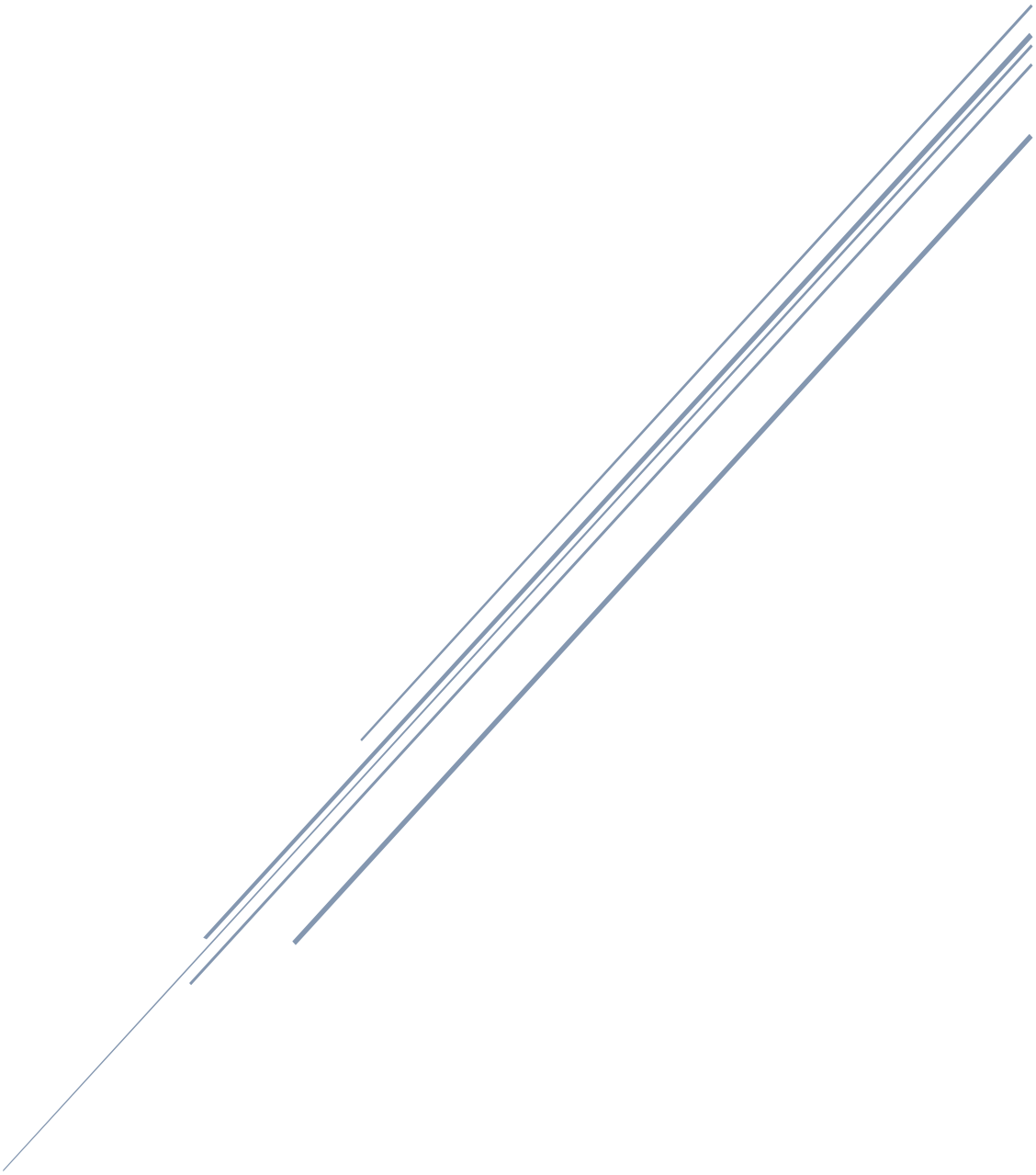
DSAP

Practica 2



Francisco Joaquín Murcia Gómez

Índice

Código empleado

En primer lugar creamos un vector estático de tamaño TAM que el usuario pasa por parámetro (siendo entre 1 y 100000) y se han creado unas variables para poder repartir el tamaño entre los diferentes procesos y rellenamos el vector.

```
//parte inicial
printf("tamaño del vector (1-100000): ");
scanf("%d", &TAM);
if(TAM<1 || TAM>MAXTAM)return 0;

int vector[TAM];//creamos un vector de punteros con tamaño TAM
TAMrepartido=TAM/numprocs;//indicamos el tamaño que le toca a cada proceso
resto=TAM%numprocs;
indice=resto+TAMrepartido;//indicamos el tamaño que va a tener el proceso padre
//srand(1);
for(int i=0;i<TAM;i++)//rellenmos el vector
{
   vector[i]=rand()%MAXENTERO+1;
}
```

Código secuencial

Para realizar la búsqueda simplemente se ha realizado dos bucles anidados que recorren el tamaño y lo repiten según el número de repeticiones.

```
for (int i=0; i<REPETICIONES; i++){//recorremos el vecor y contamos los buscados
   for (int j=0; j<TAM; j++){
      if (vector[j] == NUMBUSCADO){
         numVeces++;
      }

   }
}
```

Código paralelo

Para el código paralelo, el proceso 0 envía los datos a del vector, el tamaño que le toca a cada uno y el índice donde comienzan.

```
start_time=MPI_Wtime();//iniciamos el crono
int indiceDeHilos=indice;
//hacemops las llamadas a los procesos hijos
//enviandoles el indece donde comenzara su vecor y su tamaño para que construyan el vector
