Práctica 1: Contar

Juan José Conejero
jjcs2@alu.ua.es

Desarrollo Software en Arquitecturas Paralelas.

Resumen El objetivo de esta practica es comparar el comportamiento de un algoritmo paralelo y su version secuencial, obteniendo ciertas medidas de paralelismo como son el speed-up y la eficiencia. El problema que nos interesa es un algoritmo secuencial que cuenta el numero de veces que un determinado numero aparece en un conjunto grande de elementos almacenados en un vector. Analizaremos los problemas principales que plantea la práctica y una posible solución; se expondrán los datos resultantes de una serie de experimentos llevados a cabo en el laboratorio L01 de la Escuela Politécnica Superior y finalmente, se extraerán las conclusiones obtenidas de los resultados experimentados.

1. Introducción

Para simular una búsqueda en un conjunto mayor de datos sin necesidad de utilizar tanta memoria, lo que haremos es buscar un número determinado de veces en una misma estructura de datos (array de enteros).

La **versión secuencial** se invoca con un parámetro "tamanyo" que determina el tamaño del vector. Si fijamos un tamaño de 200.000, tendremos un vector de ese tamañoo donde se colocan números generados al azar comprendidos entre 0 y 1.000 (MAXENTERO). En este arreglo se busca el número indicado en la constante NUMBUSCADO. Todo ello se repite un número determinado de veces representado por la constante REPETICIONES.

Para la **versión paralela**, la distribución de datos debe realizarse de forma equitativa entre todos los procesos (contando el proceso root). Si el número de procesos no divide al número de elementos en el vector, el exceso de elementos se le asociará al proceso ***root***

2. Implementación

Cuando se paraleliza una aplicación, a parte del algoritmo que se debe paralelizar, se debe tener en cuenta otra serie de cuestiones entre las que se encuentran la dispersión de datos a otras máquinas, la comunicación, la sincronía y el análisis final de datos. En esta sección se analizarán estas cuestiones y como se le da solución mediante el uso de la librería **MPI**.

Algoritmo 1 Búsqueda de numero ***NUMBUSCADO*** sobre ***vector***, ***REPETICIONES*** veces

```
int buscarNumero(int *vector, int tamanyo){
    int numVeces=0, i, j;
    for (i=0; i<REPETICIONES; i++) {
        for (j=0; j<tamanyo; j++) {
            if (subvector[j] == NUMBUSCADO) {
                numVeces++;
            }
        }
    }
    return numVeces;
}
```

2.1. Búsqueda de número

En un primero análisis del problema, se puede valorar que el algoritmo principal que debe realizar la máquina, es el de la búsqueda del número. Además, esta búsqueda se ha de realizar *REPETICIONES* veces en el mismo vector. Atendiendo a estas especificaciones, el una posible implementación del algoritmo es la que figura en el **Algoritmo 1.**

Este algoritmo se realiza de la misma forma, tanto en el modelo secuencial como en el modelo paralelo. No obstante, tendrán sus diferencias como se verá a continuación.

2.2. Proceso Padre

Cuando hablamos de cómputo paralelo, se deben tener en cuenta cuestiones como la comunicación entre los diferentes procesos que se producen de forma paralela. Para ello, tiene que existir un *'árbitro'* encargado de esa comunicación y de la integridad de los datos de los procesos que intervienen. A este proceso, se le denomina **proceso padre.**

En nuestra aplicación, el proceso padre es el encargado de:

1. Comprobar argumentos del programa
2. Generación de vector con números aleatorios
3. Cálculo del proceso secuencial
4. Distribución de datos a procesos hijos
5. Unión y análisis de los datos
6. Generación de informe

El elemento 1 de la lista, no se analizará en este docmento ya que se considera es una información básica y que se sobreentiende que ya se conoce el funcionamiento de la comprobación de parámetros en C. En el código adjunto a la práctica se pueden encontrar los métodos **comprobarArgumentos** y **isNumber** con sus comentarios adicionales.

