# Une introduction à la programmation parallèle avec Open MPI et OpenMP

Alban Mancheron

# Une introduction à la programmation parallèle avec Open MPI et OpenMP

**Alban Mancheron**
**[Enseignant-Chercheur en bioinformatique à l'Université de Montpellier, linuxien depuis 1997 (convaincu depuis 1998)]**

***Alors que nos machines utilisent plusieurs processeurs et que fleurissent de plus en plus de centres de calculs distribués, la plupart des algorithmes que nous concevons sont pensés pour une exécution séquentielle. Explorons les possibilités ouvertes par l'implémentation des standards OpenMP (Open Multi-Processing) et MPI (Message Passing Interface) pour le langage C/C++.***

*/// Mots-clés ///*

Parallélisme (léger, lourd, hybride) , algorithmique distribuée, Open MPI, OpenMP, C++.

*/// Fin Mots-clés ///*

Avant d'entrer dans le vif du sujet, je tiens à préciser que, hormis une connaissance des bases du **C++**, aucune connaissance de l'algorithmique distribuée n'est requise pour aborder cet article.

## Algorithme séquentiel vs. Algorithme parallèle

Un algorithme est une méthode de résolution d'un problème par la description finie d'une séquence d'opérations (qui elle peut être infinie) **[1]**. Par exemple, le matin *je me lève et je te bouscule, tu ne te réveilles pas (comme d'habitude)* **[2]**.

*/// Début Remarque ///*

Tous les algorithmes ne sont pas toujours efficaces…

*/// Fin de la remarque ///*

Lorsque la séquence est représentable par une suite d'états totalement ordonnée, alors l'algorithme est dit séquentiel. Lorsque la suite d'états n'est que partiellement ordonnée (au moins deux actions se déroulent en parallèle) alors on parlera d'algorithme parallèle. Par exemple, *le matin le réveil se déclenche*. D'un côté ma femme se lève puis me bouscule, d'un autre côté, le réveil diffuse une musique trop forte et indépendamment les volets s'ouvrent automatiquement éclairant la pièce d'une lumière agressive. Au final, je ne me réveille toujours pas, comme d'habitude.

*/// Début Remarque ///*

Ce n'est pas parce que l'algorithme est parallèle qu'il est nécessairement plus efficace.…

*/// Fin de la remarque ///*

*In fine*, un algorithme parallèle peut être vu comme plusieurs sous-algorithmes séquentiels qui se déroulent en parallèle. On distinguera cependant les communications entre les sous-algorithmes, selon qu'ils se partagent une même zone d'informations (et potentiellement y accèdent de manière concurrente) ou bien qu'ils utilisent un vecteur de communication dédié pour se partager certaines informations. Dans le premier cas, on parle de parallélisme léger (*multithreading* en anglais) tandis que dans le second cas de parallélisme lourd (*multiprocessing* en anglais).

## Parallélisme léger

Le parallélisme léger traduit le fait que des instructions s'exécutent en parallèle tout en se partageant une même zone mémoire. Cela signifie que si plusieurs blocs d'instructions $B_1$, $B_2$, …, $B_n$ s'exécutent en parallèle,

rien n'empêche par défaut les blocs d'écrire ou lire l'état d'une variable simultanément, pouvant aboutir à une situation imprévisible et chaotique. Ainsi sont apparues les fameux *MutEx*, variables particulières dont le rôle est d'obtenir un verrou exlusif (**Mut**ual **Ex**clusion) sur la mémoire partagée. Pour faire une analogie, imaginons plusieurs voitures (nos processus parallèles) circulant sur un réseau routier (zone de mémoire partagée).Tant que chaque voiture ne rencontre pas les autres, il n'y a pas de problème. Si par contre, deux voitures empruntent la même route, sur la même voie, les problèmes peuvent survenir assez rapidement (surtout si les véhicules roulent à contre-sens l'un de l'autre). Dès lors, il est prudent avant de s'engager sur cette route que chacune des voitures puisse s'assurer un usage exclusif de celle-ci. Cela passe par une demande d'autorisation pour emprunter ladite route à un agent de la sécurité routière (le système d'exploitation). Si la route est libre, l'agent autorise la voiture qui a fait la demande et interdit à tout autre véhicule de l'emprunter. Une fois que la voiture qui s'est engagée a quitté la portion de route dangereuse, il avertit l'agent que la route est à nouveau libre. *A contrario*, si lorsqu'un véhicule effectue la demande d'accès, la route est déjà utilisée par une autre voiture, alors l'agent met le demandeur en attente jusqu'à ce que la route soit libérée. Bien évidemment, la gestion des MutEx est un sujet important. En effet, il faut tout d'abord s'assurer qu'une fois la tâche effectuée, le MutEx est bien relâché. Dans le cas contraire, le risque est de bloquer le système, certains processus demeurant en attente d'obtention du MutEx. En outre, les MutEx créent des goulots d'étranglements, il faut donc les demander le plus tard possible et les relâcher dès que possible. Pour reprendre l'analogie avec les voitures, si je demande l'autorisation exclusive de circuler sur la route menant de mon travail à mon domicile (plusieurs km) alors que la zone de danger ne représente que quelques mètres, je bloque inutilement tout le monde (ce n'est pas très grave, au moins je suis tranquille sur ma route, par contre si quelqu'un faisait ça, je le gratifierai de plusieurs noms d'oiseaux). Bien évidemment, le fait de demander un MutEx a un coût, donc il est important de veiller à regrouper les instructions sensibles dans un même bloc lorsque cela est possible.

## Parallélisme lourd

Le parallélisme lourd peut s'apparenter quant à lui au déroulement de plusieurs algorithmes distincts en simultané. Chacun des algorithmes ayant un objectif spécifique et plus ou moins indépendant des autres. Les algorithmes peuvent cependant avoir besoin d'échanger quelques informations, auquel cas un protocole de communication doit être mis en œuvre. Par exemple, pour modéliser un serveur *web*, il faut un algorithme principal qui écoute sur un port donné jusqu'à ce qu'un client se connecte. Quand un client se connecte un nouvel algorithme prend en charge le client et l'algorithme principal se remet tout simplement en écoute d'un nouveau client. Chaque prise en charge d'un client est indépendante.

Dans certains cas plus complexes, il faut tout de même que les différents processus communiquent un minimum sur leur état ou bien échangent ponctuellement des données. Cela peut s'opérer par l'utilisation de fichiers (type *fifo*) lorsque les processus s'exécutent sur la même machine (ou disposent d'un espace de stockage commun (*e.g.*, montage NFS). Dans un cas plus général, cela peut passer par une communication réseau (*e.g.*, protocoles UDP ou TCP).

## Parallélisme léger + parallélisme lourd

Et non, parallélisme léger + parallélisme lourd, ça ne fait pas du parallélisme moyen, mais c'était bien tenté !

On parle dans ce cas-là de parallélisme hybride. L'idée est de tirer parti des deux paradigmes et c'est ce qu'on va faire dans la suite donc je n'en écris pas plus pour le moment…

## Quand/Comment/Pourquoi paralléliser un algorithme ?

Dans certains cas, la conception d'algorithmes parallèles découle du bon sens et s'avère plus simple à mettre en œuvre qu'en séquentiel (*e.g.*, mise en place d'un serveur *web*). Dans d'autres cas, paralléliser les traitements n'est pas toujours intuitif. Cependant, avec l'arrivée, cette dernière décennie, d'un volume de données à traiter et analyser toujours plus important, cela devient une nécessité.

