DS 2013 Systèmes avancés

Damien Crémilleux

14 mai 2014

1 Questions à réponse courte

1.1 Parallélisme

Q1 : Quel est le principal avantage des machines à mémoire partagée sur les machines à mémoire distribuée ? Le principal avantage des machines à mémoire partagée est l'espace d'adressage unifié. Ainsi il est possible d'utiliser des variables partagées, de mettre en place de la synchronisation. C'est bien un système adapté aux tâches légères.

Q2 : Quel est le principal avantage des machines à mémoire distribuée sur les machines à mémoire partagée ? Le principal avantage des machines à mémoire distribuée est l'utilisation de communication à distance. Il s'agit donc d'un système bien adapté aux processus plus qu'aux tâches légères, et qui nécessite un réseau pour communiquer.

Q3 : Pourquoi un programme ne peut-il (en général) s'exécuter P fois plus vite sur P processeurs que sur 1 processeur ? Un programme ne peut pas en général s'exécuter P fois plus vite sur P processeurs que sur 1 processeur car toutes les tâches ne sont pas parallélisables. Et même lorsque les tâches sont parallélisables, le coût de la parallélisation (synchronisation, etc) diminue la vitesse d'exécution du programme.

1.2 MPI

Q4 : Pourquoi ne peut-on pas partager de variables en MPI ? On ne peut pas partager de variables en MPI car il s'agit seulement d'une bibliothèque de passage de message, pour des machines distribuées. Il n'y a donc pas de mémoire partagée et de variable partagée.

Q5 : Pourquoi n'utilise t'on pas le rendez-vous pour la communication en MPI ? En MPI, les communications sont non-bloquantes afin d'être plus efficaces. En outre, cela évite le risque de *deadlock* si un message se perd sur le réseau.

1.3 Système de fichiers distribués

Q6 : Quel est l'intérêt d'avoir un serveur de fichier distribué stateless ? L'intérêt d'avoir un serveur de fichier stateless (comme NFS v3) est la résistance

aux problèmes de réseau. Ainsi le serveur peut être situé dans un environnment instable, subir un crash et redémarrer sans problème.

```
When NFS was developed there were many problems with the stability
     of networks and servers. NFS was designed to be an efficient
    protocol that could successfully handle an unstable
    environment without causing harm to the client machine. To
    operate in such a potentially harsh environment, NFS was built
     to be stateless. Being stateless means that there is no "
    state" that any operation could hold NFS in. For instance, in
    NFS version 2, when you perform a write, it will not return a
    result until the data has been written onto permanent storage.
     If the write never returns, it is up to the client to decide
    to continue waiting for the write to finish, or to not make
    the write.

So being stateless means that the server can crash and the client
    will wait for the server to come back up. However, for some
    technical reasons as we'll explain below, there can be minor
    problems. The most notable one is the "Stale File Handle" error
     that can be seen after a server is rebooted. The cause of this
     one is, related to the file handles that the server uses with
    its filesystem. If those should change, then NFS mounts that
    the client will not have the correct filehandle and it will
    need to be unmounted and then remounted.
```

Q7 : Quelles sont les limitations de NFS pour une utilisation dans un cadre massivement parallèle ? NFS a été désigné pour des petites infrastructures et présente donc des limites à cause de l'approche client/serveur. Ainsi, face à un grand nombre de client, le serveur NFS se trouvera surchargé.

1.4 Mémoire virtuellement partagée

Q8 : Quelles sont les deux fonctions essentielles d'une mémoire virtuellement partagée ? Les deux fonctions essentielles d'une mémoire virtuellement partagée sont :

- Les données sont distribuées entre différentes mémoires.
- Un accès transparent est fourni aux données.

Q9 : Dans le cas d'une gestion de mémoire avec invalidation ; que se passe-t-il lorsque deux processus situés sur des processeurs différents accèdent régulièrement à une même variable ? Que peut-on faire pour améliorer l'efficacité dans ce cas ? Lorsque deux processus situés sur des processeurs différents accèdent à une même page, cela va provoquer un défaut de page pour le processus arrivant en deuxième. Si cela arrive régulièrement, les performances peuvent donc être dégradée. Afin de ne pas avoir ce problème et de garder une mémoire cohérente, il est possible de mettre en place le *multiple-writer protocol*. Ce protocole permet à plusieurs processus d'écrire en même temps sur une même page, si les modifications ont des localisations différentes.