Algoritmo 2 Variables y estructura de control de la aplicación

```
int *vector;
int i;
int idProceso, totalProcesos;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &idProceso);
MPI_Comm_size(MPI_COMM_WORLD, &totalProcesos);

if(idProceso==0){
    //Acciones proceso padre
}else{
    //Acciones proceso hijo
}
//Calculo de búsqueda del número
if(idProceso==0){
    //Recepcion resultados desde hijos
}else{
    //Envio de resultados a padre
}
```

2.2.1. Pre-condiciones de la aplicación Antes de realizar todos pasos necesarios, se debe declarar la estructura sobre la que se desarrollará la aplicación. Ésta, cuenta de dos partes:

1. Declaración de variables
2. Instrucción de control 'padre o hijo'.

En el **Algoritmo 2** se visualiza, de forma genérica como funciona la aplicación: si el id del proceso actual tiene valor 0, entonces es el padre y realiza las instrucciones necesarias. Si no, es un proceso hijo y realiza otra serie de instrucciones. Las variables son las mismas para todos los procesos aunque cada uno de ellos, le dará unos valores distintos.

2.2.2. Generación de números aleatorios La aplicación cuenta con un algoritmo el cual *rellena* el vector de números enteros, que posteriormente serán buscados por cada proceso. Una posible inicialización del vector de tamaño **tamanyo,** es la reflejada en el **Algoritmo 3.**

2.2.3. Proceso secuencial Continuando con la estructura de la aplicación, se debe calcular el tiempo de cálculo que genera el proceso secuencial del algoritmo. Para ello, tomamos un tiempo absoluto **antes** de realizar el **Algoritmo 1** y un tiempo absoluto **después** de realizarlo. La diferencia entre tiempos, es lo que necesita este algoritmo para realizarse.

Algoritmo 3 Reserva de memoria e inicialización del vector **vector**.

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

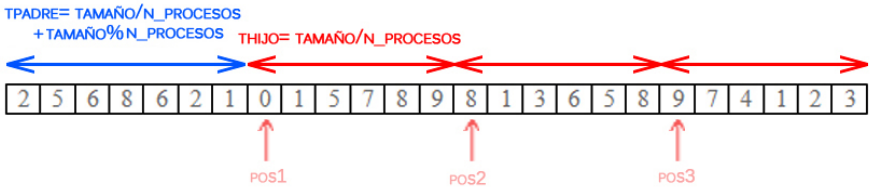


Figura 1. Representación de vector y distribucion del mismo en diferentes procesos

2.2.4. Distribución de procesos hijos El mecanismo a seguir es dividir el vector en partes iguales y enviar a cada proceso una sección de ese vector para que realice la búsqueda del número **NUMBUSCADO**. Sin embargo, esta división de partes iguales puede no ser entera. Por eso se incluye, en el padre, *tamaño MOD n_procesos*. En la **Figura 1** se puede observar una representación del método utilizado en la aplicación. En **azul** se encuentra la sección que tiene que analizar el proceso padre; en **rojo**, las secciones del vector que tienen que analizar los procesos hijos.

Una de las cuestiones principales a la hora de realizar una aplicación paralela, es la de elegir el mecanismo de comunicación que se va a utilizar. En el caso de la aplicación **contar** se utilizan dos rutinas: **send** y y **recv.**

El padre envía a los hijos una dirección de memoria que indica el punto de partida desde el cual tiene que analizar el vector, y un tamaño, que servirá de punto de finalización para el proceso. Siguiendo con la representación de la **Figura 1,** los punteros en **rosa** indican la posición de memoria que se enviará del proceso padre al proceso hijo.

Finalmente, en el **Algoritmo 4** se refleja una posible implementación del envío de los valores del proceso padre a los procesos hijos.

Una vez establecido el comportamiento de la aplicación si es el proceso **padre**, dentro de la **estructura** declarada que seguirá un flujo de control, se debe establecer que pasa si el proceso actual **es un hijo.** Esta sección del código de la aplicación, recibirá los datos del vector que posteriormente tratará en un algoritmo común. Una posible implementación de éste código es la reflejada en el **Algoritmo 5**.

Algoritmo 4 Envío de subvector del proceso padre a los procesos hijos.

```
asignacionHijos=tamanyo/totalProcesos;
asignacionFinal=auxPosicion=asignacionHijos+tamanyo%totalProcesos;

for (i=1;i<totalProcesos;i++) {
    MPI_Send( &vector[auxPosicion],asignacionHijos-1,
            MPI_LONG, i, 23, MPI_COMM_WORLD);
    auxPosicion+=asignacionHijos;
}
```

Algoritmo 5 Recepción del subvector por parte de los hijos

```
asignacionFinal=tamanyo/totalProcesos;
vector=malloc(sizeof(int)*asignacionFinal);
MPI_Recv( vector, asignacionFinal,
        MPI_LONG, MPI_ANY_SOURCE, 23, MPI_COMM_WORLD, &estado);
```

2.2.5. Unión de datos Para que la aplicación tenga sentido, han de reagruparse los datos que ha calculado por separado cada máquina. Para ello primero se ha de poner en marcha el algoritmo principal del programa, que es el del método **buscarNumero.** Una vez realizado, se pasa a la unión de estos resultados. En el **Algoritmo 6** se puede ver una posible implementación de esta especificación.

