PPR

Programación Paralela.

Práctica 2:

Implementación distribuida de un algoritmo paralelo de datos.

Autora: Cristina María Crespo Arco
Correo: cmcrespo@correo.ugr.es

Profesor: José Miguel Mantas Ruiz

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Curso 2021 - 2022

Índice

1. Información del ordenador que ha realizado las pruebas.

- Sistema Operativo: Ubuntu 18.04.6 LTS
- Tipo de Sistema Operativo: 64 bits
- Procesador: Intel(R) Core(TM) i7-7700k CPU @ 4.20GHz x 8
- Memoria RAM: DDR4 con capacidad de 64GB y velocidad de 2133 MT/s
- Compilador: mpiCC
- Compilación de cada programa:
  - mpiCC matriz_x_vector2.cpp -o matriz_x_vector2
  - mpiCC descomposicion_bidimensional.cpp -o descomposicion_bidimensional
- Ejecución: mpirun
- Compilación de cada programa:
  - mpirun -np -*numero_procesos*- matriz_x_vector2 -*n*-
  - mpirun -np -*numero_procesos*- descomposicion_bidimensional -*n*-

2. Descomposición bidimensional (por bloques 2D)

2.1. Ejercicio 1:

mplementar el algoritmo de multiplicación paralela matriz-vector basado en descomposición bidimensional usando funciones de MPI. Comienza entendiendo bien la versión (ya dada) que asume una distribución unidimensional de la matriz A, para abordar la versión que usa una distribución 2D de la matriz. Usa como punto de partida la solución del ejercicio de multiplicación matriz-vector del tutorial de MPI disponible en: `https://lsi.ugr.es/jmantas/ppr/tutoriales/tutorial_mpi.php?tuto=05_matriz_x_vector`

