Programmation parallèle hybride

Marc Tajchman

CEA - DEN/DM2S/STMF/LMES

10/12/2020

Cas envisagés

Du fait qu'il y a souvent plusieurs dispositifs matériels (clusters, machines multi-cœurs, GPU, etc.) et plusieurs outils logiciels associés, il est intéressant d'essayer de combiner leur utilisation.

En fonction de ce qui est disponible sur les machines

1. clusters (plus généralement, machines parallèles multi-nœuds) : MPI,
2. processeurs multi-cœurs : OpenMP (et autres outils : TBB, std::threads, ...),
3. accélérateurs de calcul (GPU ou autres) : Cuda, OpenCL,
4. outils PGAS ("partitionned global addressing system") : pour information (faibles performances),

on testera plusieurs combinaisons.

Machines multi-nœuds et multi-cœurs

Pour utiliser la puissance de calcul de ce type de machine, on a souvent le choix entre 2 possibilités :

- **MPI** pour le parallélisme multi-nœuds et **MPI** pour le parallélisme multi-cœurs (plusieurs processus MPI sur chaque nœud).
- **MPI** pour le parallélisme multi-nœuds et **OpenMP** pour le parallélisme multi-cœurs (un processus MPI sur chaque nœud).

Sur des simulations de taille petite ou moyenne, on préfère souvent le premier choix pour des raisons de simplicité.
Par contre, sur des simulations de (très) grande taille, le “tout MPI” atteint plus rapidement ses limites d'utilisation.

Avantages/inconvénients du tout MPI:

- ⊕ programmation MPI nœuds-cœurs identique à la programmation MPI entre les nœuds
- ⊕ les librairies MPI sont de mieux en mieux optimisées (en particulier si plieurs processus sont sur le même processeur)
- ⊖ le nombre de processus MPI augmente plus vite (n° cœurs $\times$ n° nœuds), les structures internes de MPI, la mémoire utilisée pour les communications aussi ("cellules fantomes", "halos")

On atteint actuellement les limites des machines parallèles sur des simulations avec des nombres de processus $> 10^6$.

Avantages/inconvénients de la combinaison MPI-OpenMP:

- ⊕ les possibilités en nombre de nœuds-cœurs sont plus importantes
  Le nombre total de processus MPI diminue (un seul processus MPI par nœud de calcul.)

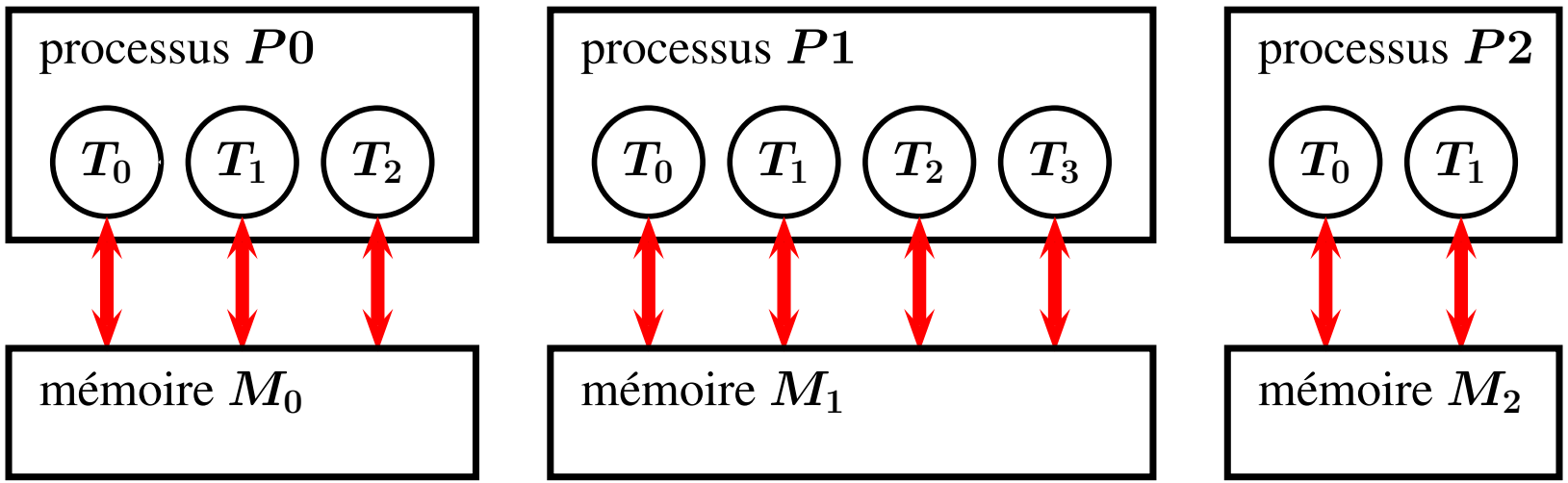
- ⊖ la programmation MPI sur les nœuds et OpenMP sur les cœurs est plus complexe

