Arquitectura de Sistemas Paralelos

Práctica 2

Programación MPI Avanzada
Comunicación de Datos

Michael Alexander Fajardo Malacatus
Gloria del Valle Cano
michael.fajardo@estudiante.uam.es & gloria.valle@estudiante.uam.es

Grado en Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
30-03-2020

Índice general

Índice de figuras

Introducción

En esta práctica abordaremos diferentes bloques que nos permitirán estudiar en más profundidad lo que la programación `MPI` nos ofrece. En el primer bloque probaremos mecanismos simples de distribución de carga y estudiaremos el rendimiento de las diferentes estrategias en términos del *speedup*. Después, en el segundo bloque probaremos diferentes nuevos tipos de datos y topologías virtuales mediante funciones que nos ofrece `MPI`. Para finalizar, en el tercer bloque, realizaremos algún ejercicio para aprender de `MPI4py` con el fin de aplicarlo en Python.

Bloque 1: Distribución de carga

Experimental

Para comprobar el *speedup* y el balanceo de carga se han creado 4 ficheros: `ficheroA.txt`, `ficheroB.txt`, `ficheroC.txt`, `ficheroD.txt`. Estos ficheros han sido creados mediante un sencillo programa en Python, `generadorArchivos.py`.
Todos los tiempos parciales que se emplean para realizar el cálculo del factorial de un número se guardan en ficheros con la siguiente nomenclatura:

1. `Fichero_Serie_NombreFichero.txt`: donde NombreFichero es el nombre del fichero con el que se ejecuta, pudiendo ser el A,B,C o D.
2. `Fichero_1_2_NombreFichero.txt`: este tipo de ficheros son los referidos al apartado 1.2.
3. `Fichero_1_3_NombreFichero.txt`: este tipo de ficheros son los referidos al apartado 1.3.
4. `Factorial_Total_NombreFichero.txt`: este tipo de ficheros, contienen el tiempo total empleado para cada cada fichero, de la ejecución en serie, paralelo y paralelo con distribución de carga.

En todos los programas, se miden dos tipos de tiempos:

1. **Tiempo parcial** que tarda el procesador en realizar el factorial del número i.
2. **Tiempo total** que emplea el procesador en realizar el todo el cálculo de todos los números del fichero.

1.1. Programa Factorial con ejecución serie o secuencial

Partiendo del fichero `factorial.c`, se ha modificado este, haciendo que lea los ficheros con los que realizar el factorial.

1.2. Distribución Fija de la carga

Para la distribución fija de la carga se utiliza la comunicación punto a punto, además de seguir el siguiente algoritmo:

1. **Maestro**
   (a) **Leer** el fichero con el que vamos a trabajar.
   (b) Abrir los ficheros en los que guardar los tiempos empleados tanto para el cálculo individual de cada número así como el total que ha sido utilizado para todos los números en conjunto.

(c) **Reparto** del número de filas con las que debe trabajar cada procesador; **dividiendo** el número de **líneas** del fichero, que corresponden a la cantidad de elementos con los que realizar el factorial, entre el número de **procesadores**.

(d) Mediante un **bucle** for se hace el **reparto** de la cantidad de **números** a los que se les calculará el factorial. Para el envío se utiliza la función `MPI_Send`.

(e) En caso de no haber repartido todos los números entre todos los esclavos, se procederá al cálculo de los números restantes.

(f) Se **recogerán** los resultados enviados por los esclavos, según el orden de envío mediante nuevamente un bucle *for* y con la función `MPI_Recv`, comprobando además si se ha recibido adecuadamente el contenido mediante la misma. `MPI_Get_elements` para constatar la cantidad recibida con la esperada.

(g) Los resultados parciales recibidos se escribirán en el fichero correspondiente.

(h) Se imprimirá el tiempo total en el fichero correspondiente.

Dentro del código del **Maestro** se hacen **tres medidas** de tiempo:

(a) Tiempo empleado en enviar el reparto entero a los esclavos.

(b) Tiempo empleado en caso no se haya hecho un reparto total de los números.

(c) Tiempo empleado en la recepción de los resultados enviados por los esclavos.

El tiempo total será la suma de estos tres valores.

2. **Esclavo**

(a) Recoger el **tamaño** del mensaje a recibir a continuación mediante las funciones `MPI_Probe` y `MPI_Get_count`.

(b) Recibir el mensaje mediante la función `MPI_Recv`.

(c) Calcular el tiempo dedicado a la realización de cada factorial y guardarlo en un array.

(d) Enviar al maestro a través de dos funciones `MPI_SSend` el array de los tiempos y el array de números con los que se ha trabajado.

Se hace uso de la función `MPI_SSend` ya que es bloqueante al emisor y se evita que se pierdan los valores, ya que al final del esclavo se libera la memoria reservada para todo el cómputo.

