CARACTERIZACIÓN DE PROBLEMAS A SOLUCIONAR MEDIANTE PROGRAMACIÓN MULTIPROCESO DE MEMORIA COMPARTIDA

AUTOR
ERIKA VIVIANA RIAÑO BEJAR

DIRECTOR
JOSE ORLANDO MALDONADO BAUTISTA

DEPARTAMENTO DE INGENIERÍAS ELÉCTRICA ELECTRÓNICA SISTEMAS Y TELECOMUNICACIONES
FACULTAD DE INGENIERÍAS Y ARQUITECTURA
INGENIERIA DE SISTEMAS
UNIVERSIDAD DE PAMPLONA
PAMPLONA
2016

Tabla de Contenido

Lista De Figuras

Lista De Tablas

1. Introducción

1.1. Definición del problema y justificación

1.1.1. Planteamiento Del Problema.

Actualmente se ha venido desaprovechando gran parte del potencial de las arquitecturas modernas que nos ofrecen las diferentes marcas de pc, ya que a pesar de que estas cuenten con varios procesadores y/o núcleos el algoritmo programado se ejecuta como si fuese un equipo monoprocesador, es por ello que se hace necesario la utilización de herramientas tales como openMP que provee directivos de compilador para la programación de multihilos, variables de ambiente y rutinas de biblioteca que controlan paralelismo en nivel de cores y nivel de procesadores, logrando con ello la caracterización de problemas computacionales susceptibles a solucionar mediante programación multiproceso de memoria compartida.

1.1.2. Justificacion.

En la actualidad la Universidad de Pamplona no cuenta con una infraestructura completamente definida que permita realizar procesos de alta complejidad. Si bien se han adquirido elementos de hardware que puden ser apovechados para tal fin, e require la instalación del software adecuado y la configuración de hardware y software, así como la capacitación y entrenamiento del recurso humano para sacar mayor provecho de estos dispositivos. Este trabajo espera aportar un material exploratorio, que pueda ser tomado como base para afrontar problemas concretos que saquen provecho al modelo de

programación de memoria compartida, mostrando como se pueden abordar ciertos problemas mediante programación paralela en OpenMP.

1.2. Objetivos

1.2.1. Objetivo general

Identificar las características propias de los problemas que pueden ser optimizados mediante la programación multiproceso de memoria compartida.

1.2.2. *Objetivos específicos*

- Revisar el estado del arte del paradigma de programación multiproceso de memoria compartida mediante la API OpenMP

- Analizar y caracterizar los problemas que son susceptibles de ser optimizados mediante este paradigma.

- Seleccionar problemas tipo dentro de la caracterización realizada y comparar el rendimiento de la implementación en paralelo frente al serial tradicional.

- Elaborar un manual de procedimientos para programación bajo este paradigma.

2. Enfoque Metodológico

Por la naturaleza de la presente investigación se adopta un enfoque **cuantitativo**, con un alcance de tipo **exploratorio**, ya que la idea surge por el interés del grupo, docentes, estudiantes e investigadores del grupo de investigación en Ciencias Computacionales, en abordar un área que si bien, lleva años de desarrollo es relativamente nueva en el contexto del programa de Ingeniería de Sistemas de la Universidad de Pamplona en la sede de Pamplona, siendo de esta forma, un área poco estudiada (en el contexto local), de manera que entre los propósitos de esta investigación está el de preparar el terreno para nuevos estudios. Dicho esto, y de acuerdo con lo consignado en (Sampieri, Collado, & Lucio, n.d.) dado el enfoque adoptado, se distinguen las siguientes etapas, cuyo desarrollo se explica en cada caso:

Planteamiento del problema, que se ha establecido mediante la descripción del problema, la justificación de la investigación y la formulación de objetivos de la misma.

Desarrollo de la perspectiva teórica, realizada mediante la revisión de la literatura para establecer un contexto, un marco teórico y un estado del arte.

La descripción del alcance de la investigación. Esta llegará a ser de tipo exploratorio, como se ha mencionado, pues se considera que al menos en el contexto local, esta es un área poco estudiada, que puede aportar soluciones innovadoras a

otros problemas ya abordados, y se espera que con el desarrollo del mismo se prepare el terreno para emprender nuevos estudios.

Formulación de la hipótesis de investigación: A saber: “Es posible identificar un conjunto de problemas característicos que se pueden resolver mediante el modelo de programación paralelo de memoria compartida”

Diseño de investigación: Aunque el alcance es exploratorio, se realizan un conjunto de pruebas experimentales, para determinar el rendimiento de algoritmos en problemas seleccionados. Para lo cual las variables a medir son, tiempo de ejecución, aceleración.

La selección de la muestra, recolección y análisis de datos, y presentación de resultados: Se explican los apartados 6 y 7 de este documento.

3. Estado Del Arte

Se ha realizado una revisión con las publicaciones más recientes relacionadas con el desarrollo de algoritmos mediante OpenMP. Se han encontrado artículos valiosos que proponen metodologías para la programación paralela con OpenMP, algunas aplicaciones recientes utilizando dicho modelo, mejoras sugeridas al modelo actual, y algunas propuestas para la enseñanza del paradigma.

En (Dorta, García, González, León, & Rodríguez, n.d.), (Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, 2009), (Eduard Ayguad, Nawal Copty, Duran Alejandro, Hoeflinger Jay, Yuan Lin, Massaioli Federico, Su Ernesto, Unnikrishnan Priya, n.d.), (Julio Monetti, n.d.)
se propone algunas metodologías específicas para el desarrollo de algoritmos paralelos. Por ejemplo, en (Julio Monetti, n.d.) muestra el uso y análisis de metodologías no clásicas de computación a través de programación concurrente observando características estructurales de software y hardware con el fin de asistir a aquellos usuarios que requieren computación de alto rendimiento. En (Dorta et al., n.d.), se presenta una breve introducción de la técnica divide y vencerás desde el punto de vista secuencial, abordándose mediante un ejemplo las diferentes posibilidades de paralelización, incrementándose el nivel de dificultad en la implementación de memoria compartida, por último se analiza el rendimiento de la propuesta secuencial frente a la paralela.
En (Eduard Ayguad, Nawal Copty, Duran Alejandro, Hoeflinger Jay, Yuan Lin, Massaioli Federico, Su Ernesto, Unnikrishnan Priya, n.d.) presenta una propuesta para definir el paralelismo de tareas mediante OpenMP, teniéndose en cuenta tres objetivos: simplicidad de

uso, simplicidad de especificación y consistencia con el resto de OpenMP. El documento muestra cómo utilizar la propuesta para paralelizar algunos de los ejemplos clásicos de paralelismo de tareas, como persecución de puntero y funciones recursivas. En (Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, 2009) se presenta diseño del modelo de tareas OpenMP, el documento presenta el diseño, evaluación e integración del modelo de tareas, incluye ejemplos y una discusión de la lógica detrás de varias opciones de diseño, se compara un prototipo de implementación del modelo de tasking con los modelos existentes y lo evaluación de varias aplicaciones, dicha comparación proporciona expresividad, flexibilidad, rendimiento y escalabilidad en el modelo de tareas.

En (Haoqiang Jina, Dennis Jespersena, Piyush Mehrotraa, Rupak Biswasa, Lei Huangb, 2011), (Patrick Carribault, Marc Pérache, 2010), (Jiangzhou He & Zhizhong, 2015), (Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, 2008) se proponen algunas aplicaciones y mejoras a los modelos de programación de memoria compartida. Por ejemplo, En (Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, 2008) los autores presentan la posibilidad de mejorar la afinidad de datos e hilos en programas OpenMP. En (Patrick Carribault, Marc Pérache, 2010) Se introduce una taxonomía extendida de la programación hibrida MPI/ OpenMP y un nuevo módulo al marco MPC que maneja un tiempo de ejecución OpenMP totalmente compatible con 2.5 a una implementación MPI 1.3. Las directrices de diseño e implementación permiten dos características: capacidad sobre suscripción incorporada con un rendimiento comparable a las implementaciones de última generación en puntos de referencia y programas OpenMP, y la posibilidad de ejecutar aplicaciones MPI / OpenMP híbridas con una sobrecarga limitada debido a la mezcla de dos modelos de programación diferentes. En

(Haoqiang Jina, Dennis Jespersena, Piyush Mehrotraa, Rupak Biswasa, Lei Huangb, 2011), Muestra un enfoque híbrido de programación de memoria compartida y memoria distribuida, una combinación de dos modelos de programación OpenMP y OpenMPI, observándose el desempeño de los benchmarks paralelos y dos aplicaciones usando este enfoque en sistemas basados en varios núcleos incluyendo un SGI Altix 4700, un IBM p575 + y un SGI Altix ICE 8200EX, además se exhibe nuevas extensiones de localidad de datos en OpenMP para mejorar la estructura jerárquica de las arquitecturas multicore. En (Jiangzhou He & Zhizhong, 2015), se propone NestedMP, un conjunto de directivas que amplía OpenMP. NestedMP especifica el número de subprocesos de cada rama anidada en paralelo de forma declarativa y permite que los sistemas de tiempo de ejecución vean toda la imagen de los árboles de tareas para realizar la asignación de tareas-núcleo basada en la localidad. Se implementó NestedMP en GCC 4.8.2, probando el rendimiento en un servidor SandyBridge de 8 vías. El resultado muestra que NestedMP mejora significativamente el rendimiento de la implementación de OpenMP de GCC.

Ejemplos de implementaciones recientes que utilizan el modelo de programación de de memoria compartida pueden ser consultados en (Atienza, Colmenar, & Garnica, 2010), (Auckenthaler et al., 2011) y (Frederico Pratasa, Pedro Trancosob, Leonel Sousaa, Alexandros, & Shid Guochun, 2012). En (Atienza et al., 2010), muestra la implementación de una heurística para el método louvain utilizando OpenMP, mediante gráficos del mundo real derivados de múltiples dominios de aplicación (internet, citas, biológicos), proporcionando velocidades absolutas de hasta 16x usando 32 hilos. En (Auckenthaler et al., 2011), se discuten las variantes de la transformación inversa de una matriz diagonal, mejorando la eficiencia paralela para un gran número de procesadores, así como la utilización por

procesador. Se modificó el algoritmo divide y vencerás de matrices tridiagonales simétricas de tal manera que pueda calcular un subconjunto de los pares propios a costo reducido. La eficacia de las modificaciones se demuestra con experimentos numéricos. En (Frederico Pratasa, Pedro Trancosob, Leonel Sousaa et al., 2012) , se expone la explotación del paralelismo de grano fino para tres aplicaciones con características distintas: una aplicación de bioinformática (MrBayes), una aplicación de dinámica molecular (NAMD) y una aplicación de base de datos (TPC-H). Los resultados indican que el rendimiento de la aplicación depende de las características de las arquitecturas paralelas y de los requisitos computacionales de las funciones básicas de las respectivas aplicaciones.

Algunas propuestas con enfoques didácticos, que pueden ser utilizados en la enseñanza de la programación en paralelo pueden consultarse en (Amit Amritkar & Tafti, 2013; Francisco Almeida, Domingo Giménez, José Miguel Mantas, n.d.). En (Amit Amritkar & Tafti, 2013; Marco Oliverio, William Spataro, Donato D'Ambrosio, Rocco Rongo , Giuseppe Spingola, 2011; Nasim Muhammad, n.d.) se muestra modelos implementados en un entorno de memoria compartida, a través de OpenMP, una interfaz de programación de aplicaciones que soporta la programación paralela, documentando y comparando el rendimiento de cada uno de los algoritmos de simulación, Se muestra que a medida que el número de núcleos de procesamiento aumentan mejora notablemente el tiempo de procesamiento. El (Francisco Almeida, Domingo Giménez, José Miguel Mantas, n.d.), se propone una estructura general de la enseñanza de la programación paralela y la inclusión de algoritmos paralelos en los grados de ingeniería informática. Así como la contribución al conocimiento y ampliación de OpenMP que en los últimos tiempos se ha convertido en la herramienta estándar para el desarrollo de programas en plataformas de multiprocesador de memoria compartida.

Recientemente se ha publicado en (Reinders & Jeffers, 2015), algunas formas eficaces para aprovechar mejor el potencial computacional de los sistemas con procesadores Intel Xeon Phi y procesadores Intel Xeon u otros procesadores multicore. Incluye resultados de alto rendimiento de los procesadores anteriormente nombrados en aplicaciones de química, ingeniería y ciencias ambientales.

4. Marco Teórico

4.1. Computación de alto rendimiento y programación de sistemas paralelos

4.1.1. Breve recuento histórico

El gobierno de los Estados Unidos ha desempeñado un papel clave en el desrrollo de computadoras de alto rendimiento. Durante la Segunda Guerra Mundial, el Ejército de U.S.A construyo la ENIAC para acelerar el cálculo de las mesas de artillería.

En los 30 años después de la Segunda Guerra Mundial, El gobierno de los U.S.A. utilizó computadoras de alto rendimiento para diseñar armas nucleares, romper códigos y realizar otras tareas. El primer supercomputador comenzó a utilizarse ampliamente con la introducción del Cray-l en 1976. El Cray-l era un procesador vectorial pipeline, no una computadora de múltiples procesadores, pero era capaz de realizar más de 100 millones de operaciones de punto flotante por segundo. A finales de los años setenta, los

supercomputadores aparecieron en las industrias intensivas en la capital. Las compañías petroleras aprovecharon a los supercomputadores para ayudarles a buscar petróleo, y los fabricantes de automóviles comenzaron a usar estos sistemas para mejorar el combustible para la eficiencia y seguridad de sus productos.

Diez años más tarde, cientos de corporaciones alrededor del mundo estaban usando supercomputadoras para apoyar sus empresas comerciales. La razón es simple: para muchas empresas, los cálculos más rápidos llevan a una ventaja competitiva. Simulaciones mas rapidas de choque pueden reducir el tiempo que un fabricante de automóviles necesita para diseñar un coche nuevo. Un diseño más rápido de fármacos puede aumentar el número de firma de patentes. Las velocidades informáticas han aumentado dramáticamente en los últimos 5 años. El ENIAC Podría realizar aproximadamente 350 mUltiplicaciones por segundo. Las supercomputadoras de hoy Son más de mil millones de veces menos, capaces de petfonn billones de punto flotante, Operaciones por segundo.

Los procesadores individuales son aproximadamente un millón de veces más rápido de lo que eran hace 50 años. La mayor parte del aumento de velocidad se debe a mayores velocidades de reloj que permiten una sola operación a realizarse más rápidamente. El aumento de velocidad restante se debe a una mayor concurrencia del sistema: que permite que el sistema funcione simultáneamente en múltiples operaciones. La historia de la computacion ha sido marcada por un rápido progreso tanto en estos frentes, como lo demuestran los microprocesadores contemporáneos de alto rendimiento.

La CPU Intel Pentium 4, por ejemplo, tiene velocidades de reloj muy superiores a I GHz, dos unidades aritméticas-lógicas (ALU s) registradas en el doble de la velocidad de reloj del

procesador central, y soporte extensivo de hardware para la ejecución especulativa fuera de orden de las instrucciones.

¿Cómo pueden los superordenadores de hoy ser mil millones de veces más rápido que el ENIAC, si procesadores individuales son sólo un millón de veces más rápido? La respuesta es simple: El aumento de la velocidad restante mil veces se logra mediante la recogida de mil procesadores en un sistema integrado capaz de resolver problemas individuales, más rápido que una sola CPU; Es decir, un ordenador paralelo.

El significado de la palabra superordenador, entonces, ha cambiado con el tiempo. En 1976 la supercomputadora significaba un Cray-l, una computadora de una sola CPU con un alto rendimiento pipeline Procesador vectorial conectado a un sistema de memoria de alto rendimiento.

Hoy en día, el supercomputador significa una computadora paralela con mil CPUs. La invención del microprocesador es un acontecimiento que la desaparición de minicomputadoras y mainframes tradicionales y estimuló el de ordenadores paralelos de bajo coste. Desde mediados de los años 90, los fabricantes de microprocesadores han mejorado el rendimiento de sus procesadores de gama alta en una tasa del 50 por ciento, manteniendo los precios más o menos constantes, El rápido aumento en velocidades de microprocesador ha cambiado completamente la cara de la informática.

Los servidores basados en microprocesadores ahora ocupan el papel anteriormente desempeñado por minicomputadoras construidas fuera de los arreglos de ventanas o fuera de la plataforma de la lógica. Incluso las computadoras mainframe se están construyendo a partir de microprocesadores. (Michael J, n.d.).

4.1.2. ¿Qué es computación de alto rendimiento?

Se usa el término más general, "Computación de alto rendimiento ", término que incluye una variedad de tipos de máquina.

Una clase de HPC consiste en máquinas poderosas, diseñadas para aplicaciones numéricas. La potencia de una computadora a gran escala proviene de la combinación de componentes electrónicos de muy alta velocidad y la arquitectura especializada. La mayoría de los diseños usan una combinación de "procesamiento vectorial" y "paralelismo" en su diseño. Un procesador vectorial es una unidad aritmética del ordenador que produce una serie de cálculos similares en un conjunto superpuesto de línea de moda, (Muchos cálculos científicos se pueden establecer de esta manera.)(John H. Gibbons, 1989)

El paralelismo utiliza varios procesadores, suponiendo que un problema se puede dividir en grandes piezas que se pueden calcular en procesadores separados.

En la actualidad, los grandes HPC de mainframe como los ofrecidos por Cray, IBM, son sólo modestamente paralelos, tener tan pocos como dos hasta ocho procesadores. La tendencia es hacia más paralelo los procesadores en estos grandes sistemas. Algunos expertos Anticipan hasta 512 máquinas de procesador apareciendo en un futuro próximo. (John H. Gibbons, 1989)

4.1.3. ¿Qué es computación paralela?

Un equipo paralelo es un sistema de procesador múltiple que supone programación paralela. Existen dos categorías importantes las multicomputadoras y los multiprocesadores centralizados. El primero es un ordenador paralelo construido a partir de múltiples ordenadores y de toda una red de interconexión. Los procesadores de diferentes computadores interactúan transmitiendo mensajes entre sí. El segundo denominado a su vez multiprocesador simétrico o SMP es un sistema integrado en el que todas las CPU comparten el acceso a una memoria global. Esta memoria compartida admite comunicación y sincronización entre procesadores. (Michael J, n.d.)

4.1.4. Arquitecturas paralelas

La taxonomía de Flynn es la clásica clasificación usada en computación paralela, la cual usa ideas familiares al campo de la computación convencional para proponer una taxonomía de arquitecturas de computadores. Se basa en el análisis del flujo de instrucciones y de datos, los cuales pueden ser simples o múltiples, originando la aparición de 4 tipos de máquinas. Es decir, esta clasificación está basada en el número de flujos de instrucciones y de datos simultáneos que pueden ser tratados por el sistema computacional durante la ejecución de un programa. Un flujo de instrucción es una secuencia de instrucciones transmitidas desde una unidad de control a uno o más procesadores. Un flujo de datos es una secuencia de datos que viene desde un área de memoria a un procesador y viceversa. Se pueden definir las variables ni y nd como el número de flujos de instrucciones y datos, respectivamente, los cuales pueden ser concurrentemente procesados en un computador. Según eso, las posibles categorías son:

SISD (Single Instruction Stream, Single Data Stream)

Representa la máquina de Von-Neumann, en la cual un único programa es ejecutado usando solamente un conjunto de datos específicos a él. Compuesto de una memoria central donde se guardan los datos y los programas, y de un procesador (unidad de control y unidad de procesamiento), En este caso, ni = nd =1. En esta plataforma sólo se puede dar un tipo de paralelismo virtual a través del paradigma de multitareas, en el cual el tiempo del procesador es compartido entre diferentes programas.(J.Aguilar, 2006)

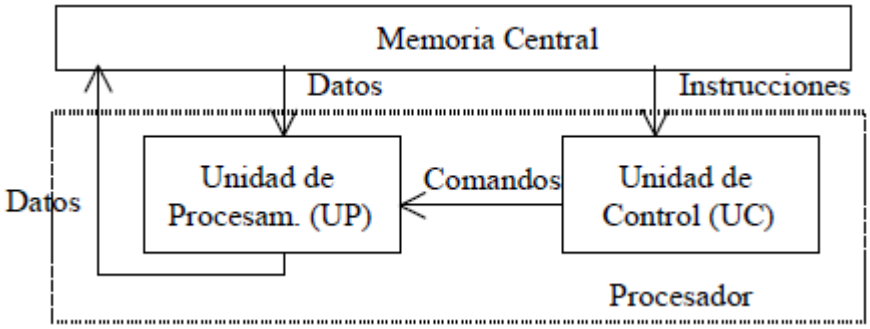


Figura 1. Arquitectura SISD (Maquina Von Neumann) (J.Aguilar, 2006)

SIMD (Single Instruction Stream, Multiple Data Stream)

Arreglo de elementos de procesamiento, todos los cuales ejecutan la misma instrucción al mismo tiempo. En este caso, ni =1 y nd > 1. El enfoque de paralelismo usado aquí se denomina *paralelismo de datos*. Los arreglos de procesadores son típicos ejemplos de esta clase de arquitectura. En estas arquitecturas, un controlador recibe y decodifica secuencias de instrucciones a ejecutar, para después enviarlas a múltiples procesadores esclavos.

El arreglo de procesadores procesa los datos que llegan a los diferentes procesadores, usando la instrucción enviada por el controlador. Los procesadores

están conectados a través de una red. Los datos a tratar pueden estar en un espacio de memoria que es común a todos los procesadores o en un espacio de memoria propio a cada unidad de procesamiento. Todos los procesadores trabajan con una perfecta sincronización. SIMD hace un uso eficiente de la memoria, y facilita un manejo eficiente del grado de paralelismo. La gran desventaja es el tipo de procesamiento (no es un tipo de procesamiento que aparece frecuentemente), ya que el código debe tener una dependencia de datos que le permita descomponerse.(J.Aguilar, 2006)

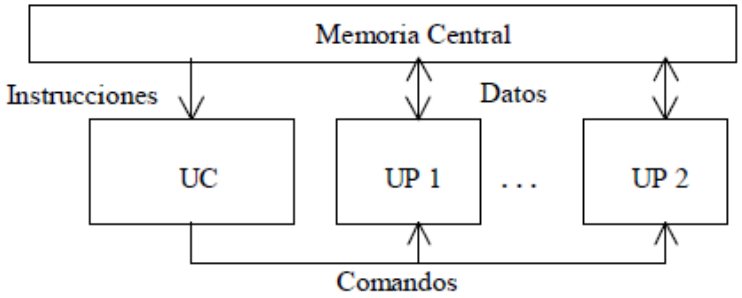


Figura 2. Arquitectura SIMD (J.Aguilar, 2006)

MISD (Multiple Instruction Stream, Single Data Stream)

Estas son computadoras con elementos de procesamiento, cada uno ejecutando una tarea diferente, de tal forma que todos los datos a procesar deben ser pasados a través de cada elemento de procesamiento para su procesamiento. En este caso, $n_i > 1$ y $n_d = 1$. Las implementaciones de esta arquitectura no existen realmente, excepto ciertas realizaciones a nivel de mecanismos de procesamiento tipo encauzamiento (*pipelining* en inglés) (internamente en los procesadores RISC, etc.) o sistemas de tolerancia a fallas.

La idea es descomponer las unidades de procesamiento en fases, en donde cada una se encarga de una parte de las operaciones a realizar. De esta manera, parte de los datos pueden ser procesados en la fase 1 mientras otros son procesados en la 2, otros en la tres, y así sucesivamente. El flujo de información es continuo y la velocidad de procesamiento crece con las etapas. Este tipo de clasificación de Flynn puede ser incluida dentro de la clasificación SIMD si se asemeja el efecto de estas máquinas, al tener múltiples cadenas de datos (los flujos) sobre las etapas de procesamiento, aunque también puede ser visto como un modo MIMD, en la medida de que hay varias unidades y flujos de datos y cada etapa está procesando datos diferentes(J.Aguilar, 2006)

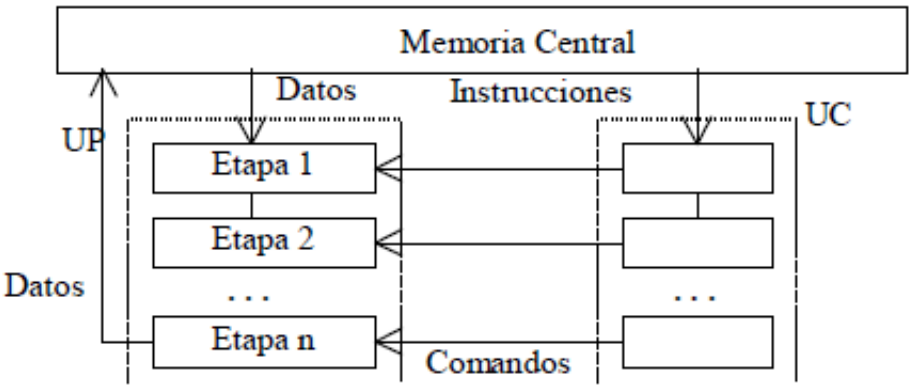


Figura 3. Arquitectura MIMD. (J.Aguilar, 2006)

MIMD (Multiple Instruction Stream, Multiple Data Stream)

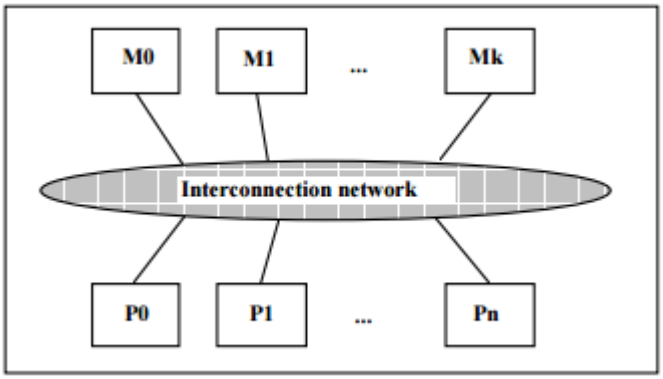


Figura 3MIMD (Sima et al., n.d.)

Es el modelo más general de paralelismo, y debido a su flexibilidad, una gran variedad de tipos de paralelismo puede ser explotados. Las ideas básicas son que múltiples tareas heterogéneas puedan ser ejecutadas al mismo tiempo, y que cada procesador opere independientemente con ocasionales sincronizaciones con otros. Está compuesto por un conjunto de elementos de procesamiento donde cada uno realiza una tarea, independiente o no, con respecto a los otros procesadores. La conectividad entre los elementos no se especifica y usualmente se explota un *paralelismo funcional*. La forma de programación usualmente utilizada es del tipo concurrente, en la cual múltiples tareas, quizás diferentes entre ellas, se pueden ejecutar simultáneamente. En este caso, ni > 1 y nd > 1. Muchos sistemas de multiprocesadores y sistemas de múltiples computadores están en esta categoría. Un computador MIMD es fuertemente acoplado si existe mucha interacción entre los procesadores, de lo contrario es débilmente acoplado.

Las MIMD se pueden distinguir entre sistemas con múltiples espacios de direcciones y sistemas con espacios de direcciones compartidos. En el primer caso, los computadores se comunican explícitamente usando pase de mensajes entre ellos

según una arquitectura NUMA (non-uniform memory access). En el segundo caso, al usarse una memoria centralizada, se trabaja con modelos UMA. En el primer caso se habla de MIMD a memoria distribuida y en el otro de MIMD a memoria compartida. Por el problema de gestión del acceso concurrente a la información compartida, se origina un cierto no determinismo en las aplicaciones sobre estas plataformas.

4.1.5. Técnicas y paradigmas de programación

Los paradigmas de programación permiten guiar el desarrollo eficiente del algoritmo paralelo, estos son estrategias para la resolución de problemas, el propósito es:

- Tener idea para resolver los diferentes problemas por medios equivalentes
- Desarrollar herramientas de diseño que ayude a programadores

Los paradigmas de programación paralela encapsulan información de los patrones de comunicación y los esquemas de descomposicion de tareas.(J.Aguilar, 2006)

- **Maestro/ Esclavo:**

  Se generan un conjunto de subtareas que es controlada por el maestro y realizadas por los esclavos. El MIMD trabaja bajo este enfoque.(J.Aguilar, 2006)

- **Divide y venceras:**

  Se divide un problema en varios subproblemas, se continúa dividiendo hasta que estos no puedan ser divididos, cuando los subproblems son resueltos hay una generación recursiva de subproblemas.

  Los subproblemas en computación paralela pueden ser resueltos al mismo tiempo en el caso que se tengan suficientes procesadores disponibles. (J.Aguilar, 2006)

- **Arbol binario:**

  Son un conjunto de elementos que se dividen en tres subconjuntos: contiene un elemento único llamado raíz, un subárbol izquierdo y un subárbol derecho.

  Los cálculos sobre N datos ($n_1, n_2, n_3 \dots n_N$) son descompuestos en cálculos sobre ($n_1, n_2, n_3 \dots n_{N/2}$) y ($n_{\frac{N}{2}+1}, \dots$) datos. El cálculo procede desde las hojas a la raíz del árbol. Cuando un nodo padre se descompone en varios nodos hijos durando varios tiempos de ejecucion, representa un problema debido a que los procesadores con nodos hijos que duren menos tiempo pueden llegar a estar largos periodos desocupados.(J.Aguilar, 2006)

4.1.6. Tecnologías disponibles

4.1.6.1. Hardware

Los procesadores más potentes actualmente:

Familia	Procesador	Caracteristicas
INTEL	Core i7-6950X	• Formado por diez núcleos con HyperThreading • Frecuencia máxima de 3 GHz que con turbo sube hasta los 3.5 GHz • Rendimiento insuperable en tareas muy pesadas como el renderizado de vídeo a resoluciones muy elevadas • Consumo de energía máximo es de 140W.
	Core i7-6900K	• Formado por ocho núcleos con HyperThreading • Frecuencia máxima de 3,2 GHz que con Boost sube hasta los 4000 MHz • Consumo de energía 140W.
	Core i7 6850K	• Formado por seis núcleos con HyperThreading • Frecuencia de 3600 MHz y que al máximo alcanza hasta los 3GHz**,** 40 LANES de PCI • Consumo de energía de 140W.
AMD	AMD FX-8350	• Formado por ocho núcleos basados en la arquitectura Piledriver • Frecuencia máxima de 4.2 GHz.
	El AMD FX-6300	• Formado por un procesador de seis núcleos basados en la arquitectura Piledriver • Frecuencia máxima de 4.1 GHz. • Consumo de 95W

Por ahora Intel se prepara para el próximo tick, un ciclo en el que se mejora la tecnología de integración y fabricación para pasar a procesos de 10 nanómetros. Eso implica más transistores en el mismo espacio. Se tendra tres familias de procesadores de 10 nm: la primera, Cannonlake, debutará en 2017. La segunda, Icelake, lo hará en 2018, mientras que la tercera y última será Tigerlake, que aparecerá en 2019. Tras ese ciclo se espera que Intel dé el salto a los microprocesadores de 5 nm en 2020.

En cuanto a tecnología de servidores tenemos:

Empresa	Equipo	Descripcion	Procesador
IBM	Power Systems S821LC Power System S822LC para Big Data	Ofrece dos procesadores POWER8 en un factor de forma de 1U para cubrir las cargas de trabajo de gran intensidad de cálculo actuales, y crecer con sus necesidades futuras. El servidor Linux de escalabilidad horizontal ofrece un diseño de servidor de alto rendimiento de datos y gran capacidad de almacenamiento, basado en estándares abiertos, para cubrir las cargas de trabajo de gran intensidad de cálculo actuales, y crecer con sus necesidades futuras. Para informática de alto rendimiento proporciona un rendimiento excepcional y ganancias en las aplicaciones con el nuevo POWER8 con	2 procesadores POWER8

	Power Systems S822LC Power System S824L	NVLink, que ofrece 2,8 veces el ancho de banda de CPU-GPU en comparación con los sistemas basados en x86. es el primer servidor Power Systems diseñado para incorporar adaptadores de alto vataje como GPUs de NVIDIA. Este servidor proporciona una nueva clase de tecnología que maximiza el rendimiento y la eficiencia para todos los tipos de cargas de trabajo de analítica de alto rendimiento e informática técnica, así como aplicaciones Java y big data	
DELL	PowerEdge R630 PowerEdge R730 PowerEdge	Con un rendimiento de 2U en un chasis compacto de 1U, ofrece una excepcional densidad y productividad, parte de la 13 generación de servidores, es ideal para la virtualización, lo que impulsa las aplicaciones empresariales de gran tamaño o la ejecución de las bases de datos transaccionales. ofrece una excelente funcionalidad en tan solo 2U de espacio de rack con la combinación de procesadores potentes, gran capacidad de memoria, opciones rápidas de almacenamiento y soporte del acelerador GPU, realiza sus funciones de manera excepcional en una variedad de entornos exigentes. El un servidor en rack básico de corta	2 Intel® Xeon® E5-2620 v4 de 2,1 GHz; memoria caché de 20 M; 8 GT/s QPI; Turbo; HT; 8 C/16 T (85 W); mem. máx. de 2133 MHz 2 Intel® Xeon® E5-2620 v4 de 2,1 GHz; memoria caché de 20 M; 8 GT/s QPI; Turbo; HT; 8 C/16 T (85 W); mem. máx. de 2133 MHz Intel® Xeon® E5-2630 v3

	R430	profundidad (24 in) que ofrece un máximo rendimiento de 2 sockets para centros de datos con restricciones de espacio	2,4 GHz, caché de 20 M, 8 GT/s QPI, Turbo, HT, 8 C/16 T (85 W) mem. máx. 1866 MHz
HP	HPE ProLiant DL20 Gen9	Es una plataforma de servidor de rack ideal para empresas y proveedores de servicios en crecimiento	Intel® Pentium®, Core i3 e Intel xeón E3-1200 v5
	HPE ProLiant DL560 Gen9	Es un servidor de alta densidad de cuatro sockets (4 S) con rendimiento equilibrado, más escalabilidad y confiabilidad en un chasis de 2 U. Ofrece mayor potencia de procesamiento, mayor capacidad de almacenamiento, ancho de banda de E/S de hasta siete ranuras PCIe Gen 3.0, hasta 3 TB de memoria, además de la inteligencia y simplicidad de la administración automatizada con HPE OneView y HPE iLO 4.	Intel® Xeon® E5-4600 v4/v3
	HPE ProLiant DL580 Gen9	Es el servidor x86 estándar de empresa de cuatro zócalos (4S) que ofrece un rendimiento superior, excelente fiabilidad y disponibilidad, y eficiencia de consolidación y virtualización increíbles, ofrece un rendimiento mejorado del procesador, hasta 6 TB de memoria, mayor ancho de banda de E/S (9 ranuras PCIe Gen 3.0) y 12 Gb/s de velocidades SAS.	Intel® Xeon® E7-4800/8800 v4/v3

LENOVO	X440	Un nodo de ejecución de 4 sockets de bajo coste y excelente rendimiento para aplicaciones informáticas que consumen una gran cantidad de memoria, virtualización y bases de datos, capacidad de memoria: Hasta 1,5 TB y 48 módulos DIMM, Almacenamiento interno: Hasta 2 TB	Intel® Xeon® E5-4600 v2 y 48 núcleos
	X240 M5	Componentes de hardware avanzados, nodo de ejecución estándar de 2 sockets, Capacidad de memoria: Hasta 1,5 TB con LRDIMM de 64 GB, Almacenamiento interno: Hasta 4,0 TB	Intel® Xeon® E5-2600 serie v3 y 36 núcleos

4.1.6.2. *Software*

En la actualidad hay muy poco software que hace uso del hardware paralelo, esto significa un problema debido a que no se puede confiar en el hardware y los compiladores para proporcionar un aumento en el rendimiento de la aplicación, Se requiere de mucho esfuerzo en el desarrollo de software tanto a nivel de desarrollo de sistemas operativos, compiladores, como librerías de programación. Los desarrolladores de software deben aprender a escribir programas que exploten las arquitecturas de memoria tanto compartida como distribuida.

La construcción del software en paralelo debe basarse en: dividir el problema en partes más pequeñas, resolviéndose cada una de manera concurrente al resto. Los programas paralelos surgen por el deseo de optimizar la performance en la resolución de un problema, es por ello que el diseño es más complejo.

La relación entre los datos y la independencia o dependencia de los resultados, determinan la multiplicidad de los algoritmos paralelos. Surgiendo preguntas tales como: ¿Cómo dividir el trabajo en tareas? ¿Cómo asignarles tareas a los procesadores?, la decisión tomada de uno influye para el proceso del siguiente y la forma de analizar cada una genera nuevos enfoques de computación paralela.(Piccoli, 2011).

Algunos de los lenguajes para la programación paralela son:

- X10: es un lenguaje de programación de alto rendimiento y alta productividad de desarrollo, desarrollado en IBM “Productive, Easy-to-use, Reliable Computing System”, supported by the DARPA High Productivity Computer Systems initiative.(Saraswat et al., 2012)

  X10 es un lenguaje de programación orientada a objetos. Diseñado para abordar el desafío central de productividad de computación de Rendimiento (HPC). (Saraswat et al., 2012)

- High performance fortran: es un lenguaje de programación cuyo nombre proviene de “FORmula TRANslator” (traductor de fórmulas), y fue desarrollado originalmente por IBM en 1954, con el objetivo de poder escribir programas de computo cientıfico en un lenguaje de alto nivel en vez de tener que recurrir a lenguaje de m´aquina o ensamblador. (Alcubierre, 2005).

Se utiliza principalmente en aplicaciones científicas y análisis numérico. Desde 1958 ha pasado por varias versiones, entre las que destacan FORTRAN II, FORTRAN IV, FORTRAN 77, Fortran 90, Fortran 95 y Fortran 2003. Las últimas versiones incluyen elementos de la programación orientada a objetos. (Santa Cruz, 2007)

El primer compilador de FORTRAN se desarrolló para una IBM 704 entre 1954 y 1957 por la empresa IBM, por un grupo liderado por John W. Backus. El lenguaje ha sido ampliamente adoptado por la comunidad científica para escribir aplicaciones con cómputos intensivos. La inclusión en el lenguaje de la aritmética de números complejos amplió la gama de aplicaciones para las cuales el lenguaje se adapta especialmente y muchas técnicas de compilación de lenguajes han sido creadas para mejorar la calidad del código generado por los compiladores de Fortran.(Santa Cruz, 2007)

- ZPL: es un lenguaje de programación paralelo que se desarrolló en la universidad de Washington entre 1992 y 2005. ZPL fue un contemporáneo de fortran de alto rendimiento (HPF), etiquetando una clase similar de aplicaciones mediante el apoyo a los cálculos paralelos de datos mediante operaciones en matrices de vista globales distribuidas entre Un conjunto de memorias de procesador distintas. ZPL se distinguió del HPF proporcionando un modelo de ejecución menos ambiguo, soporte para conjuntos de índices de primera clase y un modelo de rendimiento sintáctico que apoyaba la capacidad de identificar y razonar trivialmente sobre la comunicación. .

- Parlog: es un lenguaje de programación lógica en el sentido de que casi todas las definiciones y consultas pueden ser leídas como una sentencia de lógica predicada.

Las relaciones PARLOG se dividen en dos tipos: relaciones de una sola solución y relaciones de todas las soluciones. Una conjunción de llamadas de una sola solución de relación se puede evaluar en paralelo con las variables compartidas que actúan como canales de comunicación para el paso de enlaces parciales. Sólo se calcula una solución para cada llamada, utilizando el no determinismo de elección comprometida. Una conjunción de llamadas de relación de todas las soluciones se evalúa sin comunicación de enlaces parciales, pero todas las soluciones pueden ser encontradas por una exploración or paralela de las diferentes trayectorias de evaluación. Un constructor de conjuntos proporciona la interfaz principal entre las relaciones de una sola solución y las relaciones de todas las soluciones.

- Nesl: está destinado a ser utilizado como una interfaz portátil para la programación de una variedad de supercomputadores paralelo y vectorial, y como base para la enseñanza de algoritmos paralelos. El paralelismo se suministra a través de un conjunto simple de construcciones paralelas de datos basadas en vectores, que incluyen un mecanismo para aplicar cualquier función sobre los elementos de un vector en paralelo y un rico conjunto de funciones paralelas que manipulan vectores. (Blelloch, 2012)

  NESL admite totalmente vectores anidados y paralelismo anidado - la capacidad de tomar una función paralela y aplicarla en múltiples instancias en paralelo. El paralelismo anidado es importante para la implementación de algoritmos con estructuras de datos complejas y que cambian dinámicamente, tal como se requiere en muchos algoritmos de matriz gráfica y escasa. (Blelloch, 2012)

NESL proporciona un mecanismo para calcular el tiempo de funcionamiento asintótico para un programa en varios modelos de máquinas paralelas, incluyendo la máquina paralela de acceso aleatorio (PRAM). Esto es útil para estimar tiempos de ejecución de algoritmos en máquinas reales y, cuando se enseñan algoritmos, para suministrar una correspondencia estrecha entre el código y la complejidad teórica. (Blelloch, 2012).

Diferentes API's para programación paralela y/o distribuida:

- OpenMP:

  Es una interfaz de programa de aplicación (API) que se puede utilizar para dirigir explícitamente paralelismo de *memoria* compartida *multi-* subprocesos.

  Compuesto de tres componentes primarios de API:

  - Directivas del compilador
  - Runtime Library Routines
  - Variables de entorno

- OpenMPI

  Es una *especificación* para los desarrolladores y usuarios de las bibliotecas de paso de mensajes. Es una especificación de lo que debería ser una biblioteca. Se dirige principalmente al *modelo de programación paralelo de paso de mensajes: los* datos se

trasladan del espacio de direcciones de un proceso a otro de proceso a través de operaciones cooperativas en cada proceso.

El objetivo de la interfaz es proporcionar un estándar ampliamente utilizado para escribir programas de paso de mensajes. La interfaz intenta ser:

- Práctico
- Portátil
- Eficiente
- Flexible

- OpenCL

(Open Computing Language, en español lenguaje de computación abierto) es un estándar de programación para propósitos generales en sistemas heterogéneos, es decir a sistemas computaciones que constan de distintos tipos de arquitecturas tales como procesadores Intel o,AMD, procesadores de tarjetas gráficas, FPGA's . Consta de una interfaz de programación de aplicaciones y de un lenguaje de programación y esta está basado en lenguaje C. En OpenCL es posible crear aplicaciones con paralelismo tanto a nivel de datos como paralelismo a nivel de tareas. la finalidad de OpenCL es tener un lenguaje común entre los distintos dispositivos que pueden tener arquitecturas y formas de programación diferentes entre si.

5. Programación de memoria Compartida con OpenMP

5.1. Introducción

OpenMP es una API para la programación en paralelo de memoria compartida, esta contiene directivas de compilador para la programación de multihilos, variables de ambiente y librerías que controlan la ejecución en paralelo, donde cada hilo tiene acceso a toda la memoria disponible, viéndose el sistema como una colección de cores o procesadores donde estos tienen acceso a la misma memoria global compartida, permitiendo así el aprovechamiento incremental del código. OpenMP trabaja con estándares como Fortran, C/ C++ combinando algoritmos de forma serial y paralelo dentro del mismo código fuente.

OpenMP se basa en el modelo de ejecuión fork-join, que permite dividir una tarea con alto costo computacional, en pequeñas subtareas, de menor coste, que, al ser ejecutadas en hilos independientes, permite aprovechar los recursos en máquinas multinucleo o multiprocesador, y reducir los tiempos de procesamiento.

5.2. El modelo de ejecución Fork-Join

El paralelismo fork join, es un modelo fundamental en la computación paralela, se remonta a 1963 y desde entonces ha sido ampliamente utilizado en computación paralela.

Cuando el programa se comienza a ejecutar el hilo master esta activo, el subproceso maestro ejecuta la secuencia de las partes del algoritmo. En aquellos puntos en los que se requiere operaciones paralelas, el thread maestro fork crea hilos adicionales. El hilo maestro y los hilos creados funcionan simultaneamente a traves de la seccion paralela, al final del control paralelo los hilos creados mueren y el flujo del control vuelve al hilo maestro.

El número de hilos activos es uno en el inicio y final del programa y puede cambiar dinamicamente durante la ejecucion del programa. Los programas van desde aquellos con un solo fork/join alrededor de un solo bucle a aquellos en los que la mayoria del segmento de codigo se ejecutan en paralelo.

5.3. Estructura de bloques y clausulas

OpenMP cuenta con un modelo de programación “orientado a directivas”, esto siginfica que existen un conjunto especial de instrucciones al preprocesador conocidas como *pragmas*. Los pragmas son añadidos dentro del código fuente, sin que necesariamente hagan parte de la especificación del lenguaje que lo soporta. Si un compilador no soporta los pragmas, este es libre de ignorarlo, lo que permite que un programa escrito en forma correcta utilizando OpenMP, pueda ser ejectutado en cualquier compilador de C, ya sea que este soporte o no el API.

5.4. Directivas o constructores

5.4.1. Pragma Omp Parallel

Los fragmentos de algoritmos contenidos en este forman un grupo de threads para iniciar la ejecución en paralelo

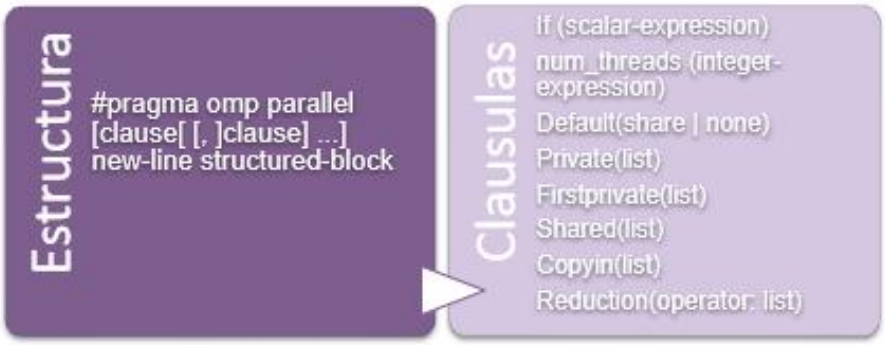


Figura 4. Esctructura y Clausulas Del Pragma Omp Parallel

5.4.2. Pragma Omp For

Permite la distribución de iteraciones al momento de crear el bucle y la ejecución de los subprocesos encontrados en el equipo de hebras.

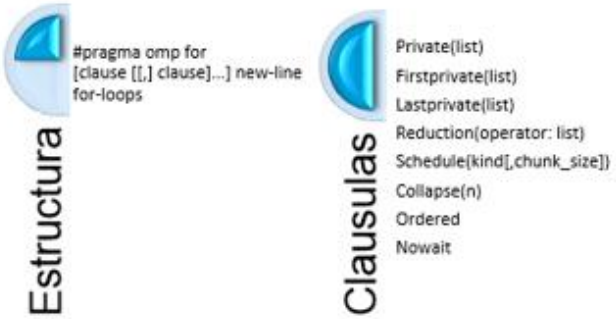


Figura 5. Estructura y Clausulas Del Pragma Omp For

5.4.3. *Pragma Omp Sections*

El grupo de bloques son distribuidos y ejecutados por el equipo de threads

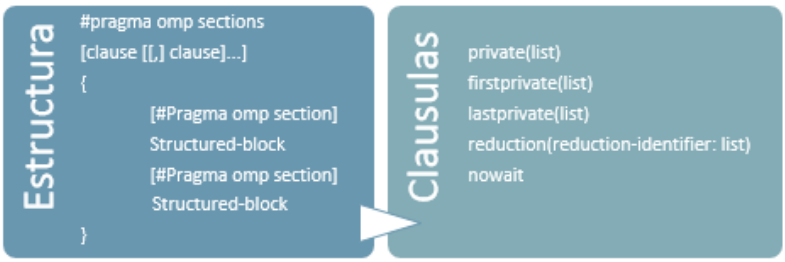


Figura 6. Estructura y Clausulas Del Pragma Omp Sections

5.4.4. Pragma Omp Single

Indica que la región paralelizada es ejecutada por una sola hebra del equipo.

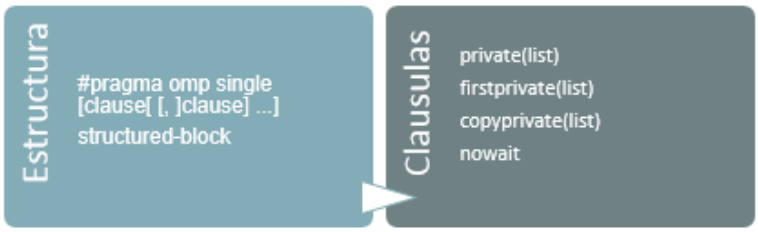


Figura 7. Estrcutura Y Clausulas Del Pragma Omp Single

5.4.5. Pragma Omp For/Sections

La abreviación (for/sections) de la construcción parallel contiene una o más loop/sections teniendo una sola declaración.

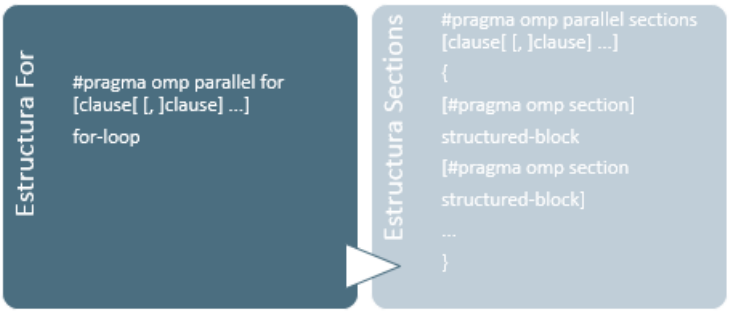


Figura 8. Esctructura Y Clausulas Del Pragma Omp For/Sections

5.4.6. Pragma Omp Task

Define una tarea. Dependiendo de las clausulas declaradas se crea el ambiente de datos para dicha tarea.

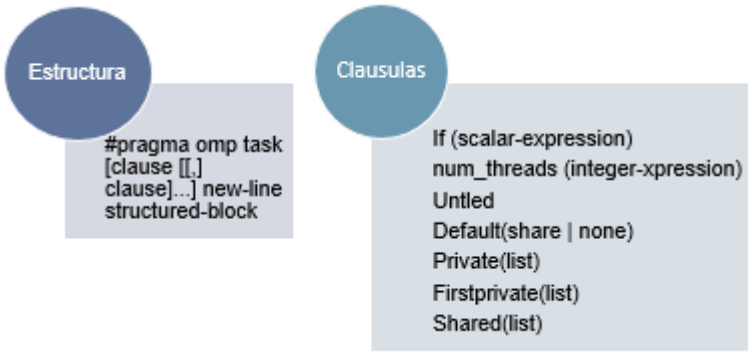


Figura 9. Estructura Y Clausulas Del Pragma Omp Task

5.4.7. Pragma Omp Taskwait

Suspende el trabajo de la tarea actual hasta que las subtareas terminen con su trabajo.

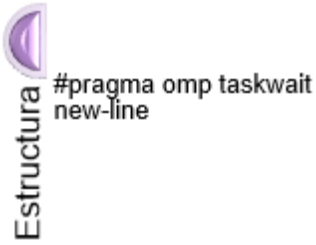


Figura 10. Estructura Del Pragma Omp TaskWait

5.5. Clausulas

5.5.1. Cláusulas de Compartición de Datos

5.5.1.1. Default (Shared | none)

Esta cláusula es usada para controlar atributos por defecto de variables de datos compartidos.

- Default(shared): Asigna atributos compartidos para todas las variables del constructor.
- Default (private): Hace todas las variables privadas por defecto

5.5.1.2. Shared (list)

Especifica los datos que deben ser compartidos entre los hilos ejecutados en la región asociada. Hay una única instancia de esas variables y cada thread lee o modifica el valor. Los elementos contenidos en la lista son compartidos entre los hilos del equipo.

5.5.1.3. Private(list)

Los elementos de la lista son replicados, así cada thread del equipo accede a una copia local de la variable, los cambios realizados en los datos no son visibles para los demás threads. Esta cláusula declara a los elementos de la lista en privados para la tarea.

5.5.1.4. *Firstprivate(list)*

Las variables contenidas en este constructor son privadas, siento estas inicializadas con el valor del elemento teniendo este el mismo nombre antes de la construcción.

5.5.1.5. *Lastprivate(list)*

Los elementos contenidos en esta cláusula son privados, los datos son accedidos después de que el constructor complete la ejecución. Es decir; el elemento original se actualiza terminada la región paralela.

5.5.1.6. *Reduction (operator: list)*

Especifica algunas formas de cálculos recurrentes ejecutados en paralelo sin modificar el código a partir de una lista de elementos separados por una coma.

A cada hilo se le da una copia privada de cada elemento, los valores finales de estos se combinan para el operador y el resultado se coloca en el valor original de la variable de reducción compartida.

La cláusula reduction debe cumplir con las siguientes condiciones:

- Tipo apropiado para el operador: bolean, int, float, double
- No debe tener tipo puntero
- Debe ser compartida

5.5.1.7. *Nowait*

Esta cláusula anula la sincronización implícita barrier al final de la asociación del constructor de trabajo compartido. Es decir; No hay sincronización de los threads al final de cada uno debido a que las operaciones no dependen del anterior.

Cuando los threads llegan al final de un constructor ellos inician otra tarea.

5.5.1.8. *Schedule*

Describe las iteraciones del bucle que se dividen entre los hilos del equipo. El programa predeterminado depende de la implementación.

STÁTIC

Las iteraciones de bucle se dividen en piezas de tamaño chunk y luego se asignan estáticamente a los hilos. Si no se especifica el fragmento, las iteraciones son iguales (si es posible) divididas contiguamente entre los subprocesos.

DYNAMIC

Las iteraciones de bucle se dividen en trozos de tamaño chunk, y se programan dinámicamente entre los hilos; Cuando un hilo finaliza un fragmento, se asigna dinámicamente otro. El tamaño del trozo predeterminado es 1.

GUIDED

Las iteraciones se asignan dinámicamente a los subprocesos en bloques que los subprocesos los solicitan hasta que no queden bloques por asignar. Similar a DYNAMIC

excepto que el tamaño de bloque disminuye cada vez que se entrega un paquete de trabajo a un hilo. El tamaño del bloque inicial es proporcional a:

Number_of_iterations / number_of_threads

Los bloques posteriores son proporcionales a

Number_of_iterations_remaining / number_of_threads

El parámetro chunk define el tamaño mínimo del bloque. El tamaño del trozo predeterminado es 1.

RUNTIME

La decisión de programación se difiere hasta el tiempo de ejecución por la variable de entorno OMP_SCHEDULE. Es ilegal especificar un tamaño de segmento para esta cláusula.

AUTO

La decisión de programación se delega en el compilador y / o en el sistema del tiempo de ejecución.

5.5.2. Cláusulas de Copia de Datos

5.5.2.1. Copyin (list)

Permite copiar el valor de los threads master de las variables privadas correspondiente a las variables privadas de otros threads. Al ser suplantada la cláusula copyin por una directiva, el

nombre de las variables privadas dentro de la copia privada de cada hebra esclavo es inicializados con el valor del thread master.

5.5.2.2. Copyprivate (list)

Transmite el valor de una variable privada de un thread a otros threads del equipo pertenecientes a la región paralela. Su principal uso es leer o inicializar datos privadas por otros threads.

5.6. Sincronización

La sincronización hace referencia a los componentes que se deben tener en cuenta para la ejecución de múltiples hilos, estos son: exclusión mutua y la sincronización de eventos.

La primera es usada cuando son modificados múltiples hilos por una variable compartida estos adquieren acceso a la misma modificando su integridad. OpenMP provee una exclusión mutua desde una directiva critical.

El segundo componente señala la ocurrencia de un evento por medio de múltiples hilos, un ejemplo para este caso es la directiva barrier; cuando un hilo alcanza la directiva espera a los demás hilos en ese punto y luego continua con la ejecución.

OpenMP provee la sincronización de otros constructores, algunos de estos muestran un modelo común de sincronización

5.6.1. Constructor Barrier

Cuando un thread alcanza la directiva espera a los demás threads a que lleguen a ese lugar y luego continua con la ejecución; es decir sincroniza en un punto a todos los threads. Muchos constructores implican un barrier en openMP, el compilador automáticamente inserta un barrier al final del constructor. Todos los threads esperan en un punto hasta que todos los del equipo de trabajo asociado con la construcción sea completado. La sintaxis en C/C++ es:

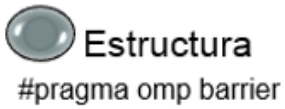


Figura 11. Estructura Del Constructor Barrier

Restricciones del constructor:

- Cada barrier debe encontrarse con todos los hilos en el equipo, o por ninguno
- La secuencia de tarea-compartidas y regiones barrier encontradas son iguales para todos los hilos en el equipo.

5.6.2. Constructor Ordered

Permite la ejecución de una estructura de bloques dentro de un loop paralelo en el orden secuencial especificado por las iteraciones del bucle. El código del bloque externo corre en paralelo, el hilo ejecuta la primera iteración de la región encontrada en el constructor, reconociendo la región sin esperar. Esta espera hasta que cada iteración de la secuencia sea completada. Además, es usado para ayuda a determinar los datos asociados al código. La sintaxis en C/C++ es:

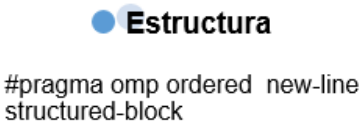


Figura 12. Estructura Del Constructor Ordered

5.6.3. Constructor Critical

Este constructor asegura que múltiples threads no actualicen los mismos datos compartidos simultáneamente, es decir; restringe que múltiples threads ejecuten el código en la misma región al mismo tiempo. La estructura de este constructor en C/C++ es:

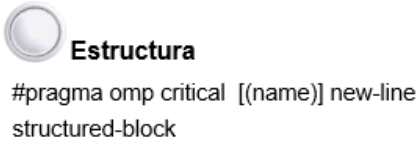


Figura 13. Estructura Del Constructor Critical

5.6.4. Constructor Atomic

Crea una memoria temporal para que múltiples threads actualicen las variables compartidas automáticamente los datos sin interferencia, para esto es necesario que el hardware soporte las lecturas de memoria local.

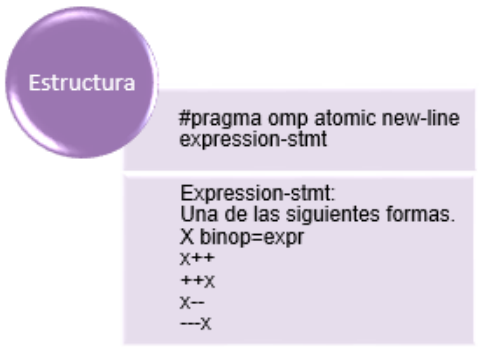


Figura 14. Estructura Del Constructor Atomic

Si un thread actualiza automáticamente un valor, a si mismo otro thread puedo hacerlo simultáneamente. Esta restricción aplica a todos los hilos ejecutados en el programa, para eso lo actualiza en una memoria localizada por esa directiva.

Cada vez que se aumente count++ el valor va incrementando hasta terminar de contar la totalidad de los threads.

```c
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    int count = 0;

    #pragma omp parallel
    {
        #pragma omp atomic
        count++;
    }

    printf("NUmero de hilos: %d\n", count);
}
```

```
erika@erika-HP-14-Notebook-PC: ~/Descargas
erika@erika-HP-14-Notebook-PC:~/Descargas$ gcc -fopenmp atomicc.c -o atomicc
erika@erika-HP-14-Notebook-PC:~/Descargas$ ./atomicc
NUmero de hilos: 4
erika@erika-HP-14-Notebook-PC:~/Descargas$
```

Figura 15. Ejemplo Del Constructor Atomic

5.6.5. Locks

La librería de rutinas locking runtime provee gran flexibilidad para sincronizar hechos usados en la sección critical o el constructor atomic. Hay dos tipos de locks:

- Simple locks: no puede ser locked si ya está en estado locked. Son declaradas omp lock t en C/C++.
- Nestable locks: puede ser locked en múltiples tiempos por los mismos threads. Son declaradas con el tipo omp nest lock t en C/C++

5.6.6. Constructor Master

Este constructor define que un bloque de código dentro de la región paralela debe ser ejecutado por un thread master, es decir un thread principal.

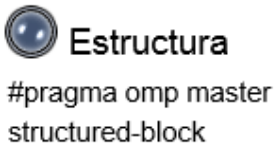


Figura 16. Estructura Del Constructor Master

5.7. Funciones de Librería

- **void omp_set_num_threads(int):**

Indica el número de threads que se deben usar en la región paralela, no especificada por la cláusula num_threads.

- **int omp_get_num_threads(void):**

Devuelve el número de threads que se estas usando actualmente en el equipo de trabajo.

- **int omp_get_max_threads(void):**

Devuelve el máximo de threads que se puedan usar en la región paralela.

- **int omp_get_threads_num(void):**

Devuelve el ID del thread actual.

- **int omp_get_num_procs(void):**

Devuelve la cantidad de procesadores disponibles.

- **int omp_in_parallel(void):**

Devuelve el valor de true o false para los siguientes casos:

True: Si el llamado a la librería se hace dentro de la región paralela.

False: en caso negativo a lo anterior.

- **int omp_set_dynamic(void):**

Permite activar o desactivar el ajuste dinámico del número de threads en una región paralela.

- **int omp_get_dynamic(void):**

Devuelve el valor de la variable para comprobar si el ajuste dinámico del número de threads está o no activado

- **int omp_set_nested(void):**

Permite activar o desactivar el paralelismo anidado, a través de la variable de control interno nest-var

- **int omp_get_nested(void):**

Devuelve el valor de la variable de control interno nest-var, en el caso de que este o no activo el paralelismo anidado

- **void omp_set_schedule (omp_sched_t kind, int modifier):**

Modifica las variables de control interno run-sched-var, debido a que afecta al planificador de las rutinas

    - Kind: Define el tipo de programación (auto, dynamic, guided, runtime, or static).
    - Modifier: De acuerdo al tipo de programación se elige el tamaño del fragmento, este valor debe ser entero positivo

- **void omp_get_schedule (omp_sched_t *kind, int *modifier):**

Retorna la (VIC) sched-var de la región paralela que lo está procesando

- **int omp_get_thread_limit(void):**

Retorna el máximo número de threads disponibles.

- **void omp_set_max_active_levels (int max_levels):**

Limita el número de niveles paralelos anidados, modifica la variable de control interno max-active-levels-var (VIC).

- **int omp_get_max_active_levels(void):**

Retorna el número máximo de niveles paralelos anidados.

- **int omp_get_level(void):**

Retorna la cantidad de regiones paralelas que encierra la llamada.

- **int omp_get_ancestor_thread_num (int level):**

Retorna el número del thread del antecesor del thread actual al que pertenece el nivel anidado.

- **int omp_get_team_size (int level):**

Retorna el tamaño equipo del thread antecesor del thread actual al que pertenece el nivel anidado

- **int omp_get_active_level(void):**

Devuelve el número del thread ancestro para un nivel anidado

5.8. Versiones de OpenMP.

- **Versión 2.0:**

  - ✓ Las comas están permitidas en las directivas de OpenMP.

  - ✓ Adición de la cláusula num_threads. Esta cláusula permite al usuario solicitar un número específico de hilos para una construcción en paralelo

  - ✓ La directiva threadprivate se ha extendido para aceptar el bloqueo estático.