Q10 : Deux techniques de gestion de la cohérence existent : le *write back* et l'invalidation. Donner des arguments en faveur ou en défaveur de chacune de ces techniques (coût des défaut de page, nombre de

messages, ...)

	write back	invalidation
avantages	Le défaut de page a uniquement lieu au premier accès	L'écriture est immédiate
inconvénients	Il faut attendre pour écrire	Beacoup de mise à jour sont envoyées à chaque écriture

2 Exercice

2.1 Question 1

Les éléments nécessaires au calcul d'une ligne de la matrice C sont la ligne de même indice de A, et toute la matrice B.

2.2 Question 2

Lors de cette phase, 2*(P-1) messages seront envoyés. Cela représente N*N*(P-1)+N, soit N*N*P floats.

2.3 Question 4

Il y a (P-1) messages envoyés, soit (P-1)*(N/size)*N floats.

2.4 Question 5

Les fonctions MPI_Gather et MPI_Scatter sont des fonctions plus évoluées que les send/receive et qui permettent de respectivement : recevoir des données de tous les membres du groupe et de les rassembler directement, envoyer par morceaux une structure à tous les membres du groupe.

2.5 Question 6

Cet algorithme n'est pas performant car il nécessite l'envoi d'un nombre important de données. Ainsi la matrice B doit être complètement envoyée à chaque processus.

```
/* Exercice - DS2013 */
/* Damien Cremilleux */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <mpi.h>

#define N 3

int main(int argc, char * argv[])
{
  int i;
  int j;
```

```
int k;
float A[N][N];
float B[N][N];
float C[N][N];

int rank, size;

MPI_Status stat;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if(size != 3) {
  printf("Ce programme ne fonctionne qu'avec 3 
      processus\n");
  exit(-1);
}

if(rank==0){
  A[0][0] = 1;
  A[0][1] = 2;
  A[0][2] = 3;
  A[1][0] = 4;
  A[1][1] = 5;
  A[1][2] = 6;
  A[2][0] = 7;
  A[2][1] = 8;
  A[2][2] = 9;

  B[0][0] = 10;
  B[0][1] = 11;
  B[0][2] = 12;
  B[1][0] = 13;
  B[1][1] = 14;
  B[1][2] = 15;
  B[2][0] = 16;
  B[2][1] = 17;
  B[2][2] = 18;

  for(i=1 ; i<size ; i++){
    MPI_Send(B, N*N, MPI_FLOAT, i ,0 , MPI_COMM_WORLD);
    MPI_Send(A+i*(N/size), N*(N/size), MPI_FLOAT, i, 0,
        MPI_COMM_WORLD);
  }

  /* affichage des matrices */
  printf("Matrice A :\n");
  for(i = 0 ; i < N ; i++) {
    for(j = 0 ; j < N ; j++) {
      printf("%f ", A[i][j]);
```

```
    }
    printf("\n");
  }
  printf("Matrice B :\n");
  for(i = 0 ; i < N ; i++) {
    for(j = 0 ; j < N ; j++) {
      printf("%f ", B[i][j]);
    }
    printf("\n");
  }
}

if(rank != 0){
  MPI_Recv(B, N*N, MPI_FLOAT, 0, MPI_ANY_TAG,
      MPI_COMM_WORLD, &stat);
  MPI_Recv(A+i*(N/size), N*(N/size), MPI_FLOAT, 0,
      MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
}


for(i = 0 ; i < N/size ; i++) {
  for(j = 0 ; j < N ; j++) {
    float res_ij;
    res_ij = 0;
    for(k = 0 ; k < N ; k++) {
      res_ij += A[i*(N/size)][k] * B[k][j];
    }
    C[rank*(N/size)+i][j] = res_ij;
  }
}

if(rank != 0) {
  MPI_Send(C + rank*(N/size), N*(N/size), MPI_FLOAT, 0,
      0, MPI_COMM_WORLD);
}

if(rank == 0) {
  for(i = 1; i < size ; i++) {
    MPI_Recv(C + i*(N/size), N*(N/size), MPI_FLOAT, i,
        MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
  }

  /* Affichage du resultat */
  printf("Matrice C :\n");
  for(i = 0 ; i < N ; i++) {
    for(j = 0 ; j < N ; j++) {
      printf("%f ", C[i][j]);
    }
    printf("\n");
  }
```

```
    }

    MPI_Finalize();

    return 0;
}
```