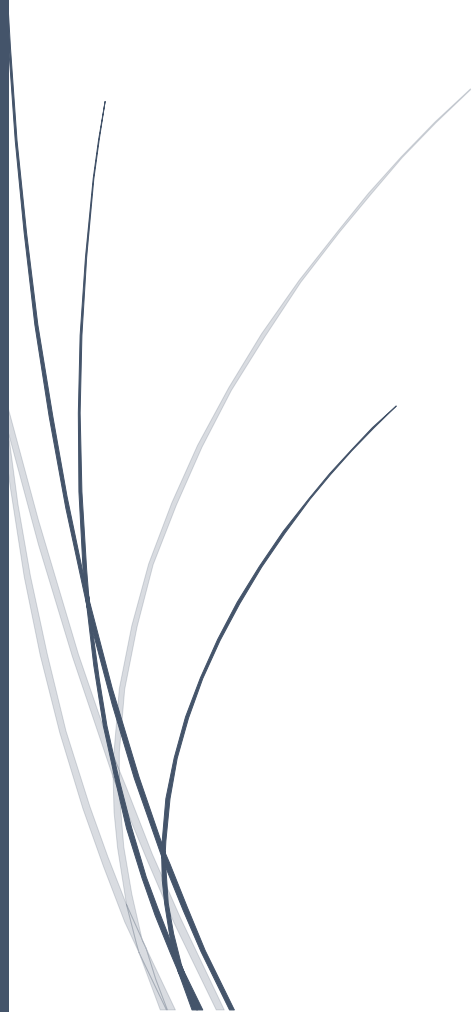
18-3-2022

PRÁCTICA 2: CONTAR

Desarrollo de software en arquitecturas paralelas



Francisco Javier Pérez Martínez
UNIVERSIDAD DE ALICANTE |GRADO EN INGENIERÍA INFORMÁTICA

Contenido

Descripción

Para el presente documento, correspondiente a la segunda práctica de la asignatura, se comparará el comportamiento de un algoritmo paralelo y su versión secuencial, obteniendo ciertas medidas de paralelismo como son el speed-up y la eficiencia.

El problema en cuestión es un algoritmo secuencial que cuenta el número de veces que un determinado número aparece en un conjunto grande de elementos almacenados en un vector.

Pasos seguidos

Cálculo secuencial

El proceso root es el encargado de leer una variable que determina el tamaño del vector de tipo entero y enviarla al resto de procesos. Esta variable se pedirá al usuario mientras el tamaño esté en el rango permitido. Además, root generará el vector de forma aleatoria con valores entre 0 y MAXENTERO (=100) y realizará el cálculo secuencial siguiendo el código del enunciado.

```c
if (myrank == 0) { // proceso root

        printf("# ------------ CALCULO SECUENCIAL ------------ #\n");

        do {
            printf("Introduce el tamaño del vector (1-100000): ");
            scanf("%d", &tamanyo);
        }while(tamanyo < 0 || tamanyo > MAXTAM);

        if (tamanyo > 0 && tamanyo <= MAXTAM) {

            vector = (int *) malloc(tamanyo * sizeof(int)); // vector dinámico con malloc.

            srand(1);

            for (int i = 0; i < tamanyo; i++) {
                vector[i] = 1 + ((double) (MAXENTERO) * rand()) / RAND_MAX;
            }

            // cálculo secuencial del proceso 0.
            numVeces = 0;
            start_time = MPI_Wtime();

            for (int i = 0; i < REPETICIONES; i++) { // repetimos la búsqueda según *REPETICIONES*.
                for (int j = 0; j < tamanyo; j++) { // recorremos el vector.
                    if (vector[j] == NUMBUSCADO) numVeces++; // si el nº en el vector es igual al NUMBUSCADO contador++.
                }
            }
            end_time = MPI_Wtime();
            sequential_time = end_time - start_time;
            printf("Veces que aparece el %d en %d repeticiones del vector de tamaño %d: %d veces, el %5.2f% \n",
                    NUMBUSCADO, REPETICIONES, tamanyo, numVeces, ((100.0*numVeces)/tamanyo) / REPETICIONES);
            printf("Tiempo de proceso: %f \n \n", sequential_time);
```

Para la declaración del vector se ha utilizado ***malloc*** para crear un vector dinámico ya que a priori no sabemos el espacio que vamos a necesitar para los procesos hijos. Esto se decidirá en tiempo de ejecución y llevado a cabo mediante la función de gestión de memoria mencionada.

