Práctica 4. Paralelismo a nivel de proceso

Esteban Platero Horcajadas
Manuel Platero Horcajadas
María Luisa Risueño González
Gacel Ivorra Rodríguez

Estudiaremos el problema de convertir un algoritmo sin incorporar paralelismo a un algoritmo paralelo basado en procesos, también evaluaremos la ganancia de rendimiento al introducir paralelismo a base de proceso frente a la solución tradicional sin paralelismo y otras soluciones paralelas

17/12/2014

Índice

1. Introducción. Objetivos de la práctica.

Partiendo de un problema, como es el desenfoque Gaussiano, que incorpora un algoritmo libre de paralelismo, realizado en la práctica 2, recorriendo cada canal de manera secuencial, nos dispondremos a solucionarlo incorporando paralelismo basado en procesos.

¿En qué consiste el desenfoque Gaussiano? Utilizando una función de Gauss para el cálculo de la transformación a aplicar a cada pixel en la imagen, se obtiene un desenfoque que es utilizado en aplicaciones de edición de imágenes.

Es un suavizado del mapa de bits, generado por algoritmos que mezclan los colores de los pixeles vecinos, provocando una pérdida de nitidez y detalles.

La imagen es considerada una matriz de píxeles con tres canales. Los valores de cada pixel de la nueva imagen será una combinación de sus vecinos en función de un radio.

De manera secuencial, obtuvimos unos tiempos, que pretendemos mejorar con la ayuda de MPI. Por sus buenas prestaciones y amplias funcionalidades para el paralelismo a nivel de proceso. Pudiendo iniciar, gestionar y finalizar procesos MPI y comunicar datos entre procesos.

Las implementaciones en MPI consisten en un conjunto de bibliotecas de rutinas que pueden ser utilizadas en programas escritos en los lenguajes de programación C, C++, Fortran y Ada. La ventaja de MPI sobre otras bibliotecas de paso de mensajes, es que los programas que utilizan la biblioteca son portables y rápidos.

La idea es crear un anillo de varios computadores que ejecuten el programa del desenfoque gaussiano de forma paralela, existiendo un nodo maestro que es el que ejecuta el programa principal y reparte las tareas.

2. Planteamiento de la solución.

Implementación:

1. Debido a que la memoria no es compartida entre los nodos, el primer paso es buscar que datos necesitan cada uno de los distintos nodos para poder realizar su parte de trabajo.

2. Como hemos decidido que la imagen a la que vamos a aplicar el desenfoque gaussiano solo la tenga un nodo, el resto de nodos necesita conocer el **alto** y **ancho** de la imagen correspondiente y recibir los datos del **canal** que le corresponda a dicho nodo.

3. Por lo tanto, el nodo maestro, enviará al resto de nodos alto, ancho y canal (**MPI_Send**). Y los nodos esperaran a recibir estos datos (**MPI_Recv**).

4. Tras esto, todos los nodos realizaran su parte de trabajo y enviaran los resultados al nodo maestro. Dicho nodo, tras realizar su parte de trabajo, esperará a recibir el resultado del resto de nodos.

5. Una vez recibidos todos los resultados, será el nodo maestro quien juntará todas las partes y guardará la imagen.

Por último, mencionar que se ha utilizado la función **MPI_Wtime()** para realizar las mediciones de tiempo de ejecución de cada nodo. Obviamente el nodo que más tiempo tardará será el nodo maestro, ya que el reparte el trabajo y espera a recibir el trabajo de todos los nodos para posteriormente juntarlo y guardar el resultado final.

Una vez realizada la nueva implementación con MPI, el primer paso para poder lanzar el programa de forma paralela es realizar la configuración necesaria para poder crear un anillo. Se llevara a cabo mediante el sistema operativo Ubuntu de Linux.

Software previo que es necesario instalar

MPICH es una implementación de MPI libre y portable que nos dará soporte para crear el anillo.

