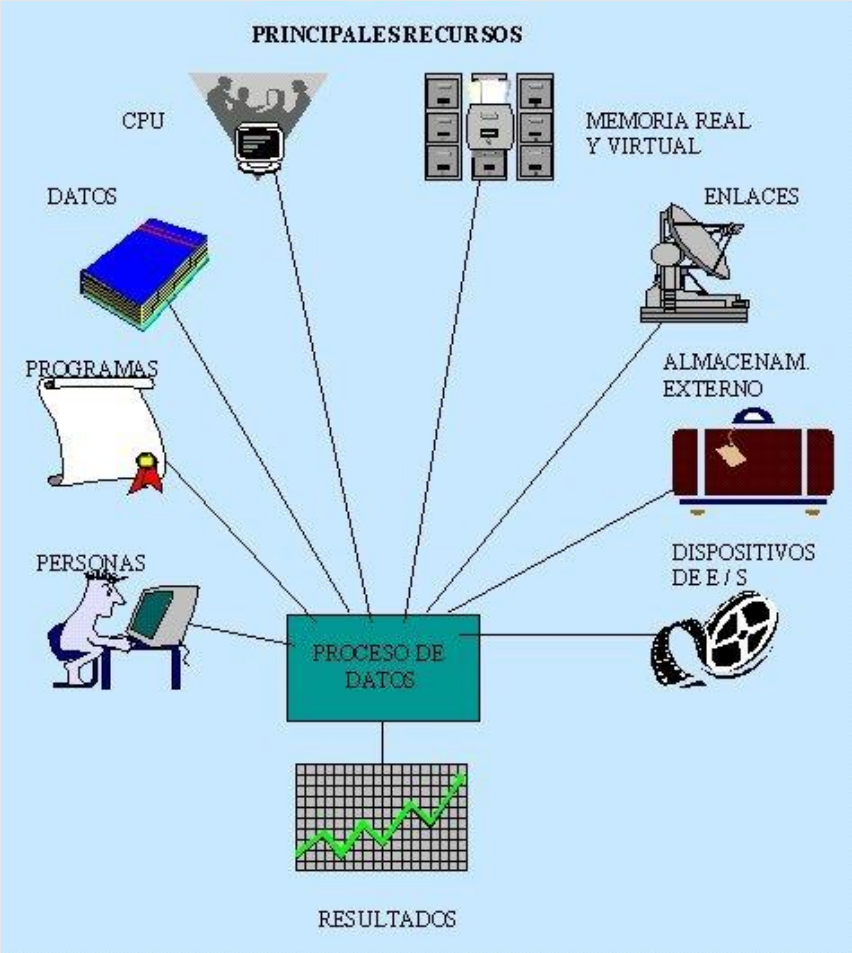
PRÁCTICA 4 – PARALELISMO EN SISTEMAS DE MEMORIA DISTRIBUIDA

INGENIERIA COMPUTADORES



Eduardo Correal Botero – Jose Miguel Gómez Lozano

Jordi Ferrández Gallego

ÍNDICE

INTRODUCCIÓN

¿QUÉ ES MPI?

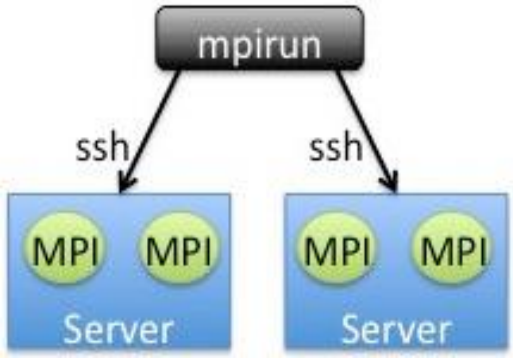
MPI significa Interfaz de Paso de Mensajes, el cual es un estándar que define la sintaxis y semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores. Es una técnica empleada en programación concurrente para aportar sincronización entre procesos y permitir la exclusión mutua. Su principal característica es que no precisa de memoria compartida. Nosotros utilizamos OpenMPI, que es una implementación open source de MPI.

¿QUÉ ES SSH?

SSH es un protocolo de administración remota que permite a un usuario controlar y modificar sus servidores remotos a través de internet. Es un protocolo seguro al utilizar técnicas criptográficas para garantizar que todas las comunicaciones hacia y desde el servidor remoto sucedan de manera encriptada.

¿PARA QUE SIRVEN MPI Y SSH?

OPENMPI consigue que nuestro programa tenga un rendimiento alto, tener todos los estándares de MPI-3.1, seguridad en los procesos y concurrencia, generación de procesos dinámicos, tolerancia a los fallos de red y procesos, etc.



SSH proporciona mecanismos para autenticar un usuario remoto, transferir entradas desde el cliente al servidor y retransmitir la salida de vuelta al cliente.

OBJETIVO

El objetivo de esta práctica es aprender a paralelizar una aplicación en un sistema de memoria distribuida utilizando técnicas de paso de mensajes. Para este objetivo usamos la API openMPI junto a SSH para conectar una serie de ordenadores en los cuales se ejecutarán tareas de forma paralela pasadas por SSH en sus procesadores.

NUESTRO CÓDIGO

En nuestro código hemos realizado la paralelización en las funciones *for*, las cuales se encargarán de una parte proporcional, siendo N el número de iteraciones, y P el número de procesadores, cada procesador hará N/P iteraciones, repartiendo de la misma forma la carga entre cada uno de los procesadores con *para_range*. A continuación podemos observar lo más significante del paralelismo implementado:

home ▸ josemygel ▸ Descargas ▸ Telegram Desktop ▸ PRACTICA MPI ▸ recta_regresion_mpi.cpp ▸ ...

```
12  //Se encarga de repartir las porciones de trabajo entre procesadores
13  void para_range(int n1, int n2, int &nprocs, int &irank, int &ista, int &iend) {
14   int iwork1 = (n2 - n1 + 1) / nprocs;
15   int iwork2 = ((n2 - n1 + 1) % nprocs);
16   ista = irank * iwork1 + n1 + min(irank, iwork2);
17   iend = iwork2 > irank ? ista + iwork1 : ista + iwork1 - 1;
18  }
19
20
21  float inline myFor(float *a1, float *a2, int n) {···
29  }
30
31  float inline Media(float array[], int n) {···
39  }
40
41
42  float inline myForPar(float *a1, float *a2, int n) {
43    float resultado = 0;
44    float resultadoGlobal = 0;
45    int i, ista, iend;
46
47    para_range(1,n,p,rango,ista,iend);
48    //Cada procesador calculará en resultado su porción de iteraciones
49    for (i = ista; i <= iend; i++) {
50      resultado += a1[i] * a2[i];
51    }
52
53    //Con MPI_Reduce sumamos resultado de cada rango en resultadoGlobal
54    MPI_Reduce(&resultado,&resultadoGlobal,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD);
55
56    return resultadoGlobal;
57  }
58
59  //MediaPar tiene repartido el trabajo igual que resultadoGlobal
60  float inline MediaPar(float array[], int n) {
61    float resultado = 0;
62    float resultadoGlobal = 0;
63    int i, ista, iend;
64
65    para_range(1,n,p,rango,ista,iend);
66    //Cada procesador calculará en resultado su porción de iteraciones
```

GIUA DE DESPLIEGE

Los requisitos para el despliegue son:

- MPI
- Código
- Archivo con las ips de hosts a las que mandar las instrucciones.
- Conexión entre las máquinas

Una máquina funcionará de *Master (Maestro)* y otra/s de *Slaves (Esclavos)*, repartiendo el trabajo a los procesadores (primero los procesadores del Master, y posteriormente los Slaves).

El despliegue se hará siguiendo los pasos siguientes:

1. Generar la clave con ssh-keygen-rsa en la maquina maestra.
2. Pasar la clave publica /.ssh/id_rsa.pub a todas las maquinas esclavas por pendrive
3. Cambiar el nombre de id_rsa.pub a authorized_keys y copiarlo a .ssh
4. Tener en todas las máquinas el programa recta_rpi
5. Compilar con el makefile en todas las maquinas en la misma carpeta (comando usado ***mpic++ exec recta_rpi.cpp***)
6. Crear el archivo hosts que contienen el nombre / ip de todas las máquinas.
7. Ejecutar en la máquina maestra con: ***mpirun -mca plm_rsh_no_tree_spawn 1 -hostfile [ArchivoConHosts] -n [NumeroDeProcesos] ./[EjecutableCompilado]***

Con esto podremos ejecutar en paralelo el programa ya compilado.

Es necesario compilar los programas en cada ordenador con cada cambio, por lo que hemos hecho para facilitar la tarea el siguiente *Makefile*:

```
EXECS=rectampi
MPICC?=mpic++ -std=c++11

rectampi: recta_regresion_mpi.cpp
    ${MPICC} -o rectampi recta_regresion_mpi.cpp

clean:
    rm ${EXECS}
```

ANÁLISIS DE RENDIMIENTO

NUESTROS COMPUTADORES

A continuación podremos observar las especificaciones de los computadores en los cuales vamos a ejecutar nuestro algoritmo y podremos comparar los resultados para obtener una comprensión perfecta de los costes computacionales que puede llevar a cabo.

Este ordenador que hemos utilizado, tenia 8 GB de memoria RAM y un Disco Duro HDD, es el ordenador del aula L12 de la clase de prácticas.

NUESTRAS PRUEBAS

Después de realizar distintas pruebas con distintos valores, hemos encontrado los mejores para nuestro algoritmo, lo cuales ocupaban hasta 7 GB de 8 GB disponibles de RAM que tenemos en los ordenadores del laboratorio. Recordamos que dichos valores corresponde a el tamaño en X e Y de los puntos generados para nuestra recta de regresión. Siendo MAX el número 1459938304 (máximo INT permitido), las pruebas que pasaremos son las siguientes:

```
// TOPE CON LA RAM PARA 8 GB (llega a 3.4 GB) 2x3.4 => 7GB
//n = MAX * 0.2;

 // TOPE CON LA RAM PARA 8 GB (llega a 2.95 GB) 2x2.95=> 5.9GB
 //n = MAX * 0.15;

// TOPE CON LA RAM PARA 8 GB (llega a 2.4 GB) 2x2.4 => 4.8GB
//n = MAX * 0.10;

// TOPE CON LA RAM PARA 8 GB (llega a 1.9 GB) 2x1.9 => 3.8GB
n = MAX * 0.05;
```

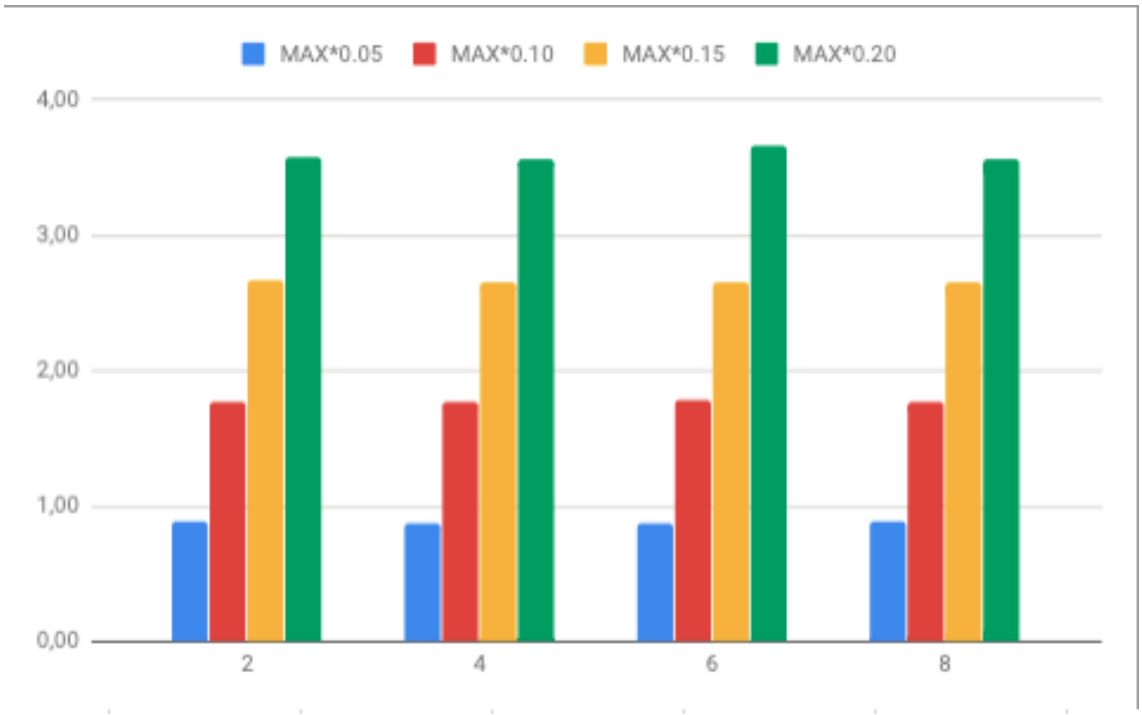
Además de utilizar dichos valores, hemos ido cambiando los procesadores lógicos que hemos utilizado, es decir, primero utilizábamos 2 procesadores en 1 computador, luego 4 en 2 computadores (2 en cada uno), luego 6 en 3 computadores (2 en cada uno) y finalmente 8 procesadores en 4 computadoras distintas.

RESULTADOS TIEMPOS SECUENCIALES

Los tiempos obtenidos han con los valores anteriores y variando los procesadores entre los computadores de forma secuencial, han sido los siguientes:

En todas nuestras pruebas, el Eje Y corresponde al tiempo, el Eje X corresponde a los procesadores utilizados, y los colores, a los valores de nuestras pruebas.

	2	4	6	8
MAX*0.05	0,89	0,88	0,88	0,89
MAX*0.10	1,77	1,77	1,78	1,77
MAX*0.15	2,67	2,66	2,65	2,66
MAX*0.20	3,58	3,57	3,66	3,56

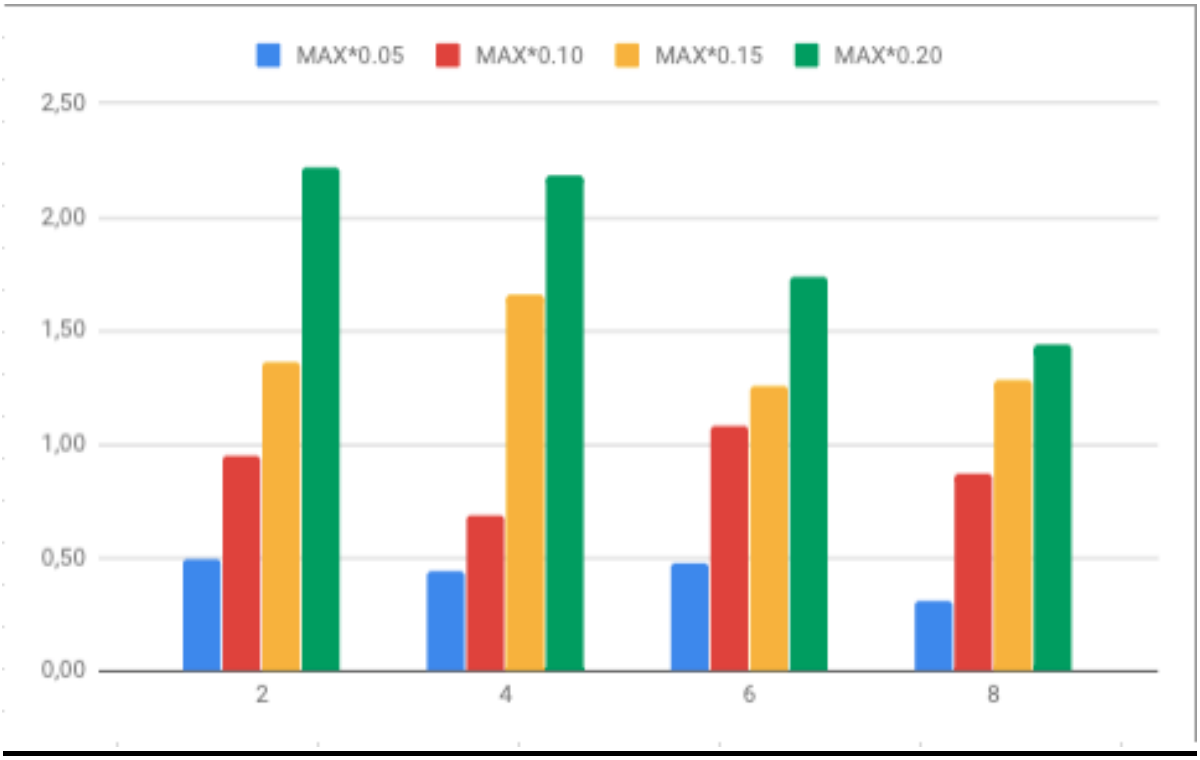


Como observamos en la gráfica, los tiempos si que son notablemente bajos, incluso si lo realizamos de forma secuencial, pero al ser secuencial, solo se realiza en el primer ordenador, es decir el tiempo se mantiene estable, ya que aunque pongan 8 procesadores, únicamente está utilizando 2, los únicos 2 del primer ordenador. En el siguiente apartado podremos comprobar como dichos tiempos mejoran de forma notoria.

RESULTADOS TIEMPOS PARALELO

Los tiempos obtenidos han con los valores anteriores y variando los procesadores entre los computadores de forma paralela, han sido los siguientes:

	2	4	6	8
MAX*0.05	0,49	0,44	0,48	0,31
MAX*0.10	0,95	0,69	1,08	0,87
MAX*0.15	1,36	1,66	1,26	1,28
MAX*0.20	2,22	2,18	1,74	1,44



Una vez comparado los tiempos obtenidos, nos damos cuenta de que la diferencia es bastante notoria entre los tiempos en paralelo y los tiempos en secuencial, siendo en algunos casos de hasta 2 segundos menos! Incluso, observamos como a medida que añadimos más procesadores, los valores de tiempo se hacen cada vez más bajos, esto significa que estamos haciendo el trabajo correctamente y la mejora temporal es más que evidente.

Evidentemente es muy importante que nuestros procesadores estén lo suficientemente capacitados para realizar estas tareas, ya que estas funciones en grandes empresas o corporaciones pueden ahorrar un grandioso tiempo que, para ellas, puede ser muy útil.

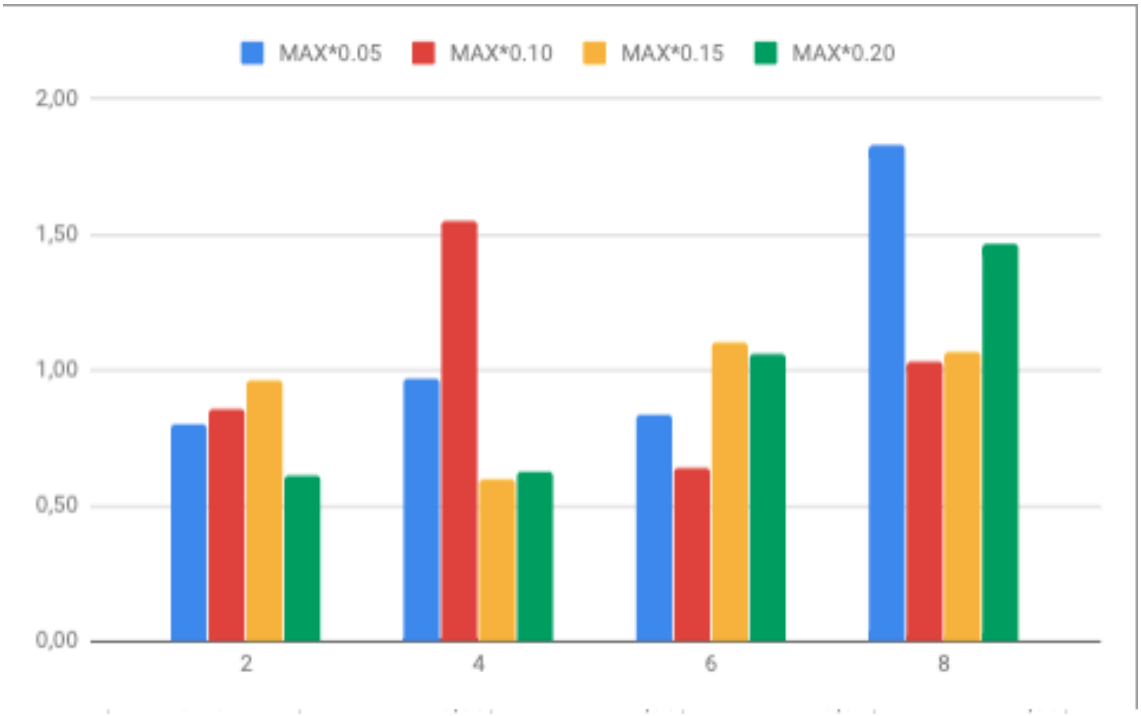
Hay que aclarar que algunos tiempos son un poco más grandes, cuando tendrían que ser más bajos, esto es debido a la congestión de la red del aula, o incluso algunos programas externos abiertos en alguna computadora que pueden llegar a consumir algo de RAM, tal y como nos explicó el profesor en clase, pero aún así, estamos bastante sorprendidos con la mejora de tiempos.

TIEMPOS CON GANANCIA Y EFICIENCIA

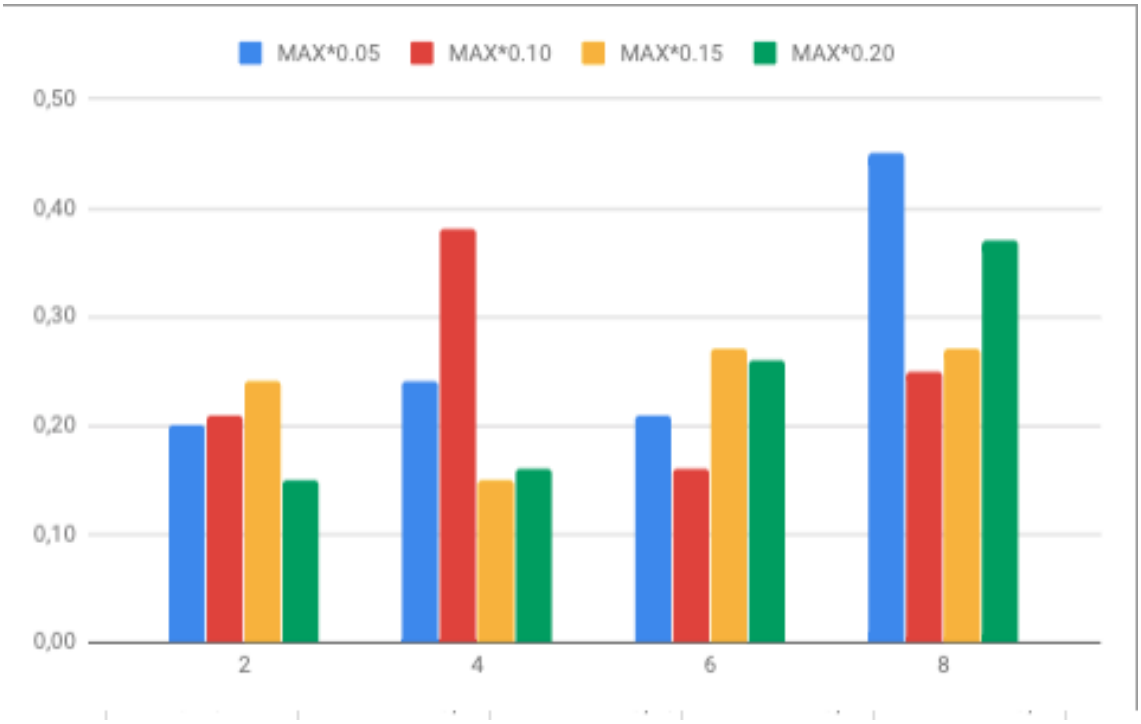
A continuación, veremos las gráficas de la ganancia y la eficiencia, entre los tiempo en secuencial y los obtenidos en paralelo.

La primera gráfica correspondera a la ganancia y la segunda gráfica a la eficiéncia.

	2	4	6	8
MAX*0.05	0,80	0,97	0,84	1,83
MAX*0.10	0,86	1,55	0,64	1,03
MAX*0.15	0,96	0,60	1,10	1,07
MAX*0.20	0,61	0,63	1,06	1,47



	2	4	6	8
MAX*0.05	0,20	0,24	0,21	0,45
MAX*0.10	0,21	0,38	0,16	0,25
MAX*0.15	0,24	0,15	0,27	0,27
MAX*0.20	0,15	0,16	0,26	0,37



Como podemos observar las dos gráficas se parecen bastante, solo difieren del valor, que podemos observar en la izquierda (Eje Y).

Observamos que la ganancia es casi 2 veces mejor en algunos casos el paralelo que el secuencial, cosa que nos sorprendió bastante, ya que es una mejora temporal bastante notable.

En cuanto a la eficiencia, observamos que se dispara bastante cuando utilizamos 8 procesadores (2 en cada uno utilizando 4 computadoras), esto se debe a que gracias a dividir las operaciones realizadas en nuestro algoritmo, el tiempo de ganancia y eficiencia son mucho mejores.

CONCLUSIÓN

Como conclusión podemos comentar la importancia de esta práctica en cualquier ámbito, tanto empresarial como individual, ya que al paralelizar y dividir entre procesadores las operaciones de cualquier algoritmo, la mejora que obtenemos es bastante grande.

Nosotros hemos obtenido un x2 de mejora, con un algoritmo bastante simple, en una empresa donde se utilicen millones de datos con supercomputadoras estos procesos que hemos seguido en esta práctica se hacen muy necesarios para poder obtener rendimientos excepcionales.

También decir que nos ha sorprendido bastante los resultados, y como a medida que aumentábamos los procesadores el tiempo mejoraba, y de forma bastante notable.

REFERENCIAS

http://users.dsic.upv.es/~jlinares/eda/complejidad%20computacional.pdf
https://www.hostinger.es/tutoriales/que-es-ssh#gref
https://www.open-mpi.org/
http://www.cenits.es/faq/preguntas-generales/que-es-mpi