Le tri rapide parallèle sur un hypercube

HES-SO//Genève, hepia ITI - orientation logiciel & systèmes complexes

---

On considère une architecture parallèle en hypercube de dimension d. On va utiliser le fait qu'un hypercube de dimension d se décompose en deux sous-hypercubes de dimension $d-1$ dont les sommets correspondants sont reliés. Cette propriété permettra à l'algorithme de partitioner le tableau à trier autour d'un élément pivot simplement en faisant que les paires de processeurs correspondants (un sur chaque sous-hypercube) se répartissent leurs éléments. Après cette répartition, les éléments inférieurs au pivot se retrouvent dans un sous-hypercube et ceux supérieurs au pivot dans l'autre. On répète ensuite la procédure sur chacun des sous-hypercubes.

Plus précisément, soit n le nombre total d'éléments à trier et $p = 2^d$ le nombre de processeurs interconnectés selon un hypercube de dimension d.

Description de l'algorithme

- **Tri local des données**
  Chaque processeur dispose de n/p données dans *dataLoc* qu'il trie localement au début.
- **Pour $i = d$ à 1, faire**
  - **Choix des pivots**
    Les processeurs de rang $j \cdot 2^i$, $0 \leq j < 2^{d-i}$, choisissent chacun comme pivot leur élément $dataLoc[\frac{n}{2p}]$.
  - **Diffusion des pivots**
    Ces processeurs sont chacun dans un sous-hypercube de dimension i (il y a 2^{d-i} sous-hypercubes). Ils diffusent chacun leur pivot aux autres $2^i - 1$ processeurs de leur sous-hypercube.
  - **Partitionnement des listes**
    Chaque processeur, après réception du pivot, partitionne sa liste d'éléments en deux listes : l'une avec les éléments inférieurs au pivot, l'autre avec ceux supérieurs au pivot.
  - **Echange de listes**
    Chaque processeur avec le $i^{\text{ème}}$ bit à 0 (respectivement à 1) envoie à son voisin suivant la dimension i sa liste des éléments supérieurs (resp. inférieurs) au pivot.
  - **Réunion de listes**
    Chaque processeur avec le $i^{\text{ème}}$ bit à 0 (resp. à 1) fait la réunion de la liste envoyée par son voisin suivant la dimension i, avec sa liste des éléments inférieurs (resp. supérieurs) au pivot. Cette réunion produit une liste triée.

Après $d = \log_2(p)$ étapes, on aboutit à une suite globalement triée selon le rang des processeurs. Attention ! Le nombre de données par processeur peut fluctuer à chaque étape.

Illustration sur un hypercube de dimension 3

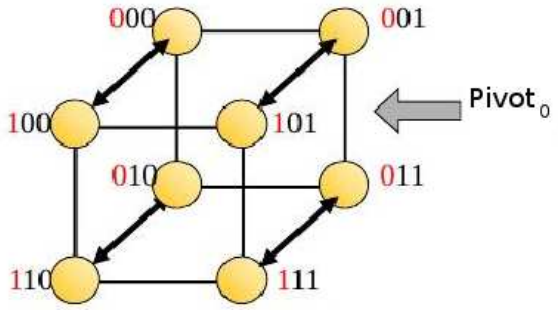

FIGURE 1 – Etape $i = 3$. Le processeur $0 = 000$ diffuse son pivot dans l'hypercube.

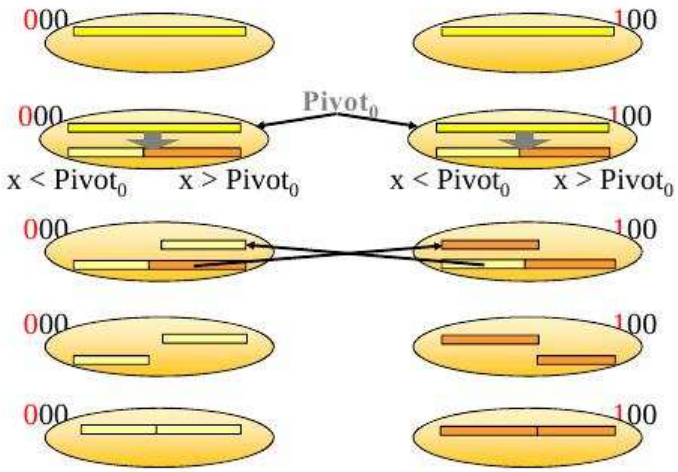

FIGURE 2 – Détail de l'étape $i = 3$ pour les processeurs $0 = 000$ et $4 = 100$.