Aunque se recomienda usar el protocolo ssh para la comunicación entre procesos debido a que contiene una capa SSL que introduce por encima del nivel TCP servicios de cifrado y autenticación de las comunicaciones, vamos a utilizar el protocolo rsh ya que no necesitamos esa seguridad adicional para nuestro caso y con esto conseguiremos unos mejores tiempos de ejecución.

Necesitamos dos elementos SW previos a la instalación de MPICH2 necesarios para que funcione:

1. Instalar los programas cliente/servidor de rsh, rlogin y rexec en todos los equipos.

apt-get install rsh-client rsh-server

2. Instalar superdemonio xinetd con configuración para rsh, rlogin y rexec correcta para la intercomunicación en la red local.

apt-get install xinetd

Configuración de MPIH2:

1. Descarga e instalación.

2. Para configuraciones **locales** usamos el fichero */etc/hosts.equiv*. Este debe listar los nombres de los equipos (hostname) que queremos interconectar entre sí mediante rsh, uno por línea.

3. Fichero *$HOME/mpd.hosts*. Debe incluir también los hosts que queremos incluir en el anillo. El orden en el que escribirlos es importante ya que cuando se lance el demonio mpd los cogera en ese orden.

4. Comando *mpdboot*: Para lanzar el demonio gestor de procesos mpd en los nodos especificados y crear el anillo usamos este comando. La sintaxis que vamos a utilizar es la siguiente:

mpdboot –n (nº nodos) – r (protocolo comunicación)

5. Para comprobar el estado del anillo MPI, utilizaremos el comando mpdtrace con la opción –l que debe listar todos los nombres de hosts donde se está ejecutando el demonio mpd.

6. Compilar aplicación con mpi:

mpicc miAplicacion.c -o miAplicacion

7. Lanzar aplicación en el anillo:

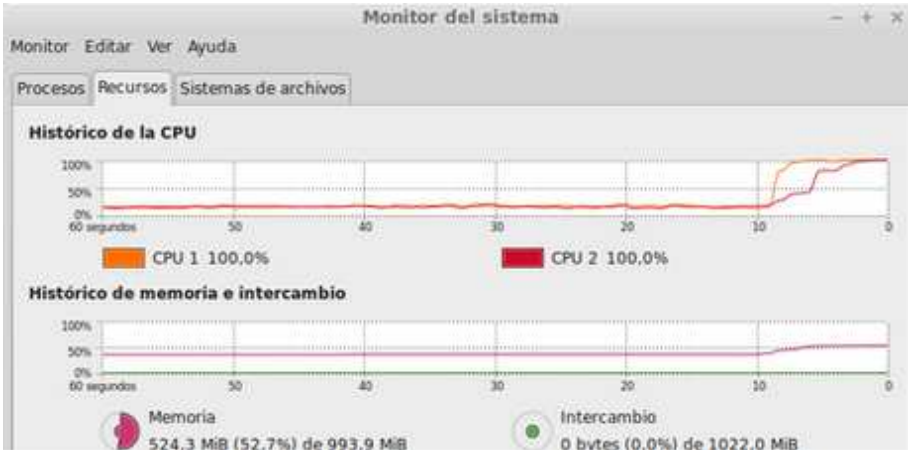
mpirun miAplicacion

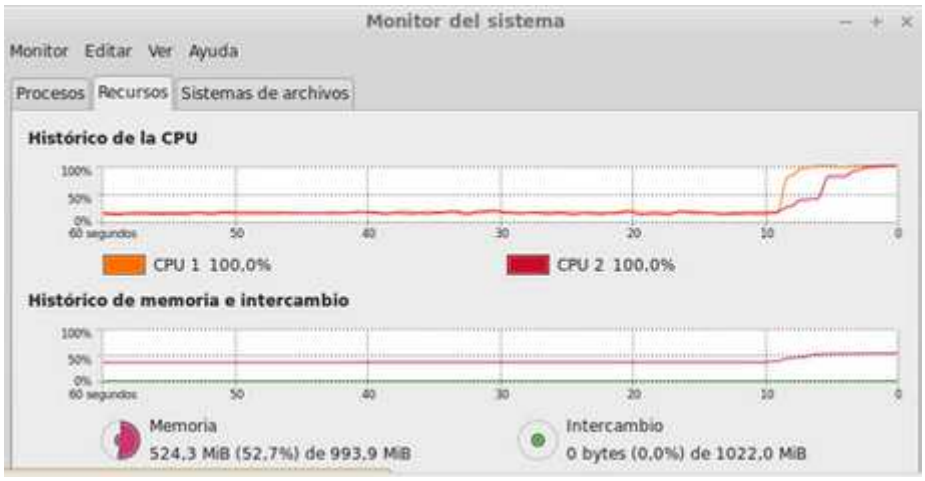
8. *mpdallexit:* Lo utilizaremos para romper/deshacer el anillo MPI (finalizar la ejecución de todos los demonios mpd en todos los nodos del anillo).

