Implementación de multiplicación de matrices utilzando OMP
Documento de análisis
Presentado por: Juan Camilo Rojas Cortés
Repositorio del proyecto: https://github.com/JuanCRdrums/HPC/tree/master/multmat/OMP

El presente documento tiene como objetivo realizar una bitácora y un análisis de los resultados obtenidos al implementar la multiplicación de matrices cuadradas utilizando paralelización con la librería OMP. En este trabajo se utilizaron diferentes variaciones de las opciones que ofrece OMP para hacer ciclos paralelos. Específicamente, se utilizaron las diferentes variaciones de *#pragma omp parallel for*, utilizando diferentes cláusulas que pueden modificar su comportamiento. A todas estas implementaciones se les validó su efectividad y su eficiencia, comparando esta última con la eficiencia de la implementación secuencial utilizada en la entrega anterior. Para evaluar la eficiencia, se utilizó un script en el que se ejecuta automáticamente el programa con 7 diferentes dimensiones y en 10 veces cada una de estas, hallando al final el promedio de las implementaciones con cada tamaño de las matrices. Para efectos prácticos, en el presente documento no se mostrarán todos los resultados, sino solamente los promedios. A continuación, se explicará cada una de las variaciones implementadas:

#pragma omp parallel for simple

En el primer acercamiento a OpenMP se utilizó la cláusula *#pragma omp parallel for* en su forma más simple. En este caso se eligió colocarlo por fuera del ciclo más externo de la implementación. Esto, en términos generales, establece un hilo de procesamiento por cada fila de las matrices y las ejecuta en un proceso diferente, teniendo una complejidad cuadrática las instrucciones hechas por cada uno de estos. Además, esta cláusula hace que, por defecto, todas las variables que están declaradas antes de ella (incluyendo las variables globales) sean compartidas por todos los procesos.

```
void multmat(int n)
{
    int tot;
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
          tot = tot + fst[i][k] * sec[k][j];
        }
        mult[i][j] = tot;
        tot = 0;
      }
    }
}
```

Figura 1. Implementación de parallel for simple en la multiplicación de matrices.

Respecto a la efectividad del algoritmo, esta implementación cumple con los valores esperados. Sin embargo, en algunas ocasiones sucede que uno de los valores de la primera fila de la matriz resultante no se guarda correctamente. Esto puede suceder por el orden en el que se ejecutan los procesos. La memoria compartida sí está funcionando bien, pero al no haber un control respecto al orden de

ejecución, puede suceder que algunos valores no se guarden de forma correcta. Es importante señalar que en este caso pasa únicamente en la primera fila de la matriz y con uno o máximo dos valores. Además, esto sucede cuando hay muchos procesos en ejecución en el sistema operativo (por ejemplo, cuando está el navegador abierto a la misma vez que se ejecutan las pruebas). Cuando hay menos procesos en ejecución además de las pruebas, la efectividad pasas a ser de un 100%.

Para evidenciar la eficiencia, a continuación se relacionan la tabla y el gráfico de speed up rate respecto a la implementación secuencial de la multiplicación de matrices:

Dimensiones	Promedio implementación secuencial (segundos)	Promedio implementación con parallel for simple (segundos)	Speed up rate
10	0,0000151	0,00077930	-0,9806236366
100	0,0097298	0,0107906	-0,0983077864
200	0,0717527	0,0720495	-0,0041193901
500	1,767927	1,367617	0,2927062182
800	12,075896	7,0557421	0,711499064
1000	25,0743312	14,4189203	0,7389881266
2000	213,1887919	124,3949927	0,7138052527

