

DE LA RECHERCHE À L'INDUSTRIE



# PROGRAMMATION HYBRIDE & MULTI-CŒURS

## PROGRAMMATION HYBRIDE

### MPI+OPENMP

AMS-I03 2016/2017

Marc Tajchman (CEA-Saclay - DEN)

[marc.tajchman@cea.fr](mailto:marc.tajchman@cea.fr)

[www.cea.fr](http://www.cea.fr)

## Introduction

- Objectifs
- Qu'est ce que l'hybridation et comment peut-elle aider ?
- Quand ?
- Hybridation MPI+OpenMP

## Notions de base de l'hybridation

- Scénario classique
- Les différentes possibilités
- OpenMP dans MPI
- MPI dans OpenMP
- Conception d'applications hybrides « from scratch »
- Exemples
- Conseil généraux

## Exemples

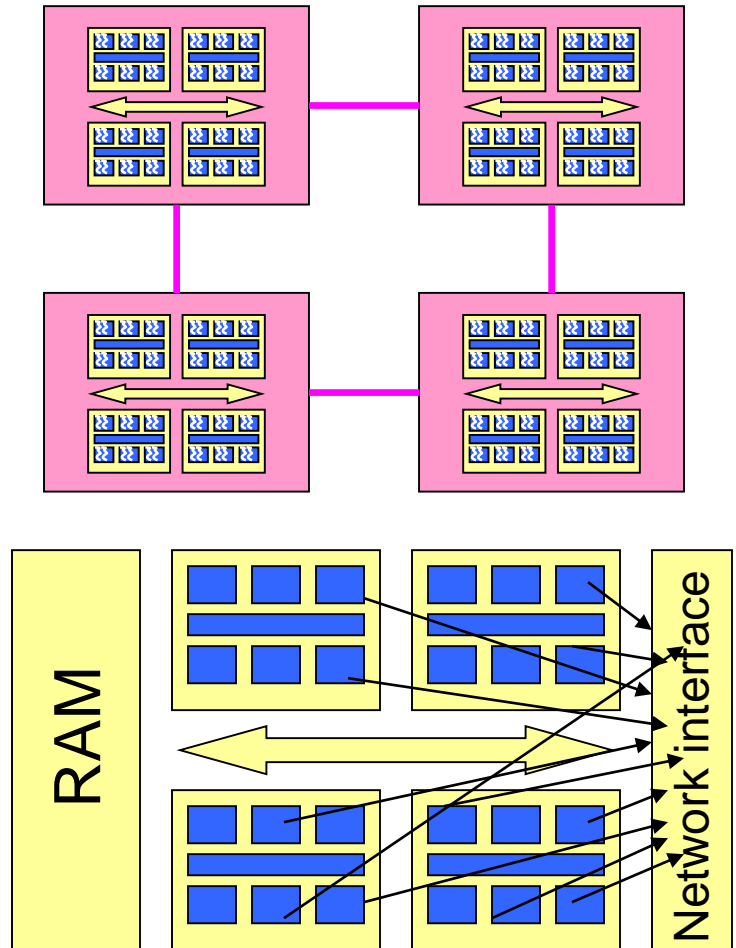
# ***INTRODUCTION***

- Comprendre la **différence** entre les modèles à **passage de message** et à **mémoire partagée**
- Apprendre les modèles de base pour **utiliser à la fois** les approches **passage de messages** et **mémoire partagée**
- Apprendre comment programmer un programme hybride;
- Questions de fond quant à **l'hybridation** des codes séquentiels existant, ou **tout-MPI**, ou **tout OpenMP**;

- C'est l'utilisation de modèles de programmation intrinsèquement différents d'une manière complémentaire, afin d'atteindre certains gains de performances pas possible autrement;
- C'est une autre façon d'utiliser différents modèles de parallélisation d'une façon qui tire parti des avantages de chacun;
- **En introduisant MPI dans les applications OpenMP** : peut aider à étendre sur plusieurs nœuds SMP;
- **En introduisant OpenMP dans les applications MPI** : peut aider à utiliser plus efficacement la mémoire partagée sur des noeuds SMP atténuant ainsi la nécessité d'une communication explicite intra-nœud;
- **En introduisant MPI et OpenMP** lors de la conception / codage d'une nouvelle application : peut aider à maximiser l'efficacité, la performance et la scalabilité;

# QUAND EST-CE QUE L'HYBRIDATION A DU SENS?

- Quand on veut « scaler » une application mémoire partagée OpenMP pour une utilisation sur plusieurs nœuds dans un cluster de SMP;
- Quand on veut réduire la sensibilité d'une application MPI à devenir « bornée » par les communications;
- Quand on conçoit un programme parallèle, dès le début;



Facilite la programmation coopérative de nœuds à mémoire partagée :

- MPI facilite la communication entre les nœuds SMP;
- OpenMP gère la charge de travail sur chacun des nœuds SMP;
- MPI et OpenMP sont utilisées en tandem pour gérer l'accès concurrentiel de l'application;

- MPI

- Standard de fait pour les communications à mémoire distribuée;
- Nécessite que la synchronisation de l'état du programme doit être traitée explicitement en raison de la nature de la mémoire distribuée;
- Les données vont au processus;

- OpenMP

- Communications implicites intra-noeud, via la mémoire partagée
- Utilisation efficace des SMP;
- Facilite la programmation multi-thread;
- Pas de surcoût de communication;
- Standard de fait et supporté par la plupart des compilateurs
- Le processus va aux données

