PRÁCTICAS
DE
INGENIERÍA
DE LOS
COMPUTADORES

PRÁCTICA 4

PARALELIZACIÓN EN NODOS
Uso de OpenMPI para la paralelización en red

Javier Bellver García (jbg39@alu.ua.es)
Alejandro Reyes Albillar (ara65@alu.ua.es)
Cesar Enrique Pozo Vásquez (cepv1@alu.ua.es)
Curso 2015 / 2016
Grado en Ingeniería Informática

Universitat d'Alacant
Universidad de Alicante

ÍNDICE

1. **Introducción** **Pág. 2**
2. **Estado del Arte** **Pág. 2**
3. **Introducción a OpenMPI** **Pág. 3**
4. **Tipos de paralelización implementados** **Pág. 5**
5. **Mediciones** **Pág. 7**
6. **Conclusión** **Pág. 9**
7. **Bibliografía** **Pág. 10**

1. Introducción

En esta última práctica se nos ha planteado que utilicemos OpenMPI para realizar la paralelización de nuestro proyecto propuesto de la segunda práctica haciendo uso de los mecanismos de paralelización en nodos en red que nos proporciona la API de OpenMPI.

OpenMPI, de la que hablaremos en detalle en una de las siguientes secciones de la práctica, es una librería que nos permite realizar una paralelización en nodos en red. Esto significa que, a diferencia de la paralización con OpenMP, la cual tratamos en la anterior práctica y que paraleliza dentro de un mismo procesador mediante hilos de ejecución, OpenMPI realiza una paralelización multicomputador. Con esto nos referimos a que cada nodo de la red es un proceso distinto en un ordenador que está conectado a dicha red.

En esta práctica trataremos el problema del cálculo de la covarianza de matrices, como hicimos en las prácticas anteriores. Debido a la naturaleza de OpenMPI y la diferencia entre este y, la paralelización de hilos frente a la paralelización en nodos ha hecho que para esta práctica hayamos optado por un enfoque diferente, centrándonos únicamente en la paralelización de datos. Más adelante explicaremos cómo hemos enfocado esta práctica.

2. Estado del arte

Con respecto al estado del arte, tras revisar otros trabajos expuestos en internet, nos damos cuenta que la mayor parte de los intentos de paralelización de este problema se dan a nivel de hilos en un sólo procesador. De este modo se han desarrollado diferentes soluciones basadas en una multitud de arquitecturas de procesador.

Muchas de estas soluciones tratan de corregir ciertos problemas del cálculo de la covarianza, como la repetición de operaciones como multiplicaciones, uso eficiente de la memoria, división del trabajo entre los diferentes hilos y la sincronización de estos hilos. Además, muchas de estas soluciones han dado muy buen resultado en la resolución de este problema.

Sin embargo, no vemos realmente un gran trabajo hecho en la paralelización del problema mediante diferentes nodos en una red dedicada. De esto podríamos deducir que, aunque no resultan prácticas por cuestiones de rendimiento probaremos en nuestra práctica si este enfoque es en verdad realmente útil o si, por el contrario, es algo inútil que no podemos aplicar en un escenario real.

3. Introducción a OpenMPI

A continuación realizaremos una breve explicación de la libería de OpenMPI para la paralelización en nodos en red. Openmpi es una librería de paso de mensajes que implementa el estándar MPI-2. Al trabajar muy bien con C++, esta librería nos es conveniente, ya que mediante el uso de llamadas a funciones de la librería se obtiene una implementación sencilla de un programa basado en el estándar MPI.

Para hablar de OpenMPI, primero debemos tratar de que es MPI, que es un estándar de Interfaz de paso de mensajes. Este estándar ha sido ideado por el MPI Forum, un foro de 40 organizaciones de los ámbitos de la investigación, venta, desarrollo y usuarios de estas librerías de paso de mensajes. Aunque no es un estándar formal del IEEE o de la ISO, MPI se ha convertido en el estándar de la industria con respecto a este tipo de librerías. Este estándar ayuda a cumplir los objetivos de portabilidad, flexibilidad y eficiencia que la industria requiere.

