Paralelización del método iterativo de Jacobi para la Ecuación de Poisson en una dimensión

Segundo Parcial

High Performance Computing

Documento de análisis

Presentado por: Juan Camilo Rojas Cortés

En la presente entrega, de forma similar que en el reto del primer parcial, se pretende realizar una paralelización de la solución iterativa de Jacobi para la Ecuación de Poisson en una dimensión. La explicación de este algoritmo puede encontrarse aquí.

En este caso se requería hacer dos implementaciones paralelas: Una utilizando OpenMP y otra utilizando MPI en varias máquinas.

Infraestructura computacional:

Para realizar la presente implementación se decidió utilizar la infraestructura computacional provista Amazon Web Services (aws). En este caso se montó virtualmente un clúster homogéneo con 5 máquinas, una con rol de maestro y 4 con role de trabajadores (worker nodes). Al ser un clúster homogéneo, todas las instancias de equipos asociadas tienen las mismas características de hardware y software. Las características utilizadas fueron las siguientes:

- **Tipo de instancia:** t2.micro
- **Núcleos de procesamiento por máquina:** 1.
- **Memoria RAM:** 1 GB.
- **Disco duro:** 8 GB.
- **Sistema operativo:** Ubuntu 18.04 (Linux/UNIX).

Explicación de la técnica de paralelización:

Si bien es cierto que las implementaciones con OMP y MPI tienen consideraciones técnicas radicalmente distintas, la idea de paralelización para el algoritmo en ambas es la misma. Para ilustrar esta idea, se mostrará la siguiente porción de código de la implementación secuencial (todo el código se puede ver en este enlace):

```
/* Fill boundary conditions into utmp */
utmp[0] = u[0];
utmp[n] = u[n];

for (sweep = 0; sweep < nsweeps; sweep += 2) {

    /* Old data in u; new data in utmp */
    for (i = 1; i < n; ++i)
        utmp[i] = (u[i-1] + u[i+1] + h2*f[i])/2;

    /* Old data in utmp; new data in u */
    for (i = 1; i < n; ++i)
        u[i] = (utmp[i-1] + utmp[i+1] + h2*f[i])/2;
}
```

Figura 1. Porción de código de la implementación secuencial.

Como se puede ver en la Figura 1, la implementación secuencial tiene un ciclo externo que itera según un parámetro llamado *número de pasos* y dos ciclos internos que tienen como objetivo final darle valores al vector *u,* cuyo tamaño también está determinado por uno de los parámetros del algoritmo. Por lo tanto, el objetivo final del algoritmo es que después de determinada cantidad de pasos se le pueda dar valores precisos al vector *u.*

Para ejecutar de forma paralela este algoritmo se tomaron los dos ciclos que están en la parte interna del programa. Las iteraciones de estos ciclos son las que se dividieron entre los diferentes hilos de ejecución. Con esto hay que tener una consideración en cuenta: Es necesario hacer dos divisiones de trabajo por cada ciclo externo. Esto quiere decir que el proceso (por iteración del ciclo externo) es el siguiente:

- Se dividen los *n-1* elementos del vector *utmp* (el primer elemento no es necesario procesarlo porque siempre es 0) en los hilos de ejecución que haya disponibles. En este caso, por convención se eligió que fueran cuatro hilos de ejecución para ambas ejecuciones. Esta elección se hizo porque en la infraestructura computacional utilizada hay un nodo *master* y cuatro *worker nodes*. Para la implementación con OMP se trabajó con la misma cantidad de hilos de procesamiento con el objetivo de que hubiese una homogeneidad en las pruebas realizadas.
- En caso de que la división entre *n-1* y el número de hilos de ejecución no sea exacta, los elementos sobrantes (el módulo de la división) se reparten equitativamente entre los hilos de ejecución.
- Cada hilo de ejecución calcula los elementos que le corresponden del vector *utmp*.
- El hilo de ejecución principal (el nodo master en caso de MPI y el proceso padre en caso de OMP) recibe los resultados de los demás hilos de ejecución y los almacena en *utmp.*
- Se repite todo el proceso desde el primer paso, pero esta vez no se reparte *utmp,* sino que se reparte *u.* Se espera a que todos los procesos terminen su cálculo y los guarda en el vector *u* principal.

