UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Estudio comparativo de las prestaciones de diferentes modelos de programación paralela

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Enrique Gil Arcas

Tutor: Federico Silla Jiménez

2014-15

Resumen

Hoy día disponemos de múltiples soluciones para el procesamiento de grandes volúmenes de datos, pero no es trivial seleccionar la más adecuada para cada tipo de situación. Debemos tener en cuenta, en cada caso, el tipo de hardware disponible, la cantidad de información que queremos procesar y la tecnología software aplicable entre otros muchos aspectos.

En este trabajo hemos profundizado en distintos paradigmas de programación tanto paralela como distribuida. Basándonos en un sencillo algoritmo de multiplicación de matrices, hemos realizado diversas implementaciones a partir de distintas soluciones software (MPI, OpenMP, Pthreads, CUDA, etc) y las hemos probado sobre diversos tipos de hardware (procesadores, tarjetas gráficas y coprocesadores vectoriales), realizando un análisis de los resultados y extrayendo conclusiones.

Palabras clave: MPI, OpenMP, Pthreads, CUDA, AVX, cluster, compartida, distribuida, paralela, coprocesador, GPU, InfiniBand.

Abstract

Nowadays, we have a lot of solutions to process big amounts of data, but it is not easy to select the most adequate on each kind of situation. We should take care of: The different available hardwares, the amount of data to process and the applicable software technology

In this study we went in depth with different paradigms for parallel and distributed programing. We have taken a simple matrix multiplication algorithm as base to develop various implementations using different software solutions (MPI, OpenMP, Pthreads, CUDA, etc) and we tested them over different types of hardware (processors, coprocessors and graphics cards) making an analysis of the results and drawing conclusions.

Tabla de contenidos

1. Introducción.

Los fabricantes de microprocesadores, como Intel o AMD, han desarrollado a lo largo de los años diseños cada vez más elaborados con la finalidad de aumentar las prestaciones de los computadores. En este sentido, cuando debido a problemas de disipación de calor ya no les fue posible continuar aumentando la frecuencia de reloj a la que trabajaban sus productos, se lanzaron a la búsqueda de nuevas formas de incrementar la potencia computacional de sus desarrollos, creando finalmente lo que hoy en día es algo habitual para todos: los procesadores multinúcleo (*multicore*).

Los procesadores multinúcleo se basan en incluir varios núcleos de computación dentro del mismo encapsulado, cada uno de ellos trabajando típicamente a frecuencias de reloj algo menores que las que se utilizarían en un chip con un único núcleo. Sin embargo, a pesar de las menores frecuencias de reloj utilizadas, las prestaciones globales del procesador multinúcleo son notablemente mayores que las de los diseños anteriores basados en un único núcleo siempre y cuando la aplicación que se ejecute en dicho procesador sea capaz de repartir sus tareas entre los diferentes núcleos del procesador, momento en el cual se puede llegar a conseguir importantes reducciones en el tiempo de ejecución.

Para conseguir que una aplicación mantenga los diferentes núcleos del procesador en uso, se suelen usar mecanismos de programación de memoria compartida basados en hilos (*threads*). Mediante este mecanismo el programador divide la computación a realizar en diferentes hilos de ejecución y asigna cada uno de estos hilos a un núcleo diferente del procesador, de forma que todos avancen en su ejecución de forma concurrente. Obviamente, hace falta mantener los diferentes hilos sincronizados, hecho que acaba complicando la programación de la aplicación. Por esta razón, se han creado entornos de programación como OpenMP, que facilitan la programación con hilos, aunque en ocasiones puede resultar en una ligera pérdida de prestaciones dado que algunas de las decisiones de programación las toma el compilador.

En cualquier caso, hay numerosos problemas de gran envergadura que incluso con el uso de los múltiples núcleos de un procesador (o con los núcleos de los diversos procesadores existentes en un computador dado) aún presentan tiempos de ejecución demasiado elevados. Por ello, y con la finalidad de reducir estos elevados tiempos de ejecución, se hace uso de mecanismos de paso de mensajes, como la librería MPI (*Message Passing Interface*), que permiten distribuir la computación a realizar entre los diferentes nodos de un clúster, usando un modelo de memoria distribuida no compartida. Nótese que en cada computador involucrado en

la ejecución de la aplicación distribuida aún se puede seguir utilizando la programación basada en hilos.

Finalmente, en los últimos años se ha dado un gran salto en la reducción del tiempo de ejecución de numerosas aplicaciones a base de usar aceleradores de diferente índole. Quizás el tipo de acelerador más utilizado sea la GPU (*Graphics Processing Units*), que no es otra cosa que una tarjeta gráfica de las comúnmente utilizadas en los ordenadores de sobremesa para dotarles de capacidades gráficas para mostrar ventanas y también para los videojuegos, pero adaptadas para su uso en computación. Cabe destacar que el uso de las GPUs es compatible con la programación basada en hilos y con la programación basada en librerías de paso de mensajes, pudiendo combinar las tres técnicas para lograr mayores reducciones del tiempo de ejecución.

En este trabajo se pretende profundizar en la computación paralela y distribuida. A lo largo de la carrera se han revisado de forma breve los conceptos más sencillos de estas técnicas. No obstante, en este trabajo se amplían de forma notable los conocimientos sobre estas técnicas especialmente a nivel práctico. Para ello se utilizan diversos paradigmas de programación y combinándolos entre sí, realizando además un estudio comparativo de los rendimientos que se obtienen en cada caso. Para ello, se parte de un algoritmo sencillo como es la multiplicación de matrices, que resulta muy útil para cargar de trabajo un clúster de computadores y se modifica para adaptarlo a los modelos de programación mencionados, de manera que el alumno realizará diversas implementaciones de dicho algoritmo usando distintas herramientas tanto software como hardware. A nivel software se utilizarán las tecnologías OpenMP, MPI, Pthreads, funciones intrínsecas para vectorización, autovectorización y CUDA. Éstas se utilizarán combinadas entre sí, usando siempre el lenguaje de programación C como nexo. Respecto a la parte hardware, se desarrollarán implementaciones para su ejecución en procesadores multinúcleo, en coprocesadores vectoriales y en GPUs.

El segundo objetivo de este trabajo ha sido probar dichas implementaciones en un entorno real en funcionamiento, para ello nos hemos servido del *cluster* que gestiona el Grupo de Arquitecturas Paralelas del departamento DISCA de la ETSINF en la Universidad Politécnica de Valencia. Cuando se trabaja con un cluster hay que tener en cuenta sus características propias como el tipo de hardware del que dispone, por ejemplo: la marca y modelo de los procesadores, si hay o no tarjetas gráficas y de qué tipo, la cantidad de memoria RAM por nodo, el tipo de red de interconexión entre nodos, si dispone de una interfaz de disco duro compartida, etc. Igual de importante es conocer el tipo de software instalado, en nuestro caso es necesario conocer la implementación disponible de MPI, ya que entre unas y otras pueden haber sutiles diferencias, como por ejemplo, el tamaño máximo de mensaje, que para nosotros es un factor determinante en nuestra tarea. También nos es vital conocer qué versión de

CUDA podemos utilizar, ya que la forma de programar en las nuevas versiones ha cambiado notablemente. Por todo ello, en adición a los dos objetivos principales, también se ha desarrollado una guía rápida de usuario que se anexa al final del trabajo, en la que se tratan los aspectos más importantes de configuración que debe tener en cuenta un usuario del citado cluster a la hora de ejecutar sus algoritmos en él. Se pretende que esta guía pueda ser utilizada por usuarios noveles que se encuentren con problemas que hemos tenido que resolver y que en muchas ocasiones estaban muy poco documentados.

2. Entornos de computación paralela y distribuida.

2.1 Computación paralela.

La computación paralela consiste en repartir la carga de trabajo de un programa sobre los múltiples *cores* de que dispone el computador con el fin de reducir todo lo posible el tiempo de ejecución del programa. Los programas informáticos paralelos son más difíciles de escribir que los secuenciales, porque la concurrencia introduce nuevos tipos de errores de software, siendo las condiciones de carrera los más comunes. Para que un programa secuencial se ejecute de forma paralela se requiere del rediseño de los algoritmos y posiblemente el replanteamiento de las estructuras de datos. Además, la comunicación y sincronización entre diferentes sub-tareas son algunos de los mayores obstáculos para obtener un buen rendimiento del programa paralelo.

2.2 Computación distribuida.

Podemos entender la computación distribuida como un caso especial de la computación paralela en el que tenemos trabajando en común un conjunto de procesadores que residen en diferentes computadores, los cuales están conectados por una red de comunicaciones. Esta red puede ser tan pequeña como la red local de un *cluster* o tan grande como Internet, permitiendo trabajar en común a computadoras que están en una misma sala o computadoras que están en diferentes continentes.

2.3 Ley de Amdahl.

A priori podríamos pensar que si un programa que se ejecuta sobre un procesador tarda, por ejemplo, 20 segundos en terminar su ejecución, al ejecutar ese mismo programa sobre dos procesadores el tiempo de ejecución sería de 10 segundos, pero en realidad no es así. Los programas para entornos paralelos poseen una parte que es susceptible de ser ejecutada en paralelo y otra que siempre lo hará de forma secuencial por lo que únicamente se podrá reducir el tiempo de ejecución de la parte paralelizable. Este comportamiento queda recogido en la ley de Amdahl que dice que la mejora obtenida en el rendimiento de un sistema debido a la

alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente.

La fórmula original de la ley de Amdahl es:

$$F = F_a \cdot \left((1 - F_m) + \frac{F_m}{A_m} \right)$$

Fa = tiempo de ejecución antiguo.

Fm = tiempo de ejecución mejorado.

Am = es el factor de mejora introducido por el subsistema mejorado.

Esta fórmula se puede reescribir usando la definición del incremento de la velocidad que viene dado por $A = F_a / F$, por lo que la fórmula anterior se puede reescribir como:

$$A = \frac{1}{(1 - F_m) + \frac{F_m}{A_m}}$$

A = es la aceleración o ganancia en velocidad conseguida en el sistema completo debido a la mejora de uno de sus subsistemas.

Fm = es la fracción de tiempo que el sistema utiliza el subsistema mejorado.

Am = es el factor de mejora introducido por el subsistema mejorado.

Esta última formula nos muestra que cuando Fm/Am tiende a cero el tiempo de ejecución ya no es mejorable añadiendo más procesadores en paralelo.

2.4 Memoria compartida y memoria distribuida.

Los procesos disponen de dos entornos de comunicación en función del tipo de acceso a la memoria que hagan: memoria compartida y memoria distribuida. En entornos de memoria compartida se permite que todos los procesadores puedan acceder a toda la memoria, posibilitando la comunicación entre procesos y evitando copias redundantes de información. Un ejemplo de sistema de memoria compartida sería un computador compuesto por varios procesadores multinúcleo. En este sistema todos los núcleos comparten la memoria RAM del

sistema. Este formato ofrece una escalabilidad limitada ya que, cuantos más *cores* tengamos trabajando juntos, más fácil es que se puedan generar colas de espera cuando intenten acceder a la memoria al mismo tiempo. Por otra parte, en entornos de memoria distribuida los procesadores solo tienen acceso a una parte de la memoria y la forma de intercambiar información entre ellos es por paso de mensajes. Un ejemplo de sistema distribuido sería un *cluster* de computadores conectados por una determinada tecnología de red. En este sistema un *core* puede acceder únicamente a la memoria de su computador y no a la de los otros computadores del *cluster*. Nótese que hoy en día es típico que un sistema distribuido esté compuesto por computadores que son en sí mismos sistemas de memoria compartida. Esta configuración ofrece una escalabilidad mucho mayor con menor costo.

Para realizar la evaluación del cluster en un entorno de memoria compartida nos hemos servido de dos herramientas basadas en el manejo de hilos o *threads* que son las API (*Application Programming Interface*) Pthreads y OpenMP. Por otro lado, para realizar las pruebas en un entorno de memoria distribuida hemos utilizado MPI (Message Passing Interface).

2.5 Hilos o *threads.*

Para responder a ¿qué es un *thread*? Primero debemos entender qué es un proceso. Un proceso podemos definirlo como un programa en ejecución. Es creado por el sistema operativo y ocupa recursos del sistema. Un proceso contiene información sobre el estado del programa y sus recursos, por ejemplo: el ID del proceso, ID del usuario, ID del grupo, registros, descriptores de fichero, heap, stack, etc.

Un thread lo definimos como un flujo independiente de instrucciones que extraemos del proceso principal que puede ser ejecutado en paralelo con otros threads de ese mismo proceso. Para que un thread pueda ejecutarse de forma independiente manteniendo sus propios registros, *stack pointer*, etc necesita duplicar algunos recursos del proceso pero el resto se comparten. En lo relativo al uso de recursos, crear un thread ofrece una sobrecarga al sistema mucho menor que crear un proceso nuevo. En arquitecturas de memoria compartida multiprocesador, los threads pueden ser usados para implementar paralelismo. Los threads intercambian información leyendo y escribiendo sobre la misma región de memoria.

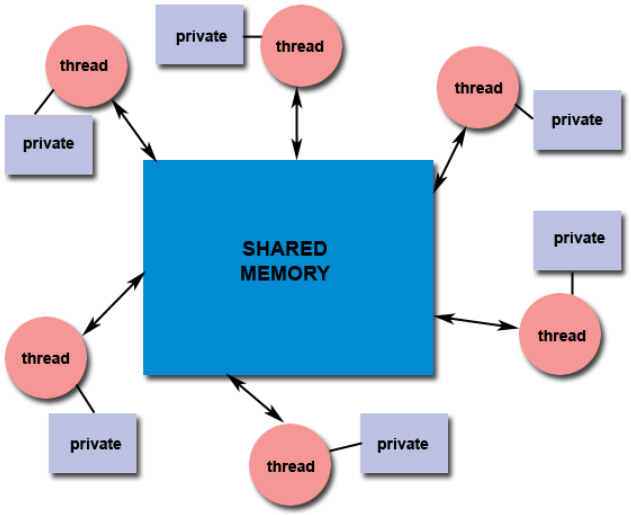

Ilustración 1: Modelo de threads con memoria compartida.

Históricamente, los fabricantes de hardware implementaban sus propias versiones propietarias de threads. Estas implementaciones diferían mucho unas de otras dificultando a los programadores el desarrollo de aplicaciones portables. En este contexto se hizo necesario el desarrollo de una interfaz homogénea que para los sistemas UNIX queda especificada por el estándar IEEE POSIX 1003.1c en 1995. Las implementaciones de este estándar son referenciadas como POSIX threads o Pthreads.

2.5.1 pthreads.h.

La librería *pthreads.h* nos ofrece una implementación de la interfaz Pthreads para el lenguaje de programación C que incluye un conjunto de tipos, funciones y constantes para el manejo de hilos.

2.5.2 OpenMP.

OpenMP es una especificación de un conjunto de directivas de compilador, rutinas y variables de entorno para el desarrollo de aplicaciones con un alto nivel de paralelismo en lenguajes como Fortran y C/C++. Utiliza principalmente una notación de directivas que se incluyen sobre el código secuencial y que utiliza el compilador para generar el código que se ejecutará en cada *thread* y gestionar el acceso de los mismos a las regiones de memoria. Esta interfaz es de mayor nivel respecto de la que nos ofrece POSIX por lo que es más fácil para el

programador desarrollar y mantener el código de programas paralelos. Por el contrario, dado que es el compilador quien toma las decisiones finales de paralelización, el control del programador es menor.

2.6 Message Passing Interface.

MPI es un protocolo de comunicaciones por paso de mensajes estándar y portable, independiente del lenguaje y usado para el desarrollo de software de ejecución en paralelo. Las ventajas que nos brinda son escalabilidad, portabilidad y un alto rendimiento. Se ha convertido en un estándar de *facto* para la comunicación entre procesos sobre arquitecturas de memoria distribuida como por ejemplo *clusters*. A partir de la versión MPI-2 se introduce con limitaciones el concepto de memoria compartida.

MPI se encarga de crear tantos procesos independientes en cada nodo como nosotros le indiquemos, los cuales trabajaran en común con un modelo de memoria distribuida, es decir, cada proceso mantiene regiones de memoria separadas y los intercambios de datos se hacen por paso de mensajes entre ellos. Este paso de mensajes debe ser programado de forma explícita, lo cual conlleva cierta dificultad añadida en numerosos casos. En nuestras implementaciones hemos utilizado una estructura de red de comunicaciones entre los nodos de tipo estrella, pero MPI nos permite trabajar con cualquier otra disposición (malla, toro, circular…). De hecho, MPI abstrae muchos de los detalles de la red al programador, simplificando así su tarea.

2.7 Hardware.

Para obtener un rendimiento óptimo en software para ejecución en paralelo es muy importante tener en cuenta a la hora de programar el hardware sobre el que se va a ejecutar el programa.

2.7.1 Red de interconexión.

Cuando utilizamos MPI sobre un conjunto de computadoras en red nos encontramos que la propia red es un factor determinante del rendimiento de nuestras aplicaciones y más aún cuanto mayor sea la cantidad de información que se ha de enviar.

2.7.1.1 Topología.

Un factor importante de la red es su topología. Cuando vamos a realizar el montaje de un *cluster* teóricamente nos gustaría tener una conexión física directa entre todos los pares de nodos del *cluster*. No obstante, si en dicho *cluster* tenemos un número elevado de equipos muy probablemente nos vamos a encontrar con limitaciones físicas y económicas. Por ello hay que recurrir a otro tipo de topologías como por ejemplo en anillo, en toro, en estrella,…

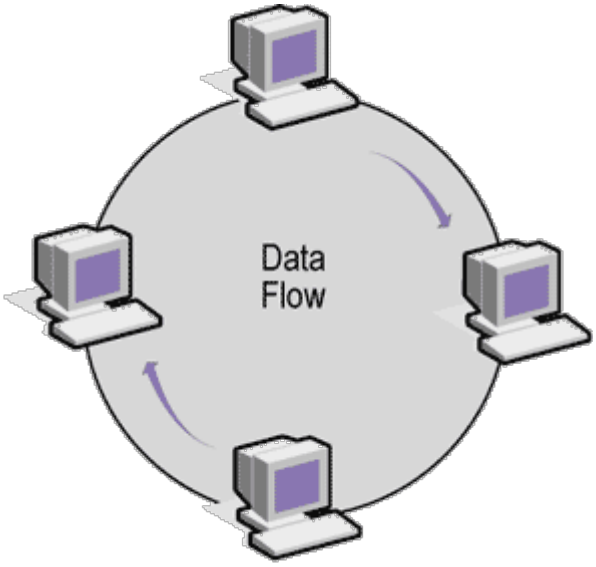

Ilustración 2: Topología anular.

Con una estructura en anillo cada terminal se conecta únicamente con otros dos miembros de la red. Esta topología tiene el inconveniente de que si queremos enviar un mensaje a un equipo que está en la parte más alejada del anillo el mensaje tendrá que pasar a través de todos los equipos que encuentre en su camino, incrementando por tanto la latencia del mismo.

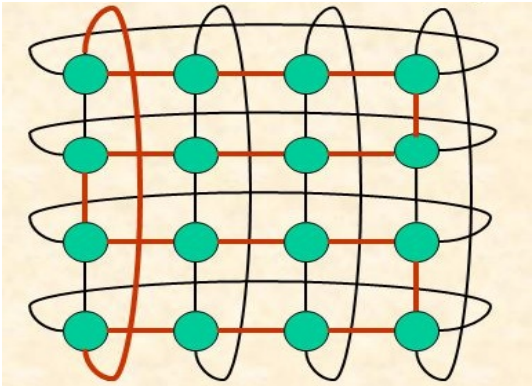

Ilustración 3: Topología toroide.

En la ilustración anterior vemos una disposición toroidal de dos dimensiones, cada equipo está conectado directamente con otros cuatro miembros de la red, de esta manera se reduce el número de saltos para comunicar los pares más alejados de la red. No obstante, el coste económico de una topología en toro es mucho más elevado que otras topologías más sencillas, como por ejemplo la topología en estrella.

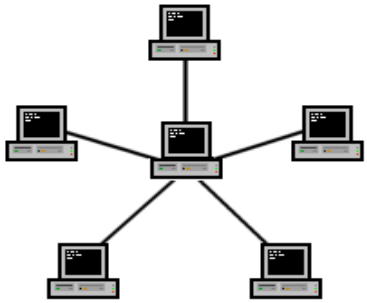

Ilustración 4: Topología en estrella.

La organización en forma de estrella reduce el número máximo de saltos para comunicar pares de nodos a dos. Como inconveniente tenemos que todo el tráfico pasa por el nodo central. Una optimización para este modelo es sustituir el nodo central por un *switch*.

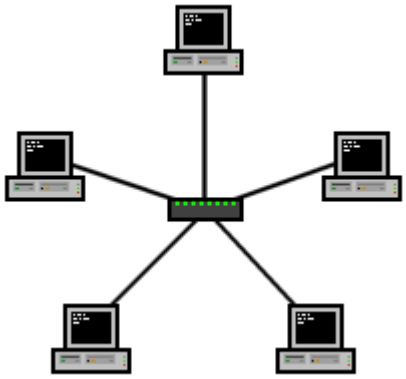

Ilustración 5: Topología en estrella con switch

En los *clusters* actuales, y debido a razones económicas de adquisición del equipamiento y de montaje y mantenimiento de la red, la topología más utilizada hoy en día es la estrella junto con otras derivadas, como pueden ser varias estrellas interconectadas. Este es el caso, por ejemplo, de redes Ethernet. Otras redes de mayores prestaciones, como InfiniBand, también pueden adaptar esta topología.

2.7.1.2 Enlaces.

Una de las principales características de la red es la tasa de bits que no es más que el número de bits que se transmiten por unidad de tiempo entre dos terminales a través de la propia

red. Cuanto mayor sea la tasa de bits, los mensajes tardarán menos tiempo en llegar y antes será posible procesarlos. El elemento hardware que va a determinar dicha tasa es el enlace, junto con las tarjetas de red (capas del nivel de enlace y físico de la pila de protocolos TCP/IP).

Típicamente se utiliza cable compuesto de pares trenzados de cobre (también se usa cable coaxial o fibra óptica) para montar redes Ethernet, en función de la categoría del cable vamos a poder trabajar con distintos estándares que nos ofrecen diferentes tasas de bits, las soluciones comerciales llegan hasta 1Gbit/s aunque se han desarrollado estándares para funcionar a 10Gbit/s.

Ilustración 6: Conector RJ-45 para cable de pares trenzados.

Sin embargo, el *cluster* sobre el que vamos a trabajar dispone de tecnología InfiniBand. InfiniBand está compuesto por enlaces serie bidireccionales que pueden añadirse en grupos de 4 o 12, llamados 4X o 12X. Estos enlaces pueden trabajar a distintos *data rates*: SDR (Single Data Rate), DDR (Double Data Rate), QDR (Quad Data Rate), FDR (Fourteen Data Rate), y EDR (Enhanced Data Rate). Actualmente se está trabajando en desarrollar HDR (High Data Rate) que podrá trabajar a 200 Gb/s en enlaces 4X.

El *cluster* de Arquitecturas Paralelas en el momento de realizar este Trabajo Final de Grado trabajaba con enlaces 4X y FDR ofreciendo 56 Gb/s en cada dirección del enlace. Recientemente se han actualizado estos enlaces para trabajar con EDR ofreciendo un ancho de banda de 100 Gb/s.

InfiniBand usa una topología conmutada de forma que varios dispositivos pueden compartir la red al mismo tiempo. Los datos se transmiten en tramas de hasta 4 kB que se agrupan para formar mensajes. Un mensaje puede ser una operación de acceso directo a memoria remota, de lectura o escritura sobre un nodo remoto, un envío o recepción por el canal, una operación de transacción reversible o una transmisión *multicast*.

Ilustración 7: Conector de enlace InfiniBand.

2.7.2 Coprocesador vectorial.

El coprocesador es un procesador suplementario al procesador principal. Realiza operaciones en coma flotante, aritméticas, gráficas, de procesamiento de señal, procesamiento de *strings*, encriptación, etc. El coprocesador por sí mismo no puede coger instrucciones de la memoria ni hacer operaciones de entrada/salida entre otras limitaciones sino que está subordinado a realizar las funciones que el procesador principal le solicite. Los procesadores actuales de Intel contienen varios coprocesadores, gracias a la gran cantidad de transistores que es posible integrar hoy en dia.

Cada nodo del *cluster* dispone de 12 coprocesadores vectoriales (uno por núcleo) los cuales son compatibles con la tecnología AVX que nos ofrece 16 registros de 256 bits que nos permiten alojar, por ejemplo, vectores de 4 *doubles* o de *floats*.

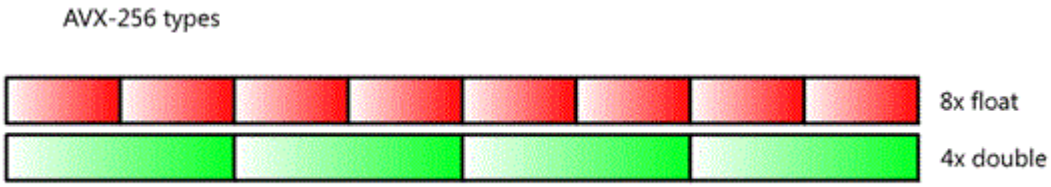

Ilustración 8: Ejemplo de registro AVX-256.

Los procesadores trabajan con un modelo SISD (*single instruction single data*) pero los coprocesadores vectoriales nos permiten ejecutar instrucciones SIMD (*single instruction multiple data*) pudiendo realizar operaciones sobre múltiples datos simultáneamente. De esta forma, por ejemplo, resulta sencillo aplicar una misma operación a los elementos de un vector.

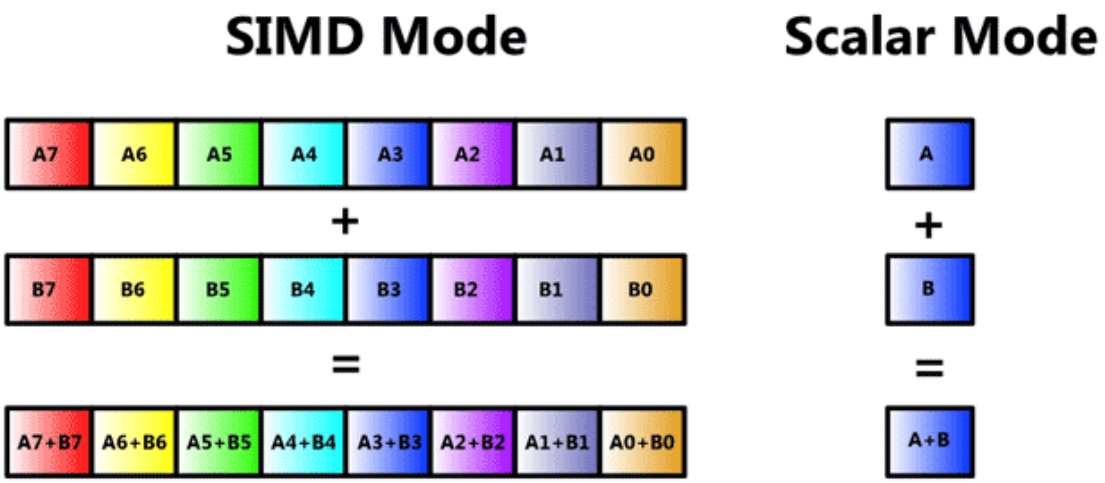

Ilustración 9: Esquema de aplicación de una operación suma, elemento a elemento, sobre dos vectores

2.7.3 GPU.

La unidad de procesamiento gráfico o GPU (Graphics Processing Unit) es un coprocesador dedicado al procesamiento de gráficos, para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos o aplicaciones 3D interactivas. La GPU implementa ciertas operaciones gráficas llamadas primitivas optimizadas para el procesamiento gráfico. Una de las primitivas más comunes para el procesamiento gráfico en 3D es el *antialiasing*, que suaviza los bordes de las figuras para darles un aspecto más realista. Adicionalmente existen primitivas para dibujar rectángulos, triángulos, círculos y arcos. Las GPU actualmente disponen de gran cantidad de primitivas, buscando mayor realismo en los efectos. De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, el procesador principal (CPU) puede dedicarse a otro tipo de cálculos.

Una de las mayores diferencias con la CPU estriba en su arquitectura. A diferencia del procesador central, que tiene una arquitectura de von Neumann, la GPU se basa en un modelo de multi-procesador. Este modelo facilita el procesamiento en paralelo y la gran segmentación que posee la GPU para sus tareas.

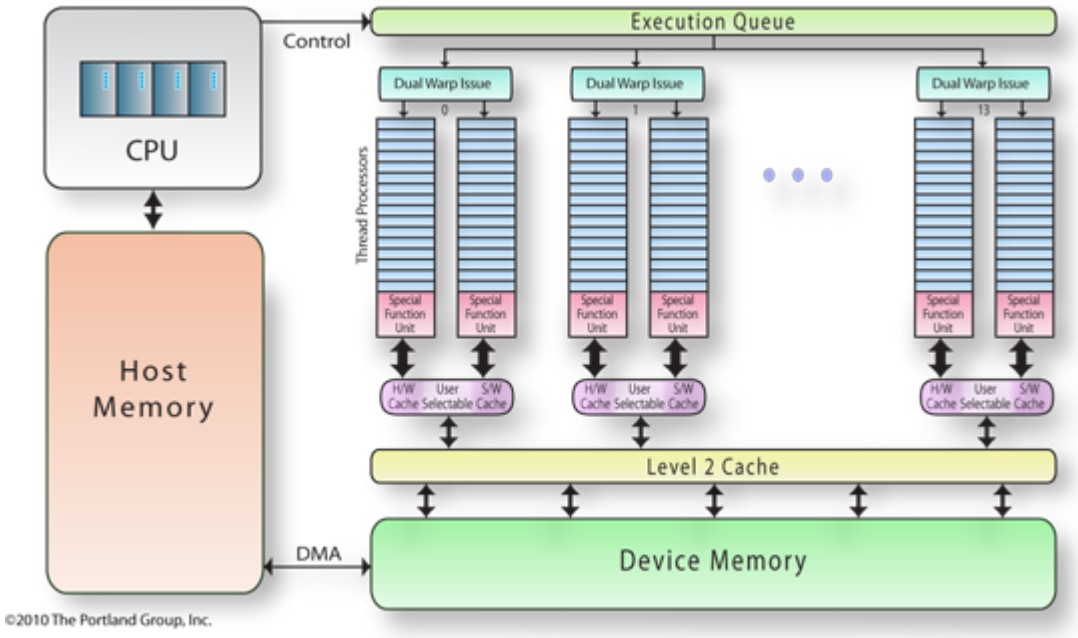

Ilustración 10: Arquitectura de una Nvidia Fermi-class GPU

En este sentido, se intenta aprovechar la gran potencia de cálculo de las GPU para aplicaciones no relacionadas con los gráficos, en lo que desde recientemente se viene a llamar GPGPU, o GPU de propósito general (General Purpose GPU, en sus siglas en inglés).

La imagen muestra una abstracción de un soporte multi-core x64+GPU para computación. Dispone de hasta 512 procesadores organizados en 16 grupos de 32 procesadores llamados multi-procesadores, estos últimos trabajan con un modelo de arquitectura SIMT (single instruction multiple thread), cuando la GPU procesa una instrucción genera grupos de 32 threads llamados warps que realizan el procesamiento de los datos en paralelo.

3. Programas realizados.

Introducción.

En este trabajo fin de grado hemos realizado diversos programas para profundizar en el conocimiento de la programación paralela y distribuida. Por ello hemos llevado a cabo diferentes tests para comprobar el comportamiento de las implementaciones que hemos desarrollado combinando diferentes paradigmas de programación paralela con modelos de memoria compartida y distribuida. Las pruebas se han llevado a cabo sobre 8 nodos con procesadores de tipo Intel Xeon de 24 cores cada uno y con 32 GB de memoria RAM. Además, cada nodo posee una GPU Nvidia Tesla K20 y una tarjeta de red InfiniBand Connect x-3 (FDR) a 56 Gbps. En dichos nodos disponemos de MvaPich2 la cual es una versión desarrollada en el *Network-Based Computing Laboratory* en la *Ohio State University* basada en *Mpich* que es una implementación de la interfaz de MPI. También se dispone de las librerías Pthreads, OpenMP y CUDA.

Para poner a prueba los diferentes entornos de programación, se ha partido de un primer programa MPI que realiza la multiplicación de dos matrices. Posteriormente este programa se ha extendido para que haga uso de threads mediante la librería Pthreads. También se ha extendido para que haga uso de la librería OpenMP. Estas implementaciones se han utilizado posteriormente como punto de partida para probar ciertos aspectos de diseño, como puede ser el uso eficiente de la memoria caché o el patrón de accesos a memoria (mediante el uso de matrices transpuestas). Finalmente se ha implementado la multiplicación de matrices, usando MPI como base, para la ejecución sobre coprocesadores vectoriales y GPUs.

Para cada implementación hemos realizados pruebas con matrices de tipo *double* de 1536x1536, 3072x3072, 6144x6144 y 12288x12288 elementos. A continuación presentamos los diferentes casos estudiados:

3.1 Test MPI.

En esta primera aproximación hemos ejecutado un algoritmo que únicamente utiliza MPI, de manera que el proceso con el *rank* = 0 (el *rank* es una etiqueta numérica que identifica inequívocamente cada proceso) es el encargado de generar las matrices A y B para después distribuirlas entre el resto de procesos. Se han generado 24 procesos en cada nodo involucrado,

tantos como cores de los que disponen. Cada proceso recibe un subconjunto de filas de la matriz A y una copia completa de la matriz B, posteriormente calcula su parte de la multiplicación matricial y finalmente devuelve su resultado parcial al proceso maestro el cual compone los resultados en la matriz C.

Simplificación del código:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
// SIZE: Alto y ancho de las matrices A,B y C
#define SIZE 3072

// Recibe el puntero de una matriz [SIZE][SIZE] y la rellena con doubles entre [0-100[
void fill_matrix(double* x) {
        long i, j;
        for (i = 0; i < SIZE; ++i) {
                for (j = 0; j < SIZE; ++j) {
                        x[i * SIZE + j] = (double)(rand() % 100);
                }
        }
}

int main(int argc, char *argv[]){
        // Declaración de variables
        int myrank, P, from, to, i, j, k;
        double *A, *B, *C, *A_local, *C_local;
        MPI_Status status;

        // Inicialización de MPI
        MPI_Init (&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Comm_size(MPI_COMM_WORLD, &P);

        // from: posición de la primera fila de la matriz A que debe calcular cada proceso
        from = myrank * SIZE/P;
        // to: posición de la última fila de la matriz A que debe calcular cada proceso
        to = (myrank+1) * SIZE/P;
        // Reserva de memoria en cada proceso para la sección de la matriz A que le corresponde a cada uno
        A_local = malloc (SIZE * (SIZE/P) * sizeof(double));
        // Reserva de memoria en cada proceso para la matriz de resultados locales
        C_local = malloc (SIZE * (SIZE/P) * sizeof(double));
        // Reserva de memoria en cada proceso para la matriz B
        B = malloc (SIZE * SIZE * sizeof(double));

        // El siguiente código solo lo ejecutara el proceso principal o master
        if (myrank==0) {
                // Reserva de memoria para la matriz A
                A = malloc (SIZE * SIZE * sizeof(double));
                // Reserva de memoria para la matriz resultado C
                C = malloc (SIZE * SIZE * sizeof(double));
                // Puebla la matriz A con valores
                fill_matrix(A);
                // Puebla la matriz A con valores
                fill_matrix(B);
        }

        // El proceso master envía una copia de la matriz B a todos los procesos
        MPI_Bcast (B, SIZE*SIZE, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        // El proceso master envía la sección que de la matriz A que le corresponde calcular a cada  proceso
        MPI_Scatter (A, SIZE*SIZE/P, MPI_DOUBLE, A_local, SIZE*SIZE/P, MPI_DOUBLE, 0,
        MPI_COMM_WORLD);
```

```
        // Cada proceso calcula el resultado de la multiplicación matricial de su parte de la matriz A y lo guarda en
        la matriz C_local
        // Iteramos por cada una de las filas que tiene que calcular cada proceso
        for (i=0; i<SIZE/P; i++){
                // La variable j nos marca la columna de B
                for (j=0; j<SIZE; j++) {
                        // La variable k nos indica la columna de A y la fila de B
                        for (k=0; k<SIZE; k++){
                                C_local[i * SIZE + j] += A_local[i * SIZE + k] * B[k * SIZE + j];
                        }
                }
        }
        // Todos los procesos envian sus resultados parciales al proceso master que los recoge en la matriz C
        MPI_Gather (C_local, SIZE*SIZE/P, MPI_DOUBLE, C, SIZE*SIZE/P, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

        // Se finalize MPI
        MPI_Finalize();
        // Fin de la ejecución
        return 0;
}
```

En el código del programa se puede observar que cada proceso MPI reserva memoria RAM para albergar la matriz B entera y la parte correspondiente de las matrices A y C. Por otra parte el proceso master inicializa las matrices A y B, a continuación envía la matriz B a todos los procesos MPI con un *broadcast* y también la parte de la matriz A que necesita cada uno con un *scatter*. Después, todos los procesos calculan la parte de la matriz resultado que tienen asignada. Para finalizar el proceso master recibe todas las secciones de la matriz resultado con un *gather* y finaliza la ejecución.

Resultados:

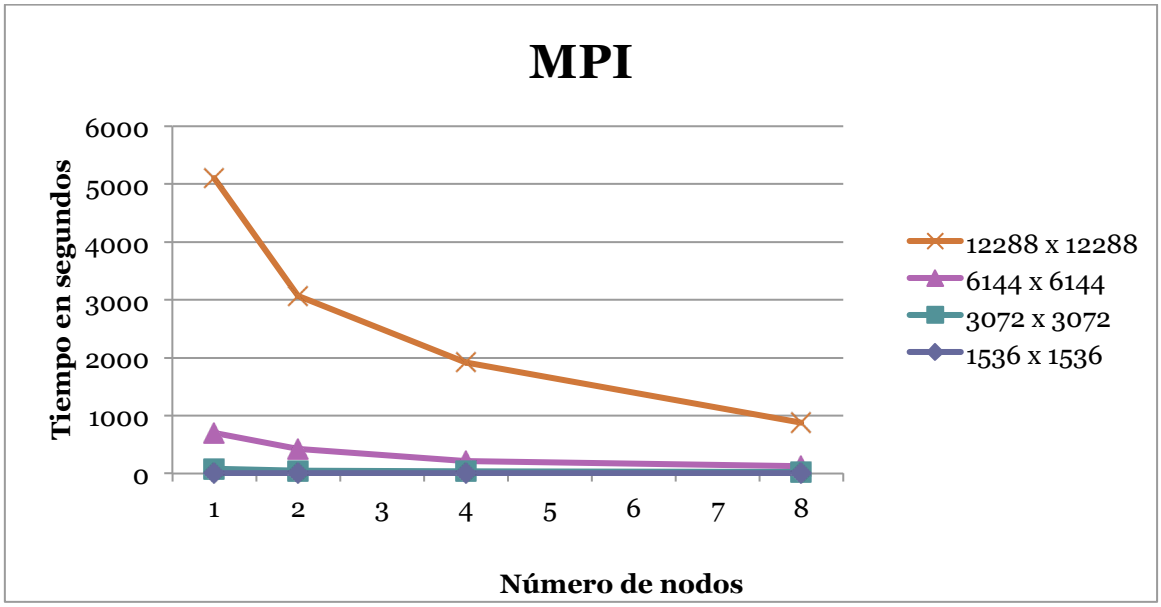

Ilustración 11: Comparativa de tiempos de ejecución de la versión en MPI.

La ilustración 11 muestra que en la ejecución con matrices de tamaño 12288x12288 con 1 y 2 nodos se obtiene una mejora de rendimiento del 66% en lugar de un 100% teórico que supondríamos ya que hemos duplicado el número de nodos. Esta diferencia es debida a que aunque el número de operaciones aritméticas en los dos casos permanece constante no ocurre igual con la cantidad de mensajes de comunicación entre procesos de MPI ya que tenemos en el primer caso 24 instancias del programa y en el segundo 48. Este hecho se repite cada vez que incrementamos el número de nodos involucrados, por lo que podemos observar como la pendiente de la curva disminuye progresivamente debido a la sobrecarga de comunicaciones.

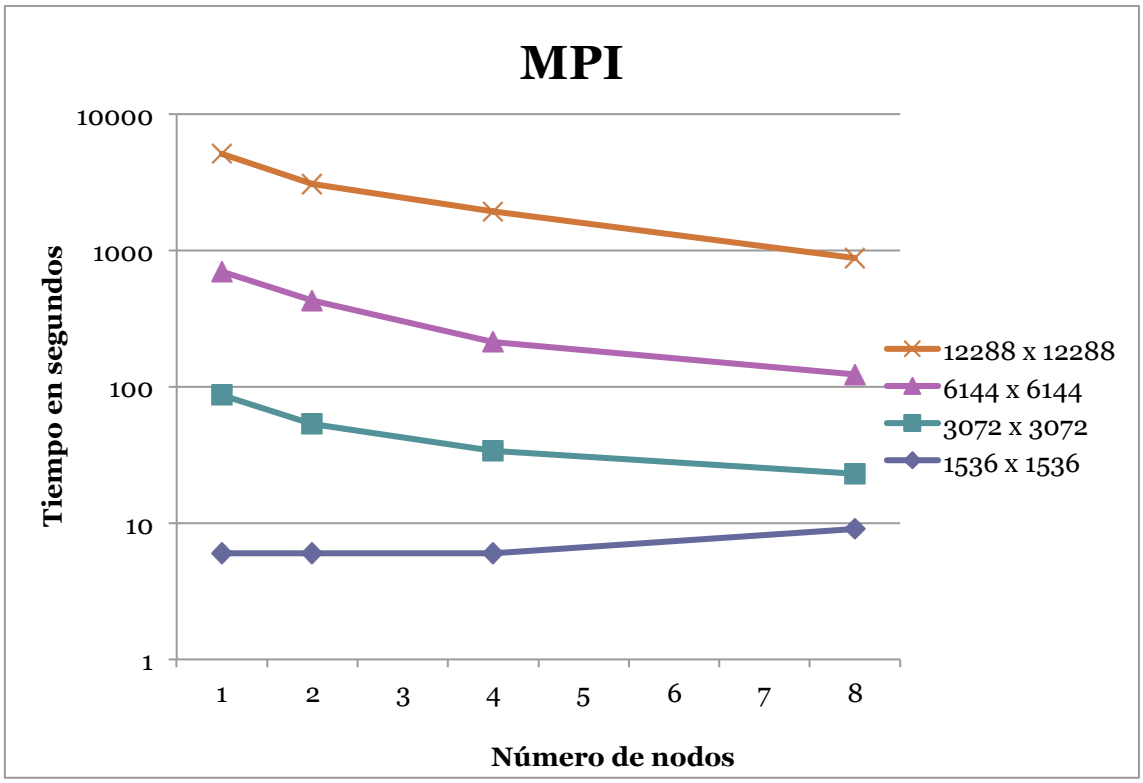

Ilustración 12: Comparativa de tiempos de ejecución de la versión en MPI (eje Y en escala logarítmica en base 10).

Dado que en la ilustración 11 es difícil distinguir las curvas para los tamaños más pequeños de matriz, la ilustración 12 muestra los mismos resultados en escala logarítmica para el eje Y.

Estudiando los datos en la ilustración 12 podemos apreciar en la prueba con tamaño de 1536x1536 que la pendiente se invierte, este hecho nos indica que el beneficio obtenido por distribuir las operaciones aritméticas sobre más nodos es inferior al coste del incremento de las comunicaciones entre nodos con lo que obtenemos un decremento del rendimiento.

Una de las primeras deficiencias que podemos apreciar en nuestro algoritmo es que al utilizar un entorno de memoria distribuida estamos replicando 24 veces la matriz B en cada nodo con el coste en mensajes y en memoria RAM que conlleva. En este sentido, nótese que aunque en cada nodo existe un dominio de memoria compartida, el uso de MPI convierte indirectamente esa memoria compartida en memoria distribuida.

3.2 Test MPI + Pthreads.

En este caso hemos utilizado una implementación mixta de memoria compartida y distribuida. Creamos un único proceso por nodo y cada uno de ellos genera 23 hilos adicionales. Todos los hilos del mismo nodo comparten las matrices A, B y Resultado. De esta manera reducimos enormemente tanto el número de mensajes como el uso de memoria.

Simplificación del código:

```
#include <mpi.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

#define SIZE 12288
// NUM_THREADS: número de hilos que creará el proceso
#define NUM_THREADS 23
double *A, *B, *C, *A_local, *C_local;
struct v {
        // from: índice de la primera fila que debe procesar el hilo
        int from;
        // to: índice de la última fila que debe procesar el hilo
        int to;
        // id: identificador del hilo
        int id;
};

void fill_matrix(double* x) {
        long i, j;
        for (i = 0; i < SIZE; ++i) {
                for (j = 0; j < SIZE; ++j) {
                        x[i * SIZE + j] = (double)(rand() % 100);
                }
        }
}

// function que calcula la parte de la multiplicación matricial que le corresponde al hilo que la ejecuta en base al parametro “param” que recibe
void *operate(void *param) {
        struct v *data = param;
        int i, j, k;
        for (i=data->from; i<data->to; i++){
                for (j=0; j<SIZE; j++) {
                        for (k=0; k<SIZE; k++){
                                C_local[i * SIZE + j] += A_local[i * SIZE + k] * B[k * SIZE + j];
```

```
                                }
                        }
                }
}

int main(int argc, char *argv[]){
        int myrank, P, namelen;
        int iam = 0, np = NUM_THREADS+1, work = 0, from = 0, to = 0, i=0, j=0, k=0, rc, provided;
        pthread_attr_t attr;
        char processor_name[MPI_MAX_PROCESSOR_NAME];
        MPI_Status status;

        // Iniciamos MPI indicándole que vamos a trabajar con threads y que queremos un nivel de  seguridad
        MPI_THREAD_FUNNELED que indica que solo el hilo principal podrá realizar llamadas MPI.
        MPI_Init_thread (&argc, &argv, MPI_THREAD_FUNNELED, &provided);

        // Comprobamos que MPI nos permite ejecutar el programa con el nivel de seguridad  que le hemos
        solicitado
        if (provided < MPI_THREAD_FUNNELED){
                printf("request: %d, provided: %d",MPI_THREAD_FUNNELED, provided );
                MPI_Abort(MPI_COMM_WORLD, 1);
        }

        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Comm_size(MPI_COMM_WORLD, &P);
        MPI_Get_processor_name(processor_name, &namelen);

        A_local = malloc (SIZE * (SIZE/P) * sizeof(double));
        C_local = malloc (SIZE * (SIZE/P) * sizeof(double));
        B = malloc (SIZE * SIZE * sizeof(double));

        if (myrank==0) {
                A = malloc (SIZE * SIZE * sizeof(double));
                 C = malloc (SIZE * SIZE * sizeof(double));
                fill_matrix(A);
                fill_matrix(B);
        }

        from = myrank * SIZE/P;
        to = (myrank+1) * SIZE/P;

        MPI_Bcast (B, SIZE*SIZE, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        MPI_Scatter (A, SIZE*SIZE/P, MPI_DOUBLE, A_local, SIZE*SIZE/P, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

        // En este punto cada nodo dispone de su copia de la matriz B y de la sección de la matriz A que le
        corresponde calcular.
        // Declaramos los hilos
        pthread_t thread[NUM_THREADS];
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

        // Lanzamos los hilos asignandoles un struct 'v'
        for(i=1; i<=NUM_THREADS; i++){
                struct v *data = (struct v *) malloc(sizeof(struct v));
                data->from =  i * (SIZE/(P*np));
                data->to = (i+1) * (SIZE/(P*np));
                data->id = (i+(myrank*(NUM_THREADS+1)));
                // Cada hilo ejecutará la function operate  sobre una parte de la matriz A
                 rc = pthread_create(&thread[i-1], &attr, operate, data);
                if (rc) {
                        printf("ERROR; return code from pthread_create() is %d\n", rc);
                        exit(-1);
                }
        }

        // Ahora es el hilo principal el que realiza su parte de la multiplicación de matricial
```

```
        struct v *data2 = (struct v *) malloc(sizeof(struct v));
        data2->from = 0;
        data2->to = (SIZE/(P*np));
        data2->id = (myrank*(NUM_THREADS+1));
        operate(data2);

        // El proceso principal espera a que todos los hilos hayan terminado
        pthread_attr_destroy(&attr);
        for(i=0; i<NUM_THREADS; i++) {
                rc = pthread_join(thread[i], NULL);
                if (rc) {
                        printf("ERROR; return code from pthread_join() is %d\n", rc);
                        exit(-1);
                }
        }

        MPI_Gather (C_local, SIZE*SIZE/P, MPI_DOUBLE, C, SIZE*SIZE/P, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
        MPI_Finalize();
        return 0;
}
```

En esta versión solo se crea un proceso MPI por nodo, con lo que solo se reserva espacio para una matriz B en cada uno en lugar de para 24, como pasaba cuando utilizábamos solo MPI. En el código primero observamos que cada proceso MPI ha reservado la memoria RAM que necesita y el proceso master a enviado la matriz B y la parte correspondiente de la matriz A a cada uno. Después, cada proceso ha creado 23 hilos asignándoles una sección de la matriz A y haciéndoles calcular parte de la matriz resultado. Por su parte cada proceso principal también ha calculado una sección de la matriz resultado. Cuando cada proceso principal ha terminado de calcular su sección espera a que también terminen todos los hilos y los destruye. Finalmente el proceso master recibe todas las secciones de la matriz resultado de los procesos principales de cada nodo.

Resultados:

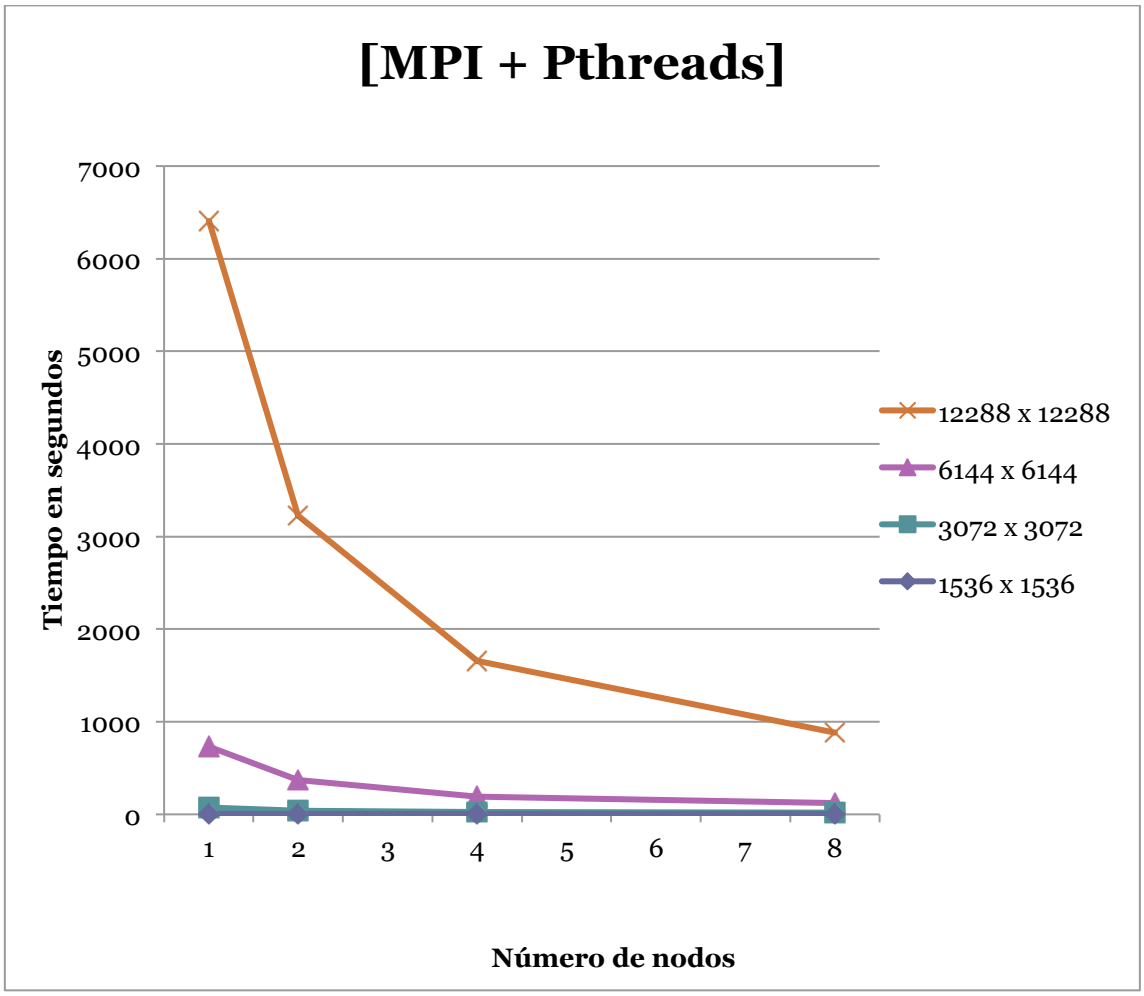

Ilustración 13: Comparativa de tiempos de ejecución de la versión en MPI combinada con Pthreads.

La ilustración 13 muestra que la versión 12288x12288 ejecutada sobre un nodo es más lenta que su homóloga de la versión distribuida con MPI, apreciamos que al sistema le cuesta más gestionar 1 proceso con 23 hilos que 24 procesos independientes en cada nodo. Sin embargo, así como en la versión anterior para 1 y 2 nodos teníamos una mejora del 66% en este caso es del 98% esto es debido a que los costes de comunicación se han reducido drásticamente.

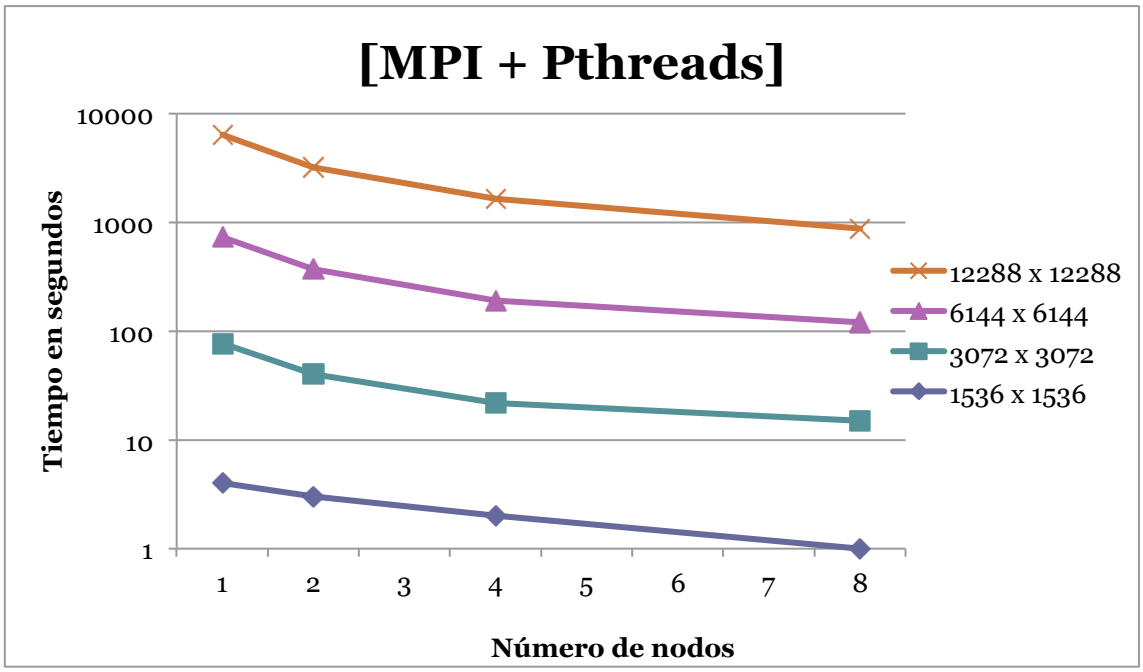

Ilustración 14: Comparativa de tiempos de ejecución de la versión en MPI combinada con Pthreads (eje Y en escala logarítmica en base 10).

La ilustración 14 muestra cómo en la prueba de MPI + Pthreads para 1536x1536, al contrario que en la versión MPI, sí que obtenemos mejoras de rendimiento.

Como podemos observar en el código, la dificultad a la hora de programar aumenta bastante al añadir la gestión de hilos a un simple algoritmo de multiplicación de matrices. Hemos obtenido mejoras de rendimiento pero lo pagamos con complejidad.

3.3 Test MPI + OpenMP.

En esta ocasión hemos utilizado OpenMP, una interfaz de más alto nivel que nos abstrae en gran medida de la gestión de los hilos pero que nos permite aprovecharnos de las bondades del uso del modelo de memoria compartida.

Código simplificado:

```
#include <stdio.h>
#include <mpi.h>
#include <omp.h>
#define SIZE 12288

void fill_matrix(double* x) {
        long i, j;
        for (i = 0; i < SIZE; ++i) {
                for (j = 0; j < SIZE; ++j) {
                        x[i * SIZE + j] = (double)(rand() % 100);
                }
```

```
		}
}

int main(int argc, char *argv[])
{
		int myrank, P, namelen;
		double *A, *B, *C, *A_local, *C_local;
		int iam = 0, iam_in = 0, rows_per_thread = 0, np = 0, from = 0, to = 0, i=0, j=0, k=0, provided;
		char processor_name[MPI_MAX_PROCESSOR_NAME];
		MPI_Status status;

		// Al igual que en la versión Pthreads solicitamos un nivel de seguridad MPI_THREAD_FUNNELED
		MPI_Init_thread (&argc, &argv, MPI_THREAD_FUNNELED, &provided);

		if (provided < MPI_THREAD_FUNNELED){
			printf("request: %d, provided: %d",MPI_THREAD_FUNNELED, provided );
			MPI_Abort(MPI_COMM_WORLD, 1);
		 }

		MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
		MPI_Comm_size(MPI_COMM_WORLD, &P);
		MPI_Get_processor_name(processor_name, &namelen);

		A_local = malloc (SIZE * (SIZE/P) * sizeof(double));
		C_local = malloc (SIZE * (SIZE/P) * sizeof(double));
		 B = malloc (SIZE * SIZE * sizeof(double));

		if (myrank==0) {
			A = malloc (SIZE * SIZE * sizeof(double));
			C = malloc (SIZE * SIZE * sizeof(double));
			fill_matrix(A);
			fill_matrix(B);
		}

		from = myrank * SIZE/P;
		to = (myrank+1) * SIZE/P;

		MPI_Bcast (B, SIZE*SIZE, MPI_DOUBLE, 0, MPI_COMM_WORLD
		MPI_Scatter (A, SIZE*SIZE/P, MPI_DOUBLE, A_local, SIZE*SIZE/P, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

		// En este punto cada nodo dispone de su copia de la matriz B y de la sección de la matriz A que le
		corresponde calcular. Cada nodo inicia el computo en paralelo indicando que todas las variables del hilo
		principal serán compartidas con el resto de hilos a excepción de las que indiquemos explicitamente   con
		private()
		#pragma omp parallel default(shared) private(iam, np, rows_per_thread, from, to, i, j, k) {
			// np: número de hilos trabajando en la seccion omp parallel
			np = omp_get_num_threads();
			// iam: identificador de cada hilo
			 iam = omp_get_thread_num();
			// rows_per_thread: cantidad de fils que deberá calcular cada hilo
			rows_per_thread =  SIZE / (P * np);
			// from: índice de la primera fila de la matriz A que le corresponde calcular a cada hilo
			from = iam * rows_per_thread;
			// to: índice de la última fila de la matriz A que le corresponde calcular a cada hilo
			to = (iam+1) * rows_per_thread;

			// Se calculan los datos
			for (i=from; i<to; i++){
				for (j=0; j<SIZE; j++) {
					for (k=0; k<SIZE; k++){
						C_local[i * SIZE + j] += A_local[i * SIZE + k] * B[k * SIZE + j];
					}
				}
			}
			// Sincronizamos todos los hilos antes de seguir
			#pragma omp barrier
```

```
        }

        MPI_Gather (C_local, SIZE*SIZE/P, MPI_DOUBLE, C, SIZE*SIZE/P, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

        MPI_Finalize();
        return 0;
}
```

Al igual que en la versión *Pthreads,* en cada nodo se mantiene una única copia de la matriz B. La diferencia principal de utilizar directivas OpenMp en lugar de la librería pthreads.h es que no es necesario inicializar explícitamente los hilos, sino que utilizando dichas directivas le indicamos al compilador que secciones del código queremos que se ejecuten en paralelo. Es el propio compilador el encargado de generar el código para la creación y gestión de los hilos. OpenMP permite especificar el *scope* de cada variable y también permite trabajar con varios tipos de secciones con distintos comportamientos como regiones paralelas, secciones críticas o *sections*.

Resultados:

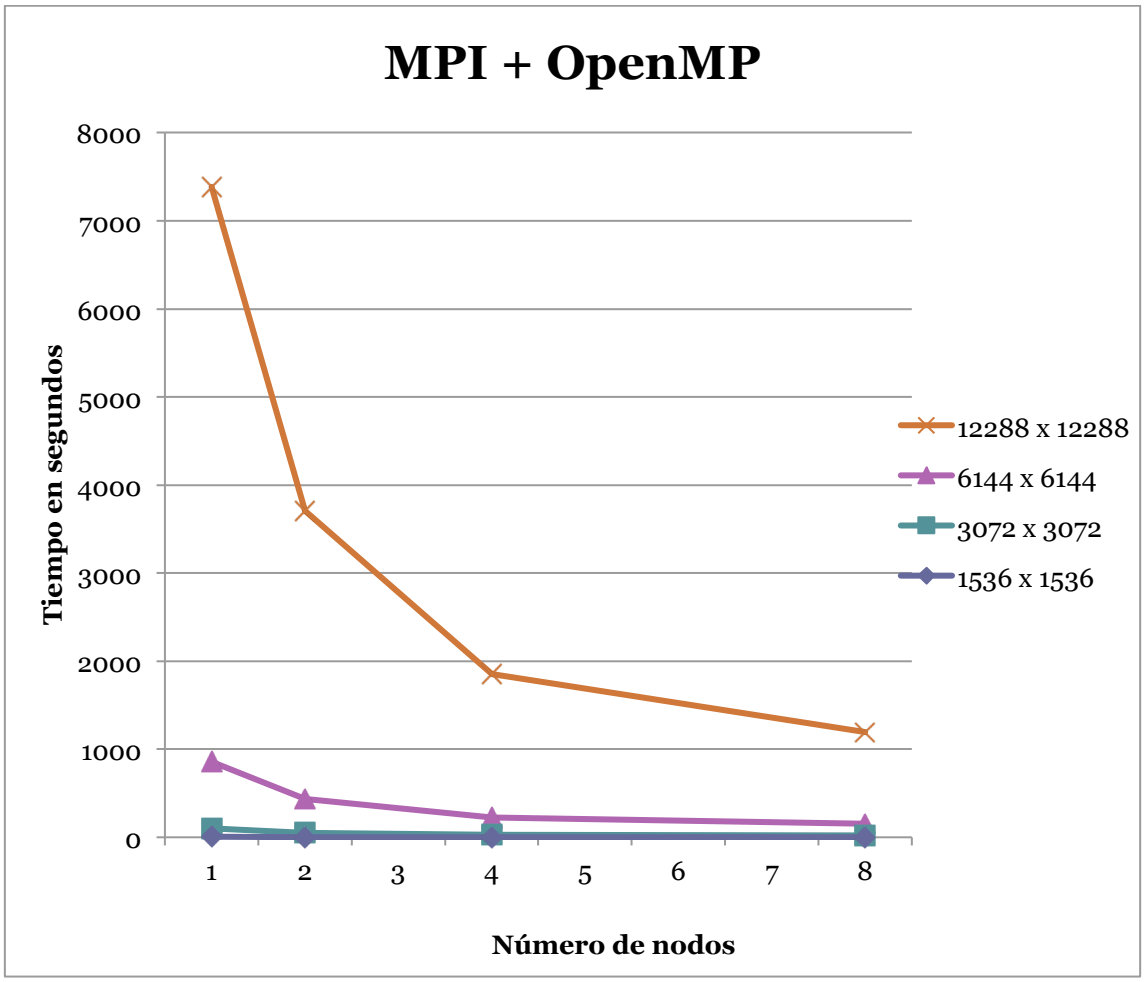

Ilustración 15: Comparativa de tiempos de ejecución de la versión en MPI combinada con OpenMP.

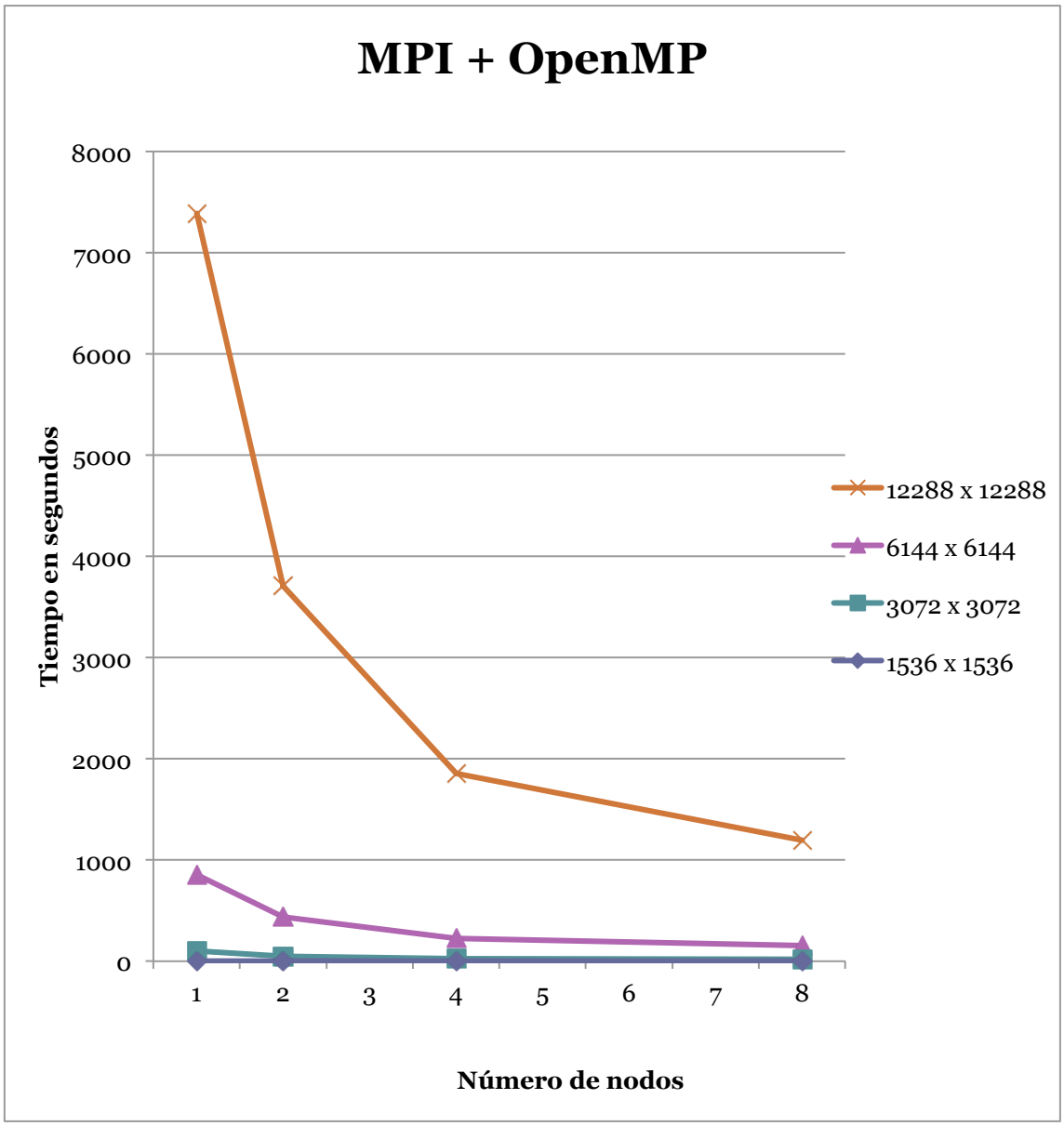

Ilustración 16: Comparativa de tiempos de ejecución de la versión en MPI combinada con OpenMP (eje Y en escala logarítmica en base 10).

En las ilustraciones 15 y 16 se muestra que hemos obtenido unos resultados ligeramente peores que en la versión MPI + Pthreads pero con un coste de implementación más bajo, a su vez sería más sencillo en un futuro realizar modificaciones sobre este código que en el utilizado en la versión de Pthreads.

3.4 Recopilación.

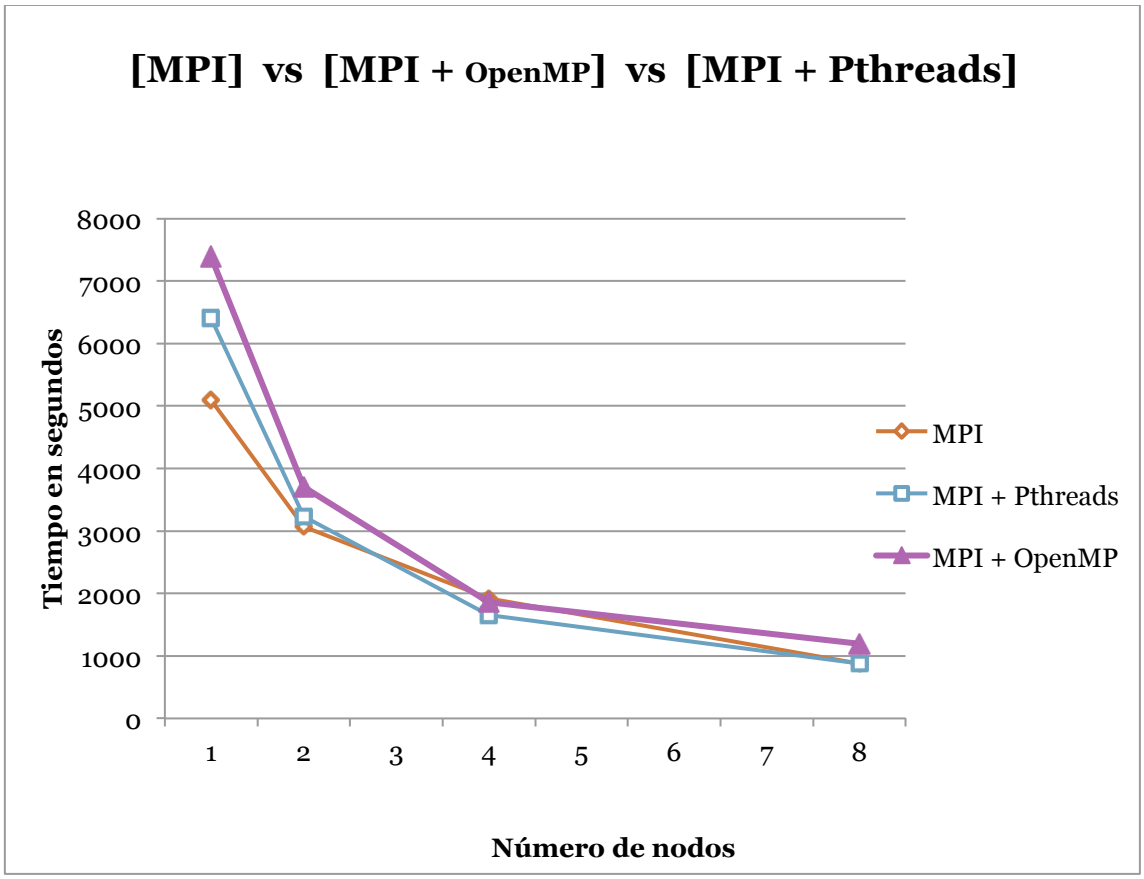

Ilustración 17: Comparativa de los tiempos de ejecución de las tres versiones para un tamaño de matriz de 12288x12288.

A modo de resumen, La ilustración 17 muestra las tres implementaciones en conjunto para el tamaño de matrices de 12288x12288 elementos, vemos cómo pese a que la versión MPI es la más eficiente ejecutada sobre un único nodo, su diferencia respecto a las versiones de memoria compartida disminuye conforme aumenta el número de nodos involucrados debido al coste en comunicaciones. En las pruebas con 8 nodos ya se observa cómo se invierte la tendencia. También vemos que OpenMP es la versión menos eficiente.

3.5 EFECTO DE LA MEMORÍA.

Hasta ahora nos habíamos centrado en distintas librerías que nos permiten trabajar con los modelos de memoria distribuida y compartida para mejorar el rendimiento de nuestro software. Pero existe otro factor determinante para maximizar la eficiencia: la gestión de la memoria caché. Es un hecho que se pierde mucho tiempo de computación mientras se hacen accesos a bloques de memoria caché los cuales pueden derivar en sustitución de bloques a

través de los distintos niveles de la jerarquía de esta memoria. Claramente no podemos ni queremos controlar manualmente el proceso de sustitución de bloques de la memoria caché, pero lo que sí que podemos hacer es preparar los datos en una estructura tal que minimice el número de bloques solicitados a la caché.

En el caso de nuestro algoritmo de multiplicación de matrices sabemos que para calcular, por ejemplo, el valor resultante en la posición de la fila 2 columna 1 de la matriz resultado lo obtenemos de multiplicar los elementos de la fila 2 de la matriz A con los elementos de la columna 1 de la matriz B. Rápidamente nos damos cuenta de que cuando el sistema solicite un bloque de memoria para leer un valor de la matriz A recibirá dicho bloque que contendrá el dato solicitado y un conjunto de datos adicionales y contiguos pertenecientes también a la misma fila de la matriz A que necesitaremos en las siguientes iteraciones de nuestro algoritmo. Sin embargo, cuando solicitemos un bloque para obtener un valor de la matriz B no podremos reutilizar el resto de valores que vienen en dicho bloque ya que, como hemos dicho, necesitamos acceder por columnas a esta matriz y nos encontramos que entre un dato y otro de los que requerimos hay tantos valores como la dimensión de la matriz.

Podemos mejorar este comportamiento fácilmente haciendo que el proceso inicial transponga la matriz B, de esta manera el cálculo a realizar sería de fila por fila en lugar de fila por columna con lo que reutilizamos cada bloque varias veces, reduciendo el número total de sustituciones de bloque y mejorando notablemente el rendimiento del programa.

Las modificaciones que hemos tenido que realizar a las tres versiones de nuestro algoritmo han sido:

- Añadir una función a la que le pasamos el puntero de la matriz B y transpone sus elementos.
- Hacer que el proceso master realice una llamada a dicha función justo después de poblar la matriz B.
- Cambiar los índices en el bucle de cálculo.

En las siguientes subsecciones vamos a analizar los beneficios de transponer la matriz B para los mismos casos vistos en las secciones anteriores.

3.5.1 Test MPI vs MPI (transpuesta).

Modificaciones del código:

```
…
void transpose_matrix(double* m){
        long i, j;
```

```
            double tmp;
            for (i=0; i<SIZE; i++) {
                        for (j=i; j<SIZE; j++){
                                    tmp = m[i * SIZE + j];
                                    m[i * SIZE + j] = m[j * SIZE + i];
                                    m[j * SIZE + i] = tmp;
                        }
            }
}
…
            if (myrank==0) {
                         A = malloc (SIZE * SIZE * sizeof(double));
                         C = malloc (SIZE * SIZE * sizeof(double));
                        fill_matrix(A);
                        fill_matrix(B);
                        transpose_matrix(B);
            }
…
            for (i=0; i<SIZE/P; i++){
                        for (j=0; j<SIZE; j++) {
                                    for (k=0; k<SIZE; k++){
                                                C_local[i * SIZE + j] += A_local[i * SIZE + k] * B[j * SIZE + k];
                                    }
                        }
            }
…
```

Resultados:

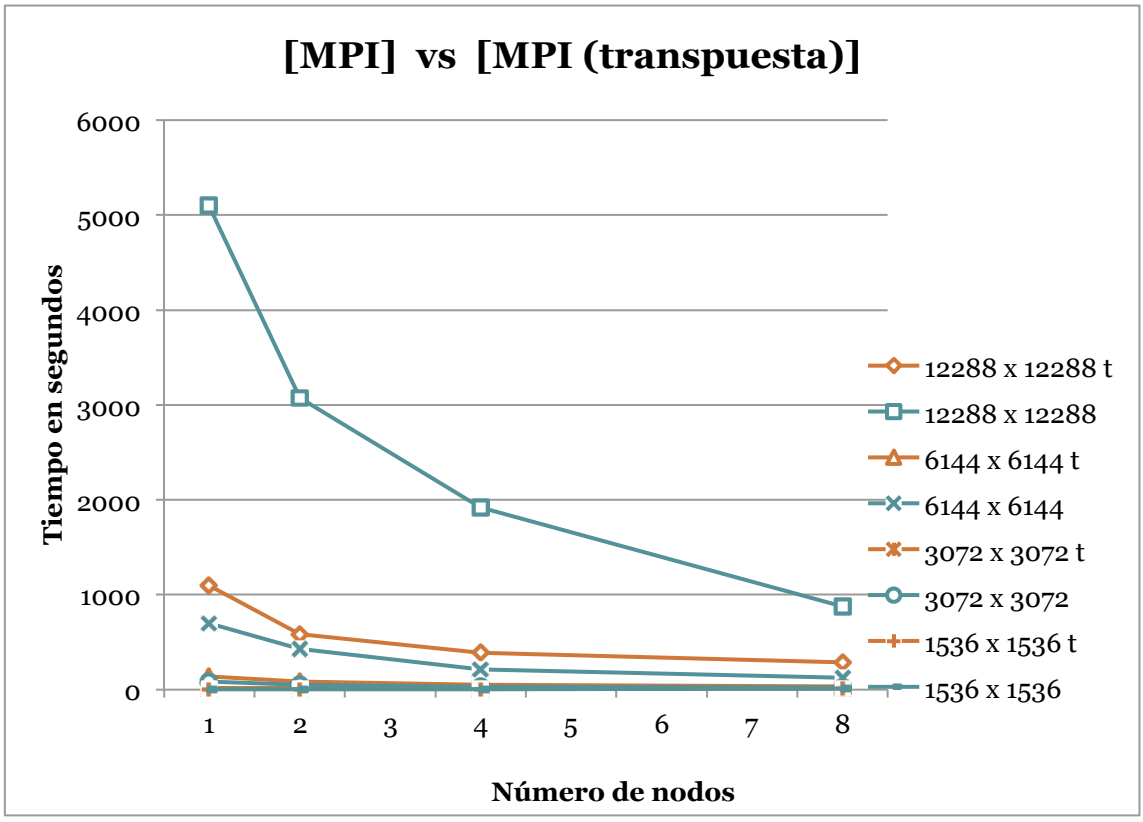

Ilustración 18: Comparativa de los tiempos de ejecución de la versión MPI con la versión MPI en la que se transpone la matriz B.

La ilustración 18 muestra mejoras de casi un 400% cuando utilizamos la implementación que facilita la reutilización de bloques en la memoria cache. Esto hace que haya un gran decremento del coste en comunicaciones entre el procesador y la memoria central, lo que aumenta la importancia relativa del coste en comunicaciones entre procesos MPI ya que este permanece intacto, por lo que la reducción del rendimiento cuando trabajamos con más nodos es mayor. Este comportamiento lo percibimos viendo cómo las curvas de la versión transpuesta son menos pronunciadas.

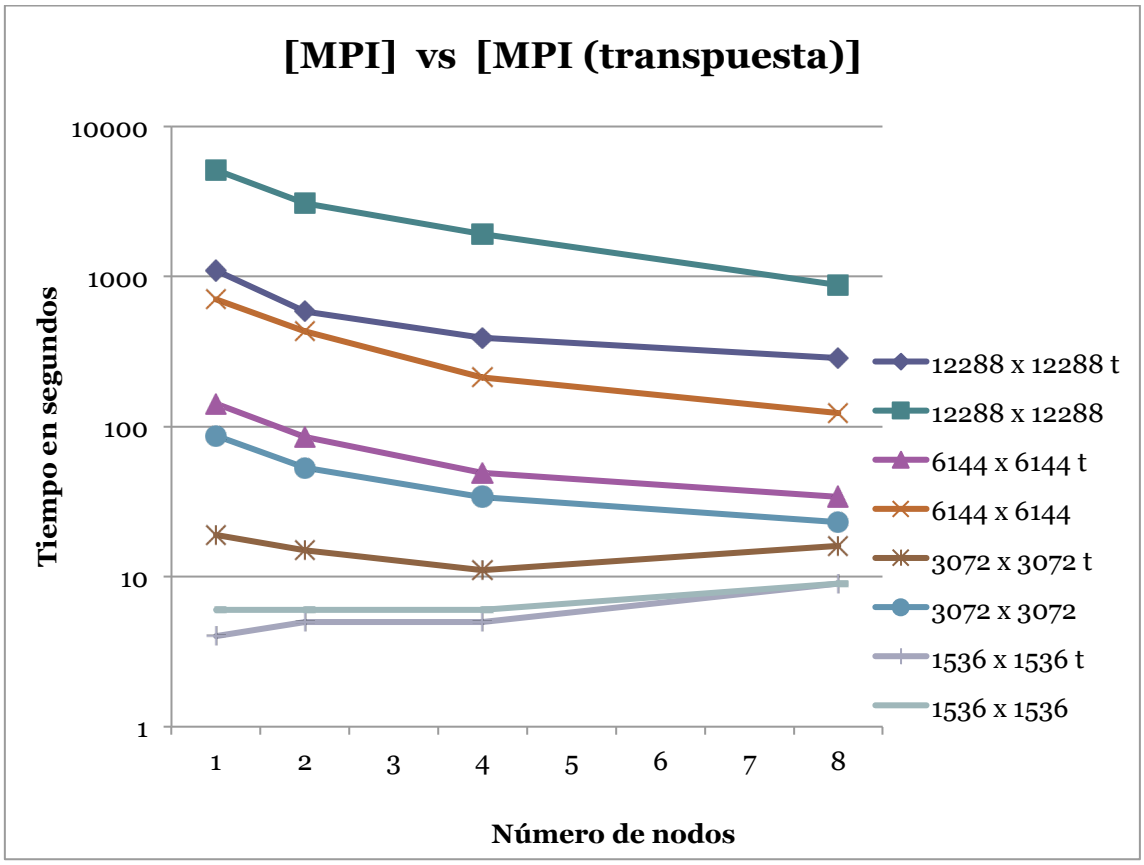

Ilustración 19: Comparativa de los tiempos de ejecución de la versión MPI con la versión MPI en la que se transpone la matriz B (eje Y en escala logarítmica en base 10).

La ilustración 19 muestra que como la computación se ha optimizado, las comunicaciones tienen un peso mayor respecto del total del tiempo de ejecución. Por tanto, la inversión de tendencia que antes se observaba en la curva de "1536x1536", ahora también se observa en la curva "3072x3072 t".

3.5.2 Test MPI + Pthreads vs MPI + Pthreads (transpuesta).

Modificaciones del código:

```
…
void transpose_matrix(double* m){
        long i, j;
        double tmp;
        for (i=0; i<SIZE; i++) {
                for (j=i; j<SIZE; j++){
                        tmp = m[i * SIZE + j];
                        m[i * SIZE + j] = m[j * SIZE + i];
                        m[j * SIZE + i] = tmp;
                }
        }
}
…
        if (myrank==0) {
                A = malloc (SIZE * SIZE * sizeof(double));
                C = malloc (SIZE * SIZE * sizeof(double));
                fill_matrix(A);
                fill_matrix(B);
                transpose_matrix(B);
        }
…
void *operate(void *param) {
        struct v *data = param;
        int i, j, k;
        for (i=data->from; i<data->to; i++){
                for (j=0; j<SIZE; j++) {
                        for (k=0; k<SIZE; k++)
                                C_local[i * SIZE + j] += A_local[i * SIZE + k] * B[j * SIZE + k];
                }
        }
}
…
```

Resultados:

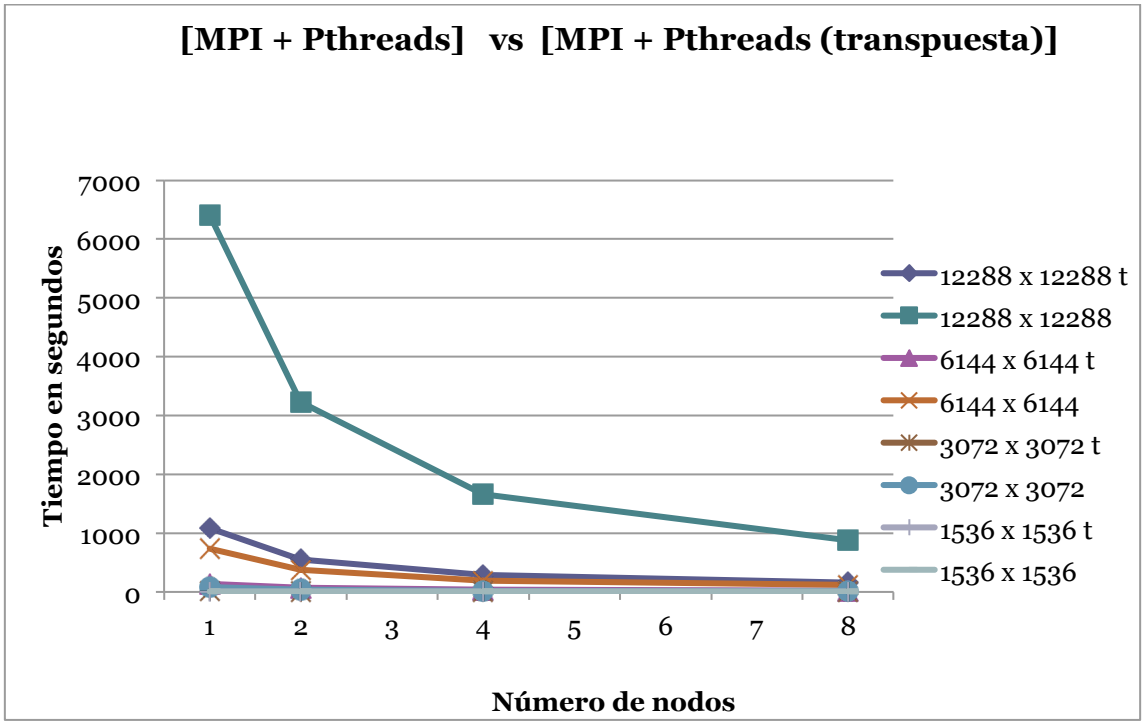

Ilustración 20: Comparativa de los tiempos de ejecución de la versión MPI + Pthreads con la versión MPI + Pthreads en la que se transpone la matriz B.

La ilustración 20 muestra que los *speed-up* llegan hasta el 500%, lo que indica que una mejora en la gestión de la memoria cache se hace más notable en un entorno de memoria compartida. Esto nos da pistas sobre por qué obteníamos mejores tiempos cuando ejecutábamos en un solo nodo con la versión MPI que con las versiones de memoria compartida. Es posible que sea por una mala gestión de la cache en el segundo caso.

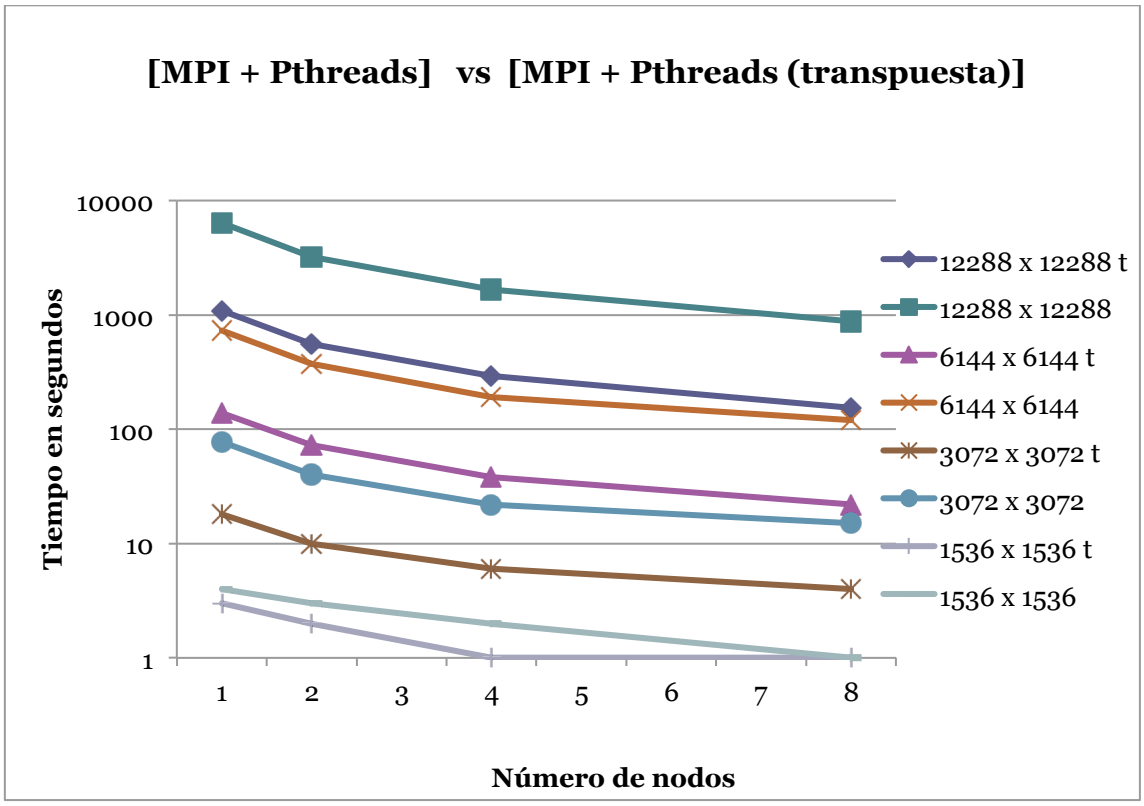

Ilustración 21: Comparativa de los tiempos de ejecución de la versión MPI + Pthreads con la versión MPI + Pthreads en la que se transpone la matriz B (eje Y en escala logarítmica en base 10).

La ilustración 21 muestra cómo las mejoras de rendimiento son más proporcionales al aplicar una optimización a la sección computacional que en la versión MPI. Esto es debido a que el peso de la parte de comunicaciones es mucho menor en el caso de modelos de memoria compartida y afecta menos a los tiempos de ejecución.

3.5.3 Test MPI + OpenMp vs MPI + OpenMP (transpuesta).

Modificaciones del código:

```
…
void transpose_matrix(double* m){
        long i, j;
        double tmp;
        for (i=0; i<SIZE; i++) {
                for (j=i; j<SIZE; j++){
```

```
                                tmp = m[i * SIZE + j];
                                m[i * SIZE + j] = m[j * SIZE + i];
                                m[j * SIZE + i] = tmp;
                        }
                }
        }
…
        if (myrank==0) {
                        A = malloc (SIZE * SIZE * sizeof(double));
                        C = malloc (SIZE * SIZE * sizeof(double));
                        fill_matrix(A);
                        fill_matrix(B);
                        transpose_matrix(B);
        }
…
                        for (i=from; i<to; i++){
                                for (j=0; j<SIZE; j++) {
                                        for (k=0; k<SIZE; k++){
                                                C_local[i * SIZE + j] += A_local[i * SIZE + k] * B[j * SIZE + k];
                                        }
                                }
                        }
…
```

Resultados:

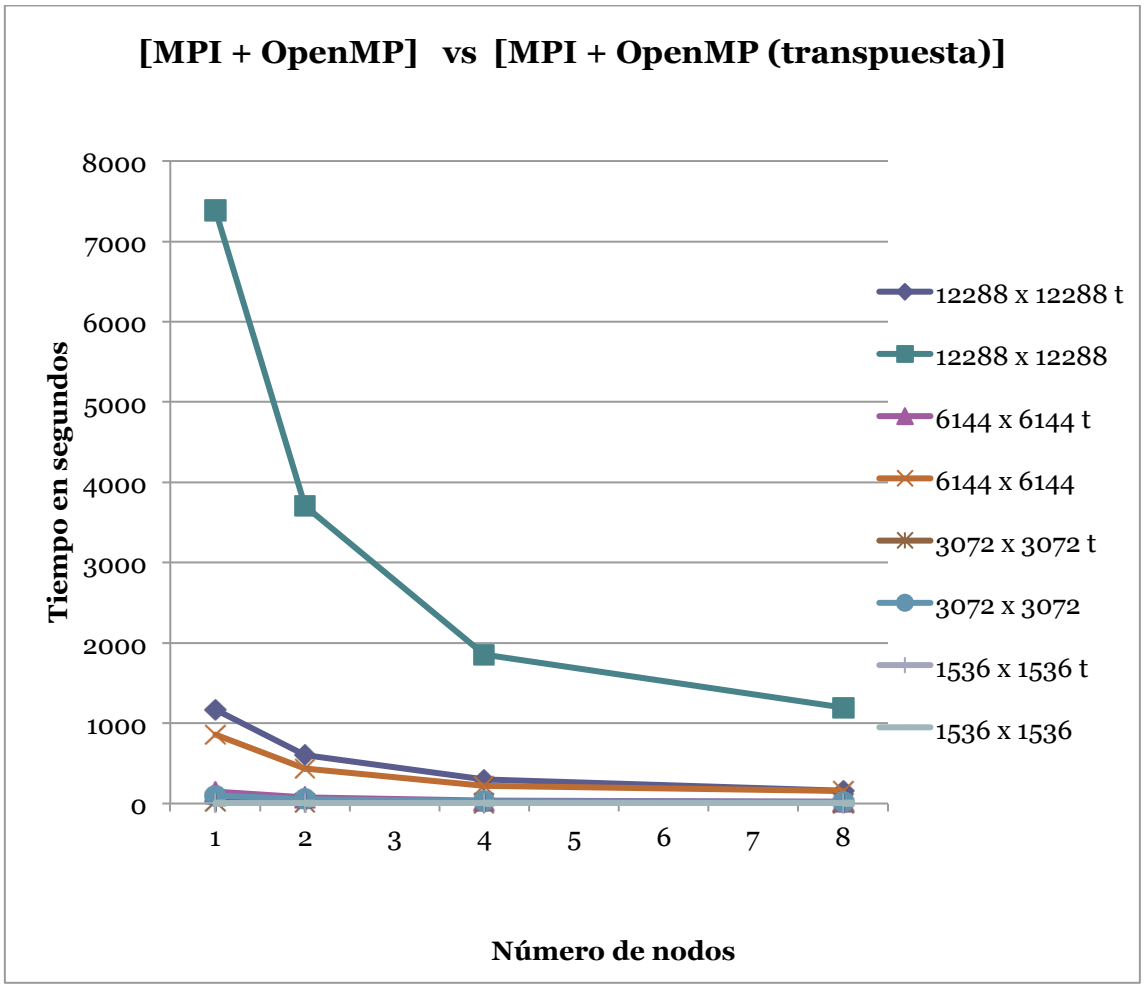

Ilustración 22: Comparativa de los tiempos de ejecución de la versión MPI + OpenMP con la versión MPI + OpenMP en la que se transpone la matriz.

En la ilustración 22 se muestra que con OpenMP obtenemos hasta un 530% de mejora del rendimiento que es ligeramente superior que la versión realizada con Pthreads.

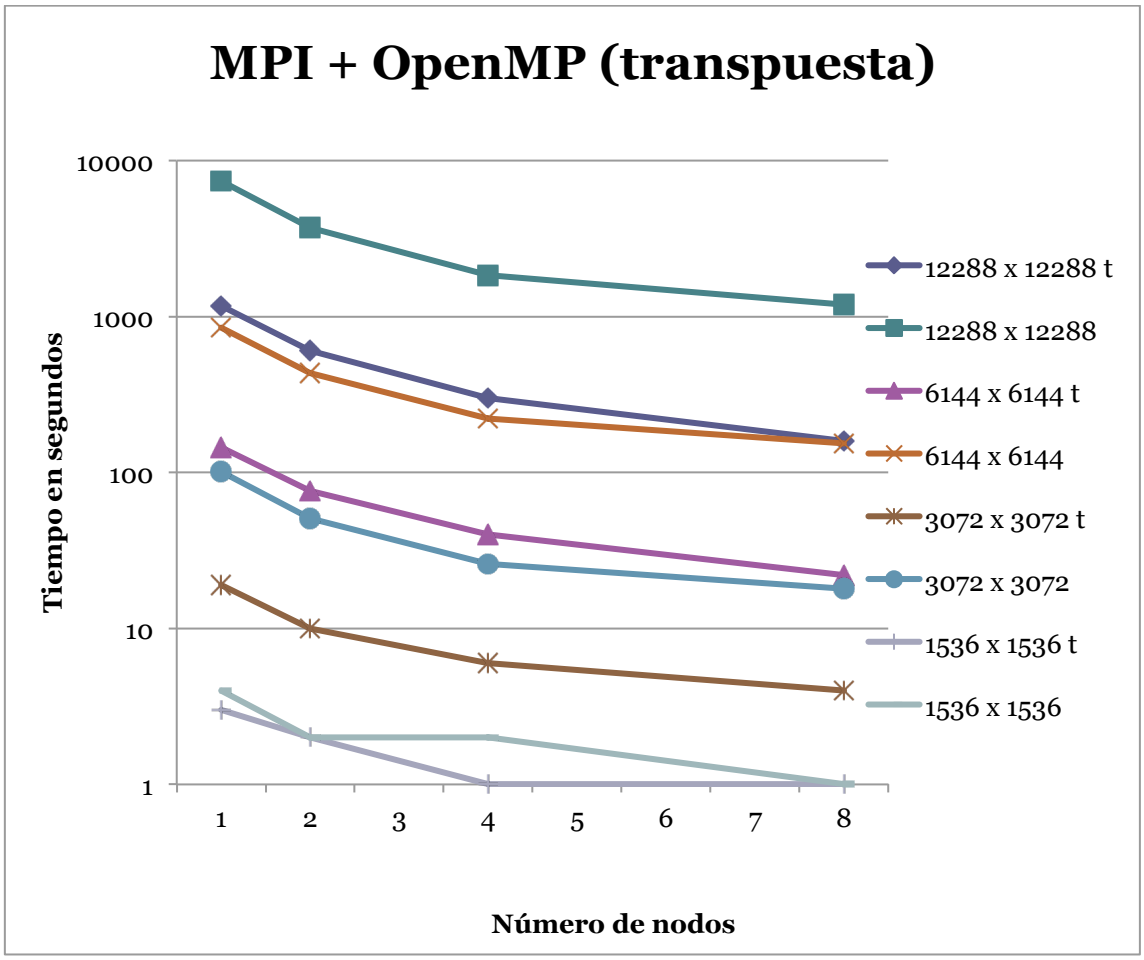

Ilustración 23: Comparativa de los tiempos de ejecución de la versión MPI + OpenMP con la versión MPI + OpenMP en la que se transpone la matriz (eje Y en escala logarítmica en base 10).

La ilustración 23 nos ofrece al igual que la versión con *Pthreads* unas mejoras más homogéneas que la versión MPI.

3.5.4 Recopilación.

A continuación mostramos dos gráficos que resumen lo visto hasta ahora sobre uso de entornos de memoria distribuida y compartida y sobre la disposición óptima de los datos.

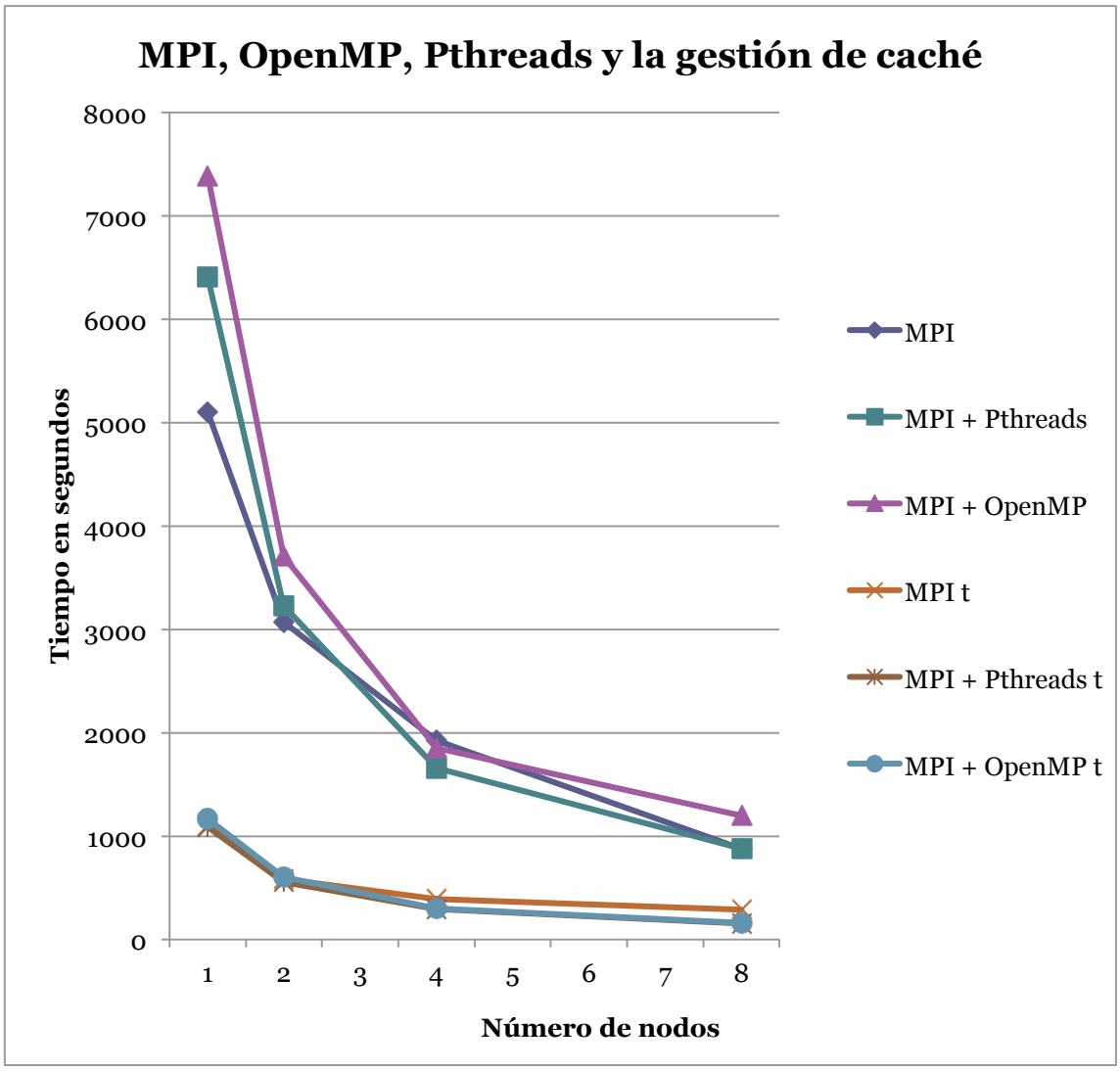

Ilustración 24: Comparativa de las versiones MPI, MPI + Pthreads y MPI + OpenMP en sus versiones normales y transpuestas para un tamaño de matriz de 12288x12288.

La ilustración 24 muestra que existe una notable diferencia de eficiencia entre las versiones que son más "amigables" con la gestión de la memoria caché y las que no. Dado que en las versiones en las que realizamos la transposición de la matriz B el peso de la parte computacional es menor, también obtenemos menores mejoras de rendimiento al distribuir la carga sobre más nodos. Esto queda reflejado en la gráfica con unas curvas con menor pendiente.

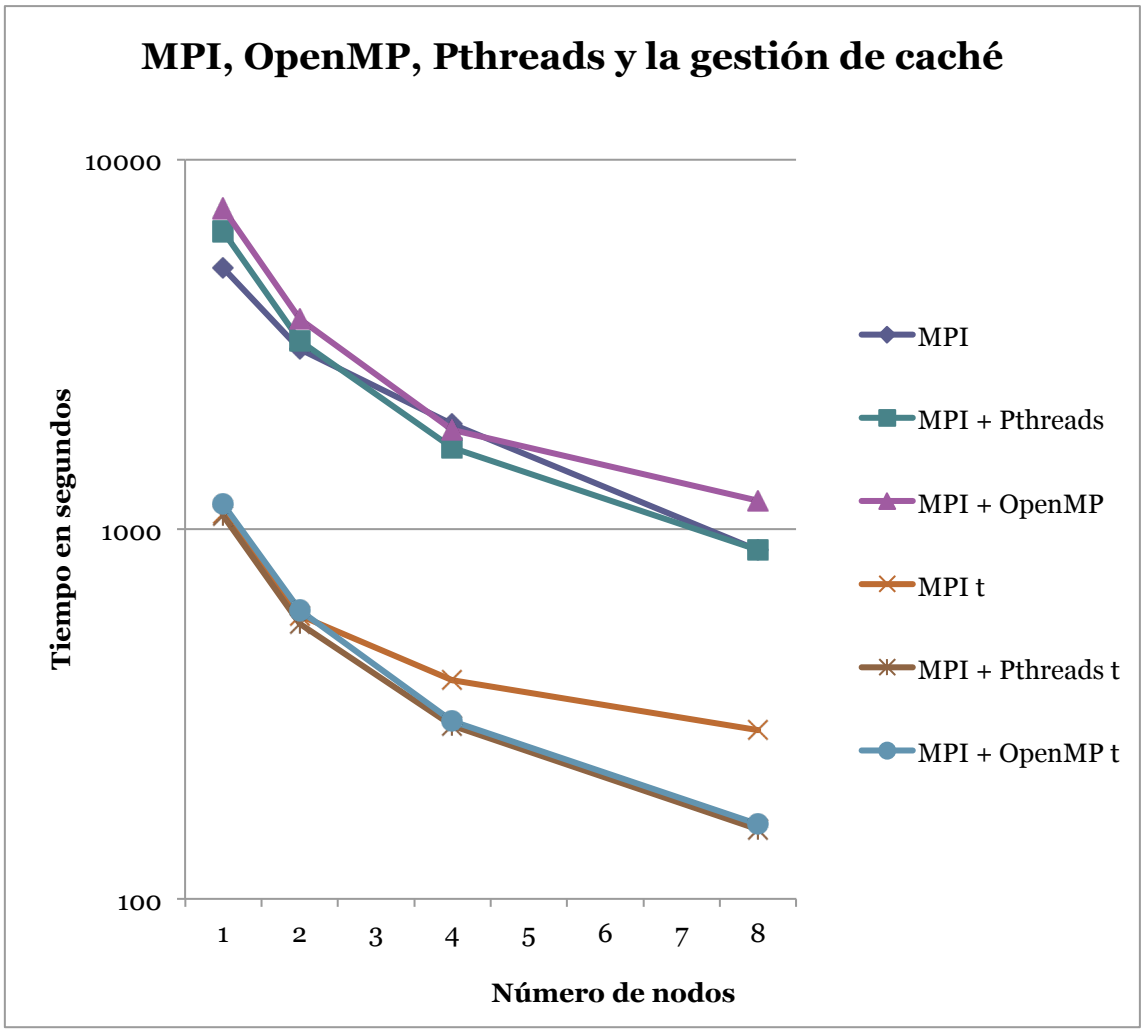

Ilustración 25: Comparativa de las versiones MPI, MPI + Pthreads y MPI + OpenMP en sus versiones normales y transpuestas para un tamaño de matriz de 12288x12288 (eje Y en escala logarítmica en base 10).

En la ilustración 25 vemos cómo las versiones de memoria compartida optimizadas se comportan mejor cuando aumentamos el número de nodos que utilizamos para repartir la carga de trabajo.

3.6 Coprocesador vectorial: funciones intrínsecas.

Como ya hemos mencionado, cada nodo del *cluster* dispone de 12 coprocesadores vectoriales los cuales son compatibles con la tecnología AVX que nos ofrece 16 vectores de 256 bits. En nuestro caso de uso utilizamos cuatro grupos de vectores, cada grupo dispone de un vector en el que cargamos 4 dobles de la matriz A, otro vector en el que cargamos 4 dobles de la matriz B y un tercer vector que almacena el resultado de la multiplicación elemento a elemento de los dos anteriores. Con esta estrategia y gracias al *pipelining* de instrucciones podemos tener simultáneamente hasta 16 multiplicaciones elemento a elemento.

En nuestra primera aproximación al uso de operaciones vectoriales nos hemos servido del conjunto de funciones intrínsecas del compilador GCC. Basándonos en la primera versión en la que solo utilizábamos MPI hemos realizado los cambios necesarios para que el cómputo se realice en el coprocesador vectorial.

Código simplificado:

```
#include <mpi.h>
#include <immintrin.h>
#include <stdio.h>
#include <stdlib.h>

#define SIZE 12288

void fill_matrix(double* x) {
	long i, j;
	for (i = 0; i < SIZE; ++i) {
		for (j = 0; j < SIZE; ++j) {
			x[i * SIZE + j] = (double)(rand() % 100);
		}
	}
}

void transpose_matrix(double* m){
	long i, j;
	double tmp;
	for (i=0; i<SIZE; i++) {
		for (j=i; j<SIZE; j++){
			tmp = m[i * SIZE + j];
			m[i * SIZE + j] = m[j * SIZE + i];
			m[j * SIZE + i] = tmp;
		}
	}
}

int main(int argc, char *argv[]) {
	int myrank, P, from, to, i, j, k;
	double *a_row, *b_row, *c_elem;
	double *A, *B, *C, *A_local, *C_local;
	MPI_Status status;

	MPI_Init (&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
	MPI_Comm_size(MPI_COMM_WORLD, &P);
```

```
	from = myrank * SIZE/P;
	to = (myrank+1) * SIZE/P;

	A_local = malloc (SIZE * (SIZE/P) * sizeof(double));
	C_local = malloc (SIZE * (SIZE/P) * sizeof(double));
	B = malloc (SIZE * SIZE * sizeof(double));

	if (myrank==0) {
		A = malloc (SIZE * SIZE * sizeof(double));
		C = malloc (SIZE * SIZE * sizeof(double));
		fill_matrix(A);
		fill_matrix(B);
		transpose_matrix(B);
	}

	MPI_Bcast (B, SIZE*SIZE, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Scatter (A, SIZE*SIZE/P, MPI_DOUBLE, A_local, SIZE*SIZE/P, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

	for (i=0; i<SIZE/P; i++){
		for (j=0; j<SIZE; j++) {
			a_row = &A_local[i * SIZE];
			b_row = &B[j * SIZE];
			calculate_matrix_element(a_row, b_row, &C_local[i * SIZE + j]);
		}
	}

	MPI_Gather (C_local, SIZE*SIZE/P, MPI_DOUBLE, C, SIZE*SIZE/P, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

	MPI_Finalize();
	return 0;
}

// Función que recibe una fila de la matriz A, otra de la matriz B y la posición de la matriz C donde se guardara el
resultado de la suma de la  multiplicación elemento a elemento de las dos filas
int calculate_matrix_element(double *a_row, double *b_row, double *c_elem){
	int index;
	*c_elem = 0.0;

	for(index=0; index<SIZE; index=index+16){
		// Inicializamos un vector de 256 bits  con cuatro doubles de una filade la matriz A
		 __m256d a_part = _mm256_set_pd(a_row[index], a_row[index+1], a_row[index+2],
		a_row[index+3]);
		// Inicializamos un vector de 256 bits  con cuatro doubles de una filade la matriz B
		 __m256d b_part = _mm256_set_pd(b_row[index], b_row[index+1], b_row[index+2],
		b_row[index+3]);
		// calculamos la multiplicación elemento a element de los dos vectores en el coprocesador
		__m256d result_part = _mm256_mul_pd(a_part, b_part);

		__m256d a_part2 = _mm256_set_pd(a_row[index+4], a_row[index+5], a_row[index+6],
		a_row[index+7]);
		__m256d b_part2 = _mm256_set_pd(b_row[index+4], b_row[index+5], b_row[index+6],
		b_row[index+7]);
		__m256d result_part2 = _mm256_mul_pd(a_part2, b_part2);

		__m256d a_part3 = _mm256_set_pd(a_row[index+8], a_row[index+9], a_row[index+10],
		a_row[index+11]);
		__m256d b_part3 = _mm256_set_pd(b_row[index+8], b_row[index+9], b_row[index+10],
		b_row[index+11]);
		__m256d result_part3 = _mm256_mul_pd(a_part3, b_part3);

		__m256d a_part4 = _mm256_set_pd(a_row[index+12], a_row[index+13], a_row[index+14],
		a_row[index+15]);
		__m256d b_part4 = _mm256_set_pd(b_row[index+12], b_row[index+13], b_row[index+14],
		b_row[index+15]);
```

```
                    __m256d result_part4 = _mm256_mul_pd(a_part4, b_part4);

                    // Extraemos los resultados del vector en un array de doubles
                    double* f = (double*)&result_part;
                    double* f2 = (double*)&result_part2;
                    double* f3 = (double*)&result_part3;
                    double* f4 = (double*)&result_part4;

                    // Sumamos todas las multiplicaciones parciales en el procesador principal acumulando el
                    resultado en la posicion de la matriz resultado que le corresponde
                    *c_elem = *c_elem + f[0] + f[1] + f[2] + f[3]+ f2[0] + f2[1] + f2[2] + f2[3]+ f3[0] + f3[1] +
f3[2] + f3[3]+ f4[0] + f4[1] + f4[2] + f4[3];
            }

   return 0;
}
```

El código pertenece a una función que recibe a una fila de la matriz A, otra fila de la matriz B y multiplica sus elementos devolviendo la suma de resultados en la posición correspondiente de la matriz C. Observamos que hemos tenido que precargar explícitamente los registros vectoriales y posteriormente extraer los resultados de ellos, con esto nos podemos hacer una idea de lo compleja que empieza a ser la implementación de algoritmos con el juego de funciones intrínsecas: tenemos que controlar el tipo de variables que queremos manejar, utilizar los vectores disponibles adecuadamente, no reutilizar vectores si aún están siendo utilizados por otras instrucciones previas, etc. Además, hemos de tener en cuenta si nos sale más rentable realizar ciertas operaciones en el procesador o en el coprocesador vectorial. Por ejemplo, tras probar diversas versiones hemos visto que nos resulta más eficiente hacer la suma de los 16 resultados parciales en el procesador que utilizando operaciones vectoriales.

Primero mostramos los resultados de la multiplicación de matrices con MPI utilizando funciones intrínsecas sin transponer la matriz B.

Resultados:

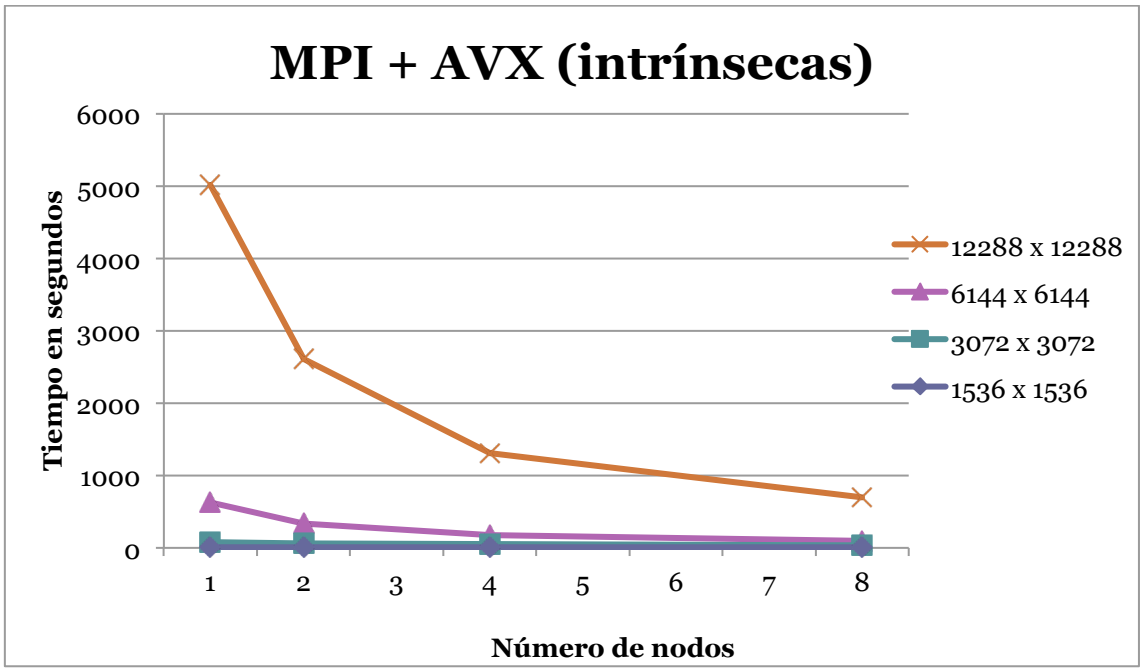

Ilustración 26: Comparativa de tiempos de ejecución de la versión en MPI combinada con intrínsecas de GCC para AVX.

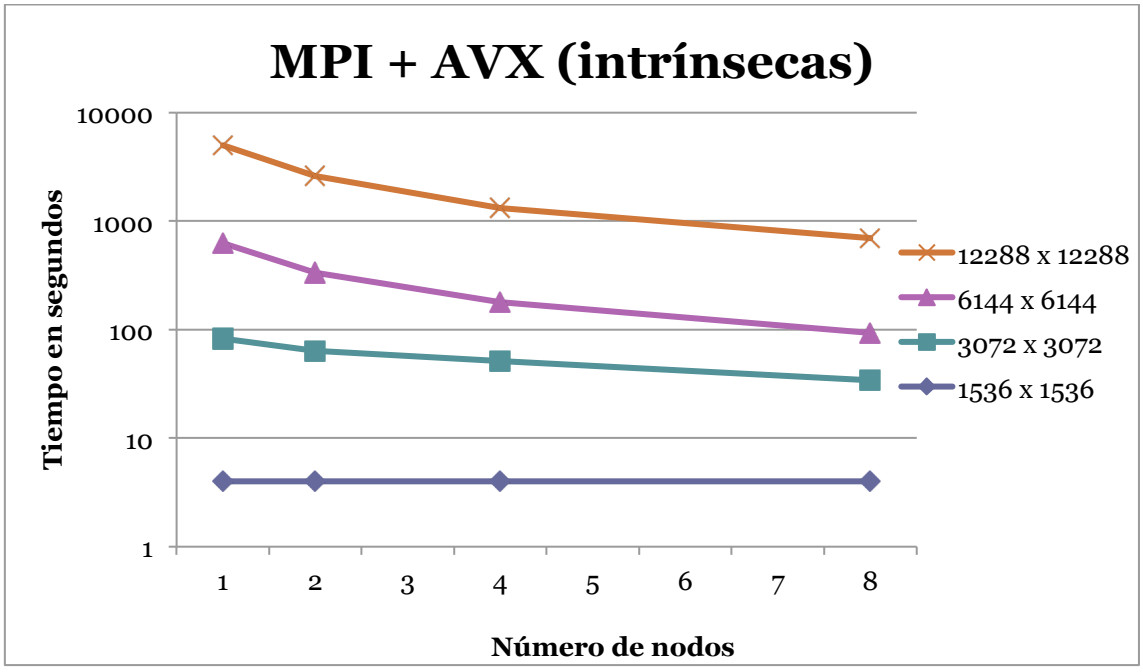

Ilustración 27: Comparativa de tiempos de ejecución de la versión en MPI combinada con intrínsecas de GCC para AVX (eje Y en escala logarítmica en base 10).

Las ilustraciones 26 y 27 muestran resultados son ligeramente mejores al realizar el cómputo sobre los coprocesadores. A continuación mostramos la comparativa de tiempos con la versión que realiza la transposición de la matriz B previa al cálculo.

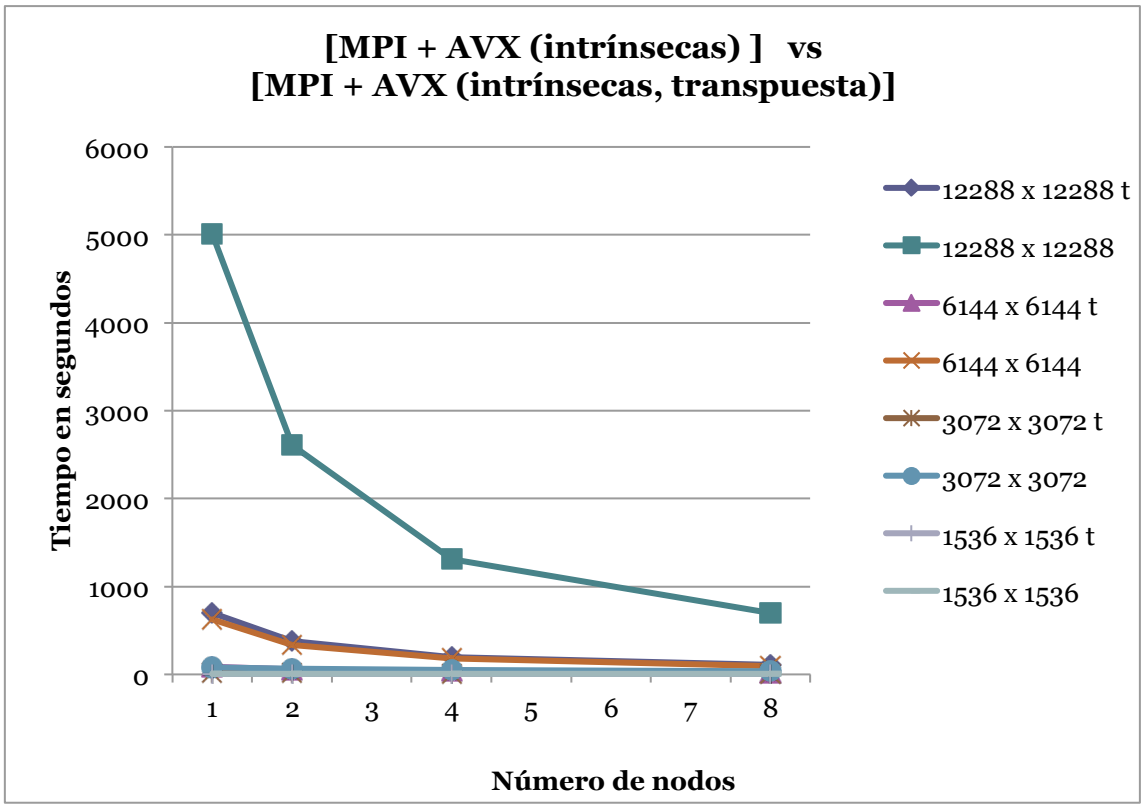

Ilustración 28: Comparativa de los tiempos de ejecución de la versión MPI + AVX (intrínsecas) con la versión MPI + AVX (intrínsecas) en la que se transpone la matriz B.

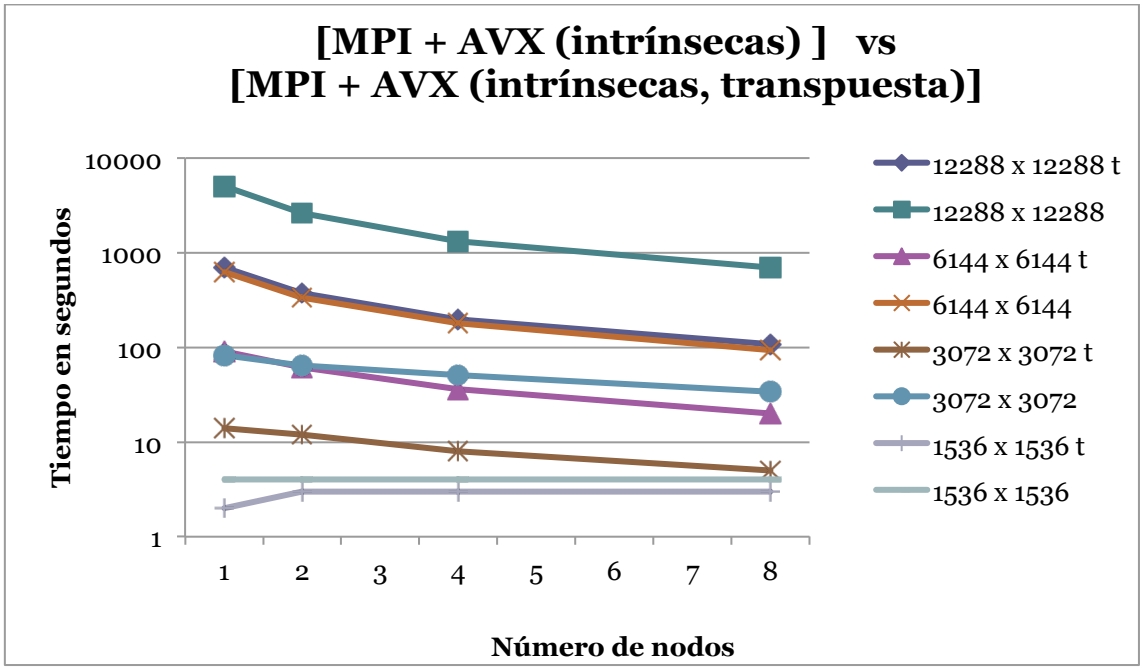

Ilustración 29: Comparativa de los tiempos de ejecución de la versión MPI + AVX (intrínsecas) con la versión MPI + AVX (intrínsecas) en la que se transpone la matriz B (eje Y en escala logarítmica en base 10).

Las ilustraciones 28 y 29 muestran, según lo esperado, una gran mejora del rendimiento al transponer la matriz, en algunos casos por encima del 600%.

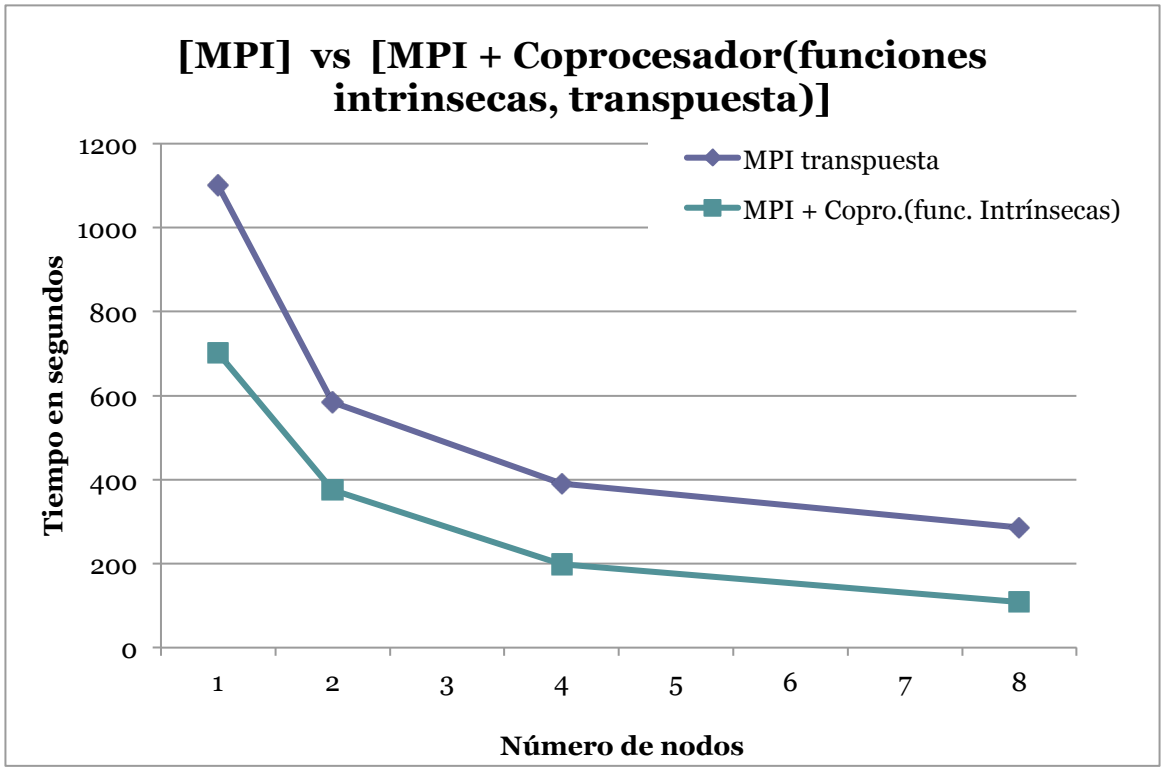

Ilustración 30: Comparativa MPI (transpuesta) con MPI con ejecución en coprocesador con funciones intrínsecas (transpuesta) para un tamaño de matriz de 12288x12288.

A modo de resumen, En la ilustración 30 se compara la versión MPI + AVX (intrínsecas, transpuesta) con su homóloga en procesador MPI (transpuesta), obtenemos aumentos de la eficiencia en torno al 60%.

Hemos comprobado que podemos mejorar el tiempo de ejecución si nuestro programa, o una parte de él, es susceptible de ser ejecutado sobre el coprocesador vectorial. Por otro lado, también sabemos que la implementación de algoritmos vectoriales es más complicada y además se reduce en gran medida su portabilidad a otros sistemas ya que habrían de ser tecnológicamente compatibles (no todos los coprocesadores soportan las mismas instrucciones vectoriales).

3.7 Coprocesador vectorial: autovectorizador.

Para llegar a un compromiso entre mejora de rendimiento y sencillez de implementación podemos ayudarnos de los autovectorizadores. GCC dispone de un autovectorizador muy sencillo de utilizar, simplemente añadiendo el *flag* –O3 en el comando de compilación obtendremos una versión de nuestro algoritmo que ejecutará sobre el coprocesador las operaciones que crea oportunas.

Las siguientes ilustraciones muestran los resultados obtenidos tras compilar la versión inicial MPI para el procesador con este *flag* de control:

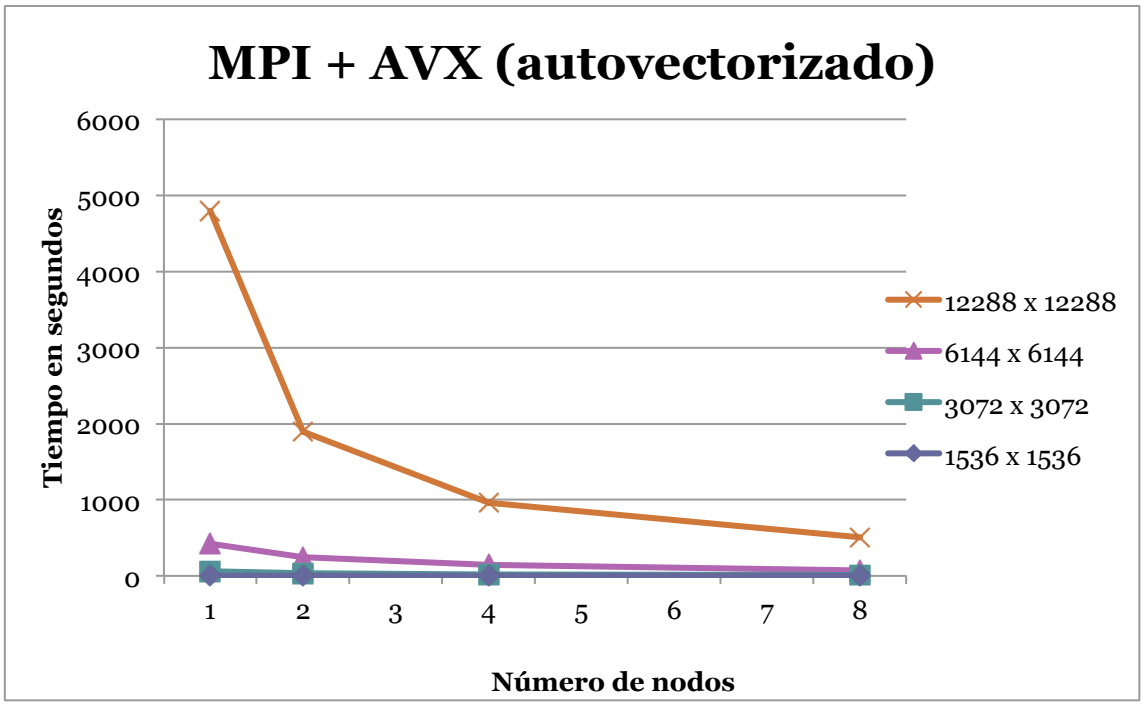

Ilustración 31: Comparativa de tiempos de ejecución de la versión en MPI autovectorizada.

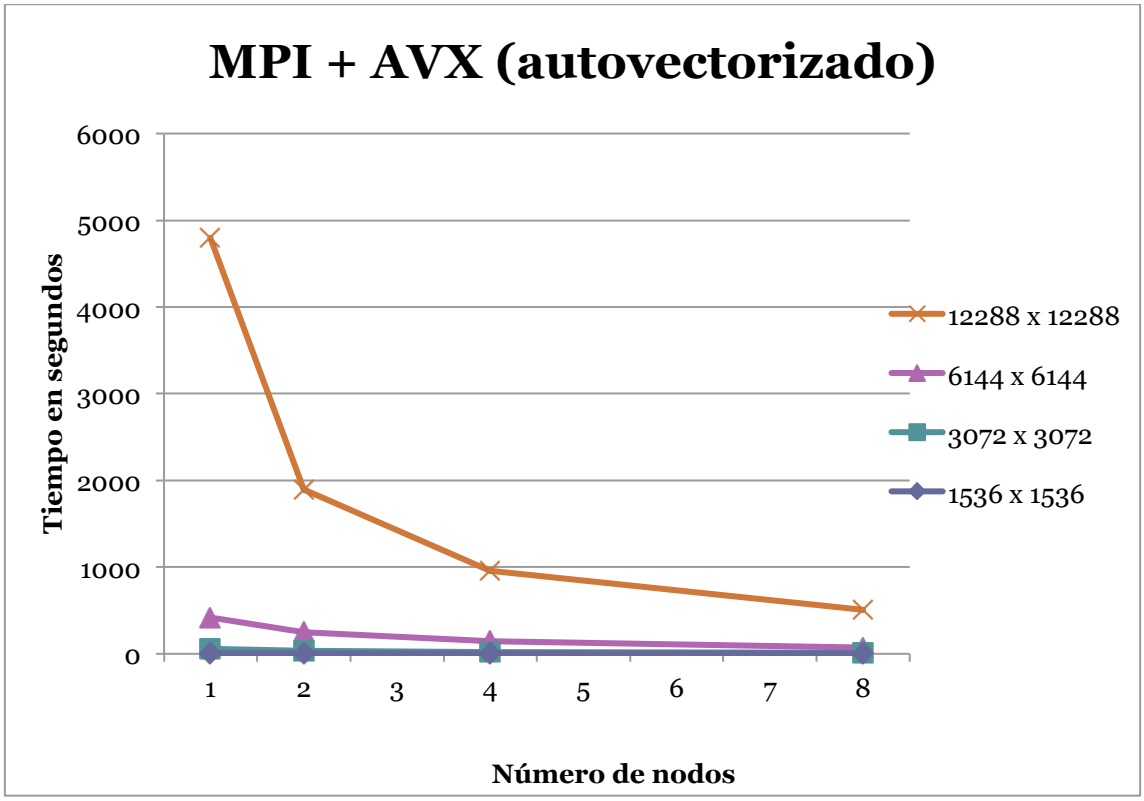

Ilustración 32: Comparativa de tiempos de ejecución de la versión en autovectorizada (eje Y en escala logarítmica en base 10).

Seguidamente se comparan los tiempos obtenidos con las versiones autovectorizadas de MPI y de MPI (transpuesta):

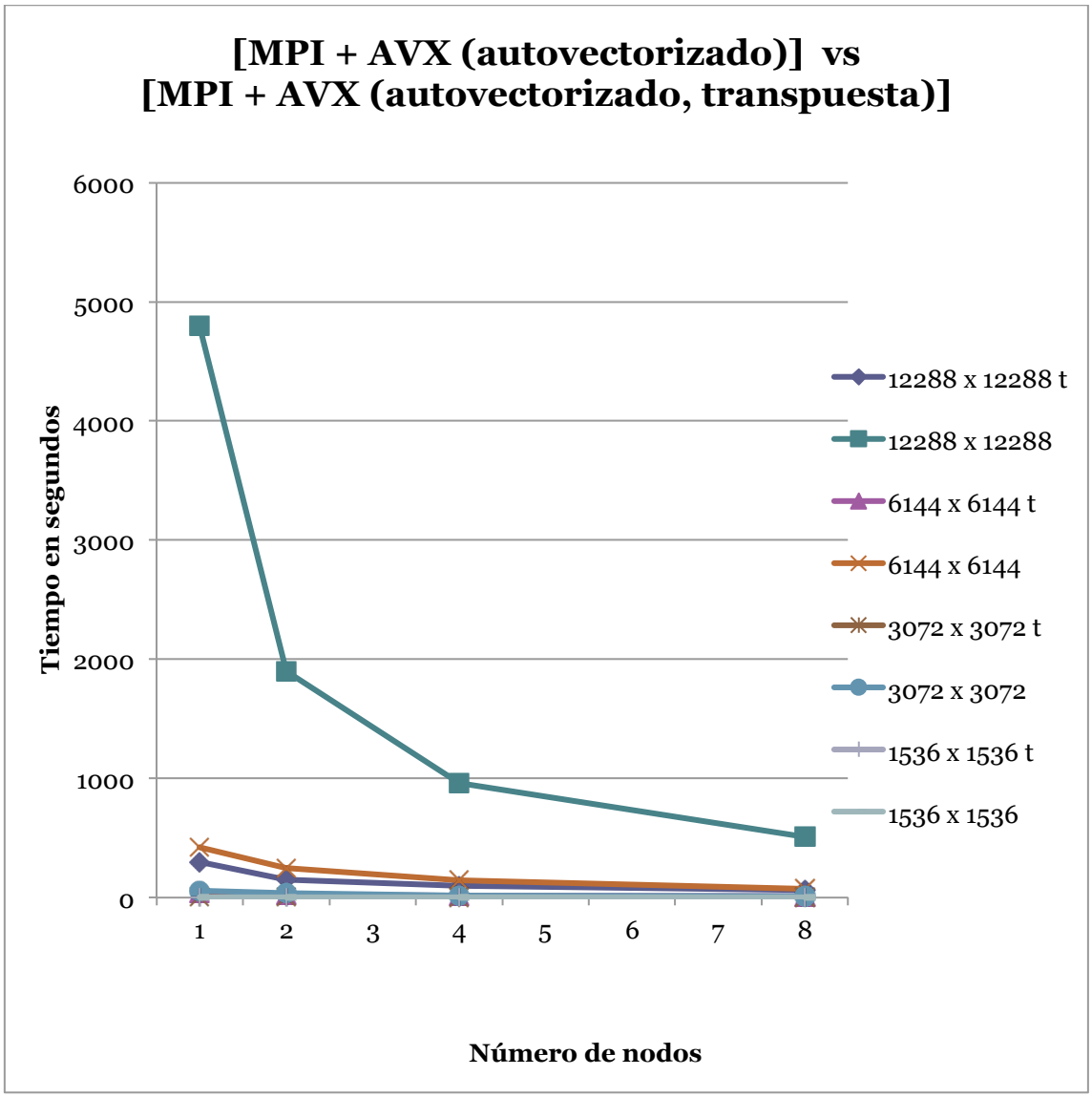

Ilustración 33: Comparativa de tiempos de ejecución de la versión en MPI + AVX (autovectorizada) y de la MPI + AVX (autovectorizada, transpuesta).

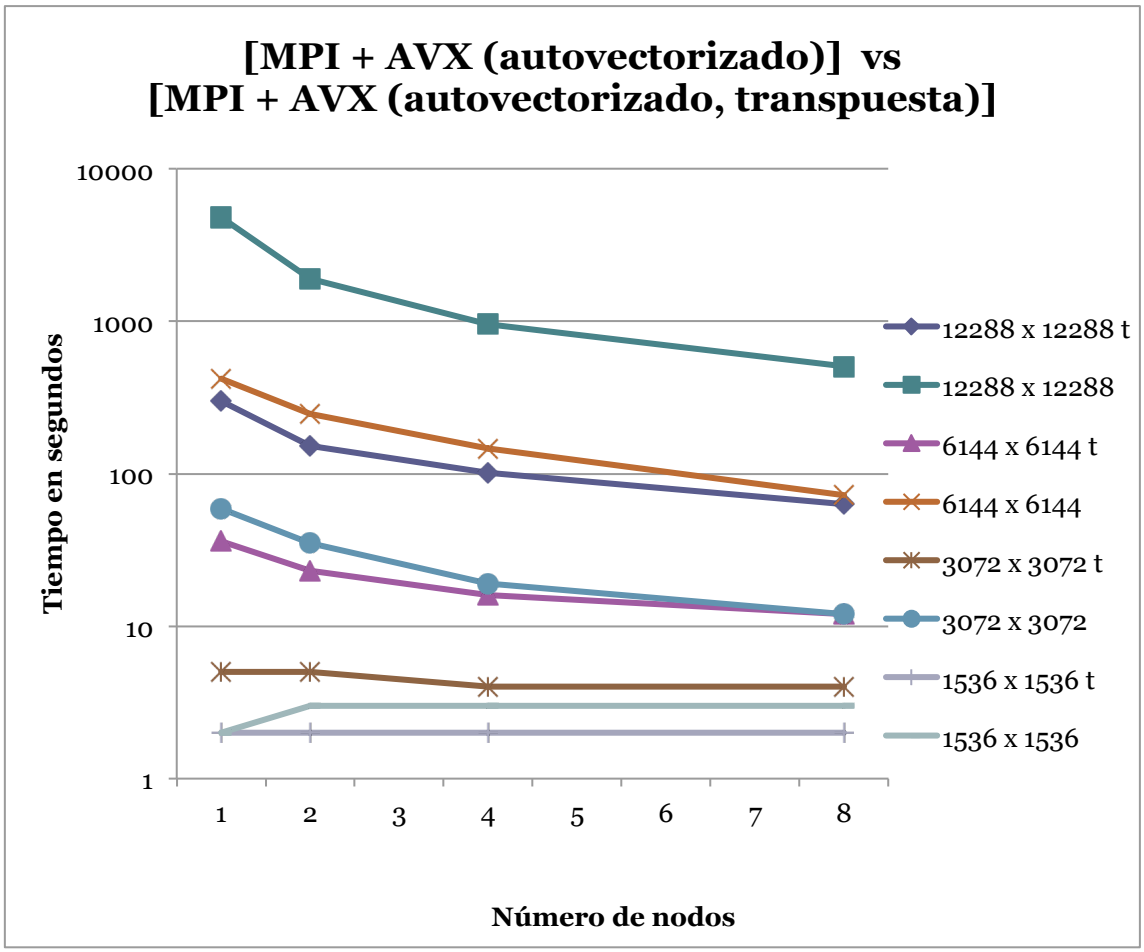

Ilustración 34: Comparativa de tiempos de ejecución de la versión en MPI + AVX (autovectorizada) y de la MPI + AVX (autovectorizada, transpuesta) (eje Y en escala logarítmica en base 10).

Las ilustraciones 33 y 34 muestran que una vez más obtenemos grandes mejoras al transponer la matriz B, en este caso la mejora entre las versiones autovectorizadas normal y transpuesta llega a ser del 1500%. El autovectorizador es mucho más eficiente cuando le facilitamos los datos de una forma óptima, como ya ocurría en las versiones para CPU.

3.8 Recopilación.

A continuación mostramos una gráfica con la recopilación de resultados de las cinco versiones vistas hasta ahora para un tamaño de matriz de 12288x12288 haciendo en todos los casos la transposición de la matriz B antes del cálculo:

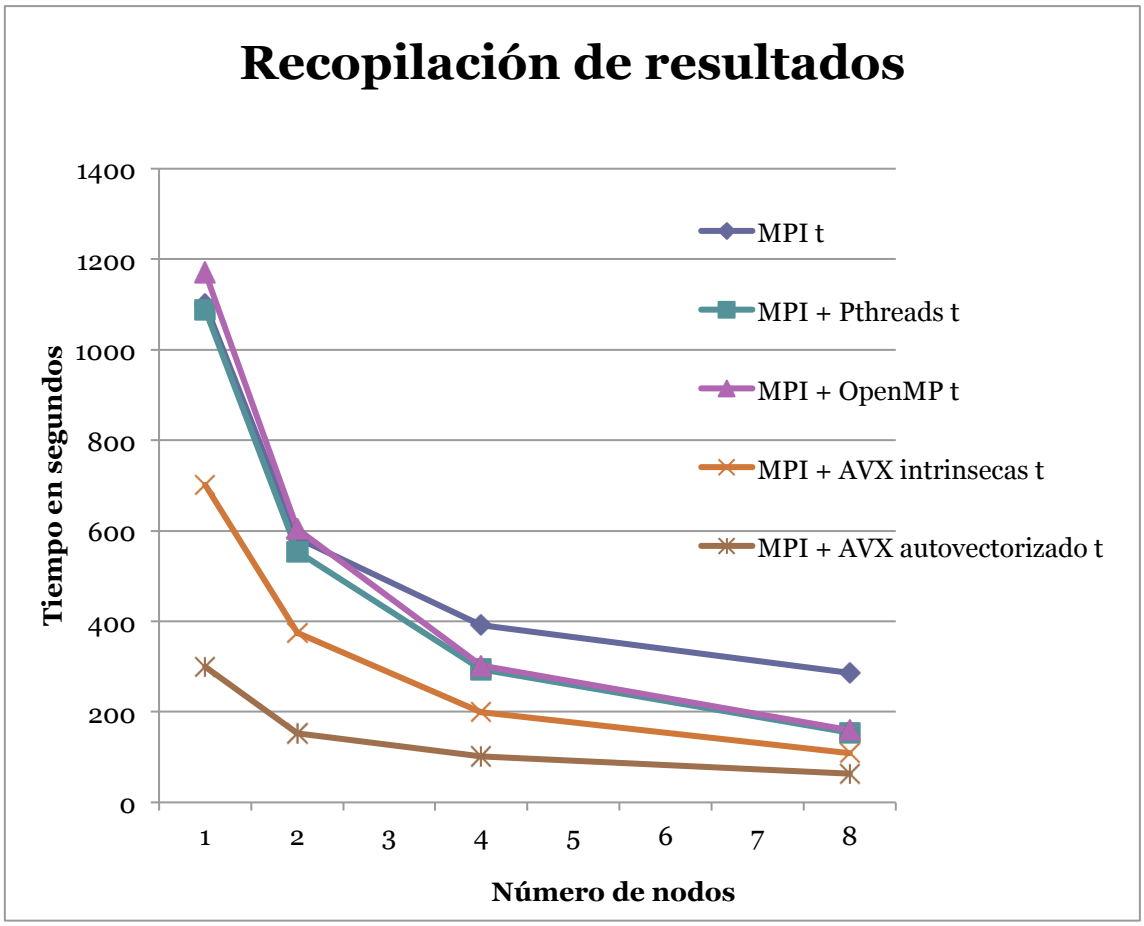

Ilustración 35: Comparativa de tiempos de ejecución para un tamaño de matriz de 12288x12288 y transponiendo la matriz B.

En la ilustración 35 se puede apreciar que hemos obtenido los mejores al utilizar el coprocesador, siendo los mejores resultados los de la versión autovectorizada. Si comparamos la versión MPI (transpuesta) para procesador con la autovectorizada la mejora llega a ser del 300%. Esto es gracias al alto grado de optimización que se realiza al compilar el código. También hemos de tener en cuenta en este caso que al no realizar ninguna modificación al código seguimos manteniendo el mismo nivel de portabilidad que teníamos con la versión para procesador.

3.9 GPU.

Nuestro último test también está centrado en el aprovechamiento de los distintos *hardwares* disponibles. En este caso hemos estudiado los rendimientos obtenidos si trasladamos el cómputo a las tarjetas gráficas. El cluster sobre el que trabajamos dispone de tarjetas gráficas de la marca Nvidia y modelo Tesla K20 con 1.17 Tflops de operaciones en coma flotante de doble precisión, 5 GB de memoria y 2496 cores. Nvidia nos facilita el lenguaje de programación CUDA.

Programar en este lenguaje requiere cierta destreza ya que se ha de tener en cuenta el modelo de tarjeta del que se dispone, los diferentes niveles de memoria con los que trabaja, se ha de gestionar el flujo de información entre la CPU y la GPU, etc. Pero CUDA nos ofrece la librería CUBLAS que implementa el estándar BLAS para el cálculo matricial.

En el cluster disponemos de la versión 5.5 de la librería CUBLAS y es con la que hemos realizado nuestra implementación. Teniendo en cuenta que cada nodo dispone de una única tarjeta gráfica solo necesitamos crear un único proceso MPI por nodo ya que no obtendríamos ningún beneficio creando una cola de trabajo.

Código simplificado:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <cublas_v2.h>
#include <cuda_runtime.h>

#define SIZE 12288

void fill_matrix(double* x) {
	long i, j;
	for (i = 0; i < SIZE; ++i) {
		for (j = 0; j < SIZE; ++j) {
			x[i * SIZE + j] = (double)(rand() % 100);
		}
	}
}

void transpose_matrix(double* m){
	long i, j;
	double tmp;
	for (i=0; i<SIZE; i++) {
		for (j=i; j<SIZE; j++){
			tmp = m[i * SIZE + j];
			m[i * SIZE + j] = m[j * SIZE + i];
			m[j * SIZE + i] = tmp;
		}
	}
}

int main(int argc, char *argv[]){
	int myrank, P, from, to, i, j, k;
	double alpha = 1.0;
	double beta = 0.0;
	double *A, *B, *C, *A_local, *C_local;
	double *devA, *devB, *devC;
	MPI_Status status;

	cublasHandle_t handle;
	cublasStatus_t cublas_status;
	cublasPointerMode_t * mode;

	MPI_Init (&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
	MPI_Comm_size(MPI_COMM_WORLD, &P);

	from = myrank * SIZE/P;
	to = (myrank+1) * SIZE/P;
```

```
	A_local = malloc (SIZE * (SIZE/P) * sizeof(double));
	C_local = malloc (SIZE * (SIZE/P) * sizeof(double));
	B = malloc (SIZE * SIZE * sizeof(double));

	if (myrank==0) {
		A = malloc (SIZE * SIZE * sizeof(double));
		C = malloc (SIZE * SIZE * sizeof(double));
		fill_matrix(A);
		fill_matrix(B);
	}

	MPI_Bcast (B, SIZE*SIZE, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Scatter (A, SIZE*SIZE/P, MPI_DOUBLE, A_local, SIZE*SIZE/P, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

	// Cada proceso reserva memoria en la GPU para su sección de la matriz A
	cudaMalloc( (void **)&devA, SIZE * (SIZE/P) * sizeof(double) );
	// Cada proceso reserva memoria en la GPU para la matriz B
	cudaMalloc( (void **)&devB, SIZE * SIZE * sizeof(double) );
	// Cada proceso reserva memoria en la GPU para la matriz resultado
	cudaMalloc( (void **)&devC, SIZE * (SIZE/P) * sizeof(double) );

	// Se copian las matrices en la GPU
	cudaMemcpy( devA, A_local, SIZE * (SIZE/P) * sizeof(double), cudaMemcpyHostToDevice);
	cudaMemcpy( devB, B, SIZE * SIZE * sizeof(double), cudaMemcpyHostToDevice);

	// Inicializamos un manejador que mantendrá el contexto de ejecución en la GPU y que iremos pasando en
	cada llamada que hagamos a funciones de la librería CUBLAS
	cublas_status = cublasCreate(&handle);
	// En la variable Cublas_status recibiremos los posibles códigos de error, hemos de comprobar que cada
	función se ejecuta correctamente
	if(cublas_status != CUBLAS_STATUS_SUCCESS){
		printf("handle create fail\n"); return 1;
	}

	// Realizamos la multiplicación matricial en la GPU indicandole cuales son los vectores y las dimensiones
	de estos
	cublas_status = cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, SIZE/P, SIZE, SIZE, &alpha,
devA, SIZE, devB, SIZE, &beta, devC, SIZE/P );
	if(cublas_status == CUBLAS_STATUS_SUCCESS){
		printf("cublasDgemm OK\n");
	}else{
		printf("cublasDgemm BAD");
	}

	// Extraemos de la GPU a memoria principal el vector resultado
	cudaMemcpy( C_local, devC,  SIZE * (SIZE/P) * sizeof(double),     cudaMemcpyDeviceToHost);

	// Liberamos el espacio reservado en la GPU
	cudaFree(devA);
	cudaFree(devB);
	cudaFree(devC);

	MPI_Gather (C_local, SIZE*SIZE/P, MPI_DOUBLE, C, SIZE*SIZE/P, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

	// Destruimos el manejador
	cublas_status = cublasDestroy(handle);
	 if(cublas_status != CUBLAS_STATUS_SUCCESS){
		printf("cublasDestroy failed\n"); return 1;
	}

	// Hemos de transponer la matriz resultado porqueCUBLAS trabaja con colum-major-order por cuestiones
	de rendimiento y nosotros necesitamos la respuesta en row-major-order
	if (myrank==0) {
		transpose_matrix(C);
```

```
        }
        MPI_Finalize();
        return 0;
}
```

Observando el código vemos que una vez que cada nodo dispone de las matrices que necesita, ha de copiarlas en la memoria de la GPU. Para ello primero reserva espacio en dicha memoria con la orden *cudaMalloc* y después copia el contenido de las matrices con *cudaMemcpy*. Posteriormente se realiza la multiplicación matricial y se copia la matriz resultado desde la GPU a la memoria principal. Finalmente todos los procesos envían sus matrices resultado parciales al proceso *master* el cual ha de transponer la matriz resultado para poder presentar los datos correctamente.

Resultados:

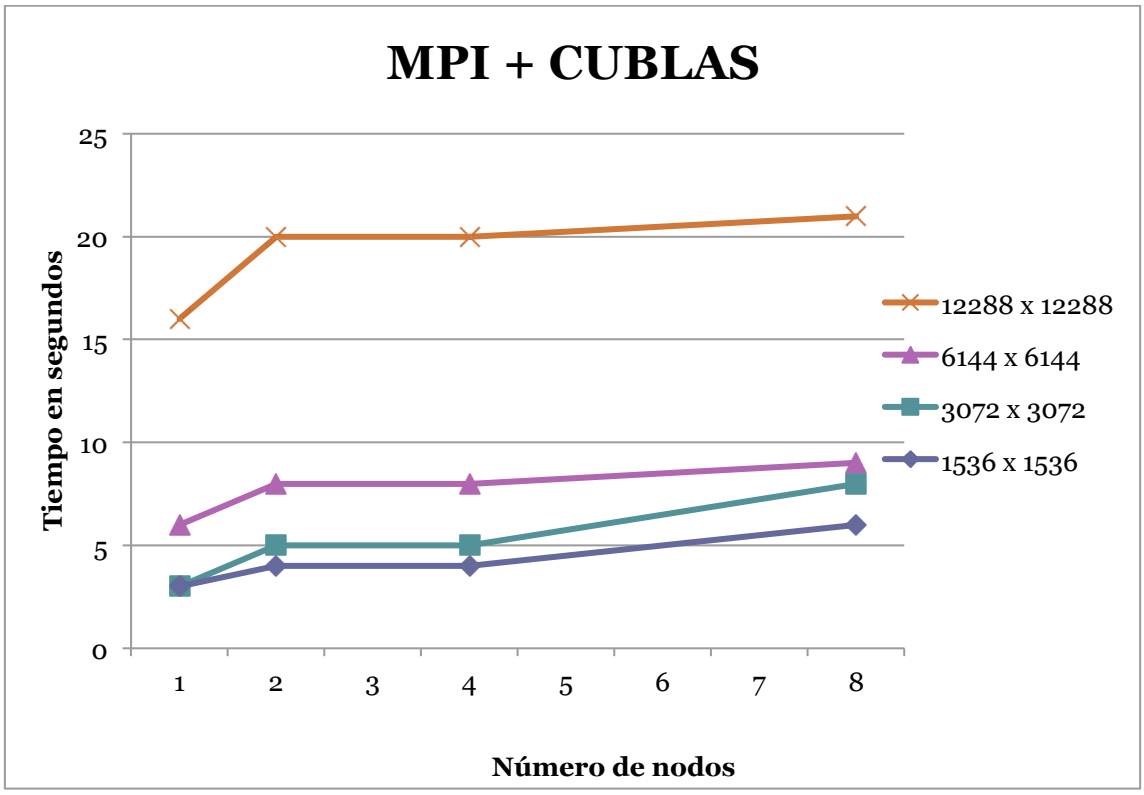

Ilustración 36: Comparativa de tiempos de ejecución de la versión MPI + CUBLAS.

La ilustración 36 muestra unos resultados espectaculares. Vemos que ninguna de las cargas de trabajo que hemos utilizado son lo suficientemente grandes como para que nos merezca la pena implicar más de un nodo en la ejecución. De hecho obtenemos peores tiempos de ejecución debido a la sobrecarga de comunicaciones.

3.10 Recopilación final.

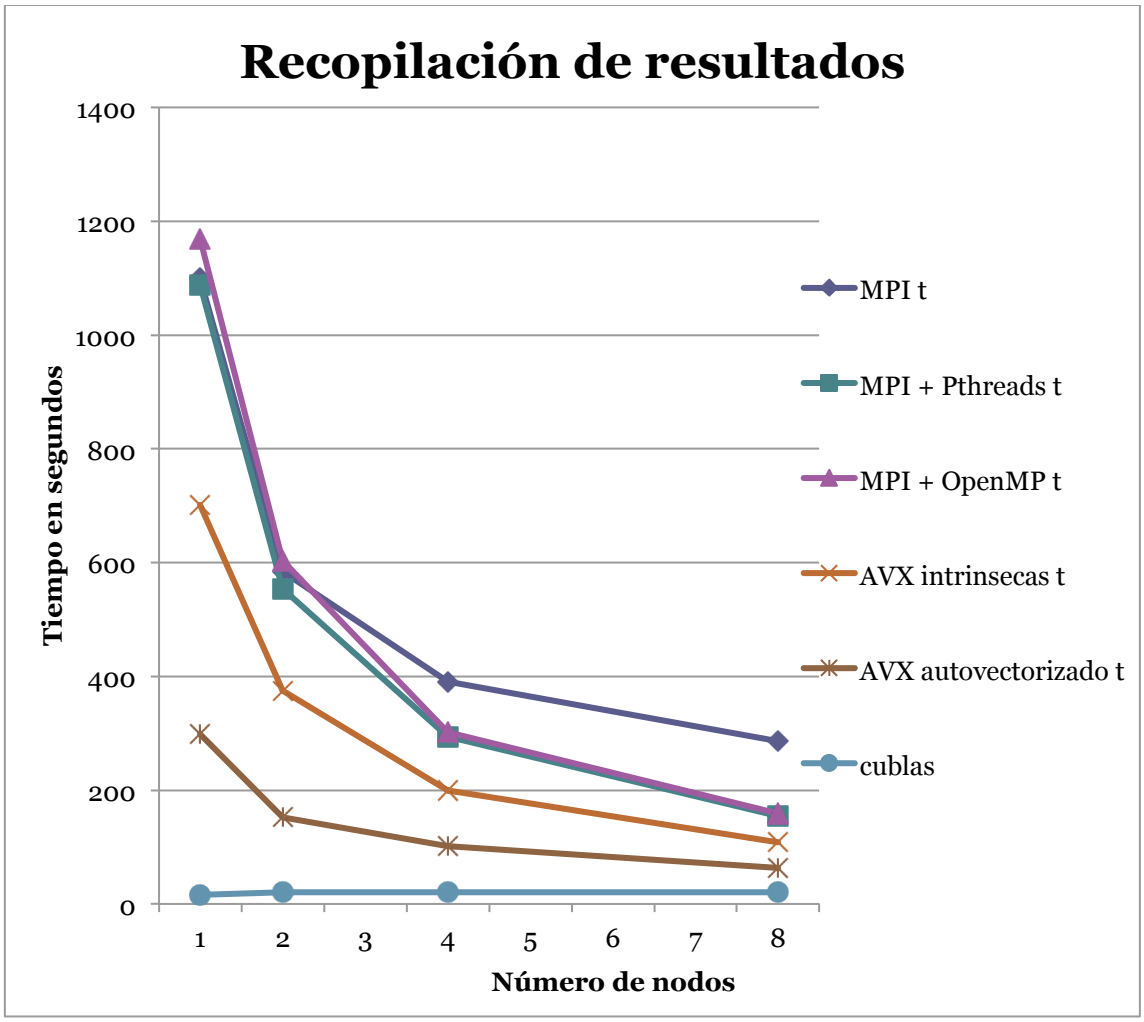

Ilustración 37 : Comparativa de tiempos de ejecución para un tamaño de matriz de 12288x12288 y transponiendo la matriz B.

La ilustración 37 muestra que CUBLAS resulta hasta 17 veces más rápido que la versión autovectorizada, la cual era la más eficiente hasta ahora. Como se puede observar en el código fuente, CUBLAS es una librería de alto nivel que nos abstrae en gran medida de CUDA, permitiendo una implementación bastante sencilla. A partir de la versión 6.0 de CUBLAS se vuelve incluso más fácil la programación ya que no requiere la gestión manual de la subida y bajada de bloques de memoria a la GPU sino que le podemos pasar la referencia de las matrices para que el sistema se encargue del resto. Además se incluye la posibilidad de trabajar con varias tarjetas sobre el mismo nodo.

4. Conclusiones.

En este Trabajo Fin de Grado el alumno ha desarrollado diversas versiones de programas paralelos y/o distribuidos para la realización de la multiplicación de matrices. El objetivo principal de este trabajo era que el alumno ampliara, de forma práctica, los conocimientos adquiridos durante la carrera relacionados con la programación paralela y distribuida. Por este motivo, el énfasis de este trabajo ha sido la resolución de un sencillo problema en diferentes entornos de programación sin preocuparnos de la dificultad del propio problema. Lo importante era afrontarlo desde distintos entornos. Las conclusiones de este trabajo se pueden dividir en tres grandes grupos:

1- Es importante la exploración de los diferentes entornos disponibles a la hora de desarrollar software. Nosotros hemos profundizado en diversos aspectos que se deben tener en cuenta como son: el aprovechamiento de los distintos tipos de *hardwares* disponibles, el uso de un modelo de memoria compartida o distribuida y el desarrollo de software intentando reducir los costes de acceso a memoria. Todo ello nos ha proporcionado una visión más amplia del contexto de la computación paralela y distribuida.

2- El trabajar en un *cluster* en funcionamiento ha requerido de una adaptación al mismo. Hemos tenido que aprender muchas cosas que se han de tener en cuenta a la hora de lanzar un programa en él. Según qué tecnología software se quiera utilizar, puede requerir previamente la configuración de múltiples parámetros por parte del usuario y si además nuestros programas utilizan una composición de estas, la dificultad aumenta considerablemente. Dado que este proceso de adaptación nos ha ocupado buena parte del esfuerzo realizado en este trabajo, hemos decidido añadir una breve guía de usuario para este cluster incluyendo las distintas configuraciones necesarias que hemos tenido que realizar. Dicha guía podrá ser consultada por futuros alumnos que se encuentren con los mismos problemas que nosotros.

3- Conclusiones sobre el rendimiento:
   a. Repartir la carga de trabajo sobre más nodos no asegura un incremento del rendimiento ya que los costes de comunicación pueden llegar a ser mayores que el beneficio obtenido por la distribución del trabajo.

b. Una correcta gestión intranodo de memoria compartida reduce considerablemente los costes de comunicación, con lo que el número máximo de nodos sobre los que podemos distribuir la carga, obteniendo mejoras de rendimiento, es mayor que en la versión puramente distribuida.
c. Las versiones de memoria compartida intranodo acaban obteniendo mejores resultados que las versiones totalmente distribuidas cuando aumentamos lo suficiente la cantidad de nodos sobre los que repartimos la carga.
d. Disponer los datos en memoria de tal manera que se maximice la reutilización de bloques que ya están en la memoria cache del procesador incrementa muy significativamente el rendimiento de los programas. Esta mejora de rendimiento todavía es más notable en un entorno de memoria compartida intranodo.
e. Es posible obtener mejoras de rendimiento realizando el cómputo sobre el coprocesador vectorial si la naturaleza de nuestro programa nos lo permite.
f. En el caso de programar con los lenguajes C o C++ y utilizando GCC como compilador, podemos desarrollar nuestro algoritmo para el cómputo en el coprocesador vectorial mediante el juego de funciones intrínsecas que trae dicho compilador. Pero esta forma es más difícil de implementar, menos portable y probablemente menos eficiente que si utilizamos el autovectorizador del que dispone el compilador.
g. En el supuesto de que nuestro software realice grandes cálculos matriciales el uso de las tarjetas gráficas marca la diferencia, son sin duda mucho más eficientes para este tipo de cálculo que los procesadores y coprocesadores vectoriales.
h. Si trabajamos con tarjetas gráficas Nvidia podemos aprovecharnos de la librería CUBLAS que nos abstrae en gran medida de programación en CUDA.

5. Bibliografía.

- **Pthreads**
  Autor: Blaise Barney, Lawrence Livermore National Laboratory
  https://computing.llnl.gov/tutorials/pthreads/
- **Intel Intrinsics Guide**
  https://software.intel.com/sites/landingpage/IntrinsicsGuide/
- **Guia para trabajar con coprocesadores vectoriales AVX**
  Autor: Matt Scarpino.
  Publicado: 20 febrero de 2015.
  http://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX
- **An Introduction to GCC Compiler Intrinsics in Vector Processing**
  Autores: George Koharchik and Kathy Jones.
  Publicado: 21 septiembre de 2012.
  http://www.linuxjournal.com/content/introduction-gcc-compiler-intrinsics-vector-processing?page=0,0
- **Auto-vectorization with gcc 4.7**
  http://locklessinc.com/articles/vectorize/
- **SGEMM Tutorial**
  http://keeneland.gatech.edu/software/sgemm_tutorial
- **CUBLAS GUIDE**
  Última revisión: 5 de marzo de 2015.
  http://docs.nvidia.com/cuda/cublas/index.html#axzz3XmUq51mU
- **CUDA C Programming Guide**
  Última actualización: 5 de marzo de 2015.
  http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz3FwllVxqd
- **MVAPICH2 2.0 User Guide**
  Última revisión: 10 de octubre de 2014.
  http://mvapich.cse.ohio-state.edu/static/media/mvapich/mvapich2-2.0-userguide.html
- **MPICH documentation guides**
  http://www.mpich.org/documentation/guides/
- **Tutorial material on MPI**
  Institución: Argonne National Laboratory
  http://www.mcs.anl.gov/research/projects/mpi/tutorial/
- **Mixing MPI and OpenMP**
  http://www.slac.stanford.edu/comp/unix/farm/mpi_and_openmp.html
- **Running MPI applications on Linux over Infiniband cluster with mvapich/ mvapich2**
  Autora: Y. Joanna Wong
  Publicado: febrero de 2010.
  http://chpc.wustl.edu/assets/files/pdf/WashU_7_mvapich.pdf
- **Basic Linear Algebra Subprograms**
  Última modificación: 3 de junio de 2015.
  https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms

Anexo: Guía de usuario para el cluster DISCA

Guía rápida de uso del Cluster de la ETSINF

Autor: Enrique Gil Arcas

1 CONEXIÓN

En este apartado se describe cómo podemos conectarnos al Cluster del grupo de Arquitecturas Paralelas de la Universidad Politécnica de Valencia.

1.1 Conexión VPN

Por mtivos de seguridad, el cluster del Grupo de Arquitectiras Paralelas únicamente acepta conexiones SSH desde direcciones IP de la UPV. Por este motivo, lo primero que necesitamos para acceder al cluster desde fuera de la Universidad es conectarnos por VPN a la misma. En el siguiente link disponemos de la explicación paso a paso https://www.upv.es/contenidos/miw/infoweb/infoacceso/dat/697481normalc.html.

Si estamos trabajando desde los ordenadores de la Universidad podemos saltarnos este paso.

1.2 Conexión SSH

Una vez conectados por VPN necesitamos abrir una sesión SSH hasta el cluster. Para ello si estamos en LINUX abrimos un terminal y tecleamos:

```
$ ssh –p PUERTO pepito@cluster.gap.upv.es
```

Donde “pepito” es el nombre de usuario que te dio la Universidad al matricularte. Nótese que por motivos de seguridad se ha cambiado el nombre real del cluster y se usa el nombre simbolico “cluster”.

A continuación os solicitará la contraseña que os ha facilitado el administrador del cluster. Si es la primera vez que te conectas primero te mostrará un mensaje por si quieres añadir la dirección a tu lista de hosts conocidos, le decimos que sí, después posiblemente te permita configurar una nueva contraseña.

Si por el contrario trabajamos desde Windows os recomiendo utilizar MOBAXTERM para crear la sesión (http://mobaxterm.mobatek.net/download.html).

[OPCIONAL:]
Si vais a conectaros al cluster recurrentemente os recomiendo generar una clave SSH y registrarla para acceder sin contraseña, el proceso se hace desde tu ordenador personal justo después de conectarte por VPN y seria:

```
$ ssh-keygen
```

Pulsamos “Enter” repetidas veces hasta que el proceso termine.

```
$ ssh-copy-id –p PUERTO pepito@cluster.gap.upv.es
```

Listo, ahora cuando conectemos por SSH ya no nos solicitara la contraseña.

1.3 Dentro del Frontend

En este momento, si todo ha ido bien, estarás logueado en el Frontend del cluster. Verás una lista de usuarios (entre los que estas tú) con los nodos que tienen disponibles para su uso.

La primera comprobación que debes hacer es que tienes un HOME para tu usuario. Para ello ejecuta:

```
$ ls /nfs/alumnos/
```

Comprueba que hay una carpeta con tu nombre. En caso contrario avisa al administrador.

Recuerda que no debes ejecutar programas en el Frontend, sino que has de hacerlo dentro de los nodos que se te han asignado. Para ello has de conectarte por SSH desde el Frontend a un nodo. Por ejemplo, si tenemos disponible el nodo "nodo7" nos conectamos a él mediante:

```
$ ssh pepito@nodo7
```

Como ya sabéis, si es la primera vez que nos conectamos nos preguntará si queremos añadir la dirección a la lista de hosts. Le decimos que sí.

Es importante que tengamos claro que el cluster usa una imagen de disco compartida. Ello hace que cualquier archivo que tengamos en nuestro HOME pueda ser accedido desde cualquier nodo del cluster, así que no es necesario copiar los archivos ejecutables en cada nodo para ejecutarlos de forma distribuida.

Por defecto la variable PATH (que contiene las rutas a directorios con archivos ejecutables) viene muy poco preparada, con lo que seguramente de primeras no podrás ejecutar programas, como por ejemplo el compilador de MPI (mpicc), hasta que añadas su ruta al PATH. En este caso sería:

```
$ export PATH=$PATH:/usr/mpi/gcc/mvapich2-2.0/bin/
```

Ten en cuenta que este comando no es permanente y has de configurar el PATH cada vez que inicies sesión. Para hacerlo permanente tendríamos que configurar el archivo ".bashrc" pero no tenemos privilegios de administrador para hacerlo.

Para concluir esta sección os dejo un comando con el que podemos subir archivos desde nuestro ordenador personal al cluster vía SSH. Para ello desde un terminal en nuestro ordenador personal (sin estar conectados por SSH con el cluster) y situados en el directorio en el que están los archivos que queremos transferir:

```
$ scp -P 3322 archivo.txt pepito@cluster.gap.upv.es:
```

2 TRABAJANDO CON MPI

Lo primero, como ya se ha mencionado antes, es añadir la ruta del directorio "bin" de MPI a la variable PATH.

```
$ export PATH=$PATH:/usr/mpi/gcc/mvapich2-2.0/bin/
```

Ahora tenemos disponibles los comandos típicos que necesitamos para trabajar con MPI. Para compilar un archivo programado en C para su ejecución en MPI utilizaremos "mpicc", por ejemplo:

```
$ mpicc –o miApp miApp.c
```

Antes de ejecutar nada se recomienda crear un "hostfile", que no es más que un archivo de texto en el que escribimos, separados por líneas, el nombre de todos los nodos en los que queremos que se ejecute el algoritmo junto con el número máximo de procesos que permitimos que se ejecuten simultáneamente en cada nodo. Por ejemplo, tenemos un archivo con el nombre "hf_mpi" que contiene:

```
nodo1:8
nodo2:8
```

Este archivo indicaría que nuestro programa se va a ejecutar en los nodos "nodo1" y "nodo2" y que, como mucho, se van a ejecutar simultáneamente 8 instancias del programa en cada nodo. Normalmente el número máximo de instancias suele ser el mismo que el número de cores que tiene disponibles cada nodo. Para ejecutar el algoritmo anterior utilizaríamos el comando:

```
$ mpiexec –hostfile hf_mpi –np 16 miApp
```

Como vemos, le hemos pasado el "hostfile" con el prefijo "-hostfile" y también le hemos indicado el número total de procesos (16) con el prefijo "-p". Mientras el algoritmo se esté ejecutando podremos comprobar que todo funciona correctamente accediendo a cada nodo involucrado en la ejecución y utilizando la orden:

```
$ top
```

Deberíamos ver, como mucho, tantas instancias del programa como el número máximo de procesos que le asignamos previamente en el "hostfile" a dicho nodo.

3 HÍBRIDOS DE MEMORIA COMPARTIDA

Si utilizamos MPI en solitario estaremos ejecutando nuestro programa en un entorno de memoria distribuida tanta inter como intra nodo ya que se crean instancias independientes del mismo que se distribuyen entre todos los cores disponibles del cluster.

Pero quizás queramos aprovecharnos de las ventajas de ejecutar nuestros programas en un entorno de memoria compartida intra nodo, es decir, creando una única instancia de ejecución en cada nodo y esa instancia generará tantos hilos como cores tenga disponibles el nodo, con lo que optimizaríamos el uso de memoria. Podemos conseguir esto combinando, por ejemplo, MPI con OMP o MPI con Pthreads.

Importante: Si vamos a optar por usar una de estas combinaciones debemos tener en cuenta que la versión de MVAPICH de que disponemos en el Cluster, por defecto, tiene configurada la afinidad de asignación de procesos a "1". Ello conlleva que todos los hilos que se ejecuten en un nodo lo hagan sobre el mismo core, lo cual sería catastrófico para nuestro propósito de distribuir la carga los más eficientemente posible. Para solucionar esto hemos de añadir la variable de entorno "MV2_ENABLE_AFFINITY=0" en el comando de ejecución de MPI (más información en pg 16 -> http://chpc.wustl.edu/assets/files/pdf/WashU_7_mvapich.pdf):

```
$ mpiexec -env MV2_ENABLE_AFFINITY 0  -np 2  --hostfile hf_mpi  miApp
```

Para detectar si se está distribuyendo bien la carga entre los cores podemos ayudarnos otra vez de la orden "top"; si por ejemplo estamos en un nodo con 8 cores y lo hemos preparado todo para que se ejecuten 8 hilos en dicho nodo, deberíamos ver que existe una única instancia del programa pero que tiene un porcentaje de consumo de CPU cercano al 800%. Si por el contrario, el porcentaje se mantiene entorno al 100%, sabremos que MPI está serializando la ejecución de todos los hilos en un mismo core.

Por otro lado, la forma de inicializar MPI en un programa híbrido es distinta que al utilizar MPI en solitario. Cuando estamos trabajando con hilos la forma de inicializar correctamente sería:

```
MPI_Init_thread (&argc, &argv, MPI_THREAD_FUNNELED, &provided);
if (provided < MPI_THREAD_FUNNELED){
    printf("request: %d, provided: %d",MPI_THREAD_FUNNELED, provided );
    MPI_Abort(MPI_COMM_WORLD, 1);
 }
```

Donde con "MPI_THREAD_FUNNELED" le indicamos a MPI el nivel mínimo de "thread safety level" con el que queremos trabajar, en este caso la variable indica nivel 1 (más información en pg 17 -> http://www.mcs.anl.gov/~balaji/talks/tutorials/2013/2013-06-16-isc-mpi.pptx) y en la variable "provided" MPI nos devuelve el nivel efectivo con el que se va a ejecutar.

Importante: Si el nivel que nos devuelve "provided" es inferior al nivel que requerimos y el programa aborta puede ser debido a que no hemos utilizado la variable de entorno "MV2_ENABLE_AFFINITY=0" a la hora de ejecutar y se están serializando los procesos con lo que MPI no nos puede dar el nivel de seguridad que necesitamos.

Por último, hemos de tener en cuenta que el archivo "hostfile" ha de indicar que en cada nodo solo se puede ejecutar un proceso, así el contenido de nuestro archivo "hf_mpi" seria por ejemplo:

```
nodo1:1
nodo2:1
```

3.1 MPI + OMP

Podemos compilar un programa híbrido de MPI y OMP añadiendo el *flag* "-fopenmp":

```
$ mpicc -fopenmp -o mpi_hello_world mpi_hello_world.c
```

Además, hemos de introducir **en todos los nodos** la variable de entorno "OMP_NUM_THREADS=X" donde X es el número de hilos que queremos que se creen dentro de cada nodo en particular. Por ejemplo, para indicar que queremos que se ejecuten 8 hilos en el nodo "nodo1" deberíamos entrar a dicho nodo vía SSH y escribir:

```
$ export OMP_NUM_THREADS=8
```

Si detectamos que los hilos se ejecutan serializados en un core pese a haber configurado la afinidad de MPI para que los hilos se distribuyan entre todos los cores, podremos ayudarnos de la variable de entorno específica para OMP "GOMP_CPU_AFFINITY=X-Y", donde X e Y son los índices de los cores que indican el rango entre los cuales se van a distribuir los hilos. Por ejemplo, para indicar que queremos que se distribuyan entre los cores del 0 al 7 escribiríamos **en cada nodo**:

```
$ export GOMP_CPU_AFFINITY=0-7
```

Finalmente para ejecutar nuestro algoritmo lo haríamos de la forma usual para programas híbridos, indicando con "-np" que queremos tantos procesos como nodos de que disponemos:

```
$ mpiexec -np 2 --hostfile hf_mpi -env MV2_ENABLE_AFFINITY 0 mpi_hello_world
```

3.2 MPI + PTHREADS

En este caso no tenemos que tener tantas consideraciones como con OMP. Podemos compilar normalmente y ejecutar el programa tal como indicamos en el principio del apartado de programas híbridos.

```
$ mpicc  -o mpi_hello_world mpi_hello_world.c
$ mpiexec -np 2 --hostfile hf_mpi -env MV2_ENABLE_AFFINITY 0 mpi_hello_world
```

4 MPI + CUDA

Para realizar pesados cálculos matriciales puede resultar muy interesante utilizar MPI para distribuir la carga entre todos los nodos y CUDA para hacer los cálculos sobre las GPU de cada nodo.

Para que en tiempo de ejecución sea visible la librería CUBLAS en todos los nodos es necesario exportar la variable de entorno LD_LIBRARY_PATH con la ruta a dicha librería:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH: /usr/local/cuda-5.5/lib64
```

Si vas a utilizar el compilador nvcc debes añadir también su ruta a la variable PATH:

```
$ export PATH=$PATH:/usr/local/cuda-5.5/bin/
```

Si por el contrario vas a compilar utilizando mpicc has de indicar los siguientes flags y rutas:

```
$ mpicc src/mpi_cublas.c -lcublas -lcudart_static  -I /usr/local/cuda-5.5/include -L /usr/local/cuda-5.5/lib64 -std=c99 -o bin/myCublasApp
```

5 MPI + COPROCESADOR VECTORIAL

Mejorar el tiempo de ejecución de nuestros programas siempre es un objetivo importante y más aún si estamos trabajando sobre un Cluster dado que estos suelen estar muy solicitados y usualmente tenemos un tiempo limitado de uso del mismo. Una buena práctica de optimización es aprovechar los coprocesadores vectoriales que tienen las CPU ejecutar cómputos en paralelo. Con ello podemos obtener *speed –ups* muy interesantes.

Para aprovecharnos de la velocidad del coprocesador tenemos dos opciones. La primera es implementar nosotros las operaciones vectoriales en nuestro código, lo cual no es trivial de hacer, conseguiríamos una mejora de rendimiento que pagaríamos con la complejidad de la implementación. La segunda es aprovecharnos del autovectorizador del que disponen las versiones modernas del compilador "gcc". Simplemente añadiendo el *flag* "-O3" el compilador intentara vectorizar todas las operaciones y bucles que sea capaz. Por ejemplo:

```
$ mpicc  -o mpi_hello_world mpi_hello_world.c –O3
```

La ejecución es completamente igual que la que haríamos si no utilizáramos el coprocesador.

```
$ mpiexec -np 2 --hostfile hf_mpi -env MV2_ENABLE_AFFINITY 0 mpi_hello_world
```

Además, podemos añadir directivas en nuestro código que faciliten al autovectorizador su trabajo de optimización y poder sacar asi mejores rendimientos (mas info en -> http://locklessinc.com/articles/vectorize/).

Si has optado por hacer una implementación mediante funciones intrínsecas de GCC lo único que debes añadir al comando de compilación es el *flag* –mavx si estás trabajando en los nodos XEON o –msse2 si estás trabajando con los nodos ATOM (diferentes arquitecturas de procesador tienen diferentes juegos de instrucciones vectoriales):

```
$ mpicc  -o mpi_hello_world mpi_hello_world.c –mavx
```

```
$ mpicc  -o mpi_hello_world mpi_hello_world.c –msse2
```


6 LINKS DE INTERÉS

MPICH:

- http://www.mpich.org/documentation/guides/

MVAPICH:

- http://mvapich.cse.ohio-state.edu/userguide/
- http://chpc.wustl.edu/assets/files/pdf/WashU_7_mvapich.pdf

CUDA:

- http://docs.nvidia.com/cuda/index.html#axzz3Yu9xIJHA
- http://docs.nvidia.com/cuda/cublas/index.html#axzz3XmUq51mU

GCC:

- http://locklessinc.com/articles/vectorize/