# LE MEILLEUR DES DEUX

- MPI pour les communications inter-nœuds;
- MPI pour l'envoi de structures de données et des schémas de communication;
- La synchronisation de l'état du programme est explicite;

- OpenMP permet une utilisation efficace du multi-threading dans un noeud.;
- OpenMP fournit une interface pour l'utilisation concurrente de chaque nœud SMP de la machine;
- La synchronisation de l'état du programme est implicite;



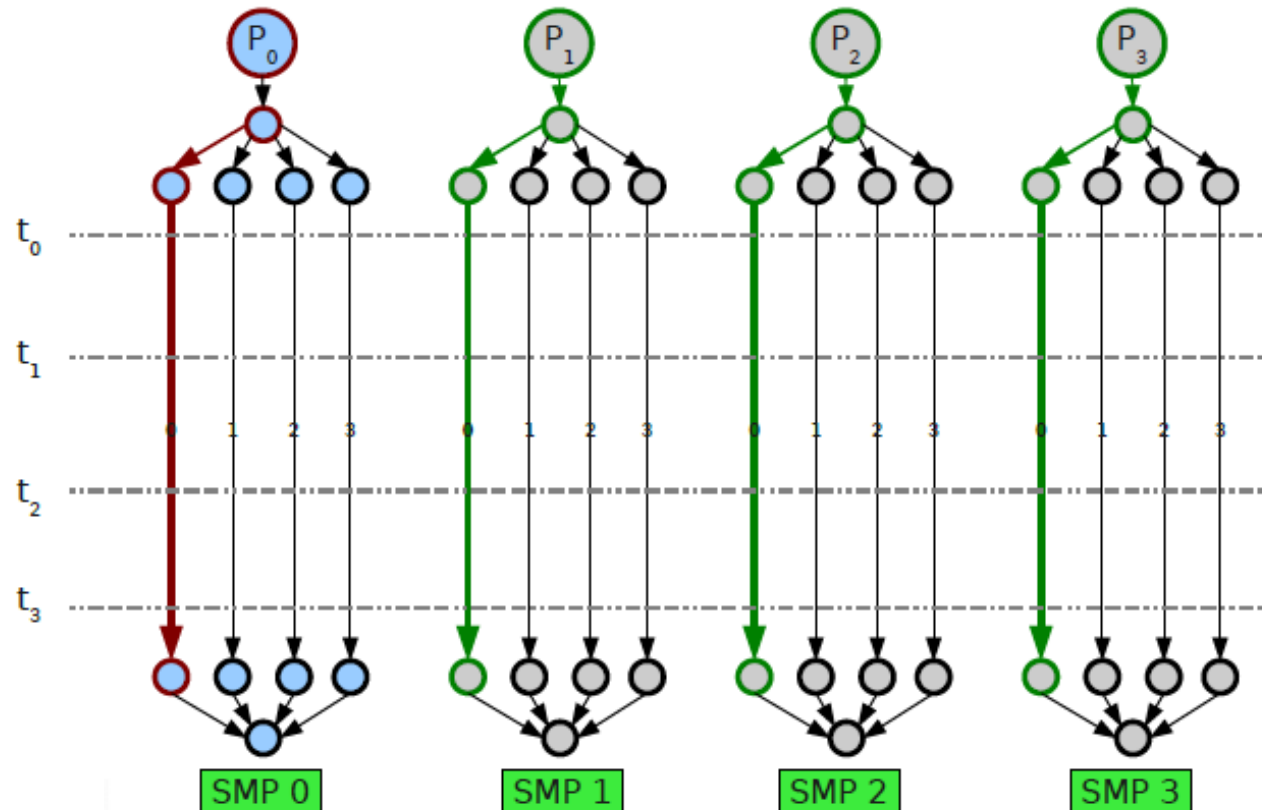
# POURQUOI UN CODE HYBRIDE OPENMP/MPI CODE EST QUELQUEFOIS PLUS LENT?

- OpenMP est moins « scalable » du fait du parallélisme implicite.
- Tous les threads sont inoccupés à part un pendant une communication :
  - Nécessité de recouvrir les communications avec du calcul pour améliorer les performances.
  - Sections critiques (donc sans parallélisme) pour les variables partagées.
- Surcoût du à la création des threads.
- Cohérence des caches, placement des données.
- Peu de librairies et de compilateurs OpenMP optimisés


# ***NOTIONS DE BASE DE L'HYBRIDATION***

# UN EXEMPLE CLASSIQUE DE SCÉNARIO

1. Un process MPI sur chacun des nœuds SMP du cluster;
2. Chaque process lance N threads sur chacun des nœuds SMP;
3. A des points de synchronisation globale, le thread maître de chaque SMP communique avec les autres;
4. Les threads continuent jusqu'à un nouveau point de synchro ou à la terminaison;



```
#include <omp.h>
#include "mpi.h"
#include <stdio.h>
#define _NUM_THREADS 4
/* Each MPI process spawns a distinct OpenMP
 * master thread; so limit the number of MPI
 * processes to one per node */
int main (int argc, char *argv[]) {
    int p,my_rank,c;
    /* set number of threads to spawn */
    omp_set_num_threads(_NUM_THREADS);
    /* initialize MPI stuff */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    /* the following is a parallel OpenMP
     * executed by each MPI process */
    #pragma omp parallel reduction(+:c)
    {
        c = omp_get_num_threads();
    }
    /* expect a number to get printed for each MPI process */
    printf("%d\n",c);
    /* finalize MPI */
    MPI_Finalize();
    return 0;
}
```

- Une application MPI existante :
  - Réaménagement du programme avec des ajouts de directives OpenMP
- Une application OpenMP existante :
  - Réaménagement du programme avec ajout de parallélisation inter-nœuds via MPI
- Une application séquentielle existante :
  - Tout est à faire.
- La manière la plus simple (et avec le moins d'introduction d'erreurs) est d'utiliser MPI en dehors des régions parallèles et d'autoriser uniquement le thread maître de communiquer entre les tâches MPI.
- On peut utiliser des appels MPI à l'intérieur de région parallèle uniquement avec des implémentations MPI thread-safe. 

