UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

TESIS

ACELERACIÓN DE ALGORITMOS UTILIZANDO TECNOLOGÍAS DE MULTIPROCESAMIENTO

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

PRESENTAN:

ARIEL ULLOA TREJO
LUIS FERNANDO PÉREZ FRANCO

DIRECTORA DE TESIS

ING. LAURA SANDOVAL MONTAÑO

CIUDAD UNIVERSITARIA. Abril 2015.

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos:

Para poder alcanzar el punto en el que ahora me encuentro, agradezco a muchas personas, que con su participación directa o indirecta, me han acompañado en el sendero de la vida. Quiero agradecer principalmente:

A mis padres:

Cuyo amor sincero es el primer recuerdo que poseo. Inculcaron en mí los valores que me han permitido alcanzar mis metas y la idea de ir siempre hacia adelante. Han sido los pilares en los que siempre me apoyo. Agradezco sus esfuerzos y sacrificios.

A mis hermanos:

Han formado parte de mi vida y siempre están conmigo incondicionalmente. Ustedes me obligaron a ser mejor cada día.

Ing. Laura Sandoval:

Quien me brindó apoyo, conocimientos, orientación y oportunidades para desempeñar mis actividades mejor. Sin usted, este trabajo habría sido imposible.

A mis amigos:

Por su incansable apoyo. Por haberme enseñado muchas cosas y aguantarme otras tantas. Gracias por confiar en mí.

Ariel Ulloa Trejo

Contenido

Introducción ***1***

1. Evolución de sistemas de multiprocesamiento ***5***

1.1 Principios del multiprocesamiento **5**

1.2 Arquitecturas de multiprocesamiento **7**

1.3 Herramientas de desarrollo **12**

Parallel Processing Shell Script (PPSS) **12**

Portable Operating System Interconnection for Unix (POSIX) **13**

OpenMP **15**

Message Passing Interface (MPI) **17**

Cilk++ **18**

Java Threads **19**

Compute Unified Device Architecture (CUDA) **20**

Open Computing Language (OpenCL) **22**

2. Fundamentos del procesamiento paralelo ***25***

2.1 Complejidad, tiempo y eficiencia de algoritmos **25**

2.2 Ley de Amdahl, granularidad y regiones paralelas **30**

2.3 Taxonomía de Flynn **33**

2.4 Procesadores multinúcleo **34**

2.5 Tarjetas Gráficas **37**

2.6 Infraestructura utilizada **38**

3. Algoritmos de ordenamiento ***41***

3.1 Algoritmo de ordenamiento por cuenta **42**

3.2 Algoritmo de ordenamiento Shell **44**

3.3 Algoritmo de ordenamiento por casilleros **47**

3. 4 Selección del algoritmo a paralelar y justificación **49**

3.5 Análisis e implementación paralela con OpenMP **49**

3.6 Análisis de resultados **52**

4. Algoritmos de búsqueda de cadenas ***57***

4.1 Algoritmo Knuth-Morris-Pratt **57**

4.2 Algoritmo Rabin-Karp **62**

4.3 Selección del algoritmo a paralelar y justificación **65**

4.4 Análisis e implementación paralela en CUDA 66
4.5 Análisis de resultados 68
5. Algoritmos celulares o evolutivos *73*
5.1 Descripción 73
5.2 Análisis e implementación serial 75
5.3 Selección del algoritmo a paralelar y justificación 76
5.4 Análisis e implementación paralela en OpenMP 76
5.5 Análisis de resultados 80
6. Algoritmos para generar números primos *83*
6.1 Algoritmo por fuerza bruta 84
6.2 Algoritmo Criba de Eratóstenes 86
6.3 Selección del algoritmo a paralelar y justificación 88
6.4 Análisis e implementación paralela en CUDA 88
6.5 Análisis de resultados 90
Conclusiones *93*
Bibliografía *97*
Anexo 1: Códigos de programas seriales *101*
1.1 Algoritmos de ordenamiento 101
1.2 Algoritmos de búsqueda de cadenas 106
1.3 Algoritmo genético (C) 110
1.4 Algoritmos para generar números primos 112
Anexo 2: Códigos de programas paralelos *115*
2.1 Algoritmo casillero-shell (C++ con OpenMP) 115
2.2 Algoritmo Rabin-Karp (CUDA C/C++) 119
2.3 Algoritmo genético (C con OpenMP) 122
2.4 Algoritmo Criba de Eratóstenes (CUDA C/C++) 124

Introducción

Actualmente existen problemas que demandan mucho tiempo de procesamiento en una computadora, por ejemplo: predicción del estado del tiempo, algoritmos para trazar rutas de aviones, simulación de fenómenos naturales, peticiones a una base de datos, análisis de información, conversión de formatos (de video principalmente), etc. Algunos de ellos, además de utilizar tiempo en CPU necesitan gran cantidad de memoria. Solucionar estos problemas puede tardar desde minutos hasta días o meses.

Desde sus inicios, la solución para resolver estos problemas fue fabricar computadoras con procesadores cada vez más rápidos, pero con el paso de los años, se alcanzaron límites físicos que impedían continuar con esta tendencia. Entonces se desarrollaron máquinas con varias unidades de procesamiento.

Actualmente, todas (o casi todas) las computadoras tienen por lo menos procesadores con dos núcleos y pueden ser adquiridas a precios accesibles. Incluso es raro encontrar dispositivos móviles (como celulares y tablets) que tengan un solo CPU.

A pesar de esto, las soluciones a muchos problemas de cómputo se realizan de manera secuencial, desperdiciando recursos hardware, esto debido a que aún es poca la cultura de desarrollo de aplicaciones que aprovechen las capacidades del procesamiento paralelo. Sin embargo, en la actualidad, existen diferentes herramientas para el desarrollo de programas paralelos que hace que esta actividad se vuelva más sencilla, atractiva y factible, obteniendo software que reduce el tiempo de ejecución considerablemente, así como soluciones de manera más rápida.

Por lo anterior, el objetivo de este trabajo de tesis es analizar algoritmos que demandan gran cantidad de memoria y tiempo de procesamiento, así como encontrar la pertinencia de mejorar su eficiencia utilizando infraestructura y herramientas de multiprocesamiento, además de difundir el uso de herramientas para desarrollar aplicaciones paralelas, dado que su uso no es tan amplio como debería.

El presente trabajo está dividido en dos partes. En la primera se estudian las bases teóricas necesarias y herramientas de desarrollo, mientras que la segunda comprende el estudio e implementación de varios algoritmos de diferentes categorías para acelerarlos, programándolos de forma paralela y obtener métricas para decidir si conviene o no hacerlo.

Para medir nuestros resultados, utilizamos máquinas convencionales, tanto portátiles como de escritorio, a las que cualquier persona puede tener acceso (como procesadores Intel i5, i7 y tarjetas gráficas Nvidia GeForce), así como equipo más robusto que corresponde a estaciones de trabajo o servidores (como procesadores Xeon o tarjeta Tesla K20c, uno de los GPUs más poderosos desarrollados hasta la fecha). Las herramientas software que utilizamos principalmente son OpenMP y CUDA, ambas extensiones de lenguaje C.

A continuación se presenta una breve descripción de cada uno de los capítulos que conforman al presente trabajo de investigación:

Capítulo 1: Evolución de sistemas de multiprocesamiento. En este capítulo abordamos los avances de sistemas de multiprocesamiento a lo largo de la historia, las diferentes arquitecturas como multinúcleos, GPUs, granjas, cómputo en la nube y supercomputadoras. También la descripción de algunas herramientas de desarrollo de programas paralelos, funciones utilizadas y ejemplo de códigos.

Capítulo 2: Fundamentos del procesamiento paralelo: Aquí se introducen conceptos clave para comprender los resultados, tales como complejidad, tiempo de ejecución y eficiencia. Éstos los utilizamos para obtener métricas y poder comparar resultados entre algoritmos del mismo tipo, así como para obtener configuraciones óptimas para ejecutar en varias unidades de procesamiento. Esto nos permitirá entender la pertinencia de utilizar todos los recursos de una computadora.

Además, abordamos la base teórica para analizar los problemas y saber cómo resolverlos, tales como la ley de Amdahl, granularidad, Taxonomía de Flynn tanto en hardware como software y tipos de descomposición. Éste último es uno de los más importantes, ya que para poder paralelar programas, debemos saber si éstos se pueden descomponer funcionalmente o por dominio. Si no se puede de alguna de estas maneras, el programa no se puede paralelar.

Capítulo 3: Algoritmos de ordenamiento. En éste quisimos salirnos de los algoritmos más comunes (como *quick, bubble y mergesort*) y estudiar tres no tan recurridos. Entonces seleccionamos a ordenamientos por inserción, por casilleros y por cuentas. Analizamos los algoritmos, los implementamos de forma secuencial, estimamos ventajas y desventajas frente a los demás y seleccionamos uno de ellos para ser paralelado. Incluimos la justificación y posteriormente todo el trabajo realizado para programar la versión paralela y presentar los resultados.

Capítulo 4: Algoritmos de búsqueda de cadenas. Es muy importante la manera en la cual manipulamos el texto, principalmente cuando se tiene una gran base de información. Por ejemplo, un solo libro de 400 páginas puede contener más de un millón de caracteres. En este capítulo se estudian los algoritmos *Knuth-Morris-Pratt* y *Rabin-Karp*. La comparación entre estos dos es muy interesante, porque aquí se puede observar con claridad porqué uno no puede ser paralelado, mientras que el otro sí. De igual manera, se obtienen las métricas del programa acelerado y se ejecuta en diferentes plataformas.

Capítulo 5: Algoritmos celulares o evolutivos. Este tipo de algoritmos busca la mejor solución a los problemas de búsqueda y optimización cuya solución está en un rango determinado y conocido. Estos algoritmos reciben su nombre de la teoría de evolución de Darwin, pues se obtienen soluciones a partir de una población inicial y se van creando generaciones (descendientes) que tienen características de sus "padres"; cabe destacar que las soluciones no son exactas y que para obtener mejores soluciones, se produce una mayor cantidad de generaciones.

Como la idea es básicamente la misma para diferentes algoritmos, sólo implementamos uno, el cual busca el máximo de una función en un rango definido. Se analiza secuencialmente y posteriormente se paralela, se obtienen métricas y se ejecuta en diferentes plataformas.

Capítulo 6: Algoritmos para generar números primos. Los números primos tienen una aplicación muy importante en la criptografía. Esto es cifrar mensajes para que sean seguros a través del medio y que sólo puedan ser descifrados y leídos por un receptor, sin que algún tercero pueda hacerlo si lo intercepta. En 1977, Ronald Rivest, Adi Shamir y Leonard Adleman encontraron una manera sencilla para encriptar mensajes. Se toman dos números primos muy grandes y se multiplican. Para descifrarlo, basta con conocer a ambos factores. La dificultad para descifrar el mensaje, aunque todo el mundo conozca el número enorme no-primo, está en encontrar a los dos primos originales.

Se estudian dos algoritmos. El primero es por *fuerza bruta*, es decir, para generar los primos hasta un entero p, hay que verificar que éste no sea divisible por algún primo anterior. El segundo es la *criba de Eratóstenes*, que consiste en seleccionar un número menor a la raíz de p que sea primo y marcar todos sus múltiplos. Al igual que los anteriores, para este algoritmo también se analizan las métricas.

Conclusiones: En este apartado incluimos las conclusiones, donde se detallan las razones por las cuales obtuvimos los resultados, así como decidir qué hardware resultó ser más eficiente, la proyección a futuro de estas tecnologías y el aprendizaje obtenido.

Bibliografía: Aquí se incluyen todas las referencias utilizadas durante el presente trabajo.

Anexo 1 y 2: En éstos se presentan los códigos de todos los programas elaborados, debidamente documentados, tanto secuencialmente como en paralelo.

1. Evolución de sistemas de multiprocesamiento

1.1 Principios del multiprocesamiento

El multiprocesamiento es la conjunción de software y hardware, ambos adaptados para la ejecución, optimización y administración de programas, los cuales tienen ciertas partes que pueden ser ejecutadas al mismo tiempo sin poner en riesgo la integridad de los resultados finales.[1]

El multiprocesamiento surgió como una alternativa para optimizar la ejecución de ciertas tareas que consumían grandes recursos y a la vez gran cantidad de tiempo.

Desde el principio de la invención de la arquitectura Von-Neumann se optó por el aumento de la frecuencia de los procesadores y en general de los componentes de un computador para incrementar su rendimiento. Al aumentar la frecuencia se procesa un mayor número de instrucciones en un determinado tiempo, pero a su vez crea otros inconvenientes como el aumento de temperatura y deterioro acelerado de los componentes de los sistemas; debido a esto se buscaron otras formas de mejorar el rendimiento sin llegar a aumentar las frecuencias a valores exorbitantes.

Uno de los primeros en proponer una solución fue el mismo Von-Neumann que a principio de 1950´s empezó a hacer analogías entre el cerebro y los computadores, proponiendo un modelo de cómputo paralelo. Así mismo Church y Turing propusieron un tiempo de cómputo similar a como trabajan las neuronas, generando las primeras teorías en lo que se basaría el actual cómputo paralelo.

En la tabla 1 se muestran hechos relevantes que influyeron al desarrollo del cómputo paralelo:

[1]Gutiérrez A. (diciembre 2009). *Sistemas de Multiprocesamiento*. Agosto 2014, de UPICSA Sitio web: http://www.sites.upiicsa.ipn.mx/polilibros/portal/polilibros/P_terminados/PolilibroFC/Unidad_VI/Unidad%20VI_4.htm#IrInicioUnidad

Tabla 1: Línea del tiempo[2]

Fecha	Acontecimiento
1956	Primer supercomputadora construida con base en transistores por IBM llamado IBM 7030
1958	John Cocke y Daniel Slotnick publican en una nota de IBM el uso del paralelismo para cálculos numéricos.
1960	E. V. Yevreinov en el Instituto de Matemáticas en Novosibirsk comienzan a trabajar en una arquitectura paralela de grano grueso.
1962	C. A. Petri describe las redes de Petri, que consisten en la teoría para describir y analizar las propiedades de sistemas concurrentes. Burroughs crea el primer multiprocesador para fines militares; que constaba de 1 a 4 procesadores con capacidad de acceder a 1-16 módulos de memoria.
1964	Se crea la CDC 6600, una supercomputadora la cual contaba con 10 procesadores, los cuales se dividían las tareas rutinarias como perforación de tarjetas y gestión de disco.
1965	Edsger Dijkstra describe el problema de las regiones críticas. Las cuales consisten en regiones que pueden ser modificadas al mismo tiempo por uno o más procesos.
1966	Sperry Rand Corporation diseña el primer multiprocesador para fines no militares con 3 CPUs y 2 controladores de entrada/salida.
1968	Edsger Dijkstra presenta la teoría de los semáforos, muestra el problema de "la cena de los filósofos" y presenta un ejemplo de la teoría de concurrencia.
1969	Compass Inc. Comienza los trabajos para intentar paralelizar FORTRAN.
1973	Se crea Basic Linear Algebra Subprograms (BLAS). BLAS, es un software que facilita los cálculos de algebra lineal; fue uno de los primeros en implementar el cómputo paralelo para la optimización de los cálculos en operaciones con vectores y matrices.
1974	Tony Hoare describe los monitores y los mecanismos de la exclusión mutua. La exclusión mutua es un mecanismo que no permite el ingreso de 2 procesos a una región crítica, evitando incoherencias en los datos.
1975	Edsger Dijkstra describe los "comandos guardados" y la concurrencia estructurada.
1976	Utpal Banerjee's formaliza el concepto de "dependencia de datos". Este concepto se refiere a que ciertos datos dependen de otros previamente procesados, por lo cual no se pueden ejecutar paralelamente.
1979	Entra en operación el primer "data-flow-multiprocesor" o en el CERT-ONERA.
1980	PFC, primer compilador orientado al multiprocesamiento basado en FORTRAN. J. T. Schwartz crea el concepto de ultra computación, la cual consiste en múltiples procesadores conectados en red y trabando a la par.
1981	Bruce J. Nelson introduce al concepto de "procedimientos remotos". Este concepto es ampliamente utilizado en los sistemas de bases de datos distribuidas actuales; permiten la visualización y modificación de datos entre dos o más entidades que no están en un mismo equipo físico.

[2] Schauer B. (septiembre 2008). *Multicore Processors – A Necessity*. Agosto 2014, de Pro Quest. Sitio web: http://cse.unl.edu/~seth/990/Pubs/multicore-review.pdf

1982	Cray Research crea la CRAY-1. Primera computadora vectorial. Estaba adaptada para hacer operaciones básicas de manera simultánea. Utilizaba Cray-FORTRAN, que fue el primer compilador de vectorización automática.
1983	En la Secretaria de Defensa de EEUU se hace mención de ADA, el cual introduce a conceptos como la comunicación y sincronización entre procesos.
1985	Cray Research produce CRAY-2. Fue una mejora de la CRAY-1, la cual mejoró su rendimiento a 1.9 GFlops.
1987	Se crea BLAZE, un lenguaje orientado a la memoria compartida. Optimizaba tareas como operaciones de arreglos, ciclos. Permitía la portabilidad entre varias arquitecturas multinúcleo.
1990	University of Tsukuba crea una supercomputadora con 432 procesadores.
1992	Nace Message-Passing Interface Forum (MPI) para generar un estándar de comunicación por mensajes. Este sistema permite la sincronización de procesos y la exclusión mutua para evitar incoherencias y bloqueos mutuos.
1997	Se crea la primera versión de OpenMP. Es una API que permite la programación multiproceso de memoria compartida. Actualmente OpenMP está disponible para distintos lenguajes de programación como C, C++ y Python.
2005	Intel produce el Pentium D, el primer procesador comercial para PCs con 2 núcleos. El Pentium D consiste en 2 procesadores Pentium 4 en un solo encapsulado y con la posibilidad de comunicación entre procesos.
2007	NVIDIA presenta CUDA. API que permite utilizar las tarjetas gráficas como procesador de datos en paralelo; aprovechando la gran cantidad de núcleos con los que cuentan.
2008	Nace OpenCL, una alternativa libre y no propietaria. OpenCL a diferencia de CUDA, que solo funciona con tarjetas NVIDIA, permite utilizar tarjetas gráficas de diferentes empresas como ATI e Intel para el procesamiento paralelo.

1.2 Arquitecturas de multiprocesamiento

La computación paralela, es muy importante actualmente, al permitir mejorar la velocidad en la solución de grandes problemas, de modo que se mejora el rendimiento de cómputo. En la actualidad existen varios medios de procesamiento paralelo/distribuido. A continuación se describen algunos.

Granjas

En algunos casos se habla de granja de procesos cuando un conjunto de procesos trabaja de manera conjunta pero independiente en la resolución de un problema. Este paradigma se asimila con el maestro-esclavo si consideramos que los procesos que constituyen la granja son los esclavos.

Actualmente tienen aplicación en:

1. Granjas de compilación: Es un conjunto de uno o más servidores que ha sido creado para compilar programas remotamente. Entre los objetivos principales se encuentran:
    - Desarrollar aplicaciones para múltiples plataformas, dado que en una granja se cuenta con diversos Sistemas Operativos.

- La arquitectura de un CPU no es una limitante.
- Utiliza mejor los *hosts* disponibles, dividiendo la carga de trabajo a pesar de que sus velocidades de CPU difieran.
- Compilación distribuida: Módulos que pueden ser compilados de forma paralela.

2. Granjas de renderizado: Es un clúster de computadoras diseñado y organizado con el propósito de hacer *renderizado* de imágenes, como se muestra en la ilustración 1. Cada nodo recibe una única tarea que no requiere de las demás. Aquí, el trabajo del maestro es repartir la carga de trabajo y posteriormente trabajar con los resultados. Una vez que un nodo termina un trabajo, el maestro debe de ser capaz de asignarle nuevas tareas. Estas actividades se deben poder realizar sin supervisión humana.

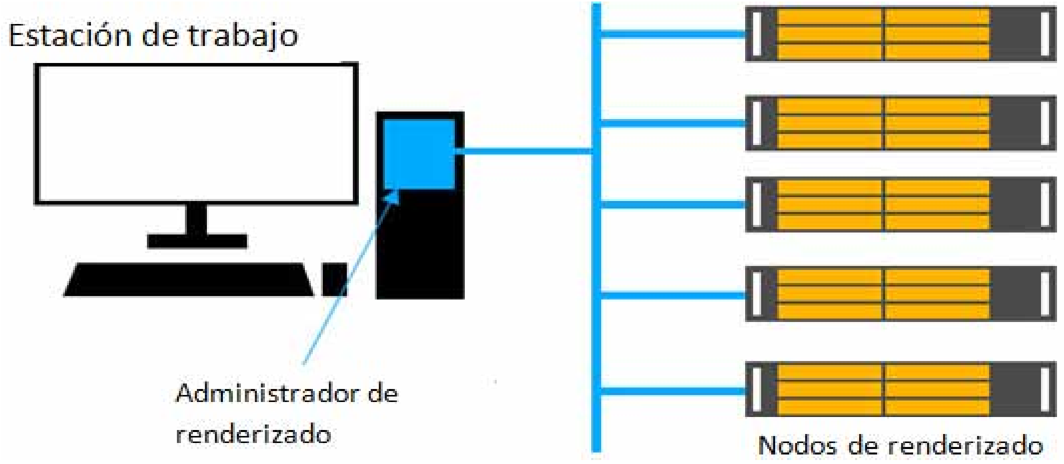


Ilustración 1. Granja de renderizado[3]

Supercomputadoras (Top 500)

En la tabla 2 se muestra el Top 10 de supercomputadoras

Tabla 2:Top 10 Supercomputadoras (junio 2014)[4]

#	Nombre	Creador	País	Año	CPU´s	Arq.	CPU	SO
1	Tianhe-2	NUDT	China	2013	3120000	Clúster	Intel Ivy Bridge	Kylin Linux
2	Titan	Cray Inc.	EUA	2012	560640	MPP	AMD x86_64	Cray Linux
3	Sequoia	IBM	EUA	2011	1572864	MPP	PowerPC	Linux
4		Fujitsu	Japón	2011	705024	Clúster	Sparc	Linux
5	Mira	IBM	EUA	2012	786432	MPP	PowerPC	Linux
6	PizDaint	Cray Inc.	Suiza	2012	115984	MPP	Intel Sandy Bridge	Cray Linux
7	Stampede	Dell	EUA	2012	462462	Clúster	Intel Sandy Bridge	Linux
8	JUQUEEN	IBM	Alemania	2012	458752	MPP	PowerPC	Linux
9	Vulcan	IBM	EUA	2012	393216	MPP	PowerPC	Linux
10		Cray Inc.	EUA	2014	225984	MPP	Intel Ivy Bridge	Cray Linux

[3] Sareesh. (2013). *What is a Render Farm?* [Imagen] Sitio web: http://wolfcrow.com/blog/what-is-a-render-farm/
[4] Top 500. (2014). *Lista de junio de 2014*. Agosto 2014. Sitio web: http://www.top500.org/

Se pueden observar las siguientes características:

- Algunos fabricantes coinciden, como Cray Inc. e IBM.
- La mayoría son de origen Estadounidense.
- Todas son relativamente nuevas (no mayores a dos años de antigüedad).
- No necesariamente un mayor número de CPU´s significa mayor poder de cómputo.
- La diferencia básica entre las arquitecturas, es que un clúster es un sistema distribuido compuesto por un conjunto de computadoras autónomas e interconectadas, trabajando juntas en forma cooperativa como único recurso integrado, mientras que en una MPP (procesamiento masivamente paralelo) los nodos no pueden correr su propio código.
- Aunque se observan diferentes fabricantes de CPU´s, el dominante es Intel.
- Todos corren alguna distribución de Linux.

Actualmente, en México las supercomputadoras con más poder de cómputo son:

- **Xiuhcóatl (IPN):** Tiene 3480 unidades de procesamiento tanto de Intel como de AMD y 16128 de coprocesamiento (tarjetas gráficas FERMI 2070). Cuenta con memoria 7200 GB de RAM.
- **Miztli (UNAM)**. Fabricada por HP con arquitectura Clúster. Cuenta con 5320 unidades de procesamiento Intel E5-2670, 16 tarjetas NVIDIA m2090 y memoria RAM de 15000 GB.
- **KanBalam (UNAM)**: Fue fabricada por HP y tiene arquitectura de Clúster. Tiene 1368 procesadores (AMD Opteron 285), 3000 GB de RAM.
- **Aitzaloa (UAM):** Tiene 270 nodos con procesadores Intel Xeon Quad-Core (1080 unidades de procesamiento) y 4320 GB de RAM total

Tecnología multicore y GPUs

El cálculo acelerado puede definirse como el uso de una unidad de procesamiento gráfico (GPU) en combinación con una CPU para acelerar aplicaciones de cálculo científico, ingeniería y empresa.[5]

El cálculo acelerado en la GPU ofrece un rendimiento sin precedentes ya que traslada las partes de la aplicación con mayor carga computacional a la GPU y deja el resto del código ejecutándose en la CPU. Desde la perspectiva del usuario, las aplicaciones simplemente se ejecutan más rápido.

Una forma sencilla de entender la diferencia entre la CPU y la GPU es comparar la forma en que procesan las tareas. Una CPU consta de unos cuantos núcleos optimizados para el procesamiento secuencial de las instrucciones, mientras que la GPU se compone de miles de núcleos, más pequeños y eficientes, diseñados para manejar múltiples tareas de forma simultánea, como se muestra en la ilustración 2:

[5] Nvidia Corporation. (2012). *Qué es el cálculo acelerado en la GPU*. Septiembre 2014, de Nvidia Sitio web: http://www.nvidia.es/object/gpu-computing-es.html

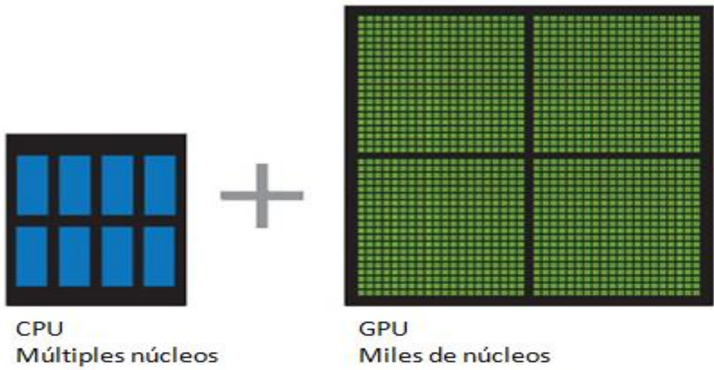


Ilustración 2. Representación de CPU y GPU[6]

Los sistemas con GPUs aceleradoras proporcionan el mejor rendimiento y la mayor eficiencia energética del actual mercado de la alta computación. Los sistemas acelerados se han convertido en la nueva norma para los entornos HPC (High Performance Computing) a gran escala y son líderes en supercomputación.

La ilustración 3 muestra los valores pico teóricos de algunas GPUs de NVIDIA comparadas contra procesadores Intel, tanto en operaciones de precisión sencilla o doble.

