Paralelización del método iterativo de Jacobi para la Ecuación de Poisson en una dimensión utilizando una arquitectura híbrida con MPI y OpenMP

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 y el segundo 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 una implementación híbrida que utilizara paralelización con memoria distribuida utilizando MPI y también tuviera uno o más segmentos con paralelización de memoria compartida haciendo uso de OpenMP.

Infraestructura computacional:

Para realizar la presente implementación se decidió utilizar la infraestructura computacional provista por Amazon Web Services (aws). En este caso se montó virtualmente un clúster homogéneo con 5 máquinas, una con rol de maestro y 4 con rol 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.xlarge
- **Núcleos de procesamiento por máquina:** 4.
- **Memoria RAM:** 16 GB.
- **Disco duro:** 8 GB.
- **Sistema operativo:** Ubuntu 18.04 (Linux/UNIX).

Explicación de la técnica de paralelización:

Para ilustrar la idea de la técnica de paralelización utilizada, se mostrará la siguiente porción de código de la implementación secuencial:

```
/* 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. Esta parte específica del algoritmo se distribuyó utilizando MPI. 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) 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.

Lo explicado anteriormente puede interpretarse como la parte externa del algoritmo. Cuando cada uno de los worker nodes recibe su carga de trabajo, allí se activa la parte de la paralelización que utiliza MPI. En este caso, los worker nodes toman la carga de trabajo que se les asigna y la dividen en 4 procesos, que son ejecutados por cada uno de sus núcleos de procesamiento. Esta división se hace de la misma forma que en la paralelización inicial con OMP: Se asigna a cada proceso el resultado de la división entera entre el número de elementos que el worker node recibió y los 4 núcleos de procesamiento; y en caso de que esta división no sea exacta, los elementos que sobran se reparten equitativamente entre todos los procesos. Después se debe esperar a que cada uno de los procesos termine su ejecución para unir los resultados y enviarlos al master, continuando así la ejecución con OMP.