- ⊖ l'amélioration des performances n'est pas toujours évidente surtout pour des nombres de processus petits ou moyens.

Exemples4/MemoireMPI : ressource mémoire utilisée par MPI (MPI_Init et MPI_Finalize, seulement, pas d'échange de messages).

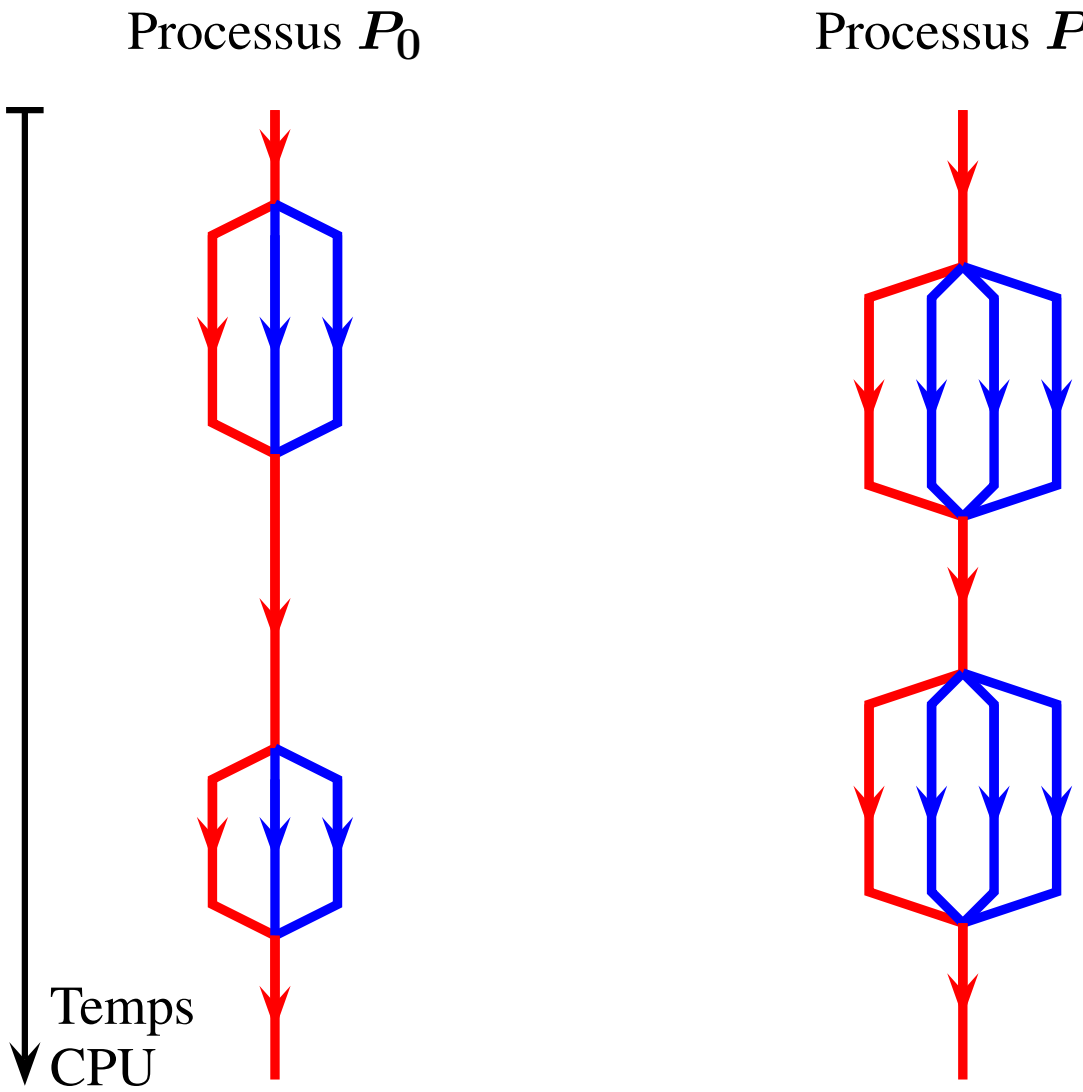
Lire le fichier README.txt pour des instructions

Combinaison des modèles MPI et OpenMP en modèle hybride MPI-OpenMP:

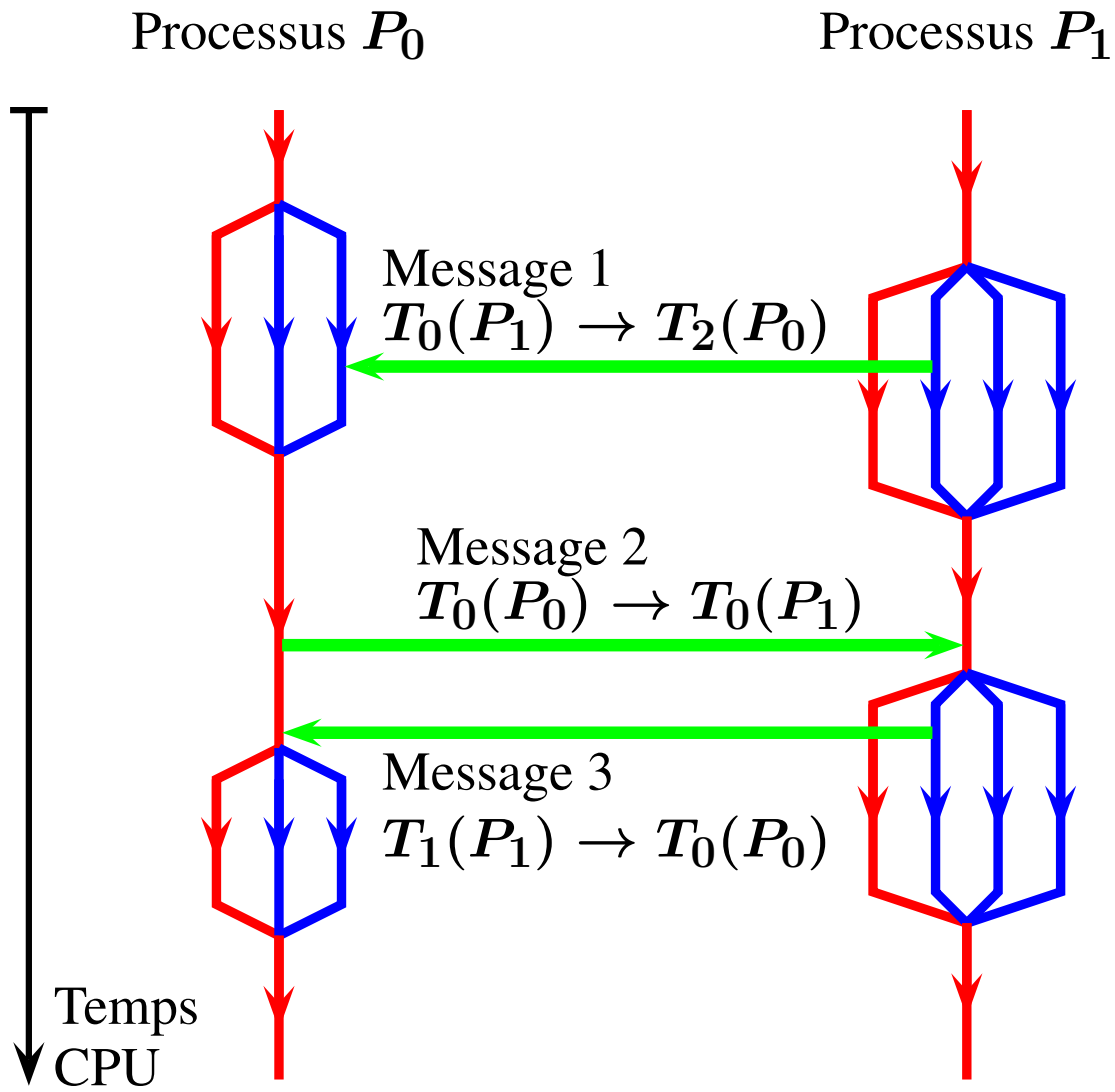


- Les différents processus P_0, P_1, ... ont chacun leur mémoire locale M_0, M_1, ...
- A l'intérieur de chaque processus P_i, un ou plusieurs threads travaillent avec la mémoire locale M_i.
- Tous les threads peuvent *a priori* envoyer et recevoir des messages MPI.

Exemple sur 2 processus, chacun avec plusieurs threads



Quelques types différents de messages



- Pas de synchronisation a priori entre les sections multi-threads entre les différents processus, donc besoin éventuel de combiner des barrières OpenMP et MPI

- 2 types de réductions dans OpenMP et MPI, donc pour faire une réduction complète, il faut combiner ces 2 réductions

- 2 threads peuvent envoyer chacun un message. Comment distinguer ces messages (l'ordre d'arrivée est inconnu) ?

- Si un message arrive dans un processus qui est dans une région multi-threads, quel thread va recevoir et traiter le message ?

Le grand nombre de situations possibles est très difficile à gérer pour ceux qui développent les libraires OpenMP et MPI

Pour cette raison, 4 niveaux de compatibilité entre OpenMP et MPI, ont été définis:

- `MPI_THREAD_SINGLE`:
  Un seul thread par processus (donc pas d'OpenMP)
- `MPI_THREAD_FUNNELED`:
  Parmi les threads, seul celui qui a initialisé MPI peut utiliser des fonctions MPI
- `MPI_THREAD_SERIALIZED`:
  A un instant donné, un seul thread peut utiliser des fonctions MPI (mais pas toujours le même thread)
- `MPI_THREAD_MULTIPLE`:
  Pas de restrictions, tous les threads peuvent utiliser les fonctions MPI.

Plus le niveau de compatibilité permet de souplesse dans l'utilisation des fonctions MPI et des pragmas OpenMP, plus la gestion interne des threads dans les processus est coûteuse.
Donc, il faut choisir le niveau le plus bas suffisant pour l'algorithme du code

Pour choisir le niveau de compatibilité, il faut remplacer

```
MPI_Init(int *argc, char ***argv)
```

par

```
MPI_Init_thread(int *argc, char ***argv,
                int required, int *provided)
```

où `required` est un entier qui peut prendre une des 4 valeurs

- `MPI_THREAD_SINGLE`,
- `MPI_THREAD_FUNNELED`,
- `MPI_THREAD_SERIALIZED`,
- `MPI_THREAD_MULTIPLE`

et `provided` est un entier qui reçoit le niveau de compatibilité effectivement proposé par la version de MPI

Il est important de comparer les valeurs de `required` et `provided` pour vérifier que le niveau demandé est effectivement disponible.