  - ✓ Las matrices de longitud variable C99 son tipos completos, por lo que se pueden especificar donde se permiten tipos completos, por ejemplo, en las listas de particulares, Firstprivate, y lastprivate.

  - ✓ Una variable privada en una región paralela puede marcarse de nuevo privada en un anidado directiva

  - ✓ Se ha agregado la cláusula copyprivate. Proporciona un mecanismo privado para transmitir un valor de un miembro de un equipo a otro miembro. Es una alternativa al uso de una variable compartida para el valor cuando proporciona una variable compartida sería difícil (por ejemplo, en una recursión requiriendo una variable diferente en cada nivel). La cláusula copyprivate sólo puede aparecer en la directiva única.

- ✓ Adición de rutinas de temporización omp_get_wtick y omp_get_wtime similares a las rutinas MPI. Estas funciones son necesarias para realizar los tiempos del reloj de pared

- ✓ Un apéndice con una lista de comportamientos definidos por la implementación en OpenMP C / C ++ ha sido añadido.

- ✓ Se aclaró que el comportamiento de omp_set_nested y omp_set_dynamic Cuando omp_in_parallel devuelve distinto de cero no está definido

- ✓ Nido de directivas aclarado cuando se usa paralelo anidado

- ✓ Las funciones de inicialización de bloqueo y destrucción de bloqueo se pueden llamar en paralelo

- **Diferencias entre la versión 2.5 y 3.0**

  - ✓ El concepto de tareas ha sido añadido al modelo de ejecución de OpenMP.

  - ✓ Se ha agregado la construcción de tarea, un mecanismo para crear tareas explícitamente.

  - ✓ Se ha agregado la construcción taskwait, que hace que una tarea espere a que todas sus tareas secundarias se completen.

- ✓ El modelo de memoria OpenMP ahora cubre la atomicidad de los accesos a. La descripción del comportamiento del volátil eliminó la descarga.

- ✓ En la versión 2.5, había una sola copia de las variables nest-var, dyn-var, nthreads-var y Run-sched-var variables de control interno (ICVs) para todo el programa. En la versión 3.0, hay una copia de estos ICV por tarea. Como resultado, El omp_set_num_threads, omp_set_nested y omp_set_dynamic las rutinas de la biblioteca de tiempo de ejecución ahora tienen efectos especificados

- ✓ Se ha cambiado la definición de región paralela activa: en la versión 3.0 a región paralela está activa si es ejecutada por un equipo que consta de más de una.

- ✓ Las reglas para determinar el número de hilos utilizados en una región paralela.

- ✓ En la versión 3.0, la asignación de iteraciones a subprocesos en una construcción de bucle con un tipo de programa estático es determinista.

- ✓ En la versión 3.0, una construcción de bucle puede estar asociada con más de una construcción perfectamente bucle anidado. El número de bucles asociados puede ser controlado por el colapso.

- ✓ Los iteradores de acceso aleatorio y las variables de tipo entero sin signo, pueden utilizarse ahora como Iteradores de bucle en bucles asociados con una construcción de bucle.

- ✓ Se ha agregado el tipo de horario auto, que da la implementación a libertad de elegir cualquier posible asignación de iteraciones en una construcción de bucle a roscas en el equipo.

- ✓ Las matrices de tamaño supuesto de Fortran tienen ahora atributos predeterminados.

- ✓ En Fortran, firstprivate está ahora permitido como argumento al valor predeterminado.

- ✓ Para los elementos de lista en la cláusula privada, las implementaciones ya no están permitidas el almacenamiento del elemento de lista original para contener el nuevo elemento de lista en el subproceso maestro. Si no se intenta hacer referencia al elemento original de la lista dentro de la región paralela, su valor se define bien a la salida de la región paralela

- ✓ En la versión 3.0, las matrices asignables de Fortran pueden aparecer en privado,

  Firstprivate, lastprivate, reduction, copyin y copyprivate Cláusulas.

- ✓ En la versión 3.0, las variables de los miembros de la clase estática pueden aparecer en un threadprivate.

- ✓ La versión 3.0 deja claro dónde y con qué argumentos, constructores y destructores de variables de tipo de clase private y threadprivate.

- ✓ Las rutinas de la biblioteca de ejecución omp_set_schedule y omp_get_schedule

  Ha sido agregado; Estas rutinas respectivamente establecen y recuperan el valor de la Run-sched-var ICV.

- ✓ Se ha agregado el thread-limit-var ICV, que controla el número máximo de que participan en el programa OpenMP. El valor de este ICV se puede establecer con La variable de entorno OMP_THREAD_LIMIT y se recupera con la Omp_get_thread_limit rutina de biblioteca de tiempo de ejecución.

- ✓ Se ha agregado el valor máximo de los niveles activos-var ICV, que controla el número de regiones anidadas paralelas activas. El valor de este ICV se puede establecer con la variable de entorno OMP_MAX_ACTIVE_LEVELS y la Omp_set_max_active_levels rutina de biblioteca de tiempo de ejecución, y se puede recuperar con la rutina de la biblioteca de tiempo de ejecución omp_get_max_active_levels.

- ✓ Se ha agregado el stacksize-var ICV, que controla el tamaño de la pila para los subprocesos creados por la implementación de OpenMP. El valor de este ICV se puede establecer con la variable de entorno OMP_STACKSIZE.

- ✓ Se ha agregado la opción wait-policy-var ICV, que controla el comportamiento deseado de los threads en espera. El valor de este ICV se puede establecer con la variable de entorno OMP_WAIT_POLICY.

- ✓ Se ha agregado la rutina de biblioteca de tiempo de ejecución omp_get_level, que devuelve el número de regiones paralelas anidadas que encierran la tarea que contiene la llamada.

- ✓ Se ha agregado la rutina de la biblioteca de tiempo de ejecución omp_get_ancestor_thread_num, que devuelve, para un nivel anidado anidado del subproceso actual, el número de subproceso del ascendiente.

- ✓ Se ha agregado la rutina de la biblioteca de tiempo de ejecución omp_get_team_size, que devuelve, para un nivel anidado anidado del hilo actual, el tamaño del equipo de hilos al que pertenece el antecesor.

- ✓ Se ha agregado la rutina de la biblioteca de tiempo de ejecución omp_get_active_level, que devuelve el número de regiones paralelas anidadas que incluyen la tarea que contiene la llamada.

- ✓ En la versión 3.0, los bloqueos son propiedad de tareas, no de subprocesos

(OpenMP, 2013)

- **Diferencias entre la versión 3.0 y 3.1**

  - ✓ Las cláusulas finales y fusionables se agregaron a la construcción de tarea para admitir la optimización de entornos de datos de tareas.

  - ✓ Se añadió el constructor taskyield para permitir puntos de programación de tareas definidos por el usuario.

  - ✓ La construcción atómica se amplió para incluir lectura, escritura y captura de formularios, y se agregó una cláusula de actualización para aplicar la forma ya existente de la construcción atómica.

  - ✓ Se modificaron las restricciones de los entornos de datos para permitir que la intención (in) y constqualified tipos para la primera cláusula privada.

  - ✓ Se modificaron las restricciones de entorno de datos para permitir Firstprivate y lastprivate.

  - ✓ Se han añadido nuevos operadores de reducción min y max para C y C ++

  - ✓ Se aclararon las restricciones de anidación para no permitir estrechamente anidados OpenMP en regiones dentro de una región atómica. Esto permite

definir la region de forma coherente con otras regiones de OpenMP para que incluyan el código en la construcción atómica.

- ✓ La rutina de la biblioteca de tiempo de ejecución omp_in_final fue añadida para apoyar la especialización de las regiones de tareas finales.

- ✓ El nthreads-var ICV se ha modificado para que sea una lista del número de subprocesos a utilizar en cada nivel de región paralela anidada. El valor de este ICV todavía se establece con la variable de entorno OMP_NUM_THREADS, pero el algoritmo para determinar el número de hilos utilizados en una región paralela ha sido modificado para manejar una lista

- ✓ Se ha agregado el enlace-var ICV, que controla si los hilos están o no enlazados a los procesadores. El valor de este ICV se puede establecer con la variable de entorno OMP_PROC_BIND.

- ✓ Se ampliaron y aclararon descripciones de ejemplos

- ✓ Sustituyó el uso incorrecto de omp_integer_kind en interfaces Fortran con selected_int_kind (8)

  (OpenMP, 2013)

- **Diferencias entre la versión 3.1 y 4.0**

- ✓ Se realizaron varios cambios a lo largo de la especificación para proporcionar apoyo fortran 2003.

- ✓ La sintaxis de la matriz C / C ++ se amplió para soportar secciones de matriz.

- ✓ La cláusula proc_bind (consulte la Sección 2.5.2 en la página 49), el OMP_PLACES y la variable omp_get_proc_bind Rutina de ejecución se agregaron al subproceso de soporte de políticas de afinidad.

- ✓ Las construcciones SIMD se agregaron para soportar el paralelismo SIMD

- ✓ Construcciones de dispositivos, el OMP_DEFAULT_DEVICE Variable de entorno, omp_set_default_device, omp_get_default_device, Omp_get_num_devices, omp_get_num_teams, omp_get_team_num y Las rutinas omp_is_initial_device se agregaron para soportar la ejecución en dispositivos.

- ✓ Se eliminaron los puntos de programación de tareas definidos para la implementación de tareas no enlazadas.

- ✓ Se agregó la cláusula depend para soportar la tarea de dependencias.

- ✓ La construcción taskgroup se añadió para sincronización de tareas más flexible.

- ✓ Se amplió la cláusula de reducción y se agregó a apoyar las reducciones definidas por el usuario.

- ✓ La construcción atómica se amplió para Swap atómico con la cláusula de captura, para permitir una nueva actualización atómica y captura, y para apoyar las operaciones atómicas consecutivas consistentes con un nuevo seq_cst cláusula.

- ✓ La construcción de cancelación, la cancelación, la omp_get_cancellation rutina de ejecución y la OMP_CANCELLATION Variable de entorno para concepto de cancelación.

- ✓ La variable de entorno OMP_DISPLAY_ENV fue añadido para mostrar el valor de los ICV asociados con el entorno OpenMP Variables. (OpenMP, 2013)

6. Aspectos de diseño en algoritmos paralelos

6.1. Metodologías de desarrollo

6.1.1. Metodología de Foster

En la mayoría de los casos gran parte de los problemas de programación contienen distintas soluciones paralelas, siendo esta la mejor solución que la propuesta por los algoritmos secuenciales. La estructura de la metodología del proceso de diseño que se describirá a continuación contiene cuatro etapas tales como: particionado, comunicación, aglomeración y mapeo, es por ello que esta se le asignó el nombre de PCAM haciendo referencia a las iniciales de cada una. El particionado y la comunicación hacen énfasis en la concurrencia y escalabilidad y la aglomeración y mapeo se basan en la localidad y rendimiento.

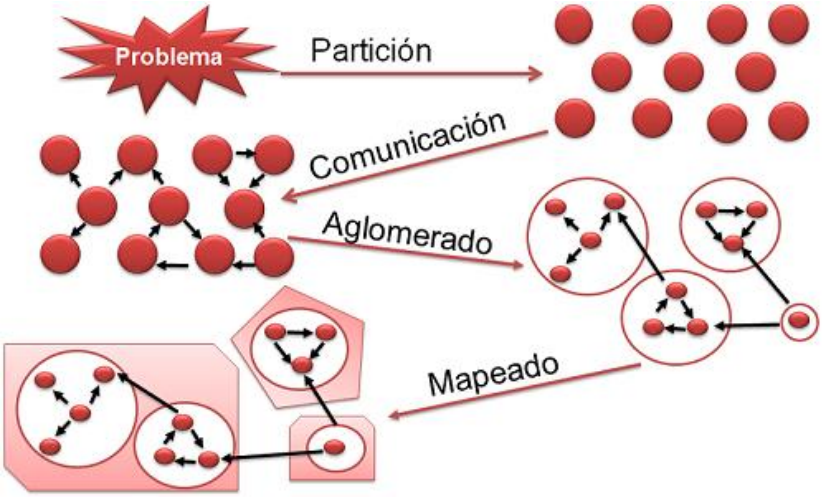


Figura 17. Modelo PCAM

Modelo PCAM: es una metodología de diseño para programas paralelos. Comenzando con un problema Especificación, desarrollamos una partición, determinamos requisitos de

comunicación, aglomeramos tareas, y finalmente se asignan las tareas a los procesadores.(Foster, 1995)

- **Particionado:**

  El cálculo asociado con el problema y los datos del mismo se descomponen en partes muy pequeñas denominadas tareas, según el tamaño de estas se les llama granularidad fina (muchas tareas pequeñas) y granularidad gruesa (pocas tareas grandes). La descomposición del grano fino proporciona mayor flexibilidad en la ejecución de algoritmos paralelos.

  En esta etapa se tienen en cuenta dos tipos de descomposición:

  - **Descomposición De Domino (salida/entrada/bloques):**

    En este caso primero se descompone los datos de un problema en pequeños pedazos del mismo tamaño si es necesario, luego se divide la computación que se va a realizar a través de la asociación del cálculo con los datos.

    Esta partición consigue un número de tareas logrando algunos datos y operaciones sobre los mismos. Habrá casos en los que una operación exija datos de diferentes tareas, para ello es necesario la comunicación de los mismos para moverlos.

Diversas formas de descomposición de los datos se pueden dar tales como: la entrada al programa, la salida de los cálculos del programa y los valores intermedios, lo importante para ello es centrarse en la estructura de datos más grandes o a la que se accede con mayor periodicidad.(Foster, 1995)

Los problemas que se le pueden dar solución a cada uno de los dominios de descomposición son:

Datos de Salida: Diferencias finitas de Jacobi

$$X_{i,j}^{t+1} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i,j+1}^{(t)}}{8} \quad (1)$$

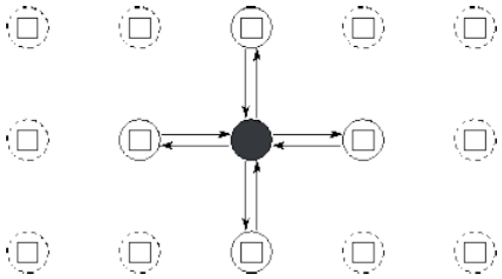


Figura 18. Diferencias Finitas De Jacobi (Foster, 1995)

En la Figura 17. Tarea y estructura de canal para un cálculo de diferencias finitas bidimensionales con Plantilla de actualización de cinco puntos. En esta simple formulación de grano fino, cada tarea encapsula un solo elemento de una cuadrícula bidimensional y debe enviar su valor a cuatro vecinos y recibir Valores de cuatro vecinos. Sólo se muestran los canales utilizados por la tarea sombreada.(Foster, 1995)

Datos de entrada: Producto escalar de dos vectores de longitud n con x tareas.

$$\boldsymbol{pe_{pid}} = \sum_{j=pid\frac{n}{t}}^{(pid+1)\frac{n}{t}-1} X_j Y_j \quad (2)$$

$$\boldsymbol{pe} = \sum_{pid=0}^{t-1} pe_{pid} \quad (3)$$

Basado en bloques: multiplicación matriz-vector ($X = AB$)

- **Descomposición Funcional:**

Representa una forma diferente y complementaria de pensar los problemas. Por esta sola razón, debe considerarse al explorar posibles algoritmos paralelos. Un enfoque en los cálculos que se van a realizar a veces puede revelar la estructura de un problema, y por lo tanto las oportunidades de optimización, que no sería obvio a partir de un estudio de datos por sí solo.

El enfoque inicial se centra en el cálculo a realizarse en lugar de los datos manipulados por el cálculo, si se obtiene éxito de dividir el cálculo en tareas disjuntas, se procede a examinar los requisitos de datos de esas tareas. Estos datos son disjuntos en el caso dado que la partición sea completa. Puede suponerse significativamente en cuyo caso será necesaria una comunicación considerable para evitar la replicación de datos. Esto es a menudo una señal de

que un enfoque de descomposición de dominio debe ser considerado en su lugar.(Foster, 1995)

Es decir, en otras palabras, el enfoque de esta descomposición se dará de la siguiente forma:

- Divide el cálculo en tareas disjuntas
- Examinan los requisitos de los datos de esas tareas
  - ✓ Si los datos son disjuntos el particionamiento es completo
  - ✓ Si los datos no son disjuntos el particionamiento es incompleto. Es necesario la comunicación para evitar la replicación de datos.

Ejemplos:

- Filtrar una lista de enteros
- Modelo climático

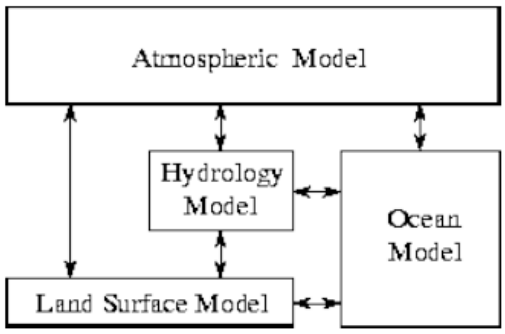


Figura 19. Descomposicion Funcional Del Modelo Del Clima (Foster, 1995)

En la Figura 18. Cada componente del modelo Puede ser pensado como una tarea separada, para ser paralelizado por la descomposición del dominio. Las flechas representan intercambios de datos entre componentes durante el cálculo: el modelo de atmósfera genera

datos de velocidad del viento que son utilizados por el modelo oceánico, el modelo oceánico genera datos de temperatura de superficie del mar que son utilizados por el modelo de atmósfera, etc.(Foster, 1995)

- **Comunicación**:

  El cálculo que se realizará en una tarea normalmente requerirá datos asociados con otra tarea. Los datos deben entonces ser transferidos entre tareas de modo que permitan que la computación continúe.

  La comunicación determina la coordinación para la ejecución de la tarea, define las estructuras y algoritmos de comunicación apropiados.

  Definir la estructura comunicacional de la aplicación es clave en el futuro desempeño de una aplicación. El diseño de la estructura comunicacional se puede descomponer en dos fases: en la primera se definen los enlaces entre las tareas. Después, se especifican los mensajes a intercambiarse entre ellos.(Foster, 1995)

  Los patrones de comunicación son:

  - Local: Es cuando una operación requiere datos de un pequeño subconjunto de tareas.

    Ejemplo: Método de diferencia finita de Jacobi. *Ver figura 17*

- Global: Hace referencia a que deben participar muchas tareas, es decir cada tarea requiere comunicarse con varias tareas Así, en una comunicación global, muchas tareas pueden participar, lo que puede conllevar a muchas comunicaciones.

Descomposición Recursiva:

- Se aplica la técnica divide y vencerás para descubrir concurrencia.
- Generación recursiva de subproblemas
- Se continúa dividiendo
- Cada tarea divide el problema o combina los resultados de los subproblemas
- Esquema de planificación basado en la granja de tareas: estructura de datos compartida (se implementa dependiendo del modelo de programación)

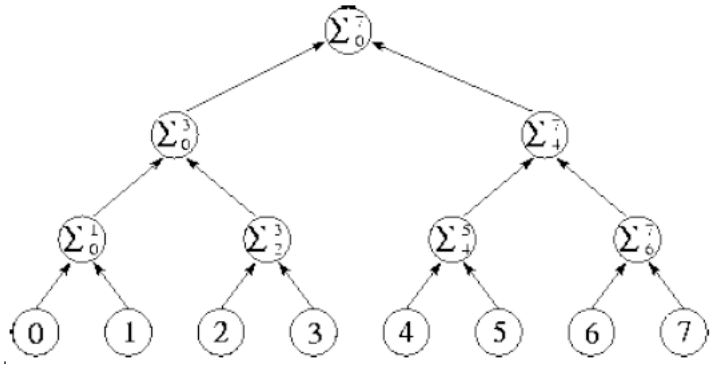


Figura 20. Estrucutura Del Arbol Divide Y Venceras

En la Figura 19 la estructura de árbol para el algoritmo de suma de división y conquista con N = 8. Entonces Los números situados en las tareas en la parte inferior del diagrama se comunican a las tareas en la fila inmediatamente superior; Cada uno realiza una adición y luego reenvía el resultado al siguiente nivel. La suma completa está disponible en la raíz del árbol después de los log^N pasos.

- Comunicación Estructurada: El patrón de comunicación de una tarea con sus vecinos forma una estructura regular.

  Ejemplo, del tipo árbol o malla.

- Comunicación No Estructurada: Las redes de comunicación puede generar grafos arbitrarios.

  Ejemplo: malla de elementos finitos para un montaje

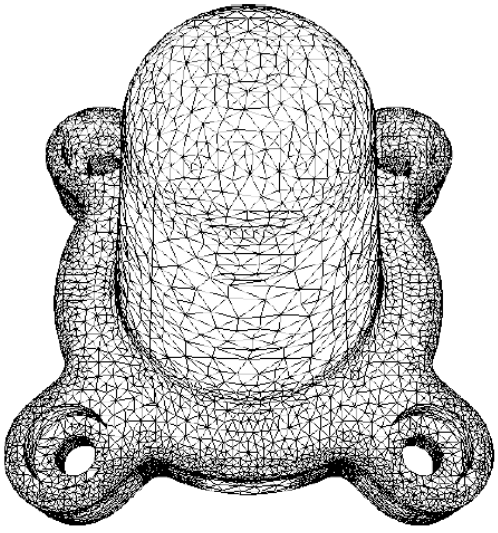


Figura 21. Elemento Finito En Una Malla Para Una Parte Del Montaje(Foster, 1995)

En la Figura 20 el elemento finito en la malla generada para una parte del montaje, cada vértice es un punto de rejilla. Un borde que conecta dos vértices representa una dependencia de datos que requerirá comunicación si los vértices se encuentran en tareas diferentes. Observe que los diferentes vértices tienen un número variable de vecinos. (Imagen cortesía de M. S. Shephard.)(Foster, 1995)

- Comunicación Dinámica: Los algoritmos se aplican con frecuencia durante el tiempo de ejecución del programa siendo estos variables, el costo de estos debe ser medidos contra sus beneficios.

- Comunicación Estática: Los interlocutores de comunicación no cambian con el tiempo
- Comunicación Asincrónica: Un consumidor obtiene datos sin la ayuda del productor.
- Comunicación Sincrónica: Coordinación entre el consumidor y el productor.

- **Aglomeración**:

  Después de la fase de partición y comunicación, el algoritmo resultante es aun abstracto debido a que no está especializado para su eficiente ejecución en el ordenador paralelo. Es ineficiente en el caso que cree muchas tareas que procesadores en el equipo destino y este no este diseñado para la ejecución eficaz de las tareas pequeñas.

En la fase de aglomeración se pasa de lo abstracto a lo concreto, se revisan las decisiones tomadas en las dos etapas anteriores con el fin de obtener un algoritmo eficiente. En particular se agrupan las tareas en tareas más grandes esto con el fin de mejorar el rendimiento y los costos de implementación.

Tres objetivos a veces conflictivos guían las decisiones relativas a la aglomeración y la replicación:

- **Aumentar la cantidad de cálculos y reducir los costos comunicacionales**

  Los costos pueden ser reducidos de dos formas: enviando menos datos o menos mensajes. Esto porque el costo comunicacional depende tanto de la cantidad de datos a transferir como del costo inicial por establecer una comunicación. Ambos costos pueden ser reducidos por agrupamiento. La replicación también puede reducir los requerimientos comunicacionales, introduciendo cálculos redundantes se pueden evitar costos comunicacionales.(J.Aguilar, 2006)

  La agrupación es también importante, cuando se analizan los requerimientos comunicacionales y se determina que ciertas tareas no pueden ejecutarse concurrentemente, por sus dependencias, en el caso de un árbol de ejecución sólo las tareas que están en el mismo nivel pueden ejecutarse concurrentemente, así, tareas en diferentes niveles se agrupan sin reducir las posibilidades de ejecución concurrente entre las tareas.(J.Aguilar, 2006)

- **Aumentar la flexibilidad con respecto a la escalabilidad y proceso de decisión de las asignaciones**

  La capacidad de crear un número variable de tareas es fundamental para que un programa sea portátil y escalable. Un buen algoritmo paralelos debe ser diseñado para ser resistente a los cambios en el número de procesadores.

  La flexibilidad debe ser útil al ajustar un código. Las tareas a menudo bloquean la espera de datos remotos, puede ser ventajoso asignar varias tareas a un procesador. Entonces, una tarea bloqueada no tiene como resultado que un procesador se vuelva inactivo, ya que otra tarea puede ser capaz de ejecutar en su lugar. De esta forma, la comunicación de una tarea se superpone con el cálculo de otra tarea. Esta técnica, denominada superposición de cálculo y comunicación.

  Otro beneficio de crear más tareas que los procesadores es que proporciona un mayor margen para las estrategias de mapeo que equilibran la carga computacional sobre los procesadores disponibles

  La flexibilidad no requiere necesariamente que un diseño siempre cree un gran número de tareas. La granularidad se puede ser controlada mediante un parámetro de tiempo de compilación o de tiempo de ejecución. Lo importante es que no incorpore límites innecesarios al número de tareas que se pueden crear.

- **Reducir costos de diseño de software**

Una preocupación adicional, que puede ser particularmente importante al paralelizar códigos secuenciales existentes, es los costos de desarrollo relativos asociados con diferentes estrategias de partición. Desde esta perspectiva, las estrategias más interesantes pueden ser aquellas que evitan cambios extensos de código.

- **Mapeo:**

Cada tarea se le asigna a un procesador con el fin de maximizar la utilización del procesador (mediante algoritmos de balanceo de carga) y minimizar los costos de comunicación y sincronización.

El mapeo es el proceso de asignar tareas a los procesadores. Si se ejecuta un programa en un multiprocesador centralizado, el sistema operativo automáticamente mapea procesos a procesadores

Los objetivos del mapeo son maximizar la utilización del procesador y minimizar
Comunicación inter procesador. La utilización del procesador es el porcentaje promedio Los procesadores del sistema están ejecutando activamente las tareas necesarias para Solución del problema. La utilización del procesador se maximiza cuando el cálculo Se equilibra uniformemente, permitiendo que todos los procesadores comiencen y terminen la ejecución al mismo tiempo.

La comunicación interprocesador aumenta cuando dos tareas conectadas por un canal de mapeado a diferentes procesadores. La comunicación inter procesador disminuye cuando dos tareas conectadas por un canal están asignadas al mismo procesador.

El objetivo en el desarrollo de algoritmos de mapeo es normalmente minimizar el tiempo de ejecución total. Dos estrategias para lograr este objetivo:

1. Tareas capaces de ejecutar simultáneamente en diferentes procesadores, para mejorar la concurrencia.
2. Tareas que se comunican frecuentemente en el mismo procesador, para aumentar localidad.

6.1.2. Metodologia De Barbara Chapman

- **Optimizar el uso de barrier:** Sin importar lo eficaz que sean los barrier, son operaciones costosas, es por ello que se hace necesario reducir su uso. La clausula nowait elimina la barrera implícita en varias contrucciones. El #pragma omp for loop tiene una barrier implícito. Primero se debe asegurar que el programa OpenMP funcione correctamente y luego utilice la cláusula nowait cuando sea posible, insertandose cuidadosamente en puntos específicos del necesario. Al hacer esto, se

debe tener cuidado en identificar y ordenar cálculos que escriben y leen la misma porción de memoria.(Barbara Chapman, Gabriele Jost, n.d.)

```
#pragma omp parallel
{
.........
#pragma omp for
for (i=0; i<n; i++)
.........
#pragma omp for nowait
for (i=0; i<n; i++)
      } /*-- End of parallel region - barrier is implied --*/
```

Figura 9. Clausula Nowait omite el barrier implícito al final del bucle (Barbara Chapman, Gabriele Jost, n.d.)

- **Evitar el constructor ordered:** Es costoso de implementar debido a que el sistema runtime debe estar pendiente que las iteraciones hayan terminado o posiblemente mantener los threads en un estado de espera hasta terminar los demás. Esto inevitamente relentiza la ejecucion del programa. (Barbara Chapman, Gabriele Jost, n.d.).

- **Evitar grandes regiones critical:** Una región critical se usa para garantizar que nos hay dos subprocesos trabajando en el mismo fragmento de código simultáneamente. Entre mas código contenga la región critical hay mas probabilidad que los threads deban eperar para entrar en ella, haciendo los tiempos de espera potenciales. (Barbara Chapman, Gabriele Jost, n.d.)

- **Maximar las regiones paralelas:** El uso indiscriminado de regiones paralelos puede dar un rendimiento suboptimo, Las grandes regiones paralelas ofrecen más oportunidades para utilizar datos en caché y proporcionan un contexto más grande para otras optimizaciones de compilador. Por lo tanto, es necesario minimizar el

número de regiones paralelas. En el caso de que se tengas multiples bucles paralelos, se debe elegir si encapsular cada bucle de la región individual o crear una región paralela que abarque todos.(Barbara Chapman, Gabriele Jost, n.d.)

- **Evitar regiones paralelas en bucles internos:** Otra técnica para mejorar el rendimiento es desplazar las regiones paralelas fuera de los bucles internos. (Barbara Chapman, Gabriele Jost, n.d.)

- **Dirección de equilibrio de carga deficiente:** Cuando la carga de trabajo es diferente para cada hilo, resulta un desequilibrio de carga. Cuando esto ocurre, los hilos esperan en el siguiente punto de sincronización hasta que se complete la más lenta. Una forma de superar este problema es usar la cláusula Schedule.(Barbara Chapman, Gabriele Jost, n.d.)

6.1.3. Metodologia De Charles Severance

- Región paralela: Los hilos simplemente aparecen entre dos sentencias de código de línea recta. La ejecución comienza con un solo hilo. (Severance, 2010).

Cuando el programa encuentra la directiva PARALLEL, los otros hilos se activan para unirse al cálculo. Así que, en cierto sentido, como la ejecución pasa la primera directiva, un hilo se convierte en cuatro. Cuatro subprocesos ejecutar las dos instrucciones entre las directivas. A medida que los subprocesos se están ejecutando independientemente, el orden en el que se muestran las sentencias de impresión es

algo aleatorio. Los hilos esperan en la directiva END PARALLEL hasta que todos los hilos hayan llegado. Una vez que todos los hilos han completado el paralelo región, un subproceso único continúa ejecutando el resto del programa.(Severance, 2010)

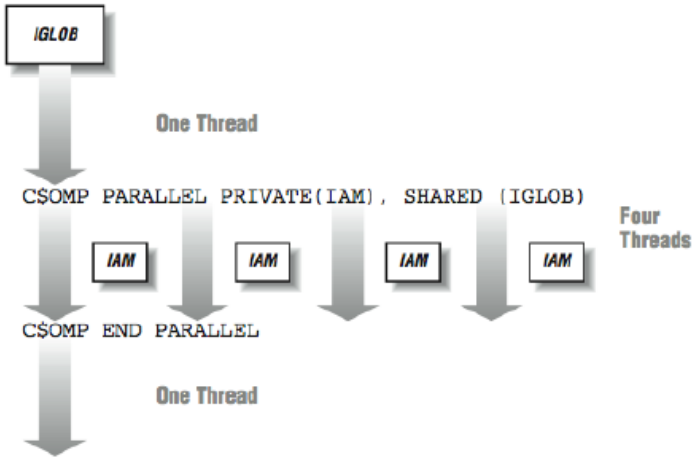


Figura 22. Interaccion De Datos En Una Regio Paralela (Severance, 2010)

En la Figura 21. El PRIVATE (IAM) indica que el IAM Variable no se comparte en todos los subprocesos, pero en su lugar, cada subproceso tiene su propia versión privada de la variable. La variable IGLOB se comparte en todos los subprocesos. Cualquier modificación de IGLOB aparece en otros hilos instantáneamente, dentro de las limitaciones de la coherencia del caché (Severance, 2010)

- Paralelo loops: Las áreas mas valiosas para ejecutar en paralelo son los bucles, al momento de tener el entorno de datos configurado para el bucle, el único problema restante que debe ser resuelto es qué hilos realizará qué iteraciones.(Severance, 2010)

- Iteracion scheduling: Hay dos técnicas básicas para dividir las iteraciones en un bucle entre threads:

  Estática: Al comienzo de un bucle paralelo, cada subproceso toma una porción continua de iteraciones del bucle en base al número de hilos que ejecutan el bucle.

  Dinámica: Cada hilo procesa un fragmento de datos y cuando se ha completado el nuevo pedazo se procesa. El tamaño del trozo puede ser variado por el programador, pero se ve durante la duración del bucle. (Severance, 2010)

6.2. Caracterizacion De Problemas

El conjunto de problemas que pueden ser abordados mediante la computación en paralelo es bastante amplio, y abarca diferentes temáticas dentro de las ciencias computacionales. Uno de los estudios más completos e interesantes que se han abordado al respecto es el realizado por Krste Asanovic y su equipo del departamento de Ciencias de la Computación e Ingeniería Eléctrica en la Universidad de California en Berkeley (Asanovic, Catanzaro, Patterson, & Yelick, 2006), el cual trata de establecer un panorama acerca de la investigación en computación paralela. En la clasificación realizada por Asanovic y su equipo, se agrupan problemas mediante un alto nivel de abstracción que pueden a su vez agrupar diferentes métodos computacionales. La clasificación reúne problemas que presentan patrones comunes de comunicación y computación en cada clase, de forma independiente de implementaciones individuales. El informe sugiere la siguiente agrupación de problemas:

- Problemas de algebra lineal densa

- Problemas de algebra lineal dispersa
- Métodos espectrales
- Métodos de n-cuerpos
- Redes estructuradas
- Redes no estructuradas
- MapReduce
- Lógica Combinacional
- Recorrido de grafos
- Programación dinámica
- Métodos de vuelta atrás, ramificación y poda
- Modelado de grafos
- Máquinas de estado finito

La anterior agrupación reúne conjuntos de problemas con patrones comunes en sus requerimientos, y puede ser considerada una caracterización de problemas suceptibles de ser paralelizados. Sin embago es una categorización aun muy genérica aplicado a diferentes modelos de computación paralela. Varios de estos problemas pueden ser abordados y resueltos eficientemente utilizando el modelo de programación paralela de memoria compartida de OpenMP. Sin embargo, cada tipo de problema debe abordarse de forma indivdual, de tal forma que se saque el mayor provecho al modelo en cuestión.

A continuación, se describe un procedimiento que puede ser utilizado, en caso de querer desarrollar un algoritmo paralelo, a partir de una versión secuencial del mismo,

6.2.1. *Procedimiento para paralelizar el código secuencial.*

Inicialmente, debe hacerse un análisis del problema a tratar y verificar si este es paralelizable, o no, para la cual se puede hacer un análisis de dependencias utilizando la metodología sugerida por Foster: que incluye los pasos sugeridos en la sección 6.1.1: particionado, comunicación, aglomeración y mapeo.

Verificado que se puede obtener una versión del algoritmo que se ejecute en paralelo, Se debe revisar tres tipos de escenarios dentro del algoritmo. El primero de ellos es la identificación de los ciclos *o loops,* suceptibles de ser paralelizados. En el caso de ciclos anidados, se debe analizar si paralelizar el bucle externo o alguno de los internos. Habitualmente da mejores resultados paralelizar el bucle externo, pero ya que no siempre se obtienen los mismos resultados, lo mejor que se puede hacer es comparar los rendimientos de ambos. En segundo lugar, pueden establecerse regiones a ser paralelizadas de forma independiente mediante la directiva *sections*. El tercer escenario corresponde al uso de directiva *task* en el que se define una tarea explicita, que puede ser ejecutada por el hilo o diferida para su ejecucion por cualquier otro hilo del equipo.

Una vez establecida la posibilidad de parelalización, los pasos a seguir son:

1. Identificar bucles, que son susceptibles de ser paralelizados. Para cada uno de ellos se puede tener en cuenta lo siguiente:
    a. En los ciclos *for*, la paralelización de grano grueso suele obtener mejores resultados que la del grano fino, es decir, utilizar la directiva *parallel for* en el ciclo más externo. Sin embargo, es buen ejercicio comparar los resultados, paralelizando ambos ciclos en forma independiente y midiendo tiempos.

b. Definir si la asignación de datos a procesar por un hilo es dinámica, estática o guiada, mediante la clausula *Schedule.*
c. Verificar cuales variables son privadas y cuales son compartidas.
d. Utilizar *reduction,* para juntar cálculos parciales, si es necesario

2. Identificar posibles secciones a paralelizar mediante la directiva *sections*. En este caso cada sección será ejecutada en paralelo por un hilo. Al final, si es necesario se puede sincronizar para que el programa continúe una vez todas las secciones en paralelo sean ejecutadas. Esta directiva es útil cuando tenemos varias secciones independientes que se ejectuan en forma serial, y que pueden ser ejecutadas en forma paralela, para reducir el tiempo de procesamiento.
3. En el caso de bloques de código con ciclos de tipo while, o llamadas recursivas, se recomienda utilizar la directiva *task*, la cual genera unidades de trabajo independientes que son asignadas a un hilo para realizar una tarea, estas pueden ejecutarse inmediatamente o ser diferidas. Aunque *task* paraleliza código recursivo, no necesariamente el resultado es un algoritmo más eficiente en tiempo de procesamiento. Es una clausula en el que se debe ser cuidadoso y realizar juiciosamente análisis de tiempo de procesamiento y el *speed up* alcanzado

6.3. Ejemplos de cálculo numérico

A continuación, se muestran algunos ejemplos de problemas de procesamiento numérico, que caen en la categoría de problemas en redes o grids estructurados, según la caracterización propuesta por (Asanovic et al., 2006). Estos son problemas que solucionan

encuaciones diferenciales parciales mediante el uso de esquemas de diferencias finitas. Se han seleccionado estos ejemplos, por el interés que existe en el grupo de investigación CICOM, de colaborar con proyectos relacionados con la solución de este tipo de problema

Para la realización de las pruebas se ejecuto en cada ejemplo un total de 30 veces por thread debido al uso de la “ley de los grandes números” en la que señala que al realizar n lanzamientos estos van a estar cerca al valor promedio, siendo 30 un número suficientemente grande para la precisión que se necesita.

Para la ejecución de los algoritmos se realizó el siguiente procesamiento:

Mediante la conexión remota a través de ssh se logró acceder al servidor ubicado en los laboratorios de alto rendimiento la universidad de pamplona, en él se realizaron las pruebas para comprobar el rendimiento de los diferentes tipos de ejercicios.

Las pruebas se realizaron en un servidor con las siguientes caracteristicas.
Servidor HP ProLiant DL180, con dos procesadores de la serie Intel Xeon E5 2600 v3, con 6 nucleos cada uno (Para un total de 12 core efectivos). Memoria 16 DDR4. Almacenamiento 1Terabyte.

La conexión al servidor se hizo de la siguiente forma:

Ssh pasarela@softwarelibre.unipamplona.edu.co

Luego se procede a ingresar al siguiente servidor

Ssh cluster@192.168.19.131

Una vez accedido se crea un directorio (carpeta) principal y dentro de este se crean tres subdirectorios con los nombres BIN, TOOLS y SOLUCIONES, el primero tendrá todos los ejecutables de los diferentes ejemplos, el segundo un archivo llamado lanzador.pl este es un archivo PERL que permite ejecutar varios ejemplos, con los threads y las iteraciones que se desee sin la necesidad a estar ejecutándolo manualmente y el tercero contendrá las carpetas generadas por el archivo PERL.

Para agregar los ejecutables a la carpeta BIN se hizo con el comando scp, este permite copiar archivos o directorios entre diferentes hosts que utilizan autenticación ssh.

Antes de enviar cada ejecutable al servidor se comprueba que el algoritmo funcione correctamente, luego se comenta las partes de código que imprime los resultados dejando solamente el tiempo de ejecución, una vez hecho lo anterior se genera el ejecutable y se envía a la carpeta BIN ubicada en el servidor clúster, esto se hace debido a que el principal interés es analizar el comportamiento de la versión secuencial frente a la paralela. Es por ello que cada uno de los archivos generados por el lanzador contiene el tiempo de ejecución.

El archivo PERL permite detallar los threads a ejecutar, las iteraciones a realizar y el nombre de cada uno de los ejecutables a los que se le desean analizar el rendimiento. Este hace el llamado a la carpeta BIN para verificar que los nombres ingresados sean iguales a los que están en la misma. Además de ello accede al directorio soluciones para guardar las diferentes carpetas generadas con sus respectivos archivos que contienen los resultados de las pruebas realizadas. El archivo se ejecuta de la forma ./pl junto con el nombre de la

carpeta en la que se desea guardar cada uno de los archivos. Al terminar la ejecución se genera un archivo por cada thread y con el respectivo nombre del ejecutable.

La capacidad del procesador del servidor permitió realizar para cada ejemplo prueba 30 iteraciones por thread, teniendo una disponibilidad de 12 threads.

Una vez se tengan los tiempos de ejecución se procede a ubicarlos en tablas, de ellas se obtienen los promedios de la versión secuencial y la paralela, así como el speed up del algoritmo paralelizado junto con estos sus respectivas gráficas, esto con el fin de analizar el comportamiento de las dos versiones del algoritmo.

El **SpeedUp** nos permite saber cuánto se acelera el sistema al incluir nuevos nodos o unidades de procesamiento, para efectos de la programación paralela es necesario tener en cuenta algunas precisiones. El tiempo de ejecución sin mejora hace referencia al tiempo que se toma el sistema en ejecutar algún programa secuencialmente, y el tiempo con mejora es básicamente el tiempo demorado en ejecutar el mismo problema de manera paralela, así tendríamos lo siguiente:

$$S(n) = \frac{Tiempo(1)}{Tiempo(n)}$$

Donde n es el número de nodos usados en la ejecución del programa en paralelo.

6.3.1. Ecuacion Del Calor En 2D

La ecuación del calor en estado estacionario que describe la distribución del calor en una placa rectangular a través del tiempo, se define por la siguiente expresión:

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0$$

Para encontrar la solución de dicha ecuación mediante un esquema de diferencias finitas, la región física, y las condiciones de contorno, son sugeridas por este diagrama;

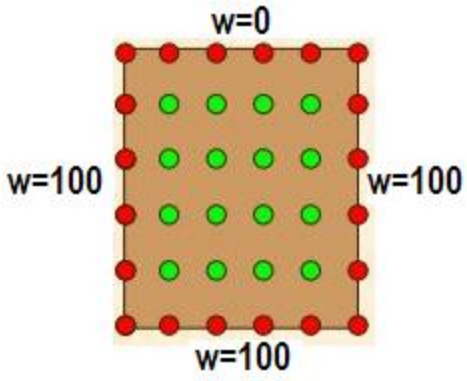


Figura 23 Región física, Condiciones de contorno

La región está cubierta con una rejilla de M por N nodos, y un N por N matriz W se utiliza para registrar la temperatura. La correspondencia entre los índices de matriz y las ubicaciones en la región se sugiere al dar los índices de las cuatro esquinas:

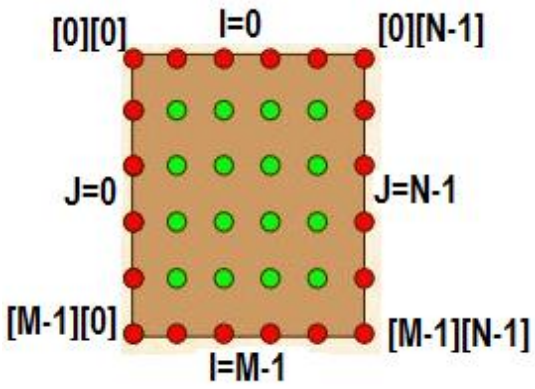


Figura 24Diagrama De Condiciones De Contorno

La solución en estado estacionario a la ecuación de calor discreta satisface la siguiente condición en un punto de rejilla interior:

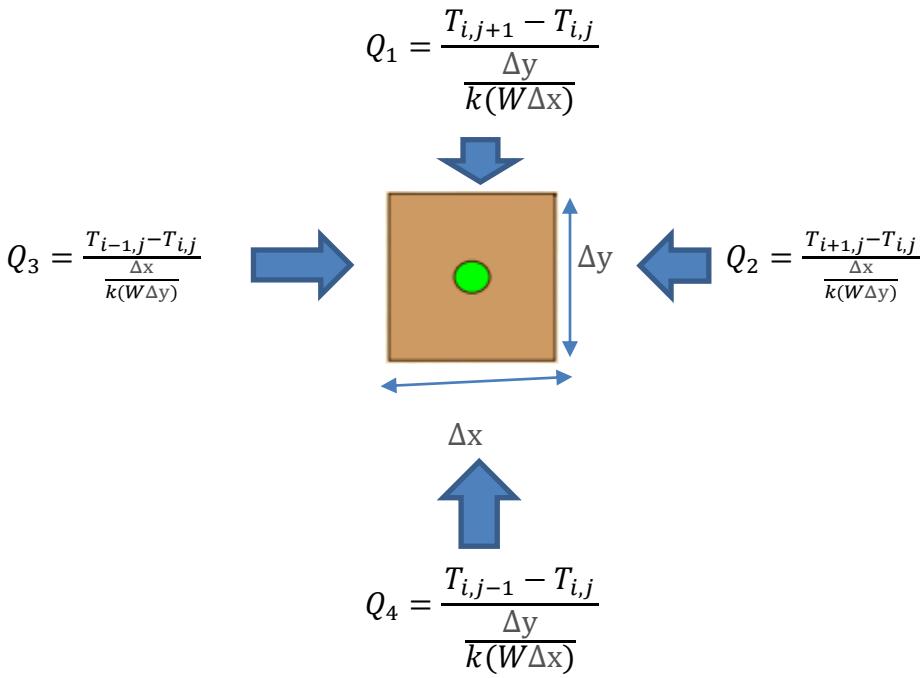


Figura 25 Punto Interior De Rejilla. Ecuacion De Calor En 2D

$$Q_1 + Q_2 + Q_3 + Q_4 = 0$$

$$\frac{\frac{T_{i,j+1}-T_{i,j}}{\Delta y}}{k(W\Delta x)} + \frac{\frac{T_{i+1,j}-T_{i,j}}{\Delta x}}{k(W\Delta y)} + \frac{\frac{T_{i-1,j}-T_{i,j}}{\Delta x}}{k(W\Delta y)} + \frac{\frac{T_{i,j-1}-T_{i,j}}{\Delta y}}{k(W\Delta x)} + \varphi\,\Delta x\Delta y W = 0$$

$$T_{i,j+1} - T_{i,j} + T_{i+1,j} - T_{i,j} + T_{i-1,j} - T_{i,j} + T_{i,j-1} - T_{i,j} + \frac{\varphi\ \Delta x^2}{k} = 0$$

$$T_{i,j+1} + T_{i+1,j} + T_{i-1,j} + T_{i,j-1} + \frac{\varphi\ \Delta x^2}{k} - 4T_{i,j} = 0$$

$$T_{i,j} = \frac{\frac{\varphi\ \Delta x^2}{k} T_{i,j+1} + T_{i+1,j} + T_{i-1,j} + T_{i,j-1}}{\mathbf{4}}$$

Siendo $T_{i,j} = w[Central]$

$T_{i,j+1} = w[Norte]$,

$T_{i+1,j} = w[Este]$,

$T_{i-1,j} = w[Oeste]$,

$T_{i,j-1} = w[Sur]$

Quedando de la siguiente forma:

$$w[Central] = \left(\frac{1}{4}\right) * (w[Norte] + w[Sur] + w[Este] + w[Oeste])$$

Donde "Central" es el índice del punto de la cuadrícula, "Norte" es el índice de su vecino inmediato al "norte", y así sucesivamente.

Dada una solución aproximada de la ecuación de calor en estado estacionario, se da una solución "mejor" reemplazando cada punto interior por la media de sus 4 vecinos - en otras palabras, usando la condición como una declaración de ASIGNACIÓN:

$$w[Central] <= \left(\frac{1}{4}\right) * (w[Norte] + w[Sur] + w[Este] + w[Oeste])$$

Si este proceso se repite con suficiente frecuencia, la diferencia entre las estimaciones sucesivas de la solución pasará a cero.

Este programa realiza una iteración de este tipo, utilizando una tolerancia especificada por el usuario, y escribe la estimación final de la solución a un archivo que se puede utilizar para el procesamiento gráfico. (Department of Scientific Computing, 2011)

Es un programa en C que ilustra el uso de la API OpenMP empleando una iteración que resuelve la ecuación de calor en estado estacionario 2D.

Este programa muestra cómo el máximo se puede computar usando una aproximación que combina variables privadas y una sección crítica.

Version paralelizada del algoritmo

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <omp.h>

int main ( int argc, char *argv[] );
/******************************************************************/
int main ( int argc, char *argv[] )
/******************************************************************/
/*
  Purpose:
    MAIN is the main program for ECUACION_CALOR_OPENMP.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    18 October 2011

  Author:
    Original C version by Michael Quinn.
    This C version by John Burkardt.

  Reference:
```

```
    Michael Quinn,
    Parallel Programming in C with MPI and OpenMP,
    McGraw-Hill, 2004,
    ISBN13: 978-0071232654,
    LC: QA76.73.C15.Q55.

  Local parameters:
    Local, double DIFF, the norm of the change in the solution from one
iteration
    to the next.
    Local, double MEAN, the average of the boundary values, used to
initialize
    the values of the solution in the interior.
    Local, double U[M][N], the solution at the previous iteration.
    Local, double W[M][N], the solution computed at the latest iteration.
*/
{
# define M 500
# define N 500

  double diff;
  double epsilon = 0.001;
  int i;
  int iterations;
  int iterations_print;
  int j;
  double mean;
  double my_diff;
  double u[M][N];
  double w[M][N];
  double wtime;
  int nthreads;
  int S = atoi(argv[1]);
  omp_set_num_threads(S);
  nthreads = omp_get_num_threads();

  printf ("\n" );
  printf ("HEATED_PLATE_OPENMP\n" );
  printf ("C/OpenMP version\n" );
  printf ("A program to solve for the steady state temperature
distribution\n" );
  printf ("over a rectangular plate.\n" );
  printf ("\n" );
  printf ("Spatial grid of %d by %d points.\n", M, N );
  printf ("The iteration will be repeated until the change is <= %e\n",
epsilon );
  printf ( "  Number of threads =%d\n", omp_get_max_threads ( ) );
  printf("Numero de threads en ejecucion = %d\n", nthreads);
/*
  Set the boundary values, which don't change.
*/
  mean = 0.0;

#pragma omp parallel shared ( w ) private ( i, j )
  {
#pragma omp for
    for ( i = 1; i < M - 1; i++ )
    {
      w[i][0] = 100.0;
```

```
    }
#pragma omp for
    for ( i = 1; i < M - 1; i++ )
    {
      w[i][N-1] = 100.0;
    }
#pragma omp for
    for ( j = 0; j < N; j++ )
    {
      w[M-1][j] = 100.0;
    }
#pragma omp for
    for ( j = 0; j < N; j++ )
    {
      w[0][j] = 0.0;
    }
/*
  Promedio de los valores límite, para llegar a un valor inicial razonable
   */
#pragma omp for reduction ( + : mean )
    for ( i = 1; i < M - 1; i++ )
    {
      mean = mean + w[i][0] + w[i][N-1];
    }
#pragma omp for reduction ( + : mean )
    for ( j = 0; j < N; j++ )
    {
      mean = mean + w[M-1][j] + w[0][j];
    }
  }
/*
  Nota de OpenMP:
   No se puede normalizar MEAN dentro de la región paralela. Este
   Sólo obtiene su valor correcto una vez que salga de la región
paralela.  Así que interrumpimos la región paralela, ponemos MEAN, y
volvemos.
*/
  mean = mean / ( double ) ( 2 * M + 2 * N - 4 );
  printf ( "\n" );
  printf ( "  MEAN = %f\n", mean );
/*
  Initialize the interior solution to the mean value.
*/
#pragma omp parallel shared ( mean, w ) private ( i, j )
  {
#pragma omp for
    for ( i = 1; i < M - 1; i++ )
    {
      for ( j = 1; j < N - 1; j++ )
      {
        w[i][j] = mean;
      }
    }
  }
/*
Iterar hasta que la nueva solución W difiera de la vieja solución U
   Por no más de EPSILON.
*/
  iterations = 0;
```

```
  iterations_print = 1;
  printf ( "\n" );
  printf ( " Iteration  Change\n" );
  printf ( "\n" );
  wtime = omp_get_wtime ( );
  diff = epsilon;

  while ( epsilon <= diff )
  {
# pragma omp parallel shared ( u, w ) private ( i, j )
    {
/*
  Save the old solution in U.
*/
# pragma omp for
      for ( i = 0; i < M; i++ )
      {
        for ( j = 0; j < N; j++ )
        {
          u[i][j] = w[i][j];
        }
      }
/*
Determine la nueva estimación de la solución en los puntos interiores.
La nueva solución W es la media de los vecinos norte, sur, este y oeste.
*/
# pragma omp for
      for ( i = 1; i < M - 1; i++ )
      {
        for ( j = 1; j < N - 1; j++ )
        {
          w[i][j] = ( u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] ) / 4.0;
        }
      }
    }
/*
C y C ++ no pueden calcular un máximo como una operación de reducción.
   Por lo tanto, definimos una variable privada MY_DIFF para cada
subproceso. Una vez que han calculado sus valores, usamos una sección CRÍTICA
Para actualizar DIFF.
*/
    diff = 0.0;
# pragma omp parallel shared ( diff, u, w ) private ( i, j, my_diff )
    {
      my_diff = 0.0;
# pragma omp for
      for ( i = 1; i < M - 1; i++ )
      {
        for ( j = 1; j < N - 1; j++ )
        {
          if ( my_diff < fabs ( w[i][j] - u[i][j] ) )
          {
            my_diff = fabs ( w[i][j] - u[i][j] );
          }
        }
      }
# pragma omp critical
      {
        if ( diff < my_diff )
```

```
        {
          diff = my_diff;
        }
      }
    }

    iterations++;
    if ( iterations == iterations_print )
    {
      printf ( "  %8d  %f\n", iterations, diff );
      iterations_print = 2 * iterations_print;
    }
  }
  wtime = omp_get_wtime ( ) - wtime;

  printf ( "\n" );
  printf ( "  %8d  %f\n", iterations, diff );
  printf ( "\n" );
  printf ( "  Error tolerance achieved.\n" );
  printf ( "  Wallclock time = %f\n", wtime );
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "HEATED_PLATE_OPENMP:\n" );
  printf ( "  Normal end of execution.\n" );

  return 0;

# undef M
# undef N
}
```

6.3.2. Ecuacion de Onda Acustica 2D

Formulación de la ecuación:

La ecuación de onda para la función $U(x,y,t)$ está dada por la expresión (Acevedo Alvaro Casasús, Juan Jose Benito Muñoz Prieto & Corvinos, 2009)

$$\frac{\partial^2 U(x,y,t)}{\partial t^2} = c^2\left(\frac{\partial^2 U(x,y,t)}{\partial x^2} + \frac{\partial^2 U(x,y,t)}{\partial y^2}\right) \quad t > 0 \quad (x,y) \in \Omega \subseteq \mathbb{R}^2$$

Con las condiciones iniciales:

$$U(x,y,0) = f_1(x,y);\ \frac{\partial U(x,y,0)}{\partial t} = f_2(x,y)$$

y la condición de contorno

$$aU(x_0,y_0,t) + bU(x_0,y_0,t) = g(t)\ en\ \Gamma$$

siendo $f_1(x,y)$, $f_2(x,y)\ y\ g(t)$ funciones conocidas c^2 una constante que representa la velocidad de propagación de la onda y Γ la frontera del domino Ω.

Solución de la ecuación de onda 2D aplicando el esquema de diferencias finitas:

Mediante el esquema de diferencias finitas centradas tenemos que para calcular el valor de U_{ij}^{n+1} se realiza mediante la siguiente expresión:

$$U_{ij}^{n+1} = 2(1-G^2)U_{i,j}^{n} + G^2\left(U_{i+1,j}^{n} + U_{i-1,j}^{n} + U_{i,j+1}^{n} + U_{i,j-1}^{n}\right) - U_{i,j}^{n-1}$$

A continuación, se muestra el algoritmo que calcula la solución de la ecuación de onda, en sobre un grid de NxM para un total de T iteraciones

En este caso partimos de un algoritmo inicial, que soluciona y modela en forma secuencial la ecuación de onda. Después de establecer las condiciones iniciales, el algoritmo en cada iteración calcula la malla solución actual, e imprime los resultados en un archivo de disco.

A continuación se muestra la implementación base.

1. Algoritmo de acceso a disco rigidodo

```
#include<stdio.h>
#include<stdlib.h>
#include <sys/time.h>
#include<math.h>

using namespace std;

int tout=1000;
int Nx=201, Nz=201;
float alpha = 1.0, beta=1.0;
float c = 10;
float dt=0.01, G=0.1;
float U1[201][201], U2[201][101], U3[201][201] ;
float dx, dz, x, z, t;
float x0=50.0, z0=50.0, r;
char namefile[15];

FILE *out;

int main()
{
struct timeval t0, t1;
double tej;
gettimeofday(&t0, 0);   // Tomamos tiempo de inicio
      //out = fopen("2Ddata.txt","w");
dx = c*dt/G;
dz = dx;

for (int i=0; i<Nx; i++)
{
  for (int j=0; j<Nz; j++)
  {
  x = i*dx-x0;
  z = j*dz-z0;
  r = sqrt(x*x+z*z);
  U2[i][j] = sin(alpha*r)*exp(-beta*r*r);
  U1[i][j] = 0.0;
  U3[i][j] = 0.0;
/* U1[i] = U2[i];*/
  }
```

```
        }
        dx = c*dt/G;
        dz = dx;

        for(int n=0; n<tout; n++)
        {
          sprintf(namefile,"../plot2d/2Ddata%04d.txt",n);
          out = fopen(namefile,"w");

        for (int i=1; i<Nx-1; i++)
        {
          for (int j=1; j<Nz-1; j++)
          {
            U3[i][j] = 2*(1-2*G*G) * U2[i][j] + G*G*(U2[i+1][j] + U2[i-1][j]+
        U2[i][j+1]+ U2[i][j-1]) - U1[i][j];
          }
        }

        for (int i=1; i<Nx-1; i++)
        {
          for (int j=1; j<Nz-1; j++)
          {
            fprintf(out,"%f ",U3[i][j]);
          }
            fprintf(out,"\n");
        }

        for (int i=1; i<Nx-1; i++)
        {
          for (int j=1; j<Nz-1; j++)
          {
            U1[i][j] = U2[i][j];
            U2[i][j] = U3[i][j];
          }
        }
        fclose (out);
        }
        gettimeofday(&t1, 0);  // Tomamos tiempo final
        tej = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec) / 1e6;
        printf("\n Tiempo de ejecucion: %1.3f ms\n\n", tej*1000);
        return (0);
    }
```

2. Tomando como base el algoritmo secuencial anterior, se reorganiza el código de forma que todos los cálculos se realicen en memoria para las *tout* iteraciones y luego se accede a disco para guardar la información, contenida en la matriz tridimensional U3.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <sys/time.h>
#include <omp.h>

#define dimx 300
#define dimz 300

using namespace std;

int tout=1000;
int Nx=201, Nz=201;
float alpha = 1.0, beta=1.0;
float c = 10;
float dt=0.01, G=0.1;
float U1[201][201], U2[201][201], U3[201][201][1000] ;
float dx, dz, x, z, t;
float x0=50.0, z0=50.0, r ;
char namefile[15];

FILE *out;

int main()
{
struct timeval t0, t1;
double tej;
int i,j,n;

gettimeofday(&t0, 0);   // Tomamos tiempo de inicio
//out = fopen("2Ddata.txt","w");
dx = c*dt/G;
dz = dx;

{
  for (i=0; i<Nx; i++)
   {
     for (j=0; j<Nz; j++)
      {
        x = i*dx-x0;
        z = j*dz-z0;
      r = sqrt(x*x+z*z);
      U2[i][j] = sin(alpha*r)*exp(-beta*r*r);
      U1[i][j] = 0.0;
      //U3[i][j] = 0.0;
      /* U1[i] = U2[i];*/
      }
    }
}
for(n=0; n<tout; n++)
 {
 {
     for ( i=1; i<Nx-1; i++)
      {
        for ( j=1; j<Nz-1; j++)
          {
```

```
            U3[i][j][n] = 2*(1-2*G*G) * U2[i][j] + G*G*(U2[i+1][j] +
U2[i-1][j]+ U2[i][j+1]+ U2[i][j-1]) - U1[i][j];
          }
      }

    for ( i=1; i<Nx-1; i++)
     {
      for ( j=1; j<Nz-1; j++)
       {
         U1[i][j] = U2[i][j];
         U2[i][j] = U3[i][j][n];
        }
     }
   }
}
/************** Acceso a  disco ************************
#pragma omp parallel for private(i,j,namefile,out)
for(n=0; n<tout; n++)
 {
   sprintf(namefile,"../plot2d/2Ddata%04d.txt",n);
   out = fopen(namefile,"w");
     for ( i=1; i<Nx-1; i++)
       {
         for ( j=1; j<Nz-1; j++)
          {
            fprintf(out,"%f ",U3[i][j][n]);
          }
         fprintf(out,"\n");
       }
   fclose (out);
 }
/*****************************************************/


gettimeofday(&t1, 0);  // Tomamos tiempo final
tej = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec) / 1e6;
printf("\n Tiempo de ejecucion: %1.3f ms\n\n", tej*1000);
return (0);
}
```

3. Una vez separado el cálculo de la solución de la ecuación, del código que imprime en disco rígido, se pasa a paralelizar en la solución de la ecuación. Inicialmente, como se recomienda en 6.2.1 se establecen las partes de código que pueden paralelizarse, en este caso, claramente se tiene una serie de ciclos *for* . El primer bloque a paralelizar, realiza la asignación de las condiciones iniciales, corresponde a un ciclo doblemente anidado. Se utiliza un *#pragma omp parallel for* en el bucle más externo. En el segundo bloque candidato a parelelizar, se realiza el cálculo por diferenias finitas.