for (int i = 1; i < numprocs; i++)
{
   MPI_Send(&TAMrepartido,1,MPI_INT,i,1,MPI_COMM_WORLD);
   MPI_Send(&vector[indiceDeHilos],TAMrepartido,MPI_INT,i,1,MPI_COMM_WORLD);
   indiceDeHilos+=TAMrepartido;

}
```

Una vez enviado todo en proceso 0 realizara la búsqueda de la misma manera que en el código secuencial.

En el código de los procesos hijos recibimos los datos del vector que nos ha enviado el proceso 0.

```
}else{//codigo de los procesos hijos
   MPI_Recv(&TAMrepartido,1,MPI_INT,0,1,MPI_COMM_WORLD,&estado);//recivimos el tamaño del vector
   int vector[TAMrepartido];//creamos el vectro que va a leer
   MPI_Recv(vector,TAMrepartido,MPI_INT,0,1,MPI_COMM_WORLD,&estado);//recivimos el trozo de vector que nos interesa para este proceso

   for (int i=0; i<REPETICIONES; i++){//contamos
```

Una vez recibido el vector realizamos la búsqueda como en el cogito secuencial.

```
for (int i=0; i<REPETICIONES; i++){//contamos
   for (int j=0; j<TAMrepartido; j++){
      if (vector[j] == NUMBUSCADO){
         cont++;
      }

   }
}
```

Finalmente enviamos al proceso 0 el contador.

```
MPI_Send(&cont,1,MPI_INT,0,1,MPI_COMM_WORLD);//enviamos el resulatdo al padre
```

Una vez calculado todo el proceso 0 recibe los datos y sumamos todo.

```
for (int i = 1; i < numprocs; i++)//recivimos y sumamos los resultados de los procesos hijos
{
   MPI_Recv(&cont,1,MPI_INT,i,1,MPI_COMM_WORLD,&estado);
   resultado+=cont;
}
```

