UNIVERSITÉ LIBRE DE BRUXELLES

INFO-F-404 : Real-Time Operating Systems

Bitonic Sort - Hydra

Auteur :

Yasin Arslan

Hakim Boulahya

Youcef Bouharaoua

Titulaire :

Joël Goossens

Assistant :

Cédric Ternon

Table des matières

Chapitre 1

Introduction

Dans le cadre du cours Info-F-404 : Real-Time Operating Systems, il nous a été demandé d'implanter et d'étudier l'algorithme de tri bitonic en parallèle en utilisant le module MPI. Pour ce faire, nous avons eu accès aux serveur Hydra de l'ULB. Ce rapport va tenter d'éclaircir nos éventuels choix lors de l'implantation ainsi que quelques explications au niveau de l'algorithme.

Chapitre 2

Implantation

Il nous a été donné au maximum (n/2) + 1 noeuds pour réaliser ce projet. Nous avons séparer ces noeuds en deux catégories : n/2 noeuds de comparaison et un noeud principal.

2.1 Approche

Pour effectuer le tri, nous procédons par étape de comparaison. Supposons avoir une séquence bitonique de longueur 8.

13|15|20|25|6|3|2|1

Pour débuter les comparaisons, nous allons diviser notre séquence en deux blocs.

13|15|20|25

6|3|2|1

Nous pouvons alors comparer les éléments de même index entre eux en envoyant chaque comparaison à un noeud de comparaison.

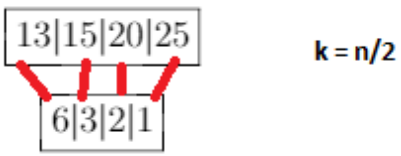

FIGURE 2.1 – Première itération. Avec k = n/2

Le noeud principal va s'occuper de réordonner les nombres dans la séquence par rapport aux résultats renvoyés par les noeuds de comparaison. Suite à la première étape de comparaison, la nouvelle séquence est la suivante :

6|3|2|1|13|15|20|25

Pour la prochaine itération, on va diviser notre k par deux, et renvoyer les éléments aux noeuds de comparaison. Une fois que k est égal à 1, nous avons fini le tri.

2.2 Algorithme

```
if ( Master_Node ) {
    INIT bloc_offset;              // size of blocs
    for i < log_2 (n) {
        bloc_offset/2
        for j < n/(k*2) {
            for k < bloc_offset {
                MPI_Send()         // send 2 elements to compare
            }
        }
    }
}
else {
    for i < log_2 (n) {
        MPI_Recv()          // Receives the elements to compare
        ...
        MPI_Send()          // Send the solution to Master_Node
    }
}
```

2.3 Performances

Quand il est fait d'une manière parallèle le tri Bitonique est en complexité logarithmique $\mathcal{O}(log^2 n)$ où n est égal au nombre de processeur. S'il est exécuté de manière séquentielle, c'est à dire sur un même processseur, sa complexité est en $\mathcal{O}(n \log^2 n)$.

2.4 Limitation

La séquence doit évidemment être biotonique c'est à dire, sous forme de liste avec au plus un maximum local et au plus un minimum local. Les points de terminaison doivent être pris en compte.

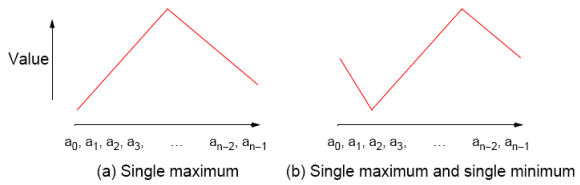

FIGURE 2.2 – Une séquence bitonique.

Une restriction au niveau de la taille de la liste est aussi présente, celle-ci doit impérativement être une puissance de deux parce car la forme la plus basique du tri demandé traite ces cas-ci.

2.5 MPI et les communications

2.5.1 Schéma de fonctionnement

Les appels MPI utilisés par notre programme sont `Recv` et `Send`. L'utilisation des appels bloquant est nécessaire pour permettre une synchronisation lors du ré arrangement effectué par le noeud principal.

La Figure 2.3 montre un aperçu des communications effectués entre le noeud principal et les noeuds de computation.

