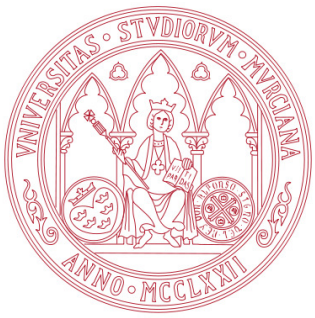

Universidad de Murcia

Facultad de Informática

Grado en Ingeniería Informática

3.er Curso

Arquitectura y Organización de Computadores

Práctica 2 - Procesadores CMP: usando la vectorización y la paralelización

Curso académico 2021-22

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores

1. Introducción

La vectorización es un proceso mediante el cual las operaciones matemáticas realizadas en los bucles internos de aplicaciones de código científico se ejecutan en paralelo usando hardware vectorial específico que se encuentra implementado en la mayoría de CPUs modernas de propósito general así como en algunos procesadores de propósito específico (e.j., Xeon Phi). El paralelismo de tipo `SIMD` (*Single Instruction Multiple Data*), también conocido como paralelismo de datos, se consigue medinte las instrucciones vectoriales implementadas en los repertorios de instrucciones de dichos procesadores. Estas instrucciones, debido a sus orígenes, también son conocidas como extensiones multimedia. Las instrucciones vectoriales disponibles en los procesadores de propósito general actuales operan sobre vectores con un número pequeño (de 2 a 16) y fijo de elementos.

Estas instrucciones vectoriales se ejemplifican mediante una única instrucción (por ejemplo, `vector add`) emitida sobre operandos vectoriales para producir resultados vectoriales. Un "vector" es un conjunto contiguo y homogéneo de datos escalares (que pueden ser de tipo entero o de coma flotante, en cualquiera de sus tamaños habituales como `float` o `double`). Los elementos contenidos en los vectores se procesan simultáneamente, con lo que el efecto neto de la vectorización es una aceleración en los cálculos proporcional al ancho del vector utilizado.

El objeto principal de la vectorización es ejecutar el código de una aplicación, fundamentalmente los bucles internos, usando instrucciones vectoriales. En los procesadores con ISA x86-64, las extensiones que añaden instrucciones vectoriales al repertorio se conocen como SSE[1] (con sus diferentes evoluciones, SSE2 a SSE4), AVX, AVX2 y AVX-512. El número de operaciones que se pueden ejecutar simultáneamente depende del tipo de los elementos y del número de bytes de los registros: 128 bits (16 bytes) para SSE, 256 bits (32 bytes) para AVX y AVX2, y 512 bits (64 bytes) para AVX-512.

Hay varias estrategias que se pueden emplear para usar código vectorial en los programas: a) programar en lenguaje ensamblador las instrucciones vectoriales donde sea necesario; b) programar usando los `intrinsics` vectoriales, que son un conjunto de funciones disponibles para lenguajes de alto nivel que corresponden directamente a instrucciones conmcretas de ensamblador; c) dejar que el compilador vectorice automáticamente; y d) usar una biblioteca optimizada que ya esté vectorizada.

Muchos compiladores vectorizan automáticamente como parte de su proceso de optimización de código. El éxito de la paralelización automática depende de algunas características del código. Determinadas construcciones de código pueden dificultar o imposibilitar que el compilador vectorice adecuadamente los bucles internos. Al mismo tiempo, el uso ineficiente de las cachés y memoria puede anular cualquier aumento de rendimiento obtenido por la vectorización. A medida que las longitudes de los vectores han aumentado en las CPUs modernas, se puede obtener más rendimiento cuando se usa código vectorial, y hay por tanto mayor interés en hacerlo.

Por otra parte, la paralelización es el proceso por el que una aplicación se *divide* en varios hilos de código que se ejecutan simultáneamente. Hace dos décadas que aparecieron los procesadores multinúcleo y, desde entonces, *paralelizar* una aplicación habitualmente ha significado utilizar los 4, 8 o 16 núcleos que tiene el procesador. En la actualidad, paralelizar una aplicación puede también significar usar decenas, cientos, o incluso miles de núcleos (usando varios chips interconectados). Esto requiere pensar de forma diferente porque muchos programas que escalaban aceptablemente en máquinas más pequeñas no funcionarán bien cuando sean escalados a máquinas grandes.

Durante muchos años, al enfoque estándar de la programación paralela se le ha llamado *descomposición del dominio del problema*. Este es un tipo de paralelización de datos. Si pensamos en los datos como organizados en un espacio N-dimensional, hay que buscar el bucle más externo del programa para dividirlo y distribuir los cálculos de ese bucle sobre las tareas (hilos) paralelas. Si hay dependencias de datos entre las tareas, los datos se comunican al final de cada iteración, siendo un factor importante la cantidad de datos a comunicar. Cuando no hay dicha dependencia entre tareas, se habla de que tenemos una aplicación masivamente paralela (*embarranssingly parallel*), siendo el caso mejor ya que podemos llegar a obtener una aceleración linealmente proporcional al número de núcleos que estemos usando[2].

[1] Las extensiones SSE definen también las instrucciones de coma flotante que se usan para operar con escalares.

[2] Siempre y cuando nos lo permita el ancho de banda a memoria.

