High Performance Computing Prof. P. Albuquerque

Algorithmes parallèles de multiplication matricielle

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

---

1. Implémentation

Il s'agit d'implémenter les algorithmes parallèles de multiplication matricielle vus en cours (Fox, Cannon et DNS). On vous met à disposition un squelette de code dans le fichier `multmat.cc`. Par rapport à ce squelette, vous ne devez rien changer en ce qui concerne ni les entrées-sorties, ni la génération aléatoire des matrices. Ainsi, pour le rendu, les seuls affichages seront ceux déjà présent dans le squelette (les trois appels à la fonction `printAll()`). Le choix de l'algorithme, l'initialisation du générateur aléatoire et le choix de la dimension des sous-matrices se fera via les arguments passés en ligne de commande. Par exemple, le lancement du programme avec l'algorithme DNS sur 27 processeurs, une graine de 17 pour le générateur aléatoire et des sous-matrices de dimension 5×5 se fera selon la syntaxe: `mpirun -np 27 multmat 3 17 5`

Remarques

- Les broadcasts des algorithmes doivent être implémentés efficacement. Pour cela, la notion de groupe de communication (`MPI_Comm`) peut servir pour communiquer au sein d'un groupe de processeurs (`MPI_Bcast(), MPI_Scatter(), MPI_Reduce()`).
- L'emploi de Octave/Matlab peut faciliter votre travail.
- Les fonctions `main()`, `randomInit()` et `printAll()` n'ont en principe pas besoin d'être modifiées.
- Faites des tests pour vérifier que votre implémentation fonctionne correctement.
- Votre code doit être bien structuré et commenté.

2. Analyse de performance théorique

Etablissez les formules pour la complexité, le speedup, l'efficacité et la fonction d'isoefficacité des trois algorithmes de multiplication matricielle Fox, Cannon et DNS.

3. Rendu et évaluation

- Placez votre code de multiplications matricielles parallèles (obligatoirement nommé `multmat.cc`) dans un dossier zippé en une archive à votre(vos) nom(s) (p.ex. fahy_hohn.zip pour un groupe formé de Axel Fahy et Rudolf Höhn). Celle-ci sera déposée sur le site du cours dans `https://cyberlearn.hes-so.ch`.
- Rendez les formules établies dans l'analyse de performance théorique.
- Vous serez aussi interrogés oralement sur votre travail.

4. Squelette de code

```
/*
Les sous-matrices <mat> de dimension <nloc>x<nloc> sur les <nbPE> processeurs
du groupe de communication <comm> sont rassemblées sur le processeur 0
qui les affiche en une grande matrice de dimension (<p>*<nloc>) x (<p>*<nloc>)
*/
void printAll(int* mat,int nloc,MPI_Comm comm,string label) {
   int nbPE,globalPE;
   MPI_Comm_size(comm,&nbPE);
   MPI_Comm_rank(MPI_COMM_WORLD,&globalPE);
   int* recv_buf = new int[nbPE*nloc*nloc];
   MPI_Gather(mat,nloc*nloc,MPI_INT,recv_buf,nloc*nloc,MPI_INT,0,comm);
   if (globalPE == 0) {
      int p = sqrt(nbPE+0.1);
      cout << label;
      for (int global=0;global<(p*nloc)*(p*nloc);global++) {
         int global_i = global/(p*nloc); int global_j = global%(p*nloc);
         int pe_i = global_i/nloc; int pe_j = global_j/nloc;
         int local_i = global_i%nloc; int local_j = global_j%nloc;
         int pe = pe_i*p+pe_j; int local = local_i*nloc+local_j;
         cout << recv_buf[pe*nloc*nloc + local] << " ";
         if ((global+1)%(p*nloc) == 0) cout << endl;
      }
   }
   delete recv_buf;
}
/*
Allocation et initialisation aléatoire d'une matrice de dimension <size>x<size>
avec des valeurs entières entre <inf> et <sup>; cette matrice est retournée
*/
int* randomInit(int size,int inf,int sup) {
   int* mat = new int[size*size];
   for (int i=0;i<size*size;i++) mat[i] = inf+rand()%(sup-inf);
   return mat;
}
/*
Algorithmes de multiplication matricielle parallèle
*/
void fox(int* matLocA,int* matLocB,int* matLocC,int nloc) { /*à compléter*/ }
void cannon(int* matLocA,int* matLocB,int* matLocC,int nloc) { /*à compléter*/ }
void dns(int* matLocA,int* matLocB,int* matLocC,int nloc) { /*à compléter*/ }
```

```
int main(int argc,char** argv) {
   MPI_Init(&argc,&argv);
   int nbPE,myPE;
   MPI_Comm_size(MPI_COMM_WORLD,&nbPE);
   MPI_Comm_rank(MPI_COMM_WORLD,&myPE);
   MPI_Comm comm_i_cte = MPI_COMM_WORLD,
            comm_j_cte = MPI_COMM_WORLD,
            comm_k_cte = MPI_COMM_WORLD;

   int algo = atoi(argv[1]); // 1 = fox, 2 = cannon, 3 = dns
   srand(atoi(argv[2])+myPE);
   int nloc = atoi(argv[3]);

   // sous-matrices de dimension <nloc> x <nloc>
   // des matrices A, B et C de dimension (<p>*<nloc>) x (<p>*<nloc>)
   int* matLocA = randomInit(nloc,-10,10);
   int* matLocB = randomInit(nloc,-10,10);
   int* matLocC = new int[nloc*nloc];

   switch(algo) {
   case 1: fox(matLocA,matLocB,matLocC,nloc); break;
   case 2: cannon(matLocA,matLocB,matLocC,nloc); break;
   case 3: // p^3 processeurs P_{ijk} avec i,j,k = 0,...,p-1
      int p = pow(nbPE+0.1,1.0/3.0);
      int j = (myPE/p)%p;  // A_{ik} sur les P_{i0k}
      MPI_Comm_split(MPI_COMM_WORLD,j,myPE,&comm_j_cte);
      int i = myPE/(p*p)); // B_{kj} sur les P_{0jk}
      MPI_Comm_split(MPI_COMM_WORLD,i,(myPE%p)*p+myPE/p,&comm_i_cte);
      int k = myPE%p;      // C_{ij} sur les P_{ij0}
      MPI_Comm_split(MPI_COMM_WORLD,k,myPE,&comm_k_cte);
      dns(matLocA,matLocB,matLocC,nloc);
   }
   printAll(matLocA,nloc,comm_j_cte,"%matrice complete A\n");
   printAll(matLocB,nloc,comm_i_cte,"%matrice complete B\n");
   printAll(matLocC,nloc,comm_k_cte,"%matrice complete C\n");

   MPI_Finalize();
   delete matLocA, matLocB, matLocC;
   return 0;
}
```