Tabla 1. Speed up rate de la implementación con parallel for simple

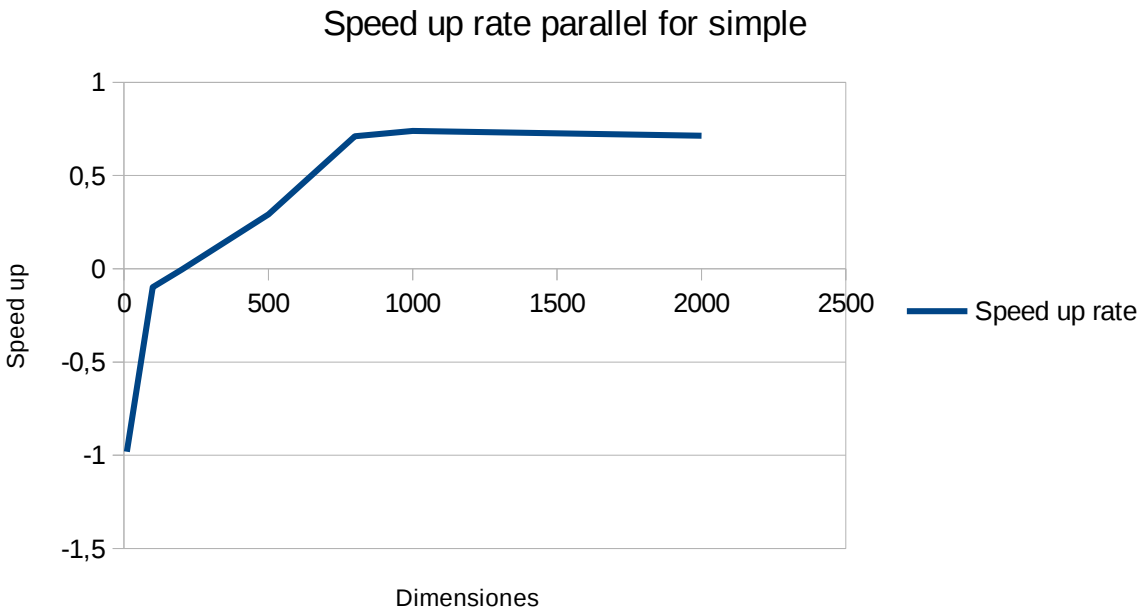

Gráfico 1. Speed up rate de la implementación con parallel for simple.

El análisis del speed up rate se realizará al final de este documento, cuando se presente la gráfica comparativa entre todos los speed up rate de las implementaciones.

#pragma omp parallel for interno

En la segunda variación del presente trabajo, se colocó la cláusula *#pragma omp parallel for* antes del primer ciclo interno. Esto quiere decir que por cada celda de la matriz resultante se genera un nuevo hilo de procesamiento, siendo la complejidad de cada uno de estos lineal. Comparativamente con la primera implementación, se generan más procesos. En la primera se generaban N procesos (siendo N el número de filas y columnas de la matriz), mientras que en esta implementación se generan NxN procesos.

```
void multmat(int n)
{
    int tot;
    for (int i = 0; i < n; i++) {
        #pragma omp parallel for
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                tot = tot + fst[i][k] * sec[k][j];
            }
            mult[i][j] = tot;
            tot = 0;
        }
    }
}
```

Figura 2. Segunda implementación en la multiplicación de matrices.

Respecto a la eficacia de este algoritmo, sucede algo similar que en la primera implementación: Hay uno o dos valores de la primera fila que no se guardan bien. Al igual que en la primera implementación, esto sucede cuando el sistema operativo está ejecutando muchos procesos aparte de las pruebas. Respecto a la eficiencia, a continuación se presentan los datos de speed up rate:

Dimensiones	Promedio implementación secuencial (segundos)	Promedio implementación con parallel for interno (segundos)	Speed up rate
10	0,0000151	0,00079850	-0,9810895429
100	0,0097298	0,0169319	-0,4253568708
200	0,0717527	0,1280401	-0,4396075917
500	1,767927	2,0777511	-0,1491151178
800	12,075896	9,4421511	0,2789348393
1000	25,0743312	18,2061966	0,3772415926
2000	213,1887919	140,9196646	0,5128391946