2. Descripción y objetivos

Esta segunda práctica tiene como principal objetivo familiarizar al alumno con las dos técnicas más importantes para obtener un alto rendimiento en las arquitecturas multinúcleo (CMP) actuales: vectorización y paralelizacion. Esto es, por un lado su capacidad de ejecutar código usando las unidades vectoriales que poseen (*fine-grained data parallelism*) y por otro lado su capacidad de usar todos los núcleos de dichos procesadores (*coarse-grained data parallelism o thread-level parallelism*). Estas dos técnicas con aplicables en todos los procesadores comerciales, desde procesadores empotrados, pasando por procesadores móviles, y llegando a los procesadores para ordenadores de sobremesa o grandes servidores.

Este boletín describe el proceso de vectorización en relación con el hardware de computación, los compiladores y las prácticas de codificación. Nos centramos en la vectorización automática utilizando un compilador optimizador, ya que esta técnica es ampliamente aplicable a los problemas de computación científica y otros. Además, el boletín presenta el manejo básico de OpenMP como medio de paralelizar un programa, proponiendo al alumno paralelizar una aplicación interesante de cálculo numérico como es el cálculo de la integral numérica por el método del punto medio. Esto nos servirá para visualizar el problema de las condiciones de carrera[3] entre procesos y diversas formas de evitarlas, así como también la aplicación del paralelismo de tareas a la resolución de forma recursiva de dicha aplicación.

Más concretamente, los objetivos de esta práctica son:

- Aprender cómo se pueden usar las técnicas de vectorización por parte de los compiladores ICC de Intel (versión 2021.3.0) y/o GCC de GNU (versión 11.0.1).
- Aprender a utilizar técnicas para facilitar la vectorización, como la regularización de bucles vectoriales, y el uso de ayudas (*hints*) al compilador que le permitan relajar algunas de las comprobaciones en tiempo de ejecución relacionadas con la alineación de datos y los *alias* en los punteros.
- Conocer las características básicas del lenguaje OpenMP que nos va a permitir explotar la capacidad multinúcleo de nuestro procesador.
- Conocer y usar la paralelización de bucles.
- Resolver los problemas de condiciones de carrera por medio de la operación de reducción, y conocer cómo hacer operaciones atómicas y cómo usar regiones críticas.
- Utilizar la paralelización de tareas para resolver problemas recursivos.
- Evaluar el rendimiento obtenido por las diversas aproximaciones paralelas realizadas para resolver el problema manejado en la parte de paralelización.

3. La vectorización automática

En este apartado de la práctica vamos a mostrar cómo facilitar la vectorización automática por parte del compilador. La vectorización automática está incluida en muchos compiladores actuales pero no siempre se activa por defecto, por requerir análisis costosos y no ser beneficiosa en algunos casos. Por ejemplo, el compilador ICC trata de vectorizar por defecto a partir del nivel de optimización `-O2`, mientras que GCC y CLang lo hacen solo a partir de la opción `-O3`.

Para verificar la efectividad de la autovectorización y ayudarnos a facilitarla, los compiladores pueden generar un informe de optimización. Dicho informe es una herramientas muy útil, puesto que se indica qué bucles se vectorizaron y qué problemas se encontraron al intentar vectorizar otros. Veremos cómo producir dichos informes en esta práctica.

[3] Se recomienda repasar éste y otros conceptos de la asignatura de Programación Concurrente.

3.1. Demostrando la vectorización automática

En este apartado vamos a ver sucintamente los beneficios que obtenemos en el rendimiento de una aplicación sencilla mediante la vectorización automática.

Para ello, ve al subdirectorio `vectorization/vectorization1`. En él, podemos encontrar un programa que mide el rendimiento obtenido a la hora de sumar un pequeño array de números en coma flotante de simple precisión. Con «`make all`» se generan ejecutables usando ICC, GCC y Clang. El código a analizar se encuentra en el fichero `test_vectorization.cpp` (el resto de ficheros contienen el código para realizar la llamada y medir el tiempo). El programa nos permite comparar varias implementaciones de una suma de arrays:

- Una implementación directa del algoritmo mediante un simple `for`, en la que no se ha prestado especial atendción para facilitar la autovectorización. Aun así, los compiladores tratan de vectorizar este bucle y lo consiguen (`sum_vectors_vectorized`).
- Una versión exactamente igual a la anterior, pero en la que se le ha indicado a los compiladores que no traten de vectorizar el bucle o la función. Esta versión nos servirá para medir el rendimiento base (`sum_vectors_no_vectorized`).
- El compilador no puede saber que los dos arrays que tiene que sumar y el array de destino son, en efecto, tres arrays totalmente independientes sin ningún tipo de solape (aliasing). Esto le obliga a suponer que cada vez que se modifica el array de destino se han podido modificar los arrays de origen, por lo que tiene que volver a cargar cualquier dato que se hubiera leído previamente. Estos accesos extra a memoria dificultan la vectorización y reducen el rendimiento en general. El lenguaje C ofrece la palabra clave `restrict` para indicar que un puntero no apunta a un objeto que se solape con otro objeto apuntado por ningún otro puntero (es decir, que no hay *aliasing*), lo que facilita la optimización. Esta palabra clave no existe en C++, pero los tres compiladores que usamos soportan la misma funcionalidad mediante una extensión usando la palabra `__restrict__`. Incluimos una versión del algoritmo que la usa (`sum_vectors_vectorized_restrict`).
- Otra cuestión que influye en la vectorización es que los datos estén o no alineados en memoria de forma correcta. En procesadores con AVX, la alineación de 32 bytes suele ser óptima, mientras que en procesadores con AVX-512 suele ser de 64 bytes. Incluso aunque los datos estén correctamente alineados en memoria, el compilador en general no tiene forma de saberlo si los recibe a través de un puntero. Cuando sabemos que es así, podemos mejorar el rendimiento indicándolo. Incluimos una versión que así lo hace usando la función `std::assume_aligned` (`sum_vectors_vectorized_aligned`).