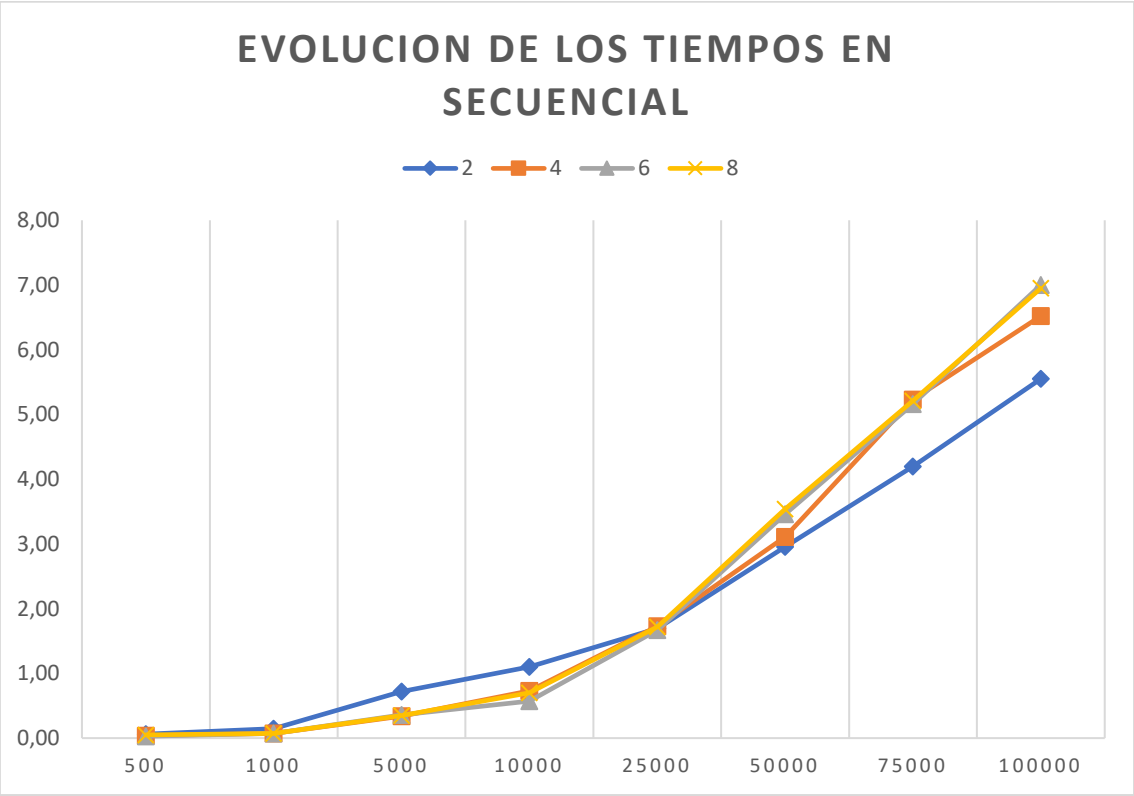
Por último, calculamos es Speed-up y la eficiencia.