Tabla 2. Speed up rate de la implementación con parallel for interno

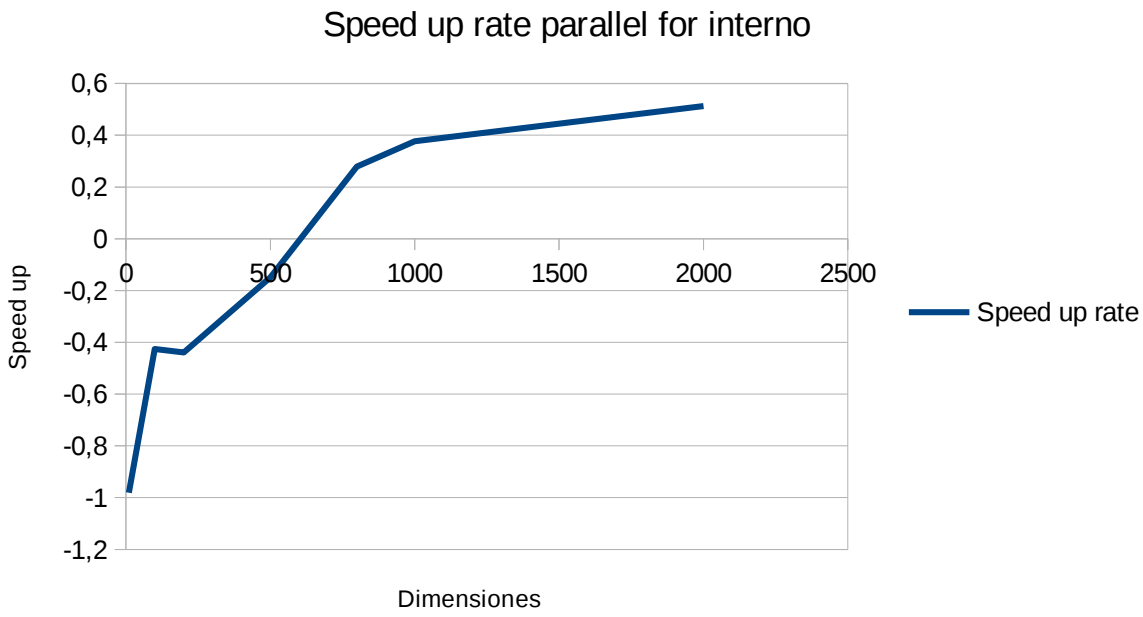

Gráfico 2. Speed up rate con la implementación del parallel for interno

El análisis del speed up rate se realizará al final de este documento, cuando se presente la gráfica comparativa entre todos los speed up rate de las implementaciones.

#pragma omp parallel for schedule

En la tercera implementación del presente trabajo se le añadió la cláusula *schedule* al *#pragma omp parallel for*. Esta cláusula permite tener un control del orden o la secuencia en la que se ejecuta cada proceso. La cláusula tiene espacio para argumentos, los cuales indican qué tipo de orden se sigue. Para esta implementación se eligió el tipo *dynamic* con un valor numérico de 1. Para entender el funcionamiento de esta cláusula se relaciona la siguiente figura:

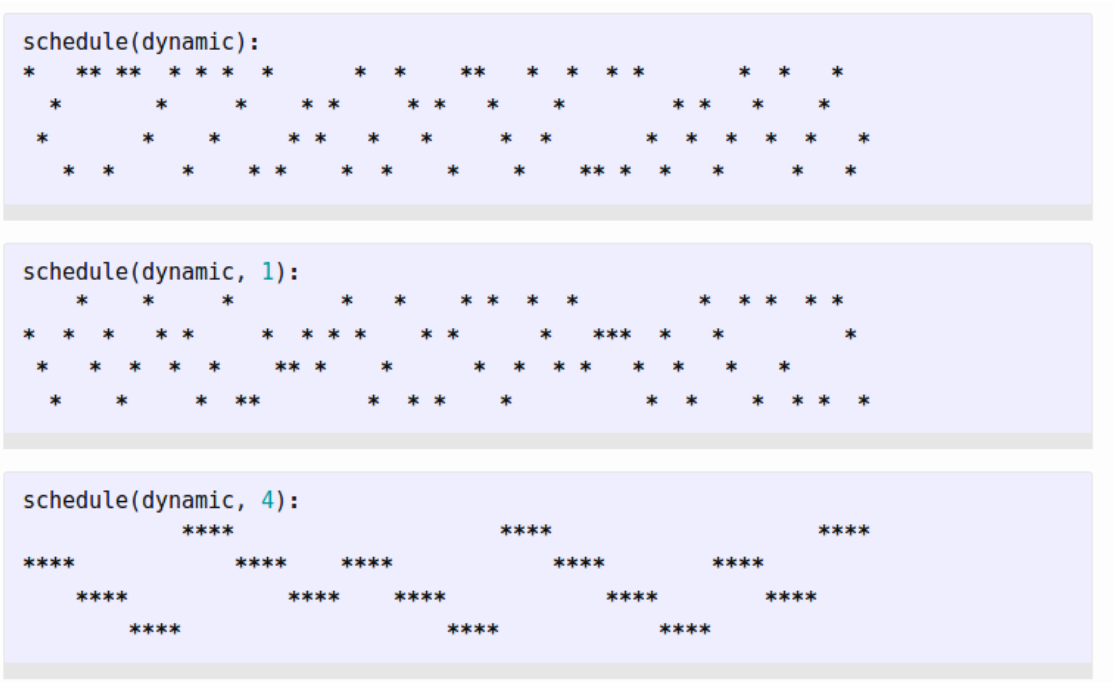

Figura 3. Ejemplos gráficos del funcionamiento de la cláusula *schedule.*
Fuente: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Cuando se coloca la cláusula schedule, OMP divide las instrucciones de cada hilo de ejecución en pequeños fragmentos, siendo estos representados en la Figura 3 con los asteriscos (*). El eje x de estas gráficas infica el tiempo de ejecución y el eje Y indica cada uno de los procesos que se ejecutan. Por lo tanto, la cláusula *dynamic* hace que estos fragmentos de los procesos se ejecuten simultáneamente pero de manera controlada. El argumento numérico de la cláusula le indica al sistema operativo cuántos de estos fragmentos de cada proceso puede ejecutar de forma seguida. Por ejemplo, cuando el valor es 4, se ejecutan 4 fragmentos seguidos de cada proceso y luego se pasa a 4 fragmentos de otro. Para la presente implementación se eligió un valor de 1, por lo que solamente se ejecuta un fragmento de cada proceso a la vez.

```
void multmat(int n)
{
    int tot;
    #pragma omp parallel for schedule(dynamic,1)
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
          tot = tot + fst[i][k] * sec[k][j];
        }
        mult[i][j] = tot;
        tot = 0;
      }
    }
}
```

Figura 4. Implementación de la cláusula *schedule* en la multiplicación de matrices.

Como se puede ver en la Figura 4, para la presente implementación se decidió poner la cláusula del for paralelo antes del primer ciclo, por lo que se establece un hilo de procesamiento por cada fila de la matriz resultante. Respecto a la eficacia, con esta cláusula se corrigen los errores presentados en las dos primeras implementaciones. Esto puede ocurrir porque, al existir un control respecto al orden de ejecución, hay mayor garantía de que todas las posiciones de memoria compartida se van a setear de forma correcta. Para analizar la eficiencia, a continuación se presentan los datos de speed up rate:

Dimensiones	Promedio implementación secuencial (segundos)	Promedio implementación con parallel for con schedule (segundos)	Speed up rate
10	0,0000151	0,00264040	-0,9942811695
100	0,0097298	0,0103101	-0,0562846141
200	0,0717527	0,07006460	0,0240934794
500	1,767927	1,3810037	0,280175426
800	12,075896	7,0578096	0,710997701
1000	25,0743312	14,0689181	0,7822501362
2000	213,1887919	119,4086942	0,7853707666

Tabla 3. Speed up rate para la implementación de parallel for con la cláusula *schédule*

