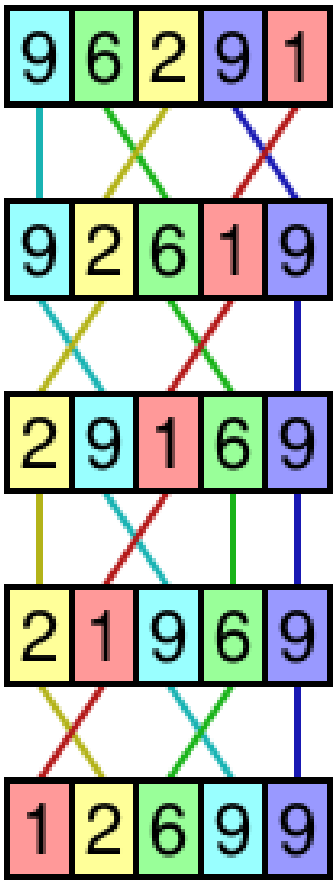
TP - Algorithmes pour le calcul parallèle

Tri pair-impair



Présenté par : Rémi CHASSAGNOL

Professeurs : Jonas KOKO et Jian Jin LI

Campus des Cézeaux. 1 rue de la Chebarde. TSA 60125. 63178 Aubière CEDEX

Table des extraits de codes

Table des matières

Introduction

L'objectif de ce TP est d'implémenter l'algorithme du *tri pair-impair* qui est un algorithme de tri parallèle. Pour ce faire, nous utiliserons la bibliothèque MPI et le langage C.

Dans ce rapport, nous commencerons par étudier un exemple de tri séquentiel qui est le *tri à bulle*. Ensuite, nous verrons l'implémentation du tri pair-impair avec MPI. Enfin, avant de conclure ce rapport, nous détaillerons les tests qui ont été réalisés pour valider les implémentations des algorithmes.

À noter que nous ne ferrons pas d'analyse d'accélération ici. Cependant, l'implémentation permet tout de même de calculer le temps passé dans les différents threads lors de l'exécution du tri parallèle.

1 Tri séquentiel

Avant de nous intéresser à l'implémentation du tri parallèle, traitons un exemple de tri séquentiel. L'algorithme que nous allons analyser est le tri à bulle.

1.1 Principe

L'objectif du tri à bulle est de faire remonter l'élément le plus petit (ou le plus grand en fonction de l'ordre que l'on souhaite) d'un sous-tableau (la bulle) vers le bord de ce sous-tableau en permutant deux éléments à chaque fois. On commence par faire remonter la bulle sur le tableau de taille N (N étant la taille du tableau en entrée). Ensuite, on fait remonter la bulle sur le sous-tableau de taille $N - 1$ et ainsi de suite jusqu'à ce qu'il n'y ait plus de sous-tableau à trier. À la fin, lorsque l'on a fait remonter toutes les bulles, le tableau principal est trié.

1.2 Implémentation

L'implémentation en C de l'algorithme est visible sur l'extrait de code 1. Ici, on peut voir que l'algorithme est constitué de deux boucles. La boucle principale (d'index `i`) permet d'itérer sur les sous-tableaux, en commençant par le tableau le plus grand. La seconde boucle (d'index `j`) permet de faire remonter la bulle. À chaque tour, si la fonction de comparaison renvoie vrai, on permute les éléments. Par exemple, si l'on cherche à trier le tableau dans l'ordre croissant, la fonction de comparaison va renvoyer vrai quand `arr[j]` $>$ `arr[j + 1]`. Dans ce cas, on permute les deux éléments pour faire remonter le plus grand vers le bord gauche du sous-tableau.

```c
void bubbleSort(int *arr, int n, int (*compare)(int, int)) {
    for (int i = n - 1; i >= 0; --i) {
        for (int j = 0; j < i; ++j) {
            if (compare(arr[j], arr[j + 1])) {
                int tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
            }
        }
    }
}
```

Extrait de code 1 – Implémentation du tri à bulle

2 Tri parallèle

À présent que nous avons vu un exemple de tri séquentiel, intéressons-nous à l'implémentation d'un tri parallèle, le tri pair-impair.

2.1 Principe

L'objectif du tri pair-impair est de donner un bout du tableau à trier à n processus. Chaque processus va trier son tableau. Ensuite, dans une première phase, chaque processus de rang rp pair va communiquer avec le processus impair de rang $rp + 1$. Les deux processus vont fusionner leurs tableaux et se partager les valeurs (on recoupe le tableau en deux). Dans, une seconde phase, c'est le processus impair de rang ri qui va procéder à la fusion et l'échange avec le processus pair de rang $ri + 1$. On répète ces deux phases $\frac{p}{2}$ fois (p étant le nombre de processus). À la fin, on fusionne les parties de tableaux de tous les processus et on obtient un tableau trié.

On peut voir sur la figure 1 la trace de l'algorithme pour le tri d'un tableau de taille 8 avec 4 processus.

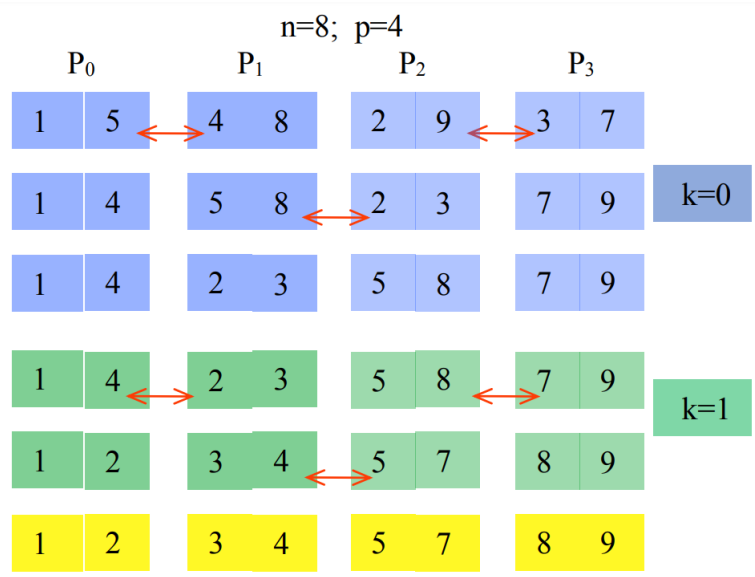


FIGURE 1 – Trace du tri pair-impair avec 4 processus

2.2 L'implémentation

L'implémentation de la fonction principale de l'algorithme est visible sur l'extrait de code 2. Sur cet extrait, certaines parties du code ont été omises pour simplifier la lecture.

```
void sortEvenOdd(int *arr, int size, bool measureTime) {
    /* variable declaration ... */

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nbprocs);
    assert(size >= nbprocs && size % nbprocs == 0); // verify if the size is valid
    subArrSize = size / nbprocs;
    /* malloc subArr and subArrShared */
    /* begin time measure ... */
    MPI_Scatter(arr, subArrSize, MPI_INT, subArr, subArrSize, MPI_INT, 0,
                MPI_COMM_WORLD);
    qsort(subArr, subArrSize, sizeof(int), &qgreater); // sort the sub array
    for (int i = 0; i < nbprocs/2; ++i) {
        if (rank % 2 == 0) {
            execEven(subArr, subArrShared, subArrSize, nbprocs, rank, tag, &status);
        } else {
            execOdd(subArr, subArrShared, subArrSize, nbprocs, rank, tag, &status);
        }
    }
    MPI_Gather(subArr, subArrSize, MPI_INT, arr, subArrSize, MPI_INT, 0,
               MPI_COMM_WORLD);
    /* end time measure + display ... */
    /* free ... */
}
```

Extrait de code 2 – Fonction `sortEvenOdd`

Dans cette fonction, on commence par récupérer le rang du processus courant et le nombre de processus. On utilise un `assert` pour vérifier que la taille du tableau en entrée est valide. Ensuite, on alloue la mémoire des sous-tableaux qui vont être utilisés par le processus courant. Le tableau `subArr` permet de stocker le sous-tableau du processus et `subArrShared` permet d'accéder au sous-tableau du processus pair (ou impair) avec lequel le processus courant communique. Le processus principal distribue les bouts de tableaux aux autres processus avec la fonction `MPI_Scatter`. Ensuite, le processus courant utilise `qsort` pour trier son sous-tableau. Une fois le sous-tableau trié, on gère les communications entre les processus dans la boucle `for` à l'aide des fonctions `execEven` et `execOdd` qui nous décrirons après. Enfin, le processus principal rassemble tous les sous-tableaux pour obtenir le tableau final trié avec la fonction `MPI_Gather`.

Pour la mesure du temps de calcul, on commence à compter avant l'appel de `MPI_Scatter` et on termine après `MPI_Gather`. Cela permet de prendre en compte le temps de calcul du thread ainsi que les temps de communications.

Comme nous l'avons mentionné, pour gérer les communications entre les processus pairs et impairs, on utilise les fonctions `execEven`, visible sur l'extrait de code 3 et `execOdd` visible sur l'extrait 4.

Si le processus est pair, c'est lui qui fusionne et échange les tableaux en premier. Il commence donc par attendre que le processus impair avec lequel il communique lui envoie son sous-tableau. Ensuite, envoie le résultat à l'autre processus. Dans la deuxième phase, c'est le processus impair qui fait la fusion. Le processus pair se contente alors juste d'envoyer son sous-tableau et de recevoir le résultat.

```
1  void execEven(/* ... */) {
2      // phase 1: the even sent to its N + 1
3      if (rank < nbprocs - 1) {
4          MPI_Recv(subArrShared, subArrSize, MPI_INT, rank + 1, tag, MPI_COMM_WORLD,
5                   status);
6          mergeSubArrays(subArr, subArrShared, subArrSize);
7          MPI_Send(subArrShared, subArrSize, MPI_INT, rank + 1, tag, MPI_COMM_WORLD);
8      }
9      // phase 2: the odd (N - 1) works with its N + 1
10     if (rank > 0) {
11         MPI_Send(subArr, subArrSize, MPI_INT, rank - 1, tag, MPI_COMM_WORLD);
12         MPI_Recv(subArr, subArrSize, MPI_INT, rank - 1, tag, MPI_COMM_WORLD,
13                  status);
14     }
15 }
```

Extrait de code 3 – Fonction `execEven`

Pour le processus impair, la communication est simplement inversée. Dans la première phase en envoie le sous-tableau et on attend le résultat du processus pair. Puis, dans la seconde phase, on fusionne les tableaux et on envoie le résultat au processus pair.

```
1  void execOdd(/* ... */) {
2      // phase 1: the event (N - 1) works with its N + 1
3      if (rank > 0) {
4          MPI_Send(subArr, subArrSize, MPI_INT, rank - 1, tag, MPI_COMM_WORLD);
5          MPI_Recv(subArr, subArrSize, MPI_INT, rank - 1, tag, MPI_COMM_WORLD,
6                   status);
7      }
8      // phase 2: the odd works to its N + 1
9      if (rank < nbprocs - 1) {
10         MPI_Recv(subArrShared, subArrSize, MPI_INT, rank + 1, tag, MPI_COMM_WORLD,
11                  status);
12         mergeSubArrays(subArr, subArrShared, subArrSize);
13         MPI_Send(subArrShared, subArrSize, MPI_INT, rank + 1, tag, MPI_COMM_WORLD);
14     }
15 }
```

Extrait de code 4 – Fonction `execOdd`

Sur la figure 2, on peut voir le schéma des communications entre les processus pair et impair comme ils sont faits avec les deux fonctions décrites précédemment.

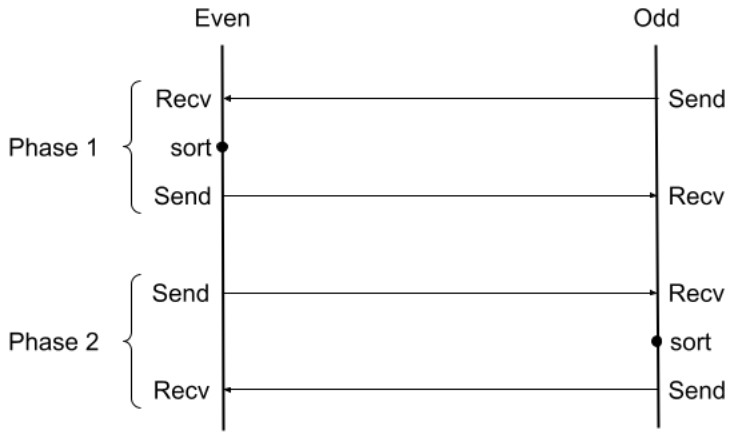


FIGURE 2 – Communications entre les processus pairs et impairs

3 Les tests

Dans cette section, nous allons présenter les tests qui ont été réalisés pour valider les implémentations des deux algorithmes. À noter que pour certains des tests, le programme utilise la fonction `generateRandomArr` qui permet de générer un tableau d'une taille donnée avec des valeurs aléatoires. La génération des valeurs aléatoire est faite à l'aide de la fonction `rand` de la bibliothèque standard. La *seed* du générateur est toujours 0 donc l'exécution du programme est reproductible. Le générateur de la bibliothèque standard est un générateur congruentiel linéaire. Ce n'est pas performant pour faire de la simulation, mais c'est suffisant pour réaliser les tests que nous détaillerons dans cette section.