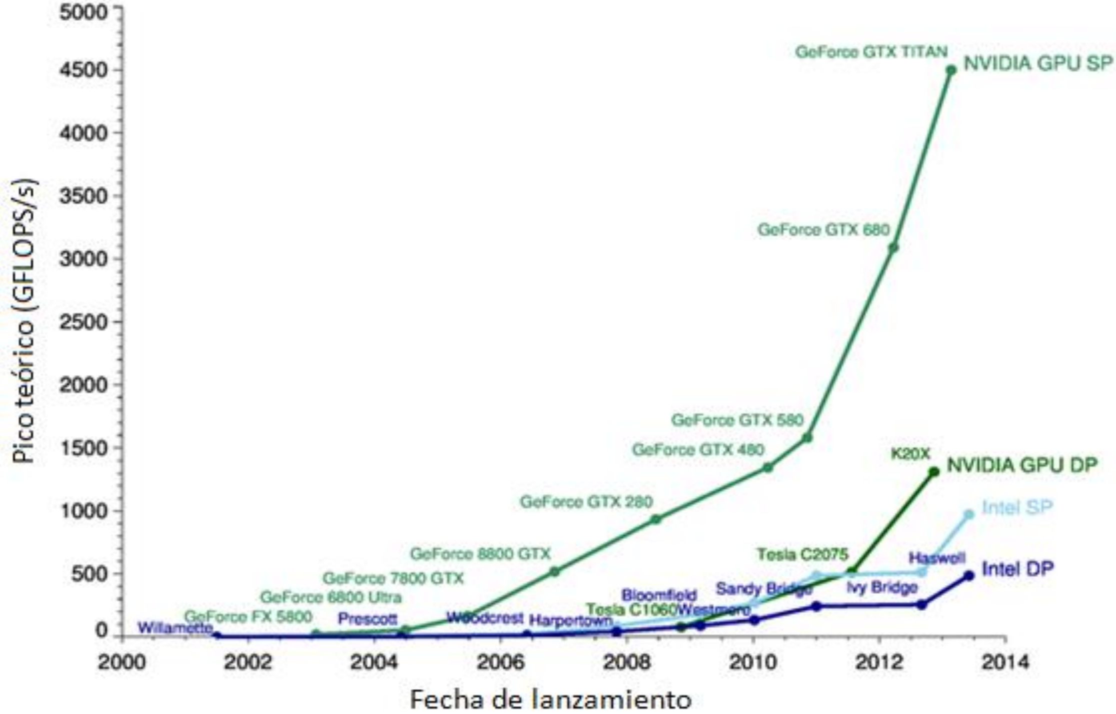


Ilustración 3. Evolución de GPUs y CPUs de Precisión Sencilla y Doble[7]

---

[6] Nvidia Corporation (2012). *Qué es el cálculo acelerado en la GPU* [imagen] Sitio web: http://www.nvidia.es/object/gpu-computing-es.html

[7] Galloy. (2013). *GPU vs CPU Performance*. [imagen] Sitio web: http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html

Cómputo en la nube

El cómputo en la nube es un nuevo paradigma, el cual se basa en el internet y sitios de procesamiento remotos a los cuales podemos tener acceso sin estar físicamente cerca y aprovechar sus recursos de procesamiento (hardware), software e información. [8]

El cómputo en la nube funciona como un servicio de procesamiento el cual permite a un usuario rentar servidores remotos, de los cuales no necesita conocer su ubicación ni configuración. Esto ha supuesto una gran ventaja ya que una empresa o usuario no necesita comprar los equipos físicamente, si no que busca un proveedor que le permita hacer uso de sus servidores enfocándose directamente a la aplicación y dejando la administración de los equipos físicos al proveedor. El cómputo en la nube es altamente configurable y escalable para los usuarios, como lo muestra la ilustración 4.

Tipos de cómputo en la nube:

- Nube pública: Es el paradigma principal del cómputo en la nube, en el cual se ofrecen aplicaciones a través de internet donde cualquier persona puede tener acceso o servicio de este sistema. Ejemplo de estas tecnologías son: IBM´s Blue Cloud, Google App Engine.
- Nube privada: Son servicios o aplicaciones, generalmente de marketing, las cuales sólo un limitado número de usuarios tienen acceso, su utilización puede tener un costo. Ejemplo de estas tecnologías son: eBay y HP CloudStart.
- Nube híbrida: Es una combinación de la nube privada y pública principalmente utilizada en empresas enfocadas a la tecnología de la información como HP, Oracle e IBM.
- Nube comunitaria: Es la asociación de múltiples usuarios que prestan o compartes sus equipos para un trabajo comunitario que permita llegar a un fin común.

Características:

- Virtualización: El cómputo en la nube se basa en el trabajo comunitario entre diferentes equipos con diferentes características de software y hardware, y que a pesar de estas diferencias deben poder trabajar como uno solo.
- Seguridad: El cómputo en la nube debe garantizar que los datos que se transmitan y procesan estén seguros en todo el proceso.
- Elasticidad: El cómputo en la nube es completamente configurable y adaptable a lo que requiera el usuario.
- Disponibilidad: El cómputo en la nube garantiza la disponibilidad de sus servicios en todo momento usando replicados y sitios redundantes por si alguno llega a fallar.

[8] Cloud Computing. (2013). *Computación en nube*. Septiembre 2014, de Computación en la nube. Sitio web: http://www.computacionennube.org/

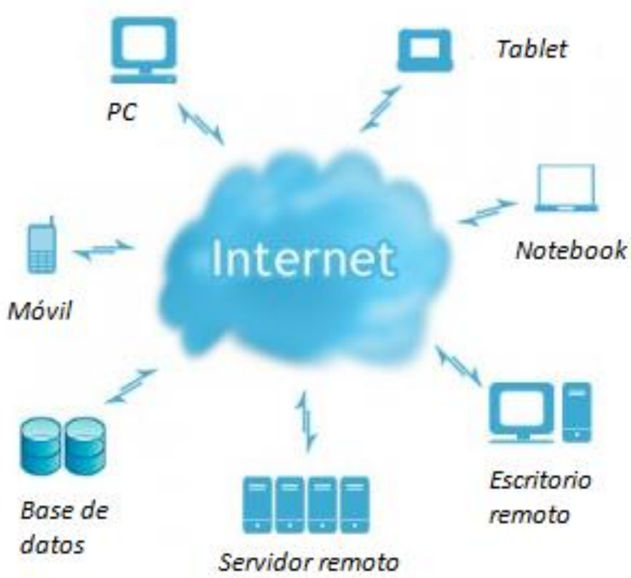


Ilustración 4. Cómputo en la nube[9]

1.3 Herramientas de desarrollo

Las necesidades de cómputo de numerosas aplicaciones obligan a desarrollar software eficiente y seguro para plataformas multiprocesador. Además, el auge de los procesadores multinúcleo y de las redes de computadoras ha aumentado la difusión del procesamiento paralelo, que cada vez está más al alcance del público en general. No obstante, para utilizar los sistemas paralelos y/o distribuidos de forma eficiente es necesaria la programación paralela.

En esta sección describiremos algunas herramientas de desarrollo en paralelo.

Parallel Processing Shell Script (PPSS)

PPSS es un script de bash shell que ejecuta los comandos, scripts o programas en paralelo. Está diseñado para hacer un uso completo de las actuales CPUs multi-core.

Detecta un número de CPUs disponibles e inicia trabajo separado por cada núcleo del CPU. PPSS puede ejecutar una misma tarea en varios nodos, tal como un clúster.

PPSS tiene una lista de elementos como entrada. Los elementos pueden ser archivos de un directorio o de las entradas en un archivo de texto. PPSS ejecuta un comando especificado por el usuario para cada elemento de la lista. En la ilustración 5 se muestra en consola.

[9] Cloud Computing (2013). Computación en nube. [imagen] Sitio web: www.computacionennube.org

```
pc@ubuntu:~$ ppss

|P|P|S|S| Distributed Parallel Processing Shell Script 2.97

usage: /usr/local/bin/ppss [[ -d <sourcedir> | -f <sourcefile> ]]  [[ -c '<comma
nd> "$ITEM"' ]]
                 [[ -C <configfile> ]] [[ -j ]] [[ -l <logfile> ]] [[ -p <# jobs
> ]]
                 [[ -q ]] [[ -D <delay> ]] [[ -h ]] [[ --help ]] [[ -r ]] [[ --d
aemon ]]

Examples:
                 /usr/local/bin/ppss -d /dir/with/some/files -c 'gzip '
                 /usr/local/bin/ppss -d /dir/with/some/files -c 'cp "$ITEM" /tmp
' -p 2
                 /usr/local/bin/ppss -f <file> -c 'wget -q -P /destination/direc
tory "$ITEM"' -p 10
```

Ilustración 5. PPSS en una terminal de Linux[10]

Este script es (sólo) útil para trabajos que pueden ser fácilmente clasificados en distintas tareas que se pueden ejecutar en paralelo. Por ejemplo, la codificación de muchos archivos *wav* a formato *mp3*, descargar un gran número de archivos, cambiar formatos de imágenes, etc.

Portable Operating System Interconnection for Unix (POSIX)

Es un estándar orientado a facilitar la creación de aplicaciones confiables y portables. La mayoría de las versiones populares de UNIX (Linux, Mac OS X) cumplen con este estándar. La biblioteca para el manejo de hilos en POSIX es *pthread*.

Los hilos POSIX tienen las siguientes características:

- Permiten la ejecución concurrente de varias secuencias de instrucciones asociadas a funciones dentro de un mismo proceso (hilo principal).
- Los hilos hermanos entre sí comparten la misma imagen de memoria, es decir:
  - Código
  - Variables de memoria global.
  - Dispositivos y archivos asociados al hilo principal antes de la creación.
- Los hilos hermanos no comparten:
  - El Contador de Programa: Cada hilo puede ejecutar instrucciones distintas.
  - Los registros del CPU.
  - La pila en la que se crean variables locales.
  - Estado: los hilos pueden estar en ejecución, listos o bloqueados.

[10] Sandoval L. (2012). *Tutorial de PPSS*. Septiembre 2014, de Facultad de Ingeniería, UNAM. Sitio web: http://lcomp89.fi-b.unam.mx/

Las funciones básicas son descritas en la tabla 3:

Tabla 3. Funciones básicas de hilos POSIX

Función	Descripción
pthread_equal	Verifica la igualdad de identificadores de hilos.
pthread_self	Devuelve el identificador del propio hilo.
pthread_create	Crea un hilo.
pthread_exit	Termina el hilo sin terminar el proceso.
pthread_join	Espera por el término de un hilo.
pthread_cancel	Termina otro hilo.
pthread_detach	Configura liberación de recursos cuando termina.
pthread_kill	Envía una señal a otro hilo

Como la creación de hilos no es una tarea estándar de UNIX, se debe incluir la cabecera del código fuente *#include<pthread.h>*y en la línea de comandos para compilar:

$$gcc\ codigoFuente.c - lpthread$$

Los hilos POSIX cuentan también con funciones para la sincronización. Éstas incluyen mecanismos de exclusión mutua (mutex), mecanismos de señalización del cumplimiento de condiciones por parte de variables y de acceso de variables que se modifican de manera exclusiva, pero leídas de forma compartida. Las funciones para el manejo de zonas de acceso exclusivo tienen el prefijo *pthread_mutex*.

Un *mutex* es una variable especial que puede tener sólo dos estados: libre (*unlocked*) u ocupado (*locked*). Es como una compuerta que permite el acceso controlado a variables o regiones de código. Si un hilo tiene el *mutex*, entonces se dice que es dueño de éste y ningún otro hilo puede acceder. Si ningún hilo lo tiene, se dice que está libre. Cada *mutex* tiene una cola de hilos que están esperando para acceder. Es por esto que el acceso a estas regiones debe de ser mínimo y que ocupen el menor tiempo posible. Las funciones se describen en la tabla 4:

Tabla 4. Funciones para exclusión mutua

Función	Descripción
pthread_mutex_destroy	Destruye la variable usada para el manejo de exclusión mutua o candado *mutex*.
pthread_mutex_init	Permite dar condiciones iníciales a un candado *mutex*.
pthread_mutex_lock	Permite solicitar acceso a la región crítica; el hilo pasa a estado bloqueado hasta su obtención.
pthread_mutex_trylock	Permite solicitar el acceso a la región crítica. El hilo retorna inmediatamente.
pthread_mutex_unlock	Permite liberar un *mutex*.

OpenMP

OpenMP es una interfaz de programación de aplicaciones (API) para la programación multiproceso de memoria compartida en múltiples plataformas. Permite añadir concurrencia a los programas escritos en C, C++ y Fortran sobre la base del modelo de ejecución *fork-join*. Está disponible en muchas arquitecturas, incluidas las plataformas de Unix y de Microsoft Windows. Se compone de un conjunto de directivas de compilador, rutinas de biblioteca, y variables de entorno que influyen el comportamiento en tiempo de ejecución.

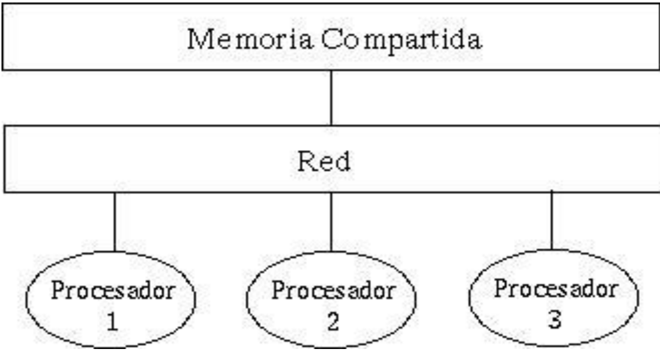


Ilustración 6. Diagrama de memoria compartida[11]

Definido conjuntamente por proveedores de hardware y de software, OpenMP es un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas, para plataformas que van desde las computadoras de escritorio hasta supercomputadoras. Una aplicación construida con un modelo de programación paralela híbrido se puede ejecutar en un clúster de computadoras utilizando OpenMP y MPI, o a través de las extensiones de OpenMP para los sistemas de memoria distribuida, como se muestra en la ilustración 7.

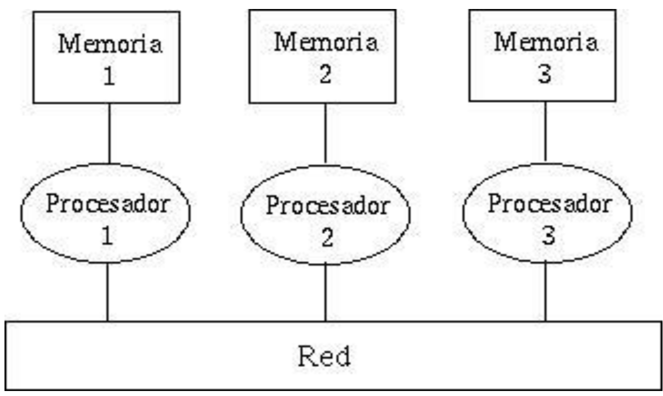


Ilustración 7. Diagrama de memoria distribuida[12]

[11]Caece (2011). Arquitectura de sistemas distribuidos. [imagen]. Sitio web: http://so2-caece.wikispaces.com/home.
[12] Ibídem.

Las funciones que se emplean están en la tabla 5:

Tabla 5. Funciones en OpenMP

Funciones	Descripción
omp_set_num_threads	Fija el número de hilos simultáneos.
omp_get_num_threads	Devuelve el número de hilos en ejecución.
omp_get_max_threads	Devuelve el número máximo de hilos que se pueden ejecutar concurrentemente en una región paralela.
omp_get_ thread_num	Devuelve el identificador del hilo.
omp_get_ num_procs	Devuelve el número de procesadores del equipo (reales o virtuales).
omp_set_ dinamic	Es un booleano que indica si el número de hilos puede crecer o decrecer dinámicamente.

La sintaxis básica para utilizar OpenMP es:

$$\#\ pragma\ omp < directiva > [cláusula[\ , \ldots] \ldots]$$

Las directivas o constructores que se emplean se muestran en la tabla 6:

Tabla 6. Directivas utilizadas en OpenMP

Constructor	Descripción
parallel	Indica la parte del código que será ejecutada por varios hilos.
for	Igual al parallel, pero optimizada para los bucles for.
section	Indica secciones que se ejecutan en paralelo. Cada una es ejecutada por un solo hilo.
single	Región de código ejecutada solamente por un hilo.
master	Región de código ejecutada solamente por un hilo, el padre.
critical	Un solo hilo puede acceder a esta región a la vez.
atomic	Similar al critical, pero se aplica solo a una localidad de memoria.
shared	Indica si una variable es compartida entre los hilos.
private	Indica si una variable es privada entre los hilos. Éstas no son inicializadas.
firstprivate	Indica si una variable es privada entre los hilos. Éstas se inicializan al valor que tenía antes de la directiva.

Cuenta también con cláusulas de *Scheduling (planificación)*, que permiten repartir la carga de trabajo de ciclos iterativos entre los hilos. Existen tres tipos y se muestra gráficamente el trabajo de un vector en 4 unidades de procesamiento y un valor *chunk* (parámetro que indica el tamaño del pedazo) igual a 2:

Static: Antes de ejecutar el bucle, las iteraciones son repartidas de forma equivalente y contigua entre los hilos.

Dynamic: Se utiliza un parámetro *chunk* que define el número de iteraciones contiguas de cada hilo.

Guided: Inicia con un bloque grande y el tamaño se reduce, hasta llegar al tamaño *chunk*. Es una variación de *Dynamic*.

Ilustración 8. Representación de planificadores en OpenMP para ciclos

Message Passing Interface (MPI)

Es una interfaz de paso de mensajes que representa un esfuerzo prometedor de mejorar la disponibilidad de un software altamente eficiente y portable para satisfacer las necesidades actuales en la computación de alto rendimiento a través de la definición de un estándar de paso de mensajes universal.[13]

Es un estándar de programación en paralelo creado en 1993. Es abierto por fabricantes y usuarios. Principalmente, incluye interfaces para FORTRAN, C y C++. La ilustración 9 muestra que MPI se encuentra entre el nivel de aplicación y de Software:

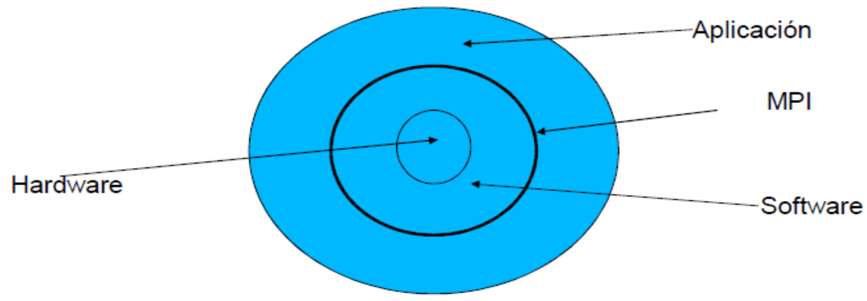


Ilustración 9. MPI es un middleware

La unidad básica son los procesos. Un proceso es una entidad formada por los siguientes dos elementos principales:

---

[13] Indiana University. (2014). *Open Source High Performance Computing*. Octubre 2014, de Open MPI. Sitio web: http://www.open-mpi.org/

1. Imagen binaria del programa, cargada parcial o totalmente en memoria física. La imagen binaria está formada por las instrucciones y datos del programa.
2. Área de memoria para almacenar datos temporales, también conocida como pila.

El funcionamiento, a grandes rasgos, es:

- A cada proceso se le asigna un identificador interno.
- Tienen espacios de memoria independientes.
- Intercambio de información por paso de mensajes.
- Para realizar la comunicación, introduce el concepto de *comunicadores*, que agrupa a los procesos implicados en una ejecución paralela. Estos procesos pueden intercambiarse mensajes.

Las funciones básicas de MPI se muestran en la tabla 7:

Tabla 7. Funciones básicas en MPI

Función	Descripción
MPI_init	Primera llamada de cada uno de los procesos MPI.
MPI_Finalize	Termina la ejecución en MPI y libera recursos.
MPI_Comm_size	Devuelve el número de procesos de un comunicador.
MPI_Comm_rank	Devuelve el identificador de un proceso dentro de un comunicador.
MPI_Send	Un proceso envía datos a otro.
MPI_Recv	Un proceso recibe datos de otro.

Cilk++

Es una extensión del lenguaje C++ para simplificar el desarrollo de aplicaciones que aprovechen de manera eficiente el ambiente de varios procesadores.

Generalmente, resuelve problemas partiéndolos en tareas que pueden ser resueltos independientemente para posteriormente integrar los resultados. Éstas pueden ser implementadas en funciones separadas o por iteraciones en un ciclo.

Cilk++ cuenta con palabras reservadas para identificar las funciones o iteraciones que pueden ser ejecutadas en paralelo.

- *cilk_spawn*: Llama a una función hija que puede ser ejecutada en paralelo con el que la llama (padre).
- *cilk_sync*: Espera a que todas las funciones hijas se completen antes de que el padre prosiga.
- *cilk_for*: Identifica un ciclo *for* cuyas iteraciones se pueden hacer en paralelo.

En la ilustración 10 se muestra la ejecución de un *for* de 8 iteraciones en *clik*:

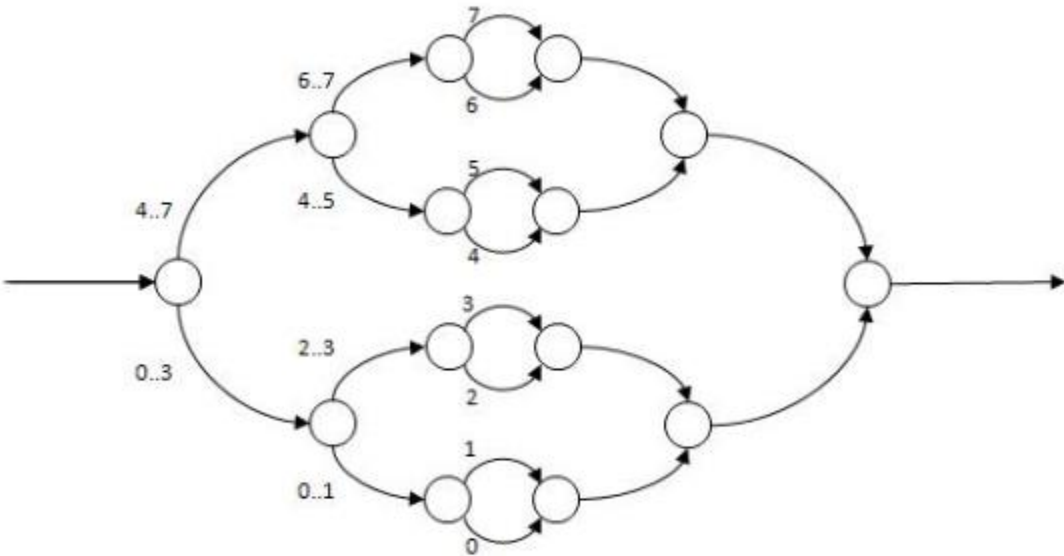


Ilustración 10. Representación de cilk_for

Así como el clásico ejemplo del cálculo de Fibonacci mediante llamadas recursivas (problema que es altamente paralelable). La primera llamada a la función *fib* se puede calcular en paralelo con la segunda, pero se tiene que esperar a que ambos terminen para poder devolver un valor. El código está en la ilustración 11:

```
int fib(int n){
    if(fib < 2)
        return n;
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x + y;

}
```

Ilustración 11. Ejemplo de cilk en el cálculo de la sucesión de Fibonacci

Java Threads

Así como un sistema operativo gestiona los procesos, la Máquina Virtual de Java (JVM) gestiona los Java Threads. Un hilo de Java es una instancia de la clase *java.lang.Thread*. Algunos de los métodos esenciales son mostrados en la tabla 8:[14]

[14] Java Tutorials. (2014). *Concurrency*. octubre 2014, de Oracle. Sitio web: http://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html

Tabla 8. Métodos para manipular Java Threads

Método	Descripción
start()	Comienza la ejecución de un hilo.
run()	Método que contiene el código a ejecutar.
yield()	Establece prioridad.
sleep()	Pausa la ejecución de un hilo.
join()	Permite al hilo "ponerse en la cola de espera" de otro hilo.

Es importante aclarar la diferencia entre los métodos *run* y *start*. En el primero se encuentra el código que va a procesar un hilo, pero éste no es ejecutado hasta que el segundo es llamado. En la ilustración 12, la clase *HellowRunnable* sólo contiene a un método, *run*, el cual contiene el código a ser ejecutado por el hilo.

El mecanismo por el cual un sistema controla la ejecución concurrente de procesos se llama planificación, y en java, éste depende de la prioridad que tengan los hilos. Éstos tienen un valor entero de uno a diez, y por defecto la prioridad de un hilo es igual a la de su padre.

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

Ilustración 12. Clase HelloRunnable

Compute Unified Device Architecture (CUDA):

CUDA es una arquitectura de cálculo paralelo de la empresa NVIDIA que aprovecha la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento extraordinario del rendimiento del sistema.[15]

Los sistemas informáticos están pasando de realizar el "procesamiento central" en la CPU a realizar "coprocesamiento" repartido entre la CPU y la GPU. Para posibilitar este nuevo paradigma computacional, NVIDIA ha inventado la arquitectura de cálculo paralelo CUDA, que ahora se incluye en las GPUs GeForce, ION Quadro y Tesla GPUs, lo cual representa una base instalada considerable para los desarrolladores de aplicaciones.

Un buen indicador de la excelente acogida de CUDA es la rápida adopción de la GPU Tesla para aplicaciones de GPU Computing. En la actualidad existen más de 700 clúster de GPUs instalados en

[15] Nvidia Corporation. (2014). *GPU Computing: The Revolution*. Octubre 2014, de Nvidia. Sitio web: http://www.nvidia.com/object/cuda_home_new.html

compañías Fortune 500 de todo el mundo, lo que incluye empresas como Schlumberger y Chevron en el sector energético o BNP Paribas en el sector bancario.

Por otra parte, la reciente llegada de los últimos sistemas operativos de Microsoft y Apple (Windows 8 y Snow Leopard) está convirtiendo el GPU Computing en una tecnología de uso masivo. En estos nuevos sistemas, la GPU no actúa únicamente como procesador gráfico, sino como procesador paralelo de propósito general accesible para cualquier aplicación.

CUDA intenta aprovechar el gran paralelismo, y el alto ancho de banda de la memoria en las GPU en aplicaciones con un gran costo aritmético frente a realizar numerosos accesos a memoria principal, lo que podría actuar de cuello de botella.

El modelo de programación de CUDA está diseñado para que se creen aplicaciones que de forma transparente escalen su paralelismo para poder incrementar el número de núcleos computacionales. Este diseño contiene tres puntos claves, que son la jerarquía de grupos de hilos, las memorias compartidas y las barreras de sincronización.[16]

La estructura que se utiliza en este modelo está definido por un *grid (malla)*, dentro de la cual hay bloques de hilos que están formados por, como máximo, 512 hilos distintos.

Cada hilo está identificado con un identificador único, que se accede con la variable *threadIdx*. Esta variable es muy útil para repartir el trabajo entre distintos hilos. *threadIdx* tiene 3 componentes (x, y, z), coincidiendo con las dimensiones de bloques de hilos. Así, cada elemento de una matriz, por ejemplo, lo podría tratar su homólogo en un bloque de hilos de dos dimensiones.

Al igual que los hilos, los bloques se identifican mediante *blockIdx* (al igual que los hilos, tienen coordenadas en x, y, z). Otro parámetro útil es *blockDim*, para acceder al tamaño de bloque. En la ilustración 13, se muestra un código que suma dos vectores.

El *kernel* en CUDA es una función la cual incluye la palabra __*global*__ en la declaración. En este caso, se utilizan las variables antes descritas para mapear los identificadores de los hilos en los elementos de dos vectores y guardar la suma de sus elementos en un tercer arreglo. Cuando el *kernel* es llamado, se especifica entre tres pares de picoparéntesis la configuración de ejecución, separando la cantidad de bloques e hilos por bloque con una coma.

En la tabla 9 se muestran las funciones básicas para elaborar un código en CUDA.

[16] Nvidia Corporation. (2014). *CUDA Toolkit Documentation v6.5*. Octubre 2014, de Nvidia. Sitio web: http://docs.nvidia.com/cuda/index.html#axzz3UsKSpImT

```
__global__ void addKernel(  int  *d_a, int *d_b, int *d_result){
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  d_result[idx] = d_a[idx] + d_b[idx];
}

void onDevice( int  *h_a, int *h_b, int *h_result ){

  int *d_a, *d_b, *d_result;

  cudaMalloc( (void**)&d_a, ARRAY_BYTES );
  cudaMalloc( (void**)&d_b, ARRAY_BYTES );
  cudaMalloc( (void**)&d_result, ARRAY_BYTES );

  cudaMemcpy( d_a, h_a, ARRAY_BYTES, cudaMemcpyHostToDevice );
  cudaMemcpy( d_b, h_b, ARRAY_BYTES, cudaMemcpyHostToDevice );

  addKernel<<<BLOCKS,THREADS>>>( d_a, d_b, d_result);

  cudaMemcpy( h_result, d_result, ARRAY_BYTES, cudaMemcpyDeviceToHost );

  cudaFree( d_a );
  cudaFree( d_b );
  cudaFree( d_result );
}
```

Ilustración 13. Ejemplo de suma de arreglos en CUDA

Tabla 9. Funciones básicas en CUDA

Función	Descripción
cudaMalloc	Asigna memoria en el GPU.
cudaMemcpy	Copia memoria; ésta puede ser del GPU al CPU o viceversa, CPU a CPU ó GPU a GPU.
cudaFree	Libera la memoria en el GPU.

Open Computing Language (OpenCL)

Consta de una interfaz de programación de aplicaciones y de un lenguaje de programación. La diferencia esencial con CUDA, es que no es propietario, es decir, el código escrito en OpenCL puede correr en tarjetas gráficas y CPUs de diferentes marcas, como se muestra en la ilustración 14. Sin embargo, CUDA obtiene mejores resultados cuando se comparan estas dos herramientas.[17] El lenguaje está basado en C, eliminando cierta funcionalidad y extendiéndolo con operaciones vectoriales.