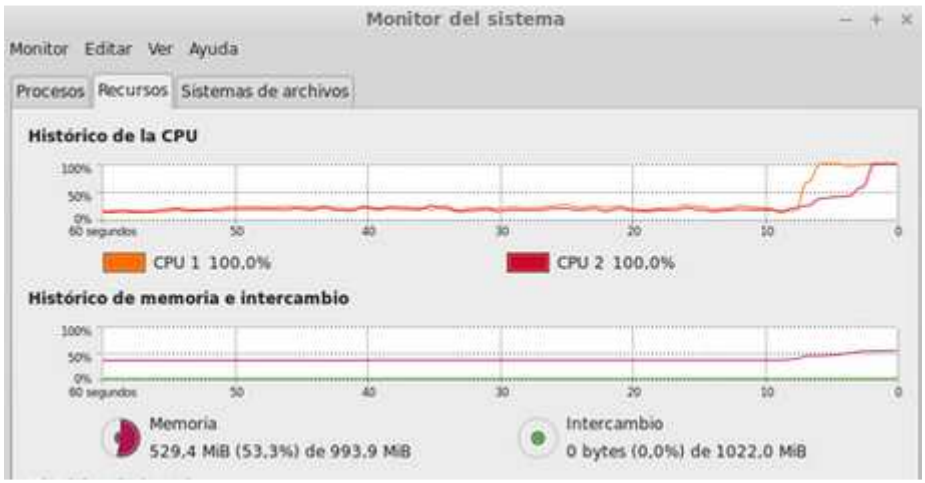
3. Resultados y evaluación del rendimiento.

A continuación vemos dos imágenes que corresponden al antes y el después de aplicar el desenfoque. Esto nos demuestra que el programa tiene un correcto funcionamiento en cuanto a los resultados que ofrece.

Para la prueba se han usado 3 nodos los cuales contienen un procesador de dos núcleos cada uno. Si nos fijamos en el porcentaje de uso de la CPU de dichos nodos vemos que efectivamente tenemos a los procesos trabajando:




Monitor del sistema
Monitor Editar Ver Ayuda
Procesos Recursos Sistemas de archivos
Histórico de la CPU
100%
50%
0%
60 segundos
50
40
30
20
10
0
CPU 1 100,0%
CPU 2 100,0%
Histórico de memoria e intercambio
Memoria
524,3 MiB (52,7%) de 993,9 MiB
Intercambio
0 bytes (0,0%) de 1022,0 MiB


Monitor del sistema
Monitor Editar Ver Ayuda
Procesos Recursos Sistemas de archivos
Histórico de la CPU
100%
50%
0%
60 segundos
50
40
30
20
10
0
CPU 1 100,0%
CPU 2 100,0%
Histórico de memoria e intercambio
Memoria
529,4 MiB (53,3%) de 993,9 MiB
Intercambio
0 bytes (0,0%) de 1022,0 MiB