Dans cet article, je présenterai deux standards permettant de faire l'un du parallélisme léger et l'autre du parallélisme lourd. Le code fourni est écrit en C++, tout d'abord parce que cela me semble cohérent d'utiliser l'un des langages les plus performants pour faire des calculs intensifs et d'autre part parce que le C++ dispose (par rapport au **C**) des facilités offertes par la **STL** (qui n'existe plus *stricto sensu* puisqu'elle a été intégrée au standard C++ **[3]**).

*/// Note ///*

Je profite au passage de l'occasion pour d'une part rappeler qu'un langage est dit de bas niveau lorsqu'il ne présente que peu d'abstraction par rapport aux contraintes matérielles (gestion des registres, du jeu d'instruction des processeurs, …) et que par opposition un langage de haut niveau permet de s'abstraire de

ces contraintes. Ainsi il en résulte que l'**Assembleur** est incontestablement un langage de bas niveau alors que le langage C est par conséquent un langage de haut niveau (qui permet d'inclure des instructions de bas niveau).

*/// Fin Note ///*

# OpenMP

Le standard **OpenMP [4]** est clairement dédié aux parallélismes légers. Pour celles et ceux qui ont déjà fait des programmes parallèles (*e.g.*, avec `pthreads` en C/C++), vous allez constater que l'interface OpenMP simplifie grandement les mécanismes de parallélisation du code. Pour ceux qui n'ont jamais fait de programmes parallèles, vous allez découvrir que vous pouvez probablement paralléliser une partie de vos programmes avec très peu d'effort.

Il est à noter que les implémentations d'OpenMP existent pour les langages C/C++ et **Fortran**. Dans la suite de cet article, je n'évoquerai que l'API C/C++ d'OpenMP.

## Principe de fonctionnement

Lorsque l'on envisage de paralléliser tout ou partie d'un algorithme, il faut identifier les blocs d'instructions qui sont complètement indépendants entre eux, ceux qui ne le sont que partiellement et ceux qui ne le sont pas du tout.

Un exemple simple d'algorithme permettant de calculer la moyenne d'une série de 20 valeurs :

```
Entrée : T un tableau de 20 entiers
Sortie : M un réel
1/ Pour chaque valeur V du tableau T
  a/ M = M + V
2/ M = M / 20
3/ Afficher le résultat
```

En l'état cet algorithme paraît difficilement parallélisable, et pourtant l'étape 1 peut être clairement parallélisée en se rappelant que la moyenne de 20 valeurs peut être obtenue en calculant indépendamment d'un côté la somme $S_1$ des 10 premières valeurs et d'une autre côté la somme $S_2$ des 10 dernières valeurs, puis en sommant $S_1$ et $S_2$. La somme $S_1$ peut également se décomposer en sommant les 5 premières valeurs d'un côté (somme $S_{1a}$) et les 5 suivantes de l'autre (somme $S_{1b}$), et ainsi de suite. Cependant, l'implémentation d'un tel algorithme (appelé « réduction ») demeure fastidieuse alors que cette décomposition reste assez courante et s'adapte à d'autres opérations (trouver le maximum parmi un ensemble de valeur, calculer leur produit, …).

L'idée d'OpenMP (en tout cas, l'idée que moi j'en ai), c'est de faire ce travail pour vous en expliquant juste au compilateur que cette partie doit être parallélisée en faisant une réduction.

Pour cela, OpenMP définit d'une part des directives et schémas de constructions et d'autre part des fonctions. Les directives sont des *pragmas* qui sont analysés par le préprocesseur et qui réécrivent les blocs d'instruction concernés selon le schéma de construction spécifié par la directive. Les fonctions permettent d'écrire des instructions spécifiques lorsqu'aucun schéma de construction ne correspond.

Ainsi l'algorithme de calcul de la moyenne précédent peut se récrire :

```
Entrée : T un tableau de 20 entiers
Sortie : M un réel
# Utiliser le schéma de construction OpenMP « réduction de somme sur la variable M »
1/ Pour chaque valeur V du tableau T
  a/ M = M + V
2/ M = M / 20
3/ Afficher le résultat
```

C'est de la magie où je ne m'y connais pas, *Bibbidi Babbidi Bou* **[5]**.

## Installation

Si vous êtes frustrés de ne pas encore avoir eu de code C++ sous les yeux, c'est probablement que j'ai réussi à vous mettre l'eau à la bouche… Encore un peu de patience, ça va venir. Mais avant cela, il faut installer la librairie de développement OpenMP pour le compilateur **gcc/g++**.

Pour les chanceux qui utilisent une distribution **Debian**/**Ubuntu**, rien de plus simple :

```
$ sudo apt install libgomp1
```

Et si vous voulez installer les symboles de *debuggage*, (typiquement pour une utilisation de **gdb** ou de

**valgrind**) :

```
$ sudo apt install libgomp1-dbg
```

Pour les autres distributions, il faudra chercher sur les forums dédiés à vos distributions.

Si vous envisagez d'utiliser un autre compilateur, alors il faudra également chercher sur les forums dédiés.

Les chanceux sont donc ceux qui n'ont pas besoin de chercher sur les forums :-p.

## L'exemple qui ne sert à rien, mais qui sert quand même

Pourquoi faire un exemple qui ne sert à rien quand on peut en faire plusieurs.

Tout d'abord, notre premier exemple a pour objet de vérifier que le compilateur (ou plus précisément l'éditeur de lien) résout correctement la liaison avec la librairie OpenMP.

Commençons par créer un fichier **helloworld.cpp** assez rudimentaire :

```
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

  for (int i = 0; i < argc; ++i) {
    cout << "L'argument " << i << " a pour valeur"
         << " '" << argv[i] << "'" << endl;
  }
  return 0;
}
```

Compilons-le avec une ligne de commande tout aussi rudimentaire :

```
$ g++ -Wall -ansi -pedantic helloworld.cpp -fopenmp -o helloworld
```

Si tout se passe bien, vous n'avez aucun avertissement sur la sortie standard (je laisse le soin aux lecteurs dubitatifs d'aller regarder l'intérêt des options **-Wall**, **-ansi** et **-pedantic** qui ne sont pas l'objet de cet article). Vous remarquerez l'option **-fopenmp** qui signifie tout simplement de gérer les directives OpenMP.