- El proceso se repite hasta que el ciclo externo llegue a su fin.

Implementación secuencial:

Para tener un parámetro de referencia respecto a la eficiencia de las implementaciones realizadas, se decidió montar la implementación secuencial en la infraestructura computacional utilizada. Para medir el rendimiento se lanzó un script que, en 10 iteraciones, ejecuta el algoritmo colocando en los dos parámetros de entrada (número de pasos y tamaño del vector) los valores 50, 500, 1000, 2500, 4000, 5000 y 10000 (se pasa el mismo valor para ambos parámetros), midiendo el tiempo en cada una de estas ejecuciones. De esta forma se tiene una buena base de resultados para realizar un análisis. Los resultados de rendimiento para la implementación secuencial fueron los siguientes:

Implementación secuencial (tiempo en segundos)							
Iteración\Tamaño	**50**	**500**	**1000**	**2500**	**4000**	**5000**	**10000**
1	0,000013	0,001147	0,004339	0,026471	0,069735	0,107372	0,430417
2	0,000013	0,001086	0,004282	0,026834	0,068508	0,107144	0,426769
3	0,000013	0,001104	0,004455	0,026955	0,068293	0,106707	0,430307
4	0,000013	0,00115	0,004383	0,026721	0,068004	0,107737	0,427167
5	0,000017	0,001061	0,004257	0,027077	0,068757	0,107927	0,430664
6	0,000013	0,001135	0,004623	0,026761	0,068425	0,106025	0,426277
7	0,000012	0,001097	0,004272	0,026623	0,068439	0,10714	0,425942
8	0,000012	0,001106	0,004255	0,026518	0,068763	0,10662	0,426294
9	0,00004	0,001165	0,004296	0,02651	0,06823	0,107393	0,426086
10	0,000018	0,001132	0,004368	0,026501	0,068187	0,105771	0,426468
Promedio	0,0000164	0,0011183	0,004353	0,0266971	0,0685341	0,1069836	0,4276391

Tabla 1. Tiempos de ejecución de la implementación secuencial

Implementación con MPI utilizando comunicación punto a punto

La primera implementación paralela realizada fue hecha haciendo uso del esquema de comunicación punto a punto. En este esquema se realiza el envío individual de datos desde el master a cada uno de los worker nodes para que estos realicen su parte respectiva. En el contexto de MPI, esto se hace utilizando las funciones *MPI_Send* y *MPI_Recieve*. Se siguió la técnica descrita anteriormente en este documento, dividiendo los vectores *utmp* en los cuatro worker nodes y esperando a que todos los resultados llegaran para continuar con la siguiente iteración del ciclo externo.

A continuación, se presentan los resultados de tiempos obtenidos en la presente implementación:

Implementación con MPI y comunicación punto a punto (tiempo en segundos)							
Iteración\Tamaño	**50**	**500**	**1000**	**2500**	**4000**	**5000**	**10000**
1	0,054129	0,560869	1,170502	3,078781	6,085468	7,334495	17,342594
2	0,098268	0,480135	1,130106	2,494469	15,103908	6,609488	16,547272
3	0,054395	0,570461	1,103555	3,015173	7,993141	6,98891	16,41329
4	0,052793	0,618362	1,101821	2,978729	9,589538	6,456399	16,779189
5	0,067095	0,612935	1,207049	2,900286	6,515518	7,626586	17,163038
6	0,061636	0,573362	1,112431	2,638719	5,522782	7,437072	16,342025
7	0,057081	0,537102	1,113431	3,395631	5,626328	6,284697	17,717831
8	0,071365	0,61196	1,031153	3,466198	16,718095	6,434656	17,450352
9	0,065201	0,571204	1,158926	1,680822	5,463156	6,489805	16,128585
10	0,05357	0,532424	1,228276	2,820843	6,734749	7,411068	15,797837
Promedio	0,0635533	0,5668814	1,135725	2,8469651	8,5352683	6,9073176	16,7682013

Tabla 2. Registro de tiempos para la implementación con comunicación punto a punto

