CS2601
Computacion Paralela y Distribuida

Laboratorio 4

Prof. Jose Antonio Fiestas Iquira

Enrique Sobrados
Cesar Madera
Stephano Wurttele

16 de junio de 2021

1. Ejercicio 1

Para la primera parte del ejercicio, se utilizo el comando MPI_Comm_size para poder obtener el numero de procesos los cuales se han inicializado, con el cual realizamos una verificación y en caso sea diferente a 2 se terminan todos los procesos. Para ello se implemento lo siguiente:

```
MPI_Comm_size(MPI_COMM_WORLD, &numprocs );
if(numprocs > 2){
    MPI_Finalize();
    return 0;
}
```

Ahora bien, para el ejercicio en si, se utilizó el send y recieve implementados de MPI para hacer las comunicaciones entre procesos. Estamos creando un array de tamaño 2^{16} y dentro del send y recieve unicamente se envia una porcion cada vez mas grande de ese mismo para evitar la creacion y destruccion de arrays con cada envio o recibo. El codigo implementado para el proceso 0, es el siguiente:

```
MPI_Ssend(&data,1,MPI_INT,1,0,MPI_COMM_WORLD);
while(num < N/2){
    num*=2;
    MPI_Recv(&data,num,MPI_INT,1,0,MPI_COMM_WORLD,&status);
    MPI_Get_count(&status,MPI_INT,&count);
    time_per_process = MPI_Wtime();
    cout<<"process "<<me<<" timer: "<<
        time_per_process-initial_time_per_process<<endl;

    num*=2;
    MPI_Ssend(&data,num,MPI_INT,1,0,MPI_COMM_WORLD);
}
MPI_Recv(&data,num*2,MPI_INT,1,0,MPI_COMM_WORLD,&status);
MPI_Get_count(&status,MPI_INT,&count);
time_per_process = MPI_Wtime();
cout<<"process "<<me<<" timer: "<<
    time_per_process-initial_time_per_process<<endl;
```

Cabe resaltar que el código para el proceso 1, es exactamente opuesto al código presentado anteriormente. El proceso comienza con el thread de rank 0, el cual envia un array conteniendo 1 elemento, el cual lo recibe el thread de rank 1 y envía un array pero del doble del tamaño, y así sucesivamente. Después de cada recieve, se esta tomando el tiempo y imprimiéndolo en pantalla.

2. Ejercicio 2

Para este problema, se utilizo send y receive, se mandaban los mensajes que en este caso lo almacenamos en una varible data que es una lista de dos elementos, el primero es el valor del mensaje que vimos en la Practica dirigida 3, la cual aumentaba en data = data + rank*100. El segundo valor es el tiempo que demora de latencia acumulado.

```
if(me==0){
    data[0] = 1000;
    MPI_Ssend(&data,2,MPI_DOUBLE,me+1,0,MPI_COMM_WORLD);
    MPI_Recv(&data,2,MPI_DOUBLE,numprocs-1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
```

```
        data[1] += MPI_Wtime();
        cout << me << ": " << data[0] << " con tiempo " << data[1] << endl;
    }
        else {
        MPI_Recv(&data,2,MPI_DOUBLE,me-1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        data[0] += 100*me;
        data[1] += MPI_Wtime();
        MPI_Ssend(&data,2,MPI_DOUBLE,(me+1)%numprocs,0,MPI_COMM_WORLD);
    }
```

El código se corrió múltiples veces con 8 procesos, en todos los casos se obtuvo el mismo tiempo de latencia de: 1.29905e+10.

3. Ejercicio 3

Para este problema, se implementaron 2 códigos para solucionarlo. Uno está basado en send's y recieve's mientras que el otro esta basado en scater y gather. En el primer código, se esta mandando cada fila por separada a los procesos correspondientes uno a la vez, de tal manera que el proceso recibe una por una las filas y hace el calculo que procede. Una vez terminado, envía la información de nuevo al proceso 0, el cual almacena la información dentro del vector de salida. En ese sentido, el proceso 0 contiene las siguientes instrucciones:

```
    for(int i = 0; i < N; i++){
            MPI_Send(&A[i][0],N,MPI_INT,i/rows_per_process+1,i,MPI_COMM_WORLD);

        }
        //receive
        for(int i = 0; i < N; i++){
            MPI_Recv(&x[i],1,MPI_INT,i/rows_per_process+1,i
                ,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
```

El proceso 0 esta encargando unicamente de enviar y recibir la informacion de los demas procesos. en ese sentido se tiene P-1 procesos y por ende el numero de filas por proceso se define como N/(P-1). En el caso de los demas procesos, se tiene el siguiente codigo:

```
int mat[N];
int var = 0;
for(int i = 0; i < rows_per_process; i++){
    var = 0;
    MPI_Recv(&mat,N,MPI_DOUBLE,0,(me-1)*rows_per_process+i,
        MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    for (int j = 0; j < N; j++ ){
        var  += mat[j]*v[j];
    }
    MPI_Send(&var, 1, MPI_INT, 0, (me-1)*rows_per_process+i, MPI_COMM_WORLD);
}
```

Cada proceso realiza, por cada fila que le toca procesar, la fila de la matriz original y la almacena en *mat* y luego procede a realizar la multiplicación, y devuelve el resultado de esa fila al proceso 0 y para poder diferenciarlos, se le añade un tag con el numero de fila correspondiente, para que después

el proceso 0 lo almacene en el array de salida.

Por otra parte, el segundo código envía todo el bloque de información conteniendo N*numero_de_procesos respectivamente a cada uno de los procesos.