- Porte le plus souvent sur le partage du travail de boucles simples;
- C'est le plus facile des deux options de réaménagement car les synchronisations du programme sont déjà traitées de façon explicite;
- Les gains dépendent de combien de boucles simples peuvent être partagés, sinon, les effets ont tendance vers l'utilisation de moins de processus MPI;
- Le nombre de processus MPI par nœud SMP dépendra du nombre de threads qu'on veut utiliser par processus;
- Le plus de gain est à attendre pour les applications limitées par les communications.
  - L'introduction d'OpenMP réduit le nombre de processus MPI qui ont besoin de communiquer,
  - Mais l'utilisation du CPU du processeur sur chaque nœud devient un problème;

- Pas aussi direct que l'inverse, car l'état global du programme doit être géré par MPI;
- Nécessite d'être précautionneux : comment chaque process va-t-il communiquer avec les autres? ;
- Peut nécessiter une refonte complète de la parallélisation;
- Un réaménagement réussi d'applications OpenMP avec MPI conduit généralement à de meilleures améliorations de performances et de scaling car le programme original en mémoire partagé exploite déjà très bien le nœud SMP.

## OpenMP dans des applications MPI :

- Relativement simple car la gestion de la mémoire distribuée des nœuds SMP est déjà réalisée.
- Les applications MPI qui sont limitées par les communications et qui sont constituées de beaucoup de boucles simples qui peuvent être partagées peuvent bénéficier d'une grosse accélération du fait de la réduction des besoins de communication entre les nœuds SMP.



## Ajout de MPI à des applications OpenMP :

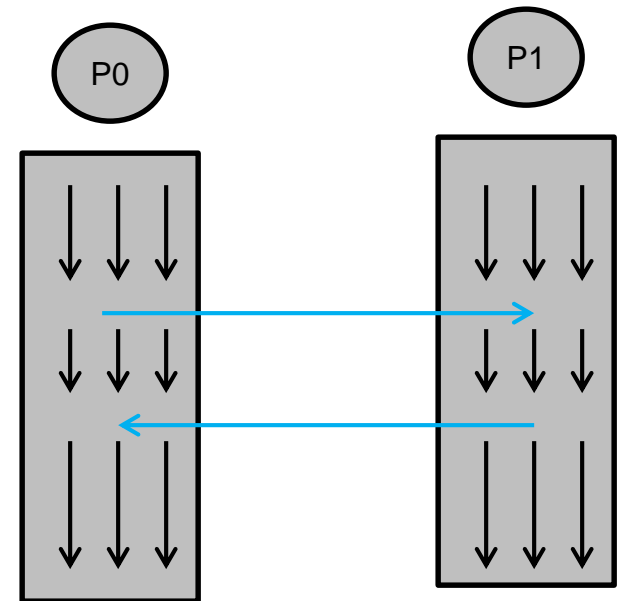
- Pas direct mais peut conduire à de bien meilleurs scaling dans bien des cas.
- Les applications OpenMP gèrent l'état du programme de manière implicite et l'introduction d'appels à MPI impliquent la gestion explicite de l'état du programme.
- De manière générale, ajouter du MPI aux applications OpenMP, conduit à reconcevoir l'application à la base afin de gérer explicitement les synchronisations entre les nœuds.
- Heureusement, la plupart de l'ancien code OpenMP pourra certainement être réutilisé.

- La reconception d'une application, utilisant soit OpenMP soit MPI, est la situation idéale, mais pas toujours possible ou même souhaitable;
- Les gains attendus sont plus importants lorsqu'on introduit du MPI dans des applications à mémoire partagée.
- Une grande attention doit être portée sur trouver le bon équilibre entre les échanges MPI et le « travail » OpenMP
- Points à traiter en priorité :
  1. Les communications entre les nœuds devra être réduit au minimum afin de maximiser le passage à l'échelle.
  2. Les calculs en mémoire partagée sur chaque nœud devront utiliser le maximum de threads durant les phases de calcul.
  3. MPI est plus efficace lors de la communication d'un petit nombre de gros messages.