Correponde a un ciclo triple. En este caso se ha probado experimentalmente que el segundo bucle ofrece mejor rendimiento en términos del speed up y tiempo de procesamiento, por tanto es allí donde se realiza el llamado a la directiva *#pragma omp parallel for.*

Un tercer bloque a paralelizar es el que realiza la reasignación entre matrices, corresponde a un doble ciclo for anidado, y como en la mayoría de los casos, conviene utilizar en el bloque externo el *#pragma omp parallel for*

En la siguiente sección se muestran los resultados en cuanto al tiempo de procesamiento y aceleración del algoritmo implementado. El código final modificado se muestra a continuación.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <sys/time.h>
#include <omp.h>

#define dimx 300
#define dimz 300

using namespace std;

int tout=1000;
int Nx=201, Nz=201;
float alpha = 1.0, beta=1.0;
float c = 10;
float dt=0.01, G=0.1;
float U1[201][201], U2[201][201], U3[201][201][1000] ;
float dx, dz, x, z, t;
float x0=50.0, z0=50.0, r ;
char namefile[15];

FILE *out;

int main()
{
struct timeval t0, t1;
```

```
double tej;
int i,j,n;

gettimeofday(&t0, 0);   // Tomamos tiempo de inicio
//out = fopen("2Ddata.txt","w");
dx = c*dt/G;
dz = dx;
   omp_set_num_threads(4);
#pragma omp parallel
{
  #pragma omp for private(i, j, x, z, r) schedule(static)
  for (i=0; i<Nx; i++)
   {
     for (j=0; j<Nz; j++)
      {
        x = i*dx-x0;
        z = j*dz-z0;
      r = sqrt(x*x+z*z);
      U2[i][j] = sin(alpha*r)*exp(-beta*r*r);
      U1[i][j] = 0.0;
      //U3[i][j] = 0.0;
      /* U1[i] = U2[i];*/
      }
    }
} // pragma
// #pragma omp parallel private(i,j)
for(n=0; n<tout; n++)
 {
   #pragma omp parallel private(i,j)
 {
    #pragma omp  for schedule(static)
     for ( i=1; i<Nx-1; i++)
      {
        for ( j=1; j<Nz-1; j++)
          {
            U3[i][j][n] = 2*(1-2*G*G) * U2[i][j] + G*G*(U2[i+1][j] + U2[i-
1][j]+ U2[i][j+1]+ U2[i][j-1]) - U1[i][j];
          }
      }

    #pragma omp  for schedule(static)
    for ( i=1; i<Nx-1; i++)
     {
      for ( j=1; j<Nz-1; j++)
       {
         U1[i][j] = U2[i][j];
         U2[i][j] = U3[i][j][n];
        }
     }
   }  // pragma
}  // for
/************** Acceso a  disco ************************
#pragma omp parallel for private(i,j,namefile,out)
for(n=0; n<tout; n++)
 {
   sprintf(namefile,"../plot2d/2Ddata%04d.txt",n);
```

```
   out = fopen(namefile,"w");
     for ( i=1; i<Nx-1; i++)
       {
         for ( j=1; j<Nz-1; j++)
          {
            fprintf(out,"%f ",U3[i][j][n]);
          }
         fprintf(out,"\n");
       }
   fclose (out);
 }
/*******************************************************/


gettimeofday(&t1, 0);  // Tomamos tiempo final
tej = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec) / 1e6;
printf("\n Tiempo de ejecucion: %1.3f ms\n\n", tej*1000);
return (0);
}
```

6.3.3. Otros ejemplos numéricos

Otro tipo de ejercicios numéricos fueron explorados para determinar el desempeño del ´codigo en el servidor utilizado. Un buen número de ejemplos, el código y el análisis de tiempos se añade como anexos a este trabajo. En este infome se ha seleccionado a manera ejemplo un algoritmo clásico de referencia: el calculo de Pi mediante sumas de Rieman. Los resultados en tiempo de procesamiento y aceleración o *speed up*, se presentan en la siguiente sección.

Calculo de Pi

El número PI se define como la integral asi:

$$\int_0^4 \frac{4}{1+x^2}$$

La aproximación de una integral mediante la suma de Riemann permite dividir el trabajo en unidades independientes, siendo un factor de precisión el número de divisiones.

Version paralela del algoritmo:

La paralelizacion se encuentran en el for en el que se comparten las variables sum, num_steps y step y contiene las variables privadas i, x.

Para el caso de obtener el algoritmo en la versión secuencial se eliminan todos los pragmas del algoritmo paralelizado.

```
#include <stdio.h>
#include <omp.h>

static long num_steps = 1000000;
#define NUM_THREADS 2
double step;
void openMP()
{
    double before = omp_get_wtime();
    int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel shared(sum, num_steps, step) private(i, x)
    {
        #pragma omp for reduction(+:sum)
        for(i=1; i<=num_steps; i++)
        {
            x = (i-0.5) * step;
            sum = sum + 4.0/(1.0 + x*x);
        }

      #pragma omp single
        pi = step * sum;
    }

    printf("PI : %f\n",pi);
    printf("%f sec OpenMP Complete.\n",omp_get_wtime()-before);
    printf ( "  Number of threads =%d\n", omp_get_max_threads ( ) );
}
int main()
{
    openMP();
    return 0;
}
```

7. Análisis de resultados

7.1. Ecuacion De Calor En 2D

En la tabla 1 se muestra el tiempo de procesamiento obtenido en 30 lanzamientos del programa que calcula la ecuación del calor, utilizando de 1 a 12 cores del servidor. La tabla 2 muestra el tiempo promedio de todos los lanzamientos, y la tabla 3 muestra el tiempo empleado por el algoritmo estrictamente secuencial, y su promedio se puede ver en la tabla 4. Una síntesis de estos resultados puede verse en la figura 25. El tiempo de procesamiento se muestra en el eje Y, mientras que el eje X corresponde al número de *cores* o núcleos empleados en cada ejecución. Se puede apreciar un descenso importante del tiempo de procesamiento a medida que aumenta el número de núcleos empleado.

La figura 26 muestra el *speedup* o aceleración del algoritmo. En el caso ideal, la gráfica de la aceleración correspondería a la identidad, de tal forma que al aumentar el número de procesadores, aumentaría en igual proporción la aceleración del algoritmo. La figura 26 muestra que la aceleración para el algoritmo paralelizado que soluciona la ecuación del calor 2D mediante diferencias finitas, esta muy cerca del ideal, por tanto, con los recursos disponibles de la máquina, el algoritmo resulta en una buena implementación.

Tabla 1 Tiempos (Segundos) Ecuacion De Calor En 2D Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	76.121	38.449	27.121	19.672	17.129	13.441	11.738	10.309	10.040	8.486	7.876	7.356
2	76.226	38.425	28.033	19.690	15.957	13.644	12.724	10.371	10.547	8.609	7.883	7.249
3	76.218	38.515	28.801	19.646	15.921	13.672	12.150	10.393	9.658	8.561	7.862	7.358
4	76.096	38.537	30.640	19.697	15.972	13.387	12.198	10.290	9.367	8.557	8.341	7.332
5	76.028	38.443	28.932	19.751	15.955	13.374	12.462	10.432	9.292	8.943	7.959	7.276
6	76.088	38.423	27.132	19.613	17.136	13.542	12.198	10.410	9.427	8.531	7.795	7.265
7	76.085	38.434	28.990	19.627	18.539	13.444	11.216	10.374	9.428	8.882	7.929	7.306
8	76.237	38.404	28.801	19.714	15.975	13.568	12.959	10.380	10.21	8.444	7.832	7.308
9	76.060	38.430	26.098	19.735	16.699	13.490	12.088	10.450	9.393	8.599	8.369	7.285
10	76.134	38.455	28.038	19.710	15.756	13.803	12.326	10.494	9.510	8.581	7.813	7.448
11	76.073	38.472	27.802	19.830	16.616	13.610	12.217	10.435	9.360	8.564	8.196	7.358
12	76.075	38.420	30.640	19.653	17.177	13.595	11.698	10.475	9.215	8.605	7.804	7.371
13	76.091	38.494	27.589	19.834	15.947	13.326	12.056	10.395	9.221	8.490	7.783	7.369
14	76.081	38.505	27.187	19.667	17.099	13.327	12.121	10.445	9.216	8.549	7.807	7.377
15	76.184	38.401	28.984	19.667	15.964	13.521	12.980	10.386	9.422	8.521	7.981	7.442
16	76.060	38.585	28.873	19.665	15.846	13.218	12.214	10.435	9.290	8.700	8.616	7.321
17	76.181	38.504	28.831	19.709	17.530	13.659	12.549	10.522	9.313	8.478	7.999	7.282
18	76.033	38.453	26.074	19.666	15.743	13.502	12.154	10.388	9.351	8.387	7.994	7.367
19	76.258	38.493	30.632	19.798	15.835	13.980	12.021	10.430	9.214	8.611	7.877	7.298

20	76.063	38.442	27.743	19.741	15.949	13.217	11.215	10.464	9.324	8.771	7.914	7.266
21	76.029	38.578	28.932	19.548	15.939	13.757	12.124	10.439	9.313	8.607	7.839	7.362
22	76.065	38.464	26.097	19.768	17.735	14.035	11.858	10.328	10.41	8.516	9.265	7.348
23	76.119	38.442	28.037	19.721	16.802	13.215	11.994	10.348	9.447	8.493	9.595	7.281
24	76.127	38.488	28.842	19.856	16.716	13.664	12.807	10.391	9.316	8.545	7.881	7.274
25	76.255	38.495	27.563	19.736	17.408	13.512	11.683	10.405	10.28	8.604	7.881	7.364
26	76.065	38.446	27.673	19.727	15.947	13.534	11.214	10.373	10.43	8.480	7.985	7.277
27	76.079	38.515	28.801	19.732	17.234	13.531	12.643	10.405	9.279	8.570	8.144	7.311
28	76.046	38.489	28.841	19.678	16.482	13.326	11.900	10.363	9.221	8.490	7.878	7.344
29	76.234	38.459	27.573	19.766	16.734	13.219	12.246	10.334	10.217	8.602	7.858	7.372
30	76.172	38.532	30.621	19.780	17.734	14.219	11.821	10.365	9.216	8.421	7.953	7.311

Tabla 2Promedio (Seg) Del Algoritmo De La Ecuacion De Calor En 2D Con OpenMP

Threads	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
Promedio Con OpenMP	***76,1***	***38,4***	***28,3***	***19,7***	***16,5***	***13,5***	***12,1***	***10,4***	***8,9***	***8,5***	***8,0***	***7,3***

Tabla 3Tiempos (Segundos) Ecuacion De Calor En 2D Secuencial

1	69.037
2	69.077
3	68.997
4	69.076
5	69.087
6	69.098
7	69.015
8	69.088
9	69.153
10	69.067
11	69.010
12	69.104
13	69.105
14	69.074
15	69.137
16	69.785
17	69.059
18	69.043
19	69.088

20	69.074
21	69.101
22	69.061
23	69.073
24	69.101
25	69.081
26	69.004
27	69.660
28	69.163
29	69.120
30	69.023

Tabla 4Promedio (Seg) Del Algoritmo De La Ecuacion De Calor En 2D Secuencial

Promedio Secuencial	***69,1***

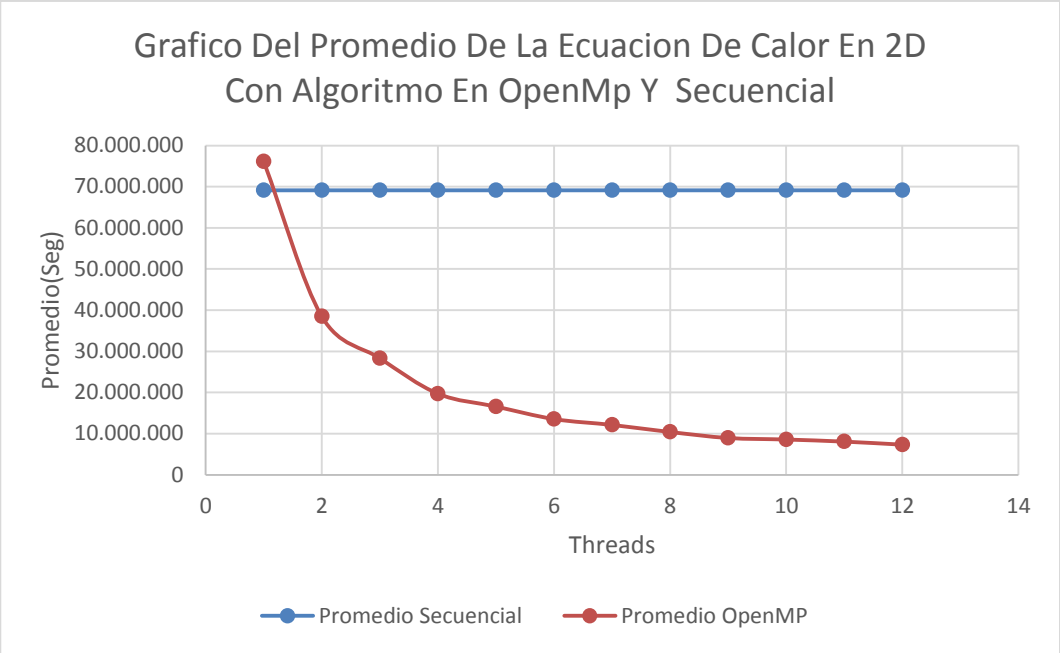


Figura 26 Promedio De La Ecuacion De Calor

Tabla 5Speed Up De La Ecuacion De Calor

Threads	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
Speed Up	***1***	***1,9818***	***2,689***	***3,8629***	***4,6121***	***5,637***	***6,289***	***7,3173***	***8,5506***	***8,953***	***9,5125***	***10,42***

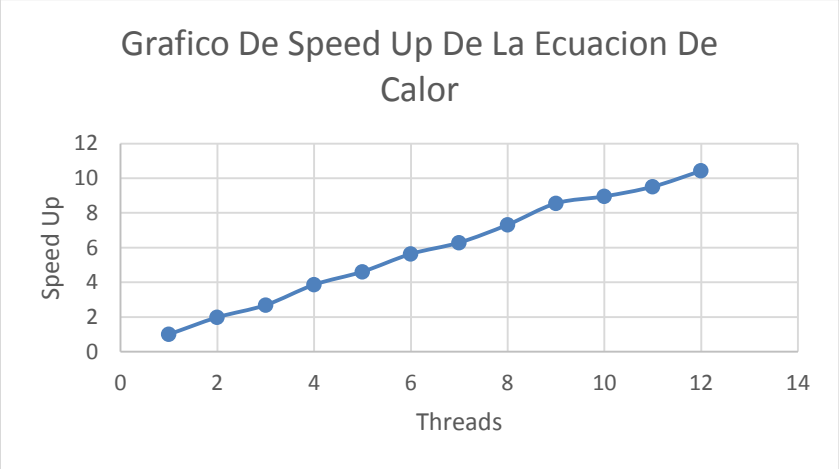


Figura 27 Speed Up De La Ecuacion De Calor

7.2. Ecuacion De Onda Acustica 2D

En la tabla 6 se muestran los resultados (tiempo de procesamiento) de 30 ejecuciones sobre el algoritmo que calcula la solución para la ecuación de onda acústuca 2D mediante el esquema de diferencias finitas. Los tiempos fueron tomados desde la asignación de las condiciones inciales sobre el *grid* respectivo, hasta que finaliza el calculo de 1000 iteraciones. El tiempo de acceso para guardar los resultados en disco no es calculado. La tabla 7 muestra el promedio de las iteraciones con un número determinado de *cores* activos, desde 1 a 12. La tabla 8 muestra el promedio del tiempo para el algoritmo secuencial. El comportamiento del algoritmo al aumentar el número de *cores* utilizados se puede apreciar de mejor forma en la figura 27. En la figura el eje X, corresponde al número de núcleos empleados, y el eje Y, al tiempo de procesamiento en milisegundos. Se observa un descenso importante del tiempo de procesamiento a medida que se incrementa el número de *cores* utilizado. La línea azul se toma como referencia y corresponde al tiempo empleado por el algoritmo secuencial. La tabla 9 muestra los datos del *speedup* del algoritmo y la figura 28 ilustra su comportamiento. Como se ha mencionado, el comportamiento ideal corresponde a la recta de la identidad, que se incrementa proporcionalmente a medida que aumenta el número de cores utilizados, de forma que el speedup para 2 cores es 2 (se duplica la aceleración), para 3 cores es 3, etc. Se puede ver que la gráfica obtenida para este caso, tiene un comportamiento similar al de la identidad, si bien no es ideal, demuestra que la solución saca provecho de los cores utilizados.

.

Tabla 6 Tiempos (Segundos) Ecuacion De La Onda Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	127.477	66.214	49.965	29.679	23.354	21.229	20.193	16.416	15.974	13.210	12.911	11.567
2	118.078	47.518	49.861	31.952	28.346	21.324	19.898	13.913	13.046	13.365	12.815	10.672
3	117.455	47.518	50.627	31.283	23.108	18.834	20.010	15.812	13.166	12.347	12.356	9.899
4	117.186	47.510	48.031	23.752	28.226	18.748	20.011	16.684	12.388	12.982	11.188	10.901
5	116.68	47.375	48.950	27.585	28.548	21.150	20.428	16.648	12.389	13.144	11.088	9.866

6	116.464	47.395	49.791	30.110	28.056	21.338	19.232	16.006	13.183	13.740	13.243	10.016
7	115.845	47.309	48.420	24.133	28.529	20.917	19.885	15.402	13.692	13.879	13.196	9.876
8	115.582	47.408	49.929	24.174	26.611	20.457	20.220	16.109	13.135	13.802	12.160	10.301
9	115.163	47.366	50.938	28.592	29.213	19.324	20.104	15.743	15.756	13.658	10.615	9.371
10	114.809	47.481	50.056	23.948	22.980	19.088	19.984	16.838	12.905	12.757	10.573	9.893
11	114.593	47.390	50.361	27.213	27.607	21.197	18.203	15.706	14.695	12.748	10.540	11.068
12	114.428	47.292	49.034	24.072	27.479	20.678	19.878	15.899	13.064	12.905	12.468	10.493
13	114.255	47.346	49.282	27.734	29.305	20.384	19.235	15.456	12.288	13.019	12.104	9.892
14	114.145	47.338	49.924	31.626	29.079	19.365	19.727	15.445	12.437	13.264	13.001	11.313
15	114.040	47.545	49.354	31.825	29.675	20.745	19.621	16.042	13.604	12.839	11.047	9.706
16	113.517	47.639	49.537	24.134	27.155	20.811	19.698	15.855	15.573	12.800	12.043	10.319
17	113.048	47.562	51.008	23.603	28.499	20.540	20.478	16.482	14.463	11.633	12.134	9.136
18	112.542	47.235	47.620	30.123	28.940	19.943	19.287	16.109	12.956	12.194	12.749	9.945
19	109.432	47.643	49.393	24.231	23.402	20.766	19.747	16.279	13.161	12.343	10.731	9.439
20	109.465	47.403	50.315	27.025	23.096	20.944	19.808	16.035	12.724	12.793	11.154	9.785
21	109.429	47.557	49.352	31.035	23.574	21.329	20.196	15.546	15.322	12.060	12.907	10.813
22	109.465	47.385	48.894	27.869	28.149	19.656	17.351	15.655	15.139	12.844	11.973	9.370
23	122.863	47.400	49.910	29.168	29.171	19.454	19.681	15.891	14.990	12.512	12.344	9.949
24	135.057	47.325	49.383	32.437	28.739	20.478	19.999	16.142	12.672	13.449	11.755	10.433
25	114.625	47.302	48.825	25.024	28.761	21.096	20.350	13.974	13.162	12.755	11.310	9.859
26	114.005	47.282	49.242	23.647	28.980	20.067	19.420	14.897	12.692	11.779	10.930	11.282
27	109.443	47.651	48.723	25.498	29.348	18.916	19.623	16.089	15.680	12.431	12.121	9.882
28	116.555	47.029	50.569	28.919	29.574	21.535	16.889	16.122	15.471	13.473	11.834	10.459
29	109.464	47.364	48.579	23.606	24.317	20.746	17.583	15.325	12.257	13.504	10.752	9.782
30	109.234	47.154	48.979	24.052	25.740	19.365	19.623	16.533	12.314	12.562	12.181	11.613

Tabla 7Promedio(mseg) De Ecuacion De La Onda Con OpenMP

Threads	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
Promedio Con OpenMP	***115145***	***48031***	***49495***	***27268***	***27252***	***20348***	***19545***	***15835***	***13677***	***12893***	***11874***	***10230***

Tabla 8Promedio(mseg) De Ecuacion De La Onda Secuencial

Promedio Secuencial	***116513***

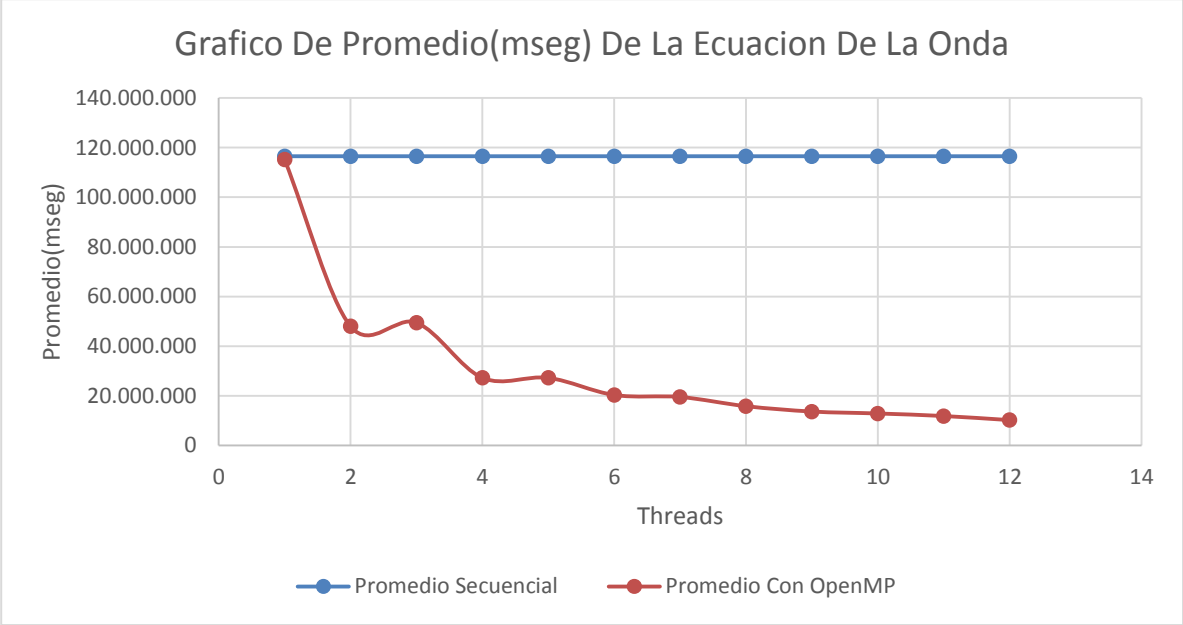


Figura 28 Promedio(mseg) De La Ecuacion De La Onda

Tabla 9Speed Up De La Ecuacion De La Onda

Threads	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
Speed Up	***1***	***2,397***	***2,326***	***4,222***	***4,225***	***5,658***	***5,891***	***7,271***	***8,418***	***8,930***	***9,696***	***11,255***

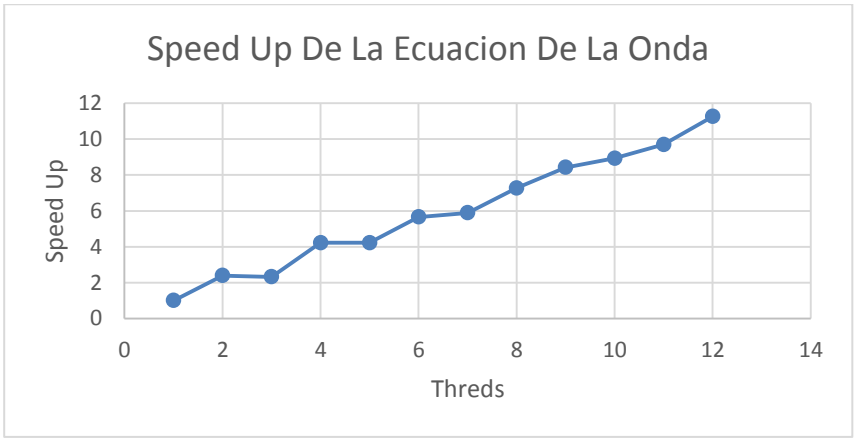


Figura 29 Speed Up De La Ecuacion De La Onda

7.3. Calculo de PI

Dentro de una gran cantidad de algoritmos consultados en la bibliografía, y cuyo análisis se adjunta en los a exos se ha incluido en este documento el clásico algoritmo que aproxima el número Pi, mediante sumas de Riemann, la cual permite realizar sumas de unidades independientes, y donde el factor de precisión depende del número de particiones. Este es un algoritmo clásico que presenta un buen desempeño en tiempo de ejecución y su *speedup*, con lo cual pude tomarse como *benchmark* o punto de referencia para apreciar el buen desempeño de los algoritmos anteriores (métodos de diferencias finitas). La tabla 10 muestra los tiempos de procesamiento obtenidos, para 30 lanzamientos, experimentando con el uso de diferentes nucleos del procesador de 1 a 12. La tabla 12 muestra los tiempos promedio y la tabla 13, el tiempo requerido por el algoritmo secuencial. La figura 29 permite ver el comportamiento que tiene el tiempo de ejecución, el eje X muestra el número de *cores* y el eje Y el tiempo en milisegundos. Se puede apreciar el descenso del tiempo, al aumentar el número de *cores* utilizados. Ña figura 30 muestra el *speedup* del algoritmo. Como en los casos anteriores se observa que la grafica es cercana a la diagonal de la identidad, sin embago, como en los casos anteriores, la pendiente decrece a mediad qu aumenta el número de *cores*, es decir el se aleja del comportamiento ideal cuando aumenta el número de nucleos de procesamiento. Esto sucede también en los ejemplos anteriores, es decir que hay pérdida de la aceleración al incrementar los cores. Sin embargo es de resaltar que los algoritmos anteriores, que hace parte de una clase de algoritmos paralelizables, definidos en la sección 6.2, y que son propios de la computación científica (solución de ecuaciones mediante esquemas de diferencias finitas) son naturalmente paralelizables, mejorando su rendimiento en términos de la reducción del tiempo de procesamiento.

Tabla 10Tiempos (Segundos) calculo de PI con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	0.011777	0.006020	0.004097	0.003195	0.002780	0.002162	0.001941	0.001727	0.001639	0.001473	0.001449	0.001333
2	0.011833	0.006068	0.004086	0.003114	0.002601	0.002227	0.001916	0.001741	0.001557	0.001494	0.001385	0.001337
3	0.011823	0.006064	0.004077	0.003110	0.002553	0.002186	0.001903	0.001724	0.001561	0.001515	0.001377	0.001316
4	0.011815	0.006046	0.004428	0.003118	0.002585	0.002156	0.001896	0.001714	0.001678	0.001454	0.001386	0.001323
5	0.011761	0.006063	0.004109	0.003157	0.002551	0.002208	0.001927	0.001743	0.001627	0.001474	0.001385	0.001384
6	0.012006	0.006014	0.004103	0.003165	0.002818	0.002149	0.001922	0.001763	0.001700	0.001468	0.001364	0.001329
7	0.012051	0.006027	0.004115	0.003101	0.002537	0.002225	0.001963	0.001733	0.001566	0.001452	0.001372	0.001389
8	0.011772	0.006056	0.004068	0.003138	0.002585	0.002169	0.001956	0.001731	0.001584	0.001484	0.001366	0.001323
9	0.011843	0.006049	0.004114	0.003126	0.002599	0.002188	0.001902	0.001795	0.001584	0.001461	0.001411	0.001323
10	0.011818	0.006068	0.004077	0.003167	0.002529	0.002154	0.001916	0.001721	0.001615	0.001451	0.001410	0.001430
11	0.011773	0.006053	0.004103	0.003126	0.002527	0.002495	0.001944	0.001761	0.001580	0.001489	0.001376	0.001328
12	0.011761	0.006047	0.004098	0.003167	0.002522	0.002189	0.001910	0.001805	0.001590	0.001481	0.001391	0.001307
13	0.012043	0.006008	0.004122	0.003129	0.002588	0.002180	0.001910	0.001779	0.001629	0.001530	0.001369	0.001325
14	0.011760	0.006059	0.004352	0.003157	0.002587	0.002150	0.001892	0.001780	0.001557	0.001456	0.001355	0.001409
15	0.011776	0.006064	0.004335	0.003159	0.002561	0.002155	0.001893	0.001785	0.001591	0.001544	0.001369	0.001325
16	0.011799	0.006047	0.004082	0.003173	0.002518	0.002155	0.001903	0.001799	0.001561	0.001478	0.001374	0.001451
17	0.011825	0.006025	0.004105	0.003154	0.002551	0.002175	0.001962	0.001714	0.001577	0.001515	0.001407	0.001327
18	0.011770	0.006102	0.004070	0.003168	0.002566	0.002185	0.001902	0.001739	0.001632	0.001493	0.001374	0.001310
19	0.011810	0.006014	0.004058	0.003205	0.002538	0.002290	0.001942	0.001779	0.001589	0.001455	0.001416	0.001327
20	0.011765	0.006008	0.004064	0.003103	0.002540	0.002154	0.001915	0.001728	0.001559	0.001510	0.001367	0.001326
21	0.012031	0.006049	0.004077	0.003097	0.002521	0.002201	0.002006	0.001721	0.001561	0.001453	0.001373	0.001312
22	0.011810	0.006036	0.004109	0.003159	0.002525	0.002175	0.001946	0.001714	0.001591	0.001460	0.001514	0.001315
23	0.011771	0.006053	0.004070	0.003099	0.002595	0.002162	0.001893	0.001724	0.001570	0.001517	0.001386	0.001309

24	0.011757	0.006054	0.004114	0.003100	0.002560	0.002201	0.001901	0.001785	0.001597	0.001452	0.001426	0.001308
25	0.012092	0.006042	0.004126	0.003092	0.002776	0.002227	0.002208	0.001719	0.001799	0.001493	0.001372	0.001351
26	0.011758	0.006018	0.004068	0.003140	0.002571	0.002153	0.001912	0.001797	0.001563	0.001538	0.001360	0.001336
27	0.011831	0.006051	0.004073	0.003152	0.002558	0.002204	0.001940	0.001732	0.001579	0.001473	0.001373	0.001311
28	0.011770	0.006075	0.004087	0.003152	0.002578	0.002360	0.001900	0.001721	0.001574	0.001455	0.001377	0.001306
29	0.011773	0.006062	0.004104	0.003152	0.002588	0.002177	0.001895	0.001727	0.001645	0.001463	0.001382	0.001321
30	0.011776	0.005999	0.004074	0.003172	0.002557	0.002155	0.001893	0.002009	0.001560	0.001453	0.001621	0.001523

Tabla 11Promedio(Seg) De PI con OpenMP

Threads	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
Promedio Con OpenMP	***0,011***	***0,006***	***0,004***	***0,003***	***0,0025***	***0,0021***	***0,0019***	***0,0017***	***0,0016***	***0,0014***	***0,0013***	***0,0013***

Tabla 12Tiempos (Segundos) del calculo de PI secuencial

1	0.011737
2	0.011750
3	0.011750
4	0.011750
5	0.011989
6	0.011741
7	0.011748
8	0.011760
9	0.011749
10	0.011749
11	0.011987
12	0.011992
13	0.011750
14	0.011750
15	0.011749
16	0.011750
17	0.011750
18	0.011749
19	0.011751
20	0.011739
21	0.011739
22	0.011751
23	0.011750

24	0.011741
25	0.011741
26	0.011739
27	0.011750
28	0.011739
29	0.011750
30	0.011988

Tabla 13Promedio(Seg) De PI Secuencial

Promedio Secuencial	***0,011***

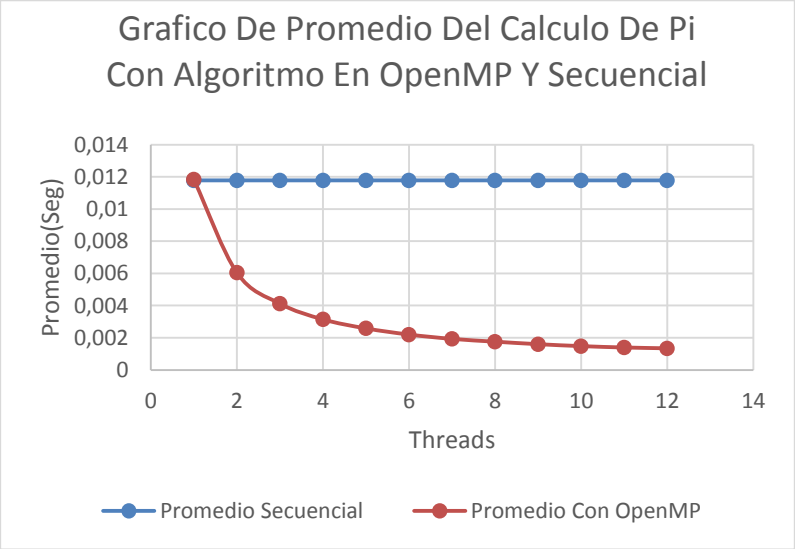


Figura 30Promedio De PI con OpenMP Y Secuencial

Tabla 14Speed Up del cálculo de PI

Threads	1	2	3	4	5	6	7	8	9	10	11	12
Speed Up	1	1,83	2,75	3,66	4,4	5,23	5,78	6,47	6,875	7,85	8,46	8,46

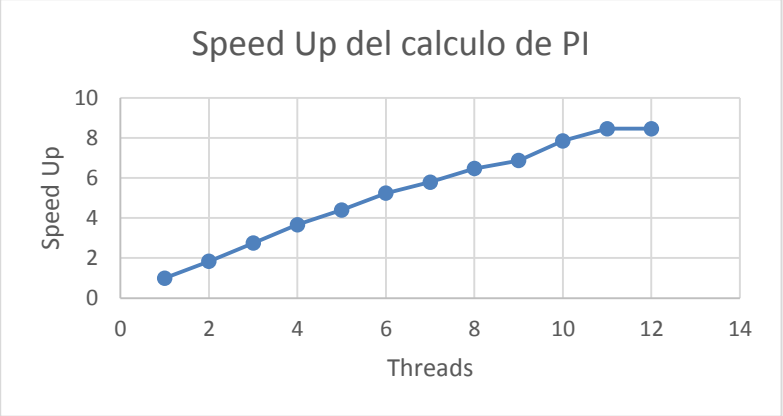


Figura 31Speed Up del calculo de PI

8. Conclusiones

- Se reviso el estado del arte relacionado con el desarrollo de algoritmos mediante el paradigma de memoria compartida a través de la API de OpenMP, encontrandose articulos valiosos de gran interés en el que se proponen metodologias para la programacion paralela con OpenMP, aplicaciones recientes utilizando este modelo, mejoras en el modelo actual asi como algunas propuestas para la enseñanza del mismo.

- Para el contexto especifico de este trabajo se tomaron difentes ejemplos tipo implementados mediante la API OpenMP en lenguaje C, en el que se les analizo el rendimiento de la versión secuencial frente a la paralela en el servidor Intel(R) Xeon(R) CPU E5-2403 v2 @ 1.80GHz ubicado en la Universidad De Pamplona, observandose que a mayor numero de threads menor tiempo de procesamiento.

- En el presente documento se explica en detalle algunos ejemplos relacionados con una de las clases de problemas sugeridos en la bibliografía. Los relacionados con *grids* estructurados, específicamente la solución de ecuaiones diferenciales parciales mediante diferencias finitas, observando que se obtienen buenos resultados en cuanto a tiempo de procesamiento y aceleración. Sin embargo, la bibliografía consultada sugiere que existe un amplio espectro de problemas que pueden ser abordados desde modelos de programación paralela, y se sale del alcance de este trabajo el mostrar ejempos en cada una de las 13 clases de problemas sugeridos en la bibliografía. A

pesar de ellos el presente trabajo ofrece una base para la exploración de problemas específicos dentro de cada clase sugerida.

- Existen 2 formas de obtener un algoritmo en paralelo, el primero paralelizar un algoritmo secuencial lo cual interesa detectar las partes con mayor coste computacional y el segundo programar en paralelo desde cero analizando las caracterisitcas propias de la aplicacion para determinar el algoritmo paralelo mas adecuado.

- Especificamente en el caso de programación de memoria compartida con OpenMP, se destacan tres escenarios para paralelizar un algoritmo secuencial. Paralelismo de bucles, que son los primeros candidatos para ser paralelizados, paralelismo de regiones y parelalismo de tareas. En cualquiera de los casos, es recomendable hacer un análisis de dependencias utilizando por ejemplo la metodología de Foster, para determinar si el problema es paralelizable o no.

9. Referencias

Acevedo Alvaro Casasús, Juan Jose Benito Muñoz Prieto, F. U., & Corvinos, L. G. (2009). Resolución de la ecuación de Ondas en 2-D y 3-D utilizando diferencias finitas generalizadas . Consistencia y Estabilidad . Alvaro Casas u Francisco Ure n Introducci ´ on Diferencias finitas generalizadas y método explícito en, *2009*, 1–8.

Alcubierre, M. (2005). Introducción a FORTRAN. *Fortran*, 34.

Amit Amritkar, S. D., & Tafti, D. (2013). Efficient parallel CFD-DEM simulations using OpenMP.

Asanovic, K., Catanzaro, B. C., Patterson, D. A., & Yelick, K. A. (2006). The Landscape of Parallel Computing Research : A View from Berkeley.

Atienza, D., Colmenar, J. M., & Garnica, O. (2010). Parallel heuristics for scalable community detection. *Parallel Computing*. https://doi.org/10.1016/j.parco.2010.07.001

Auckenthaler, T., Blum, V., Bungartz, H.-J., Huckle, T., Johanni, R., Krämer, L., … Willems, P. . (2011). Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations.

Barbara Chapman, Gabriele Jost, R. V. D. P. (n.d.). *Using OpenMP, Portable Shared Memory Parallel Programming*.

Blelloch, G. E. (2012). NESL: A Nested Data-Parallel Language.

Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, T. R. (2008). Data and thread affinity in openmp programs.

Dorta, A., García, L., González, J. R., León, C., & Rodríguez, C. (n.d.). Aproximación paralela a la técnica Divide y Vencerás.

Eduard Ayguad, Nawal Copty, Duran Alejandro, Hoeflinger Jay, Yuan Lin, Massaioli Federico, Su Ernesto, Unnikrishnan Priya, G. Z. (n.d.). A Proposal for Task Parallelism in OpenMP.

Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, G. Z. (2009). The Design of OpenMP Tasks.

Foster, I. (1995). Designing and Building Parallel Programs. *Interface*, (Noviembre). https://doi.org/10.1109/MCC.1997.588301

Francisco Almeida, Domingo Giménez, José Miguel Mantas, A. M. V. (n.d.). Sobre la situación del paralelismo y la programación paralela en los Grados de Ingeniería Informática.

Frederico Pratasa, Pedro Trancosob, Leonel Sousaa, Alexandros, S., & Shid Guochun, K. V. (2012). Fine-grain parallelism using multi-core, Cell/BE, and GPU Systems.

Haoqiang Jina, Dennis Jespersena, Piyush Mehrotraa, Rupak Biswasa, Lei Huangb, B. C. (2011). High performance computing using MPI and OpenMP on multi.

J.Aguilar, E. L. (2006). *Introducción a la Computación Paralela. Memory*. https://doi.org/10.1157/13068212

Jiangzhou He, W. C., & Zhizhong, T. (2015). NestedMP: Enabling cache-aware thread mapping for nested parallel shared memory applications.

John H. Gibbons. (1989). High Performance Computing and Networking for Science September 1989. *Performance Computing*, (September).

Julio Monetti, O. L. (n.d.). USO DE THREADS PARA LA EJECUCIÓN EN PARALELO SOBRE UNA MALLA COMPUTACIONAL.

Marco Oliverio, William Spataro, Donato D'Ambrosio, Rocco Rongo , Giuseppe Spingola, G. T. (2011). OpenMP parallelization of the SCIARA Cellular Automata lava flow model.

Michael J, Q. (n.d.). *Parallel Programming in C with MPI andOpenMP* (Vol. 1).

Nasim Muhammad, H. J. E. (n.d.). OpenMP Parallelization of a Mickens Time-Integration Scheme for a Mixed-Culture Biofilm Model and Its Performance on Multi-core and Multi-processor Computers.

OpenMP. (2013). OpenMP Application Program Interface. *The OpenMP Forum, Tech. Rep*, (July), 320. https://doi.org/10.1080/08905769008604595

Patrick Carribault, Marc Pérache, H. J. (2010). Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC.

Piccoli, M. F. (2011). *Computación de alto desempeño en GPU*. Retrieved from http://sedici.unlp.edu.ar/bitstream/handle/10915/18404/Documento_completo__.pdf?sequence=1

Reinders, J., & Jeffers, J. (2015). High Performance Parallelism Pearls, Volume One. *High Performance Parallelism Pearls*. https://doi.org/10.1016/B978-0-12-802118-7.00007-8

Sampieri, R. H., Collado, C. F., & Lucio, P. B. (n.d.). *Metodologia de*.

Santa Cruz, C. (2007). Programando en Fortran. *Fortran*, 118.

Saraswat, V., Tardieu, O., Grove, D., Cunningham, D., Takeuchi, M., & Herta, B. (2012). A Brief Introduction to X10 (for the High Performance Programmer). *The IBM Corporation*, *10*.

Severance, C. (2010). High Performance Computing. *Hpc*. Retrieved from http://www.computer.org/csdl/mags/pd/1994/03/p3085.pdf

Sima, D., Fountain, T. J., Kacsuk, P., Sima, D., Fountain, T. J., & Kacsuk, P. (n.d.). Part IV . Chapter 15 - Introduction to MIMD Architectures Thread and process-level parallel architectures are typically realised by MIMD (Multiple Instruction Multiple Data) computers . This class of parallel computers is the most general one since it permits autonomous operations on a set of data by a set of processors without any architectural restrictions . Instruction level data-parallel architectures should satisfy several constraints in order to build massively parallel systems . For example processors in array processors , systolic architectures and cellular automata should work synchronously controlled by a common clock . Generally the processors are very simple in these systems and in many cases they realise only a special function (systolic arrays , neural networks , associative processors , etc .). Although in recent SIMD architectures the complexity and generality of the applied processors have been increased , these modifications have resulted in the introduction of process-level parallelism and MIMD features into the last generation of data-parallel computers (for example CM-5), too . MIMD architectures became popular when progress in integrated circuit technology made it possible to produce microprocessors which were relatively easy and economical to connect into a multiple processor system . In the early eighties small systems , incorporating only tens of processors were typical . The appearance of Transputer in the mid-eighties caused a great breakthrough in the spread of MIMD parallel computers and even more resulted in the general acceptance of parallel processing as the technology of future computers . By the end of the eighties mid-scale MIMD computers containing several hundreds of processors become generally available . The current generation of MIMD computers aim at the range of massively parallel systems containing over 1000 processors . These systems are often called scalable parallel computers . 15 . 1 Architectural concepts The MIMD architecture class represents a natural generalisation

of the uniprocessor von Neumann machine which in its simplest form consists of a single processor connected to a single memory module . If the goal is to extend this architecture to contain multiple processors and memory modules basically two alternative choices are available : a . The first possible approach is to replicate the processor / memory pairs and to connect them via an interconnection network . The processor / memory pair is ca…, 1–12.

GUIA PARA LA REALIZACION DE LAS PRUEBAS DE LOS EJEMPLOS TIPO MEDIANTE EL PARADIGMA DE MEMORIA COMPARTIDA OPENMP

AUTOR
ERIKA VIVIANA RIAÑO BEJAR

DIRECTOR
JOSE ORLANDO MALDONADO BAUTISTA

DEPARTAMENTO DE INGENIERÍAS ELÉCTRICA ELECTRÓNICA SISTEMAS Y TELECOMUNICACIONES
FACULTAD DE INGENIERÍAS Y ARQUITECTURA
INGENIERIA DE SISTEMAS
UNIVERSIDAD DE PAMPLONA
PAMPLONA
2016

INTRODUCCIÓN

Propósito del Documento

El presente documento está dirigido a entregar las pautas para la ejecución de las pruebas a los diferentes ejemplos en el servidor ubicado en los laboratorios de alto rendimiento de la universidad de pamplona, usando el compilador gcc junto con el lenguaje C y un archivo perl.

CONCEPTOS IMPORTANTES

¿Qué Es Un Archivo PERL?

Perl es un lenguaje de programación utilizado para construir aplicaciones CGI para el web. Perl es un acrónimo de Practical Extracting and Reporting Languaje, trata de un lenguaje de programación práctico para extraer información de archivos de texto y generar informes a partir del contendido de los ficheros.

Perl es un lenguaje de programación interpretado es decir que el código de los scripts en Perl no se compila, sino que cada vez que se quiere ejecutar se lee el código y se pone en marcha interpretando lo que hay escrito. Además, es extensible a partir de otros lenguajes, ya que desde Perl se puede hacer llamadas a subprogramas escritos en otros lenguajes. También desde otros lenguajes se puede ejecutar código Perl.

¿Qué Es SCP?

Scp permite que Scp permite que los archivos se copien a, desde, o entre diferentes hosts. Utiliza ssh para la transferencia de datos y proporciona la misma autenticación y el mismo nivel de seguridad que ssh

¿Qué Es GCC?

es un conjunto de compiladores creados por el proyecto GNU. GCC es software libre y lo distribuye la Free Software Foundation (FSF) bajo la licencia general pública GPL. Originalmente GCC significaba *GNU C Compiler* (*compilador GNU de C*), porque sólo compilaba el lenguaje C. Posteriormente se extendió para compilar C++, Fortran, Ada y otros.

¿Qué Es SSH?

SSH™ (o *S*ecure *SH*ell) es un protocolo que facilita las comunicaciones seguras entre dos sistemas usando una arquitectura cliente/servidor y que permite a los usuarios conectarse a un host remotamente. SSH encripta la sesión de conexión, haciendo imposible que alguien pueda obtener contraseñas no encriptadas.

El protocolo SSH proporciona los siguientes tipos de protección:

- Después de la conexión inicial, el cliente puede verificar que se está conectando al mismo servidor al que se conectó anteriormente.

- El cliente transmite su información de autenticación al servidor usando una encriptación robusta de 128 bits.
- Todos los datos enviados y recibidos durante la sesión se transfieren por medio de encriptación de 128 bits, lo cual los hacen extremamente difícil de descifrar y leer.
- El cliente tiene la posibilidad de reenviar aplicaciones X11 desde el servidor. Esta técnica, llamada reenvío por X11, proporciona un medio seguro para usar aplicaciones gráficas sobre una red.

¿Qué Es Un Servidor?

Es un ordenador y sus programas, que está al servicio de otros ordenadores. El servidor atiende y responde a las peticiones que le hacen los otros ordenadores. Los otros ordenadores, que le hacen peticiones, serán los "clientes" del servidor.

¿Qué Es OpenMP?

OpenMP es una API para la programación en paralelo de memoria compartida, esta contiene directivas de compilador para la programación de multihilos, variables de ambiente y librerías que controlan la ejecución en paralelo, donde cada hilo tiene acceso a toda la memoria disponible, viéndose el sistema como una colección de cores o procesadores donde estos tienen acceso a la misma memoria global compartida, permitiendo así el aprovechamiento incremental del código. OpenMP trabaja con estándares como Fortran, C/ C++ combinando algoritmos de forma serial y paralelo dentro del mismo código fuente.

MobaXterm

Es un sistema de herramientas de la red incluidas en un solo archivo portable del exe. MobaXterm integra un servidor X y varios clientes de red (SSH, RDP, VNC, Telnet, rlogin, sftp y ftp), accesible a través de un terminal basado en pestañas. MobaXterm también integra un sistema completo de comandos del Unix.

Versión: 9.4

CodeBlocks

Es un *IDE libre de C, C ++ y Fortran* construido para satisfacer las necesidades más exigentes de sus usuarios. Está diseñado para ser muy extensible y completamente configurable.

Por último, un IDE con todas las características *que* necesita, con una apariencia coherente, sensación y operación a través de plataformas.

Construido en torno a un marco de complemento, Code :: Blocks se puede *ampliar con complementos* . Cualquier tipo de funcionalidad se puede agregar mediante la instalación / codificación de un complemento. Por ejemplo, la compilación y depuración de la funcionalidad ya está a cargo de los complementos

MinGW

Proporciona un conjunto completo de herramientas de programación de código abierto que es adecuado para el desarrollo de aplicaciones MS-Windows nativas y que no dependen de

ninguna DLL de C-Runtime de terceros. (Depende de una serie de DLL proporcionados por Microsoft, como componentes del sistema operativo, entre los que destaca MSVCRT.DLL, la biblioteca de tiempo de ejecución de Microsoft C. Adicionalmente, las aplicaciones threaded ***deben ser*** enviadas con una DLL de soporte de subprocesos distribuible libremente, Proporcionada como parte de MinGW).

Los compiladores de MinGW proporcionan acceso a la funcionalidad del tiempo de ejecución de Microsoft C y algunos tiempos de ejecución específicos del idioma. MinGW , siendo Minimalist, no intenta, y nunca intentará, proporcionar un entorno de tiempo de ejecución POSIX para la implementación de la aplicación POSIX en MS-Windows . Si desea implementar POSIX en esta plataforma, considere Cygwin en su lugar.

3. GUIA PARA LA REALIZACION DE LAS PRUEBAS

Mediante la conexión remota a través de ssh se logró acceder al servidor ubicado en los laboratorios de alto rendimiento la universidad de pamplona, en él se realizaron las pruebas para comprobar el rendimiento de los diferentes tipos de ejercicios.

Las pruebas se realizaron en un servidor con las siguientes características:
Servidor HP ProLiant DL180, con dos procesadores de la serie Intel Xeon E5 2600 v3, con 6 nucleos cada uno (Para un total de 12 core efectivos). Memoria 16 DDR4. Almacenamiento 1Terabyte.

Pasos En Ubuntu

Pasos Previos

1. Lo primero que se debe realizar es verificar que todos los ejercicios están ejecutándose correctamente, para luego comentar cada una de las partes de código que imprime los resultados dejando solamente el tiempo de ejecución.

2. Se copian los archivos ejecutables en la carpeta BIN en el servidor cluster mediante el comando scp.

    a. Primero se deben copiar los archivos al servidor pasarela así:

```
scp heated_plate pasarela@softwarelibre.unipamplona.edu.co:/home/pasarela
```

Figura 32Copiar ejecutable al servidor pasarela

Donde heated_plate es el nombre del ejecutable.

    b. Al tener el archivo ejecutable en el servidor pasarela se procede a copiarlo al servidor clúster en la carpeta openmp/BIN

```
scp heated_plate cluster@192.168.19.131:/home/cluster/openmp/BIN
```

Figura 33Copiar ejecutable al servidor cluster

Nota:

- En el caso que se desee copiar la carpeta BIN con todos los ejecutables al servidor pasarela.

```
scp -r BIN pasarela@softwarelibre.unipamplona.edu.co:/home/pasarela
```

Figura 34 Copiar carpeta BIN al servidor pasarela

Luego se copia la carpeta BIN con todos los archivos al servidor clúster

```
scp -r BIN cluster@192.168.19.131:/home/cluster/openmp
```

Figura 35Copiar carpeta BIN al servidor clúster

Pasos Posteriores

1. Conectarse al servidor pasarela mediante el siguiente comando

```
ssh pasarela@softwarelibre.unipamplona.edu.co
```

Figura 36Conexion al servidor pasarela

password: pasarela

3. Luego se procede a ingresar al servidor clúster:

```
ssh cluster@192.168.19.131
```

Figura 37Conexion al servidor clúster

password: master

4. Se crea un directorio (carpeta) llamado OpenMP y dentro de este los siguientes subdirectorios BIN, TOOLS y SOLUCIONES.

- BIN: Contiene todos los ejecutables de los diferentes ejemplos.
- TOOLS: Contiene un archivo llamado lanzador.pl este es un archivo PERL que permite ejecutar varios ejemplos, con los threads y las iteraciones que se desee sin la necesidad a estar ejecutándolo manualmente.
- Soluciones: Contiene las carpetas generadas por el archivo PERL.

5. Se crea el archivo lanzador.pl dentro de la carpeta TOOLS. El archivo contiene las siguientes características:

```
$Rep = 30; #veces a repetir
@Algoritmos  = ("heated_plate","PI","NumerosPrimos","MultiTask");#NOMBREeJECUTABLE
@Nucleos = (1,2,3,4,5,6,7,8,9,10,11,12);
```

Figura 38Características del archivo lanzador.pl

$Rep=30 Indica el número de iteraciones a ejecutar

@**Algoritmos**=("heated_plate","PI","NumerosPrimos","MultiTask"); es el nombre de los archivos ejecutables.

@**Nucleos = (1,2,3,4,5,6,7,8,9,10,11,12);** Cada uno de los threads a ejecutarse

Con esta instrucción se define el nombre que tendrá cada uno de los archivos generados en la carpeta soluciones.

```
foreach $Nombre (@Algoritmos) {
 foreach $nnucleos (@Nucleos) {
        $file = "$Path/Soluciones/$carpeta/$Nombre-nucleos".$nnucleos.".dat";
         print $file."\n";
          for ($k=0; $k<$Rep; $k++) {
              system("$Path/BIN/$Nombre 0 0 >> $file\n");
          }
          close($file);
    }
}
```

Figura 39Creacion del nombre de los archivos pruebas

A continuación, se muestra el archivo lanzador.pl completo:

```
#!/usr/bin/perl
if (@ARGV[0]) {
   $carpeta = "$ARGV[0]";
   chomp($carpeta);
} else {
   print "Falta el nombre de la carpeta como argumento ";
   exit (0);
};
$path0 = `pwd`;
chomp($path0);
$T = index($path0,"T")-1;
$Path = substr($path0,0,$T);

if (-d $Path."/Soluciones/".$carpeta) {
   print "la carpeta $carpeta ya existe y podria sobreescribir los datos ";
   exit (0);
} else {
    system ("mkdir $Path"."/Soluciones/".$carpeta);
};
system("clear");
&fechaactual();
$Rep = 30; #veces a repetir
@Algoritmos  = ("heated_plate","PI","NumerosPrimos","MultiTask");#NOMBREeJECUTABLE
@Nucleos = (1,2,3,4,5,6,7,8,9,10,11,12);