Para tener una mayor claridad en la mejora de rendimiento de esta implementación respecto a la versión secuencial del algoritmo, se calculó el Speed Up Rate con los tiempos obtenidos. Los resultados de Speed Up Rate se presentan en la siguiente gráfica:

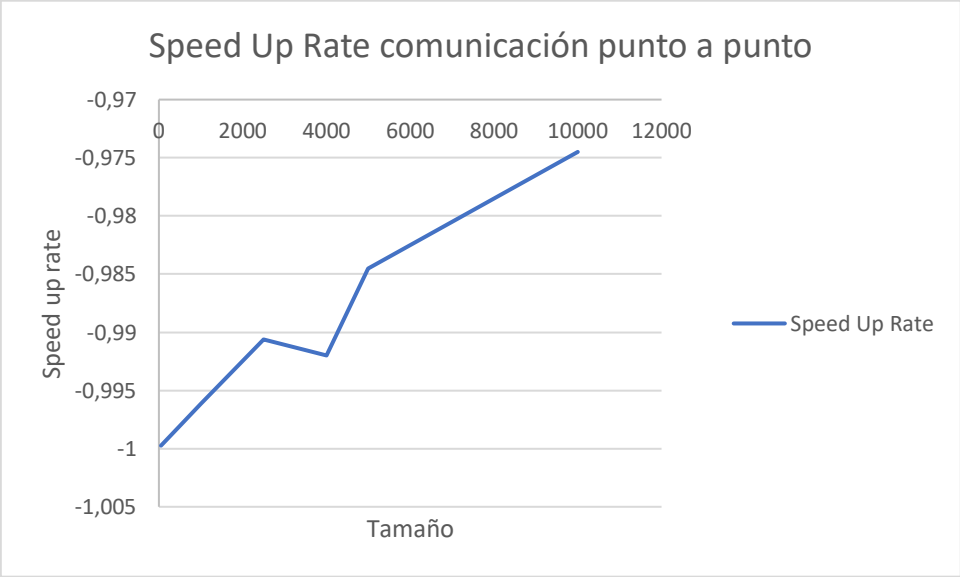

Gráfico 1. Speed Up Rate para la implementación con comunicación punto a punto

Como se puede ver en el Gráfico 1 y en la Tabla 1, no hubo ninguna mejora en la implementación punto a punto respecto a la implementación secuencial. De hecho, con un Speed Up rate tan cercano a -1 en todos los casos, se puede afirmar que la implementación con MPI punto a punto es un 100% más lenta que la implementación secuencial. Según el criterio del autor de este documento, si bien es cierto que el algoritmo es ejecutado por cuatro máquinas a la misma vez, en este caso resulta ser más lenta la ejecución porque, dada la estrategia de paralelización elegida, se hace necesario hacer muchos envíos dentro de la red, dejando una gran carga de responsabilidad a la velocidad de las comunicaciones. Además, el tipo de instancia utilizada (t2.micro) es el que tiene un rendimiento de red más bajo.

Implementación con OMP

La segunda implementación realizada para esta entrega se hizo utilizando paralelización con memoria compartida (en una sola máquina) a través de la librería OpenMP. Se utilizó la técnica de paralelización descrita al principio: Se le asignaron rangos a cada uno de los hilos de procesamiento para que procesaran los vectores *utmp* y *u*. De la misma forma que en la implementación con MPI, se debía esperar a que todos los procesos terminaran en cada iteración para continuar. Sin embargo, en este caso el factor de la comunicación es abolido, pues OMP ejecuta los algoritmos de forma concurrente en una sola máquina. Para lanzar los hilos de ejecución, en este caso se decidió utilizar la herramienta de OMP *#pragma omp parallel for Schedule* con el parámetro *dynamic* y un argumento numérico de 1. Esto quiere decir que los hilos de ejecución se lanzan en paquetes unitarios controlados, facilitando así el uso de variables compartidas sin que haya resultados erróneos. Los resultados de tiempos para la implementación con OMP fueron los siguientes:

Implementación con OMP con un solo núcleo (tiempo en segundos)							
Iteración\Tamaño	**50**	**500**	**1000**	**2500**	**4000**	**5000**	**10000**
1	0,000055	0,001548	0,00549	0,030635	0,075792	0,117943	0,465419
2	0,000056	0,001529	0,005339	0,030488	0,076136	0,11808	0,468275
3	0,000054	0,00153	0,005448	0,030744	0,076042	0,11801	0,462572
4	0,000054	0,001557	0,005471	0,030316	0,076087	0,117845	0,464692
5	0,000054	0,001592	0,005332	0,030731	0,076403	0,117265	0,466324
6	0,000056	0,001559	0,005388	0,030758	0,076508	0,118147	0,464698
7	0,000055	0,001552	0,005415	0,030383	0,076051	0,118456	0,462337
8	0,000054	0,001572	0,005429	0,030383	0,075983	0,118336	0,463005
9	0,000054	0,00161	0,005418	0,030473	0,07591	0,118359	0,464161
10	0,000054	0,001602	0,005336	0,030529	0,076128	0,1178	0,462788
Promedio	0,0000546	0,0015651	0,0054066	0,030544	0,076104	0,1180241	0,4644271

Tabla 3. Registro de tiempos para la implementación con OMP en una instancia de un núcleo de procesamiento.

Para tener una mayor claridad en la mejora de rendimiento de esta implementación respecto a la versión secuencial del algoritmo, se calculó el Speed Up Rate con los tiempos obtenidos. Los resultados de Speed Up Rate se presentan en la siguiente gráfica:

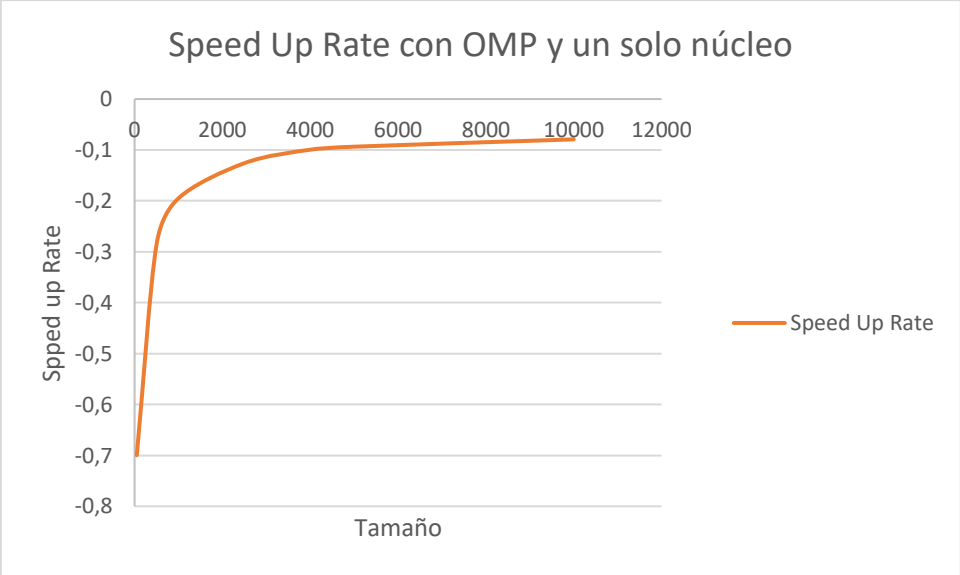

Gráfico 2. Speed Up Rate para la implementación con comunicación colectiva

Como se puede ver en el Gráfico 2, la implementación con OMP no tuvo mejoras significativas de rendimiento respecto a la implementación secuencial, pues en todos los valores tuvo tiempos de ejecución mayores. Sin embargo, es necesario tener en cuenta que las instancias utilizadas para esta práctica únicamente tienen un núcleo de procesamiento (ver información presentada al principio de este documento). Los *pragma* de OMP están diseñados de tal forma que se ejecutan en núcleos distintos del procesador. En caso de que el procesador solamente cuente con un núcleo, simplemente se lanzan los hilos de ejecución uno detrás de otro, y como es necesario crear varios procesos en el sistema operativo, lo más probable es que la ejecución con OMP termine siendo más lenta que la ejecución secuencial, como en este caso. Para validar esta información, se montó una instancia de aws que cuenta con cuatro núcleos de procesamiento, de tal forma que cada hilo de ejecución pudiera ser tratado por un núcleo distinto, haciendo el símil a la implementación con MPI en la que cada hilo era ejecutado por una máquina diferente. En esta ocasión los resultados de rendimiento fueron los siguientes:

Implementación con OMP con cuarto núcleos (tiempo en segundos)							
Iteración\Tamaño	**50**	**500**	**1000**	**2500**	**4000**	**5000**	**10000**
1	0,00188	0,003142	0,005003	0,012658	0,02704	0,038069	0,134324
2	0,001129	0,0032	0,005146	0,012001	0,027385	0,038317	0,131675
3	0,000217	0,001446	0,004022	0,012316	0,027641	0,04011	0,132459
4	0,000188	0,003222	0,005045	0,012856	0,027411	0,040043	0,132435
5	0,000797	0,00153	0,004995	0,013941	0,026003	0,039004	0,13184
6	0,000216	0,001484	0,005023	0,012091	0,027649	0,040174	0,131077
7	0,000735	0,001507	0,005015	0,011835	0,02767	0,03961	0,131722
8	0,000709	0,001489	0,003373	0,011828	0,025664	0,03945	0,131465
9	0,000647	0,001443	0,005041	0,011936	0,027458	0,039628	0,133128
10	0,000217	0,001419	0,003341	0,013779	0,027786	0,040009	0,131
Promedio	0,0006735	0,0019882	0,0046004	0,0125241	0,0271707	0,0394414	0,1321125

Tabla 3. Tiempos de ejecución para la implementación con OMP y cuatro núcleos de procesamiento.

Para tener una mayor claridad en la mejora de rendimiento de esta implementación respecto a la versión secuencial del algoritmo, se calculó el Speed Up Rate con los tiempos obtenidos. Los resultados de Speed Up Rate se presentan en la siguiente gráfica:

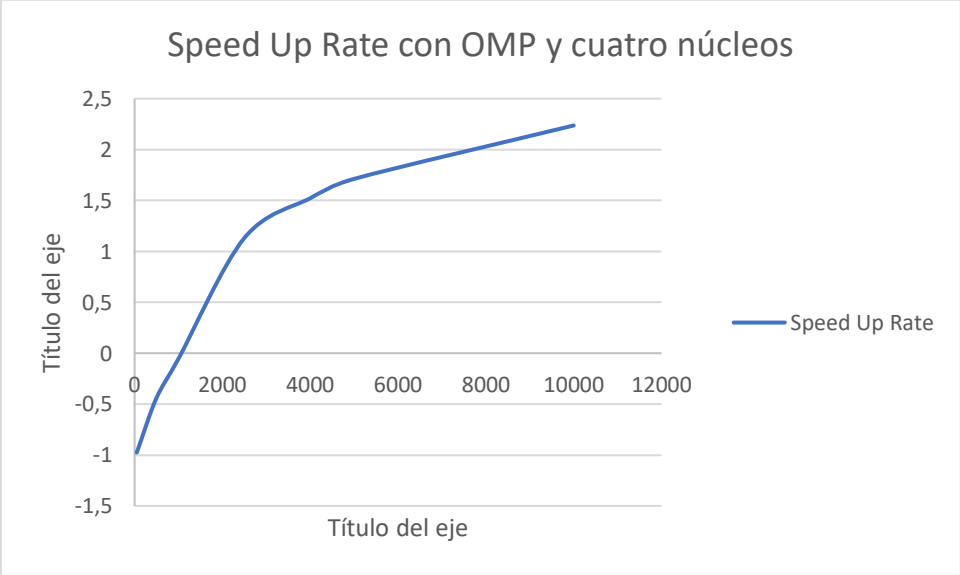

Gráfico 3. Speed Up Rate para la implementación con OMP y cuatro núcleos de procesamiento

Como se puede ver en la Tabla 3 y el Gráfico 3, cuando se ejecutó la implementación con OMP en una máquina de 4 núcleos, los resultados tuvieron mejoras significativas. De hecho, se llegó a tener un Speed Up rate superior a 2, lo que quiere decir que el algoritmo llegó a ejecutarse un 200% más rápido que la implementación secuencial.

Análisis comparativo y conclusiones

Para determinar las ventajas o desventajas de las diferentes implementaciones hechas en la presente prácita, se realizó un gráfico comparativo con los Speed Up Rate obtenidos por cada una de las ejecuciones. Este gráfico se presenta a continuación:

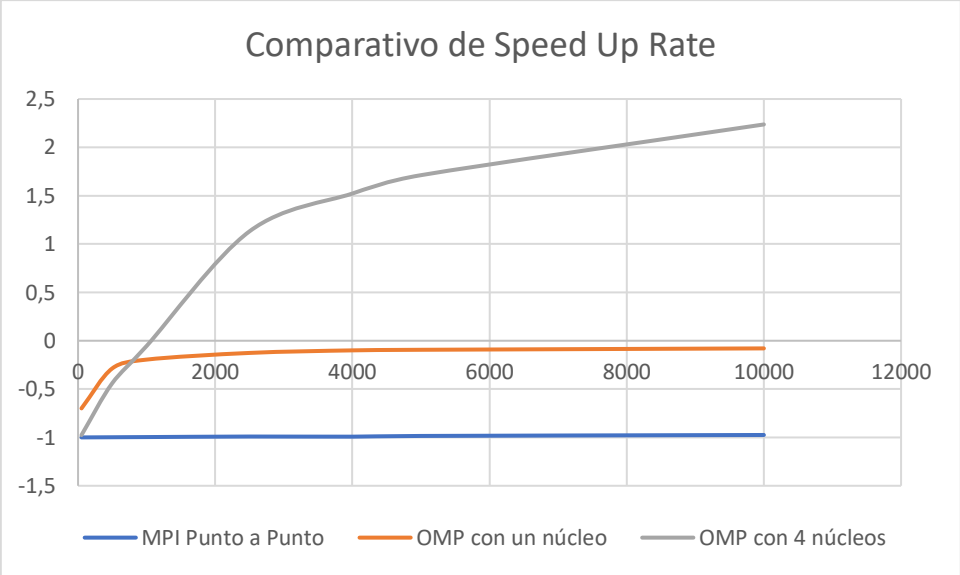

Gráfico 4. Comparativo de Speed Up Rate para las implementaciones hechas.

Según todo lo analizado en el presente documento y de acuerdo con la información presentada en el Gráfico 4, se puede concluir lo siguiente:

- Cuando se utiliza MPI es ideal que la técnica de paralelización elegida no requiera enviar y recibir muchos paquetes a través de la red. Los resultados de esta implementación no fueron buenos porque en cada iteración era necesario enviar información y esperar respuesta de todos los worker nodes dos veces. Esto quiere decir que para este caso en el que se utilizaron 4 worker nodes, en cada iteración circulaban en total 16 paquetes (8 de envío desde el máster a los worker nodes y 8 de regreso desde cada uno de estos). Esto quiere decir que, por ejemplo, para la implementación con un *n* igual a 10000, por la red debían pasar en total 160000 paquetes. El exceso de comunicación hace que la implementación lenta, especialmente si la red no tiene una velocidad alta. Este no fue el caso de la multiplicación de matrices (práctica pasada). Allí los resultados fueron bastante buenos (Speed Up Rate superior a 4), porque solamente era necesario hacer un envío a cada worker node y esperar su respectiva respuesta.
- La velocidad de ejecución de las implementaciones hechas con OpenMP dependen del número de núcleos que tenga el procesador. Para este caso la ejecución con cuatro núcleos fue ideal ya que, al tener cuatro hilos, cada uno de estos era ejecutado por un núcleo distinto. No es recomendable nunca utilizar OpenMP en procesadores de un solo núcleo.

- Para algoritmos en los que sea necesario realizar distribución de la información en repetidas ocasiones, es recomendable utilizar memoria compartida en lugar de memoria distribuida. Esto ocurre porque en el segundo caso es necesario tener en cuenta el componente del envío y recibimiento de paquetes, siendo este un factor sobre el que no se tiene mucho control. En este caso se vio, por ejemplo, que la implementación con OpenMP en una sola máquina tuvo un rendimiento considerablemente mayor que la implementación con MPI.