Eigensolvers y Solver Iterativos

Descripción

Se realizó la implementación de los siguientes algoritmos usando la biblioteca de OpenMP para realizar programación multiproceso:

1. Suma de vectores en paralelo.
2. Multiplicación de vectores elemento a elemento en en paralelo.
3. Producto punto en paralelo.
4. Multiplicación matriz vector en paralelo.
5. Multiplicación matriz matriz en paralelo

Las pruebas realizadas se obtuvieron bajo la siguiente configuración:

- Procesador: 12 × AMD Ryzen 5 1600 Six-Core Processor
- Memoria: 16 GiB de RAM

Los resultados que se presentan están basados en 3 medidas de rendimiento

- **Tiempo de ejecución**: Tiempo total de ejecución
- **Speed-Up**: $SpeedUp = \frac{Tiempo\ Lineal}{Tiempo\ Paralelo}$
- **Eficiencia**: $SpeedUp = \frac{Tiempo\ Lineal}{Tiempo\ Paralelo * Número\ de\ núcleos}$

Suma de vectores en paralelo

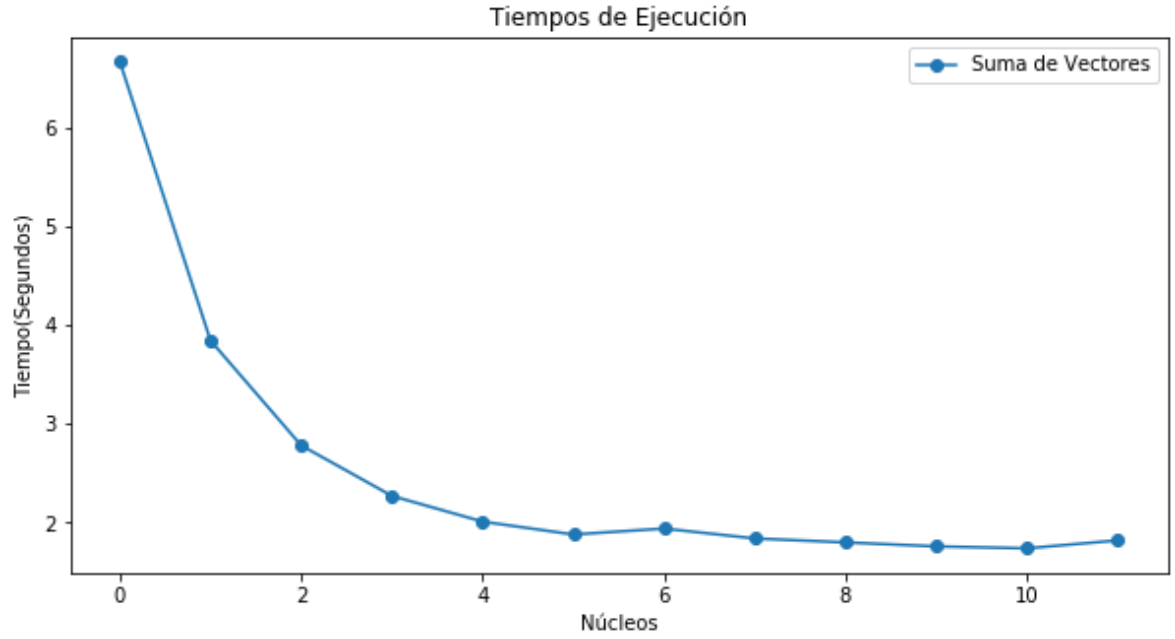
Para realizar el multiprocesamiento se usó la directiva *#pragma omp parallel for* como se observa el código:

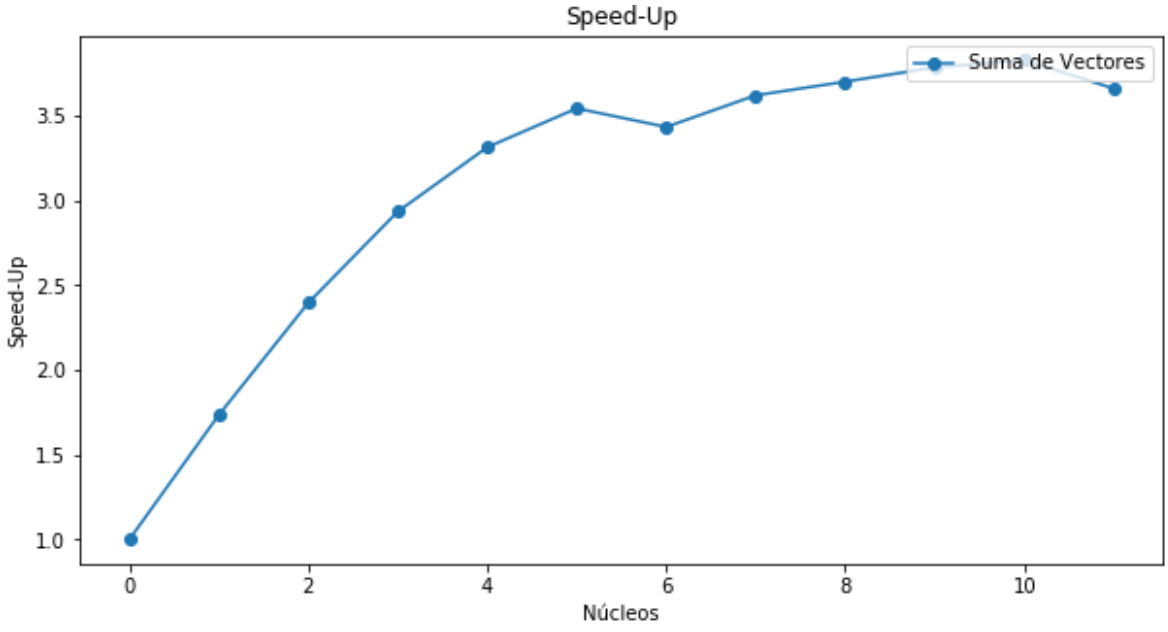
```
1 void vector_sum_parallel(double * vec_a, double * vec_b, double * vec_c, int
  size) {
2     #pragma omp parallel for
3     for (int i = 0; i < size; ++i) {
4         vec_c[i] = vec_a[i] + vec_b[i];
5     }
6 }
```