Exécutons ce programme avec quelques arguments et voyons ce que l'on obtient :

```
$ ./helloworld Premier test OpenMP pour "GNU/Linux Magazine"
L'argument 0 a pour valeur './helloworld'
L'argument 1 a pour valeur 'Premier'
L'argument 2 a pour valeur 'test'
L'argument 3 a pour valeur 'OpenMP'
L'argument 4 a pour valeur 'pour'
L'argument 5 a pour valeur 'GNU/Linux Magazine'
```

Bon, c'était vraiment un exemple qui ne sert pas à grand-chose. Mais nous allons le modifier afin de paralléliser les itérations de la seule et unique boucle du programme. Après tout, chaque boucle est indépendante et les zones mémoires accédées sont distinctes et peut donc être traitée en parallèle…

Pour cela, on insère avant la boucle une directive OpenMP (**#pragma omp**) indiquant d'utiliser le schéma de construction de boucle pour paralléliser (**parallel for**) :

```
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

#pragma omp parallel for
  for (int i = 0; i < argc; ++i) {
    cout << "L'argument " << i << " a pour valeur"
         << " '" << argv[i] << "'" << endl;
  }
  return 0;
}
```

On compile comme précédemment et on exécute avec les mêmes arguments et on observe une sortie réellement chaotique !!!

```
$ ./helloworld Deuxième test OpenMP pour "GNU/Linux Magazine"
L'argument L'argument 04 a pour valeurL'argument  ' a pour valeur./helloworld' 'pour2' a pour valeur
'L'argument test
L'argument '
13 a pour valeur 'OpenMP'
L'argument 5 a pour valeur 'GNU/Linux Magazine'
```

```
 a pour valeur 'Deuxième'
```

C'est vrai que vu sous cet angle, ça valait vraiment le coup de paralléliser cette boucle. Que se passe-t-il ? Et bien tout simplement, chaque tour de boucle est exécuté en parallèle et donc l'affichage aussi. Tous les processus légers écrivent en même temps sur la sortie standard (qui est une variable partagée, contrairement à ce qui est écrit précédemment, zut alors !!!). Il faut donc protéger l'instruction d'écriture sur la sortie standard en réclamant un MutEx, ou mieux en expliquant à OpenMP que l'instruction est une instruction critique.

```
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

#pragma omp parallel for
  for (int i = 0; i < argc; ++i) {
#pragma omp critical
    cout << "L'argument " << i << " a pour valeur"
         << " '" << argv[i] << "'" << endl;
  }
  return 0;
}
```

On retrouve bien l'accès à la variable commune (**cout**) protégé comme étant une section critique. Après compilation et exécution, on a le résultat attendu :

```
$ ./helloworld Troisième test OpenMP pour "GNU/Linux Magazine"
L'argument 1 a pour valeur 'Troisième'
L'argument 3 a pour valeur 'OpenMP'
L'argument 4 a pour valeur 'pour'
L'argument 5 a pour valeur 'GNU/Linux Magazine'
L'argument 2 a pour valeur 'test'
L'argument 0 a pour valeur './helloworld'
```

## Retour en arrière : compiler sans OpenMP

Un des gros inconvénients d'une exécution en parallèle est que cela peut devenir assez rapidement difficile à *debugger*. Fort heureusement, désactiver OpenMP à la compilation est assez facile, il suffit de compiler sans l'option **-fopenmp**. Dans ce cas, vous obtiendrez quelques avertissements :

```
$ g++ -Wall -ansi -pedantic helloworld.cpp -o helloworld
helloworld.cpp:7:0: warning: ignoring #pragma omp parallel [-Wunknown-pragmas]
 #pragma omp parallel for
 ^
helloworld.cpp:9:0: warning: ignoring #pragma omp critical [-Wunknown-pragmas]
 #pragma omp critical
 ^
```

Si comme moi vous avez du mal avec les compilations qui émettent des avertissements (c'est sale, beark) il est possible d'utiliser un mécanisme de compilation conditionnelle afin de n'utiliser les directives OpenMP que lorsque l'option **-fopenmp** est fournie. En effet, OpenMP définit la macro **_OPENMP** (qui contient l'année et le mois au format *yyyymm* de la spécification supportée par l'implémentation courante).

Ainsi, un beau code intégrant des directives OpenMP qui ne sert donc toujours à rien ressemblerait à :

```
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

#ifdef _OPENMP
  cout << "Valeur de la macro _OPENMP : '" << _OPENMP << "'" << endl;
#pragma omp parallel for
#endif
  for (int i = 0; i < argc; ++i) {
#ifdef _OPENMP
#pragma omp critical
#endif
    cout << "L'argument " << i << " a pour valeur"
         << " '" << argv[i] << "'" << endl;
  }
  return 0;
}
```

Compilez ce code avec et sans l'option **-fopenmp** et vous verrez que vous n'aurez aucun avertissement.

# Quelques fonctions OpenMP bien pratiques

Outre les directives, OpenMP propose également quelques fonctions assez utiles, telles que :
- **omp_get_max_threads()**, qui renvoie le nombre maximum de processus légers qui pourront être déclenchés après l'appel de cette fonction ;
- **omp_get_num_procs()**, qui renvoie le nombre de processeurs disponibles sur la machine au moment de l'appel ;
- **omp_get_num_threads()**, qui renvoie le nombre de processus légers correspondants à un bloc d'instruction ;
- **omp_get_thread_num()**, qui renvoie le numéro d'un processus léger dans un bloc parallèle.

Ces fonctions sont déclarées dans le fichier **omp.h** qu'il faudra bien évidemment inclure pour pouvoir les utiliser.

# Quelques schémas de construction bien pratiques

Nous allons revenir au premier algorithme permettant de calculer la moyenne de 20 valeurs, et bien évidemment ne pas commencer par se restreindre à exactement 20 valeurs.

Le programme prendra donc comme valeurs les arguments qui lui seront passés, les stockera dans un vecteur, calculera la moyenne puis l'affichera sur la sortie standard.

On commence par inclure les déclarations qui nous seront utiles :

```
#include <iostream>
#include <cstdlib> // Pour atof()
#include <libgen.h> // Pour basename()
#include <vector>
#include <omp.h>
```

On déclare utiliser l'espace de nommage standard (histoire de ne pas préfixer par **std::** ce que l'on utilisera de cet espace) :

```
using namespace std;
```

La fonction moyenne permet de prendre un vecteur de réels double précision (paramètre passé en référence constante car on travaillera sur l'original plutôt qu'une copie et qu'on ne modifiera en aucun cas les valeurs de ce vecteur). On récupère la taille du vecteur dans une variable **n**, on initialise la somme à **0** dans la variable **res**, puis on lance le calcul de la somme en appliquant le schéma de conception de boucle pour paralléliser avec réduction de l'opérateur **+** sur la variable **res**. Dit autrement, chaque itération sera traitée par un processus léger, qui travaillera sur une copie de la variable **res** (qui vaut donc **0**), puis les copies seront sommées par réduction (sommes partielles et récurrences).

```
double moyenne(const vector<double> &v) {
  size_t n = v.size();
  double res = 0;
#ifdef _OPENMP
#pragma omp parallel for reduction(+: res)
#endif
  for (size_t i = 0; i < n; ++i) {
    res += v[i];
  }
  return res / n;
}
```

Enfin le programme principal qui s'assure dans un premier temps d'avoir au moins un argument (c'est toujours mieux pour calculer une moyenne). On crée un vecteur de réels **v** de la taille du nombre d'arguments passés en option (le **-1** provient du fait que le nom du programme n'est pas à prendre en compte dans le décompte).

La lecture des arguments est faire en parallèle, mais on limitera le nombre de processus simultanés à **3** (clause **num_threads(3)**). On en profite pour ajouter un peu d'infos de *debuggage* qui nous permettent d'utiliser deux des fonctions présentées précédemment.

```
int main(int argc, char** argv) {
  if (argc < 2) {
    cerr << "usage: " << basename(argv[0]) << " <v_1> ... <v_n>" << endl
         << "Le programme calcule la moyenne des valeurs passées en argument"
         << endl;
    return 1;
  }

  vector<double> v(argc - 1);

#ifdef _OPENMP
```

```
#pragma omp parallel for num_threads(3)
#endif
  for (int i = 1; i < argc; ++i) {
    v[i - 1] = atof(argv[i]);
#ifdef _OPENMP
#pragma omp critical
    cout << "P[" << omp_get_thread_num()
         << "/" << omp_get_num_threads() << "]"
         << " argv[" << i << "]"
         << " = " << v[i - 1]
         << endl;
#endif
  }

  cout << "La moyenne des " << v.size()
       << " valeurs fournies en arguments est " << moyenne(v)
       << endl;

  return 0;
}
```

Pour changer, on compile et on teste avec quelques valeurs :

```
$ g++ -Wall -ansi -pedantic -fopenmp moyenne.cpp -o moyenne
$ ./moyenne 1 2 3 4 5 6 7 8
P[1/3] argv[4] = 4
P[0/3] argv[1] = 1
P[1/3] argv[5] = 5
P[2/3] argv[7] = 7
P[0/3] argv[2] = 2
P[1/3] argv[6] = 6
P[0/3] argv[3] = 3
P[2/3] argv[8] = 8
La moyenne des 8 valeurs fournies en arguments est 4.5
```

C'est cohérent.