Si medimos el rendimiento obtenido por las diferentes versiones de este programa podemos constatar los benficios de la autovectorización. Por ejemplo, en el ordenador `compiladorintel.inf.um.es`, usando el compilador GCC y para un tamaño de array de 512 elementos obtenemos solo 2,12 GFLOP/s con la versión no vectorizada, mientras que llegamos a 15,5 GFLOP/s vectorizando. Debido a la baja intensidad aritmética de este algoritmo, los beneficios son mucho menores para arrays más grandes que la caché porque el tiempo de acceso a memoria limita el rendimiento.

3.2. Facilitando la vectorización automática

Como hemos visto, los compiladores son capaces de vectorizar automáticamente, pero no siempre lo consiguen o no siempre tienen suficiente información para hacerlo de forma óptima. Ya hemos visto cómo es posible proporcionar alguna información útil sobre *aliasing* y alineamiento que puede ayudar a mejorar el rendimiento. Para mostrar algunas técnicas adicionales que pueden favorecer la vectorización automática, vamos a utilizar una aplicación que va a sumar 2 vectores de números en coma flotante (simple o doble precisión), y lo va a hacer de 3 formas distintas: a) utilizando un bucle para sumar todos los elementos de los vectores, b) utilizando una función *escalar* que sumará únicamente 2 elementos de cada vector, y c) utilizando una función vectorial que sumará todos los elementos de los dos vectores por medio de un bucle dentro de dicha función. Se te pide que realices los siguientes ejercicios:

1. Ve al subdirectorio `vectorization/vectorization2` donde encontrarás un fichero principal `vectorization.c` y otro fichero auxiliar `worker.cpp` en el que se encuentran las funciones descritas anteriormente. Ejecuta el comando `make` para compilar el programa y generar los informes de vectorización.

   Este apartado lo vamos a hacer únicamente usando el compilador de Intel ICC, pues el informe de vectorización que genera es mucho más legible. Si observas el contenido del fichero `Makefile` verás que al compilar con ICC se ha activado al generación de informes de optimización con el *flag* `-qopt-report`. El fichero `vectorization-icc.o.optrpt` nos da información acerca de cómo ha ido el proceso de vectorización: para cada bucle, muestra si ha sido vectorizado o no, además de información complementaria. En este caso, puedes comprobar que ha vectorizado 2 bucles, el bucle de inicialización (línea 28 del código fuente) y el bucle que realiza la suma (línea 33 del código fuente)[4] Con respecto al último bucle (línea 36) donde se hace una llamada a una función escalar, el informe nos dice que no se vectoriza precisamente por tener una llamada a una función en otro fichero con lo que el compilador no sabe si es vectorizable. Finalmente, la llamada a una función es totalmente ignorada en el proceso de vectorización.

2. Inserta el `#pragma novector` delante del bucle de la suma (línea 33) para desactivar la vectorización para ese bucle. Repite de nuevo el proceso anterior para comprobar en el informe de optimización que el bucle no ha sido vectorizado. Como se puede observar, la directiva realiza su trabajo, y el informe nos muestra que se vectoriza el bucle de la vectorización, pero no el de la suma.

3. La vectorización es una herramienta muy potente que no sólo se usa para operaciones aritméticas sencillas como sumas o multiplicaciones. Hay otras muchas operaciones aritmético-lógicas que tienen la correspondiente versión vectorizada. Asimismo, todas estas funciones permiten el uso de operaciones enmascaradas (*masked*) donde no todos los elementos del vector son usados. Para comprobarlo, vamos a usar la raíz cuadrada con una declaración `"if"` dentro del bucle. En el cuerpo del bucle de suma, reemplaza la línea 34 (b[i]=a[i]+b[i];) por el siguiente código:

   ```
   if (i %2 == 0)
       b[i] = sqrt(a[i]) + b[i];
   ```

   y repite el proceso del primer punto. Como podrás ver en el informe de vectorización, el bucle sigue estando vectorizado.

4. En este punto vamos a ver cómo vectorizar la *llamada* a una función dentro de un bucle si dicha función es apropiada para vectorizar. Si miramos en el fichero `worker.cpp` en el que se encuentran descrita esta función, vemos que simplemente suma los dos parámetros de entrada. Por tanto, en este caso caso sabemos que es adecuado vectorizar la función. Para ello, vamos a añadir `#pragma omp simd` antes del bucle que llama a `my_scalar_add()`. Si ahora comprobamos el informe de vectorización, veremos que dicho bucle ha sido vectorizado, mostrándonos la siguiente información: `SIMD LOOP WAS VECTORICED`.

5. Por último, nos vamos a fijar en el fichero `worker.cpp`. Lo primero que vamos a ver es cómo vectorizar funciones escalares. Funciones como `my_scalar_add()`, que se podrían utilizar en un estilo de paralelismo de datos, se pueden vectorizar. Este tipo de funciones se denominan "Funciones habilitadas para SIMD" (*SIMD-Enabled Functions*). Para ello, hay que añadir la palabra reservada `__attribute__((vector))` delante de la declaración de la función `my_scalar_add()` en el fichero `worker.cpp`. Si lo haces y miras el fichero `worker.optrpt`, comprobarás que la función ha sido vectorizada (`FUNCTION WAS VECTORIZED`).