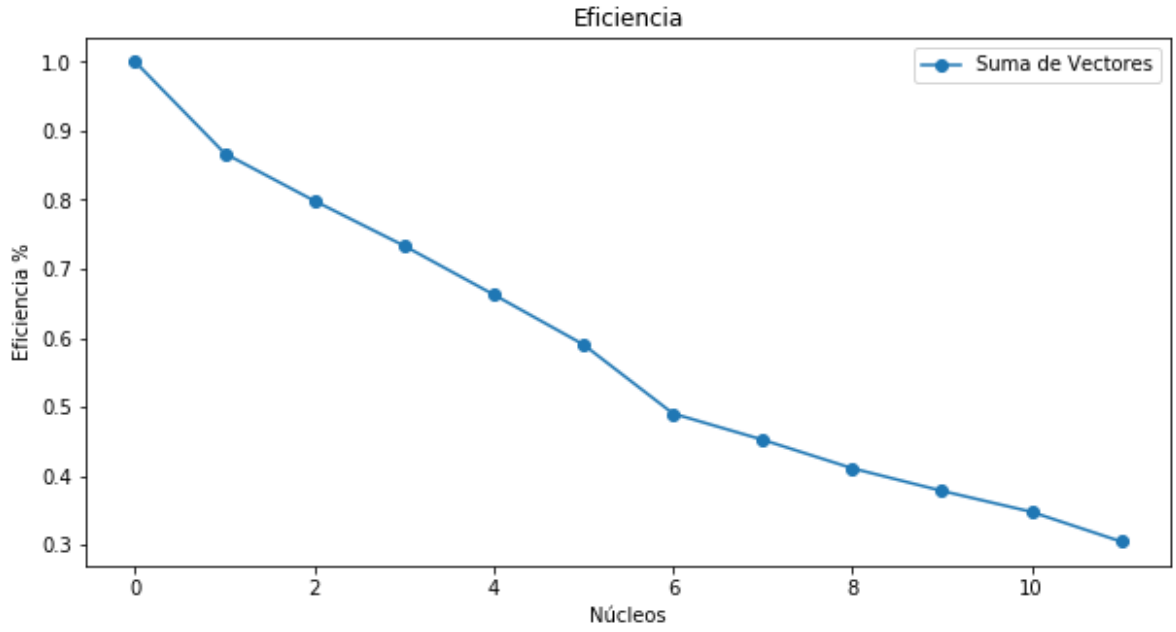
Se realizaron pruebas con tamaños de *array* de 1000000000 elementos con los siguientes resultados

```
Cores: 1: InputSize: 1000000000 Time 6.66s Memory: 7814300Kb CPU: 99%
Cores: 2: InputSize: 1000000000 Time 3.84s Memory: 7814168Kb CPU: 191%
Cores: 3: InputSize: 1000000000 Time 2.78s Memory: 7814104Kb CPU: 275%
Cores: 4: InputSize: 1000000000 Time 2.27s Memory: 7814148Kb CPU: 356%
Cores: 5: InputSize: 1000000000 Time 2.01s Memory: 7813940Kb CPU: 433%
Cores: 6: InputSize: 1000000000 Time 1.88s Memory: 7814084Kb CPU: 502%
Cores: 7: InputSize: 1000000000 Time 1.94s Memory: 7813848Kb CPU: 542%
Cores: 8: InputSize: 1000000000 Time 1.84s Memory: 7813996Kb CPU: 625%
Cores: 9: InputSize: 1000000000 Time 1.80s Memory: 7814136Kb CPU: 705%
Cores: 10: InputSize: 1000000000 Time 1.76s Memory: 7813528Kb CPU: 790%
Cores: 11: InputSize: 1000000000 Time 1.74s Memory: 7813548Kb CPU: 864%
Cores: 12: InputSize: 1000000000 Time 1.82s Memory: 7814092Kb CPU: 848%
```

Las siguientes gráficas muestran los resultados para las medidas de rendimiento mencionadas anteriormente.







Multiplicación de vectores elemento a elemento en en paralelo

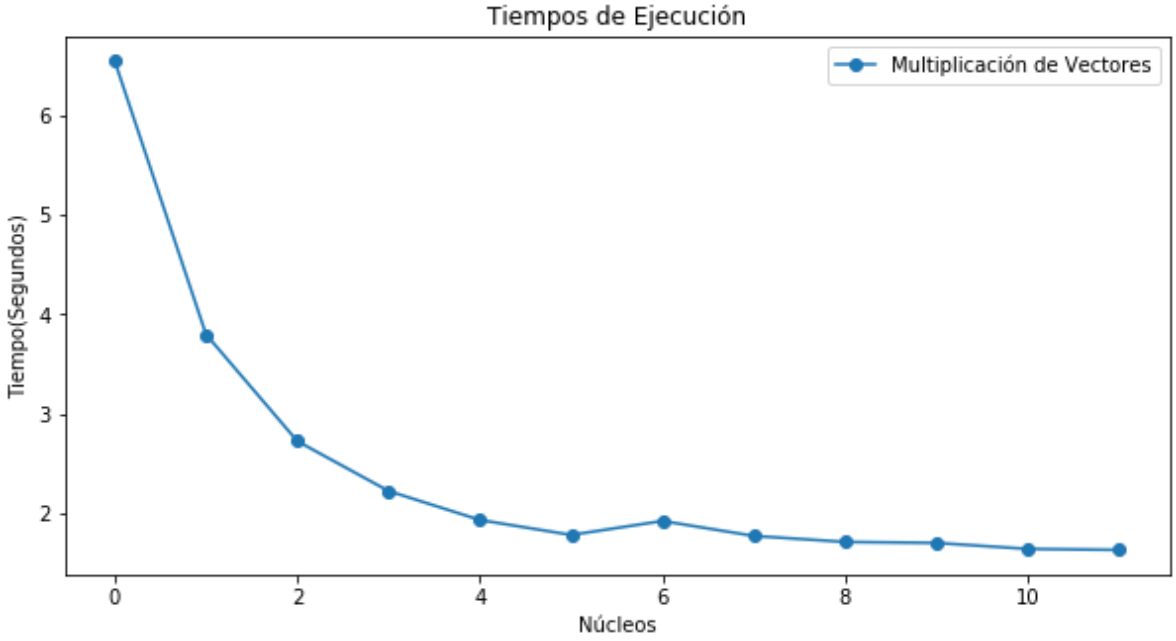
La implementación es muy parecida a la anterior, solo se necesita paralelizar un ciclo *for*:

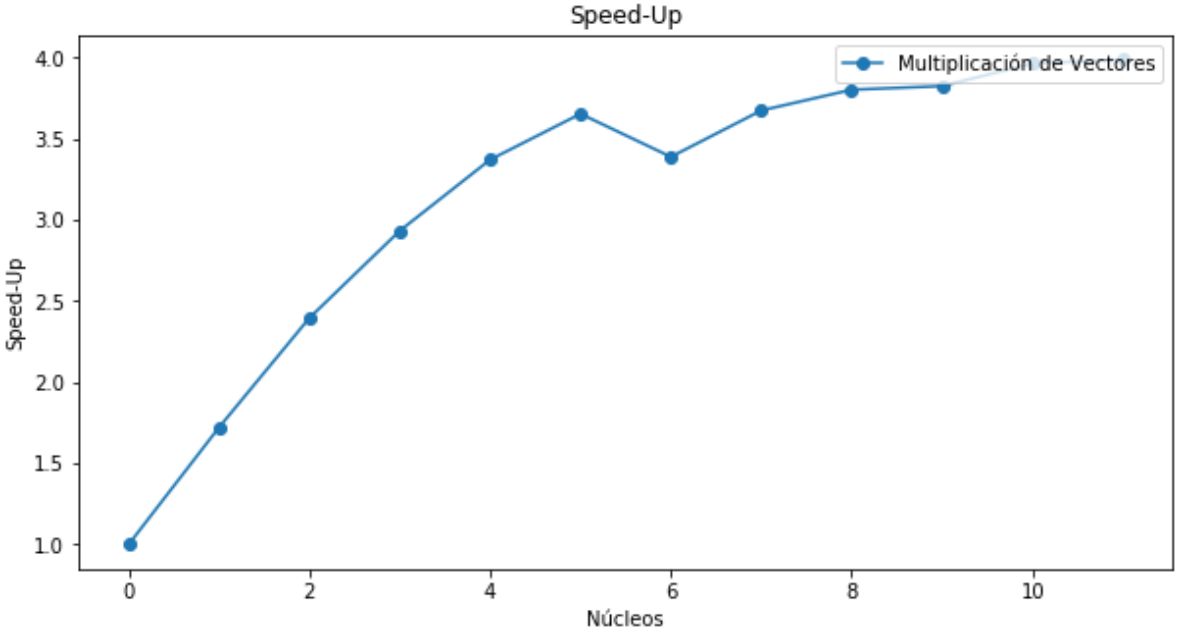
```
void vector_mult_parallel(double * vec_a, double * vec_b, double * vec_c, int size) {
    #pragma omp parallel for
    for (int i = 0; i < size; ++i) {
        vec_c[i] = vec_a[i] * vec_b[i];
    }
}
```

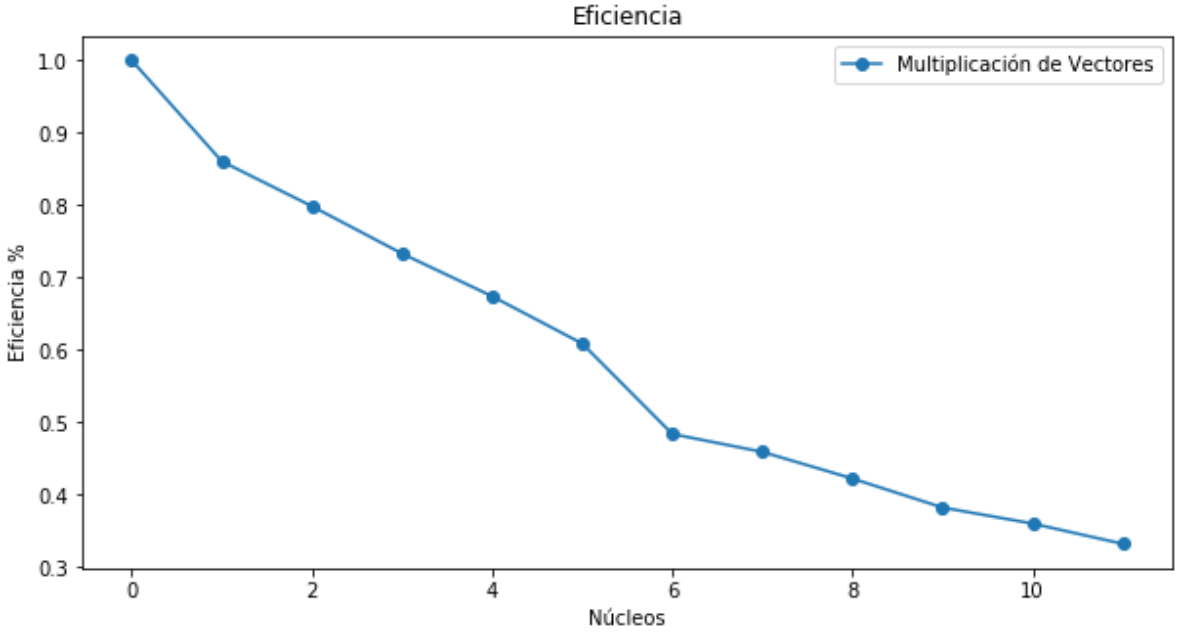
Se realizaron pruebas con tamaños de *array* de 1000000000 elementos con los siguientes resultados

```
Cores: 1: InputSize: 1000000000 Time 6.54s Memory: 7814188Kb CPU: 99%
Cores: 2: InputSize: 1000000000 Time 3.80s Memory: 7814136Kb CPU: 191%
Cores: 3: InputSize: 1000000000 Time 2.73s Memory: 7813988Kb CPU: 275%
Cores: 4: InputSize: 1000000000 Time 2.23s Memory: 7814024Kb CPU: 356%
Cores: 5: InputSize: 1000000000 Time 1.94s Memory: 7813876Kb CPU: 435%
Cores: 6: InputSize: 1000000000 Time 1.79s Memory: 7814048Kb CPU: 505%
Cores: 7: InputSize: 1000000000 Time 1.93s Memory: 7813848Kb CPU: 540%
Cores: 8: InputSize: 1000000000 Time 1.78s Memory: 7814088Kb CPU: 627%
Cores: 9: InputSize: 1000000000 Time 1.72s Memory: 7814004Kb CPU: 702%
Cores: 10: InputSize: 1000000000 Time 1.71s Memory: 7813532Kb CPU: 798%
Cores: 11: InputSize: 1000000000 Time 1.65s Memory: 7813540Kb CPU: 886%
Cores: 12: InputSize: 1000000000 Time 1.64s Memory: 7814152Kb CPU: 954%
```