Habiendo dejado ya claro que es MPI pasaremos ahora a describir pues, que es OpenMPI. OpenMPI es una librería “thread-safe”, implementación del estándar MPI-2 y de código abierto, la cual está desarrollada y mantenida por un conjunto de colectivos de la industria y académicos.

OpenMPI nos dota de diferentes niveles de soporte de multi-threading. Esto nos permite elegir, en caso de que queramos ejecutar un programa en OpenMPI, que esté paralelizado en múltiples hilos y determinar si dejamos a estos hilos ejecutar directivas de OpenMPI y con qué restricciones.

Ahora pasaremos a explicar algunos de los conceptos más importantes de OpenMPI mediante pequeños ejemplos prácticos.

Para hacer funcionar un programa en OpenMPI, lo primero que debemos hacer será importar la librería concreta en nuestro programa. En nuestro caso, ya que el programa de cálculo de covarianza de matrices está hecho en C++, importamos la librería mediante la directiva (#include “mpi.h”).

Dos conceptos realmente importantes sobre OpenMPI son los conceptos de grupos y comunicadores. Estos dos conceptos definen los grupos de procesos con los que podemos comunicarnos, definiendo aquellos a los que podemos enviar y desde los que podemos recibir mensajes. La mayor parte de rutinas de OpenMPI requieren de un comunicador para especificar el grupo concreto de procesos que estamos utilizando.

Otro concepto realmente importante, y relacionado con los dos conceptos que acabamos de tratar, es el concepto de rango. El rango es un identificador único a cada proceso, que es asignado cuando el proceso es iniciado. Los rangos también suelen llamarse “Task IDs” y se asignan de forma continua con un entero que se inicia desde el 0.

Con respecto al manejo de errores en el entorno de OpenMPI, la mayor parte de rutinas de OpenMPI tienen un return con el que devuelven un código de error. Sin embargo, cuando hay un error en MPI, el comportamiento normal del programa es, de hecho, cerrarse. Con lo cual, de forma normal no se pueden recuperar estos códigos de error. Este comportamiento puede ser sobreescrito para poder capturar estos errores.

En OpenMPI existen dos tipos de comunicación primordiales utilizados para el paso de mensajes. El primer tipo de comunicación es la comunicación punto a punto en el cual los procesos intercambian mensajes los unos con los otros a pares, enviando y recibiendo de un proceso a otro. El segundo tipo de comunicación es grupal, donde se pasan mensajes entre todos los procesos de diferentes tipos.

Dentro de la comunicación de punto a punto existen diversos subtipos de comunicación. Estos tipos son:

- Synchronous Send
- Blocking send / blocking receive
- Non-blocking send / non-blocking receive
- Buffered send
- Combined send/receive
- “Ready" send

Estos diversos modos nos permiten realizar diversos tipos de comunicación tanto síncrona como asíncrona, dándonos un amplio rango de tipos de comunicación que podemos utilizar en nuestro programa.

Con respecto a la comunicación grupal, también tenemos varios tipos de comunicación y operaciones disponibles, los cuales agrupamos en 3 tipos:

- **Synchronization** -Los procesos esperan a que todos los procesos hayan completado una tarea
- **Data Movement** - Broadcast, gather-scatter etc. .
- **Collective Computation** - Los procesos pasan ciertos datos a otro proceso que realiza una operación con estos.

4. Tipos paralelismo implementados

En este apartado analizaremos el tipo de paralelización que hemos realizado en nuestro programa utilizando la API de OpenMPI. Mostraremos este tipo de paralelización junto a diferentes trozos de nuestro código que nos servirán para explicar el tipo de paralelización en concreto y nuestra decisión de utilizarla para la resolución de nuestro problema.