# EXEMPLE CONCEPT 1 : *ROOT MPI PROCESS CONTRÔLE TOUTES LES COMMS*

- Paradigme le plus direct;
- Un process MPI par nœud SMP;
- Chaque process MPI lance un nombre fixé de threads;
- La communication entre les process MPI est gérée par le process MPI maître à des intervalles réguliers prédéterminés;
- Permet un contrôle fin des communications;

```
// do only if master thread, else wait
#pragma omp master
{ if (0 == my_rank)
  // some MPI_ call as ROOT process
else
  // some MPI_ call as non-ROOT process
}
// end of omp master
```



## EXEMPLE CONCEPT 1

```

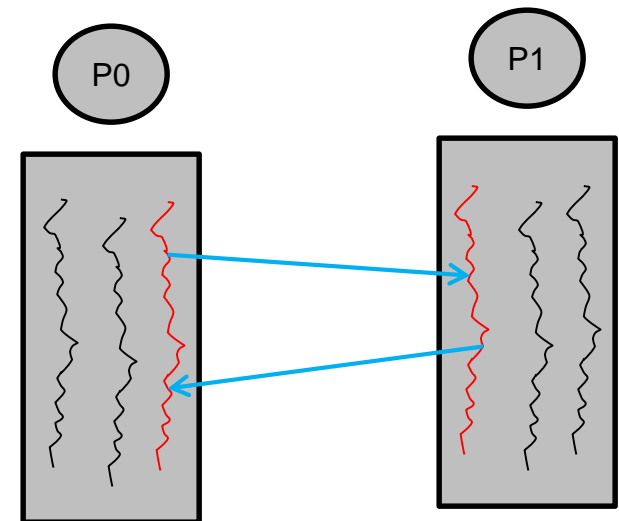
#include <omp.h>
#include "mpi.h"
#include <stdio.h>
#define _NUM_THREADS 4
int main (int argc, char *argv[]) {
    int p, my_rank, c;
    /* set number of threads to spawn */
    omp_set_num_threads( _NUM_THREADS );
    /* initialize MPI stuff */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    /* the following is a parallel OpenMP
    * executed by each MPI process
    */
    #pragma omp parallel reduction(+:c) {
        #pragma omp master {
            if ( 0 == my_rank)
                // some MPI_ call as ROOT process
                c = 1;
            else
                // some MPI_ call as non-ROOT process
                c = 2;
        }
    }
    /* expect a number to get printed for each MPI process */
    printf("%d\n", c);
    /* finalize MPI */
    MPI_Finalize();
    return 0;
}

```

## EXEMPLE CONCEPT 2 : *THREAD OPENMP MAITRE CONTROLE TOUTES LES COMMS*

- Chaque process MPI passe par son thread OpenMP maître ( 1 par nœud SMP) pour communiquer;
- Permet des communications encore plus asynchrones;
- Plus de flexibilité que dans l'exemple 1;
- Faire attention à l'efficacité des communications, mais la flexibilité peut conduire à plus d'efficacité à certains endroits;

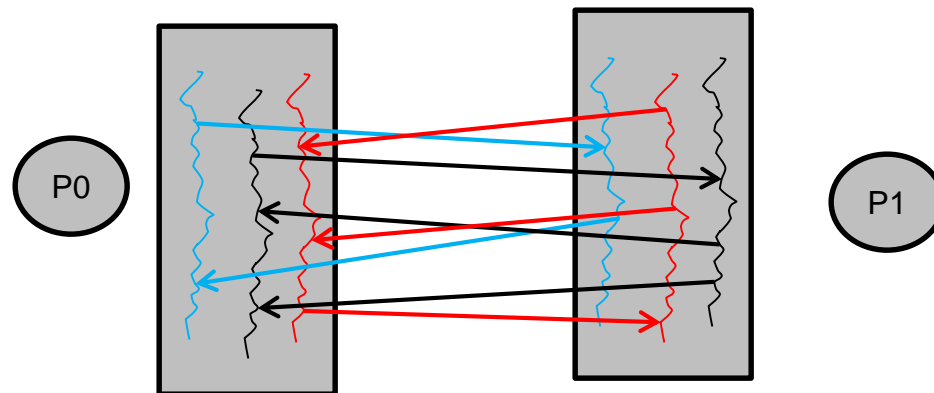
```
// do only if master thread, else wait
#pragma omp master
{
// some MPI_ call as an MPI process
}
// end of omp master
```



```
#include <omp.h>
#include "mpi.h"
#include <stdio.h>
#define _NUM_THREADS 4
int main (int argc, char *argv[]) {
    int p,my_rank;
    int c = 0;
    /* set number of threads to spawn */
    omp_set_num_threads(_NUM_THREADS);
    /* initialize MPI stuff */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    /* the following is a parallel OpenMP
    * executed by each MPI process
    */
    #pragma omp parallel {
        #pragma omp master {
            // some MPI_ call as an MPI process
            c = 1;
        }
    }
    /* expect a number to get printed for each MPI process */
    printf("%d\n",c);
    /* finalize MPI */
    MPI_Finalize();
    return 0;
}
```

## EXEMPLE CONCEPT 3 : *TOUS LES THREADS OPENMP PEUVENT CONTENIR DES APPELS MPI*

- C'est le schéma de communication le plus flexible;
- Permet un comportement complètement distribué comme avec un programme MPI pur;
- Les plus grands risques d'inefficacité sont liés à cette approche;
- Il faut faire très attention dans la gestion de quel thread communique avec quel process MPI;
- Nécessite un « marquage » spécifique pour identifier quel process MPI participe à une communication et quel thread du process MPI est impliqué, e.g., `<my_rank,omp_thread_id>`;
- Ni MPI, ni OpenMP n'offre ce genre de possibilité nativement;