Las siguientes gráficas muestran los resultados para las medidas de rendimiento mencionadas anteriormente.


Tiempos de Ejecución
Multiplicación de Vectores
Tiempo(Segundos)
Núcleos
0
2
4
6
8
10
2
3
4
5
6


Speed-Up
Multiplicación de Vectores
Speed-Up
Núcleos
0
2
4
6
8
10
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Eficiencia
Multiplicación de Vectores
Eficiencia %
Núcleos
0
2
4
6
8
10
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

Producto punto en paralelo

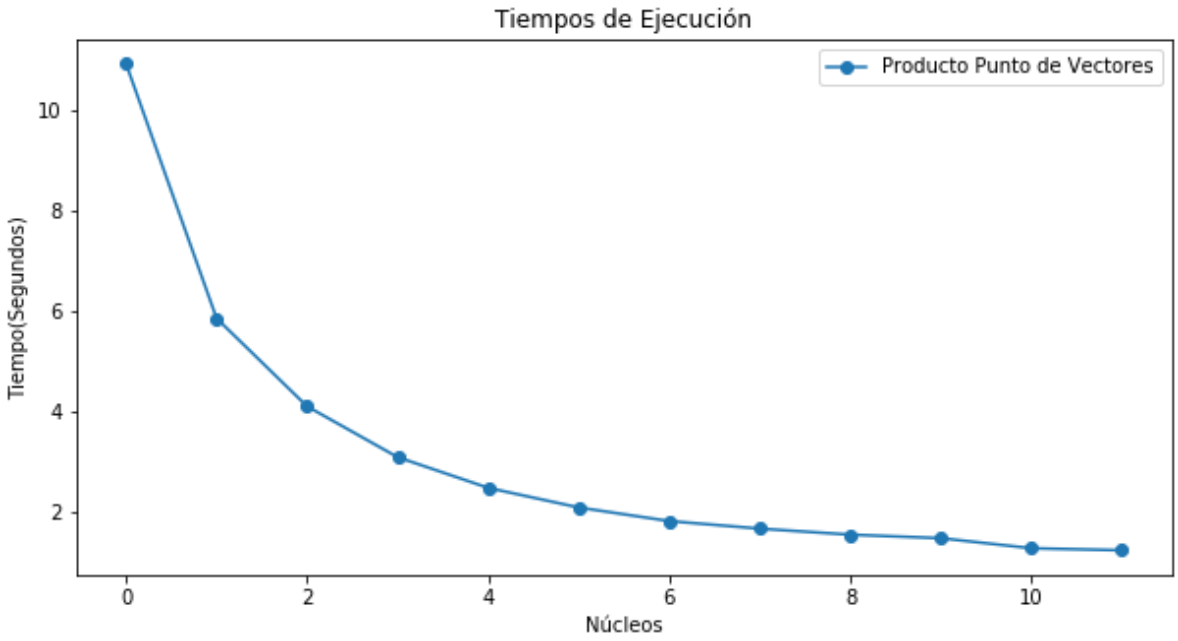
Para realizar la paralelización del producto punto es necesario evitar la concurrencia sobre la variable que guardará el resultado, para ello se usa la opción *reduction* como se muestra en el siguiente ejemplo:

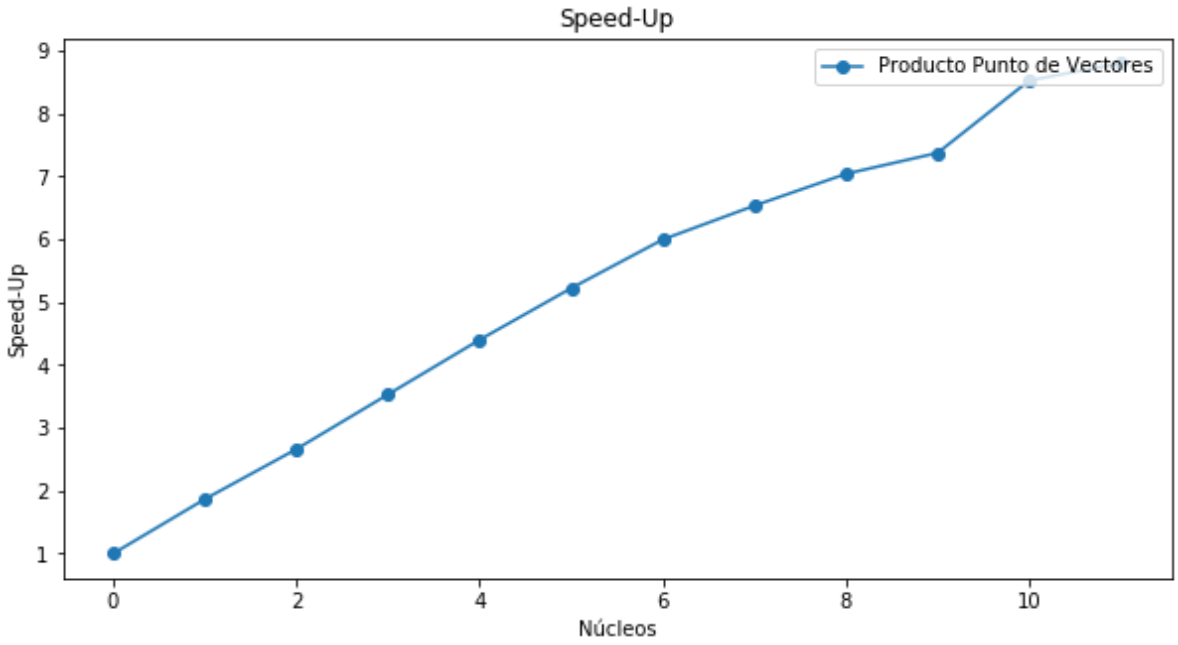
```
double vector_scalar_product(double * vec_a, double * vec_b, int size) {
    int res = 0;
    #pragma omp parallel for reduction(+:res)
    for (int i = 0; i < size; ++i) {
        res += vec_a[i] * vec_b[i];
    }
    return res;
}
```

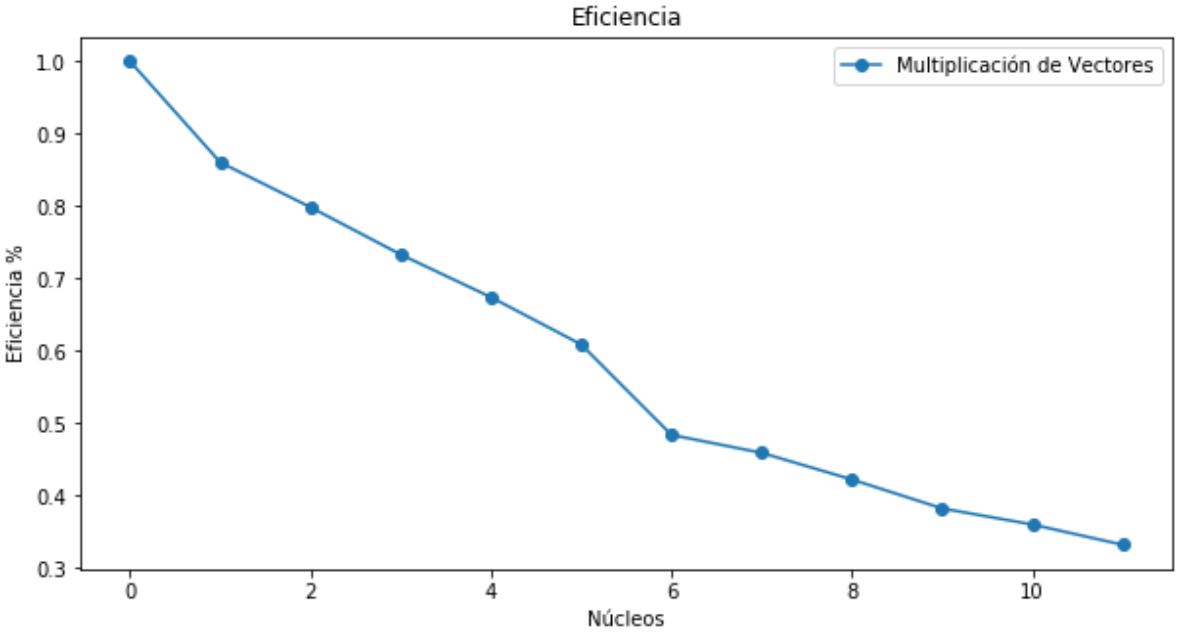
Se realizaron pruebas con tamaños de *array* de 10000000000 elementos con los siguientes resultados

```
Cores: 1: InputSize: 10000000000 Time 10.91s Memory: 1664Kb CPU: 99%
Cores: 2: InputSize: 10000000000 Time 5.85s Memory: 1696Kb CPU: 198%
Cores: 3: InputSize: 10000000000 Time 4.10s Memory: 1760Kb CPU: 295%
Cores: 4: InputSize: 10000000000 Time 3.09s Memory: 1700Kb CPU: 389%
Cores: 5: InputSize: 10000000000 Time 2.48s Memory: 1700Kb CPU: 487%
Cores: 6: InputSize: 10000000000 Time 2.09s Memory: 1844Kb CPU: 581%
Cores: 7: InputSize: 10000000000 Time 1.82s Memory: 1796Kb CPU: 668%
Cores: 8: InputSize: 10000000000 Time 1.67s Memory: 1740Kb CPU: 752%
Cores: 9: InputSize: 10000000000 Time 1.55s Memory: 1696Kb CPU: 841%
Cores: 10: InputSize: 10000000000 Time 1.48s Memory: 1680Kb CPU: 877%
Cores: 11: InputSize: 10000000000 Time 1.28s Memory: 1764Kb CPU: 1038%
Cores: 12: InputSize: 10000000000 Time 1.24s Memory: 1696Kb CPU: 1089%
```

Las siguientes gráficas muestran los resultados para las medidas de rendimiento mencionadas anteriormente.







Multiplicación matriz vector en paralelo.

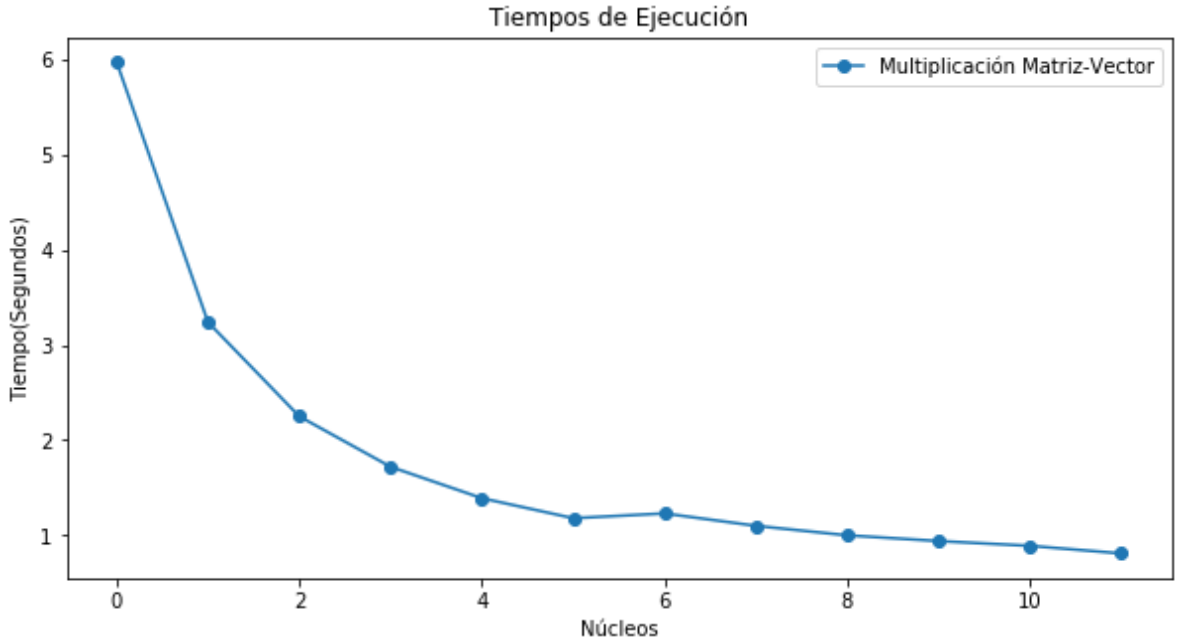
La implementación de la paralización es muy similar a las anteriores, observemos:

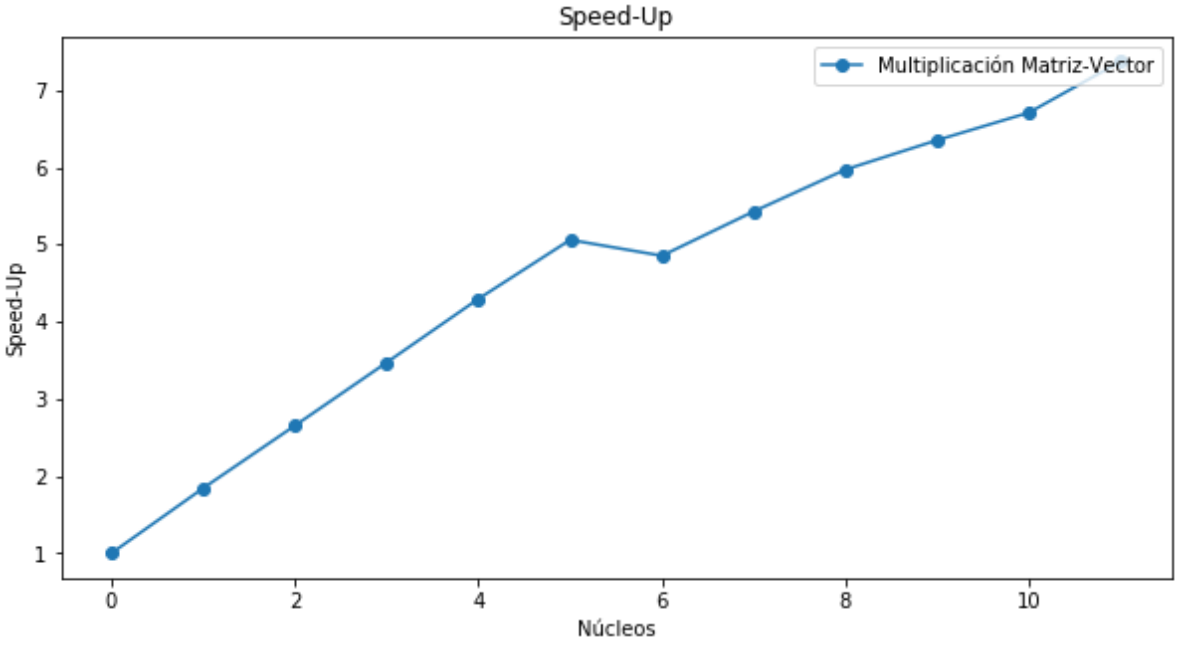
```
1  void matrix_vector_mult(double * a, double * b, double * c, int size) {
2      #pragma omp parallel for
3      for (int i = 0; i < size; i++) {
4          double sum = 0;
5          for (int j = 0; j < size; j++) {
6              sum += a[i * size + j] * b[j];
7          }
8          c[i] = sum;
9      }
10 }
```

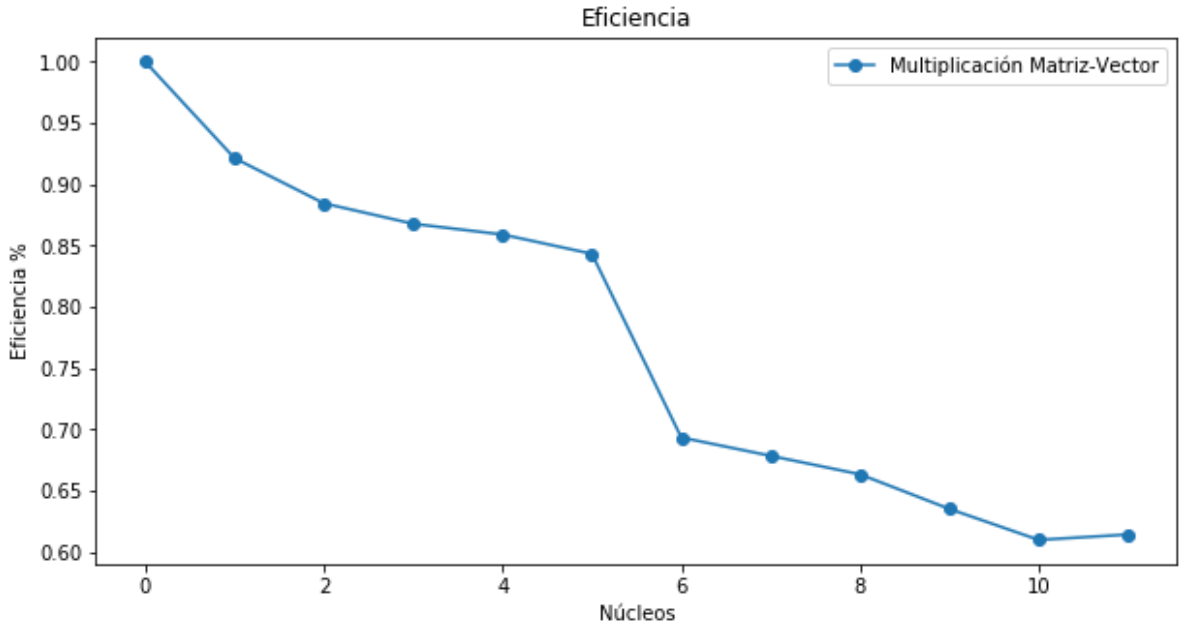
Se realizaron pruebas con tamaños de *array* de 10000000000 elementos con los siguientes resultados

```
Cores: 1: InputSize: 40000 Time 5.97s Memory: 2076Kb CPU: 99%
Cores: 2: InputSize: 40000 Time 3.24s Memory: 2004Kb CPU: 198%
Cores: 3: InputSize: 40000 Time 2.25s Memory: 1988Kb CPU: 296%
Cores: 4: InputSize: 40000 Time 1.72s Memory: 2004Kb CPU: 392%
Cores: 5: InputSize: 40000 Time 1.39s Memory: 2052Kb CPU: 487%
Cores: 6: InputSize: 40000 Time 1.18s Memory: 2056Kb CPU: 576%
Cores: 7: InputSize: 40000 Time 1.23s Memory: 2024Kb CPU: 589%
Cores: 8: InputSize: 40000 Time 1.10s Memory: 2136Kb CPU: 697%
Cores: 9: InputSize: 40000 Time 1.00s Memory: 2140Kb CPU: 805%
Cores: 10: InputSize: 40000 Time 0.94s Memory: 2076Kb CPU: 901%
Cores: 11: InputSize: 40000 Time 0.89s Memory: 2012Kb CPU: 1012%
Cores: 12: InputSize: 40000 Time 0.81s Memory: 2092Kb CPU: 1127%
```