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 el primer parámetro de entrada (tamaño del vector) los valores 500000, 5000000, 10000000, 25000000, 40000000, 50000000 y 100000000, midiendo el tiempo en cada una de estas ejecuciones. En la entrega anterior se les pasaba el mismo valor a ambos parámetros (número de pasos y tamaño del vector). Sin embargo, en el análisis realizado se identificó que un número de pasos grande hace que la utilización de OMP tenga tiempos de ejecución muy grandes ya que, para la técnica de paralelización utilizada, la cantidad de envíos a través de la red depende directamente del número de pasos. Para la presente implementación, con el objetivo de obviar la variación dada por el número de envíos de mensajes, se dejó el número de pasos en una constante de 10. De esta forma, independientemente de la ejecución, siempre va a ser necesario pasar la misma cantidad de mensajes. Los resultados de tiempos de la implementación secuencial utilizando este script de ejecución son los siguientes:

Implementación secuencial (tiempo en segundos)							
Iteración\Tamaño	**500000**	**5000000**	**10000000**	**25000000**	**40000000**	**50000000**	**100000000**
1	0,024708	0,25887	0,516303	1,262481	2,034918	2,539614	5,175815
2	0,024593	0,257176	0,516249	1,272237	2,032416	2,542017	5,151555
3	0,024764	0,255997	0,517565	1,272626	2,03752	2,540214	5,159813
4	0,024459	0,257052	0,512714	1,269691	2,031461	2,539401	5,162338
5	0,024469	0,256556	0,513646	1,267585	2,032475	2,537656	5,158661
6	0,024605	0,258206	0,513235	1,269786	2,031295	2,545706	5,148255
7	0,024557	0,256835	0,512686	1,267634	2,041143	2,541205	5,17321
8	0,024605	0,258144	0,515007	1,271808	2,038088	2,539827	5,159528
9	0,02451	0,257319	0,513592	1,270985	2,035963	2,546222	5,181367

10	0,024802	0,25724	0,520746	1,268763	2,033766	2,542282	5,159432
Promedio	0,0246072	0,2573395	0,5151743	1,2693596	2,0349045	2,5414144	5,1629974

Tabla 1. Tiempos de ejecución de la implementación secuencial

Implementación híbrida

Después de probar la implementación secuencial, se lanzó el mismo script de ejecución para la implementación explicada anteriormente en el presente documento. Los resultados de tiempos de ejecución obtenidos fueron los siguientes:

Primera implementación híbrida (tiempo en segundos)							
Iteración\Tamaño	**500000**	**5000000**	**10000000**	**25000000**	**40000000**	**50000000**	**100000000**
1	0,928938	9,477978	19,461794	48,060301	76,330768	96,131574	191,926912
2	0,924588	9,535366	19,336707	48,32713	79,222952	96,367263	192,202614
3	0,926231	9,466441	18,964536	47,533424	76,955112	95,831279	191,539975
4	0,940856	9,773646	19,292727	47,59711	76,575158	95,900881	192,053673