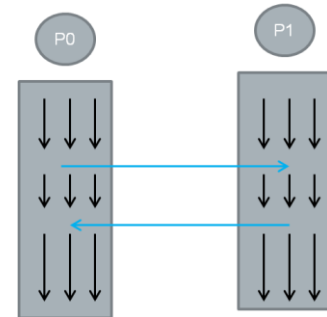
```
#include <omp.h>
#include "mpi.h"
#include <stdio.h>
#define _NUM_THREADS 4
int main (int argc, char *argv[]) {
    int p,my_rank;
    int c = 0;
    /* set number of threads to spawn */
    omp_set_num_threads(_NUM_THREADS);
    /* initialize MPI stuff */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    /* the following is a parallel OpenMP
    * executed by each MPI process
    */
    #pragma omp parallel {
        #pragma omp critical /* not required */ {
            // some MPI_ call as an MPI process
            c = 1;
        }
    }
    printf("%d\n",c);
    /* finalize MPI */
    MPI_Finalize();
    return 0;
}
```



## COMPARAISON DES EXEMPLES

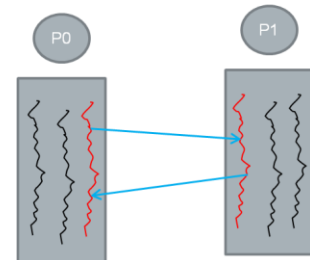
1

```
// do only if master thread, else wait
#pragma omp master
{
    if (0 == my_rank)
        // some MPI_ call as ROOT process
    else
        // some MPI_ call as non-ROOT process
}
// end of omp master
```



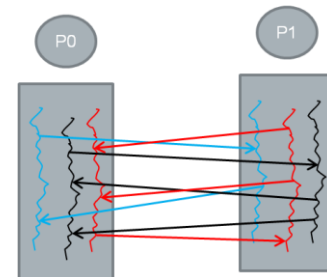
2

```
// do only if master thread, else wait
#pragma omp master
{
    // some MPI_ call as an MPI process
}
// end of omp master
```



3

```
// each thread makes a call; can utilize
// critical sections for some control
#pragma omp critical
{
    // some MPI_ call as an MPI process
}
```



- Le ratio entre communications et temps de calcul sur chaque nœud SMP doit être minimisé afin d'augmenter les capacités de scaling de l'application hybride;
- Introduire des directives OpenMP dans une application MPI est plus simple, mais les bénéfices potentiels ne sont pas aussi grand que dans le cas contraire;
- Les plus gros gains sont souvent constatés dans le cas d'une reconception complète de l'application, heureusement une grande partie du code existant peut-être réutilisé;
- Il existe énormément d'autres possibilités que les 3 présentées;
- Dans tous les cas il faut faire très attention à la programmation afin de s'assurer de la correction des schémas de communication et de leur efficacité.

# ***EXAMPLES***

***EXEMPLE 1 : HELLO WORLD***

```
#include <stdio.h>
#include "mpi.h"
#include <omp.h>

int main(int argc, char *argv[]) {
    int numprocs, rank, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int iam = 0, np = 1;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(processor_name, &namelen);

    #pragma omp parallel default(shared) private(iam, np)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Hello from thread %d out of %d from process %d out
              of %d on %s\n",
              iam, np, rank, numprocs, processor_name);
    }

    MPI_Finalize();
}
```

```
> export OMP_NUM_THREADS=4  
> mpirun -np 2 -machinefile machinefile.morab -x OMP_NUM_THREADS ./hello
```

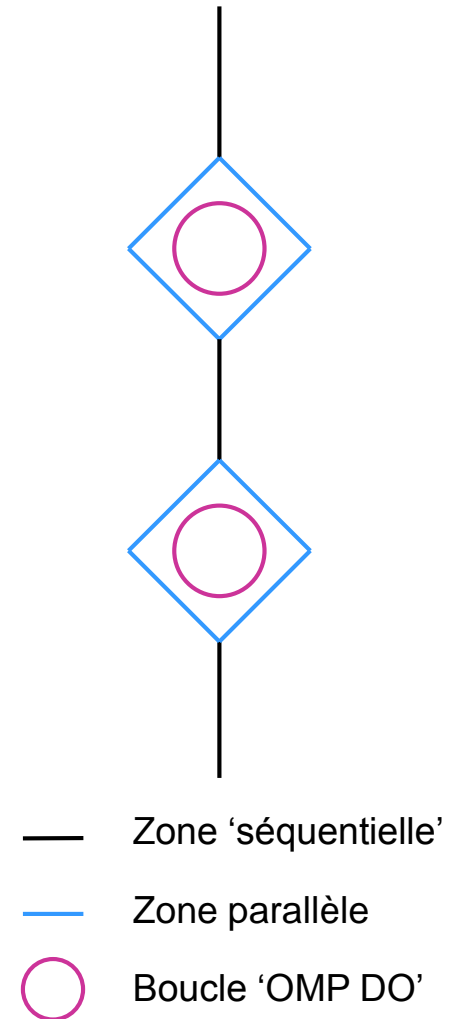
```
Hello from thread 0 out of 4 from process 0 out of 2 on morab006  
Hello from thread 1 out of 4 from process 0 out of 2 on morab006  
Hello from thread 2 out of 4 from process 0 out of 2 on morab006  
Hello from thread 3 out of 4 from process 0 out of 2 on morab006  
Hello from thread 0 out of 4 from process 1 out of 2 on morab001  
Hello from thread 3 out of 4 from process 1 out of 2 on morab001  
Hello from thread 1 out of 4 from process 1 out of 2 on morab001  
Hello from thread 2 out of 4 from process 1 out of 2 on morab001
```

- On fixe pour MPI le nombre de threads par process MPI via la variable OMP\_NUM\_THREADS.

