PPR

Programación Paralela.

Práctica 3:

Implementación distribuida de un algoritmo de equilibrado dinámico de la carga usando MPI

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: mpicxx
- Compilación de cada programa:
  - mpicxx -O3 -c bbpseq.cc
    mpicxx -O3 -c libbb.cc
    mpicxx -O3 bbseq.o libbb.o -o bbseq
  - mpicxx -O3 -c bbpar.cc
    mpicxx -O3 -c libbb.cc
    mpicxx -O3 bbpar.o libbb.o -o bbpar
- Ejecución: mpirun
- Compilación de cada programa:
  - mpirun bbseq *-n- -archivo-*
  - mpirun -np *-numero_procesos-* bbpar *-n- -archivo-*

2. Equilibrado de carga

1. Si la pila está vacía:

   - Se envía un mensaje utilizando MPI_Send al siguiente proceso con el identificador del proceso como mensaje y con la etiqueta PETICION. Esto se hace para solicitar trabajo cuando el proceso no tiene ninguna tarea asignada.

2. Se inicia un bucle while que se ejecuta mientras la pila está vacía y la variable fin es falsa. Con la finalidad de esperar a que la pila se llene de nuevo o se detecte una posible situación de finalización.

   - Se utiliza MPI_Probe para sondear cualquier fuente de mensaje y cualquier etiqueta de mensaje en el comunicador global MPI_COMM_WORLD. Esto permite verificar si hay algún mensaje pendiente de recibir.
   - Se utiliza un switch para manejar diferentes etiquetas de mensaje (MPI_TAG) que pueden llegar al proceso.
     - Si la etiqueta de mensaje es PETICION, se recibe el identificador del proceso solicitante utilizando MPI_Recv, y se envía una respuesta al siguiente proceso con una nueva solicitud de trabajo. Si el proceso solicitante es el proceso actual, se inicia la detección de una posible situación de finalización, qeu se implementará más adelante.
     - Si la etiqueta de mensaje es NODOS, se utiliza MPI_Get_count para obtener el número de elementos recibidos en el mensaje y se almacena en count. Luego, se recibe el contenido de los nodos (pila.nodos) desde la fuente especificada en el estado del mensaje (estado.MPI_SOURCE).

3. Después de salir del bucle while, se verifica si fin sigue siendo falso. Si es así, se continúa con el siguiente código:

   - Se utiliza MPI_Iprobe para verificar si hay algún mensaje pendiente de recibir. Si hay mensajes pendientes (etiqueta > 0), se ejecuta:
     - Se recibe el identificador del proceso solicitante utilizando MPI_Recv. Si el tamaño de la pila es mayor o igual a 2, se divide la pila en dos (pila.divide(pila_mitad)) y se envía la mitad de la pila (pila_mitad.nodos) al proceso solicitante utilizando MPI_Send con la etiqueta NODOS. Si la pila tiene menos de 2 elementos, se envía una nueva solicitud de trabajo al siguiente proceso.

4. Se vuelve a utilizar MPI_Iprobe para verificar si hay más mensajes pendientes. Si hay más mensajes, el bucle se repite hasta que no haya más mensajes pendientes.

2.1. Implementación del código.

```
/* ********************************************************************** */
/* ******************** Funcion Equilibrado_Carga ********************** */
/* ********************************************************************** */
void  Equilibrado_Carga(tPila &pila, bool &fin){
  MPI_Status estado;
  int solicitante;

  if(pila.vacia()){
    MPI_Send(&rank, 1, MPI_INT, ((rank+1) % size), PETICION, MPI_COMM_WORLD);

    while(pila.vacia() && !fin){
      MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &estado);

      switch(estado.MPI_TAG){
        case PETICION:
          MPI_Recv(&solicitante, 1, MPI_INT, ((rank-1 + size) % size),
                   PETICION, MPI_COMM_WORLD, &estado);

          MPI_Send(&rank, 1, MPI_INT, ((rank+1) % size), PETICION,
                   MPI_COMM_WORLD);
          if(solicitante == rank){
            //Iniciar deteccion de posible situacion de fin;
          }
          break;
        case NODOS:
          int count;
          MPI_Get_count(&estado, MPI_INT, &count);
          MPI_Recv(pila.nodos, count, MPI_INT, estado.MPI_SOURCE, NODOS,
                   MPI_COMM_WORLD, &estado);

          pila.tope = count;
          break;
      }
    }
  }

  if(!fin){
    int etiqueta;
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &etiqueta,
               &estado);

    while(etiqueta > 0){
      MPI_Recv(&solicitante, 1, MPI_INT, ((rank-1 + size) % size), PETICION,
               MPI_COMM_WORLD, &estado);
```

```
      if(pila.tamanio() >= 2){
        tPila pila_mitad;
        pila.divide(pila_mitad);
        MPI_Send(pila_mitad.nodos, pila_mitad.tope, MPI_INT, solicitante,
                 NODOS, MPI_COMM_WORLD);
      } else{
        MPI_Send(&solicitante, 1, MPI_INT, ((rank+1) % size), PETICION,
                 MPI_COMM_WORLD);
      }
      MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &etiqueta,
                 &estado);
    }
  }
}
```

3. Equilibrado de carga + Detección de fin

En esta segunda implementación de la función "Equilibrado_Carga" el código amplía la implementación de equilibrado de carga para incluir un algoritmo de detección de fin mediante el uso de un token. Los procesos intercambian el token entre sí para determinar si se ha alcanzado la condición de terminación global. Además, también se introduce la variable "solucion" para almacenar la solución óptima encontrada durante el proceso de equilibrado de carga, y poder mostrar el resultado por pantalla.

1. Se verifica si la pila está vacía. Si está vacía, se envía un mensaje MPI al siguiente proceso con el ID del proceso actual (id_proceso) y la etiqueta PETICION.

2. Luego, se inicia un bucle while que se ejecuta mientras la pila esté vacía y "fin" sea falso.

   - Se utiliza la función MPI_Probe para esperar un mensaje entrante en cualquier etiqueta y de cualquier origen utilizando el comunicador "comunicadorCarga".
   - Se utiliza una declaración switch para manejar diferentes etiquetas de mensajes MPI recibidos en la variable "estado.MPI_TAG".
     - Si la etiqueta es PETICION, se recibe el ID del proceso solicitante y se envía un mensaje de petición al siguiente proceso. Si el solicitante es igual al proceso actual (id_proceso), se establece el estadoProc en PASIVO. Si hay un token presente, se realiza un intercambio de tokens entre procesos.
     - Si la etiqueta es TOKEN, se recibe un mensaje de token del proceso siguiente. Se marca la presencia del token y, si el estadoProc es PASIVO, se realiza una verificación para determinar si se ha detectado la terminación. Si el proceso actual es el proceso 0 y el color asignado al proceso 0 y el color del token son BLANCO, se establece "fin" en verdadero y se envía un mensaje de terminación al siguiente proceso. Además, se recibe un mensaje de terminación del proceso anterior y se compara con la solución actual para actualizarla si es necesario.
     - Si la etiqueta es FIN, se recibe un mensaje de terminación del proceso anterior. Se establece "fin" en verdadero y se compara la solución recibida con la solución actual para actualizarla si es necesario. Luego, se envía un mensaje de terminación al siguiente proceso.
     - Si la etiqueta es NODOS, se obtiene la cantidad de elementos (count) del mensaje y se alamacenan en la pila. Se establece el estadoProc en ACTIVO.

3. Después del bucle while, se verifica si "fin" es falso. Si es falso, se utiliza la función MPI_Iprobe para verificar si hay mensajes entrantes. Si se encuentra un mensaje con la etiqueta PETICION, se recibe el solicitante y se realiza una comprobación adicional.

   - Si la pila tiene al menos dos elementos, se divide la pila en dos partes y se envía al proceso solicitante utilizando la etiqueta NODOS.
   - Si la pila tiene menos de dos elementos, se envía un mensaje de petición al siguiente proceso.