2.1.1. Implementación del código.

```
int main(int argc, char * argv[]) {
  ...
  float *A, // Matriz global a multiplicar
        *x, // Vector a multiplicar
        *y, // Vector resultado
        *local_A,  // Matriz local de cada proceso
        *local_x,  // Vector local de cada proceso
        *local_y,  // Porción local del resultado en  cada proceso
        *l_y;

  ...
  int numero_filas_columnas = ceil(sqrt(numeroProcesadores));
  int elementos_fila_columna = ceil(n/(sqrt(numeroProcesadores)));
  x = new float[n]; //reservamos espacio para el vector x (n floats).
  y = new float[n]; //reservamos espacio para el vector resultado final
                    // y (n floats)

  float *buf_envio = new float[n*n];

  // Proceso 0 genera matriz A y vector x
  if (id_Proceso==0)  {
    ...
    /*Defino el tipo de bloque cuadrado*/
    MPI_Datatype MPI_BLOQUE;
    MPI_Type_vector(elementos_fila_columna,
                    elementos_fila_columna,
                    n,
                    MPI_FLOAT,
                    &MPI_BLOQUE);

    /*Creo el nuevo tipo*/
    MPI_Type_commit(&MPI_BLOQUE);

    /*Empaqueta bloque a bloque en el buffer de envío*/
```

```
  for (int i = 0, posicion = 0; i < numeroProcesadores; i++) {
    /*Calculo la posicion de comienzo de cada submatriz*/
    int fila_P = i / numero_filas_columnas;
    int columna_P = i % numero_filas_columnas;
    int comienzo = (columna_P*elementos_fila_columna) +
                   (fila_P*elementos_fila_columna*elementos_fila_columna*
                   numero_filas_columnas);
    MPI_Pack(&A[comienzo],
             1,
             MPI_BLOQUE,
             buf_envio,
             sizeof(float)*n*n,
             &posicion,
             MPI_COMM_WORLD);
  }

  /*Libero el tipo bloque*/
  MPI_Type_free(&MPI_BLOQUE);
}

// Cada proceso reserva espacio para su porción de A y para el vector x
const int local_A_size = (elementos_fila_columna)*(elementos_fila_columna);
const int local_x_size = elementos_fila_columna;
const int local_y_size = elementos_fila_columna;
local_A = new float[local_A_size];//reservamos espacio para la matriz
local_x = new float[local_x_size]; //reservamos espacio para el vector x.
local_y = new float[local_y_size]; //reservamos espacio para el vector y.
l_y = new float[local_y_size]; //reservamos espacio para el vector y.

for (int i = 0; i < local_x_size; i++) {
  local_x[i] = 0;
}

// Repartimos una bloque de filas de A a cada proceso
MPI_Scatter(buf_envio, // Matriz que vamos a compartir
            sizeof(float)*local_A_size, // Numero de filas a entregar
            MPI_PACKED, // Tipo de dato a enviar
            local_A, // Vector en el que almacenar los datos
            local_A_size, // Numero de filas a recibir
            MPI_FLOAT, // Tipo de dato a recibir
            0, // Proceso raiz que envia los datos
            MPI_COMM_WORLD); // Comunicador utilizado (el global)

// Repartimos y difundimos el vector x a cada proceso
MPI_Comm comm_diagonal; // Comunicador para los elementos de la diagonal.
int diagonal = 0;

for (int i = 0; (i < numero_filas_columnas) && (diagonal == 0); i++) {
```

```
  if((i * sqrt(numeroProcesadores) + i) == id_Proceso)
    diagonal = 1;
}

// creamos un nuevo cominicador para los elementos en la diagonal
MPI_Comm_split(MPI_COMM_WORLD, // a partir del comunicador global.
               diagonal, // lo de la diagonal entraran en el comunicador
               id_Proceso, // indica el orden de asignacion de rango
                           // dentro de los nuevos comunicadores
               &comm_diagonal); // Referencia al nuevo comunicador creado.

// Repartimos el vector x entre la diagonal
MPI_Scatter(x, // Matriz que vamos a compartir
            local_x_size, // Numero de filas a entregar
            MPI_FLOAT, // Tipo de dato a enviar
            local_x, // Vector en el que almacenar los datos
            local_x_size, // Numero de filas a recibir
            MPI_FLOAT, // Tipo de dato a recibir
            0, // Proceso raiz que envia los datos
            comm_diagonal); // Comunicador utilizado, el de la diagonal

// Repartimos el vector x entre todas las columnas
MPI_Comm comm_columnas; // Comunicador para los elementos de la columnas.

int columna = floor(id_Proceso / numero_filas_columnas);

// creamos un nuevo cominicador para los elementos en la diagonal
MPI_Comm_split(MPI_COMM_WORLD, // a partir del comunicador global.
               columna, // lo de la diagonal entraran en el comunicador
               id_Proceso, // indica el orden de asignacion de rango
                           // dentro de los nuevos comunicadores
               &comm_columnas); // Referencia al nuevo comunicador creado.

MPI_Bcast(local_x, // Dato a compartir
          local_x_size, // Numero de elementos que se van a enviar y recibir
          MPI_FLOAT, // Tipo de dato que se compartira
          columna, // Proceso raiz que envia los datos
          comm_columnas); // Comunicador utilizado

// Hacemos una barrera para asegurar que todas los procesos comiencen
// la ejecucion a la vez, para tener mejor control del tiempo empleado
MPI_Barrier(MPI_COMM_WORLD);

// Inicio de medicion de tiempo
tInicio = MPI_Wtime();

for (int i = 0; i < local_y_size; i++) {
  local_y[i] = 0.0;
```

```
  for (int j = 0; j < elementos_fila_columna; j++) {
    local_y[i] += local_A[i*(elementos_fila_columna)+j] * local_x[i];
  }
}

MPI_Barrier(MPI_COMM_WORLD);
// fin de medicion de tiempo
Tpar = MPI_Wtime()-tInicio;

for (int i = 0; i < local_y_size; i++) {
  l_y[i] = 0;
}

MPI_Reduce (local_y, // Dato que envia cada proceso
            l_y, //Dato que recibe el proceso raiz
            local_y_size, // Numero de elementos que se envian-reciben
            MPI_FLOAT, // Tipo del dato que se envia-recibe
            MPI_SUM, //Operacion que se va a realizar
            columna, // proceso que va a recibir los datos
            comm_columnas); // Canal de comunicacion

// Recogemos los datos de la multiplicacion, por cada proceso sera
// un escalar y se recoge en un vector, Gather se asegura de que la
// recolección se haga en el mismo orden en el que se hace el Scatter,
// con lo que cada escalar acaba en su posicion correspondiente del
// vector.
MPI_Gather(l_y, // Dato que envia cada proceso
           local_y_size, // Numero de elementos que se envian
           MPI_FLOAT, // Tipo del dato que se envia
           y, // Vector en el que se recolectan los datos
           local_y_size, // Numero de datos que se esperan recibir por
                         // cada proceso
           MPI_FLOAT, // Tipo del dato que se recibira
           0, // proceso que va a recibir los datos
           comm_diagonal); // Canal de comunicacion

// Terminamos la ejecucion de los procesos, despues de esto solo existira
// el proceso 0
// Ojo! Esto no significa que los demas procesos no ejecuten el resto
// de codigo despues de "Finalize", es conveniente asegurarnos con una
// condicion si vamos a ejecutar mas codigo (Por ejemplo, con "if(rank==0)".
MPI_Comm_free(&comm_diagonal);
MPI_Comm_free(&comm_columnas);
MPI_Finalize();

...

delete [] buf_envio;
```

```
  delete [] local_A;
  delete [] local_x;
  delete [] local_y;
  delete [] l_y;
  delete [] x;
  delete [] y;

  return 0;
}
```

2.2. Ejercicio 2:

Realizar medidas de tiempo de ejecución sobre los dos algoritmos implementados de multiplicación matriz-vector (Descomposición 1D y descomposición 2D). Para medir tiempos de ejecución, podemos utilizar la función MPI_Wtime. Para asegurarnos de que todos los procesos comienzan su computación al mismo tiempo, o de que todos han analizado cierta fase de cómputo, podemos utilizar MPI_Barrier. Deberán realizarse las siguientes medidas:

1. Medidas para el algoritmo secuencial (P = 1).
2. Medidas para el algoritmo paralelo (P = 4, 9). Las medidas no incluirán la fase de distribución.

   Las medidas deberán realizarse para diferentes tamaños de problema, para así poder comprobar el efecto de la granularidad sobre el rendimiento de los algoritmos.

   La ganancia ebución inicial de la matriz A y el vector x desde el procesador P0, ni tampoco la fase de reunión del vector resultado y en el procesador P0. n velocidad se calculará dividiendo el tiempo de ejecución del algoritmo secuencial entre el tiempo de ejecución del algoritmo paralelo.

   Se valorará la representación gráfica de los datos numéricos de ganancia en velocidad.

2.2.1. Tabla comparativa.

- Descomposición Unidimensional:

Tiempo	P = 1 (sec.)	P = 4	P = 9	Ganancia(4)	Ganancia(9)
N = 360	0.000308961	0.000164581	0.00010432	1.87725801	2.96166603
N = 720	0.00267937	0.00145511	0.000711647	1.8413522	3.76502676
N = 1080	0,00243283	0.00177601	0.00141641	1.369829	1.71760295
N = 1440	0.0051831	0.00181841	0.00243078	2.85034728	2.13227853
N = 1800	0.00787077	0.00210772	0.00384488	3.73425787	2.04707819

- Gráfica:

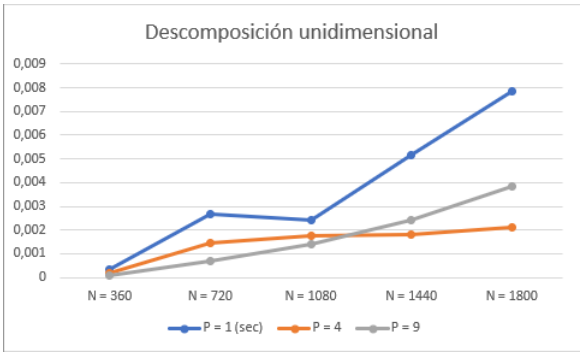

Figura 1: Gráfica descomposición unidimensional.

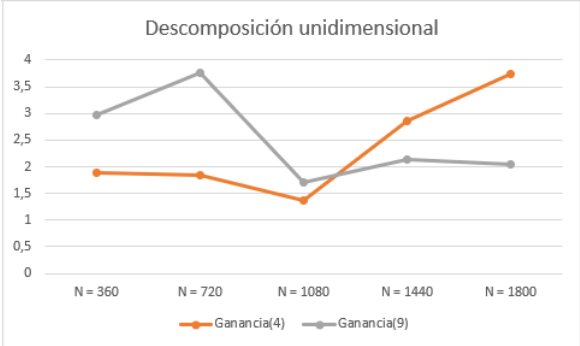

Figura 2: Gráfica ganancia descomposición unidimensional.

- Descomposición Bidimensional:

Tiempo	P = 1 (sec.)	P = 4	P = 9	Ganancia(4)	Ganancia(9)
N = 360	0.00127561	0.000664169	0.00026539	1.92061057	4.80654885
N = 720	0.00241873	0.00157898	0.000330711	1.53183068	7.31372709
N = 1080	0.0033928	0.00112484	0.00151051	3.0162512	2.24612879
N = 1440	0.00494888	0.00204862	0.00267503	2.41571399	1.85002785
N = 1800	0.00765155	0.00263693	0.00398957	2.90168871	1.91788839

- Gráfica:

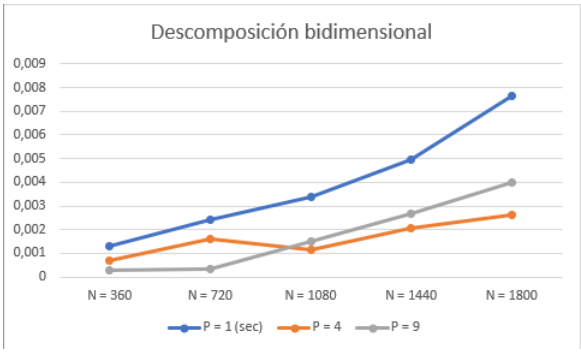

Figura 3: Gráfica descomposición bidimensional.

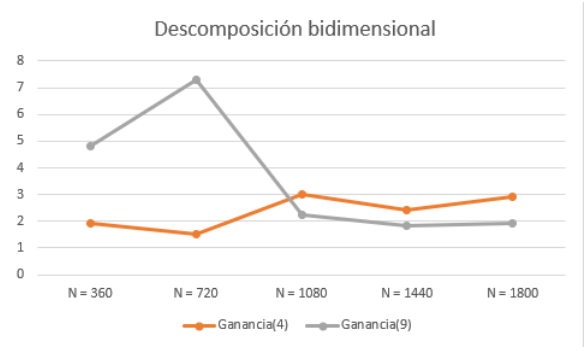

Figura 4: Gráfica ganancia descomposición bidimensional.