Cálculo paralelo

Para la implementación paralela, el primer paso seguido es comprobar que elementos se le asociarán al proceso root y cuales al resto.

En primer lugar, comprobaremos si el número de procesos divide al número de elementos en el vector, en caso de no, el exceso de elementos se le asociará al proceso root.

La variable “indice_root” contempla hasta donde va a leer el proceso root.

A partir de la línea 10 de la imagen, se le envía el tamaño “n” y la posición de donde van a empezar los procesos restantes. Para el siguiente proceso, vamos aumento el índice del siguiente proceso utilizando la variable “indiceProceso”.

El siguiente for se encarga de realizar el proceso de búsqueda del exceso de elementos asociado al proceso root.

```
1  printf("# ------------ CALCULO PARALELO ------------ #\n");
2              numVecesParalelo = 0;
3
4              n = tamanyo / numprocs; // tamaño de los vectores que se van a pasar.
5              indice_root = n + (tamanyo % numprocs); // hasta donde va a leer el proceso root.
6
7              indiceProceso = indice_root; // variable auxiliar para indicarle a cada proceso en que posición del vector debe empezar.
8              start_time = MPI_Wtime();
9
10             for (int i = 1; i < numprocs; i++) {
11                 MPI_Send(&n, 1, MPI_INT, i, i, MPI_COMM_WORLD); // le pasamos el tamaño.
12                 MPI_Send(&vector[indiceProceso], n, MPI_INT, i, i, MPI_COMM_WORLD); // donde van a empezar los procesos restantes.
13                 indiceProceso += n; // vamos aumentando el indice para el siguiente proceso.
14             }
15             // paralelo en el proceso root
16             for (int i = 0; i < REPETICIONES; i++) { // repetimos la búsqueda según *REPETICIONES*.
17                 for (int j = 0; j < indice_root; j++) { // el proceso root recorre hasta el indice indicado.
18                     if (vector[j] == NUMBUSCADO) numVecesParalelo++; // si el nº en el vector es igual al NUMBUSCADO contador++.
19                 }
20             }
```

El código que seguirá el resto de los procesos recibirá el tamaño que se le ha enviado anteriormente, después se reserva memoria siguiendo el tamaño enviado para los procesos restantes y recibimos los datos del vector.

Al igual que antes, se realiza el proceso de búsqueda, pero esta vez hasta la variable “n”, es decir, el tamaño del vector de cada proceso. Y enviamos al proceso root el número de veces que se ha encontrado el número buscado (NUMBUSCADO).

```
1  if (myrank != 0) { // codigo de los procesos restantes.
2          numVecesParalelo = 0;
3          MPI_Recv(&n, 1, MPI_INT, 0, myrank, MPI_COMM_WORLD, &estado); // recibimos el tamaño.
4          vector = (int *) malloc(n * sizeof(int)); // reservamos el tamaño de cada proceso restante con malloc.
5          MPI_Recv(vector, n, MPI_INT, 0, myrank, MPI_COMM_WORLD, &estado); // recibimos los datos del vector.
6
7          for (int i = 0; i < REPETICIONES; i++) {
8              for (int j = 0; j < n; j++) {
9                  if (vector[j] == NUMBUSCADO) numVecesParalelo++;
10             }
11         }
12         // enviamos al proceso root el numero de veces que se ha encontrado NUMBUSCADO.
13         MPI_Send(&numVecesParalelo, 1, MPI_INT, 0, myrank, MPI_COMM_WORLD);
14     }
```