Antes que nada, como ya realizamos en la anterior práctica, hemos de hacer un breve apunte con respecto a la compilación. Ya que nuestro proyecto está realizado en C++, es muy importante que, a la hora de compilar, utilicemos la directiva mpic++. Esta compilación nos permite crear un binario que es posible ejecutar en el entorno de OpenMPI.

Para esta práctica hemos utilizado el paralelismo de datos. Tras valorar otras opciones, como distribuir la carga de operaciones del cálculo de una sola matriz, caímos en la cuenta de que esta opción podría resultar en una pérdida de tiempo debido al desplazamiento de tal tamaño de datos a través de la red. Por tanto, los tipos de paralelismo aplicados en la práctica anterior resultaban inviables.

Teniendo en cuenta esto, el paralelismo de datos se nos presentaba como la mejor opción para la realización de la práctica en OpenMPI. El concepto es sencillo, en lugar de tener un único nodo que realiza las operaciones con paralelismo a nivel de bucle, como en nuestra implementación con OpenMP, o una implementación secuencial en un nodo, lo que haremos será dividir la carga de trabajo de cada nodo haciendo que este realice sólo cierto número de cálculos, dividiendo el archivo de input entre nuestros nodos.

Aquí está el código de nuestra implementación en este acercamiento al paralelismo en nodos:

```
int main(int argc,char* argv[])
{
  if(argc == 2)
  {
    int init=0, num_nodos=0, rank=0;
    init = MPI_Init(&argc,&argv);
    if (init != MPI_SUCCESS) {
      printf ("Error iniciando el programa Covarianza matrices mpi\n");
      MPI_Abort(MPI_COMM_WORLD, init);
    }

    MPI_Comm_size(MPI_COMM_WORLD, &num_nodos);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    clock_t start;
    if(rank == 0) { //Proceso master
      start = clock();
      cout << "Tiempo de comienzo de la aplicacion: " <<CPU_time(start) <<endl;
      start = clock();
    }
    else { //Proceso slave
      readMatrices(argv[1]);
      unsigned int i=0;
      vector< vector <double> > resuelta;
      for(i=0;i<matrices.size();i++)
      {
          compute_covariance_matrix(matrices[i],resuelta);
          resuelta.clear();
      }
    }
    MPI_Barrier(MPI_COMM_WORLD);
    cout << "Tiempo de finalización de la aplicación: " << (double)CPU_time(start) <<endl;
    MPI_Finalize();
  }else{
    cout << "Número de parámetros incorrecto" <<endl;
    cout << "Formato $./main nombre_de_fichero" <<endl;
    return -1;
  }
  return 0;
```

Como podemos observar, lo primero que hacemos es iniciar el contexto de OpenMPI y comprobar que este se ha ejecutado correctamente, en caso de que, por cualquier razón, mpi no se haya ejecutado correctamente, abortamos el programa.

Después, obtenemos el número de nodos que van a ejecutarse y el rango del proceso actual. En caso de que el rango sea 0, y por tanto sea el proceso master el que está ejecutando, indicamos el tiempo de inicialización del programa y esperamos en la barrera.

En caso de que el proceso sea un proceso esclavo, este lo que hace es leer su parte de la matrices a computar, realiza el proceso de cálculo y luego espera en la barrera a que el resto de procesos hayan terminado. Cuando todos los procesos llegan a la barrera, esta se desbloquea y se finaliza el programa.

5. Mediciones

En este apartado realizaremos mediciones necesarias para probar como rinde nuestra aplicación en OpenMPI, comparándola con la medición secuencial que utilizamos en la práctica 2. Todas las medidas están representadas en segundos.

Las mediciones se han realizado sobre los ordenadores del laboratorio de la EPS IV. Para el caso de la ejecución de OpenMPI, hemos realizado diversas mediciones con diferente número de nodos para comprobar el rendimiento en distintas situaciones. Tenemos que tener en cuenta que la topología de la red, en este caso, es crucial para entender los resultados. La comunicación entre los nodos tiene un papel importantísimo a la hora de los resultados, por tanto estos dependen también en gran medida de la red por la que están interconectados.

