TD4 - MPI en douceur

Introduction au Message Passing Interface

(https://www.lix.polytechnique.fr/~mohamed/)

Sameh Mohamed (https://www.lix.polytechnique.fr/~mohamed/)
X2015

Vous devrez déposer votre travail **au fur et à mesure**. Avant de pouvoir déposer, il faut s'authentifier avec son login et son mot de passe :

Login : ha-quang.le **Mot de passe :** •••••••••••••••• Connecter

Toggle Menu

Introduction au MPI - Message Passing Interface

Le MPI (The Message Passing Interface) (http://www.mpi-forum.org/), conçue en 1993-94, est une norme définissant une bibliothèque de fonctions, utilisable avec les langages C, C++ et Fortran. Elle permet d'exploiter des ordinateurs distants ou multiprocesseur par passage de messages. Elle est devenue de facto un standard de communication pour des nœuds exécutant des programmes parallèles sur des systèmes à mémoire distribuée. MPI a été écrite pour obtenir de bonnes performances aussi bien sur des machines massivement parallèles à mémoire partagée que sur des clusters d'ordinateurs hétérogènes à mémoire distribuée. Elle est disponible sur de très nombreux matériels et systèmes d'exploitation. Ainsi, MPI possède l'avantage par rapport aux plus vieilles bibliothèques de passage de messages d'être grandement portable (car MPI a été implémentée sur presque toutes les architectures de mémoires) et rapide (car chaque implémentation a été optimisée pour le matériel sur lequel il s'exécute). Depuis 1997, une nouvelle version de MPI est disponible, MPI-2, qui apporte quelques puissantes fonctionnalités supplémentaires.

Exercice 1 Hello (parallel) World

Voici le "Hello world" message (ou plutôt bienvenue en INF442) en version distribuée. Veuillez l'enregister dans un fichier nommé `PremierMPI.cpp` :

?

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

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

    int id, p, name_len;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    // Initialize MPI.
    MPI_Init(&argc, &argv);

    // Get the number of processes.
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    // Get the individual process ID.
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    std::cout << "Processeur " << processor_name << " ID = " << id << "

    // Terminate MPI.
    MPI_Finalize();

    return 0;

}
```

Compilez votre programme

```
mpic++ PremierMPI.cpp -o PremierMPI
```

Testez votre programme

Testez sur votre machine avec 4 processus : `mpirun -np 4 PremierMPI` .
Repetez l'execution plusieurs fois : on constate que l'ordre des salutations est susceptible à changer.

La chose à **absolument retenir** avant de sortir de ce TP est le fait que **l'ensemble du code entre les fonctions `MPI_Init` et `MPI_Finalize` sera lu par TOUS (!) les processus !!**

Rappelons cependant quelques autres concepts :

Communicateurs : Un communicateur désigne un ensemble de processus pouvant communiquer ensemble, et deux processus ne pourront communiquer que s'ils sont dans un même communicateur. Un communicateur initial englobe tous les processus (`MPI_COMM_WORLD`), qu'il est possible de subdiviser en communicateurs plus petits correspondants à des entités logiques. Il existe deux types de communicateurs : les intracommunicateurs et les intercommunicateurs. Les intracommunicateurs sont les communicateurs standards, alors que les intercommunicateurs servent à créer un pont entre deux intracommunicateurs.
Communications point-à-point: Les communications point-à-point permettent à deux processus à l'intérieur d'un même communicateur d'échanger une donnée (scalaire, tableau ou type dérivé). Les fonctions correspondantes sont `MPI_Send` , `MPI_Recv` et `MPI_Sendrecv` .
Communications collectives : Les communications collectives impliquent tous les processus d'un communicateur. Il est possible d'envoyer une même donnée à tous les processus (`MPI_Bcast`), de découper un tableau entre tous les processus (`MPI_Scatter`), ou d'effectuer une opération (par exemple addition) où chaque processus contribuera.

Voyons d'abord la **communications point-à-point** avec les fichiers suivants : `mpi_helloBsend.cpp` (code/mpi_helloBsend.cpp) et `mpi_helloNBsend.cpp` (code/mpi_helloNBsend.cpp). Compilez et exécutez plusieurs fois à l'aide des mêmes lignes de commandes que pour `PremierMPI.cpp`.

1. Que font ces codes ?
2. Voyez vous une différence entre les sorties de ces programmes ?
3. A présent, ouvrez les fichiers sources et cherchez à comprendre que font les différentes conditions (`if ... else`).
4. Quel est la différence fondamentale entre ces deux programmes ?

Déposez votre travail

Déposez votre fichier avec commentaires (pas de correction automatique) :

Le nom du fichier à déposer
Choose File No file chosen
Déposer
Il faut se connecter avant de pouvoir déposer

Toggle Menu

Exécution distribuée sur des machines designées

Maintenant choisissez deux machines `machine1` et `machine2` parmi les 169 PCs en salles machines (cf. cette liste des machines (sallesmachines.txt)).
Tout d'abord, se logguer sur ces deux machines en faisant au préalable `kinit` (Kerberos) sur votre console :