Finalmente, recibiremos los datos de los procesos restantes y lo sumaremos desde el proceso root para obtener el numero de veces que ha aparecido el valor en la parte paralela, almacenando dicho valor en la variable “cuenta_total”. Y luego, calcularemos el speed-up y la eficiencia con los tiempos obtenidos.

```
if (myrank == 0) { // miramos si estamos en el proceso root.
        cuenta_total = numVecesParalelo;
        for (int i = 1; i < numprocs; i++) { // recibir los datos de los procesos restantes y sumarlos en la variable cuenta_total.
            MPI_Recv(&numVecesParalelo, 1, MPI_INT, i, i, MPI_COMM_WORLD, &estado);
            cuenta_total += numVecesParalelo;
        }
        end_time = MPI_Wtime();
        parallel_time = end_time - start_time;

        printf("Veces que aparece el %d en %d repeticiones del vector de tamaño %d: %d veces, el %5.2f% \n",
            NUMBUSCADO, REPETICIONES, tamanyo, cuenta_total, ((100.0*cuenta_total)/tamanyo) / REPETICIONES);
        printf("Tiempo de proceso: %f \n \n", parallel_time);
```

Speed-up y Eficiencia

Los valores de Speed-up y Eficiencia calculados han sido obtenidos desde los ordenadores del aula. Se ha variado el numero de procesadores como se indica en el enunciado 2, 4, 6 y 8. Y también, el tamaño del vector de 10000 en 10000 hasta llegar al tamaño máximo.

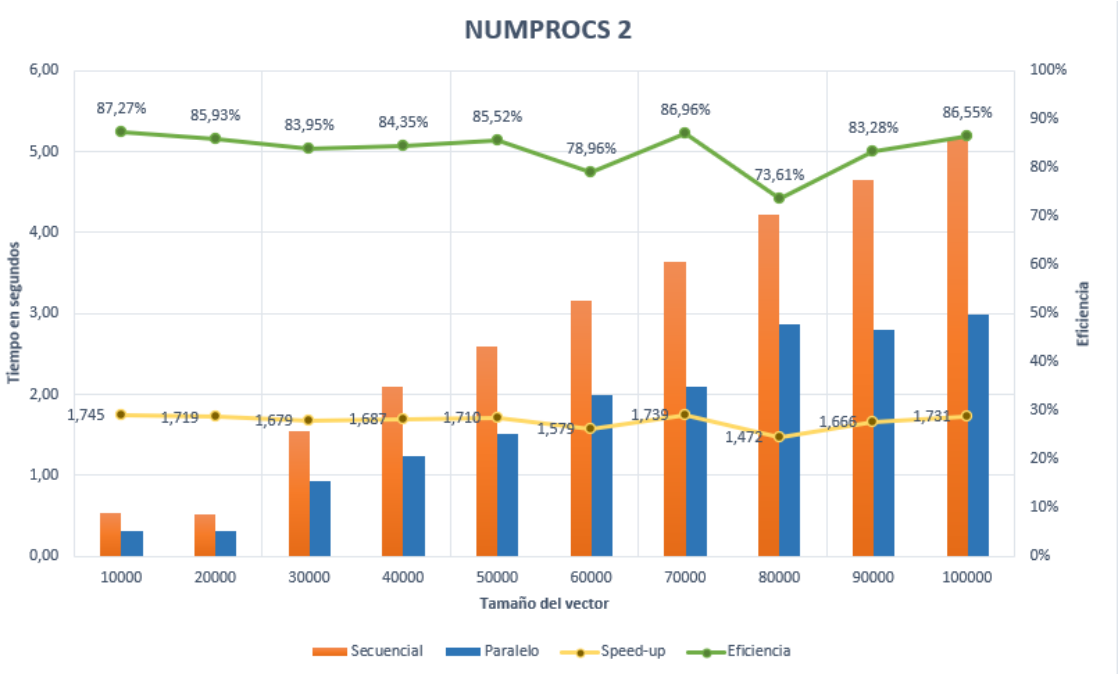
```
// Cálculo del speed-up y eficiencia.

        double sp = sequential_time / parallel_time;
        printf("Speed-up = %f \n", sp);
        printf("Eficiencia = %5.2f% \n", (sp/numprocs) * 100.0);
    }
    free(vector); // liberar vector instanciado con malloc.
    MPI_Finalize()
```