4. Comparación de resultados

Resultados

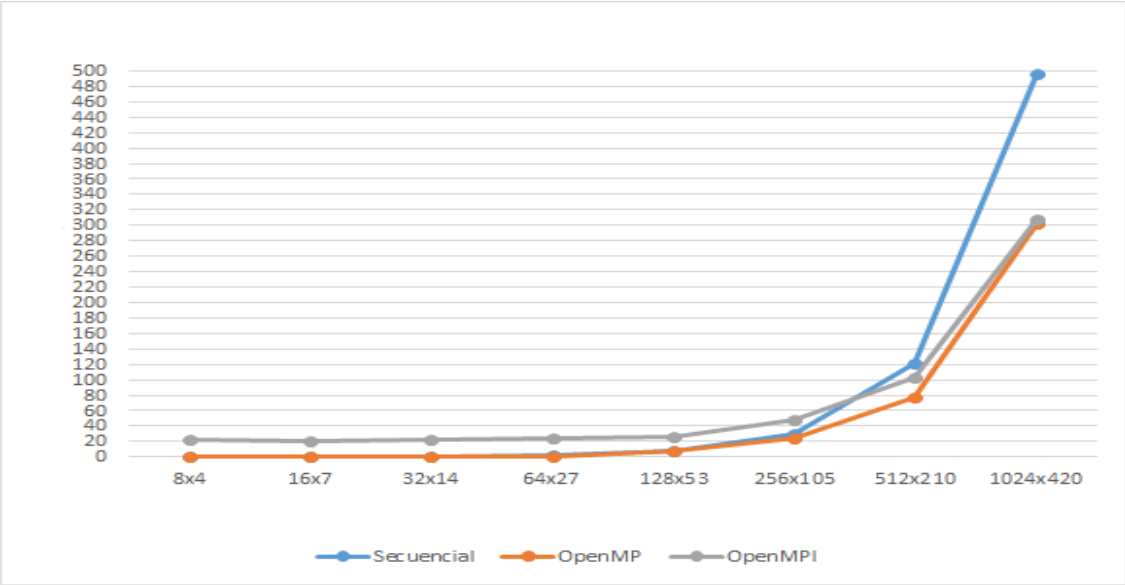
Hemos ejecutado los diferentes códigos que tenemos hasta el momento, uno que tiene un algoritmo secuencial, otro utilizando OpenMP (paralelismo a nivel de hilo) y MPI (paralelismo a nivel de proceso), para ver si obtenemos mejora en el tiempo de ejecución.

A la izquierda tenemos el tamaño de la imagen, para cada tiempo que obtenemos en la siguiente tabla, se ha realizado 5 veces el cálculo, y calculado la media, para evitar valores pico.

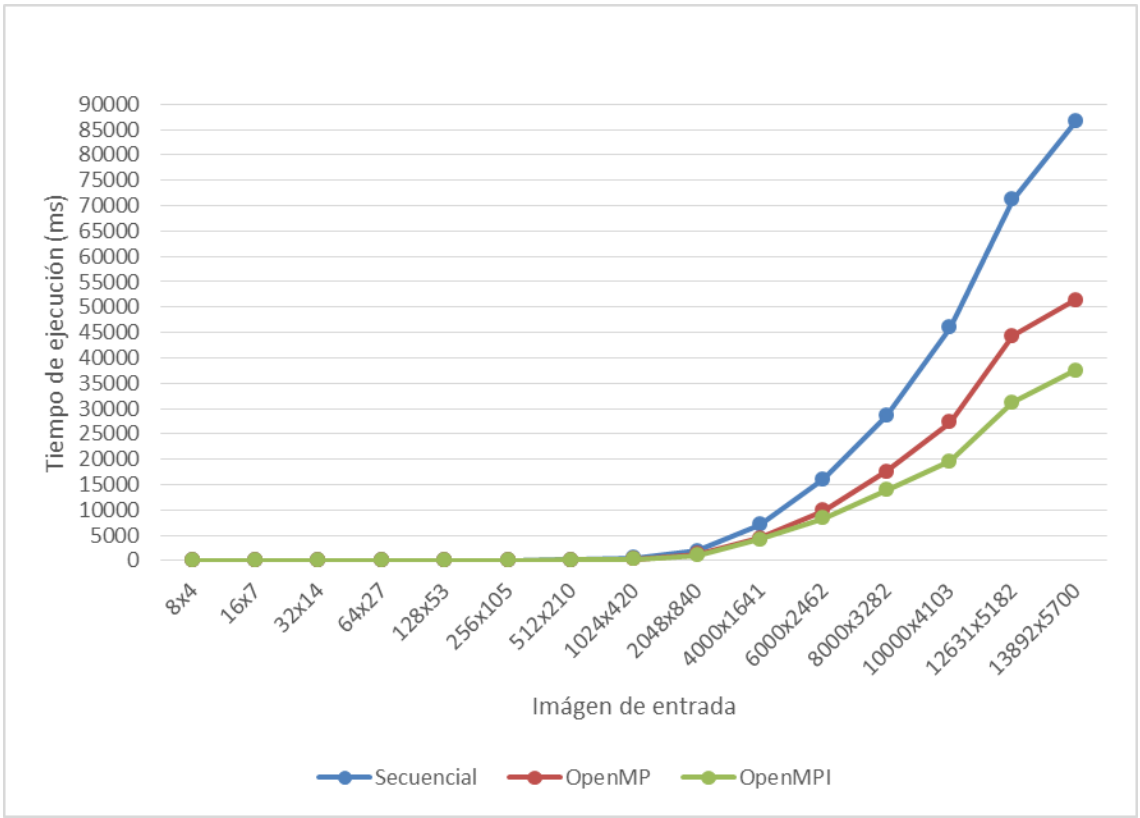
Imágen	Secuencial	OpenMP	OpenMPI	Ganancia_sec	Ganancia_OMP
8x4	0,0378	0,1236036	22,665453	0,001667736	0,005453392
16x7	0,1332	0,2603842	20,9892752	0,006346098	0,012405583
32x14	0,5292	0,2813808	23,2758046	0,022736056	0,012088983
64x27	1,6326	1,0072144	24,7951508	0,06584352	0,040621427
128x53	7,3896	7,2241482	27,0027162	0,273661359	0,267534131
256x105	29,8236	23,7278218	47,812891	0,623756468	0,49626411
512x210	120,3684	77,5213814	102,3173808	1,176421826	0,757656038
1024x420	495,5332	302,0987942	308,51388	1,606194185	0,979206492
2048x840	1976,1412	1255,208629	1005,454016	1,965421759	1,248399837
4000x1641	7131,7112	4443,043158	4183,22792	1,704834481	1,062108793
6000x2462	16017,6956	9873,470051	8323,74773	1,924336984	1,186180837
8000x3282	28489,4328	17542,37252	13893,2147	2,050600484	1,262657556
10000x4103	45913,6044	27323,25074	19642,13099	2,337506272	1,39105328
12631x5182	71181,5816	44356,65537	31188,39912	2,282309564	1,422216485
13892x5700	86607,1148	51387,8167	37539,59308	2,307087203	1,368896477

En la tabla podemos ver como a partir de un tamaño mínimo de imagen, cuando se empieza a requerir mas procesamiento, la paralelización mediante MPI resulta nos ofrece grandes mejoras de tiempos.

En la gráfica siguiente, los tamaños mostrados son los 8 primeros, ordenados ascendentemente. Hasta los 512x210 tratando los canales con MPI se empeora el tiempo, ya que al hacer tan pocas operaciones, es más costosa la comunicación entre procesos que el tiempo de hacer el cálculo. A partir de ese tamaño comienza a verse mejorado el tiempo respecto al secuencial, pero aún en este momento sigue siendo menos costoso el paralelismo a nivel de hilo, que el de a nivel de proceso.



A partir de ese tamaño (430.080px) se ve más igualado respecto al OpenMP y muy distanciado del secuencial. En la imagen inferior ya vemos tamaños más grandes, y efectivamente el tiempo se ve reducido unos 10 segundos respecto al paralelismo a nivel de hilo, y unos 40 segundos respecto al secuencial en el mejor de los casos. Por lo que hemos obtenido el resultado esperado, a los objetivos expuestos.



