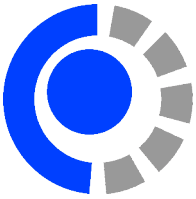
Universidad Nacional del Comahue

Facultad de Informática



Tesis de Licenciado en Ciencias de la Computación

Optimización e implementación de multiprocesamiento para una aplicación legacy de Dinámica de Fluidos

Andrés José Huayquil

Director: Lic. Eduardo Grosclaude

Neuquén Argentina

2017

Prefacio

Esta tesis es presentada como parte de los requisitos finales para optar al grado académico de *Licenciado en Ciencias de la Computación*, otorgado por la Universidad Nacional del Comahue, y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otras. La misma es el resultado de la investigación llevada a cabo en el Departamento de Ingeniería de Computadoras, de la Facultad de Informática, en el período comprendido entre Enero de 2014 y Septiembre de 2016, bajo la dirección de Lic. Eduardo Grosclaude.

Andrés José Huayquil
Facultad de Informática
Universidad Nacional del Comahue
Neuquén, 29 de Septiembre de 2016.

UNIVERSIDAD NACIONAL DEL COMAHUE

Facultad de Informática

La presente tesis ha sido aprobada el día ....., mereciendo la calificación de .....

Dedicatorias

Este trabajo está dedicado a mi esposa, por su paciencia y apoyo en estos años.

A mis padres y mi hermana, que me enseñaron a valorar los estudios y me motivaron a seguir adelante. Y en especial a mi padre que nos espera en la próxima existencia.

Y también a los amigos que han sabido dar su ayuda y consejo en el momento justo.

Resumen

Denominamos “Legacy Software”, o “Software Heredado”, a programas vigentes tras veinte, treinta y hasta cuarenta años. Generalmente son programas grandes, complejos, que desempeñan una tarea crucial en la organización a la que pertenecen. Particularmente en las áreas científicas, con problemas de cálculo intensivo, una vez que un programa arroja resultados correctos, no suelen existir modificaciones al código. Uno de los lenguajes de programación mas ampliamente adoptados por la comunidad científica es Fortran y justamente en este lenguaje están programados gran cantidad de los problemas de cómputo intensivo que han devenido en sistemas legacy.

Las aplicaciones legacy, o heredadas, debido al paso de una cierta cantidad de tiempo, enfrentan finalmente la problemática de dar respuesta a cambios ambientales, donde deben ser modernizadas o considerar terminar su ciclo de vida. El trabajo de modernizar un sistema legacy puede tener una envergadura variable, dependiendo de la complejidad del sistema y del nuevo ambiente donde vaya a funcionar. La modernización de una aplicación legacy puede verse como un proceso de optimización de la aplicación, sólo que para una plataforma diferente de aquella para la cual fue construida.

Esta tesis presenta la modernización de una aplicación científica del campo de la Dinámica de Fluidos, la cual se encuentra dentro de las aplicaciones de Cómputo de Altas Prestaciones (HPC en inglés), desarrollada entre mediados y fines de la década de 1990, escrita en lenguaje Fortran para una plataforma de computación con recursos limitados. Se describe el proceso de optimización de la aplicación, pasando en primer instancia por una optimización serial y luego por una optimización paralela, de manera que pueda aprovechar recursos que no estaban contemplados en su diseño original. La implementación de multiprocesamiento se realiza con la interfaz de programación paralela OpenMP. Se modifica el código lo menos posible para asegurar que el autor y usuario de la aplicación pueda seguir manteniéndola.

El proceso de optimización e implementación de multiprocesamiento se ilustra con pruebas de ejecución de las distintas versiones de la aplicación: versión original, versión optimizada serialmente y versión optimizada paralelamente, observando la mejora en los tiempos de ejecución y el impacto en la utilización de los recursos computacionales con distintos tamaños de problema.

Abstract

Programs which are still running after several decades are usually called Legacy Software. They are usually big, complex programs performing crucial tasks in their owner organization. Once a program is found to yield correct results, its code is seldom modified, especially in scientific areas, where compute-intensive problems often belong. Fortran is one of the most widely adopted languages in the scientific community. Many compute-intensive problems having become legacy systems are programmed in this language.

Due to the passing of time, legacy applications eventually face environmental changes which they must account for, at which time they must be modernized, or their end of life has to be considered. Modernizing a legacy system entails a work of a variable size, depending on the system's complexity and the new proposed environment. Modernizing a legacy application can be considered as an optimization process, only for a different platform than that it was built for.

This thesis introduces the modernization of a scientific application taken from the field of Fluid Dynamics, a discipline commonly related to High Performance Computing. This application was developed in the late 90s, in Fortran, for a rather resource-limited computing platform. Here the optimization process is described, starting with serial optimization, and then performing parallel optimization, so that new resources, not envisioned in the original design, can be leveraged. Multiprocessing is implemented through the OpenMP parallel programming interface. The application's code has been modified as little as possible so as to ensure that the application's author and user may continue to maintain it.

The development of optimization and multiprocessing is illustrated with execution performance tests at several stages of the process: original, serially optimized and parallel optimized versions, characterizing the execution speedup obtained and the utilization of computational resources at distinct problem sizes.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Sistemas legacy o heredados

Todos los sistemas de software deben, eventualmente, dar respuesta a cambios ambientales. El desafío del cambio en los recursos de computación disponibles no solamente se presenta en forma de restricciones, sino que al contrario, más y mejores recursos disponibles comprometen la eficiencia de los sistemas y ponen al descubierto limitaciones o vulnerabilidades de diseño que no eran evidentes en el momento en que se crearon.

Las aplicaciones *legacy*, o heredadas, debido al paso de una cierta cantidad de tiempo, enfrentan finalmente esta problemática en forma crítica, y alguien debe hacerse cargo de modernizarlas, o considerar dar por terminado su ciclo de vida y reemplazarlas. El trabajo de modernizar un sistema *legacy* puede tener una envergadura variable, dependiendo de la complejidad del sistema y del nuevo ambiente donde vaya a funcionar. La modernización de una aplicación *legacy* puede verse como un proceso de optimización de la aplicación, sólo que para una plataforma diferente de aquella para la cual fue construida.

1.2. Aplicación seleccionada

Este trabajo de tesis aborda el problema de la modernización de una aplicación científica del campo de la Dinámica de Fluidos. El programa objeto de optimización de esta tesis es de autoría de Ricardo A. Prado, docente e investigador de la Universidad Nacional del Comahue, y fue utilizado para obtener resultados para su trabajo de tesis de doctorado [Pra07] en el área de Ingeniería presentado en la Universidad de Buenos Aires en 2007. El programa analiza el comportamiento fluidodinámico de un tipo particular de turbomáquina, la turbina eólica de eje horizontal. Fue desarrollado entre los años 1999 y 2005, en un momento en el cual los recursos de computación eran restrictivos en comparación con los de hoy. Por otro lado, por el tipo de tarea que desarrolla, se encuentra entre las aplicaciones de la Computación de Altas Prestaciones (HPC, High Performance Computing) y desde que fue escrita esta aplicación, las características del software y el hardware disponibles para esta clase de actividades han avanzado notablemente.

Este programa fue construido en su momento para la plataforma computacional típica que estaba entonces al alcance de los pequeños grupos de investigación. Estos mismos grupos hoy ven la posibilidad de acceder a plataformas de características sumamente diferentes. Este trabajo de tesis intenta optimizar y modernizar la aplicación para que puedan ser aprovechados recursos que no estaban previstos en su diseño original, pero con los que hoy pueden contar sus usuarios. En especial nos referimos a las capacidades de multiprocesamiento de los equipos actuales, la mayor cantidad de memoria principal, y las nuevas capacidades de los compiladores que acompañan estos desarrollos arquitectónicos.

Como se expone en [Pra07], dicha tesis "analiza el comportamiento fluidodinámico de una turbomáquina particular: la turbina eólica". La creación del programa en estudio se justifica

en el mismo trabajo indicando que "debido a la complejidad de las ecuaciones de gobierno en ambas zonas del campo fluidodinámico, como así también de la geometría de la turbina y de sus condiciones de operación, se requiere de procesos de resolución numérica adecuados, los cuales se incorporaron en los códigos computacionales que se desarrollaron a tal efecto" [Pra07], y si bien existen otras soluciones para lo indicado, la realización del programa formaba parte del trabajo de doctorado.

La primer parte del trabajo de Prado es sobre el flujo inviscido (de viscosidad despreciable, casi nula) alrededor de la pala de la turbina. Para determinar dicho flujo, el modelo numérico de dicha tesis se formula a través del método de los paneles "por el cual la pala es discretizada mediante un número finito de paneles cuadrilaterales, los cuales son adecuadamente distribuídos a lo largo de la envergadura y cuerda de la misma" [Pra07]. Además indica que "asociado a cada panel se encuentra un anillo vorticoso de intensidad de circulación constante, conformado por segmentos de hilos vorticosos rectos. Estos anillos configuran el sistema vorticoso ligado a la pala" [Pra07].

El programa en estudio en esta tesis realiza el cálculo de las velocidades inducidas en un punto del panel para cada uno de los paneles. El cálculo es realizado mediante la ley de Biot-Savart la cual es una de las leyes fundamentales de la magnetoestática e indica el campo magnético creado por corrientes eléctricas estacionarias. Se establece una equivalencia entre Electromagnetismo y la Mecánica de Fluidos para aplicar Biot-Savart, considerando que un hilo de corriente equivale a un hilo vorticoso; a partir de esto calcula las velocidades no viscosas sobre dicho hilo vorticoso.

La ecuación principal resuelta por el programa es la del vector de velocidad inducida en cada uno de los puntos de colocación de los N paneles, donde dicho vector tiene tres componentes:

$$V_{iX}|_{pc}^{k} = \sum_{j=1}^{N_p} Ci_{Xkj}\gamma_j \qquad V_{iY}|_{pc}^{k} = \sum_{j=1}^{N_p} Ci_{Ykj}\gamma_j \qquad V_{iZ}|_{pc}^{k} = \sum_{j=1}^{N_p} Ci_{Zkj}\gamma_j \tag{1.1}$$

Mediante la integración de la Ley de Biot-Savart a lo largo de la longitud de cada filamento del sistema vorticoso completo se determinan las tres componentes cartesianas de las velocidades inducidas en cada punto P definido. El proceso de integración numérica se realiza por el método de Simpson. La regla o método de Simpson es un método de integración numérica que se utiliza para obtener la aproximación de una integral en un intervalo definido, al dividir ese intervalo en subintervalos y aproximar cada subintervalo con un polinomio de segundo grado. En el trabajo de Prado se indica que "El proceso de integración numérica de la ley de Biot-Savart para la obtención de los coeficientes de influencia de cada hilo de la estela sobre cada punto de control, [...], se realizó mediante el método de Simpson, considerando hilos de longitud finita que partían desde el borde de fuga de la pala hasta una coordenada axial igual a 40 radios R [...], y dividiendo dichos hilos en 2000 segmentos correspondientes a incrementos constantes del parámetro del helicoide" [Pra07].

Analizando el programa en sí, podemos indicar que está escrito en lenguaje Fortran, utilizando una estructura de programación secuencial y monolítica. Presenta un cuerpo principal con múltiples subrutinas, entre las cuales se pueden identificar dos de ellas que realizan el cálculo más intensivo, *estela* y *solgauss*, donde la primera realiza el cálculo de los coeficientes de influencia de los hilos libres Cix, Ciy y Ciz, y la segunda resuelve el sistema de ecuaciones para determinar las vorticidades de los paneles mediante el metodo de eliminación de Gauss para matrices. En estas subrutinas es en donde se encuentran las mayores posibilidades de optimización y paralelización para nuestro trabajo de tesis.

1.3. Objetivos y Motivación

En este trabajo se busca principalmente la optimización a través de paralelización del programa presentado, utilizando herramientas modernas sobre código *legacy*, tales como OpenMP y OpenMPI.

Se debe tener en cuenta que el programa es totalmente deterministico, es decir que cada ejecución sucesiva arroja siempre los mismos resultados. De acuerdo a lo indicado en conversaciones con el autor del programa, se observaban tiempos de ejecución altos, de los cuales no se tienen registros. En el caso de tamaño de datos de 50x50 (50 filas de 50 paneles) tardaba unas horas y en el caso de tamaño 80x80 (80 filas de 80 paneles), debido a un elevado tiempo de ejecución, se dejaba en ejecución de un día para el otro. Debido a esta falta de registros de tiempos, en este trabajo de tesis se realizaron nuevas ejecuciones del programa con ambos tamaños de datos para obtener tiempos de referencia con respecto a los siguientes pasos de optimización.

Por esto es que otro objetivo importante es la mejora en los tiempos de ejecución del programa a través del proceso de optimización y paralelización, esperando alcanzar ejecuciones en la mitad del tiempo (50 % menos) que el programa original, comparando las mediciones directamente contra las nuevas ejecuciones del código original.

Como el programa realiza el almacenamiento de datos temporales y resultados, tanto parciales como finales, en archivos planos debido a restricciones de memoria en la arquitectura original de ejecución del código, se pretende también mejorar la utilización de los recursos, maximizando el uso de memoria RAM, que actualmente es de mayor tamaño a la utilizada en 1999, y minimizando el uso de disco, buscando así mejorar el desempeño del programa.

Por lo expresado es que la optimización buscada tiene en cuenta las nuevas arquitecturas paralelas, así como la mayor disponibilidad de memoria principal en las nuevas plataformas. Se propondrá una optimización del código secuencial y a continuación una solución de ejecución paralela.

Se mantendrán las condiciones actuales de uso para el usuario. En particular, no se modificará el lenguaje de programación, de modo que el usuario y autor de la aplicación conserve la capacidad de modificar la misma.

Esta tesis surge de la necesidad de adaptar y optimizar el código *legacy*, utilizando nuevas técnicas de programación disponibles, nuevas estructuras de datos o aplicando nuevos recursos del lenguaje original en el que fue escrito. Además de permitir al código aprovechar los recursos de la máquina sobre la que se ejecuta.

En particular el código de Computo de Altas Prestaciones posee ejemplos de código *legacy* que puede y necesita ser adaptado y optimizado, dentro de la dinámica de fluidos como otros campos de la denominada Ciencia e Ingeniería Computacional. Se ampliaran estos conceptos en los siguientes capítulos.

Por último pero no menos importante, se busca que se pueda aprovechar las posibilidades de paralelización que brindan las nuevas arquitecturas disponibles en las PC actuales, permitiendo a grupos de investigación que no poseen equipamiento especializado para realizar los cálculos de sus trabajos poder obtener mejores resultados en equipos de uso general.

1.4. Marco de trabajo

El autor de la tesis que se presenta es integrante del proyecto de investigación 04/F002, Computación de Altas Prestaciones, Parte II, dirigido por la Dra. Silvia Castro. La tesis ha sido desarrollada como aporte a los objetivos de dicho proyecto:

- De formación de recursos humanos
  - Adquirir conocimientos teóricos y prácticos para el diseño, desarrollo, gestión y mejora de las tecnologías de hardware y software involucradas en la Computación de Altas Prestaciones y sus aplicaciones en Ciencia e Ingeniería Computacional.
  - Formar experiencia directa en temas relacionados con Cómputo de Altas Prestaciones, en particular, optimización y paralelización de aplicaciones científicas.
- De transferencia

- Relevar los requerimientos de Computación de Altas Prestaciones de otros investigadores en Ciencias e Ingeniería de Computadoras y cooperar en su diagnóstico y/o resolución.
- Detectar necesidades relacionadas con Computación de Altas Prestaciones en otras entidades del ámbito regional y plantear correspondientes actividades de transferencia.

1.5. Organización de la Tesis

A continuación se describe sintéticamente el contenido del resto de los capítulos comprendidos en esta Tesis.

- Capítulo 2

  Se presenta una revisión de los fundamentos y la aplicación de la Computación Paralela. Se presenta Fortran, el lenguaje de programación con el cual está implementada la aplicación seleccionada. Se describe la interfaz de programación paralela OpenMP utilizada para la solución propuesta. Se explica el proceso de optimización de una aplicación secuencial hacia una aplicación paralelizada, y por último se presenta brevemente la aplicación motivo de este trabajo de tesis.

- Capítulo 3

  Presenta el proceso de solución aplicado, dividido en dos etapas, la optimización del código Fortran y la paralelización aplicando OpenMP. Se presentan también los problemas encontrados en el proceso inherentes a la arquitectura de máquina y a la estructura de la aplicación seleccionada.

- Capítulo 4

  Se presentan ejemplos de ejecución de la solución propuesta en el capítulo 3 y comparaciones de resultados obtenidos.

- Capítulo 5

  Se presentan las conclusiones del trabajo, así como el análisis de los resultados obtenidos al aplicar la solución propuesta. Además se identifican posibles futuros trabajos derivados de esta tesis.

Capítulo 2

Analisis conceptual

2.1. Introducción

En el pasado reciente, la herramienta más común para la solución de problemas de las ciencias computacionales fue la programación en lenguajes orientados al cómputo científico, como Fortran. Cuando los recursos de computación (por ejemplo, con la aparición de la computadora personal) se hicieron accesibles a los investigadores individualmente y a grupos de investigación de modestos presupuestos, esta herramienta se hizo popular y creó un modo de trabajo estándar de facto, ampliamente extendido, en las ciencias e ingenierías.

Con frecuencia, las soluciones producidas por este modo de trabajo eran programas grandes, secuenciales y monolíticos. Los problemas abordados por esta clase de programas se caracterizan por grandes demandas de cómputo y de memoria, recursos especialmente escasos desde los comienzos de la computación. Las decisiones de programación no resultaban siempre eficientes, debido a que su autor no siempre era un profesional del área informática, sino el científico que necesitaba la solución.

La evolución técnica y económica de los sistemas disponibles para los investigadores permitió la incorporación de plataformas paralelas, primero con la posibilidad de distribuir el cómputo entre varios equipos de computación a través de una red, luego con diferentes formas de arquitecturas paralelas de hardware. En estas plataformas, la multiplicidad de los recursos enfrenta al programador con un problema de programación y de administración de recursos aún más complejo.

En este capítulo se presenta el tratamiento de estos problemas mediante la utilización de programación paralela. En la sección 2.2 se explicará en qué consiste la programación paralela, explicando sintéticamente cómo funciona un programa paralelo, cuáles son las plataformas más utilizadas de computación paralela y sus modelos de comunicación (memoria compartida, pasaje de mensajes). En la sección 2.3 se explicará brevemente el Cómputo de Altas Prestaciones. En la sección 2.4 se presentará brevemente el lenguaje de programación Fortran y se explicará su uso a nivel científico. En las secciones 2.5 y 2.6 se presentan la interfaz de programación de aplicaciones OpenMP y su implementación en Fortran respectivamente. En la sección 2.7 se mostrará en qué consiste el proceso de optimización para paralelizar una aplicación y por último en la sección **??** se presentará la aplicación científica núcleo de esta Tesis y se describirá muy brevemente el problema que resuelve.

2.2. Visión general del procesamiento paralelo

La fuerza impulsora detrás del procesamiento paralelo es lograr completar un trabajo mucho más rápidamente, al desempeñar múltiples tareas simultáneamente. La taxonomía de Flynn [WC00] es un modelo simple para considerar las arquitecturas de computadoras capaces de procesamiento paralelo.

La taxonomía, formulada en la década de los 60, ofrece una categorización completamente general de las diferentes arquitecturas de las computadoras. Propone clases de máquinas que son determinadas por la multiplicidad del flujo de instrucciones y del flujo de datos que la computadora puede procesar en un instante dado. Según el modelo, todas las computadoras pueden ser ubicadas en alguna de las siguientes cuatro clases:

- **Una instrucción, un dato (SISD)**: Constituyen un sistema de cómputo de una sola unidad de procesamiento con un único flujo de instrucciones y un único flujo de datos. Las computadoras personales, desde las últimas décadas del s. XX hasta el surgimiento de la llamada *revolución del multinucleo*, pertenecieron a esta clase.

- **Múltiples instrucciones, un dato (MISD)**: El mismo conjunto de datos es tratado por múltiples unidades de procesamiento. No se conocen sistemas de esta clase que hayan sido construidos para venta al público o en otro ámbito.

- **Una instrucción, múltiples datos (SIMD)**: Varios flujos de datos son procesados por varios procesadores, cada uno de ellos ejecutando el mismo flujo de instrucciones. Las máquinas de esta clase suelen tener un procesador que ejecuta el flujo único de instrucciones y despacha subconjuntos de esas instrucciones a todos los demás procesadores (que generalmente son de diseño mucho más simple). Cada uno de estos procesadores "esclavos" ejecuta su propio flujo de datos. Se aplican especialmente a problemas que requieren las mismas operaciones sobre una gran cantidad de datos, como ocurre con los problemas del álgebra lineal, pero no se desempeñan bien con flujos de instrucciones con ramificaciones (que es caso de la mayoría de las aplicaciones modernas). Las máquinas vectoriales y las GPUs (Graphics Processing Units) son ejemplos de esta clase.

- **Múltiples instrucciones, múltiples datos (MIMD)**: Esta denominación se aplica a máquinas que poseen múltiples procesadores capaces de ejecutar flujos de instrucciones independientes, usando flujos de datos propios y separados. Estas máquinas son mucho más flexibles que los sistemas SIMD. En la actualidad hay cientos de diferentes máquinas MIMD disponibles. Las computadoras multicore actuales y los clusters de procesadores son ejemplos de esta clase.

2.2.1. Modelos Paralelos

Desde la perspectiva del sistema operativo, hay dos medios importantes de conseguir procesamiento paralelo: múltiples procesos y múltiples hilos. Un proceso es un programa en ejecución bajo control de un sistema operativo con un conjunto de recursos asociados. Estos recursos incluyen, pero no están limitados, a estructuras de datos con información del proceso, al menos un hilo de ejecución y un espacio de direcciones virtuales conteniendo las instrucciones del programa y datos. Un hilo es un camino de ejecución o flujo de control independiente dentro de un proceso, compuesto de un contexto (que incluye un conjunto de registros) y una secuencia de instrucciones a ejecutar [WC00].

En los sistemas de computación actuales existen distintos niveles de paralelismo; por ejemplo, los procesadores VLIW y los RISC superescalares alcanzan paralelismo en el nivel de instrucción (ejecutando varias instrucciones de bajo nivel al mismo tiempo) [GG+03, WC00].
Para este trabajo de tesis utilizamos el término "procesamiento paralelo" para indicar que hay más de un hilo de ejecución ejecutándose en un único programa. Esta definición admite la implementación de procesamiento paralelo con más de un proceso. Podemos así considerar el procesamiento paralelo en tres categorías [WC00]:

- "Paralelismo de procesos": usar más de un proceso para desempeñar un conjunto de tareas.

- "Paralelismo de hilos": usar múltiples hilos dentro de un único proceso para ejecutar un conjunto de tareas.

- "Paralelismo híbrido": usar múltiples procesos, donde al menos uno de ellos es un proceso paralelo con hilos, para desempeñar un conjunto de tareas.

Si bien se puede obtener paralelismo con múltiples procesos, existen al menos dos razones potenciales para considerar paralelismo de hilos: conservación de recursos del sistema y ejecución más rápida. Los hilos comparten acceso a los datos del proceso, archivos abiertos y otros atributos del proceso. Compartir datos e instrucciones puede reducir los requerimientos de recursos para un trabajo en particular. Por el contrario, una colección de procesos a menudo deberá duplicar las áreas de datos e instrucciones en memoria para un trabajo.

2.2.1.1. Conceptos básicos de hilos

La administración de hilos es más simple que la de los procesos, ya que los hilos no poseen todos los atributos de un proceso. Pueden ser creados y destruidos mucho más rápidamente que un proceso. Los hilos tienen otros atributos importantes, relacionados con el desempeño [WC00].

Un hilo ocioso es aquel que no tiene procesamiento para hacer y está esperando su próximo conjunto de tareas. Por lo general el hilo es puesto en estado de espera (wait) de acuerdo con una variable de control, que puede tener dos valores, bloqueado o desbloqueado (debe esperar o puede continuar procesando, respectivamente).