3.1. Implementación del código.

```
/* ********************************************************************** */
/* ******************** Funcion Equilibrado_Carga ********************** */
/* ********************************************************************** */
void  Equilibrado_Carga(tPila &pila, bool &fin, tNodo &solucion){
  MPI_Status estado;
  int solicitante,
      count;
  tNodo solucion_temporal;

  color = BLANCO;
  anterior  = (id_proceso-1 + size) % size,
  siguiente = (id_proceso+1) % size;

  if(pila.vacia()){
    MPI_Send(&id_proceso, 1, MPI_INT, siguiente, PETICION, comunicadorCarga);

    while(pila.vacia() && !fin){
      MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comunicadorCarga, &estado);

      switch(estado.MPI_TAG){
        case PETICION:
          MPI_Recv(&solicitante, 1, MPI_INT, anterior, PETICION,
                   comunicadorCarga, &estado);

          MPI_Send(&solicitante, 1, MPI_INT, siguiente, PETICION,
                   comunicadorCarga);

          if(solicitante == id_proceso){
            estadoProc = PASIVO;
            if(token_presente){
              if(id_proceso == 0)
                color_token = BLANCO;
              else
                color_token = color;

              MPI_Send(NULL, 0, MPI_INT, anterior, TOKEN, comunicadorCarga);
              token_presente = false;
              color = BLANCO;
            }
          }
          break;

        case TOKEN:
          MPI_Recv(NULL, 0, MPI_INT, siguiente, TOKEN, comunicadorCarga,
                   &estado);
          token_presente = true;
```

```
        if(estadoProc == PASIVO){
          if(id_proceso==0 && color == BLANCO && color_token == BLANCO){
            //TERMINACION DETECTADA
            fin = true;
            MPI_Send(solucion.datos, Tamanio(&solucion), MPI_INT, siguiente,
                     FIN, comunicadorCarga);
            MPI_Recv(solucion_temporal.datos, Tamanio(&solucion_temporal),
                     MPI_INT, anterior, FIN, comunicadorCarga, &estado);

            if(solucion_temporal.ci() < solucion.ci()){
              CopiaNodo(&solucion_temporal, &solucion);
            }
          }
        } else{
          if(id_proceso == 0)
            color_token = BLANCO;
          else
            color_token = color;

          MPI_Send(NULL, 0, MPI_INT, anterior, TOKEN, comunicadorCarga);
          color = BLANCO;
          token_presente = false;
        }
        break;

      case FIN:
        MPI_Recv(solucion_temporal.datos, Tamanio(&solucion_temporal),
                 MPI_INT, anterior, FIN, comunicadorCarga, &estado);
        fin = true;

        if(solucion_temporal.ci() < solucion.ci()){
          CopiaNodo(&solucion_temporal, &solucion);
        }
        MPI_Send(solucion.datos, Tamanio(&solucion), MPI_INT, siguiente,
                 FIN, comunicadorCarga);
        break;

      case NODOS:
        MPI_Get_count(&estado, MPI_INT, &count);
        MPI_Recv(pila.nodos, count, MPI_INT, estado.MPI_SOURCE, NODOS,
                 comunicadorCarga, &estado);
        pila.tope = count;
        estadoProc = ACTIVO;
        break;
    }
  }
}
```

```
  if(!fin){
    int etiqueta;
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, comunicadorCarga, &etiqueta,
               &estado);

    while(etiqueta > 0){
      MPI_Recv(&solicitante, 1, MPI_INT, anterior, PETICION, comunicadorCarga,
               &estado);

      if(pila.tamanio() >= 2){
        tPila pila_mitad;
        pila.divide(pila_mitad);

        MPI_Send(pila_mitad.nodos, pila_mitad.tope, MPI_INT, solicitante,
                 NODOS, comunicadorCarga);
      } else{
        MPI_Send(&solicitante, 1, MPI_INT, siguiente, PETICION,
                 comunicadorCarga);
      }
      MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, comunicadorCarga, &etiqueta,
                 &estado);
    }
  }
}
```

4. Difusión de la cota superior

1. Se define una variable booleana llamada "hay_nueva_cota_superior" e inicializada como falso. Esta variable se utilizará para indicar si hay una nueva cota superior durante la difusión.