Todos los ordenadores del laboratorio que hemos utilizado tienen las siguientes especificaciones:

- Intel(R) Pentium(R) CPU G840 @ 2.80GHz
- Memoria RAM: 4010MiB
- Sistema operativo Linux-Ubuntu

Primero la ejecución **secuencial** en un pc del laboratorio:

Ejecución 1	Ejecución 2	Ejecución 3	Ejecución 4	Ejecución 5	Media
1’13.445’’	1’13.142’’	1’13.342’’	1’12.325’’	1’12.570’’	1’12.9648’’

Aquí tenemos los tiempos de ejecución con **dos** nodos:

Ejecución 1	Ejecución 2	Ejecución 3	Ejecución 4	Ejecución 5	Media
36.589’’	36.586’’	36.578’’	37.006’’	36.936’’	36.739’’

Aquí tenemos los tiempos de ejecución con **tres** nodos:

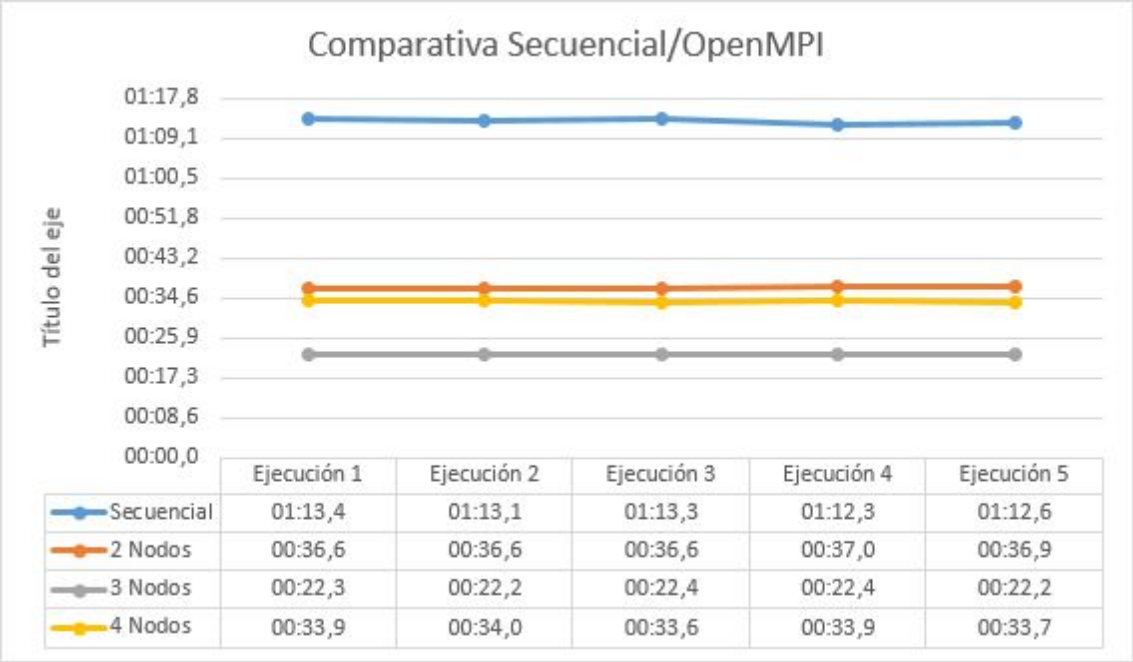
Ejecución 1	Ejecución 2	Ejecución 3	Ejecución 4	Ejecución 5	Media
22.259’’	22.182’’	22.427’’	22.370’’	22.190’’	22.2856’’

Y finalmente aquí tenemos los tiempos de ejecución con **cuatro** nodos:

Ejecución 1	Ejecución 2	Ejecución 3	Ejecución 4	Ejecución 5	Media
33.929''	33.999''	33.586''	33.873''	33.729''	33.8232''

Como podemos comprobar, la implementación paralelizada mediante OpenMPI es bastante más rápida, en cualquiera de los casos, que la implementación secuencial, siendo óptima cuando paraleliza en tres nodos, a partir de cuatro nodos vemos peores tiempos que en tres debido a la distribución desigual de la carga en esta configuración.