foreach $Nombre (@Algoritmos) {
 foreach $nnucleos (@Nucleos) {
        $file = "$Path/Soluciones/$carpeta/$Nombre-nucleos".$nnucleos.".dat";
        print $file."\n";
          for ($k=0; $k<$Rep; $k++) {
              system("$Path/BIN/$Nombre 0 0 >> $file\n");
          }
          close($file);
    }
}
&fechaactual();
exit(0);
sub fechaactual{
 my ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =  localtime(time);
 $year += 1900;
 $mon++;
 -print "$mday/$mon/$year $hour:$min:$sec\n";
}
```

Figura 40 Archivo PERL (Lanzador.pl)

6. Una vez ingresados los threads, las iteraciones y el nombre de cada uno de los ejecutables en el archivo lanzador.pl, se procede a su ejecución.

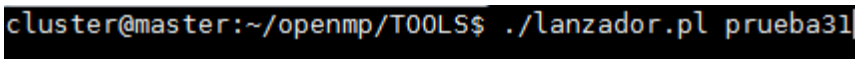


Figura 41Ejecución del archivo perl

Donde prueba31 es el nombre de la carpeta que va a contener todos los tiempos de ejecución de cada uno de los archivos ejecutables.

Al presionar enter se genera el procesamiento de cada ejecutable así:

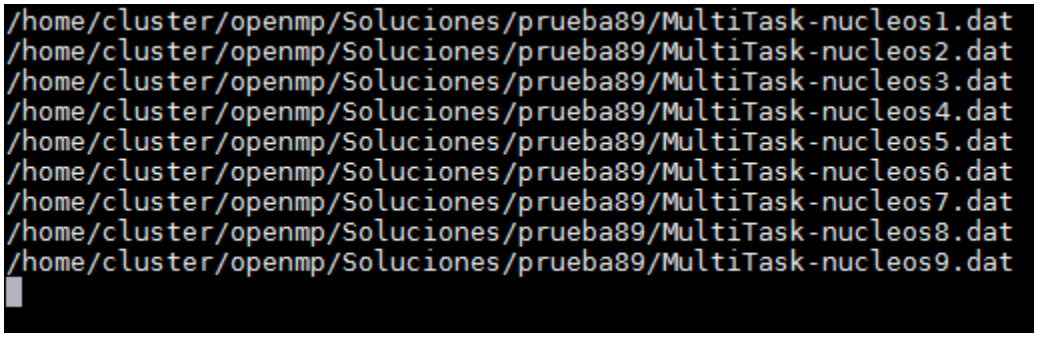


Figura 42Ejecucion de cada ejecutable mediante el archivo perl

Cabe destacar que la computadora se debe dejar solo y exclusivamente la consola con la ejecución del archivo PERL, es decir no se puede tener abierto otro programa (navegador, editores de texto, juegos, etc.), debido a que interfieren en los tiempos de procesamiento y por ende los resultados generados son erróneos.

7. Se procede a revisar la carpeta soluciones para verificar que los archivos se hayan creado correctamente.
8. Se analiza los resultados generados por cada thread

9. Ingresar los datos en Excel para hallar el promedio, el speedUp así como cada una de las gráficas, esto con el fin de analizar el comportamiento de las dos versiones del algoritmo.

Pasos En Windows

Para la realizar las pruebas en Windows se usa el programa MobaXterm_Personal_9.4, este programa soporta comandos Unix siendo esta una ventaja debido a que los pasos realizados en Ubuntu se pueden aplicar en Windows.

1. Para ingresar al servidor pasarela, presionamos el botón que dice new session

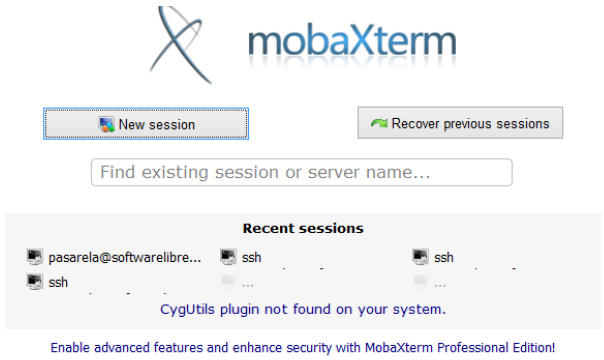


Figura 43Entorno Grafico

2. Una vez presionado el botón, se coloca el comando ***ssh pasarela@softwarelibre.unipamplona.edu.co*** para ingresar al servidor pasarela en el cuadro de texto remote host y la contraseña del servidor pasarela en este caso es ***pasarela*** en el cuadro de texto specify username

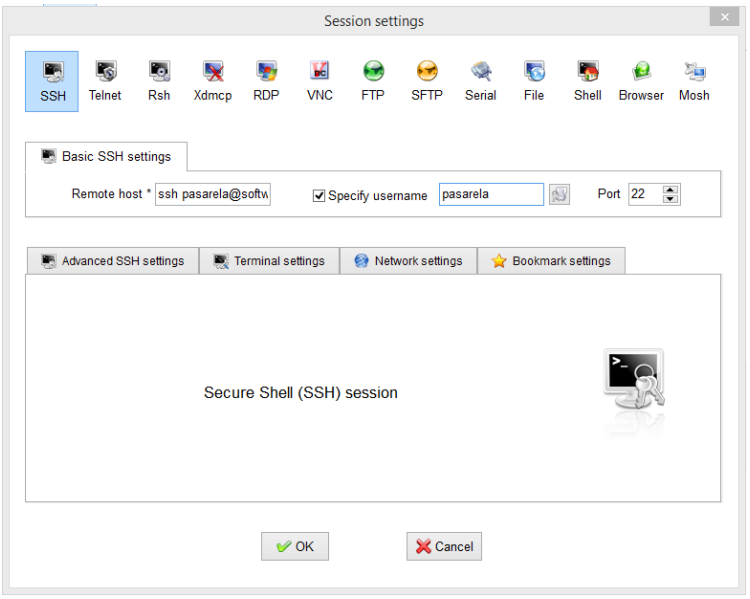


Figura 44Acceder al servidor pasarela

3. Una vez ingresado al servidor pasarela se procede a ingresar al servidor clúster

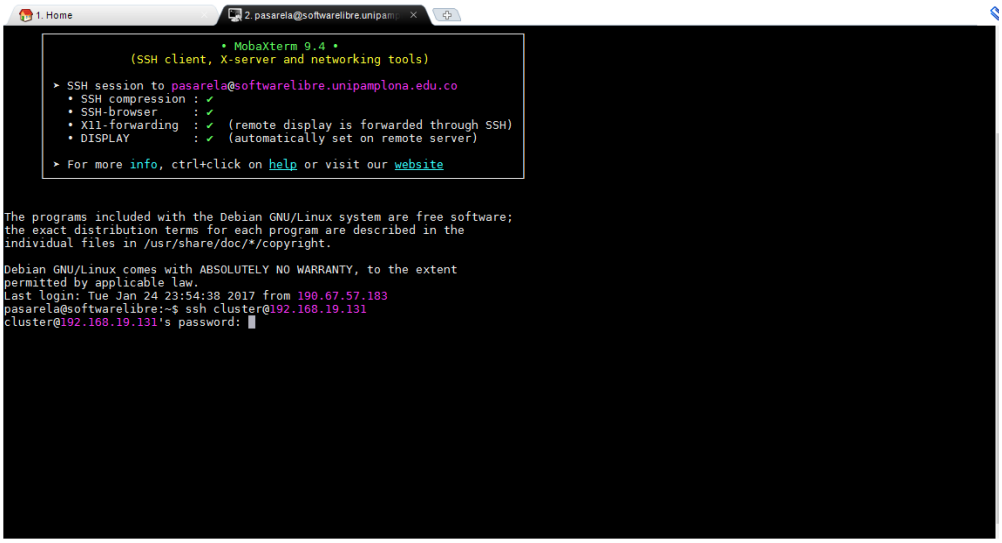


Figura 45Acceder al servidor clúster

4. Para enviar un archivo del pc al servidor pasarela, se arrastra el archivo a servidor

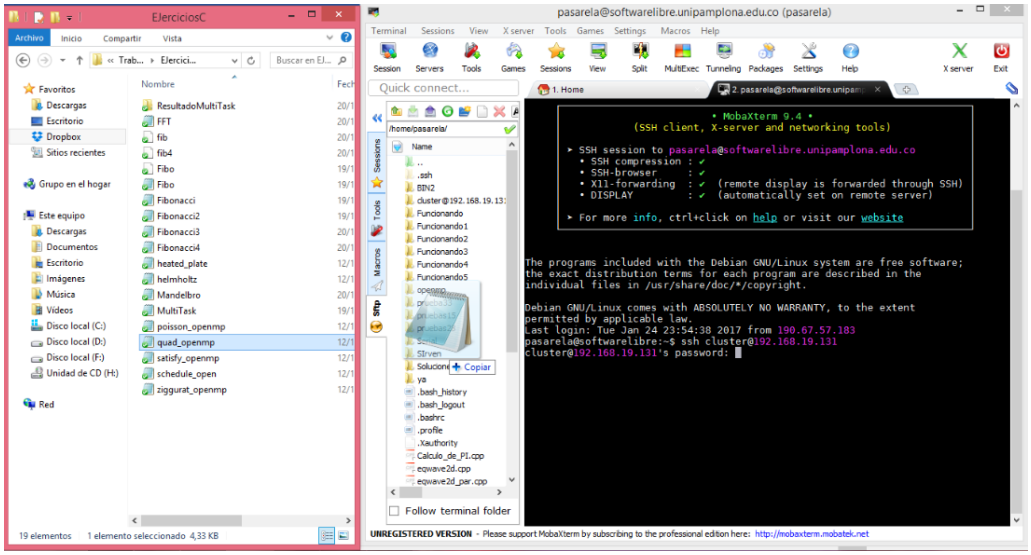


Figura 46Enviar Archivos De Pc Al Servidor

5. Se repiten los pasos realizados en Ubuntu descritos al inicio de este documento.

4. Guía Para la Instalación De OpenMP

Pasos En Windows

1. Se descarga e instala el CodeBlocks, en este caso se descarga el primero de la lista de ejecutables.

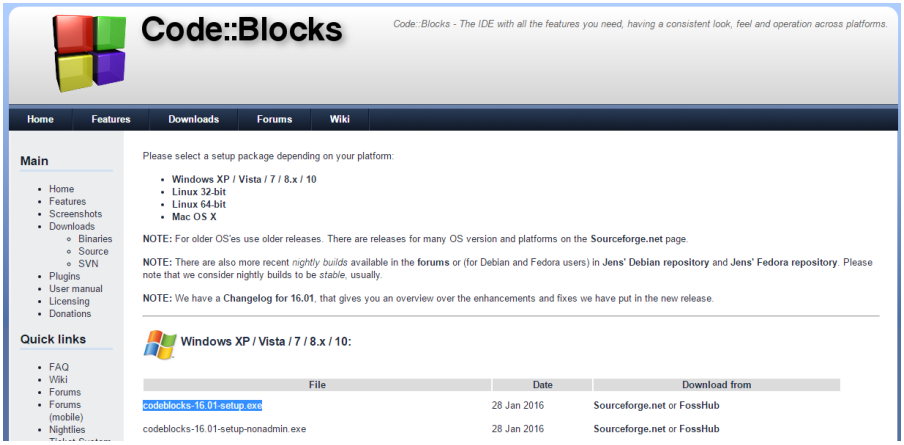


Figura 47Descargar CodeBlocks

2. Una vez presionado el botón aparece la siguiente página, se presiona Download para descargarlo

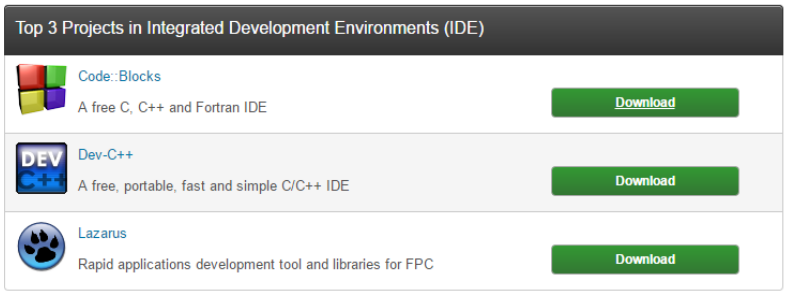


Figura 48Pagina de descarga

3. Se procede con la instalación, se presiona next

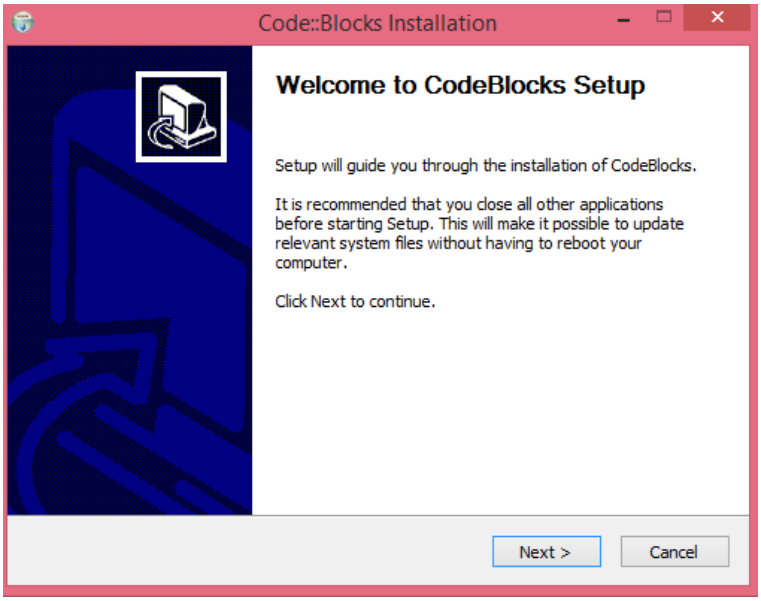


Figura 49Inicio de la instalación

4. Se presiona I Agree

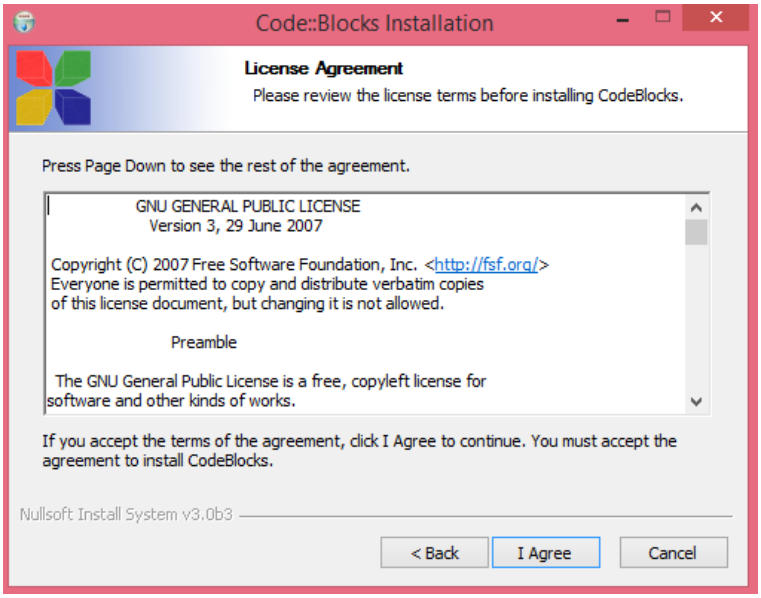


Figura 50Licencia

5. Se seleccionan todas las opciones y se presiona I Agree

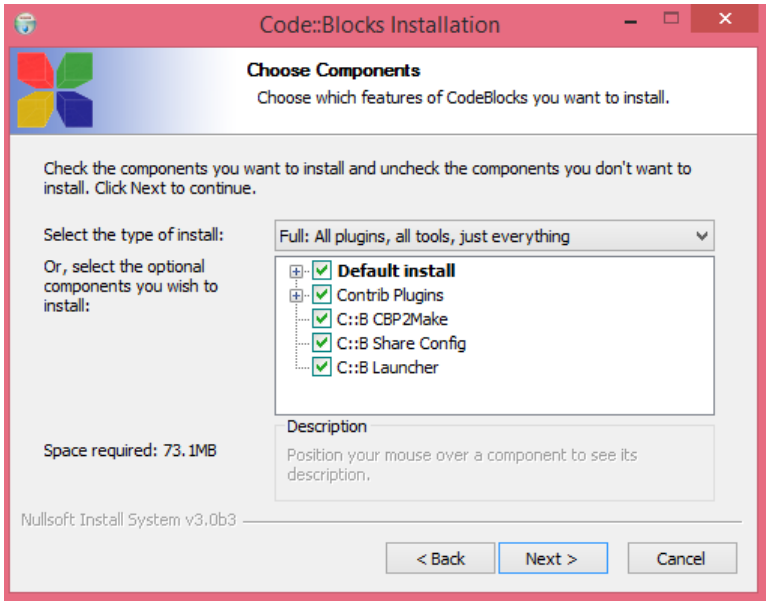


Figura 51Seleccionar componentes

6. Se ubica la carpeta donde se desea instalar y se presiona Install

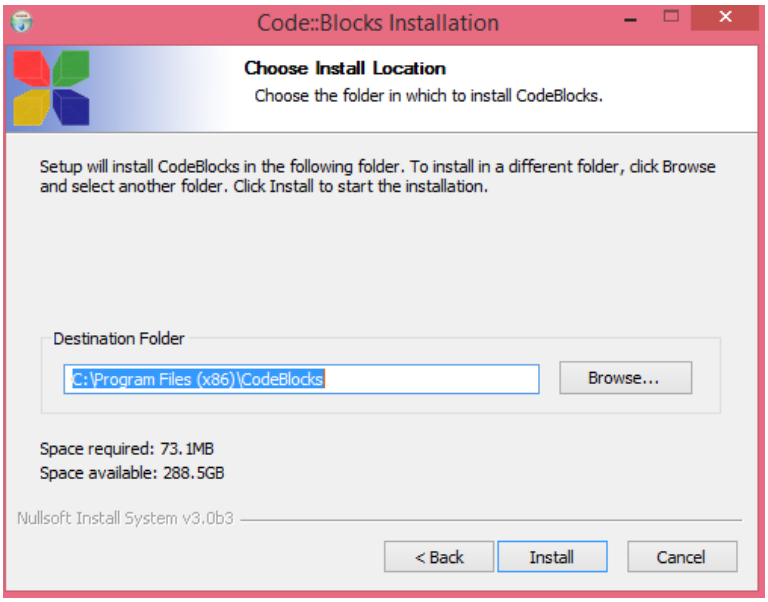


Figura 52Ubicacion de la instalación

7. Se espera a que se copien los archivos del programa y se presiona next

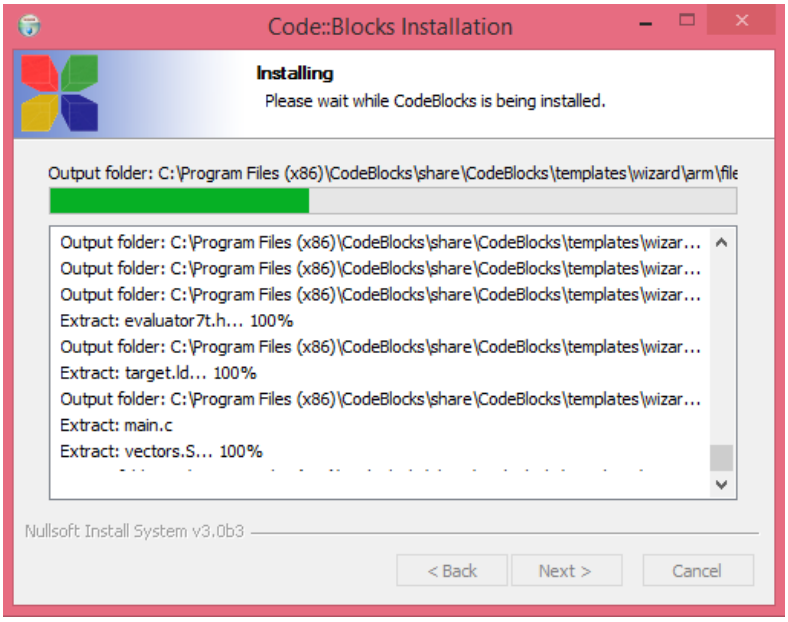


Figura 53Copiando componentes del programa

8. Se presiona el botón si.

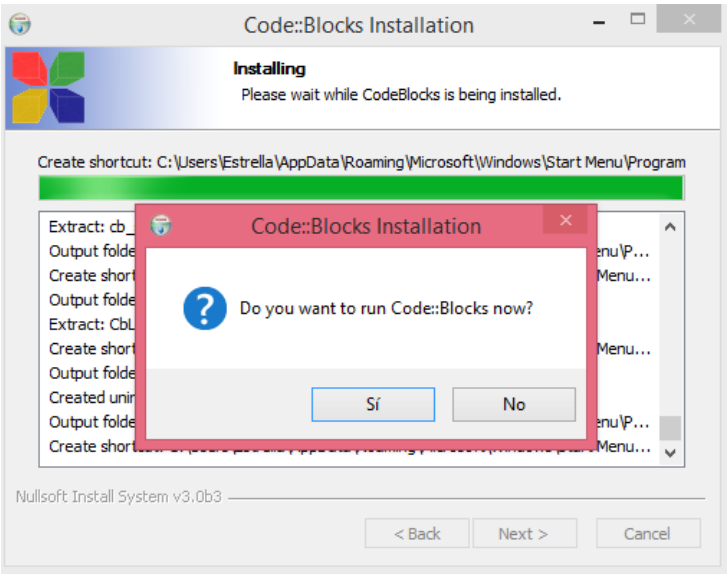


Figura 54Terminada la instalación

9. Se genera la ventana del programa.

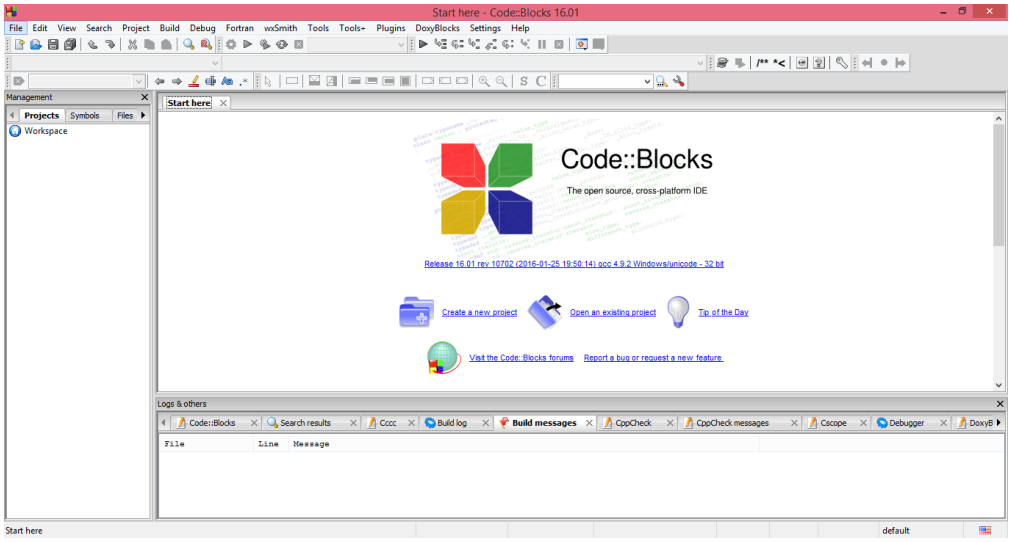


Figura 55Entorno grafico

10. Se procede a configurarlo presionando el botón settings/compiler, generando la siguiente ventana.

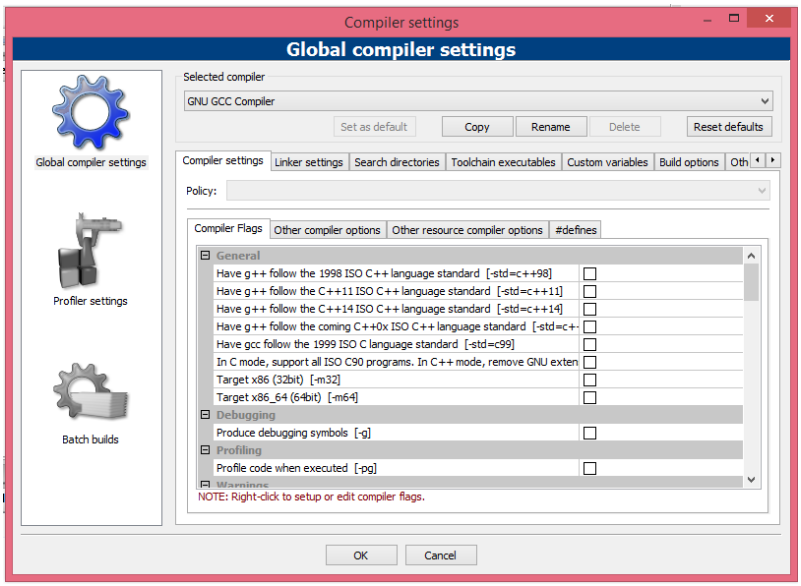


Figura 56Configuración

11. Se coloca -fopenmp en la casilla other compiler options

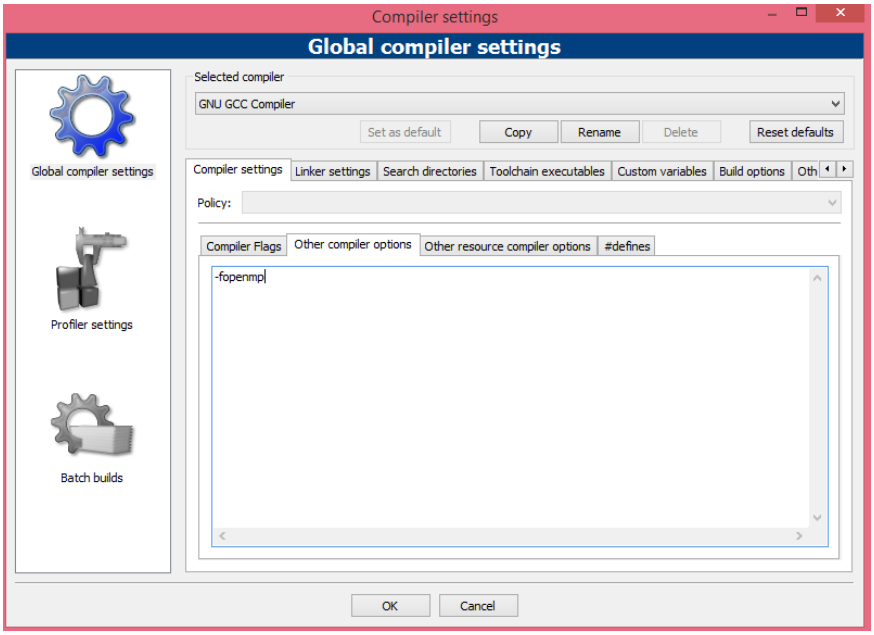


Figura 57Adicion de -fopenmp al compilador

12. Se descarga e instala MinGW

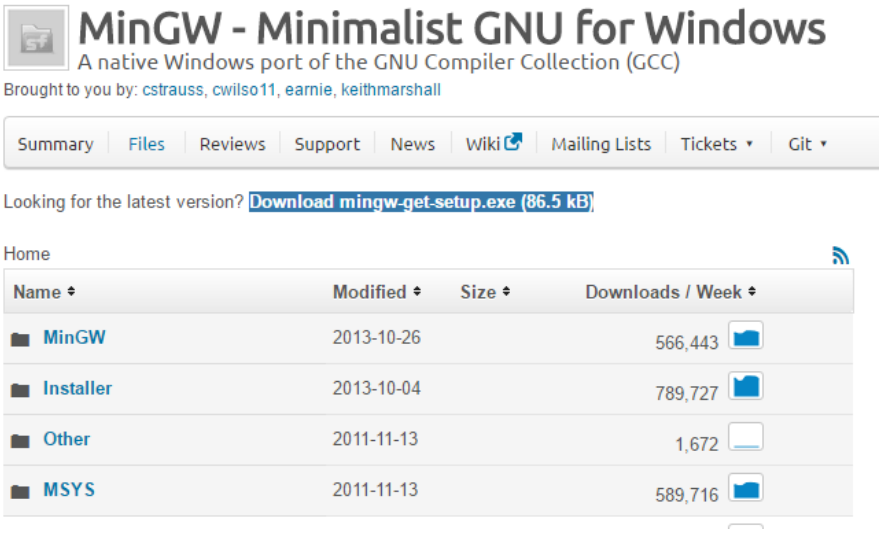


Figura 58Pagina de descargar de MinGW

13. Se inicia la instalación del programa, presionando el botón install

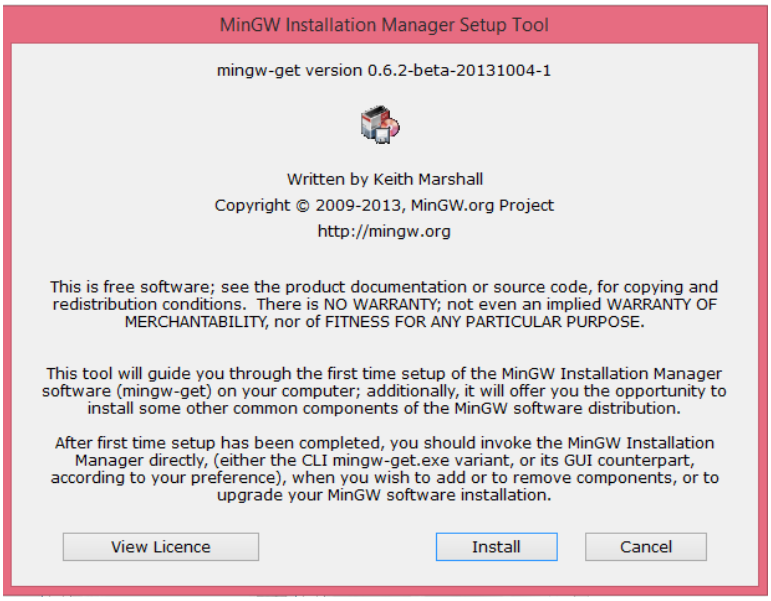


Figura 59Inicio de la instalación

14. Se presiona continúe

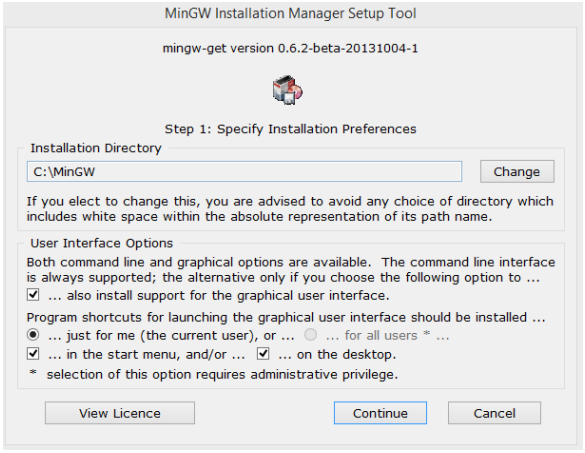


Figura 60Ubicación del archivo a instalar

15. Aparece la siguiente ventana, se espera que cargue y presionamos continúe

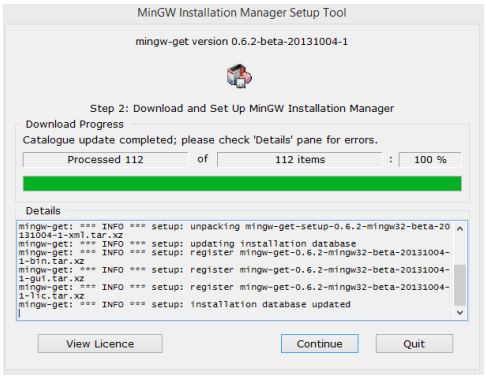


Figura 61Los archivos se están copiando

16. Se marcan todas las casillas

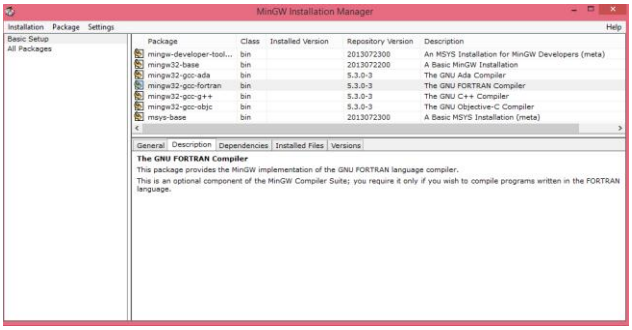


Figura 62Selección de los programas a instalar

17. Se inicia la descarga de los paquetes

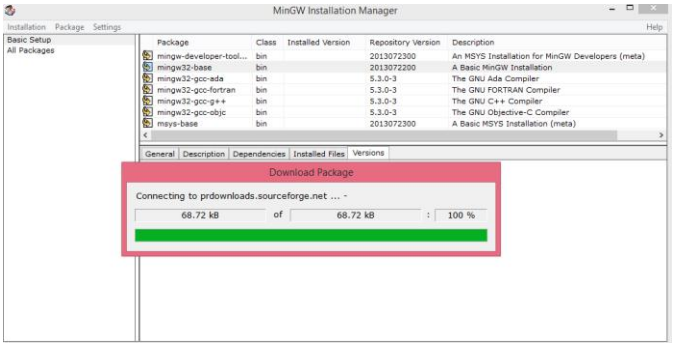


Figura 63nstalacion de los programas seleccionados

18. Una vez instalado los paquetes queda así:

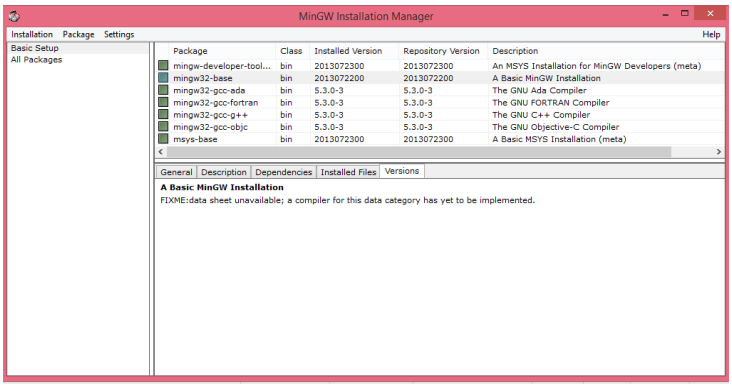


Figura 64Instalacion completada

19. Se agregan las bibliotecas con el botón add de la opción linker settings

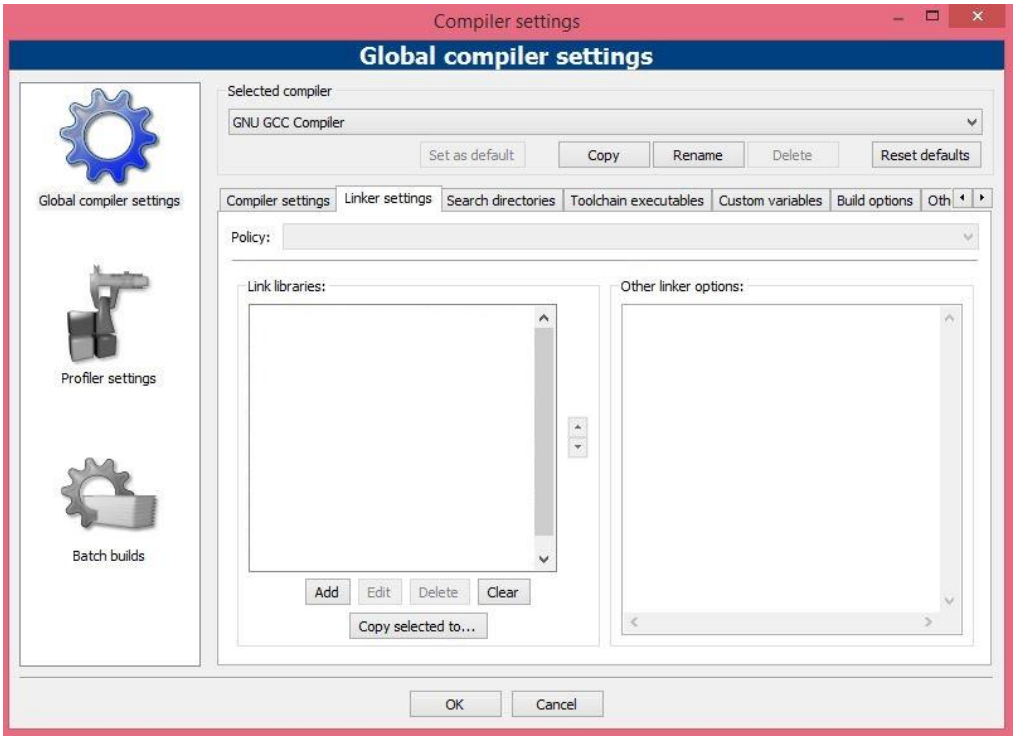


Figura 65Agregar librerías al CodeBlocks

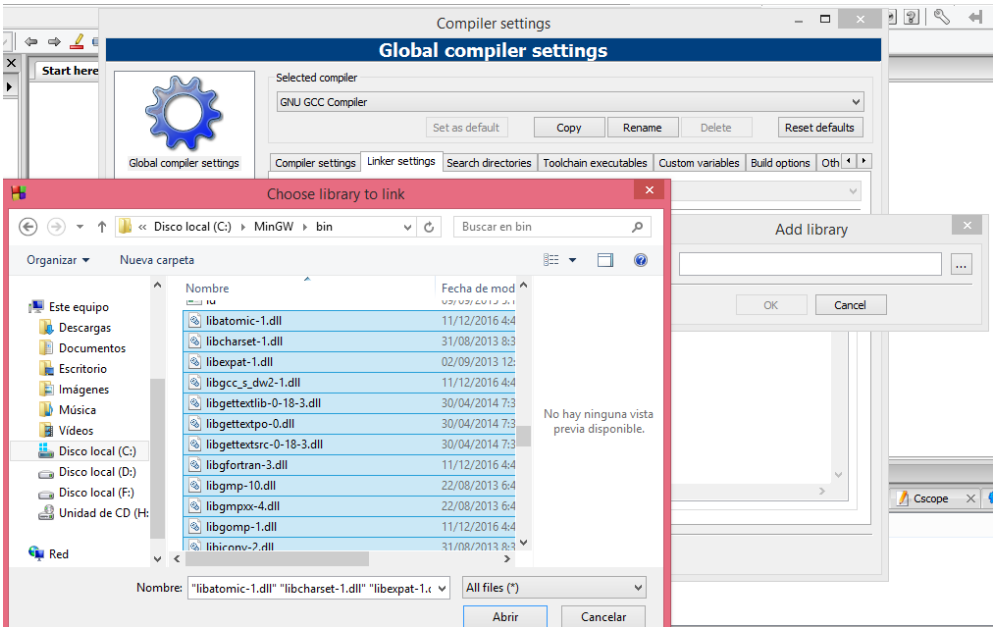


Figura 66Adicion de librerias

20. Una vez realizado lo anterior se procede a revisar un ejemplo.

Figura 67Algoritmo Hello world con 4 threads

21. Se ejecuta y compila

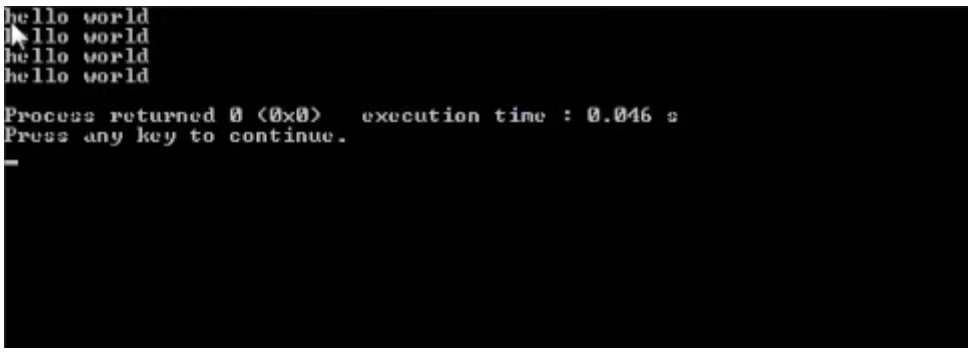


Figura 68Impresion de hello world

ANEXOS

Tabla De Contenido

Listas De Figuras

Lista De Tablas

10. Ejemplos Tipo

10.1. Multiplicacion De Matrices

Algoritmo en OpenMP: Anexos, sección 2,8
Algoritmo Secuencial: Anexos, sección 3,8
Tablas de tiempos: Anexos, sección 4,8

Tabla 15 Promedio(Seg) De Multiplicacion De Matrices Con OpenMP Y Secuencial

Threads	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Promedio Con OpenMP	*1,62*	*0,85*	*0,6*	*0,46*	*0,4*	*0,3*	*0,28*	*0,26*	*0,25*	*0,23*	*0,19*	*0,14*

Promedio Secuencial	*1,15*

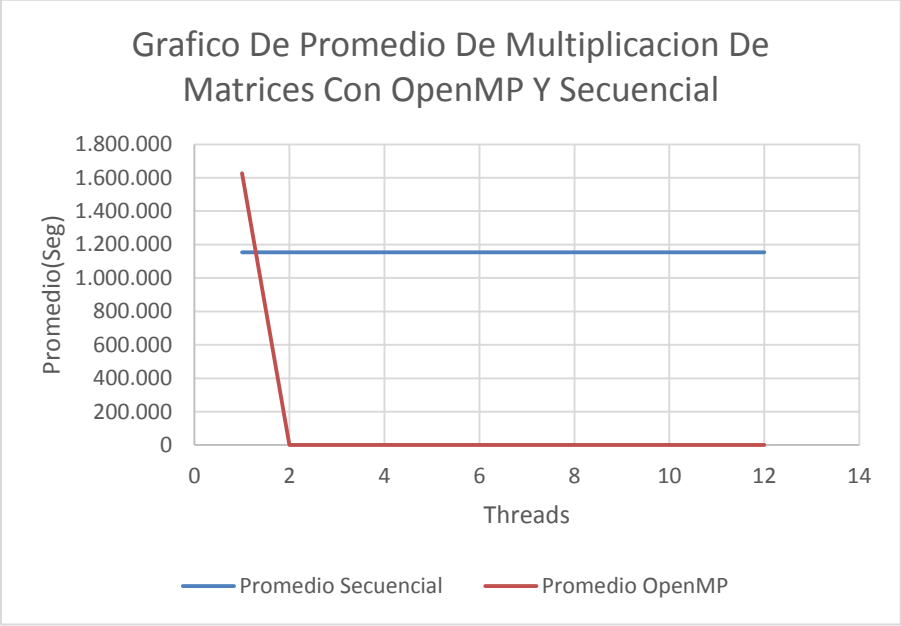


Figura 69 Promedio Multiplicacion De Matrices Con OpenMP Y Secuencial

10.2. Multiplicacion de matrices con omp for nowait

Algoritmo en OpenMP: Anexos, sección 2,15
Algoritmo Secuencial: Anexos, sección 3,15
Tablas de tiempos: Anexos, sección 4,15

Tabla 16Promedio Multiplicacion De Matrices Con Omp For Nowait

Threads	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Promedio Con OpenMP	*100,4*	*50,8*	*31,8*	*24,5*	*19,5*	*16,3*	*14,1*	*12,2*	*10,9*	*9,7*	*9*	*8,3*

Tabla 17Promedio (Seg) Multiplicacion De Matrices Secuencial

Promedio Secuencial	*51,6*

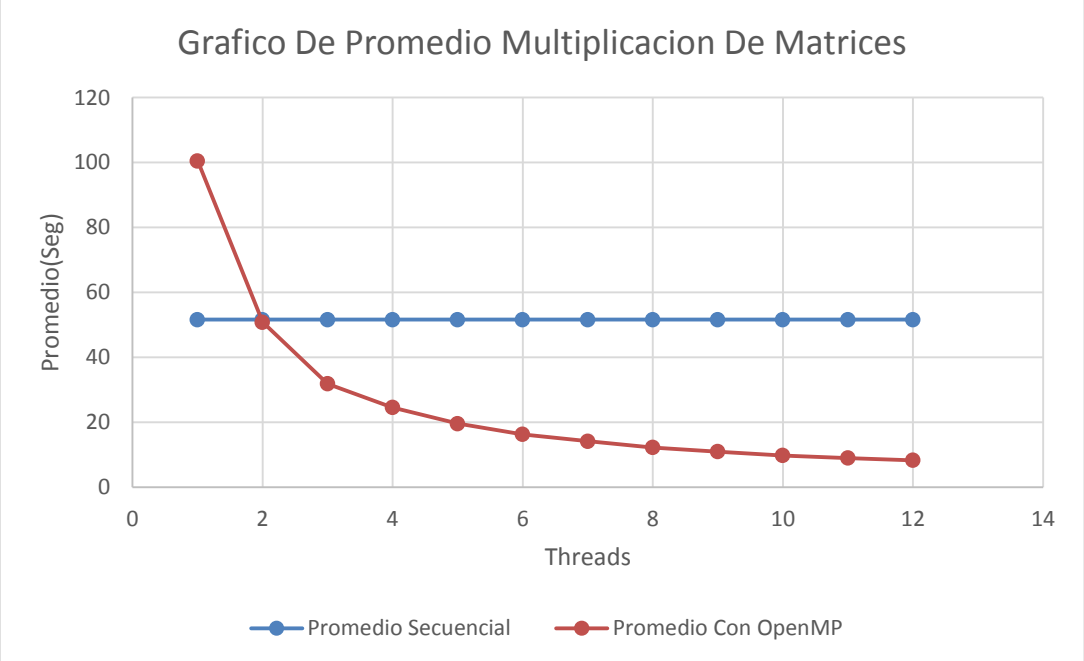


Figura 70

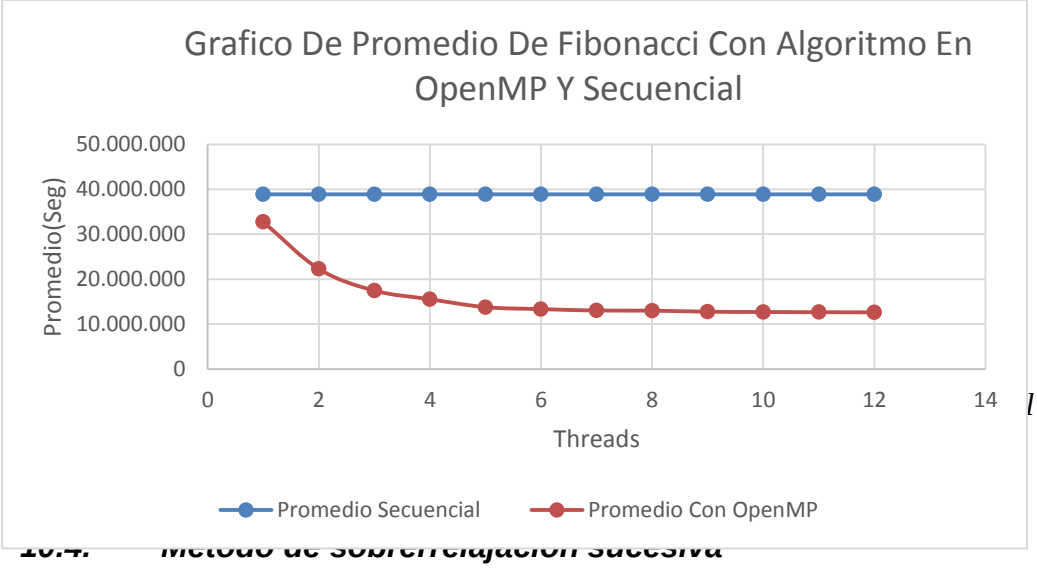
10.3. Fibonacci

Algoritmo en OpenMP: Anexos, sección 2,11
Algoritmo Secuencial: Anexos, sección 3,11
Tablas de tiempos: Anexos , sección 4,11

Tabla 18Promedio(Seg) De Fibonacci Con OpenMP Y Secuencial

Threads	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Promedio con OpenMP	*32,7*	*22,2*	*17,4*	*15,5*	*13,7*	*13,3*	*13*	*12,9*	*12,7*	*12,68*	*12,64*	*12,60*

Promedio Secuencial	*38,8*



l

10.4. Metodo de sobrerrelajacion sucesiva

Tabla 19Promedio(Seg) Del Metodo De Sobrerrelajacion Sucesiva Con OpenMP Y Secuencial

Threads	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Promedio con OpenMP	*0,92*	*0,53*	*0,43*	*0,33*	*0,26*	*0,22*	*0,21*	*0,19*	*0,17*	*0,16*	*0,15*	*0,13*

Promedio Secuencial	*0,76*

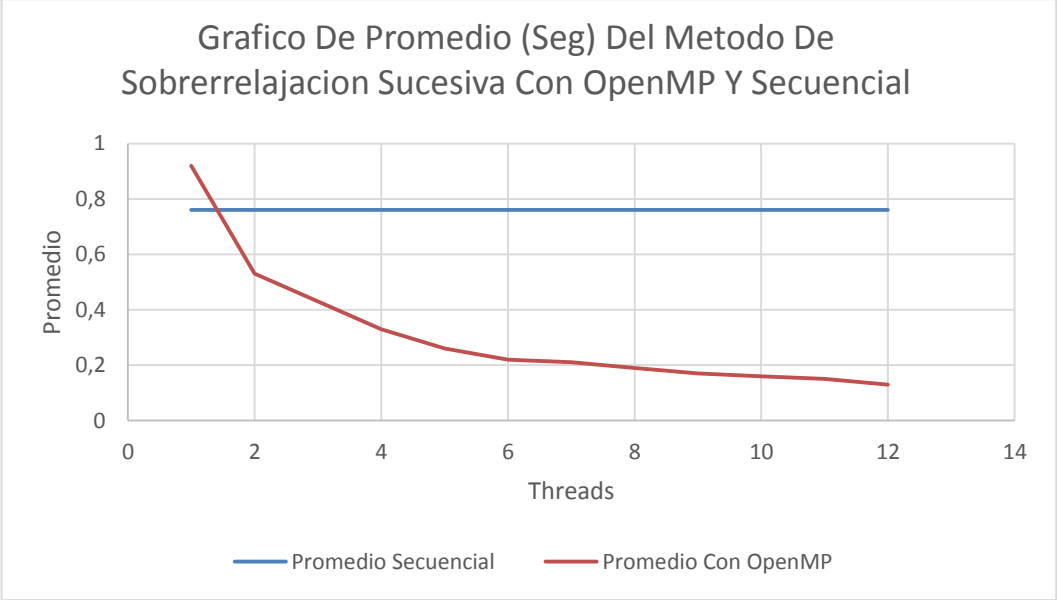


Figura 72 Metodo De Sobrerrelajacion Sucesiva

Algoritmo en OpenMP: Anexos, sección 2,10
Algoritmo Secuencial: Anexos, sección 3,10
Tablas de tiempos: Anexos, sección 4,10

10.5. Calculo de PI

El numero PI se define como la integral asi:

$$\int_0^4 \frac{4}{1+x^2}$$

La aproximación de una integral mediante la suma de Riemann permite dividir el trabajo en unidades independientes, siendo un factor de precision el numero de divisiones.

Tabla 20 Promedio(Seg) De PI con OpenMP Y Secuencial

Threads	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
Promedio Con OpenMP	***0,011***	***0,006***	***0,004***	***0,003***	***0,0025***	***0,0021***	***0,0019***	***0,0017***	***0,0016***	***0,0014***	***0,0013***	***0,0013***

Promedio Secuencial	***0,011***

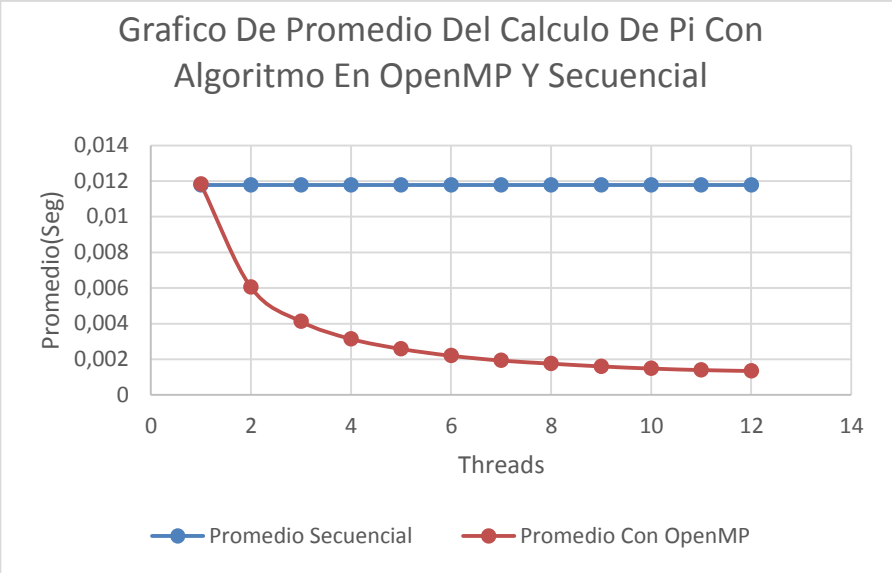


Figura 73Promedio De PI con OpenMP Y Secuencial

Algoritmo en OpenMP: Anexos, sección 2,12
Algoritmo Secuencial: Anexos, sección 3,12
Tablas de tiempos: Anexos, sección 4,12

10.6. Numeros Primos

Tabla 21 Promedio(Seg) De Numeros Primos Con OpenMP Y Secuencial

Threads	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Promedio Con OpenMP	*58,7*	*41,8*	*31,7*	*24,9*	*20,5*	*17,5*	*15,5*	*13,4*	*12,08*	*10,9*	*10,2*	*9,3*

Promedio Secuencial	*56,3*

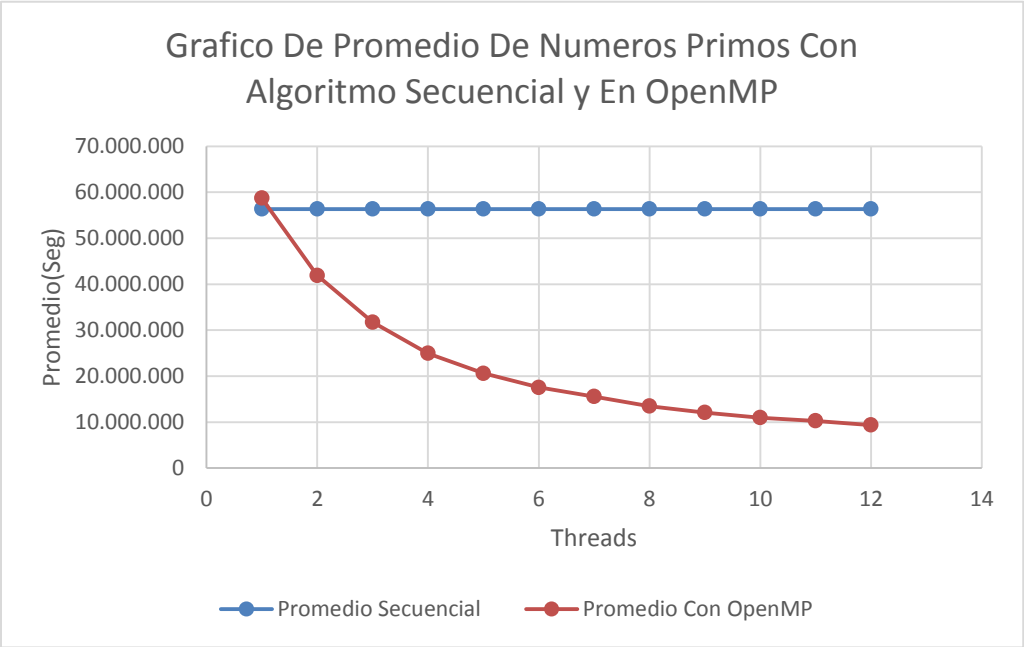


Figura 74 Promedio Numeros Primos Con OpenMP Y Secuencial

Algoritmo en OpenMP: Anexos, sección 2,13
Algoritmo Secuencial: Anexos, sección 3,13
Tablas de tiempos: Anexos, sección 4,13

10.7. Poisson

Es un programa C que calcula una solución aproximada a la ecuación de Poisson en una región rectangular, utilizando OpenMP para llevar a cabo la iteración de Jacobi en paralelo.

La versión de la ecuación de Poisson que se está resolviendo aquí es

$$-\left(\frac{d}{dx}\frac{d}{dx}+\frac{d}{dy}\frac{d}{dy}\right) U(x,y) = F(x,y) \quad \textbf{(1)}$$

Sobre el rectángulo 0 <= X <= 1, 0 <= Y <= 1, con solución exacta

$$\boldsymbol{U(x,y) = sin(pi * x * y)}\ \textbf{(2)}$$

$$\boldsymbol{F(x,y) = pi^2 * (x^2 + y^2) * sin(pi * x * y)}\ \textbf{(3)}$$

Y con condiciones de frontera de Dirichlet a lo largo de las líneas $x = 0$, $x = 1$, $y = 0$ y $y = 1$. (Las condiciones de contorno serán realmente cero en este caso, pero escribimos el problema como si no supiéramos que, Lo que hace que sea fácil cambiar el problema más tarde.)

Calculamos una solución aproximada discretizando la geometría, suponiendo que $DX = DY$, y aproximando al operador de Poisson por

$$\frac{\boldsymbol{U(i-1,j)+U(i+1,j)+U(i,j-1)+U(i,j+1)-4*U(i,j)}}{\boldsymbol{dx/dy}}\ \textbf{(4)}$$

Junto con las condiciones de frontera en los nodos fronterizos, tenemos un sistema lineal para U. Podemos aplicar la iteración de Jacobi para estimar la solución al sistema lineal.

OpenMP se utiliza en este ejemplo para llevar a cabo la iteración de Jacobi en paralelo. Tenga en cuenta que la iteración de Jacobi puede converger muy lentamente, y la lentitud aumenta a medida que la matriz se hace más grande. Por lo tanto, si debes usar la iteración de Jacobi, el paralelismo puede ayudarte. Pero también podría encontrar, en algún momento, que conseguir un mejor solucionador lineal (¡incluso uno no paralelo!) Le ayudaría más. (OPENMP C Examples of Parallel Programming with OpenMP, 2011)

Tabla 22 Promedio (Seg) De La Ecuacion De Poisson Con OpenMP Y Secuencial

Threads	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
Promedio con OpenMP	***16,1***	***10,8***	***8,2***	***6,7***	***5,6***	***5,2***	***4,8***	***4,4***	***4,3***	***4***	***3,9***	***3,7***

Promedio Secuencial	***14***

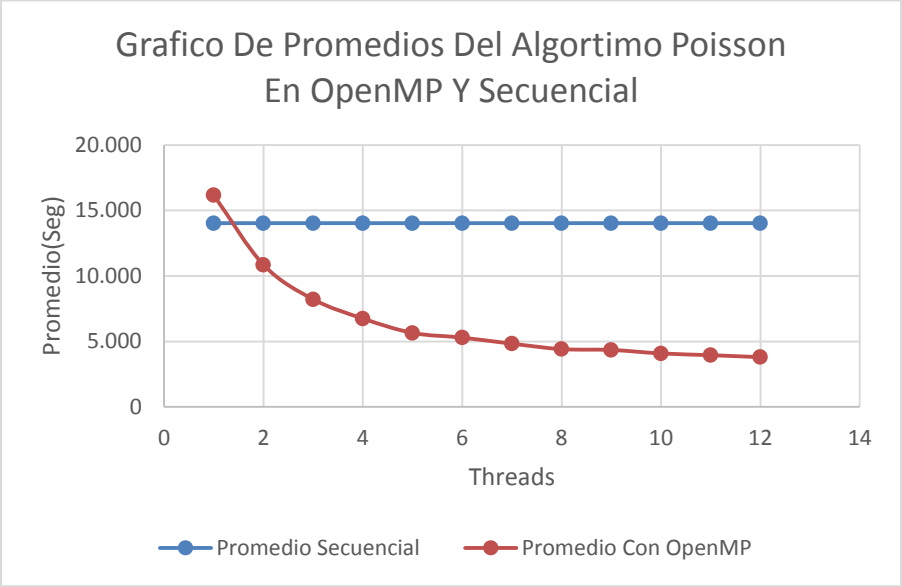


Figura 75 Ecuacion de Poisson Con OpenMP Y Secuecial

Algoritmo en OpenMP: Anexos, sección 2,1

Algoritmo Secuencial: Anexos, sección 2,1

Tablas de tiempos: Anexos, sección 3,1

11. Algoritmos Con OpenMP

11.1. Poisson

Algoritmo Poisson En OpenMP

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <time.h>
# include <omp.h>
# define NX 161
# define NY 161

int main ( int argc, char *argv[] );
double r8mat_rms ( int nx, int ny, double a[NX][NY] );
void rhs ( int nx, int ny, double f[NX][NY] );
void sweep ( int nx, int ny, double dx, double dy, double f[NX][NY],
int itold, int itnew, double u[NX][NY], double unew[NX][NY] );
void timestamp ( void );
double u_exact ( double x, double y );
double uxxyy_exact ( double x, double y );

/******************************************************************/
int main ( int argc, char *argv[] )
/******************************************************************/
{
  int converged;
  double diff;
  double dx;
  double dy;
  double error;
  double f[NX][NY];
  int i;
  int id;
  int itnew;
  int itold;
  int j;
  int jt;
  int jt_max = 20;
  int nx = NX;
  int ny = NY;
  double tolerance = 0.000001;
  double u[NX][NY];
  double u_norm;
  double udiff[NX][NY];
  double uexact[NX][NY];
  double unew[NX][NY];
  double unew_norm;
  double wtime;
  double x;
  double y;
  int S = atoi(argv[1]);
  int nthreads;
  omp_set_num_threads(S);
```

```
  nthreads = omp_get_num_threads();

  dx = 1.0 / ( double ) ( nx - 1 );
  dy = 1.0 / ( double ) ( ny - 1 );

  timestamp ();
  printf ("\n");
  printf ("POISSON_OPENMP:\n");
  printf ("C version\n");
  printf ("A program for solving the Poisson equation.\n");
  printf ("\n");
  printf ("Use OpenMP for parallel execution.\n");
  printf ("The number of processors is %d\n", omp_get_num_procs ());

 # pragma omp parallel
{
  id = omp_get_thread_num ( );
  if ( id == 0 )
  {
printf( "The maximum number of threads is %d\n",omp_get_num_threads( ));
  }
}
  printf ( "\n" );
  printf ( "  -DEL^2 U = F(X,Y)\n" );
  printf ( "\n" );
  printf ( "  on the rectangle 0 <= X <= 1, 0 <= Y <= 1.\n" );
  printf ( "\n" );
  printf ( "  F(X,Y) = pi^2 * ( x^2 + y^2 ) * sin ( pi * x * y )\n" );
  printf ( "\n" );
  printf ( "  The number of interior X grid points is %d\n", nx );
  printf ( "  The number of interior Y grid points is %d\n", ny );
  printf ( "  The X grid spacing is %f\n", dx );
  printf ( "  The Y grid spacing is %f\n", dy );
/*
  Set the right hand side array F.
*/
  rhs ( nx, ny, f );
/*
  Set the initial solution estimate UNEW.
  We are "allowed" to pick up the boundary conditions exactly.
*/
  for ( j = 0; j < ny; j++ )
  {
    for ( i = 0; i < nx; i++ )
    {
      if ( i == 0 || i == nx - 1 || j == 0 || j == ny - 1 )
      {
        unew[i][j] = f[i][j];
      }
      else
      {
        unew[i][j] = 0.0;
      }
    }
  }
  unew_norm = r8mat_rms ( nx, ny, unew );
/*
  Set up the exact solution UEXACT.
*/
```

```
  for ( j = 0; j < ny; j++ )
  {
    y = ( double ) ( j ) / ( double ) ( ny - 1 );
    for ( i = 0; i < nx; i++ )
    {
      x = ( double ) ( i ) / ( double ) ( nx - 1 );
      uexact[i][j] = u_exact ( x, y );
    }
  }
  u_norm = r8mat_rms ( nx, ny, uexact );
  //printf ( "  RMS of exact solution = %g\n", u_norm );
/*
  Do the iteration.
*/
  converged = 0;

  printf ( "\n" );
  printf ( "  Step    ||Unew||     ||Unew-U||     ||Unew-Exact||\n" );
  printf ( "\n" );

  for ( j = 0; j < ny; j++ )
  {
    for ( i = 0; i < nx; i++ )
    {
      udiff[i][j] = unew[i][j] - uexact[i][j];
    }
  }
  error = r8mat_rms ( nx, ny, udiff );
  printf ( "  %4d  %14g                  %14g\n", 0, unew_norm, error );

  wtime = omp_get_wtime ( );

  itnew = 0;

  for ( ; ; )
  {
    itold = itnew;
    itnew = itold + 500;
/*
  SWEEP carries out 500 Jacobi steps in parallel before we come
  back to check for convergence.
*/
    sweep ( nx, ny, dx, dy, f, itold, itnew, u, unew );
/*
  Check for convergence.
*/
    u_norm = unew_norm;
    unew_norm = r8mat_rms ( nx, ny, unew );

    for ( j = 0; j < ny; j++ )
    {
      for ( i = 0; i < nx; i++ )
      {
        udiff[i][j] = unew[i][j] - u[i][j];
      }
    }
    diff = r8mat_rms ( nx, ny, udiff );

    for ( j = 0; j < ny; j++ )
```

```
    {
      for ( i = 0; i < nx; i++ )
      {
        udiff[i][j] = unew[i][j] - uexact[i][j];
      }
    }
    error = r8mat_rms ( nx, ny, udiff );
   printf ( "  %4d  %14g  %14g  %14g\n", itnew, unew_norm, diff, error );

    if ( diff <= tolerance )
    {
      converged = 1;
      break;
    }
  }
  if ( converged )
  {
    printf ( "  The iteration has converged.\n" );
  }
  else
  {
    printf ( "  The iteration has NOT converged.\n" );
  }

  wtime = omp_get_wtime ( ) - wtime;
  printf ( "\n" );
  printf ( "  Elapsed seconds = %g\n", wtime );
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "POISSON_OPENMP:\n" );
  printf ( "  Normal end of execution.\n" );
  printf ( "\n" );
  timestamp ( );

  return 0;
}
/*****************************************************************/
double r8mat_rms ( int nx, int ny, double a[NX][NY] )
/*****************************************************************/
{
  int i;
  int j;
  double v;
  v = 0.0;

  for ( j = 0; j < ny; j++ )
  {
    for ( i = 0; i < nx; i++ )
    {
      v = v + a[i][j] * a[i][j];
    }
  }
  v = sqrt ( v / ( double ) ( nx * ny )  );
  return v;
}
/*****************************************************************/
void rhs ( int nx, int ny, double f[NX][NY] )
```

```
/******************************************************************/
/*
  Purpose:
    RHS initializes the right hand side "vector".

  Discussion:
    It is convenient for us to set up RHS as a 2D array.  However, each
    entry of RHS is really the right hand side of a linear system of the
    form

      A * U = F

   In cases where U(I,J) is a boundary value, then the equation is simply
    U(I,J) = F(i,j)
    and F(I,J) holds the boundary data.

    Otherwise, the equation has the form
    (1/DX^2) * ( U(I+1,J)+U(I-1,J)+U(I,J-1)+U(I,J+1)-4*U(I,J) ) = F(I,J)

    where DX is the spacing and F(I,J) is the value at X(I), Y(J) of
      pi^2 * ( x^2 + y^2 ) * sin ( pi * x * y )

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    28 October 2011

  Author:
    John Burkardt

  Parameters:
    Input, int NX, NY, the X and Y grid dimensions.
    Output, double F[NX][NY], the initialized right hand side data.
*/
{
  double fnorm;
  int i;
  int j;
  double x;
  double y;

  for ( j = 0; j < ny; j++ )
  {
    y = ( double ) ( j ) / ( double ) ( ny - 1 );
    for ( i = 0; i < nx; i++ )
    {
      x = ( double ) ( i ) / ( double ) ( nx - 1 );
      if ( i == 0 || i == nx - 1 || j == 0 || j == ny - 1 )
      {
        f[i][j] = u_exact ( x, y );
      }
      else
      {
        f[i][j] = - uxxyy_exact ( x, y );
      }
    }
  }
```

```
  fnorm = r8mat_rms ( nx, ny, f );
  printf ( "  RMS of F = %g\n", fnorm );
  return;
}
/*****************************************************************/
void sweep ( int nx, int ny, double dx, double dy, double f[NX][NY],
  int itold, int itnew, double u[NX][NY], double unew[NX][NY] )
/*****************************************************************/
/*
  Purpose:
   SWEEP carries out one step of the Jacobi iteration.

  Discussion:
    Assuming DX = DY, we can approximate
      - ( d/dx d/dx + d/dy d/dy ) U(X,Y)
    by
     ( U(i-1,j) + U(i+1,j) + U(i,j-1) + U(i,j+1) - 4*U(i,j) ) / dx / dy

    The discretization employed below will not be correct in the general
    case where DX and DY are not equal.  It's only a little more
complicated
    to allow DX and DY to be different, but we're not going to worry about
    that right now.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    14 December 2011

  Author:
    John Burkardt

  Parameters:
    Input, int NX, NY, the X and Y grid dimensions.
    Input, double DX, DY, the spacing between grid points.
    Input, double F[NX][NY], the right hand side data.
    Input, int ITOLD, the iteration index on input.
    Input, int ITNEW, the desired iteration index
    on output.
    Input, double U[NX][NY], the solution estimate on
    iteration ITNEW-1.

    Input/output, double UNEW[NX][NY], on input, the solution
    estimate on iteration ITOLD.  On output, the solution estimate on
    iteration ITNEW.
*/
{
  int i;
  int it;
  int j;

# pragma omp parallel shared ( dx, dy, f, itnew, itold, nx, ny, u, unew )
private ( i, it, j )

  for ( it = itold + 1; it <= itnew; it++ )
  {
/*
  Save the current estimate.
```

```
*/
# pragma omp for
    for ( j = 0; j < ny; j++ )
    {
      for ( i = 0; i < nx; i++ )
      {
        u[i][j] = unew[i][j];
      }
    }
/*
  Compute a new estimate.
*/
# pragma omp for
    for ( j = 0; j < ny; j++ )
    {
      for ( i = 0; i < nx; i++ )
      {
        if ( i == 0 || j == 0 || i == nx - 1 || j == ny - 1 )
        {
          unew[i][j] = f[i][j];
        }
        else
       {
       unew[i][j] = 0.25 * (
     u[i-1][j] + u[i][j+1] + u[i][j-1] + u[i+1][j] + f[i][j] * dx * dy );
        }
      }
    }
  }
  return;
}
/******************************************************************/
void timestamp ( void )
/******************************************************************/
{
# define TIME_SIZE 40

  static char time_buffer[TIME_SIZE];
  const struct tm *tm;
  time_t now;

  now = time ( NULL );
  tm = localtime ( &now );

  strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );

 printf ( "%s\n", time_buffer );

  return;
# undef TIME_SIZE
}
/******************************************************************/
double u_exact ( double x, double y )
/******************************************************************/
{
  double pi = 3.141592653589793;
  double value;
  value = sin ( pi * x * y );
  return value;
```

```
}
/*****************************************************************/
double uxxyy_exact ( double x, double y )
/*****************************************************************/
{
  double pi = 3.141592653589793;
  double value;
  value = - pi * pi * ( x * x + y * y ) * sin ( pi * x * y );
  return value;
}
# undef NX
# undef NY
```

11.2. Ecuacion De Calor En 2D

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <omp.h>

int main ( int argc, char *argv[] );
/*****************************************************************/
int main ( int argc, char *argv[] )
/*****************************************************************/
/*
  Purpose:
    MAIN is the main program for ECUACION_CALOR_OPENMP.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    18 October 2011

  Author:
    Original C version by Michael Quinn.
    This C version by John Burkardt.

  Reference:
    Michael Quinn,
    Parallel Programming in C with MPI and OpenMP,
    McGraw-Hill, 2004,
    ISBN13: 978-0071232654,
    LC: QA76.73.C15.Q55.

  Local parameters:
    Local, double DIFF, the norm of the change in the solution from one
iteration
    to the next.
    Local, double MEAN, the average of the boundary values, used to
initialize
    the values of the solution in the interior.
```

```
    Local, double U[M][N], the solution at the previous iteration.
    Local, double W[M][N], the solution computed at the latest iteration.
*/
{
# define M 500
# define N 500

  double diff;
  double epsilon = 0.001;
  int i;
  int iterations;
  int iterations_print;
  int j;
  double mean;
  double my_diff;
  double u[M][N];
  double w[M][N];
  double wtime;
  int nthreads;
  int S = atoi(argv[1]);
  omp_set_num_threads(S);
  nthreads = omp_get_num_threads();

  printf ("\n" );
  printf ("HEATED_PLATE_OPENMP\n" );
  printf ("C/OpenMP version\n" );
  printf ("A program to solve for the steady state temperature
distribution\n" );
  printf ("over a rectangular plate.\n" );
  printf ("\n" );
  printf ("Spatial grid of %d by %d points.\n", M, N );
  printf ("The iteration will be repeated until the change is <= %e\n",
epsilon );
  printf ( "  Number of threads =%d\n", omp_get_max_threads ( ) );
  printf("Numero de threads en ejecucion = %d\n", nthreads);
/*
  Set the boundary values, which don't change.
*/
  mean = 0.0;

#pragma omp parallel shared ( w ) private ( i, j )
  {
#pragma omp for
    for ( i = 1; i < M - 1; i++ )
    {
      w[i][0] = 100.0;
    }
#pragma omp for
    for ( i = 1; i < M - 1; i++ )
    {
      w[i][N-1] = 100.0;
    }
#pragma omp for
    for ( j = 0; j < N; j++ )
    {
      w[M-1][j] = 100.0;
    }
#pragma omp for
    for ( j = 0; j < N; j++ )
```

```
    {
      w[0][j] = 0.0;
    }
/*
  Average the boundary values, to come up with a reasonable
  initial value for the interior.
*/
#pragma omp for reduction ( + : mean )
    for ( i = 1; i < M - 1; i++ )
    {
      mean = mean + w[i][0] + w[i][N-1];
    }
#pragma omp for reduction ( + : mean )
    for ( j = 0; j < N; j++ )
    {
      mean = mean + w[M-1][j] + w[0][j];
    }
  }
/*
  OpenMP note:
  You cannot normalize MEAN inside the parallel region.  It
  only gets its correct value once you leave the parallel region.
  So we interrupt the parallel region, set MEAN, and go back in.
*/
  mean = mean / ( double ) ( 2 * M + 2 * N - 4 );
  printf ( "\n" );
  printf ( "  MEAN = %f\n", mean );
/*
  Initialize the interior solution to the mean value.
*/
#pragma omp parallel shared ( mean, w ) private ( i, j )
  {
#pragma omp for
    for ( i = 1; i < M - 1; i++ )
    {
      for ( j = 1; j < N - 1; j++ )
      {
        w[i][j] = mean;
      }
    }
  }
/*
  iterate until the  new solution W differs from the old solution U
  by no more than EPSILON.
*/
  iterations = 0;
  iterations_print = 1;
  printf ( "\n" );
  printf ( " Iteration  Change\n" );
  printf ( "\n" );
  wtime = omp_get_wtime ( );
  diff = epsilon;

  while ( epsilon <= diff )
  {
# pragma omp parallel shared ( u, w ) private ( i, j )
    {
/*
  Save the old solution in U.
```

```
*/
# pragma omp for
      for ( i = 0; i < M; i++ )
      {
        for ( j = 0; j < N; j++ )
        {
          u[i][j] = w[i][j];
        }
      }
/*
  Determine the new estimate of the solution at the interior points.
  The new solution W is the average of north, south, east and west neighbors.
*/
# pragma omp for
      for ( i = 1; i < M - 1; i++ )
      {
        for ( j = 1; j < N - 1; j++ )
        {
          w[i][j] = ( u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] ) / 4.0;
        }
      }
    }
/*
  C and C++ cannot compute a maximum as a reduction operation.
  Therefore, we define a private variable MY_DIFF for each thread.
  Once they have all computed their values, we use a CRITICAL section
  to update DIFF.
*/
    diff = 0.0;
# pragma omp parallel shared ( diff, u, w ) private ( i, j, my_diff )
    {
      my_diff = 0.0;
# pragma omp for
      for ( i = 1; i < M - 1; i++ )
      {
        for ( j = 1; j < N - 1; j++ )
        {
          if ( my_diff < fabs ( w[i][j] - u[i][j] ) )
          {
            my_diff = fabs ( w[i][j] - u[i][j] );
          }
        }
      }
# pragma omp critical
      {
        if ( diff < my_diff )
        {
          diff = my_diff;
        }
      }
    }

    iterations++;
    if ( iterations == iterations_print )
    {
      printf ( "  %8d  %f\n", iterations, diff );
      iterations_print = 2 * iterations_print;
    }
  }
```

```
  wtime = omp_get_wtime ( ) - wtime;

  printf ( "\n" );
  printf ( "  %8d  %f\n", iterations, diff );
  printf ( "\n" );
  printf ( "  Error tolerance achieved.\n" );
  printf ( "  Wallclock time = %f\n", wtime );
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "HEATED_PLATE_OPENMP:\n" );
  printf ( "  Normal end of execution.\n" );

  return 0;

# undef M
# undef N
}
```

11.3. Ziggurat

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <time.h>
# include <stdint.h>
# include <omp.h>

int main ( int argc, char *argv[] );
void test01 ( );
void test02 ( );
void test03 ( );
void test04 ( );
float r4_exp ( uint32_t *jsr, uint32_t ke[256], float fe[256], float we[256]
);
void r4_exp_setup ( uint32_t ke[256], float fe[256], float we[256] );
float r4_nor ( uint32_t *jsr, uint32_t kn[128], float fn[128], float wn[128]
);
void r4_nor_setup ( uint32_t kn[128], float fn[128], float wn[128] );
float r4_uni ( uint32_t *jsr );
uint32_t shr3_seeded ( uint32_t *jsr );
void timestamp ( );

/******************************************************************/
int main ( int argc, char *argv[] )
/******************************************************************/
/*
  Purpose:
    MAIN is the main program for ZIGGURAT_OPENMP.

  Licensing:
    This code is distributed under the GNU LGPL license.
```

```
  Modified:
    04 October 2013

  Author:
    John Burkardt
*/
{

  int S = atoi(argv[1]);
  int nthreads;
  omp_set_num_threads(S);
  nthreads = omp_get_num_threads();
  double wtime;
   timestamp ( );
  printf ( "\n" );
  printf ( "ZIGGURAT_OPENMP:\n" );
  printf ( "  C version\n" );
  printf ( "\n" );
  printf ( "  Number of processors available = %d\n", omp_get_num_procs ( )
);
  printf ( "  Number of threads =              %d\n", omp_get_max_threads ( )
);
  wtime = omp_get_wtime ( );
  test01 ( );
  test02 ( );
  test03 ( );
  test04 ( );
  wtime = omp_get_wtime ( ) - wtime;
/*
 Terminate.
*/
  printf ( "\n" );
  printf ( "ZIGGURAT_OPENMP:\n" );
  printf ( "  Normal end of execution.\n" );
  printf ( "  Wallclock time = %f\n", wtime );
  printf ( "\n" );
  timestamp ( );
  return 0;
}
/******************************************************************/
void test01 ( )
/******************************************************************/
{
  uint32_t jsr;
  uint32_t jsr_value;
  double mega_rate_par;
  double mega_rate_seq;
  int r;
  int r_num = 1000;
  int *result_par;
  int *result_seq;
  int s;
  int s_num = 10000;
  uint32_t *seed;
  int thread;
  int thread_num;
  double wtime_par;
  double wtime_seq;
```

```
  printf ( "\n" );
  printf ( "TEST01\n" );
  printf ( "  SHR3_SEEDED computes random integers.\n" );
  printf ( "  Since the output is completely determined\n" );
  printf ( "  by the input value of SEED, we can run in\n" );
  printf ( "  parallel as long as we make an array of seeds.\n" );
/*
  Set up the SEED array, which will be used for both sequential and
  parallel computations.
*/
# pragma omp parallel
{
# pragma omp master
  {
    thread_num = omp_get_num_threads ( );

    printf ( "\n" );
    printf ( "  The number of threads is %d\n", thread_num );
  }
}
  seed = ( uint32_t * ) malloc ( thread_num * sizeof ( uint32_t ) );
  result_seq = ( int * ) malloc ( thread_num * sizeof ( int ) );
  result_par = ( int * ) malloc ( thread_num * sizeof ( int ) );
/*
  Sequential execution.
  The sequential execution will only match the parallel execution if we can
  guarantee that the parallel threads are scheduled to execute the R loop
  consecutively.
*/
  jsr = 123456789;

  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_seq = omp_get_wtime ( );
  for ( r = 0; r < r_num; r++ )
  {
    thread = ( r % thread_num );
    jsr = seed[thread];

    for ( s = 0; s < s_num; s++ )
    {
      jsr_value = shr3_seeded ( &jsr );
    }
    result_seq[thread] = jsr_value;
    seed[thread] = jsr;
  }

  wtime_seq = omp_get_wtime ( ) - wtime_seq;
 mega_rate_seq = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_seq  /
1000000.0;
/*
  Parallel.
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
```

```
  }
  wtime_par = omp_get_wtime ( );

# pragma omp parallel \ shared ( result_par, seed ) \  private ( jsr,
jsr_value, r, s, thread )
  {
# pragma omp for schedule ( static, 1 )
    for ( r = 0; r < r_num; r++ )
    {
      thread = omp_get_thread_num ( );

      jsr = seed[thread];
      for ( s = 0; s < s_num; s++ )
      {
        jsr_value = shr3_seeded ( &jsr );
      }
      result_par[thread] = jsr_value;
      seed[thread] = jsr;
    }
  }
  wtime_par = omp_get_wtime ( ) - wtime_par;
  mega_rate_par = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_par /
1000000.0;
/*
  Report.
*/
  printf ( "\n" );
  printf ( "  Correctness check:\n" );
  printf ( "\n" );
  printf ( "  Computing values sequentially should reach the\n" );
  printf ( "  same result as doing it in parallel:\n" );
  printf ( "\n" );
  printf ( "    THREAD    Sequential      Parallel    Difference\n" );
  printf ( "\n" );
  for ( thread = 0; thread < thread_num; thread++ )
  {
printf ( "  %8d  %12d  %12d  %12d\n",  thread, result_seq[thread],
result_par[thread], result_seq[thread] - result_par[thread] );
  }
printf ( "\n" );
printf ( "  Efficiency check:\n" );
printf ( "\n" );
printf ( "  Computing values in parallel should be faster:\n" );
printf ( "\n" );
printf ( "              Sequential      Parallel\n" );
printf ( "\n" );
printf ( "      TIME:  %14f  %14f\n", wtime_seq, wtime_par  );
printf ( "      RATE:  %14f  %14f\n", mega_rate_seq, mega_rate_par );
/*
  Free memory.
*/
  free ( result_par );
  free ( result_seq );
  free ( seed );
  return;
}
/*****************************************************************/
void test02 ( )
/*****************************************************************/
```

```
/*
  Purpose:
    TEST02 tests R4_UNI.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    04 October 2013

  Author:
    John Burkardt
*/
{
  uint32_t jsr;
  uint32_t jsr_value;
  double mega_rate_par;
  double mega_rate_seq;
  int r;
  int r_num = 1000;
  float r4_value;
  float *result_par;
  float *result_seq;
  int s;
  int s_num = 10000;
  uint32_t *seed;
  int thread;
  int thread_num;
  double wtime_par;
  double wtime_seq;

  printf ( "\n" );
  printf ( "TEST02\n" );
  printf ( "  R4_UNI computes uniformly random single precision real
values.\n" );
  printf ( "  Since the output is completely determined\n" );
  printf ( "  by the input value of SEED, we can run in\n" );
  printf ( "  parallel as long as we make an array of seeds.\n" );
/*
  Set up the SEED array, which will be used for both sequential and parallel
computations.
*/
# pragma omp parallel
{
# pragma omp master
  {
    thread_num = omp_get_num_threads ( );
    printf ( "\n" );
    printf ( "  The number of threads is %d\n", thread_num );
  }
}
  seed = ( uint32_t * ) malloc ( thread_num * sizeof ( uint32_t ) );
  result_seq = ( float * ) malloc ( thread_num * sizeof ( float ) );
  result_par = ( float * ) malloc ( thread_num * sizeof ( float ) );
/*
  Sequential execution.
  The sequential execution will only match the parallel execution if we can
  guarantee that the parallel threads are scheduled to execute the R loop
  consecutively.
```

```
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_seq = omp_get_wtime ( );

  for ( r = 0; r < r_num; r++ )
  {
    thread = ( r % thread_num );
    jsr = seed[thread];

    for ( s = 0; s < s_num; s++ )
    {
      r4_value = r4_uni ( &jsr );
    }
    result_seq[thread] = r4_value;
    seed[thread] = jsr;
  }
  wtime_seq = omp_get_wtime ( ) - wtime_seq;
  mega_rate_seq = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_seq /
1000000.0;
/*
  Parallel.
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_par = omp_get_wtime ( );

# pragma omp parallel \ shared ( result_par, seed ) \ private ( jsr, r,
r4_value, s, thread )
  {

# pragma omp for schedule ( static, 1 )
    for ( r = 0; r < r_num; r++ )
    {
      thread = omp_get_thread_num ( );
      jsr = seed[thread];

      for ( s = 0; s < s_num; s++ )
      {
        r4_value = r4_uni ( &jsr );
      }
      result_par[thread] = r4_value;
      seed[thread] = jsr;
    }
  }

  wtime_par = omp_get_wtime ( ) - wtime_par;
  mega_rate_par = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_par /
1000000.0;
/*
  Report.
*/
  printf ( "\n" );
```

```
  printf ( "  Correctness check:\n" );
  printf ( "\n" );
  printf ( "  Computing values sequentially should reach the\n" );
  printf ( "  same result as doing it in parallel:\n" );
  printf ( "\n" );
  printf ( "    THREAD Sequential Parallel Difference\n" );
  printf ( "\n" );
  for ( thread = 0; thread < thread_num; thread++ )
  {
  printf ( "  %8d  %14f  %14f  %14f\n", thread,
result_seq[thread],result_par[thread], result_seq[thread] -
result_par[thread] );
  }

 printf ( "\n" );
 printf ( "  Efficiency check:\n" );
 printf ( "\n" );
 printf ( "  Computing values in parallel should be faster:'\n" );
 printf ( "\n" );
 printf ( "              Sequential      Parallel\n" );
 printf ( "\n" );
 printf ( "      TIME:  %14f  %14f\n", wtime_seq, wtime_par  );
 printf ( "      RATE:  %14f  %14f\n", mega_rate_seq, mega_rate_par );
/*
  Free memory.
*/
  free ( result_par );
  free ( result_seq );
  free ( seed );
  return;
}
/******************************************************************/
void test03 ( )
/******************************************************************/
/*
  Purpose:
    TEST03 tests R4_NOR.

  Discussion:
    The arrays FN, KN and WN, once set up by R4_NOR_SETUP, are "read only"
when
    accessed by R4_NOR.  So we only need to have one copy of these arrays,
    and they can be shared.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    04 October 2013

  Author:
    John Burkardt
*/
{
  float fn[128];
  uint32_t jsr;
  uint32_t jsr_value;
  uint32_t kn[128];
  double mega_rate_par;
```

```
  double mega_rate_seq;
  int r;
  int r_num = 1000;
  float r4_value;
  float *result_par;
  float *result_seq;
  int s;
  int s_num = 10000;
  uint32_t *seed;
  int thread;
  int thread_num;
  float wn[128];
  double wtime_par;
  double wtime_seq;

  printf ( "\n" );
  printf ( "TEST03\n" );
  printf ( "  R4_NOR computes normal random single precision real values.\n"
);
  printf ( "  Since the output is completely determined\n" );
  printf ( "  by the input value of SEED and the tables, we can run in\n" );
  printf ( "  parallel as long as we make an array of seeds and share the
tables.\n" );
/*
  Set up the SEED array and the tables, which will be used for both
sequential
  and parallel computations.
*/
# pragma omp parallel
{
# pragma omp master
  {
    thread_num = omp_get_num_threads ( );
  printf ( "\n" );
  printf ( "  The number of threads is %d\n", thread_num );
  }
}
  seed = ( uint32_t * ) malloc ( thread_num * sizeof ( uint32_t ) );
  result_seq = ( float * ) malloc ( thread_num * sizeof ( float ) );
  result_par = ( float * ) malloc ( thread_num * sizeof ( float ) );

  r4_nor_setup ( kn, fn, wn );
/*
  Sequential execution.
  The sequential execution will only match the parallel execution if we can
  guarantee that the parallel threads are scheduled to execute the R loop
  consecutively.
*/
  jsr = 123456789;

  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_seq = omp_get_wtime ( );

  for ( r = 0; r < r_num; r++ )
  {
    thread = ( r % thread_num );
```

```
    jsr = seed[thread];

    for ( s = 0; s < s_num; s++ )
    {
      r4_value = r4_nor ( &jsr, kn, fn, wn );
    }
    result_seq[thread] = r4_value;
    seed[thread] = jsr;
  }
  wtime_seq = omp_get_wtime ( ) - wtime_seq;
  mega_rate_seq = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_seq /
1000000.0;
/*
  Parallel.
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_par = omp_get_wtime ( );

# pragma omp parallel \ shared ( result_par, seed ) \ private ( jsr, r,
r4_value, s, thread )
  {
# pragma omp for schedule ( static, 1 )

    for ( r = 0; r < r_num; r++ )
    {
      thread = omp_get_thread_num ( );
      jsr = seed[thread];

      for ( s = 0; s < s_num; s++ )
      {
        r4_value = r4_nor ( &jsr, kn, fn, wn );
      }
      result_par[thread] = r4_value;
      seed[thread] = jsr;
    }
  }
  wtime_par = omp_get_wtime ( ) - wtime_par;

  mega_rate_par = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_par /
1000000.0;
/*
  Report.
*/
  printf ( "\n" );
  printf ( "  Correctness check:\n" );
  printf ( "\n" );
  printf ( "  Computing values sequentially should reach the\n" );
  printf ( "  same result as doing it in parallel:\n" );
  printf ( "\n" );
  printf ( "    THREAD    Sequential        Parallel      Difference\n" );
  printf ( "\n" );

  for ( thread = 0; thread < thread_num; thread++ )
  {
```

```
   printf ( "  %8d  %14f  %14f  %14f\n", thread,
result_seq[thread],result_par[thread], result_seq[thread] -
result_par[thread] );
  }

  printf ( "\n" );
  printf ( "  Efficiency check:\n" );
  printf ( "\n" );
  printf ( "  Computing values in parallel should be faster:\n" );
  wprintf ( "\n" );
  printf ( "              Sequential      Parallel\n" );
  printf ( "\n" );
 printf ( "      TIME:  %14f  %14f\n", wtime_seq, wtime_par  );
 printf ( "      RATE:  %14f  %14f\n", mega_rate_seq, mega_rate_par );
/*
  Free memory.
*/
  free ( result_par );
  free ( result_seq );
  free ( seed );
  return;
}
/******************************************************************/
void test04 ( )
/******************************************************************/
/*
  Purpose:
    TEST04 tests R4_EXP.

  Discussion:
    The arrays FE, KE and WE, once set up by R4_EXP_SETUP, are "read only"
when
    accessed by R4_EXP.  So we only need to have one copy of these arrays,
    and they can be shared.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    19 October 2013

  Author:
    John Burkardt
*/
{
  float fe[256];
  uint32_t jsr;
  uint32_t jsr_value;
  uint32_t ke[256];
  double mega_rate_par;
  double mega_rate_seq;
  int r;
  int r_num = 1000;
  float r4_value;
  float *result_par;
  float *result_seq;
  int s;
  int s_num = 10000;
  uint32_t *seed;
```

```
  int thread;
  int thread_num;
  float we[256];
  double wtime_par;
  double wtime_seq;

  printf ( "\n" );
 printf ( "TEST04\n" );
 printf ( "  R4_EXP computes exponential random single precision real
values.\n" );
 printf ( "  Since the output is completely determined\n" );
 printf ( "  by the input value of SEED and the tables, we can run in\n" );
 printf ( "  parallel as long as we make an array of seeds and share the
tables.\n" );
/*
  Set up the SEED array and the tables, which will be used for both
sequential
  and parallel computations.
*/
# pragma omp parallel
{
# pragma omp master
  {
    thread_num = omp_get_num_threads ( );
    printf ( "\n" );
    printf ( "  The number of threads is %d\n", thread_num );
  }
}
 seed = ( uint32_t * ) malloc ( thread_num * sizeof ( uint32_t ) );
 result_seq = ( float * ) malloc ( thread_num * sizeof ( float ) );
 result_par = ( float * ) malloc ( thread_num * sizeof ( float ) );

  r4_exp_setup ( ke, fe, we );
/*
  Sequential execution.
  The sequential execution will only match the parallel execution if we can
  guarantee that the parallel threads are scheduled to execute the R loop
  consecutively.
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_seq = omp_get_wtime ( );
  for ( r = 0; r < r_num; r++ )
  {
    thread = ( r % thread_num );
    jsr = seed[thread];
    for ( s = 0; s < s_num; s++ )
    {
      r4_value = r4_exp ( &jsr, ke, fe, we );
    }
    result_seq[thread] = r4_value;
    seed[thread] = jsr;
  }
  wtime_seq = omp_get_wtime ( ) - wtime_seq;
  mega_rate_seq = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_seq
    / 1000000.0;
```

```
/*
  Parallel.
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_par = omp_get_wtime ( );

# pragma omp parallel \ shared ( result_par, seed ) \  private ( jsr, r,
r4_value, s, thread )
  {

# pragma omp for schedule ( static, 1 )
    for ( r = 0; r < r_num; r++ )
    {
      thread = omp_get_thread_num ( );
      jsr = seed[thread];
      for ( s = 0; s < s_num; s++ )
      {
        r4_value = r4_exp ( &jsr, ke, fe, we );
      }
      result_par[thread] = r4_value;
      seed[thread] = jsr;
    }
  }
  wtime_par = omp_get_wtime ( ) - wtime_par;

  mega_rate_par = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_par
    / 1000000.0;
/*
  Report.
*/
 printf ( "\n" );
 printf ( "  Correctness check:\n" );
 printf ( "\n" );
 printf ( "  Computing values sequentially should reach the\n" );
 printf ( "  same result as doing it in parallel:\n" );
 printf ( "\n" );
 printf ( "    THREAD    Sequential        Parallel      Difference\n" );
 printf ( "\n" );

  for ( thread = 0; thread < thread_num; thread++ )
  {
printf ( "  %8d  %14f  %14f  %14f\n", thread, result_seq[thread],
result_par[thread], result_seq[thread] - result_par[thread] );
  }
printf ( "\n" );
printf ( "  Efficiency check:\n" );
printf ( "\n" );
printf ( "  Computing values in parallel should be faster:\n" );
printf ( "\n" );
printf ( "              Sequential      Parallel\n" );
printf ( "\n" );
printf ( "      TIME:  %14f  %14f\n", wtime_seq, wtime_par  );
printf ( "      RATE:  %14f  %14f\n", mega_rate_seq, mega_rate_par );
/*
  Free memory.
```

```
*/
  free ( result_par );
  free ( result_seq );
  free ( seed );
  return;
}
/******************************************************************************
**/
float r4_exp ( uint32_t *jsr, uint32_t ke[256], float fe[256], float we[256]
)
/*****************************************************************/
/*
  Purpose:
    R4_EXP returns an exponentially distributed single precision real value.

  Discussion:
    The underlying algorithm is the ziggurat method.
    Before the first call to this function, the user must call R4_EXP_SETUP
    to determine the values of KE, FE and WE.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    15 October 2013

  Author:
    John Burkardt

  Reference:
    George Marsaglia, Wai Wan Tsang,
    The Ziggurat Method for Generating Random Variables,
    Journal of Statistical Software,
    Volume 5, Number 8, October 2000, seven pages.

  Parameters:

    Input/output, uint32_t *JSR, the seed.
    Input, uint32_t KE[256], data computed by R4_EXP_SETUP.
    Input, float FE[256], WE[256], data computed by R4_EXP_SETUP.
    Output, float R4_EXP, an exponentially distributed random value.
*/
{
  uint32_t iz;
  uint32_t jz;
  float value;
  float x;
  jz = shr3_seeded ( jsr );
  iz = ( jz & 255 );

  if ( jz < ke[iz] )
  {
    value = ( float ) ( jz ) * we[iz];
  }
  else
  {
    for ( ; ; )
    {
      if ( iz == 0 )
```

```
      {
        value = 7.69711 - log ( r4_uni ( jsr ) );
        break;
      }
      x = ( float ) ( jz ) * we[iz];

      if ( fe[iz] + r4_uni ( jsr ) * ( fe[iz-1] - fe[iz] ) < exp ( - x ) )
      {
        value = x;
        break;
      }
      jz = shr3_seeded ( jsr );
      iz = ( jz & 255 );

      if ( jz < ke[iz] )
      {
        value = ( float ) ( jz ) * we[iz];
        break;
      }
    }
  }
  return value;
}
/****************************************************************/
void r4_exp_setup ( uint32_t ke[256], float fe[256], float we[256] )
/****************************************************************/
/*
  Purpose:
    R4_EXP_SETUP sets data needed by R4_EXP.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    14 October 2013

  Author:
    John Burkardt

  Reference:
    George Marsaglia, Wai Wan Tsang,
    The Ziggurat Method for Generating Random Variables,
    Journal of Statistical Software,
    Volume 5, Number 8, October 2000, seven pages.

  Parameters:
    Output, uint32_t KE[256], data needed by R4_EXP.
    Output, float FE[256], WE[256], data needed by R4_EXP.
*/
{
  double de = 7.697117470131487;
  int i;
  const double m2 = 2147483648.0;
  double q;
  double te = 7.697117470131487;
  const double ve = 3.949659822581572E-03;
  q = ve / exp ( - de );

  ke[0] = ( uint32_t ) ( ( de / q ) * m2 );
```

```
  ke[1] = 0;
  we[0] = ( float ) ( q / m2 );
  we[255] = ( float ) ( de / m2 );
  fe[0] = 1.0;
  fe[255] = ( float ) ( exp ( - de ) );
  for ( i = 254; 1 <= i; i-- )
  {
    de = - log ( ve / de + exp ( - de ) );
    ke[i+1] = ( uint32_t ) ( ( de / te ) * m2 );
    te = de;
    fe[i] = ( float ) ( exp ( - de ) );
    we[i] = ( float ) ( de / m2 );
  }
  return;
}
/**********************************************************/

float r4_nor ( uint32_t *jsr, uint32_t kn[128], float fn[128], float wn[128]
)
/**********************************************************/
/*
  Purpose:
    R4_NOR returns a normally distributed single precision real value.

  Discussion:
    The value returned is generated from a distribution with mean 0 and
    variance 1.
    The underlying algorithm is the ziggurat method.
    Before the first call to this function, the user must call R4_NOR_SETUP
    to determine the values of KN, FN and WN.
    Thanks to Chad Wagner, 21 July 2014, for noticing a bug of the form
      if ( x * x <= y * y );   <-- Stray semicolon!
      {
        break;
      }

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 July 2014

  Author:
    John Burkardt

  Reference:
    George Marsaglia, Wai Wan Tsang,
    The Ziggurat Method for Generating Random Variables,
    Journal of Statistical Software,
    Volume 5, Number 8, October 2000, seven pages.

  Parameters:
    Input/output, uint32_t *JSR, the seed.
    Input, uint32_t KN[128], data computed by R4_NOR_SETUP.
    Input, float FN[128], WN[128], data computed by R4_NOR_SETUP.
    Output, float R4_NOR, a normally distributed random value.
*/
{
  int hz;
```

```
uint32_t iz;
const float r = 3.442620;
float value;
float x;
float y;

hz = ( int ) shr3_seeded ( jsr );
iz = ( hz & 127 );

if ( fabs ( hz ) < kn[iz] )
{
  value = ( float ) ( hz ) * wn[iz];
}
else
{
  for ( ; ; )
  {
    if ( iz == 0 )
    {
      for ( ; ; )
      {
        x = - 0.2904764 * log ( r4_uni ( jsr ) );
        y = - log ( r4_uni ( jsr ) );
        if ( x * x <= y + y )
        {
          break;
        }
      }

      if ( hz <= 0 )
      {
        value = - r - x;
      }
      else
      {
        value = + r + x;
      }
      break;
    }
    x = ( float ) ( hz ) * wn[iz];

    if ( fn[iz] + r4_uni ( jsr ) * ( fn[iz-1] - fn[iz] )
      < exp ( - 0.5 * x * x ) )
    {
      value = x;
      break;
    }

    hz = ( int ) shr3_seeded ( jsr );
    iz = ( hz & 127 );
    if ( fabs ( hz ) < kn[iz] )
    {
      value = ( float ) ( hz ) * wn[iz];
      break;
    }
  }
}

return value;
```

```
}
/****************************************************************/
void r4_nor_setup ( uint32_t kn[128], float fn[128], float wn[128] )
/****************************************************************/
/*
  Purpose:
    R4_NOR_SETUP sets data needed by R4_NOR.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    14 October 2013

  Author:
    John Burkardt

  Reference:
    George Marsaglia, Wai Wan Tsang,
    The Ziggurat Method for Generating Random Variables,
    Journal of Statistical Software,
    Volume 5, Number 8, October 2000, seven pages.

  Parameters:
    Output, uint32_t KN[128], data needed by R4_NOR.
    Output, float FN[128], WN[128], data needed by R4_NOR.
*/
{
  double dn = 3.442619855899;
  int i;
  const double m1 = 2147483648.0;
  double q;
  double tn = 3.442619855899;
  const double vn = 9.91256303526217E-03;
  q = vn / exp ( - 0.5 * dn * dn );
  kn[0] = ( uint32_t ) ( ( dn / q ) * m1 );
  kn[1] = 0;
  wn[0] = ( float ) ( q / m1 );
  wn[127] = ( float ) ( dn / m1 );
  fn[0] = 1.0;
  fn[127] = ( float ) ( exp ( - 0.5 * dn * dn ) );
  for ( i = 126; 1 <= i; i-- )
  {
    dn = sqrt ( - 2.0 * log ( vn / dn + exp ( - 0.5 * dn * dn ) ) );
    kn[i+1] = ( uint32_t ) ( ( dn / tn ) * m1 );
    tn = dn;
    fn[i] = ( float ) ( exp ( - 0.5 * dn * dn ) );
    wn[i] = ( float ) ( dn / m1 );
  }
  return;
}
/****************************************************************/
float r4_uni ( uint32_t *jsr )
/****************************************************************/
/*
  Purpose:
    R4_UNI returns a uniformly distributed real value.

  Licensing:
```

```
    This code is distributed under the GNU LGPL license.

  Modified:
    04 October 2013

  Author:
    John Burkardt

  Reference:
    George Marsaglia, Wai Wan Tsang,
    The Ziggurat Method for Generating Random Variables,
    Journal of Statistical Software,
    Volume 5, Number 8, October 2000, seven pages.

  Parameters:
    Input/output, uint32_t *JSR, the seed.
   Output, float R4_UNI, a uniformly distributed random value in
    the range [0,1].
*/
{
  uint32_t jsr_input;
  float value;
  jsr_input = *jsr;
  *jsr = ( *jsr ^ ( *jsr <<   13 ) );
  *jsr = ( *jsr ^ ( *jsr >>   17 ) );
  *jsr = ( *jsr ^ ( *jsr <<    5 ) );

  value = fmod ( 0.5 + ( float ) ( jsr_input + *jsr ) / 65536.0 / 65536.0,
1.0 );

  return value;
}
/******************************************************************/
uint32_t shr3_seeded ( uint32_t *jsr )
/******************************************************************/
/*
  Purpose:
    SHR3_SEEDED evaluates the SHR3 generator for integers.

  Discussion:
    Thanks to Dirk Eddelbuettel for pointing out that this code needed to
    use the uint32_t data type in order to execute properly in 64 bit mode,
    03 October 2013.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    04 October 2013

  Author:
    John Burkardt

  Reference:
    George Marsaglia, Wai Wan Tsang,
    The Ziggurat Method for Generating Random Variables,
    Journal of Statistical Software,
    Volume 5, Number 8, October 2000, seven pages.
```

```
  Parameters:
    Input/output, uint32_t *JSR, the seed, which is updated
    on each call.
    Output, uint32_t SHR3_SEEDED, the new value.
*/
{
  uint32_t value;
  value = *jsr;
  *jsr = ( *jsr ^ ( *jsr <<   13 ) );
  *jsr = ( *jsr ^ ( *jsr >>   17 ) );
  *jsr = ( *jsr ^ ( *jsr <<    5 ) );
 value = value + *jsr;
  return value;
}
/******************************************************************/
void timestamp ( void )
/******************************************************************/
/*
  Purpose:
    TIMESTAMP prints the current YMDHMS date as a time stamp.

  Example:
    31 May 2001 09:45:54 AM

  Licensing:
    This code is distributed under the GNU LGPL license.

 Modified:
    24 September 2003

  Author:
    John Burkardt

  Parameters:
    None
*/
{
# define TIME_SIZE 40
  static char time_buffer[TIME_SIZE];
  const struct tm *tm;
  size_t len;
  time_t now;
  now = time ( NULL );
  tm = localtime ( &now );

  len = strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );
 printf ( "%s\n", time_buffer );
  return;
# undef TIME_SIZE
}
```

11.4. Satisfy

```
# include <stdlib.h>
```

```
# include <stdio.h>
# include <time.h>
# include <omp.h>

int main ( int argc, char *argv[] );
int circuit_value ( int n, int bvec[] );
void i4_to_bvec ( int i4, int n, int bvec[] );
void timestamp ( void );

/******************************************************************************/
int main ( int argc, char *argv[] )
/******************************************************************************/
/*
  Purpose:
    MAIN is the main program for SATISFY_OPENMP.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    24 March 2009

  Author:
    John Burkardt

  Reference:
    Michael Quinn,
    Parallel Programming in C with MPI and OpenMP,
    McGraw-Hill, 2004,
    ISBN13: 978-0071232654,
    LC: QA76.73.C15.Q55.
*/
{
# define N 23
  int bvec[N];
  int i;
  int id;
  int ihi;
  int ihi2;
  int ilo;
  int ilo2;
  int j;
  int n = N;
  int proc_num;
  int solution_num;
  int solution_num_local;
  int thread_num;
  int value;
  double wtime;
  int S = atoi(argv[1]);
```

```
  int thread_num;
  omp_set_num_threads(S);
  thread_num = omp_get_num_threads();

  printf ( "\n" );
  timestamp ( );
  printf ( "\n" );
  printf ( "SATISFY_OPENMP\n" );
  printf ( "  C + OpenMP version\n" );
  printf ( "  We have a logical function of N logical arguments.\n" );
  printf ( "  We do an exhaustive search of all 2^N possibilities,\n" );
 printf ( "  seeking those inputs that make the function TRUE.\n" );
 printf ( "\n" );

/*
  Compute the number of binary vectors to check.
*/
  ilo = 0;
  ihi = 1;
  for ( i = 1; i <= n; i++ )
  {
    ihi = ihi * 2;
  }
  printf ( "\n" );
  printf ( "  The number of logical variables is N = %d\n", n );
  printf ( "  The number of input vectors to check is %d\n", ihi );
  printf ( "\n" );
  printf ( "   # Processor Index -------Input Values----------\n" );
  printf ( "\n" );
/*
  Processor ID takes the interval ILO2 <= I < IHI2.
  Using the formulas below yields a set of nonintersecting intervals
  which cover the original interval [ILO,IHI).
*/
  thread_num = omp_get_max_threads ( );
  solution_num = 0;
  wtime = omp_get_wtime ( );

# pragma omp parallel \ shared ( ihi, ilo, n, thread_num ) \
  private ( bvec, i, id, ihi2, ilo2, j, solution_num_local, value ) \
  reduction ( + : solution_num )
  {
    id = omp_get_thread_num ( );
    ilo2 = ( ( thread_num - id     ) * ilo
          + (              id     ) * ihi )
          / ( thread_num          );
    ihi2 = ( ( thread_num - id - 1 ) * ilo
          + (              id + 1 ) * ihi )
          / ( thread_num          );
 printf ( "\n" );
```

```
printf ( "  Processor %8d iterates from %8d <= I < %8d.\n", id, ilo2, ihi2 );
printf ( "\n" );
/*
  Check every possible input vector.
*/
    solution_num_local = 0;
    for ( i = ilo2; i < ihi2; i++ )
    {
      i4_to_bvec ( i, n, bvec );
      value = circuit_value ( n, bvec );
      if ( value == 1 )
      {
        solution_num_local = solution_num_local + 1;
        printf ( "  %2d  %8d  %10d:  ", solution_num_local, id, i );
        for ( j = 0; j < n; j++ )
        {
          printf ( " %d", bvec[j] );
        }
        printf ( "\n" );
      }
    }
    solution_num = solution_num + solution_num_local;
  }
  wtime = omp_get_wtime ( ) - wtime;
  printf ( "\n" );
  printf ( "  Number of solutions found was %d\n", solution_num );
  printf ( "  Elapsed wall clock time (seconds) %f\n", wtime );
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "SATISFY_OPENMP\n" );
  printf ( "  Normal end of execution.\n" );
  printf ( "\n" );
  timestamp ( );
  return 0;
# undef N
}
/******************************************************************************/
int circuit_value ( int n, int bvec[] )
/******************************************************************************/
/*
  Purpose:
    CIRCUIT_VALUE returns the value of a circuit for a given input set.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    20 March 2009
```

```
  Author:
    John Burkardt

  Reference:
    Michael Quinn,
    Parallel Programming in C with MPI and OpenMP,
    McGraw-Hill, 2004,
    ISBN13: 978-0071232654,
    LC: QA76.73.C15.Q55.

  Parameters:
    Input, int N, the length of the input vector.
    Input, int BVEC[N], the binary inputs.

    Output, int CIRCUIT_VALUE, the output of the circuit.
*/
{
  int value;

  value =
       (  bvec[0]  ||  bvec[1]  )
    && ( !bvec[1]  || !bvec[3]  )
    && (  bvec[2]  ||  bvec[3]  )
    && ( !bvec[3]  || !bvec[4]  )
    && (  bvec[4]  || !bvec[5]  )
    && (  bvec[5]  || !bvec[6]  )
    && (  bvec[5]  ||  bvec[6]  )
    && (  bvec[6]  || !bvec[15] )
    && (  bvec[7]  || !bvec[8]  )
    && ( !bvec[7]  || !bvec[13] )
    && (  bvec[8]  ||  bvec[9]  )
    && (  bvec[8]  || !bvec[9]  )
    && ( !bvec[9]  || !bvec[10] )
    && (  bvec[9]  ||  bvec[11] )
    && (  bvec[10] ||  bvec[11] )
    && (  bvec[12] ||  bvec[13] )
    && (  bvec[13] || !bvec[14] )
    && (  bvec[14] ||  bvec[15] )
    && (  bvec[14] ||  bvec[16] )
    && (  bvec[17] ||  bvec[1]  )
    && (  bvec[18] || !bvec[0]  )
    && (  bvec[19] ||  bvec[1]  )
    && (  bvec[19] || !bvec[18] )
    && ( !bvec[19] || !bvec[9]  )
    && (  bvec[0]  ||  bvec[17] )
    && ( !bvec[1]  ||  bvec[20] )
    && ( !bvec[21] ||  bvec[20] )
    && ( !bvec[22] ||  bvec[20] )
    && ( !bvec[21] || !bvec[20] )
```

```
    && (  bvec[22] || !bvec[20] );

  return value;
}
/******************************************************************************/
void i4_to_bvec ( int i4, int n, int bvec[] )
/******************************************************************************/
/*
  Purpose:
    I4_TO_BVEC converts an integer into a binary vector.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    20 March 2009

  Author:
    John Burkardt

  Parameters:
    Input, int I4, the integer.
    Input, int N, the dimension of the vector.
    Output, int BVEC[N], the vector of binary remainders.
*/
{
  int i;
  for ( i = n - 1; 0 <= i; i-- )
  {
    bvec[i] = i4 % 2;
    i4 = i4 / 2;
  }
  return;
}
/******************************************************************************/
void timestamp ( void )
/******************************************************************************/
/*
  Purpose:
    TIMESTAMP prints the current YMDHMS date as a time stamp.

  Example:
    31 May 2001 09:45:54 AM

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    24 September 2003
```

```
  Author:
    John Burkardt

  Parameters:
    None
*/
{
# define TIME_SIZE 40
  static char time_buffer[TIME_SIZE];
  const struct tm *tm;
  size_t len;
  time_t now;
  now = time ( NULL );
  tm = localtime ( &now );
  len = strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );
  printf ( "%s\n", time_buffer );
  return;
# undef TIME_SIZE
}
```

11.5. Quad

```
// program which approximates an integral using a quadrature rule, and
carries out the computation in parallel using OpenMP.
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <time.h>
# include <omp.h>

int main ( int argc, char *argv[] );
double f ( double x );
double cpu_time ( void );
void timestamp ( void );

/******************************************************************/
int main ( int argc, char *argv[] )
/******************************************************************/
/*
  Purpose:
    MAIN is the main program for QUAD_OPENMP.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    14 December 2011

  Author:
    John Burkardt
*/
```

```
{
  double a = 0.0;
  double b = 10.0;
  double error;
  double exact = 0.49936338107645674464;
  int i;
  int n = 10000000;
  double total;
  double wtime;
  double x;
  int nthreads;
  int S = atoi(argv[1]);
  omp_set_num_threads(S);
  nthreads = omp_get_num_threads();

  timestamp ( );
  printf ( "\n" );
  printf ( "QUAD_OPENMP:\n" );
  printf ( "  C version\n" );
  printf ( "  Use OpenMP for parallel execution.\n" );
  printf ( "  Estimate the integral of f(x) from A to B.\n" );
  printf ( "  f(x) = 50 / ( pi * ( 2500 * x * x + 1 ) ).\n" );
  printf ( "  Number of threads =  %d\n", omp_get_max_threads ( ) );
  printf ( "\n" );
  printf ( "  A        = %f\n", a );
  printf ( "  B        = %f\n", b );
  printf ( "  N        = %d\n", n );
  printf ( "  Exact    = %24.16f\n", exact );
  wtime = omp_get_wtime ( );

  total = 0.0;

# pragma omp parallel shared ( a, b, n ) private ( i, x )
# pragma omp for reduction ( + : total )

  for ( i = 0; i < n; i++ )
  {
    x = ( ( double ) ( n - i - 1 ) * a + ( double ) ( i ) * b ) / ( double )
( n - 1 );
    total = total + f ( x );
  }
  wtime = omp_get_wtime ( ) - wtime;
  total = ( b - a ) * total / ( double ) n;
  error = fabs ( total - exact );

  printf ( "\n" );
  printf ( "  Estimate = %24.16f\n", total );
  printf ( "  Error    = %e\n", error );
  printf ( "  Time     = %f\n", wtime );
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "QUAD_OPENMP:\n" );
  printf ( "  Normal end of execution.\n" );
  printf ( "\n" );
  timestamp ( );

  return 0;
```

```
}
/******************************************************************/
double f ( double x )
/******************************************************************/
/*
  Purpose:
     F evaluates the function.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    18 July 2010

  Author:
    John Burkardt

  Parameters:
    Input, double X, the argument.
    Output, double F, the value of the function.
*/
{
  double pi = 3.141592653589793;
  double value;

  value = 50.0 / ( pi * ( 2500.0 * x * x + 1.0 ) );

  return value;
}
/******************************************************************/
double cpu_time ( void )
/******************************************************************/
/*
  Purpose:
     CPU_TIME reports the total CPU time for a program.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    27 September 2005

  Author:
    John Burkardt

  Parameters:
    Output, double CPU_TIME, the current total elapsed CPU time in second.
*/
{
  double value;

  value = ( double ) clock ( ) / ( double ) CLOCKS_PER_SEC;

  return value;
}
/******************************************************************/
void timestamp ( void )
/******************************************************************/
/*
```

```
  Purpose:
    TIMESTAMP prints the current YMDHMS date as a time stamp.

  Example:
   31 May 2001 09:45:54 AM

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    24 September 2003

  Author:
    John Burkardt

  Parameters:
    None
*/
{
# define TIME_SIZE 40

  static char time_buffer[TIME_SIZE];
  const struct tm *tm;
  time_t now;

  now = time ( NULL );
  tm = localtime ( &now );

  strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );
  printf ( "%s\n", time_buffer );
  return;
# undef TIME_SIZE
}
```

11.6. Schedule

```
// a C program which demonstrates the default, static, and dynamic methods of
"scheduling" loop iterations in OpenMP to avoid work imbalance.

# include <stdlib.h>
# include <stdio.h>
# include <omp.h>

int main ( int argc, char *argv[] );
int prime_default ( int n );
int prime_static ( int n );
int prime_dynamic ( int n );

/******************************************************************/
int main ( int argc, char *argv[] )
/******************************************************************/
/*
  Purpose:
    MAIN is the main program for SCHEDULE_OPENMP.
```

```
  Discussion:
    This program demonstrates the difference between default,
      static and dynamic scheduling for a loop parallelized in     OpenMP.
    The purpose of scheduling is to deal with loops in which there is known
    or suspected imbalance in the work load.  In this example,
    if the work is divided in the default manner between two   threads,the
    second thread has 3 times the work of the first.

    Both static and dynamic scheduling, if used, even out the work
    so that both threads have about the same load.  This could be
    expected to decrease the run time of the loop by about 1/3.

  Licensing:

    This code is distributed under the GNU LGPL license.

  Modified:
    10 July 2010

  Author:
    John Burkardt
*/
{
  int n;
  int n_factor;
  int n_hi;
  int n_lo;
  int primes;
  double time1;
  double time2;
  double time3;
  int nthreads;
  omp_set_num_threads(3);
  nthreads = omp_get_num_threads();

  printf ( "\n" );
  printf ( "SCHEDULE_OPENMP\n" );
  printf ( "  C/OpenMP version\n" );
  printf ( "  Count the primes from 1 to N.\n" );
  printf ( "  This is an unbalanced work load, particular for two threads.\n"
);
  printf ( "  Demonstrate default, static and dynamic scheduling.\n" );
  printf ( "\n" );
  printf ( "  Number of processors available = %d\n", omp_get_num_procs ( )
);
  printf ( "  Number of threads =              %d\n", omp_get_max_threads ( )
);

  n_lo = 1;
  n_hi = 131072;
  n_factor = 2;

  printf ( "\n" );
  printf ( " Default   Static Dynamic\n" );
  printf ( " N  Pi(N) Time  Time   Time\n" );
  printf ( "\n" );
  n = n_lo;

  while ( n <= n_hi )
```

```
  {
    time1 = omp_get_wtime ( );
    primes = prime_default ( n );
    time1 = omp_get_wtime ( ) - time1;
    time2 = omp_get_wtime ( );
    primes = prime_static ( n );
    time2 = omp_get_wtime ( ) - time2;
    time3 = omp_get_wtime ( );
    primes = prime_dynamic ( n );
    time3 = omp_get_wtime ( ) - time3;

printf ( " %8d  %8d  %12f  %12f  %12f\n", n,primes,time1,time2, time3 );

    n = n * n_factor;
  }
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "SCHEDULE_OPENMP\n" );
  printf ( "  Normal end of execution.\n" );

  return 0;
}
/******************************************************************/
int prime_default ( int n )
/******************************************************************/
/*
  Purpose:
    PRIME_DEFAULT counts primes, using default scheduling.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    10 July 2010

  Author:
    John Burkardt

  Parameters:
    Input, int N, the maximum number to check.
    Output, int PRIME_DEFAULT, the number of prime numbers up to N.
*/
{
  int i;
  int j;
  int prime;
  int total = 0;

# pragma omp parallel \
  shared ( n ) \
  private ( i, j, prime )

# pragma omp for reduction ( + : total )
  for ( i = 2; i <= n; i++ )
  {
    prime = 1;
```

```
    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )
      {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }

  return total;
}
/****************************************************************/
int prime_static ( int n )
/****************************************************************/
/*
  Purpose:
    PRIME_STATIC counts primes using static scheduling.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    10 July 2010

  Author:
    John Burkardt

  Parameters:
    Input, int N, the maximum number to check.
    Output, int PRIME_STATIC, the number of prime numbers up to N.
*/
{
  int i;
  int j;
  int prime;
  int total = 0;

# pragma omp parallel \ shared ( n ) \ private ( i, j, prime )
# pragma omp for reduction ( + : total ) schedule ( static, 100 )
  for ( i = 2; i <= n; i++ )
  {
    prime = 1;
    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )
      {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
  return total;
}
/****************************************************************/
int prime_dynamic ( int n )
/****************************************************************/
```

```
/*
  Purpose:
    PRIME_DYNAMIC counts primes using dynamic scheduling.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    10 July 2010

  Author:
    John Burkardt

  Parameters:
    Input, int N, the maximum number to check.
    Output, int PRIME_DYNAMIC, the number of prime numbers up to N.
*/
{
  int i;
  int j;
  int prime;
  int total = 0;

# pragma omp parallel \shared ( n ) \ private ( i, j, prime )
# pragma omp for reduction ( + : total ) schedule ( dynamic, 100 )
  for ( i = 2; i <= n; i++ )
  {
    prime = 1;
    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )
      {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
  return total;
}
```

11.7. Helmholtz

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <omp.h>

int main ( int argc, char *argv[] );
void driver ( int m, int n, int it_max, double alpha, double omega,
double tol );
void error_check ( int m, int n, double alpha, double u[], double f[] );
void jacobi ( int m, int n, double alpha, double omega, double u[],
double f[],
  double tol, int it_max );
double *rhs_set ( int m, int n, double alpha );
double u_exact ( double x, double y );
```

```
double uxx_exact ( double x, double y );
double uyy_exact ( double x, double y );

/******************************************************************/
int main ( int argc, char *argv[] )
/******************************************************************/
/*
  Purpose:
    MAIN is the main program for HELMHOLTZ.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    19 April 2009

  Author:
    Original FORTRAN77 version by Joseph Robicheaux, Sanjiv Shah.
    C version by John Burkardt
*/
{
  double alpha = 0.25;
  int it_max = 100;
  int m = 500;
  int n = 500;
  double omega = 1.1;
  double tol = 1.0E-08;
  double wtime;
  int S = atoi(argv[1]);
  int nthreads;
  omp_set_num_threads(S);
  nthreads = omp_get_num_threads();

printf ( "\n" );
printf ( "HELMHOLTZ\n" );
printf ( "  C/OpenMP version\n" );
printf ( "\n" );
printf ( "A program which solves the 2D Helmholtz equation.\n" );
printf ( "\n" );
printf ( "  This program is being run in parallel.\n" );
printf ( "\n" );
printf ( "  Number of processors available = %d\n", omp_get_num_procs
( ) );
printf ( "  Number of threads = %d\n", omp_get_max_threads() );
printf ( "\n" );
printf ( "  The region is [-1,1] x [-1,1].\n" );
printf ( "  The number of nodes in the X direction is M = %d\n", m );
printf ( "  The number of nodes in the Y direction is N = %d\n", n );
printf ( "  Number of variables in linear system M * N  = %d\n", m * n );
printf ( "  The scalar coefficient in the Helmholtz equation is ALPHA
= %f\n", alpha );
printf ( "  The relaxation value is OMEGA = %f\n", omega );
printf ( "  The error tolerance is TOL = %f\n", tol );
printf ( "  The maximum number of Jacobi iterations is IT_MAX = %d\n",
it_max );
/*
  Call the driver routine.
*/
  wtime = omp_get_wtime ( );
```

```
  driver ( m, n, it_max, alpha, omega, tol );
  wtime = omp_get_wtime ( ) - wtime;
  printf ( "\n" );
  printf ( "  Elapsed wall clock time = %f\n", wtime );
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "HELMHOLTZ\n" );
  printf ( "  Normal end of execution.\n" );

  return 0;
}
/******************************************************************/
void driver ( int m, int n, int it_max, double alpha, double omega,
double tol )
/******************************************************************/
/*
  Purpose:
    DRIVER allocates arrays and solves the problem.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    Original FORTRAN77 version by Joseph Robicheaux, Sanjiv Shah.
    C version by John Burkardt

  Parameters:
    Input, int M, N, the number of grid points in the
    X and Y directions.
    Input, int IT_MAX, the maximum number of Jacobi
    iterations allowed.
    Input, double ALPHA, the scalar coefficient in the
    Helmholtz equation.
    Input, double OMEGA, the relaxation parameter, which
    should be strictly between 0 and 2.  For a pure Jacobi method,
    use OMEGA = 1.

    Input, double TOL, an error tolerance for the linear
    equation solver.
*/
{
  double *f;
  int i;
  int j;
  double *u;
/*
  Initialize the data.
*/
  f = rhs_set ( m, n, alpha );

  u = ( double * ) malloc ( m * n * sizeof ( double ) );

# pragma omp parallel \shared ( m, n, u ) \private ( i, j )
# pragma omp for
```

```
  for ( j = 0; j < n; j++ )
  {
    for ( i = 0; i < m; i++ )
    {
      u[i+j*m] = 0.0;
    }
  }
/*
  Solve the Helmholtz equation.
*/
  jacobi ( m, n, alpha, omega, u, f, tol, it_max );
/*
  Determine the error.
*/
  error_check ( m, n, alpha, u, f );
  free ( f );
  free ( u );
  return;
}
/****************************************************************/
void error_check ( int m, int n, double alpha, double u[], double f[] )
/****************************************************************/
/*
  Purpose:

    ERROR_CHECK determines the error in the numerical solution.

  Licensing:

    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    Original FORTRAN77 version by Joseph Robicheaux, Sanjiv Shah.
    C version by John Burkardt

  Parameters:
    Input, int M, N, the number of grid points in the
    X and Y directions.

    Input, double ALPHA, the scalar coefficient in the
    Helmholtz equation.  ALPHA should be positive.

    Input, double U[M*N], the solution of the Helmholtz equation
    at the grid points.

    Input, double F[M*N], values of the right hand side function
    for the Helmholtz equation at the grid points.
*/
{
  double error_norm;
  int i;
  int j;
  double u_norm;
  double u_true;
  double u_true_norm;
```

```
  double x;
  double y;

  u_norm = 0.0;

# pragma omp parallel \
  shared ( m, n, u ) \
  private ( i, j )

# pragma omp for reduction ( + : u_norm )

  for ( j = 0; j < n; j++ )
  {
    for ( i = 0; i < m; i++ )
    {
      u_norm = u_norm + u[i+j*m] * u[i+j*m];
    }
  }

  u_norm = sqrt ( u_norm );

  u_true_norm = 0.0;
  error_norm = 0.0;

# pragma omp parallel \
  shared ( m, n, u ) \
  private ( i, j, u_true, x, y )

# pragma omp for reduction ( + : error_norm, u_true_norm)

  for ( j = 0; j < n; j++ )
  {
    for ( i = 0; i < m; i++ )
    {
      x = ( double ) ( 2 * i - m + 1 ) / ( double ) ( m - 1 );
      y = ( double ) ( 2 * j - n + 1 ) / ( double ) ( n - 1 );
      u_true = u_exact ( x, y );
      error_norm = error_norm + ( u[i+j*m] - u_true ) * ( u[i+j*m] -
u_true );
      u_true_norm = u_true_norm + u_true * u_true;
    }
  }

  error_norm = sqrt ( error_norm );
  u_true_norm = sqrt ( u_true_norm );

  //printf ( "\n" );
  //printf ( "  Computed U l2 norm :       %f\n", u_norm );
 // printf ( "  Computed U_EXACT l2 norm : %f\n", u_true_norm );
  //printf ( "  Error l2 norm:             %f\n", error_norm );

  return;
}************************************************************/
void jacobi ( int m, int n, double alpha, double omega, double u[],
double f[],
  double tol, int it_max )
/************************************************************/
/*
  Purpose:
```

```
    JACOBI applies the Jacobi iterative method to solve the linear
system.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    Original FORTRAN77 version by Joseph Robicheaux, Sanjiv Shah.
    C version by John Burkardt

  Parameters:
    Input, int M, N, the number of grid points in the
    X and Y directions.
    Input, double ALPHA, the scalar coefficient in the
    Helmholtz equation.  ALPHA should be positive.
    Input, double OMEGA, the relaxation parameter, which
    should be strictly between 0 and 2.  For a pure Jacobi method,
    use OMEGA = 1.

    Input/output, double U(M,N), the solution of the Helmholtz
    equation at the grid points.

    Input, double F(M,N), values of the right hand side function
    for the Helmholtz equation at the grid points.

    Input, double TOL, an error tolerance for the linear
    equation solver.

    Input, int IT_MAX, the maximum number of Jacobi
    iterations allowed.
*/
{
  double ax;
  double ay;
  double b;
  double dx;
  double dy;
  double error;
  double error_norm;
  int i;
  int it;
  int j;
  double *u_old;
/*
  Initialize the coefficients.
*/
  dx = 2.0 / ( double ) ( m - 1 );
  dy = 2.0 / ( double ) ( n - 1 );

  ax = - 1.0 / dx / dx;
  ay = - 1.0 / dy / dy;
  b  = + 2.0 / dx / dx + 2.0 / dy / dy + alpha;

  u_old = ( double * ) malloc ( m * n * sizeof ( double ) );

  for ( it = 1; it <= it_max; it++ )
```

```
  {
    error_norm = 0.0;
/*
  Copy new solution into old.
*/
# pragma omp parallel \ shared ( m, n, u, u_old ) \
  private ( i, j )

# pragma omp for
    for ( j = 0; j < n; j++ )
    {
      for ( i = 0; i < m; i++ )
      {
        u_old[i+m*j] = u[i+m*j];
      }
    }
/*
  Compute stencil, residual, and update.
*/
# pragma omp parallel \
  shared ( ax, ay, b, f, m, n, omega, u, u_old ) \
  private ( error, i, j )

# pragma omp for reduction ( + : error_norm )
    for ( j = 0; j < n; j++ )
    {
      for ( i = 0; i < m; i++ )
      {
/*
  Evaluate the residual.
*/
       if ( i == 0 || i == m - 1 || j == 0 || j == n - 1 )
        {
          error = u_old[i+j*m] - f[i+j*m];
        }
        else
        {
          error = ( ax * ( u_old[i-1+j*m] + u_old[i+1+j*m] )
            + ay * ( u_old[i+(j-1)*m] + u_old[i+(j+1)*m] )
            + b * u_old[i+j*m] - f[i+j*m] ) / b;
        }
/*
  Update the solution.
*/
        u[i+j*m] = u_old[i+j*m] - omega * error;
/*
  Accumulate the residual error.
*/
        error_norm = error_norm + error * error;
      }
    }
/*
  Error check.
*/
    error_norm = sqrt ( error_norm ) / ( double ) ( m * n );
    printf ( "  %d  Residual RMS %e\n", it, error_norm );

    if ( error_norm <= tol )
    {
```

```
      break;
    }
  }
  printf ( "\n" );
  printf ( "  Total number of iterations %d\n", it );

  free ( u_old );

  return;
}
/******************************************************************/
double *rhs_set ( int m, int n, double alpha )
/******************************************************************/
/*
  Purpose:

    RHS_SET sets the right hand side F(X,Y).

  Discussion:

    The routine assumes that the exact solution and its second
    derivatives are given by the routine EXACT.

    The appropriate Dirichlet boundary conditions are determined
    by getting the value of U returned by EXACT.

    The appropriate right hand side function is determined by
    having EXACT return the values of U, UXX and UYY, and setting

      F = -UXX - UYY + ALPHA * U

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    Original FORTRAN77 version by Joseph Robicheaux, Sanjiv Shah.
    C version by John Burkardt

  Parameters:
    Input, int M, N, the number of grid points in the
    X and Y directions.
    Input, double ALPHA, the scalar coefficient in the
    Helmholtz equation.  ALPHA should be positive.

    Output, double RHS[M*N], values of the right hand side function
    for the Helmholtz equation at the grid points.
*/
{
  double *f;
  double f_norm;
  int i;
  int j;
  double x;
  double y;

  f = ( double * ) malloc ( m * n * sizeof ( double ) );
```

```
# pragma omp parallel \ shared ( f, m, n ) \ private ( i, j )
# pragma omp for

  for ( j = 0; j < n; j++ )
  {
    for ( i = 0; i < m; i++ )
    {
      f[i+j*m] = 0.0;
    }
  }
/*
  Set the boundary conditions.
*/

# pragma omp parallel \
  shared ( alpha, f, m, n ) \
  private ( i, j, x, y )
  {

# pragma omp for
    for ( i = 0; i < m; i++ )
    {
      j = 0;
      y = ( double ) ( 2 * j - n + 1 ) / ( double ) ( n - 1 );
      x = ( double ) ( 2 * i - m + 1 ) / ( double ) ( m - 1 );
      f[i+j*m] = u_exact ( x, y );
    }

# pragma omp for
    for ( i = 0; i < m; i++ )
    {
      j = n - 1;
      y = ( double ) ( 2 * j - n + 1 ) / ( double ) ( n - 1 );
      x = ( double ) ( 2 * i - m + 1 ) / ( double ) ( m - 1 );
      f[i+j*m] = u_exact ( x, y );
    }

# pragma omp for
    for ( j = 0; j < n; j++ )
    {
      i = 0;
      x = ( double ) ( 2 * i - m + 1 ) / ( double ) ( m - 1 );
      y = ( double ) ( 2 * j - n + 1 ) / ( double ) ( n - 1 );
      f[i+j*m] = u_exact ( x, y );
    }

# pragma omp for
    for ( j = 0; j < n; j++ )
    {
      i = m - 1;
      x = ( double ) ( 2 * i - m + 1 ) / ( double ) ( m - 1 );
      y = ( double ) ( 2 * j - n + 1 ) / ( double ) ( n - 1 );
      f[i+j*m] = u_exact ( x, y );
    }
/*
  Set the right hand side F.
*/
# pragma omp for
```

```
    for ( j = 1; j < n - 1; j++ )
    {
      for ( i = 1; i < m - 1; i++ )
      {
        x = ( double ) ( 2 * i - m + 1 ) / ( double ) ( m - 1 );
        y = ( double ) ( 2 * j - n + 1 ) / ( double ) ( n - 1 );
        f[i+j*m] = - uxx_exact ( x, y ) - uyy_exact ( x, y ) + alpha *
u_exact ( x, y );
      }
    }
  }

  f_norm = 0.0;

# pragma omp parallel \shared ( f, m, n ) \private ( i, j )
# pragma omp for reduction ( + : f_norm )

  for ( j = 0; j < n; j++ )
  {
    for ( i = 0; i < m; i++ )
    {
      f_norm = f_norm + f[i+j*m] * f[i+j*m];
    }
  }
  f_norm = sqrt ( f_norm );

  //printf ( "\n" );
 // printf ( "  Right hand side l2 norm = %f\n", f_norm );

  return f;
}
/*****************************************************************/
double u_exact ( double x, double y )
/*****************************************************************/
/*
  Purpose:
    U_EXACT returns the exact value of U(X,Y).

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    John Burkardt

  Parameters:

    Input, double X, Y, the point at which the values are needed.