## ***EXEMPLE 2 : TRACE DE MATRICE***

## → *Parallélisation au niveau des boucles*

- Approche la plus commune avec OpenMP :
  - Facile à utiliser
  - Proche de la parallélisation automatique
  - Développement incrémental
  
- Mais :
  - En général performances médiocres





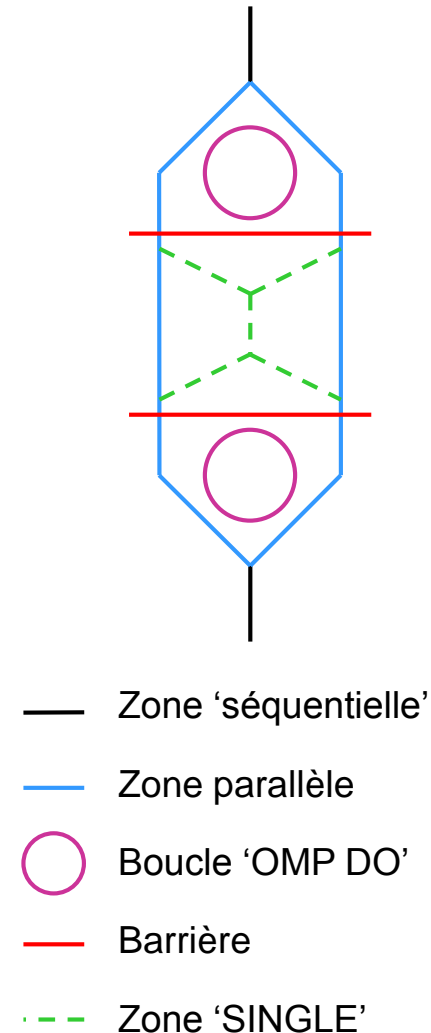
# TRACE D'UNE MATRICE – GRAIN FIN

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

void main(int argc, char ** argv) {
    int me, np, root=0;
    int N; /* On suppose que  $N = m*np$  */
    double A[N][N];
    double traceA = 0;

    /* Initialisation de A */
    /* ... */
    #pragma omp parallel for default(shared) reduction(+:traceA) {
        for (i=0; i<N; i++) {
            traceA += A[i][i]
        }
    }
    printf("La trace de A est : %f \n", traceA);
}
```

- Essaie d'obtenir de meilleures performances du modèle grain fin tout en gardant sa simplicité
- Consiste à éviter les problèmes supposés connus :
  - Pas de multiplication des régions parallèles (pour éviter les coûts de management des threads)
- Utilisation plus avertie des directives :
  - Plus seulement **!\$OMP PARALLEL DO (NoWait)**
  - Utilisation de directives de synchronisation



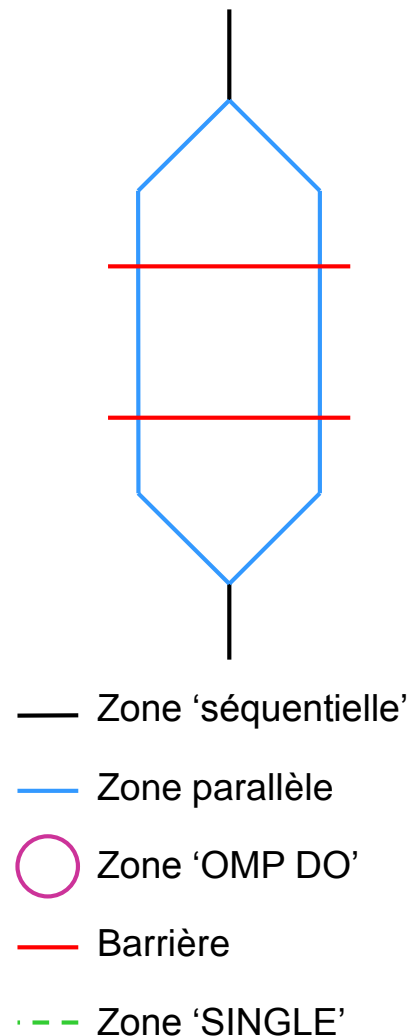
# TRACE D'UNE MATRICE : GRAIN FIN AMÉLIORÉ

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

void main(int argc, char ** argv) {
    int me, np, root=0;
    int N; /* On suppose que  $N = m*np$  */
    double A[N][N];
    double traceA = 0;
    #pragma omp parallel default(shared) {
        /* Initialisation de A */
        /* ... */
        #pragma omp for reduction(+:trace) {
            for (i=0; i<N; i++) {
                traceA += A[i][i]
            }
        }
        #pragma omp master {
            printf("La trace de A est : %f \n", traceA);
        }
    }
}
```

➔ *Exploiter la stratégie de décomposition de domaines dans un environnement multithread*

