Pour mémoire
— le site du cours : https://gaufre.informatique.univ-paris-diderot.fr/kestener/m2-genial-hpc
— aide mémoire MPI.

1 Les bases de MPI

1.1 Helloworld MPI

Revoir la présentation du calculateur (planches 63 à 69 du document `genial-intro-mpi.pdf`), notamment la commande `lstopo`.

On utilise le code `helloworld_mpi.c` situé dans `mpi/code/c/solution`.

Activités :
— Utiliser le compilateur *emballé*[1] `mpicc` pour compiler et `mpirun` pour lancer l'exécution.
— Explorer la page de manuel de `mpicc`. Exécuter la commande `mpicc --show:me`. Quelle information nous apporte-t'elle ?
— Parcourir la FAQ d'OpenMPI ; comment fait-on pour changer le compilateur sous-jacent, c'est à dire pour passer du compilateur GNU au compilateur intel par exemple ?
— — Comment modifie-t-on le nombre de tâches MPI demandées ?
— Utiliser l'option `--report-bindings` pour obtenir l'information du placement des tâches MPI, comparer d'une exécution à une autre
— Croiser l'information donnée par l'option `--report-bindings` avec celle obtenue grâce à l'appel de `MPI_Get_processor_name`
— Comment demander à `mpirun` de placer toutes les tâches MPI sur un même socket CPU ?

1.2 Performance du réseau

— **Mesure de temps de calcul.** MPI fournit l'API `MPI_Wtime` pour mesurer des temps d'exécution. Lire la page de manuel correspondante : `man MPI_Wtime`. Comment détermine-t-on la résolution temporelle associée à `MPI_Wtime` (i.e. temps minimal qui peut être mesuré) ? Quelle est la résolution temporelle effective sur `odette` ?
— **Mesure de la latence.**
La latence dans une communication point-à-point est la moitié du temps mis pour faire un aller-retour entre deux machines à travers le réseau. On considère ici qu'un aller-retour est constitué d'un appel à `MPI_Send` et `MPI_Recv` pour échanger un octet. Plus la latence est faible, meilleur est le réseau.
— Utiliser le code source `mpi/code/c/solution/mpi_latency.c` pour mesurer la latence dans les deux cas de figure :
— les deux processus MPI participant à la communication sont sur un même nœud [2]
— les deux processus MPI participant à la communication sont sur des nœuds différents (sur un supercalculateur, la latence inter-nœud est typiquement un ordre de grandeur plus élevé que la latence intra-nœud)
— Quel est l'ordre de grandeur de la latence ? Que remarquez-vous sur les mesures dans les deux cas précédents ? Sur odette on pourra comparer le cas où les deux processus MPI sont sur le **même socket**, et le cas où ils sont sur des **sockets différents**. Comment expliquer simplement les différences observées ?

1. *compiler wrapper* en anglais
2. seule cette mesure est accessible sur la machine `odette`

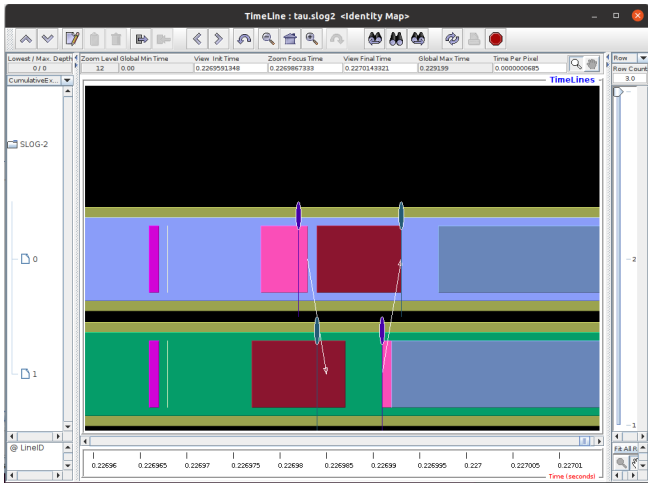

FIGURE 1 – Capture d'écran de Jumpshot. Communications bloquantes.