    Output, double U_EXACT, the value of the exact solution.
*/
{
  double value;
  value = ( 1.0 - x * x ) * ( 1.0 - y * y );
  return value;
}
```

```
/*****************************************************************/
double uxx_exact ( double x, double y )
/*****************************************************************/
/*
  Purpose:

    UXX_EXACT returns the exact second X derivative of the solution.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    John Burkardt

  Parameters:
    Input, double X, Y, the point at which the values are needed.
    Output, double UXX_EXACT, the exact second X derivative.
*/
{
  double value;
  value = -2.0 * ( 1.0 + y ) * ( 1.0 - y );
  return value;
}
/*****************************************************************/
double uyy_exact ( double x, double y )
/*****************************************************************/
/*
  Purpose:
    UYY_EXACT returns the exact second Y derivative of the solution.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    John Burkardt

  Parameters:
    Input, double X, Y, the point at which the values are needed.
    Output, double UYY_EXACT, the exact second Y derivative.
*/
{
  double value;
  value = -2.0 * ( 1.0 + x ) * ( 1.0 - x );
  return value;
}
```

11.8. Multiplicacion de Matrices

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
```

```
# include <omp.h>

int main ( int argc, char *argv[] );
void r8_mxm ( int l, int m, int n );
double r8_uniform_01 ( int *seed );

/******************************************************************/
int main ( int argc, char *argv[] )
/******************************************************************/
/*
  Purpose:
    MAIN is the main program for MXM.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    19 April 2009

  Author:
    John Burkardt
*/
{
  int id;
  int l;
  int m;
  int n;
  int nthreads;
  int S = atoi(argv[1]);
  omp_set_num_threads(S);
  nthreads = omp_get_num_threads();

  printf ( "\n" );
  printf ( "MXM\n" );
  printf ( "  C/OpenMP version.\n" );
  printf ( "\n" );
  printf ( "  Matrix multiplication tests.\n" );
  printf ( "\n" );
  printf ( "  Number of processors available = %d\n", omp_get_num_procs ( )
);
  printf ( "  Number of threads =              %d\n", omp_get_max_threads ( )
);

  l = 500;
  m = 500;
  n = 500;

  r8_mxm ( l, m, n );
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "MXM:\n" );
  printf ( "  Normal end of execution.\n" );
  return 0;
}
/******************************************************************/
void r8_mxm ( int l, int m, int n )
/******************************************************************/
```

```
/*
  Purpose:
    R8_MXM carries out a matrix-matrix multiplication in R8  arithmetic.

  Discussion:
    A(LxN) = B(LxM) * C(MxN).

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:

    13 February 2008

  Author:
    John Burkardt

  Parameters:
    Input, int L, M, N, the dimensions that specify the sizes of the A, B,
and C matrices.
*/
{
  double *a;
  double *b;
  double *c;
  int i;
  int j;
  int k;
  int ops;
  double rate;
  int seed;
  double time_begin;
  double time_elapsed;
  double time_stop;
/*
  Allocate the matrices.
*/
  a = ( double * ) malloc ( l * n * sizeof ( double ) );
  b = ( double * ) malloc ( l * m * sizeof ( double ) );
  c = ( double * ) malloc ( m * n * sizeof ( double ) );
/*
  Assign values to the B and C matrices.
*/
  seed = 123456789;
  for ( k = 0; k < l * m; k++ )
  {
    b[k] = r8_uniform_01 ( &seed );
  }
  for ( k = 0; k < m * n; k++ )
  {
    c[k] = r8_uniform_01 ( &seed );
  }
/*
  Compute A = B * C.
*/
  time_begin = omp_get_wtime ( );

# pragma omp parallel\shared ( a, b, c, l, m, n )\private ( i, j,k)
# pragma omp for
```

```
  for ( j = 0; j < n; j++)
  {
    for ( i = 0; i < l; i++ )
    {
      a[i+j*l] = 0.0;
      for ( k = 0; k < m; k++ )
      {
        a[i+j*l] = a[i+j*l] + b[i+k*l] * c[k+j*m];
      }
    }
  }
  time_stop = omp_get_wtime ( );
/*
  Report.
*/
  ops = l * n * ( 2 * m );
  time_elapsed = time_stop - time_begin;
  rate = ( double ) ( ops ) / time_elapsed / 1000000.0;

  printf ( "\n" );
  printf ( "R8_MXM matrix multiplication timing.\n" );
  printf ( "  A(LxN) = B(LxM) * C(MxN).\n" );
  printf ( "  L = %d\n", l );
  printf ( "  M = %d\n", m );
  printf ( "  N = %d\n", n );
  printf ( "  Floating point OPS roughly %d\n", ops );
  printf ( "  Elapsed time dT = %f\n", time_elapsed );
  printf ( "  Rate = MegaOPS/dT = %f\n", rate );

  free ( a );
  free ( b );
  free ( c );

  return;
}
/******************************************************************/
double r8_uniform_01 ( int *seed )
/******************************************************************/
/*
  Purpose:
    R8_UNIFORM_01 is a unit pseudorandom R8.

  Discussion:
    This routine implements the recursion
      seed = 16807 * seed mod ( 2**31 - 1 )
      unif = seed / ( 2**31 - 1 )

    The integer arithmetic never requires more than 32 bits,
    including a sign bit.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    11 August 2004

  Author:
    John Burkardt
```

```
  Reference:
    Paul Bratley, Bennett Fox, Linus Schrage,
    A Guide to Simulation,
    Springer Verlag, pages 201-202, 1983.
    Bennett Fox,
    Algorithm 647:
    Implementation and Relative Efficiency of Quasirandom
    Sequence Generators,
    ACM Transactions on Mathematical Software,
    Volume 12, Number 4, pages 362-376, 1986.

  Parameters:
    Input/output, int *SEED, a seed for the random number generator.
    Output, double R8_UNIFORM_01, a new pseudorandom variate, strictly
between
    0 and 1.
*/
{
  int k;
  double r;

  k = *seed / 127773;
  *seed = 16807 * ( *seed - k * 127773 ) - k * 2836;

  if ( *seed < 0 )
  {
    *seed = *seed + 2147483647;
  }
  r = ( double ) ( *seed ) * 4.656612875E-10;
  return r;
}
```

11.9. Producto Escalar

```
/************************************
   orf.c     OPENMP
   calculo de un producto escalar
   reparto de tares tipo orphan
************************************/
#include <omp.h>
#include <stdio.h>
#define N 10000000
float A[N], B[N], pe;
void AporB()
{
  int j;
  #pragma omp for reduction(+:pe)
  for (j=0; j<N; j++) pe += A[j] * B[j];
}
int main ( int argc, char *argv[] )
{
  int S = atoi(argv[1]);
  int nthreads;
  double wtime;
  omp_set_num_threads(S);
  nthreads = omp_get_num_threads();
  int i;
```

```
  wtime = omp_get_wtime ( );
  for (i=0; i<N; i++)
  {
    A[i] = i;
    B[i] = N-i;
  }
  pe = 0.0;
  #pragma omp parallel
  {
    AporB();
  }
   wtime = omp_get_wtime ( ) - wtime;
  //printf("\n\n     >> PE = %10.0f\n\n", pe);
  printf ( "  Wallclock time = %f\n", wtime );
}
```

11.10. Método de sobrerrelajación sucesiva

```
#include <stdio.h>
#include <omp.h>

#define width 256
#define height 256

void openMP()
{
    double before = omp_get_wtime();
    float A[width][height], B[width][height];
    int i,j,t;
    #pragma omp parallel for private(i,j) shared(B)
    for(i=0; i<width; i++){
        for(j=0; j<height; j++){
            B[i][j] = (float) (i+1)/(j+1);
        }
    }
    for(t=0; t<1000; t++)
    {
        #pragma omp parallel for private(i,j) shared(A,B)
        for(i=1; i<width-1; i++)
        {
            for(j=1; j<height-1; j++)
            {
    A[i][j] = (B[i+1][j] + B[i-1][j] + B[i][j+1] + B[i][j-1])/4;
            }
        }

        #pragma omp parallel for private(i,j) shared(A,B)
        for(i=1; i<width-1; i++)
        {
            for(j=1; j<height-1; j++)
            {
                B[i][j] = A[i][j];
            }
        }
    }
   printf("%f sec OpenMP Complete.\n",omp_get_wtime()-before);
}
```

```
int main ( int argc, char *argv[] )
{
   int nthreads;
    int S = atoi(argv[1]);
    omp_set_num_threads(S);
    nthreads = omp_get_num_threads();
    openMP();
    return 0;
}
```

11.11. Fibonacci

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <omp.h>

long long fib(long long n)
{
        if (n < 2) {
        return 1;
        }
        return fib(n - 2) + fib(n - 1);
}

int main(int argc, char ** argv)
{
        double wtime;
        int nthreads;
        omp_set_num_threads(3);
        nthreads = omp_get_num_threads();
        long long n = 0;
        wtime = omp_get_wtime ( );

        #pragma omp parallel for schedule(guided, 1)
        for (n = 0; n <= 45; n++) {
                printf("Fib(%lld): %lld\n", n, fib(n));

        }
         wtime = omp_get_wtime ( ) - wtime;
         printf ( "  Wallclock time = %f\n", wtime );
         printf ( "Number of threads =  %d\n", omp_get_max_threads ( ) );
        return 0;
}
```

11.12. Calculo de PI

```
#include <stdio.h>
#include <omp.h>

static long num_steps = 1000000;
#define NUM_THREADS 2
double step;
void openMP()
{
```

```
    double before = omp_get_wtime();
    int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel shared(sum, num_steps, step) private(i, x)
    {
        #pragma omp for reduction(+:sum)
        for(i=1; i<=num_steps; i++)
        {
            x = (i-0.5) * step;
            sum = sum + 4.0/(1.0 + x*x);
        }

      #pragma omp single
        pi = step * sum;
    }

    printf("PI : %f\n",pi);
    printf("%f sec OpenMP Complete.\n",omp_get_wtime()-before);
    printf ( "  Number of threads =%d\n", omp_get_max_threads ( ) );
}
int main()
{
    openMP();
    return 0;
}
```

11.13. Números Primos

```
# include <stdlib.h>
# include <stdio.h>
# include <omp.h>

int main ( int argc, char *argv[] );
void prime_number_sweep ( int n_lo, int n_hi, int n_factor );
int prime_number ( int n );

/******************************************************************/
int main ( int argc, char *argv[] )

/******************************************************************/
/*
  Purpose:
    MAIN is the main program for PRIME_OPENMP.

  Discussion:

    This program calls a version of PRIME_NUMBER that includes
    OpenMP directives for parallel processing.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
```

```
    06 August 2009

  Author:
    John Burkardt
*/
{
  int n_factor;
  int n_hi;
  int n_lo;
  double wtime;
  int nthreads;
  omp_set_num_threads(4);
  nthreads = omp_get_num_threads();

  printf ( "\n" );
  printf ( "PRIME_OPENMP\n" );
  printf ( "  C/OpenMP version\n" );
  printf ( "\n" );
  printf ( "  Number of processors available = %d\n", omp_get_num_procs
( ) );
  printf ( "  Number of threads =              %d\n", omp_get_max_threads
( ) );
  printf("Numero de threads en ejecucion = %d\n", nthreads);
  wtime = omp_get_wtime ( );
  n_lo = 1;
  n_hi = 131072;
  n_factor = 2;
  prime_number_sweep ( n_lo, n_hi, n_factor );

  n_lo = 5;
  n_hi = 500000;
  n_factor = 10;

  prime_number_sweep ( n_lo, n_hi, n_factor );
  wtime = omp_get_wtime ( ) - wtime;
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "PRIME_OPENMP\n" );
  printf ( "  Wallclock time = %f\n", wtime );
  printf ( "  Normal end of execution.\n" );

  return 0;
}
/******************************************************************/
void prime_number_sweep ( int n_lo, int n_hi, int n_factor )
/******************************************************************/
/*
  Purpose:
   PRIME_NUMBER_SWEEP does repeated calls to PRIME_NUMBER.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    06 August 2009

  Author:
```

```
    John Burkardt

  Parameters:
    Input, int N_LO, the first value of N.
    Input, int N_HI, the last value of N.
    Input, int N_FACTOR, the factor by which to increase N after
    each iteration.
*/
{
  int i;
  int n;
  int primes;
  double wtime;

  printf ( "\n" );
  printf ( "TEST01\n" );
  printf ( "  Call PRIME_NUMBER to count the primes from 1 to N.\n" );
  printf ( "\n" );
  printf ( "         N        Pi          Time\n" );
  printf ( "\n" );

  n = n_lo;
  while ( n <= n_hi )
  {
    wtime = omp_get_wtime ( );
    primes = prime_number ( n );
    wtime = omp_get_wtime ( ) - wtime;
    printf ( "  %8d  %8d  %14f\n", n, primes, wtime );
    n = n * n_factor;
  }
  return;
}
/**********************************************************/
int prime_number ( int n )
/**********************************************************/
/*
  Purpose:
    PRIME_NUMBER returns the number of primes between 1 and N.

  Discussion:
    A naive algorithm is used.
Mathematica can return the number of primes less than or equal to N
    by the command PrimePi[N].

                N  PRIME_NUMBER

                1           0
               10           4
              100          25
            1,000         168
           10,000       1,229
          100,000       9,592
        1,000,000      78,498
       10,000,000     664,579
      100,000,000   5,761,455
    1,000,000,000  50,847,534

 Licensing:
    This code is distributed under the GNU LGPL license.
```

```
 Modified:
    21 May 2009

 Author:
    John Burkardt

 Parameters:
  Input, int N, the maximum number to check.
  Output, int PRIME_NUMBER, the number of prime numbers up to N.
*/
{
  int i;
  int j;
  int prime;
  int total = 0;
# pragma omp parallel \shared ( n ) \ private ( i, j, prime )
  # pragma omp for reduction ( + : total )
  for ( i = 2; i <= n; i++ )
  {
    prime = 1;
    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )
      {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
  return total;
}
```

11.14. Multiplicacion de matrices con for nowait

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define DEBUG 0
#define NRA 1600                 // number of rows in matrix A
#define NCA 1600                 // number of columns in matrix A
#define NCB 1600                 // number of columns in matrix B

int main (int argc, char *argv[]) {
  int   tid, nthreads, i, j, k;
  double **a, **b, **c;
  double *a_block, *b_block, *c_block;
  double **res;
  double *res_block;
  double starttime, stoptime;

  /* Check that the user gives number of threads */
  if (argc < 2) {
    printf("Usage: %s N    where N is number of threads\n", argv[0]);
    exit(0);
  }
```

```
  else {
    nthreads = atoi(argv[1]);
  }
  printf("Number of threads is %d\n", nthreads);
a = (double **) malloc(NRA*sizeof(double *)); /* matrix a to be
multiplied */
b = (double **) malloc(NCA*sizeof(double *)); /* matrix b to be
multiplied */
c = (double **) malloc(NRA*sizeof(double *)); /* result matrix c */

a_block = (double *) malloc(NRA*NCA*sizeof(double)); /* Storage for
matrix a */
b_block = (double *) malloc(NCA*NCB*sizeof(double));
c_block = (double *) malloc(NRA*NCB*sizeof(double));

res = (double **) malloc(NRA*sizeof(double *));
res_block = (double *) malloc(NRA*NCB*sizeof(double));

for (i=0; i<NRA; i++)   /* Initialize pointers to a */
    a[i] = a_block+i*NRA;

for (i=0; i<NCA; i++)   /* Initialize pointers to b */
    b[i] = b_block+i*NCA;

for (i=0; i<NRA; i++)   /* Initialize pointers to c */
c[i] = c_block+i*NRA;

for (i=0; i<NRA; i++)   /* Initialize pointers to res */
    res[i] = res_block+i*NRA;

#pragma omp parallel shared(a,b,c,nthreads) private(tid,i,j,k)
num_threads(nthreads)
  {
  tid = omp_get_thread_num();
  if (tid == 0) {
  ////nthreads = omp_get_num_threads();
  printf("Starting matrix multiplication with %d threads\n",nthreads);
 printf("Initializing matrices...\n");
    }
    /*** Initialize matrices ***/
#pragma omp for nowait
    for (i=0; i<NRA; i++)
      for (j=0; j<NCA; j++)
   a[i][j]= (double) (i+j);
#pragma omp for nowait
    for (i=0; i<NCA; i++)
      for (j=0; j<NCB; j++)
   b[i][j]= (double) (i*j);
#pragma omp for
    for (i=0; i<NRA; i++)
      for (j=0; j<NCB; j++)
   c[i][j]= 0.0;

    if (tid == 0)
      starttime = omp_get_wtime();  /* Master thread measures the
execution time */

printf("Thread %d starting matrix multiply...\n",tid);
#pragma omp for nowait
```

```
    for (i=0; i<NRA; i++) {
      if (DEBUG) printf("Thread=%d did row=%d\n",tid,i);
      for(j=0; j<NCB; j++) {
   for (k=0; k<NCA; k++) {
     c[i][j] += a[i][k] * b[k][j];
   }
      }
    }

    if (tid == 0) {
      stoptime = omp_get_wtime();
      printf("Time for parallel matrix multiplication: %3.2f\n",
stoptime-starttime);
    }
  }   /*** End of parallel region ***/

  starttime = omp_get_wtime();
  /* Do a sequential matrix multiplication and compare the results */
  for (i=0; i<NRA; i++) {
    for (j=0; j<NCB; j++) {
      res[i][j] = 0.0;
      for (k=0; k<NCA; k++)
   res[i][j] += a[i][k]*b[k][j];
    }
  }
  stoptime = omp_get_wtime();
  printf("Time for sequential matrix multiplication: %3.2f\n", stoptime-
starttime);

  /* Check that the results are the same as in the parallel solution.
     Actually, you should not compare floating point values for equality
like this
     but instead compute the difference between the two values and check
that it
     is smaller than a very small value epsilon. However, since all
values in the
     matrices here are integer values, this will work.
  */
  for (i=0; i<NRA; i++) {
    for (j=0; j<NCB; j++) {
      if (res[i][j] == c[i][j]) {
   /* Everything is OK if they are equal */
      }
      else {
   printf("Different result %5.1f != %5.1f in %d %d\n ", res[i][j],
c[i][j], i, j);
      }
    }
  }

  /* If DEBUG is true, print the results */
  if (DEBUG) {
    printf("Result Matrix:\n");
    for (i=0; i<NRA; i++) {
      for (j=0; j<NCB; j++)
   printf("%6.1f ", c[i][j]);
      printf("\n");
    }
  }
```

```
      printf ("Done.\n");
      exit(0);
    }
```

11.15. Critical

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{
    int i, thread_id;
      int S = atoi(argv[1]);
      int nthreads;
      omp_set_num_threads(S);
      double wtime;
     int glob_nloops, priv_nloops;
    glob_nloops = 0;

    // parallelize this chunk of code
    #pragma omp parallel private(priv_nloops, thread_id)
    {
        wtime = omp_get_wtime ( );
        priv_nloops = 0;
        thread_id = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        // parallelize this for loop
        #pragma omp for
        for (i=0; i<100000; ++i)
        {
            ++priv_nloops;
        }
        // make this a "critical" code section
        #pragma omp critical
        {
           //  printf("Thread %d is adding its iterations (%d) to sum (%d),
", thread_id, priv_nloops, glob_nloops);
            glob_nloops += priv_nloops;
           //  printf(" total nloops is now %d.\n", glob_nloops);
        }
    }
    wtime = omp_get_wtime ( ) - wtime;
   //  printf("Total # loop iterations is %d\n", glob_nloops);
    printf ( "  Wallclock time = %f\n", wtime );
    return 0;
}
```

11.16. Ecuancion De La Onda

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <sys/time.h>
```

```
#include <omp.h>

#define dimx 300
#define dimz 300

using namespace std;

int tout=1000;
int Nx=201, Nz=201;
float alpha = 1.0, beta=1.0;
float c = 10;
float dt=0.01, G=0.1;
float U1[201][201], U2[201][201], U3[201][201][1000] ;
float dx, dz, x, z, t;
float x0=50.0, z0=50.0, r ;
char namefile[15];

FILE *out;

int main()
{
struct timeval t0, t1;
double tej;
int i,j,n;

gettimeofday(&t0, 0);   // Tomamos tiempo de inicio
//out = fopen("2Ddata.txt","w");
dx = c*dt/G;
dz = dx;
   omp_set_num_threads(4);
#pragma omp parallel
{
  #pragma omp for private(i, j, x, z, r) schedule(static)
  for (i=0; i<Nx; i++)
   {
     for (j=0; j<Nz; j++)
      {
        x = i*dx-x0;
        z = j*dz-z0;
          r = sqrt(x*x+z*z);
          U2[i][j] = sin(alpha*r)*exp(-beta*r*r);
          U1[i][j] = 0.0;
          //U3[i][j] = 0.0;
          /* U1[i] = U2[i];*/
      }
    }
} // pragma
// #pragma omp parallel private(i,j)
for(n=0; n<tout; n++)
```

```
{
  #pragma omp parallel private(i,j)
{
   #pragma omp  for schedule(static)
    for ( i=1; i<Nx-1; i++)
     {
       for ( j=1; j<Nz-1; j++)
         {
           U3[i][j][n] = 2*(1-2*G*G) * U2[i][j] + G*G*(U2[i+1][j] + U2[i-1][j]+ U2[i][j+1]+
U2[i][j-1]) - U1[i][j];
         }
     }

   #pragma omp  for schedule(static)
   for ( i=1; i<Nx-1; i++)
    {
     for ( j=1; j<Nz-1; j++)
      {
        U1[i][j] = U2[i][j];
        U2[i][j] = U3[i][j][n];
       }
    }
  }  // pragma
}  // for
/************** Acceso a  disco ************************
#pragma omp parallel for private(i,j,namefile,out)
for(n=0; n<tout; n++)
 {
   sprintf(namefile,"../plot2d/2Ddata%04d.txt",n);
   out = fopen(namefile,"w");
     for ( i=1; i<Nx-1; i++)
       {
         for ( j=1; j<Nz-1; j++)
          {
            fprintf(out,"%f ",U3[i][j][n]);
          }
         fprintf(out,"\n");
       }
   fclose (out);
 }
/*********************************************************/


gettimeofday(&t1, 0);  // Tomamos tiempo final
tej = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec) / 1e6;
printf("\n Tiempo de ejecucion: %1.3f ms\n\n", tej*1000);
return (0);
}
```

12. Algoritmos Secuencial

12.1. Poisson

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <time.h>

# define NX 161
# define NY 161
int main(int argc, char ** argv);
double r8mat_rms ( int nx, int ny, double a[NX][NY] );
void rhs ( int nx, int ny, double f[NX][NY] );
void sweep ( int nx, int ny, double dx, double dy, double f[NX][NY],
int itold, int itnew, double u[NX][NY], double unew[NX][NY] );
void timestamp ( void );
double u_exact ( double x, double y );
double uxxyy_exact ( double x, double y );

/******************************************************/
int main(int argc, char ** argv)
/******************************************************/
{
  int converged;
  double diff;
  double dx;
  double dy;
  double error;
  double f[NX][NY];
  int i;
  int id;
  int itnew;
  int itold;
  int j;
  int jt;
  int jt_max = 20;
  int nx = NX;
  int ny = NY;
  double tolerance = 0.000001;
  double u[NX][NY];
  double u_norm;
  double udiff[NX][NY];
  double uexact[NX][NY];
  double unew[NX][NY];
  double unew_norm;
  double wtime;
  double x;
  double y;
  dx = 1.0 / ( double ) ( nx - 1 );
  dy = 1.0 / ( double ) ( ny - 1 );
/*
  Print a message.
*/
  timestamp ( );
  printf ( "\n" );
  printf ( "POISSON_SECUENCIAL:\n" );
```

```
  printf ( "  C version\n" );
  printf ( "  A program for solving the Poisson equation.\n" );
  printf ( "\n" );
  printf ( "  -DEL^2 U = F(X,Y)\n" );
  printf ( "\n" );
  printf ( "  on the rectangle 0 <= X <= 1, 0 <= Y <= 1.\n" );
  printf ( "\n" );
   printf ( "  F(X,Y) = pi^2 * ( x^2 + y^2 ) * sin ( pi * x * y )\n" );
   printf ( "\n" );
   printf ( "  The number of interior X grid points is %d\n", nx );
   printf ( "  The number of interior Y grid points is %d\n", ny );
   printf ( "  The X grid spacing is %f\n", dx );
   printf ( "  The Y grid spacing is %f\n", dy );
/*
  Set the right hand side array F.
*/
  rhs ( nx, ny, f );
/*
  Set the initial solution estimate UNEW.
  We are "allowed" to pick up the boundary conditions exactly.
*/
  for ( j = 0; j < ny; j++ )
  {
    for ( i = 0; i < nx; i++ )
    {
      if ( i == 0 || i == nx - 1 || j == 0 || j == ny - 1 )
      {
        unew[i][j] = f[i][j];
      }
      else
      {
        unew[i][j] = 0.0;
      }
    }
  }
  unew_norm = r8mat_rms ( nx, ny, unew );
/*
  Set up the exact solution UEXACT.
*/
  for ( j = 0; j < ny; j++ )
  {
    y = ( double ) ( j ) / ( double ) ( ny - 1 );
    for ( i = 0; i < nx; i++ )
    {
      x = ( double ) ( i ) / ( double ) ( nx - 1 );
      uexact[i][j] = u_exact ( x, y );
    }
  }
  u_norm = r8mat_rms ( nx, ny, uexact );
  printf ( "  RMS of exact solution = %g\n", u_norm );
/*
  Do the iteration.
*/
  converged = 0;
  printf ( "\n" );
  printf ( "  Step ||Unew||  ||Unew-U||  ||Unew-Exact||\n" );
  printf ( "\n" );
  for ( j = 0; j < ny; j++ )
  {
```

```
    for ( i = 0; i < nx; i++ )
    {
      udiff[i][j] = unew[i][j] - uexact[i][j];
    }
  }
  error = r8mat_rms ( nx, ny, udiff );
  printf ( "  %4d  %14g  %14g\n", 0, unew_norm, error );
  wtime = omp_get_wtime ( );
  itnew = 0;
  for ( ; ; )
  {
    itold = itnew;
    itnew = itold + 500;
/*
  SWEEP carries out 500 Jacobi steps in parallel before we come
  back to check for convergence.
*/
    sweep ( nx, ny, dx, dy, f, itold, itnew, u, unew );
/*
  Check for convergence.
*/
    u_norm = unew_norm;
    unew_norm = r8mat_rms ( nx, ny, unew );
    for ( j = 0; j < ny; j++ )
    {
      for ( i = 0; i < nx; i++ )
      {
        udiff[i][j] = unew[i][j] - u[i][j];
      }
    }
    diff = r8mat_rms ( nx, ny, udiff );
    for ( j = 0; j < ny; j++ )
    {
      for ( i = 0; i < nx; i++ )
      {
        udiff[i][j] = unew[i][j] - uexact[i][j];
      }
    }
    error = r8mat_rms ( nx, ny, udiff );

printf ( " %4d %14g %14g %14g\n",itnew,unew_norm, diff, error );
    if ( diff <= tolerance )
    {
      converged = 1;
      break;
    }
  }

 if ( converged )
  {
    printf ( "converged.\n" );
  }
 else
  {
    printf ( "NOT converged.\n" );
  }
  wtime = omp_get_wtime ( ) - wtime;
  printf ( "\n" );
  printf ( "%g\n", wtime );
```

```
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "POISSON_SECUENCIAL:\n" );
  printf ( "  Normal end of execution.\n" );
  printf ( "\n" );
  timestamp ( );
  return 0;
}
/**********************************************************/
double r8mat_rms ( int nx, int ny, double a[NX][NY] )
/**********************************************************/
/*
  Purpose:
    R8MAT_RMS returns the RMS norm of a vector stored as a matrix.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    01 March 2003

  Author:
    John Burkardt

  Parameters:
    Input, int NX, NY, the number of rows and columns in A.
    Input, double A[NX][NY], the vector.
    Output, double R8MAT_RMS, the root mean square of the entries of A.
*/
{
  int i;
  int j;
  double v;
  v = 0.0;

  for ( j = 0; j < ny; j++ )
  {
    for ( i = 0; i < nx; i++ )
    {
      v = v + a[i][j] * a[i][j];
    }
  }
  v = sqrt ( v / ( double ) ( nx * ny )  );
  return v;
}
/**********************************************************/
void rhs ( int nx, int ny, double f[NX][NY] )
/**********************************************************/
/*
Purpose:
    RHS initializes the right hand side "vector".

Discussion:
    It is convenient for us to set up RHS as a 2D array.  However, each
    entry of RHS is really the right hand side of a linear system of the
form
    A * U = F
```

```
    In cases where U(I,J) is a boundary value, then the equation is
simply
    U(I,J) = F(i,j)
    and F(I,J) holds the boundary data.
    Otherwise, the equation has the form
    (1/DX^2)*(U(I+1,J)+U(I-1,J)+U(I,J-1)+U(I,J+1)-4*U(I,J))= F(I,J)
    where DX is the spacing and F(I,J) is the value at X(I), Y(J) of
      pi^2 * ( x^2 + y^2 ) * sin ( pi * x * y )

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    28 October 2011

  Author:
    John Burkardt

  Parameters:
    Input, int NX, NY, the X and Y grid dimensions.
    Output, double F[NX][NY], the initialized right hand side data.
*/
{
  double fnorm;
  int i;
  int j;
  double x;
  double y;
/*
  The "boundary" entries of F store the boundary values of the solution.
  The "interior" entries of F store the right hand sides of the Poisson
equation.
*/
  for ( j = 0; j < ny; j++ )
  {
    y = ( double ) ( j ) / ( double ) ( ny - 1 );
    for ( i = 0; i < nx; i++ )
    {
      x = ( double ) ( i ) / ( double ) ( nx - 1 );
      if ( i == 0 || i == nx - 1 || j == 0 || j == ny - 1 )
      {
        f[i][j] = u_exact ( x, y );
      }
      else
      {
        f[i][j] = - uxxyy_exact ( x, y );
      }
    }
  }
  fnorm = r8mat_rms ( nx, ny, f );

   printf ( "  RMS of F = %g\n", fnorm );

  return;
}
/*****************************************************/
void sweep ( int nx, int ny, double dx, double dy, double f[NX][NY],
int itold, int itnew, double u[NX][NY], double unew[NX][NY] )
/*****************************************************/
```

```
/*
  Purpose:
   SWEEP carries out one step of the Jacobi iteration.

  Discussion:
    Assuming DX = DY, we can approximate
      - ( d/dx d/dx + d/dy d/dy ) U(X,Y)
    by
   (U(i-1,j)+U(i+1,j)+U(i,j-1) + U(i,j+1)-4*U(i,j) )/dx / dy
The discretization employed below will not be correct in the    general
case where DX and DY are not equal.  It's only a little more complicated
to allow DX and DY to be different, but we're not going to worry about
that right now.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    14 December 2011

  Author:
    John Burkardt

  Parameters:
    Input, int NX, NY, the X and Y grid dimensions.
    Input, double DX, DY, the spacing between grid points.
    Input, double F[NX][NY], the right hand side data.
    Input, int ITOLD, the iteration index on input.
    Input, int ITNEW, the desired iteration index
    on output.
    Input, double U[NX][NY], the solution estimate on
    iteration ITNEW-1.
    Input/output, double UNEW[NX][NY], on input, the solution
    estimate on iteration ITOLD.  On output, the solution estimate on
    iteration ITNEW.
*/
{
  int i;
  int it;
  int j;

  for ( it = itold + 1; it <= itnew; it++ )
  {
/*
  Save the current estimate.
*/
    for ( j = 0; j < ny; j++ )
    {
      for ( i = 0; i < nx; i++ )
      {
        u[i][j] = unew[i][j];
      }
    }
/*
  Compute a new estimate.
*/
    for ( j = 0; j < ny; j++ )
    {
      for ( i = 0; i < nx; i++ )
```

```
      {
        if ( i == 0 || j == 0 || i == nx - 1 || j == ny - 1 )
        {
          unew[i][j] = f[i][j];
        }
        else
        {
          unew[i][j] = 0.25 * (
            u[i-1][j] + u[i][j+1] + u[i][j-1] + u[i+1][j] + f[i][j] * dx
* dy );
        }
      }
    }
  }
  return;
}
/******************************************************/
void timestamp ( void )
/******************************************************/
/*
  Purpose:
    TIMESTAMP prints the current YMDHMS date as a time stamp.

  Example:
    31 May 2001 09:45:54 AM

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    24 September 2003

  Author:
    John Burkardt

  Parameters:
    None
*/
{
# define TIME_SIZE 40

  static char time_buffer[TIME_SIZE];
  const struct tm *tm;
  time_t now;
  now = time ( NULL );
  tm = localtime ( &now );
  strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );
  printf ( "%s\n", time_buffer );
  return;
# undef TIME_SIZE
}
/******************************************************/
double u_exact ( double x, double y )
/******************************************************/
/*
  Purpose:
    U_EXACT evaluates the exact solution.

  Licensing:
```

```
    This code is distributed under the GNU LGPL license.

  Modified:
    25 October 2011

  Author:
    John Burkardt

  Parameters:
    Input, double X, Y, the coordinates of a point.
    Output, double U_EXACT, the value of the exact solution
    at (X,Y).
*/
{
  double pi = 3.141592653589793;
  double value;
  value = sin ( pi * x * y );
  return value;
}
/******************************************************/
double uxxyy_exact ( double x, double y )
/******************************************************/
/*
  Purpose:
    UXXYY_EXACT evaluates ( d/dx d/dx + d/dy d/dy ) of the exact
solution.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    25 October 2011

  Author:
    John Burkardt

  Parameters:
    Input, double X, Y, the coordinates of a point.
    Output, double UXXYY_EXACT, the value of
    ( d/dx d/dx + d/dy d/dy ) of the exact solution at (X,Y).
*/
{
  double pi = 3.141592653589793;
  double value;
  value = - pi * pi * ( x * x + y * y ) * sin ( pi * x * y );
  return value;
}
# undef NX
# undef NY
```

12.2. Ecuacion De Calor En 2D

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>

int main(int argc, char ** argv);
```

```
/******************************************************************/
int main(int argc, char ** argv)
/******************************************************************/
/*
   MAIN is the main program for Ecuacion_Calor_OPENMP.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    18 October 2011

  Author:
    Original C version by Michael Quinn.
    This C version by John Burkardt.

  Reference:
    Michael Quinn,
    Parallel Programming in C with MPI and OpenMP,
    McGraw-Hill, 2004,
    ISBN13: 978-0071232654,
    LC: QA76.73.C15.Q55.

  Local parameters:

    Local, double DIFF, the norm of the change in the solution from one
iteration
    to the next.

    Local, double MEAN, the average of the boundary values, used to
initialize
    the values of the solution in the interior.

    Local, double U[M][N], the solution at the previous iteration.

    Local, double W[M][N], the solution computed at the latest iteration.
*/
{
# define M 500
# define N 500
  double diff;
  double epsilon = 0.001;
  int i;
  int iterations;
  int iterations_print;
  int j;
  double mean;
  double my_diff;
  double u[M][N];
  double w[M][N];
  double wtime;

  printf ( "\n" );
  printf ( "ECUACION_CALOR_SECUENCIAL\n" );
  printf ( "  C/OpenMP version\n" );
  printf ( "  A program to solve for the steady state temperature
distribution\n" );
  printf ( "  over a rectangular plate.\n" );
  printf ( "\n" );
```

```
  printf ( "  Spatial grid of %d by %d points.\n", M, N );
  printf ( "  The iteration will be repeated until the change is
<= %e\n", epsilon );
*/

/*
  Set the boundary values, which don't change.
*/
  mean = 0.0;
  {
    for ( i = 1; i < M - 1; i++ )
    {
      w[i][0] = 100.0;
    }
    for ( i = 1; i < M - 1; i++ )
    {
      w[i][N-1] = 100.0;
    }
    for ( j = 0; j < N; j++ )
    {
      w[M-1][j] = 100.0;
    }
    for ( j = 0; j < N; j++ )
    {
      w[0][j] = 0.0;
    }
/*
  Average the boundary values, to come up with a reasonable
  initial value for the interior.
*/
    for ( i = 1; i < M - 1; i++ )
    {
      mean = mean + w[i][0] + w[i][N-1];
    }
    for ( j = 0; j < N; j++ )
    {
      mean = mean + w[M-1][j] + w[0][j];
    }
  }
  mean = mean / ( double ) ( 2 * M + 2 * N - 4 );
  printf ( "\n" );
  printf ( "  MEAN = %f\n", mean );
/*
  Initialize the interior solution to the mean value.
*/
  {
    for ( i = 1; i < M - 1; i++ )
    {
      for ( j = 1; j < N - 1; j++ )
      {
        w[i][j] = mean;
      }
    }
  }
/*
  iterate until the  new solution W differs from the old solution U
  by no more than EPSILON.
*/
  iterations = 0;
```

```
    iterations_print = 1;
    printf ( "\n" );
    printf ( " Iteration  Change\n" );
    printf ( "\n" );
    diff = epsilon;
    while ( epsilon <= diff )
    {
      {
  /*
    Save the old solution in U.
  */
        for ( i = 0; i < M; i++ )
        {
          for ( j = 0; j < N; j++ )
          {
            u[i][j] = w[i][j];
          }
        }
  /*
    Determine the new estimate of the solution at the interior points.
    The new solution W is the average of north, south, east and west
  neighbors.
  */
        for ( i = 1; i < M - 1; i++ )
        {
          for ( j = 1; j < N - 1; j++ )
          {
w[i][j] = ( u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] ) / 4.0;
          }
        }
      }
  /*
    C and C++ cannot compute a maximum as a reduction operation.
    Therefore, we define a private variable MY_DIFF for each thread.
    Once they have all computed their values, we use a CRITICAL section to
  update DIFF.
  */
      diff = 0.0;
      {
        my_diff = 0.0;
        for ( i = 1; i < M - 1; i++ )
        {
          for ( j = 1; j < N - 1; j++ )
          {
            if ( my_diff < fabs ( w[i][j] - u[i][j] ) )
            {
              my_diff = fabs ( w[i][j] - u[i][j] );
            }
          }
        }
        {
          if ( diff < my_diff )
          {
            diff = my_diff;
          }
        }
      }
      iterations++;
      if ( iterations == iterations_print )
```

```
      {
        printf ( "  %8d  %f\n", iterations, diff );
        iterations_print = 2 * iterations_print;
      }
    }
    printf ( "\n" );
    printf ( "  %8d  %f\n", iterations, diff );
    printf ( "\n" );
    printf ( "  Error tolerance achieved.\n" );
    /*
    Terminate.
  */
    printf ( "\n" );
    printf ( "ECUACION_CALOR_OPENMP:\n" );
    printf ( "  Normal end of execution.\n" );
    return 0;

  # undef M
  # undef N
  }
```

12.3. Ziggurat

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <time.h>
# include <stdint.h>
int main(int argc, char ** argv);
void test01 ( );
void test02 ( );
void test03 ( );
void test04 ( );
float r4_exp ( uint32_t *jsr, uint32_t ke[256], float fe[256], float
we[256] );
void r4_exp_setup ( uint32_t ke[256], float fe[256], float we[256] );
float r4_nor ( uint32_t *jsr, uint32_t kn[128], float fn[128], float
wn[128] );
void r4_nor_setup ( uint32_t kn[128], float fn[128], float wn[128] );
float r4_uni ( uint32_t *jsr );
uint32_t shr3_seeded ( uint32_t *jsr );
void timestamp ( );

/******************************************************************/
int main(int argc, char ** argv)
/******************************************************************/
/*
  Purpose:
    MAIN is the main program for ZIGGURAT_OPENMP.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    04 October 2013

  Author:
    John Burkardt
```

```
*/
{
  double wtime;
  timestamp ( );
  printf ( "\n" );
  printf ( "ZIGGURAT_SECUENCIAL:\n" );
  printf ( "  C version\n" );
  wtime = omp_get_wtime ( );
  test01 ( );
  test02 ( );
  test03 ( );
  test04 ( );
  wtime = omp_get_wtime ( ) - wtime;

/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "ZIGGURAT_SECUENCIAL:\n" );
  printf ( "  Normal end of execution.\n" );
  printf ( " %f\n", wtime );
  printf ( "\n" );
  timestamp ( );

  return 0;
}
/**************************************************************/
void test01 ( )
/**************************************************************/
/*
  Purpose:
    TEST01 tests SHR3_SEEDED.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    04 October 2013

  Author:
    John Burkardt
*/
{
  uint32_t jsr;
  uint32_t jsr_value;
  double mega_rate_par;
  double mega_rate_seq;
  int r;
  int r_num = 1000;
  int *result_par;
  int *result_seq;
  int s;
  int s_num = 10000;
  uint32_t *seed;
  int thread;
  int thread_num;
  double wtime_par;
  double wtime_seq;
```

```
  printf ( "\n" );
  printf ( "TEST01\n" );
  printf ( "  SHR3_SEEDED computes random integers.\n" );
  printf ( "  Since the output is completely determined\n" );
  printf ( "  by the input value of SEED, we can run in\n" );
  /*
  Set up the SEED array, which will be used for both sequential and
  parallel computations.
*/
  seed = ( uint32_t * ) malloc ( thread_num * sizeof ( uint32_t ) );
  result_seq = ( int * ) malloc ( thread_num * sizeof ( int ) );
  result_par = ( int * ) malloc ( thread_num * sizeof ( int ) );
/*
  Sequential execution.
  The sequential execution will only match the parallel execution if we
can
  guarantee that the parallel threads are scheduled to execute the R loop
  consecutively.
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_seq = omp_get_wtime ( );
  for ( r = 0; r < r_num; r++ )
  {
    thread = ( r % thread_num );
    jsr = seed[thread];
    for ( s = 0; s < s_num; s++ )
    {
      jsr_value = shr3_seeded ( &jsr );
    }
    result_seq[thread] = jsr_value;
    seed[thread] = jsr;
  }
  wtime_seq = omp_get_wtime ( ) - wtime_seq;
  mega_rate_seq = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_seq
/ 1000000.0;
/*
  Parallel.
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_par = omp_get_wtime ( );
  shared ( result_par, seed ) \
  private ( jsr, jsr_value, r, s, thread )
  {
    for ( r = 0; r < r_num; r++ )
    {
      thread = omp_get_thread_num ( );
      jsr = seed[thread];
      for ( s = 0; s < s_num; s++ )
      {
        jsr_value = shr3_seeded ( &jsr );
      }
```

```
      result_par[thread] = jsr_value;
      seed[thread] = jsr;
    }
  }
  wtime_par = omp_get_wtime ( ) - wtime_par;
  mega_rate_par = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_par
/ 1000000.0;
/*
  Report.
*/
printf ( "\n" );
printf ( "  Correctness check:\n" );
printf ( "\n" );
printf ( "  Computing values sequentially should reach the\n" );
printf ( "  same result as doing it in parallel:\n" );
printf ( "\n" );
printf ( "  THREAD Sequential Parallel    Difference\n" );
printf ( "\n" );

for ( thread = 0; thread < thread_num; thread++ )
  {
printf ( "  %8d  %12d  %12d  %12d\n",  thread, result_seq[thread],
result_par[thread], result_seq[thread] - result_par[thread] );
  }
printf ( "\n" );
printf ( "  Efficiency check:\n" );
printf ( "\n" );
printf ( "  Computing values in parallel should be faster:\n" );
printf ( "\n" );
printf ( "               Sequential      Parallel\n" );
printf ( "\n" );
printf ( "TIME:  %14f  %14f\n", wtime_seq, wtime_par  );
printf ( "RATE:  %14f  %14f\n", mega_rate_seq, mega_rate_par );
/*
  Free memory.
*/
  free ( result_par );
  free ( result_seq );
  free ( seed );
  return;
}
/****************************************************************/
void test02 ( )
/****************************************************************/
/*
  Purpose:
    TEST02 tests R4_UNI.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    04 October 2013

  Author:
    John Burkardt
*/
{
  uint32_t jsr;
```

```
  uint32_t jsr_value;
  double mega_rate_par;
  double mega_rate_seq;
  int r;
  int r_num = 1000;
  float r4_value;
  float *result_par;
  float *result_seq;
  int s;
  int s_num = 10000;
  uint32_t *seed;
  int thread;
  int thread_num;
  double wtime_par;
  double wtime_seq;

 printf ( "\n" );
 printf ( "TEST02\n" );
 printf ( "  R4_UNI computes uniformly random single precision real
values.\n" );
 printf ( "  Since the output is completely determined\n" );
 printf ( "  by the input value of SEED, we can run in\n" );
 printf ( "  parallel as long as we make an array of seeds.\n" );
/*
  Set up the SEED array, which will be used for both sequential and
  parallel computations.
*/
{
  {
    thread_num = omp_get_num_threads ( );
    printf ( "\n" );
  }
}
  Seed = ( uint32_t * ) malloc ( thread_num * sizeof ( uint32_t ) );
  result_seq = ( float * ) malloc ( thread_num * sizeof ( float ) );
  result_par = ( float * ) malloc ( thread_num * sizeof ( float ) );
/*
  Sequential execution.
  The sequential execution will only match the parallel execution if we
can
  guarantee that the parallel threads are scheduled to execute the R loop
  consecutively.
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_seq = omp_get_wtime ( );
  for ( r = 0; r < r_num; r++ )
  {
    thread = ( r % thread_num );
    jsr = seed[thread];
    for ( s = 0; s < s_num; s++ )
    {
      r4_value = r4_uni ( &jsr );
    }
    result_seq[thread] = r4_value;
    seed[thread] = jsr;
```

```
  }
  wtime_seq = omp_get_wtime ( ) - wtime_seq;
  mega_rate_seq = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_seq
/ 1000000.0;
/*
  Parallel.
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_par = omp_get_wtime ( );
  shared ( result_par, seed ) \
  private ( jsr, r, r4_value, s, thread )
  {
    for ( r = 0; r < r_num; r++ )
    {
      thread = omp_get_thread_num ( );
      jsr = seed[thread];
      for ( s = 0; s < s_num; s++ )
      {
        r4_value = r4_uni ( &jsr );
      }
      result_par[thread] = r4_value;
      seed[thread] = jsr;
    }
  }

  wtime_par = omp_get_wtime ( ) - wtime_par;
  mega_rate_par = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_par
/ 1000000.0;
/*
  Report.
*/
  printf ( "\n" );
  printf ( "  Correctness check:\n" );
  printf ( "\n" );
printf ( "  Computing values sequentially should reach the\n" );
printf ( "  same result as doing it in parallel:\n" );
printf ( "\n" );
printf ( "    THREAD    Sequential Parallel Difference\n" );
printf ( "\n" );

  for ( thread = 0; thread < thread_num; thread++ )
  {
   printf ( "  %8d  %14f  %14f  %14f\n", thread,
result_seq[thread],result_par[thread], result_seq[thread] -
result_par[thread] );
  }

printf ( "\n" );
printf ( "  Efficiency check:\n" );
printf ( "\n" );
printf ( "Computing values in parallel should be faster:'\n" );
printf ( "\n" );
printf ( "              Sequential      Parallel\n" );
printf ( "\n" );
printf ( " TIME:  %14f  %14f\n", wtime_seq, wtime_par  );
```

```
printf ( " RATE:  %14f  %14f\n", mega_rate_seq, mega_rate_par );
/*
  Free memory.
*/
  free ( result_par );
  free ( result_seq );
  free ( seed );

  return;
}
/******************************************************/
void test03 ( )
/******************************************************/
/*
  Purpose:
    TEST03 tests R4_NOR.

  Discussion:
    The arrays FN, KN and WN, once set up by R4_NOR_SETUP, are "read
only" when
    accessed by R4_NOR.  So we only need to have one copy of these
arrays,
    and they can be shared.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    04 October 2013

  Author:
    John Burkardt
*/
{
  float fn[128];
  uint32_t jsr;
  uint32_t jsr_value;
  uint32_t kn[128];
  double mega_rate_par;
  double mega_rate_seq;
  int r;
  int r_num = 1000;
  float r4_value;
  float *result_par;
  float *result_seq;
  int s;
  int s_num = 10000;
  uint32_t *seed;
  int thread;
  int thread_num;
  float wn[128];
  double wtime_par;
  double wtime_seq;

  printf ( "\n" );
  printf ( "TEST03\n" );
  printf ( "  R4_NOR computes normal random single precision real
values.\n" );
  printf ( "  Since the output is completely determined\n" );
```

```
  printf ( "  by the input value of SEED and the tables, we can run
in\n" );
  printf ( "  parallel as long as we make an array of seeds and share the
tables.\n" );
/*
  Set up the SEED array and the tables, which will be used for both
sequential
  and parallel computations.
*/
{
  {
    thread_num = omp_get_num_threads ( );
    printf ( "\n" );
    printf ( "  The number of threads is %d\n", thread_num );
  }
}
  seed = ( uint32_t * ) malloc ( thread_num * sizeof ( uint32_t ) );
  result_seq = ( float * ) malloc ( thread_num * sizeof ( float ) );
  result_par = ( float * ) malloc ( thread_num * sizeof ( float ) );

  r4_nor_setup ( kn, fn, wn );
/*
  Sequential execution.
  The sequential execution will only match the parallel execution if we
can
  guarantee that the parallel threads are scheduled to execute the R loop
  consecutively.
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_seq = omp_get_wtime ( );
  for ( r = 0; r < r_num; r++ )
  {
    thread = ( r % thread_num );
    jsr = seed[thread];
    for ( s = 0; s < s_num; s++ )
    {
      r4_value = r4_nor ( &jsr, kn, fn, wn );
    }
    result_seq[thread] = r4_value;
    seed[thread] = jsr;
  }
  wtime_seq = omp_get_wtime ( ) - wtime_seq;
  mega_rate_seq = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_seq
/ 1000000.0;
/*
  Parallel.
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_par = omp_get_wtime ( );
  shared ( result_par, seed ) \
  private ( jsr, r, r4_value, s, thread )
```

```
  {
    for ( r = 0; r < r_num; r++ )
    {
      thread = omp_get_thread_num ( );
      jsr = seed[thread];
      for ( s = 0; s < s_num; s++ )
      {
        r4_value = r4_nor ( &jsr, kn, fn, wn );
      }
      result_par[thread] = r4_value;
      seed[thread] = jsr;
    }
  }
  wtime_par = omp_get_wtime ( ) - wtime_par;
  mega_rate_par = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_par
/ 1000000.0;
/*
  Report.
*/
printf ( "\n" );
printf ( "  Correctness check:\n" );
printf ( "\n" );
printf ( "  Computing values sequentially should reach the\n" );
printf ( "  same result as doing it in parallel:\n" );
printf ( "\n" );
printf ( " THREAD  Sequential  Parallel      Difference\n" );
printf ( "\n" );
for ( thread = 0; thread < thread_num; thread++ )
  {
printf ( "  %8d  %14f  %14f  %14f\n", thread,
result_seq[thread],result_par[thread], result_seq[thread] -
result_par[thread] );
  }
printf ( "\n" );
printf ( "  Efficiency check:\n" );
printf ( "\n" );
printf ( "  Computing values in parallel should be faster:\n" );
wprintf ( "\n" );
printf ( "              Sequential      Parallel\n" );
printf ( "\n" );
printf ( "      TIME:  %14f  %14f\n", wtime_seq, wtime_par  );
printf ( "      RATE:  %14f  %14f\n", mega_rate_seq, mega_rate_par );
/*
  Free memory.
*/
  free ( result_par );
  free ( result_seq );
  free ( seed );
  return;
}
/****************************************************************/
void test04 ( )
/****************************************************************/
/*
  Purpose:
    TEST04 tests R4_EXP.

  Discussion:
```

```
    The arrays FE, KE and WE, once set up by R4_EXP_SETUP, are   "read
only" when accessed by R4_EXP.  So we only need to have one copy of these
arrays, and they can be shared.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    19 October 2013

  Author:
    John Burkardt
*/
{
  float fe[256];
  uint32_t jsr;
  uint32_t jsr_value;
  uint32_t ke[256];
  double mega_rate_par;
  double mega_rate_seq;
  int r;
  int r_num = 1000;
  float r4_value;
  float *result_par;
  float *result_seq;
  int s;
  int s_num = 10000;
  uint32_t *seed;
  int thread;
  int thread_num;
  float we[256];
  double wtime_par;
  double wtime_seq;

  printf ( "\n" );
  printf ( "TEST04\n" );
  printf ( "  R4_EXP computes exponential random single precision real
values.\n" );
  printf ( "  Since the output is completely determined\n" );
  printf ( "  by the input value of SEED and the tables, we can run
in\n" );
  printf ( "  parallel as long as we make an array of seeds and share the
tables.\n" );
/*
  Set up the SEED array and the tables, which will be used for both
sequential
  and parallel computations.
*/
{
  {
    thread_num = omp_get_num_threads ( );
     printf ( "\n" );
     printf ( "  The number of threads is %d\n", thread_num );
  }
}
  seed = ( uint32_t * ) malloc ( thread_num * sizeof ( uint32_t ) );
  result_seq = ( float * ) malloc ( thread_num * sizeof ( float ) );
  result_par = ( float * ) malloc ( thread_num * sizeof ( float ) );
```

```
  r4_exp_setup ( ke, fe, we );
/*
  Sequential execution.
  The sequential execution will only match the parallel execution if we
can
  guarantee that the parallel threads are scheduled to execute the R loop
  consecutively.
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_seq = omp_get_wtime ( );
  for ( r = 0; r < r_num; r++ )
  {
    thread = ( r % thread_num );
    jsr = seed[thread];
    for ( s = 0; s < s_num; s++ )
    {
      r4_value = r4_exp ( &jsr, ke, fe, we );
    }
    result_seq[thread] = r4_value;
    seed[thread] = jsr;
  }
  wtime_seq = omp_get_wtime ( ) - wtime_seq;
  mega_rate_seq = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_seq
    / 1000000.0;
/*
  Parallel.
*/
  jsr = 123456789;
  for ( thread = 0; thread < thread_num; thread++ )
  {
    seed[thread] = shr3_seeded ( &jsr );
  }
  wtime_par = omp_get_wtime ( );
  shared ( result_par, seed ) \
  private ( jsr, r, r4_value, s, thread )
  {
    for ( r = 0; r < r_num; r++ )
    {
      thread = omp_get_thread_num ( );
      jsr = seed[thread];
      for ( s = 0; s < s_num; s++ )
      {
        r4_value = r4_exp ( &jsr, ke, fe, we );
      }
      result_par[thread] = r4_value;
      seed[thread] = jsr;
    }
  }
  wtime_par = omp_get_wtime ( ) - wtime_par;
  mega_rate_par = ( double ) ( r_num ) * ( double ) ( s_num ) / wtime_par
    / 1000000.0;
/*
  Report.
*/
  printf ( "\n" );
```

```
  printf ( "  Correctness check:\n" );
  printf ( "\n" );
  printf ( "  Computing values sequentially should reach the\n" );
  printf ( "  same result as doing it in parallel:\n" );
  printf ( "\n" );
  printf ( "    THREAD    Sequential        Parallel
Difference\n" );
  printf ( "\n" );

  for ( thread = 0; thread < thread_num; thread++ )
  {
    printf ( "  %8d  %14f  %14f  %14f\n", thread, result_seq[thread],
result_par[thread], result_seq[thread] - result_par[thread] );
  }

  printf ( "\n" );
  printf ( "  Efficiency check:\n" );
  printf ( "\n" );
  printf ( "  Computing values in parallel should be faster:\n" );
  printf ( "\n" );
  printf ( "              Sequential      Parallel\n" );
  printf ( "\n" );
  printf ( "      TIME:  %14f  %14f\n", wtime_seq, wtime_par  );
  printf ( "      RATE:  %14f  %14f\n", mega_rate_seq, mega_rate_par );
/*
  Free memory.
*/
  free ( result_par );
  free ( result_seq );
  free ( seed );
  return;
}
/****************************************************************/
float r4_exp ( uint32_t *jsr, uint32_t ke[256], float fe[256], float
we[256] )
/****************************************************/
/*
  Purpose:
    R4_EXP returns an exponentially distributed single precision real
value.

  Discussion:
    The underlying algorithm is the ziggurat method.
    Before the first call to this function, the user must call
R4_EXP_SETUP
    to determine the values of KE, FE and WE.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    15 October 2013

  Author:
    John Burkardt

  Reference:
    George Marsaglia, Wai Wan Tsang,
    The Ziggurat Method for Generating Random Variables,
```

```
    Journal of Statistical Software,
    Volume 5, Number 8, October 2000, seven pages.

  Parameters:
    Input/output, uint32_t *JSR, the seed.
    Input, uint32_t KE[256], data computed by R4_EXP_SETUP.
    Input, float FE[256], WE[256], data computed by R4_EXP_SETUP.
    Output, float R4_EXP, an exponentially distributed random value.
*/
{
  uint32_t iz;
  uint32_t jz;
  float value;
  float x;
  jz = shr3_seeded ( jsr );
  iz = ( jz & 255 );
  if ( jz < ke[iz] )
  {
    value = ( float ) ( jz ) * we[iz];
  }
  else
  {
    for ( ; ; )
    {
      if ( iz == 0 )
      {
        value = 7.69711 - log ( r4_uni ( jsr ) );
        break;
      }
      x = ( float ) ( jz ) * we[iz];

      if ( fe[iz] + r4_uni ( jsr ) * ( fe[iz-1] - fe[iz] ) < exp ( -
x ) )
      {
        value = x;
        break;
      }
      jz = shr3_seeded ( jsr );
      iz = ( jz & 255 );
      if ( jz < ke[iz] )
      {
        value = ( float ) ( jz ) * we[iz];
        break;
      }
    }
  }
  return value;
}
/**************************************************/
void r4_exp_setup ( uint32_t ke[256], float fe[256], float we[256] )
/**************************************************/
/*
  Purpose:
    R4_EXP_SETUP sets data needed by R4_EXP.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
```

```
    14 October 2013

  Author:
    John Burkardt

  Reference:
    George Marsaglia, Wai Wan Tsang,
    The Ziggurat Method for Generating Random Variables,
    Journal of Statistical Software,
    Volume 5, Number 8, October 2000, seven pages.

  Parameters:
    Output, uint32_t KE[256], data needed by R4_EXP.
    Output, float FE[256], WE[256], data needed by R4_EXP.
*/
{
  double de = 7.697117470131487;
  int i;
  const double m2 = 2147483648.0;
  double q;
  double te = 7.697117470131487;
  const double ve = 3.949659822581572E-03;
  q = ve / exp ( - de );
  ke[0] = ( uint32_t ) ( ( de / q ) * m2 );
  ke[1] = 0;
  we[0] = ( float ) ( q / m2 );
  we[255] = ( float ) ( de / m2 );
  fe[0] = 1.0;
  fe[255] = ( float ) ( exp ( - de ) );
  for ( i = 254; 1 <= i; i-- )
  {
    de = - log ( ve / de + exp ( - de ) );
    ke[i+1] = ( uint32_t ) ( ( de / te ) * m2 );
    te = de;
    fe[i] = ( float ) ( exp ( - de ) );
    we[i] = ( float ) ( de / m2 );
  }
  return;
}
/**************************************************/
float r4_nor ( uint32_t *jsr, uint32_t kn[128], float fn[128], float
wn[128] )
/**************************************************/
/*
  Purpose:
    R4_NOR returns a normally distributed single precision real value.

  Discussion:
    The value returned is generated from a distribution with mean 0 and
    variance 1.
    The underlying algorithm is the ziggurat method.
    Before the first call to this function, the user must call
R4_NOR_SETUP
    to determine the values of KN, FN and WN.
    Thanks to Chad Wagner, 21 July 2014, for noticing a bug of the form
      if ( x * x <= y * y );   <-- Stray semicolon!
      {
        break;
      }
```

```
  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 July 2014

  Author:
    John Burkardt

  Reference:
    George Marsaglia, Wai Wan Tsang,
    The Ziggurat Method for Generating Random Variables,
    Journal of Statistical Software,
    Volume 5, Number 8, October 2000, seven pages.

  Parameters:
    Input/output, uint32_t *JSR, the seed.
    Input, uint32_t KN[128], data computed by R4_NOR_SETUP.
    Input, float FN[128], WN[128], data computed by R4_NOR_SETUP.
    Output, float R4_NOR, a normally distributed random value.
*/
{
  int hz;
  uint32_t iz;
  const float r = 3.442620;
  float value;
  float x;
  float y;
  hz = ( int ) shr3_seeded ( jsr );
  iz = ( hz & 127 );
  if ( fabs ( hz ) < kn[iz] )
  {
    value = ( float ) ( hz ) * wn[iz];
  }
  else
  {
    for ( ; ; )
    {
      if ( iz == 0 )
      {
        for ( ; ; )
        {
          x = - 0.2904764 * log ( r4_uni ( jsr ) );
          y = - log ( r4_uni ( jsr ) );
          if ( x * x <= y + y )
          {
            break;
          }
        }
        if ( hz <= 0 )
        {
          value = - r - x;
        }
        else
        {
          value = + r + x;
        }
        break;
```

```
      }
      x = ( float ) ( hz ) * wn[iz];

      if ( fn[iz] + r4_uni ( jsr ) * ( fn[iz-1] - fn[iz] )
        < exp ( - 0.5 * x * x ) )
      {
        value = x;
        break;
      }
      hz = ( int ) shr3_seeded ( jsr );
      iz = ( hz & 127 );
      if ( fabs ( hz ) < kn[iz] )
     {
        value = ( float ) ( hz ) * wn[iz];
        break;
      }
    }
  }
  return value;
}
/*****************************************************/
void r4_nor_setup ( uint32_t kn[128], float fn[128], float wn[128] )
/*****************************************************/
/*
  Purpose:
    R4_NOR_SETUP sets data needed by R4_NOR.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    14 October 2013

  Author:
    John Burkardt

  Reference:
    George Marsaglia, Wai Wan Tsang,
    The Ziggurat Method for Generating Random Variables,
    Journal of Statistical Software,
    Volume 5, Number 8, October 2000, seven pages.

  Parameters:
    Output, uint32_t KN[128], data needed by R4_NOR.
    Output, float FN[128], WN[128], data needed by R4_NOR.
*/
{
  double dn = 3.442619855899;
  int i;
  const double m1 = 2147483648.0;
  double q;
  double tn = 3.442619855899;
  const double vn = 9.91256303526217E-03;
  q = vn / exp ( - 0.5 * dn * dn );
  kn[0] = ( uint32_t ) ( ( dn / q ) * m1 );
  kn[1] = 0;
  wn[0] = ( float ) ( q / m1 );
  wn[127] = ( float ) ( dn / m1 );
  fn[0] = 1.0;
```

```
  fn[127] = ( float ) ( exp ( - 0.5 * dn * dn ) );
  for ( i = 126; 1 <= i; i-- )
  {
    dn = sqrt ( - 2.0 * log ( vn / dn + exp ( - 0.5 * dn * dn ) ) );
    kn[i+1] = ( uint32_t ) ( ( dn / tn ) * m1 );
    tn = dn;
    fn[i] = ( float ) ( exp ( - 0.5 * dn * dn ) );
    wn[i] = ( float ) ( dn / m1 );
  }
  return;
}
/****************************************************************/
float r4_uni ( uint32_t *jsr )
/****************************************************************/
/*
  Purpose:
    R4_UNI returns a uniformly distributed real value.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    04 October 2013

  Author:
    John Burkardt

  Reference:
    George Marsaglia, Wai Wan Tsang,
    The Ziggurat Method for Generating Random Variables,
    Journal of Statistical Software,
    Volume 5, Number 8, October 2000, seven pages.

  Parameters:
    Input/output, uint32_t *JSR, the seed.
    Output, float R4_UNI, a uniformly distributed random value in
    the range [0,1].
*/
{
  uint32_t jsr_input;
  float value;
  jsr_input = *jsr;
  *jsr = ( *jsr ^ ( *jsr <<   13 ) );
  *jsr = ( *jsr ^ ( *jsr >>   17 ) );
  *jsr = ( *jsr ^ ( *jsr <<    5 ) );
  value = fmod ( 0.5 + ( float ) ( jsr_input + *jsr ) / 65536.0 /
65536.0, 1.0 );
  return value;
}
/****************************************/
uint32_t shr3_seeded ( uint32_t *jsr )
/****************************************/
/*
  Purpose:
    SHR3_SEEDED evaluates the SHR3 generator for integers.

  Discussion:
    Thanks to Dirk Eddelbuettel for pointing out that this code needed to
```

```
    use the uint32_t data type in order to execute properly in 64 bit
mode,
    03 October 2013.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    04 October 2013

  Author:
    John Burkardt

  Reference:
    George Marsaglia, Wai Wan Tsang,
    The Ziggurat Method for Generating Random Variables,
    Journal of Statistical Software,
    Volume 5, Number 8, October 2000, seven pages.