3.1 Les macros de test

Les tests ont été écrits à l'aide des macros définies dans le fichier `quick-test-c/quicktest.h`. Leur utilisation est assez simple.

Pour créer une fonction de test, on utilise la macro `Test`. Cette dernière prend en paramètre l'identifiant de la fonction. Dans ces fonctions, on peut utiliser les macros `assert` et `MPI_assert`. Ces macros prennent en paramètre la condition à tester. Si la condition est fausse, le test échoue et on passe au test suivant. La fonction `MPI_assert` réalise le test uniquement pour le processus principal. On peut voir un exemple sur l'extrait 5.

```
1  Test(my_function) {
2      assert(1 == 2); // test si 1 == 2
3      MPI_assert(1 == 2); // le test n'est exécuté que par le processus 0
4      return 0;
5  }
```

Extrait de code 5 – Exemple de fonction de test

Pour lancer les tests, il faut utiliser la macro `TestRun` qui prend en paramètre l'identifiant de la fonction de test à lancer et une description. À noter qu'il faut d'abord initialiser les tests avec la macro `TestInit` (ou `MPI_TestInit`) et il faut appeler `TestEnd` (ou MPI_TestEnd) à la fin. La fonction `main` doit renvoyer `TEST_RESULT`. Un exemple est visible sur l'extrait 6.

```
1  int main(int argc, char **argv) {
2      MPI_Init( &argc, &argv );
3      MPI_TestInit();
4
5      TestRun(my_function, "exemple de lancement de test");
6
7      MPI_TestEnd();
8      MPI_Finalize();
9      return TEST_RESULT;
10 }
```

Extrait de code 6 – Lancement des tests

À la fin, le programme affiche le résultat des tests et les erreurs s'il y en a.

3.2 Tri à bulle

Dans le cas du tri à bulle, une seule fonction de test a été écrite. Elle permet de tester les cas suivants :

- cas général avec un tableau généré de façon aléatoire.
- cas où le tableau ne comporte que des 0 (permet de tester que le programme ne plante pas).
- pire cas où le tableau est trié dans l'ordre inverse.

Ici, on ne teste qu'avec des petits tableaux étant donné que l'algorithme est très simple et relativement lent. À noter également que ce test n'est exécuté que par le processus principal comme l'algorithme est séquentiel.

3.3 Tri à pair-impair

Pour le tri pair-impair, plusieurs fonctions de test ont été écrites. Elles permettent de tester les cas suivants :

- cas où la taille de tableau est égale au nombre de processus.
- cas où il n'y a la même valeur dans le tableau.
- cas général avec un tableau généré aléatoirement.
- pire cas où le tableau est trié dans l'ordre inverse.
- cas où le tableau est déjà trié.

Plus de tests ont été réalisés pour cet algorithme, car il est plus complexe étant donné la parallélisation.

3.4 Calcul du temps d'exécution

Le `Makefile` fournit avec le code permet de créer deux exécutables. Un exécutable `tests` qui permet d'exécuter les tests et un exécutable `perf`. Il permet de voir le temps d'exécution des différents processus pour le tri d'un tableau de taille $100,000,000$. La mesure de performance n'était pas demandée pour ce TP, cependant, le programme permet tout de même de mesurer l'accélération en changeant la taille du tableau généré aléatoirement ou le nombre de processus à l'exécution.

3.5 Lancement du programme

Pour lancer le programme, on utilise le script `run`. Si on ne lui donne pas de paramètre, il lance les tests. Sinon, il lance l'exécutable `perf`. Par défaut, on utilise 4 processus, mais on peut changer cette valeur dans le script si besoin.

Le programme peut aussi être lancé sur un cluster SLURM en utilisant le script `run-slurm`.

Conclusion

Dans ce rapport, nous avons présenté deux algorithmes de tri ainsi que leurs implémentations et les tests permettant leurs validations. Nous avons commencé par présenter le tri à bulle qui est un algorithme séquentiel peu efficace. Ensuite, nous avons traité l'algorithme du tri pair-impair parallèle implémenté avec MPI.