```
1 kinit
2 ssh machine1
3 ssh machine2
```

Puis vérifier que vous n'avez plus besoin de rentrer votre MDP (Mot de Passe) :

```
1 ssh machine1
2 exit
```

On peut alors exécuter notre premier programme sur ces deux machines avec cette syntaxe (attention aux espaces) :

```
mpirun -np 5 -host machine1,machine2 PremierMPI
```

Bravo ! Vous venez d'exécuter votre premier programme MPI sur une architecture à mémoire distribuée.

Exécution distribuée aux ressources auto-allouées (SLURM)

Qu'est-ce que c'est le **SLURM** (http://slurm.schedmd.com/)? C'est un système pour la gestion des ressources dans les réseaux du calcul distribué.
Pour nous, les avantages sont évidents. Avec `mpirun -host machine1,...machineN ...` (comme ci-dessus) on ne peut que se croiser les doigts en espérant que tous nos collègues n'essaient pas d'utiliser les mêmes machines (ce qui peut épuiser vite nos ressources). Avec SLURM, le choix des machines est automatisé, et optimisé pour minimiser les conflits et gaspillages.

On préfère donc lancer nos programmes sous SLURM, en utilisant la commande `salloc` .

Pour lancer votre programme avec SLURM, entrez la commande suivante :

```
salloc -n 6 --ntasks-per-node=2 mpirun PremierMPI
```

Cette commande lance six processus (`-n 6`), deux sur chacun des trois noeuds (`--ntasks-per-node=2`), sur des noeuds sélectionnés automatiquement par SLURM.

Attention : Bien évidemment, nos ressources dans les salles infos sont limitées et partagées: chaque machine n'a que 4 processeurs, et il n'y a que 24 machines par salle. Soyez sage (et sympa avec vos collègues) en choisissant un nombre plutôt modeste de processus à lancer!

Exercice 2 Calcul de π

À présent, on va estimer π à partir de l'identité mathématique

$$\int_0^1 \frac{4}{(1+x^2)} dx = \pi$$

L'intégrale est approchée en la discrétisant en n intervalles par la somme de Riemann

$$\frac{1}{n}\sum_{i=0}^{n-1} f\left(\frac{i+1/2}{n}\right)$$

où $f(x) = \frac{4}{1+x^2}$. Le processus avec l'identifiant $0 \leq r \leq p-1$ va calculer la somme partielle avec les indices $i \equiv r \pmod{p}$. Il faut donc communiquer n à tous les processus avec `Bcast` puis une fois que les processus auront fini de calculer leur portion, il faut sommer tous les résultats avec `Reduce` .

`MPI_Bcast`

Rappelons l'opération `MPI_Bcast` (OpenMPI doc ici (https://www.open-mpi.org/doc/v1.5/man3/MPI_Bcast.3.php)). Son signature est :

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm
                                        comm)
```

Ses arguments sont :

- `buffer` : l'adresse du "buffer" des données à transmettre à tout processus.
- `count` et `datatype` : le nombre et type d'elements stocké dans `buffer` .

  Rappelons que les types sont les suivants:

MPI datatype	C datatype
`MPI_CHAR`	`char`
`MPI_WCHAR`	`wchar_t`
`MPI_SHORT`	`short` (signed)
`MPI_INT`	`int` (signed)
`MPI_LONG`	`long` (signed)
`MPI_SIGNED_CHAR`	`char` (signed)
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`

- `root` : le rank du processus root.
- `comm` : le communicateur (typiquement `MPI_COMM_WORLD`).