  Parameters:
    Input/output, uint32_t *JSR, the seed, which is updated
    on each call.
    Output, uint32_t SHR3_SEEDED, the new value.
*/
{
  uint32_t value;
  value = *jsr;
  *jsr = ( *jsr ^ ( *jsr <<   13 ) );
  *jsr = ( *jsr ^ ( *jsr >>   17 ) );
  *jsr = ( *jsr ^ ( *jsr <<    5 ) );
  value = value + *jsr;
  return value;
}
/******************************************************************************
******/
void timestamp ( void )
/******************************************************************************
******/
/*
  Purpose:
    TIMESTAMP prints the current YMDHMS date as a time stamp.

  Example:
    31 May 2001 09:45:54 AM

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    24 September 2003

  Author:
    John Burkardt

  Parameters:
    None
*/
{
# define TIME_SIZE 40
```

```
  static char time_buffer[TIME_SIZE];
  const struct tm *tm;
  size_t len;
  time_t now;
  now = time ( NULL );
  tm = localtime ( &now );
  len = strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );
 printf ( "%s\n", time_buffer );
  return;
# undef TIME_SIZE
}
```

12.4. Satisfy

```
# include <stdlib.h>
# include <stdio.h>
# include <time.h>

int main(int argc, char ** argv);
int circuit_value ( int n, int bvec[] );
void i4_to_bvec ( int i4, int n, int bvec[] );
void timestamp ( void );

/******************************************/
int main(int argc, char ** argv)
/******************************************/
/*
  Purpose:
    MAIN is the main program for SATISFY_OPENMP.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    24 March 2009

  Author:
    John Burkardt

  Reference:
    Michael Quinn,
    Parallel Programming in C with MPI and OpenMP,
    McGraw-Hill, 2004,
    ISBN13: 978-0071232654,
    LC: QA76.73.C15.Q55.
*/
{
# define N 23

  int bvec[N];
  int i;
  int id;
  int ihi;
  int ihi2;
  int ilo;
  int ilo2;
  int j;
  int n = N;
```

```
  int proc_num;
  int solution_num;
  int solution_num_local;
  int thread_num;
  int value;
  double wtime;
  printf ( "\n" );
  timestamp ( );
  printf ( "\n" );
  printf ( "SATISFY_SECUENCIAL\n" );
  printf ( "  C + OpenMP version\n" );
  printf ( "  We have a logical function of N logical arguments.\n" );
  printf ( "  We do an exhaustive search of all 2^N possibilities,\n" );
  printf ( "  seeking those inputs that make the function TRUE.\n" );
  printf ( "\n" );
 /*
  Compute the number of binary vectors to check.
*/
  ilo = 0;
  ihi = 1;
  for ( i = 1; i <= n; i++ )
  {
    ihi = ihi * 2;
  }
  printf ( "\n" );
  printf ( "  The number of logical variables is N = %d\n", n );
  printf ( "  The number of input vectors to check is %d\n", ihi );
  printf ( "\n" );
  printf ( "   # Processor       Index    ---------Input Values----------
--------------\n" );
  printf ( "\n" );
/*
  Processor ID takes the interval ILO2 <= I < IHI2.
  Using the formulas below yields a set of nonintersecting intervals
  which cover the original interval [ILO,IHI).
*/
  thread_num = omp_get_max_threads ( );
  solution_num = 0;
  wtime = omp_get_wtime ( );
  shared ( ihi, ilo, n, thread_num ) \
  private ( bvec, i, id, ihi2, ilo2, j, solution_num_local, value ) \
  reduction ( + : solution_num )
  {
    id = omp_get_thread_num ( );

    ilo2 = ( ( thread_num - id     ) * ilo
           + (              id     ) * ihi )
           / ( thread_num          );

    ihi2 = ( ( thread_num - id - 1 ) * ilo
           + (              id + 1 ) * ihi )
           / ( thread_num          );

    printf ( "\n" );
    printf ( "  Processor %8d iterates from %8d <= I < %8d.\n", id, ilo2,
ihi2 );
    printf ( "\n" );
/*
  Check every possible input vector.
```

```
*/
    solution_num_local = 0;
    for ( i = ilo2; i < ihi2; i++ )
    {
      i4_to_bvec ( i, n, bvec );

      value = circuit_value ( n, bvec );

      if ( value == 1 )
      {
        solution_num_local = solution_num_local + 1;
    printf ( "  %2d  %8d  %10d:  ", solution_num_local, id, i );
        for ( j = 0; j < n; j++ )
        {
          printf ( " %d", bvec[j] );
        }
        printf ( "\n" );
      }
    }
    solution_num = solution_num + solution_num_local;
  }
  wtime = omp_get_wtime ( ) - wtime;
printf ( "\n" );
printf ( "  Number of solutions found was %d\n", solution_num );
printf ( " %f\n", wtime );
/*
  Terminate.
*/
printf ( "\n" );
printf ( "SATISFY_SECUENCIAL\n" );
printf ( "  Normal end of execution.\n" );
printf ( "\n" );
timestamp ( );
return 0;
# undef N
}
/******************************************************/
int circuit_value ( int n, int bvec[] )
/******************************************************/
/*
  Purpose:
    CIRCUIT_VALUE returns the value of a circuit for a given input set.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    20 March 2009

  Author:
    John Burkardt

  Reference:
    Michael Quinn,
    Parallel Programming in C with MPI and OpenMP,
    McGraw-Hill, 2004,
    ISBN13: 978-0071232654,
    LC: QA76.73.C15.Q55.
```

```
  Parameters:
    Input, int N, the length of the input vector.
    Input, int BVEC[N], the binary inputs.
    Output, int CIRCUIT_VALUE, the output of the circuit.
*/
{
  int value;
  value =
       (  bvec[0]  ||  bvec[1]  )
    && ( !bvec[1]  || !bvec[3]  )
    && (  bvec[2]  ||  bvec[3]  )
    && ( !bvec[3]  || !bvec[4]  )
    && (  bvec[4]  || !bvec[5]  )
    && (  bvec[5]  || !bvec[6]  )
    && (  bvec[5]  ||  bvec[6]  )
    && (  bvec[6]  || !bvec[15] )
    && (  bvec[7]  || !bvec[8]  )
    && ( !bvec[7]  || !bvec[13] )
    && (  bvec[8]  ||  bvec[9]  )
    && (  bvec[8]  || !bvec[9]  )
    && ( !bvec[9]  || !bvec[10] )
    && (  bvec[9]  ||  bvec[11] )
    && (  bvec[10] ||  bvec[11] )
    && (  bvec[12] ||  bvec[13] )
    && (  bvec[13] || !bvec[14] )
    && (  bvec[14] ||  bvec[15] )
    && (  bvec[14] ||  bvec[16] )
    && (  bvec[17] ||  bvec[1]  )
    && (  bvec[18] || !bvec[0]  )
    && (  bvec[19] ||  bvec[1]  )
    && (  bvec[19] || !bvec[18] )
    && ( !bvec[19] || !bvec[9]  )
    && (  bvec[0]  ||  bvec[17] )
    && ( !bvec[1]  ||  bvec[20] )
    && ( !bvec[21] ||  bvec[20] )
    && ( !bvec[22] ||  bvec[20] )
    && ( !bvec[21] || !bvec[20] )
    && (  bvec[22] || !bvec[20] );

  return value;
}
/**************************************************/
void i4_to_bvec ( int i4, int n, int bvec[] )
/**************************************************/
/*
  Purpose:
    I4_TO_BVEC converts an integer into a binary vector.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    20 March 2009

  Author:
    John Burkardt

  Parameters:
    Input, int I4, the integer.
```

```
    Input, int N, the dimension of the vector.
    Output, int BVEC[N], the vector of binary remainders.
*/
{
  int i;

  for ( i = n - 1; 0 <= i; i-- )
  {
    bvec[i] = i4 % 2;
    i4 = i4 / 2;
  }

  return;
}
/**************************************************/
void timestamp ( void )
/**************************************************/
/*
  Purpose:
    TIMESTAMP prints the current YMDHMS date as a time stamp.

  Example:
    31 May 2001 09:45:54 AM

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    24 September 2003

  Author:
    John Burkardt

  Parameters:
    None
*/
{
# define TIME_SIZE 40
  static char time_buffer[TIME_SIZE];
  const struct tm *tm;
  size_t len;
  time_t now;
  now = time ( NULL );
  tm = localtime ( &now );
len = strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );

 printf ( "%s\n", time_buffer );

  return;
# undef TIME_SIZE
}
```

12.5. Quad

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <time.h>
```

```
int main(int argc, char ** argv);
double f ( double x );
double cpu_time ( void );
void timestamp ( void );

/************************/
int main(int argc, char ** argv)
/************************/
/*
  Purpose:
    MAIN is the main program for QUAD_OPENMP.
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    14 December 2011
  Author:
    John Burkardt
*/
{
  double a = 0.0;
  double b = 10.0;
  double error;
  double exact = 0.49936338107645674464;
  int i;
  int n = 10000000;
  double total;
  double wtime;
  double x;

  timestamp ( );
  printf ( "\n" );
  printf ( "QUAD_SECUENCIAL:\n" );
  printf ( "  C version\n" );
  printf ( "  Use OpenMP for parallel execution.\n" );
  printf ( "  Estimate the integral of f(x) from A to B.\n" );
  printf ( "  f(x) = 50 / ( pi * ( 2500 * x * x + 1 ) ).\n" );
  printf ( "\n" );
  printf ( "  A        = %f\n", a );
  printf ( "  B        = %f\n", b );
  printf ( "  N        = %d\n", n );
  printf ( "  Exact    = %24.16f\n", exact );

  wtime = omp_get_wtime ( );
  total = 0.0;
  for ( i = 0; i < n; i++ )
  {
 x =((double )( n - i - 1 ) * a + ( double ) ( i ) * b ) / ( double ) ( n
- 1 );
  total = total + f ( x );
  }
  wtime = omp_get_wtime ( ) - wtime;
  total = ( b - a ) * total / ( double ) n;
  error = fabs ( total - exact );
  printf ( "\n" );
  printf ( "  Estimate = %24.16f\n", total );
  printf ( "  Error    = %e\n", error );
  printf ( " %f\n", wtime );
/*
  Terminate.
```

```
*/
  printf ( "\n" );
  printf ( "QUAD_SECUENCIAL:\n" );
  printf ( "  Normal end of execution.\n" );
  printf ( "\n" );
  timestamp ( );
  return 0;
}
/******************************************************/
double f ( double x )
/******************************************************/
/*
  Purpose:
    F evaluates the function.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    18 July 2010

  Author:
    John Burkardt

  Parameters:
    Input, double X, the argument.
    Output, double F, the value of the function.
*/
{
  double pi = 3.141592653589793;
  double value;
  value = 50.0 / ( pi * ( 2500.0 * x * x + 1.0 ) );
  return value;
}
/******************************************************/
double cpu_time ( void )
/******************************************************/
/*
  Purpose:
   CPU_TIME reports the total CPU time for a program.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    27 September 2005

  Author:
    John Burkardt

  Parameters:
    Output, double CPU_TIME, the current total elapsed CPU time in
second.
*/
{
  double value;
  value = ( double ) clock ( ) / ( double ) CLOCKS_PER_SEC;
  return value;
}
```

```
/******************************************************/
void timestamp ( void )
/******************************************************/
/*
  Purpose:
    TIMESTAMP prints the current YMDHMS date as a time stamp.
  Example:
    31 May 2001 09:45:54 AM
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    24 September 2003
  Author:
    John Burkardt
  Parameters:
    None
*/
{
# define TIME_SIZE 40
static char time_buffer[TIME_SIZE];
const struct tm *tm;
time_t now;
now = time ( NULL );
tm = localtime ( &now );
strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );
printf ( "%s\n", time_buffer );
return;
# undef TIME_SIZE
}
```

12.6. Schedule

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

int main(int argc, char ** argv);
int prime_default ( int n );
int prime_static ( int n );
int prime_dynamic ( int n );
/******************************************************/
int main(int argc, char ** argv)
/******************************************************/
/*
  Purpose:
    MAIN is the main program for SCHEDULE_OPENMP.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    10 July 2010

  Author:
    John Burkardt
*/
{
  int n;
  int n_factor;
  int n_hi;
```

```
  int n_lo;
  int primes;
  double time1;
  double time2;
  double time3;
  double wtime;

  printf ( "\n" );
  printf ( "SCHEDULE_SECUENCIAL\n" );
  printf ( "  C/OpenMP version\n" );
  printf ( "  Count the primes from 1 to N.\n" );
  printf ( "  This is an unbalanced work load, particular for two
threads.\n" );
  printf ( "  Demonstrate default, static and dynamic scheduling.\n" );
  printf ( "\n" );

  n_lo = 1;
  n_hi = 131072;
  n_factor = 2;

  printf ( "\n" );
  printf ( " Default        Static       Dynamic\n" );
  printf ( "N     Pi(N)          Time          Time          Time\n" );
  printf ( "\n" );

  n = n_lo;
 wtime = omp_get_wtime ( );
  while ( n <= n_hi )
  {
    time1 = omp_get_wtime ( );
    primes = prime_default ( n );
    time1 = omp_get_wtime ( ) - time1;
    time2 = omp_get_wtime ( );
    primes = prime_static ( n );
    time2 = omp_get_wtime ( ) - time2;
    time3 = omp_get_wtime ( );
    primes = prime_dynamic ( n );
    time3 = omp_get_wtime ( ) - time3;

    printf ( "  %8d  %8d  %12f  %12f  %12f\n", n, primes, time1, time2,
time3 );
    n = n * n_factor;
  }
     wtime = omp_get_wtime ( ) - wtime;
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "SCHEDULE_SECUENCIAL\n" );
  printf ( "%f\n", wtime );
  printf ( "  Normal end of execution.\n" );
  return 0;
}
/******************************************************/
int prime_default ( int n )
/******************************************************/
/*
  Purpose:
    PRIME_DEFAULT counts primes, using default scheduling.
```

```
  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    10 July 2010

  Author:
    John Burkardt

  Parameters:
    Input, int N, the maximum number to check.
  Output, int PRIME_DEFAULT, the number of prime numbers up to N.
*/
{
  int i;
  int j;
  int prime;
  int total = 0;
  shared ( n ) \
  private ( i, j, prime )
  for ( i = 2; i <= n; i++ )
  {
    prime = 1;
    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )
      {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
  return total;
}
/**************************************************/
int prime_static ( int n )
/**************************************************************/
/*
  Purpose:
    PRIME_STATIC counts primes using static scheduling.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    10 July 2010

  Author:
    John Burkardt

  Parameters:
    Input, int N, the maximum number to check.
    Output, int PRIME_STATIC, the number of prime numbers up to N.
*/
{
  int i;
  int j;
```

```
  int prime;
  int total = 0;
  shared ( n ) \
  private ( i, j, prime )
  for ( i = 2; i <= n; i++ )
  {
    prime = 1;
    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )
      {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
  return total;
}
/**********************************************************/
int prime_dynamic ( int n )
/**********************************************/
/*
  Purpose:
    PRIME_DYNAMIC counts primes using dynamic scheduling.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    10 July 2010

  Author:
    John Burkardt

  Parameters:
    Input, int N, the maximum number to check.
    Output, int PRIME_DYNAMIC, the number of prime numbers up to N.
*/
{
  int i;
  int j;
  int prime;
  int total = 0;
  shared ( n ) \
  private ( i, j, prime )
  for ( i = 2; i <= n; i++ )
  {
    prime = 1;

    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )
      {
        prime = 0;
        break;
      }
    }
    total = total + prime;
```

```
  }
  return total;
}
```

12.7. Helmholtz

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>

int main(int argc, char ** argv);
void driver ( int m, int n, int it_max, double alpha, double omega,
double tol );
void error_check ( int m, int n, double alpha, double u[], double f[] );
void jacobi ( int m, int n, double alpha, double omega, double u[],
double f[],
double tol, int it_max );
double *rhs_set ( int m, int n, double alpha );
double u_exact ( double x, double y );
double uxx_exact ( double x, double y );
double uyy_exact ( double x, double y );
/**************************/
int main(int argc, char ** argv)
/*****************************************************/
/*
  Purpose:
    MAIN is the main program for HELMHOLTZ.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    19 April 2009

  Author:
    Original FORTRAN77 version by Joseph Robicheaux, Sanjiv Shah.
    C version by John Burkardt
*/
{
  double alpha = 0.25;
  int it_max = 100;
  int m = 500;
  int n = 500;
  double omega = 1.1;
  double tol = 1.0E-08;
  double wtime;
  printf ( "\n" );
  printf ( "HELMHOLTZ\n" );
  printf ( "  C/OpenMP version\n" );
  printf ( "\n" );
  printf ( "  A program which solves the 2D Helmholtz equation.\n" );
  printf ( "\n" );
  printf ( "  This program is being run in parallel.\n" );
  printf ( "\n" );
  printf ( "\n" );
  printf ( "  The region is [-1,1] x [-1,1].\n" );
  printf ( "  The number of nodes in the X direction is M = %d\n", m );
```

```
  printf ( "  The number of nodes in the Y direction is N = %d\n", n );
  printf ( "  Number of variables in linear system M * N  = %d\n", m *
n );
  printf ( "  The scalar coefficient in the Helmholtz equation is ALPHA
= %f\n", alpha );
  printf ( "  The relaxation value is OMEGA = %f\n", omega );
  printf ( "  The error tolerance is TOL = %f\n", tol );
  printf ( "  The maximum number of Jacobi iterations is IT_MAX = %d\n",
it_max );
/*
  Call the driver routine.
*/
  wtime = omp_get_wtime ( );
  driver ( m, n, it_max, alpha, omega, tol );
  wtime = omp_get_wtime ( ) - wtime;
  printf ( "\n" );
  printf ( "%f\n", wtime );
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "HELMHOLTZ\n" );
  printf ( "  Normal end of execution.\n" );
  return 0;
}
/******************************************************/
void driver ( int m, int n, int it_max, double alpha, double omega,
double tol )
/******************************************************/
/*
  Purpose:
    DRIVER allocates arrays and solves the problem.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    Original FORTRAN77 version by Joseph Robicheaux, Sanjiv Shah.
    C version by John Burkardt

  Parameters:
    Input, int M, N, the number of grid points in the
    X and Y directions.
    Input, int IT_MAX, the maximum number of Jacobi
    iterations allowed.
    Input, double ALPHA, the scalar coefficient in the
    Helmholtz equation.
    Input, double OMEGA, the relaxation parameter, which
    should be strictly between 0 and 2.  For a pure Jacobi method,
    use OMEGA = 1.
    Input, double TOL, an error tolerance for the linear
    equation solver.
*/
{
  double *f;
  int i;
```

```
  int j;
  double *u;
/*
  Initialize the data.
*/
  f = rhs_set ( m, n, alpha );
  u = ( double * ) malloc ( m * n * sizeof ( double ) );
  shared ( m, n, u ) \
  private ( i, j )
  for ( j = 0; j < n; j++ )
  {
    for ( i = 0; i < m; i++ )
    {
      u[i+j*m] = 0.0;
    }
  }
/*
  Solve the Helmholtz equation.
*/
  jacobi ( m, n, alpha, omega, u, f, tol, it_max );
/*
  Determine the error.
*/
  error_check ( m, n, alpha, u, f );
  free ( f );
  free ( u );
  return;
}
/******************************************************/
void error_check ( int m, int n, double alpha, double u[], double f[] )
/******************************************************/
/*
  Purpose:
    ERROR_CHECK determines the error in the numerical solution.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    Original FORTRAN77 version by Joseph Robicheaux, Sanjiv Shah.
    C version by John Burkardt

  Parameters:
    Input, int M, N, the number of grid points in the
    X and Y directions.
    Input, double ALPHA, the scalar coefficient in the
    Helmholtz equation.  ALPHA should be positive.
    Input, double U[M*N], the solution of the Helmholtz equation
    at the grid points.
    Input, double F[M*N], values of the right hand side function
    for the Helmholtz equation at the grid points.
*/
{
  double error_norm;
  int i;
  int j;
```

```
  double u_norm;
  double u_true;
  double u_true_norm;
  double x;
  double y;
  u_norm = 0.0;
  shared ( m, n, u ) \
  private ( i, j )
  for ( j = 0; j < n; j++ )
  {
    for ( i = 0; i < m; i++ )
    {
      u_norm = u_norm + u[i+j*m] * u[i+j*m];
    }
  }
  u_norm = sqrt ( u_norm );
  u_true_norm = 0.0;
  error_norm = 0.0;
  shared ( m, n, u ) \
  private ( i, j, u_true, x, y )
  for ( j = 0; j < n; j++ )
  {
    for ( i = 0; i < m; i++ )
    {
      x = ( double ) ( 2 * i - m + 1 ) / ( double ) ( m - 1 );
      y = ( double ) ( 2 * j - n + 1 ) / ( double ) ( n - 1 );
      u_true = u_exact ( x, y );
      error_norm = error_norm + ( u[i+j*m] - u_true ) * ( u[i+j*m] -
u_true );
      u_true_norm = u_true_norm + u_true * u_true;
    }
  }
  error_norm = sqrt ( error_norm );
  u_true_norm = sqrt ( u_true_norm );
  printf ( "\n" );
  printf ( "  Computed U l2 norm :       %f\n", u_norm );
  printf ( "  Computed U_EXACT l2 norm : %f\n", u_true_norm );
  printf ( "  Error l2 norm:             %f\n", error_norm );
  return;
}
/*****************************************************/
void jacobi ( int m, int n, double alpha, double omega, double u[],
double f[],
  double tol, int it_max )
/*****************************************************/
/*
  Purpose:
    JACOBI applies the Jacobi iterative method to solve the linear
system.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    Original FORTRAN77 version by Joseph Robicheaux, Sanjiv Shah.
    C version by John Burkardt
```

```
  Parameters:
    Input, int M, N, the number of grid points in the
    X and Y directions.
    Input, double ALPHA, the scalar coefficient in the
    Helmholtz equation.  ALPHA should be positive.
    Input, double OMEGA, the relaxation parameter, which
    should be strictly between 0 and 2.  For a pure Jacobi method,
    use OMEGA = 1.
    Input/output, double U(M,N), the solution of the Helmholtz
    equation at the grid points.
    Input, double F(M,N), values of the right hand side function
    for the Helmholtz equation at the grid points.
    Input, double TOL, an error tolerance for the linear
    equation solver.
    Input, int IT_MAX, the maximum number of Jacobi
    iterations allowed.
*/
{
  double ax;
  double ay;
  double b;
  double dx;
  double dy;
  double error;
  double error_norm;
  int i;
  int it;
  int j;
  double *u_old;
/*
  Initialize the coefficients.
*/
  dx = 2.0 / ( double ) ( m - 1 );
  dy = 2.0 / ( double ) ( n - 1 );
  ax = - 1.0 / dx / dx;
  ay = - 1.0 / dy / dy;
  b  = + 2.0 / dx / dx + 2.0 / dy / dy + alpha;
  u_old = ( double * ) malloc ( m * n * sizeof ( double ) );
  for ( it = 1; it <= it_max; it++ )
  {
    error_norm = 0.0;
/*
  Copy new solution into old.
*/
  shared ( m, n, u, u_old ) \
  private ( i, j )
    for ( j = 0; j < n; j++ )
    {
      for ( i = 0; i < m; i++ )
      {
        u_old[i+m*j] = u[i+m*j];
      }
    }
/*
  Compute stencil, residual, and update.
*/
  shared ( ax, ay, b, f, m, n, omega, u, u_old ) \
  private ( error, i, j )
```

```
    for ( j = 0; j < n; j++ )
    {
      for ( i = 0; i < m; i++ )
      {
/*
  Evaluate the residual.
*/
        if ( i == 0 || i == m - 1 || j == 0 || j == n - 1 )
        {
          error = u_old[i+j*m] - f[i+j*m];
        }
        else
        {
          error = ( ax * ( u_old[i-1+j*m] + u_old[i+1+j*m] )
            + ay * ( u_old[i+(j-1)*m] + u_old[i+(j+1)*m] )
            + b * u_old[i+j*m] - f[i+j*m] ) / b;
        }
/*
  Update the solution.
*/
        u[i+j*m] = u_old[i+j*m] - omega * error;
/*
  Accumulate the residual error.
*/
        error_norm = error_norm + error * error;
      }
    }
/*
  Error check.
*/
    error_norm = sqrt ( error_norm ) / ( double ) ( m * n );
    printf ( "  %d  Residual RMS %e\n", it, error_norm );
    if ( error_norm <= tol )
    {
      break;
    }
  }
 printf ( "\n" );
 printf ( "  Total number of iterations %d\n", it );
  free ( u_old );
  return;
}
/******************************************************/
double *rhs_set ( int m, int n, double alpha )
/******************************************************/
/*
  Purpose:
    RHS_SET sets the right hand side F(X,Y).

  Discussion:
    The routine assumes that the exact solution and its second
    derivatives are given by the routine EXACT.
    The appropriate Dirichlet boundary conditions are determined
    by getting the value of U returned by EXACT.

    The appropriate right hand side function is determined by
    having EXACT return the values of U, UXX and UYY, and setting
      F = -UXX - UYY + ALPHA * U
```

```
  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    Original FORTRAN77 version by Joseph Robicheaux, Sanjiv Shah.
    C version by John Burkardt

  Parameters:
    Input, int M, N, the number of grid points in the
    X and Y directions.
    Input, double ALPHA, the scalar coefficient in the
    Helmholtz equation.  ALPHA should be positive.
    Output, double RHS[M*N], values of the right hand side function
    for the Helmholtz equation at the grid points.
*/
{
  double *f;
  double f_norm;
  int i;
  int j;
  double x;
  double y;
  f = ( double * ) malloc ( m * n * sizeof ( double ) );
  shared ( f, m, n ) \
  private ( i, j )
  for ( j = 0; j < n; j++ )
  {
    for ( i = 0; i < m; i++ )
    {
      f[i+j*m] = 0.0;
    }
  }
/*
  Set the boundary conditions.
*/
  shared ( alpha, f, m, n ) \
  private ( i, j, x, y )
  {
    for ( i = 0; i < m; i++ )
    {
      j = 0;
      y = ( double ) ( 2 * j - n + 1 ) / ( double ) ( n - 1 );
      x = ( double ) ( 2 * i - m + 1 ) / ( double ) ( m - 1 );
      f[i+j*m] = u_exact ( x, y );
    }
    for ( i = 0; i < m; i++ )
    {
      j = n - 1;
      y = ( double ) ( 2 * j - n + 1 ) / ( double ) ( n - 1 );
      x = ( double ) ( 2 * i - m + 1 ) / ( double ) ( m - 1 );
      f[i+j*m] = u_exact ( x, y );
    }
    for ( j = 0; j < n; j++ )
    {
      i = 0;
      x = ( double ) ( 2 * i - m + 1 ) / ( double ) ( m - 1 );
```

```
      y = ( double ) ( 2 * j - n + 1 ) / ( double ) ( n - 1 );
      f[i+j*m] = u_exact ( x, y );
    }
    for ( j = 0; j < n; j++ )
    {
      i = m - 1;
      x = ( double ) ( 2 * i - m + 1 ) / ( double ) ( m - 1 );
      y = ( double ) ( 2 * j - n + 1 ) / ( double ) ( n - 1 );
      f[i+j*m] = u_exact ( x, y );
    }
/*
  Set the right hand side F.
*/
    for ( j = 1; j < n - 1; j++ )
    {
      for ( i = 1; i < m - 1; i++ )
      {
        x = ( double ) ( 2 * i - m + 1 ) / ( double ) ( m - 1 );
        y = ( double ) ( 2 * j - n + 1 ) / ( double ) ( n - 1 );
        f[i+j*m] = - uxx_exact ( x, y ) - uyy_exact ( x, y ) + alpha *
u_exact ( x, y );
      }
    }
  }
  f_norm = 0.0;
  shared ( f, m, n ) \
  private ( i, j )
  for ( j = 0; j < n; j++ )
  {
    for ( i = 0; i < m; i++ )
    {
      f_norm = f_norm + f[i+j*m] * f[i+j*m];
    }
  }
  f_norm = sqrt ( f_norm );
  printf ( "\n" );
  printf ( "  Right hand side l2 norm = %f\n", f_norm );
  return f;
}
/*****************************************************/
double u_exact ( double x, double y )
/*****************************************************/
/*
  Purpose:
    U_EXACT returns the exact value of U(X,Y).

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    John Burkardt

  Parameters:
    Input, double X, Y, the point at which the values are needed.
    Output, double U_EXACT, the value of the exact solution.
*/
```

```
{
  double value;
  value = ( 1.0 - x * x ) * ( 1.0 - y * y );
  return value;
}
/*******************************************/
double uxx_exact ( double x, double y )
/*******************************************/
/*
  Purpose:
    UXX_EXACT returns the exact second X derivative of the solution.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    John Burkardt

  Parameters:
    Input, double X, Y, the point at which the values are needed.
    Output, double UXX_EXACT, the exact second X derivative.
*/
{
  double value;
  value = -2.0 * ( 1.0 + y ) * ( 1.0 - y );
  return value;
}
/******************/
double uyy_exact ( double x, double y )
/******************/
/*
  Purpose:
    UYY_EXACT returns the exact second Y derivative of the solution.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 November 2007

  Author:
    John Burkardt

  Parameters:
    Input, double X, Y, the point at which the values are needed.
    Output, double UYY_EXACT, the exact second Y derivative.
*/
{
  double value;
  value = -2.0 * ( 1.0 + x ) * ( 1.0 - x );
  return value;
}
```

12.8. Multiplicacion de matrices

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>

int main(int argc, char ** argv);
void r8_mxm ( int l, int m, int n );
double r8_uniform_01 ( int *seed );

/************/
int main(int argc, char ** argv)
/**********************/
/*
  Purpose:
    MAIN is the main program for MXM.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    19 April 2009

  Author:
    John Burkardt
*/
{
  int id;
  int l;
  int m;
  int n;
  printf ( "\n" );
  printf ( "MXM\n" );
  printf ( "  C/Secuencial version.\n" );
  printf ( "\n" );
  printf ( "  Matrix multiplication tests.\n" );
  printf ( "\n" );
  l = 500;
  m = 500;
  n = 500;

  r8_mxm ( l, m, n );
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "MXM:\n" );
  printf ( "  Normal end of execution.\n" );

  return 0;
}
/*************************************************/
void r8_mxm ( int l, int m, int n )
/*****************************************/
/*
  Purpose:
    R8_MXM carries out a matrix-matrix multiplication in R8 arithmetic.

  Discussion:
    A(LxN) = B(LxM) * C(MxN).
```

```
  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    13 February 2008

  Author:
    John Burkardt

  Parameters:
    Input, int L, M, N, the dimensions that specify the sizes of the
    A, B, and C matrices.
*/
{
  double *a;
  double *b;
  double *c;
  int i;
  int j;
  int k;
  int ops;
  double rate;
  int seed;
  double time_begin;
  double time_elapsed;
  double time_stop;
/*
  Allocate the matrices.
*/
  a = ( double * ) malloc ( l * n * sizeof ( double ) );
  b = ( double * ) malloc ( l * m * sizeof ( double ) );
  c = ( double * ) malloc ( m * n * sizeof ( double ) );
/*
  Assign values to the B and C matrices.
*/
  seed = 123456789;
  for ( k = 0; k < l * m; k++ )
  {
    b[k] = r8_uniform_01 ( &seed );
  }
  for ( k = 0; k < m * n; k++ )
  {
    c[k] = r8_uniform_01 ( &seed );
  }
/*
  Compute A = B * C.
*/
  time_begin = omp_get_wtime ( );
  shared ( a, b, c, l, m, n ) \
  private ( i, j, k )
  for ( j = 0; j < n; j++)
  {
    for ( i = 0; i < l; i++ )
    {
      a[i+j*l] = 0.0;
      for ( k = 0; k < m; k++ )
      {
        a[i+j*l] = a[i+j*l] + b[i+k*l] * c[k+j*m];
      }
```

```
    }
  }
  time_stop = omp_get_wtime ( );
/*
  Report.
*/
  ops = l * n * ( 2 * m );
  time_elapsed = time_stop - time_begin;
  rate = ( double ) ( ops ) / time_elapsed / 1000000.0;

  printf ( "\n" );
  printf ( "R8_MXM matrix multiplication timing.\n" );
  printf ( "  A(LxN) = B(LxM) * C(MxN).\n" );
  printf ( "  L = %d\n", l );
  printf ( "  M = %d\n", m );
  printf ( "  N = %d\n", n );
  printf ( "  Floating point OPS roughly %d\n", ops );
  printf ( "%f\n", time_elapsed );
  printf ( "  Rate = MegaOPS/dT = %f\n", rate );
  free ( a );
  free ( b );
  free ( c );
  return;
}
/******************************************************/
double r8_uniform_01 ( int *seed )
/******************************************************/
/*
  Purpose:
    R8_UNIFORM_01 is a unit pseudorandom R8.

  Discussion:
    This routine implements the recursion
      seed = 16807 * seed mod ( 2**31 - 1 )
      unif = seed / ( 2**31 - 1 )
    The integer arithmetic never requires more than 32 bits,
    including a sign bit.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    11 August 2004

  Author:
    John Burkardt

  Reference:
    Paul Bratley, Bennett Fox, Linus Schrage,
    A Guide to Simulation,
    Springer Verlag, pages 201-202, 1983.

    Bennett Fox,
    Algorithm 647:
    Implementation and Relative Efficiency of Quasirandom
    Sequence Generators,
    ACM Transactions on Mathematical Software,
    Volume 12, Number 4, pages 362-376, 1986.
```

```
    Parameters:
      Input/output, int *SEED, a seed for the random number generator.
      Output, double R8_UNIFORM_01, a new pseudorandom variate, strictly
  between
      0 and 1.
  */
  {
    int k;
    double r;
    k = *seed / 127773;
    *seed = 16807 * ( *seed - k * 127773 ) - k * 2836;
    if ( *seed < 0 )
    {
      *seed = *seed + 2147483647;
    }
    r = ( double ) ( *seed ) * 4.656612875E-10;
    return r;
  }
```

12.9. Producto Escalar

```
#include <stdio.h>
#define N 10000000
float A[N], B[N], pe;
void AporB()
{
  int j;
  for (j=0; j<N; j++) pe += A[j] * B[j];
}
int main(int argc, char ** argv)
{
  double wtime;
  int i;
  wtime = omp_get_wtime ( );
  for (i=0; i<N; i++)
  {
    A[i] = i;
    B[i] = N-i;
  }
  pe = 0.0;
  {
    AporB();
  }
   wtime = omp_get_wtime ( ) - wtime;
  printf("\n\n     >> PE = %10.0f\n\n", pe);
  printf ( "%f\n", wtime );
}
```

12.10. Método de sobrerrelajacion sucesiva

```
#include <stdio.h>
#define width 256
#define height 256

void Sequential()
{
```

```
    float A[width][height], B[width][height];
    int i,j,t;

    for(i=0; i<width; i++){
        for(j=0; j<height; j++){
            B[i][j] = (float) (i+1)/(j+1);
        }
    }
     for(t=0; t<1000; t++)
    {
        for(i=1; i<width-1; i++)
        {
            for(j=1; j<height-1; j++)
            {
       A[i][j] = (B[i+1][j] + B[i-1][j] + B[i][j+1] + B[i][j-1])/4;
            }
        }
         for(i=1; i<width-1; i++)
        {
            for(j=1; j<height-1; j++)
            {
                B[i][j] = A[i][j];
            }
        }
    }
    printf("%f sec Sequential Complete.\n");
}
 int main ( int argc, char *argv[] )
{
    Sequential();
    return 0;
}
```

12.11. Fibonacci

```
    #include <stdio.h>
    #define width 256
    #define height 256

    void Sequential()
    {
        float A[width][height], B[width][height];
        int i,j,t;
        for(i=0; i<width; i++){
            for(j=0; j<height; j++){
                B[i][j] = (float) (i+1)/(j+1);
            }
        }
         for(t=0; t<1000; t++)
        {
            for(i=1; i<width-1; i++)
            {
                for(j=1; j<height-1; j++)
                {
       A[i][j] = (B[i+1][j] + B[i-1][j] + B[i][j+1] + B[i][j-1])/4;
                }
            }
```

```
        for(i=1; i<width-1; i++)
        {
            for(j=1; j<height-1; j++)
            {
                B[i][j] = A[i][j];
            }
        }
    }


 printf("%f sec Sequential Complete.\n");

}
int main ( int argc, char *argv[] )
{
     Sequential();
     return 0;
}
```

12.12. Calculo de PI

```
#include <stdio.h>

static long num_steps = 1000000;
double step;

void Sequential()
{
    int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for(i=1; i<=num_steps; i++)
    {
        x = (i-0.5) * step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("PI : %f\n",pi);
  }

int main()
{
    Sequential();
    return 0;
}
```

12.13. Numeros Primos

```
# include <stdlib.h>
# include <stdio.h>
int main(int argc, char ** argv);
void prime_number_sweep ( int n_lo, int n_hi, int n_factor );
int prime_number ( int n );
```

```
/************************/
int main(int argc, char ** argv)
/****************************/
/*
  Purpose:
    MAIN is the main program for PRIME_OPENMP.

  Discussion:
    This program calls a version of PRIME_NUMBER that includes
    OpenMP directives for parallel processing.

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    06 August 2009

  Author:
    John Burkardt
*/
{
  int n_factor;
  int n_hi;
  int n_lo;
  double wtime;
  int nthreads;
  printf ( "\n" );
  printf ( "PRIME_SECUENCIAL\n" );
  printf ( "  C/OpenMP version\n" );
  printf ( "\n" );
  wtime = omp_get_wtime ( );
  n_lo = 1;
  n_hi = 131072;
  n_factor = 2;

  prime_number_sweep ( n_lo, n_hi, n_factor );
  n_lo = 5;
  n_hi = 500000;
  n_factor = 10;
  prime_number_sweep ( n_lo, n_hi, n_factor );
  wtime = omp_get_wtime ( ) - wtime;
/*
  Terminate.
*/
  printf ( "\n" );
  printf ( "PRIME_SECUENCIAL\n" );
  printf ( " %f\n", wtime );
  printf ( "  Normal end of execution.\n" );

  return 0;
}
/*************************************************/
void prime_number_sweep ( int n_lo, int n_hi, int n_factor )
/*************************************************/
/*
  Purpose:
   PRIME_NUMBER_SWEEP does repeated calls to PRIME_NUMBER.

  Licensing:
```

```
    This code is distributed under the GNU LGPL license.

  Modified:
    06 August 2009

  Author:
    John Burkardt

  Parameters:
    Input, int N_LO, the first value of N.
    Input, int N_HI, the last value of N.
    Input, int N_FACTOR, the factor by which to increase N after
    each iteration.
*/
{
  int i;
  int n;
  int primes;
  double wtime;

  printf ( "\n" );
  printf ( "TEST01\n" );
  printf ( "  Call PRIME_NUMBER to count the primes from 1 to N.\n" );
  printf ( "\n" );
  printf ( "         N        Pi          Time\n" );
  printf ( "\n" );
  n = n_lo;
  while ( n <= n_hi )
  {
    wtime = omp_get_wtime ( );
    primes = prime_number ( n );
    wtime = omp_get_wtime ( ) - wtime;
    printf ( "  %8d  %8d  %14f\n", n, primes, wtime );
    n = n * n_factor;
  }
  return;
}
/**********************************************************/
int prime_number ( int n )
/**********************************************************/
/*
  Purpose:
    PRIME_NUMBER returns the number of primes between 1 and N.

  Discussion:
    A naive algorithm is used.
    Mathematica can return the number of primes less than or equal to N
    by the command PrimePi[N].

                N  PRIME_NUMBER

                1           0
               10           4
              100          25
            1,000         168
           10,000       1,229
          100,000       9,592
        1,000,000      78,498
       10,000,000     664,579
```

```
      100,000,000   5,761,455
    1,000,000,000  50,847,534

  Licensing:
    This code is distributed under the GNU LGPL license.

  Modified:
    21 May 2009

  Author:
    John Burkardt

  Parameters:
    Input, int N, the maximum number to check.
    Output, int PRIME_NUMBER, the number of prime numbers up to N.
*/
{
  int i;
  int j;
  int prime;
  int total = 0;
  shared ( n ) \
  private ( i, j, prime )
  for ( i = 2; i <= n; i++ )
  {
    prime = 1;
    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )
      {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
  return total;
}
```

12.14. Multiplicacion de matrices For Nowait

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

#define DEBUG 0
#define NRA 1600                 // number of rows in matrix A
#define NCA 1600                 // number of columns in matrix A
#define NCB 1600                 // number of columns in matrix B

int main(int argc, char ** argv) {
  int   tid, nthreads, i, j, k;
  double **a, **b, **c;
  double *a_block, *b_block, *c_block;
  double **res;
  double *res_block;
  double starttime, stoptime;

  /* Check that the user gives number of threads */
```

```
/*if (argc < 2) {
    printf("Usage: %s N    where N is number of threads\n", argv[0]);
    exit(0);
  }
  else {
    nthreads = atoi(argv[1]);
  }
  printf("Number of threads is %d\n", nthreads);
*/
  a = (double **) malloc(NRA*sizeof(double *)); /* matrix a to be
multiplied */
  b = (double **) malloc(NCA*sizeof(double *)); /* matrix b to be
multiplied */
  c = (double **) malloc(NRA*sizeof(double *)); /* result matrix c */

  a_block = (double *) malloc(NRA*NCA*sizeof(double)); /* Storage for
matrix a */
  b_block = (double *) malloc(NCA*NCB*sizeof(double));
  c_block = (double *) malloc(NRA*NCB*sizeof(double));

  res = (double **) malloc(NRA*sizeof(double *));
  res_block = (double *) malloc(NRA*NCB*sizeof(double));

  for (i=0; i<NRA; i++)   /* Initialize pointers to a */
    a[i] = a_block+i*NRA;

  for (i=0; i<NCA; i++)   /* Initialize pointers to b */
    b[i] = b_block+i*NCA;

  for (i=0; i<NRA; i++)   /* Initialize pointers to c */
    c[i] = c_block+i*NRA;

  for (i=0; i<NRA; i++)   /* Initialize pointers to res */
    res[i] = res_block+i*NRA;

  starttime = omp_get_wtime();
  /* Do a sequential matrix multiplication and compare the results */
  for (i=0; i<NRA; i++) {
    for (j=0; j<NCB; j++) {
      res[i][j] = 0.0;
      for (k=0; k<NCA; k++)
   res[i][j] += a[i][k]*b[k][j];
    }
  }
  stoptime = omp_get_wtime();
  printf("Time for sequential matrix multiplication: %3.2f\n", stoptime-
starttime);

  /* Check that the results are the same as in the parallel solution.
     Actually, you should not compare floating point values for equality
like this
     but instead compute the difference between the two values and check
that it
     is smaller than a very small value epsilon. However, since all
values in the
     matrices here are integer values, this will work.
  */
  for (i=0; i<NRA; i++) {
```

```
        for (j=0; j<NCB; j++) {
          if (res[i][j] == c[i][j]) {
       /* Everything is OK if they are equal */
          }
          else {
       printf("Different result %5.1f != %5.1f in %d %d\n ", res[i][j],
    c[i][j], i, j);
          }
        }
      }
      /* If DEBUG is true, print the results */
      if (DEBUG) {
        printf("Result Matrix:\n");
        for (i=0; i<NRA; i++) {
          for (j=0; j<NCB; j++)
       printf("%6.1f ", c[i][j]);
          printf("\n");
        }
      }
      printf ("Done.\n");
      exit(0);
    }
```

12.15. Critical

```
#include <stdio.h>
int main(int argc, char ** argv)
{
    int i;
       double wtime;


    int glob_nloops, priv_nloops;
    glob_nloops = 0;
        {
        wtime = omp_get_wtime ( );
        priv_nloops = 0;
        for (i=0; i<100000; ++i)
        {
            ++priv_nloops;
        }
         {
            glob_nloops += priv_nloops;
         }
    }
   printf("Total # loop iterations is %d\n", glob_nloops);
    return 0;
}
```

12.16. Ecuacion De La Onda

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <sys/time.h>

using namespace std;

int tout=1000;
int Nx=201, Nz=201;
float alpha = 1.0, beta=1.0;
float c = 10;
float dt=0.01, G=0.1;
float U1[201][201], U2[201][201], U3[201][201][1000] ;
float dx, dz, x, z, t;
float x0=50.0, z0=50.0, r ;
char namefile[15];

FILE *out;

int main()
{
struct timeval t0, t1;
double tej;
int i,j,n;

gettimeofday(&t0, 0);   // Tomamos tiempo de inicio
//out = fopen("2Ddata.txt","w");
dx = c*dt/G;
dz = dx;
{
  for (i=0; i<Nx; i++)
   {
     for (j=0; j<Nz; j++)
      {
        x = i*dx-x0;
        z = j*dz-z0;
          r = sqrt(x*x+z*z);
          U2[i][j] = sin(alpha*r)*exp(-beta*r*r);
          U1[i][j] = 0.0;
          //U3[i][j] = 0.0;
          /* U1[i] = U2[i];*/
      }
    }
}
for(n=0; n<tout; n++)
 {
 {
     for ( i=1; i<Nx-1; i++)
      {
```

```
       for ( j=1; j<Nz-1; j++)
         {
           U3[i][j][n] = 2*(1-2*G*G) * U2[i][j] + G*G*(U2[i+1][j] + U2[i-1][j]+ U2[i][j+1]+
U2[i][j-1]) - U1[i][j];
         }
     }
   for ( i=1; i<Nx-1; i++)
    {
     for ( j=1; j<Nz-1; j++)
      {
        U1[i][j] = U2[i][j];
        U2[i][j] = U3[i][j][n];
       }
    }
  }
/************** Acceso a  disco ************************
for(n=0; n<tout; n++)
 {
   sprintf(namefile,"../plot2d/2Ddata%04d.txt",n);
   out = fopen(namefile,"w");
    for ( i=1; i<Nx-1; i++)
      {
        for ( j=1; j<Nz-1; j++)
         {
           fprintf(out,"%f ",U3[i][j][n]);
         }
        fprintf(out,"\n");
      }
   fclose (out);
 }
/*********************************************************/