Las siguientes gráficas muestran los resultados para las medidas de rendimiento mencionadas anteriormente.







Multiplicación matriz matriz en paralelo

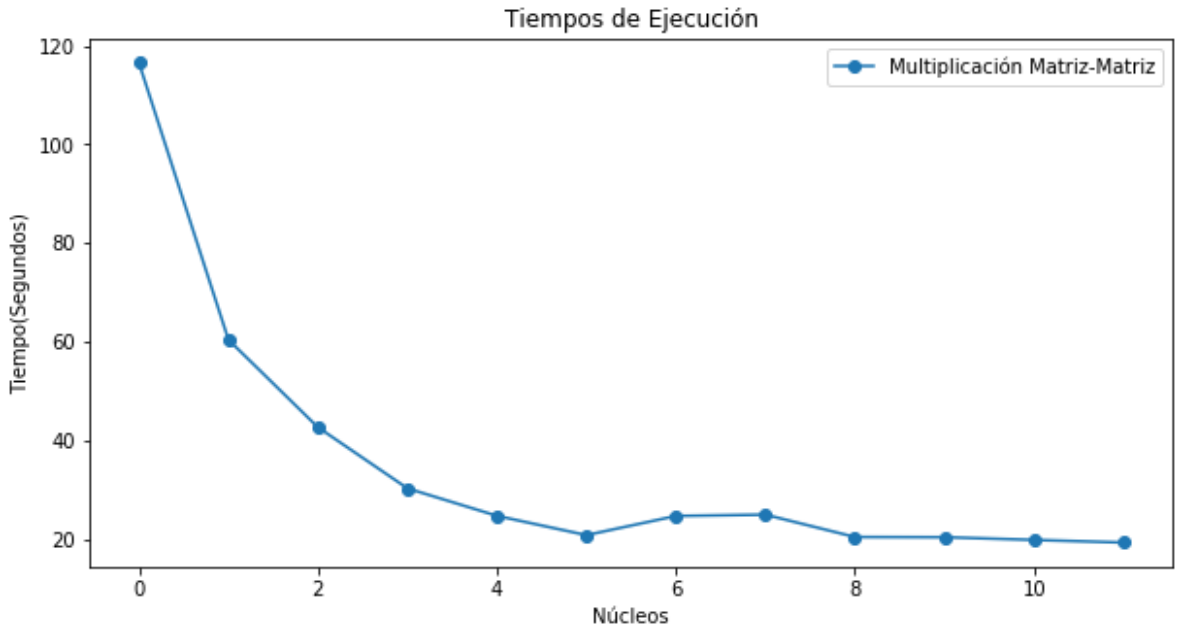
El código de paralelización es el siguiente:

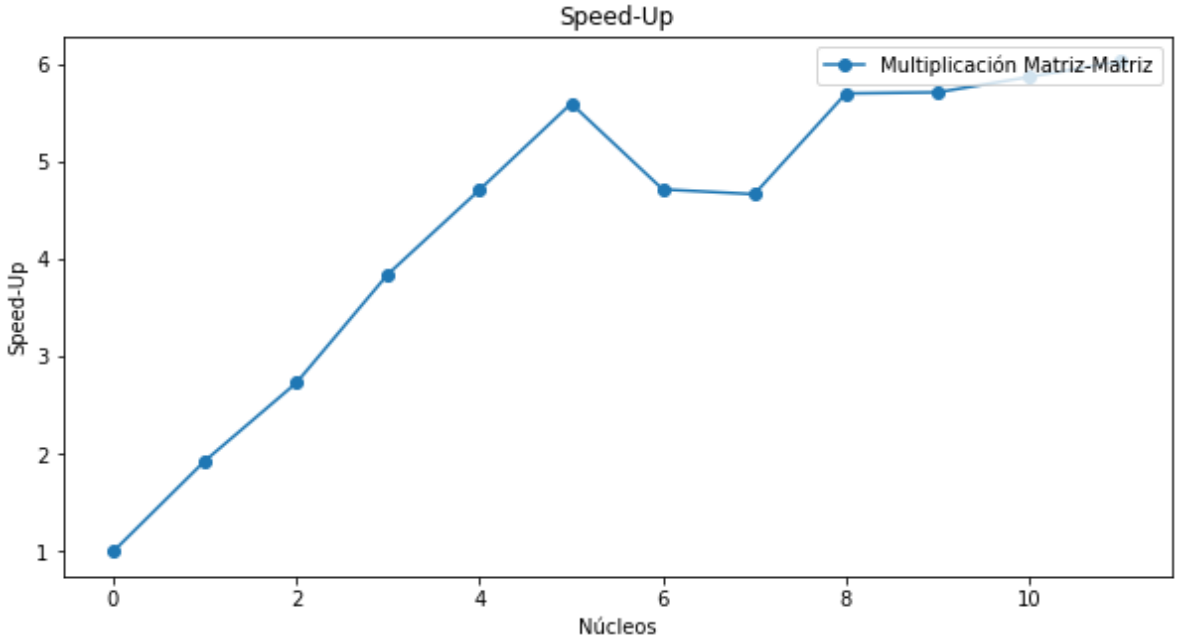
```
void matrix_matriz_mult(double * a, double * b, double * c, int size) {
    #pragma parallel for
    for (int i = 0; i < size * size; i++) {
        c[i] = 0;
    }
    #pragma omp parallel for
    for (int i = 0; i < size; i++) {
        for (int k = 0; k < size; k++) {
            for (int j = 0; j < size; j++) {
                c[i * size + j] += a[i * size + k] * b[k * size + j];
            }
        }
    }
}
```