Algoritmo 6 Envío y recepción de datos para emisión de informe final.

```
    if(idProceso==0){
        for (i=1;i<totalProcesos;i++) {
            MPI_Recv ( &numVecesRecibida, 1,
                    MPI_LONG, MPI_ANY_SOURCE, 23, MPI_COMM_WORLD, &estado);
            numVeces += numVecesRecibida;
        }
        //Emitir informe
    }else{
        MPI_Send( &numVeces, 1,
            MPI_LONG, 0, 23, MPI_COMM_WORLD);
    }
```

3. Resultados

Todos los análisis realizados se realizan sobre un vector, de distinto tamaño, 10000 veces. Esto *simula* un tamaño del vector 10000 veces mayor sin ocupar memoria. Los resultados se reflejan en las tablas siguientes:

Tamaño del vector: 1000000

N Procesos	t. Secuencial (segs.)	t. Paralelo (segs.)	Speed-Up	Eficiencia
2	20,5326	10,4817	1,9588	97,94 %
4	20,4881	5,3797	3,8084	95,21 %
6	20,4005	3,6481	5,5921	93,20 %
8	20,3373	2,827	7,1940	89,92 %

Tamaño del vector: 2000000

N Procesos	t. Secuencial (segs.)	t. Paralelo (segs.)	Speed-Up	Eficiencia
2	41,0474	20,8957	1,9644	98,22 %
4	41,0174	10,75	3,8156	95,39 %
6	40,9835	7,2935	5,6192	93,65 %
8	40,9817	5,6567	7,2448	90,56 %

Tamaño del vector: 3000000

N Procesos	t. Secuencial (segs.)	t. Paralelo (segs.)	Speed-Up	Eficiencia
2	61,3097	31,2572	1,9615	98,07 %
4	61,4389	16,0409	3,8301	95,75 %
6	61,2304	10,9577	5,5879	93,13 %
8	61,3874	8,5051	7,2177	90,22 %

Tamaño del vector: 4000000

N Procesos	t. Secuencial (segs.)	t. Paralelo (segs.)	Speed-Up	Eficiencia
2	81,8595	41,6783	1,9641	98,20 %
4	82,0672	21,4127	3,8326	95,82 %
6	81,9771	14,6375	5,6005	93,34 %
8	81,5199	11,2938	7,2181	90,23 %

Atendiendo a las tablas, se generan los gráficos que se pueden ver en el **Anexo I** que acompaña este documento.

4. Conclusión

Para concluir, el estudio de una aplicación de forma secuencial comparada con la misma aplicación de forma paralela, nos permite tener una visión **más global del problema** y nos permite mecanismos de abstracción para descomponer el problema y que pueda ser ejecutado por un conjunto de máquinas diferentes.

Tanto si la paralelización es mas eficiente, como si no lo es, la descomposición del problema supone una **capa adicional** a la hora de diseñar un algoritmo y permitir por tanto posible mejoras sobre la aplicación inicial. En el caso de la práctica aquí expuesta, la mejora es visible desde los primeros resultados, con lo que se puede concluir que el algoritmo es eficiente y supone una mejora respecto al algoritmo ejecutado de forma secuencial.

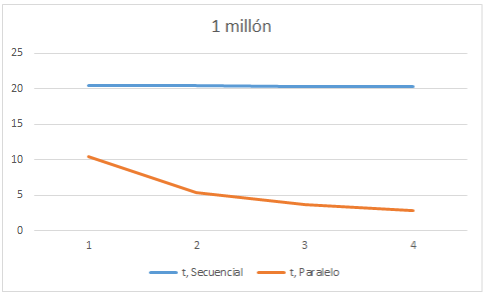
En caso de ser necesario, estas **herramientas** y **protocolos** (MPI y sus rutinas) tienen una potencia extraordinaria para **reducir en tiempo y en costes** las posibles soluciones que un problema de gran envergadura de cálculo requiera.

