Calcul haute performance : notions de base (HPC)

Cours 1 bis : compléments sur MPI

P. Fortin – Sorbonne Université

M1 Info SFPN / MAIN4

All-Gather

```
int MPI_Allgather(
    void* sbuf, int scount, MPI_Datatype stype,
    void* rbuf, int rcount, MPI_Datatype rtype,
    MPI_Comm comm)
```

- Equivalent de MPI_Gather() mais tous les processus récupèrent le résultat final.

Exemple de All-Gather à 5 processus

```
main(int argc, char **argv) {
  char msg[10];
  char envoi = 97;     /* = 61h : code ascii de a */
  int my_rank;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  envoi += (char) my_rank;
  MPI_Allgather(&envoi, 1, MPI_CHAR,msg, 1, MPI_CHAR,
                MPI_COMM_WORLD);

  msg[5] = '\0';      /* fin de chaine */
  printf("Dans #%d : contenu de msg = %s\n", my_rank, msg);

  MPI_Finalize();
}
```

Tous les processus récupèrent : abcde

Algorithmes pour All-Gather

- Modélisation du temps de communications de N octets à travers le réseau : T(N) = a + N / b
  - a : latence de la communication (réseau, OS, librairie MPI ...)
  - b : débit (bande passante) en octets/s
  - On considère des réseaux full-duplex : émissions et réceptions simultanées sur le même lien réseau
- Algorithme de l'anneau :
  - P-1 étapes pour P processus qui ont chacun n / P données
  - Première étape : le processus de rang r envoie « ses » données à (r+1)%P, et reçoit les données de (r-1)%P
  - Etape « i » : le processus r envoie à (r+1)%P les données reçues à l'étape i-1
  - Etape P-1 : chaque processus a reçu les données de tous les autres

Algorithmes pour All-Gather (2)

- Analyse de l'algorithme de l'anneau :
  - À chaque étape, chaque processus envoie et reçoit n / P données
  - Au total :

    T(n, P, anneau) = (P-1).a + (P-1).n / (P.b)
  - Analyse :
    - Contribution due à la bande passante : optimale car chaque processus doit recevoir n / P données de P-1 processus
    - Possible d'améliorer la contribution due à la latence ?

Algorithmes pour All-Gather (3)

- Algorithme du doublage récursif (2rec) :
  - Hypothèse : P est une puissance de 2
  - log2(P) = k étapes
  - A l'étape « i » (0 ≤ i < k) : le processus r envoie toutes les contributions connues au processus de rang : t = r XOR 2^i, et reçoit toutes les contributions connues de t en les ajoutant à l'ensemble de ses contributions connues
  - Chaque processus envoie et reçoit k messages dont la taille double à chaque étape → quantité totale de données envoyée :

$$\sum_{i=0}^{k-1} 2^i . n/P = \left(2^k - 1\right) . n/P = \left(P - 1\right) . n/P$$

  - Au total :

T(n, P, 2rec) = log2(P).a + (P-1).n / (P.b)

Algorithmes pour All-Gather (4)

- Comparaison des deux algorithmes :
  - Petites valeurs de n :
    - temps de communication dominé par la latence
      → intérêt de l'algorithme du doublage récursif
  - Grandes valeurs de n :
    - temps de communication dominé par le débit
      → théoriquement similaires
    - mais attention à la congestion réseau pour les communications « distantes » de 2rec : par exemple sur un anneau physique de processeurs (liens directs entre voisins)

Algorithmes pour Broadcast de *n* données

- Algorithme *flat-tree* :
  - La racine envoie les n données aux P-1 autres processus successivement
  - Temps total : T(n, P, flat-tree) = (P-1).(a + n / b)
- Algorithme basé sur un arbre binaire :
  - P processus : $\lceil \log 2(P) \rceil$ étapes
  - Première étape : la racine de rang r envoie les n données à (r+P/2)%P
  - A l'étape « i » ($0 \leq i < k$) : la racine et les processus qui ont reçu les données à l'étape i-1 deviennent racines de leur sous-arbre, et envoient les données au processus « au milieu » de leur sous-arbre
  - Temps total : T(n, P, arbre) = $\lceil \log 2(P) \rceil.(a+n/b)$
  - Contribution due à la latence : optimale → algorithme bien adapté aux petites valeur de n
  - Pour les grandes valeurs de n : est-il possible de réduire la contribution due au débit ?