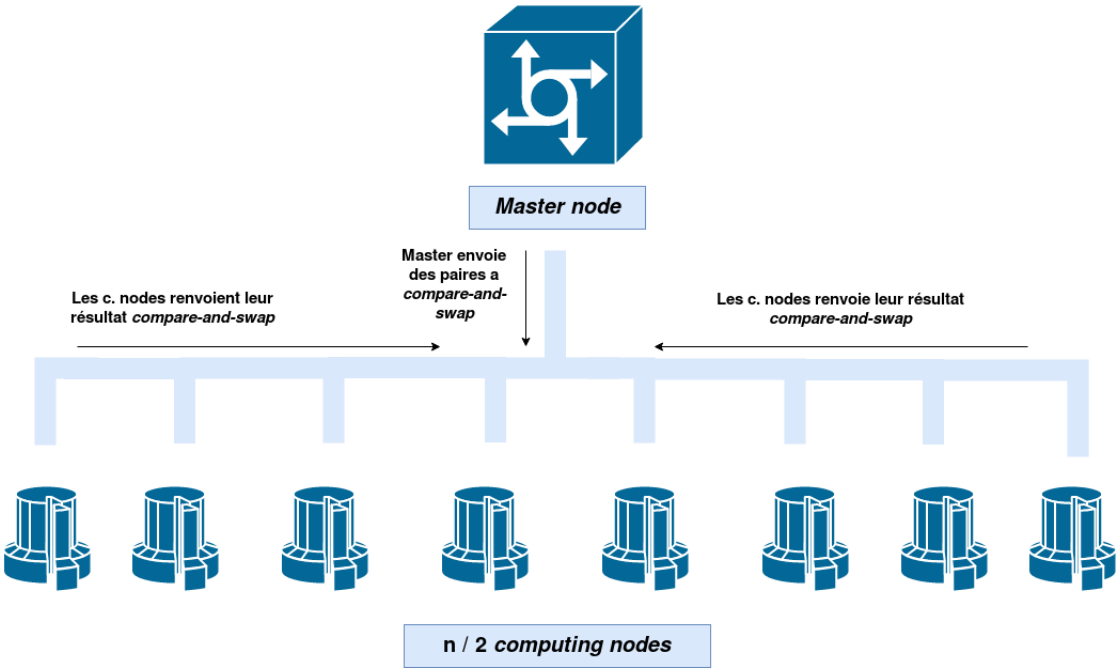

FIGURE 2.3 – Communication effectuée entre le noeud principal et les noeuds de computation.

Pour chaque itération le rôle du noeud principal est de choisir une paire dans la liste à trier pour chaque noeud de computation. Chaque noeud de computation attend de

recevoir une paire et effectué un *compare-and-swap* croissant, et renvoie le résultat au noeud principal. Comme montré sur la Figure 2.3, les actions de comparaison s'effectuent en parallèle. Au total le programme effectue $log_2(n)$ échange de message entre le noeud principal et les noeuds n'exécution.

2.5.2 Les appels MPI

Les Figures ci-dessous montrent les différents appels MPI effectués par les noeuds. La taille des *buffers* utilisés est constant et égal à 2, car uniquement les paires sont envoyés via MPI. Il est important qu'aucun tag n'est utilisé ici, car aucun message n'est à ignorer pour tous les noeuds.

```
MPI_Send(cmpbuf, 2, MPI_INT, cnode, 0, MPI_COMM_WORLD);
```

[H]

FIGURE 2.4 – Appel d'envoi effectué par le noeud **principal** : les destinataires sont les noeuds de computation.

```
MPI_Recv(cmpbuf,2,MPI_INT,cnode,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
```

[H]

FIGURE 2.5 – Appel de réception effectué par le noeud **principal** : les envoyeurs sont les noeuds de computation

```
MPI_Recv(buf,2, MPI_INT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
```

FIGURE 2.6 – Appel d'envoi effectué par les noeuds de **computation** : le destinataire est égal à O *i.e* le rank du noeud principal

```
MPI_Send(buf,2, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

FIGURE 2.7 – Appel de réception effectué par le noeud **principal** : renvoi une paire au noeud 0 *i.e.* le noeud principale

2.6 Difficultés rencontrées

2.6.1 Parallélisation de bitonic-sort

Le plus gros soucis rencontré lors de l'implantation de l'algorithme était de définir le processus de parallélisation du triage. Comme indiqué dans le cours, l'algorithme du *bitonic-sort* peut être implanté en utilisant initialement la fonction $Merge_{2k}$ permet de

séparer la séquence bitonique en 2 sous-séquences contenant les *compare-and-swap* maximum et minimum et de relancer la fonction $Merge_k$ sur les sous-séquences.

Une première idée fut d'implanter cette algorithme de récursion en effectuant le $Merge_{2k}(A)$ dans le noeud principal et ensuite d'envoyer à deux noeuds de computation les deux sous-séquences, et ceux-ci de manière récursive. Le problème de cette implantation, est que lorsque un noeud fini l'exécution du `Merge` et envoie les sous-séquences, il n'est plus sollicité et donc sera en *standby* en attendant que les autres noeuds finissent leur *merge*. C'est pour cela que nous avons préféré implanter la parallélisation des *compare-and-swap*.

2.7 Exemple d'execution