2. Si la variable "difundir_cs_local" es verdadera y no hay un retorno de cota superior pendiente, se envía el valor de "U" (cota superior) al proceso siguiente utilizando la función MPI_Send(). Luego, se establece la variable "pendiente_retorno_cs" como verdadera y "difundir_cs_local" como falso.

3. Se utiliza la función MPI_Iprobe() para verificar si hay algún mensaje recibido pendiente del proceso anterior con cualquier etiqueta.

4. Si hay mensajes recibidos pendientes, se utiliza la función MPI_Recv() para recibir el valor de la nueva cota superior en la variable "nueva_cota" desde el proceso anterior.

5. Si la nueva cota superior ("nueva_cota") es menor que la cota superior actual ("U"), se actualiza "U" con el nuevo valor y se establece la variable "hay_nueva_cota_superior" como verdadera.

6. Si el origen del mensaje es el proceso actual y la variable "difundir_cs_local" es verdadera, se envía el valor actualizado de "U" al proceso siguiente utilizando la función MPI_Send(). Luego, se establece "pendiente_retorno_cs" como verdadera y "difundir_cs_local" como falso.

7. Si el origen del mensaje es el proceso actual pero "difundir_cs_local" es falso, se establece "pendiente_retorno_cs" como falso, lo que indica que no hay un retorno pendiente.

8. Si el origen del mensaje no es el proceso actual, se envía el valor actualizado de "U" al proceso siguiente.

9. Se repite desde el tercer punto hasta que no haya más mensajes pendientes.

10. Al final, se devuelve el valor de "hay_nueva_cota_superior".

4.1. Implementación del código.

```
/* ********************************************************************** */
/* ****************** Funcion Difusion_Cota_Superior  ******************* */
/* ********************************************************************** */
bool Difusion_Cota_Superior(int &U){
  MPI_Status estado;
  bool hay_nueva_cota_superior = false;

  int nueva_cota,
      etiqueta,
      origen_mensaje;
```

```
  anterior  = (id_proceso-1 + size) % size,
  siguiente = (id_proceso+1) % size;

  if(difundir_cs_local && !pendiente_retorno_cs){
    MPI_Send(&U, 1, MPI_INT, siguiente, id_proceso, comunicadorCota);
    pendiente_retorno_cs = true;
    difundir_cs_local = false;
  }

  MPI_Iprobe(anterior, MPI_ANY_TAG, comunicadorCota, &etiqueta, &estado);
  origen_mensaje = estado.MPI_TAG;

  while(etiqueta > 0){
    MPI_Recv(&nueva_cota, 1, MPI_INT, anterior, origen_mensaje,
             comunicadorCota, &estado);

    if(nueva_cota < U){
      U = nueva_cota;
      hay_nueva_cota_superior = true;
    }

    if(origen_mensaje == id_proceso && difundir_cs_local){
      MPI_Send(&U, 1, MPI_INT, siguiente, id_proceso, comunicadorCota);
      pendiente_retorno_cs = true;
      difundir_cs_local = false;
    }else if(origen_mensaje == id_proceso && !difundir_cs_local){
      pendiente_retorno_cs = false;
    }else{
      MPI_Send(&U, 1, MPI_INT, siguiente, origen_mensaje, comunicadorCota);
    }

    MPI_Iprobe(anterior, MPI_ANY_TAG, comunicadorCota, &etiqueta, &estado);
    origen_mensaje = estado.MPI_TAG;
  }

  return hay_nueva_cota_superior;
}
```

5. Estudio experimental

5.1. Tabla comparativa.

- Implementación secuencial (bbseq.cc):

Ciudades	10	20	30	35	40
Tiempo	0,000586317	0,0266513	0,0968124	1,28347	3,6873
Iteraciones	207	3755	6957	71107	158556