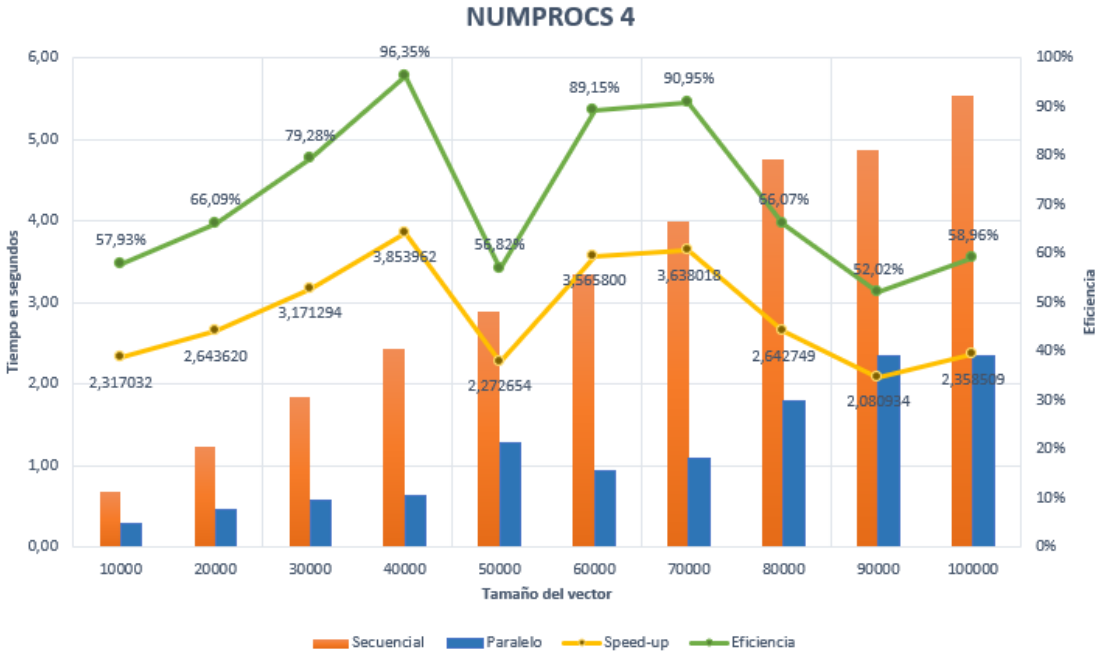
Cálculo y gráficas

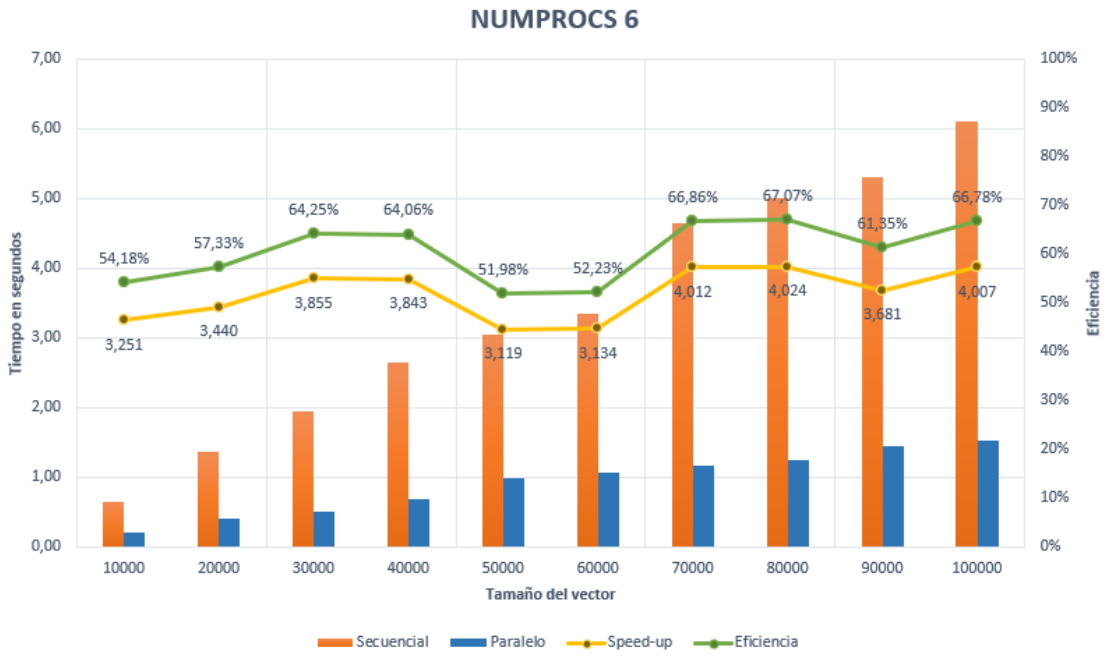
NUMPROCS	2				4			
DATOS	Secuencial	Paralelo	Speed-up	Eficiencia	Secuencial	Paralelo	Speed-up	Eficiencia
10000	0,535915	0,307029	1,745	87,27%	0,681284	0,294033	2,317032	57,93%
20000	0,522799	0,304187	1,719	85,93%	1,234946	0,467142	2,643620	66,09%
30000	1,552751	0,924794	1,679	83,95%	1,842801	0,581088	3,171294	79,28%
40000	2,102717	1,246474	1,687	84,35%	2,426547	0,629624	3,853962	96,35%
50000	2,592298	1,515524	1,710	85,52%	2,894963	1,273825	2,272654	56,82%
60000	3,150769	1,995167	1,579	78,96%	3,347527	0,938787	3,565800	89,15%
70000	3,642299	2,094333	1,739	86,96%	3,992503	1,097439	3,638018	90,95%
80000	4,221279	2,867194	1,472	73,61%	4,752442	1,798295	2,642749	66,07%
90000	4,650899	2,792354	1,666	83,28%	4,861993	2,336447	2,080934	52,02%
100000	5,160588	2,981166	1,731	86,55%	5,526286	2,343127	2,358509	58,96%

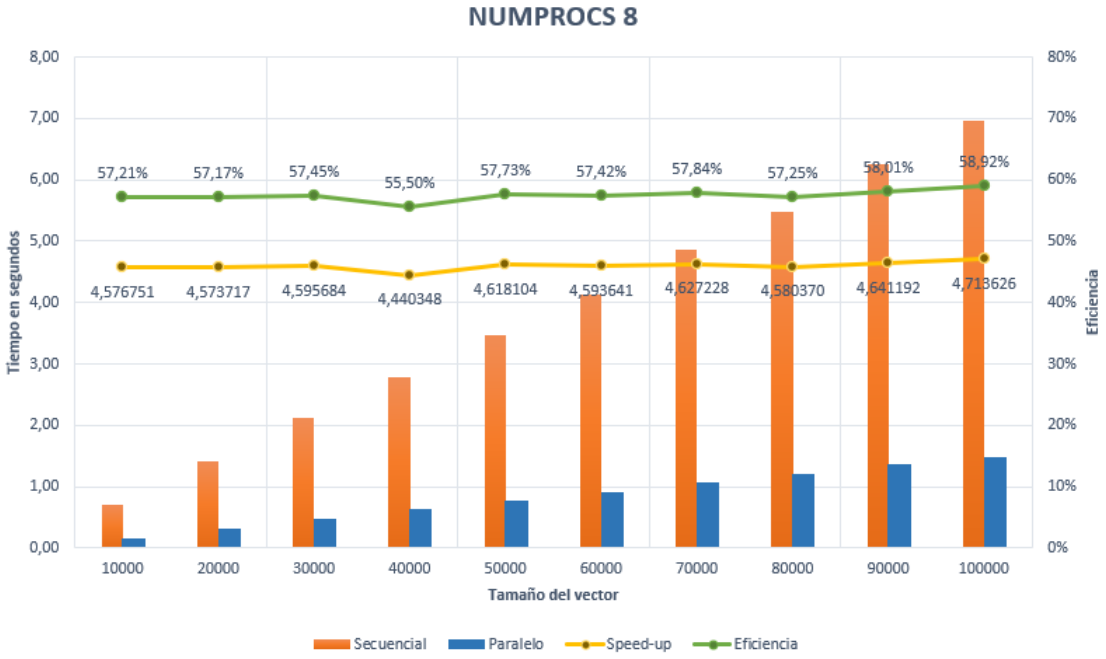
6				8			
Secuencial	Paralelo	Speed-up	Eficiencia	Secuencial	Paralelo	Speed-up	Eficiencia
0,645565	0,198578	3,251	54,18%	0,706568	0,154382	4,577	57,21%
1,376249	0,400069	3,440	57,33%	1,408069	0,307861	4,574	57,17%
1,944715	0,504492	3,855	64,25%	2,128546	0,463162	4,596	57,45%
2,644293	0,688024	3,843	64,06%	2,786105	0,627452	4,440	55,50%
3,056524	0,980097	3,119	51,98%	3,471992	0,751822	4,618	57,73%
3,355705	1,070898	3,134	52,23%	4,124125	0,89779	4,594	57,42%
4,646725	1,158275	4,012	66,86%	4,854069	1,049023	4,627	57,84%
5,001204	1,24283	4,024	67,07%	5,489908	1,198573	4,580	57,25%
5,314399	1,443709	3,681	61,35%	6,262063	1,349236	4,641	58,01%
6,112538	1,525619	4,007	66,78%	6,970869	1,478876	4,714	58,92%