5	0,930605	9,69911	19,368173	47,726529	76,751115	95,817687	192,012434
6	0,924862	9,677222	19,110137	47,74122	77,004155	95,958106	192,809585
7	0,923437	9,451104	18,969658	48,344166	76,882904	95,605265	192,207747
8	0,971622	9,694231	19,255491	47,73455	76,841562	95,135811	192,179787
9	0,977563	9,769859	19,351826	47,569733	76,301097	95,80263	192,034061
10	0,971952	9,609623	19,124879	47,971921	76,241708	96,019125	191,564177
Promedio	0,9420654	9,615458	19,2235928	47,8606084	76,9106531	95,8569621	192,053097

Tabla 2. Registro de tiempos para la implementación híbrida

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:

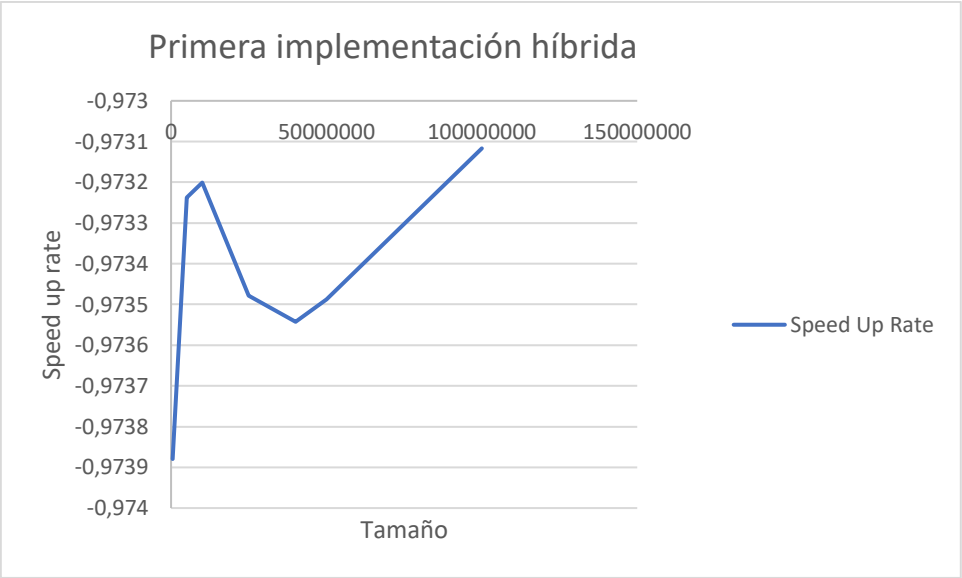

Gráfico 1. Speed Up Rate para la primera implementación híbrida

Como se puede ver en el Gráfico 1 y en la Tabla 1, no hubo ninguna mejora en esta implementación 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. A pesar de que se dejó como constante el número de envíos en la red, este sigue siendo muy alto, pues por cada iteración (son 10 en este caso) es necesario hacer envíos y recibir información de todos los worker nodes dos veces.

Para comprobar que el factor de la comunicación hace que el algoritmo sea lento para este caso específico, se decidió hacerle una modificación al algoritmo: En este caso, el vector utmp se llenaría únicamente por el nodo master, siguiendo la misma técnica de distribución por procesos en MPI que se sigue en cada worker node. Después de tener los resultados de utmp, este se envía a cada uno de los worker nodes, siguiéndose el mismo proceso de la primera implementación. De esta manera, el número de envíos en la red se reduce a la mitad. Los resultados de speed up rate obtenidos con esta modificación son los siguientes:

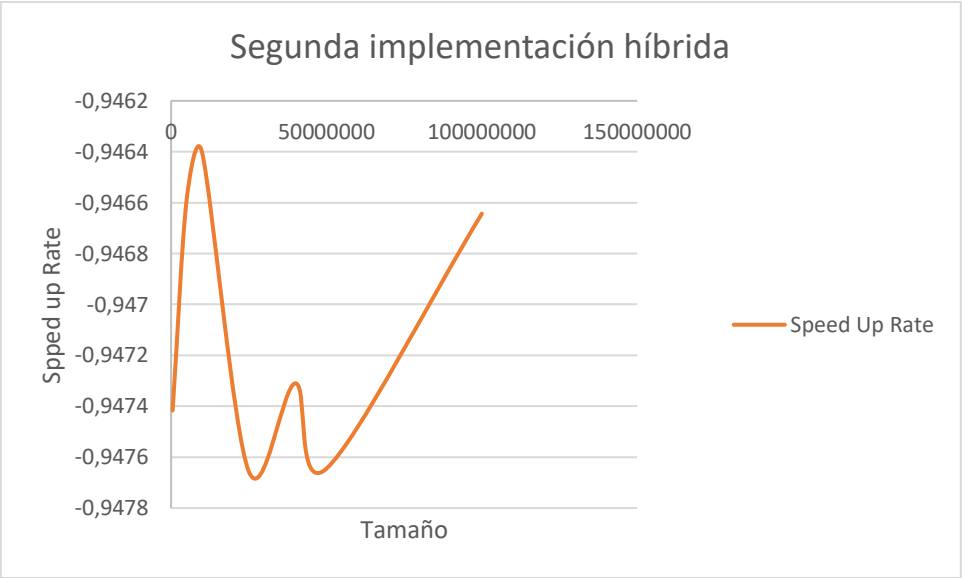

Gráfico 2. Speed Up Rate para la segunda implementación híbrida

Como se puede ver en el gráfico 2, los resultados de Speed Up Rate para la segunda implementación híbrida siguen siendo bastante malos. Sin embargo, sí existe una leve mejoría si se les compara con los resultados de la primera implementación. Esta comparación se puede ver en el siguiente gráfico:

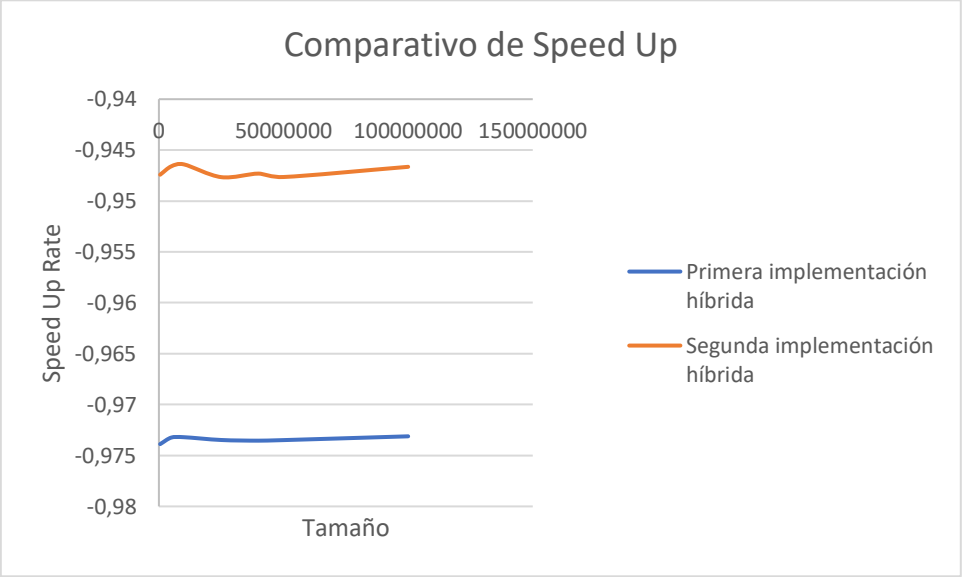

Gráfico 3. Comparativo de Speed Up Rate.

Conclusiones

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). Al dejar el parámetro de número de pasos en 10, era necesario hacer 80 envíos y recibos de información, teniendo en cuenta que el ciclo exterior se ejecutaría 5 veces,
- 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.