Los hilos ociosos pueden ser suspendidos o quedar en espera activa ("spinning" o "busy waiting") [WC00]. Los hilos suspendidos liberan el procesador donde se estaban ejecutando. Los que están en espera activa comprobarán repetidamente la variable para ver si ya están desbloqueados, sin liberar el procesador para otros procesos. Como resultado de esto el rendimiento del sistema puede degradarse drásticamente. Sin embargo, reiniciar un hilo suspendido puede llevar cientos, si no miles, de ciclos de procesador.

Otro atributo de los hilos es la afinidad, que consiste en la posibilidad de ligar un hilo con un mismo procesador al ser reiniciado luego de una suspensión [WC00]. La capacidad de manejar la afinidad permite mantener al hilo, siempre que sea posible, ejecutando sobre el mismo dispositivo de cómputo cada vez que vuelve al modo activo. De esta manera el hilo aprovecha los datos que habían sido puestos en cache durante su ejecución previa. De otro modo, debería volver a cargar todos los datos necesarios para reanudar su ejecución desde la memoria a la cache del nuevo procesador, con el costo de sobrecarga correspondiente.

2.2.1.2. Hilos POSIX

Recién en 1995 se estableció un estándar para la programación de hilos, a pesar de que los sistemas operativos ya venían implementando hilos. El estándar es parte de la normativa "POSIX (Portable Operating System Interface)", en particular la porción POSIX 1003.1c[1] incluye las funciones y las Interfaces de Programación de Aplicación ("APIs") que soportan múltiples flujos de control dentro de un proceso. Los hilos creados y manipulados vía este estándar son generalmente indicados como "pthreads". Con anterioridad al establecimiento de este estándar, las APIs de hilos eran específicas del fabricante del hardware, lo que hacía muy difícil la portabilidad de las aplicaciones paralelas con hilos. Combinado con la complejidad de reescribir aplicaciones para utilizar y aprovechar el control explícito de hilos, el resultado era que muy pocas aplicaciones paralelas utilizaban el modelo de hilos [WC00].

[1] `http://http://www.unix.org/version3/ieee_std.html`

2.2.1.3. Paralelismo basado en directivas del compilador

El uso de directivas del compilador para conseguir paralelismo tiene el fin de aliviar la complejidad y los problemas de la portabilidad. En el paralelismo orientado a directivas, la mayoría de los mecanismos paralelos se ponen en marcha a través del compilador (generación de hilos, generación de construcciones de sincronización, etc.). Es decir, el compilador traduce las directivas de compilador en las llamadas al sistema necesarias para la administración de los hilos, y realiza cualquier reestructuración del código que sea necesaria. Estas directivas proveen una manera simple de lograr paralelismo.

El estándar "OpenMP" [Ope13] para directivas de compilador paralelas ha promovido el uso de esta forma de programación paralela. Antes de la aparición de este estándar, las directivas de compilador eran específicas del fabricante del hardware, lo que dificultaba la portabilidad. Tanto la implementación explícita de hilos paralelos (como pthreads) como el paralelismo basado en directivas (como OpenMP) se benefician del uso de memoria compartida.

2.2.1.4. Paralelismo de memoria compartida

El paralelismo de hilos depende de la existencia de la memoria compartida para la comunicación entre hilos. Otro modelo paralelo anterior también utiliza memoria compartida, pero entre procesos. Este paralelismo de procesos típicamente se logra a través del uso de las llamadas al sistema "fork()" y "exec()" o sus análogas. Por ello se lo denomina generalmente como el modelo "fork/exec". La memoria es compartida entre los procesos en virtud de las llamadas al sistema "mmap()" (derivada de Berkeley UNIX) o "shmget()" (de System V UNIX) [WC00].

2.2.1.5. Pasaje de Mensajes

El modelo fork/exec no implica la existencia de memoria compartida. Los procesos pueden comunicarse a través de interfaces de E/S tales como las llamadas al sistema "read()" y "write()". Esto puede darse a través de archivos regulares o a través de alguna otra forma estándar de comunicación entre procesos como los "sockets". La comunicación a través de archivos resulta fácil entre procesos que comparten un sistema de archivos, pudiendo extenderse a varios sistemas al utilizar un sistema de archivos compartido como NFS. Los sockets son usualmente un medio de comunicación más eficiente entre procesos ya que eliminan gran parte del costo de realizar operaciones sobre un sistema de archivos.

Estas dos técnicas comunes dependen de que el proceso escriba, o envíe, el dato a ser comunicado hacia un archivo o socket. Este acto de comunicación se considera un mensaje, esto es, el proceso emisor está enviando un mensaje al proceso receptor. De ahí el nombre de pasaje de mensajes para este modelo.

Han existido diferentes implementaciones de bibliotecas de pasaje de mensajes, como PARMACS (para macros paralelas) y PVM (Parallel Virtual Machine) [WC00, GG+03]. Luego, en 1994, surge "MPI" [For12] en un intento de brindar una API estándar de pasaje de mensajes. MPI pronto desplazó a PVM y fue adoptando algunas de las ventajas de ésta. Aun cuando fue pensada principalmente para máquinas de memoria distribuida, tiene la ventaja de que puede ser aplicada también en máquinas de memoria compartida. MPI está destinado a paralelismo de procesos, no paralelismo de hilos.

2.2.2. Infraestructuras de hardware para paralelismo

Históricamente, las arquitecturas de computadoras paralelas han sido muy diversas. Existen aún diversas arquitecturas disponibles comercialmente hoy en día. En las siguientes secciones se dará un panorama general de las arquitecturas paralelas.

2.2.2.1. Clusters

Un cluster es una colección interconectada de equipos independientes que son utilizadas como un solo recurso de computación. Un ejemplo común de cluster es simplemente un conjunto de estaciones de trabajo conectadas por una red de área local [WC00].

Un aspecto positivo de un cluster es que en general cada nodo está de por sí bien balanceado en términos de procesador, sistema de memoria y capacidades de E/S (ya que cada nodo es una computadora). Otra ventaja es el costo: un cluster puede consistir de estaciones de trabajo estándar, de fácil adquisición. Del mismo modo, la interconexión de los nodos puede ser resuelta con tecnologías de red local estándar como Ethernet, FDDI, etc. Los clusters también resultan escalables.

La capacidad y el desempeño de las interconexiones son dos puntos críticos de los clusters. Las operaciones que acceden a datos residentes en el mismo nodo donde está ejecutándose la aplicación serán relativamente rápidas; acceder a datos residentes en otros nodos resulta algo muy diferente. Los datos remotos deberán ser transferidos vía llamadas al sistema para pasaje de mensajes. Esto implica los costos de la sobrecarga del mecanismo de llamadas al sistema y de la latencia de las comunicaciones, que dependen de la tecnología de la red subyacente.

La administración del sistema presenta otro problema. Sin software especial para esta tarea, es complejo administrar el sistema. El software debe instalarse en cada nodo individual, lo cual puede ser un proceso lento y costoso (por ejemplo, si se necesita una licencia por cada nodo).

2.2.2.2. Multiprocesadores

Las computadoras multiprocesador suponen un complemento para los clusters como también una arquitectura para programación paralela. En un multiprocesador, todos los procesadores acceden a todos los recursos de la máquina [WC00].

En los sistemas con modo dual de operación, las instrucciones privilegiadas se ejecutan en modo privilegiado o kernel. El proceso, que corre en modo de usuario, logra esto mediante llamadas al sistema, lo que provoca que el sistema operativo tome control sobre el hilo de ejecución del programa por un período de tiempo. En un sistema multiprocesador, si sólo una CPU a la vez puede ejecutar en modo kernel, aparece un cuello de botella que transformará la computadora multiprocesador en un sistema de un solo procesador. Un equipo multiprocesador debe ser capaz de que todos sus procesadores ejecuten en modo kernel.

Al proveer un solo espacio de direcciones para las aplicaciones, un sistema multiprocesador puede hacer el desarrollo de aplicaciones más fácil que en un sistema con múltiples e independientes espacios de direcciones como un cluster.

2.3. Cómputo de Altas Prestaciones

Luego de revisar el concepto de la computación paralela, las formas en que se categoriza y sus distintos modelos, definimos lo que podría llamarse una consecuencia de su evolución: el cómputo de altas prestaciones.

Un gran problema transversal a las Ciencias e Ingenierías Computacionales es la aplicación eficiente de modernas herramientas de cómputo paralelo y distribuido. La respuesta a este problema está condensada en el concepto de Computación de Altas Prestaciones (High Performance Computing, o HPC) que abarca todos aquellos principios, métodos y técnicas que permiten abordar problemas con estructuras de datos complejas y con gran utilización de recursos (tiempo de CPU, memoria, disco).

Tradicionalmente, el ámbito donde surgían los productos de la ciencia y de la ingeniería eran los laboratorios. Combinando la teoría y la experimentación, con cálculos hechos a mano o apoyándose en herramientas de cálculo rudimentarias, se aplicaba el conocimiento de la física, la matemática, la biología, para obtener y validar nuevos conocimientos. La aparición de las computadoras ofreció una nueva y potente forma de hacer ciencia e ingeniería: la ejecución de programas que utilizan modelos matemáticos y soluciones numéricas para resolver los problemas.

Así surgen herramientas como la simulación numérica, proceso de modelar matemáticamente un fenómeno de la realidad, y ejecutar experimentos virtuales a partir del modelo implementado en computador. En cada disciplina podemos encontrar experimentos que, por ser de alto costo, complejos, peligrosos o simplemente impracticables, hacen de la simulación numérica una herramienta de enorme valor. De la misma manera, las computadoras permiten la obtención de resultados concretos para problemas de cálculo imposibles de abordar en forma manual.

2.3.1. Ciencia e Ingeniería Computacional

La situación descripta en los párrafos anteriores ha dado auge a un nuevo campo interdisciplinario denominado Ciencia e Ingeniería Computacional (Computational Science & Engineering, CSE), que es la intersección de tres dominios: matemática, ciencias de la computación y las diferentes ramas de las ciencias o ingenierías. La Ciencia e Ingeniería Computacional usa herramientas de las ciencias de la computación y las matemáticas para estudiar problemas de las ciencias físicas, sociales, de la Tierra, de la vida, de las diferentes disciplinas ingenieriles, etc.

Durante la presente década, la Ciencia e Ingeniería Computacional ha visto un desarrollo espectacular. Puede decirse que las tecnologías de cómputo y de comunicaciones han modificado el campo científico de una manera que no admite retroceso, sino que, al contrario, la superación y extensión de esas tecnologías resulta vital para poder seguir haciendo ciencias como las conocemos hoy.

Gracias a estos avances tecnológicos los científicos pueden trascender sus anteriores alcances, extender sus resultados y abordar nuevos problemas, antes intratables. Entre los métodos de la Ciencia e Ingeniería Computacional se incluyen:

- Simulaciones numéricas, con diferentes objetivos:
  - Reconstrucción y comprensión de los eventos naturales conocidos: terremotos, incendios forestales, maremotos, etc.
  - Predicción del futuro o de situaciones inobservables: predicción del tiempo, comportamiento de partículas subatómicas.
- Ajustes de modelos y análisis de datos
  - Sintonización de modelos o resolución de ecuaciones para reflejar observaciones, sujetas a las limitaciones del modelo: prospección geofísica, lingüística computacional.
  - Modelado de redes, en particular aquellas que conectan individuos, organizaciones o sitios web.
  - Procesamiento de imágenes, inferencia de conceptos y discriminantes: detección de características del terreno, de procesos climatológicos, reconocimiento de patrones gráficos.
- Optimización
  - Análisis y mejoramiento de escenarios conocidos, como procesos técnicos y de manufactura.

A estos métodos, cuya aplicación hoy ya es corriente en las ciencias e ingenierías, se suman ciertos problemas, denominados "grandes desafíos", y cuya solución tiene amplio impacto sobre el desarrollo de esas disciplinas. Estos problemas pueden ser tratados por la aplicación de técnicas y recursos de Computación de Altas Prestaciones. Algunos de los campos donde aparecen estos problemas son:

- En la dinámica de fluidos computacional, en el diseño de aeronaves, la predicción del tiempo a corto y largo plazo, en la recuperación eficiente de minerales y muchas otras aplicaciones.
- Cálculos de estructuras electrónicas, para el diseño de nuevos materiales como catalíticos químicos, agentes inmunológicos o superconductores.
- Cómputos que permitan comprender la naturaleza fundamental de la materia y de los procesos de la vida.
- Procesamiento simbólico, incluyendo reconocimiento del habla, visión por computadora, comprensión del lenguaje natural, razonamiento automatizado y herramientas varias para diseño, manufactura y simulación de sistemas complejos.

La resolución de estos problemas involucra conjuntos masivos de datos, una gran cantidad de variables y complejos procesos de cálculo; por otro lado, es de carácter abierto, en el sentido de que siempre aparecerán escenarios de mayor porte o mayor complejidad para cada problema. Estos métodos y sus técnicas particulares exigen la utilización de recursos de computación hasta hoy excepcionales, como lo han sido las supercomputadoras, los multiprocesadores y la colaboración de una gran cantidad de computadoras a través de las redes, en diferentes niveles de agregación como clusters, multiclusters y grids.

2.4. Fortran

Muy popular en la programación científica y la computación de alto desempeño (HPC), el lenguaje Fortran surge a mediados de la década de 1950, siendo uno de los lenguajes de programación más antiguos utilizados aún hoy por científicos de todo el mundo. Se lo clasifica como un lenguaje de programación de alto nivel (considerado el primero de ellos en aparecer), de propósito general e imperativo. La programación imperativa describe un programa en términos del estado del programa y las sentencias que cambian dicho estado, como descripto por una máquina de Turing.

Fue desarrollado para aplicaciones científicas y de ingeniería, campos que dominó rápidamente, siendo durante todo este tiempo ampliamente utilizado en áreas de cómputo intensivo, tales como el análisis de elementos finitos, predicción numérica del clima, dinámica de fluidos computacional o física computacional. Ampliamente adoptado por científicos para escribir programas numéricamente intensivos, impulsó a los constructores de compiladores a generar código más rápido y eficiente. La inclusión de un tipo de dato complejo (COMPLEX) lo hizo especialmente apto para aplicaciones técnicas como la Ingeniería Eléctrica.

En las áreas científicas, con típicos problemas de cálculo intensivo, una vez que un programa alcanza un estado de computación correcta (i.e. arroja los resultados deseados), no suelen ocurrir modificaciones del código. En el caso de Fortran, su adopción por la comunidad científica derivó en la construcción de programas que permanecen vigentes tras 20, 30 y hasta 40 años. Estos constituyen lo que denominamos "Legacy Software". Se ha definido al Legacy Software como:

- Software crítico que no puede ser modificado eficientemente [Gol98].
- Cualquier sistema de información que significativamente resiste las modificaciones y la evolución para alcanzar requerimientos de negocio nuevos y constantemente cambiantes [BS95].

Algunas características de los sistemas legacy son:

- La resistencia al cambio del software.
- La complejidad inherente.
- La tarea crucial desempeñada por el software en la organización.
- El tamaño del sistema, generalmente mediano o grande.

Es común hallar software hecho en Fortran que ha estado ejecutándose en ambientes de producción por décadas. Durante ese período, el software puede necesitar cambios de diferente tipo: mejoras, correcciones, adaptaciones y prevenciones. Para todas estas tareas se necesita conocimiento y comprensión del sistema. En la era multinucleo y muchos núcleos (decenas a cientos de núcleos), los cambios de software se hacen más y más complejos [MT12]. Debido a que Fortran ha estado tantos años vigente, ha pasado por un proceso particular de estandarización en el cual cada versión previa del estándar cumple con el vigente. Este proceso de estandarización permite que un programa en Fortran 77 compile en los compiladores modernos de Fortran 2008 [MT12]. Gracias a estas características es que el lenguaje está, aún hoy y a pesar de ser relativamente poco visible, en una posición sólida y bien definida. Actualmente hay un gran conjunto de programas Fortran ejecutándose en ambientes productivos de universidades, empresas e instituciones de gobierno. Algunos buenos ejemplos son programas de modelo climático, simulaciones de terremotos, simulaciones magnetohidrodinámicas, etc. La mayoría de estos programas han sido construidos años o décadas atrás y sus usuarios necesitan que sean modernizados, mejorados y/o actualizados. Esto también implica que estos programas sean capaces de aprovechar las arquitecturas de procesadores modernas y específicamente, el equipamiento para procesamiento numérico.

2.4.1. Evolución del lenguaje

Fortran ha evolucionado de una versión inicial con 32 sentencias para la IBM 704, entre los que estaban el condicional IF y el IF aritmético de 3 vías, el salto GO TO, el bucle DO, comandos para E/S tanto formateada como sin formato (FORMAT, READ, WRITE, PRINT, READ TAPE, READ DRUM, etc), y de control del programa (PAUSE, STOP, CONTINUE), y tipos de datos todos numéricos, hasta llegar al último estandar Fortran, ISO/IEC 1539-1:2010, conocido informalmente como Fortran 2008, donde fueron incorporándose características como tipos de datos CHARACTER, definición de arrays, subrutinas, funciones, recursividad, modularidad, hasta nuevas sentencias que soportan la ejecución de alta performance como DO CONCURRENT, coarrays (un modelo de ejecución paralela). La definición del estándar Fortran se encuentra en el sitio de NAG (The Numerical Algorithms Group)[2].

El lenguaje utilizado en el programa de estudio de este trabajo de tesis está basado en el estándar Fortran 77, aunque presenta libertades presentes en Fortran 90, como la utilización de minúsculas indiferentemente y el formato libre de escritura. Esto es posible, como explicamos en la sección previa, gracias a que cada versión nueva del compilador soporta los estándares previos. Se presentan en el anexo A las sentencias de Fortran más utilizadas en la aplicación objeto de estudio.

2.5. OpenMP

OpenMP es una Interfaz de Programación de Aplicaciones, o API por sus siglas en inglés, la cual provee un modelo portable y escalable para el desarrollo de aplicaciones paralelas de

[2] `http://www.nag.co.uk/sc22wg5/`

memoria compartida. La API soporta C/C++ y Fortran en una gran variedad de arquitecturas. Es utilizada para aplicar de manera directa multithreads en memoria compartida[3].

La especificación de OpenMP [Ope13] pertenece, es escrita y mantenida por la OpenMP Architecture Review Board, que es la unión de las compañías que tienen participación activa en el desarrollo del estándar para la interfaz de programación en memoria compartida [Her02].

No es un nuevo lenguaje de programación, sino que es una notación que puede ser agregada a un programa secuencial en Fortran, C o C++ para describir cómo el trabajo debe ser compartido entre los hilos que se ejecutarán en diferentes procesadores o núcleos y para organizar el acceso a los datos compartidos cuando sea necesario. La inserción apropiada de las características de OpenMP en un programa secuencial permitirá a muchas, si no a la mayoría de las aplicaciones, beneficiarse de una arquitectura de memoria compartida, a menudo con mínimas modificaciones al código. Uno de los factores del éxito de OpenMP es que es comparativamente sencillo de usar, ya que el trabajo más complicado de armar los detalles del programa paralelo son dejados para el compilador. Tiene además la gran ventaja de ser ampliamente adoptado, de manera que una aplicación OpenMP va a poder ejecutarse en muchas plataformas diferentes [CJvdP08].

Las directivas de OpenMP permiten al usuario indicarle al compilador cuales instrucciones ejecutar en paralelo y como distribuir las mismas entre los hilos que van a ejecutar el código. Una de estas directivas es una instrucción en un formato especial que es entendido por compiladores OpenMP solamente. De hecho luce como un comentario para un compilador Fortran regular, o una directiva pragma para un compilador C/C++, de manera que el programa puede ejecutarse como lo hacía previamente si el compilador no conoce OpenMP. Generalmente se puede rápida y fácilmente crear programas paralelos confiando en la implementación para que trabaje los detalles de la ejecución paralela. Pero no siempre es posible obtener alta performance con una inserción sencilla, incremental de directivas OpenMP en un código secuencial. Por esta razón OpenMP incluye varias características que habilitan al programador a especificar más detalle en el código paralelo.

2.5.1. La idea de OpenMP

Un hilo es una entidad en tiempo de ejecución que es capaz de ejecutar independientemente un flujo de instrucciones. OpenMP trabaja en un cuerpo más grande de trabajo que soporta la especificación de programas para ser ejecutados por una colección de hilos cooperativos. El sistema operativo crea un proceso para ejecutar un programa: reservará algunos recursos para este proceso, incluyendo páginas de memoria y registros para almacenar valores de objetos. Si múltiples hilos colaboran para ejecutar un programa, compartirán los recursos, incluyendo el espacio de direcciones, del correspondiente proceso. Los hilos individuales necesitan muy pocos recursos por si mismos: un contador de programa y un área de memoria para guardar variables especificas del hilo (incluyendo registros y una pila) [CJvdP08]. OpenMP intenta proveer facilidad de programación y ayudar al usuario a evitar un número de potenciales errores de programación, ofreciendo un enfoque estructurado para la programación multithread. Soporta el modelo de programación llamado fork-join, el cual podemos ver en la Fig. 2.1.

Bajo este enfoque, el programa inicia como un solo hilo de ejecución (denominado hilo inicial), igual que un programa secuencial. Siempre que se encuentre una construcción paralela de OpenMP por el hilo mientras ejecuta su programa, se crea un equipo de hilos (ésta es la parte fork), se convierte en el maestro del equipo, y colabora con los otros miembros del mismo para ejecutar el código dinámicamente encerrado por la construcción. Al final de ésta, sólo el hilo original, o maestro del equipo, continúa; todos los demás terminan (ésta es la parte join). Cada porción del código encerrada por una construcción paralela es llamada una región paralela.

[3] `https://computing.llnl.gov/tutorials/openMP/`

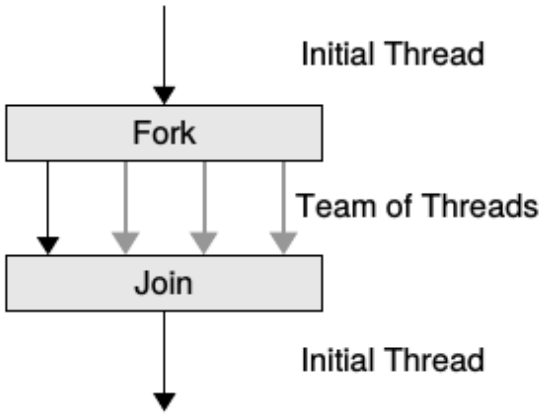


Figura 2.1: Modelo fork-join

2.5.2. Conjunto de construcciones paralelas

La API OpenMP comprende un conjunto de directivas del compilador, rutinas de bibliotecas de tiempo de ejecución, y variables de ambiente para especificar paralelismo de memoria compartida. Muchas de las directivas son aplicadas a un bloque estructurado de código, una secuencia de sentencias ejecutables con una sola entrada en la parte superior y una sola salida en la parte inferior en los programas Fortran, y una sentencia ejecutable en C/C++ (que puede ser una composición de sentencias con una sola entrada y una sola salida). En otras palabras, el programa no puede ramificarse dentro o fuera de los bloques de código asociados con directivas. En Fortran el inicio y el final del bloque aplicable de código son marcados explícitamente por una directiva OpenMP [CJvdP08].

2.5.2.1. Crear equipos de Hilos

Un equipo de hilos es creado para ejecutar el código en una región paralela de un programa OpenMP. El programador simplemente especifica la región paralela insertando una directiva *parallel* inmediatamente antes del código que debe ser ejecutado en paralelo para marcar su inicio; en los programas Fortran el final también es indicado por una directiva end parallel. Información adicional puede ser provista junto con la directiva *parallel*, como habilitar a los hilos a tener copias privadas de algún dato por la duración de la región paralela. El final de una región paralela es una barrera de sincronización implícita: esto significa que ningún hilo puede progresar hasta que todos los demás hilos del equipo hayan alcanzado este punto del programa [CJvdP08]. Es posible realizar anidado de regiones paralelas.

2.5.2.2. Compartir trabajo entre Hilos

Si el programador no especifica como se compartirá el trabajo en una región paralela, todos los hilos ejecutarán el código completo redundantemente, sin mejorar los tiempos del programa. Para ello OpenMP cuenta con directivas para compartir trabajo que permiten indicar como se distribuirá el cómputo en un bloque de código estructurado entre los hilos. A menos que el programador lo indique explícitamente, una barrera de sincronización existe implícitamente al final de las construcciones de trabajo compartido [CJvdP08].