## Aller plus loin...

Comme vous l'aurez compris, une directive OpenMP s'écrit sous forme d'un *pragma* juste avant l'instruction concernée. Si on souhaite faire porter une directive sur un bloc d'instruction, il suffit d'encapsuler le bloc d'instruction entre accolades.

L'objet de cet article n'est pas de détailler l'API complète d'OpenMP, mais d'illustrer son utilisation. Si vous avez compris le principe de fonctionnement, il ne vous reste qu'à lire la « *summary card* » disponible sur le site d'OpenMP **[4]** pour avoir la liste exhaustive des directives, schémas de construction et fonctions disponibles.

# MPI

Contrairement à la parallélisation légère, la parallélisation lourde implique surtout de prévoir un protocole de communication entre les différents processus. C'est ce que propose le standard MPI (*Message Passing Interface*) **[6]**.

## Open MPI

Il existe plusieurs implémentations du standard MPI. Comme vous l'aurez compris, Open MPI **[7]** est une implémentation libre et open source de ce protocole. Il en résulte qu'en théorie, je parlerai tout autant de MPI que d'Open MPI (et que de toute autre implémentation) dans la suite.

Écrire un programme avec Open MPI se résume à trois grandes étapes :

1. Initialisation de la communication entre les processus ;
2. Instructions à exécuter ;
3. Terminer la communication entre les processus.

Au sein d'un groupe de communication, certaines fonctions de MPI nécessitent que tous les processus du groupe communiquent ensemble. Ces routines sont dites **collectives**. *A contrario*, certaines fonctions peuvent s'exécuter indépendamment de tout ou partie des autres processus. Ces routines sont alors dites **non-collectives**.

*/// Note ///*

Par défaut, MPI définit un groupe de communication universel regroupant tous les processus. Il est possible de créer d'autres communicateurs et ainsi de définir des sous-groupes de processus. C'est pourquoi MPI distingue les communications dites intra-communicantes (au sein d'un même groupe) et des communications dites inter-communicantes (entre des groupes distincts). Cette subtilité ne sera pas développée dans cet article, c'était juste pour info.

*/// Fin Note ///*

Une des difficultés existante que l'on rencontre fréquemment lorsque l'on établit des communications entre processus, c'est le blocage de la communication. Cela peut se produire si deux (ou plusieurs) processus parlent en même temps. MPI définit deux catégories de communications. Les communications **synchrones** (*i.e.*, si Mickey parle à Dingo – Pluto c'est son chien – alors Dingo doit écouter Mickey, et seulement ensuite Mickey peut s'occuper de Pluto) des communications **asynchrones** (*i.e.*, si Mickey parle à Dingo, Dingo pourra écouter son message plus tard, cela n'empêche pas Mickey d'aller plus tôt s'occuper de Pluto).

## La documentation

C'est ici que les choses se compliquent. Jusqu'à présent, je n'ai jamais trouvé de tutoriel satisfaisant sur MPI (sauf pour dire bonjour au monde et calculer des moyennes, ce qui n'est quand même pas très passionnant – sauf quand c'est vraiment bien écrit ;-) ) et la documentation disponible sur le site d'Open MPI **[7]** manque clairement d'exemples (on dirait mes documentations de code – et je documente le minimum syndical). Donc il reste surtout à tester, coder, *debugger* et re-tester.

## Installation

Pour les moyennement chanceux qui utilisent une distribution Ubuntu, rien de plus simple pour avoir une version relativement obsolète :

```
$ sudo apt install libopenmpi-dev openmpi-bin openmpi-doc
```

Ceux qui sont sous Debian (sid) auront la chance d'avoir la dernière version. Ceci étant, à partir de la version 1.8.0 d'Open MPI, tout ce dont nous auront besoin pour cet article est fonctionnel. Au moment de la rédaction de cet article, j'utilise la version 1.10.2 (*package* officiel de la distribution Ubuntu 16.04). Rien n'interdit bien évidemment de télécharger les sources, de les compiler et de les installer.

Pour les autres distributions, il faudra chercher sur les forums dédiés à vos distributions.

Si vous envisagez d'utiliser un autre compilateur, alors il faudra également chercher sur les forums dédiés.

À noter également que depuis la version 2.0.0, les numéros majeurs et mineurs de version d'Open MPI correspondent aux spécifications de MPI. Enfin, les liaisons syntaxiques du C++ (*bindings*) sont déconseillés depuis la version 2.2 de la spécification (c'est dommage, j'aime bien ce sucre syntaxique) et ne seront plus implémentés pour les nouvelles fonctionnalités. Dans cet article, les codes Open MPI fournis utiliseront la syntaxe C++ quand celle-ci est disponible. Si vous installez vous-même Open MPI à partir des sources (récentes), pensez à passer l'option **`--enable-mpi-cxx`** au script **`./configure`**.

## L'exemple qui ne sert à rien, mais qui sert quand même

Un bel exemple qui ne sert à rien : lancer des processus qui disent simplement bonjour.

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

using namespace std;

int main(int argc, char** argv) {

  // Initialisation de la communication Open-MPI
  MPI::Init();
  // ou MPI::Init(argc, argv);

  int rank = MPI::COMM_WORLD.Get_rank();
  int nb_process = MPI::COMM_WORLD.Get_size();

  char proc_name[MPI_MAX_PROCESSOR_NAME];
  int proc_name_len;
  MPI::Get_processor_name(proc_name, proc_name_len);

  char lib_version[MPI_MAX_LIBRARY_VERSION_STRING];
```

```
  int lib_version_len;
  MPI_Get_library_version(lib_version, &lib_version_len);

  int major, minor;
  MPI::Get_version(major, minor);

  cout << "Le processus " << (rank + 1) << "/" << nb_process
       << " qui tourne sur la machine " << proc_name
       << " et utilise la librairie " << lib_version
       << " implémentant le standard " << major << "." << minor
       << " vous dit BONJOUR !"
       << endl;

  // Fin de la communication Open-MPI
  MPI::Finalize();

  return 0;
}
```

La routine **MPI::Init()** permet d'initialiser la communication entre les processus (c'est typiquement une routine collective). Cela permet notamment de créer un communicateur universel **MPI::COMM_WORLD** qui permettra à tous les processus de communiquer. Chaque processus peut récupérer son rang (méthode **Get_rank()**) au sein d'un communicateur (entre **0** et **n-1** pour **n** processus), ainsi que le nombre total de processus de ce communicateur (méthode **Get_size()**).

Chaque processus peut également récupérer le nom de la machine sur laquelle il est exécuté (**MPI::Get_processor_name()**), la version de la librairie Open MPI installée (**MPI_Get_library_version()**) ou encore la version du standard MPI que cette librairie implémente (**MPI::Get_version()**).

Enfin, la fin de la communication est actée par la routine (**MPI::Finalize()**).

La compilation d'un programme utilisant Open MPI doit être effectuée via un compilateur dédié (en réalité, c'est une surcouche de votre compilateur préféré) : **mpic++** alias **mpicxx** (pour le C++) ou **mpicc** (pour le C). Ce compilateur exploite pleinement les variables d'environnement usuelles **CXX**/**CC** pour le compilateur à utiliser en interne.

Donc pour compiler notre programme :

```
$ mpic++ -Wall -ansi -pedantic -std=c++11 helloworld-mpi.cpp -o helloworld-mpi
```

Si on l'exécute (un peu bêtement), on obtient une instance qui affiche les informations demandées :

```
$ ./helloworld-mpi
Le processus 1/1 qui tourne sur la machine spartacus et utilise la librairie Open MPI v1.10.2,
package: Open MPI buildd@lgw01-57 Distribution, ident: 1.10.2, repo rev: v1.10.1-145-g799148f, Jan
21, 2016 implémentant le standard 3.0 vous dit BONJOUR !
```

Et oui, ma machine s'appelle **spartacus** (parce que c'est une bête de compét'), mais ce n'était pas ça la bonne question. La bonne question était : mais comment lance-t-on plusieurs processus ? Pour cela, il faut utiliser le programme **mpirun**, dont la syntaxe est **mpirun [options mpirun] <programme> [args prog]**.

```
$ mpirun -np 2 ./helloworld-mpi
Le processus 2/2 qui tourne sur la machine spartacus et utilise la librairie Open MPI v1.10.2,
package: Open MPI buildd@lgw01-57 Distribution, ident: 1.10.2, repo rev: v1.10.1-145-g799148f, Jan
21, 2016 implémentant le standard 3.0 vous dit BONJOUR !
Le processus 1/2 qui tourne sur la machine spartacus et utilise la librairie Open MPI v1.10.2,
package: Open MPI buildd@lgw01-57 Distribution, ident: 1.10.2, repo rev: v1.10.1-145-g799148f, Jan
21, 2016 implémentant le standard 3.0 vous dit BONJOUR !
```

L'option **-np** permet de spécifier le nombre de processus à lancer. Il est également possible de spécifier plusieurs hôtes possibles (machines devant être accessibles en réseau et disposer du même programme – typiquement sur un lecteur réseau) avec l'option **-host**, ou encore de fournir un fichier contenant ces informations, et bien d'autres possibilités de configuration encore.

C'est bien beau tout ça, mais si on faisait des processus qui communiquent, ce serait mieux…

Donc je propose d'attaquer un autre exemple qui ne sert à rien : calculer (inefficacement) la moyenne d'une série de notes !

Pour cela, détaillons l'écriture du fichier **moyenne-mpi.cpp**, qui commencera comme il se doit par quelques inclusions :

```
01: #include <mpi.h>
02: #include <iostream>
03: #include <cstdlib> // Pour atof() et atexit()
04: #include <libgen.h> // Pour basename()
05: #include <vector>
```

La première ligne permet d'inclure toute l'API Open MPI. La seconde nous permettra de faire des entrées/sorties (donc ici surtout des sorties). Les troisième et quatrième lignes serviront respectivement à

transformer les arguments (chaînes de caractère) en réels, enregistrer des fonctions à exécuter à la fin du programme et à récupérer le nom du programme (un petit **man 3 atof** / **man 3 atexit** / **man 3 basename** devrait suffire à comprendre l'utilisation de ces fonctions). Enfin, nous allons stocker (choix plus que discutable) les valeurs passées en argument dans des vecteurs.

Afin de ne pas nous embêter à préfixer les objets/fonctions de l'espace de nom **std**, nous allons expliquer que nous utiliserons cet espace de nom :

```
07: using namespace std;
```

Pour calculer la moyenne, il faut d'abord calculer la somme d'une série de valeur :

```
09: double somme(const vector<double> &v) {
10:
11:   if (!MPI::Is_initialized()) {
12:     throw "MPI n'est pas initialisé";
13:   }
14:   size_t n = v.size();
15:   double res = 0;
16:   for (size_t i = 0; i < n; ++i) {
17:     res += v[i];
18:   }
19:   return res;
20: }
```

La fonction **somme()** prend en entrée une référence constante sur un vecteur de réels double précision (ainsi le vecteur en entrée ne sera pas recopié et nous garantissons qu'il ne sera pas modifié).

Au passage, nous nous assurerons que la fonction **somme()** est appelée après que l'environnement MPI a été initialisé (vérification à la ligne 11 ; si le test échoue – donc que l'environnement n'est pas initialisé – alors on lance une exception (ligne 12)). Le reste du code est somme toutes assez classique.

Je profite de cet article pour présenter une astuce permettant de finaliser proprement les communications MPI quelle que soit la raison mettant fin aux instances. Pour cela, il suffit de définir une fonction ne prenant aucun paramètre et ne renvoyant rien, dont le corps contient le code qui devra être exécuté à la fin de l'exécution du programme :

```
22: void mpi_ending() {
23:   // Fin de la communication Open-MPI
24:   MPI::Finalize();
25: }
```

Puis, après avoir initialisé la communication Open MPI, il suffit d'empiler cette fonction dans les fonctions à exécuter à la fin du programme.

```
27: int main(int argc, char** argv) {
28:
29:   // Initialisation de la communication Open MPI
30:   MPI::Init();
31:   atexit(mpi_ending);
```

Comme dans le précédent exemple, il est de bon ton de récupérer le rang du processus courant et le nombre total de processus qui tournent en parallèle :

```
33:   int rank = MPI::COMM_WORLD.Get_rank();
34:   int nb_process = MPI::COMM_WORLD.Get_size();
```

Nous nous assurons que le programme (ou plutôt les programmes) ont été appelés avec au moins un argument, ce qui est toujours mieux pour calculer une moyenne. Dans le cas contraire, seul le processus ayant le rang **0** (et celui-ci existe nécessairement) affichera un message d'usage sur la sortie d'erreur. Le programme renverra **1** au système pour signifier un fonctionnement anormal et du fait de l'utilisation de la fonction **atexit()** à la ligne 31, nous avons la garantie que tous les processus termineront correctement la session MPI.

```
36:   if (argc < 2) {
37:     if (!rank) {
38:       cerr << "usage: " << basename(argv[0]) << " <v_1> ... <v_n>" << endl
39:            << "Le programme calcule la moyenne des valeurs passées en argument"
40:            << endl;
41:       return 1;
42:     }
43:   }
```

Si le programme arrive jusqu'ici, c'est qu'au moins un argument a été saisi. Nous allons donc commencer à identifier quel processus travaillera sur quelles données… Nous savons que le programme a été lancé avec **argc** arguments, incluant le nom de l'exécutable. Ainsi le nombre de valeurs à traiter est égal à (**argc – 1**). Nous savons également qu'il y a **nb_process** en cours d'exécution pour traiter ces valeurs. Nous allons donc répartir ces valeurs équitablement sur chacun des processus. Chaque processus aura au moins **q** (= **(argc – 1) / nb_process**) valeurs à traiter. Il reste **r** (= **(argc – 1) % nb_process**) à répartir sur les processus (calculs des lignes 45 et 46). Arithmétiquement, **r** est strictement inférieur à **nb_process**. Aussi,

chaque processus ayant un rang (**rank**) strictement inférieur à **r** traitera **q + 1** valeurs et chaque processus ayant un rang supérieur ou égal à **r** traitera **q** valeurs (calcul de la valeur **vec_size** à la ligne 47). Il reste à calculer l'indice de la première valeur à traiter par le processus au rang **rank**. Si **rank** est strictement inférieur à **r**, alors les processus précédents traiteront **(q + 1) * rank** valeurs. Si **rank** est supérieur ou égal à **r**, alors les processus précédents traiteront **(q * rank + r)** valeurs. Ces calculs se factorisent assez facilement à la ligne 48. Nous en profitons pour afficher une sortie de *debuggage* indiquant l'intervalle des indices traités par le processus en cours (lignes 50 à 53).