Apple creó la especificación original y fue desarrollada en conjunto con AMD, IBM, Intel y NVIDIA. Apple la propuso al Grupo Khronos para convertirla en un estándar abierto y libre de derechos. El

[17] Karimi K. (2013*). A Performance Comparison of CUDA and OpenCL*. Agosto 2014, de D-Wave Systems Inc. Sitio web: http://arxiv.org/ftp/arxiv/papers/1005/1005.2581.pdf

16 de junio de 2008 Khronos creó el Compute Working Group para llevar a cabo el proceso de estandarización. En 2013 se publicó la versión 2.0 del estándar.

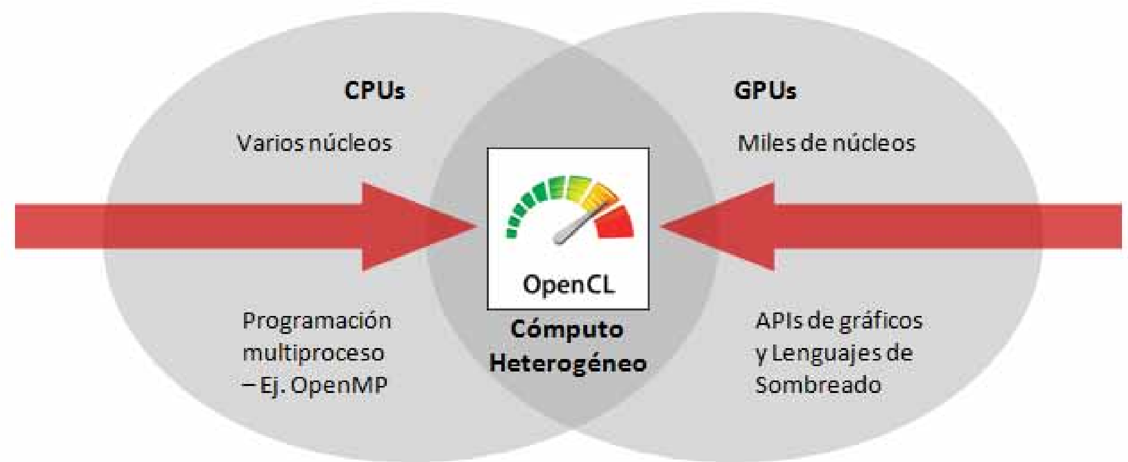


Ilustración 14. OpenCL aprovechando CPUs multicore y GPUs[18]

El modelo utilizado está basado en definir dominios computacionales N-dimensionales (funciones en C) y en ejecutar un *kernel* (muy similar a CUDA). Un ejemplo está en la ilustración 15; la multiplicación de elementos de dos arreglos de forma tradicional y con OpenCL:

Ciclo tradicional

```
void vectorMult(
    const float* a,
    const float* b,
    float* c,
    const unsigned int count)
{
    for(int i=0; i<count; i++)
        c[i] = a[i] * b[i];
}
```

Implementación paralela

```
kernel void vectorMult(
    global const float* a,
    global const float* b,
    global float* c)
{
    int id = get_global_id(0);

    c[id] = a[id] * b[id];
}
```

Ilustración 15. Comparación de un ciclo tradicional y su implementación en OpenCL[19]

---

[18] Gold Standard Group. (2014). *OpenCL*. Septiembre 2014, de Khronos. Sitio web: https://www.khronos.org/opencl/

[19] Pariabi. (2014). Translating Traditional Program Logic to OpenCL. [imagen]. Sitio web: http://www.hafizpariabi.com/2014/02/translating-traditional-program-logic.html

2. Fundamentos del procesamiento paralelo

En este capítulo se hace una revisión de los conceptos, elementos y plataformas de desarrollo tanto de software como de hardware, que se utilizarán como base de análisis para optimizar los algoritmos. Cabe señalar que las tecnologías aquí mostradas no son las únicas que existen, éstas fueron seleccionadas ya que son las más comunes y de más fácil acceso.

2.1 Complejidad, tiempo y eficiencia de algoritmos

En esta sección se explican algunos conceptos relacionados con el cómputo paralelo. A lo largo del capítulo se describen con detalle características inherentes a los algoritmos, al software y al hardware. El objetivo es facilitar la explicación de los algoritmos propuestos en los siguientes capítulos.

Tiempo de ejecución

El tiempo de ejecución se refiere a cuánto tiempo tardó el algoritmo en completarse. Éste es afectado por diversos factores, como la cantidad de datos de entrada, la cantidad de código generado por el compilador para crear el código objeto, la rapidez con la que las instrucciones son ejecutadas por el procesador y la complejidad (descrita más adelante).

Existen dos estudios posibles del tiempo. El primero es una medida a priori, es decir teórica, que consiste en acotar el tiempo de ejecución por arriba o por abajo. La segunda es una medida a posteriori, es decir real, que consiste en tomar el tiempo inicial y el final para calcular la diferencia entre ellos.

Como se mencionó, el proceso de obtener el tiempo de ejecución depende de varios aspectos; se pudiera pensar en calcularlo para diferentes dispositivos, sin embargo esto no sería un buen indicador debido a que está más supeditado a la complejidad.

Complejidad

Como no existe una computadora que sirva de parámetro para medir tiempos de ejecución, éste no puede ser expresado en segundos u otra unidad de tiempo. La complejidad de un algoritmo consiste en el número de instrucciones simples (asignaciones, comparaciones, sumas, restas, multiplicaciones, divisiones, etc.) que se ejecutan con base en los datos de entrada.

En la tabla 10 se muestran los tiempos de ejecución de acuerdo a la complejidad con base en los datos de entrada; consideremos un procesador que ejecuta 1 millón de operaciones por segundo:

Tabla 10. Ejemplos de tiempos de ejecución con base en la complejidad

F(n) \ n	10	20	50	100	Ejemplos
1	0.000001 s	0.000001 s	0.000001 s	0.000001 s	Suma primeros n números (Gauss).
n	0.00001 s	0.00002 s	0.00005 s	0.0001 s	Recorrer una lista.
n log (n)	0.00001 s	0.000026 s	0.000085 s	0.0002 s	Mergesort
n^2	0.0001 s	0.0004 s	0.0025 s	0.01 s	Ordenamiento por burbuja, eliminación Gauss-Jordan.
n^3	0.001 s	0.008 s	0.125 s	1 s	
2^n	0.001 s	1.04 s	35.6 a	2.6 MEU	Serie Fibonacci recursiva.
3^n	0.059 s	58 min	22735 Ma	10^{18} MEU	
n!	3.6 s	77146 a	$6*10^{40}$ MEU	$2*10^{134}$ MEU	Algoritmo del viajero (fuerza bruta).
s : segundos a: años Ma: millones de años MEU: millones de veces Edad del Universo					

¿Por qué tienen esa complejidad?

Suma primeros n números (Gauss): Cuando era niño, se le encargó a Gauss la tarea titánica de sumar los primeros 100 números, pero se dio cuenta de que 1 + 100 es igual a 2 + 99 y a 3 + 98. Como hay 50 de estas parejas, fácilmente dedujo una fórmula para sumar los primeros *n* números.[20] La complejidad es constante, pues sin importar el valor de *n*, el resultado se obtiene de aplicar la fórmula:

$$suma_n = (n+1) * \frac{n}{2}$$

Recorrer una lista: Para llegar al último elemento de una lista, se tiene que pasar por todos los previos. Por ejemplo, para llegar al 4º elemento de una lista, se debe de pasar primero por el 1, 2 y 3; por lo tanto la complejidad es del orden de *n*.

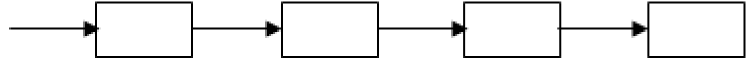


Ilustración 16. Representación de una lista

[20] Sara. (2007). *Gauss y la suma de los n primeros*. Febrero 2015. Sitio web: https://sferrerobravo.wordpress.com/2007/11/24/gauss-y-la-suma-de-los-n-primeros/

Mergesort: El ordenamiento por mezcla se basa en el principio "divide y vencerás". Consiste en dividir recursivamente el arreglo inicial en subarreglos hasta llegar a la unidad básica (arreglos de un elemento) y se comparan de uno en uno, hasta llegar al arreglo solución.[21]

La complejidad está dada por:

- Se tiene que copiar el arreglo inicial a uno auxiliar y cuando se tenga la solución, copiarla al inicial: T(2n).
- Como el arreglo se va dividiendo a la mitad, se obtiene la unidad básica después de un tiempo de T(n/2), y para obtener el arreglo solución se comparan estos valores en un tiempo T(n/2). Es decir, 2T(n/2).
- Al final, obtenemos: $T(n) = T(2n) + 2\,T(n/2)$.
- Por lo tanto:

$$O(n \log (n))$$

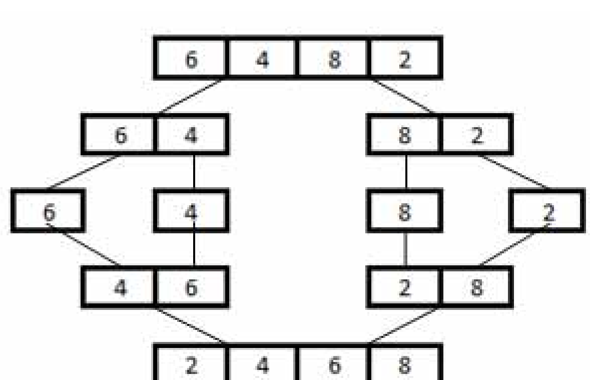


Ilustración 17. Representación de Mergesort

Ordenamiento por burbuja, eliminación Gauss-Jordan: Estos algoritmos tienen esta complejidad básicamente porque en el código del programa existe un ciclo anidado en otro. Es decir, el ciclo interno se ejecuta tantas veces como lo indique el externo.

```
for( i = 0; i < n; i++)
    for( j = 0; j < m; j++)
        b[i][j] = rand() % 2;
```

Ilustración 18. El ciclo interno se ejecuta tantas veces como lo indique el externo

Si $n = m$, se tiene complejidad $O(n^2)$; para tener complejidad de $O(n^3)$, existen tres ciclos anidados, etc.

[21] Department of Computer Science, Kent State University. (2010). *Design and Analysis of Algorithms.* Febrero 2015. Sitio web: http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/mergeSort.htm

Serie Fibonacci recursiva: Para obtener el valor Fibonacci de un número *n*, basta con sumar los valores Fibonacci de *n-1* y *n-2*; a pesar de lo sencillo que parece, la complejidad es muy grande, y se debe a que para calcular *n-1*, se tiene que calcular *n-2* y *n-3*, etc.

Para calcular la complejidad calcularemos las raíces de la ecuación característica asociada a la ecuación en recurrencia:[22]

$$\mathrm{f(n)} = \mathrm{f(n-1)} + \mathrm{f(n-2)}$$

$$cuya\ ecuación\ asociada\ es\ x^2 = x + 1$$

Una de las raíces es:

$$\varphi = \frac{(1+\sqrt{5})}{2} \approx 1.61803$$

Éste es conocido como el número áureo y se cumple que.

$$\varphi^2 = \varphi + 1$$

Así como para cualquier *n* mayor a 2.

$$\varphi^n = \varphi^{n-1} + \varphi^{n-2}$$

Es decir, la progresión geométrica (φ^n) satisface la misma ecuación en recurrencia que la función de Fibonacci f(n) = f(n-1) + f(n-2). Por lo que la complejidad es de:

$$O(\varphi^n)$$

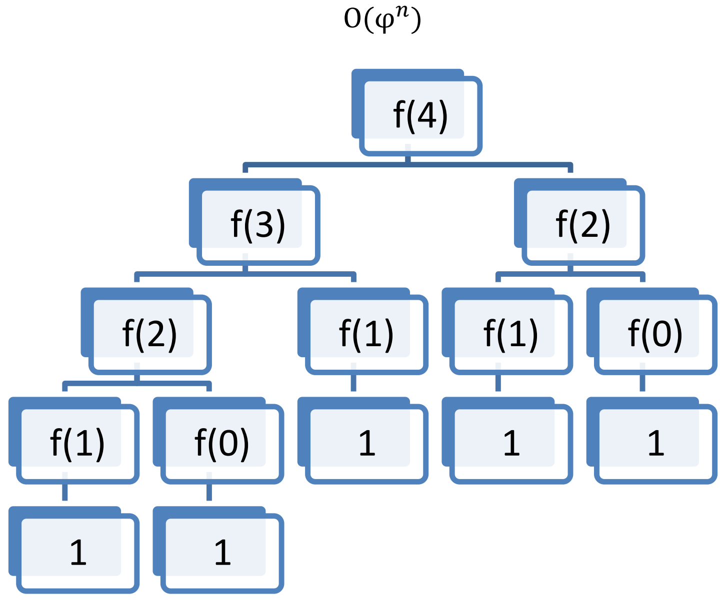


Ilustración 19. Cálculo de Fibonacci. n = 4

[22] Aznar E. (2007). *Números de Fibonacci*. Febrero 2015. Sitio web: http://www.ugr.es/~eaznar/fibo.htm

Algoritmo del viajero (fuerza bruta): Un agente viajero desea visitar un conjunto de ciudades, asignándoles un costo por visitar ciudades contiguas (distancia de traslado entre dos ciudades). Para esta solución se propusieron 2 condiciones: regresar a la misma ciudad de la cual partió y no repetir ciudades.[23] La solución es encontrar el camino más corto posible. La complejidad del algoritmo es factorial.

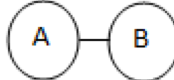


Ilustración 20. Algoritmo con dos ciudades

En la ilustración 20 se tienen dos ciudades y las posibilidades para el viajero son *ABA* o *BAB*. Es decir, sus opciones son 2 ó 2!, mientras que en la ilustración 21 se tienen las opciones *ABCA, ACBA, BACB, BCAB, CABC y CBAC*, que son 6 posibilidades ó 3!.

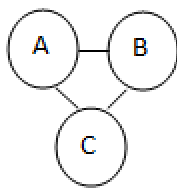


Ilustración 21. Algoritmo con tres ciudades

Eficiencia

Para calcular la eficiencia de un algoritmo, debemos entender lo que es el *speedup*. Es la relación entre el tiempo de ejecución de un algoritmo en una unidad de procesamiento con respecto al que le toma en *n* unidades:

$$Speedup = \frac{T(1)}{T(n_{procesadores})}$$

Generalmente, este valor es mayor a 1. Aunque no es recomendable tomarlo como referencia para decidir si un algoritmo se pueda paralelar o no, es evidente que no conviene procesar un algoritmo en más de una unidad de procesamiento si éste es menor a 1.

La eficiencia la mediremos como la relación del *speedup* en *n* procesadores con respecto a esos *n procesadores*:

$$Eficiencia = \frac{Speedup_{n\ procesadores}}{n}$$

[23] Fuentes A. (2011). *Problema del agente viajero*. Febrero 2015. De Universidad Autónoma del Estado de Hidalgo. Sitio web: http://www.uaeh.edu.mx/scige/boletin/tlahuelilpan/n3/e5.html

2.2 Ley de Amdahl, granularidad y regiones paralelas

Ley de Amdahl

Uno supondría que si un programa se ejecutó en cierto tiempo *t* en una unidad de procesamiento, entonces cuando corra en dos unidades tardará *t/2* y la mitad nuevamente si es en 4. Bueno, esto no es cierto, dado que hay código en todos los programas que no puede ser ejecutado en forma paralela.[24]

La Ley de Amdahl es un modelo matemático que describe la relación entre la aceleración esperada de la implementación paralela de un algoritmo con respecto al serial.

$$T(n) = T(1)\left(B + \frac{1-B}{n}\right)$$

donde

B – porcentaje de la parte estrictamente serial [0, 1].
n - número de hilos en ejecución.
T(n) - tiempo en *n* hilos.
T(1) - tiempo serial.

Es decir, el tiempo en *n* hilos está determinado por el porcentaje del código que es estrictamente serial (B) más el porcentaje del código que se puede realizar en paralelo (1-B) entre el número de hilos en ejecución (n); finalmente, este valor se multiplica por el tiempo de ejecución de manera secuencial T(1).

Entonces, el *speedup* teórico está dado por:[25]

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1-B}{n}\right)} = \frac{1}{B + \frac{1-B}{n}}$$

Digamos, por ejemplo, que un programa que tarda 20 horas en ejecutarse tiene una sección estrictamente serial que tarda una hora (5% del tiempo) y el resto se puede ejecutar en paralelo (95%). Entonces, utilizando cualquier cantidad de procesadores, el programa tardará más de una hora en ejecutarse, por lo que tiene un *speedup* máximo de 20X. Esto se puede observar en la ilustración 22, así como diferentes porcentajes de paralelismo.

---

[24] Díaz G.(2008). *Ley de Amdahl y Ley de Moore*. De Universidad de los Andes. Octubre 2014. Sitio web: http://webdelprofesor.ula.ve/ingenieria/gilberto/paralela/05_LeyDeAmdahlYMoore.pdf
[25] Liu J. (2011). *Estimation of theoretical maximum speedup ratio for parallel computing of grid-based distributed hydrological models*. Octubre, 2013. Sitio web: http://dl.acm.org/citation.cfm?id=2527901

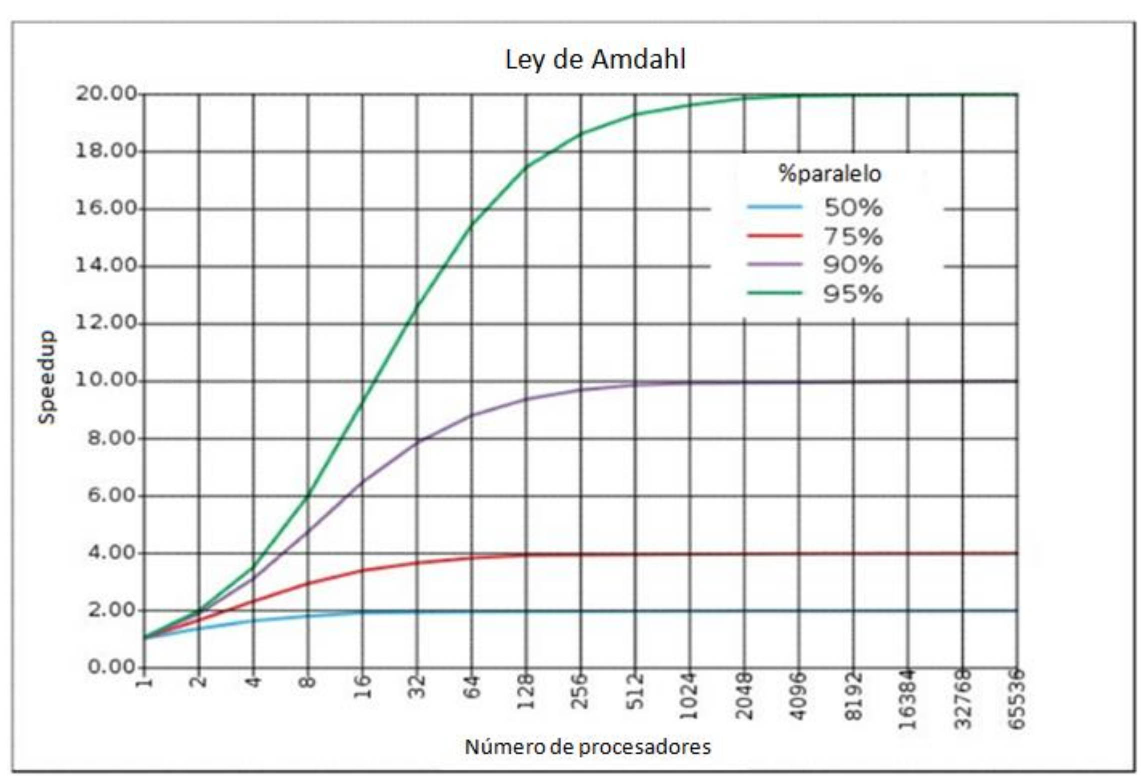


Ilustración 22. Representación gráfica de la Ley de Amdahl y speedup[26]

Niveles de paralelismo y granularidad

El paralelismo se clasifica en diversos niveles atendiendo principalmente al número de instrucciones que son ejecutadas de forma simultánea. Podemos tener los siguientes niveles:[27]

- De tarea o trabajo: Programas completos ejecutándose al mismo tiempo en diferentes procesadores
- De programa: Partes del mismo programa ejecutándose en paralelo en diversos procesadores del sistema
- De instrucción: Las instrucciones independientes se ejecutan al mismo tiempo en diferentes procesadores; o una determinada instrucción se descompone en subinstrucciones y éstas son ejecutadas en paralelo.
- De bit: Paralelismo de más bajo nivel, es invisible para el usuario ya que se logra a través del hardware.

Por otro lado, se le conoce como granularidad al tamaño relativo de las unidades de cómputo que son ejecutadas al mismo tiempo, es decir la granularidad se mide de acuerdo al grosor o a la fineza con que se dividen las tareas en un sistema de cómputo paralelo. Se distinguen los siguientes grados de granularidad:

---

[26] Fundación Wikimedia. (2014). Ley de Amdahl. [imagen]. Sitio web: http://es.wikipedia.org/wiki/Ley_de_Amdahl

[27] Sandoval L. (2012). *Propuesta de un Modelo de Desarrollo de Sistemas de Software de Procesamiento Paralelo/Distribuido*. De Facultad de Ingeniería, Universidad Nacional Autónoma de México.

- Grano fino: Se trabaja a nivel de ciclos e instrucciones.
- Grano medio: Una aplicación es dividida en tareas, funciones o rutinas. Requiere alto grado de coordinación e interacción entre las partes.
- Grano grueso: Grupo de procesos o aplicaciones completamente independientes ejecutándose al mismo tiempo. No existe comunicación entre dichas aplicaciones.

Identificación de secciones paralelas

Para poder paralelizar un programa adecuadamente, primero es necesario analizarlo para saber qué secciones de código se pueden modificar. Una buena técnica es buscar aquellas que tardan más tiempo, como ciclos o funciones con gran cantidad de operaciones. Existen dos tipos de descomposiciones para realizar procesamiento en paralelo. Se puede entender por descomposición a dividir el problema en tareas más simples e independientes:

Descomposición de dominio

Se concentra en partir los datos. Es decir, dividir los datos en "piezas pequeñas" con el mismo tamaño para operar con ellos. Por ejemplo, operación sobre vectores o el cálculo de *pi* utilizando la integral:[28]

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

Para obtener valores muy aproximados y precisos, se calcula el área de la mayor cantidad de rectángulos posible con suma de Riemann; este trabajo puede ser distribuido en varias unidades de procesamiento, como se muestra en la ilustración 23:

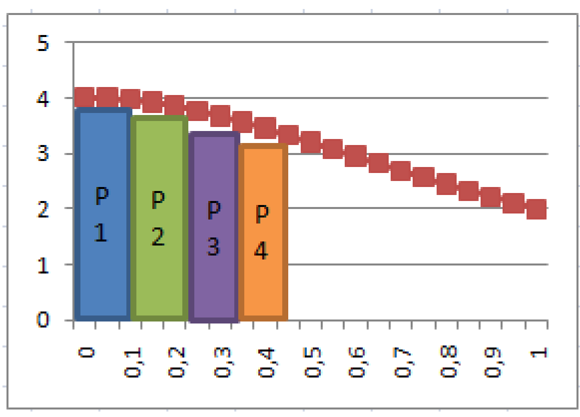


Ilustración 23. Descomposición de Dominio

[28] Guerrero D. (2010). *Programación Distribuida y Paralela*. Universidad de Granada. Septiembre 2014. Sitio web: http://lsi.ugr.es/jmantas/pdp/tutoriales/tutorial_mpi.php?tuto=03_pi

Descomposición funcional

Consiste en dividir el problema en tareas disjuntas (que una no dependa de las demás). En este tipo de descomposición se presentan dos casos:

- Particionamiento completo: Los datos son disjuntos.
- Particionamiento incompleto: Los datos no son disjuntos. Los procesos requieren de comunicación para realizar las tareas.

Un ejemplo de descomposición funcional es el cálculo de Fibonacci de forma recursiva. Por definición, el valor Fibonacci de cualquier número es la suma de los dos anteriores:[29]

$$f(n) = f(n-1) + f(n-2)$$

En la ilustración 24, por ejemplo, si calculamos el valor *f(5)*, tendríamos un árbol de la siguiente forma, donde cada rama puede ser calculada de forma paralela con el resto, pues no hay dependencia:

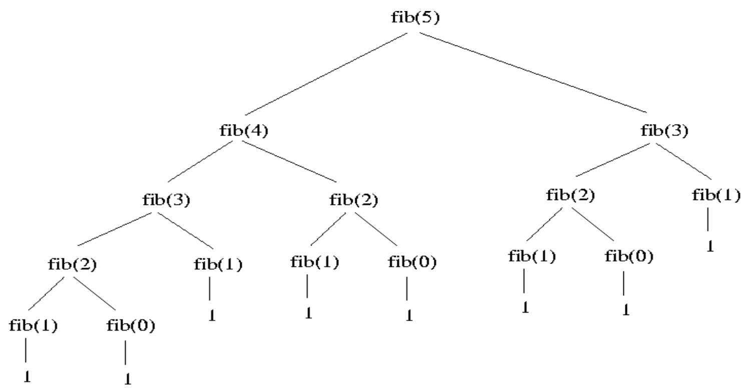


Ilustración 24. Cálculo de Fibonacci con Descomposición Funcional

2.3 Taxonomía de Flynn

La taxonomía de Flynn es una clasificación de las diferentes arquitecturas que implementan los procesadores basada en la forma en que procesan los datos y las instrucciones.[30]

1. SISD (Single Instruction Single Data): Es la versión convencional del computador de Von Neumann, el cual solo consta de una unidad de procesamiento, por lo que solo puede procesar una sola instrucción con un conjunto de datos a la vez.

[29] Cáceres A. (2005). *La serie Fibonacci*. Agosto 2014, de CINVESTAV Sitio web: http://computacion.cs.cinvestav.mx/~acaceres/courses/estDatosCPP/node38.html

[30] Idelsohn S. (2004). *Parallel Processing: Fynn´s Taxonomy.* Septiembre 2014. Sitio web: https://web.cimne.upc.edu/groups/sistemes/Servicios%20de%20calculo/Barcelona/public/14716537-Flynns-Taxonomy-and-SISD-SIMD-MISD-MIMD.pdf

2. SIMD (Single Instruction Multiple Data): A partir de esta arquitectura se implementan dos o más unidades de procesamiento. Esta arquitectura contiene un único bus de control, lo que restringe a que todas las unidades de procesamiento hagan una misma operación pero con diferentes datos cada una.
3. MISD (Multiple Instruction Single Data): Esta Arquitectura es la opuesta a SIMD, tiene un único bus de datos, pero no de control, lo que permite ejecutar diferentes operaciones a un mismo conjunto de datos a la vez.
4. MIMD (Multiple Instruction Multiple Data): Esta es la arquitectura más compleja, la cual tiene buses independientes para cada unidad de procesamiento, lo que permite que cada unidad haga una operación completamente independiente; diferentes operaciones a diferentes datos.

En este capítulo solo se tratarán dos elementos en hardware, que si bien no son los únicos, serán los que ocuparemos a lo largo del ejercicio.

2.4 Procesadores multinúcleo

El procesador es la unidad más importante dentro de un computador, el cual se encarga de procesar y transformar los datos almacenados en memoria. Desde sus inicios, su arquitectura y elementos se han mantenido constantes, salvo algunas pequeñas variaciones para optimizar su funcionamiento.

Las partes principales de un procesador son:[31]

Unidad Aritmética Lógica (ALU): Es el elemento responsable de realizar operaciones binarias aritméticas (suma, resta, división, multiplicación, etc.) y lógicas (AND, OR, XOR, etc.) entre números binarios. Actualmente esta unidad ha sido suplantada por las FPU (Unidad de Punto Flotante) las cuales son capaces de ejecutar las mismas operaciones, pero con números de coma flotante y no solo enteros; estas unidades contienen varias ALUs integradas.

Unidad de Control (CU): Es la unidad que cuida y mantiene un funcionamiento armonizado entre todos los elementos que conforman el procesador, entre sus tareas más importantes esta unidad dicta qué operaciones realiza la ALU y sobre qué datos, además controla los datos que salen y entran para su procesamiento. Los procesadores modernos separan esta unidad de control, pudiendo tener múltiples unidades específicas para cada tarea a controlar.

Memoria interna: Consiste en bloques de memoria de escaso tamaño pero capaz de trabajar a grandes velocidades, es usada en los cálculos internos o intermedios del procesador y agiliza el procesamiento de las instrucciones, evitando el cuello de botella que generaría usar directamente la memoria RAM. LA mayoría de los procesadores actuales implementan dos niveles de memoria:

[31] Smeen M. (2010). *CPU Architecture*. Octubre 2014. Sitio web: http://pgdcce.blogspot.mx/2010/09/cpu-architecture.html

- Cache L1: se divide en dos partes, para instrucciones y para datos, es el nivel más bajo de memoria y el que está en contacto directo con la CU y la ALU.
- Cache L2: es una memoria de mayor tamaño que la L1, y permite almacenar datos e instrucciones próximos a ejecutarse o ya procesados próximos a salir del procesador.

Controlador Entrada/Salida: Este elemento permite obtener y entregar datos a partir de un medio de almacenamiento externo, por ejemplo la memoria RAM. Esto es necesario ya que la memoria interna es muy pequeña para tener programas complejos, es por eso que se guardan en una memoria de mayor capacidad fuera del procesador.

Bus interno: Es el medio por el cual todos los elementos mencionados anteriormente se comunican, generalmente este bus se divide en tres partes:

- Bus de datos: Permite transición de los datos a operar, desde el exterior (Memoria RAM), hasta la memoria interna, y de ésta a la ALU.
- Bus de direccionamiento: Permite la localización de un dato dentro de la memoria RAM o interna para poder ser direccionado y éste sea depositado en el bus de datos.
- Bus de instrucciones: Este bus parte de la unidad de control, y es el medio por el cual manda las instrucciones o comandos a los diferentes elementos para coordinar su funcionamiento. En la ilustración 25 se muestra un esquema de éstos.

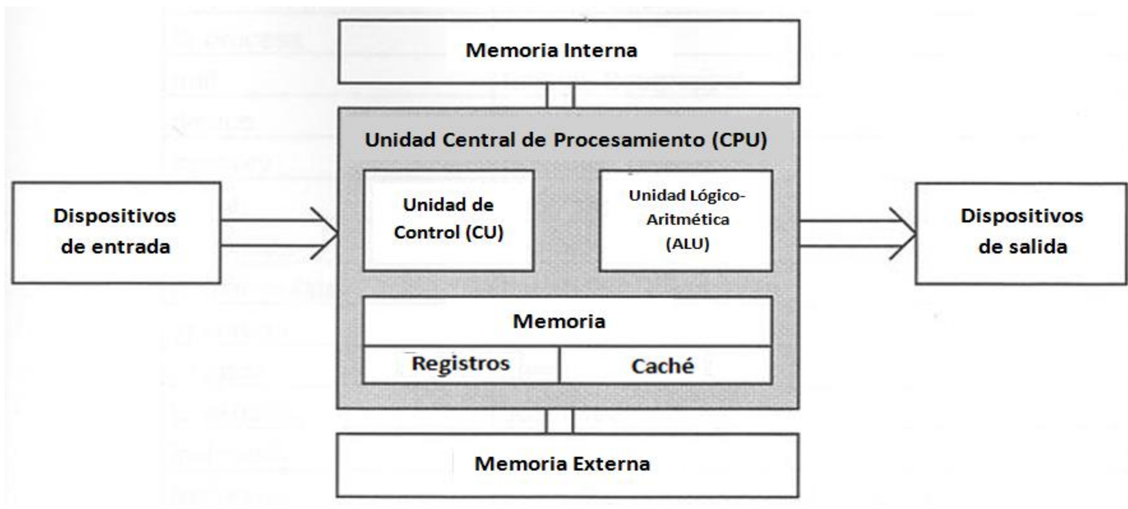


Ilustración 25. Esquema básico de un procesador[32]

Esta configuración ha sufrido pocos cambios y sólo han sido alterados dos elementos para mejor su rendimiento:

- Se ha elevado la frecuencia de MHz a GHz, logrando que las instrucciones se ejecuten en un tiempo menor debido a que un ciclo de reloj dura menos.
- Se ha modificado la arquitectura interna de la ALU o FPU y de la unidad de control, permitiendo que una instrucción determinada necesite un menor número de ciclos de reloj para completarse.

[32] Ibídem.

En la década de los noventa, se empezó la investigación de nuevas formas de mejorar el rendimiento de las computadoras dando origen a los computadores con múltiples unidades de procesamiento.

Las primeras computadoras de multiprocesamiento y algunas actuales tienen dos o más procesadores independientes, los cuales comparten la memoria RAM como canal de comunicación, ya sea para traspaso de datos o tareas de sincronización. Estos procesadores, al ser completamente independientes, pueden ejecutar sus propias instrucciones sobre diferentes datos o los mismos inclusive.

Esta arquitectura de multiprocesamiento tiene varias desventajas, entre ellas es que los procesadores dependen de un programa alojado en la memoria RAM para poder ser sincronizados y que además ésta funge como canal de comunicación pudiendo ser muy lenta para algunas tareas.

Debido a esto, se generó una nueva arquitectura de hardware en la cual, dentro de un mismo encapsulado se cuenta con dos o más ALU´s o FPU´s las cuales pueden estar controladas por una o más unidades de control. La conjunción de una ALU o FPU con una CU (Unidad de Control) se le llama *core* o núcleo, aunque muchas veces no es necesario una unidad de control por núcleo. Para mejorar la comunicación entre unidades de procesamiento, cada núcleo cuenta con uno o dos bloques de memoria para cálculos e instrucciones internos, pero además, dentro de la memoria interna se cuenta con un bloque de mayor tamaño accesible para todos los núcleos, de tal forma que permite compartir datos, sincronización y comunicación entre núcleos.[33]

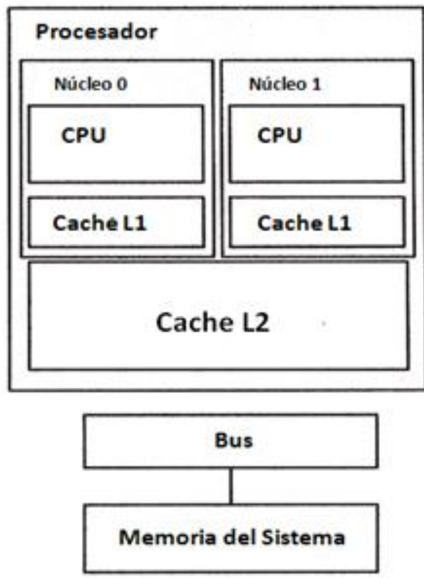


Ilustración 26. Procesador multinúcleo

[33] Domeika M. (2006). *Development and Optimization Techniques for Multicore Processors*. Octubre 2014. Intel Corp. Sitio web:. http://www.embedded.com/design/mcus-processors-and-socs/4006702/Development-and-Optimization-Techniques-for-Multicore-Processors

Esta arquitectura es similar a la que se utiliza actualmente en los procesadores de Intel, y sobre éstos correremos las herramientas de software para paralelar nuestros algoritmos.

2.5 Tarjetas Gráficas

La tarjeta gráfica es un elemento de software que anteriormente era una expansión, pero actualmente forma parte de los componentes estándar de una computadora.[34] Tiene la función de procesar y mostrar los gráficos a nuestro monitor, quitándole este trabajo al procesador permitiendo que éste se enfoque en tareas más importantes.

El procesamiento de gráficos a bajo nivel implica, en su mayoría, operaciones de coma flotante, por lo que las tarjetas gráficas están optimizadas para este tipo de operaciones.

La arquitectura de una tarjeta gráfica difiere bastante a la de un procesador convencional, ya que ésta se basa en el modelo circulante, facilitando el trabajo en paralelo. [35]Las tarjetas gráficas se componen, al igual que los procesadores de 4 elementos básicos:

- FPU o ALU: El igual que en los procesadores, es la unidad que ejecuta las operaciones sobre los datos, ya sean aritméticos o lógicos. En el caso de las tarjetas gráficas, están optimizados para hacer operaciones de coma flotante. Cada tarjeta gráfica puede tener cientos o miles de estas unidades.
- Unidad de control: Esta unidad permite la sincronización y correcto funcionamiento de cada FPU o ALU. En una tarjeta gráfica puede haber una sola unidad de control para todas las unidades de procesamiento, o puede haber varias que controlen a un grupo determinado de éstas, creando bloques de procesamiento.
- Memoria interna: Al igual que en un procesador, las tarjetas gráficas cuentan con diferentes niveles de memoria:
    - A nivel ALU o FPU: sólo para cálculos locales en cada unidad de procesamiento.
    - A nivel bloque: Un grupo determinado de unidades de procesamiento pueden acceder a este nivel de memoria.
    - A nivel general: Todas las unidades de procesamiento y sus respectivos bloques tienen acceso a este espacio de memoria. Esta memoria es la de mayor tamaño y comúnmente se le llama memoria de video.

[34] Orts S. (2005). *Introducción a la Computación paralela con GPUS*. Septiembre 2014. Universidad de Alicante. Sitio web: http://www.dtic.ua.es/jgpu11/material/sesion1_jgpu11.pdf
[35]. Charte F. (2009). *Tipos de Shaders - Procesadores de Geometría*. Septiembre 2014. Sitio web: http://fcharte.com/Default.asp?noticias=2&a=2009&m=12&d=15

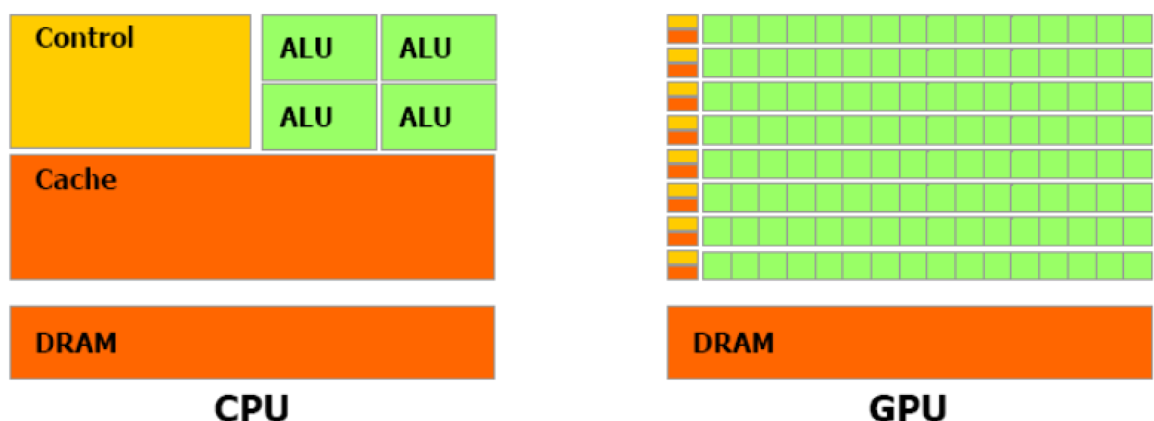


Ilustración 27. CPU vs GPU[36]

En su origen, las tarjetas gráficas eran exclusivamente para el procesamiento de gráficos, pero nuevas técnicas en software y hardware han permitido ejecutar programas en los cuales se requiera el procesamiento de coma flotante a una gran cantidad de datos. Las tarjetas gráficas son excelentes para estas tareas ya que sus unidades de procesamiento están optimizadas para las operaciones de coma flotante, y como tienen una gran cantidad de estas unidades, es posible trabajar o ejecutar un mismo set de instrucciones a una gran cantidad de datos al mismo tiempo.

2.6 Infraestructura utilizada

En este apartado, describiremos las características del hardware que empleamos para probar los códigos. Hemos separado a CPUs en la tabla 11 y GPUs en la 12:

Tabla 11. Especificaciones de CPUs

Intel Xeon E5620[37]		
# de núcleos	4	
# de hilos	8	
Frecuencia del reloj	2.4 GHz	
Instrucciones	64 bits	
Fecha de lanzamiento	Marzo 2010	

[36] Pankaj. (2008). CPU vs GPU. Septiembre 2014. E2matrix Research Lab. [imagen]. Sitio web: http://www.e2matrix.com/blog/?p=133
[37] Intel® Xeon® Processor E5620. Intel. Consultado en marzo 2015. http://ark.intel.com/products/47925/Intel-Xeon-Processor-E5620-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI#@specifications.

Intel i7 4770k[38]	
# de núcleos	4
# de hilos	8
Frecuencia del reloj	3.5 GHz
Instrucciones	64 bits
Fecha de lanzamiento	Junio 2013
Intel i5 4200U[39]	
# de núcleos	2
# de hilos	4
Frecuencia del reloj	1.6 GHz
Instrucciones	64 bits
Fecha de lanzamiento	Junio 2013

Tabla 12. Especificaciones de GPUs

NVIDIA Tesla K20c[40]	
# de núcleos	2688
Frecuencia del reloj	732 MHz
Memoria	6 GB
Fecha de lanzamiento	Noviembre 2012
NVIDIA GeForce GTX 680[41]	
# de núcleos	1536
Frecuencia del reloj	1006 MHz
Memoria	2 GB
Fecha de lanzamiento	marzo 2012
NVIDIA Tegra K1[42]	
# de núcleos	192
Frecuencia del reloj	-
Memoria	2 GB
Fecha de lanzamiento	julio 2012

[38] Intel® Core® i7 4770k. Intel. Marzo 2015. Sitio web: http://ark.intel.com/products/75123/Intel-Core-i7-4770K-Processor-8M-Cache-up-to-3_90-GHz.
[39] Intel® Core® i5 4200U.Intel.Marzo 2015. Sitio web: http://ark.intel.com/es/products/75459/Intel-Core-i5-4200U-Processor-3M-Cache-up-to-2_60-GHz
[40] Tesla K20X GPU Accelerator. Board Specification. Marzo 2015. Sitio web: http://www.nvidia.com/content/PDF/kepler/Tesla-K20X-BD-06397-001-v07.pdf
[41] Nvidia GeForce GTX 680. NVIDIA. Marzo 2015. Sitio web: http://www.nvidia.es/object/geforce-gtx-680-es.html
[42] NVIDIA Tegra K1. NVIDIA. Marzo 2015. Sitio web: www.nvidia.es/object/tegra-k1-processor-es.html.

NVIDIA GeForce 740M[43]		
# de núcleos	384	
Frecuencia del reloj	810 MHz	
Memoria	2 GB	
Fecha de lanzamiento	abril 2012	

[43]NVIDIA GeForce TM 740M.Notebook Check. Marzo 2015. Sitio web: http://www.notebookcheck.net/NVIDIA-GeForce-GT-740M.89900.0.html.

3. Algoritmos de ordenamiento

En este capítulo iniciaremos el análisis de algoritmos a fin de optimizarlos en un ambiente de procesamiento paralelo, basándonos en conceptos, herramientas y plataformas de desarrollo ad hoc.

La ordenación es una operación que consiste en disponer de un conjunto de datos y acomodarlos, ya sea en orden ascendente o descendente, con respecto a una llave. Por ejemplo, en una guía telefónica se tienen datos como nombre, número de teléfono, dirección, código postal, etcétera, los cuales están ordenados de forma ascendente por la llave "nombre", comenzando con el apellido paterno.

De acuerdo a la ubicación de los datos, existen dos tipos de ordenamiento:[44]

- Interno: Cuando los datos están almacenados en un arreglo, lista ligada o árbol.
- Externo: Cuando los datos están en un archivo.

En esta sección hablaremos de ordenación interna. En ésta, existe una gran cantidad de algoritmos, donde lo más importante es saber cuál elegir. Ponemos tres de ellos con diferentes características, donde generalmente se consideran sólo dos:

- Menor tiempo de ejecución.
- Menor número de instrucciones o menor dificultad de programación.

Se dice que un algoritmo de ordenación A es mejor que otro B si hace menos comparaciones entre los elementos.

[44] Díaz A. (2008). *Análisis y Complejidad de Algoritmos*. Septiembre 2014. CINVESTAV. Sitio web: http://delta.cs.cinvestav.mx/~adiaz/anadis/Sorting2.pdf

3.1 Algoritmo de ordenamiento por cuenta

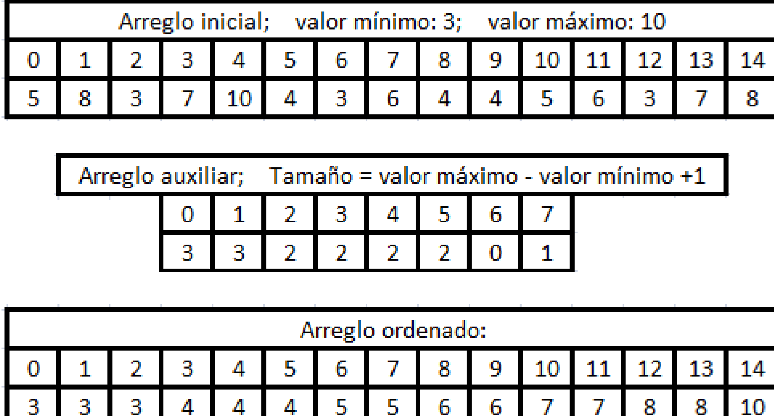
Descripción

Este algoritmo fue creado por Harold H. Steward en 1954.[45]

Se trata de un algoritmo de ordenamiento que cuenta las ocurrencias de cada elemento de un arreglo a ordenar en un arreglo auxiliar. Tiene las siguientes características:

- Los elementos a ordenar deben ser contables (por ejemplo, los números enteros).
- Debe tener un rango de números definido, es decir, un valor máximo y uno mínimo.

Para mostrarlo, veremos el siguiente ejemplo:



Arreglo inicial; valor mínimo: 3; valor máximo: 10														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5	8	3	7	10	4	3	6	4	4	5	6	3	7	8

Arreglo auxiliar; Tamaño = valor máximo - valor mínimo +1							
0	1	2	3	4	5	6	7
3	3	2	2	2	2	0	1

Arreglo ordenado:														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	3	3	4	4	4	5	5	6	6	7	7	8	8	10

Ilustración 28. Ordenamiento por cuenta

Como entrada se tiene un arreglo de enteros desordenado con un valor mínimo 3 y máximo de 10. Entonces creamos un arreglo auxiliar de tamaño: máximo - mínimo +1, donde se contarán las ocurrencias de cada elemento. Para hacer esto, se mapea con:

$$Arreglo_{auxiliar}[\,Arreglo_{inicial}[\,i\,] - \,mínimo\,] + +;$$

Es decir, el valor mínimo corresponde al elemento 0 del arreglo auxiliar y el máximo al último. Se puede observar que este arreglo debe ser inicializado en 0.

Una vez contados los elementos, se vacían en otro arreglo al que llamamos ordenado, donde simplemente se barren todos los elementos hasta agotar los que se contaron, como se muestra en la ilustración 29.

[45] Aldana C. (2010). *Algoritmos de Ordenamiento: Countingsort*. Octubre 2014 Sitio web: http://upcanalisisalgoritmos.wikispaces.com/file/view/ALGORITMOS+DE+ORDENAMIENTO+COUNTINGSORT.pdf

Implementación serial

```
void contar(int *a, int *b){
    int i;
    for(i = 0; i < tamano; i++)
        b[ a[i]-min ]++;
}
```

Ilustración 29. Función contar

Se considera que se tiene en arreglo *a*, que puede estar ordenado o no y de un arreglo *b*, previamente inicializado en ceros. Éste tiene el tamaño de $valor_{máximo}$- $valor_{mínimo}$+1. Si esta diferencia es muy grande, entonces el tamaño de *b* lo es también, lo que se convertiría en un inconveniente. Por el contrario, si la diferencia es pequeña, el algoritmo utiliza muy poca memoria.

La función de la ilustración 30, cuenta las ocurrencias de cada elemento dentro de *a*.

$$Entonces, la\ función: O(n)$$

```
void ordenar(int *b, int *c){
    int i, j=0, k=0;
    for(i = 0; i < (max - min + 1); i++){
        while( b[i] > 0 ){
            c[j] = i + min;
            b[i]--;
            j++;
        }

}
```

Ilustración 30. Función ordenar

Posteriormente (opcional), si se desea, se puede vaciar en un tercer arreglo las ocurrencias de cada elemento. Esta función tiene complejidad *O(max - min + 1)*, pero como es opcional, se considera solamente la función contar.

3.2 Algoritmo de ordenamiento Shell

Descripción

El algoritmo Shellsort es una mejora al algoritmo de inserción (inserta elemento por elemento a su posición final, haciendo un movimiento a la vez), el cual pretende mejorar la eficiencia del algoritmo cuando tenemos un conjunto de datos muy dispersos.[46]

El algoritmo pretende evitar que los elementos sean trasladados a su posición final con un movimiento a la vez, dando saltos de mayor tamaño definido por un factor “k”. De esta manera se ahorra tiempo en la colocación de un elemento que está muy lejos de su posición ya que evitamos el movimiento uno a uno. Un ejemplo se muestra en la tabla 13:

Arreglo inicial: N=10

Tabla 13. Arreglo inicial

Inicial:	98	77	26	94	10	82	22	7	47	22

El factor k puede definirse aleatoriamente y definir un valor arbitrario para decrementarlo, pero se ha comprobado que el algoritmo es más eficiente si el valor k empieza desde la mitad del tamaño del arreglo y decrece con un factor de 2. La tabla 14 muestra el resultado dividir entre 2:

Primera pasada: **k=N/2 =5**

Tabla 14. Primera pasada

98	77	26	94	10
82	22	7	47	22

Ordena: La comparación se realiza con el elemento *(i)* y el elemento *(i+k)*, intercambiando posiciones si resulta mayor el elemento en el índice (i), como se muestra en la tabla 15.

Tabla 15. Ordenamiento

82	22	7	47	10
98	77	26	94	22

Aquí podemos observar las virtudes del algoritmo ya que el elemento con el número (7) se movió 5 lugares dejándolo más cerca de su posición final.

---

[46] Commons C.. (2009). *Shell sort*. Octubre 2014, de Conoce3000 Sitio web: http://www.conoce3000.com/html/espaniol/Libros/PascalConFreePascal/Cap08-03-Ordenamiento%20Shell%20%28Shell%20sort%29.php

La tabla 16 tiene al arreglo después de la primera pasada:

Tabla 16. Arreglo después de la primera pasada

Arr:	82	22	7	47	10	98	77	26	94	22

Segunda Pasada: **k=k/2=2**

En la tabla 17, el factor “k” ha cambiado a un valor de 2, por lo que el salto será de dos unidades.

Tabla 17. Segunda pasada

82	22
7	47
10	98
77	26
94	22

Ordena: En la tabla 18 se tiene un mayor número de elementos por colocar con saltos más pequeños. Aquí se puede ver el comportamiento del algoritmo de inserción normal.

Tabla 18. Ordenamiento

7	22
10	22
77	26
82	47
94	98

Arreglo después de la penúltima pasada, se puede observar que algunos elementos ya están en su posición y otros solo necesitan un número reducido de movimientos para colocarse en la posición final, como se muestra en la tabla 19.

Tabla 19. Arreglo después de la penúltima pasada

Arr:	7	22	10	22	77	26	82	47	94	98

Tercera Pasada: k=k/2=1

El algoritmo termina hasta que “k” toma el valor de 1. Esto significa que la última pasada será exactamente igual al algoritmo de inserción.

Ordena: Algoritmo completamente ordenado en la tabla 20 después de la última pasada:

Tabla 20. Arreglo después de la última pasada

Arr:	7	10	22	22	26	77	47	82	94	98

Implementación serial

```
//Shellsort serial
void shellsort(int *a,int tam){
    for (int k = tam / 2; k > 0; k /= 2){
        for (int i = k; i < tam; i++){
            for (int j = i - k; j >= 0 && a[j] > a[j + k]; j -= k) {
                int temp = a[j];
                a[j] = a[j + k];
                a[j + k] = temp;
            }
        }
    }
}
```

Ilustración 31. Código serial

El algoritmo Shellsort, cuyo código se muestra en la ilustración 31, es una modificación al algoritmo de inserción el cual nos garantiza brincos más grandes, y por ende un menor tiempo en la colocación de los elementos a su posición original. Dependiendo del tamaño de los brincos y su variación se pueden tener diferentes complejidades, tal como lo se describe en la tabla 21:

Tabla 21: Complejidad

Nombre	Patrón	Ejemplo	Complejidad del algoritmo:
Shell	$\frac{N}{2^k}$	$\frac{N}{2},\frac{N}{4},\frac{N}{8}\cdots$	$O(n^2)$
Hibbard	2^k-1	1,3,7,15	$O(n^{\frac{3}{2}})$
Pratt	Números sucesivos de la forma 2^p3^p	1,2,3,4,6,8,9,12	$O(n\,log^2(n))$

```
for i in range(N-1)
    j = i;
    while j > -1
        if a[j] > a[j+1]
        tmp = a[j]
        a[j] = a[j+1]
        a[j+1] = tmp
    j = j-1
print str(a)
```

Ilustración 32. Algoritmo de inserción

En el peor de los casos y con la distribución de Shell, este algoritmo se comporta exactamente igual al algoritmo de inserción mostrado en la ilustración 32.

3.3 Algoritmo de ordenamiento por casilleros

Descripción

Este algoritmo es un método previo que se aplica a nuestro conjunto de datos para que otro algoritmo (o el mismo *bucketsort* recursivamente) ordenen varios conjuntos de datos con un número reducido de elementos.[47]

El algoritmo *bucketsort* genera casilleros, los cuales solo pueden contener números de un determinado rango, y distribuye nuestros elementos iníciales en cada uno de los casilleros. De esta forma un algoritmo auxiliar se dedica a ordenar casillero por casillero.

Ejemplo. Arreglo inicial en la tabla 22:

Tabla 22. Arreglo inicial

35	89	17	13	53	37	60	66	26	79

Creación de los casilleros: El número de casilleros se escoge arbitrariamente, y puede irse probando varios números para ver la opción que más nos convenga. Para este ejemplo se utilizaran 5 bloques. Se representan en la tabla 23:

Tabla 23.Creación de los casilleros

N° de casillero	1	2	3	4	5
Rango	0-19	20-39	40-59	60-79	80-99

Llenado de los casilleros: Cada elemento se coloca en el casillero que le corresponda según el rango definido, tal como en la tabla 24:

Tabla 24. Llenado de los casilleros

N° de casillero	1		2			3	4			5
Elementos	17	13	35	37	26	53	60	66	79	89

Ordenamiento: En la tabla 25, cada casillero se ordena individualmente con un algoritmo alterno:

Tabla 25. Ordenamiento interno

N° de casillero	1		2			3	4			5
Elementos	13	17	26	35	37	53	60	66	79	89

Finalmente, en la tabla 26 se concatenan los elementos de cada casillero en orden:

Tabla 26. Concatenación de bloques

13	17	26	35	37	53	60	66	79	89

---

[47] Black P. (2009). *Bucket sort*. Octubre 2014, de NIST Sitio web: http://xlinux.nist.gov/dads/HTML/bucketsort.html

Implementación serial

```
//LLenado de los casilleros
for (int i = 0; i < N; i++){
    tc1[(int)(a[i] / (R/B))].push_back(a[i]);
    tc2[(int)(a[i] / (R/B))].push_back(a[i]);
}
```

Ilustración 33. Asignación de casilleros

Donde:

- R: es el rango.
- B: es el número de bloques.
- tc1 y tc2: es el conjunto de bloques.
- a: es el arreglo original

El algoritmo *bucketsort* mostrado en la ilustración 33 hace una comparación de cada elemento con cada bloque para determinar en cuál pertenece. En la tabla 27 se analiza la complejidad.

Tabla 27. Complejidad de asignación de casilleros

Peor de los casos	Mejor de los casos
$O(n)$	$O(cn)$

Donde *c* es el número de casilleros

El utilizar otro algoritmo para ordenar los casilleros individualmente, este influye en el cálculo de la complejidad del algoritmo, obteniendo la tabla 28:

Tabla 28. Complejidad de ordenamiento y concatenación

Peor de los casos	Mejor de los casos
$O(n * D(n/c))$	$O(cn * D(n/c))$

Donde *D()* es la complejidad del algoritmo utilizado.

En dado caso que el algoritmo tenga el mismo número de casilleros que de elementos, la complejidad disminuye notablemente:

$$O(n)$$

3. 4 Selección del algoritmo a paralelar y justificación

Para los algoritmos de ordenamiento, decidimos seleccionar el algoritmo *shellsort* y por casilleros.

Justificación:

El ordenamiento por casilleros es un claro ejemplo de la metodología divide y vencerás, el cual divide nuestros elementos en bloques completamente independientes entre sí. Esto, en teoría, nos facilitaría la implementación del algoritmo mediante una herramienta cómputo paralelo.

Como se comentó en la descripción del algoritmo de ordenamiento por casilleros, se basa de un algoritmo auxiliar para el ordenamiento interno de los casilleros. Para este caso utilizaremos el *shellsort* ya que ha probado ser un algoritmo eficiente.