```
[INFO]Running bitonic sort with 2k=16
[INFO]Starting bitonic sort on [14, 16, 15, 11, 9, 8, 7,\\
5, 4, 2, 1, 3, 6, 10, 12, 13]
[INFO]Sending [14, 4] to 1
[INFO]Sending [16, 2] to 2
[INFO]Sending [15, 1] to 3
[INFO]Sending [11, 3] to 4
[INFO]Sending [9, 6] to 5
[INFO]Sending [8, 10] to 6
[INFO]Sending [7, 12] to 7
[INFO]Sending [5, 13] to 8
[INFO]A=[14, 16, 15, 11, 9, 8, 7, 5, 4, 2, 1, 3, 6, 10, 12, 13]
[INFO] Node #1: [14, 4] -> [4, 14]
[INFO]Master recieved from 1: [4, 14]
[INFO]Master recieved from 2: [2, 16]
[INFO] Node #2: [16, 2] -> [2, 16]
[INFO] Node #3: [15, 1] -> [1, 15]
[INFO] Node #4: [11, 3] -> [3, 11]
[INFO]Master recieved from 3: [1, 15]
[INFO] Node #5: [9, 6] -> [6, 9]
[INFO] Node #6: [8, 10] -> [8, 10]
[INFO] Node #7: [7, 12] -> [7, 12]
[INFO]Master recieved from 4: [3, 11]
[INFO]Master recieved from 5: [6, 9]
[INFO]Master recieved from 6: [8, 10]
[INFO]Master recieved from 7: [7, 12]
[INFO] Node #8: [5, 13] -> [5, 13]
```

```
[INFO] Master recieved from 8: [5, 13]
[INFO] Sending [4, 6] to 1
[INFO] Sending [2, 8] to 2
[INFO] Sending [1, 7] to 3
[INFO] Sending [3, 5] to 4
[INFO]  Node #1: [4, 6] -> [4, 6]
[INFO]  Node #2: [2, 8] -> [2, 8]
[INFO] A=[4, 2, 1, 3, 6, 8, 7, 5, 14, 16, 15, 11, 9, 10, 12, 13]
[INFO] Sending [14, 9] to 5
[INFO] Sending [16, 10] to 6
[INFO] Sending [15, 12] to 7
[INFO] Sending [11, 13] to 8
[INFO] A=[4, 2, 1, 3, 6, 8, 7, 5, 14, 16, 15, 11, 9, 10, 12, 13]
[INFO]  Node #4: [3, 5] -> [3, 5]
[INFO]  Node #3: [1, 7] -> [1, 7]
[INFO]  Node #7: [15, 12] -> [12, 15]
[INFO]  Node #5: [14, 9] -> [9, 14]
[INFO]  Node #6: [16, 10] -> [10, 16]
[INFO]  Node #8: [11, 13] -> [11, 13]
[INFO]  Node #2: [2, 3] -> [2, 3]
[INFO] Master recieved from 1: [4, 6]
[INFO] Master recieved from 2: [2, 8]
[INFO] Master recieved from 3: [1, 7]
[INFO] Master recieved from 4: [3, 5]
[INFO] Master recieved from 5: [9, 14]
[INFO] Master recieved from 6: [10, 16]
[INFO] Master recieved from 7: [12, 15]
[INFO] Master recieved from 8: [11, 13]
[INFO] Sending [4, 1] to 1
[INFO] Sending [2, 3] to 2
[INFO] A=[4, 2, 1, 3, 6, 8, 7, 5, 9, 10, 12, 11, 14, 16, 15, 13]
[INFO] Sending [6, 7] to 3
[INFO] Sending [8, 5] to 4
[INFO] A=[4, 2, 1, 3, 6, 8, 7, 5, 9, 10, 12, 11, 14, 16, 15, 13]
[INFO] Sending [9, 12] to 5
[INFO]  Node #1: [4, 1] -> [1, 4]
[INFO]  Node #3: [6, 7] -> [6, 7]
[INFO] Sending [10, 11] to 6
[INFO] A=[4, 2, 1, 3, 6, 8, 7, 5, 9, 10, 12, 11, 14, 16, 15, 13]
```