Hay que indicar que estos valores han sido obtenidos compilando el código utilizando la opción -O2 (optimización de nivel 2).

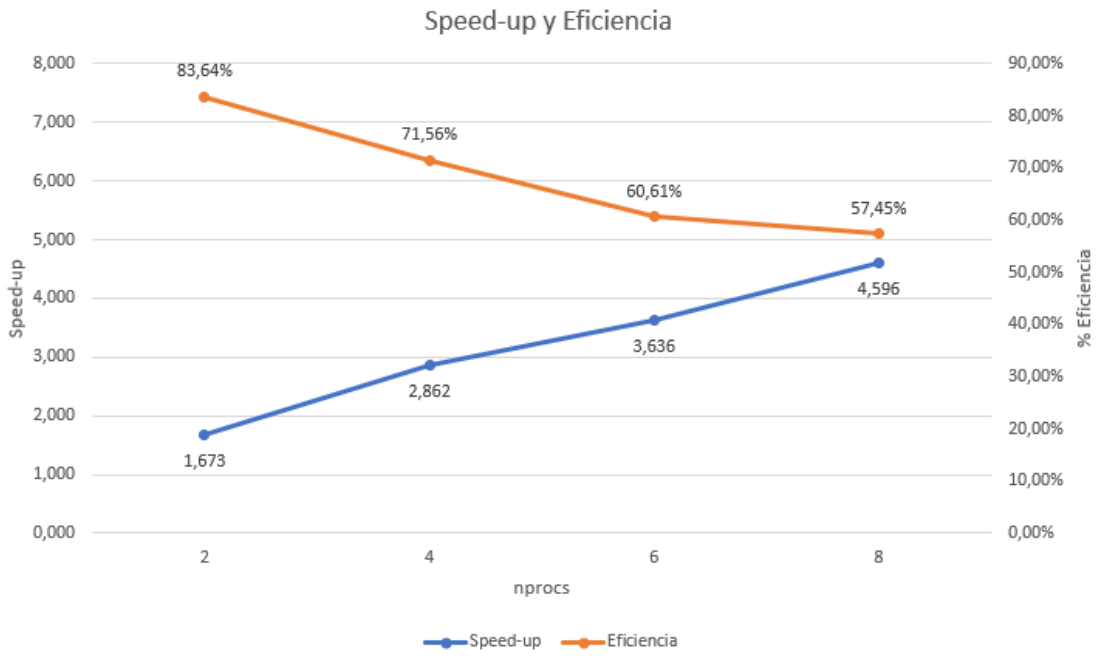








A partir de las tablas obtenidas, generalmente podemos ver como los tiempos en secuencial tardan más en comparación a los tiempos en paralelo. En cuanto a la eficiencia entre los diferentes números de procesadores usados, podemos apreciar que usando 2 procesadores el código se comporta de una forma mas eficiente ya que el valor de speed-up obtenido se aproxima más al número de procesadores usados a diferencia de cuando aumentamos el número de procesadores, por ejemplo, en las tablas de 6 y 8 nprocs.



Esta última tabla, contempla una comparativa entre la media obtenida a partir de los valores calculados en cada número de procesos usados del speed-up y eficiencia.

Realizando un último cálculo de la media de todas las eficiencias, obtenemos que el código implementado posee un 70% aproximadamente de eficiencia en cuanto a la ejecución paralela. Finalmente, fijándonos en la tabla, podemos observar que usando 2 procesadores tenemos una mayor eficiencia a diferencia de si vamos usando más procesadores.