3.5 Análisis e implementación paralela con OpenMP

El algoritmo por casilleros se puede resumir en unos cuantos pasos:

- División de nuestro dominio de datos en cada casillero
- Ordenamiento individual de cada casillero
- Unión del resultado de cada casillero

Estos pasos deben de ser ejecutados uno a uno y sin alterar su orden, de esta manera se garantiza el correcto funcionamiento del algoritmo así como resultados concisos y sin variaciones para una misma entada.

Como todo algoritmo, la entrada y salida de datos se ejecuta de manera serial, pero en este algoritmo es posible paralelizar lo siguiente pasos:

1. División de domino: En este punto, de manera serial, se iba recorriendo todo el arreglo de elementos, uno a uno y colocándolos en sus respectivos bloques. Esta parte es altamente paralelizable ya que cada unidad de procesamiento puede tomar el control de uno o más bloques y cada uno puede tomar los datos que le pertenecen.
2. Ordenamiento individual por casillero: En modo serial, los bloques eran ordenados uno a uno. De manera paralela es posible ordenar varios bloques simultáneamente ya que son completamente independientes entre sí.

En estos dos pasos es altamente recomendable paralelizar ya que son las partes que más tiempo consumen y cada una de estas actividades son completamente independientes gracias a el propio algoritmo (manejo de bloques completamente independientes entre sí).

Este algoritmo contiene un conjunto mínimo de pasos los cuales son completamente seriales, por tal motivo una división de trabajo funcional sería imposible. La propia naturaleza del algoritmo nos indica que una división de dominio para paralelizar sería lo más adecuado; esto debido a que el

algoritmo maneja bloques, los cuales contiene una porción de los datos y trabajan con ella de manera independiente a los demás.

Como se muestra en la ilustración 34 cada procesador toma a uno o más bloques para su ejecución y éstos a su vez toman un conjunto de datos.

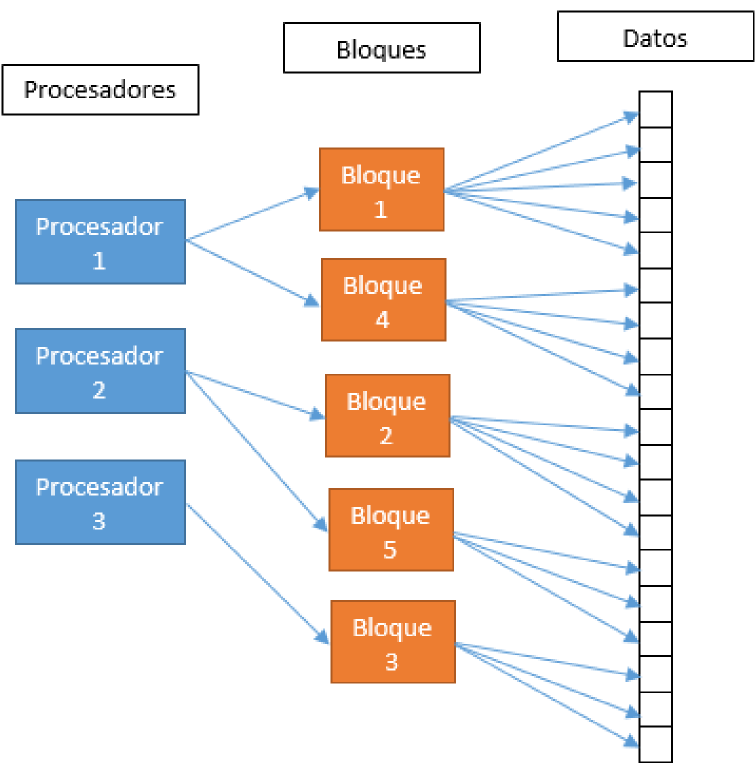


Ilustración 34. Representación de la ejecución en paralelo

En el funcionamiento del algoritmo se puede observar que todos los bloques generados hacen los mismos pasos:

- Selección de datos con los que trabajara
- Ordenamiento
- Salida de los datos

Cada uno de estos pasos los hace cada bloque indistintamente pero con un conjunto de datos diferente, de tal manera que al final de sus operaciones cada uno tendrá una porción del arreglo ordenado. Esto nos recuerda a la taxonomía de Flynn y en específico a SIMD.

El hardware y software que usaremos como herramienta para paralelizar será un procesador multinúcleo y OpenMP. Esto debido a que OpenMP nos permite realizar una división de tareas adecuada, de tal forma que cada núcleo del procesador ejecute un bloque diferente, ordenando varios bloques simultáneamente.

Implementación serial de los dos algoritmos para ordenar un arreglo de N elementos en la ilustración 35.

```
int *a,*b;
double st,sp;
std::vector<int> tc1[B],tc2[B];
a = arreglo(N, R, 'r');

//LLenado de los casilleros
for (int i = 0; i < N; i++){
    tc1[(int)(a[i] / (R/B))].push_back(a[i]);
    tc2[(int)(a[i] / (R/B))].push_back(a[i]);
}

//Convierte un vector en un arreglo simple
int **c1=conVtoA(tc1);
int **c2=conVtoA(tc2);

//se obtiene el tamaño de cada casillero
int *t=tamVec(tc1);

//implementación con medición de tiempo
st=omp_get_wtime();
for(int i=0;i<B;i++)
    shellsort(c2[i],t[i]);

a=conArr2to1(c2,t);
sp=omp_get_wtime();

tiempo[0]=sp-st;
```

Ilustración 35. Implementación serial de los dos algoritmos

Como se mencionó, la mejor forma de paralelizar el algoritmo es distribuyendo el ordenamiento de cada casillero entre los diferentes núcleos o unidades de procesamiento.

Como se puede observar en la implementación serial, el ordenamiento de cada bloque se hace uno y mediante un *for* para ir recorriendo casillero por casillero. OpenMP tiene una función específica que nos permite distribuir las tareas que se hacen en un *for*, dividiendo el dominio de nuestros datos, en este caso los bloques. Esto nos asegura que cada núcleo ejecutará el ordenamiento individual de uno o varios bloques.

En la ilustración 36 se muestra el código en versión paralela, con las directivas de OpenMP y con medición de tiempo.

```
//implementación paralela con medición de tiempo
st=omp_get_wtime();
for(int i=0;i<B;i++)
    shellsort(c2[i],t[i]);

a=conArr2to1(c2,t);
sp=omp_get_wtime();

tiempo[0]=sp-st;

st=omp_get_wtime();
omp_set_num_threads(H);
#pragma omp parallel
{
#pragma omp for
for(int i=0;i<B;i++)
    shellsort(c1[i],t[i]);
}
b=conArr2to1(c1,t);
sp=omp_get_wtime();

tiempo[1]=sp-st;
```

Ilustración 36. Implementación paralela con medición de tiempo

3.6 Análisis de resultados

El algoritmo fue ejecutado con diferentes configuraciones, variando tanto número de bloques, como número de hilos. De estas pruebas se obtuvieron los algoritmos ordenados y se compararon los resultados seriales contra los paralelos, garantizando que ambos produjeron resultados correctos e iguales. Además, para la comparación de eficiencia se tomó el tiempo que tardaba cada prueba, arrojando los resultados de las tablas 29, 30 y 31:

Tabla 29. Resultados 10 bloques

Ordenamiento				
Hilos	1	2	4	8
Bloques	10	10	10	10
Configuración	(1,10)	(2,10)	(4,10)	(8,10)
Tiempo (s)				
Intel Core i5	0.2782	0.1726	0.1369	0.1356
Intel Xeon	0.440201	0.224319	0.140535	0.14184
Intel Core i7	0.2107	0.1352	0.087	0.0824
Speedup				
Intel Core i5	1	1.61181924	2.03214025	2.05162242
Intel Xeon	1	1.96238838	3.13232291	3.10350395
Intel Core i7	1	1.55843195	2.42183908	2.55703883
Eficiencia				
Intel Core i5	1	0.80590962	0.50803506	0.2564528
Intel Xeon	1	0.98119419	0.78308073	0.38793799
Intel Core i7	1	0.77921598	0.60545977	0.31962985

Tabla 30. Resultados 100 bloques

Ordenamiento				
Hilos	1	2	4	8
Bloques	100	100	100	100
Configuración	(1,100)	(2,100)	(4,100)	(8,100)
Tiempo (s)				
Intel Core i5	0.1677	0.1063	0.0719	0.0702
Intel Xeon	0.258283	0.133076	0.072018	0.080044
Intel Core i7	0.131	0.0806	0.0458	0.0463
Speedup				
Intel Core i5	1	1.57761054	2.33240612	2.38888889
Intel Xeon	1	1.94086838	3.5863673	3.22676278
Intel Core i7	1	1.62531017	2.86026201	2.82937365
Eficiencia				
Intel Core i5	1	0.78880527	0.58310153	0.29861111
Intel Xeon	1	0.97043419	0.89659182	0.40334535
Intel Core i7	1	0.81265509	0.7150655	0.35367171

Tabla 31: Resultados 1000 bloques

Ordenamiento				
Hilos	1	2	4	8
Bloques	1000	1000	1000	1000
Configuración	(1,1000)	(2,1000)	(4,1000)	(8,1000)
Tiempo (s)				
Intel Core i5	0.1677	0.1063	0.0719	0.0702
Intel Xeon	0.258283	0.133076	0.072018	0.080044
Intel Core i7	0.131	0.0806	0.0458	0.0463
Speedup				
Intel Core i5	1	1.57761054	2.33240612	2.38888889
Intel Xeon	1	1.94086838	3.5863673	3.22676278
Intel Core i7	1	1.62531017	2.86026201	2.82937365
Eficiencia				
Intel Core i5	1	0.78880527	0.58310153	0.29861111
Intel Xeon	1	0.97043419	0.89659182	0.40334535
Intel Core i7	1	0.81265509	0.7150655	0.35367171

Gráficas de resultados:

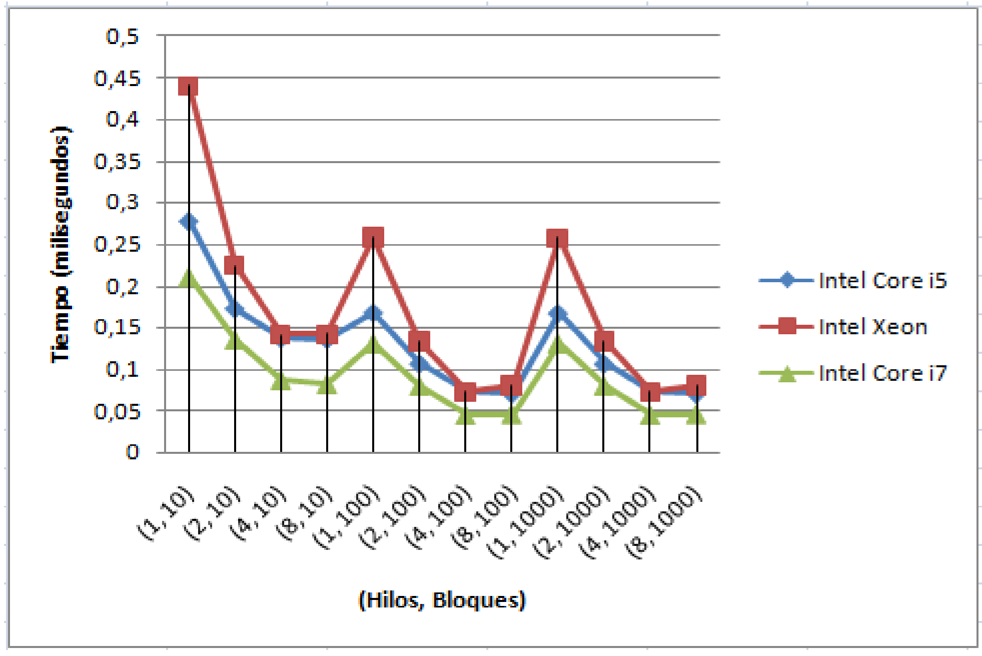


Ilustración 37. Gráfica de tiempo

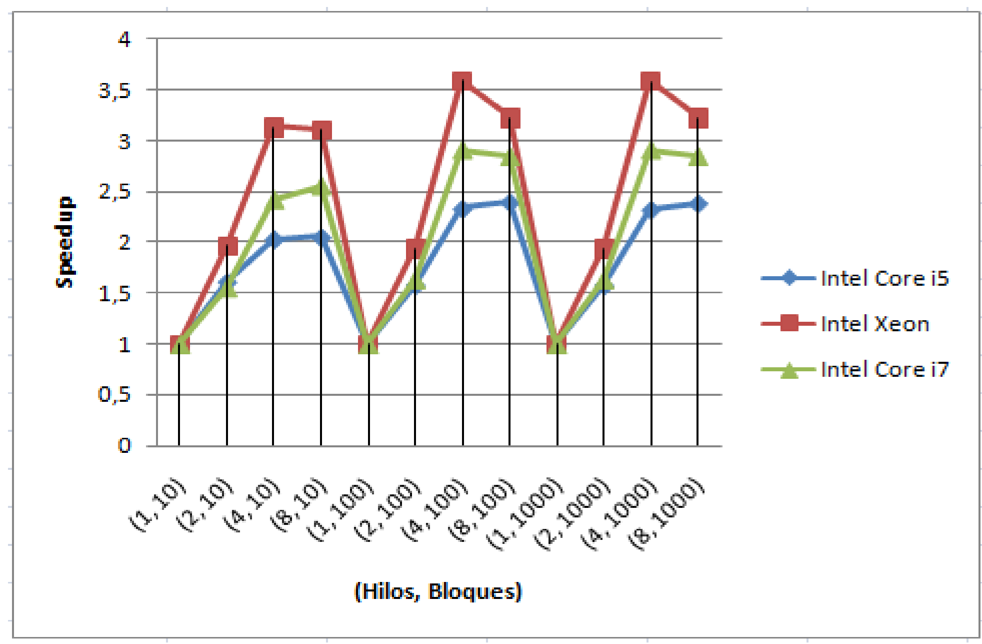


Ilustración 38. Gráfica speedup

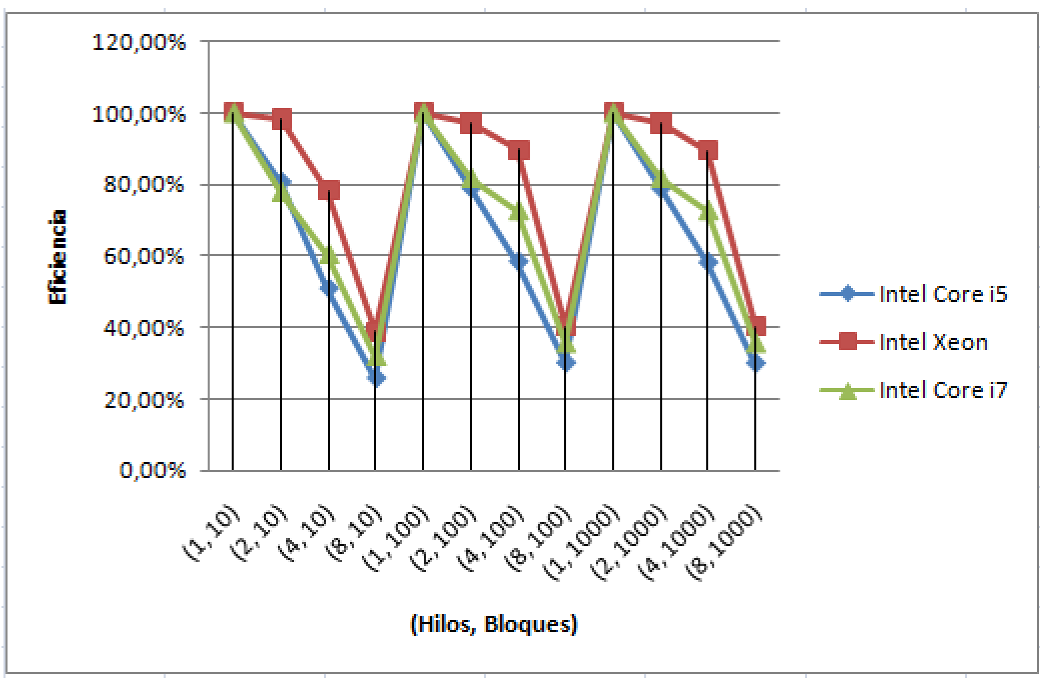


Ilustración 39. Gráfica eficiencia

Mediante las gráficas y tablas presentadas anteriormente se puede observar que el propio algoritmo permite una disminución de tiempos considerable al aumentar el número de bloques. A su vez se puede observar que el tiempo de ejecución mejoró notablemente con 2 hilos respecto a la ejecución serial (1 hilo); asimismo existe una disminución del tiempo de ejecución al ejecutar con 4 hilos.

Respecto a la ejecución con 8 hilos se pueden observar datos discrepantes, en los cuales hay un aumento o disminución del tiempo, aclarando que estas diferencias de tiempo son muy pequeñas. Esto nos indica que el número de hilos óptimo para esta configuración de hardware y software está entre 4 y 8.

Por las especificaciones de los procesadores, los cuales tienen 4 núcleos, el número de hilos ideal sería 4, esto se comprueba mediante las gráficas, observando una disminución considerable respecto a la ejecución serial.

4. Algoritmos de búsqueda de cadenas

Se puede definir como cadena a los datos que se descomponen en pequeñas partes identificables. Este tipo de datos se caracteriza fácilmente por el hecho de que se pueden escribir en forma de series lineales (generalmente muy largas) de caracteres.

Cuando queremos manipular textos se vuelve muy importante cómo lo vamos a hacer porque éstos tienden a ser muy grandes. Por ejemplo, supongamos que un libro tiene 400 páginas, 40 líneas por página y a su vez, 70 caracteres por línea (incluyendo letras, dígitos, caracteres especiales). Esto da un total de más de un millón de caracteres en un libro.

En esta sección se proponen y describen dos algoritmos de búsqueda de cadenas para su estudio, devolviendo todas las ocurrencias de los patrones a buscar dentro de un texto.

4.1 Algoritmo Knuth-Morris-Pratt

Descripción

Este algoritmo busca subcadenas dentro de una cadena. Éste se apoya en una tabla llamada "de fallos", en la cual se guarda información de la cadena a buscar con el objetivo de que cada carácter en el texto sea comparado sólo una vez.[48]

El algoritmo originalmente fue elaborado por Donald Knuth y Vaughan Pratt y de modo independiente por James H. Morris en 1977, pero lo publicaron juntos los tres.

La mejora que muestra este algoritmo respecto al de fuerza bruta (comparar cada elemento del patrón a buscar con cada elemento del texto), es que aprovecha información que tiene la cadena que será buscada dentro de sí misma. Cuando ocurre un fallo en la comparación entre ésta y el texto, se da una cierta cantidad de saltos para que ningún elemento de él sea analizado más de una vez.

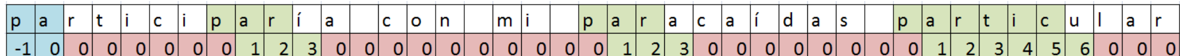
[48] Bustos B. (2008). *Algoritmo Knuth-Morris-Pratt*. Agosto 2014, de Universidad de Chile. Sitio web: http://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/BusqTexto/

El método clave para lograrlo, consiste en haber comprobado algún trozo de la cadena donde se busca con algún trozo de la cadena que se busca, lo que nos proporciona en qué sitios potenciales puede existir una nueva coincidencia, sobre el sector analizado que indica fallo.

Dicho de otro modo, partiendo del texto a buscar, elaboramos una lista con todas las posiciones, de salto atrás que señalen cuánto se retrocede desde la posición actual del texto a buscar.

Ejemplo de tabla de fallo:

Supongamos que queremos buscar la cadena: *"participaría con mi paracaídas particular"*, entonces la tabla quedaría como en la ilustración 40.



p	a	r	t	i	c	i	p	a	r	í	a		c	o	n		m	i	
-1	0	0	0	0	0	0	0	1	2	3	0	0	0	0	0	0	0	0	0

p	a	r	a	c	a	í	d	a	s		p	a	r	t	i	c	u	l	a	r
0	1	2	3	0	0	0	0	0	0	0	0	1	2	3	4	5	6	0	0	0

Ilustración 40. Ejemplo de tabla de fallo

Las primeras posiciones se marcan con -1 y 0, que sirven como indicadores para no poder dar un salto mayor al de la longitud de la cadena a buscar cuando ocurra un fallo.

Se marca con ceros los caracteres hasta que se encuentre aquél con el que inicia la cadena.

Cuando se encuentra el carácter inicial, las posiciones subsecuentes (mientras coincidan con los que siguen al inicial) son marcadas con la distancia que existe con éste hasta que haya un "fallo".

Ejemplo de ejecución del algoritmo, desde la tabla 32 hasta la 44:

Tabla 32. Paso 1: Como no hay coincidencia, "Cadena" se mueve a la posición: índice + (Posición inicial) - Tabla[índice].

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Texto	b	a	c	b	a	b	a	b	a	b	a	c	a	c	b
Cadena	a	b	a	b	a	c	a								
Índice	0	1	2	3	4	5	6								
Tabla	-1	0	0	1	2	3	0								

Tabla 33. Paso 2: Como hay coincidencia, "Cadena" no se mueve a la derecha.

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Texto	b	a	C	b	a	b	a	b	a	b	a	c	a	c	b
Cadena		a	b	a	b	a	c	a							
Índice		0	1	2	3	4	5	6							
Tabla		-1	0	0	1	2	3	0							

Tabla 34. Paso 3: Como no hay coincidencia, "Cadena" se mueve a la posición: índice + (Posición inicial) - Tabla[índice].

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Texto	b	a	c	b	a	b	a	b	a	b	a	c	a	c	b
Cadena		a	b	a	b	a	c	a							
Índice		0	1	2	3	4	5	6							
Tabla		-1	0	0	1	2	3	0							

Tabla 35. Paso 4: Como no hay coincidencia, "Cadena" se mueve a la posición: índice + (Posición inicial) - Tabla[índice].

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Texto	b	a	c	b	a	b	a	b	a	b	a	c	a	c	b
Cadena				a	b	a	b	a	c	a					
Índice				0	1	2	3	4	5	6					
Tabla				-1	0	0	1	2	3	0					

Tabla 36. Paso 5: Como hay coincidencia, "Cadena" no se mueve a la derecha.

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Texto	b	a	c	b	a	b	a	b	a	b	a	c	a	c	b
Cadena					a	b	a	b	a	c	a				
Índice					0	1	2	3	4	5	6				
Tabla					-1	0	0	1	2	3	0				

Tabla 37. Paso 6: Como hay coincidencia, "Cadena" no se mueve a la derecha.

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Texto	b	a	c	b	a	b	a	b	a	b	a	c	a	c	b
Cadena					a	b	a	b	a	c	a				
Índice					0	1	2	3	4	5	6				
Tabla					-1	0	0	1	2	3	0				

Tabla 38. Paso 7: Como hay coincidencia, "Cadena" no se mueve a la derecha.

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Texto	b	a	c	b	a	b	a	b	a	b	a	c	a	c	b
Cadena					a	b	a	b	a	c	a				
Índice					0	1	2	3	4	5	6				
Tabla					-1	0	0	1	2	3	0				

Tabla 39. Paso 8: Como hay coincidencia, "Cadena" no se mueve a la derecha.

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Texto	b	a	c	b	a	b	a	b	a	b	a	c	a	c	b
Cadena					a	b	a	b	a	c	a				
Índice					0	1	2	3	4	5	6				
Tabla					-1	0	0	1	2	3	0				

Tabla 40: Paso 9: Como hay coincidencia, "Cadena" no se mueve a la derecha.

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Texto	b	a	c	b	a	b	a	b	a	b	a	c	a	c	b
Cadena					a	b	a	b	a	c	a				
Índice					0	1	2	3	4	5	6				
Tabla					-1	0	0	1	2	3	0				

Tabla 41. Paso 10: Como no hay coincidencia, "Cadena" se mueve a la posición: índice + (Posición inicial) - Tabla[índice].

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Texto	b	a	c	b	a	b	a	b	a	b	a	c	a	c	b
Cadena					a	b	a	b	a	c	a				
Índice					0	1	2	3	4	5	6				
Tabla					-1	0	0	1	2	3	0				

Tabla 42. Paso 11: Como hay coincidencia, "Cadena" no se mueve a la derecha.

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Texto	b	a	c	b	a	b	a	b	a	b	a	c	a	c	b
Cadena							a	b	a	b	a	c	a		
Índice							0	1	2	3	4	5	6		
Tabla							-1	0	0	1	2	3	0		

Tabla 43. Paso 12: Como hay coincidencia, "Cadena" no se mueve a la derecha.

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Texto	b	a	c	b	a	b	a	b	a	b	a	c	a	c	b
Cadena							a	b	a	b	a	c	a		
Índice							0	1	2	3	4	5	6		
Tabla							-1	0	0	1	2	3	0		

Tabla 44. Paso 13: Como se alcanzó el final de la cadena a buscar, se regresa la posición inicial del texto.

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Texto	b	a	c	b	a	b	a	b	a	b	a	c	a	c	b
Cadena							a	b	a	b	a	c	a		
Índice							0	1	2	3	4	5	6		
Tabla							-1	0	0	1	2	3	0		

Análisis e implementación serial

La primera parte del algoritmo consiste en calcular la tabla de fallos, codificada en la ilustración 41. La función recibe la cadena que se va a buscar en el texto y un arreglo F de enteros, donde se marcan los caracteres coincidentes al inicio del patrón. Se tiene un ciclo iterativo *do...while()* que se ejecuta *m* veces, donde *m* es la cantidad de caracteres del patrón.

```
void precalcular_tablaKMP(unsigned char *P, int *F){
    int pos = 2; //se comienza a analizar en la posición 2
    int cnd = 0; //índice en P del sig. carácter del actual

    F[0] = -1;
    F[1] = 0; //Los primeros caracteres no se analizan

    do{
        if( P[pos -1] == P[cnd]){ //Siguiente candidato coincidente en la cadena
            cnd++;
            F[pos] = cnd;
            pos++;
        }
        else if( cnd > 0){ //Cuando fallan las coincidencias
            cnd = F[cnd];
        }
        else{ //no se hallaron candidatos coincidentes
            F[pos] = 0;
            pos++;
        }
    }while( pos <= strlen(P) );
}
```

Ilustración 41. Función para calcular la tabla de fallos

$$Entonces, la\ función: O(m)$$

Una vez que se tiene la tabla, se procede a buscar las ocurrencias. La función se encuentra en la ilustración 42. Ésta recibe el texto donde se va a realizar la búsqueda, la(s) cadena(s) a buscar, la tabla de fallos y un arreglo donde se guardan las ocurrencias.

```
void busquedaKMP(unsigned char *T, unsigned char *P, int *F, int *ocurrencia){
    int k = 0, i = 0; //Variables para examinar T y P respectivamente
    int j = 0;
    if( strlen(T) >= strlen(P) ){
        precalcular_tablaKMP(P, F);
        while( (k+i) < strlen(T) ){
            if( P[i] == T[k+i] ){
                if( i == (strlen(P)-1) ){
                    ocurrencia[j] = k;  // Ocurrencia
                    j++;
                }
                i++;
            }
            else{
                k = k + i - F[i];
                if( i > 0)
                    i = F[i];
            }
        }
    }
    ocurrencia[j] = -1;  //Final del texto
}
```

Ilustración 42. Función que realiza la búsqueda utilizando la tabla de fallos

Se puede ver un ciclo *while()* en la función, que itera *n = k + i* veces, que representan la posición en el texto y la del patrón. Se puede observar que cuando ocurre un fallo, se dan *F[i] saltos* (los que indica la tabla de fallos).

$$Entonces, la\ función: O(n)$$

Por lo tanto, el algoritmo Knuth-Morris-Pratt tiene una complejidad de:

$$O(n+m)$$

4.2 Algoritmo Rabin-Karp

Descripción

En este algoritmo, el problema se plantea de tal forma que sólo se busque una clave. Lo que se hace es evaluar una función *hash* para cada cadena de *m* caracteres del texto y compararla contra el valor de la(s) cadena(s) que se está buscando.[49]

La función *hash* que utilizaremos para el algoritmo, mapea una cadena de entrada a un valor entero positivo. A continuación la descripción: supongamos que buscamos un número de longitud *m* y cuya base es *d* dentro de un texto t[i , ..., M-1].

$$t_0 = t[i] * d^{m-1} + t[i+1] * d^{m-2} + \cdots + t[i+M-1]$$

[49] Sedgewick R. (1995). *Algoritmos en C++.* Princeton: Ediciones Díaz de Santos.