5. Conclusiones

MPI permite ejercer un mayor control sobre el código del programa, dando lugar al aprovechamiento de los recursos de las máquinas. Utilizando el paralelismo de procesos mejora el tiempo de ejecución y mejora el rendimiento frente al secuencial.

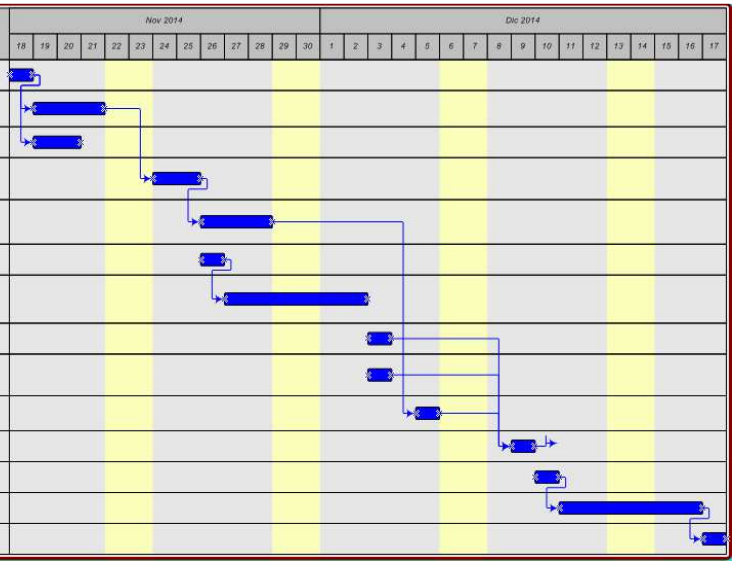
Se obtiene mejora frente al paralelismo de hilos.

También hemos comprobado, que a mayor carga computacional, se comporta mejor el paralelismo de procesos.

6. Diagrama de Gantt

En un primer momento, realizamos un análisis del algoritmo que implementamos en la práctica 1, mientras que nos informábamos acerca de MPI, posteriormente seleccionamos las partes con mayor coste computacional, y usamos MPI para esta adaptación y tuvimos que implementar un anillo, que incorporara varias computadoras, debido a esto, las pruebas se realizaron en las sesiones de la práctica. Una vez tuvimos todas las pruebas generamos unas gráficas y analizamos el rendimiento.

ID	Nombre de tarea	Comienzo	Fin	Duración
1	Leer Práctica.	18/11/2014	18/11/2014	1d
2	Análisis del algoritmo, Práctica 2.	19/11/2014	21/11/2014	3d
3	Buscar información MPI.	19/11/2014	20/11/2014	2d
4	Buscar partes con mayor peso computacional.	24/11/2014	25/11/2014	2d
5	Uso de MPI para adaptar el código.	26/11/2014	28/11/2014	3d
6	Creación del anillo.	26/11/2014	26/11/2014	1d
7	Incorporar paralelismo a nivel de proceso.	27/11/2014	02/12/2014	4d
8	Pruebas sin paralelizar.	03/12/2014	03/12/2014	1d
9	Pruebas con paralelismo a nivel de hilo.	03/12/2014	03/12/2014	1d
10	Pruebas con paralelismo a nivel de proceso.	05/12/2014	05/12/2014	1d
11	Gráficas.	09/12/2014	09/12/2014	1d
12	Análisis de rendimiento	10/12/2014	10/12/2014	1d
13	Memoria.	11/12/2014	16/12/2014	4d
14	Presentación y entrega.	17/12/2014	17/12/2014	1d



6. Referencias y bibliografías.

http://dvbmonkey.wordpress.com/2009/03/02/an-open-mpi-master-servant-example/

http://dvbmonkey.wordpress.com/2009/02/27/getting-started-with-open-mpi-on-fedora/

https://docs.it4i.cz/anselm-cluster-documentation/software/mpi-1/Running_OpenMPI

http://blogs.ua.es/

http://wiki.lazarus.freepascal.org/MPICH

http://www.sc.ehu.es