1.3. Distribución dinámica de la carga a demanda

Para la distribución dinámica de la carga a demanda nuevamente se ha utilizado una comunicación punto a punto. Sin embargo el algoritmo es diferente al del apartado anterior:

1. **Maestro**

(a) **Leer** el fichero con el que trabajar.

(b) Abrir los ficheros en los que guardar los tiempos empleados tanto para el cálculo individual de cada número así como el total que ha sido utilizado para todos los números en conjunto.

(c) **Envío** de un número a cada procesador.

(d) Mediante un bucle *while* se hace la recepción del tiempo y el número enviados por el primer esclavo en terminar, con la función `MPI_Recv`.

(e) Los resultados parciales recibidos, se escribirán en el fichero correspondiente.

(f) **Envío** del siguiente número a calcular,con `MPI_Send` al esclavo que haya terminado primero.

(g) Se **recogerán** los resultados enviados por los esclavos, según el orden de envío mediante nuevamente un bucle *for* y con la función `MPI_Recv`.
Motivo: al principio se envía un número a cada procesador, y a continuación en el bucle se hace el envío al primero en acabar por tanto siempre quedarán n-2 recv que hacen falta. Además por la forma en la que se ha programado el bucle *while*, termina enviando el último número sin recogerlo por tanto pasan a ser n-1 los *recieve* que hacen falta.

(h) Se imprimirá el tiempo total en el fichero correspondiente.

Dentro del código del **Maestro** se hacen **dos medidas** de tiempo:

(a) Tiempo empleado en enviar el reparto de un número a cada esclavo.

(b) Tiempo empleado en el cómputo de los números restantes.

El tiempo total será la suma de estos dos valores.

2. **Esclavo**

(a) Crear bucle infinito en el que se saldrá sólo cuando se reciba un -1 por parte del maestro.

(b) Recibir el número mediante la función `MPI_Recv`

(c) Calcular el tiempo dedicado a la realización de cada factorial y

(d) enviar al maestro el tiempo y número a través de dos funciones `MPI_SSend`

Se hace uso de la función `MPI_SSend` ya que es bloqueante al emisor y se evita que se pierdan los valores.

Conclusiones del primer bloque

Se ha tomado como prueba de mejor *speedup* la siguiente figura:

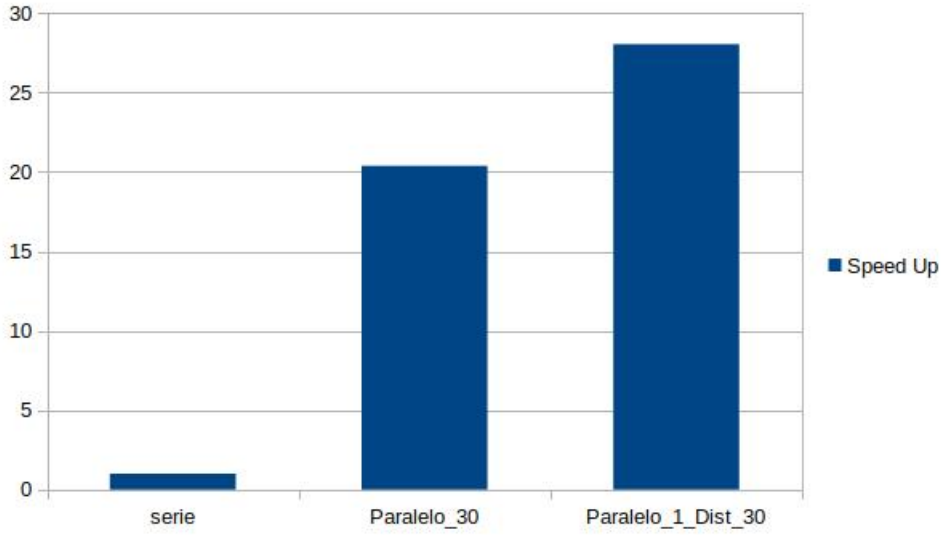

Figura 1: Speedup (primer bloque)

Esta figura corresponde con la ejecución del archivo con el mayor de números grandes. Como se puede observar el *speedup* conseguido por la paralelización es de 20 veces más que la ejecución en serie e incluso casi 30 veces más en caso del balanceo de carga dinámica. Esto se debe a que aunque se haga un reparto entre los esclavos en el balanceo de carga estático, interiormente se ejecuta secuencialmente con lo que tiempo que tarda es del más lento de todos ellos.

Esto se soluciona con el balanceo de carga dinámico, debido a que siempre se le ofrece tarea al primero que termine, por lo que se dará mas carga al esclavo más rápido y esto hará que los más lentos tenga menos carga de trabajo y por tanto el rendimiento es notoriamente superior que el balanceo de carga fijo.

Bloque 2: Tipos de datos derivados

Nota: los nombres de variables y el diseño de los ejercicios 2.1, 2.2, 2.3 y 2.4, se corresponden con los diagramas del enunciado.

2.1. Contiguous Derived Data Type

Este tipo de dato puede ser útil para hacer eficiente el envío de mensajes que contienen datos contiguos, por ejemplo de diversos tipos, transformándolo en una especie de *typemap*. Usando este tipo de dato podemos retornar una fila de una matriz, distribuyendo una fila diferente a cada uno de los procesos. Para este fin, hemos recurrido a las siguientes funciones:

- `int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)`: para crear este tipo de dato, en nuestro caso, `rowtype`.
- `int MPI_Type_commit(MPI_Datatype *datatype)`: para confirmar el nuevo tipo de dato.
- `int MPI_Type_free(MPI_Datatype *datatype)` para liberar el dato.

De esta manera, en `ejercicio2.1.c`, hemos llevado a cabo un envío y recepción bloqueantes, enviando cada una de las filas.

2.2. Vector Derived Data Type

La diferencia con respecto del anterior caso, es que aquí se utiliza un tipo de dato diferente con el fin de extraer las columnas:
`int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)`
Este tipo de dato consiste en una serie de elementos del mismo tipo de datos repetidos con un determinado avance o salto, lo que nos permite extraer `columntype` más fácilmente.
(`ejercicio2.2.c`)

2.3. Indexed Derived Data Type

En este caso se ha creado `indextype` con el fin de poder extraer porciones variables de una matriz (de tamaño 6), distribuyéndolo a todas las tareas.
`int MPI_Type_indexed(int count, int blocklens[], int indices[], MPI_Datatype old_type, MPI_Datatype *newtype);`
(`ejercicio2.3.c`)

2.4. Struct Derived Data Type

Este ejercicio se ha realizado de la misma manera que los anteriores pero con la novedad de poder pasar estructuras de datos, en nuestro caso, `Particle`. Nos hemos declarado un nuevo tipo de dato, `particletype`, y `extent`, una constante `MPI_Aint`, para obtener extent del tipo de dato.
`int MPI_Type_struct(int count,`
`const int *array_of_blocklengths, const MPI_Aint *array_of_displacements, const MPI_Datatype *array_of_types, MPI_Datatype *newtype)`: para la construcción del tipo de dato (estructura).
`int MPI_Type_extent(MPI_Datatype datatype,MPI_Aint *extent);`: para la extensión del tipo de dato.
(`ejercicio2.4.c`)

2.5. Topologías virtuales

Las topologías virtuales se basan en comunicadores y grupos MPI. En concreto, las cartesianas son útiles para aplicaciones con patrones de comunicación específicos, aquellos que coinciden con una estructura de topología MPI. Se pueden representar en forma de cuadrícula (con o sin límites cíclicos), en las que cada proceso está conectado a sus vecinos.
En lo que a este ejercicio se refiere, hemos creado una cuadrícula de dos dimensiones de 4x4 y de tamaño 16 procesadores, haciendo que cada proceso intercambie su rango con cuatro vecinos. Para ello, hemos hecho uso de las siguientes funciones y tipos de datos necesarios para este fin, además de las necesarias para establecer el paso de información:

- `MPI_Request`: para definir las peticiones para la comunicación.
- `MPI_Comm`: para definir el nuevo comunicador, `c_cartesiana`.
- `int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const int periods[], int reorder, MPI_Comm *comm_cart)`: crea un nuevo comunicador y sincroniza los procesos involucrados.
- `int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])`: convierte el rango de proceso en coordenadas de cuadrícula de proceso, según las dimensiones especificadas.
- `int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)`: para encontrar vecinos del procesador, devolviendo el rango de origen y destino del siguiente, dada una dirección y una número en el que se desplaza.

Dado el tamaño del problema y la necesidad de devolver el control a la ejecución del programa lo antes posible, es necesario que el envío y la recepción sean no bloqueantes.
(`ejercicio2.5.c`)

Bloque 3: Aplicaciones avanzadas de MPI

Conjuntos de clasificadores

Dadas las ventajas de MPI por un lado y de Python por el otro, se ha tratado de aprender a utilizar `MPI4py`. A pesar de encontrar mucha información acerca de esta librería, no hemos sabido resolver en totalidad el problema aplicado al campo de aprendizaje automático, pero no por ello queríamos dejar de intentarlo.
Tratando de resolverlo, hemos creado un clasificador de votos por mayoría *from scratch*, ayudándonos de la librería de *sklearn*, cuyo código inicial (VotingClassifier) y con MPI (VotingClassifierMPI) se pueden encontrar en el Notebook adjunto, `VotingClassifierMPI.ipynb`. El código con MPI se encuentra separado en `VotingClassifierMPI.py` (aquí sólo VotingClassifierMPI).

3.4. Implementación del método *fit*

- Para enviar los datos a los esclavos, hemos utilizado `comm.send(classifiers, dest=0, tag=11)` a través del maestro.
- A través de los esclavos, `comm.recv(source=proc, tag=11)`, siendo *proc* el número de procesadores.