`MPI_Reduce`

Rappelons l'opération `MPI_Reduce` (OpenMPI doc ici (https://www.open-mpi.org/doc/v1.5/man3/MPI_Reduce.3.php)). Son signature est :

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op
                           op, int root, MPI_Comm comm) .
```

Ses arguments sont :

- `sendbuf` : L'adresse du "send buffer": un bloc de memoire rempli avec les données à transmettre au processus root (i.e. les données à reduire).
- `recvbuf` : L'adresse du "receive buffer", un bloc de memoire à remplir avec le resultat chez le root. Devrait être la même taille que `sendbuf` .
- `count` et `datatype` : Le nombre et type d'elements stocké dans `sendbuf` (et à recevoir dans `recvbuf`).
- `MPI_Op op` parmi :
  - `MPI_MAX` : maximum
  - `MPI_MIN` : minimum
  - `MPI_SUM` : sum
  - `MPI_LAND` : logical and
  - `MPI_LOR` : logical or
- `root` : le rank du processus root.
- `comm` : le communicateur pour cette opération (typiquement `MPI_COMM_WORLD` .

Calcul de π

Finalisez le programme à trou suivant en mettant les bons arguments dans `Bcast` et `Reduce` (enregistrez votre programme dans `pi442.cpp`).

?

```cpp
#include <mpi.h>
#include <cmath>
#include <iostream>
#include <limits>

static const double PI = 3.14159265358979323846264338;

int main(int argc, char *argv[]) {
  int rank, size;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // pour afficher tout double a precision maximal :
  std::cout.precision(std::numeric_limits<double>::digits10 + 1);

  while (1) {
    double pi;
    int n;

    if (rank == 0) {
      std::cout
          << "Entrez le nombre d'intervalles (ou un entier < 1 pour quit
          << std::endl;
      std::cin >> n;
    }

    MPI_Bcast(...); // TODO : ARGUMENTS A BCAST

    if (n < 1) {
      break;
    }

    double h = 1.0 / (double)n;
    double sum = 0.0;
    for (int i = rank; i < n; i += size) {
      double x = h * ((double)i + 0.5);
      sum += (4.0 / (1.0 + x * x));
    }
    double mypi = h * sum;

    MPI_Reduce(...); // TODO : ARGUMENTS A REDUCE

    if (rank == 0) {
      std::cout << "pi est approche par " << pi << ", l'erreur est de "
                << std::abs(pi - PI) << std::endl;
    }
  }

  MPI_Finalize();

  return 0;
}
```

Compilez votre programme

```
mpic++ pi442.cpp -o pi442
```

Testez votre programme

On vérifie que le programme donne le résultat escompté :

```
$ salloc -n 8 mpirun pi442
salloc: Granted job allocation 6785
Entrez le nombre d'intervalles (ou un entier < 1 pour quitter) :
1
pi est approche par 3.2, l'erreur est de 0.05840734641020706
Entrez le nombre d'intervalles (ou un entier < 1 pour quitter) :
10
pi est approche par 3.142425985001098, l'erreur est de 0.0008333314113051493
Entrez le nombre d'intervalles (ou un entier < 1 pour quitter) :
100
pi est approche par 3.141600986923125, l'erreur est de 8.333333331389525e-06
Entrez le nombre d'intervalles (ou un entier < 1 pour quitter) :
10000
pi est approche par 3.141592654423125, l'erreur est de 8.333316259268031e-10
Entrez le nombre d'intervalles (ou un entier < 1 pour quitter) :
1000000
pi est approche par 3.141592653589891, l'erreur est de 9.769962616701378e-14
Entrez le nombre d'intervalles (ou un entier < 1 pour quitter) :
-1
salloc: Relinquishing job allocation 6785
```

Déposez votre travail `pi442.cpp`

Le nom du fichier à déposer

Choose File No file chosen

Déposer

Il faut se connecter avant de pouvoir déposer

Exercice 3 Calcul (séquentiel) du volume d'une molécule

On modélise une molécule $M = \cup_{i=1}^{n} B_i$ par une union de n boules $B_1, \ldots, B_n$ avec

$$B_i = \text{Boule}(C_i = (x_i, y_i, z_i), r_i) = \{P = (x, y, z) \in \mathbb{R}^3 \ : \ \|C_i - P\| \leq r_i\}$$

où $\|(x, y, z)\| = \sqrt{x^2 + y^2 + z^2}$. On suppose que les données pour les n boules se trouvent déjà en mémoire vive dans un tableau (elles sont générées dans les programmes à compléter). On cherche à *approximer* le volume de M par une intégration stochastique de Monte-Carlo.

Les fichiers `Molecule.h` (code/Molecule.h) et `Molecule.cpp` (code/Molecule.cpp) définissent une classe `Boule` . Une molecule n'est qu'un `std::vector<Boule>` . Une suite des molecules sont prédefinis dans le répertoire `molecules` télécharger `molecules.zip` (molecules.zip) et l'unzipper dans le repertoire où vous travaillez (`unzip molecules.zip`). La fonction `readMolecule` sert à charger une `std::vector<Boule>` depuis l'un des fichiers.

Monte Carlo. On rappelle que la méthode de Monte Carlo désigne une famille de méthodes algorithmiques visant à calculer une valeur numérique approchée en utilisant des procédés aléatoires, c'est-à-dire des techniques probabilistes. Cette méthode s'appuie principalement sur *la loi des grands nombre*, qu'on rappelle : Soit X_i une suite de variables aléatoires i.i.d alors

$$\frac{1}{n} \sum_{i=0}^{n} X_i \to_{n \to \infty} \mathbb{E}(X_0) \ .$$

Ainsi pour calculer l'aire d'un ensemble Set il suffit de prendre des variable de la forme $1_{X_i \in Set}$ (qui vaut 1 si la varible X_i appartient à Set et 0 sinon) avec X_i des variables uniformes dont le support

contient Set. Les grands avantages de cette méthode sont :

- qu'il suffit de savoir simuler la loi des X_i pour calculer son espérance;
- qu'à aucun moment on ne prend en compte la dimension de ces variables; et
- que c'est parfaitement parallélisable !

Cette méthode s'applique donc aisément au calul de π par exemple : en effet, soit un point M de coordonnées (x, y), où $0 < x < 1$ et $0 < y < 1$. On tire aléatoirement les valeurs de x et y. Le point M appartient au disque de centre $(0, 0)$ de rayon 1 si et seulement si $x^2 + y^2 \leq 1$. La probabilité que le point M appartienne au disque est $\pi/4$. En faisant le rapport du nombre de points dans le disque au nombre de tirages, on obtient une approximation du nombre $\pi/4$ si le nombre de tirages est grand.

Un algorithme Monte Carlo séquentiel

Complétez dans les fichiers `Molecule.cpp` (code/Molecule.cpp) et `MonteCarloSequentiel.cpp` (code/MonteCarloSequentiel.cpp) (en cherchant les tags `TODO`) la version *séquentielle* de cet algorithme pour calculer le volume d'une molécule M :

- On calcule (a) un boîte englobante BB de M telle que $M \subset \mathrm{BB}$, et (b) son volume $\mathrm{vol}(\mathrm{BB})$.
- On tire aléatoirement e échantillons uniformément dans BB, et on calcule le nombre d'échantillons e' contenu dans $\cup_{i=1}^{n} B_i$. On rappelle qu'on tire aléatoirement un échantillon `x` de type `double` dans $[0, 1)$ avec le code suivant :

```
double x = rand()/(double)RAND_MAX;
```

  Pour initialiser le générateur aléatoire avec une graine ("seed") `s` , on fait le suivant avant le premier appel à `rand` :

```
unsigned int s = 0; // par exemple/defaut
srand(s); // pour pouvoir tester et reproduire le comportement du pr
```

- On donne l'approximation $\mathrm{vol}(M) \simeq \frac{e'}{e}\mathrm{vol}(\mathrm{BB})$ (qui converge vers $\mathrm{vol}(M)$ quand $e \to \infty$).

Ecrivez en commentaire en tête du fichier sa complexité en fonction de n et e.

Compilez votre code

Compilez votre code à l'aide du `Makefile` (code/Makefile):

```
make testExercice3
```

Testez votre code

Vous pouvez executer votre programme comme `./testExercice3 e moleculefile` où `e` est le nombre des tirages et `moleculefile` est un fichier de molécule. Vous allez voir que prendre des valeurs plus et plus grandes pour `e` donne des meilleures approximations.

Les molécules fournies sont :

- `sphere.mol` (molecules/sphere.mol) donne une seule boule, de rayon 1. Sa volume est donc $4\pi/3 \approx 4.188790204786390984616857844 37$, et on attend quelque chose comme :

```
$ ./testExercice3 10000000 molecules/sphere.mol
volume : 4.1894096
```

- `disjoint.mol` (molecules/disjoint.mol) donne deux boules disjointes, dont l'une de rayon 1 et l'autre de rayon 2. La volume de cette molécule est donc $12\pi \approx 37.699111843077518861551720 5994$, et on attend quelque chose comme :

```
$ ./testExercice3 10000000 molecules/disjoint.mol
volume : 38.1158392
```

- `nested.mol` (molecules/nested.mol) est un exemple dégéneré. Il donne deux boules, et le plus petit (rayon 2) se trouve entièrement dans l'autre, qui a rayon 20. La volume est donc $3200\pi/3 \approx 33510.321638291127876934862755 0$, et on attend quelque chose comme :

```
$ ./testExercice3 10000000 molecules/nested.mol
volume : 33515.2768
```

- `three.mol` (molecules/three.mol) donne une molécule à trois boules. Sa volume est approximé par 36.836331648, alors on attend quelque chose comme :

```
$ ./testExercice3 10000000 molecules/three.mol
volume : 36.8431936
```

- `rand.mol` (molecules/rand.mol) donne une molécule plus compliquée, à 16 boules. Sa volume est approximé par 2087.633964400247, alors on attend quelque chose comme :

```
$ ./testExercice3 10000000 molecules/rand.mol
volume : 2083.032005002669
```

Déposez vos fichiers

Déposez ici votre fichier `Molecule.cpp` :

Le nom du fichier à déposer
Choose File No file chosen
Déposer
Il faut se connecter avant de pouvoir déposer

Déposez ici votre fichier `MonteCarloSequentiel.cpp` :

Le nom du fichier à déposer
Choose File No file chosen
Déposer
Il faut se connecter avant de pouvoir déposer

Exercice 4 Calcul distribué

Complétez dans le fichier `MonteCarloDistrib.cpp` (code/MonteCarloDistrib.cpp) la version *distribué* de cet algorithme sur p processeurs à mémoire distribuée, en utilisant MPI, et en *partitionnant les requêtes Monte-Carlo* mais *pas les Boules*.

- Tout processus lit le molecule spécifié sur la ligne de commande, et calcule sa boîte englobante;
- Le processus root décide combien de requêtes Monte-Carlo sera traité par chacun des processeurs (attention, le nombre de requêtes e n'est pas forcement divisible par le nombre de processeurs), et communique ces valeurs aux processus avec un appel à `MPI_Scatter`
- Tout processus génère ses propres requêtes pour calculer son propre `ePrime`. Pour que ça soit utile (et qu'ils ne génèrent pas tous les mêmes requêtes), il est impératif que chaque processus a une suite différente de nombres pseudo-aléatoires, donc un appel à `srand` unique ; on assure cela par `srand(taskid)`.
- Les divers valeurs de `ePrime` local sont sommés pour le `ePrime` global par un appel à `MPI_Reduce`.

MPI_Scatter

Rappelons l'opération `MPI_Scatter` (OpenMPI doc ici (https://www.open-mpi.org/doc/v1.5/man3/MPI_Scatter.3.php)). Son signature est :

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
          int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Ses arguments sont

- `sendbuf` : l'adresse du "buffer" des données à transmettre.
- `sendcount` et `sendtype` : le nombre et type d'élèments du buffer à transmettre à *chaque* processus.
- `recvbuf` : l'adresse du buffer où chaque processus reçoit sa partie des données.
- `recvcount` et `recvtype` : le nombre et type des éléments stocké dans `recvbuf` .
- `root` : le rank du processus root.
- `comm` : le communicateur (typiquement `MPI_COMM_WORLD`).

En général, `sendcount` est égal à le nombre d'éléments dans `sendbuf` divisé par le nombre de processus.

Compilez votre programme

```
make testExercice4
```

Testez votre programme

Vous exécuterez votre programme *localement* avec `mpirun`:
`mpirun -np 8 testExercice4 e moleculefile` , avec les mêmes e et `moleculeFile` que vous avez utilisé pour Exercice 3. Comparez les résultats, et le temps de calcul pour des e plus grands.

Déposez vos fichiers

Déposez votre fichier `MonteCarloDistrib.cpp` ici.

Le nom du fichier à déposer
Choose File No file chosen
Déposer
Il faut se connecter avant de pouvoir déposer