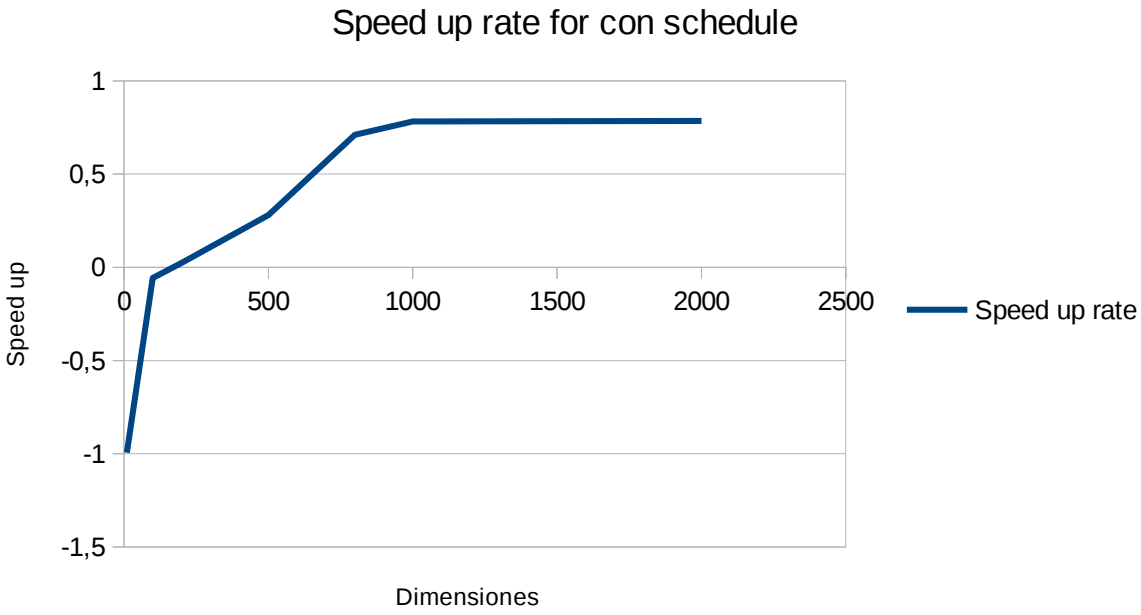

Gráfico 3. Speed up rate para la implementación con parallel for con la cláusula *schedule*

El análisis del speed up rate se realizará al final de este documento, cuando se presente la gráfica comparativa entre todos los speed up rate de las implementaciones.

#pragma omp for nowait

En la cuarta implementación se agregó la cláusula *nowait*. Esta cláusula se usa para que no se tenga que esperar a la finalización de ciertos procesos para iniciar otros. Es recomendable cuando los ciclos que se implementan son independientes. Por ejemplo, en este caso se puede implementar con éxito, pues al dividir los procesos por filas, no existe una codepenencia entre ellos. Por lo tanto, en casos como este, la cláusula *nowait* puede llegar a aumentar la eficiencia de los algoritmos que se ejecutan concurrentemente.

```
void multmat(int n)
{
    int tot;
    #pragma omp parallel
    #pragma omp for nowait
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
          tot = tot + fst[i][k] * sec[k][j];
        }
        mult[i][j] = tot;
        tot = 0;
      }
    }
}
```

Figura 5. Implementación de la cláusula *nowait* en la multiplicación de matrices.

Como se puede evidenciar en la figura 5, para utilizar la cláusula *nowait* es necesario definir primero la región paralela y después definir el ciclo (no se puede utilizar la abreviación *#pragma omp parallel for*). Respecto a la eficacia de la implementación, funciona perfectamente bien, sin problemas de sincronización o de espacios de memoria que no se modifiquen correctamente. Para analizar la eficiencia, se presenta la información del speed up rate.

Dimensiones	Promedio implementación secuencial (segundos)	Promedio implementación con parallel for con no wait (segundos)	Speed up rate
10	0,0000151	0,00522810	-0,9971117614
100	0,0097298	0,0151537	-0,3579257871
200	0,0717527	0,07636210	-0,0603624049
500	1,767927	1,3801876	0,2809323892
800	12,075896	7,7298317	0,5622456567
1000	25,0743312	14,8813778	0,6849468871
2000	213,1887919	117,8326515	0,8092505701

Tabla 4. Speed up rate con la implementación de parallel for agregando la cláusula *nowait*