— **Mesure de la bande passante mémoire.**
La bande passante mémoire mesure le débit en MegaBytes par seconde des messages échangés entre deux processus MPI.
  — Utiliser le code source situé dans `mpi/code/c/bandwidth` pour effectuer la mesure.
  — Utiliser le script `gnuplot` (`mpi_bandwidth.gp`) pour visualiser la dépendance du débit en fonction de la taille en bytes du message échangé.

2 Communications point à point

2.1 Helloworld2 - Tracing et visualisation avec `tau/jumpshot`

On souhaite visualiser une trace temporelle d'exécution d'un programme MPI avec l'outil TAU [3].

— **Activité :** [4]
Suivre les instructions de l'entête du fichier `helloworld_mpi2.c`,

```
module load tau
export TAU_MAKEFILE=$TAU_ROOT/x86_64/lib/Makefile.tau-mpi
export TAU_TRACE=1
tau_cc.sh -o helloworld_mpi2 helloworld_mpi2.c
mpirun -np 2 ./helloworld_mpi2
```

— Utiliser l'outil `jumpshot` pour visualiser les traces temporelles de l'exécution du programme. Lancer `jumpshot -merge` ; l'option merge va rassembler en un seul fichier au format `slog2` les traces temporelles générées par chaque porcessus MPI.

Vous devriez avoir une interface graphique semblable à la figure 1.

— Modifier le code pour que la trace temporelle ressemble à la figure 2 : les processus de rang 0 et 1 font tous les deux un envoi non-bloquant suivi d'une reception bloquante.

3. Un tutoriel sur TAU : https://www.vi-hps.org/cms/upload/material/tw38/TAU.pdf
4. Pensez à mettre la commande `module use /opt/modulefiles` dans votre `.bashrc` pour ne plus avoir à la tapper à chaque connection sur `odette`

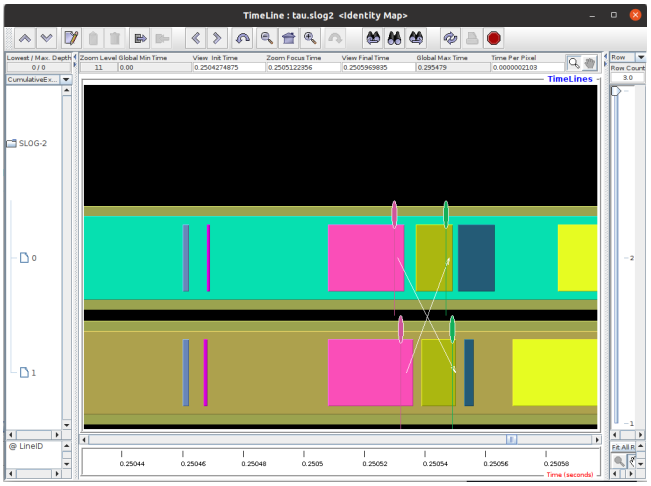

FIGURE 2 – Capture d'écran de Jumpshot. Communications non-bloquantes.

2.2 Helloworld3 - situation de *deadlock* - Eager protocol

— Utiliser le code `helloworld_mpi3.c`. Les processus MPI de rang 0 et 1 échangent deux messages sous forme d'un tableau de taille N contenant des entiers.
— Compiler et exécuter le code pour différentes valeurs de N. On constatera que le programme est bloqué pour des valeurs de N au dessus d'un certain seuil.
L'explication de ce comportement est dû au comportement de l'implantation MPI, mettant en œuvre la technique dite du *eager protocol*. Voir les planches du cours.
— Utiliser l'option `--mca btl_vader_eager_limit 8192` de `mpirun` pour changer la limite du comportement *eager* et constater que ce paramètre peut résoudre une situation de blocage.

Questions supplémentaires :
— Que se passe-t'il si l'émetteur envoie un message de taille N mais que le récepteur s'attend à un message de taille N' ? Expérimenter les deux cas $N' > N$ et $N' < N$. Que constatez-vous ?
— Modifier légèrement le code en plaçant un appel à `MPI_Get_count` juste après `MPI_Recv` pour vérifier la taille effective du message reçu.

2.3 Helloworld4 - envoie d'un message de taille variable

Le processus qui reçoit doit allouer la mémoire avant de recevoir effectivement les données.
— solution #1 : faire deux envois de message, un premier message contenant un entier pour spécifier la taille des données du deuxième message
— solution #2 : ne faire qu'un seul envoi de message, mais le récepteur fait un appel à `MPI_Probe` et `MPI_Get_count` avant `MPI_Recv` pour déterminer la taille du message et allouer la mémoire.

Le fichier `helloworld_mpi4.c` implante la solution #1.
Activités :
— Implanter la solution #2.

2.4 Exercice de parallélisation

2.4.1 Communication en anneau

Ecrire un programme (en prenant comme base `helloworld.c`) qui implante une communication en anneau.
Plus précisément, on suppose que le processus de rang 0 initialise une variable entière nommée `jeton` et ensuite l'envoie au processus de rang 1 qui l'incrémente avant de lui-même la renvoyer au processus de rang 2 ; et ainsi de suite.

— Écrire le programme correspondant et faire en sorte que chaque processus affiche à l'écran la valeur actuelle du jeton.
— Vérifier la validité du programme quand on change le nombre de processus MPI.

2.4.2 Parallélisation de la méthode des trapèzes

— Utiliser le fichier `mpi/code/c/trapeze.c`. Éditer, compiler et exécuter la version séquentielle.
— Proposer une version parallèle en utilisant que des appels à `MPI_Send` et `MPI_Recv`.

2.4.3 Multiplication de matrices carrées (facultatif) : $A * B = C$

— Utiliser la version séquentielle de l'algorithme (`matrix_multiply.c`).
On se propose d'utiliser la stratégie de parallélisation suivante : on répartit les colonnes de B sur les différents processus ; la matrice A est envoyée à tous les processus. Chaque processus calcule une partie des colonnes de la matrice résultat $C = A * B$. On ré-assemble ensuite la matrice C sur le processus 0 pour affichage et comparaison avec la version séquentielle.
— Variante 1 (simplifiée) : le rang 0 envoie toute la matrice B
— Variante 2 (optimisée) : le rang 0 n'envoie que les parties utilisées de la matrice B à chaque rang k.
— NB : on pourra décider que le processus de rang 0 ne participe pas au calcul de multiplication de matrice, il se contente d'*orchestrer* les envois et réceptions des donnéees.

3 Communications Collectives

— **Méthode des trapèzes** :
Revisiter la méthode des trapèzes en remplaçant les appels à `MPI_Send` et `MPI_Recv` par un appel à une routine collective. Quel type de communication collective reconnait-on parmi ceux énoncés dans le cours ? (planches 49)
— **Calcul de π** :
On fournit une version séquentielle d'un algorithme de calcul de π par tirage aléatoire. Proposer une version parallèle en utilisant un appel à une routine de communication collective.

3.1 Visualisation de trace temporelle avec TAU / Jumpshot

On pourra également visualiser la trace temporelle associée à l'exécution d'un broadcast MPI (`test_bcast.c`). On pourra faire plusieurs constatation :

— il n'y a pas de synchronisation globale ; certains processus MPI terminent la communication collective (`MPI_Bcast`) avant d'autres.

— ce n'est pas parce qu'un processus a fini sa participation à la communication collective que celle-ci est terminée.
— dans le cas où la taille du message est petite, certains processus peuvent avoir terminé la communication, alors que d'autres n'ont même pas commencé !

Utiliser le code `mpi_bcast_compare.c` pour mesurer le temps d'exécution d'un `MPI_Bcast` et le comparer à une implantation naïve de l'opération *broascast*.

4 Exercice d'équilibrage de charge

On suppose que chaque tâche MPI possède un tableau de taille différente (constitué d'entiers) schématisant une décomposition de domaine unidimensionnelle.

On souhaite faire un équilibrage de charge, i.e. faire un sorte que chaque tâche MPI après équilibrage possède un nouveau tableau ayant la même taille partout (à une unité près). On suppose dans un premier temps que les données sont trop volumineuses pour être rassemblées dans la mémoire d'un seul processeur.

On souhaite proposer un algorithme, qui réalise l'équilibrage et écrire le programme MPI le mettant en œuvre.

1. Commencer par déterminer la taille du nouveau tableau de **chaque processus MPI.** [5]. On pourra écrire un programme MPI où chaque processus MPI affiche à l'écran la taille avant et après le calcul de la nouvelle taille de fichier.
2. Pour chaque processus MPI, déterminer quels sont les autres processus avec qui il va devoir communiquer. [6]
   On pourra commencer par modifier le programme précédent pour que chaque processus MPI p affiche à l'écran l'information : combien d'éléments du tableau sont échangées avec le processus $p-1$ et avec le processus $p+1$.
3. Ensuite implanter les communications nécessaires pour redistribuer les données.
4. Implanter les tests qui montrent le bon fonctionnement du programme

5 Exercice Master-Worker

On souhaite calculer l'ensemble de Mandelbrot de manière parallèle avec MPI.

Problème :

— Si on fait un découpage statique du domaine, et que l'on répartit le travail avec un sous domaine par processus MPI, on obtient un déséquilibre de charge, parce que certains sous-domaines vont être calculés très vite et les processus MPI vont terminer rapidement de calculer alors que d'autres vont prendre beaucoup plus de temps et ainsi limiter les performances globales.
— **activité :** utiliser tau pour compiler le code et jumpshot pour visualiser la trace temporelle et constater le déséquilibrage de charge.

Le code `mpi_mandelbrot_naive.c` fournit la version naive de calculer l'ensemble de mandelbrot avec une décomposition de domaine.

On se propose d'implanter un algotithme de type Master-Worker pour obtenir un meilleur équilibrage de charge :

5. Indication : on pourra utiliser `MPI_Reduce`, puis effectuer une division entière.
6. Indication : on pourra utiliser `MPI_Scan`

— On découpe le domaine global en petits sous domaines de telle sorte que le nombre total de sous-domaines soit significativement plus grand que le nombre de processus MPI (e.g. 16×16)
— les processus dit *worker* (i.e. tous sauf *master*) implantent une boucle infinie, qui effectue trois choses :

1. réception (`MPI_Recv`) des infos pour traiter un bloc
2. calcul du bloc
3. envoi au processus master les résultats du bloc.

— le processus 0 (dit master) implante un algorithme en 3 phases :

1. phase 1 (initialisation) : envoi une première salve de travaux (les coordonnées des blocs à traités) en utilisant `MPI_Send`
2. phase 2 (bloc central) : tant qu'il reste des blocs à traiter, réception des résultats (de la phase d'initialisation) avec un `MPI_Recv + MPI_ANY_SOURCE`, recopie des résultats dans une grande image et envoi d'un nouveau bloc à traiter au processus qui vient de répondre (on utilisera le champ `status.MPI_SOURCE` pour le connaître)
3. phase 3 (terminaison) : réception des derniers résultats + envoi s'un signal terminaison pour dire au workers qu'il peuvent interrompre leur boucle infinie

Activités :
— Implanter cette version master/worker du calcul de l'ensemble de Mandelbrot. On pourra utiliser les planches 20 à 29 du document https://www.cse-lab.ethz.ch/wp-content/uploads/2018/03/HPC
— Utiliser tau/jumpshot pour vérifier que l'on obtient un bien meilleur équilibrage de charge
— À nombre de processus MPI égal, comment dépend le temps de calcul global de la taille des blocs (on pourra étudier les cas où les blocs sont de taille 8×8, 16×16 et 32×32. Quels sont avantages et inconvénients à diminuer et réduire la taille des blocs ?

6 Notions de communicateur MPI

— Reprendre le code sur la communication en anneau (cf. section 2.4.1).
— Lire la page de manuel de `MPI_Comm_split`, et implanter la création d'un communicateur qui regroupe les processus de rang pair / impair (dans `MPI_COMM_WORLD`).
— Faire circuler un jeton à l'intérieur des processus de rang pair.