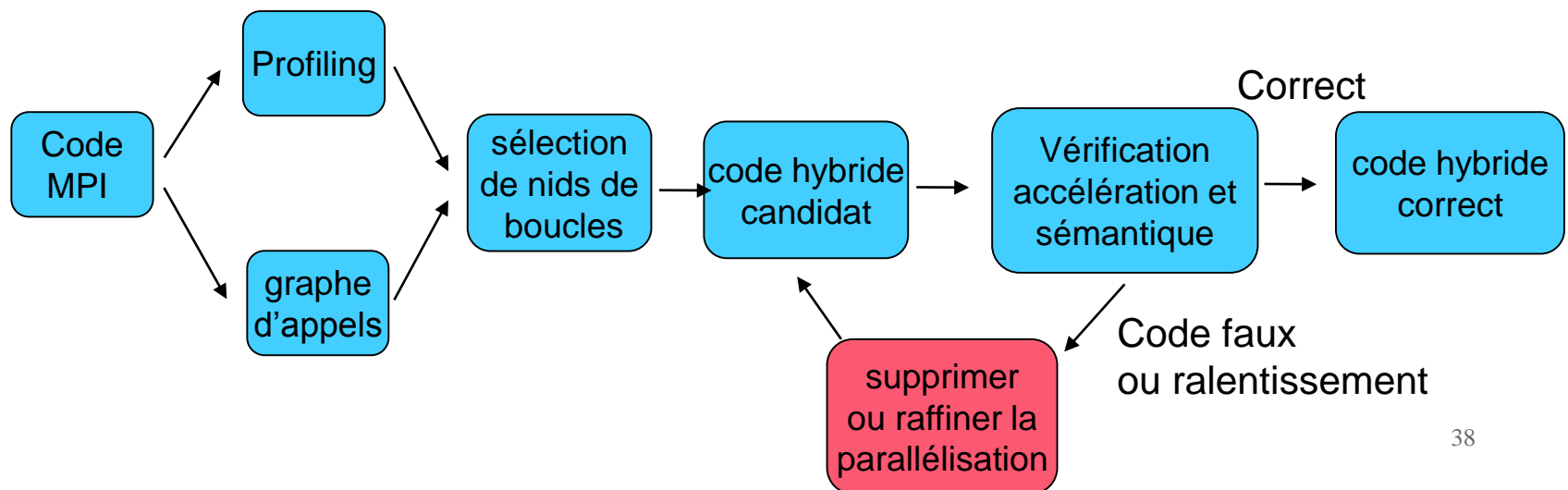
- Décomposition de domaines et partage du travail identiques à MPI
- Echanges de données réécrits en utilisant la mémoire partagée
- Si une version de MPI du programme existe déjà, possibilité d'écrire le programme OpenMP SPMD à partir de celle-ci



# TRACE D'UNE MATRICE : SPMD

```
#include <stdio.h>
#include <omp.h>
void main(int argc, char ** argv) {
    int me, np, root=0, i1, i2;
    int N; /* On suppose que  $N = m*np$  */
    double A[N][N];
    double traceA = 0, traceloc[MAX_PROC];
    #pragma omp parallel default(shared) private(i1, i2, traceA){
        /* Initialisation de A */
        np = omp_get_num_threads();
        me = omp_get_thread_num();
        i1 = (N/np)*me;
        i2 = i1 + (N/np);
        trace_loc[me] = 0;
        for (i=i1; i<i2; i++)  trace_loc[me] += A[i][i];
        #pragma omp barrier
        /* Une reduction 'manuelle' */
        for (i=0; i<np; i++) traceA +=trace_loc[i];
        #pragma omp master { printf("La trace de A est : %f \n",    traceA);
        }
    } /* fin de la region parallele */
}
```

- Sélection des nids de boucle à paralléliser à partir de profiling
- Parallélisation de la la boucle la plus extérieure possible (appels de fonction imbriqués)
- Certains nids de boucle sont transformés pour devenir parallélisables ou pour atteindre une accélération raisonnable (échange éclatement, fusion de boucles, etc.)
- Certaines parties du corps de boucle sont modifiés pour éviter des synchronisations coûteuses ou réduire les pénalités mémoire (ex: le faux partage - false sharing)
- Parallélisation du remplissage/vidage des tampons de communication
- Modification ou changement de l'algorithme



# TRACE D'UNE MATRICE : HYBRIDE GRAIN FIN (1)

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

void main(int argc, char ** argv) {
    int me, np, root=0;
    int N; /* On suppose que  $N = m*np$  */
    double A[N][N];
    double buffer[N], diag[N];
    double traceA, trace_loc;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    tranche = N/np;

    /* Initialisation de A faite sur 0 */
    /* ... */
```

## TRACE D'UNE MATRICE : HYBRIDE GRAIN FIN (2)

```
/* On bufferise les éléments diagonaux depuis le  
processus maître */
```

```
if (me == 0) {  
    for (i=root; i<N; i++)  
        buffer[i] = A[i][i];  
}
```

```
/* L'opération de scatter permet de distribuer la  
diagonale bufférisée entre les processus */
```

```
MPI_Scatter(  
    buffer, tranche, MPI_DOUBLE,  
    diag,    tranche, MPI_DOUBLE, MPI_COMM_WORLD);
```



# TRACE D'UNE MATRICE : HYBRIDE GRAIN FIN (3)

```
/* On calcule la trace locale sur chaque processeur */
trace_loc = 0;

#pragma omp parallel for default(shared)
reduction(+:trace_loc) {
    for (i = 0; i < tranche; i++)
        trace_loc += diag[i];
}