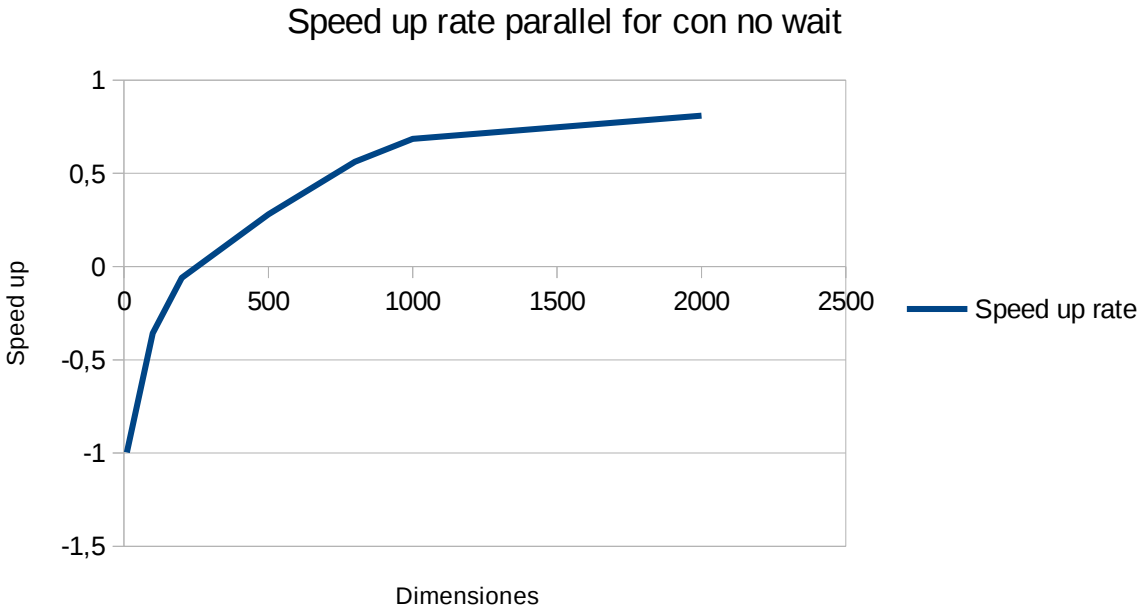

Gráfico 4. Speed up rate para la implementación con parallel for agregando la cláusula *nowait*

El análisis del speed up rate se realizará al final de este documento, cuando se presente la gráfica comparativa entre todos los speed up rate de las implementaciones.

#pragma omp parallel for con cláusula default

Por último, se agregó la cláusula *default.* Esta hace que se produzca un fenómeno llamado *random access* para los iteradores. En el caso de la multiplicación de matrices, indica que ninguna variable es compartida por defecto, por lo que debe indicarse siempre cuales son las variables que se comparten.

```
void multmat(int n)
{
    int tot;
    #pragma omp parallel for default(none) shared(n,tot,mult,fst,sec)
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
          tot = tot + fst[i][k] * sec[k][j];
        }
        mult[i][j] = tot;
        tot = 0;
      }
    }
}
```

Figura 6. Implementación de parallel for con la cláusula *default* para la multiplicación de matrices

En cuanto a eficiencia y eficacia, esta implementación se comporta de forma muy similar al parallel for simple; pues, para este caso puntual, todas las variables definidas por fuera del pragma son compartidas. A continuación, los datos de speed up rate:

Dimensiones	Promedio implementación secuencial (segundos)	Promedio implementación con parallel for con random access (segundos)	Speed up rate
10	0,0000151	0,00145430	-0,9896169979
100	0,0097298	0,0113748	-0,1446179273
200	0,0717527	0,07005550	0,0242265061
500	1,767927	1,3753154	0,2854702274
800	12,075896	7,0550402	0,7116693396
1000	25,0743312	14,1343465	0,774000036
2000	213,1887919	117,4955419	0,8144415392

Tabla 5. Speed up rate para implementación con random access.

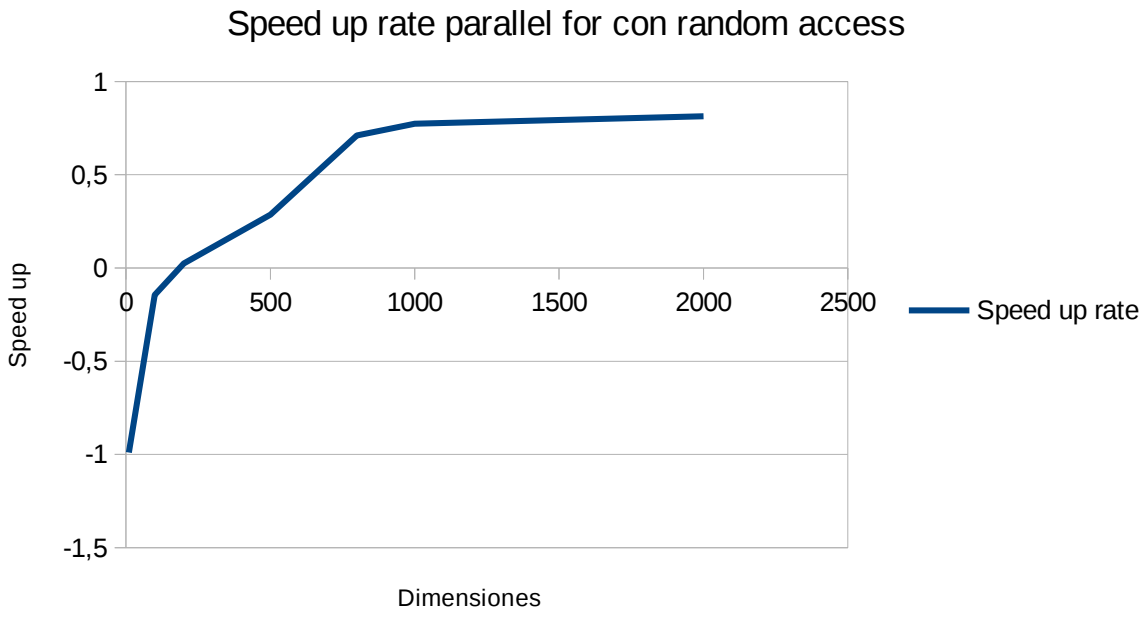

Gráfico 5. Speed up rate para implementación con random access.