   A continuación, nos vamos a fijar en el bucle que hay dentro de la función `my_vector_add()`. Si vamos hacia el final del informe de vectorización (`worker-icc.o.optrpt`) llegaremos a la parte relativa a esta función. En efecto, vemos que el bucle se está vectorizando, aunque de una forma especial. El compilador usa *Multiversioning* para producir dos implementaciones para el bucle,

[4] Como curiosidad, si hubieras puesto el *flag* `-O3`, el compilador realiza una optimización mayor, *fusionando* los dos bucles anteriores y después vectorizándolos.

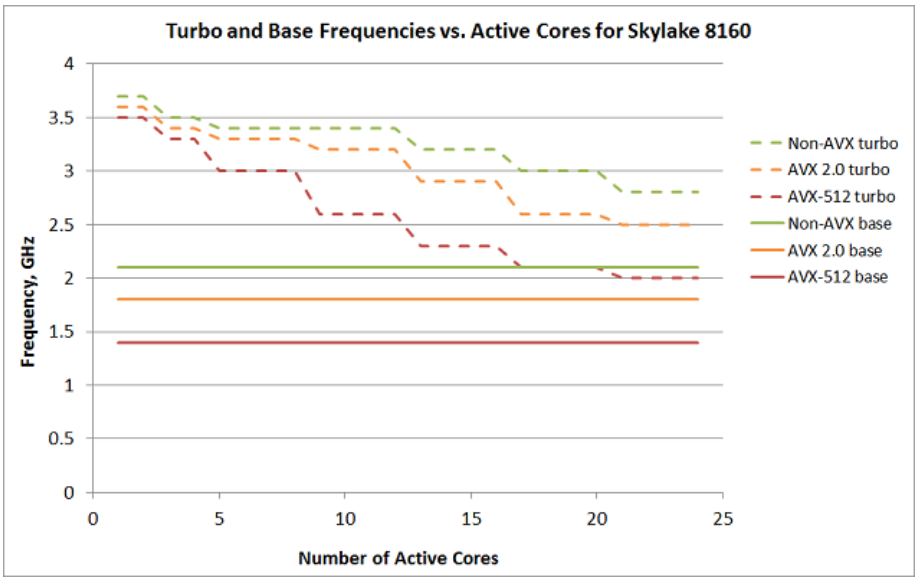

Figura 1: Escalado de las frecuencias base y turbo para diferentes escenarios de vectorización.

una secuencial y otra vectorizada, e inserta en el fichero ejecutable una comprobación en tiempo de ejecución para determinar qué versión utilizar. Esto lo hace porque la función `my_vector_add()` es llamada con 2 vectores, y en tiempo de compilación no puede determinar si es seguro realizar las operaciones de forma vectorizada (debido a que no sabe si los dos vectores son *alias* de la misma dirección, por lo que entonces no sería seguro implementar la versión vectorial). Es decir, que aunque en el fichero `vectorization.cpp` no hemos hecho nada para decirle que dicha función la ejecute de forma vectorizada (el informe de optimización `vectorization-icc.o.optrpt` no señalaba que la llamada a la función podía estar vectorizada), al hacer *multiversioning* puede usar la versión vectorial en tiempo de ejecución si comprueba que no hay riesgo de *aliasing* en las direcciones de los datos de entrada de la función.

El problema de usar un código con *multiversioning* es que la comprobación en tiempo de ejecución puede hacernos perder rendimiento en dicho código (a veces de forma significativa). Si el programador sabe que va a usar dicha función siempre de forma segura, se puede impedir que el compilador use *multiversioning* y genere siempre la versión vectorial, añadiendo la sentencia (`#pragma ivdep`) antes del bucle de la función (que le dice al compilador que *ignore las dependencias* entre las direcciones de las variables que se usan en dicho bucle). Nota: El `#pragma ivdep` es específico para un bucle, por lo que cualquier otro bucle subsiguiente de una función que no tenga dicho *pragma* podría no ser vectorizado. Revisa el informe de optimización `worker-icc.o.optrpt` después de añadir dicha sentencia para confirmar los resultados y que se ha eliminado el *multiversioning*.

Finalmente, ten en cuenta que los procesadores modernos incorporan técnicas de escalado dinámico del voltaje y la frecuencia (DVFS), por ejemplo, en función de la carga de trabajo que estén ejecutando en un instante dado. Esto puede afectar al rendimiento esperado por la vectorización. Puede ocurrir que al aumentar el tamaño de los vectores (e.j, de 256 a 512 bits) no consigamos doblar el rendimiento. El uso de vectores más anchos consume más energía y genera más calor, con lo que un procesador que integre mecanismos de DVFS puede tratar de compensar el calor generado aplicando una reducción de la frecuencia, especialmente cuando hay un uso intensivo de todos los núcleos. La figura 1 ilustra este efecto para el procesador Intel Skylake 8160 donde se muestra cómo varía la frecuencia en varios escenarios: no usando vectorización, activando AVX2 (vectores de 256 bits), o activando AVX-512 (vectores de 512 bits).

4. Usando el paralelismo

En este apartado de la práctica vamos a explotar al máximo la capacidad multinúcleo de nuestro procesador, pues es una de las principales características que poseen los procesadores actuales.

4.1. Descubriendo el paralelismo

Para empezar, vamos a familiarizarnos con las características básicas del lenguaje OpenMP[5] que es el interfaz que nos van a permitir usar los diferentes núcleos del procesador.