Con evaluar el polinomio obtenemos el valor *hash* de la primera subcadena, y para continuar con el de la siguiente simplemente:

$$t_1 = t_0 - t[i] * d^{m-1} + t[i+m]$$

Es decir, no se vuelve a evaluar el polinomio, sino que se quita el valor *hash* del carácter de la izquierda y se suma el de la derecha (en este caso porque hacemos barrido del texto de izquierda a derecha), lo que consume un tiempo constante.

En nuestro caso de estudio, tenemos que la cardinalidad de nuestro conjunto (dado que éste debe normalizarse) es de: 27 letras, 5 letras acentuadas, 8 signos de puntuación (, . - ! ? etc.) y 10 dígitos::

$$d = 50$$

La ilustración 43 muestra la función que obtiene el *hash* de cada subcadena y se ejemplifica la conversión de algunas cadenas a un número positivo:

```
int valor(char *t, int m, int d){
    int i, v = 0;
    for( i = 0; i < m; i++ )
        v = ( d*v + t[i] );
    return v;
}
```

Ilustración 43. Función que calcula el valor *hash* de una cadena

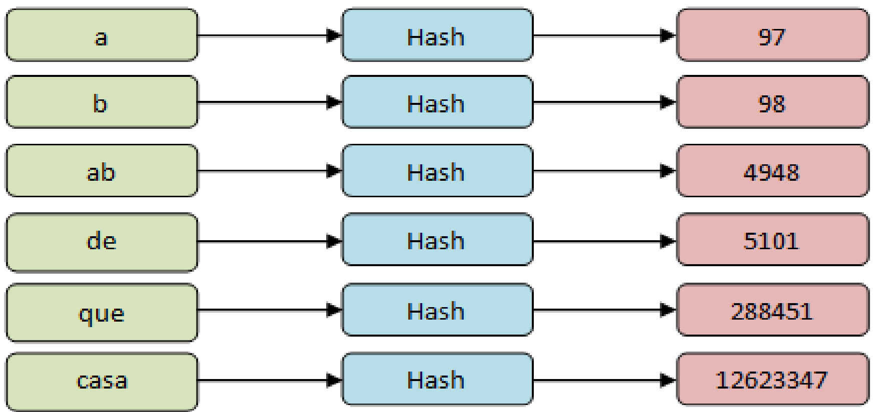


Ilustración 44. Ejemplo de la función

En la ilustración 44 se puede observar que el valor *hash* aumenta rápidamente al aumentar el tamaño de la(s) cadena(s) a buscar. Es por esto que utilizaremos un número primo para obtener el módulo del valor de la siguiente manera:

$$t_0 = (t[i] * d^{m-1} + t[i+1] * d^{m-2} + \cdots + t[i+M-1]) \,\%\, q$$

donde *q* es un número primo tal que:

$$q * d < 2^{32}$$

esto es para evitar el desbordamiento con números mayores a la capacidad de un *integer* (4 bytes = 4,294,967,296).

$$Entonces, \ q = 33{,}554{,}393, para\ cumplir\ con\ q * d < 2^{32}$$

Es importante considerar que, aunque la probabilidad de que ocurra es muy baja, es necesario que cuando se halle una igualdad, se deben comparar las cadenas, dado que puede ser que no sean las mismas aunque su valor coincida.

Análisis e implementación serial

La función valor se utiliza para calcular el *hash* de las cadenas. Recibe una subcadena t, el tamaño de ésta y la cardinalidad del conjunto de los caracteres del texto de entrada (letras, dígitos, signos de puntuación). La función calcula el valor de la subcadena sumando m factores.

```
int valor(char *t, int m, int d){
    int i, v = 0;
    for( i = 0; i < m; i++ )
        v = ( d*v + t[i] );
    return v;
}
```

Ilustración 45. Función que calcula el valor *hash*deuna cadena

$$Entonces, la\ función: O(m)$$

La función para realizar la búsqueda recibe al texto y al patrón. Primero se calcula el *hash* del patrón y de los primeros *m* caracteres y se obtiene el módulo, que consume tiempo constante.

Después, existe un *for* que itera *n* - *m* veces (la longitud del texto menos la del patrón), pero dentro de ella se llama a la función valor, que se ejecuta la misma cantidad de veces. Es decir, este algoritmo tiene la complejidad de la función.

```
int busqueda_RK( char *T, char *P ){
    const int d = 50; //cardinalidad del conjunto
    const int q = 33554393; //número primo grande que cabe en 32 bits
    int n = strlen(T);
    int m = strlen(P);
    int p, t, i;

    p = valor(P, m, d) % q; //Valor hash del patrón
    t = valor(T, m, d) % q; //Valor hash de los primeros m caracteres del texto

    for( i = 0; i <= (n-m); i++ ){
        if( p == t )
            printf("%d\n", i);
        if( i < (n-m) )
            t = valor(&T[i], m, d) % q;
    }
}
```

Ilustración 46. Función que realiza la búsqueda calculando los valores de las subcadenas en el texto

Por lo tanto, el algoritmo Rabin-Karp tiene una complejidad de:

$$O((n - m) * m)$$

4.3 Selección del algoritmo a paralelar y justificación

De entre los dos algoritmos que ya estudiamos, creemos que es conveniente utilizar Rabin-Karp para paralelarlo.

Justificación:

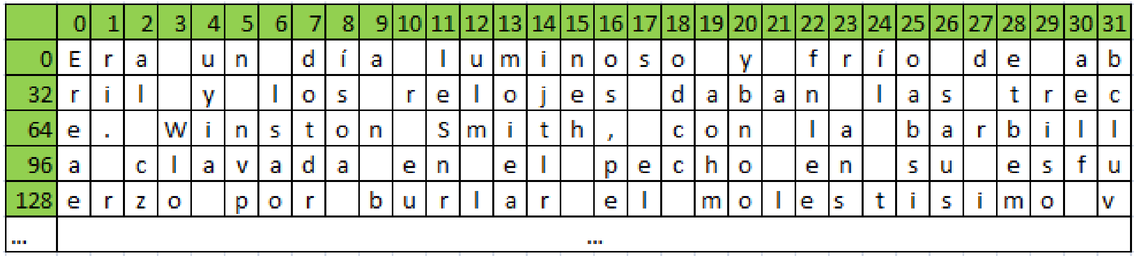
A pesar de que el algoritmo Knuth-Morris-Pratt tiene complejidad menor, se puede observar que si se selecciona aleatoriamente cualquier parte del texto que no sea el inicio, no es posible determinar si hay coincidencia o no, porque depende de caracteres previos. Es decir, en Knuth-Morris-Pratt no se puede tener descomposición de dominio ni funcional, pues la función de búsqueda depende de la que calcula la tabla de fallos.

En contraste, en el algoritmo Rabin-Karp se puede hacer descomposición de dominio, dividiendo el texto completo entre las unidades de procesamiento y calcular de forma independiente los valores *hash* para compararlos con el del patrón.

4.4 Análisis e implementación paralela en CUDA

Como ya se mencionó en la parte de justificación, se puede aplicar descomposición de dominio en el algoritmo Rabin-Karp. Ésta se describe a continuación:

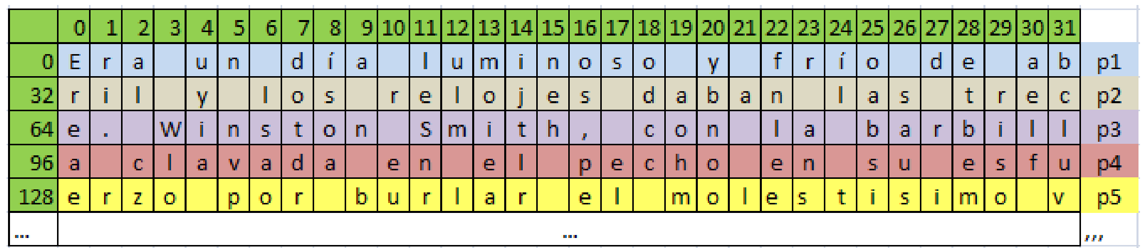
Supongamos que tenemos el siguiente texto, el cual puede ser visto como un arreglo de caracteres, tal como en la ilustración 47:



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	E	r	a		u	n		d	í	a		l	u	m	i	n	o	s	o		y		f	r	í	o		d	e		a	b
32	r	i	l		y		l	o	s		r	e	l	o	j	e	s		d	a	b	a	n		l	a	s		t	r	e	c
64	e	.		W	i	n	s	t	o	n		S	m	i	t	h	,		c	o	n		l	a		b	a	r	b	i	l	l
96	a		c	l	a	v	a	d	a		e	n		e	l		p	e	c	h	o		e	n		s	u		e	s	f	u
128	e	r	z	o		p	o	r		b	u	r	l	a	r		e	l		m	o	l	e	s	t	i	s	i	m	o		v
...	...																															

Ilustración 47. Representación de un texto en un arreglo de caracteres

Se puede hacer una descomposición de dominio en la cual, una unidad de procesamiento trabaje sobre 32 caracteres consecutivos para obtener su valor *has,* como en la ilustración 48:



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
0	E	r	a		u	n		d	í	a		l	u	m	i	n	o	s	o		y		f	r	í	o		d	e		a	b	p1
32	r	i	l		y		l	o	s		r	e	l	o	j	e	s		d	a	b	a	n		l	a	s		t	r	e	c	p2
64	e	.		W	i	n	s	t	o	n		S	m	i	t	h	,		c	o	n		l	a		b	a	r	b	i	l	l	p3
96	a		c	l	a	v	a	d	a		e	n		e	l		p	e	c	h	o		e	n		s	u		e	s	f	u	p4
128	e	r	z	o		p	o	r		b	u	r	l	a	r		e	l		m	o	l	e	s	t	i	s	i	m	o		v	p5
...	...																																,,,

Ilustración 48. Texto dividido en diferentes unidades de procesamiento

De esta forma, una vez aplicada una función para normalizar el texto (es decir, convertir todo a minúsculas) se pueden realizar búsquedas; por ejemplo, si se busca la cadena "*día*", entonces la unidad *p1* hallaría una coincidencia. Si se busca la cadena "*y*", las unidades *p1* y *p2* devolverían las ubicaciones de éstas.

El problema está si se buscan cadenas que son "cortadas" al dividir el dominio y que sus caracteres sean analizados por diferentes unidades de procesamiento. Por ejemplo, si se busca la cadena *"abril"*, ésta no sería hallada a pesar de que sí existe en el texto.

Una solución es que cada unidad de procesamiento trabaje con una cantidad de caracteres *n* más la longitud de la cadena patrón *m*. En este caso, los procesadores trabajarían con los elementos mostrados en la tabla 45:

Tabla 45. Opción para hacer descomposición de dominio

Procesador	Operación	Resultado
p1	(0, 31 + 5)	(0, 36)
p2	(32, 63 + 5)	(32, 68)
p3	(64, 95 + 5)	(64, 100)
p4	(96, 127 + 5)	(96, 132)
p5	(128, 159 + 5)	(128, 164)

Con esta alternativa, no importa si una cadena es "cortada" y *"abril"* sería hallada por *p1*.

Para poder implementar el código en CUDA, hay que hacer algunas modificaciones al serial. Lo primero es implementar una función que aloje y copie las variables ya inicializadas del CPU al GPU. Después, que llame a la función que realizará la búsqueda, copie los resultados del GPU al CPU y libere los recursos utilizados, como se muestra en el código de la ilustración 49.

```
void dispositivo( char *T, char *P, int *ocurrencia, int *n, int *m, int *p,
                  int *i, int BLOQUES, int HILOS){
    //Apuntadores en el dispositivo
    char *d_T, *d_P;
    int *d_n, *d_m, *d_ocurrencia, *d_p, *d_i;

    // Asignamos memoria en el GPU
    cudaMalloc( (void**)&d_T, *n * sizeof(char) );
    cudaMalloc( (void**)&d_P, *m * sizeof(char) );
    cudaMalloc( (void**)&d_n, sizeof(int) );
    cudaMalloc( (void**)&d_m, sizeof(int) );
    cudaMalloc( (void**)&d_p, sizeof(int) );
    cudaMalloc( (void**)&d_ocurrencia, tam_ocurr * sizeof(int) );
    cudaMalloc( (void**)&d_i, sizeof(int) );

    // Copia memoria del CPU al GPU
    cudaMemcpy( d_T, T, *n * sizeof(char), cudaMemcpyHostToDevice );
    cudaMemcpy( d_P, P, *m * sizeof(char), cudaMemcpyHostToDevice );
    cudaMemcpy( d_n, n, sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy( d_m, m, sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy( d_p, p, sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy( d_ocurrencia, ocurrencia, tam_ocurr*sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy( d_i, i, sizeof(int), cudaMemcpyHostToDevice );

    //Llamada al kernel
    kernel_RK<<< BLOQUES, HILOS >>>( d_T, d_P, d_n, d_m, d_ocurrencia, d_p, d_i );

    //Copia del GPU al CPU
    cudaMemcpy( ocurrencia, d_ocurrencia, tam_ocurr*sizeof(int), cudaMemcpyDeviceToHost );

    // Liberamos memoria del GPU
    cudaFree( d_T );
    cudaFree( d_P );
    cudaFree( d_n );
    cudaFree( d_p );
    cudaFree( d_m );
    cudaFree( d_i );
    cudaFree( d_ocurrencia );
}
```

Ilustración 49. Función que asigna, copia, ejecuta y libera memoria en el GPU

El *kernel* está implementado en la función de la ilustración 50:

```
__global__ void kernel_RK( char *d_T, char *d_P, int *d_n, int *d_m,
                          int *d_ocurrencia, int *d_p, int *d_i){
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int inf = *d_n * tid / (gridDim.x * blockDim.x) - 1;
    int sup = *d_n * (tid + 1) / (gridDim.x * blockDim.x) + 1;
    int t = valor( &d_T[inf], *d_m ) % q;

    while( inf < (sup + *d_m)){
        if( t == *d_p && *d_i < tam_ocurr){
            atomicExch( &d_ocurrencia[ *d_i ], inf );
            atomicAdd( d_i, 1 );
        }
        inf++;
        t = valor( &d_T[inf], *d_m ) % q;
    }
}
```

Ilustración 50. Función kernel

La variable *tid* calcula el índice de cada hilo; de esta forma, cada hilo es identificado de forma única. Se utiliza este valor para calcular el límite inferior en la variable *inf* y el superior en *sup*, tal como se muestra en la tabla 14.

El ciclo *while* permite que cada hilo itere sobre los elementos que le corresponden. En caso de que los valores de la cadena patrón y la subcadena del texto coincidan, se deben utilizar operaciones atómicas; es decir, solo un hilo en cada bloque puede acceder a ellas a la vez para evitar la condición de carrera y que dos hilos almacenen ubicaciones diferentes de coincidencias en la misma localidad de memoria.

Como se puede observar en la ilustración 51, la función *valor()* es llamada en el kernel, por lo que ésta debe tener modificadores para poder ser utilizada tanto por el CPU como por el GPU:

```
__host__ __device__ int valor(char *t, int m ){
    int i, v = 0;
    for( i = 0; i < m; i++ )
        v = ( d*v + t[i] );
    return v;
}
```

Ilustración 51. Función para calcular el valor hash con modificadores __host__ y __global__

4.5 Análisis de resultados

El código en CUDA puede ser ejecutado con diferentes configuraciones, variando la cantidad de bloques e hilos por bloque; la multiplicación de estos da como resultado el total de hilos en ejecución. Para la toma de medidas, a continuación, se han elegido algunas configuraciones representativas en diferentes GPUs:

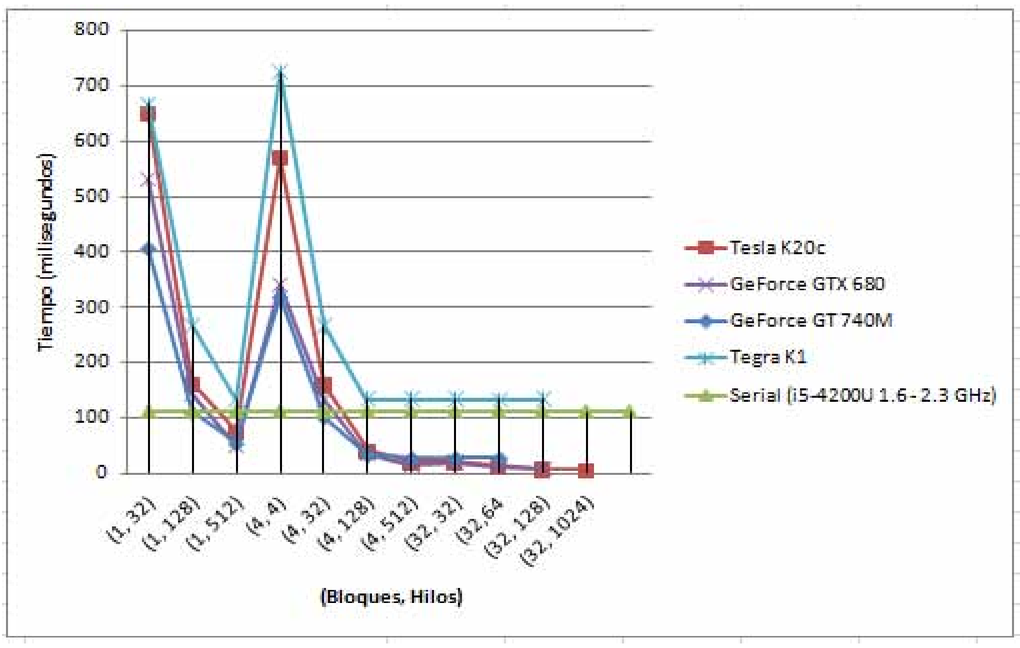


Ilustración 52. Tiempo de ejecución del *kernel* en milisegundos con diferentes configuraciones

En estas gráficas, el dominio es la configuración de ejecución, que está compuesto por *(bloques, hilos)*. Se puede calcular el total de hilos en ejecución multiplicándolos. Vale la pena comentar que para obtener mayor detalle en las gráficas, se han ignorado algunas configuraciones que tomaban demasiado tiempo, como (1, 1) y (1, 4), pero permanecen las más representativas. En la ilustración 52 se puede observar que el tiempo de ejecución serial es inferior al paralelo en todas las GPUs cuando en total hay pocos hilos, pero esto cambia cuando se aumenta el número de éstos, principalmente en la Tesla K20c y en GeForce GTX 680.

Esto se debe a que una GPU tiene menor velocidad de reloj comparada con un CPU, pero se puede observar la reducción significativa de tiempos cuando todas las unidades de procesamiento comienzan a trabajar. Esto es, 192 núcleos en Tegra K1, 1536 para GTX 680 y 2688 para Tesla K20c.

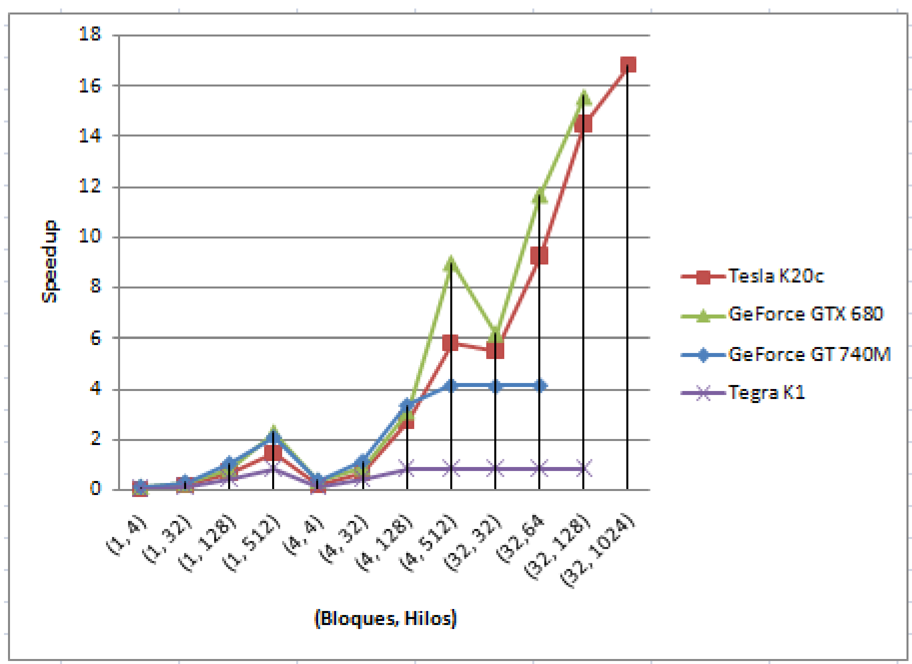


Ilustración 53. Speedup de las diferentes GPUs

Esta segunda gráfica (ilustración 53) muestra el *speedup* comparado con la versión serial (tiempo de ejecución del procesador Intel i5). De la gráfica se pueden concluir los siguientes aspectos:

- Tanto Tegra K1 como GT 740M alcanzan un límite en el *speedup*; es decir, todos sus núcleos están trabajando y el programa ya no puede ser acelerado.
- Tesla K20c es la GPU que más puede acelerar la aplicación, llegando casi a 17x.
- La GPU GTX 680 es la tarjeta que tiene mayor aceleración comparada con las demás, debido a que tiene la mayor velocidad de reloj. Como se puede observar, esta ventaja se pierde cuando el dispositivo Tesla hace uso de todos sus núcleos.
- Al ser una GPU diseñada para dispositivos móviles (es decir, que consuma muy poca energía), la GPU Tegra K1 no alcanza a llegar a 1 en speedup; esto significa que a pesar de hacer uso de todo su poder, éste no puede ser comparado con el procesador de una computadora convencional.
- La GPU Tesla K20c es la que soporta la configuración más grande, por eso el resto de las tarjetas no tienen las últimas medidas.

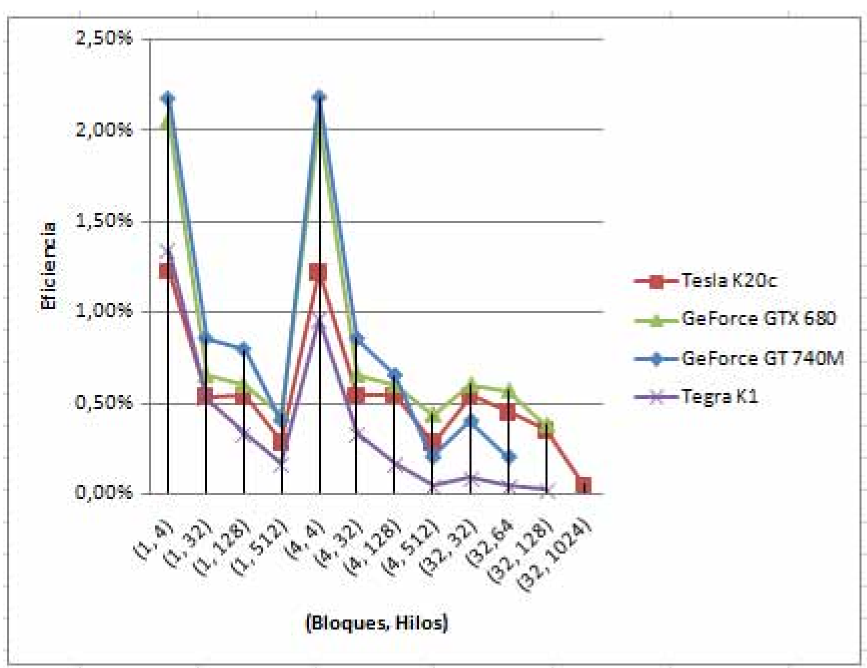


Ilustración 54. Eficiencia

Esta última gráfica, la ilustración 54, es muy interesante. En ella se muestra la eficiencia, que es una medida relacionada a los beneficios versus los recursos utilizados. Una medida cercana al 100% significa que todos los recursos fueron utilizados todo el tiempo de ejecución.

Al tomar como patrón de tiempo serial a un CPU, la eficiencia de una GPU es realmente baja. La razón es la siguiente: Si se utilizan 2 unidades de procesamiento, uno espera que una aplicación sea ejecutada en la mitad de tiempo comparada con la versión serial, si se utilizan cuatro unidades, la cuarta parte, etc. En este caso, para acelerar la aplicación 17x (Ilustración 14), se utilizó una configuración (32, 1024), es decir 32,768 hilos en ejecución. Principalmente esto se debe a que la velocidad de reloj de un GPU es menor a la de un CPU.

A pesar de que la eficiencia es realmente baja, consideramos que una aplicación debe de ser acelerada con una tarjeta gráfica si es que se tiene el recurso.

5. Algoritmos celulares o evolutivos

Este tipo de algoritmos busca la mejor solución a los problemas de búsqueda y optimización cuya solución está en un rango determinado y conocido.

Se basan en la teoría de la evolución de Darwin, la cual menciona que sólo los individuos más fuertes sobreviven y pueden dejar descendencia; por lo tanto cada generación de individuos es más fuerte y más adaptada a su entorno.

Haciendo una analogía con la teoría de la evolución, un individuo es una solución a nuestro problema la cual, dependiendo de su eficiencia se descarta o tiene la posibilidad de dejar herencia, garantizando que cada generación tenga una solución mejor.

5.1 Descripción

Es indispensable señalar que los algoritmos genéticos no nos garantizan la mejor solución general a nuestro problema, pero sí la más próxima dependiendo de la generación en la que nos encontremos.[50]

Ejemplo del algoritmo: Se desea encontrar el máximo de una función $f(x) = -2x^3 - 3x^2 + 12x + 10$ dentro del rango $[0,31]$.

1. Codificación: Se hace una analogía entre las posibles soluciones (los números entre 0 y 31) a un forma que sea fácilmente manipulable por el algoritmo genético, generalmente se usa un código binario.
   En este caso, cada individuo se conformará de 5 bits, los cuales representan los números entre 0 y 31.
   Ejemplo: 10101=21
2. Población inicial: se crea una serie de individuos, en este caso 6 (por practicidad para la explicación del algoritmo), a los cuales se les asignarán valores aleatoriamente, como en la tabla 46:

[50] Dorronsoro B. (2002). *Algoritmos Evolutivos Celulares*. Málaga: ETSI Informática.

Tabla 46. Población inicial

N°:	1	2	3	4	5	6
Individuo:	11001	01001	01101	00101	01110	01110

3. Selección: En la tabla 47, se realiza una serie de competencias entre dos individuos para determinar quién es el más fuerte y por lo tanto sobrevive.

Tabla 47. Selección

Competencia	Individuo	Valor codificado	Evaluación f(x)	Ganador
1	11001	25	-32815	
	01001	9	-1583	*
2	01101	13	-4735	
	00101	5	-255	*
3	01110	14	-5898	*
	01110	14	-5898	

4. Reproducción: Se generan 6 nuevos individuos a partir de las características de estos 3, como en la tabla 48. Se usan números aleatorios para determinar qué porcentaje de las características hereda de un individuo y de otro.

Tabla 48. Reproducción

Individuo 1	Individuo 2	%	Nuevo individuo
01001	00101	2	01101
01001	01110	1	01110
00101	01001	2	00001
00101	01110	4	00100
01110	01001	2	01001
01110	00101	4	01111

Tabla 49. Nueva población

01101	01110	00001	00100	01001	01111

5. La Nueva población de la tabla 49 vuelve a tener 6 individuos, los cuales están mejor adaptados (se acercan más a la solución de nuestro problema). Para obtener un mejor resultado es necesario repetir los pasos 2, 3 y 4 con la nueva población; de esta manera garantizamos una serie de individuos más cerca de nuestra solución. Después de 4 iteraciones a los pasos [2 - 4] se obtiene la población de la tabla 50:

Tabla 50. Población después de cuatro iteraciones

Individuo	Valor codificado	Evaluación f(x)
00000	0	10
00001	1	17
00001	1	17
00001	1	17
00001	1	17
00000	0	10

En la tabla 51 podemos observar que los individuos tienden a una configuración específica. Por lo tanto podemos concluir que nuestra mejor solución al problema es::

Tabla 51. Población solución

Individuo	Valor codificado	Evaluación f(x)
00001	1	17

5.2 Análisis e implementación serial

La complejidad de un algoritmo celular es muy variada y depende totalmente de los métodos que se utilizan en cada etapa. Éstas están en las ilustraciones 55 y 56.

```
//Poblacion inicial
for( i = 0; i < POBLACION; i++)
    for( j = 0; j < ATTRIBS; j++)
        b[i][j] = rand() % 2;
```

Ilustración 55. Población inicial

```
//Se crean nuevas generaciones
for( i = 0; i < GENERACIONES; i++){
    j = 0;
    imprimir( "Poblacion", b, i, 1);
    while( j < POBLACION){
        //Se seleccionan los ganadores
        if( fun( binToDec( b[2*j] ) ) > fun( binToDec( b[2*j+1] ) ) )
            for( k = 0; k < ATTRIBS; k++)
                ganadores[j][k] = b[2*j][k];
        else
            for( k = 0; k < ATTRIBS; k++)
                ganadores[j][k] = b[2*j+1][k];
        j++;
    }
    imprimir("Ganadores", ganadores, i, 2);

    //Combinaciones de ganadores para una nueva generacion
    for( n = 0; n < POBLACION/2; n++ ){
        //Seleccionamos un ganador diferente al actual
        do{
            k = rand() % (POBLACION/2);
        }while( k == n );
        //Combinacion entre ganadores
        gen = rand() % (ATTRIBS-2) + 1;
        for( l = 0; l < gen; l++ )
            temp[2*n][l] = ganadores[n][l];
        for( ; l < ATTRIBS; l++ )
            temp[2*n][l] = ganadores[k][l];

        gen = rand() % (ATTRIBS-2) + 1;
        for( l = 0; l < gen; l++ )
            temp[2*n+1][l] = ganadores[k][l];
        for( ; l < ATTRIBS; l++ )
            temp[2*n+1][l] = ganadores[n][l];
    }

    //Reemplazar la poblacion anterior con la nueva
    for( k = 0; k < POBLACION; k++)
        for( l = 0; l < ATTRIBS; l++)
            b[k][l] = temp[k][l];
}
```

Ilustración 56. Competencia y reproducción

Tabla 52 Complejidad de cada método

Etapa	Complejidad
Población inicial	P(n)
Competencia	C(m) (anidad en la cantidad de generaciones)
Reproducción	R(m) (anidad en la cantidad de generaciones)

$$T(n + n * (m + m))$$

En la tabla 52 se muestra la complejidad de cada método, donde *n* es el número de generaciones que se desean obtener y *m* es la cantidad de la población. La complejidad que se busca en cada etapa depende del número de generaciones que se quieran calcular.

$$Entonces, el\ algoritmo: O(n * m)$$

5.3 Selección del algoritmo a paralelar y justificación

En este capítulo hemos presentado un problema al cual daremos solución.