`MPI_Init(...)` est équivalent à `MPI_Init_thread(..., MPI_THREAD_SINGLE. ...)`

Dans les cas où plusieurs threads envoient/reçoivent des messages, **il faut utiliser les tags** pour distinguer ces messages.

Sans utiliser le tags, parfois cela marche, parfois non.

Situation analogue aux échanges MPI asynchrones (`MPI_Isend/MPI_Irecv`). D'ailleurs, MPI utilise aussi de threads en interne pour ce type d'échanges.

Exemples4/Hybrides : différents niveaux de combinaison MPI-OpenMP

Remarque:

Les implémentations de MPI et OpenMP ont fait beaucoup de progrès dans les versions récentes.

Même si on exécute un code qui utilise un niveau de compatibilité plus élevé que celui demandé dans `MPI_Init_thread`, *cela peut marcher, mais ce n'est pas garanti.*

Par sécurité, demander un niveau de compatibilité suffisant.

Suivant les algorithmes du code et d'OpenMP utilisés, on choisira entre les niveaux

- `MPI_THREAD_FUNNELED` pour pouvoir faire des appels MPI dans les régions séquentielles d'OpenMP grain fin, ou dans les régions contrôlées par les pragma OpenMP `MASTER`
- `MPI_THREAD_SERIALIZED` pour pouvoir faire des appels MPI dans les régions contrôlées par les pragma OpenMP `SINGLE`
- `MPI_THREAD_MULTIPLE` pour pouvoir faire des appels MPI dans les régions parallèles d'OpenMP générales

Exemple MPI-OpenMP grain fin

```
1  MPI_Init(..., MPI_FUNNELED, ...);
2  int nX = // nombre de points locaux
3  doubl dT, dT_local;
4  for (iT=0; iT < nT; iT++)
5  {
6    dT_local = calcul_dt(u);
7    MPI_Allreduce(&dT_local,&dT, ...)
8
9  #pragma omp parallel for
10   for (iX=0; iX < nX-1; iX++)
11     v[iX]=f(u[iX-1],u[iX],u[iX+1],dT);
12
13   echange(u, v);
14   //echanges MPI sur frontiere locale
15 }
```

Exemple MPI-OpenMP grain grossier (version 1)

```
1  MPI_Init(..., MPI_FUNNELED, ...);
2  int nX = // nombre de points locaux (MPI)
3
4  #pragma omp parallel default(shared)
5  {
6    int iT, iX;
7    for (iT=0; iT < nT; iT++) {
8
9  #pragma omp master
10   {
11     dT_local = calcul_dt(u);
12     MPI_Allreduce(&dT_local, &dT, ...)
13   }
14 #pragma omp barrier
```

Exemple MPI-OpenMP grain grossier (version 1, suite)

```
15    #pragma omp for
16    for (iX=1; iX < nX-1; iX++)
17      v[iX] = f(u[iX-1],u[iX],u[iX+1],dT)
18
19    #pragma omp master
20    {
21      echange(u, v);
22      //echanges MPI sur frontiere locale
23    }
24  }
```

Exemple MPI-OpenMP grain grossier (version 2)

```
1  MPI_Init(..., MPI_SERIALIZED, ...);
2  int nX = // nombre de points locaux (MPI)
3
4  #pragma omp parallel default(shared)
5  {
6    int iT, iX;
7    for (iT=0; iT < nT; iT++) {
8
9    #pragma omp single
10   {
11     dT_local = calcul_dt(u);
12     MPI_Allreduce(&dT_local, &dT, ...)
13   }
```

Exemple MPI-OpenMP grain grossier (vrsion 2, suite)

```
14
15    #pragma omp for
16    for (iX=1; iX < nX-1; iX++)
17      v[iX] = f(u[iX-1],u[iX],u[iX+1],dT)
18
19    #pragma omp single
20    {
21      echange(u, v);
22      // echanges MPI sur frontiere locale
23    }
24  }
```

Exemple4/PoissonMPI : code MPI auquel est ajoutée une parallélisation OpenMP grain fin