/* On peut alors effectuer la somme globale */

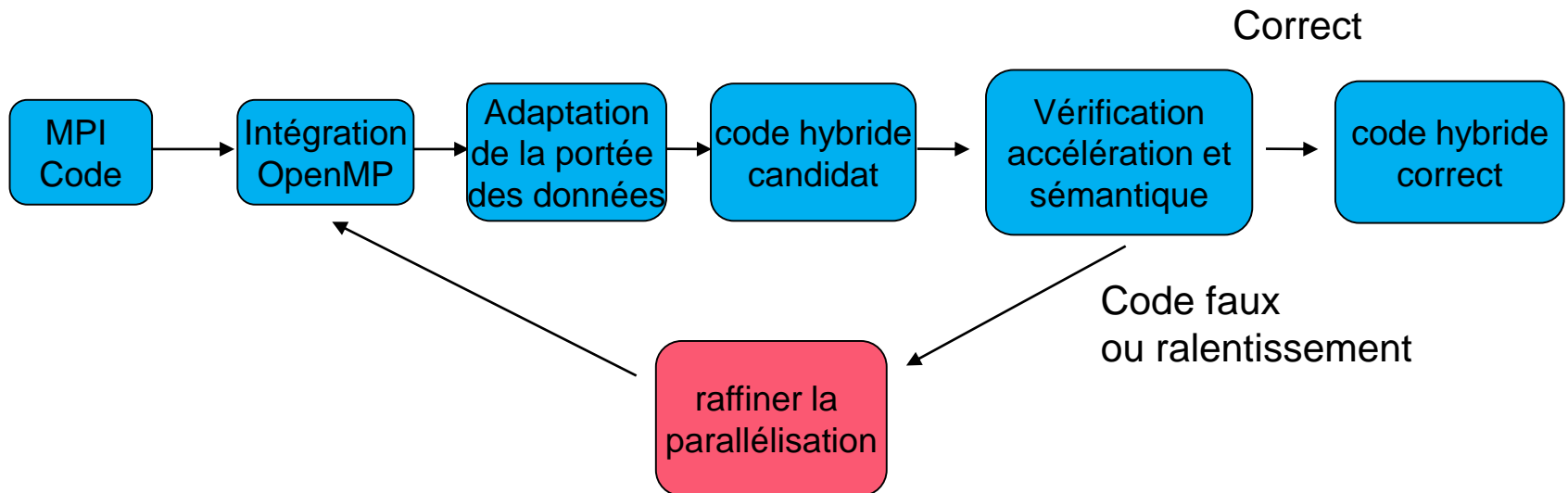
    MPI_Reduce(&trace_loc, &traceA, 1, MPI_DOUBLE, MPI_SUM, root,
MPI_COMM_WORLD);

    if (me == root)
        printf("La trace de A est : %f \n", traceA);

    MPI_Finalize();
}
```

# EFFORT DE PARALLÉLISATION POUR LE GROS GRAIN (OPENMP SPMD)

- La plupart du programme est dans une région parallèle
- Les variables sont SHARED par défaut (attention aux synchronisations)
- Les buffers et les fonctions de communication restent celles du programme MPI initial mais les communications sont à l'intérieur de la région parallèle → 1 seul thread communique (le thread maître)
- Les nids de boucles ne sont pas modifiées par rapport au programme MPI (il n'y a donc pas de directives !\$OMP DO) - les bornes de boucles sont calculées relativement au numéro de thread



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

void main(int argc, char ** argv) {
    int me, me_omp, np_omp, np, root=0;
    int i1, i2
    int N; /* On suppose que  $N = m*np$  */
    double A[N][N];
    double buffer[N], diag[N];
    double traceA, trace_loc_omp[MAX_PROCS], trace_loc;
    #pragma omp parallel default (shared) private(me_omp, np_omp, i1, i2) {
    #pragma omp master {
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &me);
        MPI_Comm_size(MPI_COMM_WORLD, &np);
    }
    me_omp = get_omp_thread_num();
    np_omp = get_omp_num_threads();
```

```
tranche = N/np;

i1 = (tranche/np_omp)*me_omp;
i2 = i1 + (tranche/np_omp);

/* Initialisation de A faite sur 0 */
/* ... */
/* On bufferise les éléments diagonaux depuis le processus maître */

if (me == 0) {
    #pragma omp master {
        for (i=root; i<N; i++)
            buffer[i] = A[i][i];
    }
}
```

```
#pragma omp master {  
    MPI_Scatter(  
        buffer, tranche, MPI_DOUBLE,  
        diag,   tranche, MPI_DOUBLE, MPI_COMM_WORLD);  
}  
  
trace_loc_omp[me_omp] = 0;  
for (i = i1; i < i2; i++)  
    trace_loc_omp[me_omp] += diag[i];  
  
/* Reduction locale puis globale */  
#pragma omp barrier  
#pragma omp master {  
    for (i = 0; i < np_omp; i++)  
        trace_loc += trace_loc_omp[i];  
    MPI_Reduce(&trace_loc, &traceA, 1, MPI_DOUBLE, MPI_SUM, root,  
MPI_COMM_WORLD);  
}
```

```
if (me == root) {  
#pragma omp master {  
    printf("La trace de A est : %f \n", traceA);  
}  
}  
  
#pragma omp master {  
    MPI_Finalize();  
}  
} /* fin du omp parallel */  
}
```

- Pour éviter la corruption des données, les routines d'une application temps-réel doivent être réentrantes.
- Une fonction réentrante est une fonction dont les résultats ne sont pas altérés si elle est exécutée par plusieurs tâches.
- Mécanismes pour assurer la réentrance d'une fonction :
  - Utiliser seulement des variables locales;
  - Garder par des sémaphores l'accès aux variables partagées (globales ou statiques);
  - Désactiver les interruptions ou l'ordonnanceur avant d'accéder aux variables globales ou statiques.

