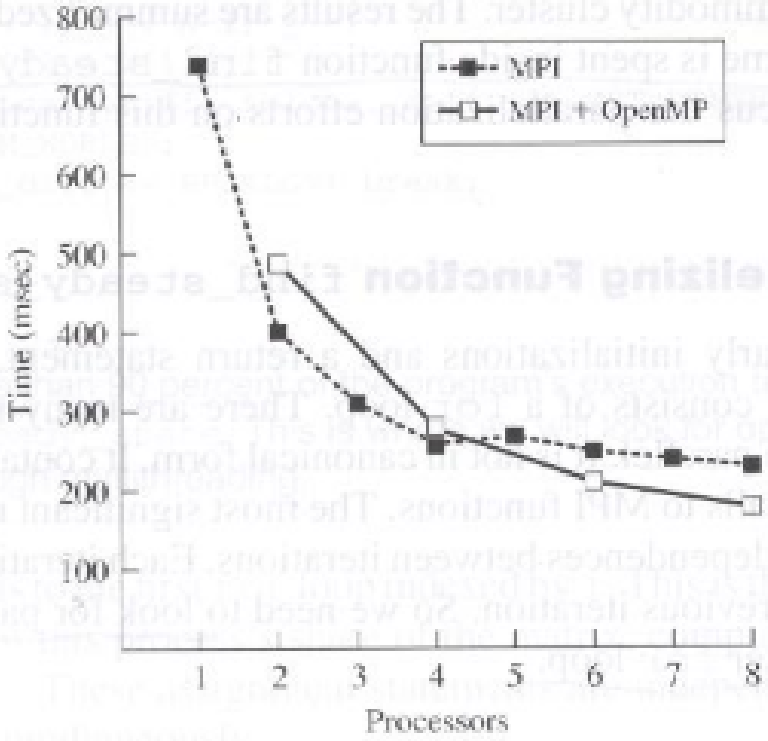
- Résolution itérative de **Ax=b**, avec **A** définie positive d'ordre n, en (au plus) n étapes
  → phase de calcul la plus coûteuse : produits matrice-vecteur
- Algorithme parallèle multiprocessus :
  - A chaque itération : 2 produits matrice-vecteur avec A
    - **A** décomposée en blocs de ligne sur les processus
    - Vecteur **b** (et autres vecteurs temporaires) répliqués sur tous les processus
  - Communications requises : après chacun des produits matrice-vecteur, réplication (vers tous les autres processus) des résultats partiels dans chaque processus → MPI_Allgather
- Mode hybride multithread-multiprocessus : parallélisation des produits matrice-vecteur avec OpenMP

Méthode du gradient conjugué avec une programmation hybride MPI-OpenMP

- Résultats : système dense de 768 équations sur grappe de 4 noeuds bi-processeur

  - MPI pur :
    jusqu'à 4 processus
    → 1 processus/noeud

  - MPI-OpenMP :
    1 processus x 2 threads



(d'après *Parallel Programming in C with MPI and OpenMP*, M. J. Quinn)