Desarrollo de aplicaciones con técnicas de programación paralela para el análisis del procesamiento 3D de imágenes de microscopía.

LUIS FELIPE HERNÁNDEZ ANILLO

Desarrollo de aplicaciones con técnicas de programación paralela para el análisis del procesamiento 3D de imágenes de microscopía.

LUIS FELIPE HERNÁNDEZ ANILLO

UNIVERSIDAD TECNOLÓGICA DE BOLÍVAR
PROGRAMA DE INGENIERÍA DE SISTEMAS
FACULTAD DE INGENIERÍA
CARTAGENA DE INDIAS
2015

Desarrollo de aplicaciones con técnicas de programación paralela para el análisis del procesamiento 3D de imágenes de microscopía.

LUIS FELIPE HERNÁNDEZ ANILLO

Proyecto para optar al título de Ingeniero de Sistemas

ISAAC ZUÑIGA SILGADO

MAGISTER EN ADMINISTRACIÓN DE EMPRESAS

HERNANDO ALTAMAR MERCADO

MAGISTER EN FISICA.

UNIVERSIDAD TECNOLÓGICA DE BOLÍVAR
PROGRAMA DE INGENIERÍA DE SISTEMAS
FACULTAD DE INGENIERÍA
CARTAGENA DE INDIAS
2015

Nota de aceptación:

_____

_____

_____

_____

_____

DEDICATORIA

Este trabajo de grado está dedicado a mis padres Elmy Anillo y Alberto Hernández, quienes siempre estuvieron ahí para darme el apoyo necesario para superar cada uno de los obstáculos que se fueron presentando a lo largo de mi carrera.

A mi hermana y tía Neida por apoyarme de manera incondicional, porque ellas como mis padres son el motivo y razón que me han llevado a seguir día a día para alcanzar esta meta de ser profesional.

A todos los que mencioné, son el motivo y la razón que me ha llevado a seguir día a día, para alcanzar mis ideales de superación, ellos fueron quienes en los momentos más difíciles me dieron su amor y compresión para poderlos superar, quiero también dejar a cada uno de ellos una enseñanza que cuando se quiere alcanzar algo en la vida, no hay tiempo ni obstáculo que lo impida para poderlo lograr**.**

AGRADECIMIENTOS

Este trabajo no habría sido posible sin la influencia directa o indirecta de muchas personas a las que les agradezco profundamente por estar presentes en las distintas etapas de su elaboración, así como en el resto de nuestras vidas.

Le agradezco a los profesores Isaac Zuñiga y Hernando Altamar por manifestar su interés en dirigir este trabajo de grado, por su confianza, colaboración y apoyo en el proceso de realización del mismo.

A todos los docentes de la Universidad Tecnológica de Bolívar que compartieron sus conocimientos, dentro y fuera de clase, haciendo posible que nuestra formación profesional se resuma en satisfacciones académicas e inquietudes insatisfechas en continua indagación.

A nuestros amigos y compañeros, en especial a Freddy Mendoza, quienes trabajaron conmigo hombro a hombro durante varios años poniendo lo mejor de su energía y empeño por el bien de mi formación profesional, a quienes compartieron su confianza, tiempo, y los mejores momentos que vivimos durante esta etapa como estudiantes de pregrado, dentro y fuera del campus.

CONTENIDO

Lista de tablas

Lista de figuras

Glosario

RAM: La memoria de acceso aleatorio (en inglés: random-access memory, cuyo acrónimo es RAM) es la memoria desde donde el procesador recibe las instrucciones y guarda los resultados.

IMÁGENES DE MICROSCOPÍA: Microscopía es la construcción y empleo del microscopio, cuando hablamos de imágenes de microscopía nos referimos a imágenes tomadas por un microscopio.

MÁQUINA VIRTUAL: En informática una máquina virtual es un software que emula a una computadora y puede ejecutar programas como si fuese una computadora real. Este software en un principio fue definido como "un duplicado eficiente y aislado de una máquina física". La acepción del término actualmente incluye a máquinas virtuales que no tienen ninguna equivalencia directa con ningún hardware real.

PROGRAMACIÓN PARALELA: Es una técnica de programación que consiste en el uso de varios procesadores trabajando en conjunto para dar solución a una tarea en común.

IDE: Integrated development environment por sus siglas en inglés, Ambiente de desarrollo interactivo o Entorno de desarrollo integrado, es una aplicación de software que proporciona servicios integrales para facilitarle al programador de computadora el desarrollo de software.

LENGUAJES DE PROGRAMACIÓN: Es un lenguaje formal diseñado para expresar procesos que pueden ser llevados a cabo por máquinas como las computadoras.

SSH: Secure Shell por sus siglas en inglés, Interprete de órdenes segura, es el nombre de un protocolo y del programa que lo implementa, sirve para acceder a máquinas remotas a través de una red.

LENGUAJE COMPILADO: Es un lenguaje cuyas implementaciones son normalmente compiladores (traductores que generan código de máquina a partir del código fuente).

LENGUAJE INTERPRETADO: Es un lenguaje que es ejecutado paso a paso del código fuente, donde no se lleva a cabo una traducción en la preejecución.

SINTAXIS: Es la parte de la gramática que estudia las reglas y principios que gobiernan la combinatoria de constituyentes sintácticos y la formación de unidades superiores a estos, como los sintagmas y oraciones gramaticales.

COMPILAR: Traducir un lenguaje de alto nivel a código absoluto o lenguaje binario.

Introducción.

La programación paralela, lleva bastante tiempo siendo desarrollada e implementada, sin embargo, en los últimos años ha crecido el interés debido a las limitaciones físicas que impiden el aumento de la frecuencia en los procesadores.

La programación paralela nace para solucionar los problemas que requieren mucho tiempo de ejecución y grandes recursos informáticos. A través del uso simultáneo de procesadores se resuelven problemas de manera más rápida que lo que se puede realizar en un solo procesador. La programación paralela se basa en la división del problema en pequeñas partes para ser resueltos cada uno de estos en paralelo.

El paralelismo ha sido utilizado con éxito en muchos campos como la informática de alto rendimiento, servidores, aceleradores gráficos, y muchos sistemas embebidos. El punto de inflexión multinúcleo, sin embargo, afecta a la totalidad del mercado, en particular el espacio del cliente, donde el paralelismo no ha sido previamente extendido. Los programas con millones de líneas de código debe ser convertido o reescrito para aprovechar el paralelismo, sin embargo, tal como se practica hoy en día, la programación en paralelo para el cliente es una tarea difícil, puesto que es realizada por pocos programadores. Los programas paralelos son notoriamente difíciles para poner a prueba, debido a los tipos de datos, intercalaciones no deterministas y modelos complejos de memoria.

En el grupo de investigación de física aplicada y procesamiento de imágenes y señales de la Universidad Tecnológica se vienen desarrollando proyectos de inspección de muestras microscópicas metálicas que requieren el procesamiento de un gran volumen de datos. El procesamiento de estos datos se hace supremamente lento cuando se utilizan técnicas de programación secuenciales.

Actualmente la Universidad Tecnológica de Bolívar no cuenta con una guía que permita probar, comparar y seleccionar la herramienta adecuada para desarrollar aplicaciones utilizando técnicas de programación paralela. Lo cual dificulta el trabajo de los grupos de investigación que trabajan en esta área del conocimiento en proyectos que requieren mejorar los tiempos de respuesta de sus aplicaciones.

Este trabajo intenta solucionar estos inconvenientes trazándonos los siguientes objetivos

Objetivo General

> Elaborar una guía de desarrollo de aplicaciones utilizando técnicas de programación paralela mediante el análisis del procesamiento 3D de imágenes de microscopía.

Objetivos Específicos

- Establecer el tipo de paralelización que se ajusta al problema del procesamiento 3D de imágenes de microscopía.
- Desarrollar una aplicación paralela para el procesamiento 3D de imágenes de microscopía.
- Comparar el rendimiento entre un software desarrollado con técnicas de programación secuencial versus programación paralela.
- Crear una guía de alto nivel para desarrollar aplicaciones con técnicas de programación paralela desde cero, y para la implementación de técnicas de programación paralela en aplicaciones desarrolladas con técnicas de programación secuencial.

A nivel regional, local e institucional, existen entidades y grupos de investigación que requieren de herramientas tanto de software como de hardware para evaluar y mejorar los tiempos de respuesta y de procesamiento de los proyectos de investigación que adelantan. Esto impulsa el desarrollo inmediato de trabajos que puedan servir de base a nuevas investigaciones e intereses haciendo uso adecuado de la programación paralela. A nivel nacional e internacional, este trabajo servirá para establecer un puente con las instituciones que estén trabajando en este campo de las ciencias computacionales.

Con el fin de mejorar los tiempos de respuesta al realizar el procesamiento 3D de imágenes de microscopía, con las cuales se realiza inspección de superficie, control de calidad, análisis de corrosión, entre otras actividades, se emplearán técnicas de programación paralela.

En el procesamiento 3D de imágenes de microscopía se han encontrado el siguiente inconveniente:

- Demora en el procesamiento de obtención de resultados en la estimación de topografías de objetos microscópicos.

En síntesis, una vez terminado este trabajo de investigación se pondrá al servicio de la comunidad el software de procesamiento 3D de imágenes de microscopía y la guía de desarrollo de aplicaciones paralelas.

A continuación podemos ver de forma introductoria la estructura de los capítulos que componen este documento:

Capítulo 1. Sistemas formadores de imágenes y métricas de enfoque. En esta sección se describen los conceptos básicos relacionados con los sistemas formadores de imágenes y métricas de enfoque, las diferentes técnicas de forma por enfoque, desenfoque o acomodación, así como el diagrama de bloque que nos ilustrará el proceso para realizar el procesamiento 3D de imágenes de microscopía.

Capítulo 2. Introducción a la programación paralela. En esta sección se describen los conceptos básicos relacionados con programación paralela, como son los diferentes tipos de programación paralelas existentes, las arquitecturas, los principales y no principales lenguajes que soportan la programación paralela, así como las principales leyes sobre las que se rige el paralelismo.

Capítulo 3. Programación paralela versus programación secuencial. En esta sección se encontrarán las principales ventajas y desventajas de la programación paralela como de la programación secuencial, adicional a esto encontremos las diferentes pruebas realizadas donde se compararan ambas técnicas con diferentes números de procesadores.

Capítulo 4. Guía para desarrollar aplicaciones implementando programación paralela. En esta sección se encontrará una guía para la instalación del entorno de desarrollo de programación paralela usando C++, OpenMP, MPI, QT y OpenCV, así como también encontraremos metodologías para desarrollar programas paralelos, el procedimiento para paralelizar un programa secuencial y el procedimiento para desarrollar un programa paralelo desde cero.

Capítulo 1

Sistemas formadores de imágenes y métricas de enfoque

La configuración óptica de un sistema formador de imágenes incluye lentes que producen la imagen de la escena en una posición fija, donde se coloca el dispositivo de registro que puede ser una película fotográfica o sensor CCD o CMOS para un registro digital. Un sistema sencillo formador de imagen lo constituye una lente de longitud focal *f*, ella formará la imagen de un objeto ubicado a la distancia z_o, medida desde la lente, a la distancias z_i medida desde la misma la lente. En la siguiente imagen se ilustra la geometría de un sistema formador de imagen constituidos de lentes.

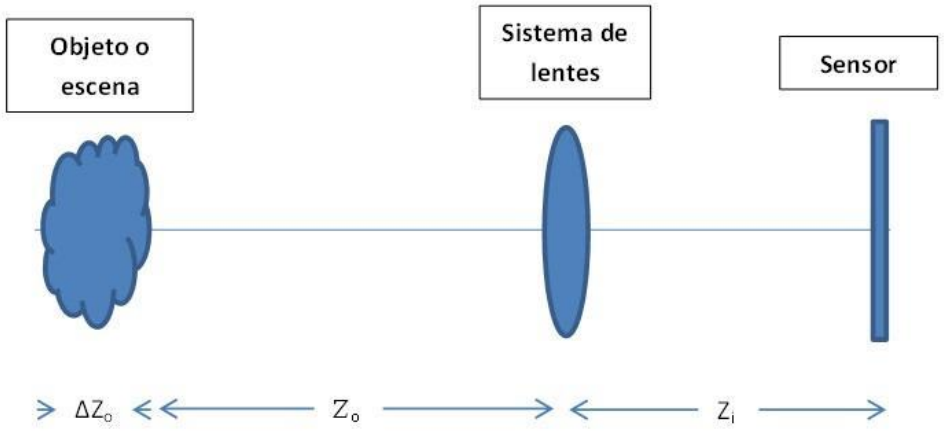


Figura 1. Geometría de un sistema formador de imagen constituido de lentes.

La geometría del sistema constituido por una lente delgada conduce a una ecuación conocida como la ecuación de las lentes delgadas:

$$\frac{1}{z_0} + \frac{1}{Z_i} = \frac{1}{f'},$$

Donde *f* en la distancia focal de la imagen, z_o y z_i las distancias objeto y distancia imagen medidas desde la lente.

Si el objeto es un objeto plano, es decir, sin ningún relieve, el sistema formará la imagen plana perfecta a la distancia z_i, dicha imagen se vuelve borrosa o difusa y pierde nitidez a medida que el objeto se mueve a uno u otro lado de la posición donde se aprecia la mejor imagen. Todo sistema tiene un rango Δz_o conocido como profundidad de campo, sobre el eje óptico de la lente, en el que a juicio de un observador, la imagen del objeto puede considerarse bien formada.

Ahora si el objeto tiene relieve entonces ocupa un volumen tridimensional y solo será nítida la porción de la imagen correspondiente a la porción del objeto que se

encuentre dentro de la profundidad de campo del sistema formador de imagen. De modo que si se pretende inspeccionar todo el objeto, debe variarse la distancia relativa entre objeto y el sistema formador de imagen de modo que todos los puntos del objeto ingresen a la profundidad de campo del sistema. Se logra el mismo efecto variando la distancia relativa entre el sistema formador de imagen y la posición del dispositivo de registro de la imagen. Esta situación se ilustra en la siguiente imagen donde se muestran dos imágenes de la misma escena registradas para diferente posición del objeto, se puede apreciar que en cada imagen existe una región nítida y una región borrosa o difusa.