```
[INFO] Node #4: [8, 5] -> [5, 8]
[INFO] Node #5: [9, 12] -> [9, 12]
[INFO] Node #8: [16, 13] -> [13, 16]
[INFO] Node #2: [4, 3] -> [3, 4]
[INFO] Node #3: [6, 5] -> [5, 6]
[INFO]Sending [14, 15] to 7
[INFO]Sending [16, 13] to 8
[INFO]A=[4, 2, 1, 3, 6, 8, 7, 5, 9, 10, 12, 11, 14, 16, 15, 13]
[INFO]Master recieved from 1: [1, 4]
[INFO]Master recieved from 2: [2, 3]
[INFO]Master recieved from 3: [6, 7]
[INFO]Master recieved from 4: [5, 8]
[INFO]Master recieved from 5: [9, 12]
[INFO]Master recieved from 6: [10, 11]
[INFO]Master recieved from 7: [14, 15]
[INFO]Master recieved from 8: [13, 16]
[INFO]Sending [1, 2] to 1
[INFO]A=[1, 2, 4, 3, 6, 5, 7, 8, 9, 10, 12, 11, 14, 13, 15, 16]
[INFO]Sending [4, 3] to 2
[INFO]A=[1, 2, 4, 3, 6, 5, 7, 8, 9, 10, 12, 11, 14, 13, 15, 16]
[INFO]Sending [6, 5] to 3
[INFO]A=[1, 2, 4, 3, 6, 5, 7, 8, 9, 10, 12, 11, 14, 13, 15, 16]
[INFO]Sending [7, 8] to 4
[INFO]A=[1, 2, 4, 3, 6, 5, 7, 8, 9, 10, 12, 11, 14, 13, 15, 16]
[INFO]Sending [9, 10] to 5
[INFO]A=[1, 2, 4, 3, 6, 5, 7, 8, 9, 10, 12, 11, 14, 13, 15, 16]
[INFO]Sending [12, 11] to 6
[INFO] Node #1: [1, 2] -> [1, 2]
[INFO] Node #7: [14, 15] -> [14, 15]
[INFO] Node #4: [7, 8] -> [7, 8]
[INFO] Node #6: [10, 11] -> [10, 11]
[INFO] Node #6: [12, 11] -> [11, 12]
[INFO] Node #5: [9, 10] -> [9, 10]
[INFO]A=[1, 2, 4, 3, 6, 5, 7, 8, 9, 10, 12, 11, 14, 13, 15, 16]
[INFO]Sending [14, 13] to 7
[INFO]A=[1, 2, 4, 3, 6, 5, 7, 8, 9, 10, 12, 11, 14, 13, 15, 16]
[INFO] Node #8: [15, 16] -> [15, 16]
[INFO] Node #7: [14, 13] -> [13, 14]
[INFO]Sending [15, 16] to 8
```

```
[INFO]A=[1, 2, 4, 3, 6, 5, 7, 8, 9, 10, 12, 11, 14, 13, 15, 16]
[INFO]Master recieved from 1: [1, 2]
[INFO]Master recieved from 2: [3, 4]
[INFO]Master recieved from 3: [5, 6]
[INFO]Master recieved from 4: [7, 8]
[INFO]Master recieved from 5: [9, 10]
[INFO]Master recieved from 6: [11, 12]
[INFO]Master recieved from 7: [13, 14]
[INFO]Master recieved from 8: [15, 16]
Sorted sequence : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

2.8 How to

Pour pouvoir tester le programme il suffit de lancer la commande :

sh compile.sh

Chapitre 3

Conclusion

Ce projet est venu parfaitement compléter les exercices fait en séance pratique. Nous avons eu une première approche très basique qui nous a permis de bien comprendre le fonctionnement d'Hydra ainsi que du module MPI. Il nous a ensuite été assez aisé d'appliquer ces connaissances pour arriver à produire l'algorithme demandé.