Justificación:

Sólo se presentó un algoritmo, debido a que todos los algoritmos de esta categoría son muy similares entre sí. Se tratará un problema que tenga mayor volumen de información y que consuma más tiempo de procesamiento para observar la diferencia.

5.4 Análisis e implementación paralela en OpenMP

Para paralelar el algoritmo, utilizaremos los siguientes datos de entrada:

Dato	Valor
Población	256
Atributos (binario)	16
Generaciones	20000

Así como una función que demande mayor tiempo de procesamiento. Seleccionamos una cuyo máximo relativo no se encuentra en un extremo y el dominio en el que buscaremos será de [0, 65535], que es representable en 16 bits. La función es:

$$f(x) = -x^4 + 50000x^3 + 20000x^2 + 18x + 5$$

Cuya gráfica es mostrada en la ilustración 57:

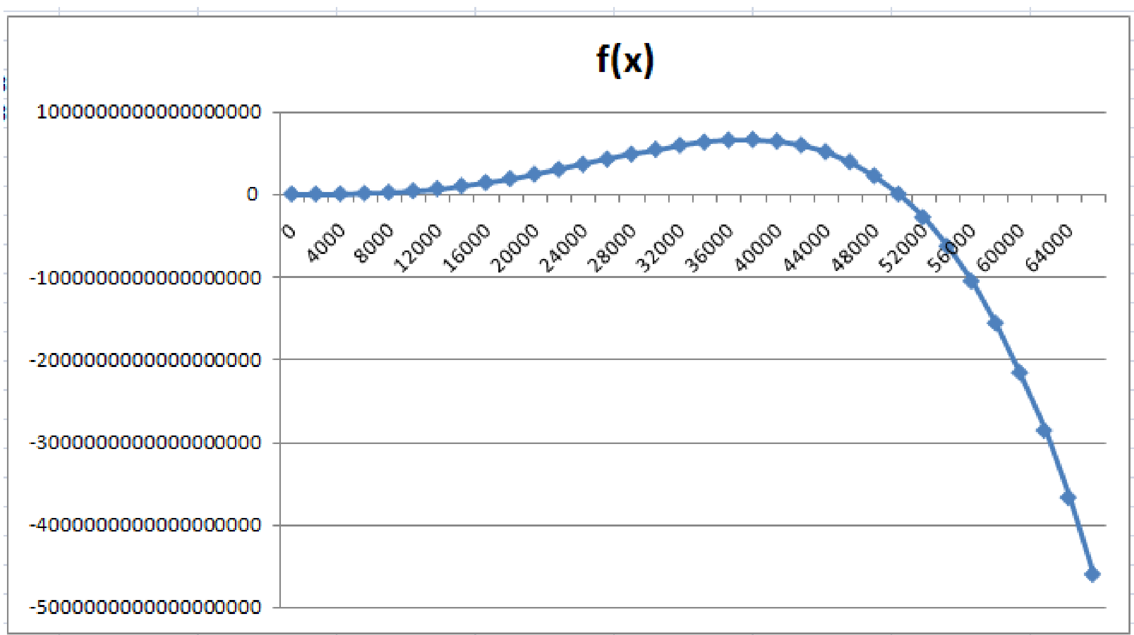


Ilustración 57. Gráfica de la función

Y el rango es representable en una variable tipo *long long* (64 bits). Se puede apreciar que el máximo relativo es cercano a 38000.

Las funciones principales son:

1. Crear la población inicial
2. Producir nuevas generaciones.
3. Selección de ganadores
4. Reproducción entre ganadores.

En el caso de la población inicial, consideramos que no es necesario aplicar modificaciones, porque el máximo de operaciones está dado por *Población * Atributos*, es decir, 4096, que son relativamente pocas.

No es posible paralelar de forma alguna la producción de nuevas generaciones. Esto se debe a que hay dependencia, tal como se representa en la ilustración 58. Para comenzar a procesar la generación 1, se deben de tener a los ganadores de la generación 0, porque cada nueva generación depende de la combinación de características de los ganadores previos.

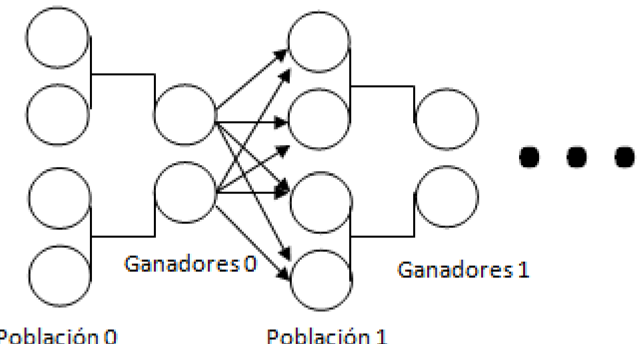


Ilustración 58. La generación 1 depende de la generación 0

Sin embargo, si es posible hacerlo con los métodos restantes: Selección y reproducción de ganadores. Pero de manera similar a la producción de nuevas generaciones, la reproducción depende de la selección. Es decir, para comenzar a ejecutar el segundo, es necesario que el primero haya concluido. Esto se muestra en la ilustración 59:

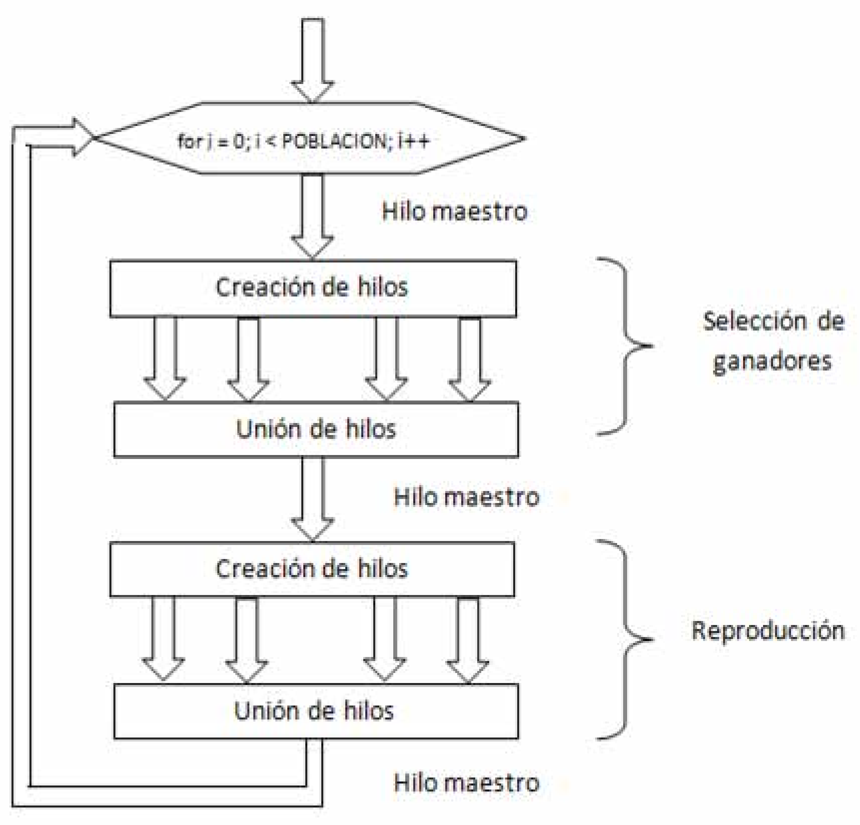


Ilustración 59. Representación del paralelismo de los métodos

Para traducir esto al código de la ilustración 60, se utilizaron dos directivas de OpenMP. Ambas tienen tres constructores.

```
//Se crean nuevas generaciones
for( i = 0; i < GENERACIONES; i++){
    //imprimir( "Poblacion", b, i, 1);

    //Se seleccionan los ganadores
    #pragma omp parallel for firstprivate(b) lastprivate(b) private(k)
    for( j = 0; j < POBLACION; j++ ){
        if( fun( binToDec( b[2*j] ) ) > fun( binToDec( b[2*j+1] ) ) )
            for( k = 0; k < ATTRIBS; k++)
                ganadores[j][k] = b[2*j][k];
        else
            for( k = 0; k < ATTRIBS; k++)
                ganadores[j][k] = b[2*j+1][k];
    }

    //imprimir("Ganadores", ganadores, i, 2);

    //Combinaciones de ganadores para una nueva generacion
    #pragma omp parallel for firstprivate(ganadores) lastprivate(temp) private(gen, k, l)
    for( n = 0; n < POBLACION/2; n++ ){
        //Seleccionamos un ganador diferente al actual
        do{
            k = rand() % (POBLACION/2);
        }while( k == n );
        //Combinacion entre ganadores
        gen = rand() % (ATTRIBS-2) + 1;
        for( l = 0; l < gen; l++ )
            temp[2*n][l] = ganadores[n][l];
        for( ; l < ATTRIBS; l++ )
            temp[2*n][l] = ganadores[k][l];

        gen = rand() % (ATTRIBS-2) + 1;
        for( l = 0; l < gen; l++ )
            temp[2*n+1][l] = ganadores[k][l];
        for( ; l < ATTRIBS; l++ )
            temp[2*n+1][l] = ganadores[n][l];
    }

    //Reemplazar la poblacion anterior con la nueva
    for( k = 0; k < POBLACION; k++)
        for( l = 0; l < ATTRIBS; l++)
            b[k][l] = temp[k][l];
}
```

Ilustración 60. Código con directivas de OpenMP

- *firstprivate*: Al crear la región paralela, todos los hilos tienen una instancia de la matriz indicada entre paréntesis. Ésta está inicializada con los valores que tenía antes de la directiva.
- *lastprivate*: Cuando termina la ejecución de la región, el hilo maestro almacena en la matriz indicada entre paréntesis los datos que cada hilo modificó.
- *private*: Cada hilo tiene una instancia de las variables indicadas entre paréntesis.

5.5 Análisis de resultados

Los resultados que se obtuvieron fueron los esperados. No conviene paralelar este algoritmo, debido a la gran cantidad de hilos que se crean en cada generación. Aunque éstos tienen una carga de trabajo importante, el tiempo de creación y espera (cuando concluyen sus tareas) es mayor.

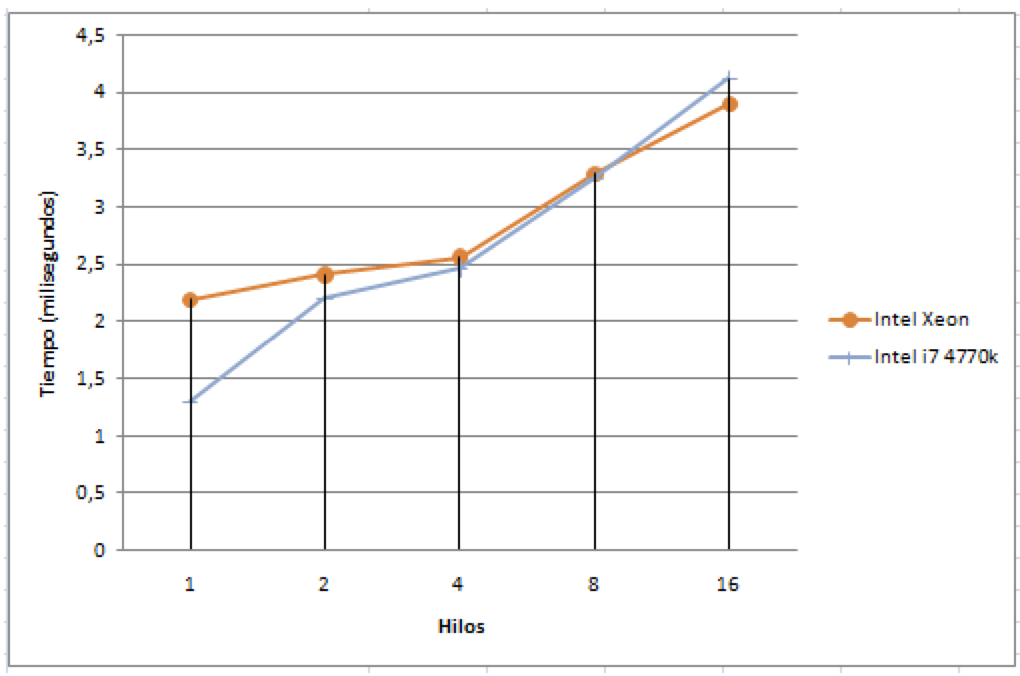


Ilustración 61. Tiempo de ejecución con diferentes procesadores

En la gráfica de la ilustración 61 se puede observar que el programa tarda más en completar la tarea a mayor cantidad de hilos en ejecución, debido a que el tiempo requerido en el proceso de creación/término de cada hilo es mayor al tiempo que le toma a cada uno completar su tarea.

El procesador i7 es más rápido que el Xeon hasta los 4 hilos, pero la ventaja se pierde cuando éstos aumentan. Consideramos que esto se debe a que el primero tiene una mayor velocidad de reloj, pero el segundo tarda menos en crear hilos.

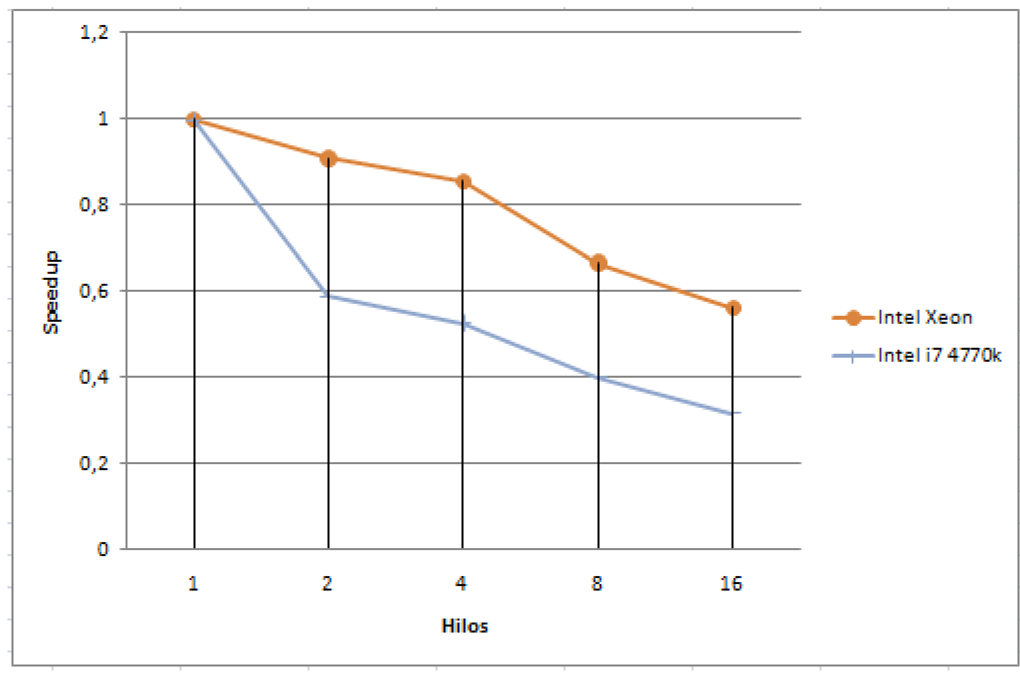


Ilustración 62. Speedup con diferentes cantidades de hilos

La gráfica de speedup describe la aceleración. Ésta es negativa en ambos casos porque el programa tarda más a mayor cantidad de hilos.

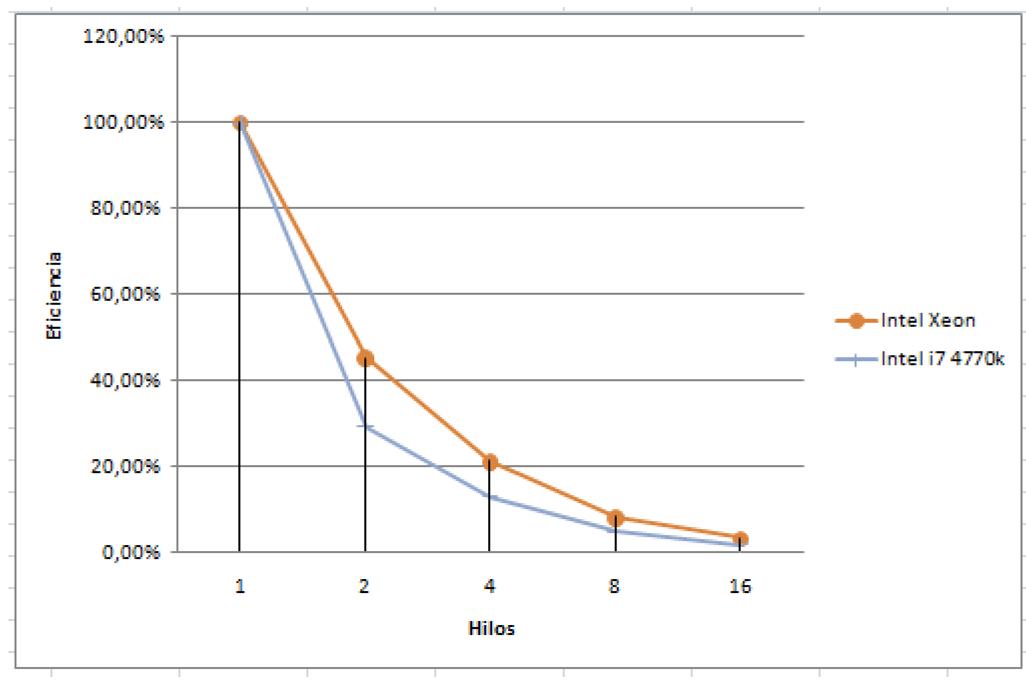


Ilustración 63. Eficiencia

Por último, el algoritmo, al ser más lento con mayor cantidad de hilos y al utilizar más recursos, tiene muy baja eficiencia.

Como era de esperarse, la *creación y unión* de hilos afecta negativamente en el rendimiento, a pesar de que se incrementen los datos a procesar y la función demande mayor tiempo de procesamiento.

6. Algoritmos para generar números primos

Un número primo es aquél que pertenece a los naturales mayores a 1 y que son divisibles únicamente entre dos valores: el mismo y 1.

Una característica de éstos es que se trata de un conjunto infinito.[51] Se cree que Euclides hizo la primer demostración alrededor del año 300 a.C. Él utilizó el método de reducción al absurdo, es decir, supuso que hay un número primo p que es el último número primo y halló que eso es imposible.

Supongamos que p es el número primo más grande y construyamos otro número q, resultado de multiplicar todos los números primos hasta el último, p; y después sumarle 1:

$$q = (2 * 3 * 5 * 7 * 11 * \dots * p) + 1$$

Evidentemente q no es divisible por ningún primo, pues siempre tendría como residuo 1; luego q es divisible sólo por 1 y por sí mismo, es decir, q es primo.

Por otra parte q es mayor que p. Por lo tanto, p no es el mayor número primo. Se puede concluir entonces que el conjunto de números primos es infinito.

Aplicación en criptografía:

La teoría de los números primeros es una de las pocas áreas de la matemática pura que ha encontrado aplicación directa en el mundo real, concretamente en la criptografía.[52] La criptografía estudia los métodos para cifrar mensajes secretos de manera que solo puedan ser descifrados por el receptor, y por nadie más que los pueda interceptar. El proceso de cifrado requiere el uso de una clave secreta; lo más corriente es que para descifrar el mensaje, al receptor solo le hace falta aplicar la clave al revés. Con este procedimiento, la clave de cifrado y descifrado es el elemento más débil de la cadena de seguridad. En primer lugar, el emisor y el receptor han de ponerse de

[51] Herrero P. (2007). *La prueba de Euclides*. Noviembre 2014, de Universidad de Murcia. Sitio web: http://www.um.es/docencia/pherrero/mathis/primos/infinitos_primos.htm
[52] Pina C. (2009). *Curiosidades*. Noviembre, 2014, de Estany. Sitio web: http://pinux.info/primos/curiosidades.html

acuerdo sobre los detalles de la clave y la transmisión de esta información es un proceso arriesgado. Si un tercero, un enemigo, puede interceptar la clave mientras se está intercambiando, podrá traducir todo aquello que se comunique desde entonces. En segundo lugar se han de cambiar las claves de vez en cuando para preservar la seguridad de las transmisiones y cada vez que esto ocurre hay un nuevo riesgo de que la clave sea interceptada.

El problema de la clave gira en torno al hecho de que aplicarla en un sentido cifrará el mensaje y aplicarla en el sentido contrario lo descifrará; es decir, que descifrar un mensaje es casi tan fácil como cifrarlo. A pesar de ello, la experiencia nos dice que hay muchas situaciones cotidianas en que descifrar es mucho más difícil que cifrar.

Durante la década de los setenta, Whitfield Diffie y Martin Hellman se propusieron encontrar un proceso matemático que fuese fácil de llevar a término en una dirección, pero muy difícil de realizar en la dirección opuesta. Un proceso como éste formaría la clave perfecta para los mensajes cifrados. Por ejemplo, yo podría tener la clave dividida en dos partes y publicar la parte correspondiente al cifrado. Cualquiera podría enviarme mensajes cifrados, pero solo yo conocería la parte descifradora de la clave.

En 1977 Ronald Rivest, Adi Shamir y Leonard Adleman, un equipo de matemáticos y científicos informáticos del Massachusetts Institute of Technology, se dieron cuenta que los números primos eran la base ideal para un proceso de cifrado fácil y descifrado difícil.

Si quisiera tener mi propia clave, tendría que tomar dos números primos muy grandes, de hasta 80 dígitos cada uno, y los multiplicaría para encontrar un número no primo más grande. Para cifrar el mensaje solo haría falta conocer el número grande no primo; para descifrarlo haría falta conocer los dos números primos originarios que fueron multiplicados, conocidos como factores primos. Ahora puedo publicar el número grande no primo (parte cifradora de la clave) y guardarme los dos factores primos (parte descifradora). Lo que cuenta es que aunque todo el mundo pueda conocer el número grande no primo, la dificultad de obtener los números primos sería inmensa.[53]

6.1 Algoritmo por fuerza bruta

Descripción

Si nos remitimos a la definición de número primo, es aquél que sólo es divisible entre él mismo y uno. El algoritmo consiste en recorrer la primera mitad de un número *n* buscando algún divisor. Si no se encuentra, *n* es primo.

Ejemplo: Si queremos conocer si el 77 es un número primo, basta con recorrer los primos anteriores; si alguno tiene como residuo 0, significa que no es primo, tal como se muestra en la tabla 53:

---

[53] Gómez J. (2010). *Números primos y criptografía*. Noviembre 2014, de Parte Aguas. Sitio web: http://jlgs.com.mx/articulos/ciencia/numeros-primos-y-criptografia/

Tabla 53 Ejemplo para hallar primos

Divisor primo	Residuo
2	1
3	2
5	2
7	0

Análisis e implementación serial

El algoritmo consiste básicamente en recorrer desde el número 2 (el primer primo) hasta alcanzar el límite deseado *n*, agregando todos los números primos hallados a una lista, como se muestra en la ilustración 64:

```
for( i = 0; i < N/2; i++)
    a[i] = 0;

a[0] = 2 ;

for( i = 3; i < N; i++ )
    for( j = 0; j < N/2; j++ ){
        if( i % a[j] == 0 )
            break;
        if( a[j+1] == 0 ){
            a[j+1] = i;
            break;
        }
    }
```

Ilustración 64. Implementación serial del algoritmo de fuerza bruta

El primer ciclo llena la lista de primos con ceros, que serán utilizados como valores control. Se añade el 2 a la lista y se comienza a iterar desde el 3 hasta el límite deseado. Anidado, existe otro ciclo que recorre los primos ya encontrados, para saber si el valor *i* se agrega o no.

Entonces, el primer ciclo tiene complejidad de *O(n)*, así como el segundo. Esto se debe a que, aunque los primos son menos que los naturales, no existe una relación que nos diga cuántos primos existen hasta un número *n*.

$$Por\ lo\ tanto: O(n^2)$$

6.2 Algoritmo Criba de Eratóstenes

Descripción

Al igual que el de fuerza bruta, este algoritmo recibe como entrada un número *n* hasta el cual se desean conocer los primos. Se comienza con el número 2 (el primer primo) y en una lista de tamaño *n* se marcan todos sus múltiplos. Después, se hace lo mismo con los siguientes números no marcados hasta que se alcanza el valor raíz de *n*. Al final, los números no marcados son primos.

Ejemplo. Se quieren conocer los números primos hasta el 100. Los valores pueden verse como un arreglo, representado desde la tabla 54 hasta la 58:

Tabla 54. Representación de los n números

		2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Tabla 55. Se selecciona el primer elemento no marcado y se marcan sus múltiplos

		2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Tabla 56. Se selecciona el siguiente elemento no marcado y se marcan sus múltiplos

		2	**3**	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Tabla 57. Se selecciona el siguiente elemento no marcado y se marcan sus múltiplos

		2	3	4	**5**	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Tabla 58. Se selecciona el siguiente elemento no marcado y se marcan sus múltiplos

		2	3	4	5	6	**7**	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

El algoritmo termina cuando se alcanza el valor raíz cuadrada de n, en este caso 10. Los elementos que quedan sin marca son los primos.

Análisis e implementación serial

El código de está compuesto por un ciclo que itera desde *k = 2* (el primer primo) hasta que *k * k* sea menor o igual que *n* (es decir, hasta que *k* alcance la raíz cuadrada de *n*). Anidado, hay otro ciclo que itera n veces marcando los múltiplos, tal como en la ilustración 65.

```
for( k = 2; k*k <= N; k++ ){
    i = k + 1;
    while( i < N){
            if( i % k == 0 )
                h_prueba[i] = MARK;
        i++;
    }
}
```

Ilustración 65 Implementación serial de la criba de Eratóstenes

$$Entonces, el\ algoritmo\ tiene\ complejidad\ O(\sqrt{n} * n)$$

6.3 Selección del algoritmo a paralelar y justificación

De los dos algoritmos planteados, consideramos que es mejor paralelar la criba de Eratóstenes.

Justificación:

Este algoritmo es altamente paralelable. El ciclo anidado se puede descomponer en dominio (los elementos a marcar son independientes entre sí). Esto parece indicar que una buena herramienta sería CUDA nuevamente, para aprovechar la característica del algoritmo.