Mostramos a continuación una gráfica comparativa de los tiempos anteriormente mostrados.



	Ejecución 1	Ejecución 2	Ejecución 3	Ejecución 4	Ejecución 5
Secuencial	01:13,4	01:13,1	01:13,3	01:12,3	01:12,6
2 Nodos	00:36,6	00:36,6	00:36,6	00:37,0	00:36,9
3 Nodos	00:22,3	00:22,2	00:22,4	00:22,4	00:22,2
4 Nodos	00:33,9	00:34,0	00:33,6	00:33,9	00:33,7

La tabla muestra los datos en el formato mm:ss,ms.

El speedup comparativo entre la implementación secuencial y la implementación paralela según el número de nodos es:

Speedup_dos = Latencia_secuencial / Latencia_dos = 1.12.9648 / 36.739 = 1.98603

Speedup_tres = Latencia_secuencial / Latencia_tres = 1.12.9648 / 22.2856 =3.27407

Speedup_cuatro = Latencia_secuencial / Latencia_cuatro =1.12.9648 / 33.8232 =2.15724

6. Conclusión

Acabaremos la memoria con unas breves conclusiones sobre nuestro trabajo y el uso de OpenMPI para la resolución del problema, el cálculo de la covarianza de un gran conjunto de matrices.

Empezábamos la práctica con desconocimiento de si esta sería correctamente paralelizable en un entorno en red y, aunque es cierto que la paralelización que realizamos en la práctica anterior no resulta óptima en este entorno, lo que sí que es cierto es que la paralelización de datos sobre este entorno multicomputador resulta en una gran ganancia de tiempo con respecto a la implementación secuencial en un sólo nodo.

Es quizás algo sorprendente el resultado que hemos obtenido en los tiempos al utilizar cuatro nodos, sin embargo dos causas importantes dan una explicación a lo que puede haber pasado. La primera es que, con cuatro nodos, la muestra de matrices que estamos utilizando para las pruebas resulta desigual, lo cual puede ralentizar la aplicación ya que el resto de procesos deben esperar a aquellos que se quedan atrás para que esta finalice. La segunda es el entorno de pruebas en el que hemos estado, ya que hemos realizado las pruebas el mismo día que los otros grupos y las colisiones de paquetes en la red habrán influido en los tiempos que hemos tomado como referencia.

Con respecto al uso de OpenMPI, nos hemos encontrado con una librería muy sólida que abstrae de manera muy cómoda el manejo de los diferentes nodos en un mismo programa a través de la red. Esta abstracción nos permite trabajar con una red de computadores al igual que si desarrolláramos para un sólo computador.

Aquí acaban nuestras conclusiones con respecto a la práctica final de la asignatura de Ingeniería de los computadores. Finalmente incluiremos en el siguiente apartado la bibliografía y enlaces que hemos utilizado para la realización de esta práctica.

7. Bibliografía

- Tutorial y código de “hello world”

  http://mpitutorial.com/tutorials/mpi-hello-world/

- Tutorial MPI

  https://www.dartmouth.edu/~rc/classes/intro_mpi/hello_world_ex.html

- Tutorial de MPI

  https://computing.llnl.gov/tutorials/mpi/

- Tutorial de MPI con parte teórica

  https://source.ggy.bris.ac.uk/mediawiki/index.php?title=Install_and_configure_MPI&redirect=no#Configuration_of_MPI

  -Estado del arte

  http://arxiv.org/ftp/arxiv/papers/1303/1303.2285.pdf

  http://scicomp.stackexchange.com/questions/5464/parallel-computation-of-big-covariance-matrices