Probablemente el método más común de trabajo compartido sea distribuir el trabajo en un bucle DO (Fortran) o for (C/C++) entre los distintos hilos de un equipo. El programador inserta la directiva apropiada inmediatamente antes de los bucles que vayan a ser compartidos entre los hilos dentro de una región paralela. Todas las estrategias de OpenMP para compartir el trabajo

en un bucle asignan uno o más conjuntos disjuntos de iteraciones a cada hilo. El programador puede, si así lo desea, especificar el método para particionar el conjunto de iteración.

2.5.2.3. Modelo de memoria de OpenMP

OpenMP se basa en el modelo de memoria compartida, por ello, por defecto los datos son compartidos por todos los hilos y son visibles para todos. A veces es necesario tener variables que poseen valores específicos por hilo. Cuando cada hilo tiene una copia propia de una variable, donde potencialmente tenga valores distintos en cada uno de ellos, decimos que la variable es privada; por ejemplo, cuando un equipo de hilos ejecuta un bucle paralelo cada hilo necesita su propio valor de la variable de control de iteración. Este caso es tan importante que el propio compilador fuerza que así sea; en otros casos el programador es quien debe determinar cuales variables son compartidas y cuales privadas [CJvdP08].

2.5.2.4. Sincronización de Hilos

En ocasiones es necesario sincronizar los hilos para asegurar el acceso en orden a los datos compartidos y prevenir corrupción de éstos. Asegurar la coordinación de hilos necesaria, es uno de los desafíos más fuertes de la programación de memoria compartida [CJvdP08]. OpenMP provee, por defecto, sincronización implícita haciendo que los mismos esperen al final de una construcción de trabajo compartido o región paralela hasta que todos los hilos en el equipo terminan su porción de trabajo. Más difícil de conseguir en OpenMP es coordinar las acciones de un subconjunto de los hilos ya que no hay soporte explícito para esto. Otras veces es necesario asegurar que solo un hilo a la vez trabaja en un bloque de código. OpenMP tiene varios mecanismos que soportan este tipo de sincronización.

2.5.2.5. Otras características

Subrutinas y funciones pueden complicar la utilización de APIs de Programación Paralela. Una de las características innovadoras de OpenMP, es el hecho de que las directivas pueden ser insertadas dentro de procedimientos que son invocados desde dentro de una región paralela.
Para algunas aplicaciones puede ser necesario controlar el número de hilos que ejecutan la región paralela. OpenMP permite al programador especificar este número previo a la ejecución del programa a través de una variable de ambiente, luego de que el cómputo ha iniciado a través de una librería de rutinas, o al comienzo de regiones paralelas. Si no se hace esto, la implementación de OpenMP utilizada elegirá el número de hilos a utilizar [CJvdP08].

2.6. OpenMP en Fortran

Fueron presentados Fortran y OpenMP, en esta sección se ve como se complementan, mostrando cual es el formato de las directivas de OpenMP en Fortran.

2.6.1. Centinelas para directivas de OpenMP y compilación condicional

El estándar OpenMP ofrece la posibilidad de usar el mismo código fuente con un compilador que implementa OpenMP como con uno normal. Para ello debe ocultar las directivas y comandos de una manera que un compilador normal no pueda verlas. Para ello existen las siguientes directivas centinelas [Her02]

```
!$OMP

!$
```

Como el primer caracter es un signo de exclamación "!", un compilador normal va a interpretar las líneas como comentarios y va a ignorar su contenido. Pero un compilador compatible con OpenMP identificará la sentencia y procederá como sigue [Her02]

- !$OMP : el compilador compatible con OpenMP sabe que la información que sigue en la línea es una directiva OpenMP. Se puede extender una directiva en varias líneas utilizando el mismo centinela frente a las siguientes líneas y usando el método estándar de Fortran para partir líneas de código:

```
!$OMP PARALLEL DEFAULT(NONE) SHARED(A, B) &
!$OMP REDUCTION(+:A)
```

Es obligatorio incluir un espacio en blanco entre la directiva centinela y la directiva OpenMP que le sigue, sino la línea será interpretada como un comentario.

- !$: la línea correspondiente se dice que está afectada por una compilación condicional. Quiere decir que su contenido estará disponible para el compilador en caso de que sea compatible con OpenMP. Si esto ocurre, los dos caracteres del centinela son reemplazados por dos espacios en blanco para que el compilador tenga en cuenta la línea. Como en el caso anterior, podemos extender la línea en varias líneas como sigue:

```
!$ interval = L * OMP_get_thread_num() / &
!$                        (OMP_get_num_threads() - 1)
```

Nuevamente el espacio en blanco es obligatorio entre la directiva de compilación condicional y el código fuente que le sigue.

2.6.2. El constructor de región paralela

La directiva más importante en OpenMP es la que define las llamadas regiones paralelas. Para entender mejor qué es la región paralela se puede observar una representación en la Fig. 2.2.
Podemos definir que cuando se crea la región paralela, se crean los hilos que ejecutarán paralelamente el código que esté definido dentro de la misma.

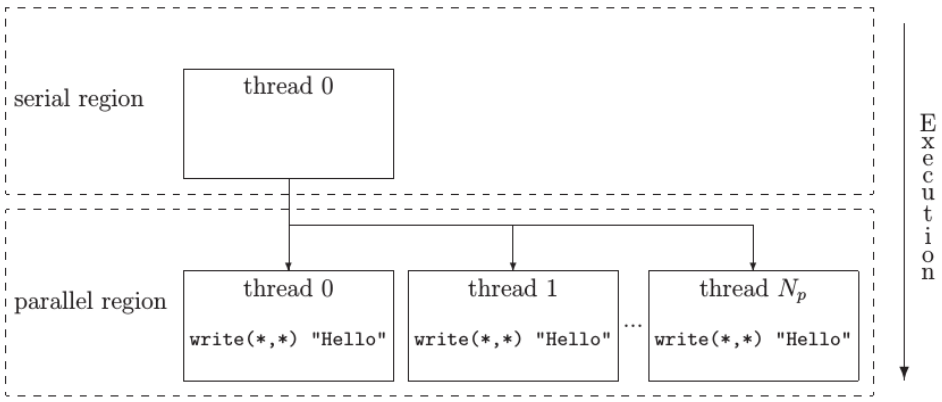


Figura 2.2: Representación de la Región Paralela.
Fuente: [Her02].

Ya que la región paralela necesita ser creada/abierta y destruida/cerrada, dos directivas son necesarias en Fotran: !$OMP PARALLEL / !$OMP END PARALLEL. El código de la Fig. 2.2 se puede ver a continuación:

```
!$OMP PARALLEL
          write(*,*) ``Hola''
!$OMP END PARALLEL
```

Como el código entre las dos directivas es ejecutado por cada hilo, el mensaje *Hola* aparece en la pantalla tantas veces como hilos estén siendo usados en la región paralela. Al comienzo de la región paralela es posible imponer cláusulas que fijan ciertos aspectos de la manera en que ésta va a trabajar, por ejemplo el alcance de las variables, el número de hilos, etc. La sintaxis a usar es la siguiente:

```
          !$OMP PARALLEL  clause1 clause2 ...
          ...
          !$OMP END PARALLEL
```

Las cláusulas permitidas en la directiva de apertura !$OMP PARALLEL son las siguientes:

- PRIVATE(lista)
- SHARED(lista)
- DEFAULT(PRIVATE | SHARED | NONE)
- FIRSTPRIVATE(lista)
- COPYIN(lista)
- REDUCTION(operador:lista)
- IF(expresión_escalar_lógica)
- NUM_THREADS(expresión_escalar_entera)

La directiva !$OMP END PARALLEL indica el final de la región paralela, la barrera implícita mencionada antes en el capitulo. En este punto es donde ocurre la sincronización entre el equipo de hilos y son terminados todos excepto el hilo maestro que continua con la ejecución del programa.

2.6.3. Directiva !$OMP DO

Es una directiva de trabajo compartido, por lo cual al encontrarla en el código el trabajo es distribuido en un equipo de hilos. Debe ser ubicada dentro del alcance de una región paralela para ser efectiva, si no, la directiva aún funcionará pero el equipo será de un solo hilo. Esto se debe a que la creación de nuevos hilos es una tarea reservada a la directiva de creación de la región paralela. Esta directiva hace que el bucle *Do* inmediato sea ejecutado en paralelo. Por ejemplo:

```
          !$OMP DO
                  do 1   i = 1, 1000
                  ...
            1 continue
          !$OMP END DO
```

distribuye el bucle *Do* entre los diferentes hilos, cada hilo computa una parte de las iteraciones. Por ejemplo si usamos 10 hilos, entonces generalmente cada hilo computa 100 iteraciones del bucle do. El hilo 0 desde 1 a 100, el hilo 1 desde 101 a 200 y así sucesivamente. Podemos ver esto en la Fig. 2.3.

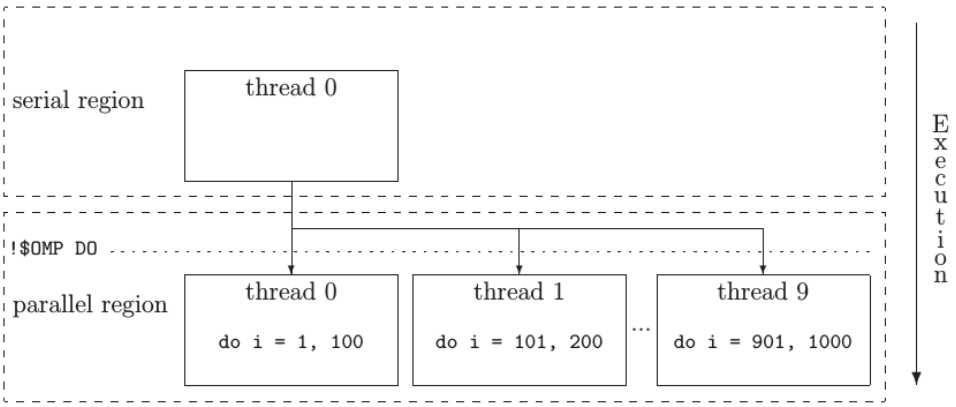


Figura 2.3: Representación gráfica de la directiva OMP DO.
Fuente: [Her02].

Dentro del trabajo de esta tesis es la directiva principal, al tratarse los bucles *Do* de una de las principales construcciones utilizadas en el programa estudiado.

La directiva !$OMP DO tiene asociadas cláusulas al igual que la directiva parallel, que permiten indicar el comportamiento de la construcción de trabajo compartido. La sintaxis es similar:

```
!$OMP DO clause1 clause2 ...
...
!$OMP END DO end_clause
```

Las cláusulas de inicio pueden ser cualquiera de las siguientes:

- PRIVATE(lista)
- FIRSTPRIVATE(lista)
- LASTPRIVATE(lista)
- REDUCTION(operador:lista)
- SCHEDULE(tipo, pedazo)
- ORDERED

Adicionalmente a estas cláusulas de inicio, se puede agregar a la directiva de cierre la cláusula NOWAIT para evitar la sincronización implícita. También se evita el refresco de las variables compartidas, implícito en la directiva de cierre, por lo que se debe tener cuidado de cuando utilizar la cláusula NOWAIT. Se pueden evitar problemas con la directiva de OpenMP !$OMP FLUSH que fuerza el refresco de las variables compartidas en memoria por los hilos.

2.6.4. Clausulas Atributo de Alcance de Datos

2.6.4.1. PRIVATE(lista)

A veces, ciertas variables van a tener valores diferentes en cada hilo. Esto sólo es posible si cada hilo tiene su propia copia de la variable. Esta cláusula fija qué variables van a ser consideradas variables locales de cada hilo. Por ejemplo, para indicar que las variables a y b tendrán diferentes valores en cada hilo, i.e., serán locales/privadas a cada hilo, utilizamos el siguiente código:

```
!$OMP PARALLEL PRIVATE(a, b)
```

Cuando una variable se declara como privada, un nuevo objeto del mismo tipo es declarado por cada hilo del equipo y usado por cada hilo dentro del alcance de la directiva que lo declare (la región paralela en el ejemplo anterior) en lugar de la variable original. El código anterior se representa en la Fig. 2.4.

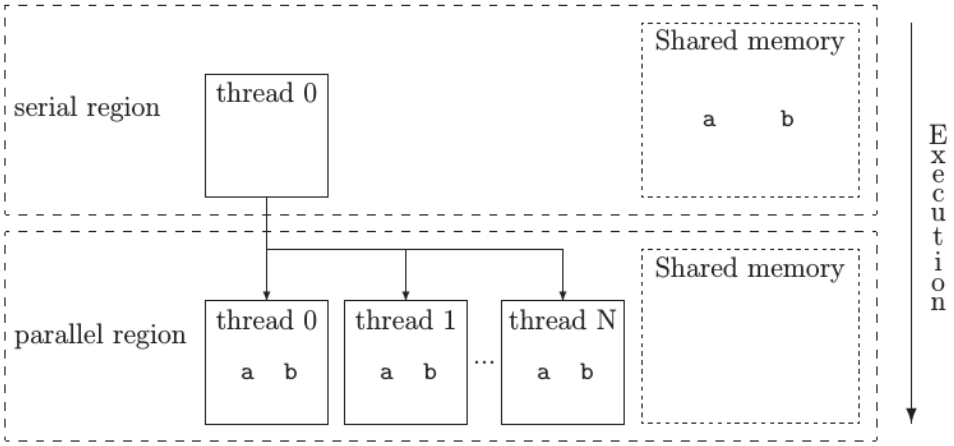


Figura 2.4: Representación gráfica de la cláusula PRIVATE.
Fuente: [Her02].

El hecho de que un nuevo objeto es creado por cada hilo puede ser algo que genere mucho consumo de recursos. Por ejemplo, si se utiliza un array de 5Gb (algo común en simulaciones numéricas directas y otras) y es declarado como privado en una región paralela con un equipo de 10 hilos, entonces el requerimiento de memoria será de 55Gb, algo no disponible en todas las maquinas SMP.

Las variables utilizadas como contadores en los bucles *Do* o comandos *forall*, o son declaradas THREADPRIVATE, se convierten automáticamente en privadas para cada hilo, aun cuando no hayan sido declaradas en un cláusula PRIVATE.

2.6.4.2. SHARED(lista)

Contrario a lo visto en la situación previa, a veces hay variables que deben estar disponibles para todos los hilos dentro del alcance de una directiva, debido a que su valor es necesario para todos los hilos o porque todos los hilos deben actualizar su valor. Por ejemplo:

```
!$OMP PARALLEL SHARED(c, d)
```

indica que las variables c y d son vistas por todos los hilos en el alcance de las directivas !$OMP PARALLEL / !$OMP END PARALLEL. Podemos observar en la Fig. 2.5, la representación del ejemplo.

Una variable declarada como compartida (shared) no consume recursos extras, ya que no se reserva nueva memoria y su valor antes de la directiva inicial es conservado. Es decir que todos los hilos acceden a la misma ubicación de memoria para leer y escribir la variable. Debido a que más de un hilo puede escribir en la misma ubicación de memoria al mismo tiempo, resulta en un valor indefinido de la variable. A esto se lo llama una condición de carrera, y debe ser siempre evitado por el programador.

2.6.4.3. DEFAULT (PRIVATE |SHARED |NONE)

Cuando la mayoría de las variables dentro del alcance de una directiva va a ser privada o compartida, entonces sería engorroso incluir todas ellas en una de las cláusulas previas. Para

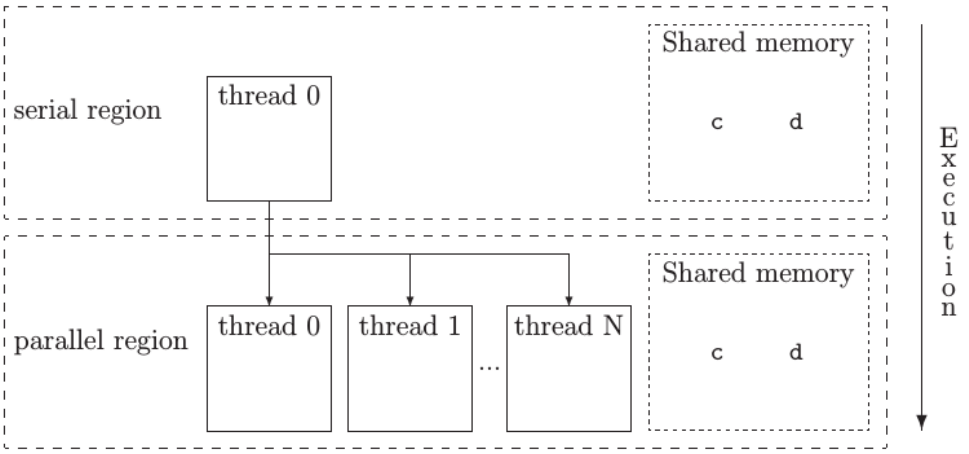


Figura 2.5: Representación gráfica de la cláusula SHARED.
Fuente: [Her02].

evitar esto, es posible especificar que hará OpenMP cuando no se especifica nada sobre una variable, i.e. un comportamiento por defecto. Por ejemplo:

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(a)
```

indica que todas las variables excepto "a" van a ser privadas, mientras que "a" será compartida por todos los hilos dentro del alcance de la región paralela. Si no se especifica ninguna cláusula DEFAULT, el comportamiento por defecto es como si DEFAULT(SHARED) fuera especificado. Como veremos en el capitulo 3 de este trabajo de tesis, esto puede variar en implementaciones y debe ser investigado más a fondo. A las opciones PRIVATE y SHARED se le agrega una tercera: NONE. Especificando DEFAULT(NONE) requiere que cada variable en el alcance de la directiva debe ser explícitamente listada en una de las cláusulas PRIVATE o SHARED al principio del alcance de la directiva (exceptuando variables declaradas THREADPRIVATE o los contadores de los bucles).

2.6.4.4. FIRSTPRIVATE(lista)

Como mencionamos previamente, las variables privadas tienen un valor indefinido al comienzo del alcance de un par de directivas de inicio y cierre. Pero a veces es de interés que esas variables locales tengan el valor de la variable original antes de la directiva de inicio. Esto se consigue incluyendo la variable en una cláusula FIRSTPRIVATE como:

```
a = 2
b = 1
!$OMP PARALLEL PRIVATE(a) FIRSTPRIVATE(b)
```

En este ejemplo, la variable "a" tiene un valor indefinido al inicio de la región paralela, mientras que "b" tiene el valor especificado en la región serial precedente, es decir "b = 1". Podemos ver este ejemplo en la Fig. 2.6.

Al incluir la variable en una cláusula FIRSTPRIVATE al inicio del alcance de una directiva toma automáticamente el estatus de PRIVATE en dicho alcance y no es necesario incluirla en una cláusula PRIVATE explícitamente. Al igual que con las variables PRIVATE debe tenerse en cuenta el costo de la operación desde el punto de vista computacional, al realizarse una copia de la variable y transferir la información almacenada a la nueva variable.

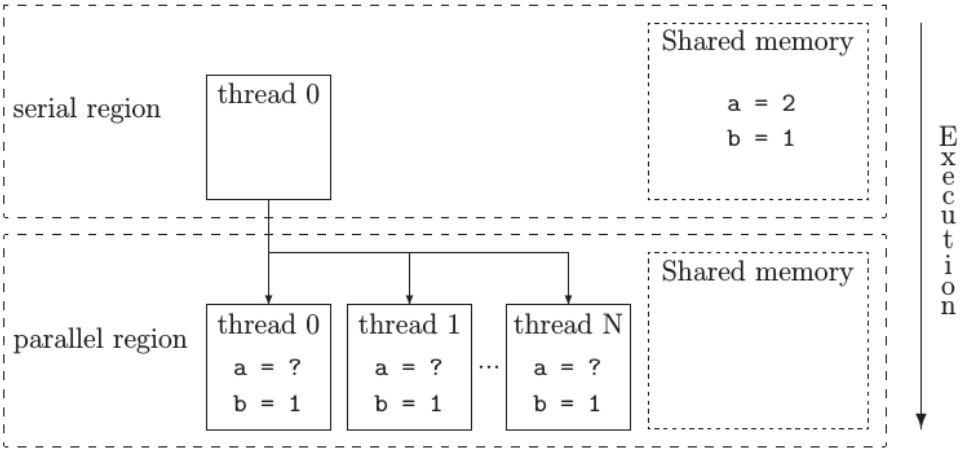


Figura 2.6: Representación gráfica de las cláusulas PRIVATE y FIRSTPRIVATE. Fuente: [Her02]

2.6.5. Otras Construcciones y Cláusulas

Existen más construcciones de trabajo compartido, de sincronización y de ambiente de datos, y más cláusulas en OpenMP, las cuales exceden el alcance de este trabajo de tesis y que pueden ser consultadas en el estándar OpenMP [Ope13] o en [Her02].

2.7. Proceso de optimización

Dependiendo del propósito de una aplicación y de la forma como será utilizada, suelen considerarse tres principios de optimización del desempeño [GS01]

- Resolver el problema más rápidamente
- Resolver un problema más grande en el mismo tiempo
- Resolver el mismo problema en el mismo tiempo, pero utilizando una cantidad menor de recursos del sistema

En aplicaciones de HPC, obtener resultados más rápidamente es crucial para los usuarios. Por ejemplo, para un ingeniero, representa una diferencia considerable poder repetir una simulación en el transcurso de una noche en lugar de esperar varios días para que la simulación termine. El tiempo ganado puede ser aprovechado para modificar el diseño, correr experimentos de mayor tamaño, resolver problemas con conjuntos de datos más grandes, u obtener resultados más precisos. Por otro lado, cuando el tamaño del problema y el tiempo de ejecución se mantengan constantes, una aplicación optimizada consumirá menos recursos para completar su ejecución [GS01].

El proceso de optimización tiene algunas etapas fundamentales: "desarrollo de la aplicación, optimización serial, y optimización paralela". La primera etapa abarca el diseño, programación y consideraciones de portabilidad de la aplicación, es decir, elección de algoritmos y estructuras de datos para resolver el problema. En el caso de este trabajo de tesis, esa etapa fue llevada a cabo por el autor de la aplicación en que basamos nuestro estudio. Las etapas de optimización serial y optimización paralela son parte de este trabajo de tesis. En la Fig. 2.7, podemos observar las etapas del proceso de optimización.

Una decisión importante para el proceso de optimización es contemplar en qué plataforma o conjunto de ellas se implementará la aplicación. Esta decisión incluye seleccionar sistema operativo y arquitectura de ejecución. La optimización será más focalizada mientras más puntuales

sean las decisiones tomadas, limitando el rango de plataformas en las cuales el programa puede ejecutarse [GS01]. Una vez que el programa produzca resultados correctos, estará listo para ser optimizado. Se seleccionarán un conjunto de casos de test para validar que el programa continúe arrojando resultados correctos y se los utilizará repetidamente en el transcurso de la optimización.

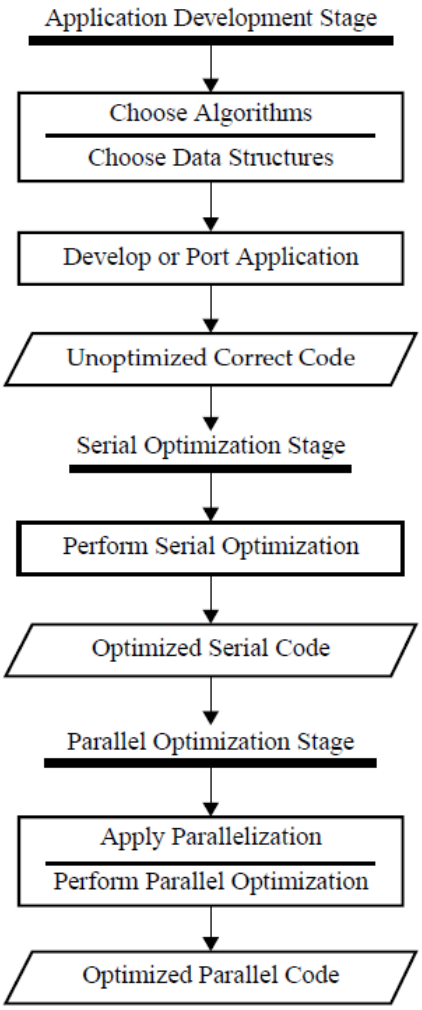


Figura 2.7: Etapas de Optimización y Desarrollo de una Aplicación.
Fuente: [GS01].

También se debe seleccionar un conjunto de casos para llevar a cabo pruebas de tiempo. Puede ser necesario que este conjunto sea diferente a los utilizados para validar el programa. Los casos de test para el cronometraje de tiempo podrían ser varios "benchmarks" que representen adecuadamente el uso del programa. Se utilizarán estos benchmarks para medir el nivel de desempeño básico o "línea de base", de manera de disponer de datos fiables para utilizar más tarde en las comparaciones de código optimizado y código original. De esta manera, se puede medir el efecto de la optimización.

2.7.1. Optimización Serial

La Optimización Serial es un proceso iterativo que involucra medir repetidamente un programa seguido por la optimización de sus partes críticas de rendimiento. La Fig. 2.8 resume las tareas de optimización y da un diagrama de flujo simplificado para el proceso de optimización serial.

Una vez que las mediciones de rendimiento de la línea de base se han obtenido, el esfuerzo de optimización debe iniciarse mediante la compilación de todo el programa con opciones seguras.

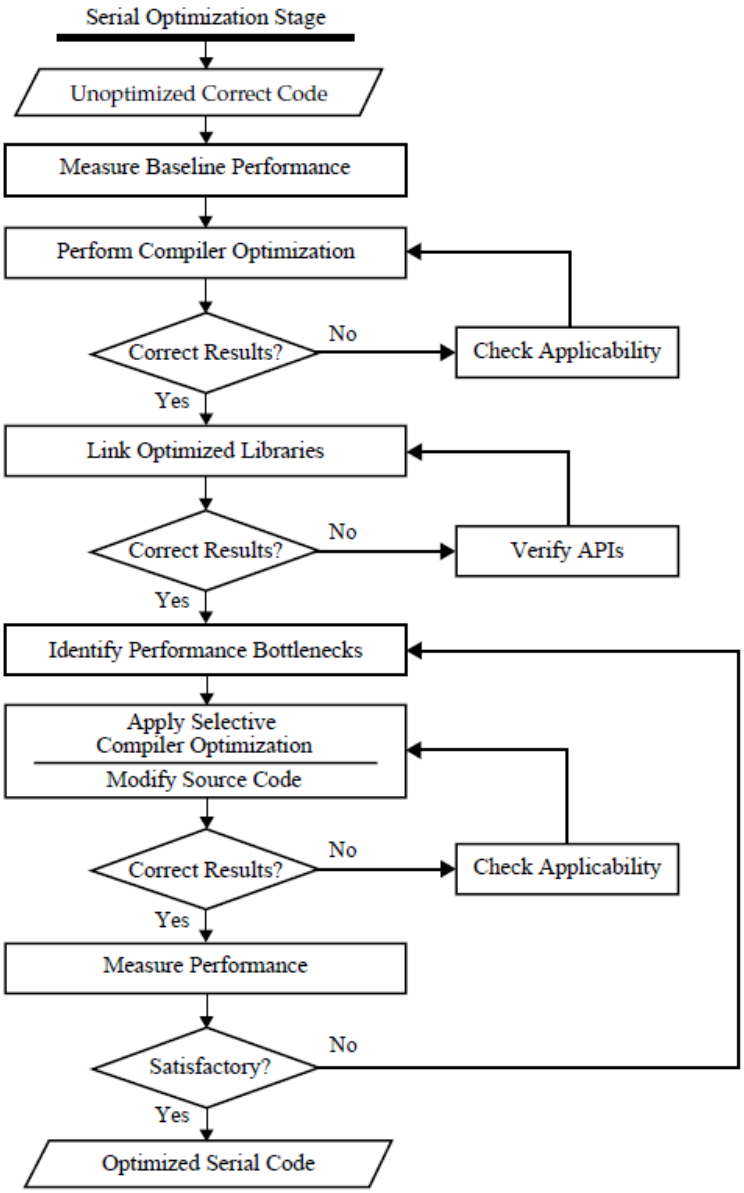


Figura 2.8: Proceso de Optimización Serial.
Fuente: [GS01].

Lo siguiente sería enlazar librerías optimizadas. (Este enlace es una manera sencilla de llevar implementaciones altamente optimizadas de operaciones estándar en un programa) Luego de esto se debe verificar que los resultados preservan la correctitud del programa. Este paso incluye verificar que el programa realiza llamadas a las Interfaces de Programación de Aplicación (APIs sus siglas en inglés) adecuadas en las librerías optimizadas. Además, es recomendable que sea medido el desempeño del programa para verificar que es lo que ha mejorado.

El siguiente paso es identificar partes de desempeño críticas en el código. El perfilado (profiling en inglés) del código fuente puede ser usado para determinar cuales partes del código son las que toman más tiempo para ejecutarse. Las partes identificadas son excelentes objetivos para enfocar el esfuerzo de optimización, y las mejoras resultantes de desempeño pueden ser significativas. Otra técnica muy útil para identificar estas partes de código críticas es el monitoreo de la actividad del sistema y el uso de los recursos del sistema [GS01].

2.7.1.1. Metodología de medición

Al trabajar en optimizar la performance de una aplicación, es esencial usar varias herramientas y técnicas que sugieran qué partes del programa necesitan ser optimizadas, comparar el desempeño antes y luego de la optimización y mostrar que tan eficientes han sido los recursos del sistema utilizados por el código optimizado [GS01].

El primer paso en el proceso de afinación de la aplicación es cuantificar su desempeño. Este paso es alcanzado usualmente estableciendo un desempeño base y fijando expectativas apropiadas de cuanta mejora en el desempeño es razonable alcanzar. Para programas científicos, las métricas de mayor interés son usualmente el tiempo reloj (tiempo de respuesta) de un solo trabajo y aquellos que relacionan el desempeño de la aplicación a picos teóricos de desempeño de la CPU [GS01]. A través de benchmarks es que podemos realizar análisis de la performance de la aplicación. Una guía importante a seguir es que las mediciones deben ser reproducibles dentro de un rango de tolerancia esperado. Establecido esto se definen las siguientes reglas generales:

- Seleccionar cuidadosamente los conjuntos de datos a utilizar. Deben representar adecuadamente el uso de la aplicación.
- Al igual que en las mediciones en otros campos de la ingeniería, la incertidumbre también se aplica a las mediciones de desempeño de programas de computadora. El simple hecho de tratar de medir un programa se entromete en su ejecución y posiblemente lo afecta de manera incierta.
- Siempre que sea posible, ejecutar los benchmarks desde un sistema de archivos tipo tmpfs (/tmp) o algún sistema de archivos montado localmente. Ejecutar una aplicación desde un sistema de archivos montado por red introduce efectos de red irreproducibles en el tiempo de ejecución.
- Actividades de paginado e intercambio a disco deben ser monitoreadas mientras se ejecuta el benchmark, ya que éstas pueden desvirtuar completamente la medición.
- Las mediciones de "respuesta del programa" deben ser desempeñadas en una manera dedicada, sin otros programas o aplicaciones ejecutándose.
- Las características del sistema deben ser registradas y guardadas.

2.7.1.2. Herramientas de medición

Antes de analizar el desempeño de la aplicación, se debe identificar los parámetros que deben ser medidos y elegir herramientas acordes a las mediciones. Las herramientas de medición de desempeño pueden ser divididas en tres grupos [GS01] basados en su función:

- Herramientas de temporizador, que miden el tiempo utilizado por un programa de usuario o sus partes. Pueden ser herramientas de línea de comando o funciones dentro del programa.
- Herramientas de perfilado, que utilizan resultados de tiempo para identificar las partes de mayor utilización de una aplicación.
- Herramientas de monitoreo, que miden la utilización de varios recursos del sistema para identificar "cuellos de botella" que ocurren durante la ejecución.

Existen otras formas de categorizar estas herramientas, como puede ser basados en los requerimientos para su uso (herramientas que operan con binarios optimizados, o que requieren insertarse en el código fuente, etc), o incluso dividirlas en dos grupos:

- Herramientas de medición de desempeño serial.
- Herramientas de medición de desempeño paralelo.

2.7.1.3. Herramientas de medición de tiempo

El paso fundamental para evaluar comparativamente y poner a punto el desempeño de un programa, es medir con precisión la cantidad de tiempo utilizado ejecutando el código. Generalmente uno está interesado en el tiempo total utilizado para correr un programa, así como en el tiempo utilizado en porciones del programa. Para medir el programa completo es necesario usar herramientas que midan con precisión el tiempo transcurrido desde el comienzo de la ejecución del mismo. En GNU/Linux utilizamos la herramienta "time" para dicho propósito. La forma de utilizar time es ejecutarlo desde una terminal de GNU/Linux pasando como parámetro el comando que debe medir tal cual como el comando es ejecutado normalmente. Por ejemplo:

```
$ time  find / -name ‘‘syslog’’
```

El comando siendo medido realiza su ejecución normalmente. Al finalizar su ejecución, el comando time muestra por salida estándar tres valores (Fig. 2.9):

- "real": el tiempo real transcurrido entre el inicio y la finalización de la ejecución.
- "user": el tiempo de usuario del procesador.
- "sys": el tiempo de sistema del procesador.

```
h4ndr3s@gondolin:~$ time find . -name "invisidos*"
./t3sis/tesis/source/invisidos2fin.for
./t3sis/tesis/source/invisidos2fin_OMP_def.for
./t3sis/tesis/source/invisidos2fin_OMP-origfuncionando.for
./t3sis/tesis/source/invisidos2fin_OMP_80.for

real    0m0.089s
user    0m0.060s
sys     0m0.024s
```

Figura 2.9: Ejemplo de ejecución del comando *time*.

2.7.1.4. Herramientas de perfilado de programa

El perfilado muestra cuales funciones son las más costosas en las ejecuciones de una aplicación. Es necesario utilizar para la medición casos de test representativos y múltiples, de manera de obtener resultados significativos. En GNU/Linux se cuenta con la herramienta "gprof" para realizar perfilado de aplicaciones. Para utilizarla, un programa debe estar compilado con la opción "-pg". Luego se ejecuta el programa una vez y genera un archivo llamado gmon.out en el directorio de ejecución el cual es utilizado por el comando gprof para generar el reporte de perfilado para esa ejecución. La sintaxis de gprof es:

```
$ gprof  <programa_ejecutable>  [<ruta_a_gmon.out>]
```

Si no se le pasa la ruta a gmon.out, por defecto utiliza el directorio desde donde es invocado gprof. Un ejemplo de este proceso puede verse a continuación:

```
$ gfortran -pg foo.for -o foo
$ foo
$ gprof foo
```

La salida de gprof es por salida estándar y bastante extensa, por lo cual es aconsejable redirigirla a un archivo. Consta de tres partes: la primera parte lista las funciones ordenadas de acuerdo

al tiempo que consumen, junto con sus descendientes (tiempo inclusivo). La segunda parte lista el tiempo exclusivo para las funciones (tiempo empleado ejecutando la función) junto con los porcentajes de tiempo total de ejecución y número de llamadas. La última parte da un índice de todas las llamadas realizadas en la ejecución.

2.7.2. Optimización Paralela

Luego de que la aplicación está optimizada para procesamiento serial, su tiempo de ejecución puede ser reducido aún más permitiendo que se ejecute en varios procesadores. Las técnicas más usadas comúnmente para paralelización son el uso explícito de hilos, el uso de directivas al compilador y el pasaje de mensajes [GS01]. En la Fig. 2.10 se ve ilustrado el proceso de optimización paralela.

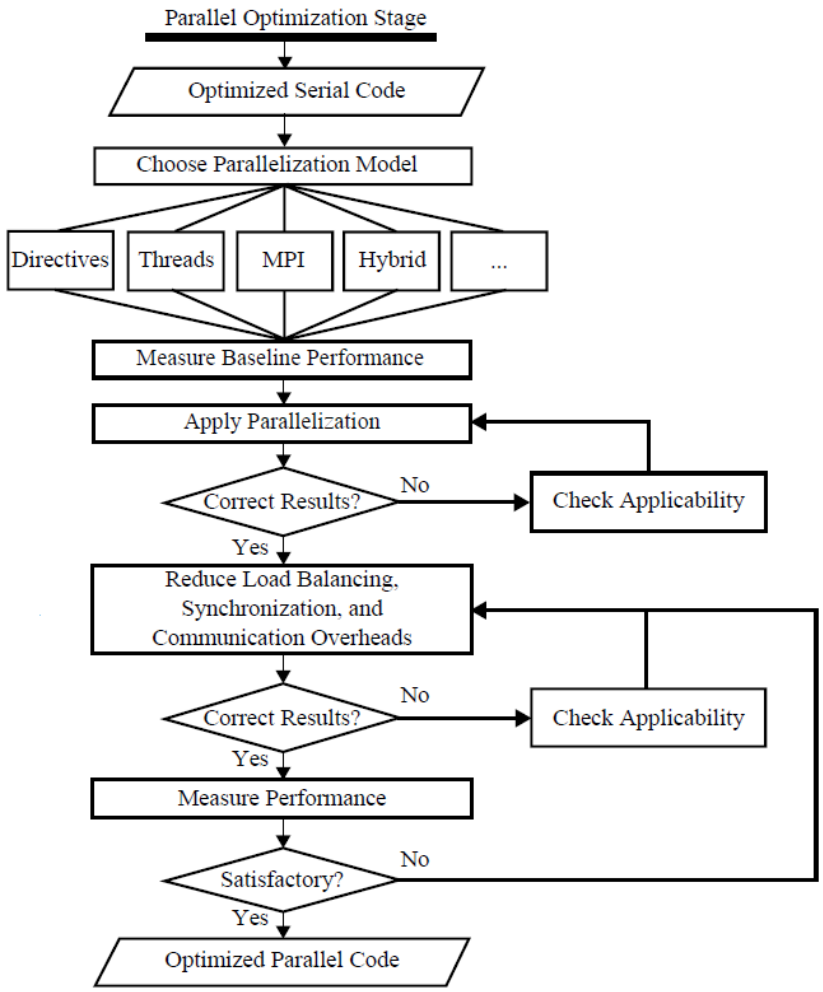


Figura 2.10: Proceso de Optimización Paralela.
Fuente: [GS01].

El primer paso es elegir un modelo, identificar que partes del programa deben ser paralelizadas y determinar como dividir la carga de trabajo computacional entre los diferentes procesadores. Dividir la carga de trabajo computacional es crucial para el desempeño, ya que determina los gastos generales de comunicación, sincronización y de desequilibrios de carga resultantes en un programa paralelizado. Generalmente, una división de trabajo de "nivel grueso" es recomendada debido a que minimiza la comunicación entre las tareas paralelas, pero en algunos casos, un enfoque de este tipo lleva a un balanceo de carga muy pobre; un nivel más fino en la división de

la carga de trabajo puede llevar a un mejor balanceo de carga y desempeño de la aplicación.

Luego de seleccionado un modelo de paralelización e implementado, lo siguiente es optimizar su desempeño. Similar a la optimización serial, este proceso es iterativo e involucra mediciones repetidas seguidas de aplicar una o más técnicas de optimización para mejorar el desempeño del programa. Las aplicaciones paralelas, sin importar el modelo utilizado, necesitan que exista comunicación entre los procesos o hilos concurrentes. Se debe tener cuidado de minimizar los gastos extras en comunicación y asegurar una sincronización eficiente en la implementación; también se debe minimizar el desequilibrio de cargas entre las tareas paralelas, ya que esto degrada la escalabilidad del programa. También es necesario considerar temas como migración y programación de procesos, y coherencia de cache. Las librerías del compilador pueden ser utilizadas para implementar versiones paralelas de funciones usadas comúnmente, tanto en aplicaciones multithread como multiproceso.

Los cuellos de botella de un programa paralelo pueden ser muy diferentes de los presentes en una versión secuencial del mismo programa. Además de gastos extra específicos de la paralelización, las porciones lineales (o secuenciales) de un programa paralelo pueden limitar severamente la ganancia de velocidad de la paralelización. En tales situaciones , hay que prestar atención a esas porciones secuenciales para mejorar el desempeño total de la aplicación paralela. Por ejemplo, consideremos la solución directa de N ecuaciones lineales. El costo computacional escala en el orden de $O(N^3)$ en la etapa de descomposición de la matriz y en el orden de $O(N^2)$ en la etapa de sustitución inversa. En consecuencia, la etapa de sustitución inversa apenas se nota en el programa secuencial, y el desarrollador paralelizando el programa justificadamente se enfoca en la etapa de descomposición de la matriz. Posiblemente, como resultado del trabajo de paralelización, la etapa de descomposición de la matriz se vuelve más eficiente que la etapa de sustitución inversa. El desempeño total y velocidad del programa de resolución directa ahora está limitado por el desempeño de la etapa de sustitución inversa. Para mejorar aún más el desempeño, la etapa de sustitución inversa debería convertirse en el foco de optimización y posiblemente un trabajo de paralelización [GS01].

Capítulo 3

Optimización e implementación de multiprocesamiento

3.1. Introducción

En los capítulos anteriores presentamos la problemática por la cual surge la idea y la necesidad de paralelizar una aplicación, así como las herramientas a utilizar, en nuestro caso OpenMP bajo Fortran. La elección del lenguaje Fortran se debe a que el usuario de la aplicación utilizada es también su programador, de manera que, para hacer el cambio lo más transparente posible, se decide no alterar este aspecto del programa.

Con Fortran como base, y teniendo en cuenta la estructura del programa con un análisis inicial del mismo, debido a las características de programa estructurado, monolítico y no modularizado, se elige orientar la solución a aplicar concurrencia en un entorno de Memoria Compartida y dejar habilitada la ejecución paralela en un equipo multiprocesador.

La aplicación bajo estudio utiliza archivos de datos en disco para guardar resultados, tanto parciales como finales. Esta actividad de entrada/salida introduce importantes demoras en el tiempo de respuesta que necesitamos considerar. En este capítulo describiremos el proceso seguido para la optimización del código Fortran en lo relacionado con el manejo de los archivos. Esta primera fase de optimización permitirá la paralelización de segmentos de código.

También realizaremos un análisis del perfil de ejecución de la aplicación, con una herramienta de perfilado que permitirá identificar qué subrutinas son las que más tiempo consumen y cuáles son las más indicadas para aplicar la paralelización.

Por último veremos la forma como se ha aplicado OpenMP a las partes seleccionadas de la aplicación. Explicaremos por qué han sido seleccionadas ciertas construcciones específicas del código y las razones de modificar algunas estructuras de control para hacer más eficiente la utilización de la memoria y de la CPU.

3.2. Análisis de la aplicación

Como se explicó en la sección 2.7, se debe determinar la plataforma en que debería ejecutarse la aplicación, estableciendo versión de sistema operativo y arquitectura. La aplicación recibida fue utilizada por su programador en arquitectura x86 de 32 bits, bajo sistema operativo GNU/Linux, específicamente con la distribución CentOS.

Lo primero fue obtener resultados base de ejecuciones de la aplicación bajo ese entorno, a fin de tener una referencia para la comparación de resultados. El autor de la aplicación nos indicó que la misma es completamente determinística, con lo cual la aplicación, con los mismos datos de entrada provistos, debe arrojar los mismos resultados en todas las ejecuciones. El autor nos proveyó con dos conjuntos de resultados correctos con los cuales se puede verificar la aplicación.

Para llevar a cabo el trabajo de la Tesis se seleccionó el entorno GNU/Linux, con la distribución Slackware de 64 bits como base, a la cual no fue necesario agregar componentes ni efectuar ninguna compilación especial. Se verificó que la aplicación entregada por el usuario compilara correctamente sin ninguna modificación en esta plataforma y arrojara, para los datos de entrada, exactamente los mismos resultados que en su entorno original.

Como vimos en el capítulo anterior, lo primero antes de optimizar es tener una aplicación que produzca resultados correctos. En nuestro caso se nos presentó una aplicación ya depurada y funcionando correctamente, así que pasamos a la parte de optimización, donde se deben seleccionar previamente los casos de test para validar que la optimización sigue produciendo resultados correctos.

Como se indicó previamente en esta sección, contamos con dos conjuntos de resultados provistos por el autor de la aplicación que serán los casos de test, los cuales se identifican por dos parámetros, *nr* y *no* que definen, respectivamente, la cantidad total de palas y de nodos sobre los cuales se va a realizar la simulación. Con estos parámetros se definen los casos de test, con valores iguales para ambos datos: nr y $no = 50$ en el primer caso de test, nr y $no = 80$ en el segundo caso.

Estos valores también definen variables globales comunes de la aplicación denominadas *maxir* y *maxio* que se establecen a los valores *nr+1* y *no+1* respectivamente. Los valores están codificados directamente en la aplicación y no se utiliza ningún tipo de constante simbólica que los defina, algo que sería más adecuado para su tratamiento y para tener un código más limpio; esto no se modificó y se mantuvo el tratamiento original de los valores para alterar lo menos posible el código.

Por el mismo motivo, tampoco se modificó la obtención de los valores de entrada para las simulaciones a partir de un archivo de texto.

3.2.1. Análisis de perfilado

Como paso preliminar de la optimización realizamos análisis de la aplicación con la herramienta de perfilado gprof, para poder comparar los principales puntos de consumo de tiempo con anterioridad a la optimización y luego de la misma. De esta forma se pretende seleccionar una o varias subrutinas para la paralelización y observar de qué manera cambia el comportamiento de la aplicación con la optimización.

Los datos obtenidos mediante gprof en esta etapa muestran que la subrutina *estela* resulta ser la que consume el mayor porcentaje, 79,83 % del tiempo de ejecución de la aplicación. Le sigue la subrutina *solgauss* con un 14,36 %. Estos datos se pueden observar en la Fig. 3.1.

Con estos resultados se pudo inferir en esta primer revisión que estas dos subrutinas son las candidatas a ser optimizadas con procesamiento paralelo.

Para las pruebas se utilizaron dos computadoras de escritorio distintas, ambas de arquitectura multiprocesador. El primer equipo posee un procesador AMD Phenom II con 4 núcleos y 4GB de memoria RAM. El segundo equipo consta de un procesador Intel Core i3 con 2 núcleos con SMT [EEL+97] (cada núcleo con 2 hilos de ejecución) y 6 Gb de RAM. Las especificaciones completas son provistas en el Capítulo 4 donde se analizan los resultados obtenidos.

La salida de la Fig. 3.1 fue obtenida en el primer equipo. Realizamos el mismo análisis de perfilado sobre el segundo equipo, y observamos que la mayor porción del tiempo sigue siendo consumida por la subrutina *estela* seguida por *solgauss* casi en los mismos porcentajes, 74,26 % y 16,84 % respectivamente. También es de notar la mejora en los tiempos de ejecución. Esto se puede observar en la Fig. 3.2.

A esta altura del trabajo contamos con "código correcto no optimizado" (Unoptimized Correct Code) [GS01], de modo que, siguiendo las etapas del proceso de optimización (ilustradas en la Fig. 2.7) visto en el capítulo 2, debemos efectuar una optimización serial para obtener código optimizado. Luego de esto podremos pasar a la etapa de "Optimización Paralela", donde aplicaremos paralelización al código para obtener justamente código paralelo optimizado.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
79.83	768.00	768.00	10	76.80	76.80	estela_
14.36	906.17	138.16	1	138.16	138.16	solgauss_
3.56	940.38	34.21	248750000	0.00	0.00	segmento_
1.80	957.70	17.32	10	1.73	5.15	anillo_
0.23	959.91	2.21	10	0.22	0.22	coefin_
0.19	961.71	1.80	10	0.18	0.18	veloc_
0.03	961.97	0.26	1	0.26	138.42	circulac_
0.01	962.09	0.12	1	0.12	0.12	geomest_
0.00	962.09	0.00	50899	0.00	0.00	radloc_
0.00	962.09	0.00	8	0.00	0.00	puntos1_
0.00	962.09	0.00	2	0.00	0.00	ploteo1_
0.00	962.09	0.00	2	0.00	0.00	ploteo2_
0.00	962.09	0.00	2	0.00	0.00	presion2_
0.00	962.09	0.00	1	0.00	962.09	MAIN__
0.00	962.09	0.00	1	0.00	0.00	cargas_
0.00	962.09	0.00	1	0.00	0.00	circo_
0.00	962.09	0.00	1	0.00	0.00	coord3d_
0.00	962.09	0.00	1	0.00	0.00	gammas_
0.00	962.09	0.00	1	0.00	0.00	hilo_
0.00	962.09	0.00	1	0.00	0.00	input_
0.00	962.09	0.00	1	0.00	0.00	palas_
0.00	962.09	0.00	1	0.00	0.00	panel_
0.00	962.09	0.00	1	0.00	0.00	presion1_
0.00	962.09	0.00	1	0.00	0.00	velsuper_

Figura 3.1: Salida de *gprof* en el primer equipo.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
74.26	156.55	156.55	10	15.65	15.65	estela_
16.84	192.04	35.49	1	35.49	35.49	solgauss_
5.59	203.83	11.79	248750000	0.00	0.00	segmento_
2.27	208.61	4.78	10	0.48	1.66	anillo_
0.50	209.67	1.06	10	0.11	0.11	coefin_
0.46	210.63	0.96	10	0.10	0.10	veloc_
0.05	210.74	0.11	1	0.11	35.60	circulac_
0.04	210.82	0.08	50899	0.00	0.00	radloc_
0.00	210.82	0.00	8	0.00	0.00	puntos1_
0.00	210.82	0.00	2	0.00	0.00	ploteo1_
0.00	210.82	0.00	2	0.00	0.00	ploteo2_
0.00	210.82	0.00	2	0.00	0.00	presion2_
0.00	210.82	0.00	1	0.00	210.82	MAIN__
0.00	210.82	0.00	1	0.00	0.00	cargas_
0.00	210.82	0.00	1	0.00	0.00	circo_
0.00	210.82	0.00	1	0.00	0.00	coord3d_
0.00	210.82	0.00	1	0.00	0.00	gammas_
0.00	210.82	0.00	1	0.00	0.08	geomest_
0.00	210.82	0.00	1	0.00	0.00	hilo_
0.00	210.82	0.00	1	0.00	0.00	input_
0.00	210.82	0.00	1	0.00	0.00	palas_
0.00	210.82	0.00	1	0.00	0.00	panel_
0.00	210.82	0.00	1	0.00	0.00	presion1_
0.00	210.82	0.00	1	0.00	0.00	velsuper_

Figura 3.2: Salida de *gprof* en el segundo equipo.

En las siguientes secciones veremos cómo realizamos estas dos etapas del proceso para obtener nuestra aplicación de estudio en forma optimizada paralela.

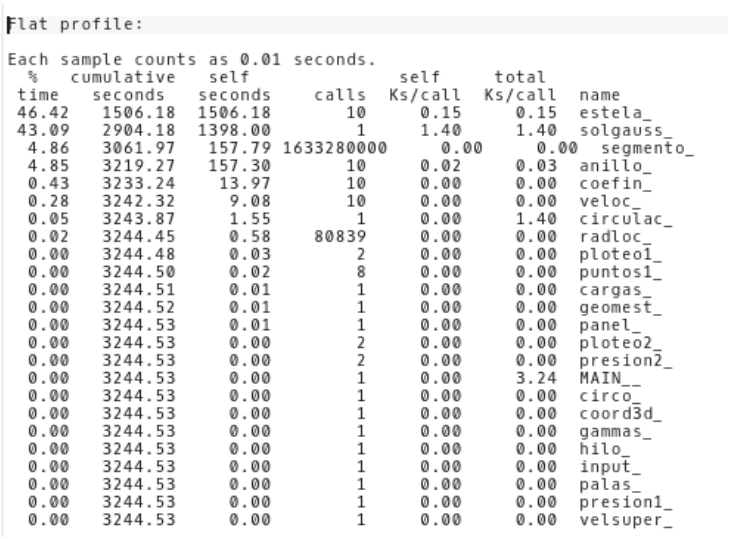
3.2.2. Perfilado de aplicación para el problema de tamaño de 80x80

Realizamos un nuevo análisis de perfilado con la herramienta *gprof* sobre la aplicación adaptada al problema de tamaño 80x80 paneles, ya que esto puede afectar el comportamiento de las subrutinas.

Luego de compilar la aplicación con la opción "-pg" activada, la ejecutamos y obtenemos el archivo *gmon.out* de salida. Con esto podemos generar la información del perfilado, el cual indica que la subrutina estela es la que más porcentaje del tiempo se ejecuta seguida de solgauss, pero esta vez los porcentajes cambian completamente. En la Fig. 3.3 podemos observar que estela se ejecuta 46,42 % del tiempo mientras que solgauss ahora ocupa un 43,09 %, esto es mucho más que el 14,36 % en PC1 o el 16,84 % en PC2 obtenido por solgauss para la versión de 50x50.

Este cambio que se produce en la ejecución al agrandar el tamaño del problema, tendrá impacto en los tiempos de las distintas versiones de la aplicación como veremos en el siguiente

capitulo.



Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ks/call	total Ks/call	name
46.42	1506.18	1506.18	10	0.15	0.15	estela_
43.09	2904.18	1398.00	1	1.40	1.40	solgauss_
4.86	3061.97	157.79	1633280000	0.00	0.00	segmento_
4.85	3219.27	157.30	10	0.02	0.03	anillo_
0.43	3233.24	13.97	10	0.00	0.00	coefin_
0.28	3242.32	9.08	10	0.00	0.00	veloc_
0.05	3243.87	1.55	1	0.00	1.40	circulac_
0.02	3244.45	0.58	80839	0.00	0.00	radloc_
0.00	3244.48	0.03	2	0.00	0.00	ploteo1_
0.00	3244.50	0.02	8	0.00	0.00	puntos1_
0.00	3244.51	0.01	1	0.00	0.00	cargas_
0.00	3244.52	0.01	1	0.00	0.00	geomest_
0.00	3244.53	0.01	1	0.00	0.00	panel_
0.00	3244.53	0.00	2	0.00	0.00	ploteo2_
0.00	3244.53	0.00	2	0.00	0.00	presion2_
0.00	3244.53	0.00	1	0.00	3.24	MAIN__
0.00	3244.53	0.00	1	0.00	0.00	circo_
0.00	3244.53	0.00	1	0.00	0.00	coord3d_
0.00	3244.53	0.00	1	0.00	0.00	gammas_
0.00	3244.53	0.00	1	0.00	0.00	hilo_
0.00	3244.53	0.00	1	0.00	0.00	input_
0.00	3244.53	0.00	1	0.00	0.00	palas_
0.00	3244.53	0.00	1	0.00	0.00	presion1_
0.00	3244.53	0.00	1	0.00	0.00	velsuper_

Figura 3.3: Salida de *gprof*. Aplicación para problema de tamaño 80x80.

3.3. Optimización Serial del código Fortran

La optimización serial es "un proceso iterativo que involucra medir repetidamente un programa seguido de optimizar sus porciones críticas" [GS01]. Obtenidas las mediciones iniciales del comportamiento, debemos utilizar las opciones del compilador en línea de comandos que permitan la mayor optimización del código (en recursos y velocidad de ejecución) y vincular el código objeto con bibliotecas optimizadas. En el trabajo de tesis se intenta reducir al mínimo las modificaciones al código, por lo cual las opciones que se utilizarán para la compilación serán únicamente las referidas a la infraestructura de programación paralela de OpenMP.

Por lo demás, la aplicación no hace uso de ninguna otra biblioteca que no se cuente entre las que utiliza regularmente el compilador para construir la aplicación. Como buscamos observar el impacto de optimizar serialmente el código y aplicar paralelización, no se utilizan bibliotecas que pudieran optimizar otras partes del programa.

3.3.1. Análisis del acceso a datos de la aplicación

Al analizar los resultados de ejecución de la aplicación observamos que maneja gran cantidad de archivos en disco, tanto de texto como binarios (temporales). En el directorio de la aplicación aparecen 34 archivos de extensión TXT, 9 archivos PLT, 5 archivos OUT y 8 archivos TMP. A éstos se agregan el propio archivo fuente .for, el ejecutable invisidosExe, el archivo con los datos de entrada entvis2f.in, y el que utilizamos para almacenar los datos de gprof, invisidosExegprof. En la Fig. 3.4 se puede observar el listado del directorio de ejecución.

Los tamaños de la mayoría de los archivos van desde 8 KB hasta 2 MB, pero los archivos TMP pueden alcanzar un tamaño de varios megabytes (se observan algunos del orden de los cientos de megabytes). Esto evidencia que una corrida de la aplicación intercambia un volumen significativo de datos entre la aplicación y el sistema de archivos.

Con el fin de localizar las subrutinas mas adecuadas para la optimización, relevamos la relación entre cada archivo y las subrutinas que lo acceden, indicando las operaciones realizadas (escritura, lectura, lectura/escritura, rewind). De este relevamiento se obtiene la lista de archivos que únicamente son escritos en disco, y los que son además leídos, por cada subrutina. La relación de archivos y subrutinas se muestra en la tabla 3.1.

```
h4ndr3s@gondolin:~/pruebas/oldone\$ ls
alfa.txt        cp2.txt          integ1.txt              vel02int.txt
arco.txt        cpei.plt         integ2.txt              vel02pvn.txt
cindg.tmp       cpei.txt         invisidos2fin.for          velcapae.txt
circo.txt       cr.txt           invisidosExe*           velcapai.txt
cix1.tmp        entvis2f.in      invisidosExegprof       velindad.plt
cix2.tmp        estel1.txt       palas.plt               velindad.txt
ciy1.tmp        estel2.txt       palest.plt              velpotad.txt
ciy2.tmp        estel3.txt       panel.plt               velresad.plt
ciz1.tmp        fuerza.plt       panel.txt               veltotal.txt
ciz2.tmp        fzas.txt         pres.plt                velxyz.txt
co.txt          gama.txt         salida2.out             vix.txt
coefg.tmp       gdifo.out        subr.out                viy.txt
coefp.txt       gdifr.out        vector.plt              viz.txt
coord.txt       gmon.out         vel01ext.txt            vn.txt
cp075c.txt      go.out           vel01int.txt
cp1.txt         gr.out           vel02ext.txt
```

Figura 3.4: Listado del directorio luego de la ejecución del programa

Fortran ofrece los archivos regulares, o *archivos externos* soportados en disco (*External Files*), pero también los *archivos internos* (*Internal Files*), que son cadenas de caracteres o arreglos de cadenas de caracteres, localizados en memoria principal. Los *archivos internos* se manejan con las mismas funciones que los *archivos externos*, y la única restricción para su uso es la cantidad de memoria virtual del sistema. Como la latencia de los accesos a disco magnético es, normalmente, al menos cinco órdenes de magnitud mayor que la de los accesos a memoria principal [Gre13], cambiando la definición de los archivos en disco a *archivos internos* (siempre que la restricción de tamaño del sistema de memoria virtual lo permita) conseguimos una mejora sustancial de desempeño de la aplicación, sin ninguna modificación importante al código ni al comportamiento del programa.

3.3.2. Optimización por adaptación de archivos externos a internos

La primera decisión tomada para la optimización del código es reducir el impacto de los accesos a archivos en disco que son leídos y además escritos por la aplicación. No efectuaremos ninguna modificación sobre los archivos que son únicamente escritos por las subrutinas, con cuatro excepciones: la escritura de los archivos *integ1.txt* e *integ2.txt* en la subrutina *estela*, retrasada hasta el final de la misma, y los archivos *salida2.out*, que guarda resultados de la ejecución a medida que avanza, y *subr.out*, que recoge lo mostrado en salida estándar. Estos archivos se guardarán en objetos de tipo Archivo Interno (Internal File) de Fortran y su escritura se demorará hasta la finalización del programa. La elección de no pasar más archivos a archivos internos es para evitar un incremento elevado en la cantidad de memoria utilizada por la aplicación.

Un caso especial es el archivo interno *outstd* que lleva lo impreso en salida estándar dentro de algunas subrutinas (estela, geomest, etc), y es mostrado por pantalla al retornar dichas subrutinas al programa principal.

Luego, todo archivo externo que sea escrito y leído durante la ejecución de la aplicación será mantenido por un archivo interno. La única modificación necesaria al código será el cambio de las referencias a los archivos en las sentencias “write”, “read” y “rewind”. En la Fig. 3.5 se muestra un ejemplo de código previo a la modificación, y en la Fig. 3.6 el código ya modificado.

Como se ve, reemplazamos el archivo *subr.out* representado por el identificador de unidad 15 por el archivo interno denominado subrout.

Como se ha dicho, el archivo externo *subr.out* pasa a ser manejado como un archivo interno, que como se ve en la declaración de la Fig. 3.6, es un arreglo de 500 cadenas de 60 caracteres como máximo. La variable nsubr mantiene la posición en el archivo interno a ser escrita, y el argumento “1” en los comandos write es un formato de escritura definido dentro del programa

Archivos	Función	Operación
cpei.plt, cpei.txt	presion2	write only
palest.plt	palas, geomest	write only
palas.plt, alfa.txt, coord.txt	palas	write only
panel.plt, panel.txt	panel	write only
pres.plt, cp075c.txt, coefp.txt, cp1.txt, cp2.txt	presion1	write only
velxyz.txt, velpotad.txt, velindad.txt velcapae.txt, velcapai.txt, vix.txt viy.txt, viz.txt	veloc	write only
estel1.txt, estel2.txt, estel3.txt	geomest	write only
gama.txt	circulac	write only
	gammas	read only
veltotal.txt, vel01ext.txt, vel01int.txt vel02ext.txt, vel02int.txt, vel02pvn.txt	velsuper	write only
velresad.plt	ploteo2	write only
arco.txt	circulac	write only
	circo	read only
fzas.txt, fuerza.plt, vector.plt	cargas	write only
coefg.tmp, cindg.tmp	circulac	write y rewind
	solgauss	read y rewind
cix1.tmp, ciy1.tmp, ciz1.tmp	anillo	write y rewind
	coefin	read y rewind
	-	-
cix2.tmp, ciy2.tmp, ciz2.tmp	coefin	write y rewind
	circulac	read y rewind
	veloc	read y rewind
salida2.out	input	write only
	cargas	write only
	anillo	write only
	veloc	write only

Tabla 3.1: Relación archivos y funciones de la aplicación

como se explicaba en el capítulo anterior. En la tabla 3.2 vemos cómo quedan las equivalencias de los archivos externos y su correspondiente cambio a archivo interno.

El proceso fue realizado primero en la subrutina Estela, buscando mejorar sus tiempos al convertir el manejo de los archivos *integ1.txt* e *integ2.txt* en archivos internos, retrasando la escritura en disco de los datos hasta el final de la subrutina. Lo primero que se observa luego de esta modificación es un comprensible incremento del uso de memoria de la aplicación, pasando de un uso de 200 a 202 MB, originalmente, sin aplicar ninguna modificación, a utilizar 205 MB con la modificación indicada en el tratamiento de los archivos. Es un cambio en principio poco significativo, pero con las modificaciones sucesivas se verá el impacto en la utilización de memoria.

De acuerdo a la tabla de funciones y archivos, y al análisis efectuado mediante gprof, procedimos a modificar las subrutinas Solgauss y Circulac que son las que leen y escriben los archivos TMP respectivamente, archivos que consumen la mayor cantidad de espacio en disco de los utilizados por la aplicación. Antes de realizar el cambio directamente, analizamos qué estructura sería la más adecuada para alojar los resultados, ya que los archivos TMP eran binarios sin formato, que transportaban valores calculados de una subrutina a otra.

```
      open(unit=15,file='subr.out')
      ...
      write(15,1)
      write(6,1)
```

Figura 3.5: Ejemplo de código sin modificar

```
      character subrout(500)*60   ! Internal File
      ...
      write(subrout(nsubr),1)
      nsubr=nsubr+1
!     write(15,1)
      write(6,1)
```

Figura 3.6: Ejemplo de código modificado para utilizar archivo interno

Seleccionamos primero los archivos coefg.tmp y cindg.tmp (definidos como units 40 y 41 respectivamente al principio de la aplicación original) ya que eran los de menor tamaño de todos los archivos tipo TMP. Como observamos en la tabla 3.1, los archivos mencionados son escritos en la subrutina "circulac" y leídos en *solgauss* (además de los rewind).

La subrutina circulac, como indica en sus comentarios realiza el cálculo de la circulación asociada a la estela y a cada anillo vorticoso. Está dividida en tres partes, siendo la primer parte la que realiza la escritura de los archivos coefg.tmp y cindg.tmp, y donde para estos cálculos lee los archivos tmp cix2.tmp, ciy2.tmp y ciz2.tmp, los cuales no son modificados en esta etapa. La segunda parte realiza la resolución de un sistema de npa*npa ecuaciones algebraicas y lo hace llamando a la subrutina solgauss que veremos a continuación. En la tercer parte con los resultados obtenidos se calculan otros valores que se escriben en otros archivos de resultados.

Como la subrutina "circulac" es la que crea los archivos coefg.tmp y cindg.tmp analizamos las estructuras de control utilizadas para generar dichos archivos.

El bucle externo controlado por el "do 1" realiza el equivalente a *npan* iteraciones, con lo cual podemos concluir que el archivo determinado por la unit 41 (lo sabemos por el write(41)), es decir cindg.tmp, almacena un total de *npan* resultados. El bucle interno controlado por el "do 2" realiza *npan* * *npan* iteraciones, por lo tanto el archivo determinado por la unit 40 (write(40)), i.e. coefg.tmp, almacena *npan* * *npan* resultados.

Analizado esto podemos definir que los tamaños de los archivos internos para dichos archivos serán de *npan* y *npan* * *npan*. Luego podemos ver que las variables *coefg* y *cindg* que almacenan los resultados para escribir en los archivos no están tipificadas explícitamente en el código, con lo cual observamos en el bloque common de toda la aplicación (repetido en cada subrutina) que se realiza la siguiente declaración:

```
    implicit real*8 (a-h,o-z)
```

la que indica que cualquier variable no tipificada definida en el código cuyo nombre comience con una letra entre los rangos indicados (a-h y o-z) será declarada, implícitamente, como real * 8, por lo cual podemos asegurar que *coefg* y *cindg* son de tipo real * 8. Con esto determinado podemos declarar archivos internos de tipo real * 8 de tamaños *npan* y *npan* * *npan* para reemplazar a cindg.tmp y coefg.tmp respectivamente:

```
    real*8 cindgtmp(npan),coefgtmp(npan*npan)
```

siendo cindgtmp el archivo interno para cindg.tmp y coefgtmp el archivo interno para coefg.tmp.

Ahora debemos reemplazar las escrituras de los archivos binarios en disco con los archivos internos de la siguiente manera, donde existían las siguientes operaciones de escritura:

Archivo en Disco	archivo interno
integ1.txt	integ1
integ2.txt	integ2
salida2.out	salida2out
subr.out	subrout
gama.txt	gamastr
circo.txt	circostr
coefg.tmp	coefgtmp
cindg.tmp	cindgtmp
cix1.tmp	cix1tmp
ciy1.tmp	ciy1tmp
ciz1.tmp	ciz1tmp
cix2.tmp	cix2tmp
ciy2.tmp	ciy2tmp
ciz2.tmp	ciz2tmp
<salida estándar>	outstd

Tabla 3.2: Equivalencias Archivo en Disco a Archivo Interno.

```
   write(40)coefg
   write(41)cindg
```

reemplazamos con el siguiente código:

```
   coefgtmp(incoefg)=coefg
   cindgtmp(npa)=cindg
```

respectivamente.

La variable *incoefg* es utilizada para marcar la posición en el array de *npan* * *npan* elementos, archivo interno coefgtmp, por cada vez que entramos en el bucle interior. Como es un array de dimensión 1 (igual al archivo binario que reemplaza) es necesario tener guardada la última posición accedida por cada iteración del bucle externo. Para el archivo interno cindgtmp que reemplaza a cindg.tmp con utilizar la variable "npa" es suficiente, ya que lleva exactamente la posición en el array por cada iteración (es la variable de control del bucle).

En el siguiente extracto de código observamos las estructuras DO mencionadas que aparecen al principio de "circulac" [Pra07]:

```
   do 1 npa=1,npan
   do 2 nv =1,npan
   [...]
   coefg= sumbcx*vnx(npa)+sumbcy*vny(npa)+sumbcz*vnz(npa)
   write(40)coefg
 2 continue
   [...]
   cindg= (-1.)*(vtgx(npa,1)*vnx(npa)+vtgy(npa,1)*vny(npa)+
 &              UU*vnz(npa))
   write(41)cindg
 1 continue
```

Como explicamos, la segunda parte de "circulac" llama a la subrutina *solgauss*, y previamente

habíamos dicho que los archivos cindg.tmp y coefg.tmp que estamos reemplazando son escritos por la primera subrutina y leídos por la segunda. En *solgauss* el cambio es simple, tenemos dos bucles anidados que iteran de la misma manera que en "circulac", sólo que leen los datos almacenados en los archivos TMP. Luego de esto hacen rewind de los archivos para que vuelvan a quedar disponibles para lectura al principio de los mismos. A continuación podemos ver el código original [Pra07]:

```
   m=npan+1

   do 1 i=1,npan
   do 2 j=1,npan
   read(40)cfg
   coefg(i,j)=cfg
 2 continue
   read(41)cig
   coefg(i,m)=cig
 1 continue

   rewind(40)
   rewind(41)
```

Aquí se leen ambos archivos para armar una matriz con la variable denominada coefg, la cual tiene *npan* filas y npan+1 columnas, realizando lo siguiente: en cada fila almacena en los primeros *npan* valores, o primeras *npan* columnas, los datos obtenidos de coefg.tmp, y en último lugar, columna npan+1, el dato obtenido de cindg.tmp.

Para permitir que *solgauss* pueda trabajar con el cambio que introdujimos es necesario que reciba de alguna manera las referencias a los archivos internos. Esto lo conseguimos simplemente pasando por parámetro los mismos. El cambio en el código sería el siguiente:
Código original definición de subrutina

```
   subroutine solgauss(npan,gama)
    ...
```

Código modificado

```
   subroutine solgauss(npan,gama,tmpcoefg,tmpcindg)
   ...
   real*8 tmpcoefg(mxro*mxro),tmpcindg(mxro)
```

Aquí tmpcoefg y tmpcindg son los nombres con los que identifica la subrutina a los archivos internos, y ambos arrays deben ser declarados explícitamente en la sección correspondiente.

Luego de que *solgauss* conoce la existencia de los archivos internos necesarios, modificamos los bucles de control para que los utilicen.
El código visto previamente de solgauss quedó de la siguiente manera:

```
    m=npan+1
    incfg=1

    do 1 i=1,npan
    do 2 j=1,npan
    !read(40)cfg
    incfg=((i-1)*npan)+j
    cfg=tmpcoefg(incfg)
    coefg(i,j)=cfg
    incfg=incfg+1
```

```
2 continue
   !read(41)cig
   cig=tmpcindg(i)
   coefg(i,m)=cig
1 continue

   !rewind(40)
   !rewind(41)
```

Como indicábamos, el cambio no es complicado. Lo primero que hicimos fue la inclusión de una variable de control "incfg" inicializada en 1 con la cual mantener la posición de la cual debe leerse desde tmpcoefg (que reemplaza a coefg.tmp) la próxima vez que se ingresa al bucle de control; luego cambiamos las sentencias read en disco de los archivos de texto por el acceso a los archivos internos (en memoria), utilizando una variable auxiliar extra para leer el dato y luego ingresarlo en la matriz "coefg". La variable auxiliar es utilizada para salvar errores aleatorios encontrados en los datos asignados al utilizar una asignación directa del archivo interno a la matriz "coefg". La variable "incfg" es utilizada para seguir la posición del archivo interno "tmpcoefg", ya que la posición del archivo interno "tmpcindg" puede ser llevada utilizando la variable de control del bucle, en este caso "i".

Estos cambios y ajustes para el recambio de archivos de texto por archivos internos (arrays en memoria) se realizó por cada uno de los archivos indicados en la tabla 3.2.

En su mayor parte el cambio es simple y consiste en modificar unas pocas líneas de código, como por ejemplo las que mantienen los archivos *subr.out* y *salida2.out* para postergar la escritura en disco de dichos archivos. Esos archivos internos son subrout y salida2out respectivamente, para los cuales agregamos la siguiente definición en el bloque "common":

```
   character salida2out(102)*95, subrout(500)*60
```

Y luego al utilizarlos llevar junto con ellos un contador que mantenga la posición siguiente para escribir, al cual llamamos nsubr para subrout:

```
   write(subrout(nsubr),1)
   nsubr=nsubr+1
```

y nsld2 para salida2out:

```
   write(salida2out(nsld2),21)indice,ncapa
   write(salida2out(nsld2+1),'(a1)') ""
   nsld2=nsld2+2
```

En estos ejemplos, recordamos del capítulo 2 que el número ubicado en el comando "write" al lado del archivo interno es una etiqueta de formato. La cantidad de elementos de estos arrays se corresponde con la cantidad de líneas que genera el archivo en disco.

En el resto del código el tratamiento de estos archivos es similar, variando solamente de acuerdo a qué datos deben ser escritos en el mismo, como observamos en los archivos internos que vimos previamente, los que reemplazan a coefg.tmp y cindg.tmp.

Un caso especial son los archivos internos cix1tmp, cix2.tmp, ciy1tmp, ciy2.tmp, ciz1tmp y ciz2.tmp, para los cuales sus homónimos archivos en disco (cix1.tmp, cix2.tmp, y así sucesivamente) son definidos en el programa original como "unformatted", i.e., sin formato, con lo cual se generan archivos en disco de tipo binario. Para obtener el mismo comportamiento en nuestros archivos internos debimos tener el cuidado de escribir en ellos sin dar formato a lo ingresado, i.e., los valores ingresan tal cual son generados por el programa. Veamos un ejemplo con cix1tmp.

El código para escribir los valores en el programa original es el siguiente:

```
        do 114 npa=1,npan
        do 113 nv=1,npan

        write(42)cix(npa,nv)
        ...
113 continue
114 continue
```

La apertura del archivo cix1.tmp le asigna al principio del programa la unidad 42 para referencia posterior en el programa y de ahí el descriptor utilizado por el write, mientras que la matriz "cix" es generada por cálculos previos. Al asignar directamente y no dar un formato a utilizar en el comando write, estamos escribiendo los valores "crudos" para ser almacenados.

El código en el programa optimizado es:

```
    common  ... cix1tmp(maxro*maxro), ...
    ...
        kon=1
        do 114 npa=1,npan
        do 113 nv=1,npan

        cix1tmp(kon)=cix(npa,nv)
        ...
        kon=kon+1
113 continue
114 continue
```

Aquí referenciamos primero la definición del archivo interno cix1tmp, y no se define un tipo por defecto, por lo que, como explicamos en párrafos anteriores, toma el tipo implicito real*8 definido en el bloque "common" de cada subrutina.

El tamaño del archivo interno (maxro * maxro) es definido por el mismo bucle que lo genera, que itera desde 1 a "npan" dentro de otro bucle que itera la misma cantidad de veces, i.e., genera npan*npan elementos en cix1tmp. La variable maxro definida "common" y con valor previamente asignado es equivalente a npan, y maxro es preferida a ésta ya que en el bloque de definición *npan* aún no tiene asignado su valor.

Por último la variable "kon" oficia de contador de posiciones para el archivo interno.

Luego de igual manera modificamos el código donde el archivo interno es leido por su equivalente interno.

El código original sería:

```
   read(42)cinfx
```

Optimizado con archivo interno:

```
   cinfx=cix1tmp(kon)
```

Donde nuevamente la variable "kon" lleva la posición dentro del archivo interno.

De igual manera son manejados los demás archivos externos binarios como archivos internos, los cuales mantienen la información necesaria en memoria y no en disco. El tiempo de lectura y escritura de dichos archivos decrece considerablemente, pasando de tiempos de acceso medidos en milisegundos para un disco rígido, a tiempos de acceso en nanosegundos para la memoria RAM, lo cual implica un aumento teórico de velocidad en varios órdenes de magnitud.

Obviamente esto trae aparejado una necesidad mayor de memoria RAM para el proceso ya que ésta debe ser capaz de contener la totalidad de los datos temporales que antes se contenían

en disco, creciendo dicha necesidad proporcionalmente con el tamaño del problema calculado. Por ello inferimos que es posible que ante un tamaño suficientemente grande del problema, su cálculo no sea viable en ciertos equipos. Tratamos este tema en el capítulo 5.

Por los motivos recién indicados,en el trabajo de optimización se decidió no pasar la totalidad de los archivos externos a archivos internos y no diferir su escritura al final de la ejecución del programa, sino que se seleccionaron los más críticos a efectos del cálculo: aquellos que eran escritos y leídos durante la ejecución del programa, y manteniendo como archivos externos todos aquellos de lectura exclusiva o escritura exclusiva.

En la tabla 3.3 se enumeran los archivos que se decidió manejar mediante un archivo interno y el motivo de dicha decisión:

Archivo en Disco	Archivo Interno	Motivo del Cambio
integ1.txt	integ1	Mejorar tiempo de subr. estela
integ2.txt	integ2	Mejorar tiempo de subr. estela
salida2.out	salida2out	Diferir escritura
subr.out	subrout	Diferir escritura
gama.txt	gamastr	Diferir escritura
circo.txt	circostr	Diferir escritura
coefg.tmp	coefgtmp	Evitar escrituras y lecturas de disco
cindg.tmp	cindgtmp	Evitar escrituras y lecturas de disco
cix1.tmp	cix1tmp	Evitar escrituras y lecturas de disco
ciy1.tmp	ciy1tmp	Evitar escrituras y lecturas de disco
ciz1.tmp	ciz1tmp	Evitar escrituras y lecturas de disco
cix2.tmp	cix2tmp	Evitar escrituras y lecturas de disco
ciy2.tmp	ciy2tmp	Evitar escrituras y lecturas de disco
ciz2.tmp	ciz2tmp	Evitar escrituras y lecturas de disco
<salida estándar>	outstd	Diferir salida estándar de algunas subrutinas

Tabla 3.3: Decisiones para cambio de Archivo en Disco a Archivo Interno.

Una vez realizados los cambios indicados, verificamos que los resultados siguieran siendo los correctos. Para ello comparamos los archivos de salida de la aplicación original con los generados por la misma aplicación, verificando que produzcan la misma salida. Al ser archivos de texto esto puede ser realizado con aplicaciones como *diff* o *vimdiff* [1], los cuales permiten verificar que ambos archivos tienen el mismo contenido o no.

Una vez verificado esto, a continuación pasamos a la siguiente etapa de optimización.

3.4. Optimización Paralela para Multiprocesamiento

Con el primer paso de optimización realizado es posible llevar a cabo la optimización paralela del código con el modelo de programación paralela seleccionado.

Como vimos en la sección 3.2.1, de acuerdo al resultado de la herramienta gprof, el código candidato para ser optimizado en ese primer momento era principalmente la subrutina *estela*, seguida de *solgauss*. Si compilamos nuestro programa nuevamente con el profiler de GNU (gprof), pero con la optimización de los archivos internos, obtenemos que la subrutina *estela* sigue siendo la de mayor peso en la ejecución, seguida de solgauss, incluso en porcentajes bastante aproximados a los obtenidos para el programa original. Esto lo podemos observar en la Fig. 3.7.

[1] Para una referencia de los comandos en GNU/Linux ver su manual: "man diff" o "man vimdiff".

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
75.38	172.69	172.69	10	17.27	17.27	estela_
17.13	211.94	39.25	1	39.25	39.25	solgauss_
4.89	223.14	11.20	248750000	0.00	0.00	segmento_
2.18	228.14	5.00	10	0.50	1.62	anillo_
0.19	228.57	0.43	10	0.04	0.04	coefin_
0.17	228.95	0.38	10	0.04	0.04	veloc_
0.03	229.03	0.08	50899	0.00	0.00	radloc_
0.03	229.10	0.07	1	0.07	39.32	circulac_
0.00	229.10	0.00	8	0.00	0.00	puntos1_
0.00	229.10	0.00	2	0.00	0.00	ploteo1_
0.00	229.10	0.00	2	0.00	0.00	ploteo2_
0.00	229.10	0.00	2	0.00	0.00	presion2_
0.00	229.10	0.00	1	0.00	229.10	MAIN__
0.00	229.10	0.00	1	0.00	0.00	cargas_
0.00	229.10	0.00	1	0.00	0.00	circo_
0.00	229.10	0.00	1	0.00	0.00	coord3d_
0.00	229.10	0.00	1	0.00	0.00	gammas_
0.00	229.10	0.00	1	0.00	0.08	geomest_
0.00	229.10	0.00	1	0.00	0.00	hilo_
0.00	229.10	0.00	1	0.00	0.00	input_
0.00	229.10	0.00	1	0.00	0.00	palas_
0.00	229.10	0.00	1	0.00	0.00	panel_
0.00	229.10	0.00	1	0.00	0.00	presion1_
0.00	229.10	0.00	1	0.00	0.00	velsuper_

Figura 3.7: Resultado de *gprof* en el código optimizado serialmente.

3.4.1. Análisis de la subrutina Estela

En la definición de la subrutina *estela*, el código documentado del programa indica que ésta realiza "cálculo de los coeficientes de influencia de los hilos libres". Los cálculos realizados dentro de la subrutina son numerosos y complejos, por lo cual utilizaremos un pseudocódigo para poder observar los puntos más importantes dentro de la subrutina que pueden ser candidatos a ser paralelizados. En la Fig. 3.8 aparece el pseudocódigo anotado de la subrutina estela.

Analizando el pseudocódigo podemos observar que la subrutina tiene partes bien diferenciadas. Un inicio, estableciendo valores iniciales y cálculos parciales, y luego un bloque conformado por dos bucles principales; dentro de ellos es donde se encuentran las estructuras que pueden ser paralelizadas.

El bucle inicial calcula los datos en fx, fy, fz, denom y dista; luego calcula términos pares e impares, y finaliza con el denominado cálculo de coeficientes de la estela x,y,z.

El cálculo de coeficientes parece ser el más complejo de los puntos indicados, pero si observamos bien, sólo se ejecuta una vez en todo el programa, cuando "indice" es igual a 1 (la variable "ib" siempre tiene valor 1, por lo cual no la contamos). Dicha variable "indice" es global al programa y controla las etapas por las que pasa, toma valores de 1 a 10 y no repite los valores.

Por otra parte, el cálculo de coeficientes se hace sobre los valores valx, valy y valz, realizando sobre ellos una sumatoria, con lo cual se crea una dependencia de datos entre el cálculo de un valor y los cálculos previos, ya que para obtener el valor de valx en un momento, es necesario el valor previo de valx. Si realizamos una paralelización del código tendríamos un problema en los límites de los distintos threads.

Por ejemplo, al dividir los datos en porciones de 100 elementos, el thread que calcula los valores 101 a 200 de un bucle necesita conocer el valor de la sumatoria en el valor 100 para poder iniciar con valores correctos su cálculo, y dicho valor 100 puede no existir aún en el momento en que se lo necesita (porque el thread encargado de su cálculo puede no haber finalizado o siquiera iniciado).

En el cálculo de los términos pares e impares se presenta el mismo problema de dependencia de datos que aparece en el cálculo de coeficientes. Cuando calculamos, por ejemplo, six, necesitamos conocer el valor previo de six en ese momento.

Existen técnicas y formas de transformar el código que permiten en algunos casos poder reprogramar una porción del mismo para que pueda ser paralelizable a pesar de tener esta dependencia de dato. Debido al potencial gran cambio necesario en el código para subsanar

```
1       subrutina estela()
2       definición de variables globales y constantes;
3
4       begin
5           Do de i=1 a 2500
6           Do de j=1 a 51
7                   ciex(i,j) = 0
8                   ciey(i,j) = 0
9                   ciez(i,j) = 0
10          end do
11          end do
12          %cálculos parciales?
13          ib = 1
14
15      Do de ir=1 a 2500
16      Do de npa=1 a 51
17          Do de ik=1 a 2001
18                  genera fx(ik), fy(ik), fz(ik), dista(ik), denom(ik)
19          end do
20
21      # sumatoria de términos impares six, siy, siz
22          six,siy,siz = 0
23          Do de ik=2 a 2000
24                  six = six + fx(ik)/denom(ik)
25                  siy = siy + fy(ik)/denom(ik)
26                  siz = siz + fz(ik)/denom(ik)
27                  ik = ik +2
28          end do
29      # sumatoria de términos pares spx, spy, spz
30          spx,spy,spz = 0
31          Do de ik=3 a 2000
32                  spx = spx + fx(ik)/denom(ik)
33                  spy = spy + fy(ik)/denom(ik)
34                  spz = spz + fz(ik)/denom(ik)
35                  ik = ik +2
36          end do
37          calculo ciex(ir,npa), ciey(ir,npa), ciez(ir,npa)
38          if (indice = 1) and (ib = 1) then  ## se ejecuta solo 1 vez
39          # calculo coeficientes de la estela x, y, z
40                  if (i = nr) and (j = nr/2) then
41                  # calculo para i=50 y j=25
42                  # nr depende del tamaño del problema,
43                  # en este caso el tamaño es 50
44                          inicializa valx,valy,valz
45                          Do de ik=1 a 2001
46                                  escribe archivo integ1.txt con varios valores incluyendo
                                        valx,valy,valz
47                                  if (ik =/= 2001) then
48                                          const=1
49                                          if (ik == 2000) then
50                                                  const=0.5
51                                          endif
52                                          valx = valx + fx(ik+1)/denom(ik+1)*otros valores
53                                          valy = valy + fy(ik+1)/denom(ik+1)*otros valores
54                                          valz = valz + fz(ik+1)/denom(ik+1)*otros valores
55                                  else
56                                          escribe integ1.txt con varios valores sin valx,
                                                valy,valz
57                                          pero con ciex,ciey,ciez(i,j)
58                                  endif
59                  else
60                          if (i = nr/2) and (j = nr+1) then
61                          # Luego (si no entró en el anterior if) el calculo es para i = 25
                                 y j=51
62                                  Repite mismo trabajo pero escribiendo integ2.txt
63                          endif
64                  endif
65          endif
66      end do
67      end do
68      end
```

Figura 3.8: Pseudocódigo de la subrutina *estela*.

el problema de la dependencia, y al requisito de no modificar el código de maneras que puedan volverlo ilegible para el usuario, es que no se avanzó sobre estas áreas de la subrutina. La solución a este problema puede ser motivo de un trabajo futuro que se pondrá a consideración en el capítulo 5.

Luego de descartar estos puntos como las zonas a paralelizar en la subrutina *estela* nos quedamos con el bucle de la Fig. 3.8 que genera los arrays fx, fy, fz, denom y dista (lineas 17 a 19), ya que cada valor generado de estos arrays no depende de otros previos dentro de los arrays.

3.4.2. Optimización con OpenMP de subrutina Estela

Seleccionado el bucle a paralelizar hicimos un análisis de los datos que intervienen para poder realizar una optimización correcta. Realizamos varias pruebas para definir las directivas OpenMP correctas, quedando definido un conjunto de datos que debe ser compartido por cada thread lanzado por OpenMP y ciertas variables que deben ser privadas de cada uno de ellos.

El bloque de código seleccionado para optimizar es el siguiente (ha sido abreviado):

```
1      do 3 ik=1,kult
2      fx(ik)=[calculo con valores de varias matrices]
3      fy(ik)=[calculo con valores de varias matrices]
4      fz(ik)=[calculo con valores de varias matrices]
5      fz(ik)=(-1.)*fz(ik)
6      dist2=[calculo con valores de varias matrices]
7      dista(ik)=dsqrt(dist2)
8      denom(ik)=dista(ik)**3
9    3 continue
```

Este bucle es el primer bucle interno de dos iteraciones mayores que incluyen más cálculos con otras estructuras, las cuales dependen de los resultados obtenidos en este primer bucle.

Se calculan tres arrays llamados fx, fy y fz, un valor dist2, y dos arrays más basados en el valor de dist2, llamados dista y denom.

Los cálculos de los tres primeros arrays y del valor dist2 dependen de varios otros arrays ya calculados previamente, y que la subrutina obtiene por el área de datos común con el resto de partes del programa Fortran, además de utilizar funciones propias del lenguaje.

En un primer análisis del bloque de código observamos una posible dependencia de datos en las líneas (5) y (8) del código anterior. En la primera, el cálculo de fz(ik) depende de sí mismo y en la segunda el valor de denom(ik) depende del valor de dista(ik) que depende de dist2. Si bien es posible que no surgieran problemas con estos valores, para evitar resultados inesperados, decidimos analizar y modificar si fuera necesario para evitar la dependencia, siempre que el cambio no fuera significativo, como reescribir la estructura de control completa o varias líneas con nuevas instrucciones.

La dependencia de datos en la línea (5) pudo solucionarse rápida y elegantemente. La línea multiplica el valor en fz(ik) por -1, por lo cual es posible agregar este cálculo al final de la línea (4) quedando de la siguiente manera:

```
fz(ik)=([cálculo con valores de varias matrices])*(-1.)
```

En el caso de la línea (8) el análisis es distinto: la dependencia se encuentra en el valor de dista(ik) el cual es calculado en el paso previo y depende del cálculo del valor dist2. Además, se trata de un cálculo simple con una función interna del lenguaje Fortran. Se podría utilizar un cálculo intermedio y luego asignar el resultado a dista(ik) y denom(ik), por ejemplo:

```
var_aux = dsqrt(dist2)
dista(ik) = var_aux
```

```
    denom(ik) = var_aux**3
```

Pero enfrentamos la indeterminación del valor inicial de var_aux, y cómo afecta a cada bloque paralelo cuando realicemos la optimización con OpenMP. Esto se puede resolver llevando un control de la variable en el bloque declarativo de OpenMP e inicializando la variable cada vez que es utilizada, lo que agrega carga de control al bloque de código (tanto en OpenMP como en el código Fortran normal). Si el cálculo a realizar con dist2 fuera de mayor complejidad podría justificarse la utilización de una variable auxiliar intermedia, pero como es un cálculo sencillo que utiliza una función interna de Fortran a la cual se le envía un solo valor, se puede resolver de la siguiente manera:

```
      dista(ik) = dsqrt(dist2)
      denom(ik) = (dsqrt(dist2))**3
```

Se puede entender mejor la dependencia de datos y la necesidad de controlar ciertas variables en los bloques paralelizados al observar un problema importante que surgió durante el trabajo de tesis, el cual incluso no estaba a simple vista.

Al realizar la optimización paralela los resultados del programa eran distintos a los de la ejecución normal. Los resultados deben ser iguales, dado que el programa es completamente determinístico; por lo cual se buscaron muchas formas diferentes con directivas de OpenMP de controlar la ejecución de los threads en este bloque seleccionado para optimización, para que los datos no se contaminaran, pero siempre arribando al mismo resultado erróneo.

El problema se encontró en otra porción de código que parecía bastante simple de paralelizar y sin necesidad de control alguno. Al iniciar, la subrutina estela utiliza dos estructuras DO anidadas que inicializan con valor 0 tres arrays (ciex, ciey y ciez), por lo cual con una estructura OMP PARALLEL DO de OpenMP debería bastar para paralelizar el cálculo y obtener una mejora, si bien poco considerable, en performance.

El problema surge porque la inicialización a 0 se realiza a través de una variable llamada "cero" definida en otra parte del código con el valor 0. Al lanzarse los threads de OpenMP dicha variable pasó a tener un valor indeterminado para cada thread, trayendo consigo datos espurios a los cálculos siguientes donde los arrays intervienen. Al comentar las directivas OpenMP que encerraban dichos bloques DO los resultados del programa volvieron a ser correctos.

Si bien el comportamiento por defecto de OpenMP debería ser compartir entre todos los threads las variables en memoria del programa principal, no ocurrió en este caso con la variable "cero", y no se encontró una explicación para este hecho. Investigar estas particularidades, cómo una implementación del estándar OpenMP difiere de otras, y qué problemas acarrean estas diferencias, puede ser motivo de una extensión futura de este trabajo de tesis.

Con las modificaciones indicadas el bucle ya estaba en condiciones de ser paralelizado con OpenMP.

Lo primero que realizamos, como se planificó en el capítulo 2, es indicar el comienzo de la región paralela y su final:

```
!$OMP PARALLEL
[bucle paralelizado]
!$OMP END PARALLEL
```

Ahora debíamos agregar las directivas para indicar que la región paralela debía ser una estructura DO, por lo que agregamos las directivas DO de OpenMP:

```
!$OMP PARALLEL
!$OMP DO

[bucle paralelizado]
```

```
!$OMP END DO
!$OMP END PARALLEL
```

Al realizar estos cambios en el código para el bloque indicado, conseguimos una gran mejora en el tiempo empleado, pero los resultados aún no eran correctos. Teniendo en cuenta esto debemos considerar qué variables son compartidas por los distintos threads del proceso y cuál es su alcance, para evitar discrepancias en los resultados.

En el bloque de código observamos que para realizar el cálculo de los arrays son necesarios varios otros arrays y variables, los que ya poseen valores previos. Además utiliza las variables de control ir y npa de los bloques DO exteriores donde está anidado nuestro bloque de código, utilizadas para recorrer los arrays indicados en la Fig. 3.8. Podemos ver esto en la tabla 3.4.

Tipo Variable	Variables
Arrays	pcx, pcy, pcz xe, ye, ze re, fi
Variable float	c0
Variables de control	ir, npa

Tabla 3.4: Variables necesarias para el código paralelizado.

El primer interrogante era saber si los datos se deben compartir entre todos los threads o deben ser privados. Si observamos todos los arrays y variables externos que se utilizan para el cálculo, los threads deben compartir su valor; si los definiéramos como PRIVATE su valor sería indefinido para cada thread, y si fuera como FIRSTPRIVATE aun cuando los valores fueran correctos, la cantidad de recursos necesarios para la ejecución se multiplicaría por la cantidad de threads que estuvieran en ejecución, ya que cada uno tendría una copia de cada variable.

Luego, los arrays modificados dentro del bloque son escritos por cada thread, pero cada thread accede a las posiciones definidas por la variable de control del bloque DO que estamos paralelizando, ik, la cual tendrá un valor para cada thread específico; por ejemplo si dividimos un DO de 100 iteraciones en 2 threads, la variable de control ik tendrá valor inicial de 0 para un thread y 50 para el otro.

Esto nos lleva a que los arrays modificados dentro del bloque también puedan ser compartidos por todos los threads, ya que sólo son accedidos indexados por la variable ik la cual, como indicamos, será distinta para cada thread, con lo cual cada uno accederá a modificar posiciones de los arrays distintas.

Por todo esto, concluimos que la gran mayoría de arrays y variables son compartidas por todos los threads, y la dependencia de datos entre éstos es inexistente (los arrays escritos no son leídos, los arrays y variables leídas no son modificadas), con lo cual definimos en la instrucción OpenMP de inicio del bloque paralelo como DEFAULT(SHARED) para todas las variables utilizadas dentro. Si bien éste es el comportamiento por defecto que asume el estándar OpenMP, lo dejamos declarado explícitamente, no sólo por legibilidad, sino para evitar que una eventual implementación de OpenMP de un compilador genere resultados incorrectos, por ejemplo como ocurre con la variable "cero" que vimos en el problema explicado previamente en esta misma sección. Definimos entonces el bloque de código paralelo de la siguiente manera:

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO

[bucle paralelizado]
```

```
!$OMP END DO
!$OMP END PARALLEL
```

Con esta definición tenemos que todas las variables (arrays y variables comunes) serán compartidas por todos los threads.

En un siguiente nivel de análisis, vemos que hay variables que necesitan definirse privadas de cada thread, principalmente la variable dist2 que es calculada dentro de cada thread en cada una de las iteraciones. Si fuera una variable compartida, todos los threads escribirían en ella en orden impredecible, llevando a resultados erróneos. Sólo para ejemplificar, supongamos que el thread 1 calcula la variable dist2 en una iteración, luego escribe el valor de dista(ik) con dist2; en ese momento el thread 4 calcula y escribe dist2. Cuando el thread 1 va a escribir el valor de denom(ik), dist2 ya tiene un valor completamente distinto al que había calculado el thread 1 previamente. Por esto declaramos a dist2 como PRIVATE.

Para evitar un problema similar al de la variable "cero" decidimos declarar las variables de control ir y npa, y la variable ncapa como FIRSTPRIVATE, de manera que sean privadas de cada thread y tengan desde el principio su valor original. El código resultante es el siguiente:

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO FIRSTPRIVATE(ir,npa,ncapa) PRIVATE(dist2)

[bucle paralelizado]

!$OMP END DO
!$OMP END PARALLEL
```

Luego de estos cambios, la ejecución del nuevo código dio resultados correctos comparados con la ejecución original. Nuevamente comparamos con *diff* o *vimdiff* los archivos de salida de la aplicación original con la nueva versión. De esta manera paralelizamos parte del bloque de código que más tiempo consumía de toda la aplicación.

En el capítulo 4 consideraremos la comparación de tiempos obtenidos para cada uno de los códigos y soluciones a pequeños contratiempos encontrados.

Capítulo 4

Experimentación y Análisis de Resultados

4.1. Introducción

En este capitulo se presentan los resultados obtenidos en cada etapa de la optimización, especialmente el speedup de la aplicación así como la utilización de recursos. Se mostrará también el estado del sistema durante las distintas ejecuciones de la aplicación.

Debido a que la optimización se realizó en varios pasos se mostrarán los resultados iniciales, parciales y finales del proceso. De esta manera es posible ver el impacto de cada parte de la optimización en el código legacy. Como indican [GG+03] la optimización previa del código serial es necesaria para evitar efectos indeseados en las mediciones, y puede representar un factor de aceleración de la aplicación de entre 2X y 5X, es decir, dos a cinco veces más rápido. La aplicación fue modificada lo menos posible en el proceso de optimización por lo cual no se alcanza toda la mejora posible en una recodificación, pero como se explicó anteriormente, se trató de hacer los cambios lo mas transparente posibles al usuario y creador de la aplicación. El código de la aplicación objeto de estudio de esta tesis es entregado junto con los resultados generados para dos conjuntos de datos de entrada, uno para un problema definido de una paleta dividida en 50 lineas de 50 paneles, de ahora en más un problema de tamaño 50x50, y otra para uno de 80 lineas de 80 paneles, problema de tamaño 80x80, siendo estos valores definidos en un archivo que sirve de entrada de datos para la aplicación.

Para este trabajo de tesis se elige trabajar principalmente con el conjunto de datos resultante del caso de cantidad de paneles 50x50, sin embargo se presentan observaciones obtenidas de una prueba en uno de los equipos para el caso de tamaño 80x80. El usuario y creador de la aplicación indicó que la ejecución de la aplicación con el problema de tamaño de 50x50 paneles demoraba en el orden de horas de ejecución. El problema de tamaño de 80x80 se dejaba ejecutando de un día para el otro. Como se explicó anteriormente, no hay datos de las ejecuciones del usuario, por lo cual se toman ejecuciones del código original en las arquitecturas de prueba para referencia.

4.2. Arquitecturas de prueba

Las pruebas se llevaron a cabo en dos equipos para obtener resultados que permitieran realizar una mejor evaluación del proceso de optimización. Las computadoras utilizadas fueron una PC y una Notebook, ambas multiprocesador y con arquitectura de 64 bits. A continuación la descripción de los equipos:

- Equipo 1 (PC Clon):
  - Procesador AMD Phenom II x4 955 x86_64

    - 4 núcleos reales.
    - Frecuencia máxima de 3.2 Ghz.
    - Release date: Abril del 2009.
  - Mother ASUS M4A785TD-V EVO
  - 4Gb RAM DDR3 1333Mhz.
  - HD SATA II 3Gbps.
  - USB 2.0 (480 Mbps)

- Equipo 2 (Notebook):
  - Procesador Intel Core i3-370M x86_64
    - 2 núcleos reales + 2 hilos de control por núcleo.
    - Frecuencia máxima de 2.4 Ghz
    - Release date: Junio del 2010.
  - Mother Dell 0PJTXT-A11.
  - 6Gb RAM DDR3 1333Mhz.
  - HD SATA II 3Gbps.
  - USB 2.0 (480 Mbps)

Nos referiremos en adelante al primer equipo como PC1 y al segundo equipo como PC2. Se utilizó una versión Live USB de Slackware Linux como sistema operativo para las pruebas. Como disco de almacenamiento sobre el que corría la aplicación se utilizó un Flash Drive USB, en el cual se crearon los archivos durante la ejecución.

Una nota sobre la arquitectura del procesador de PC2. En este caso el procesador tiene dos núcleos, pero al ofrecer dos hilos de control por núcleo, el sistema operativo los ve como si tuviera disponibles cuatro núcleos. El procesador luego distribuye los recursos disponibles sobre cada hilo de acuerdo a lo solicitado por el sistema operativo.

4.3. Mediciones

Para las mediciones de tiempo se utilizó el comando *time*[1] de manera de poder evaluar el tiempo real consumido por la aplicación en las diferentes etapas del trabajo de tesis: programa original, optimizado serialmente, optimizado paralelamente. Mostraremos los tiempos en los equipos seleccionados para las pruebas y las mejoras en desempeño que obtuvimos en el programa en cada iteración de la optimización. Para ambos equipos se realizaron mediciones del tamaño de problema de 50x50, y para el tamaño de problema de 80x80 se utilizó el equipo PC2. Realizamos pruebas con ambos tamaños de datos para poder determinar la escalabilidad de la solución aplicada, además de poder verificar como impacta en el equipo el cambio de tamaño del problema.

Para el tamaño de problema de 80x80 paneles, como se indicó en el capítulo 3, el archivo con los datos de entrada para la ejecución de la aplicación, *entvis2f.in*, posee una única modificación con respecto al mismo archivo para el tamaño de problema de 50x50, se define $nr = 80$ y $no = 80$.

Luego mediante el análisis de las diferencias entre los códigos de la versión de tamaño 50x50 contra la de 80x80, observamos que el código en los bloques "common" de Fortran indica lo siguiente:
Para el caso 50x50

[1] Para una referencia del comando en GNU/Linux ver su manual: "man 1 time".

```
parameter (maxir=51,maxio=51,...
```

Para el caso 80x80

```
parameter (maxir=81,maxio=81,...
```

Como indicamos en el capítulo 3, *maxir* y *maxio* son lo mismo que nr+1 o no+1, lo cual sería una manera más simple de definirlo. Debido a que la definición de estos valores está fija, literalmente, en cada bloque common de todo el código, es que para las optimizaciones, serial y paralela, de la aplicación con tamaño de problema 80x80, se debe cambiar en todo el código cada una de las definiciones de *maxir* y *maxio*.

Luego de adaptado esto se puede compilar cada versión de la aplicación para el tamaño de problema de 80x80 de la misma manera que la versión de tamaño 50x50.

También se incluyen muestras del estado de los archivos en disco luego de la ejecución del programa, el estado de la memoria y la CPU en plena ejecución del programa, para mostrar los resultados de las optimizaciones realizadas.

4.3.1. Estado inicial y primeras mediciones

Lo primero que hicimos fue compilar y ejecutar el programa original para calcular el tiempo inicial de referencia para el resto del trabajo, resguardando de una posible reescritura a los datos originales, que luego utilizaremos para poder verificar la correctitud de las distintas versiones del proceso de optimización. Acerca de esto, lo que se realizó fue una comparación de los resultados producidos en los archivos de salida de cada versión de la aplicación con los originales obtenidos por el usuario, verificando que sean exactamente los mismos.

En ambos equipos realizamos la compilación con el siguiente comando:

```
$ gfortran -o serial invisidos2fin.for
```

Esto crea un archivo ejecutable llamado “serial”. Para poder lanzar el ejecutable y poder verificar el tiempo lo realizamos con el comando:

```
$ time ./serial
```

4.3.1.1. Tiempos

En la Fig. 4.1 se puede observar el tiempo resultante calculado por el comando *time*, donde se obtiene un tiempo total de ejecución (linea “real”) para el tamaño 50x50 en PC1 de 21 min. 48 seg. y en PC2 de 22 min. 56 seg.

Para la versión de tamaño 80x80 podemos observar también en la Fig. 4.1 que la ejecución en el equipo PC2 indica un tiempo de ejecución de 225 min. 43 seg., es decir 3 hs. 45 min. 43 seg. El tamaño del problema se incrementa de 2500 paneles a 6400 paneles, un incremento de factor 2.56 veces, pero el tiempo se hace exponencial, en un factor de 9.78.

Podemos observar que con un cambio en la arquitectura del procesador (PC1 con 4 núcleos reales, PC2 con 2 núcleos y 2 hilos de control por núcleo) se incurre en una demora de 1m8s. Se tomó otra muestra con el equipo PC2 y se obtuvo un resultado similar, 23 min. 1 seg. por lo que podríamos indicar que la diferencia persiste y se mantiene dentro de un margen de tiempo. Esta diferencia observada se debe posiblemente a la mayor velocidad del procesador en PC1 o al mayor gasto (*overhead*) de tiempo en la administración y comunicación de los hilos por la arquitectura SMT de PC2. También sería de interés investigar el uso de la jerarquía de memoria, especialmente de las caches, en ambos procesadores.

```
real    21m48.109s
user    19m3.067s
sys     0m29.685s
live@PC1 $
```

(a) Equipo PC1

```
real    22m56.392s
user    20m7.858s
sys     0m32.917s
live@PC2 $
```

(b) Equipo PC2

```
real    225m43.721s
user    174m29.803s
sys     3m11.953s
live@PC2 $
```

(c) Equipo PC2 - Tamaño 80x80

Figura 4.1: Tiempos de la versión serial original.

Como el programa es serial, siempre utilizó en su ejecución el mismo núcleo o hilo de ejecución. En la Fig. 4.2 podemos observar una captura del comando *top* en PC1, donde se puede ver la aplicación original en ejecución sobre la CPU2.

```
op - 18:03:42 up 12 min,  4 users,  load average: 1.08, 0.61, 0.54
asks: 230 total,   3 running, 227 sleeping,   0 stopped,   0 zombie
Cpu0  :  4.7 us,  3.0 sy,  0.0 ni,  0.0 id, 91.9 wa,  0.0 hi,  0.3 si,  0.0 st
Cpu1  :  3.7 us,  1.7 sy,  0.0 ni, 61.7 id, 32.9 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu2  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu3  :  3.7 us,  1.7 sy,  0.0 ni, 93.9 id,  0.0 wa,  0.0 hi,  0.7 si,  0.0 st
iB Mem :  3784744 total,   212916 free,   812508 used,  2759320 buff/cache
iB Swap:        0 total,        0 free,        0 used.  2151860 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2224 live      20   0  222344 160692   2840 R 100.0  4.2   0:35.99 serial
 2178 live      20   0  799944  89120  65360 S   5.0  2.4   0:03.21 spectacle
 1727 live      20   0 3167828  99684  68324 S   4.3  2.6   0:13.76 kwin_x11
 1486 root      20   0  221396  55296  30544 S   4.0  1.5   0:18.63 Xorg
```

Figura 4.2: Comando *top*: Aplicación original en subrutina *estela*.

4.3.1.2. Archivos en disco

La ejecución genera para ambos tamaños de problema (50x50 y 80x80 paneles) todos los archivos utilizados para cálculos intermedios y resultados finales así como los temporales con los que el programa trabaja.

La ejecución serial del programa original generó la misma cantidad de archivos, 58 archivos (Fig. 4.3) entre los ".txt", ".plt", ".out" y los ".tmp", esto es así por el determinismo del programa. No contamos el archivo ejecutable ni el de datos de ingreso "entvis2f.in".

El tamaño en disco ocupado si difiere entre los tamaños de problema. Como podemos observar en la Fig. 4.3, para el tamaño de 50x50, tanto en PC1 como en PC2 el tamaño de los archivos fue de 684 Mb, donde el mayor tamaño era ocupado por los ocho archivos ".tmp", de los cuales siete ocupan 96 Mb cada uno para un total de 672 Mb.

Para el tamaño de 80x80 el espacio en disco utilizado fue de 4415Mb o 4.3GB, siendo los archivos ".tmp" los que ocupaban 4375MB, siete de los ocho archivos pesando 625MB cada uno.

4.3.1.3. Memoria RAM

La cantidad de memoria consumida por la aplicación para el tamaño de problema de 50x50 al iniciar en cada equipo es de 217 MB en PC1 y lo mismo en PC2. Cuando durante la ejecución la aplicación ingresa en la subrutina *solgauss* la memoria se incrementa a 255 MB. Y al salir de esta subrutina la memoria baja a 217 MB nuevamente. La salida por pantalla de la aplicación nos permite saber en que subrutina se encuentra, por ello en tiempo de ejecución podemos determinar

```
[h4ndr3s@darkstar /tesis/clon/Serial]$  ls
alfa.txt   ciz2.tmp    cpei.txt     gdifo.out   panel.plt     vel02ext.txt  veltotal.txt
arco.txt   coefg.tmp   cr.txt       gdifr.out   panel.txt     vel02int.txt  velxyz.txt
cindg.tmp  coefp.txt   entvis2f.in  gmon.out    pres.plt      vel02pvn.txt  vix.txt
circo.txt  coord.txt   estel1.txt   go.out      salida2.out   velcapae.txt  viy.txt
cix1.tmp   co.txt      estel2.txt   gr.out      serial        velcapai.txt  viz.txt
cix2.tmp   cp075c.txt  estel3.txt   integ1.txt  subr.out      velindad.plt  vn.txt
ciy1.tmp   cp1.txt     fuerza.plt   integ2.txt  vector.plt    velindad.txt
ciy2.tmp   cp2.txt     fzas.txt     palas.plt   vel01ext.txt  velpotad.txt
ciz1.tmp   cpei.plt    gama.txt     palest.plt  vel01int.txt  velresad.plt
[h4ndr3s@darkstar /tesis/clon/Serial]$  du -sh
684M    .
```

(a) PC1

```
[h4ndr3s@darkstar /tesis/dell/Serial]$  ls
alfa.txt   ciz2.tmp    cpei.txt     gdifo.out   panel.plt     vel02ext.txt  veltotal.txt
arco.txt   coefg.tmp   cr.txt       gdifr.out   panel.txt     vel02int.txt  velxyz.txt
cindg.tmp  coefp.txt   entvis2f.in  gmon.out    pres.plt      vel02pvn.txt  vix.txt
circo.txt  coord.txt   estel1.txt   go.out      salida2.out   velcapae.txt  viy.txt
cix1.tmp   co.txt      estel2.txt   gr.out      serial        velcapai.txt  viz.txt
cix2.tmp   cp075c.txt  estel3.txt   integ1.txt  subr.out      velindad.plt  vn.txt
ciy1.tmp   cp1.txt     fuerza.plt   integ2.txt  vector.plt    velindad.txt
ciy2.tmp   cp2.txt     fzas.txt     palas.plt   vel01ext.txt  velpotad.txt
ciz1.tmp   cpei.plt    gama.txt     palest.plt  vel01int.txt  velresad.plt
[h4ndr3s@darkstar /tesis/dell/Serial]$  du -sh
684M    .
```

(b) PC2

Figura 4.3: Original: Lista de archivos y tamaño del directorio por equipo. Tamaño 50x50.

el estado de la memoria para el proceso. Justamente la rutina *solgauss* representa el máximo en la cantidad de memoria consumida por la aplicación.

Para el tamaño 80x80, los datos observados muestran que en memoria RAM la aplicación llega a ocupar 1293 Mb o 1.26 GB fuera de la subrutina *solgauss* y 1581 Mb dentro de la subrutina.

Estos datos se obtienen del comando *pmap*[2] aplicado sobre el proceso en ejecución, por ejemplo si la aplicación tiene PID 2228:

```
$ pmap -x 2228
```

Pmap reporta información del mapa de memoria de un proceso, dando en su última línea un total en Kbytes de la memoria utilizada, importándonos la primer columna donde indica el total de memoria utilizada por el proceso. Por ejemplo en la Fig. 4.4 vemos el resultado para cada equipo mientras se ejecutaba la aplicación original para el tamaño de problema de 50x50. El comando "top" también permite observar el mismo valor que indica "pmap" en su columna VIRT.

```
[datos de la aplicación]
------------- ------- ------- -------
total kB       222212  157080  154368
```

(a) Equipo PC1

```
[datos de la aplicación]
------------- ------- ------- -------
total kB       222472  209160  206240
```

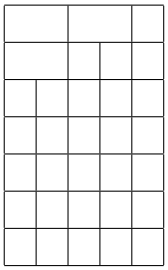
(b) Equipo PC2

Figura 4.4: Información del comando *pmap* en cada equipo.

En la tabla 4.2 se muestran los datos recopilados hasta el momento en el trabajo de tesis, todos de la aplicación serial original. Las dos subsecciones siguientes mostrarán como evolucionó con la optimización, tomando como base para comparación los tiempos y tamaños obtenidos en esta primer etapa.

[2]Para una referencia del comando en GNU/Linux ver su manual: "man pmap".

Tabla 4.1: My caption



Tamaños de problema		tamaño 50x50		tamaño 80x80
Equipos		PC1	PC2	PC2
Archivos generados		58	58	
Esp. en disco utilizado			684Mb	684Mb
Memoria	Ejecución en *solgauss*	255Mb	255Mb	
	Resto de la ejecución	217Mb	217Mb	
CPUs utilizadas		1	1	
Tiempo total de ejecución		21m48s	22m56s	

Tabla 4.2: Datos de ejecución de la aplicación original en ambos equipos.

```
real    16m2.124s
user    16m0.894s
sys     0m0.259s
live@PC1 $
```

(a) Equipo PC1

```
real    17m4.161s
user    17m2.631s
sys     0m0.428s
live@PC2 $
```

(b) Equipo PC2

Figura 4.5: Tiempo de la versión optimizada serialmente.

4.3.2. Optimización serial y mediciones intermedias

Luego de realizar la optimización serial se tomaron nuevamente mediciones. La compilación se realizó con el mismo comando ya que en esta etapa aún no tenemos la adición de ninguna optimización paralela.

```
$ gfortran -o optserial invisidos2fin_optSerial.for
```

Y nuevamente para medir el tiempo del programa ejecutamos la aplicación con la instrucción time.

```
$ time ./optserial
```

El tiempo obtenido en PC1 fue de 16m2.124s, lo que representa una ganancia de 5m36s aproximadamente sobre la versión serial original de la aplicación en el mismo equipo, teniendo entonces un factor de 1.35 de mejora en el tiempo. En la computadora PC2 los tiempos obtenidos fueron de 17min 4.161seg. Se observa una mejora sobre la versión serial original de 5m 52s aproximadamente, o un factor de 1.34 de mejora en el tiempo (Fig. 4.5).

Se puede ver que el factor de mejora alcanzado entre el original serial y el optimizado es muy similar entre ambos equipos, con una diferencia de solo 0.01, y que es levemente mejor en PC1. Estrictamente hablando de los tiempos de ejecución del código optimizado serialmente, entre los equipos la diferencia observada es de 1m2s, nuevamente a favor de PC1.

Al observar el directorio donde se ejecuta la aplicación, observamos que luego de la optimización serial han desaparecido los archivos ".tmp" (Fig. 4.6), ya que ahora lleva los cálculos intermedios en la memoria para poder mejorar los tiempos de acceso. El resto de archivos (50 en total) siguen creándose, pero al demorar la escritura de los archivos utilizados para ir mostrando y almacenando la salida por pantalla, tanto como los que son leidos y escritos y obtienen resultados finales, se logra evitar el acceso constante al disco a través de la ejecución de la aplicación, para tener sólo que hacerlo una vez por archivo al finalizar la ejecución del programa o una subrutina en particular. El tamaño ocupado por los archivos del programa ahora fue de 17 Mb tanto en PC1 como en PC2 (Fig. 4.6), observando nuevamente el impacto de no generar los archivos ".tmp".

```
[h4ndr3s@darkstar /tesis/clon/Ifiles]$  ls
alfa.txt    cp2.txt      estel3.txt  gr.out      pres.plt      vel02pvn.txt  velxyz.txt
arco.txt    cpei.plt     fuerza.plt  ifiles      salida2.out   velcapae.txt  vix.txt
circo.txt   cpei.txt     fzas.txt    integ1.txt  subr.out      velcapai.txt  viy.txt
coefp.txt   cr.txt       gama.txt    integ2.txt  vector.plt    velindad.plt  viz.txt
coord.txt   datos.txt    gdifo.out   palas.plt   vel01ext.txt  velindad.txt  vn.txt
co.txt      entvis2f.in  gdifr.out   palest.plt  vel01int.txt  velpotad.txt
cp075c.txt  estel1.txt   gmon.out    panel.plt   vel02ext.txt  velresad.plt
cp1.txt     estel2.txt   go.out      panel.txt   vel02int.txt  veltotal.txt
[h4ndr3s@darkstar /tesis/clon/Ifiles]$  du -sh
17M     .
```

(a) PC1

```
[h4ndr3s@darkstar /tesis/dell/Ifiles]$  ls
alfa.txt    cp2.txt      estel3.txt  gr.out      pres.plt      vel02pvn.txt  velxyz.txt
arco.txt    cpei.plt     fuerza.plt  ifiles      salida2.out   velcapae.txt  vix.txt
circo.txt   cpei.txt     fzas.txt    integ1.txt  subr.out      velcapai.txt  viy.txt
coefp.txt   cr.txt       gama.txt    integ2.txt  vector.plt    velindad.plt  viz.txt
coord.txt   datos.txt    gdifo.out   palas.plt   vel01ext.txt  velindad.txt  vn.txt
co.txt      entvis2f.in  gdifr.out   palest.plt  vel01int.txt  velpotad.txt
cp075c.txt  estel1.txt   gmon.out    panel.plt   vel02ext.txt  velresad.plt
cp1.txt     estel2.txt   go.out      panel.txt   vel02int.txt  veltotal.txt
[h4ndr3s@darkstar /tesis/dell/Ifiles]$  du -sh
17M     .
```

(b) PC2

Figura 4.6: Opt. Serial: Lista de archivos y tamaño del directorio por equipo.

Observando la memoria en esta versión del programa obtenemos que consume 552 Mb mientras está en *solgauss* y 504MB el resto del tiempo, tanto en PC1 como PC2 (Fig. 4.7). Esto significa un incremento en la cantidad de memoria utilizada, en esta versión optimizada serialmente con respecto a la versión serial original, de 297MB cuando el programa está en la subrutina *solgauss* y de 287MB antes o después de dicha subrutina. Este incremento se debe a los archivos ".tmp" que ya no utiliza mas en disco y debe llevar en memoria como internal files.

```
[datos de la aplicación]
------------- ------- ------- -------
total kB       516392  504060  501276
```

(a) Equipo PC1

```
[datos de la aplicación]
------------- ------- ------- -------
total kB       516524  504308  501332
```

(b) Equipo PC2

Figura 4.7: Comando *pmap* con la aplicación optimizada serialmente (fuera de *solgauss*).

Nuevamente en el caso de la CPU podemos observar que un solo procesador es el encargado de realizar la tarea ya que aún no se optimiza paralelamente. En la Fig. 4.8 podemos observar como ejemplo, la ejecución de la aplicación optimizada serialmente en PC1, en el momento que está dentro de la subrutina *estela*.

La Tabla 4.3 resume la información obtenida de la optimización serial de la aplicación. En la siguiente sección se ven los resultados de la optimización paralela mediante OpenMP.

	PC1	PC2
Archivos generados	50	50
Esp. en disco utilizado	17Mb	17Mb
Memoria	Ejecución en *solgauss*: 552Mb	552Mb
	Resto de la ejecución: 504Mb	504Mb
CPUs utilizadas	1	1
Tiempo total de ejecución	16m02s	17m04s

Tabla 4.3: Datos de ejecución de la aplicación optimizada serialmente.

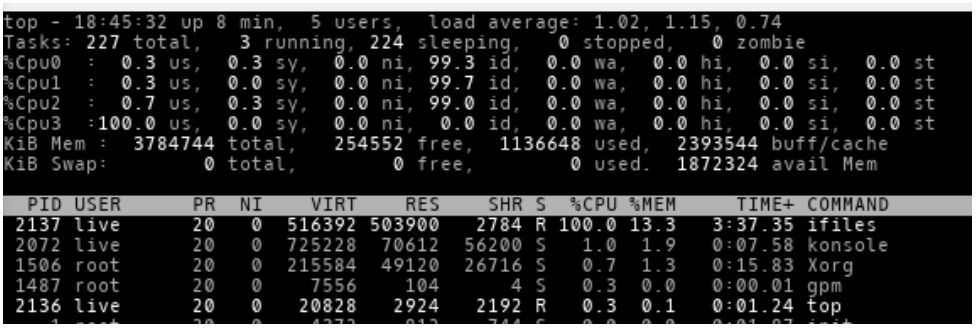


Figura 4.8: Comando *top*: Aplicación opt. serialmente en subrutina *estela*.

4.3.3. Optimización Paralela y mediciones finales

Finalmente realizamos las pruebas con la versión optimizada paralelamente del programa. Para esta prueba cambiamos la forma de compilar el programa ya que se debe indicar que aprovechará las directivas de OpenMP, esto lo realizamos pasando el parámetro "-fopenmp" al comando de compilación, de la siguiente manera:

```
$ gfortran -fopenmp -o paralelo invisidos2fin_optOMP.for
```

Al terminar tenemos un ejecutable listo para aprovechar la paralelización que brinda OpenMP. Nuevamente se ejecutó la aplicación con el comando time, de manera de obtener el tiempo de ejecución. La ejecución se hizo sin limitar la cantidad de threads creados en OpenMP, es decir que la aplicación se ejecutó aprovechando todos los threads disponibles por defecto, es decir uno por cada núcleo disponible (cuatro threads en cada equipo).

```
$ time ./paralelo
```

Un contratiempo que ocurrió en este paso fue que al ingresar en la parte paralelizada, la aplicación incurrió en un error de "segmentation fault". El problema ocurre por el tamaño máximo definido en el kernel Linux de la pila para un proceso, el cual por defecto es de 8192Kb. La solución es previo a la ejecución de la aplicación, definir el tamaño máximo de la pila en "unlimited" con el siguiente comando:

```
$ ulimit -s unlimited
```

Luego de establecido el parámetro, la ejecución de la aplicación es correcta.

Los resultados de "time" para PC1 indicaron un tiempo de ejecución de 6m5.294s (Fig. 4.9). Al comparar con los 21m48.109s que tomó en su versión original podemos observar 15m42s de mejora aproximada, obteniendo un factor de 3.58 de mejora en el desempeño, lo cual es muy superior a la ganancia inicial con la optimización serial. En PC2 obtuvimos 8m50.822s de tiempo de ejecución

(Fig. 4.9), mientras el programa original tomó 22m56.392s, es decir aproximadamente 14m6s más rápida la versión paralela, obteniendo un factor de 2.59 de mejora en el desempeño. La diferencia de tiempo de ejecución entre la aplicación optimizada paralelamente en PC1 y PC2 es de 2m45s, observándose esta vez una diferencia de tiempo considerable. Se podría investigar la incidencia de los 4 núcleos reales del procesador AMD en PC1 contra los 2 núcleos reales y 2 hilos de control por núcleo en el procesador Intel de PC2. Ambos procesadores brindan a OpenMP cuatro hilos, pero los recursos son asignados de manera diferente.

```
real    6m5.294s
user    17m38.896s
sys     0m0.872s
live@PC1 $
```

(a) Equipo PC1

```
real    8m50.822s
user    28m21.227s
sys     0m4.812s
live@PC2 $
```

(b) Equipo PC2

Figura 4.9: Tiempo de la versión optimizada paralelamente con OpenMP.

En el consumo de CPU esta vez podemos observar diferencia entre los programas seriales y uno paralelizado. Se han activado todos los núcleos disponibles en el equipo al momento de entrar en la zona de la subrutina *estela* (Fig. 4.10), ya sean núcleos reales (PC1) o virtuales (PC2). Como ya indicamos, la activación de los núcleos no fue administrada de manera directa con directivas OpenMP por lo cual todos los núcleos disponibles fueron utilizados, pero como se indicaba en el capitulo 2, hay más directivas que pueden ser estudiadas de OpenMP que podrían ser utilizadas para disminuir o incrementar la cantidad de hilos generados en una región paralela y estudiar el impacto y la utilización de los recursos en el multiprocesador.

```
top - 19:11:24 up 6 min,  4 users,  load average: 2.04, 1.64, 0.87
Tasks: 227 total,   2 running, 225 sleeping,   0 stopped,   0 zombie
%Cpu0  : 99.0 us,  0.3 sy,  0.0 ni,  0.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  : 99.3 us,  0.3 sy,  0.0 ni,  0.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  : 99.0 us,  0.7 sy,  0.0 ni,  0.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  3784744 total,   233924 free,  1097132 used,  2453688 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  1912120 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2158 live      20   0  480008 455548   3184 R 392.7 12.0   4:20.25 omp
 1512 root      20   0  216048  49928  27236 S   1.3  1.3   0:13.70 Xorg
 1757 live      20   0 3268256 112144  74148 S   1.3  3.0   0:09.87 kwin_x11
 2153 live      20   0  809044 104500  65348 S   1.0  2.8   0:02.99 spectacle
 1709 live      20   0 1528888  73308  58524 S   0.3  1.9   0:03.21 kded5
 1771 live      20   0 3816896 185208  93736 S   0.3  4.9   0:15.88 plasmashell
```

Figura 4.10: Comando *top*: Aplicación optimizada con OpenMP en subrutina *estela*.

El directorio de ejecución del programa queda igual que en la versión optimizada serialmente (Ver Fig. 4.6) ya que en esta nueva versión se han agregado las directivas OpenMP utilizadas y no se ha tocado el código serial ni el tratamiento de los archivos. Lo mismo ocurre con el tamaño ocupado por los archivos en disco (17MB).

```
[datos de la aplicación]
------------- ------- ------- -------
total kB       480008   455704   452516
```

(a) Equipo PC1

```
[datos de la aplicación]
------------- ------- ------- -------
total kB       480144   455828   452576
```

(b) Equipo PC2

Figura 4.11: Comando *pmap* sobre aplicación optimizada con OpenMP (fuera de *solgauss*).

Observando la memoria el programa consume 516MB de RAM durante la subrutina *solgauss* y 468MB en el resto de su ejecución en ambos equipos (Fig. 4.11). Con respecto al original esto indica un incremento de 261MB de memoria mientras está en *solgauss* y 251MB en el resto de la

ejecución. Al comparar con la aplicación optimizada serialmente se observó que el consumo de memoria es menor en la versión con OpenMP. Ocupa 36MB menos durante la ejecución, tanto si se ejecuta en *solgauss* como en el resto del tiempo. Podría investigarse esta diferencias en la memoria en la optimización que realiza el compilador en el código para utilizar las directivas de OpenMP.

Finalmente podemos ver en la Fig. 4.12 los datos resumidos de las tres versiones de la aplicación.

Tamaño de problema 50x50		Serial		Opt. Serial		Opt. Paralela	
		PC1	PC2	PC1	PC2	PC1	PC2
Archivos generados		58		50		50	
Tamaño en disco		684Mb		17Mb		17Mb	
Memoria utilizada	Fuera de solgauss	217Mb		504Mb		468Mb	
	En solgauss:	255Mb		552Mb		516Mb	
Procesadores utilizados		1		1		4	
Tiempo de ejecución		21m48seg	22m56seg	16m2seg	17m4seg	6m5seg	8m50seg
Factor de mejora		-	-	1.35	1.34	3.58	2.59

Figura 4.12: Resumen datos totales de las aplicaciones.

4.4. Conclusión

En este capítulo hemos presentado distintas pruebas de ejecución de la aplicación bajo estudio durante el proceso de su optimización, distinguiendo tres etapas: aplicación original, aplicación optimizada serialmente y aplicación optimizada paralelamente. Además se utilizaron dos plataformas de hardware distintas para dar mayor amplitud a la prueba y poder observar el comportamiento de la aplicación con distinto hardware. También se realizó una prueba con un tamaño de problema mayor para ver el impacto de la paralelización y se pudo ver el impacto en la memoria RAM, además de encontrar un perfilado distinto que en la versión de tamaño de problema menor.
Se han podido tomar mediciones de tiempo y de recursos para presentar conclusiones en el siguiente capítulo del trabajo realizado.

Para finalizar se puede afirmar que la aplicación desde su versión original hasta la versión optimizada y paralelizada resultante de este trabajo de tesis, ha obtenido una mejora en su velocidad de ejecución en un factor de 1.73 en el tamaño de problema mayor (80x80 paneles), y entre 2.59 y 3.58 en su versión de menor tamaño (50x50 paneles).

Capítulo 5

Conclusiones y Trabajos Futuros

En este capítulo se presentan las principales conclusiones de esta tesis. En la última sección se presentan las potenciales líneas futuras de acción que puedan complementar el trabajo desarrollado.

5.1. Conclusiones

La paralelización de una aplicación legacy es una tarea compleja con muchos aspectos. Se realiza en varias etapas incrementales que van impactando en mayor o menor medida en los resultados. La finalidad de la paralelización puede ser buscar mejorar la performance, la utilización de recursos, la calidad de los resultados, etc. El presente trabajo se enfocó en la mejora de utilización de recursos durante la etapa de optimización serial, y de la performance en la etapa de optimización paralela. Sin embargo, en ambas etapas se obtuvieron mejoras parciales en ambos aspectos.

Para poder realizar el trabajo de tesis fue necesario estudiar y aprender el lenguaje Fortran; luego, estudiar el código de la aplicación objeto de estudio para comprender su forma de trabajar, y finalmente poder realizar los cambios necesarios sin modificar substancialmente el código. Recorriendo este camino se adquirieron conocimientos sobre el estándar OpenMP y sobre la optimización de aplicaciones para Computación de Altas Prestaciones.

Para poder enfrentar el trabajo de optimización adecuadamente fue necesario un análisis de la aplicación bajo estudio. Tomamos el programa original, ejecutándolo con un tamaño del problema de 50x50, para obtener mediciones que dieran la base de comparación para las etapas de optimización. Luego se aplicó la optimización tal como se describió en el capítulo 3: optimización del código serial en primer lugar, y paralelización de una porción del código posteriormente. Esta paralelización se realizó aplicando las directivas de OpenMP a la subrutina estela.

Luego de la primera etapa, de optimización serial, se logró una mejora en el tiempo de ejecución, aunque todavía reducida (factor de mejora o speedup de 1.34). Al realizar la optimización paralela obtuvimos un mayor speedup (con valores de 3.58 en un equipo de pruebas y 2.59 en otro). Estos factores de mejora estaban más de acuerdo con la intuición, ya que la optimización afectaba a una subrutina que consumía entre un 74 % y un 79 % del tiempo de ejecución original.

Al ocuparnos del uso de discos y memoria, trasladamos los archivos desde el filesystem, residente en disco, a la memoria RAM para un acceso mas rápido. Sin embargo esta modificación no tuvo un impacto tan notable como se esperaba. Se duplicó aproximadamente la utilización de memoria RAM, sin que esto representara un inconveniente dados los equipos utilizados. En equipos con 1 GB de RAM (en la actualidad es habitual contar con 2 GB o más) el programa seguiría teniendo memoria suficiente para ejecutarse usando el tamaño de problema de 50x50.

Por último, sobre uno de los equipos se llevaron a cabo pruebas con un tamaño de problema mayor (80x80), sobre la versión original, optimizada serialmente y optimizada paralelamente. Primero se realizó un perfilado, donde se pudo observar que los porcentajes de tiempo de las

subrutinas estela y solgauss casi se habían equiparado (46 % y 43 % respectivamente). En las pruebas esto mostró que la mejora de tiempo al paralelizar no fue tan grande como con el tamaño de problema menor, lo cual lleva a la necesidad de paralelizar solgauss para poder obtener mayor mejora en tamaños más grandes de problema para la aplicación. También podemos concluir que claramente el incremento de tamaño del problema y la optimización impactan en la memoria del sistema y debe tenerse en cuenta la cantidad de RAM disponible si el usuario quisiera incrementar aún más el tamaño del problema.

5.2. Trabajos Futuros

En función del trabajo de tesis se identificaron algunos aspectos que permitirían extender el trabajo realizado. Estos aspectos se detallan a continuación:

- Analizar si al aumentar aún más el tamaño del problema, con perfilado de la aplicación original, siguen invirtiéndose los tiempos de las subrutinas, y ver si es necesario optimizar de otra manera. También utilizar herramientas de perfilado de aplicaciones paralelas con OpenMP, como "ompp".

- Otra línea de trabajo podría ser proponer una recodificación de la aplicación para aprovechar mejoras en el lenguaje Fortran y otras bibliotecas existentes para la realización de los cálculos.

- Se podría paralelizar la subrutina solgauss y analizar si es necesario recodificar la subrutina o si se puede optimizar en su estado original. Además analizar si la ganancia en performance en el tamaño de problema menor es o no despreciable y en problemas de mayor tamaño, qué speedup puede obtenerse.

- Realizar la optimización utilizando conjuntamente OpenMP y la API MPI, y analizar la posibilidad de utilizar un cluster para la ejecución de la aplicación aprovechando la capacidad de cómputo de varios nodos.

- Investigar cómo una implementación del estándar OpenMP difiere de otras y qué problemas se presentan, teniendo en cuenta el problema visto en el cap 3 con la inicialización en cero de arrays utilizando OpenMP.

Bibliografía

[BS95] M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems.* Morgan Kaufmann Publishers, 1995.

[CJvdP08] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP.* The MIT Press, 2008.

[EEL+97] Susan Eggers, Joel Emer, Henry Levy, Jack Lo, Rebecca Stamm, and Dean Tullsen. Simultaneous multithreading: A platform for next-generation processors. Technical report, IEEE Micro, 1997.

[For12] Message Passing Interface Forum. Mpi: A message-passing interface standard. `https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, 2012.

[GG+03] Ananth Grama, Anshul Gupta, et al. *Introduction to Parallel Computing, Second Edition.* Addison-Wesley, 2003.

[Gol98] N. E. Gold. The meaning of legacy systems. Technical report, Univ. of Durham, Dept. of Computer Science, 1998.

[Gre13] Brendan Gregg. *Systems Performance: Enterprise and the Cloud.* Prentice Hall, 2013.

[GRS15] C. García-Recio and L. L. Salcedo. Notas de fortran 77. Technical report, Univ. de Granada, Departamento de Física Atómica, Molecular y Nuclear, 2015.

[GS01] Rajat P. Garg and Ilya Sharapov. *Techniques for Optimizing Applications: High Performance Computing.* Prentice-Hall PTR, 2001.

[Her02] Miguel Hermmans. Parallel programming in fortran 95 using openmp. Technical report, Universidad Politeécnica de Madrid, Departamento de Motopropulsión y Termofluidodinámica, School of Aeronautical Engineering, Abril 2002.

[MT12] Mariano Méndez and Fernando G. Tinetti. A gpu approach to fortran legacy systems. Technical report, Universidad Nacional de La Plata, Facultad de Informática, Agosto 2012.

[Ope13] Architecture Review Board OpenMP. Openmp application program interface. `http://openmp.org/wp/openmp-specifications/`, 2013.

[Pra07] Ricardo A. Prado. *Desarrollo de un código de interacción viscosa-invíscida orientado a turbomaquinaria.* PhD thesis, Universidad de Buenos Aires, Mayo 2007.

[Que00] Jesús García Quesada. Manual de fortran 77. Technical report, Univ. de Las Palmas de Gran Canaria, Edificio de Informática y Matemáticas, 2000.

[WC00] Kevin R. Wadleigh and Isom L. Crawford. *Software Optimization for High Performance Computing.* Prentice-Hall PTR, 2000.

Anexo A

Referencia del Lenguaje Fortran

En este anexo se presenta una referencia resumida del lenguaje Fortran en su estándar 77, el cual es utilizado en la aplicación objeto de estudio de este trabajo de tesis. Particularmente se detallan las sentencias más utilizadas en la aplicación en cuestión. Para mayor referencia se puede ampliar consultando [Que00], [GRS15] o la definición del estándar, que puede consultarse en `https://www.fortran.com/F77_std/f77_std.html`.

A.1. Estructuras de Especificación

A.1.1. COMMON

Define una o más áreas contiguas de memoria, o bloques. También define el orden en el que las variables, arrays y records aparecen en un bloque común. Dentro de un programa, puede haber un bloque COMMON sin nombre, pero si existen más, se les ha de asignar un nombre. Esta instrucción, seguida por una serie de instrucciones de especificación, asigna valores iniciales a entidades de bloques comunes con nombre y a la vez, establece y define estos bloques.

La sintaxis es:

```
COMMON [/nomb/] list [[,]/[nomb1]/list1] . . .
```

donde:
nomb es un nombre simbólico.
list es una lista de nombres de variables, nombres de arrays y declaradores de array.
Cuando se declaran bloques comunes con el mismo nombre en diferentes unidades de programa, estos comparten la misma área de memoria cuando se combinan en un programa ejecutable.

A.2. Estructuras de Control

A.2.1. DO indexado

Controla el procesamiento iterativo, o sea, las instrucciones de su rango se ejecutan un número especificado de veces. Tiene la forma:

```
DO [s[,]] v = e1 , e2 [,e3 ]
```

donde:
s es la etiqueta de una instrucción ejecutable, que ha de estar en la misma unidad de programa.

v es una variable entera o real, que controla el bucle (índice).
e1 ,e2 ,e3 son expresiones aritméticas.
La variable **v** es la variable de control, **e1** es el valor inicial que toma **v**, **e2** es el valor final y **e3** es el incremento o paso, que no puede ser cero. Si se omite **e3**, su valor por defecto es 1. El rango de una DO incluye todas las instrucciones que siguen a la misma DO hasta la instrucción terminal, la última del rango.
La instrucción terminal no puede ser:

- una GOTO incondicional o asignada.
- un IF aritmético.
- un bloque IF.
- ELSE , ELSE IF , END IF , RETURN , STOP, END , otra DO.

El número de ejecuciones del rango de una DO, llamado contador de iteraciones viene dado por: MAX(INT($(e2 - e1 + e3)/e3$), 0)
donde INT(x) representa la función parte entera de x. Y las etapas seguidas en la ejecución son las siguientes:

1. Se evalúa el contador = INT($(e2 - e1 + e3)/e3$)
2. Se hace v = e1
3. Si contador es mayor que cero, entonces:
   a) Ejecutar las instrucciones del rango del bucle
   b) Asignar v = v + e3
   c) Decrementar el contador (contador=contador-1). Si contador es mayor que cero, repetir el bucle.

A.2.2. GOTO incondicional

Las instrucciones GOTO transfieren el control dentro de una unidad de programa. Dependiendo del valor de una expresión, el control se transfiere, bien a la misma instrucción siempre, o bien a una de un determinado conjunto de instrucciones. En el caso del GOTO incondicional, transfiere el control a la misma instrucción cada vez que se ejecuta. Tiene la forma:

```
GOTO s
```

donde **s** es la etiqueta de una instrucción ejecutable que está en la misma unidad de programa de la instrucción GOTO.

A.2.3. Sentencias IF

Transfieren el control condicionalmente, o bien ejecutan condicionalmente una instrucción o bloque de instrucciones. Nos interesan dos tipos:

- IF aritmético
- IF lógico

La decisión de transferir el control o ejecutar la sentencia o bloque de sentencias está basada en la evaluación de una expresión en la instrucción IF.

A.2.3.1. IF aritmético

Transfiere el control condicionalmente a una de tres sentencias, según sea el valor de la expresión que aparece en la instrucción IF . Tiene la forma:

```
IF (e) s1 ,s2 ,s3
```

donde:
e es una expresión aritmética (de cualquier tipo salvo compleja, lógica o caracter).
s1, s2, s3 son etiquetas de instrucciones ejecutables de la misma unidad de programa.

- las tres etiquetas **s1, s2, s3** son obligatorias, aunque no tienen que ser distintas.
- se evalúa la expresión **e** y se transfiere el control a una de las tres etiquetas como se ve en la tabla A.1.

Si el valor de e es	El control pasa a
menor que cero	etiqueta s1
igual a cero	etiqueta s2
mayor a cero	etiqueta s3

Tabla A.1: Evaluación de IF aritmético.

A.2.3.2. IF lógico

Ejecuta condicionalmente una única sentencia dependiendo del valor de la expresión lógica que aparece en la instrucción IF. Tiene la forma:

```
IF (e) sentencia
```

donde:
e es una expresión lógica.
sentencia es una sentencia Fortran completa, ejecutable, excepto una instrucción DO, END DO, bloque IF u otro IF lógico.

- Se evalúa la expresión lógica **e**. Si su valor es verdadero, se ejecuta "sentencia".
  Si es falso, se transfiere el control a la siguiente instrucción ejecutable después del IF, sin ejecutarse "sentencia".

A.3. Entrada Salida y Manejo de Archivos

En Fortran el término archivo se usa para cualquier cosa que se pueda manejar con READ o WRITE: el término cubre no sólo los ficheros de datos almacenados en disco o cinta sino también periféricos tales como impresoras o terminales.

Antes de que pueda ser usado, un Archivo externo se ha de conectar vía una instrucción OPEN a una unidad de I/O (valores entre 1 y 99).
Existen unidades preconectadas con valores por defecto, como 5=teclado y 6=pantalla. Los archivos son referenciados vía sus números de unidad.

```
OPEN(UNIT=1, FILE=?B:INPUT.DAT?, STATUS=?OLD?)
OPEN(UNIT=9, FILE=?PRINTOUT?, STATUS=?NEW?)
```

Se debe tener en cuenta que la conexión entre un fichero y una unidad persiste hasta que:

- el programa termina (STOP,END).
- otra instrucción OPEN conecta otro archivo a la misma unidad.
- se ejecuta una instrucción CLOSE para esa unidad.

Las unidades de E/S son un recurso global que puede ser utilizado por cualquier unidad de programa, que usarán todas el mismo número de unidad (se le puede pasar a un procedimiento como un argumento).

A.3.1. Formato

El programador puede establecer un formato específico para manejar la entrada/salida a través de la instrucción FORMAT, contrario a la manera libre o sin formato como READ(*,*). La instrucción tiene la forma:

```
label  format(fmt1,fmt2,...,fmtn)
```

donde:
label es una etiqueta que referencia a la sentencia format.
fmt1, fmt2 hasta fmtn son expresiones de formato que pueden indicar un tipo de dato o ser una cadena de caracteres. Por ejemplo para dar formato a la salida que imprime un resultado, podría utilizarse la siguiente sentencia:

```
157  format('El total es = ', I10)
```

En el ejemplo lo que está entre ' ' es una cadena de caracteres y la expresión de formato es I10. Se establece que:

- La sentencia FORMAT debe tener etiqueta (ej. 157). FORMAT puede estar en cualquier lugar en la unidad de programación (pero después de PROGRAM, SUBROUTINE o FUNCTION y antes de END). No es ejecutable.
- En FORMAT, El caracter X indica dejar un espacio (uso: 1X, 2X, etc, pero no X sin número delante). El caracter / pasa una línea (// pasa dos líneas, etc).
- Tipos de datos (los números son simplemente ejemplos):

I6	Datos tipo INTEGER
F13.6	Datos tipo REAL y REAL*8
E13.6 o D13.6	Datos tipo REAL y REAL*8. Escribe con exponente: -0.320E-04
G13.6	El compilador elige escribir como F13.6 o como E13.6 (o D13.6).
L5	Datos tipo LOGICAL. En escritura produce T o F, en lectura acepta T, F, .TRUE. y .FALSE.

A Datos tipo CHARACTER

A5 En lectura, lee los 5 últimos caracteres (es decir, los que están a la derecha). En escritura, escribe los 5 primeros caracteres.

- Los paréntesis dentro de formatos indican repetición de esa parte del formato:

```
      WRITE(*,15) (I,A(I),B(I),I=1,10)\\
15    FORMAT(I10,/,2(1X,E20.16))
```