```
45:   int q = (argc - 1) / nb_process;
46:   int r = (argc - 1) % nb_process;
47:   size_t vec_size = q + (rank < r);
48:   size_t vec_start = (q * rank) + ((rank < r) ? rank : r);
49:
50:   cout << "P" << rank << ": "
51:        << "[" << vec_start
52:        << ", " << (vec_start + vec_size - 1) << "]"
53:        << endl;
```

Maintenant que chaque processus connaît son nombre de valeurs à traiter ainsi que l'indice de la première valeur à traiter, nous construisons un vecteur de la bonne dimension (ligne 55) que nous initialisons en convertissant les arguments de la ligne de commande (chaînes de caractère C) en réels (lignes 57 à 59).

```
55:   vector<double> v(vec_size);
56:
57:   for (size_t i = 0; i < vec_size; ++i) {
58:     v[i] = atof(argv[i + vec_start]);
59:   }
```

Chaque processus est ainsi capable de calculer la somme des valeurs dont il assure le traitement dans la variable **m**. Nous aurons également besoin d'une variable **m2** pour calculer la réduction de la somme à travers tous les processus (l'initialisation à **0** n'est pas une nécessité dans l'absolu, mais cela fait partie de mes bonnes pratiques de programmation).

```
61:   double m = somme(v), m2 = 0;
```

La réduction passe simplement par l'appel (ligne 63) de la méthode **Reduce** de l'objet **MPI::COMM_WORLD** (qui est le communicateur universel). Le premier argument est l'adresse de la zone mémoire contenant les valeurs initiales. Le second argument est l'adresse de la zone mémoire qui contiendra le résultat de la réduction. Le troisième argument définit le nombre de valeurs initiales fournies par le processus (donc ici, nous n'avons qu'une seule valeur) et le quatrième argument définit le type de valeurs à réduire (ici, le type double (**MPI_DOUBLE)** – voir encadré sur les types de données échangeables entre les processus MPI). Le cinquième argument définit la réduction à appliquer (ici, une somme (**MPI_SUM**) – voir encadré sur les types de réductions proposées entre les processus MPI). Enfin le dernier argument spécifie le rang du processus qui récupérera le résultat de l'opération de réduction (ici le processus **0** – qui existe nécessairement).

*/// Début encadré ///*

# Les type de données échangeables entre les processus MPI

Open MPI permet d'échanger des données de n'importe quel type, y compris des structures composites. Les types primitifs sont pré-existants (**MPI_INT**, **MPI_FLOAT**, **MPI_UNSIGNED_SHORT**, **MPI_BOOL**, …). Les types de données sont de type **MPI::Datatype** (et non les types natifs du langage) et Open MPI propose des fonctions (méthodes statiques de la classe **MPI::Datatype** dans le cas de l'API C++) permettant de définir n'importe quel type de données. Il existe également des types correspondant à des paires de types primitifs tels que **MPI_FLOAT_INT**, **MPI_DOUBLE_INT**, **MPI_2INT**, … servant notamment aux opérations de réduction **MPI_MAXLOC** et **MPI_MINLOC** (voir note sur les réductions proposées entre les processus MPI).

*/// Fin encadré///*

*/// Début encadré ///*

# Les type de réductions proposées entre les processus MPI

Open MPI permet d'effectuer plusieurs opérations de réduction. Toutes ne sont pas valides selon le type des données échangées.

| Nom MPI | Signification |
|---|---|
| **MPI_MAX** | Maximum |
| **MPI_MIN** | Minimum |

| | |
|---|---|
| **MPI_SUM** | Somme |
| **MPI_PROD** | Produit |
| **MPI_LAND** | Et logique |
| **MPI_BAND** | Et binaire |
| **MPI_LOR** | Ou logique |
| **MPI_BOR** | Ou binaire |
| **MPI_LXOR** | Ou exclusif logique |
| **MPI_BXOR** | Ou exclusif binaire |
| **MPI_MAXLOC** | Valeur maximale et position globale parmi tous les processus |
| **MPI_MINLOC** | Valeur minimale et position globale parmi tous les processus |

Open MPI permet de définir nos propres opérations de réduction. Le cas échéant, ces opérations sont présumées associatives. Il est possible de spécifier si ces opérations sont également commutatives, dans le cas contraire, ces opérations sont appliquées par ordre croissant de processus.

*/// Fin encadré ///*

Enfin, toujours à des fins de *debuggage*, nous afficherons les valeurs calculées/récupérées dans les variables **m** et **m2** pour le processus courant.

```
63:   MPI::COMM_WORLD.Reduce(&m, &m2, 1, MPI_DOUBLE, MPI_SUM, 0);
64:
65:   cout << "P" << rank << ": "
66:        << "somme locale " << m << ", "
67:        << "somme globale " << m2
68:        << endl;
```

Il est à noter que chaque processus travaille à son rythme. Toutefois, les opérations collectives (**Init**, **Reduce**, **Finalize**, …) imposent aux processus de s'attendre mutuellement. Toutefois, avant d'afficher le résultat final du calcul de la moyenne, il est préférable de s'assurer que tous les processus ont fini d'écrire les informations de *debuggage* (lignes 65 à 68). Pour cela, il suffit de poser une barrière (ligne 70), c'est-à-dire une opération collective qui ne fait rien de plus que d'empêcher un processus de poursuivre tant que tous les autres processus ne sont pas au même point du programme.

```
70:   MPI::COMM_WORLD.Barrier();
```

L'usage de barrières est pratique, puisqu'il permet de synchroniser l'exécution des processus, toutefois – et l'analogie avec la barrière est très pertinente – cela les ralentis également. Ainsi, mettre une barrière juste avant l'opération de réduction ne présente aucun intérêt puisque la réduction elle-même fait office de barrière. De même, si le code de *debuggage* (lignes 65 à 68) est à terme supprimé, alors la barrière de la ligne 70 ne présente plus aucune utilité non plus.

Revenons au calcul de la moyenne. Suite à l'opération de réduction, seul le processus de rang **0** connaît le résultat de la réduction (on aurait pu utiliser la méthode **Allreduce** qui distribue le résultat à tous les processus, mais dans le cas présent, cela n'aurait rien apporté). Seul le processus de rang **0** (ligne 72) va donc diviser la somme globale par le nombre de valeurs (ligne 73 ; je rappelle à toutes fins utiles que nous avons garanti que **argc** est supérieur ou égal à **2** – ligne 36 – et donc que **(argc – 1)** est non nul). Enfin, ce processus va afficher la moyenne sur la sortie standard.