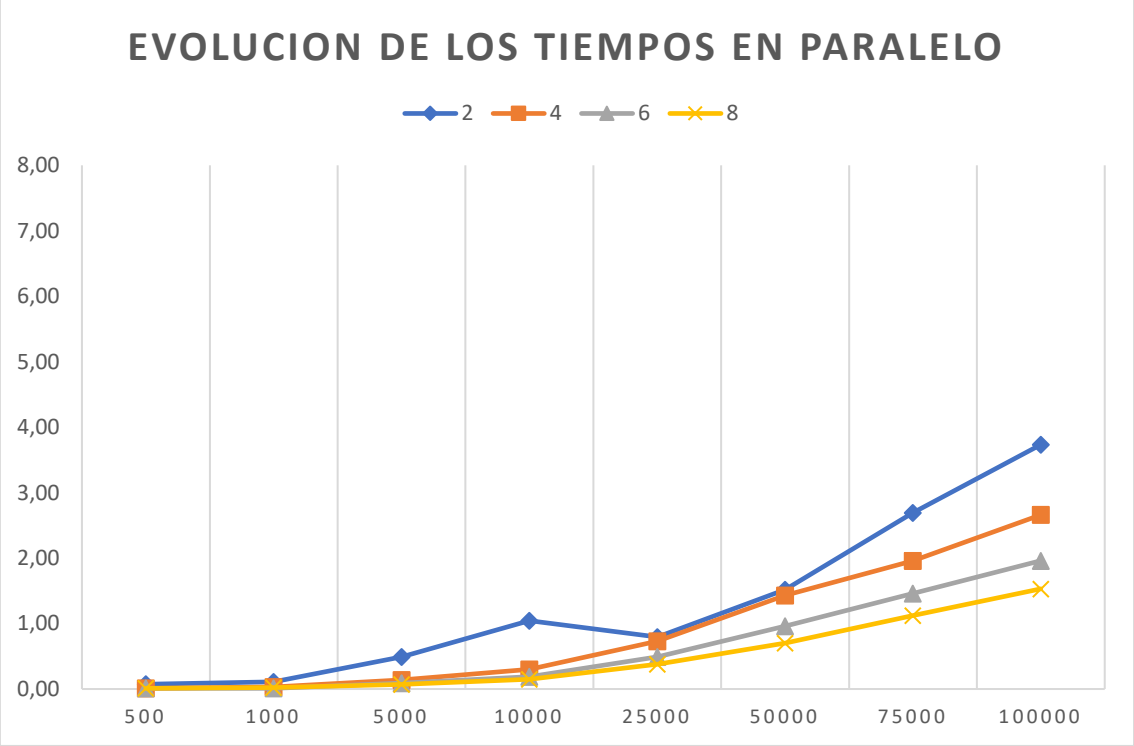
```
printf("============Informe===========\n\n");
double sp = secuencial / paralelo;
printf("Speed-up= %f \n",sp);
printf("Eficiencia= %f \n",sp/(double)numprocs);
```

Resultados

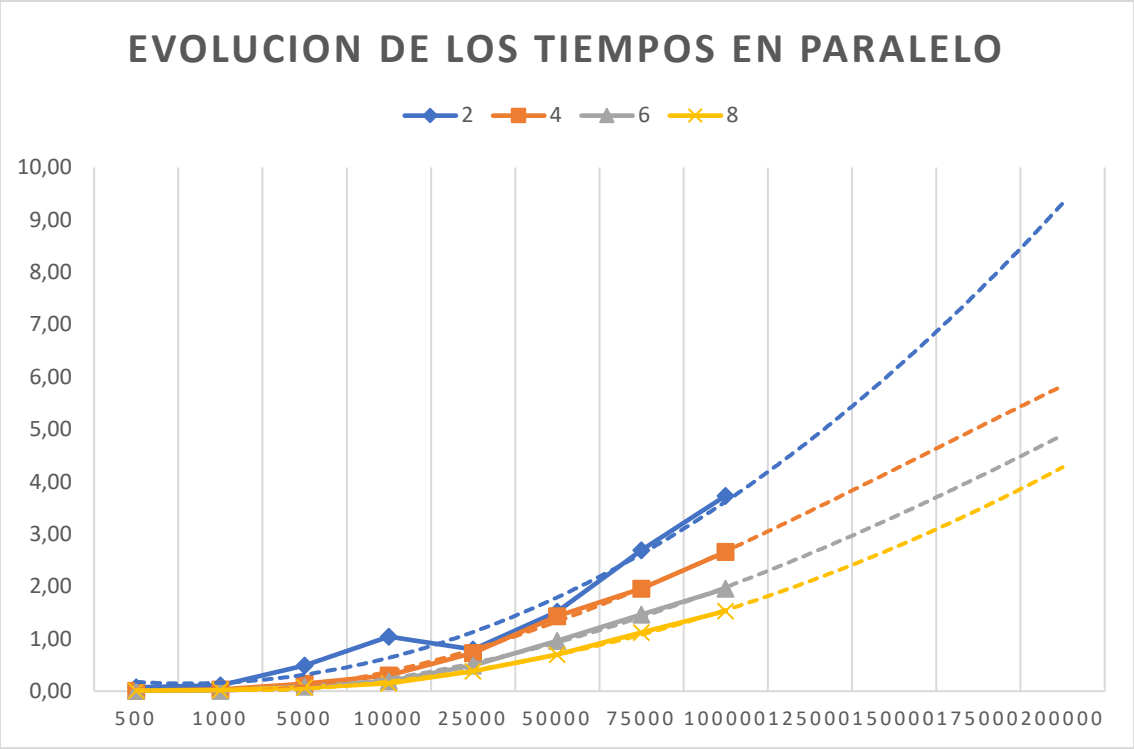
Para los resultados se ha probado con un tamaño de vector de 500, 1000, 1500, 10000, 15000, 25000, 50000, 75000 y 100000 elementos en dual-core, tetra-core y Hexa-core y octa-core. Se ha ejecutado cada prueba 5 veces y se ha hecho la mediana de estas para así tener un resultado más realista y evitar cualquier pico de rendimiento esporádico. En estas ejecuciones tomamos el tiempo de la ejecución secuencial, de la paralela y calculamos el Speed-up y la eficiencia.