El análisis del speed up rate se realizará al final de este documento, cuando se presente la gráfica comparativa entre todos los speed up rate de las implementaciones.

Comparación de rendimiento entre las implementaciones

Con el fin de comparar el rendimiento entre las diferentes implementaciones realizadas en el presente trabajo, se realizó el Gráfico 6, que compara los speed up rates obtenido por cada una de las variaciones utilizadas.

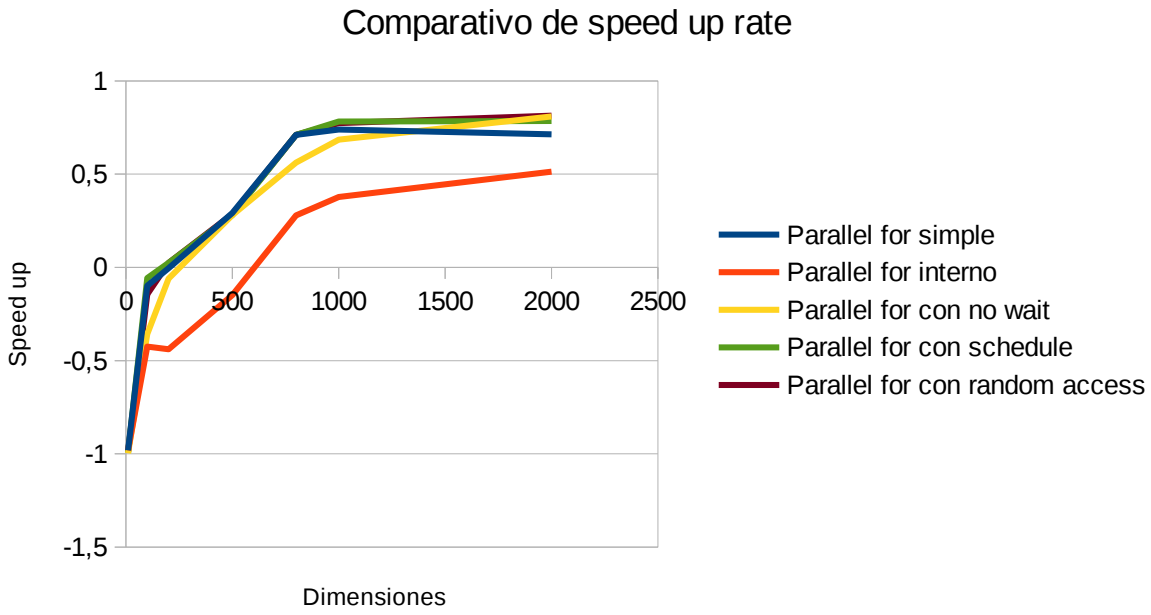

Gráfico 6. Comparatido de speed up rate de todas las implementaciones

Lo primero que se puede identificar en el Gráfico 6 es que la implementación con el paralelismo por celda (colocando el ciclo paralelo más interno) tiene un rendimiento menor que el resto de implementaciones. Por lo tanto, se puede concluir que la mejor opción de paralelismo en la multiplicación de matrices es generar un hilo de procesamiento por cada fila de la matriz resultante.

Por otro lado, es posible ver que el rendimiento de todas las cláusulas en las que se implementó paralelismo por filas es bastante similar. Llama la atención el comportamiento del speed up rate cuando se utilizó la cláusula *nowait*: Es menor que en el resto de cláusulas cuando las dimensiones son relativamente pequeñas, pero cuando crece el número de procesos que se ejecutan, termina siendo más eficiente que sus competidores. No se realizaron pruebas con más de 2000 filas y columnas, pero a juzgar por la tendencia creciente que presenta el speed up rate, se podría predecir que es posible lograr rendimientos mayores a medida que las dimensiones de las matrices aumentan. Por lo tanto, según lo evidenciado en el presente trabajo, la mejor opción de paralelismo con OMP para la multiplicación de matrices es utilizar concurrencia por filas y añadir la cláusula *nowait.*