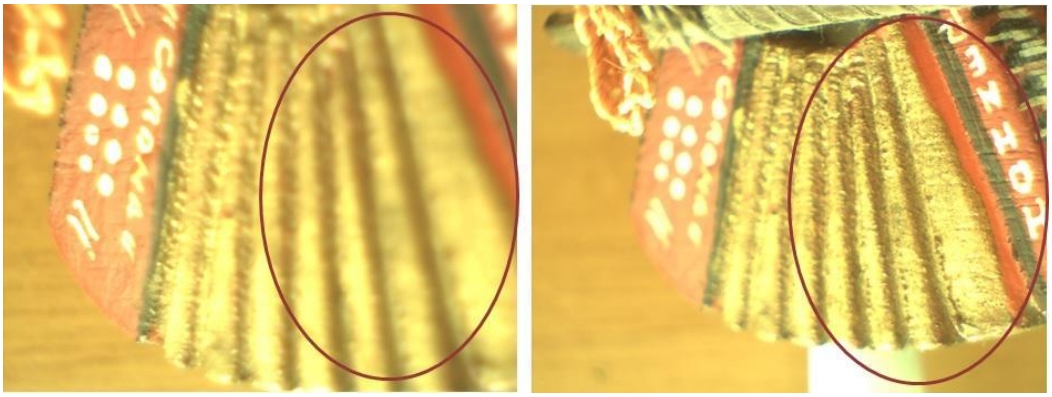


Figura 2. Porción elíptica desenfocada y enfocada.

En visión artificial o computacional las técnicas dedicadas a la recuperación de la información tridimensional de una escena o de un objeto a partir de una serie de imágenes son conocidas como técnicas de forma por enfoque, desenfoque o acomodación; (SFF) o (SFD). Con esas técnicas se puede estimar la forma, el movimiento de un objeto, nitidez, textura y otros atributos de la escena.

SFF o SFD permite la recuperación de la forma a partir de múltiples imágenes de la misma escena, las cuales son registradas variando la geometría, generalmente la distancia de la escena al sistema formador de imagen junto con el dispositivo de captura de imagen.

En algoritmos basados en SFD[1] se estima la distancia de un punto midiendo el grado de borrosidad en una o dos imágenes mientras que en algoritmos basados en SFF[2] se requiere la búsqueda del mejor ajuste de la geometría de captura de imágenes que conduce a la mejor posición de enfoque para cada punto. Por esta razón, estas técnicas requieren enfocar un punto lo mejor posible por lo tanto hay que buscar esa condición en una imagen específica de una serie de imágenes. Así se consigue una mejor calidad de la forma recuperada pero se requiere de un mayor costo computacional.

[1] A. Pentland, A new sense for depth of field, IEEE Trans. Pattern Anal. Mach. Intell. 9 (4) (1987) 523–531

[2] E.P. Krotkov, Focusing, Int. J. Comput. Vision 1 (3) (1987) 223–237.

Si se pretende recuperar la información tridimensional de un objeto utilizando técnicas de SFF se requiere de adquirir una colección de imágenes variando el grado de enfoque que se consigue al mover el sistema lente-dispositivo de registro respecto al objeto. La colección de imágenes constituye una información tridimensional codificada en la que cada imagen codifica la distancia z o profundidad y la porción nítida de la imagen codifica las dimensiones laterales del objeto.

Cada punto del objeto es enfocado hasta que en el espacio imagen se apreciar su mejor imagen, luego sobreviene su desenfoque y se aprecia que el punto vuelve a tornarse borroso a medida que el objeto se desplaza a lo largo del eje óptico. Para recuperar la información tridimensional se usa una métrica de enfoque que debe obtener su valor máximo, para cada pixel de la imagen, en una posición de enfoque en la cual se parecía el mejor contraste y nitidez de la imagen. Esa posición de enfoque da información de la distancia estimada z_o del punto objeto enfocado. El criterio se usa hasta para cada pixel en el plano imagen haya conseguido su posición de máximo valor máximo de la métrica de enfoque.

1.1 MÉTRICAS DE ENFOQUE

Una métrica de enfoque se define como la cantidad que evalúa localmente la nitidez de una región alrededor de un pixel y es un valor asignado a un pixel. Para calcularlo se toma una región de píxeles vecinos, comúnmente llamada ventana, y se evalúa la nitidez y dicho valor se asigna al centro de la ventana. El valor de la métrica varía a medida que el pixel se desenfoca.

Existen una variedad de métricas de enfoque las cuales se han propuesta para evaluarlas tanto en el dominio espacial como en el dominio de las transformadas[3]. Algunos de los más utilizados son, Suma del laplaciano modificado (SML), gradiente Tenenbaum (TEN), y Varianza del nivel de gris (GLV).

El algoritmo SML se basa en el operador laplaciano el cual por su propia definición en segundas derivadas, puede ocurrir que las segundas derivadas respecto a *X* y respecto a *Y* se cancelen entre sí, este problema fue resuelto por Nayar y Nakagawa quienes propusieron un algoritmo basado en el laplaciano modificado (ML) definido como:

$$ML(x,y) = \left|\frac{\partial^2 g(x,y)}{\partial x^2}\right| + \left|\frac{\partial^2 g(x,y)}{\partial y^2}\right|$$

Donde *g(x,y)* es el nivel de gris en la coordenadas (x,y) de una imagen.

Para hacer que los algoritmos sean robustos ante el ruido, los algoritmos se evalúan en el píxel *p(x,y)* de coordenadas *(x,y)* como la suma de la respectiva

[3] S. Nayar, Y. Nakagawa, Shape from focus, IEEE Trans. Pattern Anal. Mach. Intell. 16 (8) (1994) 824–831.

métrica en una ventana *V(x,y)* centrada en dicha coordenada. De ese modo las métricas de enfoque más utilizadas son:

Suma del laplaciano modificado SML:

$$SML(x_0, y_0) = \sum_{p(x,y) \in V(x_0,y_0)} ML(x, y)$$

Tenenbaum propuso una métrica basada en el operador gradiente Sobel definida como:

$$TEN(x_0, y_0) = \sum_{p(x,y) \in V(x_0,y_0)} \left[G_x^2(x, y) + G_y^2(x, y)\right]$$

Donde G_x y G_y son las respuestas al filtro Sobel en las direcciones horizontal y vertical respectivamente. La métrica GLV se basa en las variaciones de los valores de niveles de gris, ese valor es mayor cuanto más nítida sea la imagen, Esta métrica se define como:

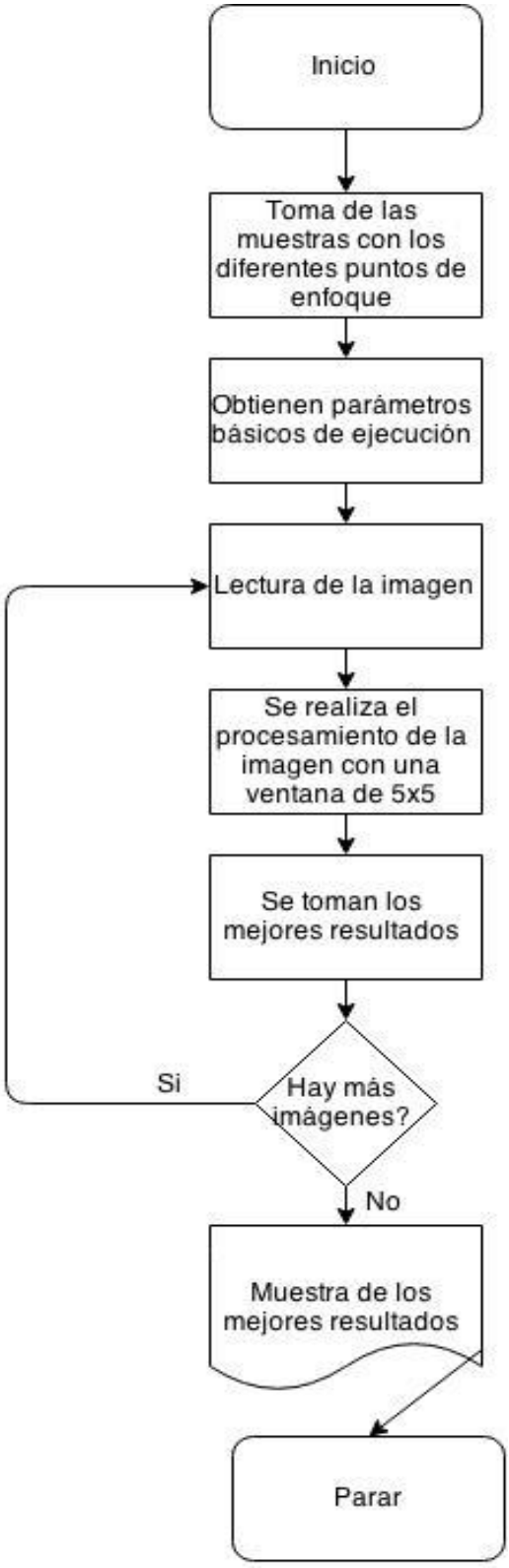
$$GLV(x_0, y_0) = \sum_{p(x,y) \in V(x_0,y_0)} \left[g(x, y) - \mu_{V(x_0,y_0)}\right]^2$$

Donde μ es el valor medio de los valores de nivel de gris en la ventana $V(X_0, Y_0)$.

Todas las métricas de enfoque se computan para cada imagen bidimensional (2D) del volumen de imágenes registradas a diferentes posiciones del objeto.

La estimación de profundidad, al igual que todos los aspectos de la visión por computador tienen algunos errores inherentes. Algunas fuentes de este error son el ruido en la imagen que conducen a imprecisión en los cálculos, ambigüedad en la interpretación de información de profundidad. Otros errores se deben a las aberraciones de lente de la cámara, etc., todos esos errores crean una incertidumbre en el resultado.

1.2 Diagrama de bloque del procesamiento 3D de imágenes de microscopía.



Tomas de muestras con los diferentes puntos de enfoque: En este punto se toman las muestras (fotos) necesarias con los diferente puntos de enfoque.

Obtención de parámetros básicos de ejecución: Se establecen los parámetros básicos como son el número de imágenes a procesar, el ancho y alto de estas mismas.

Lectura de la Imagen: Se hace la lectura de la imagen a la cual se le va hacer procesada

Se realiza el procesamiento de la imagen con una ventana de 5x5: Se realiza el procesamiento de la imagen en ventanas de 5x5 pixeles, lo cual quiere decir que se divide la imagen en bloques de 5x5 para hacer el respectivo procesamiento.

Se toman los mejores resultados: Una vez terminada de procesar la imagen se toman los mejores datos y son almacenados, los demás datos son desechados.

Hay más imágenes: En este punto se verifican si hay más imágenes para ser procesadas y repetir el proceso desde la lectura de imágenes o en caso contrario mostrar los resultados obtenidos del procesamiento.

Muestra de los mejores resultados: Para mostrar los resultados se muestra una gráfica 3D con los mejores valores obtenidos en el procesamiento de la imagen.

Capítulo 2

Introducción a la programación paralela.

2.1 Conceptos básicos.

La programación paralela es el uso de varios procesadores trabajando en conjunto para dar solución a una tarea en común, la programación paralela consiste en dividir el trabajo y que cada procesador realice una porción del problema, los datos se intercambian por una red de interconexión o a través de la memoria.

El rendimiento de los computadores tradicionales secuenciales se está saturando debido a que las aplicaciones de hoy en día necesitan realizar trabajos más complejos como son las simulaciones, predicción de tiempo, el análisis 3D de imágenes de microscopía, entre otras, para lo cual la solución es usar técnicas te programación paralela logrando así obtener mayor eficiencia, siempre y cuando los algoritmos estén diseñados de manera adecuada.

Los programas que implementan técnicas de programación paralela disminuyen los tiempos de ejecución, pero dicha disminución está limitada, la ley de Amdahl establece que cuando la fracción de un trabajo en serie es pequeña, y la denominamos s, la máxima aceleración alcanzable (incluso para un número infinito de procesadores) es sólo 1/s. Adicional a esto nos dice que en todo problema existe una parte secuencial, fs, en la que no es posible utilizar la potencia de los p procesadores de un sistema. De modo que la aceleración se ve limita como se muestra a continuación:

$$A = \frac{1}{(1 - f_m) + \frac{f_m}{a_m}}$$

Donde el factor de mejora está representado por a_m, y f_m es el tiempo de ejecución mejorado, el cual representa la parte secuencial. [4]

Por ejemplo, si en un programa de ordenador el tiempo de ejecución de un cierto algoritmo supone un 30% del tiempo de ejecución total del programa, y conseguimos hacer que este algoritmo se ejecute en la mitad de tiempo se tendrá:

[4] http://www.exa.unicen.edu.ar/catedras/arqui2/arqui2/filminas/Rendimiento%20de%20sistemas%20paralelos.pdf 18 febrero del 2015.

a_m= 2 f_m= 0.3 A = 1.18

Es decir que la velocidad de ejecución mejora en un factor 1.18, esta ley se mide en unidades genéricas, es decir los resultados no son porcentajes, ni unidades de tiempo. Por otra parte, la ley de Gustafson nos dice que cualquier problema suficientemente grande puede ser eficientemente paralelizado. Esta ley establece que:

$$S(P) = P - \alpha * (P - 1)$$

Donde P es el número de procesadores, S la aceleración y α la parte secuencia.

Por ejemplo suponemos que un programa consiste de α =0.67 partes fraccionales de un código serial, y 0.33 partes fraccionales de código paralelo. ¿Qué velocidad se espera para este programa cuando este corre a N=10 procesadores en paralelo?

$$S=10 - 0.67*(10-1) = 3.97$$

Esta ley está ligada a la ley de Amdahl, que establece que una mejora de rendimiento obtenida debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que utiliza dicho componente, ofreciendo una visión pesimista del procesamiento paralelo, por el contrario la ley de Gustafson ofrece un nuevo punto de vista y así una visión positiva de las ventajas del procesamiento paralelo.

La taxonomía de Flynn es una clasificación de arquitecturas de computadores la cual las clasifica por el número de instrucciones concurrentes (control) y los flujos de datos disponibles en las arquitecturas. En la siguiente tabla podemos observar la clasificación:

	Una Instrucción	**Múltiples instrucciones**
Un dato	SISD	MISD
Múltiples datos	SIMD	MIMD

Tabla 1. Taxonomía de Flynn.

Una instrucción, un dato (SISD): Computador secuencial que no explota el paralelismo en las instrucciones ni en el flujo de datos. El funcionamiento se muestra en la siguiente imagen:

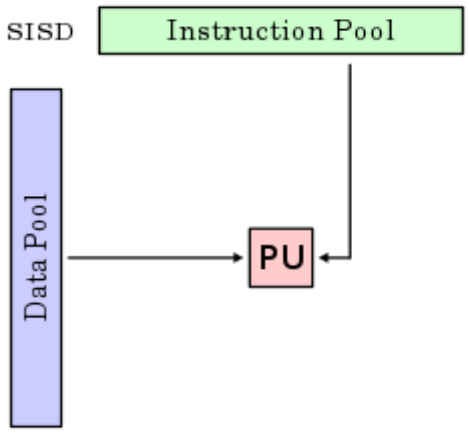


Figura 3. Imagen del comportamiento de SISD.

Múltiples instrucciones, un dato (MISD): Poco común debido al hecho de que la efectividad de los múltiples flujos de instrucciones suele precisar de múltiples flujos de datos. Sin embargo es usada en paralelismo redundante. El funcionamiento se muestra en la siguiente imagen:

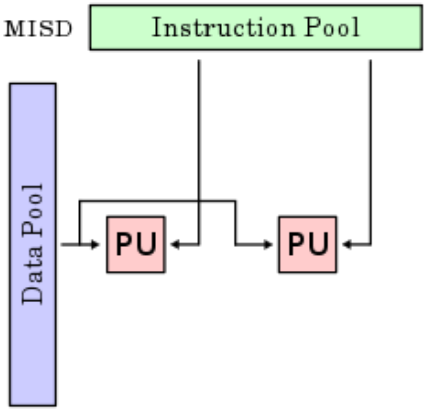


Figura 4. Imagen del comportamiento de MISD.

Una instrucción, múltiples datos (SIMD): Un computador que explota varios flujos de datos dentro de un único flujo de instrucciones, para realizar operaciones que pueden ser paralelizadas de manera natural. El funcionamiento se muestra en la siguiente imagen:

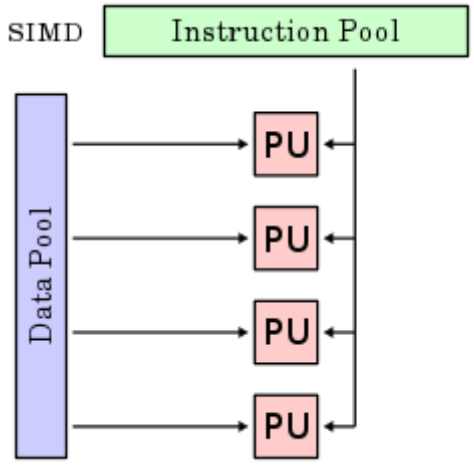


Figura 5. Imagen del comportamiento de SIMD.

Múltiples instrucciones, múltiples datos (MIMD): Varios procesadores autónomos que ejecutan simultáneamente instrucciones diferentes sobre datos diferentes. El funcionamiento se muestra en la siguiente imagen:

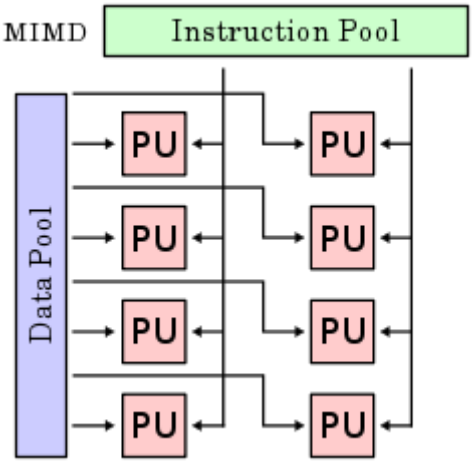


Figura 6. Imagen del comportamiento de MIMD.

2.2 Tipos de paralelismo

Las técnicas de programación paralela pueden aplicarse a un programa de diferentes maneras entre las cuales tenemos:

2.2.1 Paralelismo a nivel de bit

El paralelismo a nivel de bit se basa en el tamaño de la palabra que es capaz de manejar el procesador ya sea 8 bits, 16 bits, 32 bits o 64 bits, entre más grande sea el tamaño de la palabra menos instrucciones son llevadas a cabo por el procesador para realizar una operación determinada.

En otros tiempos entre 1970 y 1986 la aceleración en la arquitectura de computadores se lograba duplicando el tamaño de la palabra en la computadora, lo cual indica la cantidad de información que puede manejar por ciclo del reloj. Debido a esto, los microprocesadores de 4 bits fueron reemplazados por los microprocesadores de 8 bits, después estos fueron sustituidos por los de 16 bits, por los de 32 bits y por último los de 64 bits, el cual se ha convertido en un estándar en la computación de propósito general durante la última década.

2.2.2 Paralelismo a nivel de instrucción

Se ejecutan tantas instrucciones como se pueda, teniendo en cuenta que una no dependa de la otra, y de los procesadores con que se cuenta, uno de los objetivos de los compiladores es encontrar estas instrucciones para agilizar su ejecución.
Este tipo de paralelismo consiste en una técnica que busca combinar instrucciones de bajo nivel, este se encarga de ejecutar la instrucción en un núcleo del sistema de tal forma que al ser ejecutadas simultáneamente no afecten el resultado final del software, y conseguir más velocidad y el máximo aprovechamiento del hardware. En la siguiente imagen se puede observar el funcionamiento de este paralelismo.

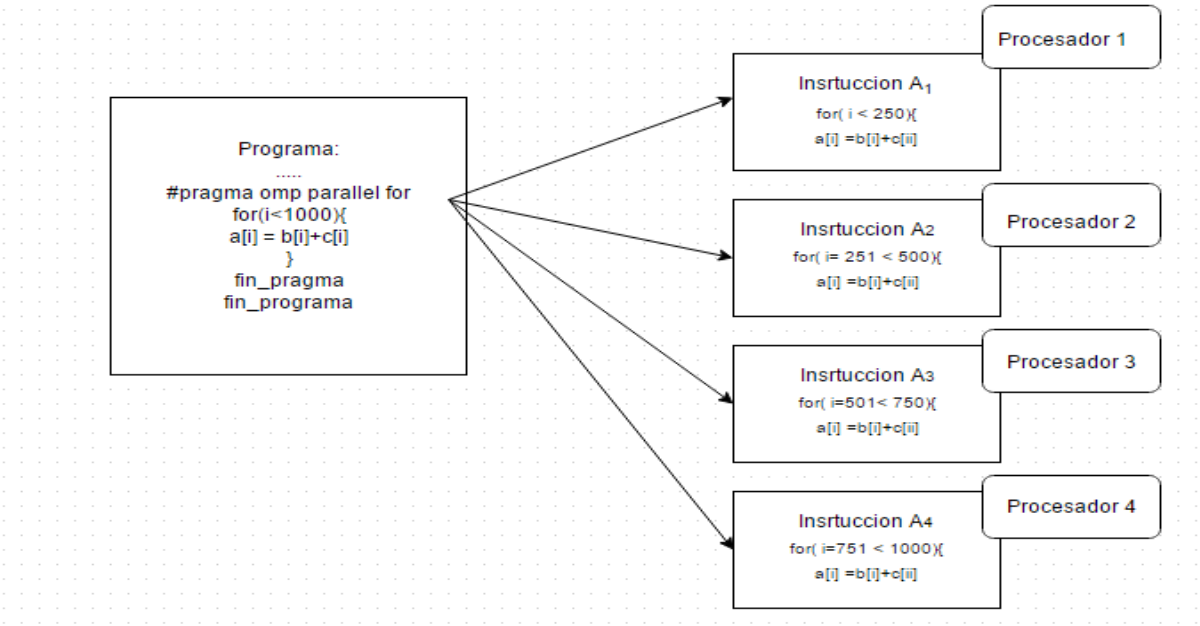


Figura 7. Representación del paralelismo a nivel de instrucción.

Como observamos en el pseudocódigo la instrucción #pragma omp parallel for se encarga de informar al compilador que la sección que viene a continuación es la parte a paralelizar. Este a su vez se encarga de enviar a cada procesador disponible la instrucción para ser ejecutada. Cuando esta es enviada a cada uno de los procesadores, se divide en secciones iguales al número de procesadores disponibles y son llevadas a cabo por cada procesador.

Entre las principales limitaciones del paralelismo a nivel de instrucción tenemos:

- **Estructurales:** El hardware no tiene la capacidad de soportar las técnicas de programación paralela, como son los computadores que tienen un único procesador.

- **De control:** Se presentan problemas para establecer la instrucción correcta que se debe ejecutarse después de cada instrucción, ya que estos son originados a partir de instrucciones de salto condicional que se encargan de determinar la secuencia de las instrucciones, por ejemplo, si una instrucción A es dependiente del control de una instrucción B, el resultado de B determina si se va a ejecutar la instrucción A.

- **De datos:** Al ejecutar las instrucciones de la aplicación, los datos de estas no deben depender unos de otros, por ejemplo, si alguno de los operando fuente o de lectura de una instrucción que llamaremos B, es el operando

destino o de escritura de una instrucción anterior que llamaremos A, entonces B no puede comenzar su ejecución hasta que A no haya finalizado.

De las características importantes que debemos resaltar del paralelismo a nivel de instrucción son:

- El programador puede conocer las unidades lógicas de su programa y de esta manera darle un tratamiento especial.
- Relativamente tiene un bajo costo de implementación, requiere pequeñas sentencias de código dentro de las unidades funcionales.
- Si se utiliza con pipelining[5] puede doblar, triplicar o cuadruplicar su efectividad.

2.2.3 Paralelismo de datos

El fuerte del paralelismo a nivel de datos proviene de la constatación de que algunas aplicaciones actúan sobre estructuras de datos como son vectores, matrices, entre otras, repitiendo cálculos sobre los elementos de estas estructuras. La idea es explotar la regularidad de datos, para esto se realizan cálculos sobre distintos datos.

En el paralelismo de datos se asocian directamente los datos a los núcleos de los computadores. Dado que los cálculos se ejecutan en paralelo sobre procesadores idénticos, es posible centralizar el control. Teniendo en cuenta que los datos son similares, la operación paralelizada toma el mismo tiempo sobre cada uno de los núcleos del computador, y el controlador puede enviar de manera síncrona la operación a ejecutar a todos los núcleos. En la siguiente imagen podemos observar el comportamiento del paralelismo a nivel de instrucción.

[5] Pipelining o segmentación: se conoce como segmentación al acto y consecuencia de dividir o formar porciones.

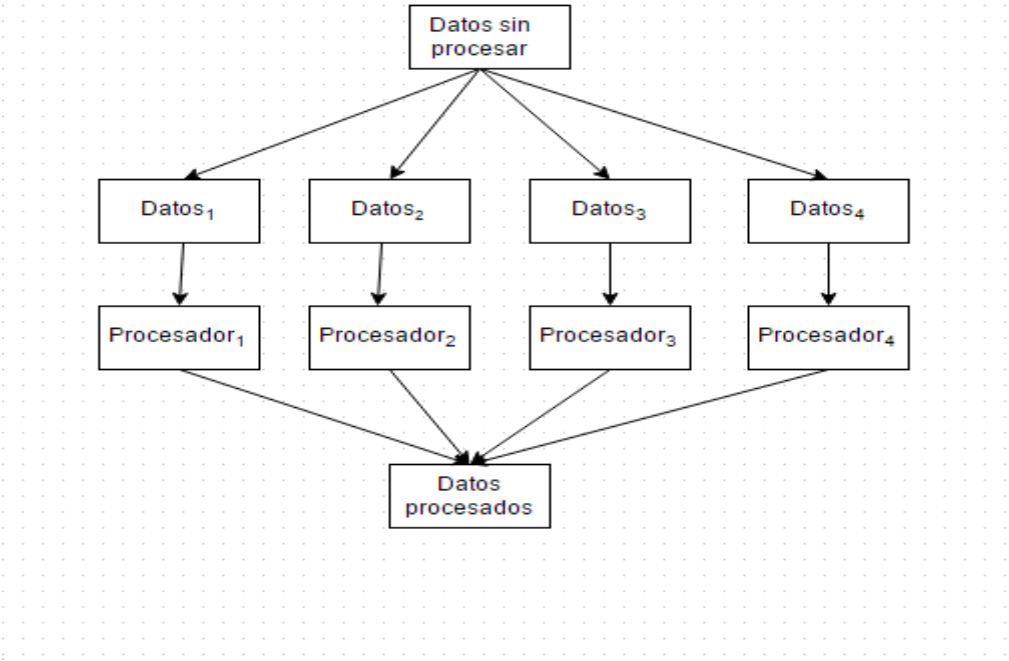


Figura 8. Representación del paralelismo a nivel de datos.

Las principales limitaciones que se encuentran en el paralelismo de datos son:

- Manejo de vectores: Dado el caso que el tamaño de la memoria de la máquina sea menor que el tamaño de los datos vectoriales, hay que dividirlos, teniendo así que sumar un tiempo extra de asignación y gestión de tareas.
- Manejo de escalares: Las aplicaciones que manipulan datos vectoriales usan frecuentemente datos escalares[6], y cuando se procesan trabajan un único núcleo y el resto permanecen en ocio.
- Operaciones de difusión[7] y reducción[8]: Las operaciones de comunicación grupales sobre datos vectoriales normalmente no son realizadas en paralelo, pero en algunos casos depende más de la topología de la plataforma.

Algunas de las características importantes que se deben considerar cuando se desea explotar este tipo de paralelismo son las siguientes:

[6] Datos escalares: un dato escalar es un determinado valor que no varía a lo largo del programa, una variable, o un campo, que solamente puede almacenar un valores en un cierto momento.

[7] Difusión: método que se encarga de enviar dato o datos a los procesadores.

[8] Reducción: método que se encarga de combinar varios datos en un solo procesador.

- Distribución de datos: Debido a que los datos son normalmente numerosos, mucho mayores al número de núcleos del computador, esto obliga a repartir los datos entre los diferentes núcleos disponibles. La homogeneidad de la estructura de los datos permite distribuirlos de manera regular.
- Ejecución síncrona: este tipo de paralelismo es clásicamente usado en máquinas tipo SIMD (una instrucción, múltiples datos) con control centralizado, por lo que se puede ver como un flujo de control sobre múltiples datos. Para este tipo en particular la sincronización es de manera automática al existir un solo controlador.

Para el desarrollo del software se utilizará el paralelismo a nivel de instrucciones ya que este es el modelo que se apega a nuestra librería de OpenMP, además con esto podemos aprovechar el beneficio de ejecutar tantas operaciones como nos sea posible para lograr agilizar nuestro trabajo, ya que el software de procesamiento 3D de imágenes de microscopía convierte la imagen en una matriz y realiza diferentes operaciones en cada una de las posiciones de dicha matriz y ninguna de estas operaciones tiene dependencia, podemos utilizar todos los procesadores que tengamos disponibles y sacar el máximo beneficio del paralelismo a nivel de instrucciones.

2.2.4 Paralelismo a nivel de tareas

El paralelismo a nivel de tareas es una técnica que se centra en la distribución de unidades de ejecución (hilos) en un conjunto de nodos diferentes que trabajan en paralelo. Es una técnica de paralelización que contrasta con el paralelismo a nivel de datos que consiste en la distribución del conjunto de datos, exclusivamente.

El paralelismo a nivel de tarea se alcanza cuando cada procesador ejecuta diferentes hilos sobre un conjunto de datos. Trabajen o no sobre el mismo conjunto de datos, los hijos se comunican entre sí mientras trabajan. La comunicación se lleva a cabo a través de un flujo de comunicaciones controlado y pre-establecido.

Si la idea es ejecutar un código en un sistema de dos procesadores que llamaremos A y B, y se pretende realizar dos tareas X y Y, es posible indicarle al procesador A que realice la ejecución de la tarea X y al procesador B que haga lo propio con la tarea Y de manera simultánea. Este nivel de paralelismo enfatiza la naturaleza paralela de la computación en contraste con al paralelismo a nivel de datos. En la siguiente imagen podemos observar el funcionamiento del paralelismo a nivel de tareas

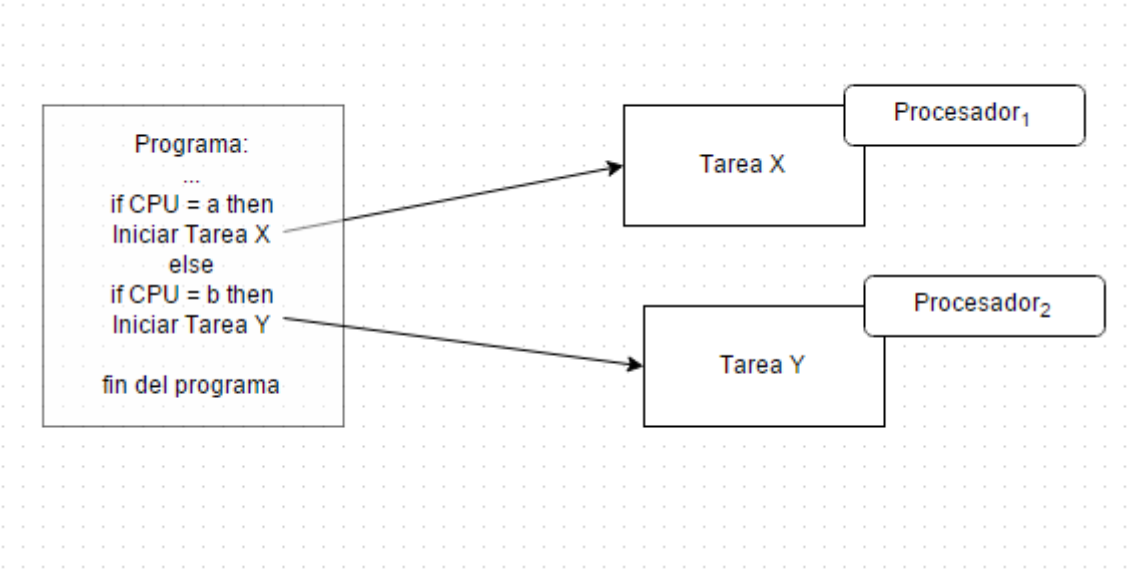


Figura 9. Representación del paralelismo a nivel de tareas.

2.3 Arquitecturas paralelas.

Las arquitecturas paralelas, son técnicas que descomponen los procesos secuenciales en subprocesos y éstos son ejecutados en segmentos especiales dedicados que operan en forma concurrente con los otros segmentos. Esta arquitectura también es conocida como líneas paralelas, cada línea puede considerarse como un conjunto de segmentos de procesamientos por el que fluye información binaria. Entre las arquitecturas paralelas encontramos:

2.3.1 Sistemas distribuidos

Los sistemas distribuidos son sistemas donde los componentes de hardware y de software, que se encuentran en el ordenador conectados a la red se comunican y coordinan sus acciones mediante paso de mensajes para cumplir su tarea. La comunicación de estos se lleva a cabo mediante un protocolo establecido por un esquema cliente-servidor.

Entre las principales características tenemos que son sistemas concurrentes, esta característica permite que todos los recursos que se encuentran disponibles en la red puedan ser utilizados simultáneamente por los usuarios que actúan en la red. Otra característica es que no tienen un reloj global para realizar la transferencia de mensajes entre los diferentes componentes, de esta tarea se encargan los mismos componentes, esto trae consigo otra característica, los fallos independientes de los componentes, esto nos garantiza que cuando un elemento falla los demás sigan funcionando hasta que nuestra tarea termine su proceso de ejecución.

Diseñar un sistema distribuido es crear aplicaciones de software que utilizando servicios y ayudándose de la conectividad, participen y se integren en este entorno de forma transparente a las plataformas de proceso y de almacenamiento de datos, dotándolas de los recursos necesarios para gestionar de forma integrada con el resto de sistema distribuido.

Las funciones básicas de un sistema distribuido son la capacidad de encontrar mecanismos para la asignación de tareas a los procesadores, que pueden estar dentro o fuera del equipo que está ejecutando el programa, además de este servicio también se encuentran los servicios de comunicación, sistema de ficheros, servicios de nombres, servicios de sincronización y coordinación, entre otros.

2.3.2 Arquitectura multinúcleo

Estas arquitecturas se caracterizan por tener más de un motor de procesamiento en un único chip, convirtiéndose en el método más habitual de aumentar las prestaciones de un dispositivo mientras se mantiene el consumo energético. Un microprocesador que combina dos o más procesadores independientes en un solo paquete es considerado multinúcleo.

Estos dispositivos permiten que el ordenador exhiba cierta forma de paralelismo en un nivel de hilos, al incluir múltiples microprocesadores en paquetes físicos separados, esto se conoce comúnmente como multiprocesamiento a nivel de chip, en general estos permiten que una computadora trabaje con multiprocesamiento, es decir procesamiento en simultáneo con dos o más procesadores. Por ejemplo, nos permite que aplicaciones de seguridad como son antivirus o anti espías estén ejecutándose al mismo tiempo que las aplicaciones que el usuario necesita para trabajar, sin causar un gran impacto en el rendimiento del sistema.

2.3.3 Clúster

Son sistemas independientes unidos entre sí, para formar un único sistema. Estos equipos manejan diferentes configuraciones entre las que tenemos alta disponibilidad, balanceo de carga, alto rendimiento y escalabilidad, siendo alta disponibilidad y alto rendimiento las configuraciones más utilizadas.

Esta tecnología ha evolucionado en apoyo de actividades, como son aplicaciones de supercomputo, software de misiones críticas, servidores web y comercio electrónico hasta base de datos de alto rendimiento, entre muchos otros usos. Esta tecnología surge como resultado de microprocesadores económicos de alto rendimiento, redes de altas velocidades, desarrollo de aplicaciones de alto desempeño, así como la creciente necesidad de mayor potencia computacional.

La construcción de un clúster es fácil y económico debido a su flexibilidad, pueden ser clúster homogéneos (cuentan con las mismas características de software y hardware) y clúster heterogéneo (tienen diferentes características tanto de software y hardware). La principal ventaja de esta tecnología es que adquirirla es bastante económico.

Un clúster hace uso de diferentes componentes para funcionar, entre estos tenemos:

- Nodos, estos son ordenadores de escritorio o servidores, estos pueden establecer un clúster con cualquier tipo de máquina.
- Sistema operativo, debe ser un entorno multiusuario, entre más fácil sea el manejo del sistema menos problemas se presentaran.
- Conexiones de red, estas pueden ser muy variadas, se pueden utilizar conexiones simples Ethernet con placas de red comunes o sistemas de alta velocidad como Fast Ethernet, Gigabit Ethernet, entre otros.
- Middleware, es el software encargado de interactuar con el sistema operativo y las aplicaciones brindando al usuario una única supercomputadora. Este software optimiza el sistema y provee herramientas de mantenimiento para procesos pesados como son migraciones, balanceo de cargas, tolerancia a fallos, entre otros.[9]

2.3.4 Grid

Esta tecnología permite utilizar de forma coordinada todo tipo de recurso entre los cuales tenemos cómputo, almacenamiento y aplicaciones específicas, que no están sujetas a un control centralizado. Este sentido es una nueva forma de computación distribuida, en la cual los recursos pueden ser heterogéneos y se encuentran conectados por redes de área extensa como es internet. Esta tecnología fue creada inicialmente para un ámbito científico pero el mercado comercial se interesó también en él.

El término grid se refiere a una infraestructura que permite la integración y el uso colectivo de ordenadores de alto rendimiento, redes y bases de datos que son propiedad y son administrados por diferentes instituciones. La colaboración entre las instituciones envuelve un intercambio de datos o de tiempo de computación, el propósito de esta tecnología grid es facilitar la integración de recursos computacionales. Las asociaciones para formar grid se dan entre universidades, laboratorios de investigación o entre algunas empresas.

[9] http://www.solingest.com/blog/cluster-de-servidores-que-es-y-como-funciona, Solingest, Revisado Mayo 23 del 2014

El objetivo de las grid es poder utilizar recursos remotos para permitir realizar las tareas que no se pueden realizar en nuestra máquina o centro de trabajo. La idea va más allá de intercambiar ficheros, se trata de tener acceso directo a software, ordenadores, datos remotos, entre otros, si no, también a las herramientas para analizarlos y la potencia de cálculo necesaria para utilizarlas.

El funcionamiento de las grid descansa en software middleware, que asegura la comunicación transparente entre diferentes dispositivos repartidos entre todo el mundo. La infraestructura grid integra un motor de búsqueda que se encarga de buscar no solo los datos, si no también busca las herramientas para analizarlos y la potencia necesaria para utilizarlas. Al final él se encarga de distribuir las tareas a cualquier lugar de la red en la que exista suficiente capacidad y devolverá los resultados al usuario.[10]

2.4 Lenguajes que soportan programación paralela.

Como la mayoría de cosas, los lenguajes de programación también van recibiendo mejoras a través del tiempo y de las necesidades que la tecnología va presentando. Los lenguajes de programación que dan soporte a la programación paralela cada día aumentan debido a sus grandes ventajas, enumerar todos y hablar acerca de cada uno sería casi imposible, por lo cual hablaremos de algunos de ellos.

Algunos de los principales lenguajes que soportan programación paralela

2.4.1 C++

C++ es un lenguaje de programación diseñado con la intención de extender al exitoso lenguaje de programación C, con mecanismos que permiten la manipulación de objetos. Poco a poco a través del tiempo fueron añadiendo más paradigmas lo cual le dio el nombre de lenguaje multiparadigma. Para realizar aplicaciones utilizando las técnicas de programación paralela, existen diferentes librerías entre las cuales tenemos OpenMP y MPI.

OpenMP está disponible en muchas arquitecturas, incluidas las plataformas de Unix y Microsoft Windows. La librería se compone de un conjunto de directivas, rutinas de biblioteca, y variables de entorno que ayudan en el cambio del tiempo de ejecución. Algunas de las ventajas que obtenemos al utilizar OpenMP son las siguientes:

[10] http://www.ramonmillan.com/tutoriales/gridcomputing.php#funcionamientocomputaciongrid, Ramón Jesús Millán Tejedor, Revisado el Mayo 26 del 2014.

- La descomposición de los datos es automática
- Admite la paralelización incremental del código
- Mismo código fuente para las versiones serial y paralela
- Elevada portabilidad

Para realizar el proceso de paralelización debemos saber que la sentencia paralela se establece con un **#pragma**, esta es la forma de informar al compilador que la parte que viene a continuación es la parte a paralelizar, es decir, quién traduce la sentencia de OpenMP es el propio compilador en tiempo de compilación. Esta sentencia está acompañada por otras palabras reservadas que podemos utilizar para mejorar su funcionamiento, entre las cuales tenemos:

- #pragma omp: Esta es la sentencia clave que identifica todas las directivas de OpenMP donde se establece la paralelización, esta sentencia es obligatoria.
- #pragma omp parallel: Parallel es una directiva que indicar a OpenMP cuándo inicializar la pila/team de hilos.
- #pragma omp for: Indica que el bucle for que sigue a la declaración del #pragma, debe ser paralelizado.

Adicional a esto existen unas funciones de gran importancia como son las siguientes:

```
OMP_GET_NUM_THREADS()        //Regresa el número actual de hilos.
OMP_GET_THREAD_NUM()         //Regresa el identificador de ese hilo.
OMP_SET_NUM_THREADS (n)     //Indica el número de hilo.
```

El estándar de OpenMP es definido y revisado por el comité OpenMP Architecture Review board que está compuesto actualmente por:

- U.S. DoD: Aeronautical Systems Center
- COMPunity (Comunidad de usuarios de OpenMP)
- Edinburgh Parallel Computing Centre (EPCC)
- Laboratorio NASA Ames
- ST Microelectronics: The Portland Group
- Fujitsu
- Hewlett Packard
- International Business Machines (IBM)
- Intel Corporation
- KAI Software Lab (KSL)
- NEC Corporation
- Silicon Graphics Inc. (SGI)
- Sun Microsystems
- Universidad RWTH Aachen

MPI es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en

programas que exploten la existencia de múltiples procesadores. Esta técnica es empleada para aportar sincronización entre procesos y permitir exclusión mutua, de manera similar a como se hace con semáforos, monitores, etc.

Su principal característica es que no precisa de memoria compartida, por lo que es muy importante en la programación de sistemas distribuidos, tiene tres elementos principales que intervienen en el paso de mensajes, estos procesos son, el proceso que envía, el que recibe y el mensaje. Algunas de las características de MPI son:

- Estandarización.
- Portabilidad.
- Buenas prestaciones.
- Amplia funcionalidad.
- Existencia de implementación libre.

Los procesos invocan diferentes funciones MPI que permiten realizar actividades como son iniciar, gestionar , finalizar procesos MPI, comunicar datos entre procesos, realizar operaciones de comunicación entre grupos de procesos y crear tipos arbitrarios de datos.

La estructura de un programa MPI se debe incluir:

- En el encabezado #include <mpi.h>
- Se debe inicializar y terminar MPI con las funciones MPI Init y MPI Finalice respectivamente.

Podemos observar en la siguiente imagen un ejemplo de cómo iniciar y finalizar con las funciones anteriormente nombradas utilizando los lenguajes C y C++

```
/*** C  **/
int MPI Init(int * pargv, char *** pargv)
int MPI Finalize(void)

/*** C++ **/
void MPI::Init(int & argv,char & ** argv)
void MPI::Init()
void MPI::Finalize()
```

Funciones básica de comunicación.

La forma de comunicación en MPI es a través de mensajes que contienen datos. La forma más simple es la comunicación es punto a punto, donde se envía un

mensaje de un proceso a otro, esto se realiza usando las funciones MPI Send y MPI Recv. En la siguiente imagen podemos observar la implementación de estas dos funciones.

```
void MPI::Intracomm::Send(void *buf, int count,MPI::Datatype dtype,int dest,int tag)

MPI::Status MPI::Intracomm::Recv(void *buf, int count,MPI Datatype dtype,int rc,int tag)
```

Adicional a las anteriores funciones básicas de comunicación tenemos funciones y argumentos que nos pueden servir de mucha utilidad a la hora de implementar programación paralela como son los siguientes:

```
void * buf          //Buffer de envío o recepción
int count           //número de datos
MPI Datatype dtype  //tipo de datos
int dest/src        //rango del nodo al que se envía/recibe
int tag             //etiqueta que diferencia al mensaje para ignorarla en MPI Recv se puede pasar MPI ANY TAG
MPI Comm comm       //comunicador (en C++ corresponde a this)
MPI Status * stat   //status de la recepción (solo MPI Recv) para ignorarlo pasar
MPI STATUS IGNORE   //(en C++ es devuelto por la funcion)
```

A continuación un ejemplo completo de un programa que implementa la librería MPI

```
#include <mpi.h>
#include <iostream>
using namespace std;

int main(int argc,char **argv){

MPI::Init(argc,argv);
const int size=MPI::COMM_WORLD.Get_size();
const int rank=MPI::COMM_WORLD.Get_rank();
cout<<"Hola mundo, este es el rango "<<rank<<" de "<<size<< endl;

if(rank==1){
double data=3.14;
MPI::COMM_WORLD.Send(&data,1,MPI::DOUBLE,0,27);
}
if(rank==0){
double data;
MPI::COMM_WORLD.Recv(&data,1,MPI::DOUBLE,1,MPI::ANY_TAG);
cout << "El rango 0 dice " << data << endl;
}
MPI::Finalize();
return 0;
}
```

En estos tipos de algoritmos se pueden presentar un deadlock[11], este ocurre cuando un proceso queda esperando un mensaje que nunca recibirá. Un caso típico se puede observar en el código siguiente, que debido al comportamiento ambiguo de MPI Send puede o no producir un deadlock.

```
if(rank==0) {
MPI_COMM_WORLD.Send(vec1,vecsize,MPI::DOUBLE,1,0);
MPI_COMM_WORLD.Recv(vec2,vecsize,MPI::DOUBLE,1,MPI::ANY_TAG);
}
if(rank==1) {
MPI_COMM_WORLD.Send(vec3,vecsize,MPI::DOUBLE,0,0);
MPI_COMM_WORLD.Recv(vec4,vecsize,MPI::DOUBLE,0,MPI::ANY_TAG);
}
```

[11] Deadlock: punto muerto o interbloqueo, bloqueo permanente de un conjunto de procesos o hilos de ejecución en un sistema concurrente.

2.4.2 FORTRAN

Es un lenguaje de programación de alto nivel, con propósito general, procedimental e imperativo, que está guiado especialmente a los cálculos numéricos y a la computación científica. Fortran es uno de los lenguajes más populares en el área de la computación de alto rendimiento y además es un lenguaje usado para evaluar el desempeño de las supercomputadoras más rápidos del mundo. Fortran tiene dos librerías para implementar la programación paralela, entre las cuales tenemos OpenMP y MPI.

Fortran ante problemas de grandes dimensiones de datos, puede obtener mejores rendimientos al fraccionar los datos, no exige reescritura de código y existe una extensión de OpenMP para realizar esta operación. Una segunda opción o alternativa que tenemos es la librería MPI, la cual es un esquema de paso de mensaje. Pueden realizarse simulaciones en cualquiera de las dos arquitecturas.

Fortran utiliza varios métodos para realizar la sincronización de hilos entre los cuales tenemos:

- !$OMP MASTER / !$OMP END MASTER: sólo el hilo 0 (el master) ejecuta el código.
- !$OMP CRITIAL / !$OMP END CRITICAL : asegura que sólo un hilo ejecuta una acción en concreto (escribir fichero, leer de teclado). El resto espera a que el anterior termine para poder ejecutar ese fragmento de código.
- !$OMP BARRIER : debe existir para todos los hilos o para ninguno. Todos esperan hasta que llegan a ese punto. Si sólo uno tiene la barrera, se produce un deadlock.
- !$OMP ATOMIC : asegura que sólo un hilo actualiza una variable compartida.

Estas funciones son accesibles en tiempo de ejecución en donde se controla y consulta desde el interior del programa al entorno de ejecución paralelo.

- OMP_set_num_threads: Fija el número de hilos que serán usados por las regiones paralelas en el código.
- OMP_get_num_threads: Obtiene el número de hilos que están ejecutándose en la región paralela desde la que es llamada.
- OMP_get_thread_num: Obtiene el identificador del hilo actual dentro de la región paralela.
- OMP_get_num_procs: Devuelve el número de procesadores disponible para el programa.
- OMP_set_dynamic: Activa el ajuste dinámico del número de hilos para

ejecutar regiones paralelas.

- OMP_get_dynamic: Obtiene el estado del mecanismo de ajuste dinámico de hilos.
- MP_set_nested: Activa o desactiva el anidado de regiones paralelas.
- OMP_get_nested: Obtiene el estado del mecanismo anterior.

Adicional a estas funciones existen otras para realizar bloqueos como son:

- OMP_init_lock: Inicializa una variable que será asociada con un bloqueo.
- OMP_set_lock: Cando un hilo llama a esta subrutina, adquiere la posesión del bloqueo si está libre. Si no lo está, espera a que lo esté.
- OMP_unset_lock(variable) : Libera la propiedad de un bloqueo.
- OMP_destry_lock(variable): Des-inicializa la variable asociada con el bloqueo.

Variables de entorno

Se puede controlar algunos aspectos de la ejecución de programas OpenMP asignando valores a determinadas como:

- export OMP_NUM_THREADS: Fija el valor de una variable de entorno.
- echo $OMP_NUM_THREADS: Muestra el valor de la variable de entorno.
- OMP_NUM_THREADS: Especifica el número de hilos que serán usados durante la ejecución de las regiones paralelas definidas dentro de un programa OpenMP.
- OMP_SCHEDULE: Afecta a la forma en que funciona la directiva.
- !$OMP DO y !$OMP PARALLEL DO. Especifica la forma en la que ha de repartirse el trabajo.
- OMP_DYNAMIC: en máquinas SMP, dónde diferentes programas se ejecutan simultáneamente, es posible ajustar el número de hilos de forma dinámica para aprovechar al máximo la máquina.
- OMP_NESTED: especifica el comportamiento del programa en caso de regiones paralelas anidadas.

A continuación un ejemplo sencillo de la implementación de OpenMP en fortran

```
 4 program Main
 5 use omp_lib
 6 implicit none
 7 integer(kind = OMP_lock_kind) :: lck
 8 integer(kind = OMP_integer_kind) :: ID
 9 call OMP_init_lock(lck)
10 !$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
11 ID = OMP_get_thread_num()
12 call OMP_set_lock(lck)
13 write(*,*) "My thread is ", ID
14 call OMP_unset_lock(lck)
15 !$OMP END PARALLEL
16 call OMP_destroy_lock(lck)
17 end program Main
```

2.4.3 JAVA

Java es un lenguaje de programación de propósito general, concurrente, orientado y basado en clases, diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Este lenguaje es considerado uno de los lenguajes más populares en uso, particularmente para programas de cliente-servidor de web.

Java Parallel Processing Framework es un framework para computación en grids para Java, enfocado en el buen rendimiento y facilidad de uso. El framework provee un conjunto de herramientas, extensibles y modificables, adicional a esto ofrece APIs para facilitar la paralelización de CPUs en aplicaciones de cálculo intensivo, y distribuir su ejecución sobre una red de nodos heterogéneos.

Algunas de las características son:

- Una API fácil de utilizar para crear tareas para su ejecución en paralelo.
- Un conjunto de APIs e interfaces de usuario para la administración y monitoreo de los servidores.
- Escalabilidad, hasta alcanzar un número arbitrario de procesadores.
- El framework es deployment-free: no requiere instalar el código de tu aplicación en un servidor, únicamente conectarse al servidor y cualquier cambio o actualización se carga automáticamente.
- Tratamiento de fallos y recuperación incluidos en todos los componentes del framework (clientes, servidores y nodos).
- APIs totalmente documentadas, guías de administración y desarrollo.

- Corre en cualquier plataforma que soporte Java 2 Plataform Standard Edition 5.0 (J2SE 1.5) o posterior.

Algunos detalles extras:

- Estado del desarrollo: Beta.
- Licencia: GNU General Public License (GPL), GNU Library or Lesser General Public License (LGPL).
- Sistema Operativo: Todo aquel que soporte la máquina virtual de Java.
- Interfaz de usuario: Java Swing.

Java también soporta la librería MPI, como podemos observar en el siguiente código:

```
 4 import mpi.* ;
 5 class Hello{
 6 static public void main(String [] args) throws MPIException{
 7                 MPI.Init( args ) ;
 8 int myrank= MPI.COMM_WORLD.Rank();
 9 if( myrank == 0 ){
10 char [] message = "Hello, there".toCharArray() ;
11 MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR,1, 99);
12       }
13                 else{
14    char [] message = new char[20];
15    MPI.COMM_WORLD.Recv(message, 0, 20, MPI.CHAR, 0,99);
16           System.out.println("received:" + new String (message)+ ":");
17 }
18 MPI.Finalize();
19     }
20 }
```

2.4.4 MATLAB

MatLab significa Matrix Laboratory, una definición clara de MatLab es que se trata de un lenguaje de alto nivel para cálculos numéricos y simbólicos. Matlab fue creado por Cleve Moler en los años ´70, quien fuese creador de las rutinas LINPACK y EISPACK, las cuales dieron origen a Matlab, el cual actualmente es distribuido por MathWorks, Inc. desde 1984.

Algunas características importantes sobre Matlab, es relevantemente lento en comparación con FORTRAN, ya que Matlab es un lenguaje de interpretación. Al programar con Matlab es más corta la escritura y existe una gran variedad de ToolBoxes que son simples de usar y entender. Las versiones desde donde Matlab dio soporte la programación paralela son:

MATLAB 7.6 → 2008

MATLAB 7.7 → 2008

MATLAB 7.8 → 2009

MATLAB 7.9 →2009

MATLAB 7.10→ 2010

MATLAB 7.11 → 2010

MATLAB 7.12 → 2011

MATLAB 7.13 → 2011

MATLAB 7.14 → 2012

MATLAB 7.15 → 2013

MATLAB 7.16 → 2014

Matlab cuenta con librerías para la implementación de programación paralela entre las cuales tenemos:

- Parallel Computing Toolbox
- MatLab Distributed Computing Server

La librería “Parallel Computing” de MatLab, nos permite desde un cliente de Matlab trabajar con varias sesiones de Matlab a la vez. “Parallel Computing Toolbox”, permite trabajar hasta con 4 “workers” en una máquina local, además de la sesión del cliente, aunque si es en un clúster donde se quieren tener los “workers”, se trabajará a través del software ”MatLab Distributed Computing Server”, que nos permitirá trabajar con todos aquellos “workers” que el clúster posea.

El equipo en donde el trabajo y sus tareas están definidos, se le conoce como sesión cliente. A menudo, “Parallel Computing Toolbox” utiliza la sesión de cliente para llevar a cabo la definición de puestos de trabajo y tareas. “MatLab Distributed Computing Server” realiza la ejecución de su trabajo mediante la evaluación de cada una de sus tareas, devolviendo el resultado a su sesión de cliente.

Parallel Virtual Machine: Es el más popular en el entorno académico, y estándar, probablemente debido al cuidado entorno de control de ejecución y otras prestaciones. Como características principales, sus autores destacan que es portable, soporta interoperación en conjunto heterogéneo de ordenadores, es escalable, permite modificación dinámica de la configuración, soporta grupos dinámicos de tareas, envío de señales, múltiples buffers de mensajes y trazado, y dispone de ganchos para soportar esquemas de tolerancia a fallos y modificar la política de despacho de tareas.

Message Passing Interface: El estándar MPI se concentra en definir la sintaxis y semántica de las rutinas de paso de mensajes que serían de utilidad a los usuarios de procesadores masivamente paralelos (MPP). El objetivo es proveer a los fabricantes con un conjunto claramente definido de rutinas que puedan implementar eficientemente en su arquitectura, basándose en soporte hardware específico. No define ningún entorno software para desarrollo de aplicaciones paralelas, manejo de tareas, configuración de E/S paralela. Algunas implementaciones del estándar MPI son LAM (Local Area MultiComputer), y MPICH (MPI Chameleon).

En un Intel iPSC de 128 nodos con 0.5MB de memoria distribuida cada uno, la conclusión resultó ser que se tardaba más tiempo en distribuir y re ensamblar los datos que en el propio cálculo, al menos para los tamaños de matriz que cabían en memoria del host. El siguiente intento reveló un problema más grave.

En el Ardent Titan (supercomputadora), con memoria compartida, se puso de manifiesto que una sesión típica de MATLAB emplea más tiempo en tareas no paralelizables (intérprete y visualización) que en paralelizables (cálculo). La conclusión es que sólo para tareas de alta granularidad merece la pena paralelizar el cálculo en el lugar obvio: el bucle externo. Realizar los cambios en MATLAB para manejar esto transparentemente no presenta interés comercial para MATLAB, ya que la inmensa mayoría de usuarios preferiría que los esfuerzos se dediquen a optimizar la versión monoprocesador.

En el siguiente código podemos observar la implementación de un programa en paralelo:

```
3 N = 100;
4 M = 200;
5 a = zeros(N, 1);
6 matlabpool open 2
7 tic;
8 parfor i = 1:N
9  a (i) = a(i) + max(eig(rand(M)));
10 end
11 toc;
12 matlabpool close
```

2.4.5 Otros lenguajes

C#: Este lenguaje a partir de su versión de framework 4 mejoró la compatibilidad para realizar programación paralela, proporcionando un runtime, nuevos tipos de biblioteca de clases y nuevas herramientas de diagnóstico, simplificando así el

desarrollo de aplicaciones paralelas, sin que el programador tenga que intervenir directamente con los subprocesos ni con el bloque de subprocesos.[12]

Python: Este lenguaje incorpora librerías para desarrollar aplicaciones paralelas como son OpenMP, MPI, CUDA y OpenCL, además este lenguaje cuenta con una extensión (Cython) que permite escribir funciones Python con variaciones y compilarlo.[13]

Axum: Este es un nuevo lenguaje de programación lanzado por Microsoft que está orientado al paradigma de la programación paralela y concurrente. Está basado en la arquitectura de la web y los principios de aislamiento, para incrementar la seguridad de las aplicaciones, capacidad de respuesta, escalabilidad y productividad. Su sintaxis es muy similar a C, ligando C# e inspirado en ADA.

Ahora que conocemos las librerías y los principales lenguajes de programación que soportan el procesamiento en paralelo podemos determinar el lenguaje que se adapta a la solución de nuestro problema.

En nuestro caso hemos decidido usar C++ por sus múltiples librerías y portabilidad a los diferentes sistemas operativos, existe una muy buena librería como es OpenMP que ésta compuesta por un conjunto de directivas para especificar las regiones de código a paralelizar, esto hace que sea fácil de implementar, también nos ayuda a tener un mejor provecho en la asignación de hilos es a nivel de loop (ciclos) paralelos y regiones paralelas (las que se especifica con el #pragma).

Además C++ nos brinda varios beneficios en cuento a librerías para realizar procesamiento 3D de imágenes, que para nuestro caso es primordial debido a que el software que vamos a desarrollar hace un procesamiento 3D de un conjunto de imágenes de microscopía. Otro beneficio que obtenemos al utilizar este lenguaje es su licencia ya que es de uso libre, y que es un lenguaje compilado y no ejecutado, ganando así algo de tiempo en ejecución.

2.5 Generalidades del proceso de división del trabajo en varias tareas

El proceso de dividir el trabajo en varias tareas se realiza la mayoría de las veces de manera automática en tiempo de ejecución, está acción de dividir la tarea es

[12] http://msdn.microsoft.com/es-es/library/dd460693(v=vs.110).aspx, Microsoft, Revisado Mayo 10 del 2014

[13] http://2013.es.pycon.org/media/programacion-paralela.pdf, Pedro Varo Herero, Revisado Mayo 8 del 2014

realiza de manera automática por librerías como OpenMP, o pueden ser establecidas por el programador de manera manual.

El comportamiento de programa funcionando donde el compilador se encarga de dividir el trabajo en el número de procesadores disponibles suponiendo que el número de procesadores disponibles es 4, entonces el compilador dividirá y ejecutará la aplicación de la siguiente manera.

```
//thread  0
for (int i = 0; i< 250; i++){
a[i] = b[i] + c[i]; }
//thread 1
for (int i = 250; i< 500; i++){
a[i] = b[i] + c[i]; }
//thread 2
for (int i = 500; i< 750; i++){
a[i] = b[i] + c[i]; }
//thread 3
for (int i = 750; i< 1000; i++){
a[i] = b[i] + c[i]; }
```

Figura 10. Comportamiento de un ciclo paralelizado.

Capítulo 3

Programación paralela versus programación secuencial

Existen muchos problemas computacionales que requieren de un importante número de recursos donde la programación paralela cumple un papel importante, esta puede dividir el problema en N partes que pueden ser ejecutadas en los diferentes procesadores que se encuentren disponibles, obteniendo así un máximo rendimiento.

En la programación secuencial, hay un único procesador que ejecuta instrucciones de programas de manera secuencial (Paso a Paso). Algunas operaciones, sin embargo, tienen múltiples pasos que no tienen dependencias de tiempo y por lo tanto se pueden separar en múltiples tareas y ser ejecutadas simultáneamente. Por ejemplo, la adición de un número a todos los elementos de una matriz, no requiere del resultado de ninguno de los demás elementos. Los elementos de la matriz puede ser puesto a disposición de varios procesadores, y realizar las sumas simultáneamente, con estos se tienen los resultados disponibles más rápido que si todas las operaciones se hubieran realizado en serie.

La primera limitante de la programación secuencial es la velocidad de ejecución del procesador, para obtener mejores resultados los procesadores tendrían un costo bastante elevado, adicional a eso su creación es bastante compleja, por esto los creadores de hardware decidieron crear varios procesadores en vez de un único procesador con gran desempeño.

Las arquitecturas informáticas actuales están confiando cada vez más en el paralelismo a nivel de hardware para mejorar el rendimiento. Sin embargo esta tecnología también tiene sus ventajas y desventajas.

3.1 Ventajas y desventajas de la programación paralela

3.1.1 Ventajas.

- Velocidad, una de las grandes ventajas de utilizar la programación paralela es que esta mejora el tiempo de ejecución de los programas.
- Mayor rendimiento, al usar la programación paralela se utilizan todos los recursos disponibles aumentando así el rendimiento.
- Solución de problemas concurrentes, algunos problemas como simulaciones, sistemas de control entre otros, son más fáciles de abordar con la programación paralela.
- Uso de recursos no locales.
- Ahorro de dinero.

3.1.2 Desventajas.

- Difícil implementación, la programación paralela es desarrollada por pocos programadores.
- No todos los problemas se pueden paralizar.
- No contiene modelos establecidos, estos depende de las características de los sistemas en paralelo, el hardware, entre otros factores.

3.2 Ventajas y desventajas de la programación secuencial.

3.2.1 Ventajas.

- Diseño descendente, el problema se descompone en etapas o estructuras jerárquicas.
- Fáciles de implementar, lo que reduce tiempo en el desarrollo.
- Reducción de complejidad en las pruebas.
- Facilidad de mantenimiento.

3.2.1 Desventajas.

- Velocidad de transmisión, la velocidad de una computadora serial depende directamente de la cantidad de datos que pueden moverse a través del hardware.
- Límites a la miniaturización, la tecnología de los procesadores está permitiendo que un número creciente de transistores puedan ser colocado en un chip. Sim embargo se tiene un límite a nivel de componentes moleculares.
- Limitaciones económicas, cada vez es más caro fabricar un procesador más rápido. El uso de un mayor número de procesadores moderadamente rápidos para conseguir el mismo o mejor rendimiento es menos costoso.

3.3 Pruebas realizadas.

Para observar las ventajas de utilizar programación paralela se realizaron varias pruebas, cada una de estas pruebas están detalladas a continuación.

3.3.1 Primera prueba: Comparación de la programación secuencial versus programación paralela.

Esta primera prueba consiste en medir el tiempo de ejecución de dos aplicaciones desarrolladas con dos técnicas de programación (Programación secuencial y Programación paralela), con el fin de medir la eficacia de estas técnicas a la hora de resolver un problema.

La aplicación construida con las dos técnicas de programación se encarga de contar el número de letras que contiene un texto, el cual está en escrito en un documento con formato ".txt", para la realización de las pruebas se tomaron en cuenta las vocales como las letras que deberían ser contadas en el texto. Se realizaron 5 tomas de tiempo por cada letra y se promediaron para tomar el tiempo de ejecución de cada técnica de programación. Adicional a esto se tomó la precaución de borrar la caché antes de cada una de las ejecuciones realizadas, para esto utilizamos la línea de comando el cual debe ser ejecutado como usuario root: *sync && echo 3 > /proc/sys/vm/drop_caches*

Tiempo(segundos) de ejecución con programación secuencial	**Tiempo(segundos) de ejecución con programación paralela (2 núcleos)**	**Tiempo(segundos) de ejecución con programación paralela (4 núcleos)**	**Vocal utilizada para la búsqueda**
2.276	1,256	1.156	A
2.357	1,458	1.201	E
2.500	1,254	1.075	I
2.380	1,548	1.302	O
2.488	1,354	1.128	U

Tabla 2. Resultados de la primera prueba.

En la tabla podemos observar los tiempos de ejecución en cada una de las pruebas realizadas utilizando las dos técnicas de programación, adicional a esto se utilizaron diferentes números de procesadores para realizar las pruebas utilizando programación paralela, como observamos en la tabla los tiempos de ejecución son mucho más bajos cuando utilizamos programación paralela y aumentamos el número de procesadores, que cuando utilizamos programación secuencial y menos números de procesadores, de esta manera podemos concluir que al utilizar técnicas de programación paralela podemos ahorrarnos tiempo de ejecución en las aplicaciones, mientras estás cumplan los parámetros adecuados.

Código de las aplicaciones.

Código utilizando programación secuencial.

```
#include <iostream>
#include <fstream>
using namespace std;


int main () {
     fstream ficheroEntrada;
     char letra,letra2;
     int sum=0;

     ficheroEntrada.open ("quijote.txt", ios::in);
     if (ficheroEntrada.is_open()) {
     while (! ficheroEntrada.eof() ) {
     ficheroEntrada >> letra;
          if(letra == 'u'){
          sum++;
         }
       }
     ficheroEntrada.close();
     }
     else cout << "Fichero inexistente" << endl;

     cout <<" Hay  "<<sum<<" "<<"a, en el texto";
     return 0;
}
```

Código utilizando programación paralela.

```
#include <iostream>
#include <fstream>
#include <omp.h>

using namespace std;
int main () {
    fstream ficheroEntrada;
    char letra,letra2;
    int sum=0;

 ficheroEntrada.open ("quijote.txt", ios::in);
 if (ficheroEntrada.is_open()) {
    #pragma omp parallel
    {
        while (! ficheroEntrada.eof() ) {
        ficheroEntrada >> letra;
            if(letra == 'u'){
                sum++;
            }
        }
     }
    ficheroEntrada.close();
}
else cout << "Fichero inexistente" << endl;
cout <<" Hay  "<<sum<<" "<<"en el texto"<<endl;
return 0;
}
```

3.3.2 Segunda prueba: Comparación de dos lenguajes de programación que soportan programación paralela(C++, Matlab).

Para realizar esta prueba se desarrollaron dos aplicaciones para el procesamiento 3D de imágenes de microscopia con dos lenguajes de programación C++ y Matlab, en ambas aplicaciones se utilizaron técnicas de programación paralela. Para comprobar la ejecución de las aplicaciones se le realizó el proceso a la siguiente imagen

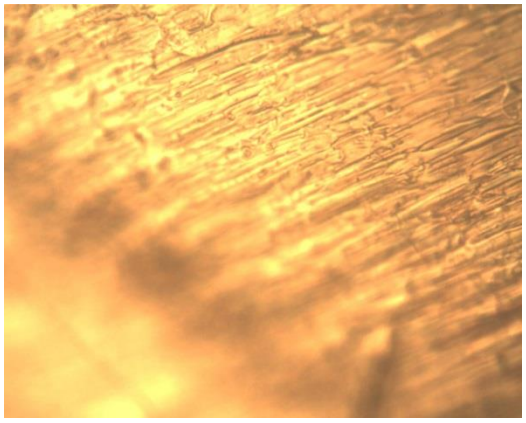


Figura 11. Imagen original a la cual se le aplico el procesamiento.

En la cual podemos observar que está contiene una parte enfocada y otra desenfocada, una vez realizado el proceso con ambas aplicaciones obtuvimos las siguientes imágenes

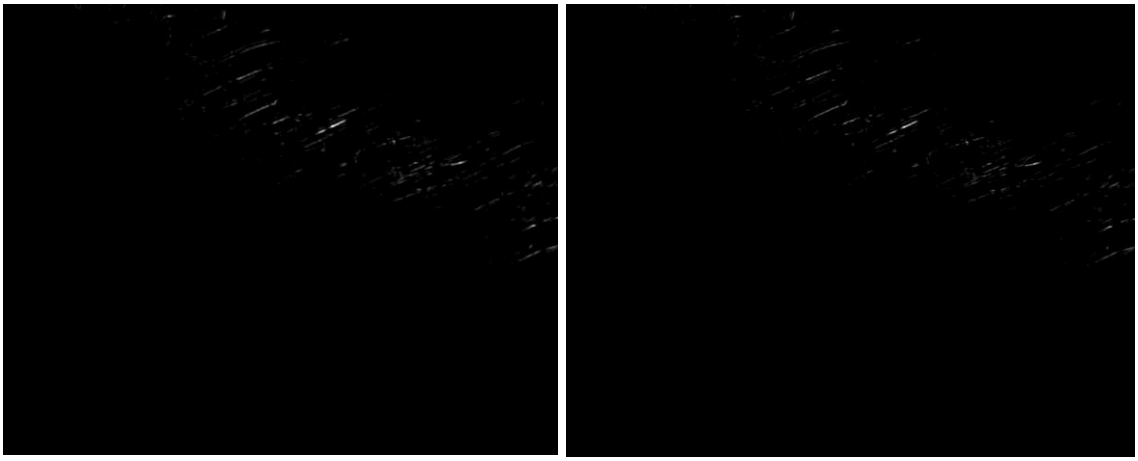


Figura 12. Imagen de resultados del procesamiento con Matlab(izquierda) y C++(derecha)

Como se puede observar en ambas imágenes se muestran la parte enfocada de la imagen original, con lo cual podemos comprobar que ambas aplicaciones están

ejecutando el procesamiento de manera correcta. Para realizar las pruebas se utilizaron máquinas virtuales con las siguientes características:

Ubuntu 14.04 LTS x64	Windows 7 Ultimate x64
80 GB de disco duro	80 GB de disco duro
10240 MB de memoria RAM	10240 MB de memoria RAM
Procesadores de 2.0GHz	Procesadores de 2.0GHz

Tabla 3. Características de las máquinas virtuales.

Las pruebas se realizaron con diferentes números de procesadores para demostrar la eficacia de cada lenguaje a la hora de implementar la programación paralela y para observar el comportamiento de las aplicaciones cuando se usa un mayor número de procesadores. En la siguiente tabla podemos observar estos resultados:

Número de Procesadores	**C++**	**Matlab 2013b**
12	3m 33 s	185m 42s
12	3m 33s	185 m 39s
12	3m 30s	185m 40s
10	3m 41s	222m 23s
10	3m 49s	222m 31s
10	3m 45s	222m 20s
8	4m24s	248m 21s
8	4m 15s	248m 38s
8	4m30s	248m 19s
6	5m4s	344m 43s
6	5m0s	344m 35s
6	5m8s	344m 39s
4	6m 20s	501m 25s
4	6m22s	501m 32s
4	6m13s	501m 29s
2	9m22s	953m 55s
2	9m15s	954m 01s
2	9m22s	953m 49s

Tabla 4. Resultados de la segunda prueba.

De los datos de la tabla se puede concluir que el código realizado en el lenguaje de C++ es más rápido que el realizado en Matlab, esto se debe a que Matlab debe realizar varias procesos para poder llevar al cabo el proceso de paralelización, entre estos procesos se encuentra el montaje de un clúster de manera local, adicional a esto Matlab es ejecutado en una máquina virtual que se encarga de traducir el código a lenguaje maquina lo cual también genera un tiempo extra a la hora ejecutar el código y convierte a Matlab en un lenguaje interpretado. A diferencia de Matlab, C++ no tiene que hacer ninguna de estas actividades para ejecutar el programa en paralelo.

Otra conclusión que podemos sacar es que entre mayor sea el número de procesadores utilizados para ejecutar el programa la diferencia con respecto al tiempo va disminuyendo considerablemente. Como observamos en la tabla la diferencia entre usar 2 procesadores y 4 procesadores son 3 minutos aproximadamente, pero si observamos la diferencia entre usar 12 procesadores y 10 procesadores apenas son de 10 segundos, la diferencia en ambos casos son solamente 2 procesadores pero respecto al tiempo los cambios son muy grandes, esto se debe a cada problema tiene un límite de procesadores a utilizar, una vez pasado este limite el usar o no más procesadores no hará obtener mejores resultados con respecto al tiempo, como lo explican las leyes de Amdahl y Gustafson.

Capítulo 4

Guía para desarrollar aplicaciones implementando programación paralela.

4.1 Instalación y configuración del entorno de desarrollo para aplicaciones paralelas.

Para llevar a cabo la instalación y configuración del entorno de desarrollo para aplicaciones paralelas se deben realizar los siguientes pasos, primero debemos realizar la instalación de las librerías OpenMP y MPI, seguido de la instalación y configuración del entorno de desarrollo, para este caso utilizaremos el IDE QT, (existen otros IDE que podemos configurar para desarrollar nuestras aplicaciones como son Kdevelop, Code::Block, entre otros). Adicional a las dos librerías instalaremos una tercera librería llamada OpenCV para el manejo de imágenes, la instalación de esta librería es opcional.

4.1.1 Instalación OpenMP y MPI

Para realizar la instalación de las librerías de OpenMP y MPI primero actualizamos los paquetes del sistema e instalamos el gestor de paquetes Synaptic, para llevar a cabo esto debemos tecleamos desde la terminal lo siguiente:

```
~$ sudo apt-get update
~$ sudo apt-get upgrade
~$ sudo apt-get install Synaptic
```

Una vez terminada la instalación del gestor de paquetes Synaptic, abrimos el gestor, marcamos e instalamos los siguientes paquetes:

- *binutils*
- *automake*
- *autoconf*
- *g++*
- *gcc*
- *m4*
- *libtool*
- *konsole*
- *MPI:* libopenmpi1*.6-db*

Para finalizar pasamos a instalar los siguientes paquetes adicionales requeridos para la instalación de las librerías OpenMP y MPI, por lo cual debemos instalarlos, una vez terminado de instalar estos paquetes continuamos instalando los paquetes de OpenMP, MPI y ssh desde la terminal, para ello debemos teclear los siguientes comandos:

```
~$ sudo apt-get install libopenmpi-dev openmpi-bin openmpi-doc
~$ sudo apt-get install ssh
```

```
~$ ssh-keygen -t dsa
~$ cd ~/.ssh
~$ cat id_dsa.pub
```

Una vez realizados los pasos anteriores podemos dar por terminado el proceso de instalación de las librerías OpenMP y MPI.

4.1.2 Instalación del IDE QT

Terminada la instalación de las librerías pasamos a instalar el entorno de desarrollo QT, este se puede instalar de dos maneras diferentes, como primera opción podemos dirigimos al centro de software de Ubuntu, en el buscamos el entorno de desarrollo QT creator y lo instalamos, o bien si queremos tener la última versión del entorno seguimos los siguientes pasos:

1. Descargamos el instalador de QT desde la terminal con el siguiente comando:

```
~$ wget http://download.qt-project.org/official_releases/qt/5.0/5.0.2/qt-linux-opensource-5.0.2-x86-offline.run
```

2. Asignamos permisos al archivo que acabamos de descargar

```
~$ chmod +x qt-linux-opensource-5.0.2-x86-offline.run
```

3. Ejecutamos el archivo descargado

```
~$ ./qt-linux-opensource-5.0.2-x86-offline.run
```

Si existen problema con la instalación escribir las siguientes líneas:

```
~$ sudo apt-add-repository ppa:canonical-qt5-edgers/qt5-proper
~$ sudo apt-get update
~$ sudo apt-get install qt-sdk
```

4. Una vez instalado el entorno de desarrollo QT, pasaremos a comprobar las instalaciones realizadas para esto debemos crear un proyecto nuevo, para este caso creamos un proyecto de tipo Plain C++ Project, una vez creado lo configuramos para compilar la librería de OpenMP modificando el archivo .pro que se encuentra dentro del proyecto, abrimos el documento y al final le adicionamos las siguientes líneas:

```
QMAKE_LFLAGS += -fopenmp
QMAKE_CXXFLAGS += -fopenmp
LIBS += -fopenmp
```

Una vez terminado estos pasos podemos realizar las pruebas respectivas para probar nuestra instalación. Para ello modificamos el archivo main.cpp del proyecto agregando las siguientes líneas de código:

```
#include <iostream>
#include <omp.h>
using namespace std;

#define N 12
int i, tid, nth,A[N];

int main(){
    for (i=0; i<N; i++) A[i]=0;
    #pragma omp parallel private(tid) shared (A)
    {
        nth=omp_get_num_threads();
        tid= omp_get_thread_num();
        cout << "Hilo" <<tid <<" de " <<nth<<endl;
        A[tid] = 10 +tid;
        cout <<"El hilo " <<tid<<" ha terminado"<<endl;
    }
    for(i=0; i<N; i++) cout <<"A["<<i<<"]="<<A[i]<< endl;


}
```

Ahora debemos compilar y ejecutar el código para observar que todo se encuentre bien instalado. Si por alguna razón no funciona verificar los pasos anteriores.

4.1.3 Instalación de OpenCV

OpenCV es una librería de libre uso tanto comercial como académica desarrollada por Intel, soportada por varios lenguajes como C, C++, Python y Java, se ha utilizado en infinidad de aplicaciones. Desde sistemas de seguridad con detección de movimiento, hasta aplicativos de control de procesos donde se requiere reconocimiento de objetos.

Para realizar la instalación de la librería de OpenCV debemos seguir estos pasos:

1. Realizamos la instalación de paquetes adicionales requerido para la instalación de la librería OpenCV, para ello tecleamos desde la terminal:

```
~$ sudo apt-get update
```

```
~$ sudo apt-get install build-essential cmake libv4l-dev pkg-config
```

```
~$ sudo apt-get instal libgtk2.0-dev libtiff4-dev libjasper-dev
libavformat-dev libswscale-dev libavcodec-dev libjpeg-dev libpng-dev
```

2. Ahora pasamos a descargar la librería OpenCV para Linux, para esto tecleamos lo siguiente en la terminal:

```
~$ cd Descargas (Se puede reemplazar la carpeta de Descargas)
~$ wget https://github.com/Itseez/opencv/archive/2.4.8.tar.gz
```

3. Asignamos los permisos y descomprimimos el archivo descargado utilizando los siguientes comandos:

```
~$ tar -xvf opencv-2.4.8.tar.gz
~$ cd opencv-2.4.8/
```

4. Creamos una carpeta con el nombre release y nos ubicamos en ella, para esto tecleamos lo siguiente en la terminal:

```
~$ mkdir release
~$ cd release
```

5. Ahora debemos generar el makefile, para esto tecleamos lo siguiente:

```
~$ cmake -D CMAKE_BUILD_TYPE=RELEASE -D
CMAKE_INSTALL_PREFIX=/usr/local ..
```

6. Compilamos e instalamos (este proceso podría tarde bastante), esto lo hacemos tecleando desde la terminal:

```
~$ make
~$ sudo make install
~$ sudo ldconfig
```

7. Para verificar que la librería se instaló tecleamos desde la terminal lo siguiente:

```
~$ pkg-config opencv –libs
```

8. Ahora pasamos a realizar la configuración del entorno de desarrollo QT, para esto modificamos el documento .pro y le adicionamos la siguiente línea:

```
LIBS += `pkg-config opencv –libs`
```

Después pasamos al archivo main.cpp e incluimos las librerías de OpenCV que son las siguientes:

```
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
```

Una vez terminado podemos dar por concluido la instalación y configuración del entorno de desarrollo para aplicaciones paralelas, con las librerías OpenMP y OpenCV.

4.2 Desarrollo de programas paralelos.

4.2.1 Metodología para desarrollar programas paralelos

Cuando vamos a desarrollar aplicaciones paralelas debemos tener en cuenta los siguientes aspectos:

- Definir el enfoque que mejor se adapte a nuestra arquitectura.
- Separar los tiempos de comunicación de entradas y salidas, los cálculos en la aplicación de ser esto posible.
- Definir las características del paralelismo si es escondido (está relacionado a los tipos de datos vectoriales y arreglos) o explícito (refleja el paralelismo que provee la máquina), cuál es su fuente de paralelismo, entre otros.

El diseño de programas paralelos no es fácil y requiere una buena creatividad, el algoritmo paralelo desarrollado puede ser muy diferente al desarrollado en su versión secuencial. Los pasos que se darán a continuación se repiten en el ciclo del diseño, existen cuatro pasos que se repiten en la metodología espiral, los cuales son descomposición, comunicación, agrupamiento y asignación. Los primeros dos pasos estudian el problema de la escalabilidad y concurrencia mientras los dos pasos restantes son los encargados de la localidad y mejoras en el rendimiento computacional. La flexibilidad de esta metodología radica en el permitir que las tareas puedan ser creadas o eliminas dinámicamente.

El mecanismo espiral nos permite realizar un diseño óptimo, basados en dos premisas de programas paralelos:

- Explotar el conocimiento que se adquiere durante el desarrollo.
- Diseñar inicialmente las partes claves de la aplicación.

Aprovechar la experiencia y conocimiento que ganamos sobre el hardware y software que utilizaremos durante el desarrollo de la aplicación nos permitirá

realizar un diseño final más eficiente, partiendo del hecho que al minimizar las restricciones posibles se pueden generar ideas iniciales de diseño que a su vez pueden generar un impacto fundamental en las características finales del código a generar.

Por otro lado el diseño incremental permite ir mejorando el código, además de considerar solo al inicio las partes importantes de nuestra aplicación. La idea es diseñar los módulos claves y una vez estos estén correctos las otras piezas de código pueden ir incorporándose lentamente.

Una vez explicado en que consiste el mecanismo espiral explicaremos los cuatros pasos anteriormente nombrados, los cuales son:

- **La descomposición:** es una fase que es consiste en vislumbrar las oportunidades de paralelismo al diseñar el mayor número de pequeñas tareas posibles con las cuales se podrá determinar el máximo paralelismo al que podemos llegar en nuestra aplicación.

- **La comunicación:** se puede diseñar en fases, para la primera fase tenemos que definir los diferentes enlaces entre cada una de nuestras tareas y después en la segunda fase podemos especificar cuáles son los mensajes a intercambiar entre ellos.

- **El agrupamiento:** pasamos a revisar las decisiones tomadas en los anteriores pasos de descomposición y comunicación para tener una ejecución óptima en nuestra máquina. Por lo general se determina si se debe agrupar las tareas identificadas en la fase de descomposición o si los datos deben ser replicados. Estas reducciones pueden llevar consigo la posibilidad de que exista una tarea por núcleo dependiendo el tipo de la plataforma que se trabaje.

- **En la asignación:** se especifica donde será ejecutada cada tarea, estas son asignadas a los núcleos de manera que logremos maximizar la utilización de los procesadores y minimizar los costos comunicacionales. Estos problemas no existen en máquinas con un solo núcleo o en computadoras de memoria compartida o distribuida con automática planificación de tareas.

Hay dos estrategias que podemos seguir a la hora de diseñar aplicaciones paralelas, ellas son diferenciadas por las consideraciones iniciales en las que están basadas, la primera es cuando desarrollamos una aplicación paralela desde cero y la otra es cuando llevamos a cabo la paralelización en un programa secuencial.

4.2.2 Desarrollando un programa paralelo desde cero.

En esta estrategia vamos a desarrollar una aplicación desde cero, partiendo de la definición del problema a resolver. Nos llegamos a preguntar ¿Es más difícil desarrollar una aplicación utilizando programación paralela que programación secuencial?, algunas veces sí y otras no, eso depende porque algunos lenguajes paralelos ganan en el poder expresivo para manipular grandes volúmenes de información logrando así la posibilidad de escribir código sin preocuparse de definir los lazos de ejecución ya que estos son hechos naturalmente en lenguajes paralelos.

Las decisiones que tomamos durante la planeación tienen gran impacto durante la implementación y en el futuro desempeño de la misma. Algunas de las cosas que podríamos llegar a preguntarnos son:

- ¿Con cuales lenguaje de programación cuento?
- ¿Qué tan complejo y grande es el problema?
- ¿Se puede garantizar la ejecución eficaz de la aplicación?
- ¿Cuánto esfuerzo es requerido para llegar a implementarla?
- ¿El diseño es escalable?

Es importante tener en cuenta que un diseño de algoritmo eficiente en una máquina A, puede llegar a tener malos rendimientos en una máquina B.

Para ir conociendo los pasos que debemos seguir para realizar un programa utilizando programación paralela desde cero, debemos desarrollar una aplicación que nos muestre en que procesador se está ejecutando el proceso, para ello lo que haremos es mostrar un mensaje que nos indique esta información.

Primero debemos recordar las funciones que nos van ayudar a realizar este programa como son las siguientes:

- Omp_get_num_threads()
- Omp_get_threads_num()

Estas funciones nos ayudarán a saber el número de hilos que tenemos, y saber desde que hilos estamos trabajando respectivamente. Para esto empezaremos

por crear una estructura base para realizar programas en C++, una vez terminado debería verse así:

```
#include <iostream>
#include <omp.h>
using namespace std;
int main(){


return 0;
}
```

Una vez creada la estructura de nuestro programa pasamos a indicar el proceso de paralelización, para esto haremos debemos agregar estas líneas dentro del main

```
#pragma omp parallel private(tid) shared (A)
{
}
```

Una vez terminado pasamos a realizar el código que se encargará de mostrar los mensajes, para esto debemos agregar las siguientes líneas dentro de la sentencia anterior:

```
nth=omp_get_num_threads();
tid= omp_get_thread_num();
cout << "Hilo" <<tid <<" de " <<nth<<endl;
```

Una vez agregadas estas líneas deberíamos tener algo como esto

```
#include <iostream>
#include <omp.h>
using namespace std;
int main(){

    #pragma omp parallel private(tid) shared (A)
    {
        nth=omp_get_num_threads();
        tid= omp_get_thread_num();
        cout << "Hilo" <<tid <<" de " <<nth<<endl;
    }

return 0;
}
```

Después de tener todo el código debemos realizar la compilación y ejecución del programa, esto lo podemos realizar desde la terminal de Linux o desde el IDE. Para realizarlo desde el IDE le damos clic al botón Build Project o utilizamos el acceso rápido control + B, una vez compilado pasamos a ejecutar el programa, para ello le daremos clic al botón run o utilizamos el acceso rápido control + R. Una vez ejecutado el código los resultados deberían verse parecido a la siguiente imagen.

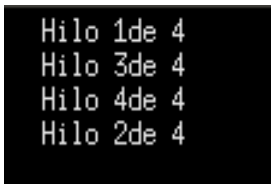


Figura 13. Imagen donde se muestran los resultados de la ejecución.

La otra forma de realizar este proceso es desde la terminal para esto debemos entrar en ella y posicionarnos en la carpeta donde se encuentra nuestro código, una vez estando ahí pasamos a teclear la siguiente línea:

g++ -o "nombre de salida" "nombre ejecutable.cpp" -fopenmp

Para este caso en particular, llamamos el archivo cpp codigo_paralelo.cpp y el ejecutable ejecutable_codigo, entonces la línea deberá quedar así:

g++ -o ejecutable_codigo codigo_paralelo.cpp -fopenmp

Si todo sale bien hasta este punto ya tenemos el programa compilado, ahora para ejecutar el programa tecleamos lo siguiente:

./ejecutable_codigo

Apenas tecleamos esto podremos observar en la terminal los resultados de la ejecución.

La sentencia *#pragma omp parallel* puede ir acompañada de otras palabras reservadas como son:

- schedule *(type [,chunk]):* Describe cómo las iteraciones del bucle se dividen entre los hilos del equipo. La programación predeterminada depende de la implementación.
- ordered: Específica que las iteraciones del bucle que deben ser ejecutados como si se estuviera utilizando programación secuencial.
- private *(list):* Los datos que se encuentran en la región paralela son privados para cada hilo, esto significa que cada hilo tendrá una copia local

que la usará como variable temporal. Una variable privada no es inicializada y tampoco se mantiene fuera de la región paralela. Por definición, en OpenMP el contador de iteraciones es privado.

- firstprivate *(list):* Cumple lo mismo que la directiva private pero se inicializa con el valor original.
- *shared *(list):* Los datos de la región paralela son compartidos, lo que significa que son visibles y accesibles por todos los hilos. Por definición, todas las variables que trabajan en la región paralela son compartidas excepto el contador de iteraciones.
- *reduction *(operator: list):* Lo que hará la librería será repartir el número total de iteraciones del bucle entre los distintos hilos.
- collapse *(n):* Especifica cuántos bucles en un bucle anidado deben ser colapsados en un solo espacio de iteraciones grande
- nowait: Si se especifica, entonces los hilos no se sincronizan en el extremo del bucle en paralelo.

En el siguiente código podemos observar en funcionamiento algunas de estas palabras:

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000

main ()
{

int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
  {

  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  }  /* end of parallel section */

}
```

Existen otras sentencias que nos ayudan a controlar mejor el proceso de paralelización con OpenMP entre las cuales tenemos:

- omp_get_max_threads() : Esta sentencia nos permite obtener el número de procesadores que tenemos disponibles para la ejecución del programa.
- omp_set_num_threads(nProcessors): Esta sentencia nos permite establecer el número de procesadores que utilizaremos en la ejecución de la aplicación, donde “nProcessors” corresponde a un número entero.
- #pragma omp sections: Con esta sentencia le indicamos al compilador que encontraremos secciones que deben ser paralelizadas. Estas secciones son indicadas por medio del siguiente sentencia #pragma omp section.

Para tener más clara la estructura de la paralelización por secciones tenemos el siguiente ejemplo:

```
#include <omp.h>
#define N     1000

main ()
{

int i;
float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }

#pragma omp parallel shared(a,b,c,d) private(i)
  {

  #pragma omp sections nowait
    {

    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

    }  /* end of sections */

  }  /* end of parallel section */

}
```

4.2.3 Paralelizando un programa secuencial

En esta estrategia partimos de una aplicación construida secuencialmente, la cual se desea paralelizar. Existen diversas maneras para llegar a cumplir nuestro objetivo, entre las cuales tenemos paralelización automática, librerías paralelas y recodificación de partes.

Paralelización automática: dependiendo de la máquina paralela, existen tareas que deben realizarse con el fin de obtener un código paralelo eficiente para la aplicación secuencial. Algunas de las principales tareas que debemos llevar a cabo son:

- Distribuir datos entre los diferentes núcleos
- Establecer referencias a memorias cercanas para hacer un eficiente uso de la jerarquía de nuestra memoria.
- Crear cadenas de control
- Sincronizar el acceso de nuestra memoria.

Estas tareas están unidas a las características de la máquina paralela donde estaremos trabajando. El objetivo de esta estrategia es eliminar la responsabilidad del programador a la hora de paralelizar el código, en la medida que esto sea posible.

Librerías paralelas: la idea base es que normalmente el tiempo de cómputo en las aplicaciones está concentrada en una sola parte del mismo. Si tratamos de optimizar esas partes, por ejemplo, produciendo versiones paralelas eficientes de las mismas en forma de rutinas, podríamos realizar librerías paralelas de ellas. Este enfoque es utilizado principalmente por las maquinas vectoriales para las cuales se ha desarrollado rutinas óptimas.

Este enfoque puede llegar a trabajarse muy bien debido a que solo hay que concentrarse en pequeñas partes, además es fácil lograr su escalabilidad ante nuevos avances tecnológicos.

Recodificación de partes: teniendo en cuenta la plataforma disponible rehacer parte del código es otra posibilidad a tener en cuenta. Pero para esto debemos tener en cuenta algunos detalles como la no dependencia entre las iteraciones, los datos, si la aplicación es susceptible a explotar un grano fino, entre otras.

Conociendo las estrategias que podemos utilizar para desarrollar una aplicación con técnicas de programación paralela, las podemos poner en práctica con un ejercicio bastante sencillo.

Para comenzar desarrollaremos un programa bastante sencillo utilizando programación secuencial. El programa se encargará de sumar dos vectores y el resultado de esta operación se guardará en un tercer vector. Los vectores serán de tipo double, y dicho programa lo desarrollaremos en C++, con el entorno gráfico QT, pero podemos usar otros entornos de programación como son CodeBlock, kdevoloped, entre otros, como podemos realizarlo desde la terminal de Linux usando el editor de texto nano.

Una vez realizado el programa deberíamos obtener algo tal como se muestra a continuación.

```
#include<iostream>
using namespace std;

int main(){
double a[100], b[100],c[100];

for (int i=0;i<100;i++){
    a[i]=1;
    b[i]=2;
}
for (int i=0;i<100;i++){
    c[i]= a[i]+b[i];
}


for (int i=0;i<100;i++){
    cout <<c[i];
}
return 0;
}
```

Como podemos observar en la imagen nuestro código se encarga de sumar dos vectores de 1000 posiciones cada uno. El siguiente paso a realizar es identificar la parte del código que puede ser paralelizada. Ahora si miramos detenidamente el código tiene 3 estructura de control paralelo (for), lo cual nos indica que podemos llegar a realizar la paralelización del código. Continuando con la observación, nos damos cuenta que no existe dependencia de datos, lo cual nos da una buena opción para llevar a cabo el proceso de paralelización.

Una vez identificado el sector de código candidato a la paralelización pasamos a realizar el proceso de paralelización, para este caso utilizamos *#pragma omp parallel for* para indicar cuál será nuestra parte a paralelizar. Una vez realizado esto el código debería verse de la siguiente manera.

```
#include<iostream>
using namespace std;

int main(){
double a[100], b[100],c[100];

for (int i=0;i<100;i++){
    a[i]=1;
    b[i]=2;
}
#pragma omp parallel for
for (int i=0;i<100;i++){
    c[i]= a[i]+b[i];
}


for (int i=0;i<100;i++){
    cout <<c[i];
}
return 0;
}
```

Como observamos en la imagen hemos utilizado paralelismo automático, cuando le agregamos la nueva línea de código le indicamos al compilador que esa es la sección a paralelizar y este será el que se encargue de hacerlo. El compilador se encargará de crear tantos hilos como procesadores tenga disponible y de esta manera llevar a cabo la paralelización. Adicional a esto se realiza el proceso de paralelización del código en el for encargado de realizar la suma ya que esta es la parte que depara mayor capacidad de cómputo.

Ahora pasamos a realizar la compilación y ejecución del código, lo podemos hacerlo de dos formas, la primera es hacer la compilación y ejecución desde el entorno de desarrollo. Para esto le damos clic al botón Build Project o utilizamos el acceso rápido control + B, una vez compilado pasamos a ejecutar el programa, para ello le daremos clic al botón run o utilizamos el acceso rápido control + R, al realizar este último paso aparecerá un nueva ventana donde se verá reflejado la suma de los dos vectores, como podemos observar en la siguiente imagen

```
33333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333
Process returned 0 (0x0)   execution time : 0.048 s
Press any key to continue.
```

Figura 14. Resultados de la ejecución.

Otra forma de hacer la compilación y ejecución de la aplicación es por medio de la terminal, para esto debemos entrar en ella y posicionarnos en la carpeta donde se encuentra nuestro código, una vez estando ahí pasamos a teclear la siguiente línea:

```
g++ -o "nombre de salida" "nombre ejecutable.cpp" –fopenmp
```

Para este caso en particular, llamamos el archivo cpp codigo_paralelo.cpp y el ejecutable ejecutable_codigo, entonces la línea deberá quedar así:

```
g++ -o ejecutable_codigo codigo_paralelo.cpp –fopenmp
```

Si todo sale bien hasta este punto ya tenemos el programa compilado, ahora para ejecutar el programa tecleamos lo siguiente:

```
./ejecutable_codigo
```

Apenas tecleamos esto podremos observar en la terminal los resultados de la ejecución.

De esta manera podemos decir que hemos realizado nuestra primera aplicación utilizando programación paralela. Cabe aclarar que nosotros podemos realizar códigos muchos más complejos donde tengamos más control sobre la parte paralelizada, por ejemplo, podemos indicar el número exacto de hilos que deseemos utilizar siempre y cuando tengamos el mismo número o mayor de procesadores disponibles, podemos establecer si nuestras variables serán compartidas por todos los procesos, entre otras cosas.

Conclusiones

Los resultados obtenidos en los mapas de enfoque calculado con C++ y utilizando Matlab, arrojan las misma partes enfocadas de la imagen. Y dado el desempeño en tiempo obtenido con C++ establece entonces que esta es la mejor opción para el cálculo de mapas de enfoque, esto garantiza realizar el procesamiento de imágenes para la reconstrucción tridimensional de objetos micróscopicos, mediante mapas de enfoque, en menos tiempo que el que se obtiene utilizando la herramienta Toolbox Parallel Computing de Matlab.

A partir del desarrollo de esta investigación se puede concluir que el uso de programación paralela es adecuado a la hora de resolver problemas que requieren de procesar un gran volumen de datos, aplicaciones como el procesamiento 3D de imágenes de microscopía, simulaciones de clima, entre otras. También podemos darnos cuenta que la programación paralela tiene un límite de procesadores a utilizar dependiendo del tipo del problema.

A pesar de la dificultad de implementar las técnicas de programación paralelas valen la pena implementarlas porque la ganancia en tiempos de ejecución son bastantes considerables dependiendo de cada problema. Adicional se observó que las técnicas de programación paralela cuando son realizadas por lenguajes compilados y no interpretados dan mejores resultados respecto al tiempo de ejecución.

Incrementar el número de procesadores no siempre nos va a recortar el tiempo de respuesta de nuestro programa, existe un límite de procesadores a utilizar dependiendo de la magnitud de cada problema, que pueden llegar a ser medidos por diferentes leyes como lo son la de Amdahl o Gustafson.

Utilizando programación paralela se aprovecha al máximo los recursos físicos con los que se dispone, logrando así mejores resultados, como son un mayor rendimiento y ahorro de energía, entre otros.

Se cuenta con una guía de desarrollo base para realizar programas utilizando técnicas de programación paralela en C++ con la librería OpenMP y el entorno de desarrollo Qt, donde se tienen diferentes metodologías que nos ayudarán a la hora de establecer los bloques de código que pueden paralelizarse adecuadamente.

Y en cuanto a la metodología utilizada para desarrollar este proyecto de grado se puede decir que no se presentó inconveniente alguno que pudiera dificultar el logro de los objetivos propuesto. Más sin embargo,

Recomendaciones

Dado el desempeño de la programación paralela con C++, OpenCV y OpenMP para el cálculo de los mapas de enfoque, se puede ampliar la aplicación al diseño de aplicaciones para la reconstrucción 3D de Objetos de microscopía.

Con el bajo costo y la posibilidad de cálculo en las actuales Unidades de Procesamiento Gráfico (GPU, por su sigla en inglés), intentar abordar el problema mediante el cálculo de los mapas de enfoque desde la GPU.

A pesar de que existen diferentes IDE que pueden adaptarse para desarrollar aplicaciones paralelas utilizando las librería de OpenMP, OpenCV y MPI, recpomendamos utilizar el programa Qt que la consideramos una excelente alternativa, debido a que su configuración es bastante sencilla. IDEs como Code::Block, Kdevelop, entre otros requieren de una configuración mucho más compleja y precisa.

Para trabajops de investigación futuros, se recomienda profundizar en la realización de pruebas rigurosas de los principales lenguajes de programación que soportan las técnicas de programación paralela para comprobar sus ventajas y desventajas, analogías y diferencias. Adicional a esto probar la programación paralela bajo diferentes entornos (Sistemas operativos como Linux y Windows).

Bibliografía

Bernal, Cesar Augusto. Metodología de la investigación - Segunda edición

Biermann, Enrique. Metodología de la investigación y el trabajo científico Editorial: UNAB.

Gutiérrez, Fidel Barboza, El Trabajo de Grado una oportunidad para seguir aprendiendo Editorial: Universidad Distrital Francisco José de Caldas

Rohit Chandra, Leonardo, y otros. Parallel Programming in OpenMP.

Wen-mei W. Hwu. Programming massively parallel processors,

W.Davidson, Michael. Microscopía, (28 de Noviembre 2004), Recuperado el 21de Enero de 2014 de http://micro.magnet.fsu.edu/primer/anatomy/introduction.html

Fernández Baldomero, Javier. Parallel Virtual machine toolbox, Recuperado el 12 de Diciembre de 2014 de http://www.ugr.es/~jfernand/investigacion/papers/Users99.pdf

INSTITUTO COLOMBIANO DE NORMALIZACIÓN Y CERTIFICACIÓN. Norma tecnica colombiana 5613: Referencias bibliograficas, contenido, forma y estructura. Bogota D.C., Colombia : El Instituto, 2008.

NAMAKFOROOSH, Mohammad Naghi. Metodología de la investigación. 2 ed. México: Limusa, 2010. ISBN 978-968-18-5517-8

HERNANDEZ Sampieri, Roberto; FERNANDEZ Collado, Carlos y BAPTISTA Lucio, Pilar. Metodología de la Investigación. 3 ed. México: McGraw-Hill, 2003. p. 4 - 296

QUESADA Herrera, José. Redacción y presentación del Trabajo Intelectual. 2 ed. Madrid: Paraninfo, 1987. p. 97 - 142

SALKIND, Neil J. Métodos de Investigación. 3 ed. México: Prentice Hall Hispanoamérica, 1999. p. 10 - 258