Algorithmes pour Broadcast de *n* données (suite)

- Algorithme de Van de Geijn :
  - Broadcast des n données (avec n « grand ») réalisé avec Scatter + All-Gather, ici avec l'hypothèse : P est une puissance de 2
  - Scatter réalisé avec un arbre binaire complet
    - log2(P) = k étapes
    - A l'étape « i » (0 ≤ i < k) : les processus concernés émettent des messages de taille n / 2^(i+1).
    - Au total :

$$\sum_{i=0}^{k-1}\left(a+n/\left(2^{i+1}.b\right)\right)=k.a+n/b\sum_{i=1}^{k}\left(1/2^{i}\right)=k.a+n/b\left(\frac{1-\left(1/2\right)^{k+1}}{1-1/2}-1\right)$$

$$=k.a+n/b\left(1-1/2^{k}\right)\ =\log 2\left(P\right).a+\left(P-1\right).n/\left(P.b\right)$$

  - n « grand » → algorithme de l'anneau pour All-Gather
  - Temps total pour le Broadcast :

    T(n, P, VdG) = a.(P-1+log2(P)) + 2.(P-1).n / (P.b)
  - Contribution due au débit désormais ≤ 2 n/b (où n/b est une borne inférieure)

Les systèmes de fichiers parallèles

- Système de fichiers classique (séquentiel) :
  - Accès en lecture concurrents : supportés mais non effectués en parallèles
  - Accès en écriture concurrents : non supportés
- Système de fichiers parallèle :
  - Accès concurrents (lecture et écriture) supportés et effectués en parallèles
  - Verrous disponibles pour gérer les conflits
- Schémas possibles d'E/S pour une application MPI :
  - E/S séquentielle : un seul processus effectue toutes les E/S
    - Pas de parallélisme pour les E/S, communications requises, problème d'extensibilité mémoire ...
  - E/S individuelles : chaque processus a ses propres fichiers
    - Besoin de pré/post-traitement sur les données (volumineuses)
  - E/S parallèles : plusieurs processus accèdent à des parties distinctes d'un même fichier

MPI-IO

- Référence : *Message Passing Interface (MPI)*, D. Lecas, I. Dupays, M. Flé, IDRIS, Décembre 2015 ; section 8.
- MPI-IO (MPI-2) permet :
  - De paralléliser les E/S sur un système de fichiers parallèle
  - D'implémenter des algorithmes efficaces pour les E/S (recouvrement des E/S par du calcul)
  - A la bibliothèque MPI de mettre en œuvre automatiquement certaines optimisations (regroupement de petites requêtes, tampons d'E/S ...)
  - Sur un système de fichiers séquentiel : MPI-IO permet de simuler un système de fichier parallèle → facilité de programmation (mais sans parallélisme au niveau des E/S)

MPI-IO (2)

- Avec MPI-IO, on peut :
  - Faire des opérations d'E/S non-collectives ou collectives
    - Opérations collectives : peuvent être bien plus efficaces que le schéma équivalent en non-collectif
  - Faire des opérations d'E/S bloquantes ou non-bloquantes
    - Opérations non-bloquantes → recouvrement E/S par calcul
  - Utiliser les types MPI (types pré-définis ou types dérivés) directement pour écrire/lire dans les fichiers
- Position dans le fichier repérée
  - soit par un décalage (*offset*) par rapport au début du fichier,
  - soit par un indicateur de position :
    - soit individuel (propre à chaque processus)
    - soit partagé entre les processus
- Avantages de MPI-IO : facilité de programmation, portabilité et performance