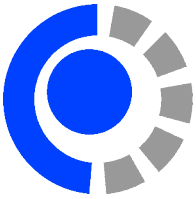
Universidad Nacional del Comahue

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

Nombre del Director

[Nombre del CoDirector]

Neuquén Argentina

2016

Prefacio

Esta tesis es presentada como parte de los requisitos finales para optar al grado académico de *Licenciado/a 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 .... (, el Departamento ... y el Departamento...), de la Facultad de Informática, en el período comprendido entre mes-año-inicio y mes-año-fin, bajo la dirección de ... (y la codirección de ....).

Nombre del Tesista
Facultad de Informática
Universidad Nacional del Comahue
Neuquén, día de Mes de Año.

UNIVERSIDAD NACIONAL DEL COMAHUE

Facultad de Informática

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

Dedicatorias

Aquí van las dedicatorias del alumno. Esta página es opcional.

Agradecimientos

Aquí van los Agradecimientos del alumno. Esta página es opcional.

Resumen

Descripción de la tesis de hasta 500 palabras.

Deberá contener mínimamente motivación, objetivo, resultado y conclusiones.

Abstract

Description of the dissertation of up to 500 words.

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. La aplicación seleccionada es un programa creado y utilizado por un investigador de la Universidad Nacional del Comahue como culminación de una tesis de Doctorado. El programa analiza el comportamiento fluidodinámico de un tipo particular de turbomáquina, la turbina eólica de eje horizontal. Fue desarrollado a fines de la década de 1990, 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.

Esta aplicación fue construida en su momento para la plataforma 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 intentará 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.

1.3. Propuesta de modernización

La aplicación está codificada en lenguaje Fortran, utilizando una estructura de programación secuencial y monolítica. El almacenamiento de datos temporales y resultados tanto parciales como finales se hace en archivos planos. Éstas son algunas respuestas de diseño a las restricciones presentadas por las plataformas de la época en que fue originalmente construida la aplicación; y son otros tantos interesantes puntos de intervención para adecuarla a las arquitecturas actuales, intentando mejorar sus prestaciones.

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 mantenerla.

1.4. Motivación de la tesis

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 maquina 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

Antecedentes

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 adaptados 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 2.1 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 2.2 se presentará brevemente el lenguaje de programación Fortran y se explicará su uso a nivel científico. En las secciones 2.3 y 2.4 se presentan la interfaz de programación de aplicaciones OpenMP y su implementación en Fortran respectivamente. En la sección 2.5 se mostrará en qué consiste el proceso de optimización para paralelizar una aplicación y por último en la sección 2.6 se presentará la aplicación cientifica 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 [REF] 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:

- "Single Instruction, Single Data (SISD)" - "Una instrucción, un dato". Constituyen un sistema de cómputo de un solo procesador 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 multicore", pertenecieron a esta clase.

- "Multiple Instruction, Single Data (MISD)" - "Múltiples instrucciones, un dato". El mismo conjunto de datos es tratado por múltiples procesadores. No se conocen sistemas de esta clase que hayan sido construidos para venta al público o en otro ámbito.

- "Single Instruction, Multiple Data (SIMD)" - "Una instrucción, múltiples datos". 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.

- "Multiple Instruction, Multiple Data (MIMD)" - "Múltiples instrucciones, múltiples datos". 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, un espacio de direcciones virtuales conteniendo las instrucciones del programa y datos, y al menos un hilo de ejecución. 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.

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). 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:

- "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.

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.

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), es decir que debe esperar o puede continuar procesando, respectivamente.

Los hilos ociosos pueden ser suspendidos o quedar en espera activa ("spinning" o "busy waiting"). 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. 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.

Hilos POSIX

Recién en 1995 se estableció un estándar para la programación de hilos, a pesar de que los sistemas operativos venían implementando hilos desde hacía décadas. El estándar es parte de la normativa "POSIX (Portable Operating System Interface)", en particular la porción POSIX 1003.1c [REF]. 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 dificil 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.

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" [REF] 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.

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).

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) [REF]. Luego, en 1994, surge "MPI" [REF] 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. Aún 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 básicas disponibles comercialmente hoy en día. En las siguientes secciones se dará un panorama general de las arquitecturas paralelas.

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.

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, agregándose nodos al sistema paralelo simplemente agregando más estaciones de trabajo.

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).

Otro aspecto es la carencia de una imagen única del sistema. Nos referimos a la capacidad de que el cluster deba verse como un sistema solo y no como un conjunto de computadoras. Un usuario que ingresa en el sistema puede hacerlo siempre desde la misma estación de trabajo, con lo cual no verá diferencias, pero si lo realiza desde distintas estaciones puede ver un ambiente muy distinto al cambiar de equipo. Por lo general se busca mantener los archivos utilizados por el usuario disponibles cada vez que ingrese, sin importar el equipo en el que esté, lo que trae mayor complejidad al cluster.

Multiprocesadores Simétricos (SMPs)

Las computadoras multiprocesador suponen una alternativa para evitar los problemas de los clusters. En un multiprocesador, todos los procesadores acceden a todos los recursos de la máquina. Cuando los procesadores son todos iguales y cada uno tiene acceso igualitario a los recursos de la computadora, el sistema se llama Multiprocesador Simétrico (SMP por la sigla en inglés).

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 SMP debe ser capaz de que todos sus procesadores ejecuten en modo kernel.

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

Un sistema SMP tendrá múltiples procesadores, pero en realidad no tiene múltiples sistemas de E/S o de memoria. Ya que en los SMP se tiene acceso igual o uniforme a la memoria, se dice que son máquinas de Acceso Uniforme a Memoria (UMA - Uniform Memory Access). UMA implica que todos los procesadores pueden acceder a la memoria con la misma latencia.

Buses y Crossbars

Hay diferentes tecnologías que permiten conectar los dispositivos de cómputo entre sí y con el resto de los dispositivos del sistema. Un bus puede ser visto como un conjunto de líneas de conexión utilizado para conectar varios periféricos de la computadora. La comunicación entre los recursos físicos se hace comúnmente con un bus. Generalmente hay dos grupos de buses en un sistema: buses de E/S y de memoria. Los de E/S típicamente son largos, pueden tener diferentes tipos de dispositivos conectados y normalmente acatan un estándar. Por el otro lado, los buses de

memoria son cortos, de alta velocidad y usualmente optimizados, para maximizar el desempeño conjunto entre el procesador y la memoria.

Otro método común de interconectar dispositivos es a través de crossbars. Un crossbar se asemeja a un conjunto múltiples buses independientes que conectan cada uno de los módulos en el multiprocesador. La implementación de un crossbar en hardware es sumamente complicada, debido a que debe permitir tantas comunicaciones independientes como sea posible mientras arbitra los pedidos múltiples de acceso a un mismo recurso, tal como un banco de memoria. Todos los posibles caminos no conflictivos deben ser permitidos simultáneamente, pero esto supone un desarrollo complejo del hardware.

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 cómputo complejas y de altos requerimientos.

El objetivo del proyecto HPC en la FaI es adquirir conocimientos para el diseño, desarrollo, gestión y mejora de las tecnologías de hardware y software involucradas en la Computación de Altas Prestaciones (CAP) y sus aplicaciones en Ciencias e Ingeniería Computacional.

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.

Hubo un tiempo cuando un analista numérico podía escribir código para implementar un algoritmo. El código era miope ya que era escrito solamente con la idea de implementar el algoritmo. Existía cierta consideración en la performance, pero generalmente era más una idea de ahorrar la memoria de la computadora. La memoria era el comodity más preciado en los albores de la computación. "Tunear" el código para la arquitectura subyacente no era una consideración de primer orden. A medida que las arquitecturas de computadoras evolucionaron, la tarea de codificación tuvo que ser ampliada para abarcar la explotación de la arquitectura. Esto fue necesario con el fin de obtener el rendimiento que se exige de códigos de aplicación.

Los centros de computadoras de cada día mejoran los sistemas con más procesadores y más rápidos, más memoria, y subsistemas de E/S mejorados, solo para descubrir que el rendimiento de la aplicación mejora un poco, si es que lo hace. Luego de algo de análisis por los vendedores de sistemas o software, encontraron que sus aplicaciones simplemente no estaban diseñadas para explotar las mejoras en la arquitectura de la computadora. A pesar de los desarrolladores haber leído textos sobre computación de alto desempeño y haber aprendido el significado de las palabras de moda, acrónimos, benchmarks y conceptos abstractos, nunca se les dieron los detalles sobre como diseñar o modificar realmente software que pueda beneficiarse de las mejoras en la

arquitectura de la computadora.

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 el 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 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, lingÃ¼ística 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:

- Dinámica de fluidos computacional, para el diseño de aeronaves, la predicción del tiempo a términos cortos o largos, para 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 [N. E. Gold. The meaning of legacy systems. Univ. of Durham, Dept. of Computer Science, 1998x.]
- Cualquier sistema de información que significativamente resiste las modificaciones y la evolución para alcanzar requerimientos de negocio nuevos y constantemente cambiantes [M. L. Brodie and M. Stonebraker. Migrating legacy systems. Morgan Kaufmann Publishers, 1995].

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 se deteriora gradualmente y puede necesitar cambios de diferente tipo, como: mejoras, correcciones, adaptaciones y prevenciones. Para todas estas tareas se necesita conocimiento y comprensión del sistema. En la era multi-core y many-core, los cambios de software se hacen más y más complejos.

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. 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 release 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). Si se desea ahondar en la definición del estándar Fortran el mismo puede encontrarse en el sitio de NAG (The Numerical Algorithms Group) quienes alojan el home oficial para los estándares de Fortran [referencia al sitio http://www.nag.co.uk/sc22wg5/] [estándares http://www.nag.co.uk/sc22wg5/links.html].

El lenguaje utilizado en el programa de estudio de este trabajo de tesis está basado en el estándar Fortran 77. Se presentan a continuación algunas de las sentencias de Fortran más utilizadas en la aplicación de estudio de esta tesis como referencia para el capitulo siguiente.

Manejo de archivos

(voy a explicarlo mejor, en parrafos onda subsecciones, por ahora lo dejo así)
Aca agregar lo que está en el CAP2.md

Estructuras de control

Aca agregar lo que está en el CAP2.md

Otros

Aca agregar lo que está en el CAP2.md

2.5. OpenMP

"https://computing.llnl.gov/tutorials/openMP/" OpenMP es una Interfaz de Programación de Aplicaciones, o API por sus siglas en Ingles, la cual provee un modelo portable y escalable

para el desarrollo de aplicaciones paralelas de memoria compartida. La API soporta C/C++ y Fortran en una gran variedad de arquitecturas. Es utilizada para de manera directa aplicar multihilos en memoria compartida. "f95 openmpv1 v2 dot pdf" La especificación de OpenMP 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. Mas información en www.openmp.org. "Using OpenMP-oct2007" 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 como el trabajo debe ser compartido entre los hilos que se ejecutaran 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, sino 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.

Las directivas de OpenMP permiten al usuario indicarle al compilador que 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 diretiva 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 multiples hilos colaboran para ejecutar un programan, 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). 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 multihilo. Soporta el modelo de programación llamado fork-join, el cual podemos ver en la figura 2.3.1.aa.

Figura 2.1 de pag 49 using openmp

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 (esta es la parte fork), se convierte en el maestro del equipo, y colabora con los otros miembros del equipo para ejecutar el código dinámicamente encerrado por la construcción. Al final de la construcción, solo el hilo original, o maestro del equipo, continua; todos los demás terminan (esta es la parte join). Cada porción del código encerrada por una construcción paralela es llamada una región paralela.

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.

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 diretiva 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. Es posible realizar anidado de regiones paralelas.

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 computo 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.

Probablemente el método más común de trabajo compartido es 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.

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 tienen 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.

Sincronización de Hilos

Sincronizar los hilos es necesario en ocasiones para asegurar el acceso en orden a los datos compartidos y prevenir corrupción de datos. Asegurar la coordinación de hilos necesaria es uno

de los desafíos más fuertes de la programación de memoria compartida. OpenMP provee, por defecto, sincronización implícita haciendo que los hilos 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. A esto se lo conoce como una barrera. Mas 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.

Otras caracteristicas

Subrutinas y funciones pueden complicar la utilización de APIs de Programación Paralela. Una de las características innovativas 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 a 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.

2.6. OpenMP en Fortran

F95_OpenMPv1_v2.

Fueron presentados Fotran y OpenMP, en esta sección se ve como se complementan, mostrando cuales 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 sentinelas:

```
!$OMP

!$
```

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

- !$OMP : el compilador compatible con OpenMP sabe que la información que sigue en la linea es una directiva OpenMP. Se puede extender una directiva en varias lineas utilizando el mismo centinela frente a las siguientes lineas y usando el método estándar de Fortran para partir lineas 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 linea será interpretada como un comentario.

- !$: la linea 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 linea. Como en el caso anterior podemos extender la linea en varias lineas 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. Ya que la región paralela necesita ser creada/abierta y destruida/cerrada, dos directivas son necesarias en Fotran: !$OMP PARALLEL / !$OMP END PARALLEL. Un ejemplo de su uso:

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

Agregar figura 1.1 de F95_OpenMPv1v2 donde muestra la representación de la región.

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 la región paralela 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 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 el trabajo es distribuido en un equipo de hilos al encontrar el código esta directiva. 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 figura 2.4.3.1 (figura 2.1 de f95_openmpv1_v2). 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

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 que variables van a ser consideradas variables locales de cada hilo. Por ejemplo:

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

indica que las variables a y b tendrán diferentes valores en cada hilo, serán locales a cada hilo. 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. (figura para mostrar esto? Figura 3.1 del pdf) 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, aún cuando no hayan sido declaradas en un cláusula PRIVATE.

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. (mostrar gráficamente con figura 3.2 del pdf?) 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.

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 evitar esto, es posible especificar que hará OpenMP cuando no se especifica nada sobre una variable, es posible especificar 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).

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 figura 2.4.4.4.1 (figura 3.4 del pdf) 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.

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 o en libros como "Parallel Programming in Fortran 95 using OpenMP" [referencia a F95_openmpv1_v2].

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 [REF GySh]:

- 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.

El proceso de optimización tiene algunas etapas fundamentales: "desarrollo de la aplicación, optimización serial, y optimización paralela" (Fig. XX [REF G y Sh]). 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 las que serán explicadas brevemente en esta subsección.

Figura 2.5.aa (pag 51 garg y sharapov)

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. 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.

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 figura 3.bb resume las tareas de optimización y da un diagrama de flujo simplificado para el proceso de optimización serial.

Figura 2.5.bb (pag 52 garg y sharapov)

Una vez que las mediciones de rendimiento de la linea de base se han obtenido, el esfuerzo de optimización debe iniciarse mediante la compilación de todo el programa con opciones seguras. Siguiente, linkear librerías optimizadas. (Este linkeo es una manera sencilla de llevar implementaciones altamente optimizadas de operaciones estandar 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 Ingles) adecuadas en las librerías optimizadas. Además, es recomendable que sea medido el desempeño del programa para verificar que ha mejorado.

El siguiente paso es identificar partes de desempeño críticas en el código. El perfilado (profiling en Ingles) 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.

Metodología de medición

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

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 cientificos, la 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. A traves 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. Con esto en mente 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 y swapeo deben ser monitoreadas mientras se ejecuta el benchmark, ya que estas pueden desvirtuar completamente la medición.
- Las mediciones de "respuesta del programa" deben ser desempeñadas en una manera dedicada, sin otros programas o aplicaciones ejecutandose.
- Las caracteristicas del sistema deben ser registradas y guardadas.