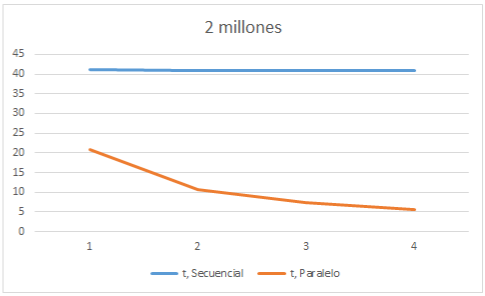
Además, dependiendo tanto del **speed-up** como de la **eficiencia**, podemos desarrollar un modelo de trabajo en el que prime el tiempo de cálculo si nos centramos en el primero, o en el que prime la eficiencia y el coste si nos centramos en el segundo factor.

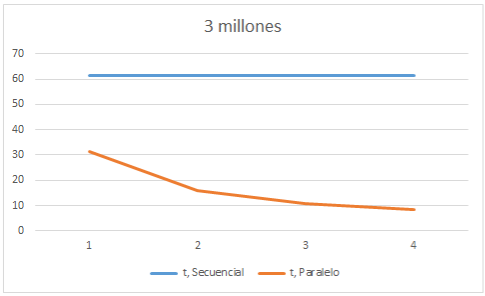
Finalmente como valoración personal, me gustaría añadir que conocía MPI a un nivel muy bajo y, aunque por el momento no se ha profundizado lo suficiente, me ha parecido fascinante el poder de cálculo y las posibilidades de cara al futuro que puede tener este tipo de mecasnismos. No dudaré en usarlo en posibles implementaciones si se me presenta la oportunidad.

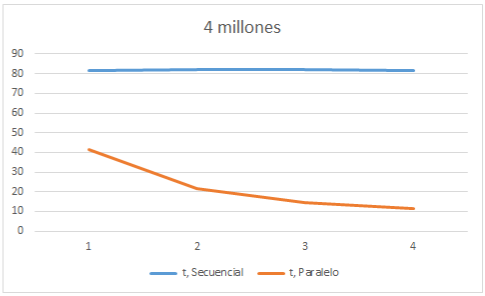
Anexo I: Tabla

Tablas con los tiempos tanto **paralelo** como **secuencial** con diferentes cargas de vector:

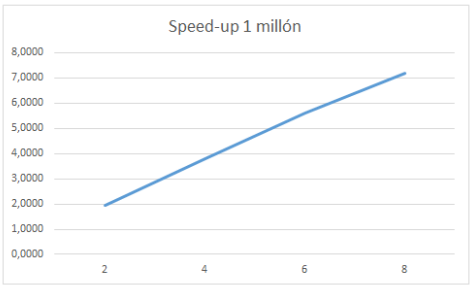


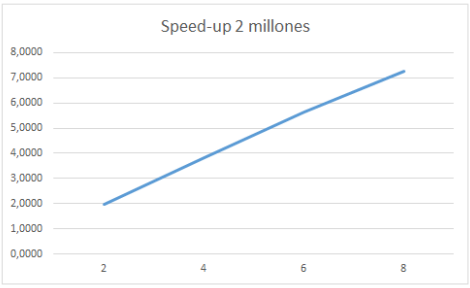


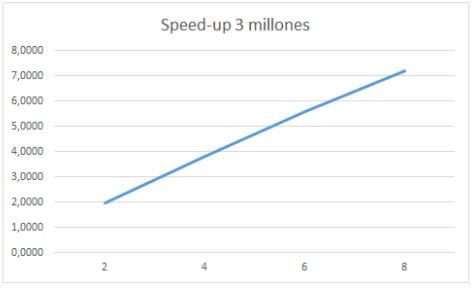


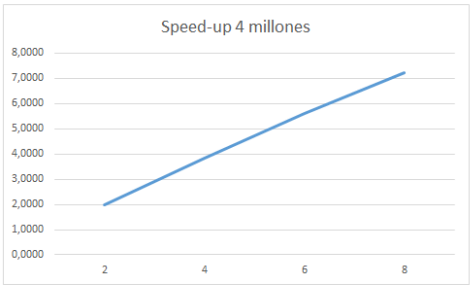


Tablas con los **speed-up** con diferentes cargas de vector:









Tablas con las estimaciones de **eficiencia** con diferentes cargas de vector: