PRÁCTICA PROFESIONAL SUPERVISADA

TEMA

Investigación de Framework Charm++ para Programación Paralela

LUGAR DE REALIZACIÓN:

Laboratorio de Computación - FCEFyN - UNC.

TUTOR:

Wolfmann, Gustavo.

INTEGRANTES:

Aguilar, Mauricio.

Tarazi, Pedro Esequiel

CARRERA:

Ingeniería en Computación.

Contenido

Tabla de Ilustraciones

Resumen

Este documento presenta una investigación del Framework Charm++, basado en C++, el cual permite codificar programas paralelos mediante la utilización de objetos. Estos objetos se comunican por medio de paso de mensajes asincrónicos, y son migrables gracias a un sistema de tiempo de ejecución adaptativo, el cual optimiza el rendimiento del programa.

Charm++ cuenta con balanceo de carga automático, permite portabilidad de código, interoperabilidad con MPI y OpenMP, entre otras ventajas, lo que hace de este framework un atractivo paradigma de programación paralela.

Se aplicó esta metodología para comparar el funcionamiento de un algoritmo de uso difundido con otras librerías de programación paralela conocidas, tales como OpenMP y MPI.

Introducción

La computación paralela es el uso simultáneo de múltiples recursos computacionales para resolver un problema. Se distingue de la computación secuencial en que varias operaciones pueden ocurrir simultáneamente. El clásico uso del paralelismo es el diseño de programas eficientes para resolver problemas científicos, los cuales requieren de una gran capacidad de procesamiento y de espacio de memoria, debido a las complejas operaciones que se deben realizar. Otro uso clásico es el de las gráficas generadas por computadoras, ya que la generación de fotogramas requiere de una gran cantidad de cálculos matemáticos, suponiendo una tarea muy compleja para un solo procesador.

En la actualidad, los fabricantes de hardware han dejado de gastar grandes cantidades de silicio en el CPU para algoritmos avanzados que mejoren las instrucciones pre-fetch y comenzaron a utilizar CPUs más pequeños y simples, logrando colocar más de uno en una misma oblea de silicio, obteniendo así lo que se conoce como CPU multi-core. Escribir un programa que optimice el uso de un CPU multi-core con librerías como OpenMP o MPI es mucho más complejo que escribir un programa secuencial, ya que requiere que haya comunicación y sincronización entre las tareas que se han paralelizado, así como una fuerte supeditación al hardware sobre el que se ejecuta. Esta dificultad nos llevó a pensar e investigar formas de aprovechar la programación paralela.

Con la intención de contar con una librería que ofrezca ventajas sobre este tipo de problemáticas e investigar el desempeño en relación a OpenMP y MPI, que son las más utilizadas en la actualidad, se llevó a cabo el estudio y análisis de un novedoso framework llamado Charm++. Se trata de un sistema de programación paralela basado en C++ fundado sobre un modelo de programación de objetos migrables con un poderoso sistema de tiempo de ejecución adaptativo, que ofrece una serie de ventajas como portabilidad, tolerancia a fallos, balanceo automático de carga, reusabilidad y modularidad.

Charm++ permite escribir un programa en C++ mediante paso de mensajes, sin embargo, el programador vé estos mensajes como simples llamadas a funciones. Es decir, Charm++ facilita la paralelización de un algoritmo, pero la pregunta clave es: ¿A qué costo? ¿Que debemos sacrificar para programar de manera más sencilla?

En el presente trabajo, se realizó el análisis comparativo de un algoritmo de uso difundido, el método de ordenamiento conocido como Merge Sort, usando el framework

Charm++ y OpenMP. A continuación, se hará una introducción teórica a la programación paralela en Charm++ y el algoritmo utilizado, para luego mostrar los resultados obtenidos.

Marco Teórico

¿Qué es Charm++?

Charm++ es un paradigma de programación paralela por paso de mensajes asíncronos orientado a objetos. ¿Que significa todo esto? Un *paradigma de programación* es una manera de escribir un programa, esto quiere decir que Charm++ NO es un lenguaje de programación en sí mismo. Usa C++ como su lenguaje base.

Por *paralelo* queremos decir que es usado para escribir aplicaciones que hacen uso de múltiples hilos en ejecución. Estos pueden ser varios hilos compartiendo tiempo en un único elemento de procesamiento, o varios hilos compartiendo muchos elementos de procesamiento. En términos de Charm++, el término "paralelo" simplemente implica que varias "cosas" pueden ocurrir en un momento dado.

Orientado a objetos significa lo mismo que en el lenguaje de programación C++. El programa se descompone en una colección lógica de objetos que interactúan unos con otros. En Charm++, hay objetos especiales llamados ***chares***. Cada objeto chare puede contener algún estado, enviar y recibir mensajes, y realizar alguna tarea en respuesta a un mensaje recibido.

Por *paso de mensajes asincrónicos* queremos decir que los objetos chare se comunican entre sí por envío de mensajes. La palabra clave asincrónico quiere decir que los mensajes son enviados de una manera que es asíncrona para la ejecución del código asociado con los objetos chare. Por ejemplo, un chare, luego de enviar un mensaje, continúa su ejecución mientras el mensaje viaja hacia el chare objetivo). Lo opuesto también ocurre: un objeto chare puede recibir un mensaje desde otro objeto chare en cualquier momento, independientemente de lo que el chare receptor esté haciendo en un momento en particular. Un punto importante de esto es que, desde la perspectiva del programador, el envío de un mensaje de un objeto chare a otro, es simplemente una llamada a una función miembro del chare objetivo.

¿Cómo funciona y cómo se compone un programa Charm++?

Desde la perspectiva del programador, un programa Charm++ es simplemente una colección de objetos chare. Cada objeto chare tiene algún estado asociado a él. Como se dijo anteriormente, los objetos chare se comunican entre sí por envío de mensajes. Cuando un objeto chare particular recibe un mensaje, éste ejecutará un método de entrada para

procesar el mensaje. Este método de entrada puede realizar uno o más operaciones o cálculos, puede enviar más mensajes a otros objetos chare, puede almacenar en un buffer el contenido de un mensaje para posterior procesamiento o puede no hacer nada en absoluto.

La ejecución comienza con un chare especial llamada ***main chare***.

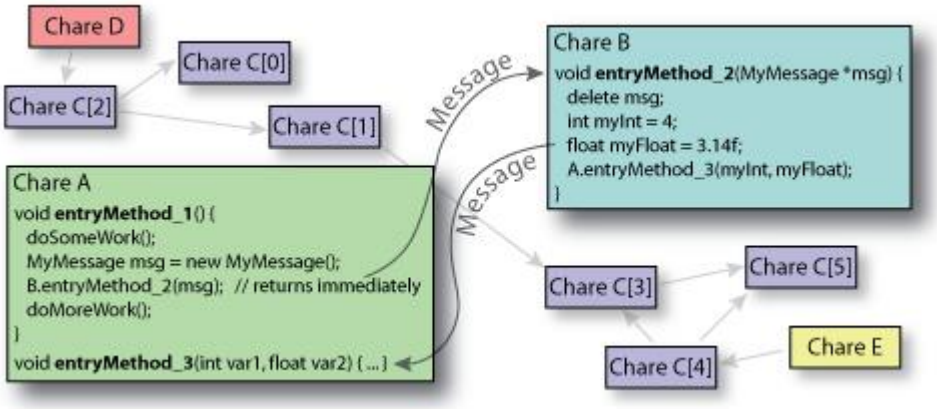

Figura 1 – Paso de Mensajes entre objetos Chare.

Las aplicaciones Charm++ son simplemente escritas en términos de los objetos chares. El programador no tiene que preocuparse acerca del número de elementos de procesamiento que estarán disponibles cuando el programa sea ejecutado o qué tipo de interconexión será usada para conectar los elementos de procesamiento. Todos estos detalles se dejan a un sistema en tiempo de ejecución inteligente de Charm++ llamado Charm++ Runtime System. El Charm++ RTS se encarga de asignar cada uno de los chares a los elementos de procesamiento físico, enrutar mensajes a través de la interconexión, etc.

Objeto Chare

Cada uno de los objetos chare en la aplicación Charm++ puede ser pensado como entidades inteligentes. Cada objeto chare tiene su propio estado (datos locales a él). Los objetos chare no pueden acceder directamente al estado de otros objetos chares, sin embargo, cualquier objeto chare puede comunicarse directamente con cualquier otro objeto chare a través de los métodos de entrada. El conjunto global de objetos chare es comúnmente denominado como “espacio global de objetos”. Cualquier objeto chare en el espacio global de objetos puede requerir información sobre el estado de otros objetos chare a través del envío de mensajes.

Mensajes

Los objetos chare se comunican entre sí por envío de mensajes. En Charm++, este proceso también es denominado como *invocación de método remoto*, ya que un objeto chare simplemente llama a uno de los métodos de entrada del objeto chare receptor.

Métodos de Entrada

Los métodos de entrada son funciones miembro especiales de la clase chare. La diferencia entre una función miembro normal C++ y un método de entrada Charm++ es que los métodos de entrada Charm++ actúan como puntos de recepción de los mensajes. Cuando un objeto chare hace una invocación a un método remoto sobre otro objeto chare, el dato a enviar se ha empaquetado en un mensaje y pasado al objeto chare receptor. Una vez que el objeto chare receptor recibe el mensaje, el método de entrada es invocado y la información en el mensaje es pasada a ese método de entrada.

Cada una de las clases chare también tiene un constructor definido como método de entrada (básicamente lo mismo que un constructor en C++). Cuando un objeto chare es creado, este método de entrada constructor es automáticamente invocado por el Charm++ Runtime System para crear el objeto chare. La ejecución de una aplicación Charm++ comienza con la ejecución del constructor del objeto chare main.

Hay dos aspectos de los métodos de entrada que los hace diferentes de las funciones miembro estándar de C++. Primero, los métodos de entrada no retornan valores. Segundo, desde la perspectiva de un objeto chare llamante, se retorna inmediatamente de la llamada a los métodos de entrada y el objeto chare llamante continúa su ejecución. Esto no significa que el método de entrada objetivo ha sido ejecutado, sino que un mensaje a sido enviado al objeto chare el cual causará que el método de entrada objetivo se ejecute en algún momento en el futuro. Estos dos aspectos de los métodos de entrada están directamente relacionados al natural paso de mensajes asíncrono de Charm++.

Proxies

Debido a que los objetos chare actuales en el espacio global de objetos en un programa Charm++ se distribuyen a través de varios elementos de procesamiento, no siempre es posible que dos objetos chare se comuniquen entre sí. En su lugar, para que un objeto chare invoque un método de entrada sobre otro objeto chare, el objeto chare emisor debe primero tener una referencia llamada proxy al objeto chare objetivo. Los

objetos proxy ocultan los detalles de la comunicación actual para el programador. Los métodos de entrada son llamados sobre el objeto proxy como si el objeto chare en sí fuera local al procesador físico actual. El Charm++ Runtime System se encarga de localizar el objeto chare real en el espacio global de objetos en nombre del objeto chare que llama.

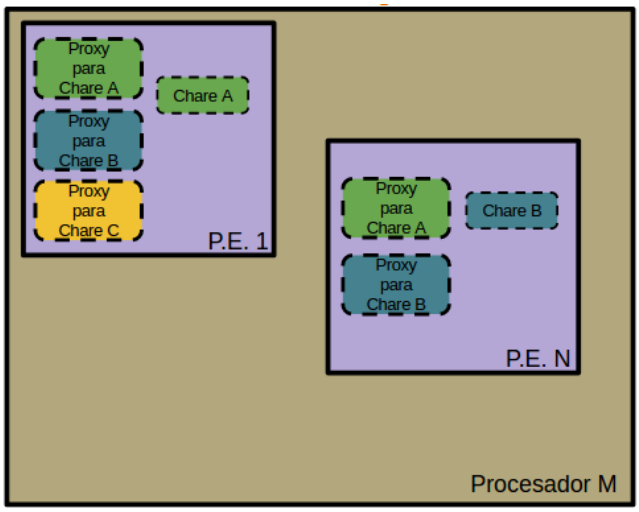

Figura 2 – Objetos Proxie.

Proceso de compilación de un programa Charm++

Charmc es una herramienta de Charm++ que envuelve al compilador nativo. Por compilador nativo queremos decir cualquier compilador que está instalado sobre la máquina donde la aplicación Charm++ está siendo compilada. El propósito de charmc es permitir a los makefiles usados compilar programas Charm++ portátiles. El programador escribe el archivo Makefile utilizando el comando charmc en lugar de directamente llamar al compilador C++ nativo. Charmc está disponible sobre todas las plataformas en que el Charm++ Runtime System esté disponible. Charmc también provee algunas capacidades adicionales además de envolver al compilador nativo C++.

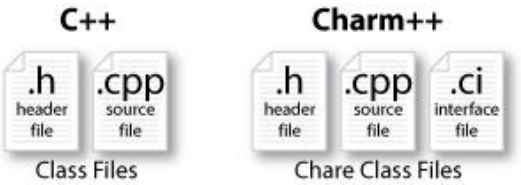

Figura 3 – Partes de un programa C++ y de un programa Charm++.

El archivo Interface es procesado por una herramienta Charm++ llamada charmxi que genera archivos “decl.h” y “def.h”. Estos archivos contienen código auto-generado, que enlaza el código de la aplicación del programador con el del Charm++ Runtime System. Este código autogenerado es incluído en el archivo fuente asociado cuando el archivo fuente es compilado. La Figura 4 muestra el proceso de compilación para una

simple clase chare. Primero, el archivo interfaz es procesado por la herramienta charmxi la cual auto-genera algunos códigos necesarios para el Charm++ Runtime System. Luego, el archivo fuente, con las líneas #include asociadas por los archivos generados, es compilado de una manera similar a una clase C++. El resultado es un archivo objeto que contiene la clase chare.

Después de que todas las clases chare para una aplicación han sido compiladas, charmc se utiliza una vez más para vincularlos. Es durante este paso de vinculación donde el Charm++ Runtime System es también vinculado en la aplicación.

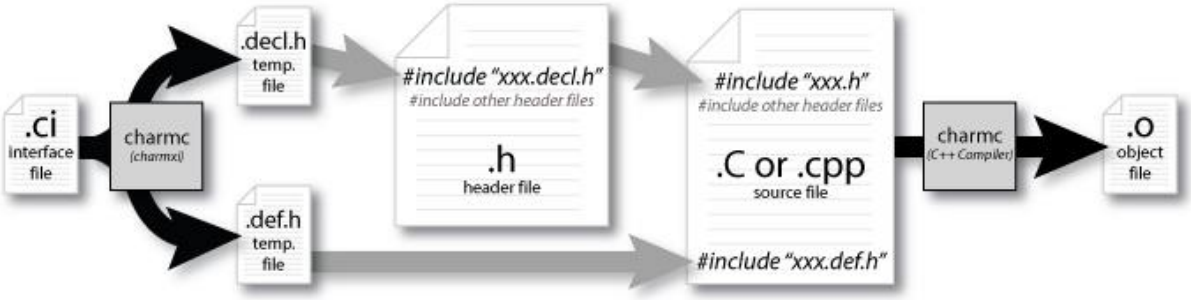

Figura 4 – Proceso de compilación de un programa Charm++.

¿Cómo funciona el algoritmo Merge Sort?

MergeSort es un algoritmo de ordenamiento basado en la técnica *divide y vencerás*. La idea es dividir el problema en subproblemas más pequeños. Si inicialmente tenemos la lista de elementos desordenada, y la dividimos a la mitad, nos quedarán 2 sub-listas desordenadas. Luego, realizamos otra vez la misma acción: dividimos las sub-listas resultados en 4 nuevas sub-listas, y así sucesivamente. Esto se realizará hasta que lleguemos a una sub-lista con 1 o 0 elementos en ella, que por defecto ya estará ordenada, y como ya está ordenada, la mezclamos con la de al lado, que también está ordenada, y así sucesivamente vamos ordenando las sub-listas hacia arriba para llegar al caso base. Ver Figura 5.

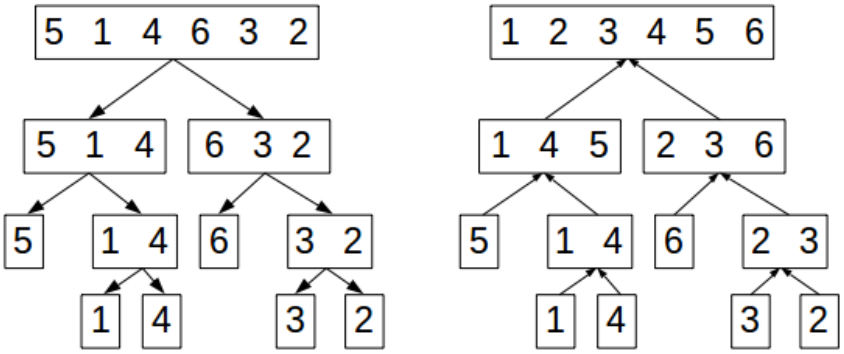

Figura 5 – Merge Sort, división y unión.

Implementación

A la hora de implementar el algoritmo Merge Sort en un programa de manera secuencial, se hace por medio de la recursividad. En Charm++, paralelizar esta recursividad no es explícitamente posible, por lo que se deben aplicar otro tipo de estrategias, de acuerdo a su modelo de programación.

La idea básica es dividir la cantidad de elementos en tantos objetos chare como quiera el usuario. Al finalizar la parte de división, cada objeto chare tendrá una cierta cantidad de elementos, un subconjunto de la cantidad total de elementos desordenados. El siguiente paso es que cada objeto chare ordene su propio subconjunto de elementos mediante algún método de ordenamiento secuencial. En nuestro caso, usamos el algoritmo Merge Sort secuencial.

Luego de esto, comienza la etapa de comparación. Un objeto chare, llamado A, enviará el valor de su último elemento al objeto chare de su derecha, llamado B. Este, comparará dicho valor recibido con su primer valor. B le enviará su subconjunto de valores a A, y la indicación de si debe reordenar el nuevo conjunto de valores. Si necesita ser ordenado, comparará y ordenará de manera secuencial.

Lo explicado anteriormente se realizará en todos los casos, hasta que el primer objeto chare tenga el conjunto inicial, ordenado. En la Figura 6 se explica gráficamente el método usado.

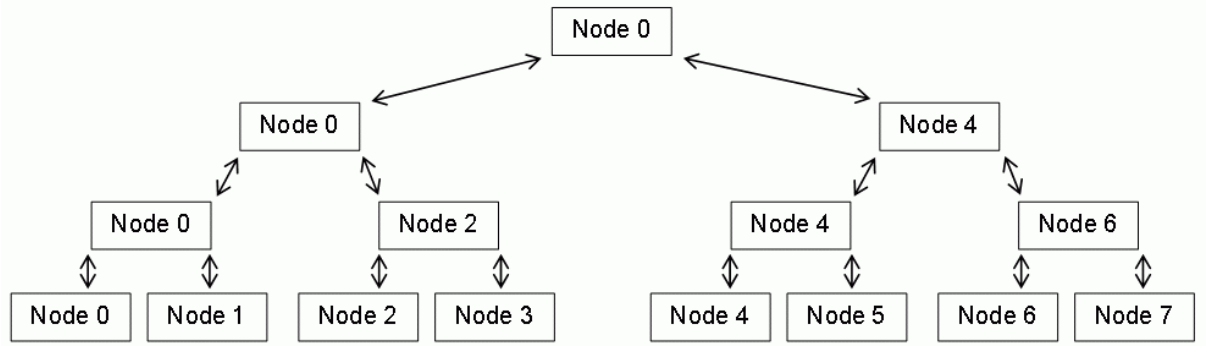

Figura 6 – Merge Sort, estrategia utilizada en Charm++.

Más adelante, mostraremos los resultados obtenidos, y una comparación del uso del mismo algoritmo en OpenMP.

En la Figura 7 se muestra un diagrama de secuencias, donde se puede ver cómo se implementa el algoritmo en Merge Sort en Charm++, y cuáles son los métodos implementados.

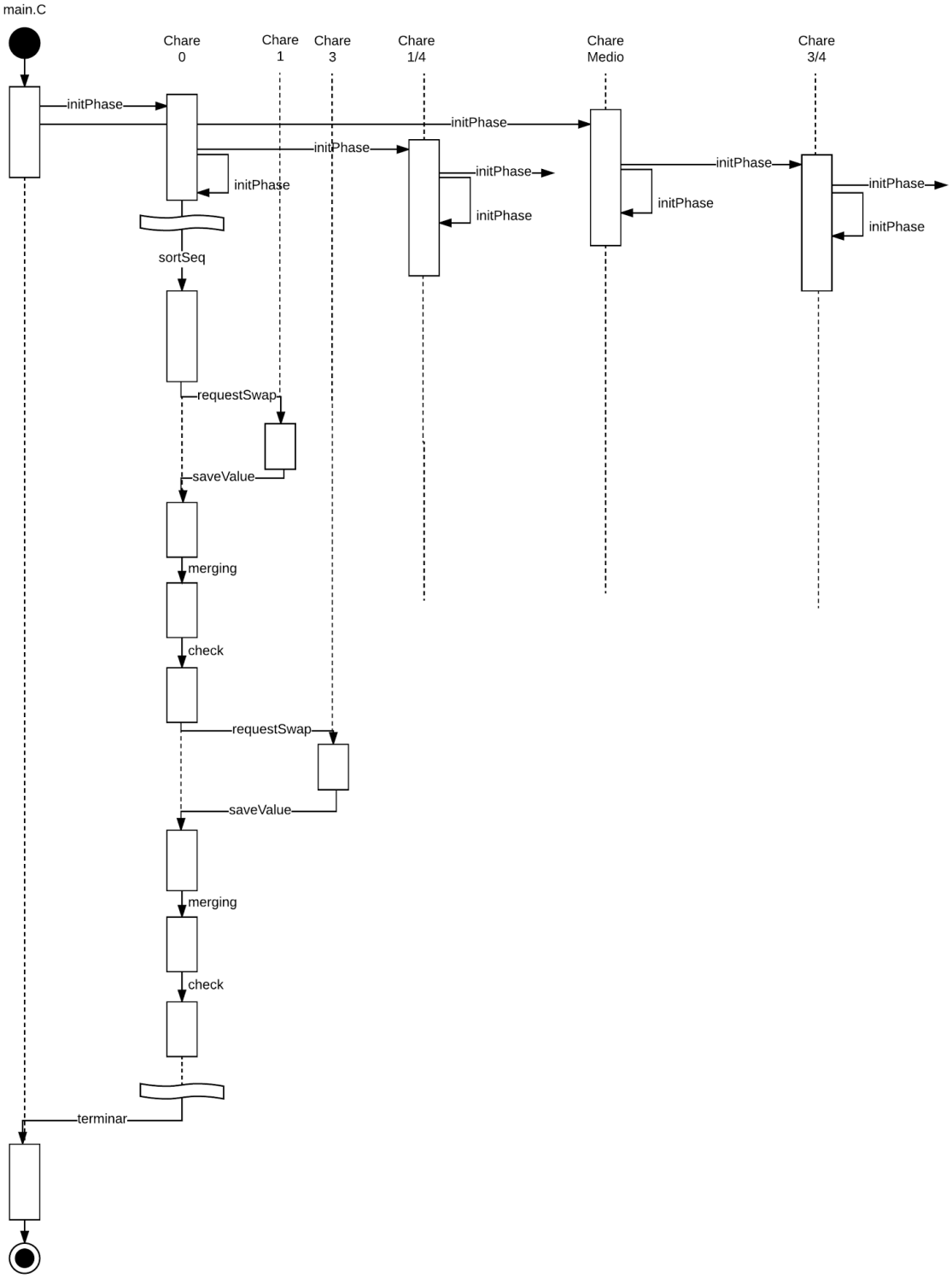

Figura 7 – Diagrama de Secuencias del programa realizado con Charm++.

Análisis de Resultados

Se hicieron comparaciones del tiempo que tarda en realizar el ordenamiento de datos mediante el algoritmo Merge Sort entre Charm++ y OpenMP.

Las ejecuciones se realizaron teniendo en cuenta tres variables:

1. *Cantidad de elementos a ordenar*
2. *Cantidad de procesos en paralelo*
3. *Cantidad de elementos por cada objeto chare* (en Charm++) o por división (en OpenMP).

Otro dato importante a tener en cuenta es que todas las ejecuciones se realizaron en la misma máquina. Las características principales de la misma son:

- 32 procesadores AMD Opteron(™) 6128 800MHz - 512KB Cache
- 33 GB de memoria RAM

Primer Análisis

El primer análisis se realiza con una cantidad fija de elementos para ambos programas, **variando la cantidad de procesos y la cantidad de elementos** por cada objeto chare:

- Cantidad de elementos: 1.048.576. (2^{20})
- Cantidad de procesos: de 1 a 32.
- Cantidad de elementos para cada objeto chare: de 32 a 32.768 (2^{15}).

En la Figura 8 y Figura 9 se puede ver un gráfico de los tiempos obtenidos con ambos programas. A los fines estadísticos, se realizaron 10 ejecuciones para cada variante, para obtener así un número más preciso. En el anexo, se muestra las tablas con los tiempos promedios.

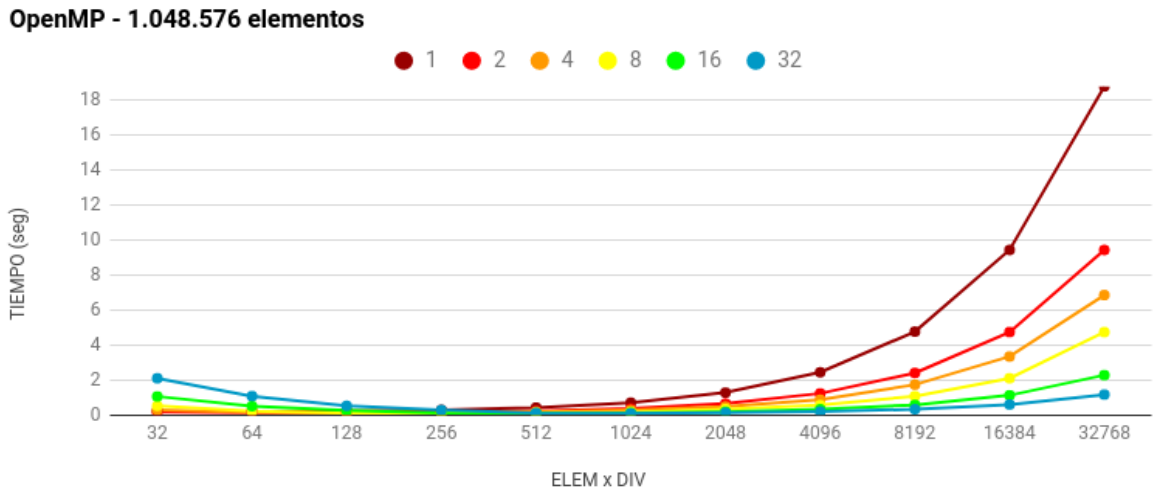

Figura 8 – Grafica de ejecuciones con OpenMP, con 1.048.576 elementos.

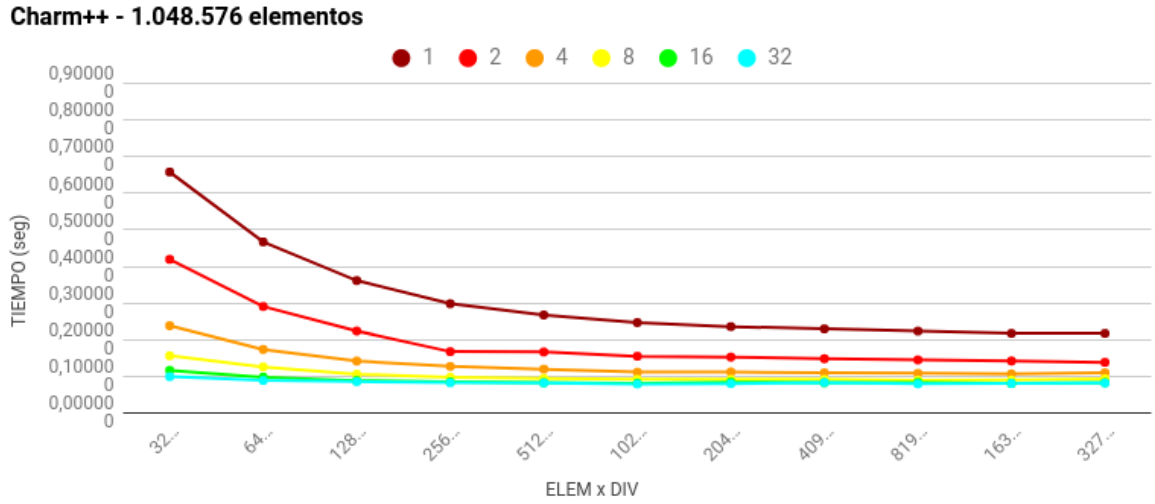

Figura 9 - Grafica de ejecuciones con Charm++, con 1.048.576 elementos.

En las figuras anteriores, se puede apreciar que Charm++ tiene un comportamiento mucho más óptimo a medida que aumenta la cantidad de elementos dentro de cada objeto chare o división, esto hace que la cantidad de objetos chare o divisiones disminuyan y dé como resultado que la comunicación entre objetos chare sea menor, traduciéndose en mejores tiempos. Esto no quiere decir que los tiempos sigan disminuyendo si se le da todo el trabajo a un solo objeto chare. Aunque en la gráfica no se alcanza a notar, luego de llegar al punto óptimo, los tiempos aumentan un poco, aunque no tanto en comparación con OpenMP, que lo hace en forma exponencial luego de llegar a su punto óptimo.

También puede observarse que los tiempos disminuyen notablemente a medida que se utilizan más elementos de procesamiento. Esto se debe a que los objetos chare o las tareas de OpenMP se distribuyen de manera uniforme sobre los procesadores, el paralelismo aumenta, y la comunicación se agiliza.

Un dato a tener en cuenta es que el punto óptimo para ambos está en 1024 elementos por cada objeto chare o división. Tomando este punto óptimo, *Charm++ tiene un mejor rendimiento que OpenMP, disminuyendo el tiempo de cálculo en un 14,57%*.

Resultados:

Tiempo para Secuencial: 0,170 segundos.

Tiempo óptimo para OpenMP: 0,091621 segundos, **1,86** de Speedup.

Tiempo óptimo para Charm++: 0,078268 segundos, **2,17** de Speedup.

Segundo Análisis

El segundo análisis propone ver cómo varían los tiempos de ambos programas si se mantienen fijos la cantidad de procesos y la cantidad de elementos por cada objeto chare o división, y se **varía la cantidad de elementos**:

- Cantidad de elementos: de 1.048.576 a 268.435.456 (2^{28}).
- Cantidad de procesos: 32.
- Cantidad de elementos por objeto chare: 1024.

En la Figura 10 se puede ver un gráfico comparativo entre Charm++, OpenMP y el programa Secuencial, ejecutado sobre un solo procesador a la vez. Nuevamente, a los fines estadísticos, se realizaron 10 ejecuciones para obtener tiempos más precisos. En el Anexo se puede ver la tabla con los tiempos.

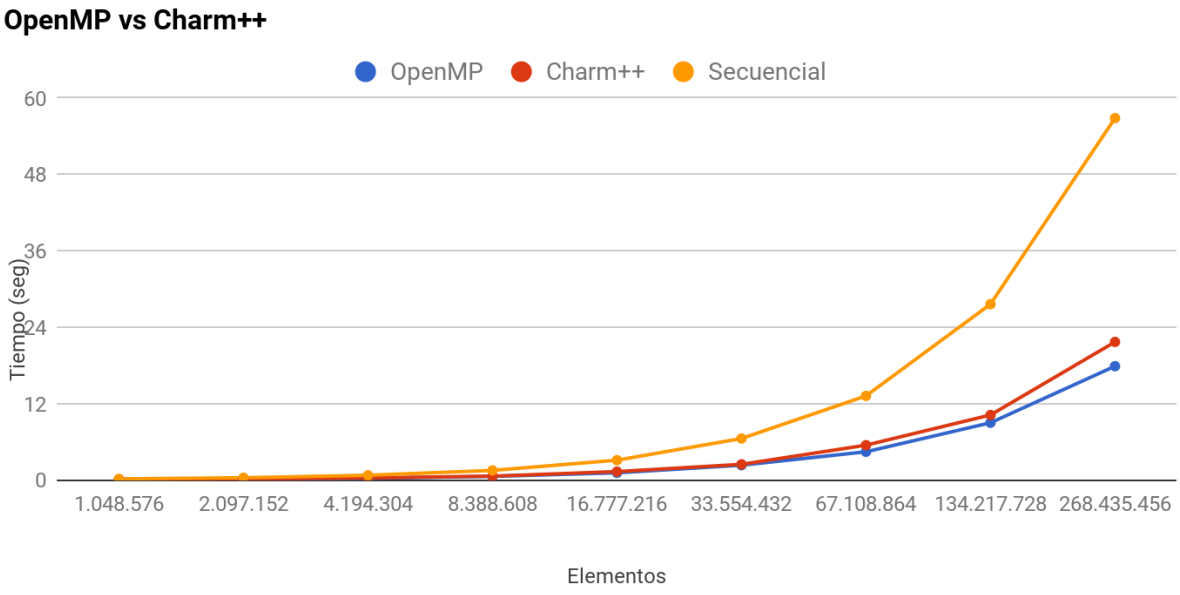

Figura 10 – OpenMP vs Charm++ vs Secuencial.

A medida que aumenta la cantidad de elementos, OpenMP comienza a superar a Charm++ en cuanto a rendimiento, llegando a obtener casi un 19% de mejora en el tiempo respecto a Charm++.

Respecto al secuencial, ambos mejoran notablemente su rendimiento. El Speedup de Charm++ llega a un máximo de 2,706, mientras que el de OpenMP llega 3,180 de Speedup como máximo. Se nota claramente que ambos programas elevan con amplio margen el rendimiento del algoritmo.

Tercer Análisis

Por último, un tercer análisis fue realizado variando la cantidad de elementos, pero en forma lineal (no en potencia de 2):

- Cantidad de elementos: 1.000.000 a 100.000.000.
- Cantidad de procesos: 32.
- Cantidad de elementos por objeto chare: 1024.

En la Figura 11 se puede ver un gráfico comparativo entre Charm++ y OpenMP. En el Anexo se agrega la tabla con los tiempos tomados:

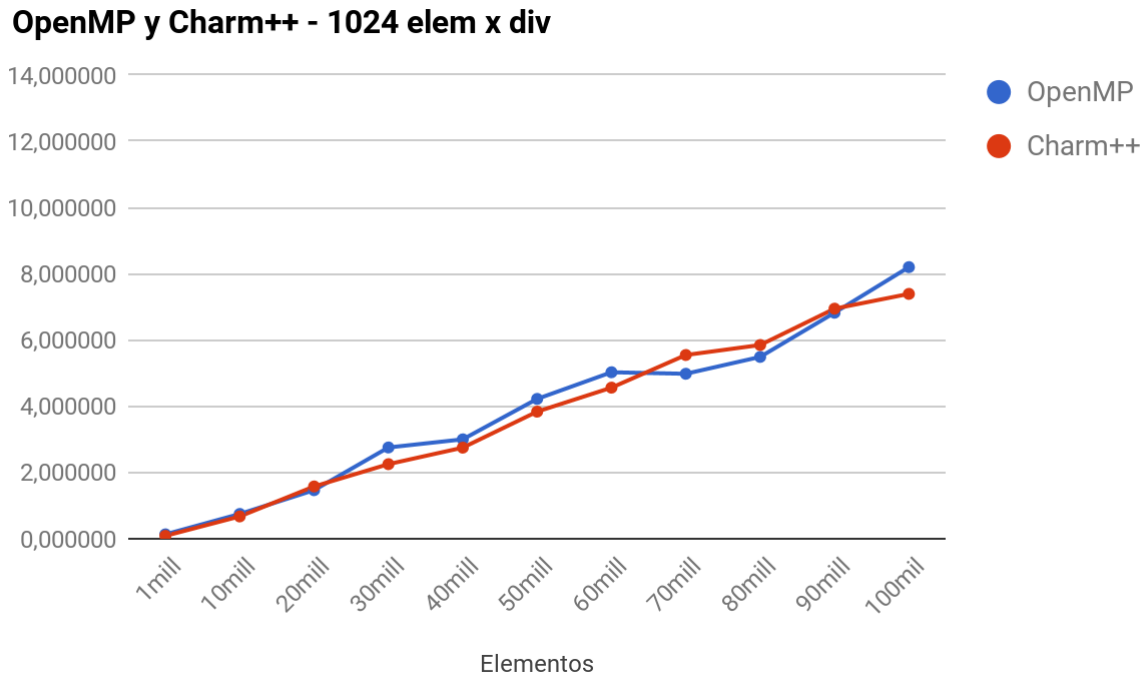

Figura 11 – OpenMP vs Charm++, con sus óptimos.

Se puede ver que Charm++ tiene tendencia a un comportamiento más lineal respecto a OpenMP. Esto es un punto a favor de Charm++, porque de esta manera permite

al programador hacer un análisis de cuánto puede tardar un programa sin siquiera ejecutarlo, con técnicas de interpolación o extrapolación.

Se hizo el mismo análisis, pero esta vez con más elementos por división. El resultado se observa en la Figura 12 y la Figura 13. De esto se desprende, que en OpenMP el rendimiento se ve afectado notablemente con el aumento de la cantidad de elementos por división, mientras Charm++ los tiempos se mantienen iguales, y hasta en algunos puntos, disminuye.

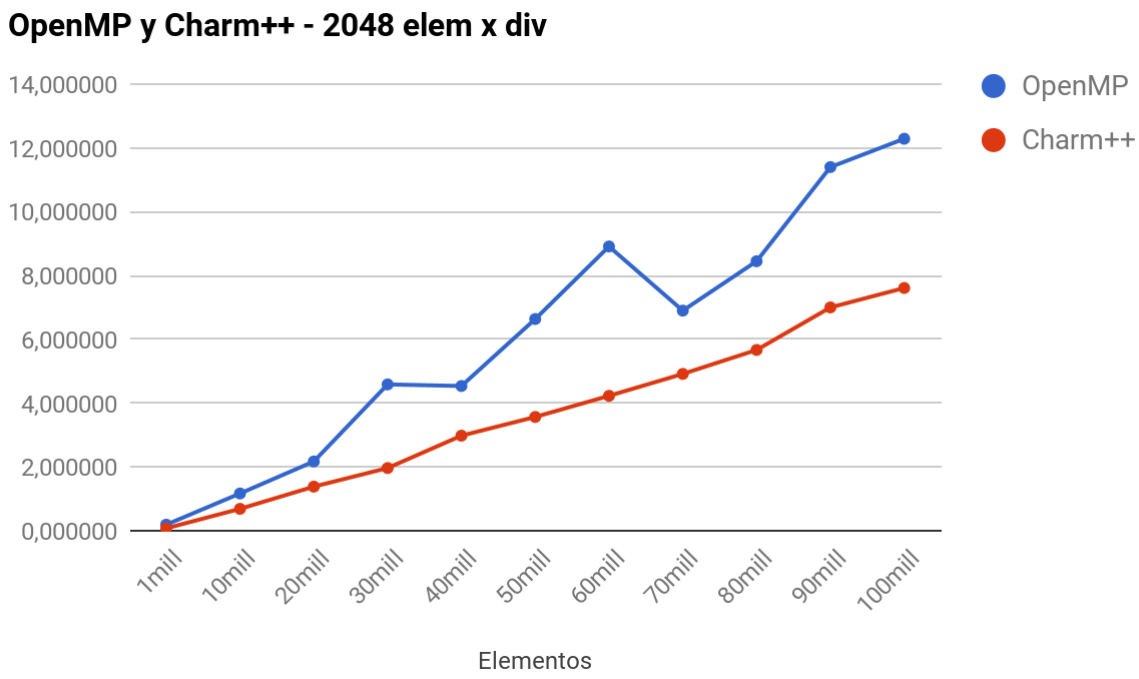

Figura 12 – OpenMP vs Charm++, 2048 elementos por división.

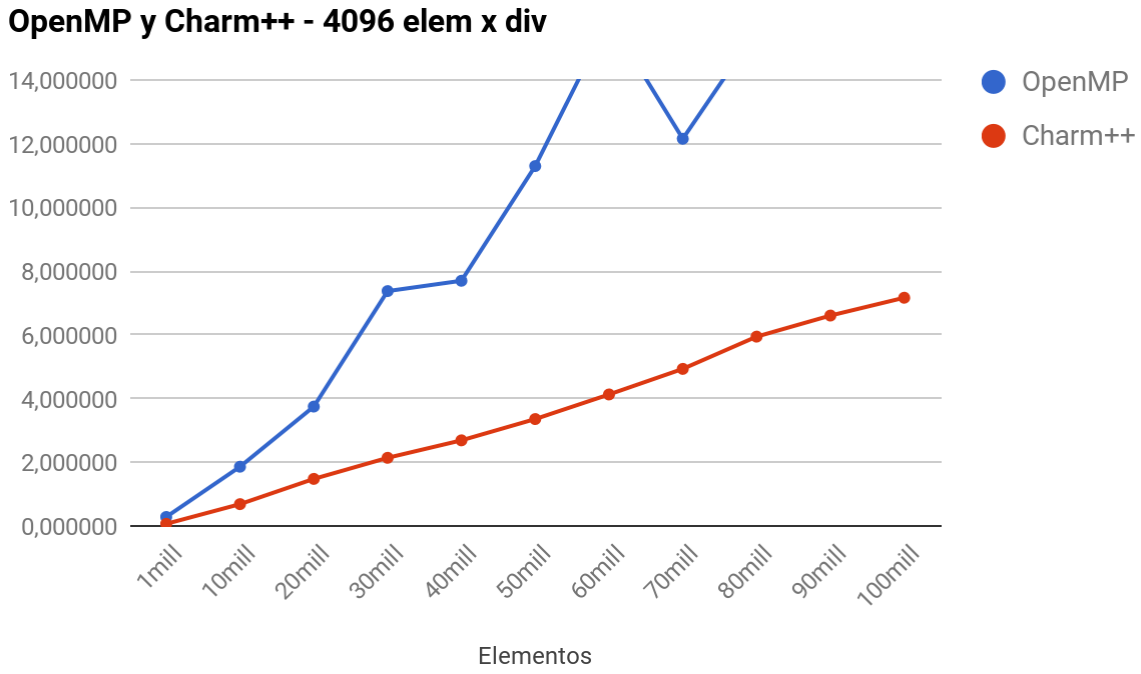

Figura 13 - OpenMP vs Charm++, 4096 elementos por división.

Esto se debe a que en OpenMP la cantidad óptima de elementos por división es 1024, mientras que Charm++ prácticamente no se ve afectado por esta variable, como se muestra a continuación en la Figura 14.

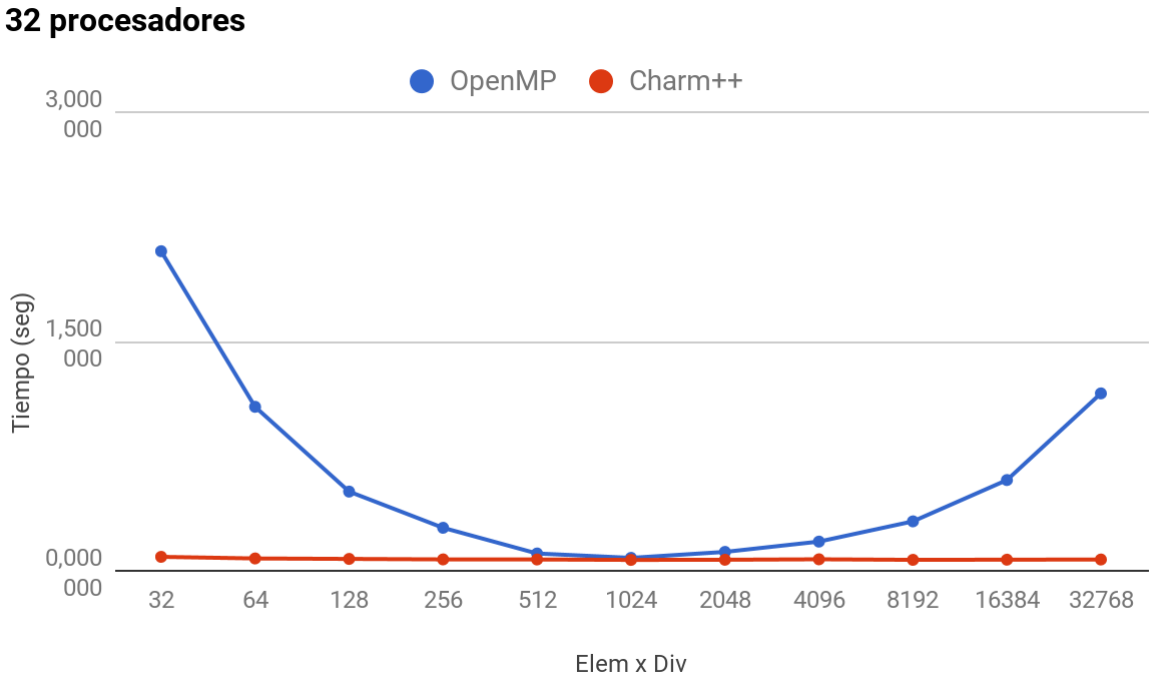

Figura 14 – Punto óptimo de OpenMP y de Charm++.

Conclusión

Del primer análisis se desprende que Charm++ no se ve tan afectado como OpenMP por el aumento de la cantidad de elementos por división, y esto se debe a que se minimiza la comunicación entre objetos chare, siendo ésta la tarea de mayor coste en Charm++. En OpenMP, el aumento de elementos por división provoca que el número de tareas decrezca y los tiempos de espera aumenten.

Del segundo análisis se tiene que los Speedup obtenidos son relativamente similares, y en ambos casos se mejora notablemente el tiempo de ejecución respecto del secuencial.

Del tercer análisis podemos deducir que Charm++ presenta una tendencia a un comportamiento lineal, mientras que OpenMP se ve caracterizada por saltos más acentuados. La linealidad de Charm++ permite predecir los tiempos de ejecución antes de utilizarlo.

Con todo esto podemos decir que Charm++ es un framework robusto, que se adapta fácilmente al aumento en la cantidad de trabajo gracias a su balanceo de carga automático, mientras que OpenMP presenta variaciones importantes cuando se varía la cantidad de tareas, debido a las barreras que siempre esperan a la tarea más lenta o a la tarea con más trabajo.

Charm++ tiene la gran ventaja de permitir ejecutar este mismo trabajo de una manera muy simple sobre varios nodos, y minimizar aún más el tiempo de ejecución. En cambio, OpenMP está limitado a un solo nodo.

Otros puntos a favor de Charm++ son la portabilidad y la compatibilidad. El primero se refiere a que permite programar en paralelo con una independencia bastante importante respecto al hardware sobre el que se ejecuta. El segundo, se refiere a la característica que permite la coexistencia de Charm++ con OpenMP, MPI u otras librerías en un mismo sistema de tiempo de ejecución.

La limitación más importante de Charm++ es que sólo es gratuito para uso educativo y de investigación, por lo que, si uno desea realizar un proyecto con fines económicos, debe optar por pagar una licencia o por usar OpenMP o MPI, por nombrar algunos ejemplos.

En cuanto al costo de utilizar Charm++ respecto a OpenMP, podemos destacar entonces que, en este caso, ambas librerías poseen un rendimiento similar.

Charm++ es una herramienta que no tiene nada que envidiarles a librerías muy usadas como OpenMP o MPI, e incluso tiene mayores ventajas que éstas. Es totalmente recomendable para adentrarse en el amplio mundo de la programación paralela, con una forma de programación sencilla, sin uso de directivas especiales, y con una comunidad que crece día a día.

Anexo A

CANTIDAD DE ELEMENTOS = 1.048.576						OPENMP					
ELEM x DIV / PROCESADORES	32	64	128	256	512	1024	2048	4096	8192	16384	32768
1	0,212063	0,210048	0,230262	0,290788	0,425254	0,706585	1,279810	2,436135	4,758439	9,412376	18,734519
2	0,184366	0,142611	0,140160	0,166142	0,229254	0,368267	0,654372	1,232947	2,400452	4,729692	9,410574
4	0,298189	0,159420	0,128249	0,139506	0,184624	0,283639	0,479265	0,874627	1,732248	3,335086	6,841266
8	0,529598	0,221038	0,125062	0,106337	0,126793	0,161676	0,338045	0,566987	1,076152	2,097215	4,725784
16	1,060068	0,509007	0,261248	0,113534	0,094531	0,121565	0,190579	0,334739	0,577418	1,126645	2,276109
32	2,097426	1,079652	0,525169	0,288331	0,120307	**0,091621**	0,130619	0,198091	0,330392	0,600710	1,167455

Figura 15 – Tabla de ejecuciones en OpenMP con 1.048.576 elementos.

CANTIDAD DE ELEMENTOS = 1.048.576						CHARM++					
ELEM x DIV / PROCESADORES	32 (32768 chares)	64 (16384 chares)	128 (8192 chares)	256 (4096 chares)	512 (2048 chares)	1024 (1024 chares)	2048 (512 chares)	4096 (256 chares)	8192 (128 chares)	16384 (64 chares)	32768 (32 chares)
1	0,658474	0,467110	0,361349	0,297902	0,267056	0,246315	0,235263	0,229167	0,223051	0,216672	0,216672
2	0,419497	0,290297	0,223207	0,166731	0,165871	0,154175	0,151800	0,147394	0,144153	0,141228	0,137425
4	0,237835	0,172724	0,140995	0,126234	0,118846	0,110795	0,111092	0,109214	0,107461	0,105641	0,108975
8	0,155432	0,124858	0,104897	0,096586	0,093564	0,091001	0,091100	0,090904	0,088613	0,089095	0,092764
16	0,115438	0,096915	0,087467	0,083809	0,081865	0,080194	0,083509	0,081717	0,082324	0,079428	0,081983
32	0,098310	0,087787	0,084774	0,081605	0,081160	**0,078268**	0,079308	0,082315	0,078692	0,079761	0,080851

Figura 16 - Tabla de ejecuciones en Charm++ con 1.048.576 elementos.

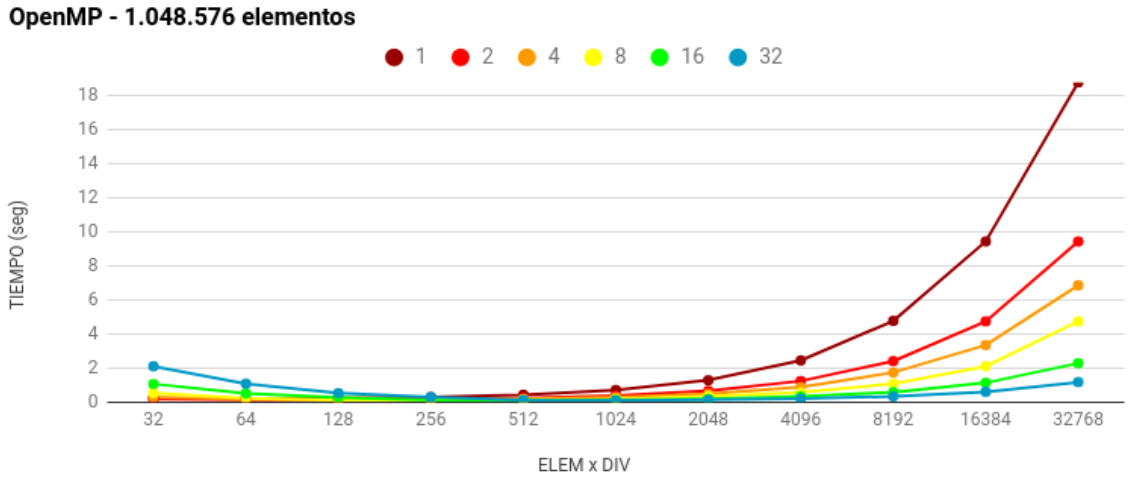

Figura 17 – Grafica de ejecuciones con OpenMP, con 1.048.576 elementos.

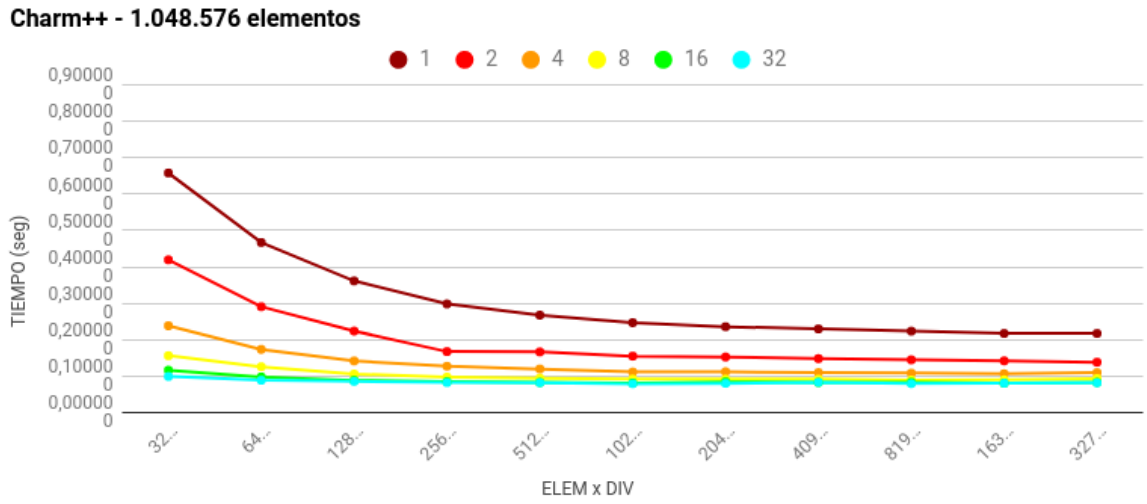

Figura 18 - Grafica de ejecuciones con Charm++, con 1.048.576 elementos.

Anexo B

		Elementos	1.048.576	2.097.152	4.194.304	8.388.608	16.777.216	33.554.432	67.108.864	134.217.728	268.435.456
Cores	32	Librería									
Elem x Div	1024	OpenMP	0,091621	0,160788	0,264859	0,565982	1,121746	2,320424	4,425176	8,982868	17,867215
		Charm++	0,078268	0,170803	0,313523	0,602293	1,314574	2,453891	5,46341	10,204098	21,675442
		Secuencial	0,17	0,35	0,74	1,5	3,11	6,51	13,2	27,61	56,82
	OpenMP	Speedup	1,855	2,177	2,794	2,650	2,772	2,806	2,983	3,074	3,180
	Charm++	Speedup	2,172	2,049	2,360	2,490	2,366	2,653	2,416	2,706	2,621

Figura 19 – Tabla de ejecuciones en óptimos de OpenMP, Charm++ y Secuencial.

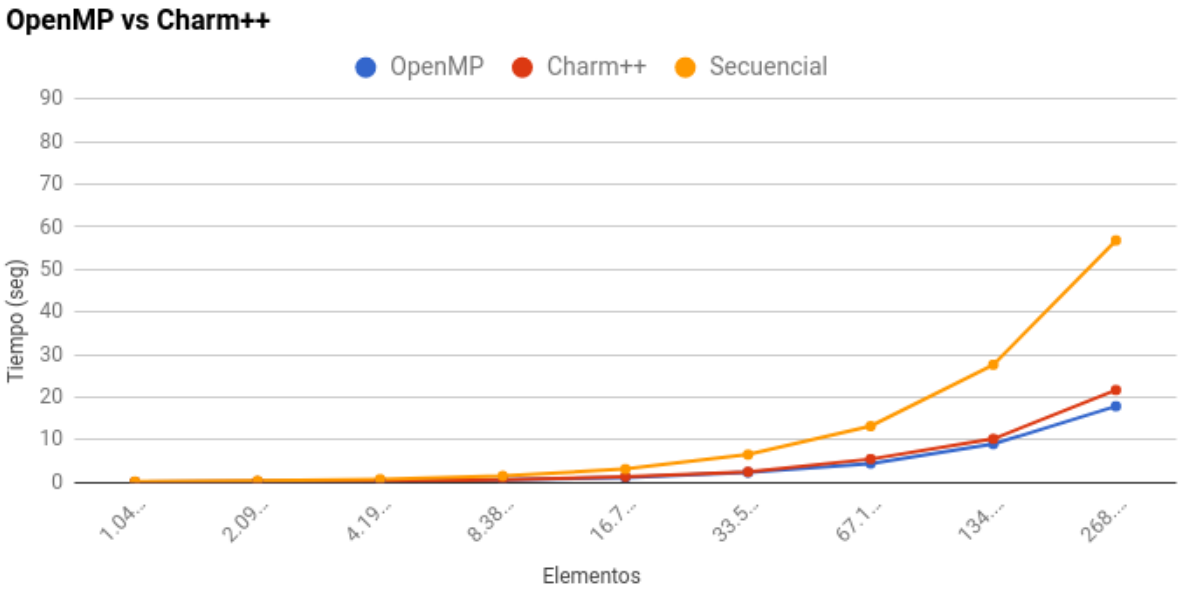

Figura 20 - Grafica de ejecuciones en óptimos de OpenMP, Charm++ y Secuencial.

Anexo C

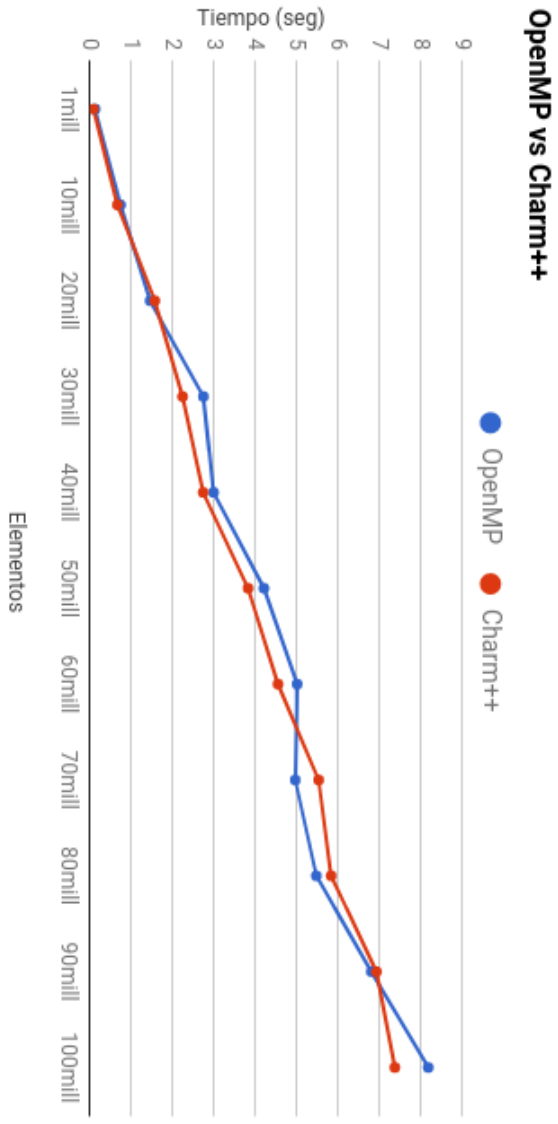

Figura 22 – Grafica de OpenMP vs Charm++ en óptimo.

Cores	32									
Elem x Div	1024									
Elementos	1mill	10mill	20mill	30mill	40mill	50mill	60mill	70mill	80mill	90mill	100mill
Librería											
OpenMP	0,123533	0,739533	1,45782	2,748256	2,993664	4,215137	5,020684	4,974392	5,484217	6,819096	8,19653
Charm++	0,08175	0,6645	1,57004	2,244121	2,742871	3,830818	4,555453	5,541756	5,843495	6,941412	7,388162

Figura 211 – Tabla de OpenMP vs Charm++ en óptimo.

Anexo D

CANTIDAD DE ELEMENTOS = 134.217.728						OPENMP	
ELEMxDIV / PROCESADORES	32	64	128	256	512	1024	2048
1	36,572972	36,273119	39,204894	46,515831	63,759597	99,694028	173,231680
2	25,929450	21,932568	21,485498	24,875686	33,146532	51,068975	87,756101
4	33,456163	20,381101	17,292374	19,095065	24,469501	39,752907	62,234720
8	54,522474	24,364085	13,844349	13,445886	17,086466	24,758082	40,369566
16	127,982709	57,130159	28,816152	9,928501	9,774823	14,745855	20,092670
32	276,303753	131,084279	72,171861	32,778862	12,656983	**8,982868**	13,109668

Figura 22 – Tabla de OpenMP con 134.217.728 elementos.

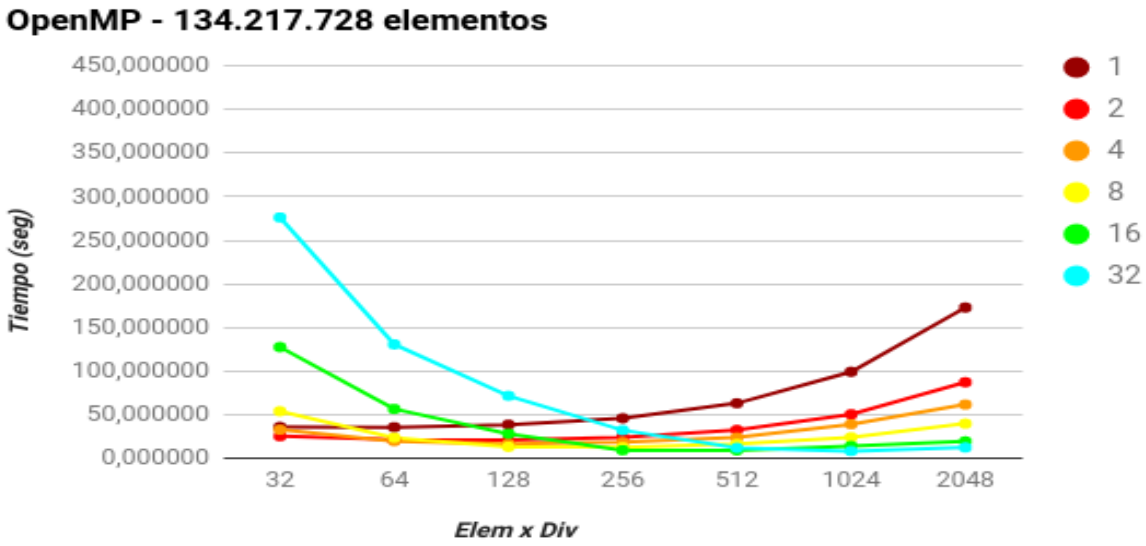

Figura 23 - Grafica de OpenMP con 134.217.728 elementos.

CANTIDAD DE ELEMENTOS = 134.217.728						CHARM++	
ELEMxDIV / PROCESADORES	32 (4194304 chares)	64 (2097152 chares)	128 (1048576 chares)	256 (524288 chares)	512 (262144 chares)	1024 (131072 chares)	2048 (65536 chares)
1	2.348,055664	2.535,383301	1.398,765869	578,013123	401,050018	177,044083	75,439087
2	589,472290	963,755676	484,512939	156,474197	113,673370	56,703190	34,029457
4	125,814278	181,858978	97,570602	43,850422	32,675629	23,535835	18,603153
8	39,471130	47,372040	31,376135	19,127716	16,256638	14,792120	13,213698
16	18,044893	19,095070	14,562688	12,398428	11,136772	11,538560	10,736136
32	13,992332	12,947989	11,660295	10,899199	10,459783	10,204098	10,694845

Figura 24 - Tabla de Charm++ con 134.217.728 elementos.

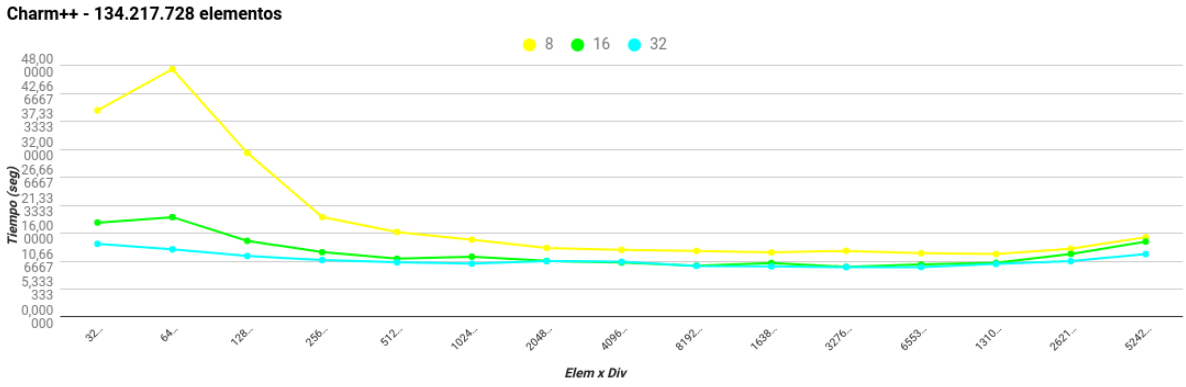

Figura 25 - Grafica de Charm++ con 134.217.728 elementos.

Bibliografía y Links de Interés

Bibliografía

http://charmplusplus.org/

http://charm.cs.illinois.edu/manuals/html/charm++/A.html

http://charmplusplus.org/tutorial/

http://charm.cs.illinois.edu/manuals/html/charm++/manual-1p.html

http://charm.cs.illinois.edu/

http://penguin.ewu.edu/~trolfe/ParallelMerge/ParallelMerge.html

Links de interés

https://github.com/AguilarTarazi/ParallelMergeSort

Glosario

Balanceo de carga: Se refiere a que cada procesador tenga la misma cantidad de procesos ejecutándose. Reparto equitativo de procesos.

Chare: Es una entidad independiente que tiene su propio estado. El acceso a cada una de estas entidades se hace mediante otro objeto denominado Proxie.

Elemento de Procesamiento: Es una representación lógica de un procesador físico. Es un proceso.

Framework: Es un paradigma de programación, es decir, una forma de programar. No es un lenguaje en sí mismo.

Memoria Caché: Área de almacenamiento dedicada a los datos usados o solicitados con más frecuencia para su recuperación a gran velocidad.

Memoria RAM: Memoria principal de la computadora, donde residen programas y datos, sobre la que se pueden efectuar operaciones de lectura y escritura.

Multicore: Mas conocido como Procesador Multinúcleo. Es el que combina más de un microcontrolador en una misma pastilla. Son independientes entre sí.

MPI: ("Message Passing Interface", Interfaz de Paso de Mensajes) es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores.

OpenMP: es una interfaz de programación de aplicaciones (API) para la programación multiproceso de memoria compartida en múltiples plataformas

Proceso: Es un programa en ejecución.

Proxie: Es un objeto que permite la comunicación entre objetos chare, estando no ambos en el mismo procesador.