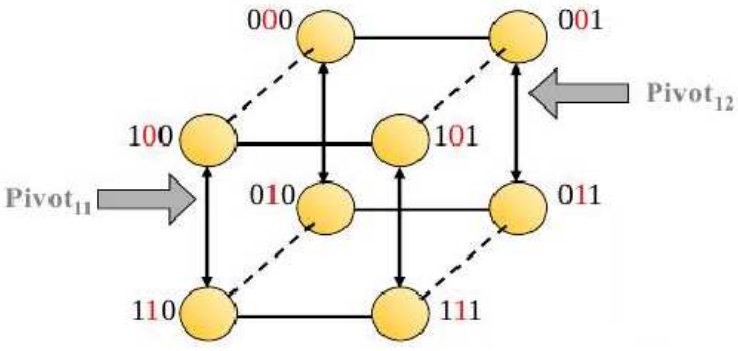

FIGURE 3 – Etape $i = 2$. Les processeurs $0 = 000$ et $4 = 100$ diffusent chacun leur pivot dans leur sous-hypercube (il y en a 2).

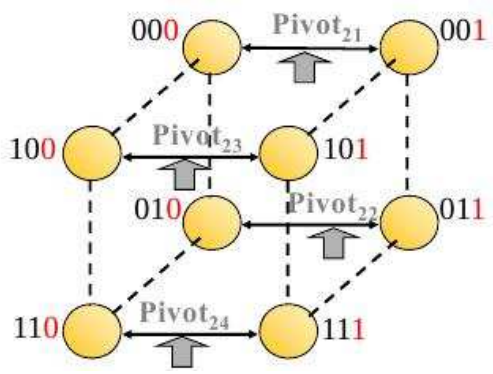

FIGURE 4 – Etape finale $i = 1$. Les processeurs $0 = 000$, $2 = 010$, $4 = 100$ et $6 = 110$ diffusent chacun leur pivot dans leur sous-hypercube (il y en a 4).

1. Implémentation de l'algorithme

Implémentez en utilisant MPI les fonctions du programme suivant :

```
void reunion(int* data1,int taille1,
             int* data2,int taille2,int* result) {  }

void partition(int pivot, int* data, int taille,
               int* dataInf,int& taille1,
               int* dataSup,int& taille2) {  }

void exchange(int* data,int& taille,int etape) {  }

void diffusion(int pivot,int etape) {  }

void quickSort(int* data,int& taille) {  }

void printAll(int* data,int taille) {
   int p,myPE;
   MPI_Comm_size(MPI_COMM_WORLD,&p);
   MPI_Comm_rank(MPI_COMM_WORLD,&myPE);
   int* res;
   if (myPE == 0) res = new int[p*taille];
   MPI_Gather(data,taille,MPI_INT,res,taille,MPI_INT,0,MPI_COMM_WORLD);
   if (myPE == 0)
      for (int k=0;k<p*taille;k++) cout << res[k] << endl;
   if (myPE == 0) delete res;
}
```

```
int main(int argc,char** argv) {
  MPI_Init(&argc,&argv);
  int myPE;
  MPI_Comm_rank(MPI_COMM_WORLD,&myPE);
  int seed = atoi(argv[1])+myPE;
  srand(seed);
  int tailleLoc = atoi(argv[2]); // tailleLoc vaut n/p
  int* dataLoc = new int[tailleLoc];
  for (int k=0;k<tailleLoc;k++) dataLoc[k] = rand()%1000;
  quickSort(dataLoc,tailleLoc);
  printAll(dataLoc,tailleLoc);
  MPI_Finalize();
  return 0;
}
```

Remarques

- Pour le tri local, utilisez la fonction séquentielle `sort()` de la librairie `algorithm`.
- N'envoyez à chaque étape que le nombre nécessaire de données.
- Vous devez implémenter chacune des fonctions ci-dessus et vous pouvez au besoin en modifier les entêtes. En particulier, la fonction `diffusion()` doit être codée similairement à un broadcast sur un hypercube.
- Les fonctions `main()` et `printAll()` ne doivent pas être modifiées.
- Faites des tests pour vérifier que votre implémentation fonctionne correctement.
- Votre code doit être bien structuré et commenté.

2. Analyse de performance

Etablissez les formules pour la complexité, le speedup, l'efficacité et la fonction d'isoefficacité de l'algorithme du tri rapide parallèle sur un hypercube. On suppose que l'algorithme est implémenté sur une machine parallèle à $p = 2^d$ processeurs interconnectés selon un hypercube de dimension d, et on note n le nombre total de données à trier.

Rendu et évaluation

- Placez votre code du tri rapide parallèle (obligatoirement nommé `par_quicksort.cc`) et vos codes de tests des fonctions dans une archive zip à votre(vos) nom(s) (p.ex. fahy_hohn.zip pour un groupe formé de Axel Fahy et Rudolf Höhn). Cette archive devra être déposée dans le dossier **TP_Quicksort** via l'onglet **Travaux** du cours **High Performance Computing** sur le site **dokeos.eig.ch**.
- Rendez les formules établies dans l'analyse de performance.
- Vous serez aussi interrogés oralement sur votre travail.