Herramientas de medición

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

- Herramientas de temporizador, que miden el tiempo utilizado por un programa de usuario o sus partes. Pueden ser herramientas de linea 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.

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 programa. 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:

- "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.

Podemos ver en la figura 2.5.1.3.x un ejemplo de ejecución del comando time.

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 multiples, 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 compilada 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 en la figura 2.5.1.4.x

figura 2.5.1.4.x

```
$ 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 secuencial, su tiempo de ejecución puede ser reducido aún más permitiendo que se ejecute en varios procesadores. Las técnicas más usadas comunmente para paralelización son el uso explícito de hilos, el uso de directivas al compilador y el pasaje de mensajes. En la figura 3.cc se vé ilustrado el proceso de optimización paralela.

Figura 3.cc (garg y sharapov pag 55)

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 balanceo de carga muy pobre; un nivel más fino en el particionamiento 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; Minimizar el desequilibrio de cargas entre las tareas paralelas, ya que esto degrada la escalabilidad del programa. También se debe 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 comunmente, tanto en aplicaciones multihilo 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 extras 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$\hat{3}$) en la etapa de descomposición de la matriz y en el orden de O(N$\hat{2}$) en la etapa de sustitución adelante-atrás. En consecuencia, la etapa de sustitución adelante-atrás 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 adelante-atrás. El desempeño total y velocidad del programa de resolución directa ahora está limitado por el desempeño de la etapa de sustitución adelante-atrás. Para mejorar aún más el desempeño, la etapa de sustitución adelante-atrás debería convertirse en el foco de optimización y posiblemente un trabajo de paralelización.

2.8. Caso de Estudio: Modelización del Flujo Invíscido

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 en el área de Ingeniería [Prado, mayo 2007] presentado en la Universidad de Buenos Aires en 2007. Como se expone en dicho trabajo, la tesis "analiza el comportamiento fluidodinámico de una turbomáquina particular: la turbina eólica". La creación del programa se justifica en el mismo trabajo porque "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".

El modelo matemático que implementa el programa es la "Ley de Biot-Savart", que indica el campo magnético creado por corrientes eléctricas estacionarias. Es una de las leyes fundamentales de la magnetoestática. En particular, en el trabajo de Prado se aplica a una modelización del flujo invíscido (de viscosidad despreciable, casi nula) alrededor de la pala de la turbina. El modelo numérico se formula a través del método de los paneles. El objetivo de la aplicación de la Ley de Biot-Savart en el trabajo es el cálculo de las velocidades de flujo inducidas en un punto para cada panel de la pala.

El programa realiza el cálculo de una integración 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.

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 la programación, 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 creador, 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 rutinas 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

Primero se analizó la aplicación para poder proceder con su optimización y paralelización. Como se explicó en la sección 2.5, determinamos la plataforma en que debería ejecutarse la aplicación, determinando 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.

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.

Con esto ya verificado se avanzó en el trabajo de Tesis hacia el análisis propiamente dicho de la aplicación.

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, por lo cual no debimos preocuparnos por esta parte, 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.

Contamos con dos casos de test provistos por el creador de la aplicación, 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 = no = 50 en el primer caso de test, y nr = no = 80 en el segundo caso.

Estos valores también definen unas variables globales comunes de la aplicación llamadas "maxir" y "maxio" que se fijan a 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 el tratamiento de dichos valores 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 rutinas 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 rutina estela() resulta ser la que consume el mayor porcentaje, 79,83 % del tiempo de ejecución de la aplicación. Le sigue la rutina solgauss() con un 14,36 %. Estos datos se pueden observar en la figura 3.x

Agregar imagen 3.x (Dropbox/Tesis/andres_scripts/tesis/oso_fortran/salida.gprof) - explicar los datos y luego apuntar al gráfico y no al reves

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

Para las pruebas se utilizaron dos computadoras de escritorio distintas, ambas multiprocesadores. El primer equipo posee un procesador AMD Phenom II con 4 nucleos y 4GB de memoria RAM. El segundo equipo consta de un procesador Intel Core i3 con 2 núcleos (2 hilos cada procesador) 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.x 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 rutina 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.x1.

Agregar imagen 3.x1 (tesis/oldone/omg2k14gprof)

A esta altura del trabajo contamos con çódigo correcto no optimizado"(Unoptimized Correct Code)[ref Techniques for Optimizing Apps: HPC], de modo que, siguiendo las etapas del proceso de optimización 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. Estos pasos se ilustran en la Fig 3.y.

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

imagen 3.y del pdf de estos muchachos (figura 1-1, hoja 51 del pdf)

3.3. Optimización Serial del código Fortran

La optimización serial es ün proceso iterativo que involucra medir repetidamente un programa seguido de optimizar sus porciones críticas"[REF Techniques for]. Obtenidas las mediciones iniciales del comportamiento, debemos optimizar las opciones de compilación y vincular el código objeto con bibliotecas optimizadas. En el trabajo de tesis intentamos 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.

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

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 los puntos de aplicación de 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.y.

Agregar Tabla 3.y de archivos y subrutinas y sus operaciones

Fortran ofrece los archivos regulares, o archivos externos soportados en disco (External Files), pero también los archivos internos o Internal Files, que son cadenas de caracteres o arreglos de cadenas de caracteres, localizados en memoria principal. Los Internal Files 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 [REF Systems Performance, BRENDAN GREGG], cambiando la definición de los archivos en disco a Internal Files (siempre que la restricción de tamaño del sistema de memoria virtual lo permita) conseguimos una mejora sustancial de performance 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 dos excepciones: 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 Internal File de Fortran y su escritura se demorará hasta la finalización del programa.

Luego, todo archivo o External File que sea escrito y leído durante la ejecución de la aplicación será mantenido por un Internal File. 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.xxx se muestra un ejemplo de código previo a la modificación, y en la Fig. 3.xxy el código ya modificado. Imagen 3.xxx mostraría:

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

    Imagen 3.xxy mostrar\'ia:
      character subrout(500)*60   ! definici\'on del Internal File
      ...
      write(subrout(nsubr),1)
      nsubr=nsubr+1
    !      write(15,1)
      write(6,1)
```

Como se ve, reemplazamos el archivo "subr.out" representado por el identificador de unidad 15 por el Internal File denominado subrout. Como se ha dicho, el External File subr.out pasa a ser manejado como un Internal File, que como se ve en la declaración de la Fig. 3.xxy, es un arreglo de 500 cadenas de 60 caracteres como máximo. La variable nsubr mantiene la posición en el internal file a ser escrita, y el argumento "1" en los comandos write es un formato de escritura definido dentro del programa como se explicaba en el capítulo anterior. En la tabla 3.yy vemos cómo quedan las equivalencias de los External Files y su correspondiente cambio a Internal File.

Tabla 3.yy con tabla de External a Internal file

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 internal files, 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 alojarlos resultados, ya que los archivos TMP eran binarios sin formato, que transportaban valores calculados de una subrutina a otra.

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.y, los archivos mencionados son escritos en la subrutina "circulac" y leidos 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 nuestros Internal Files 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 explicitamente 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 Internal Files 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 Internal File para cindg.tmp y coefgtmp el Internal File para coefg.tmp.

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

```
      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, internal file 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 internal file 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" [Prado2005 - líneas 2806-2841]:

```
      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 rutina 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:

```
      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 internal files. 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 Internal Files, y ambos arrays deben ser declarados explícitamente en la sección correspondiente.

Luego de que solgauss conoce la existencia de los internal files 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 internal files (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 internal file a la matriz "coefg". La variable "incfg" es utilizada para seguir la posición del internal file "tmpcoefg", ya que la posición del internal file "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.yy.

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 esta 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 3.3.2.yy se enumeran los archivos que se decidió manejar mediante un archivo interno y el motivo de dicha decisión:

- Tabla 3.3.2.yy con los archivos modificados con archivos internos. Tendra archivo externo - archivo interno - tipo de cambio/observaciones: ej, solo utilizado en memoria, o escritura diferida

Una vez realizados los cambios indicados, verificamos que los resultados siguieran siendo los correctos. 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 mayor peso en la ejecución, seguida de solgauss, incluso en porcentajes bastante aproximados a los obtenidos para el programa original (Fig. 3.4.x).

FIgura 3.4.x (pruebas/newone/estela_intfiles)

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. 3.4.xx aparece el pseudocódigo anotado de la subrutina estela.

Figura 3.4.1.xx