```
72:   if (!rank) {
73:     m2 /= (argc - 1);
74:     cout << "Moyenne calculée : " << m2 << endl;
75:   }
```

Il reste à remercier une fois de plus la fonction **atexit** de faire le travail de finalisation de la communication MPI tout en renvoyant **0** au système d'exploitation pour lui signifier que le programme s'est correctement exécuté.

```
77:   return 0;
78: }
```

Compilons maintenant ce programme :

```
$ mpic++ -Wall -ansi -pedantic -std=c++11 moyenne-mpi.cpp -o moyenne-mpi
```

Et exécutons-le d'abord sans MPI :

```
$ ./moyenne-mpi 1 2 3 4 5 6 7 8
P0: [0, 7]
P0: somme locale 28, somme globale 28
```

```
Moyenne calculée : 3.5
```

Cela semble fonctionner (c'est heureux).

Essayons maintenant dans l'environnement MPI, avec un seul processus :

```
$ mpirun -np 1 ./moyenne-mpi 1 2 3 4 5 6 7 8
P0: [0, 7]
P0: somme locale 28, somme globale 28
Moyenne calculée : 3.5
```

On obtient le même résultat, ce qui est plutôt rassurant.

Augmentons le nombre de processus, si possible avec une valeur qui ne soit pas un diviseur de **8** pour éviter de tomber dans un cas trop particulier :

```
$ mpirun -np 3 ./moyenne-mpi 1 2 3 4 5 6 7 8
P0: [0, 2]
P1: [3, 5]
P1: somme locale 12, somme globale 0
P2: [6, 7]
P2: somme locale 13, somme globale 0
P0: somme locale 3, somme globale 28
Moyenne calculée : 3.5
```

On peut vérifier que nos 8 valeurs sont correctement réparties sur les 3 instances. On s'aperçoit également que seul le processus de rang **0** connaît la valeur de **m2** (tous les autres n'ont pas modifié la valeur par défaut de **m2** – quelle bonne habitude de programmation que de systématiquement initialiser ses variables ;-) ).

La question que nous avons jusqu'alors passé sous silence est de savoir ce qu'il se passe si nous exécutons notre programme avec plus d'instances que de valeurs. Eh bien, théoriquement ça fonctionne, puisque dans ce cas, les premiers processus auront une seule valeur à traiter et les suivants n'en auront pas. Vérifions rapidement :

```
$ mpirun -np 10 ./moyenne-mpi 1 2 3 4 5 6 7 8
P3: [3, 3]
P4: [4, 4]
P0: [0, 0]
P1: [1, 1]
P1: somme locale 1, somme globale 0
P2: [2, 2]
P3: somme locale 3, somme globale 0
P5: [5, 5]
P2: somme locale 2, somme globale 0
P5: somme locale 5, somme globale 0
P8: [8, 7]
P9: [8, 7]
P9: somme locale 0, somme globale 0
P8: somme locale 0, somme globale 0
P7: [7, 7]
P7: somme locale 7, somme globale 0
P6: [6, 6]
P6: somme locale 6, somme globale 0
P4: somme locale 4, somme globale 0
P0: somme locale 0, somme globale 28
Moyenne calculée : 3.5
```

Cela fonctionne encore. C'était un petit pas dans le calcul de la moyenne, mais un grand pas dans le calcul distribué.

# OpenMP ou Open MPI ?

La question qui pourrait se poser maintenant serait de savoir s'il est préférable d'utiliser OpenMP plutôt qu'Open MPI, ou inversement. Je pourrais faire une réponse de normand en expliquant que cela dépend du contexte, et c'est finalement la seule véritable bonne réponse, mais dans le cas présent, l'objectif est d'illustrer les deux paradigmes. Par conséquent, la réponse sera pour cet article de ne pas les opposer, mais de les utiliser conjointement (le « ou » n'étant pas exclusif en français, le titre de cette section n'est donc finalement pas fallacieux).

## Parallélisme hybride

Pour cela, nous allons reprendre les codes précédents et les fusionner allégrement sans se poser trop de questions.

Plutôt que de détailler le code, je commenterai le fichier de patch créé à partir de la commande :

```
$ diff --unified moyenne-mpi.cpp moyenne-hybride.cpp > mpi2hybride.patch
```

Pour ceux qui ne sont pas familiers de cette commande (pourtant bien pratique), le premier élément composant la sortie du **diff** (avec l'option **--unified**) est l'indication des fichiers concernés par le différentiel ainsi que leurs dates respectives de dernière modification. Le premier fichier est préfixé de **---** et le second de **+++**. Dans la suite, chaque ligne présente uniquement dans le premier fichier sera préfixée d'un **-** et chaque ligne présente uniquement dans le second fichier sera préfixée d'un **+**. Autour de chaque différence, un contexte de 3 lignes identiques est également affiché, les lignes sont alors préfixées d'un espace.

```
--- moyenne-mpi.cpp    2018-07-25 00:12:25.242172198 +0200
+++ moyenne-hybride.cpp       2018-07-25 22:27:42.278741734 +0200
```

Ensuite, chaque bloc de différence commence par un entête entre **@@**. Cet entête précise les numéros de la première ligne de chaque bloc et le nombre de lignes correspondant dans chaque fichier (le **-** et le **+**).

Le premier bloc de différence montre que j'ai ajouté l'inclusion du fichier **omp.h**.

```
@@ -3,6 +3,7 @@
 #include <cstdlib> // Pour atof() et atexit()
 #include <libgen.h> // Pour basename()
 #include <vector>
+#include <omp.h>

 using namespace std;
```

Le second bloc montre que la boucle permettant de calculer la somme des valeurs du vecteur **v** (dans la fonction **somme**) est parallélisée avec un schéma de réduction par OpenMP.

```
@@ -13,6 +14,9 @@
   }
   size_t n = v.size();
   double res = 0;
+#ifdef _OPENMP
+#pragma omp parallel for reduction(+: res)
+#endif
   for (size_t i = 0; i < n; ++i) {
     res += v[i];
   }
```

Le troisième et dernier bloc montre que l'initialisation du vecteur **v** (dans la fonction **main**) est parallélisée avec 3 processus légers et qu'une information de *debuggage* est affichée.

```
@@ -54,8 +58,19 @@

   vector<double> v(vec_size);

+#ifdef _OPENMP
+#pragma omp parallel for num_threads(3)
+#endif
   for (size_t i = 0; i < vec_size; ++i) {
     v[i] = atof(argv[i + vec_start]);
+#ifdef _OPENMP
+#pragma omp critical
+    cout << "P" << rank << "[" << omp_get_thread_num() + 1
+         << "/" << omp_get_num_threads() << "]: "
+         << "v[" << i << "] = " << v[i]
+         << " = argv[" << i  + vec_start << "]"
+         << endl;
+#endif
   }

   double m = somme(v), m2 = 0;
```

Outre une lecture centrée sur les différences entre les deux fichiers, ce fichier peut être utilisé par la commande **patch** pour créer le fichier **moyenne-hybride.cpp** à partir du fichier **moyenne-mpi.cpp**.