  - Gráfica:

Figura 1: Gráfica tiempo-ciudades

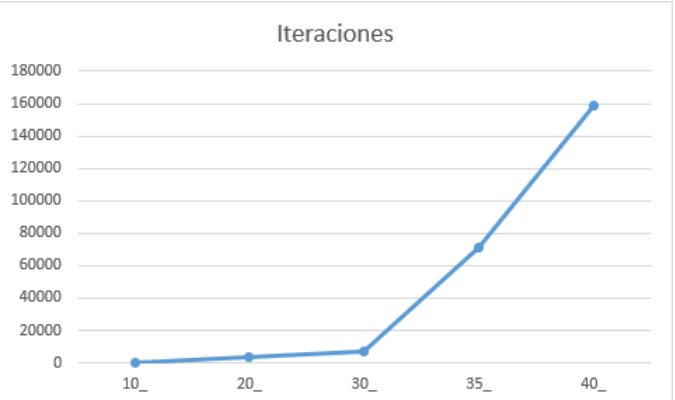

Figura 2: Gráfica iteraciones-ciudades

- Implementación en paralelo 2 procesos (bbpar.cc):

Ciudades	10	20	30	35	40
Tiempo	0,00393239	0,0215866	0,0954288	0,846955	3,93055
Iteraciones P0	155	2463	6484	40962	158843
Iteraciones P1	141	2435	6374	40843	158466
Iteraciones_total	296	4898	12858	81805	317309

  - Gráfica:

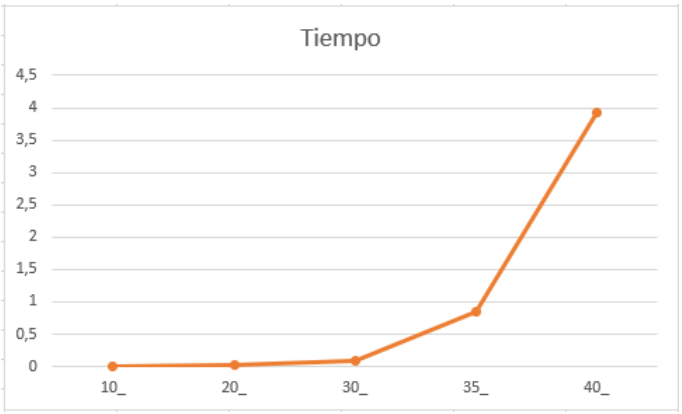

Figura 3: Gráfica tiempo-ciudades

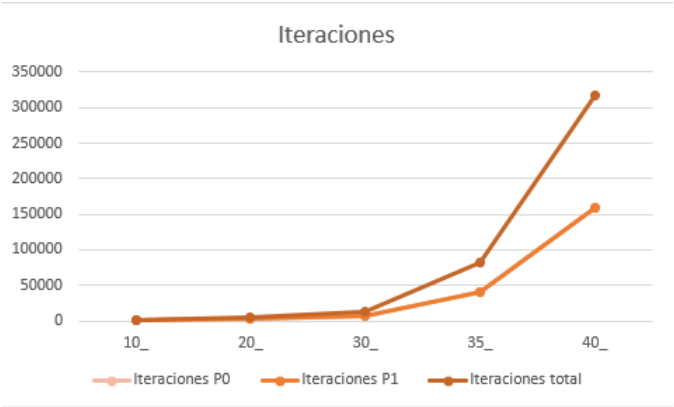

Figura 4: Gráfica iteraciones-ciudades

- Implementación en paralelo 3 procesos (bbpar.cc):

Ciudades	10	20	30	35	40
Tiempo	0,000504443	0,0168948	0,0783965	0,661151	1,83114
Iteraciones P0	115	2179	5064	32242	73485
Iteraciones P1	91	2173	5027	32270	74050
Iteraciones P2	111	2199	5078	32054	68425
Iteraciones_total	317	6551	15169	96566	215960

  - Gráfica:

Figura 5: Gráfica tiempo-ciudades

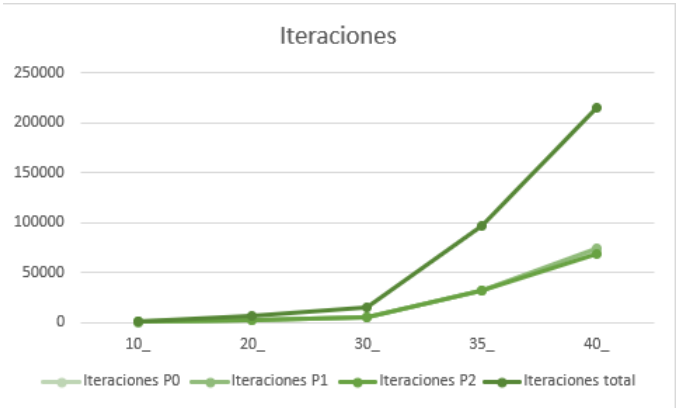

Figura 6: Gráfica iteraciones-ciudades

5.2. Comparativa entre algoritmos.

- Gráfica:
  - Gráfica implementación en paralelo:

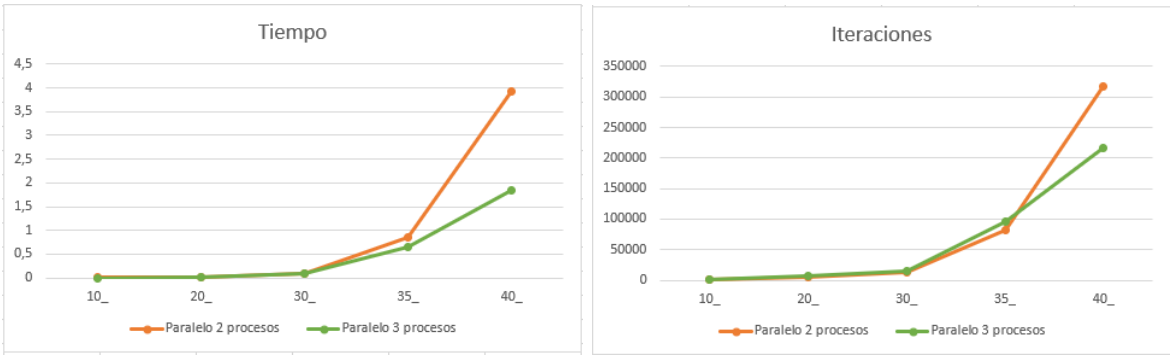

Figura 7: Gráfica comparativa

  - Gráfica comparativa secuencial-paralelo:

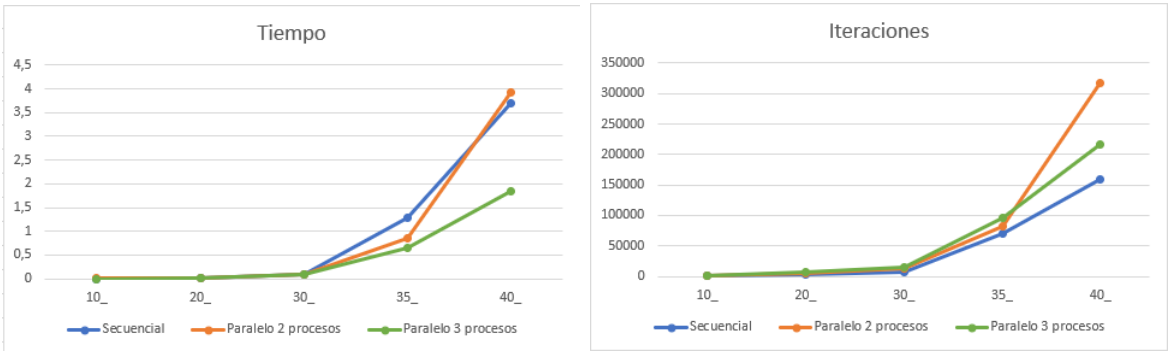

Figura 8: Gráfica comparativa