Para ello, ve al sub-directorio `OpenMP-basics` de la práctica, y ejecuta el comando `make` para compilar nuestro programa de prueba. A continuación ejecuta `./hello` y observa la salida. Como verás, nos muestra que tenemos "-1 hilos disponibles", y nos saluda desde dicho hilo "-1". Edita el fichero `hello.cpp` y realiza lo siguiente:

1. Modifica la definición de la variable `total_threads` para que la primera sentencia `print` imprima el valor correcto del número de hilos disponibles; para ello usa la función `omp_get_max_threads()` de OpenMP. A continuación, modifica la definición de la variable `thread_id` para que la sentencia que imprime `"Hello World"` indique el número real de cada hilo que ejecuta dicha instrucción; para ello usa la función `omp_get_thread_num()` de OpenMP. Recuerda que debes añadir el archivo de cabecera `#include <omp.h>` en el código fuente de la aplicación. Si vuelves a compilar por medio de `make` y observas la salida tras ejecutar el nuevo binario, verás que ahora ya obtienes el número de hilos del ordenador (4 en el caso de la máquina `compiladorintel.inf.um.es`), aunque solo el primer hilo (con identificador 0) nos saluda desde el programa.

2. Para conseguir que todos los hilos de la aplicación ejecuten la instrucción `printf("Hello world from thread%d", thread_id);` se debe paralelizar dicha sentencia `print`. Para ello, vamos a utilizar el *pragma* de OpenMP `#pragma omp parallel`. Dicho *pragma* hace que todos los hilos de ejecución realicen en paralelo la siguiente instrucción después del *pragma*. Si queremos ejecutar más de 1 instrucción después del *pragma*, hay que encerrar entre llaves dichas instrucciones[6]. Añade dicho *pragma* y comprueba que te *saludan* todos los hilos del programa.

3. Se puede usar el modificador `num_threads()` para que la sentencia `#pragma omp parallel` controle el número de hilos que ejecutan en paralelo una instrucción. Si haces la prueba y pones `#pragma omp parallel num_threads(2)`, observaras que la instrucción `print` de "Hello World" se imprime únicamente desde la mitad de los hilos disponibles. *Nota*: Si el argumento de este modificador fuera un valor no válido (<1), entonces el compilador produce un error: `expected positive integer`.

4. Por último, también se puede controlar el número de hilos de ejecución para una aplicación por medio de la variable de entorno `OMP_NUM_THREADS`. Modifica el valor de dicha variable (por medio de ejecutar el siguiente comando `export OMP_NUM_THREADS=n`, donde `n` es el número de hilos para los que queremos ejecutar el programa paralelo. Haz varias pruebas para ejecutar la aplicación con diferente número de hilos. Asegúrate de desactivar la variable `OMP_NUM_THREADS` al acabar.

Nota: La herramienta `gnome-system-monitor` te puede ser muy interesante para ver el uso que estás haciendo de los diferentes núcleos (*cores*) del sistema.

4.2. Usando el paralelismo de datos

A continuación, vamos a usar el lenguaje OpenMP para cosas más interesantes. Concretamente, vamos a usar como ejemplo una aplicación que realiza la integral de la función $1/sqrt(x)$ con respecto a x de forma numérica por medio de calcular el área que hay por debajo de la función. Lo haremos para el

[5] Para afianzar mejor los conceptos de OpenMP, te puede venir bien dentro del material de prácticas revisar una carpeta denominada OpenMP Tutorial, que tiene también diversos ejercicios resueltos.

[6] Ten en cuenta que la primera llave no puede estar en la misma línea del *pragma*.

intervalo x = 0,0 a x = 1,0 dividido en un número muy elevado de puntos (1.000.000.000), utilizando el método del punto medio como el valor de la función en dicho intervalo. Dado que el número de puntos es tan elevado, esta aplicación es ideal para realizar un paralelismo de tareas. Aunque como veremos, no es tan sencillo, pues nos aparecerá el problema de las condiciones de carrera en el código (muy habitual en este tipo de paralelismo). Para solucionarlo, usaremos la operación *reducción* que es una forma sencilla de hacerlo.

Para empezar a realizar este apartado, ve al sub-directorio `OpenMP-reduction` de la práctica y realiza las siguientes pruebas:

1. En primer lugar vamos a probar una versión secuencial (sin usar *pragmas* de OpenMP) para confirmar que el resultado obtenido por el código es correcto. Haz un `make` para compilar y ejecuta el binario generado (`./integral`). Vemos que obtenemos un error de tan solo 0,00001913 con respecto al cálculo teórico de la integral, lo que indica que se ha calculado la función con una muy buena aproximación.

   Para paralelizar esta aplicación, vamos a usar lo aprendido en el apartado anterior, y vamos a añadir un *pragma* que permita ejecutar en paralelo. En este caso, si añadimos `#pragma omp parallel` antes del bucle `for`, lo único que conseguiríamos sería una redundancia en los cálculos, pues todos los hilos ejecutarían el mismo bucle y calcularían el mismo valor de la integral.

   Lo que queremos hacer en este caso es usar el paralelismo de datos ¡tenemos 1.000.000.000 de puntos! para que cada hilo calcule en paralelo un trozo de dicha integral. Por tanto, vamos a utilizar otro *pragma* de OpenMP para dividir la ejecución de un bucle y que se realice en paralelo. Añadimos `#pragma omp parallel for` justo antes de la instrucción del bucle `for` para paralelizar el bucle. Hacemos `make` para compilar y ejecutamos la aplicación, obteniendo un resultado de 0,31783725 con un error de 1,68216275.