```
$ patch --input mpi2hybride.patch --output moyenne-hybride.cpp
patching file moyenne-hybride.cpp (read from moyenne-mpi.cpp)
```

Il reste à compiler notre nouveau programme (sans oublier l'option **-fopenmp**) :

```
$ mpic++ -Wall -ansi -pedantic -std=c++11 moyenne-hybride.cpp -fopenmp -o moyenne-hybride
```

Puis à exécuter notre programme (sans **mpirun**) :

```
$ ./moyenne-hybride 1 2 3 4 5 6 7 8
P0: [0, 7]
P0[3/3]: v[6] = 6 = argv[6]
P0[1/3]: v[0] = 0 = argv[0]
P0[2/3]: v[3] = 3 = argv[3]
P0[3/3]: v[7] = 7 = argv[7]
P0[1/3]: v[1] = 1 = argv[1]
```

```
P0[2/3]: v[4] = 4 = argv[4]
P0[1/3]: v[2] = 2 = argv[2]
P0[2/3]: v[5] = 5 = argv[5]
P0: somme locale 28, somme globale 28
Moyenne calculée : 3.5
```

Puis avec **mpirun** :

```
$ mpirun -n 3 ./moyenne-hybride 1 2 3 4 5 6 7 8
P0: [0, 2]
P1: [3, 5]
P2: [6, 7]
P1[1/3]: v[0] = 3 = argv[3]
P1[2/3]: v[1] = 4 = argv[4]
P1[3/3]: v[2] = 5 = argv[5]
P0[3/3]: v[2] = 2 = argv[2]
P0[2/3]: v[1] = 1 = argv[1]
P0[1/3]: v[0] = 0 = argv[0]
P2[2/3]: v[1] = 7 = argv[7]
P2[1/3]: v[0] = 6 = argv[6]
P1: somme locale 12, somme globale 0
P2: somme locale 13, somme globale 0
P0: somme locale 3, somme globale 28
Moyenne calculée : 3.5
```

*That's (presque) All, Folks !!!*

# Et dans la vraie vie ?

Quelle question ! Vous lisez GLMF, vous êtes donc dans la vraie vie… En tout cas, c'est ce qu'aurait pu dire un certain Blaise P. s'il avait vécu à notre époque. Toutefois, j'ai volontairement omis une complication majeure liée à la programmation hybride avec OpenMP et Open MPI.

En effet, il n'y a pas de soucis tant que la parallélisation légère n'émet ni ne reçoit de messages MPI. Là où cela se complique, c'est effectivement lorsqu'au moins un parmi plusieurs processus légers d'un même processus lourd doit communiquer dans l'univers MPI. Dans ce cas, l'initialisation MPI doit être effectuée avec **MPI::Init_thread()** au lieu de **MPI::Init()**. La méthode statique **MPI::Init_thread()** – à l'instar de la méthode statique **MPI::Init()** – existe en deux versions, l'une qui prend en paramètres les arguments **argc**, **argv** de la fonction **main** (et ne s'en sert absolument pas) et une qui ne les prends pas. Cependant, contrairement à la méthode statique **MPI::Init()**, la méthode statique **MPI::Init_thread()** requiert un argument supplémentaire qui correspond au niveau de parallélisme léger supporté. Les quatre niveaux proposés sont représentés par quatre constantes :

| | |
|---|---|
| **MPI_THREAD_SINGLE** | Un seul processus léger est exécuté (donc pas de parallélisme léger – *a minima* lors des communications MPI). |
| **MPI_THREAD_FUNNELED** | Seul le processus léger qui a appelé la méthode **MPI::Init_thread()** communiquera dans l'univers MPI. |
| **MPI_THREAD_SERIALIZED** | Lorsque plusieurs processus légers sont en cours d'éxécution, un seul parmi eux communiquera dans l'univers MPI. |
| **MPI_THREAD_MULTIPLE** | Tous les processus légers peuvent communiquer parallèlement dans l'univers MPI. |

Bien évidemment, plus le niveau autorise des opérations de communication complexes, plus cela impacte les performances.

Comme vous l'aurez compris, l'appel de **MPI::Init()** équivaut à l'appel de **MPI::Init_thread()** au niveau **MPI_THREAD_SINGLE**.

Là où les choses se compliquent encore, c'est que toutes les implémentations de MPI ne gèrent pas nécessairement le 4$^{e}$ niveau (qui crée des complications qui ne sont pas nécessairement encore très bien gérées). Le cas échéant, l'initialisation se fera au 3$^{e}$ niveau.

Pour savoir si votre implémentation d'Open MPI supporte le niveau **MPI_THREAD_MULTIPLE**, il suffit de le lui demander :

```
$ ompi_info | grep -i thread
          Thread support: posix (MPI_THREAD_MULTIPLE: no, OPAL support: yes, OMPI progress:
no, ORTE progress: yes, Event lib: yes)
   FT Checkpoint support: no (checkpoint thread: no)
```

```
$
```

Comme vous pouvez le constater, chez moi, je n'ai pas cette chance…

Mais vous imaginez bien que le programme doit pouvoir détecter si le niveau demandé est supporté pour éventuellement afficher un message d'avertissement, dérouler une stratégie alternative, ou tout simplement quitter proprement. C'est pour cette raison que la méthode statique **`MPI::Init_thread()`** retourne la valeur du niveau qui a été réellement utilisé par l'initialisation.

Ainsi, un code judicieux dans ce cas serait :

```
if (MPI::Init_thread(MPI_THREAD_SERIALIZED) != MPI_THREAD_SERIALIZED) {
  cerr << "Je ne peux pas continuer à travailler dans de telles conditions." << endl;
  MPI::COMM_WORLD.Abort(EXIT_FAILURE);
}
```

Notez au passage que cet exemple m'a permis de subrepticement glisser la méthode **`Abort()`** du communicateur **`MPI::COMM_World`** dont l'intérêt réside dans l'arrêt de tous les processus, y compris de ceux qui travaillaient tranquillement sans rencontrer de problème particulier.

# Alternatives

Bien évidemment, certains langages intègrent plus ou moins implicitement et naturellement les paradigmes de parallélisme, tels que **NodeJS** ou **Go**. Comme toujours, le choix du langage doit dépendre des objectifs et non de notre champ de compétence ou de notre affinité. Ce n'est pas parce que je n'aime pas le **Java** que je m'interdis de programmer dans ce langage (bon, c'est un mauvais exemple parce qu'il y a toujours une alternative préférable Java :-p ).

Ceci étant, si le problème que l'on souhaite résoudre est simple et ne nécessite que peu de ressources matérielles, le choix reste assez libre. Dès lors que le volume de données à traiter devient colossal (aire haire hère ère du *BigData*) le besoin de performance devient primordial et le passage à des langages tels que le C++ voire le C (qui contrairement à ce que l'on peut lire parfois – y compris dans d'illustres revues comme GNU/Linux Mag, Linux Pratique HS, … – est un langage de haut niveau) sont une nécessité et les implémentations des standards OpenMP et Open MPI sont clairement opportunes.

# Références


**[1]** « *Qu'est-ce qu'un algorithme ?* » Philippe Flajolet et Étienne Parizot (**http://interstices.info/jcms/c_5776/qu-est-ce-qu-un-algorithme**)

**[2]** « *Comme d'habitude* », novembre 1967. Chanson écrite par Gilles Thibaut et Claude François, musique de Jacques Revaux et Claude François.

**[3]** **http://en.cppreference.com/**

**[4]** **http://www.openmp.org/**

**[5]** « *Bibbidi-Bobbidi-Boo* », 1950. Chanson écrite et composée par Mack David, Al Hoffman et Jerry Livingston1 pour le long-métrage Disney Cendrillon.

**[6]** **http://www.mpi-forum.org/**

**[7]** **http://www.open-mpi.org/**