```
subrutina estela()
```

```
definicion variables globales y constantes;

begin
    Do de i=1 a 2500
    Do de j=1 a 51
            ciex(i,j) = 0
            ciey(i,j) = 0
            ciez(i,j) = 0
    end do
    end do
    cálculos parciales
    ib = 1

Do de ir=1 a 2500
Do de npa=1 a 51
    Do de ik=1 a 2001
            genera fx(ik), fy(ik), fz(ik), dista(ik), denom(ik)
    end do

# sumatoria terminos impares six, siy, siz
    six,siy,siz = 0
    Do de ik=2 a 2000
            six = six + fx(ik)/denom(ik)
            siy = siy + fy(ik)/denom(ik)
            siz = siz + fz(ik)/denom(ik)
            ik = ik +2
    end do
# sumatoria terminos pares spx, spy, spz
    spx,spy,spz = 0
    Do de ik=3 a 2000
            spx = spx + fx(ik)/denom(ik)
            spy = spy + fy(ik)/denom(ik)
            spz = spz + fz(ik)/denom(ik)
            ik = ik +2
    end do
    calculo ciex(ir,npa), ciey(ir,npa), ciez(ir,npa)
    if (indice = 1) and (ib = 1) then  ## se ejecuta solo 1 vez
    # calculo coeficientes de la estela x, y, z
            if (i = nr) and (j = nr/2) then
            # calculo para i=50 y j=25
            # nr depende del tamaño del problema,
            # en este caso el tamaño es 50
                    inicializa valx,valy,valz
                    Do de ik=1 a 2001
                            escribe archivo integ1.txt con varios valores incluyendo
                                valx,valy,valz
                            if (ik =/= 2001) then
                                    const=1
                                    if (ik == 2000) then
                                            const=0.5
                                    endif
                                    valx = valx + fx(ik+1)/denom(ik+1)*otros valores
                                    valy = valy + fy(ik+1)/denom(ik+1)*otros valores
                                    valz = valz + fz(ik+1)/denom(ik+1)*otros valores
                            else
                                    escribe integ1.txt con varios valores sin valx,
                                        valy,valz
                                    pero con ciex,ciey,ciez(i,j)
                            endif
            else
                    if (i = nr/2) and (j = nr+1) then
                    # Luego (si no entr\'o en el anterior if) el calculo es para i =
                        25 y j=51
                            Repite mismo trabajo pero escribiendo integ2.txt
                    endif
            endif
    endif
end do
end do
end
```

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 (capítulo 2 OpenMP), 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 dependencias 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 de código para que pueda ser paralelizable a pesar de tener estas dependencias de datos. Debido al potencial gran cambio necesario en el código para subsanar el problema de la dependencia de datos, 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 figura 3.4.1.xx que genera los arrays fx, fy, fz, denom y dista, 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) de la fig. 3.4.2.xx. 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)=([cÃ¡lculo 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. Podemos ver en la figura 3.4.2.yy cuales son:

arrays: pcx, pcy, pcz, xe, ye, ze, re, fi Variable: c0

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 figura 3.4.1.yy.

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.

```
!$OMP PARALLEL DEFAULT(SHARED)
!$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 queda como vemos en la Fig. 3.4.2.zz

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

Capítulo 4

Nombre del cuarto capítulo

Apartir de aquí el contenido del capítulo 4.

Capítulo 5

Nombre del quinto capítulo

Apartir de aquí el contenido del capítulo 5.