Comparativa de los tiempos obtenidos

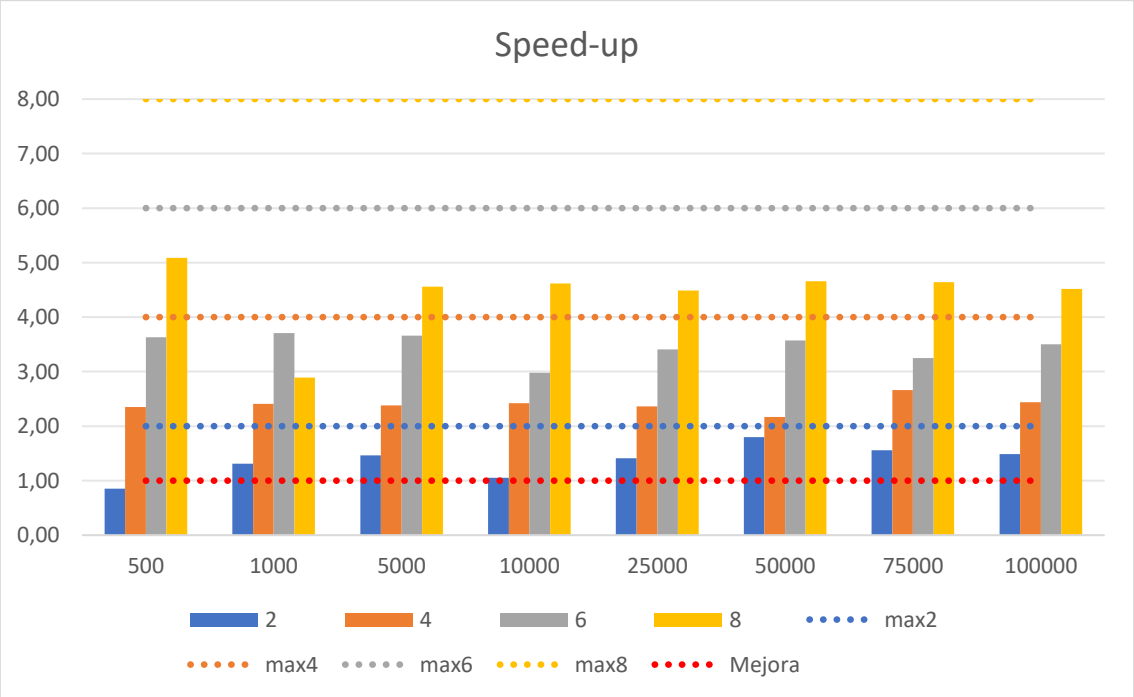




Como podemos observar si ejecutamos secuencialmente salen tiempos mucho mas altos que en su contraparte paralela, también tiene ascienden de una forma más homogénea. Podemos observar también que en las ejecuciones secuenciales con más de dos núcleos salen mejores tiempos que en su contraparte paralela que se separa más que el resto. La separación es tal que si extrapolamos los resultados interpolando estos hasta un tamaño 200000 de vector podemos observar esta separación de la tendencia general.