gettimeofday(&t1, 0);  // Tomamos tiempo final
tej = (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec) / 1e6;
printf("\n Tiempo de ejecucion: %1.3f ms\n\n", tej*1000);
return (0);
}
```

13. Tablas De Datos

13.1. Poisson

Tabla 23 Poisson En OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	16.250	10.803	8.497	6.740	5.927	5.283	4.830	4.471	4.375	4.023	3.988	3.739
2	16.191	10.936	8.174	6.748	6.180	5.303	4.836	4.444	4.185	3.988	3.920	3.865
3	16.134	10.763	8.298	6.745	6.201	5.283	4.772	4.414	4.270	4.124	3.905	3.738
4	16.148	10.777	8.165	6.701	6.088	5.272	4.962	4.358	4.289	4.132	3.942	3.781
5	16.125	10.826	8.421	6.675	6.036	5.277	4.887	4.440	4.339	4.067	3.910	3.916
6	16.107	10.908	8.228	6.697	5.700	5.330	4.694	4.436	4.188	4.085	3.948	4.099
7	16.123	10.882	8.142	6.705	5.916	5.289	4.708	4.414	4.298	4.084	3.975	3.744
8	16.148	10.797	7.821	6.783	4.589	5.310	4.658	4.428	4.444	4.070	3.942	3.737
9	16.169	10.823	7.959	6.739	4.739	5.279	4.890	4.405	4.162	4.249	3.937	3.755
10	16.182	10.743	8.194	6.715	5.881	5.223	4.825	4.393	4.389	4.001	3.930	3.871
11	16.174	10.783	8.342	6.749	5.283	5.280	4.869	4.366	4.465	4.042	3.969	3.733
12	16.191	10.800	8.230	6.733	5.903	5.294	4.801	4.404	4.392	4.067	3.967	3.755
13	16.099	10.747	8.142	6.783	5.521	5.315	4.807	4.369	4.479	4.044	3.935	3.763
14	16.196	10.772	7.989	6.767	6.019	5.250	4.745	4.463	4.409	4.011	3.966	3.550
15	16.153	10.775	8.450	6.715	4.624	5.241	4.689	4.401	4.205	4.270	3.933	3.766
16	16.200	10.749	8.202	6.740	6.221	5.286	4.797	4.395	5.051	4.164	3.925	3.731
17	16.176	10.773	8.372	6.759	6.137	5.310	4.910	4.382	4.345	4.069	3.930	3.725
18	16.140	10.789	8.492	6.737	6.129	5.372	4.900	4.410	4.253	4.007	3.997	4.329
19	16.119	10.787	8.287	6.759	6.034	5.265	4.805	4.424	4.322	3.974	3.993	3.748
20	16.165	10.870	8.079	6.759	6.010	5.251	4.801	4.372	4.205	4.191	3.965	3.734
21	16.071	10.769	7.719	6.790	6.024	5.312	4.795	4.409	4.341	4.279	3.998	3.722
22	16.187	10.768	8.341	6.715	5.821	5.228	4.877	4.406	4.372	4.032	3.965	4.103
23	16.168	10.729	8.186	6.730	5.901	5.271	4.858	4.444	4.223	4.035	3.941	3.725
24	16.144	10.821	8.355	6.777	4.558	5.294	4.814	4.423	4.372	4.107	3.959	3.762
25	16.137	10.722	8.369	6.848	5.923	5.242	4.852	4.425	4.416	4.063	3.912	3.711
26	16.149	10.790	8.102	6.732	4.988	5.274	4.884	4.402	4.233	4.134	3.968	3.710
27	16.154	10.816	8.488	6.680	6.061	5.325	4.780	4.398	4.499	4.033	3.918	3.711
28	16.066	10.803	8.092	6.672	4.615	5.389	4.835	4.373	4.370	4.029	3.942	3.735
29	16.151	10.826	7.985	6.767	5.800	5.244	4.856	4.374	4.205	4.027	3.903	3.795
30	16.169	12.253	8.224	6.863	4.593	5.380	4.958	4.424	4.315	4.007	3.964	3.779

Tabla 24 Poisson Secuencial

1	13.982
2	13.968
3	14.032
4	14.030
5	14.087
6	14.013
7	14.079
8	14.014
9	13.961
10	13.990
11	13.999
12	14.013
13	13.992
14	13.963
15	14.063
16	14.064
17	14.022
18	14.072
19	14.042
20	14.014
21	14.036
22	13.969
23	14.023
24	14.022
25	14.067
26	14.065
27	14.052
28	14.068
29	14.023
30	14.044

13.2. Ecuacion de Calor En 2D

Tabla 25Ecuacion De Calor 2D En OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	76.121992	38.449320	27.121635	19.672325	17.129199	13.441622	11.738030	10.309583	10.040696	8.486148	7.876465	7.356938
2	76.226778	38.425503	28.033684	19.690282	15.957017	13.644060	12.724657	10.371883	10.547673	8.609122	7.883588	7.249810
3	76.218954	38.515473	28.801211	19.646022	15.921659	13.672637	12.150118	10.393801	9.658567	8.561096	7.862645	7.358348
4	76.096959	38.537124	30.640853	19.697409	15.972761	13.387843	12.198478	10.290319	9.367019	8.557681	8.341622	7.332942
5	76.028014	38.443465	28.932998	19.751351	15.955376	13.374730	12.462700	10.432543	9.292523	8.943011	7.959969	7.276592
6	76.088242	38.423363	27.132747	19.613928	17.136499	13.542622	12.198518	10.410490	9.427580	8.531576	7.795355	7.265279
7	76.085240	38.434331	28.990542	19.627228	18.539380	13.444928	11.216030	10.374232	9.428026	8.882752	7.929341	7.306852
8	76.237725	38.404820	28.801165	19.714415	15.975323	13.568470	12.959932	10.380256	10.217672	8.444376	7.832580	7.308213
9	76.060379	38.430480	26.098638	19.735036	16.699822	13.490060	12.088444	10.450434	9.393490	8.599888	8.369023	7.285990
10	76.134045	38.455671	28.038642	19.710824	15.756213	13.803637	12.326478	10.494490	9.510794	8.581093	7.813644	7.448835
11	76.073465	38.472941	27.802991	19.830836	16.616569	13.610297	12.217857	10.435251	9.360487	8.564517	8.196276	7.358577
12	76.075500	38.420030	30.640953	19.653029	17.177003	13.595969	11.698574	10.475236	9.215467	8.605251	7.804092	7.371910
13	76.091456	38.494565	27.589010	19.834008	15.947567	13.326060	12.056340	10.395764	9.221443	8.490391	7.783425	7.369942
14	76.081630	38.505651	27.187412	19.667039	17.099490	13.327643	12.121518	10.445374	9.216523	8.549985	7.807116	7.377320
15	76.184765	38.401721	28.984317	19.667039	15.964970	13.521370	12.980844	10.386685	9.422154	8.521975	7.981753	7.442358
16	76.060137	38.585945	28.873123	19.665964	15.846134	13.218097	12.214700	10.435368	9.290612	8.700029	8.616588	7.321928
17	76.181773	38.504702	28.831028	19.709036	17.530428	13.659087	12.549932	10.522811	9.313240	8.478417	7.999471	7.282563
18	76.033387	38.453968	26.074135	19.666234	15.743234	13.502864	12.154378	10.388392	9.351161	8.387589	7.994892	7.367473
19	76.258159	38.493368	30.632173	19.798037	15.835743	13.980469	12.021340	10.430946	9.214482	8.611019	7.877665	7.298876
20	76.063415	38.442835	27.743189	19.741043	15.949719	13.217630	11.215474	10.464368	9.324353	8.771812	7.914170	7.266751
21	76.029020	38.578725	28.932185	19.548601	15.939899	13.757522	12.124923	10.439651	9.313411	8.607446	7.839308	7.362914
22	76.065225	38.464084	26.097321	19.768787	17.735705	14.035421	11.858383	10.328535	10.411706	8.516551	9.265676	7.348833
23	76.119023	38.442547	28.037641	19.721757	16.802607	13.215464	11.994330	10.348422	9.447642	8.493050	9.595923	7.281145
24	76.127974	38.488898	28.842532	19.856569	16.716810	13.664732	12.807818	10.391091	9.316129	8.545788	7.881733	7.274316
25	76.255477	38.495940	27.563189	19.736680	17.408573	13.512676	11.683702	10.405917	10.289061	8.604404	7.881347	7.364470
26	76.065235	38.446059	27.673165	19.727753	15.947321	13.534276	11.214501	10.373795	10.43266	8.480166	7.985203	7.277618
27	76.079204	38.515885	28.801741	19.732800	17.234685	13.531676	12.643223	10.405876	9.279165	8.570740	8.144612	7.311078
28	76.046835	38.489407	28.841754	19.678931	16.482607	13.326522	11.900930	10.363778	9.221912	8.490259	7.878768	7.344359
29	76.234890	38.459870	27.573756	19.766420	16.734810	13.219032	12.246818	10.334584	10.21760	8.602123	7.858669	7.372648
30	76.172480	38.532092	30.621732	19.780763	17.734674	14.219421	11.821438	10.365668	9.216143	8.421655	7.953870	7.311366

Tabla 26Ecuacion De Calor En 2D

1	69.037.408
2	69.077.471
3	68.997.674
4	69.076.197
5	69.087.392
6	69.098.432
7	69.015.624
8	69.088.847
9	69.153.070
10	69.067.119
11	69.010.658
12	9.104.245
13	69.105.709
14	69.074.057
15	69.137.422
16	69.785.588
17	69.059.973
18	69.043.287
19	69.088.383
20	69.074.987
21	69.101.334
22	69.061.650
23	69.073.076
24	69.101.184
25	69.081.679
26	69.004.774
27	69.660.383
28	69.163.131
29	69.120.254
30	69.023.805

13.3. Ziggurat

Tabla 27Ziggurat Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	5.472695	4.435132	4.234358	3.877727	3.656462	3.527374	3.476307	3.399373	3.304626	3.220230	3.293155	3.244155
2	5.471569	4.480172	4.205713	3.835043	3.618547	3.546293	3.454482	3.383534	3.303619	3.263071	3.217164	3.200343
3	5.426878	4.504622	4.290065	3.905168	3.682197	3.562911	3.450073	3.401614	3.306842	3.228518	3.229275	3.169014
4	5.467941	4.492071	4.110544	3.834233	3.745120	3.552812	3.373212	3.401365	3.360477	3.228733	3.228926	3.227810
5	5.464652	4.617691	4.190168	3.786674	3.680165	3.515259	3.372521	3.363453	3.292523	3.252518	3.263776	3.223896
6	5.464270	4.492601	4.139379	3.744734	3.634241	3.494762	3.365594	3.382891	3.292980	3.243574	3.256208	3.168890
7	5.433231	4.505385	4.136537	3.767147	3.699591	3.593933	3.385742	3.355199	3.306600	3.322170	3.223549	3.189780
8	5.425451	4.504920	4.028322	3.857257	3.671225	3.592723	3.373520	3.363052	3.296451	3.212640	3.234556	3.165014
9	5.463606	4.494864	4.258365	3.843888	3.691428	3.539507	3.415111	3.302133	3.265419	3.226673	3.246067	3.218279
10	5.419153	4.493087	4.161155	3.821745	3.770457	3.519593	3.458745	3.337614	3.256435	3.301193	3.248099	3.168104
11	5.430967	4.510965	4.286382	3.878560	3.720132	3.497485	3.494963	3.348225	3.304708	3.216313	3.248585	3.236944
12	5.461917	4.476860	4.207319	3.826651	3.658479	3.541344	3.478775	3.371664	3.301822	3.259416	3.248583	3.210919
13	5.421982	4.506078	4.212359	3.746562	3.655746	3.554004	3.456722	3.363974	3.264925	3.221443	3.245315	3.171188
14	5.418838	4.507914	4.167440	3.819912	3.615097	3.526004	3.472048	3.377993	3.252019	3.233882	3.252886	3.193512
15	5.422563	4.491527	4.168066	3.799327	3.684542	3.541066	3.505116	3.315181	3.252019	3.240053	3.255467	3.179509
16	5.591958	4.474714	4.195073	3.758371	3.685777	3.631183	3.457205	3.340289	3.298226	3.247997	3.261831	3.201693
17	5.427993	4.523547	4.197109	3.740546	3.679295	3.573288	3.428352	3.387832	3.307466	3.249406	3.237637	3.164692
18	5.424482	4.544032	4.086047	3.844036	3.661357	3.480087	3.457029	3.342826	3.305247	3.251562	3.262049	3.221554
19	5.474330	4.580958	4.380132	3.841545	3.650993	3.566739	3.431050	3.355906	3.263281	3.245248	3.230913	3.217595
20	5.429043	4.475605	4.177682	3.801862	3.683888	3.573388	3.483583	3.395780	3.260316	3.212781	3.247146	3.172560
21	5.423670	4.514399	4.204681	3.749406	3.664671	3.601911	3.410555	3.359365	3.254498	3.281226	3.257628	3.208701
22	5.419753	4.490506	4.285103	3.745134	3.684300	3.519753	3.502643	3.383255	3.251398	3.244202	3.225746	3.170610
23	5.439430	4.515551	4.176712	3.847326	3.609400	3.507652	3.451708	3.367917	3.250205	3.286872	3.260341	3.199148
24	5.417392	4.484827	4.293869	3.779189	3.666172	3.477539	3.400797	3.394681	3.237668	3.231640	3.246705	3.154021
25	5.425672	4.488118	4.198195	3.806509	3.715623	3.500387	3.399828	3.367738	3.285615	3.229668	3.251187	3.220158
26	5.426326	4.507329	4.041963	3.908396	3.688862	3.524599	3.464983	3.348872	3.309114	3.246685	3.252406	3.171163
27	5.417263	4.496433	4.277325	3.845325	3.656929	3.524997	3.455131	3.377328	3.260415	3.253589	3.241842	3.220046
28	5.422947	4.494222	4.195784	3.878570	3.623770	3.549040	3.412048	3.361736	3.259619	3.237703	3.244842	3.145198
29	5.439680	4.581549	4.305216	3.831908	3.621517	3.516468	3.486745	3.397310	3.299772	3.253674	3.250397	3.206538
30	5.424849	4.538971	4.209438	3.819754	3.607334	3.474753	3.399384	3.373831	3.324591	3.248826	3.256234	3.165156

Tabla 28Ziggurat Con Algoritmo Secuencial

1	5.395.345
2	5.350.592
3	5.350.358
4	5.355.542
5	5.353.078
6	5.352.196
7	5.350.167
8	5.361.826
9	5.395.606
10	5.347.336
11	5.347.244
12	5.346.955
13	5.350.745
14	5.344.339
15	5.353.106
16	5.356.353
17	5.348.582
18	5.395.018
19	5.345.124
20	5.354.578
21	5.360.593
22	5.353.112
23	5.360.211
24	5.348.873
25	5.415.375
26	5.394.084
27	5.355.636
28	5.358.659
29	5.352.380
30	5.357.796

4.4. Satisfy

Tabla 29 Satisfy Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	1.171768	0.609747	0.416924	0.334171	0.306616	0.262923	0.209393	0.193230	0.196546	0.173942	0.159595	0.149406
2	1.158702	0.616206	0.433078	0.369530	0.281015	0.276201	0.249558	0.202711	0.196694	0.178532	0.162826	0.142956
3	1.143361	0.605360	0.429991	0.345037	0.280559	0.275160	0.239148	0.199797	0.197072	0.178597	0.157933	0.153772
4	1.162912	0.630076	0.454714	0.361838	0.309151	0.261420	0.211690	0.205257	0.196350	0.163825	0.164119	0.150842
5	1.142805	0.642253	0.424393	0.366293	0.285896	0.254155	0.230634	0.221496	0.196938	0.178136	0.163124	0.148670
6	1.185492	0.636612	0.417312	0.344291	0.304391	0.233756	0.209185	0.215770	0.187988	0.178392	0.162573	0.149976
7	1.169096	0.608521	0.493996	0.349659	0.298439	0.262139	0.233186	0.189329	0.170991	0.178612	0.162571	0.147613
8	1.177435	0.680071	0.423998	0.357829	0.301450	0.243227	0.236784	0.205551	0.194065	0.179531	0.164741	0.152843
9	1.174911	0.641705	0.460128	0.357768	0.299137	0.246704	0.217955	0.215134	0.197154	0.178622	0.163382	0.151522
10	1.178877	0.618391	0.434050	0.347368	0.291954	0.243850	0.219302	0.201278	0.180750	0.178357	0.163822	0.148500
11	1.168759	0.611437	0.429698	0.331013	0.274686	0.238565	0.240633	0.206259	0.196035	0.178476	0.164450	0.149309
12	1.179567	0.629577	0.432871	0.347787	0.309480	0.271263	0.225945	0.221390	0.196886	0.171470	0.162511	0.149101
13	1.161848	0.599139	0.427836	0.330048	0.285023	0.247630	0.232085	0.221353	0.182733	0.171582	0.162552	0.153832
14	1.168134	0.641319	0.418315	0.359781	0.286070	0.233661	0.209236	0.189190	0.182808	0.170717	0.163347	0.151924
15	1.168816	0.597747	0.518488	0.329681	0.279197	0.253434	0.208499	0.192092	0.195720	0.178393	0.165324	0.147540
16	1.172872	0.661537	0.476478	0.356816	0.299926	0.247110	0.224960	0.195161	0.188905	0.178875	0.162587	0.150773
17	1.110456	0.625492	0.418610	0.353166	0.275165	0.235954	0.227581	0.200062	0.197218	0.178683	0.165846	0.149308
18	1.182823	0.605127	0.431106	0.354474	0.299751	0.248880	0.235319	0.200526	0.193144	0.178770	0.163315	0.150868
19	1.144330	0.640120	0.434503	0.366182	0.288761	0.235294	0.251149	0.215354	0.197079	0.174011	0.166911	0.154265
20	1.180636	0.628019	0.435742	0.340342	0.290750	0.231348	0.228478	0.201704	0.190798	0.178286	0.167401	0.152303
21	1.139848	0.612673	0.433318	0.360013	0.307040	0.242445	0.229825	0.193878	0.197161	0.165548	0.168427	0.144948
22	1.173310	0.597776	0.423155	0.361459	0.284472	0.262237	0.249468	0.216190	0.197267	0.172779	0.162348	0.149794
23	1.149108	0.605512	0.455505	0.338347	0.286395	0.252234	0.236002	0.217664	0.182741	0.170656	0.166317	0.151759
24	1.165251	0.627716	0.449669	0.325348	0.288441	0.256250	0.215040	0.221843	0.197290	0.178584	0.163479	0.151974
25	1.149268	0.616779	0.421801	0.328354	0.286575	0.257038	0.223992	0.221795	0.197761	0.162516	0.162558	0.149498
26	1.163759	0.620538	0.425539	0.325776	0.271161	0.260360	0.216779	0.190558	0.182587	0.163612	0.164073	0.153139
27	1.168283	0.640309	0.454765	0.359188	0.289862	0.237499	0.209875	0.222186	0.193055	0.177012	0.162215	0.149026
28	1.168847	0.619861	0.442782	0.334942	0.283070	0.274719	0.231034	0.191862	0.196952	0.165936	0.165324	0.152183
29	1.151217	0.636644	0.422115	0.347599	0.293000	0.267737	0.220795	0.222253	0.196960	0.170369	0.165310	0.148867
30	1.145867	0.618423	0.448088	0.334392	0.276421	0.275710	0.218055	0.186925	0.196732	0.178671	0.162777	0.150832

Tabla 30 Satisfy Secuencial

1.173.865
1.103.116
1.091.075
1.092.062
1.092.108
1.103.890
1.102.719
1.091.531
1.102.466
1.103.982
1.103.924
1.091.653
1.092.426
1.103.875
1.100.896
1.105.453
1.092.385
1.092.458
1.092.261
1.092.032
1.091.463
1.090.272
1.092.234
1.091.704
1.091.567
1.210.653
1.091.565
1.101.794
1.104.150
1.103.881

4.5. Quad

Tabla 31 Quad Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	0.267541	0.149246	0.099533	0.074753	0.059625	0.050236	0.042718	0.037458	0.033339	0.030094	0.027417	0.025220
2	0.270790	0.148947	0.099693	0.074733	0.060201	0.050107	0.042751	0.037485	0.033565	0.030068	0.027349	0.025665
3	0.277319	0.149021	0.099746	0.074500	0.059940	0.049994	0.042737	0.037489	0.033664	0.030040	0.027370	0.025164
4	0.255184	0.146068	0.099671	0.074712	0.059872	0.049779	0.042746	0.037691	0.033662	0.030285	0.027496	0.025419
5	0.253652	0.149859	0.099671	0.074709	0.060107	0.050526	0.042958	0.037567	0.033718	0.030162	0.027374	0.025454
6	0.259328	0.148954	0.099658	0.074781	0.060215	0.050082	0.042726	0.037689	0.033324	0.030036	0.027388	0.025194
7	0.260496	0.148939	0.099805	0.074745	0.059631	0.050050	0.042948	0.037571	0.033336	0.030038	0.027582	0.025221
8	0.276228	0.148956	0.099709	0.074610	0.059636	0.050297	0.042975	0.037443	0.033600	0.030296	0.027369	0.025383
9	0.242239	0.149073	0.099661	0.074704	0.060395	0.050500	0.042977	0.037453	0.033580	0.030078	0.027378	0.025156
10	0.251431	0.164689	0.099460	0.074759	0.059949	0.049783	0.042759	0.037450	0.033577	0.030288	0.027363	0.025452
11	0.237812	0.148937	0.099853	0.074727	0.059646	0.049779	0.042932	0.037440	0.033317	0.030072	0.027361	0.025466
12	0.271131	0.148972	0.099687	0.074688	0.059896	0.050017	0.043260	0.037462	0.033332	0.030394	0.029682	0.025392
13	0.266011	0.144135	0.099645	0.074736	0.059891	0.050133	0.042738	0.037472	0.033565	0.030265	0.027370	0.025442
14	0.278102	0.149020	0.099647	0.074688	0.060207	0.050042	0.042949	0.037453	0.033832	0.030262	0.027369	0.025124
15	0.264580	0.146017	0.099728	0.074697	0.059916	0.050235	0.042715	0.037670	0.033579	0.030287	0.027364	0.025177
16	0.252877	0.149019	0.099705	0.074477	0.060133	0.049785	0.043184	0.037462	0.033308	0.030044	0.027358	0.025184
17	0.267971	0.149037	0.103887	0.074732	0.060122	0.049773	0.045683	0.037717	0.035970	0.030282	0.027600	0.025158
18	0.270088	0.148993	0.099962	0.074512	0.060147	0.049991	0.042934	0.037443	0.033359	0.030037	0.027401	0.025307
19	0.275589	0.148952	0.103970	0.074516	0.059634	0.052353	0.042753	0.037434	0.033353	0.030036	0.027377	0.025368
20	0.245503	0.148945	0.099813	0.074725	0.065232	0.050067	0.042929	0.037445	0.033332	0.030036	0.027357	0.025496
21	0.271831	0.148945	0.099755	0.074717	0.059880	0.050023	0.042703	0.037463	0.033559	0.030044	0.027365	0.025367
22	0.207701	0.148957	0.099700	0.074694	0.059899	0.050475	0.042709	0.037456	0.033571	0.030024	0.027589	0.025436
23	0.253646	0.148973	0.099481	0.074629	0.059899	0.050001	0.042705	0.037428	0.036154	0.030044	0.027356	0.025114
24	0.237316	0.149005	0.099667	0.074698	0.060158	0.050294	0.042935	0.037459	0.033347	0.030099	0.027351	0.025137
25	0.248894	0.148991	0.099706	0.074534	0.059643	0.050447	0.042690	0.039767	0.033580	0.030082	0.027379	0.025503
26	0.248760	0.142529	0.099678	0.074731	0.059899	0.050027	0.042968	0.037483	0.033546	0.030061	0.027366	0.025399
27	0.249251	0.148952	0.099582	0.074743	0.059928	0.049784	0.042709	0.037458	0.033364	0.030275	0.027377	0.025604
28	0.268835	0.148993	0.099745	0.074711	0.060051	0.049795	0.043168	0.037502	0.033334	0.030040	0.027586	0.025472
29	0.263242	0.148991	0.099681	0.074692	0.060112	0.050215	0.043023	0.037704	0.033315	0.030055	0.027350	0.025353
30	0.247879	0.148959	0.099454	0.074480	0.059641	0.049985	0.042714	0.037712	0.033324	0.030053	0.027357	0.025191

Tabla 32 Quad Secuencial

1	0,548067
2	0,548137
3	0,548045
4	0,547846
5	0,547931
6	0,548046
7	0,547857
8	0,548055
9	0,547925
10	0,547813
11	0,597423
12	0,548099
13	0,54777
14	0,547875
15	0,548024
16	0,547884
17	0,548117
18	0,617133
19	0,547919
20	0,548258
21	0,547898
22	0,548048
23	0,547801
24	0,547848
25	0,547922
26	0,547861
27	0,548015
28	0,54786
29	0,547699
30	0,548173

4.6. Schedule

Tabla 33 Schedule Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	16.643484	9.937278	7.491780	5.667634	4.451651	3.771539	3.335682	3.023837	2.793588	2.484868	2.474897	2.150450
2	16.603492	9.941995	7.316628	5.454295	4.501538	3.948406	3.298554	2.958292	2.657179	2.496870	2.248419	2.092689
3	16.561633	9.903277	6.989740	5.744006	4.618973	4.115962	3.344977	3.086311	2.702600	2.624529	2.346874	2.083573
4	16.568604	9.905090	7.442292	5.487239	4.462315	3.928503	3.288740	3.031434	2.671340	2.483235	2.347536	2.159930
5	16.606508	9.928374	7.003288	5.445325	4.464873	3.778779	3.477702	3.051636	2.688974	2.445765	2.304075	2.095478
6	16.562195	9.897473	7.016965	5.498125	4.429645	3.868643	3.458906	3.031490	2.683354	2.534860	2.369575	2.150460
7	16.626288	9.913554	7.203199	5.569465	4.763954	3.926610	3.420361	3.038534	2.677601	2.434036	2.268286	2.081922
8	16.599501	9.929648	7.048097	5.681361	4.802569	4.064501	3.341420	3.022828	2.673352	2.545500	2.371353	2.093589
9	16.586122	9.934714	7.295751	5.550555	4.449506	3.755303	3.358240	3.046002	2.638591	2.507436	2.362630	2.207395
10	16.568058	9.965448	7.389163	5.555374	4.753282	3.976263	3.336139	2.983477	2.770998	2.477292	2.471227	2.150430
11	16.572455	9.899669	7.483943	5.460387	4.431613	3.889505	3.352723	3.004037	2.699853	2.440769	2.341188	2.278543
12	16.566789	9.900590	7.371620	5.587778	4.751372	3.731647	3.522270	3.155442	2.752875	2.585906	2.329968	2.054176
13	16.591607	9.861921	6.974149	5.480580	4.432558	3.913657	3.346960	2.945661	2.686524	2.446979	2.455444	2.147867
14	16.599587	9.890781	7.363171	5.609807	4.635947	3.814194	3.482776	3.062086	2.718460	2.516515	2.391952	2.090185
15	16.564575	9.975772	7.058790	5.474577	4.426876	3.755843	3.423443	3.011550	2.670332	2.472613	2.273468	2.088768
16	16.560738	9.930486	7.357311	5.480229	4.462023	3.879178	3.291202	2.948091	2.800138	2.449154	2.338417	2.163231
17	16.596251	9.913490	7.201565	5.602570	4.477732	3.818013	3.322125	2.950079	2.645764	2.505229	2.374527	2.188204
18	16.610904	9.942919	7.372100	5.554789	4.626955	3.746989	3.335612	3.068483	2.658275	2.500498	2.303623	2.124634
19	16.574061	9.870693	7.390378	5.664375	4.677460	3.924717	3.478692	2.986171	2.999722	2.434325	2.392165	2.082573
20	16.552562	9.878264	7.236450	5.631848	4.624487	3.846942	3.321519	2.956952	2.665540	2.495380	2.293977	2.159030
21	16.559297	9.944271	7.399200	5.453172	4.946521	3.939317	3.319521	3.021302	2.655638	2.453015	2.257473	2.145588
22	16.576264	9.978169	7.186051	5.782107	4.521691	4.065445	3.351776	3.011811	2.684353	2.483194	2.387049	2.054928
23	16.588379	9.864246	7.008425	5.816262	4.579376	3.921910	3.377129	3.020413	2.682826	2.529173	2.333763	2.089592
24	16.616076	9.971104	6.994107	5.648775	4.747902	3.788510	3.355877	2.945399	2.645223	2.498282	2.401887	2.074350
25	16.575483	9.954260	7.120750	5.730471	4.747652	3.870844	3.308465	3.024747	2.748419	2.433749	2.299142	2.173492
26	16.597372	9.962096	7.071691	5.626510	4.437394	3.909642	3.353875	2.969284	2.718649	2.479301	2.325707	2.066599
27	16.557119	9.970925	7.210736	5.541198	4.524195	3.878188	3.337680	3.060578	2.650066	2.520432	2.386540	2.230422
28	16.548756	9.863430	7.084171	5.475429	4.469637	4.098311	3.363947	2.935254	2.677859	2.519414	2.290280	2.214675
29	16.589468	9.967615	7.163717	5.683963	4.859914	4.043871	3.326064	2.967334	2.720903	2.478352	2.428163	2.172374
30	16.602587	9.893953	6.945797	5.736593	4.468060	3.858967	3.338770	3.034939	2.662481	2.499424	2.358211	2.206893

Tabla 34 Schedule Secuencial

1	16.491.223
2	16.537.136
3	16.516.553
4	16.583.311
5	16.503.509
6	16.539.857
7	16.493.340
8	16.537.547
9	16.491.589
10	16.536.869
11	16.490.800
12	16.491.435
13	16.583.094
14	16.490.213
15	16.537.213
16	16.491.897
17	16.493.051
18	16.586.029
19	16.493.417
20	16.591.295
21	16.560.198
22	16.492.038
23	16.540.168
24	16.492.734
25	16.493.590
26	16.537.035
27	16.491.439
28	16.536.772
29	16.536.992
30	16.536.598

4.7. Helmholtz

Tabla 35 Helmholtz Con OpenMP

		2	3	4	5	6	7	8	9	10	11	12
1	1.014796	0.546405	0.376806	0.239436	0.192576	0.161713	0.139128	0.122302	0.115914	0.098290	0.091072	0.083346
2	0.953569	0.476654	0.319098	0.239696	0.192320	0.161814	0.139214	0.122791	0.110087	0.098342	0.090856	0.083949
3	0.950677	0.476671	0.318756	0.239802	0.198347	0.162284	0.147375	0.122698	0.114090	0.098358	0.091518	0.083668
4	0.951751	0.476331	0.320620	0.240435	0.192511	0.161996	0.139050	0.122417	0.114702	0.098734	0.091654	0.084450
5	0.951235	0.477053	0.323813	0.240097	0.195263	0.161915	0.154213	0.122315	0.109643	0.098570	0.090417	0.083200
6	1.013665	0.476729	0.319127	0.239671	0.192147	0.162058	0.147539	0.122355	0.119175	0.098349	0.090776	0.092560
7	0.951094	0.476235	0.319301	0.240360	0.192288	0.162048	0.154697	0.122689	0.109215	0.099594	0.091237	0.085288
8	0.950828	0.477226	0.319857	0.239706	0.198308	0.161831	0.148113	0.122235	0.112880	0.098628	0.091052	0.083799
9	0.948895	0.477010	0.319112	0.239698	0.192448	0.162056	0.139058	0.122352	0.110306	0.098027	0.090930	0.083960
10	0.949767	0.476643	0.319510	0.239507	0.192544	0.161833	0.154233	0.122532	0.110371	0.098498	0.090880	0.084391
11	0.949798	0.479174	0.319202	0.239886	0.191967	0.161695	0.142635	0.123099	0.110271	0.099817	0.091390	0.083499
12	0.949410	0.476380	0.368296	0.239947	0.192056	0.161827	0.141395	0.122304	0.113035	0.100024	0.090927	0.083335
13	0.949974	0.515945	0.319385	0.239864	0.219052	0.162092	0.139341	0.122622	0.112304	0.099719	0.090648	0.084157
14	0.949769	0.476611	0.318941	0.239427	0.192713	0.161782	0.139089	0.122198	0.117551	0.100546	0.090644	0.083801
15	0.949410	0.476336	0.389842	0.241469	0.191930	0.161826	0.139068	0.122542	0.110045	0.100726	0.090848	0.083981
16	0.995220	0.476566	0.319619	0.239831	0.192490	0.162279	0.139363	0.122659	0.119913	0.100458	0.091114	0.084419
17	0.949362	0.475978	0.318885	0.239996	0.192246	0.161944	0.139286	0.122461	0.119835	0.100064	0.090993	0.084263
18	0.949622	0.476542	0.339517	0.239951	0.192557	0.161630	0.139125	0.122458	0.110372	0.100102	0.091386	0.094831
19	0.950955	0.476124	0.349901	0.240250	0.219987	0.161961	0.139235	0.122421	0.110094	0.100040	0.091824	0.084592
20	0.949355	0.476625	0.372720	0.239614	0.192499	0.162107	0.139177	0.122361	0.110100	0.100252	0.090780	0.084904
21	0.949229	0.476392	0.375405	0.240121	0.192494	0.161704	0.139139	0.122194	0.110475	0.100283	0.091257	0.085619
22	0.949407	0.476211	0.364854	0.239608	0.192501	0.162515	0.138992	0.122307	0.110501	0.100068	0.091798	0.085003
23	0.950679	0.476784	0.319205	0.239659	0.191936	0.162671	0.139439	0.122822	0.110346	0.100080	0.092723	0.085071
24	0.988920	0.477079	0.382940	0.239970	0.211957	0.161732	0.139005	0.122529	0.110380	0.099990	0.091160	0.081180
25	0.949043	0.476097	0.319619	0.239997	0.192233	0.162141	0.139106	0.122486	0.110699	0.101138	0.090873	0.089313
26	0.949777	0.476109	0.318865	0.240010	0.192181	0.161588	0.139261	0.123219	0.110825	0.099651	0.090892	0.088984
27	0.948888	0.476273	0.377824	0.291139	0.198009	0.161634	0.139131	0.123557	0.110172	0.099920	0.091025	0.087117
28	0.949019	0.476848	0.349901	0.243953	0.192723	0.161992	0.132660	0.122551	0.111416	0.100303	0.090928	0.089725
29	0.949156	0.476678	0.319645	0.242558	0.192434	0.162100	0.138029	0.122298	0.111322	0.099958	0.091540	0.089313
30	0.949768	0.477169	0.377824	0.239908	0.191587	0.161902	0.132992	0.122342	0.110750	0.099764	0.090692	0.087117

Tabla 36 Helmholtz Secuencial

1	0,743973
2	0,742455
3	0,747571
4	0,743883
5	0,795779
6	0,747855
7	0,745236
8	0,749114
9	0,743949
10	0,744253
11	0,748275
12	0,748175
13	0,842387
14	0,749223
15	0,747034
16	0,747953
17	0,747912
18	0,741813
19	0,747046
20	0,747056
21	0,746914
22	0,745333
23	0,743358
24	0,741995
25	0,746772
26	0,748206
27	0,747916
28	0,748068
29	0,74293
30	0,742336

4.8. Multiplicacion De Matrices

Tabla 37 Multiplicacion De Matrices Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	1.631934	0.830388	0.569452	0.465943	0.382508	0.298164	0.304186	0.272503	0.213611	0.236278	0.218655	0.175013
2	1.613352	0.850279	0.604000	0.440473	0.401627	0.298418	0.344316	0.271973	0.253082	0.238542	0.223924	0.148716
3	1.641922	0.837198	0.662070	0.478209	0.419238	0.298004	0.347567	0.235861	0.262497	0.235071	0.191503	0.175176
4	1.643003	0.890909	0.564637	0.503032	0.399992	0.299025	0.331629	0.231467	0.265813	0.236963	0.152917	0.156141
5	1.633969	0.854277	0.554741	0.473011	0.427151	0.297876	0.322244	0.229479	0.244081	0.234364	0.163145	0.180252
6	1.634691	0.850505	0.566185	0.455346	0.438088	0.292719	0.333489	0.230112	0.269871	0.236629	0.155288	0.194676
7	1.628976	0.857501	0.562580	0.460782	0.422446	0.292068	0.318422	0.229349	0.257029	0.233700	0.180665	0.184114
8	1.643117	0.859525	0.562720	0.464774	0.397073	0.284799	0.305466	0.229092	0.247267	0.239676	0.173700	0.134080
9	1.601907	0.860495	0.562091	0.446550	0.427886	0.284955	0.286092	0.237613	0.241156	0.233267	0.195440	0.146233
10	1.617072	0.837394	0.538759	0.456348	0.399700	0.295964	0.245119	0.251111	0.231640	0.235275	0.168301	0.155981
11	1.635415	0.831266	0.596940	0.478094	0.432106	0.326931	0.238960	0.246317	0.232415	0.234078	0.182314	0.146890
12	1.598993	0.838059	0.661760	0.480837	0.398570	0.327783	0.266997	0.240328	0.237896	0.235501	0.159077	0.160156
13	1.620787	0.855751	0.635605	0.459159	0.390526	0.304501	0.267118	0.235586	0.238155	0.238278	0.172440	0.177652
14	1.641072	0.833191	0.626846	0.474716	0.422248	0.297365	0.263471	0.236947	0.237422	0.234182	0.147996	0.139100
15	1.620889	0.848332	0.607159	0.450848	0.397987	0.299261	0.253877	0.235235	0.236441	0.232312	0.160421	0.152899
16	1.616047	0.862951	0.585838	0.473979	0.448657	0.290971	0.284730	0.239480	0.252378	0.235561	0.168078	0.163301
17	1.606303	0.868013	0.591159	0.470940	0.464018	0.293238	0.295249	0.255527	0.266665	0.233260	0.149692	0.136139
18	1.631560	0.850246	0.520937	0.464751	0.454923	0.293882	0.322197	0.256185	0.268501	0.236035	0.163304	0.149673
19	1.617149	0.863593	0.521146	0.462958	0.415614	0.292225	0.334625	0.216606	0.260993	0.233557	0.170732	0.160028
20	1.640843	0.845186	0.521068	0.453734	0.394071	0.306020	0.339378	0.216633	0.247286	0.236046	0.175936	0.145274
21	1.669083	0.832801	0.577203	0.452528	0.464385	0.304384	0.347664	0.216518	0.239813	0.233509	0.187376	0.156142
22	1.615017	0.864123	0.655839	0.479134	0.477934	0.310397	0.346970	0.212733	0.240966	0.236203	0.190117	0.176721
23	1.644694	0.842167	0.642064	0.476618	0.432631	0.303786	0.312644	0.218328	0.228141	0.233624	0.175326	0.136456
24	1.631176	0.835494	0.629047	0.480990	0.435669	0.299588	0.320083	0.201855	0.204321	0.236169	0.189059	0.153018
25	1.641327	0.834320	0.690945	0.475787	0.466363	0.299885	0.341210	0.204139	0.220953	0.233695	0.179090	0.164661
26	1.618072	0.859250	0.591640	0.476658	0.387897	0.299780	0.347051	0.203951	0.220511	0.237334	0.190940	0.150976
27	1.627945	0.855789	0.650848	0.443251	0.317789	0.298838	0.344953	0.211684	0.184553	0.233256	0.149240	0.193142
28	1.618195	0.866875	0.583426	0.477791	0.315057	0.293696	0.270863	0.207122	0.182339	0.236053	0.159921	0.200646
29	1.605563	0.868338	0.546022	0.451143	0.315501	0.292490	0.232655	0.208260	0.191208	0.233497	0.146667	0.177782
30	1.638621	0.831953	0.537982	0.452569	0.315198	0.293799	0.231661	0.201341	0.180600	0.236153	0.162332	0.222343

Tabla 38 Multiplicacion De Matrices Secuencial

1	1.158.506
2	1.190.716
3	1.148.154
4	1.146.216
5	1.145.616
6	1.145.407
7	1.148.210
8	1.147.560
9	1.146.074
10	1.146.196
11	1.147.799
12	1.147.842
13	1.145.639
14	1.151.959
15	1.207.664
16	1.146.118
17	1.145.853
18	1.148.467
19	1.147.650
20	1.147.508
21	1.146.520
22	1.191.758
23	1.147.630
24	1.146.242
25	1.146.753
26	1.145.830
27	1.148.164
28	1.164.666
29	1.146.245
30	1.148.177

4.9. Producto Escalar

Tabla 39 Producto Escalar Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	0.200740	0.132641	0.135014	0.122928	0.125057	0.112500	0.106517	0.105254	0.104785	0.104379	0.103566	0.103958
2	0.142384	0.136087	0.134833	0.123230	0.124883	0.112622	0.106459	0.105101	0.104311	0.104393	0.103468	0.104160
3	0.142558	0.141603	0.134763	0.123173	0.155590	0.112558	0.106290	0.105158	0.104253	0.104280	0.103745	0.104061
4	0.142540	0.132951	0.134738	0.123169	0.110677	0.112584	0.106226	0.105314	0.104626	0.150706	0.103822	0.103900
5	0.142651	0.137341	0.134937	0.123152	0.110553	0.112571	0.106136	0.105110	0.104597	0.150178	0.103938	0.103781
6	0.142496	0.143445	0.134804	0.123339	0.110421	0.112583	0.106042	0.105110	0.104537	0.104255	0.103938	0.103919
7	0.142656	0.147819	0.134832	0.127154	0.114589	0.108482	0.106287	0.105265	0.104567	0.104556	0.103872	0.103963
8	0.142185	0.148005	0.134766	0.129155	0.114708	0.108645	0.106369	0.105334	0.104476	0.104483	0.104214	0.104122
9	0.142237	0.133480	0.134841	0.128822	0.114763	0.108674	0.106260	0.105227	0.104632	0.104323	0.103886	0.104213
10	0.142240	0.133631	0.135013	0.129132	0.114758	0.108726	0.106240	0.105292	0.104873	0.104758	0.103615	0.104192
11	0.142503	0.138309	0.134910	0.128719	0.114837	0.108617	0.106514	0.105249	0.104446	0.104493	0.103715	0.104156
12	0.142240	0.144056	0.121069	0.128890	0.114805	0.108637	0.106296	0.105270	0.104450	0.104696	0.103788	0.104003
13	0.142162	0.147505	0.120828	0.128791	0.114769	0.108693	0.106313	0.105417	0.104757	0.104409	0.103661	0.103765
14	0.142315	0.147941	0.120736	0.128892	0.114795	0.108508	0.106187	0.105287	0.104531	0.104044	0.103881	0.103793
15	0.142364	0.133599	0.124739	0.128877	0.114777	0.108533	0.106293	0.105114	0.104501	0.104199	0.103853	0.104004
16	0.142387	0.137998	0.129372	0.128752	0.114822	0.108490	0.105989	0.105235	0.104454	0.104135	0.103919	0.103971
17	0.142241	0.143707	0.129246	0.128855	0.114690	0.108043	0.106439	0.105403	0.104665	0.103944	0.103730	0.104055
18	0.142310	0.147569	0.129248	0.128678	0.114674	0.107916	0.106367	0.105245	0.104606	0.104223	0.103697	0.104185
19	0.142337	0.147995	0.129269	0.129043	0.114754	0.154425	0.106327	0.127316	0.104483	0.103661	0.103569	0.104196
20	0.142244	0.133612	0.129221	0.128686	0.114761	0.153755	0.106381	0.150564	0.104204	0.104048	0.103790	0.104049
21	0.142296	0.138187	0.129237	0.128652	0.114832	0.153988	0.106239	0.147039	0.104685	0.104257	0.103805	0.103821
22	0.142324	0.178984	0.129471	0.128811	0.114810	0.151195	0.106500	0.105640	0.104668	0.104093	0.103811	0.103678
23	0.142332	0.178304	0.129138	0.128878	0.114819	0.108581	0.106218	0.105464	0.104485	0.103718	0.103811	0.104056
24	0.142233	0.134507	0.129095	0.128679	0.114707	0.108209	0.106381	0.105345	0.104481	0.104220	0.103856	0.104021
25	0.142332	0.139254	0.129483	0.128664	0.114690	0.108381	0.106422	0.105733	0.104614	0.104219	0.104212	0.104126
26	0.142263	0.145431	0.129095	0.128817	0.114674	0.108393	0.106105	0.105546	0.104520	0.104330	0.103852	0.104340
27	0.142556	0.147810	0.129514	0.128897	0.114820	0.108407	0.106320	0.105750	0.104503	0.104255	0.103587	0.104187
28	0.142288	0.133699	0.129366	0.128702	0.114872	0.108338	0.106409	0.105502	0.104644	0.104223	0.103785	0.104160
29	0.142194	0.137696	0.129300	0.128756	0.114861	0.108556	0.106370	0.105223	0.104238	0.104172	0.103888	0.104029
30	0.142244	0.143582	0.129255	0.128800	0.114912	0.108438	0.106332	0.105172	0.104568	0.104254	0.103739	0.103855

Tabla 40 Producto Escalar Secuencial

1	0.141607
2	0.142534
3	0.142812
4	0.142545
5	0.142445
6	0.142432
7	0.142626
8	0.143112
9	0.142675
10	0.142662
11	0.142842
12	0.142605
13	0.142993
14	0.142833
15	0.142989
16	0.142719
17	0.142649
18	0.142976
19	0.142696
20	0.142642
21	0.143071
22	0.142555
23	0.142718
24	0.142649
25	0.142726
26	0.142676
27	0.142674
28	0.142924
29	0.142820
30	0.142943

4.10. Método de sobrerrelajacion sucesiva

Tabla 41 Metodo de Sobrerrelajacion Sucesiva Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	1.020581	0.554124	0.434487	0.308813	0.294984	0.247421	0.244857	0.202344	0.176840	0.168404	0.154331	0.160234
2	1.019188	0.527199	0.407683	0.320244	0.316640	0.238611	0.218933	0.207867	0.189077	0.173345	0.155656	0.138872
3	1.025370	0.547350	0.414999	0.331829	0.268792	0.234131	0.223640	0.211460	0.190912	0.168052	0.172335	0.154304
4	0.990415	0.568172	0.391735	0.335692	0.250106	0.237242	0.227899	0.211190	0.183321	0.175662	0.154102	0.141638
5	0.946812	0.534123	0.422984	0.334420	0.292163	0.231514	0.224210	0.201589	0.182078	0.159018	0.156723	0.148152
6	0.955596	0.558377	0.390024	0.297098	0.269273	0.224835	0.228908	0.203549	0.178920	0.173931	0.151710	0.142985
7	0.943200	0.556082	0.376564	0.311316	0.298216	0.223131	0.227924	0.206669	0.192069	0.174975	0.161784	0.149905
8	0.945360	0.524831	0.418037	0.329140	0.254551	0.218276	0.209850	0.204992	0.186051	0.160891	0.156983	0.152882
9	0.943346	0.555787	0.385780	0.330407	0.311265	0.225444	0.209187	0.213645	0.186204	0.170013	0.153570	0.142358
10	0.949772	0.563768	0.418060	0.320978	0.256885	0.218484	0.221965	0.178516	0.185197	0.169470	0.154234	0.146206
11	0.943026	0.537320	0.393921	0.332402	0.293658	0.222390	0.200792	0.192137	0.187108	0.169459	0.162887	0.141481
12	0.942744	0.558157	0.455512	0.295518	0.311310	0.222707	0.208910	0.192137	0.191824	0.168581	0.154286	0.142478
13	0.944857	0.568199	0.482740	0.311822	0.279553	0.227373	0.223600	0.200580	0.186390	0.158369	0.154568	0.147354
14	1.009923	0.538827	0.385148	0.323561	0.261061	0.226720	0.204576	0.203199	0.179716	0.174330	0.162192	0.169872
15	0.947798	0.554565	0.371283	0.333407	0.290762	0.216872	0.213601	0.202664	0.182939	0.167595	0.154281	0.150298
16	0.944773	0.573703	0.409370	0.295599	0.266998	0.218491	0.213288	0.203093	0.187217	0.171014	0.152619	0.143781
17	0.945528	0.534309	0.509038	0.329898	0.254535	0.220279	0.239676	0.178700	0.183364	0.169259	0.172876	0.141919
18	.944731	0.559135	0.494969	0.339752	0.310740	0.243859	0.207281	0.198866	0.182461	0.160699	0.153839	0.142420
19	0.942967	0.527205	0.458632	0.335736	0.320432	0.219496	0.207651	0.203522	0.175176	0.180271	0.154411	0.141958
20	1.024426	0.556963	0.381543	0.311067	0.250976	0.222047	0.209257	0.202742	0.182958	0.182204	0.155806	0.151025
21	0.941375	0.573095	0.452529	0.319960	0.264215	0.225858	0.224075	0.205977	0.182684	0.174571	0.163590	0.164110
22	0.945425	0.531923	0.416552	0.321333	0.249833	0.225749	0.205071	0.197950	0.186614	0.158070	0.156897	0.159945
23	0.941353	0.558866	0.394892	0.332859	0.310694	0.233298	0.232412	0.203230	0.182955	0.168720	0.155891	0.140647
24	0.943347	0.524936	0.509284	0.294058	0.300480	0.260042	0.200494	0.202430	0.170001	0.177382	0.155986	0.149906
25	0.945682	0.551485	0.481924	0.314496	0.273797	0.251463	0.225888	0.207353	0.188912	0.157781	0.165407	0.150794
26	0.941654	0.576457	0.480399	0.339081	0.257507	0.239391	0.224452	0.180889	0.194383	0.177736	0.164812	0.142896
27	0.944050	0.534104	0.480399	0.327510	0.288251	0.237798	0.223151	0.195872	0.188495	0.168779	0.152568	0.143077
28	0.949583	0.557556	0.415528	0.334243	0.274666	0.263871	0.225963	0.202955	0.165185	0.174160	0.168906	0.142672
29	0.950239	0.554719	0.436928	0.297435	0.276032	0.246563	0.226400	0.203111	0.181868	0.168798	0.154003	0.141883
30	0.944944	0.530085	0.401459	0.314294	0.254459	0.216847	0.229790	0.208946	0.197512	0.155280	0.146616	0.142112

Tabla 42 Metodo de Sobrerrelajacion Sucesiva Secuencial

1	0,772156
2	0,789109
3	0,785123
4	0,780754
5	0,772314
6	0,780384
7	0,772476
8	0,780482
9	0,772205
10	0,772479
11	0,772701
12	0,771732
13	0,77966
14	0,779935
15	0,78081
16	0,77153
17	0,771989
18	0,780241
19	0,771805
20	0,772337
21	0,880113
22	0,772482
23	0,772344
24	0,772657
25	0,772182
26	0,772705
27	0,780435
28	0,850476
29	0,77268
30	0,772762

4.11. Calculo De PI

Tabla 43 Calculo De PI Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	0.011777	0.006020	0.004097	0.003195	0.002780	0.002162	0.001941	0.001727	0.001639	0.001473	0.001449	0.001333
2	0.011833	0.006068	0.004086	0.003114	0.002601	0.002227	0.001916	0.001741	0.001557	0.001494	0.001385	0.001337
3	0.011823	0.006064	0.004077	0.003110	0.002553	0.002186	0.001903	0.001724	0.001561	0.001515	0.001377	0.001316
4	0.011815	0.006046	0.004428	0.003118	0.002585	0.002156	0.001896	0.001714	0.001678	0.001454	0.001386	0.001323
5	0.011761	0.006063	0.004109	0.003157	0.002551	0.002208	0.001927	0.001743	0.001627	0.001474	0.001385	0.001384
6	0.012006	0.006014	0.004103	0.003165	0.002818	0.002149	0.001922	0.001763	0.001700	0.001468	0.001364	0.001329
7	0.012051	0.006027	0.004115	0.003101	0.002537	0.002225	0.001963	0.001733	0.001566	0.001452	0.001372	0.001389
8	0.011772	0.006056	0.004068	0.003138	0.002585	0.002169	0.001956	0.001731	0.001584	0.001484	0.001366	0.001323
9	0.011843	0.006049	0.004114	0.003126	0.002599	0.002188	0.001902	0.001795	0.001584	0.001461	0.001411	0.001323
10	0.011818	0.006068	0.004077	0.003167	0.002529	0.002154	0.001916	0.001721	0.001615	0.001451	0.001410	0.001430
11	0.011773	0.006053	0.004103	0.003126	0.002527	0.002495	0.001944	0.001761	0.001580	0.001489	0.001376	0.001328
12	0.011761	0.006047	0.004098	0.003167	0.002522	0.002189	0.001910	0.001805	0.001590	0.001481	0.001391	0.001307
13	0.012043	0.006008	0.004122	0.003129	0.002588	0.002180	0.001910	0.001779	0.001629	0.001530	0.001369	0.001325
14	0.011760	0.006059	0.004352	0.003157	0.002587	0.002150	0.001892	0.001780	0.001557	0.001456	0.001355	0.001409
15	0.011776	0.006064	0.004335	0.003159	0.002561	0.002155	0.001893	0.001785	0.001591	0.001544	0.001369	0.001325
16	0.011799	0.006047	0.004082	0.003173	0.002518	0.002155	0.001903	0.001799	0.001561	0.001478	0.001374	0.001451
17	0.011825	0.006025	0.004105	0.003154	0.002551	0.002175	0.001962	0.001714	0.001577	0.001515	0.001407	0.001327
18	0.011770	0.006102	0.004070	0.003168	0.002566	0.002185	0.001902	0.001739	0.001632	0.001493	0.001374	0.001310
19	0.011810	0.006014	0.004058	0.003205	0.002538	0.002290	0.001942	0.001779	0.001589	0.001455	0.001416	0.001327
20	0.011765	0.006008	0.004064	0.003103	0.002540	0.002154	0.001915	0.001728	0.001559	0.001510	0.001367	0.001326
21	0.012031	0.006049	0.004077	0.003097	0.002521	0.002201	0.002006	0.001721	0.001561	0.001453	0.001373	0.001312
22	0.011810	0.006036	0.004109	0.003159	0.002525	0.002175	0.001946	0.001714	0.001591	0.001460	0.001514	0.001315
23	0.011771	0.006053	0.004070	0.003099	0.002595	0.002162	0.001893	0.001724	0.001570	0.001517	0.001386	0.001309
24	0.011757	0.006054	0.004114	0.003100	0.002560	0.002201	0.001901	0.001785	0.001597	0.001452	0.001426	0.001308
25	0.012092	0.006042	0.004126	0.003092	0.002776	0.002227	0.002208	0.001719	0.001799	0.001493	0.001372	0.001351
26	0.011758	0.006018	0.004068	0.003140	0.002571	0.002153	0.001912	0.001797	0.001563	0.001538	0.001360	0.001336
27	0.011831	0.006051	0.004073	0.003152	0.002558	0.002204	0.001940	0.001732	0.001579	0.001473	0.001373	0.001311
28	0.011770	0.006075	0.004087	0.003152	0.002578	0.002360	0.001900	0.001721	0.001574	0.001455	0.001377	0.001306
29	0.011773	0.006062	0.004104	0.003152	0.002588	0.002177	0.001895	0.001727	0.001645	0.001463	0.001382	0.001321
30	0.011776	0.005999	0.004074	0.003172	0.002557	0.002155	0.001893	0.002009	0.001560	0.001453	0.001621	0.001523

Tabla 44 Calculo De PI Secuencial

1	0.011737
2	0.011750
3	0.011750
4	0.011750
5	0.011989
6	0.011741
7	0.011748
8	0.011760
9	0.011749
10	0.011749
11	0.011987
12	0.011992
13	0.011750
14	0.011750
15	0.011749
16	0.011750
17	0.011750
18	0.011749
19	0.011751
20	0.011739
21	0.011739
22	0.011751
23	0.011750
24	0.011741
25	0.011741
26	0.011739
27	0.011750
28	0.011739
29	0.011750
30	0.011988

Fibonacci

Tabla 45 Fibonacci Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	32.814402	22.281113	17.532450	15.491555	13.679168	13.644183	12.950393	12.959104	12.810415	12.700982	12.653880	12.600676
2	32.720471	22.238347	17.405722	15.536564	13.792269	13.671224	13.051116	12.956159	12.731610	12.591391	12.693717	12.609801
3	32.747388	22.331923	17.451502	15.487902	13.673373	13.688624	12.964163	13.008446	12.700627	12.626767	12.635565	12.615049
4	32.725856	22.270695	17.402729	15.466612	13.996063	13.661502	13.038151	13.006119	12.702830	12.660247	12.636977	12.665199
5	32.700188	22.268695	17.593387	15.513528	13.696812	13.648228	12.967935	12.976612	12.862078	12.942888	12.653298	12.614624
6	32.766147	22.260761	17.469469	15.468533	13.645518	13.637394	13.011027	12.955547	12.783853	12.686654	12.678524	12.561844
7	32.769326	22.210332	17.441519	15.537501	13.661596	13.661927	12.972419	12.978255	12.757730	12.645310	12.638836	12.580017
8	32.857057	22.251558	17.520805	15.472568	13.732934	13.638057	12.961169	12.978255	12.753255	12.664010	12.651792	12.639920
9	32.785619	22.262240	17.344787	15.534820	13.648203	13.644382	13.036193	12.996337	12.770503	12.709962	12.648493	12.658033
10	32.736223	22.228794	17.370002	15.490191	13.714140	13.686704	13.078678	12.980501	12.705628	12.640304	12.628633	12.611156
11	32.868591	22.290228	17.381965	15.677925	13.906917	13.670013	12.986940	12.981528	12.739459	12.650344	12.616432	12.658408
12	32.714287	22.213875	17.362269	15.587848	13.684986	13.657716	13.232456	12.992425	12.676633	12.625682	12.654891	12.603739
13	32.784171	22.247601	17.408685	15.491035	13.649469	13.667077	12.978712	12.976999	12.746847	12.633766	12.590890	12.613425
14	32.720079	22.371415	17.356867	15.511917	13.679575	13.673340	13.270425	12.963863	12.761787	12.813187	12.651092	12.654743
15	32.717148	22.329485	17.427720	15.531432	13.703475	13.679431	12.981081	13.016800	12.733067	12.916358	12.614309	12.604743
16	32.738199	22.347089	17.457539	15.527775	15.140841	13.678681	13.009560	13.000843	12.691159	12.676367	12.624760	12.598074
17	32.766434	22.354151	17.388111	15.502385	14.091824	13.657044	12.977001	12.983312	12.673704	12.608495	12.673666	12.603223
18	32.683934	22.265220	17.409269	15.508655	13.711681	13.643182	12.989680	12.974319	12.886278	12.654813	12.644020	12.578754
19	32.750698	22.285431	17.404920	15.545985	13.651214	13.694776	13.059445	12.989493	12.893833	12.967543	12.643609	12.543149
20	32.823728	22.246789	17.434174	15.526974	13.700159	13.678238	12.955088	12.958229	12.757781	12.633574	12.625041	12.620300
21	32.727267	22.309998	17.386607	15.541304	13.659313	13.667648	13.023054	13.019943	12.796173	12.623502	12.629113	12.585581
22	32.715684	22.202515	17.456571	15.545005	13.653628	13.639118	12.981721	13.016471	12.738236	12.643992	12.716299	12.591476
23	32.762613	22.353421	17.429688	15.455370	13.691850	13.694660	13.057063	13.060072	12.709435	12.625291	12.646486	12.637677
24	32.744645	22.342372	17.657957	15.479674	13.687314	13.659638	12.960470	12.952670	12.684005	12.597326	12.649706	12.636315
25	32.718381	22.388633	17.368037	15.464081	13.672389	13.675686	12.938646	13.019628	12.734269	12.862684	12.706322	12.613833
26	32.713319	22.358574	17.349700	15.546756	13.722813	13.673273	12.977373	12.970819	12.705407	12.721290	12.653042	12.573547
27	32.759132	22.359491	17.433303	15.480485	13.733788	13.691647	13.018037	13.036098	12.692656	12.582789	12.618080	12.586000
28	32.777281	22.193197	17.955920	16.114938	13.632609	13.634223	12.926900	12.977044	12.679964	12.539946	12.628022	12.618339
29	32.723938	22.279797	17.407212	15.511066	13.664357	13.670745	13.008627	13.001850	12.692707	12.681470	12.614245	12.599532
30	32.710634	22.260819	17.360905	15.480344	13.714594	3.658132	12.948966	12.982362	12.739060	12.644614	12.619868	12.546643

Tabla 46 Fibonacci Secuencial

40.037.951
39.861.293
39.860.661
39.907.378
39.909.006
39.867.819
39.866.597
39.814.938
39.813.999
39.894.539
39.886.902
39.821.283
39.916.540
39.913.262
39.912.978
39.946.059
39.879.610
39.788.571
39.853.459
39.820.318
39.913.637
39.949.256
39.965.236
39.821.021
39.820.130
39.866.457
39.820.714
39.866.851
39.887.074
39.860.211

4.12. Números Primos

Tabla 47 Numeros Primos Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	58.776686	43.142822	31.806367	24.937928	20.582876	17.413373	15.316693	13.393860	12.127851	10.932749	10.396373	9.315297
2	58.776540	43.198494	31.710407	25.011228	21.039324	17.675313	15.552282	13.496409	12.079044	10.937066	10.084662	9.319314
3	58.754904	43.210823	31.726882	24.956058	20.570286	17.517302	15.349149	13.466036	12.084205	10.901864	10.054950	9.322651
4	58.747816	43.362650	31.746722	24.915191	20.556000	17.461125	15.349149	13.506057	12.113090	10.890353	10.187136	9.451522
5	58.723006	43.12365	31.806780	24.917068	20.544231	17.433179	15.792395	13.451207	12.042091	10.861553	10.333462	9.248486
6	58.675676	43.233568	31.713954	24.961208	21.054323	17.550513	15.340151	13.447281	12.058297	10.972612	10.647140	9.381044
7	58.717770	43.146709	31.740032	24.907921	20.540009	17.520919	15.324565	13.453345	12.001336	11.241134	10.123712	9.418507
8	58.669174	43.256433y	31.713545	24.954056	20.555678	17.551492	15.361781	13.454873	12.126453	10.983808	10.175806	9.314062
9	58.745905	43.146213	31.729876	24.969977	20.517069	17.680699	16.690996	13.468454	11.987401	10.979185	10.261571	9.490670
10	58.736813	43.362543	31.801563	24.923228	20.573421	17.481635	15.408131	13.459555	12.103266	10.931883	10.667779	9.329739
11	58.435616	43.144321	31.703284	24.935068	20.519848	17.468155	15.464112	13.489686	12.103266	10.963985	10.061215	9.628654
12	58.734003	43.233435	31.828482	24.981995	20.502233	18.690226	16.073591	13.387644	12.045546	10.947131	10.171138	9.519138
13	58.755710	43.266270	31.721350	24.972417	21.032596	17.554566	15.349375	13.652738	12.044831	10.936241	10.030275	9.359555
14	58.733903	43.147234	31.720943	24.998833	20.540324	17.461867	15.337970	13.460111	12.125342	10.954000	10.093362	9.297141
15	58.736812	43.140965	31.817532	24.967539	20.542431	17.540405	15.337970	13.461433	12.433368	10.961136	10.037807	9.458025
16	58.343636	43.298455	31.821002	24.958977	20.597049	17.420640	15.320387	13.462183	12.080134	10.933974	10.123200	9.363301
17	58.635474	43.210005	31.709802	24.923046	20.510569	17.459012	15.554327	13.492034	12.044824	10.937463	10.091827	9.323076
18	58.727810	43.260974	31.729803	24.913018	20.519323	17.514197	15.465231	13.468077	12.048598	10.898075	10.572807	9.276942
19	58.767720	43.148904	31.717543	24.981446	20.547832	17.576242	15.793478	13.479604	12.073817	11.023222	10.444078	9.408411
20	58.736786	43.260012	31.717658	24.934177	20.519786	17.451690	15.340163	13.460109	12.074145	10.979899	9.962266	9.343881
21	58.755826	43.362650	31.747652	24.923046	20.540945	17.425342	15.349143	13.481014	12.069951	10.931896	10.584761	9.364652
22	58.666616	43.176543	31.820324	24.917870	20.597956	17.469424	15.320387	13.456741	12.067140	10.891796	10.163221	9.305974
23	58.656734	43.149203	31.721154	24.997845	20.519845	17.477344	15.337298	13.417625	12.180172	10.966789	10.164186	9.319225
24	58.766903	43.367608	31.706543	24.968925	20.544857	17.457497	16.073210	13.450569	12.082194	10.950254	10.174325	9.284375
25	58.629653	43.216432	31.746543	24.933121	20.549054	17.494721	15.320393	13.447623	12.076419	11.026212	10.777298	9.252063
26	58.722905	43.194325	31.710043	24.972994	20.578245	17.484025	16.690675	13.452770	12.076419	11.201982	10.200022	9.323474
27	58.736826	43.141604	31.821743	24.972446	20.501834	17.459127	15.337760	13.499547	12.253846	10.915814	10.238043	9.346065
28	58.732345	43.362654	31.720032	24.972420	20.543465	17.459897	15.408356	13.483971	12.044987	10.927298	10.300565	9.408468
29	58.711903	43.192134	31.701245	24.925177	20.578876	17.551006	16.073547	13.460653	11.986488	10.977651	10.508315	9.485505
30	58.877686	43.210654	31.710222	24.946078	20.502236	17.584782	15.362583	13.465703	12.047033	10.921390	10.003497	9.343643

Tabla 48 Numeros Primos Secuencial

56.212.455
56.447.406
56.445.608
56.303.492
56.358.499
56.371.320
56.314.170
56.293.054
56.285.144
56.346.501
56.361.278
56.395.166
56.330.495
56.347.446
56.329.208
56.281.967
56.465.059
56.378.842
56.275.413
56.382.198
56.369.938
56.333.489
56.332.131
56.400.328
56.312.964
56.309.401
56.442.050
56.342.455
56.333.037
56.283.412

4.13. Multiplicacion de matrices For Nowait

Tabla 49 Multiplicacion De Matrices Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	100.98	50.80	33.25	24.86	19.55	16.33	14.31	12.25	11.14	9.79	9.00	8.34
2	101.09	50.91	32.52	24.65	19.61	16.30	14.34	12.52	10.92	9.76	8.92	8.36
3	101.00	50.99	33.07	24.50	19.33	16.20	14.10	12.26	10.88	9.82	9.01	8.35
4	101.02	50.66	33.26	24.50	19.82	16.26	14.06	12.22	10.92	9.72	9.03	8.31
5	100.83	51.00	33.43	24.81	19.73	16.32	14.32	12.22	11.13	9.79	9.07	8.32
6	100.77	50.95	32.03	24.39	19.51	16.31	14.13	12.23	10.93	9.74	8.96	8.29
7	100.89	50.93	32.55	24.38	19.32	16.23	14.33	12.53	10.91	9.83	9.02	8.32
8	100.03	50.86	32.71	24.63	19.62	16.42	14.05	12.23	10.84	9.78	9.06	8.29
9	100.76	50.77	33.22	24.36	19.80	16.37	14.34	12.24	10.91	9.75	9.03	8.30
10	100.14	50.52	32.28	24.40	19.34	16.31	14.00	12.21	11.17	9.71	8.91	8.27
11	99.78	50.94	32.82	24.89	19.58	16.26	14.11	12.24	10.84	9.78	9.01	8.36
12	99.98	50.92	32.03	24.39	19.76	16.32	14.30	12.21	10.91	9.80	9.02	8.39
13	99.63	50.88	32.56	24.32	19.64	16.41	14.05	12.26	10.96	9.84	9.08	8.32
14	100.77	50.82	32.83	24.35	19.51	16.28	14.35	12.22	10.94	9.82	9.10	8.22
15	99.71	50.76	33.41	24.67	19.71	16.36	14.01	12.50	10.92	9.77	8.90	8.32
16	100.61	50.78	32.71	24.37	19.36	16.39	14.14	12.20	10.87	9.72	9.05	8.34
17	100.26	50.91	33.53	24.51	19.55	16.43	14.31	12.26	10.98	9.82	9.04	8.27
18	100.75	50.88	2.19	24.81	19.53	16.22	14.05	12.23	11.10	9.77	9.07	8.33
19	100.47	50.96	32.57	24.52	19.70	16.24	14.01	12.21	10.95	9.80	8.96	8.34
20	100.64	50.87	33.48	24.32	19.53	16.36	14.04	12.20	10.93	9.75	9.02	8.36
21	99.99	50.56	33.42	24.69	19.66	16.34	14.36	12.51	10.91	9.74	9.01	8.30
22	100.57	50.91	32.28	24.57	19.37	16.22	14.12	12.23	11.12	9.78	9.08	8.35
23	100.60	50.67	33.48	24.87	19.57	16.32	14.32	12.20	11.13	9.78	8.92	8.29
24	100.57	50.56	32.59	24.55	19.62	16.30	14.13	12.52	10.92	9.81	9.05	8.33
25	100.47	50.77	33.06	24.31	19.83	16.24	14.30	12.24	10.87	9.77	9.05	8.34
26	99.99	50.87	33.46	24.66	19.37	16.40	14.07	12.23	10.91	9.75	9.07	8.30
27	100.57	50.83	32.28	24.82	19.73	16.43	14.02	12.21	11.16	9.79	8.93	8.32
28	100.60	50.94	33.27	24.50	19.54	16.35	14.12	12.50	10.89	9.82	9.02	8.31
29	100.57	50.92	33.08	24.58	19.81	16.26	14.33	12.22	10.91	9.76	9.03	8.21
30	100.47	50.95	32.51	24.61	19.64	16.23	14.35	12.23	11.11	9.79	8.90	8.32

Tabla 50 Multiplicacion De Matrices Secuencial

51.79
51.70
51.68
51.74
51.55
51.61
51.60
51.60
51.54
51.57
51.58
51.69
51.63
51.59
51.61
51.63
51.68
51.59
51.54
51.63
51.66
51.56
51.58
51.60
51.66
51.67
51.61
51.67
51.74
51.68

4.14. Critical

Tabla 51 Critical Con OpenMP

	1	2	3	4	5	6	7	8	9	10	11	12
1	0.000592	0.000321	0.000204	0.000158	0.000135	0.000118	0.000104	0.00009	0.000091	0.000089	0.000078	0.000082
2	0.000590	0.000299	0.000219	0.000166	0.000137	0.000116	0.000106	0.000095	0.000089	0.000084	0.000077	0.000076
3	0.000603	0.000299	0.000206	0.000159	0.000135	0.000113	0.000096	0.000094	0.000087	0.000080	0.000075	0.000073
4	0.000590	0.000301	0.000205	0.000159	0.000138	0.000112	0.000105	0.000098	0.000089	0.000081	0.000081	0.000079
5	0.000590	0.000300	0.000205	0.000158	0.000128	0.000113	0.000106	0.000089	0.000085	0.000084	0.000079	0.000077
6	0.000590	0.000301	0.000204	0.000158	0.000136	0.000119	0.000106	0.000094	0.000089	0.000081	0.000076	0.000071
7	0.000604	0.000301	0.000204	0.000157	0.000137	0.000118	0.000105	0.000094	0.000091	0.000085	0.000082	0.000077
8	0.000591	0.000301	0.000205	0.000156	0.000130	0.000117	0.000099	0.000097	0.000090	0.000084	0.000078	0.000074
9	0.000592	0.000300	0.000207	0.000169	0.000137	0.000120	0.000103	0.000095	0.000089	0.000084	0.000077	0.000076
10	0.000590	0.000302	0.000236	0.000157	0.000128	0.000129	0.000103	0.000096	0.000096	0.000078	0.000074	0.000075
11	0.000591	0.000311	0.000221	0.000157	0.000128	0.000116	0.000104	0.000098	0.000088	0.000082	0.000079	0.000083
12	0.000591	0.000302	0.000203	0.000167	0.000136	0.000117	0.000103	0.000090	0.000088	0.000083	0.000093	0.000076
13	0.000590	0.000300	0.000204	0.000158	0.000129	0.000117	0.000099	0.000095	0.000084	0.000085	0.000080	0.000078
14	0.000602	0.000313	0.000206	0.000165	0.000128	0.00113	0.000103	0.000095	0.000087	0.000076	0.000074	0.000073
15	0.000591	0.000304	0.000204	0.000157	0.000128	0.000118	0.000103	0.000093	0.000091	0.000081	0.000080	0.000072
16	0.000591	0.000301	0.000219	0.000168	0.000136	0.000111	0.000105	0.000095	0.000086	0.000081	0.000076	0.000079
17	0.000590	0.000301	0.000205	0.000170	0.000129	0.000118	0.000104	0.000091	0.000087	0.000086	0.000075	0.000074
18	0.000591	0.000301	0.000206	0.000166	0.000128	0.000118	0.000105	0.000095	0.000081	0.000080	0.000079	0.000071
19	0.000590	0.000300	0.000215	0.000170	0.000128	0.000118	0.000099	0.000092	0.000086	0.000087	0.000076	0.000076
20	0.000590	0.000314	0.000204	0.000156	0.000131	0.000119	0.000106	0.000098	0.000083	0.000087	0.000082	0.000076
21	0.000605	0.000324	0.000220	0.000166	0.000137	0.000113	0.000102	0.000096	0.000085	0.000083	0.000080	0.000078
22	0.000591	0.000301	0.000220	0.000156	0.000129	0.000118	0.000104	0.000094	0.000089	0.000079	0.000078	0.000074
23	0.000591	0.000300	0.000204	0.000159	0.000129	0.000116	0.000097	0.000095	0.000082	0.000086	0.000081	0.000078
24	0.000591	0.000323	0.000205	0.000166	0.000137	0.000113	0.000105	0.000093	0.000082	0.000082	0.000080	0.000081
25	0.000591	0.000300	0.000204	0.000169	0.000130	0.000110	0.000098	0.000094	0.000089	0.000083	0.000079	0.000077
26	0.000591	0.000323	0.000204	0.000166	0.000139	0.000112	0.000103	0.000095	0.000089	0.000081	0.000078	0.000081
27	0.000590	0.000321	0.000217	0.000169	0.000130	0.000120	0.000104	0.000092	0.000088	0.000086	0.000085	0.000074
28	0.000604	0.000300	0.000205	0.000156	0.000141	0.000112	0.000105	0.000099	0.000091	0.000079	0.000081	0.000074
29	0.000590	0.000300	0.000218	0.000165	0.000136	0.000119	0.000102	0.000098	0.000087	0.000083	0.000080	0.000077
30	0.000590	0.000304	0.000220	0.000157	0.000128	0.000117	0.000106	0.000097	0.000090	0.000083	0.000078	0.000076

Tabla 52 Critical Secuencial

0.000626
0.000584
0.000584
0.000584
0.000584
0.000598
0.000626
0.000626
0.000584
0.000618
0.000626
0.000584
0.000626
0.000584
0.000584
0.000584
0.000615
0.000585
0.000584
0.000584
0.000584
0.000584
0.000626
0.000833
0.000584
0.000584
0.000584
0.000584
0.000584
0.000584

4.15. Ecuacion De La Onda Con OpenMP

Tabla 53 Tiempos(mseg) De La Ecuacion De La Onda

	1	2	3	4	5	6	7	8	9	10	11	12
1	127.477.438	66.214.843	49.965.587	29.679.749	23.354.003	21.229.917	20.193.703	16.416.578	15.974.926	13.210.308	12.911.622	11.567.591
2	118.078.395	47.518.133	49.861.100	31.952.383	28.346.777	21.324.352	19.898.942	13.913.202	13.046.966	13.365.973	12.815.374	10.672.373
3	117.455.770	47.518.226	50.627.645	31.283.582	23.108.232	18.834.228	20.010.028	15.812.377	13.166.751	12.347.891	12.356.670	9.896.832
4	117.186.822	47.510.308	48.031.703	23.752.292	28.226.490	18.748.579	20.011.636	16.684.572	12.388.618	12.982.486	11.188.557	10.901.511
5	116.685.389	47.375.106	48.950.205	27.585.253	28.548.237	21.150.643	20.428.586	16.648.772	12.389.625	13.144.505	11.088.195	9.866.296
6	116.464.475	47.395.719	49.791.151	30.110.204	28.056.949	21.338.177	19.232.439	16.006.364	13.183.304	13.740.532	13.243.856	10.016.485
7	115.845.273	47.309.006	48.420.462	24.133.257	28.529.450	20.917.900	19.885.976	15.402.345	13.692.043	13.879.470	13.196.363	9.876.854
8	115.582.074	47.408.575	49.929.959	24.174.336	26.611.214	20.457.938	20.220.917	16.109.171	13.135.268	13.802.097	12.160.528	10.301.371
9	115.163.570	47.366.350	50.938.795	28.592.465	29.213.713	19.324.495	20.104.726	15.743.867	15.756.177	13.658.483	10.615.373	9.371.266
10	114.809.623	47.481.757	50.056.032	23.948.340	22.980.130	19.088.827	19.984.227	16.838.091	12.905.011	12.757.136	10.573.228	9.893.438
11	114.593.127	47.390.676	50.361.014	27.213.275	27.607.961	21.197.670	18.203.228	15.706.767	14.695.014	12.748.113	10.540.765	11.068.358
12	114.428.748	47.292.421	49.034.330	24.072.576	27.479.522	20.678.831	19.878.120	15.899.576	13.064.441	12.905.717	12.468.024	10.493.458
13	114.255.452	47.346.679	49.282.033	27.734.880	29.305.040	20.384.391	19.235.257	15.456.242	12.288.201	13.019.909	12.104.762	9.892.115
14	114.145.182	47.338.671	49.924.860	31.626.244	29.079.282	19.365.953	19.727.589	15.445.487	12.437.141	13.264.091	13.001.843	11.313.902
15	114.040.831	47.545.033	49.354.892	31.825.618	29.675.723	20.745.181	19.621.599	16.042.758	13.604.284	12.839.593	11.047.865	9.706.559
16	113.517.397	47.639.878	49.537.413	24.134.600	27.155.855	20.811.578	19.698.022	15.855.706	15.573.692	12.800.399	12.043.715	10.319.079
17	113.048.010	47.562.512	51.008.582	23.603.438	28.499.811	20.540.330	20.478.092	16.482.059	14.463.390	11.633.532	12.134.142	9.136.576
18	112.542.131	47.235.420	47.620.751	30.123.786	28.940.926	19.943.925	19.287.896	16.109.364	12.956.481	12.194.472	12.749.574	9.945.875
19	109.432.101	47.643.727	49.393.178	24.231.281	23.402.706	20.766.539	19.747.810	16.279.278	13.161.604	12.343.279	10.731.964	9.439.499
20	109.465.938	47.403.311	50.315.053	27.025.601	23.096.171	20.944.302	19.808.908	16.035.485	12.724.960	12.793.298	11.154.549	9.785.512
21	109.429.632	47.557.783	49.352.856	31.035.062	23.574.807	21.329.484	20.196.908	15.546.999	15.322.557	12.060.686	12.907.772	10.813.076
22	109.465.000	47.385.010	48.894.143	27.869.425	28.149.904	19.656.371	17.351.791	15.655.446	15.139.140	12.844.669	11.973.013	9.370.017
23	122.863.906	47.400.800	49.910.138	29.168.721	29.171.267	19.454.966	19.681.392	15.891.421	14.990.380	12.512.146	12.344.088	9.949.553
24	135.057.942	47.325.716	49.383.835	32.437.933	28.739.671	20.478.509	19.999.040	16.142.967	12.672.832	13.449.476	11.755.158	10.433.395
25	114.625.877	47.302.069	48.825.397	25.024.707	28.761.377	21.096.973	20.350.357	13.974.868	13.162.783	12.755.126	11.310.555	9.859.364
26	114.005.431	47.282.825	49.242.090	23.647.312	28.980.573	20.067.542	19.420.282	14.897.097	12.692.348	11.779.281	10.930.930	11.282.216
27	109.443.200	47.651.565	48.723.967	25.498.264	29.348.054	18.916.761	19.623.773	16.089.601	15.680.396	12.431.980	12.121.010	9.882.480
28	116.555.452	47.029.387	50.569.537	28.919.328	29.574.647	21.535.890	16.889.633	16.122.692	15.471.371	13.473.370	11.834.554	10.459.750
29	109.464.638	47.364.317	48.579.771	23.606.078	24.317.916	20.746.152	17.583.881	15.325.579	12.257.555	13.504.381	10.752.622	9.782.629
30	109.234.632	47.154.171	48.979.556	24.052.733	25.740.971	19.365.047	19.623.057	16.533.287	12.314.462	12.562.347	12.181.584	11.613.502