   Como puedes ver, la aplicación se ha ejecutado mucho más rápido, pero produce un resultado erróneo. El motivo debe estar relacionado con la paralelización del código. Para confirmar esta sospecha, ejecuta la misma aplicación con 1 hilo usando la siguiente variable de entorno por medio de este comando: `export OMP_NUM_THREADS=1`. Ahora deberías ver que la aplicación produce el resultado correcto. Asegúrate de borrar la variable de entorno después (`unset OMP_NUM_THREADS`), de lo contrario todas las ejecuciones posteriores con cualquier aplicación en la misma sesión de terminal serán secuenciales (ejecutadas por un único hilo).

2. ¿Dónde está el problema? El problema viene por usar una variable global (`integral`) para realizar una operación de *reducción*, que implica que dicha variable sea accedida en paralelo por todos los hilos. Cada uno de ellos añade la parte de calculo que ha realizado, pero dicho acceso paralelo no se realiza en exclusión mutua y se producen *condiciones de carrera* que nos impiden obtener el resultado correcto.

   Para solucionar la operación de *reducción* de una variable hay que proceder de la siguiente forma: cada hilo tiene una copia privada de la variable que se reduce (*integral* en este caso), que se utiliza para almacenar el resultado intermedio (en este caso, la suma parcial). Al terminar el bucle, las variables privadas son *reducidas*, es decir, se combinan de una manera segura en una variable compartida.

   Para entenderlo bien, vamos a implementarlo por nosotros mismos. Para ello, primero separa el `#pragma omp parallel for` en dos *pragmas* distintos, un `#pragma omp parallel` y un `#pragma omp for`. Esto es debido a que el cálculo y la reducción se deben hacer en el ámbito de la misma región paralela. A continuación, crea la variable *partial_sum* privada para cada uno de los hilos. La solución más fácil es simplemente declarar *partial_sum* dentro del ámbito del `#pragma omp parallel` antes del `#pragma omp for`. Un método alternativo, adecuado cuando todas las variables se declaran en el comienzo de la función, para hacer que las variables de fuera de la región paralela sean privadas para cada hilo es usar la clausula *firstprivate* de la orden `#pragma omp parallel`. Por último, vamos a *reducir* (es decir, sumar) los valores de *partial_sum* en cada hilo en la variable *integral*. Para asegurar la exclusión mutua y evitar las *condiciones de carrera* hay

que realizar dicha *reducción* de forma atómica. Para ello, hay que añadir la línea `#pragma omp atomic` o `#pragma omp critical` justo antes de realizar la reducción para asegurar que este paso final es serializado y se realiza por tanto en exclusión mutua. Compila y ejecuta la aplicación con múltiples hilos y comprueba que sigue funcionando más rápido que el secuencial pero ahora nos ofrece un resultado correcto.

3. Al ser tan frecuente este tipo de operación, el lenguaje OpenMP ofrece la cláusula `reduction` a la orden `#pragma omp parallel for`. En dicha cláusula hay que especificar la operación de reducción que se quiere llevar a cabo, así como las variables compartidas que van a ser *reducidas* en dicho bucle. De esta forma, automáticamente y sin hagamos nada más, el compilador se encarga de hacer los pasos descritos en el apartado anterior. Añade esta cláusula con los argumentos adecuados (`#pragma omp parallel for reduction(+: integral)`). Compila y ejecuta la aplicación con múltiples hilos y comprueba que sigue funcionando más rápido que el secuencial, sigue dando un resultado correcto, pero con una programación mucho más fácil.

Nota: Para esta aplicación particular, no hay una diferencia observable en el rendimiento entre usar la instrucción `#pragma omp atomic` y `#pragma omp critical`. Sin embargo, funcionan de forma bastante diferente. `#pragma omp atomic` sólo puede aplicarse a algunas operaciones escalares, y asegura al programador que las direcciones de memoria utilizadas en la operación se actualizan atómicamente. `#pragma omp critical` asegura que todas las operaciones en su ámbito de aplicación, independientemente del tipo, son llevadas a cabo por un solo hilo a la vez. En otras palabras, *atomic* es un bloqueo en los datos, mientras que *critical* es un bloqueo en una *región crítica* de instrucciones. Si la operación es compatible con el uso de *atomic*, es recomendable utilizar `#pragma omp atomic` para lograr un mayor rendimiento.

4.3. Usando el paralelismo de tareas

En este apartado de la práctica vamos a ver cómo usar el paralelismo de tareas. Una forma habitual y eficiente de aplicarlo usando OpenMP es por medio de implementar la recursión de tareas de forma paralela. El código fuente proporcionado es similar al del apartado anterior, haciendo uso de una función que explota la recursividad. El objetivo, por tanto, de este apartado es paralelizar el cálculo de la integral numérica de forma recursiva utilizando el paralelismo de tareas que nos ofrece OpenMP. Para realizar este apartado, ve al sub-directorio *OpenMP-tasks* de la práctica y examina el código en el que se ha implementado el calculo numérico de una integral por medio de la llamada a una función recursiva. La implementación dada es secuencial. Compila mediante `make` y ejecuta la aplicación para comprobar que obtenemos un error mínimo y que funciona correctamente de forma secuencial.

Vamos a paralelizar entonces el programa utilizando la funcionalidad de la recursividad paralela que ofrece OpenMP:

1. Empezaremos por la función `recursive_integral()`. Esta función tiene una sentencia `if` con 2 partes, siendo la parte del `else` la que nos sirve para llamar recursivamente a la función, devolviendo el valor de la función en la variable `child_integral`. Por ello, antes de dicha llamada hay que añadir la instrucción `#pragma omp task`. Debido a que cada llamada tendrá su propia copia, debemos antes inicializar dicha variable, y asegurar por medio de usar la clausula `shared(child_integral)` que todas las tareas creadas comparten dicha variable (pues el comportamiento por defecto en una región `omp task` es hacer una copia privada). Observa que cuando se paraleliza una función recursiva solo se crea una tarea para reducir el número total de tareas paralelas. Ten en cuenta que en cada llamada recursiva creas una nueva tarea.

   Para evitar las *condiciones de carrera* en la suma de la variable `integral`, se añade la instrucción `#pragma omp taskwait` antes de realizar dicha suma, pues dicha sentencia *fuerza* un punto de sincronización, ya que la tarea *padre* no puede continuar hasta que hayan acabado las tareas *hijas*.

   Con esto finalizamos la paralelización de la función recursiva, pero nos quedan un par de detalles a tener en cuenta. Por una parte, la sentencia `#pragma omp task` siempre debe estar dentro de una región `#pragma omp parallel`. Para ello, en el programa principal, antes de hacer la primera

llamada a la función añadimos dicha sentencia. Pero entonces todos los hilos llamarían a la función recursiva. Para evitar este efecto indeseable, lo que hacemos es que un único hilo haga la llamada a la función recursiva por medio de realizar dicha llamada precedida de `#pragma omp single` o `#pragma omp master`.

2. Una vez paralelizada la aplicación, ejecútala para comprobar que el resultado es correcto y que es mucho más rápida que la versión secuencial. *Sugerencia*: para observar una diferencia más clara, amplía el tiempo de ejecución inicializando `nSteps` a 10.000.000.000 puntos.

4.4. Evaluando el rendimiento de la paralelización

Para acabar esta práctica, vamos a evaluar el rendimiento obtenido por la paralelización (en términos de tiempo) de las dos versiones anteriores de la integración numérica de la raíz cuadrada. Para medir el tiempo de ejecución, puedes aprovechar la función `omp_get_wtime()` que nos proporciona OpenMP, aunque también lo podríamos hacer a través del comando `time` (el valor *real* proporcionado suele ser una buena aproximación al tiempo de ejecución del programa).

Para realizar este apartado, ve al sub-directorio `OpenMP-time`. Ejecuta `make` para compilar las dos versiones proporcionadas (en este caso hemos usado el compilador ICC, puedes probar con los otros compiladores). Para remarcar las diferencias hemos usado un `nsteps` de 10e10. En la máquina `compiladorintel.inf.um.es` y usando 4 hilos, se observa que la versión que usa reducción (paralelización por datos) ofrece una mejora de un 30 % sobre la versión recursiva (paralelización por tareas). Esto nos muestra como, si es posible, la paralelización de datos es mas competitiva que la de tareas.

5. Ejercicios pedidos

Accede al directorio `ejercicios` y realiza los dos ejercicios siguientes.

1. En el subdirectorio `vectorización` hay un pequeño programa que tiene 6 bucles.

   - Para cada uno de los bucles del programa, identifica las dependencias entre iteraciones que tiene y explica cómo afectarían a una posible vectorización automática.
   - Utiliza el `Makefile` incluido para compilar el programa con el compilador ICC y generar el informe de optimización. Prueba el programa y comenta el contenido del informe de optimización. Para cada bucle mencionado en el informe, explica por qué ha sido vectorizado o por qué no se ha podido vectorizar. En especial, para los bucles en los que se hubieran identificado dependencias entre iteraciones en el punto anterior:
     - Explica en caso de que haya sido vectorizado qué transformaciones ha aplicado automáticamente el compilador.
     - Y en caso de que no haya sido vectorizado, indica si crees que existe alguna transformación manual posible para que se pudiera vectorizar.

2. En el directorio `paralelización` hay un pequeño programa que simula la propagación del calor en una superficie rectangular. Se supone que la temperatura de cada uno de los bordes de la superficie se mantiene constante y el programa calcula la temperatura final de cada punto de la superficie representada como una matriz. Para ello utiliza un algoritmo iterativo tipo *stencil*, el cual actualiza cada uno de los elementos de la matriz en función del valor previo del mismo elemento y sus vecinos.

   El código principal del algoritmo se encuentra en el fichero `heat.cpp`, que será el único que se necesitará modificar. Además, el fichero `matrix.h` proporciona un tipo de datos para almacenar las matrices utilizadas por el algoritmo, mientras que el fichero `main.cpp` incluye el código necesario para ejecutar el programa y medir el tiempo empleado en realizar la simulación. El programa permite configurar múltiples parámetros de la ejecución mediante opciones de la línea de comandos (ver `main.cpp`).

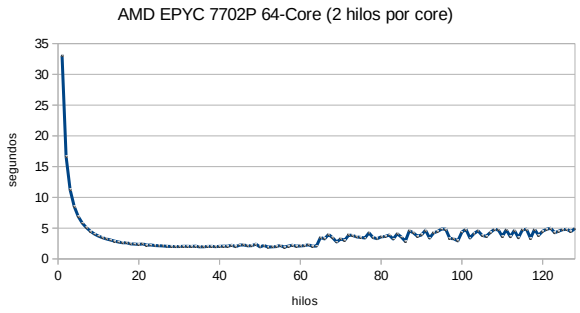

Figura 2: Tiempo de ejecución vs nº de hilos

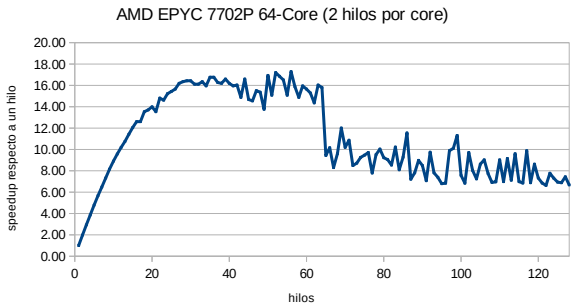

Figura 3: Escalabilidad de la aplicación

Además del programa, se incluye un `Makefile` para compilarlo usando tanto el compilador GCC como el ICC. Se incluyen también scripts para automatizar la comprobación de que los resultados son correctos y para automatizar la medida del tiempo de ejecución del programa usando diferente número de hilos (generando ficheros `tsv` que se pueden utilizar para generar gráficas fácilmente con multitud de programas, incluyendo cualquier hoja de cálculo). Se aconseja el uso de un ordenador que tenga 4 cores o más. Finalmente, se incluyen tres casos de prueba y su salida correspondiente, que se utilizarán para comprobar la corrección del programa modificado y para medir su rendimiento. El `Makefile` incluye objetivos para ejecutar todos los tests (`make tests` o `make tests-icc`) y para realizar la toma de tiempos (`make times` o `make times-icc`).

Para completar este ejercicio, realiza lo siguiente:

- El procedimiento `solve` de `heat.cpp` tiene 3 bucles. Identifícalos y explica para cada uno de ellos si es candidato para la paralelización usando OpenMP. Indica en cada caso qué `pragma` (o `pragmas`) sería necesario añadir para realizar la paralelización correctamente. Presta especial cuidado a la clasificación de las variables (ya sean de reducción, compartidas o privadas).
- Prueba a paralelizar individualmente cada uno de los bucles paralelizables y prueba también a paralelizar todos los bucles paralelizables a la vez. Para cada versión resultante del programa, mide el tiempo de ejecución con diferente número de hilos de cada uno de los tres casos de prueba incluidos y genera las correspondientes gráficas mostrando el tiempo de ejecución y la escalabilidad obtenida (similar a las figuras 2 y 3). Comenta los resultados.

5.1. Entrega

Tras finalizar los ejercicios anteriores, se pide entregar lo siguiente:

- Un documento en formato PDF que será el documento principal utilizado a la hora de evaluar la práctica. Dicho fichero debe incluir:
  - Información sobre la autoría de la práctica.
  - Las respuestas a los ejercicios solicitados.
  - Un breve informe comentando los aspectos positivos de la práctica, así como cualquier aspecto negativo y cosas que has echado en falta en la misma. La longitud del informe no debe exceder las 2000 palabras.
- El código fuente de cualquier programa o script desarrollado (o modificado) para la realización de la práctica. Estos ficheros se mirarán opcionalmente para comprobar cualquier aspecto que no esté claro en el documento PDF.
- Un fichero de texto llamado `README` identificando todos los ficheros fuente incluidos con instrucciones claras para su compilación.

Los ficheros solicitados se deben entregar en un único archivo comprimido en formato `.tar.gz` o `.zip`. No se admitirán otros formatos distintos a los indicados.

Para el envío, se tiene que utilizar la herramienta de *Tareas* del Aula Virtual y envíar el trabajo dentro de la fecha prevista. Se permite reenviar dicho fichero dentro del plazo establecido, por si después de haberlo enviado se detectasen errores.

5.2. Recursos

En la zona de Recursos del Aula Virtual, dentro de la carpeta denominada *Practicas/Practica*2 puedes encontrar todo lo necesario para realizar esta práctica.

6. Criterios de evaluación

Esta práctica se evaluará teniendo en cuenta los siguientes criterios de evaluación (sobre 10 puntos):

- Realización y corrección de los apartados pedidos (*6 puntos*).
- Presentación y claridad de las explicaciones (*1 punto*).
- Aportación de ideas originales en las explicaciones a los apartados realizados (*1 punto*).
- Realización de alguna actividad *extra* relacionada con los ejercicios de la práctica demostrando curiosidad (*1 punto*).
- Comentarios sobre la práctica, incluyendo aspectos negativos y positivos (*1 punto*).

Créditos

- Esta práctica toma ideas de las secciones 3.1 y 3.2 del libro «*Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors*». 2nd Edition. Colfax International, 2015.
- El tutorial «*A "Hands-on" Introduction to OpenMP*» de Tim Mattson, de Intel Corp., que puedes encontrar dentro del material de prácticas en una carpeta denominada OpenMP Tutorial puede ser de utilidad, junto a los diversos ejercicios resueltos que tiene.
- La página web de *The National Center for Supercomputing Applications (NCSA)* `http://www.ncsa.illinois.edu` tiene unos tutoriales muy interesantes (`https://www.citutor.org/browse.php`) relacionados con la vectorización, paralelización, evaluación de rendimiento, etc
- Asimismo, en la página web de *The Cornell University Center for Advanced Computing (CAC)* `https://cvw.cac.cornell.edu/topics` hay un material relacionado con la vectorización y paralelización de aplicaciones muy adecuado.