- Primero se procede a enviar la información con un broadcast y un scatter del vector y la matriz correspondientemente:

```
MPI_Scatter( (int**)A , N*rows_per_process , MPI_INT , (int**)lA ,
    N*rows_per_process , MPI_INT ,0 , MPI_COMM_WORLD);
MPI_Bcast((int*)v,N,MPI_INT,0,MPI_COMM_WORLD);
```

- Luego se procede a hacer la multiplicación matricial localmente con la información recibida:

```
for (int i = 0;i<rows_per_process;i++){
        for (int j = 0;j<N;j++){
            lx[i]+=lA[i][j]*v[j];
        }
    }
```

- Por ultimo se utiliza el gather para juntar todas las respuestas de los procesos:

```
MPI_Gather( (int*)lx, rows_per_process , MPI_INT , (int*)x , rows_per_process
, MPI_INT , 0 , MPI_COMM_WORLD);
```

El costo computacional promedio corriendo el código 5 veces utilizando un N de 120 y 7 procesos(1 que distribuye y 6 que operan) es : 0,003739693 segundos.

El speedup real es: Ts(n) = 0,000259161 / Tp(n) = 0,000578403 = 0,448063029. La eficiencia real es: 0,448063029/7 = 0,064009004.

Por otra parte, para la ejecución del segundo código, el promedio de correr el codigo 5 veces en paralelo utilizando un N de 120 para 8 procesos (todos operan) es : 0.000275191 segundos.

El speedup real es: Ts(n) = 0,00005647 / Tp(n) = 0,000175191 = 0,32233391 La eficiencia real es: 0,32233391/8 = 0,040291739

4. Ejercicio 4

4.1. Implementación

La *norma-infinito*, en palabras simples, busca encontrar la fila en una matriz cuya suma es máxima en relación al resto de filas. En ese sentido, se puede conceptualizar que la forma de paralelizar el problema es calcular el máximo en cada fila en procesos distintos (o distribuir el número de la forma más uniforme posible y calcular el máximo de cada uno de esos subgrupos). En ese sentido, el algoritmo se distribuye en los siguientes pasos:

1. Distribución de la data por proceso
2. Cálculo de suma por proceso y comparación de ser necesaria
3. Recopilación y definición de máximo

En el código consideramos adicionalmente un paso 0 que sería la población de la matriz, sin embargo no lo contamos para el análisis de tiempo porque no es parte del algoritmo de *norma-infinito*.

Distribución de la data por proceso

En el siguiente bloque de código, donde asumimos que la matriz ya esta poblada y existe por default en el nodo maestro, utilizamos en principio una condicional para saber si nos encontramos en el nodo maestro. De si estarlo, enviamos la información de la matriz a todos los otros procesos usando la directiva *MPI_Send*, así como asignamos el arreglo para su propio nodo más adelante con un for tradicional.
Además, en caso no sea el nodo maestro, estos reciben el número de filas que tengan que recibir de acuerdo al número de procesos y las dimensiones de la matriz usando la directiva *MPI_Recv*. Estas se guardan en *nums*, matriz manejada por todos los procesos para sus datos del proceso.

```
if(me==0){
    int A[N][N]; // Lo asumimos como ya poblado por no ser parte del algoritmo

    for(int i = rows_per_process; i < N; ++i){
        MPI_Send(&A[i],N,MPI_INT,i/rows_per_process, 0, MPI_COMM_WORLD);
    }
}
else{
    for(int i = 0; i < rows_per_process ; ++i)
        MPI_Recv(&nums[i],N,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}
```

Cálculo de suma por proceso y comparación de ser necesaria

Este es el código que se ejecuta en todos los procesos, tal que se recorren todas las filas que se le han sido asignadas, calcula la suma de cada una y las compara entre ellas para obtener un máximo local.

```
for(int i = 0; i < rows_per_process; ++i){
    sum = 0.0;
    for (int j = 0; j < N; ++j)
        sum += fabs(nums[i][j]);
    if(sum > data)
        data = sum;
}
```

Recopilación y definición de máximo

Finalmente, se usa la directiva *MPI_Reduce*, donde se recolectan los resultados locales por proceso y se calcula el mínimo de ellos en el nodo principal, donde se imprime posteriormente.

```
MPI_Reduce(&data, &res, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
if(me==0)
    printf("Res is %d\n",res);
```

4.2. Costos

El costo computacional de las primera, segunda, y tercera parte son $\frac{N}{P}$, $\frac{N}{P}$ y P respectivamente. En ese sentido, la complejidad total sería $\frac{N}{P} + \frac{N}{P} + P$. Considerando que N siempre será un múltiplo de

P, tal que $N = K * P$, la complejidad sería de $K + K + P$. Ahora, asumiendo que K será usualmente mayor que P, la complejidad será un $O(K)$, donde K es el ratio de elementos frente a los procesos.
El costo de comunicación sería N de distribución, pues cada linea debe ser enviado a su respectivo proceso, y P de recepción, pues debe recopilar los resultados de cada proceso. Nuevamente, considerando $N = K * P$, el costo de comunicación sería $K * P + P = P * (1 + K)$, nuevamente teniendo un $O(K)$ de acuerdo al ratio. Recordar para ambos casos que:

$$N > K > P$$

4.3. Speedup y eficiencia

Finalmente el speedup, considerando que la complejidad serial es de N^2, sería igual a $\frac{N^2}{K}$. Haciendo que el problema tienda al infinito, el K crecería igualmente asintoticamente hasta alcanzar N, generando un speedup de $\frac{N^2}{N} = N$. Finalmente, la eficiencia sería igual a N/P.

5. Enlaces externos

`https://github.com/cesar214567/Proyecto4Paralela`