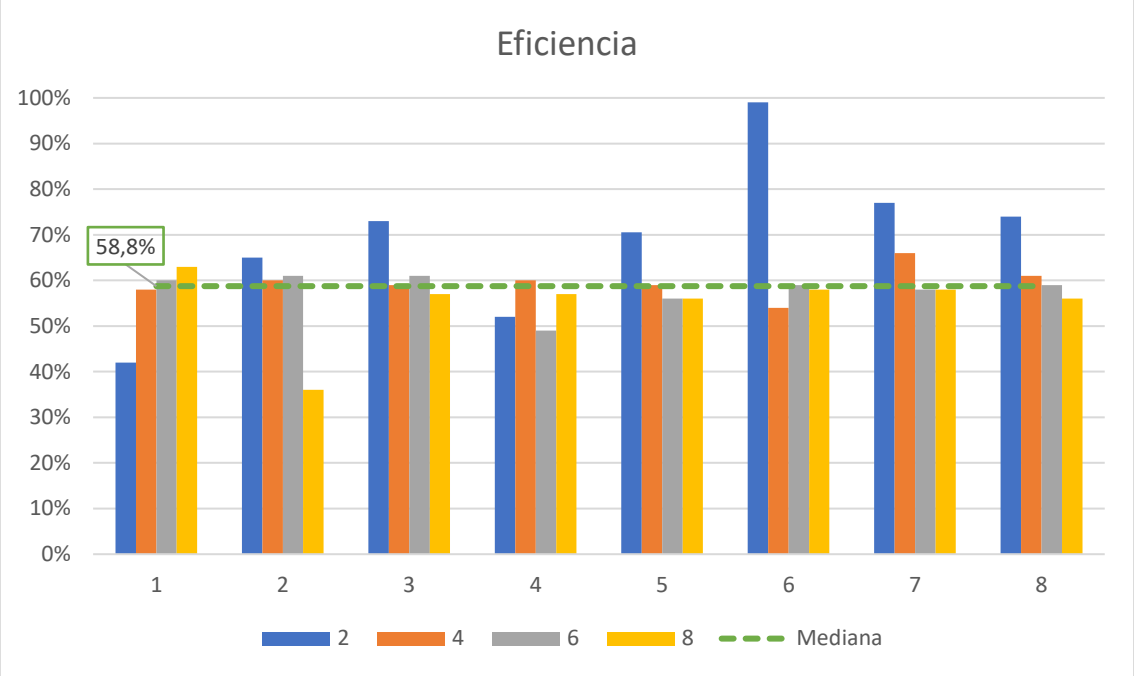


Speed-up y eficiencia



En la grafica podemos ver los diferentes Speed-ups (SP) también observamos los límites de estos, puesto que los SPs no pueden ser superiores al número de núcleos empleados y también observamos el punto de donde se consideraría mejora (con SP=1).

Vemos que con 2 núcleos cuando tenemos poca carga de trabajo es más rentable realizarlo de forma secuencial. También observamos que el código no es capaz de sacar partido a los núcleos ya que el SP es un poco más de la mitad de la "perfección de cada núcleo". Esto lo podemos ver más detalladamente con la eficiencia.



Aquí podemos observar lo ya mencionado. La eficiencia en ejecución paralela es de aproximadamente casi un 60% (calculado con la mediana para evitar valores atípicos). De hecho, si comparamos la evolución del SP con la eficiencia encontramos que cuantos más núcleos empleemos, menos eficiente se vuelve. Teniendo la máxima eficiencia con 2 núcleos.