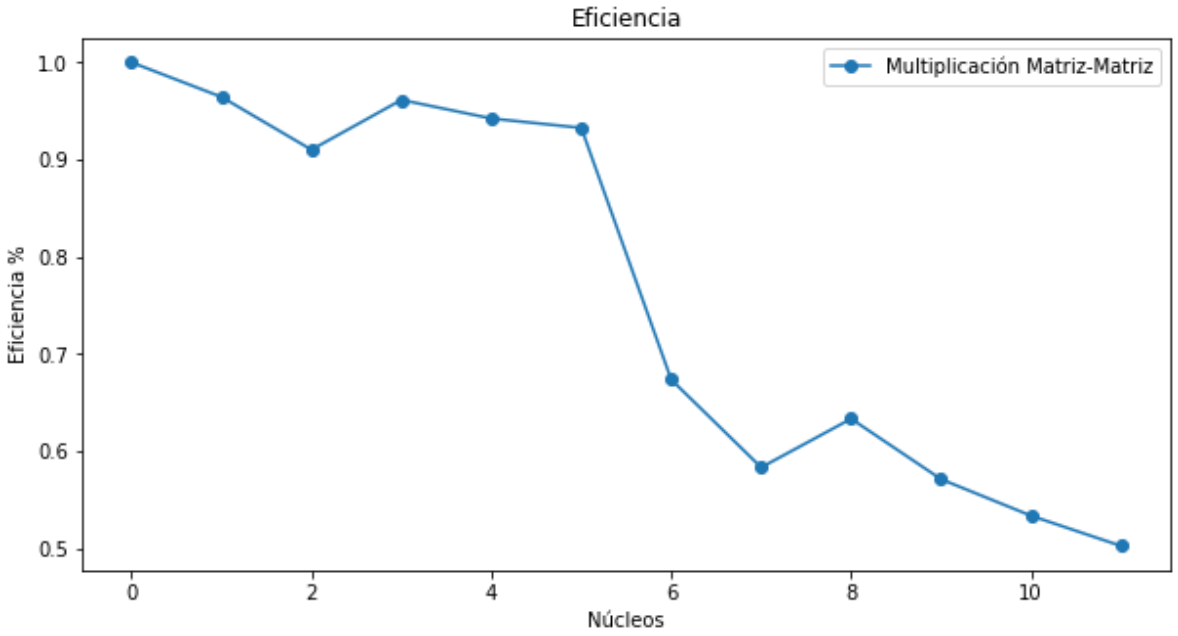
Se realizaron pruebas con tamaños de *array* de 10000000000 elementos con los siguientes resultados

```
Cores: 1: InputSize: 3000 Time 116.56s Memory: 72044Kb CPU: 99%
Cores: 2: InputSize: 3000 Time 60.43s Memory: 71988Kb CPU: 199%
Cores: 3: InputSize: 3000 Time 42.69s Memory: 72040Kb CPU: 290%
Cores: 4: InputSize: 3000 Time 30.31s Memory: 72056Kb CPU: 393%
Cores: 5: InputSize: 3000 Time 24.74s Memory: 72052Kb CPU: 489%
Cores: 6: InputSize: 3000 Time 20.83s Memory: 72072Kb CPU: 579%
Cores: 7: InputSize: 3000 Time 24.72s Memory: 72060Kb CPU: 553%
Cores: 8: InputSize: 3000 Time 24.98s Memory: 72084Kb CPU: 610%
Cores: 9: InputSize: 3000 Time 20.45s Memory: 72104Kb CPU: 786%
Cores: 10: InputSize: 3000 Time 20.41s Memory: 72120Kb CPU: 901%
Cores: 11: InputSize: 3000 Time 19.86s Memory: 72228Kb CPU: 1016%
Cores: 12: InputSize: 3000 Time 19.33s Memory: 72248Kb CPU: 1141%
```

Las siguientes gráficas muestran los resultados para las medidas de rendimiento mencionadas anteriormente.







Observaciones y mejoras

La máquina en dónde se realizaron las pruebas tiene un procesador con 6 núcleos físicos y 6 lógicos, se puede observar en las gráficas de medidas de rendimiento que el uso mayor a 6 threads no hay mejora significativa, comprobando que el uso de los núcleos lógicos no aportan demasiado a la paralelización.

Por otro lado, analizando la estructura de los códigos anteriores puede ser válidos paralelizar los ciclos anidados, por ejemplo:

```
void matrix_vector_mult(double * a, double * b, double * c, int size) {
    #pragma omp parallel for
    for (int i = 0; i < size; i++) {
        double sum = 0;
        #pragma omp parallel for reduction(+:sum)
        for (int j = 0; j < size; j++) {
            sum += a[i * size + j] * b[j];
        }
        c[i] = sum;
    }
}
```

Sin embargo, el tener más threads no siempre significa que será más rápido, en este caso, la carga de trabajo de cada thread tiene que ser atendida por algún procesador, así, mientras los 6 procesadores estén en uso, las tareas de los threads que no son atendidos están esperando en la cola de procesamiento, siendo innecesario así el paralelizar los ciclos anidados.