6.4 Análisis e implementación paralela en CUDA

Se puede notar que el primer ciclo del algoritmo no se puede paralelar, pues para conocer el siguiente elemento no marcado, los múltiplos de los primos previos ya deben estar marcados. Es el ciclo anidado el que será explotado. Por ejemplo, en el primer paso del ejemplo, hilos diferentes pueden marcar los múltiplos en verde de la ilustración 66:

		2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Ilustración 66 Diferentes hilos pueden marcar los múltiplos del primo seleccionado

```
cudaStream_t stream1;
HANDLER_ERROR_ERR(cudaStreamCreate (&stream1));

HANDLER_ERROR_ERR( cudaMalloc( &d_a, BYTES ) );
HANDLER_ERROR_ERR( cudaMemcpy( d_a, h_a, BYTES, cudaMemcpyHostToDevice));

for( k = 2; k*k <= N; k++)
    kernelCriba<<<64, 256, 0, stream1 >>>(k, d_a);

HANDLER_ERROR_ERR( cudaMemcpy( h_a, d_a, BYTES, cudaMemcpyDeviceToHost));
HANDLER_ERROR_ERR( cudaFree( d_a ) );

HANDLER_ERROR_ERR( cudaStreamDestroy( stream1 ) );
```

Ilustración 67 Creación del stream, alojar, copiar, ejecutar el kernel y liberar memoria

Para poder implementar el código en CUDA, hay que hacer algunas modificaciones al serial. Lo primero es implementar una función que aloje y copie las variables ya inicializadas del CPU al GPU (ilustración 67). Se puede notar que se crea un *stream*. Éste sirve para garantizar que una llamada al *kernel* se ejecuta después de que termine la anterior, pues hay que recordar que la ejecución de éste es asíncrona (el código continúa al llamarlo).Después, que se llame a la función que realizará la búsqueda, se copian los resultados del GPU al CPU y se liberan los recursos utilizados.

```
int *h_a = (int*)malloc( N * sizeof(int) );
int i;

for( i = 0; i < N; i++)
    h_a[i] = UNMARK;
```

Ilustración 68 Inicialización del arreglo

Cabe mencionar que todos los elementos están inicialmente desmarcados, como lo indica el código de la ilustración 68.

```
__global__ void kernelCriba(int k, int *d_a){
    int i = threadIdx.x + blockIdx.x*blockDim.x;

    while(i < N){
        if(k*k <= i){
            if(i%k == 0)
                d_a[i] = MARK;
        }
        i+=blockDim.x*gridDim.x;
    }
}
```

Ilustración 69 Función *kernel* de la Criba de Eratóstenes

La función *kernel* de la ilustración 69 obtiene el índice de cada hilo en la variable *i* y posteriormente itera sobre el arreglo *n* veces buscando y marcando los múltiplos.

6.5 Análisis de resultados

El código en CUDA puede ser ejecutado con diferentes configuraciones, variando la cantidad de bloques e hilos por bloque; la multiplicación de estos dos da como resultado el total de hilos en ejecución. Para la toma de medidas, a continuación, se han elegido algunas configuraciones representativas en diferentes GPUs:

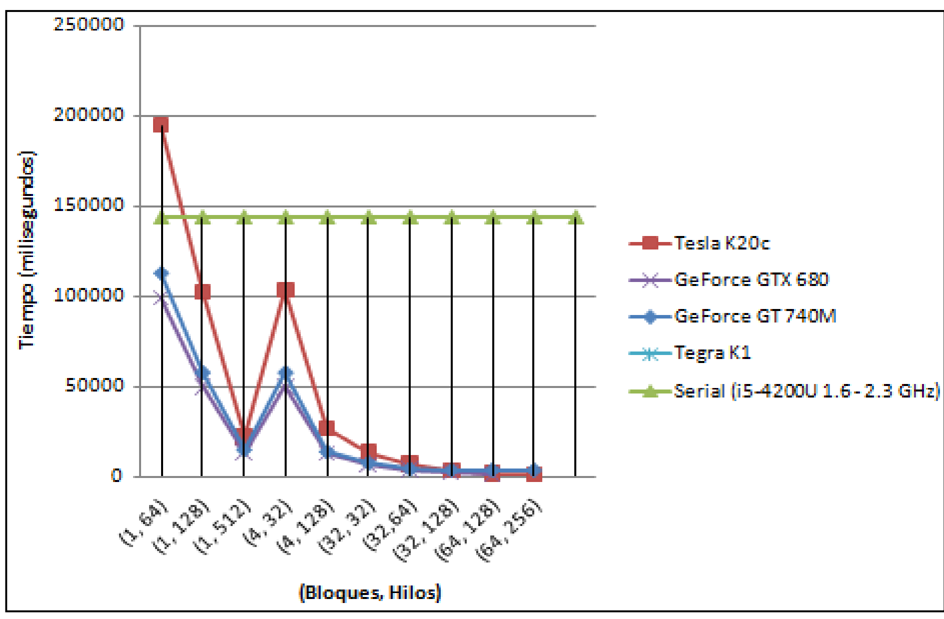


Ilustración 70. Tiempo de ejecución del kernel en milisegundos con diferentes configuraciones

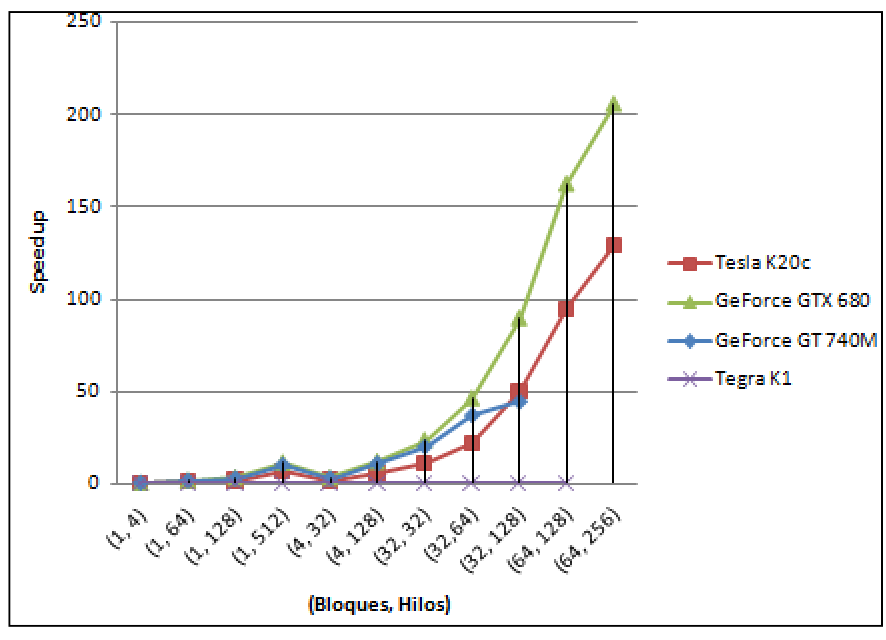


Ilustración 71. Speedup de las diferentes GPUs

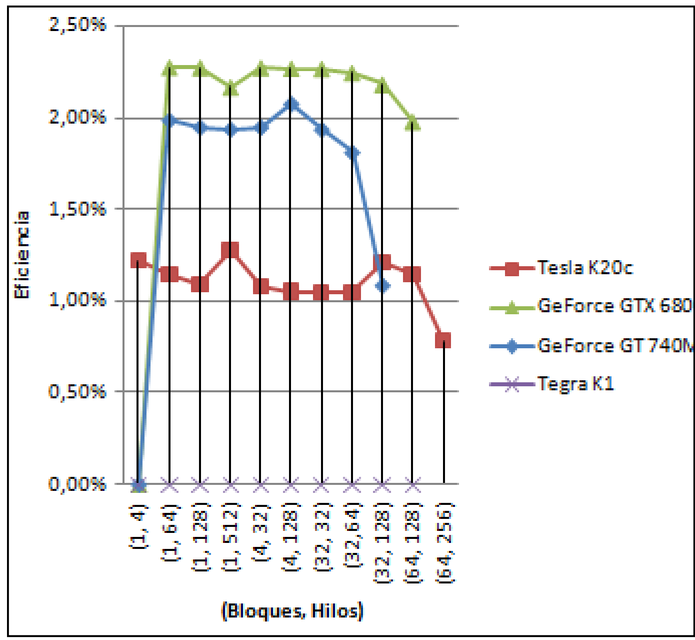


Ilustración 72. Eficiencia

Las gráficas de la ilustración 70 y 71 son medidas con respecto al tiempo serial del procesador i5. De las gráficas se pueden concluir los siguientes aspectos:

- Tanto Tegra K1 como GT 740M alcanzan un límite en el *speedup*; es decir, todos sus núcleos están trabajando y el programa ya no puede ser acelerado.
- GTX 680 es la GPU que más puede acelerar la aplicación, llegando casi a 200x.
- La GPU GTX 680 es la tarjeta que tiene mayor aceleración comparada con las demás, debido a que tiene la mayor velocidad de reloj. Como se puede observar.
- Al ser una GPU diseñada para dispositivos móviles (es decir, que consuma muy poca energía), la GPU Tegra K1 no alcanza a llegar a 1 en *speedup*; esto significa que a pesar de hacer uso de todo su poder, éste no puede ser comparado con el procesador de una computadora convencional.
- A pesar de que la GPU Tesla K20c soporta las configuraciones de ejecución más grandes, la GTX 680 puede acelerar mejor la aplicación, pues el código es demasiado paralelable y esta tarjeta aprovecha su mayor velocidad de reloj todo el tiempo de ejecución.

Conclusiones

El trabajo realizado a lo largo del presente documento, nos permitió acercarnos a diferentes tecnologías de multiprocesamiento, tanto en hardware como en software. En la primera utilizamos tarjetas gráficas principalmente, pero cuando el problema se tornaba muy complejo para programar por la transferencia de información CPU-GPU, sincronización y comunicación de hilos, acudimos a procesadores multinúcleos. Para la segunda utilizamos CUDA, pyCUDA y OpenMP.

Podemos concluir que el lenguaje C/C++ es superior a Python para realizar tareas de forma paralela. Algunos algoritmos los implementamos en Python, que es muy fácil de programar, pero es un lenguaje interpretado y tiene poco soporte para la extensión pyCUDA. A diferencia de C/C++, que, a pesar de ser un poco más complejo, es un lenguaje compilado (lo que implica que se ejecute hasta 100 veces más rápido en muchos casos) y tiene mucho soporte para desarrolladores por parte de Nvidia, así como diversas herramientas para facilitar la programación paralela.

Se puede observar que la tarjeta GeForce GTX 680 es superior en tiempos comparada contra la Tesla K20c, a pesar de que la segunda tiene más núcleos y mayor poder de cómputo. Concluimos que es debido a que nuestras aplicaciones no necesitan datos de doble precisión, donde la Tesla tiene 1,17 Tflops y sería sumamente superior a cualquier otra; la aparente superioridad de la 680 se debe a que, al ser una tarjeta diseñada para videojuegos, tiene una mayor velocidad de reloj y puede operar más rápido. Esta ventaja se pierde cuando se envían hilos en ejecución en configuraciones tales como 64 bloques y 256 hilos (es decir, 16384 hilos en ejecución).

A pesar de que una GPU acelera ciertas aplicaciones mucho más que un procesador multinúcleo, tiene una eficiencia muy baja. Concluimos que esto se debe a que el *speedup* no se comporta de manera lineal: si un problema tarda t tiempo en un procesador, no tarda $t/2$ en dos o $t/4$ en 4, sino más (como se explicó en el capítulo 2). La eficiencia es realmente baja debido a que se utilizan miles de unidades de procesamiento y el parámetro es un código serial ejecutado en un CPU; es decir, a una velocidad de reloj superior.

Como era de esperarse, el desempeño de un GPU diseñado para dispositivos móviles es menor que el de los demás. Éste tiene un consumo de energía relativamente bajo, pero con un buen rendimiento. Aún así, explotando sus 192 núcleos CUDA, el programa alcanza tiempos similares a los que un CPU de una computadora de escritorio proporciona.

Paralelar algunos algoritmos puede resultar más complejo que en otros. Tenemos que resolver, en primer lugar, si se pueden descomponer en dominio o de manera funcional y reconocer las partes de código que son estrictamente secuenciales. Al hacer esto, tenemos que considerar los posibles cambios, posteriormente diseñar una solución y elegir la herramienta adecuada. Elegir la herramienta adecuada es muy importante, pues el no hacerlo podría implicar demasiado esfuerzo a la hora de implementar. Finalmente, programar la solución. Como vimos a lo largo de los capítulos, algunos programas se aceleran contundentemente, mientras que otros no tanto. Entonces, concluimos que tenemos que valorar el esfuerzo que se tiene que hacer en contra de los beneficios que se obtendrán y decidir si vale la pena o no.

Además, la razón de elaborar los programas que fueron implementados en OpenMP es porque antes los codificamos con CUDA, pero debido a problemas de sincronización de los hilos, a partir de configuraciones de ejecución no muy elevadas, comenzaban a arrojar resultados inesperados. Concluimos que este es un buen ejemplo de esfuerzo versus beneficio, porque a pesar de que es más sencillo programar en OpenMP que en CUDA, un equipo tendría que ser muy robusto en cuando a procesador multinúcleo para igualar a un GPU.

Concluimos que, una vez identificado el tipo de descomposición que se puede realizar al problema, conviene más paralelar algoritmos que demandan mayor tiempo de procesamiento que en copia de memoria. Esto con base al análisis del algoritmo genético (donde la copia de memoria es permanente durante la ejecución) contra la Criba de Eratóstenes (donde sólo hay una copia de memoria y el resto es tiempo de ejecución).

Otra conclusión a la que pudimos llegar es que nuestros códigos funcionarán cada vez en una mayor cantidad de equipos de cómputo, tan sólo instalándoles las herramientas necesarias. Inicialmente, consideramos que probar los programas es diferentes plataformas y computadoras podría ser un problema, pero no lo fue. En la actualidad, prácticamente todos los equipos tienen dos o más unidades de procesamiento, y con la eventual reducción de precios sobre éstos, creemos que cada vez será más factible adquirir equipos robustos (con CPU multinúcleo, mayor memoria RAM y GPU), además de que Nvidia está incursionando en el mercado de móviles con GPUs muy eficientes, de alto rendimiento (similar al de consolas de videojuegos) y baratos.

A pesar de que programar en paralelo es más sencillo hoy que hace algunos años, continúa siendo más complejo que el paradigma secuencial, como se aprecia a lo largo del presente trabajo. Creemos que con el paso del tiempo esta barrera será cada vez menor, pues eventualmente se simplificarán estas extensiones del lenguaje de programación para que mayor cantidad de desarrolladores las utilicen en sus aplicaciones. Inclusive hoy día, el cómputo paralelo es fundamental para el desarrollo de nuevas tecnologías: automóviles con piloto automático, química molecular, control aéreo, cómputo en la nube, simulaciones del cuerpo humano, del universo, etc. Todas estas tareas serían imposibles sin el cómputo paralelo.

Hemos aprendido bastante del presente trabajo de tesis. Por ejemplo, para desarrollar programas donde lo más importante es el desempeño un lenguaje compilado es mejor que uno interpretado, se obtienen mejores resultados donde el algoritmo demanda mayor tiempo de procesamiento

comparado con el que tarda en copiar datos a memoria; por el contrario, los peores resultados se obtienen cuando la creación y unión de hilos es periódica. Conocimos también diferentes plataformas y dispositivos hardware para probar los programas y que dentro de una misma categoría, existen algoritmos con diferentes características para seleccionar y paralelar.

Bibliografía

- Gutiérrez A. (diciembre 2009). *Sistemas de Multiprocesamiento*. Agosto 2014, de UPICSA Sitio web: http://www.sites.upiicsa.ipn.mx/polilibros/portal/polilibros/P_terminados/PolilibroFC/Unidad_VI/ Unidad%20VI_4.htm#IrInicioUnidad
- Schauer B. (septiembre 2008). *Multicore Processors – A Necessity*. Agosto 2014, de Pro Quest. Sitio web: http://cse.unl.edu/~seth/990/Pubs/multicore-review.pdf
- Top 500. (2014). *Lista de junio de 2014*. Agosto 2014. Sitio web: http://www.top500.org/
- Nvidia Corporation. (2012). *Qué es el cálculo acelerado en la GPU*. Septiembre 2014, de Nvidia Sitio web: http://www.nvidia.es/object/gpu-computing-es.html
- Nvidia Corporation (2012). *Qué es el cálculo acelerado en la GPU* [imagen] Sitio web: http://www.nvidia.es/object/gpu-computing-es.html
- Cloud Computing. (2013). *Computación en nube*. Septiembre 2014, de Computación en la nube. Sitio web: http://www.computacionennube.org/
- Sandoval L. (2012). *Tutorial de PPSS*. Septiembre 2014, de Facultad de Ingeniería, UNAM. Sitio web: http://lcomp89.fi-b.unam.mx/
- Indiana University. (2014). *Open Source High Performance Computing*. Octubre 2014, de Open MPI. Sitio web: http://www.open-mpi.org/
- Java Tutorials. (2014). *Concurrency*. octubre 2014, de Oracle. Sitio web: http://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html
- Nvidia Corporation. (2014). *GPU Computing: The Revolution*. Octubre 2014, de Nvidia. Sitio web: http://www.nvidia.com/object/cuda_home_new.html
- Nvidia Corporation. (2014). *CUDA Toolkit Documentation v6.5*. Octubre 2014, de Nvidia. Sitio web: http://docs.nvidia.com/cuda/index.html#axzz3UsKSpImT
- Karimi K. (2013*). A Performance Comparison of CUDA and OpenCL*. Agosto 2014, de D-Wave Systems Inc. Sitio web: http://arxiv.org/ftp/arxiv/papers/1005/1005.2581.pdf
- Gold Standard Group. (2014). *OpenCL*. Septiembre 2014, de Khronos. Sitio web: https://www.khronos.org/opencl/
- Sara. (2007). *Gauss y la suma de los n primeros*. Febrero 2015. Sitio web: https://sferrerobravo.wordpress.com/2007/11/24/gauss-y-la-suma-de-los-n-primeros/
- Department of Computer Science, Kent State University. (2010*). Design and Analysis of Algorithms.* Febrero 2015. Sitio web: http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/ mergeSort.htm
- Aznar E. (2007). *Números de Fibonacci*. Febrero 2015. Sitio web: http://www.ugr.es/~eaznar/fibo.htm

- Fuentes A. (2011). *Problema del agente viajero*. Febrero 2015. De Universidad Autónoma del Estado de Hidalgo. Sitio web: http://www.uaeh.edu.mx/scige/boletin/tlahuelilpan/n3/e5.html
- Díaz G.(2008). *Ley de Amdahl y Ley de Moore*. De Universidad de los Andes. Octubre 2014. Sitio web: http://webdelprofesor.ula.ve/ingenieria/gilberto/paralela/05_LeyDeAmdahlYMoore.pdf
- Liu J. (2011*). Estimation of theoretical maximum speedup ratio for parallel computing of grid-based distributed hydrological models.* Octubre, 2013. Sitio web: http://dl.acm.org/citation.cfm?id=2527901
- Sandoval L. (2012). *Propuesta de un Modelo de Desarrollo de Sistemas de Software de Procesamiento Paralelo/Distribuido*. De Facultad de Ingeniería, Universidad Nacional Autónoma de México.
- Guerrero D. (2010). *Programación Distribuida y Paralela*. Universidad de Granada. Septiembre 2014. Sitio web: http://lsi.ugr.es/jmantas/pdp/tutoriales/tutorial_mpi.php?tuto=03_pi
- Cáceres A. (2005). *La serie Fibonacci*. Agosto 2014, de CINVESTAV Sitio web: http://computacion.cs.cinvestav.mx/~acaceres/courses/estDatosCPP/node38.html
- Idelsohn S. (2004). *Parallel Processing: Fynn´s Taxonomy.* Septiembre 2014. Sitio web: https://web.cimne.upc.edu/groups/sistemes/Servicios%20de%20calculo/Barcelona/public/14716537-Flynns-Taxonomy-and-SISD-SIMD-MISD-MIMD.pdf
- Smeen M. (2010). *CPU Architecture*. Octubre 2014. Sitio web: http://pgdcce.blogspot.mx/2010/09/cpu-architecture.html
- Domeika M. (2006). *Development and Optimization Techniques for Multicore Processors*. Octubre 2014. Intel Corp. Sitio web:. http://www.embedded.com/design/mcus-processors-and-socs/4006702/Development-and-Optimization-Techniques-for-Multicore-Processors
- Orts S. (2005). *Introducción a la Computación paralela con GPUS*. Septiembre 2014. Universidad de Alicante. Sitio web: http://www.dtic.ua.es/jgpu11/material/sesion1_jgpu11.pdf
- Charte F. (2009). *Tipos de Shaders - Procesadores de Geometría*. Septiembre 2014. Sitio web: http://fcharte.com/Default.asp?noticias=2&a=2009&m=12&d=15
- Díaz A. (2008). *Análisis y Complejidad de Algoritmos*. Septiembre 2014. CINVESTAV. Sitio web: http://delta.cs.cinvestav.mx/~adiaz/anadis/Sorting2.pdf
- Aldana C. (2010). *Algoritmos de Ordenamiento: Countingsort*. Octubre 2014 Sitio web: http://upcanalisisalgoritmos.wikispaces.com/file/view/ALGORITMOS+DE+ORDENAMIENTO+COUNTINGSORT.pdf
- Commons C.. (2009). *Shell sort*. Octubre 2014, de Conoce3000 Sitio web: http://www.conoce3000.com/html/espaniol/Libros/PascalConFreePascal/Cap08-03-Ordenamiento%20Shell%20%28Shell%20sort%29.php
- Black P. (2009). *Bucket sort*. Octubre 2014, de NIST Sitio web: http://xlinux.nist.gov/dads/HTML/bucketsort.html

- Bustos B. (2008). *Algoritmo Knuth-Morris-Pratt*. Agosto 2014, de Universidad de Chile. Sitio web: http://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/BusqTexto/
- Sedgewick R. (1995). *Algoritmos en C++.* Princeton: Ediciones Díaz de Santos.
- Dorronsoro B. (2002). *Algoritmos Evolutivos Celulares*. Málaga: ETSI Informática.
- Herrero P. (2007). *La prueba de Euclides*. Noviembre 2014, de Universidad de Murcia. Sitio web: http://www.um.es/docencia/pherrero/mathis/primos/infinitos_primos.htm
- Pina C. (2009). *Curiosidades*. Noviembre, 2014, de Estany. Sitio web: http://pinux.info/primos/curiosidades.html
- Gómez J. (2010). *Números primos y criptografía*. Noviembre 2014, de Parte Aguas. Sitio web: http://jlgs.com.mx/articulos/ciencia/numeros-primos-y-criptografia/

Anexo 1: Códigos de programas seriales

1.1 Algoritmos de ordenamiento

Ordenamiento por cuenta (C)

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define tamano 100
#define min 3
#define max 10

//Función que genera números aleatorios en el rango [min, max]
void generar_numeros(int *a){
    int i;
    for(i = 0; i < tamano; i++){
        a[i] = rand()%(max - min + 1) + min;
        printf("%5d", a[i]);
    }
}

//El arreglo para contar se inicializa a ceros
void inicializar(int *b){
    int i;
    for(i = 0; i < (max-min + 1); i++)
        b[i] = 0;
}

//En b, se cuentan las ocurrencias de los números de a
void contar(int *a, int *b){
    int i;
    for(i = 0; i < tamano; i++)
        b[ a[i]-min ]++;
}

//En el arreglo c, se acomodan las ocurrencias
void ordenar(int *b, int *c){
    int i, j=0, k=0;
    for(i = 0; i < (max - min + 1); i++){
        while( b[i] > 0 ){
            c[j] = i + min;
            b[i]--;
            j++;
        }

}

//Imprime el resultado
void resultado( int *c ){
    int i;
    for(i = 0; i < tamano; i++)
        printf("%5d", c[i]);
}
```

```
50
51  int main(int argc, char *argv[]){
52      int *a, *b, *c;
53
54      //Arreglo inicial, para contar y resultado
55      a = (int *)malloc(tamano * sizeof(int));
56      b = (int *)malloc( (max - min + 1) * sizeof(int) );
57      c = (int *)malloc(tamano * sizeof(int));
58
59      //Semilla para generar números aleatorios
60      srand( time(NULL) );
61              //Las impresiones son opcionales
62      printf("Original:\n");
63      generar_numeros( a );
64      inicializar( b );
65      contar( a, b );
66      printf("\n");
67      ordenar( b, c );
68      printf("\nOrdenados:\n");
69      resultado( c );
70      printf("\n");
71      return 0;
72  }
73
```

Ordenamiento casilleros -shell

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include <time.h>
4   #include <vector>
5   #include <omp.h>
6
7   #define N 1000000
8   #define R 10000
9   #define B 1000
10  #define H 8
11  #define P 10
12
13  //Shellsort serial
14  void shellsort(int *a,int tam){
15      int i, j, k, temp;
16      for (k = tam / 2; k > 0; k /= 2)
17          for (i = k; i < tam; i++)
18              for (j = i - k; j >= 0 && a[j] > a[j + k]; j -= k) {
19                  temp = a[j];
20                  a[j] = a[j + k];
21                  a[j + k] = temp;
22              }
23  }
24
25  //Convierte una matriz (doble apuntador) a un arreglo (un apuntador)
26  int *conArr2to1(int **c, int *t){
27      int *tmp = new int[N], ind = 0, i, j;
28      for ( i = 0; i < B; i++){
29          for ( j = 0; j < t[i]; j++)
30              tmp[ind + j] = c[i][j];
31          ind += t[i];
32      }
33      return tmp;
34  }
35
36  //Obtiene el tamaño de cada vector en un arreglo de vectores
37  int *tamVec( std::vector<int> *a){
38      int *tmp = new int[B], i;
39      for (i = 0; i < B; i++)
40          tmp[i] = a[i].size();
41      return tmp;
42  }
43
44  //Convierte un arreglo de vectores a una matriz
45  int **conVtoA( std::vector<int> *a){
46      int **tmp = new int*[B], i;
47      for (i = 0; i < B; i++){
48          tmp[i] = new int[a[i].size()];
49          std::copy(a[i].begin(),a[i].end(),tmp[i]);
50      }
51      return tmp;
52  }
53
```

```
54  //Generador de arreglos aleatorios, 1 o 0
55  int *arreglo(int tam, int rango, char tipo){
56      int *tmp, i;
57      tmp = new int[tam];
58  
59      if (tipo == 'z'){
60          for (i = 0; i < tam; i++)
61              tmp[i] = 0;
62      }
63      else if (tipo == 'o'){
64          for (i = 0; i < tam; i++)
65              tmp[i] = 1;
66      }
67      else if (tipo == 'r'){
68          srand( time(NULL) );
69          for (i = 0; i < tam; i++)
70              tmp[i] = rand()%rango;
71      }
72      else{
73          printf("Error en el tipo");
74      }
75      return tmp;
76  }
77  
78  //Imprime un arreglo
79  void imprimeArr(int *a,int tam){
80      int i;
81      for (i = 0; i < tam; i++){
82          printf("%i ", a[i]);
83      }
84      printf("\n");
85  }
86  
87  void prueba(double *tiempo){
88      int *a, i;
89      double st, sp;
90      std::vector<int> tc1[B];
91      a = arreglo( N, R, 'r');
92  
93      //LLenado de los casilleros
94      for (i = 0; i < N; i++)
95          tc1[(int)( a[i]/(R/B) )].push_back( a[i] );
96  
97      //Convierte un vector en un arreglo simple
98      int **c1 = conVtoA(tc1);
99  
100     //se obtiene el tamaño de cada casillero
101     int *t = tamVec(tc1);
102 
103     //implementación paralela con medición de tiempo
104     st = omp_get_wtime();
105     for(i = 0; i < B; i++)
106         shellsort(c1[i], t[i]);
107 
108     a = conArr2to1(c1,t);
109     sp = omp_get_wtime();
110 
111     *tiempo = sp-st;
112 }
```

```
113
114  int main(){
115      double t[P], p1 = 0.0;
116
117      for(int i=0;i<P;i++)
118          prueba(&t[i]);
119
120      for(int i=0;i<P;i++)
121          printf("%lf\n",t[i]);
122
123      for(int i=0;i<P;i++)
124          p1+=t[i];
125
126      printf("Promedio de tiempos:\n");
127      printf("%lf\n",p1/P);
128
129      return 0;
130  }
131
```

1.2 Algoritmos de búsqueda de cadenas

Knuth-Morris-Pratt (C)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   void normalizar(unsigned char *T){  //Función que convierte a minúsculas
6       for( ; *T; ++T){
7           *T = tolower(*T);
8       }
9   }
10
11  void precalcular_tablaKMP(unsigned char *P, int *F){
12      int pos = 2; //se comienza a analizar en la posición 2
13      int cnd = 0; //índice en P del sig. carácter del actual
14
15      F[0] = -1;
16      F[1] = 0; //Los primeros caracteres no se analizan
17
18      do{
19          if( P[pos -1] == P[cnd]){ //Siguiente candidato coincidente en la cadena
20              cnd++;
21              F[pos] = cnd;
22              pos++;
23          }
24          else if( cnd > 0){ //Cuando fallan las coincidencias
25              cnd = F[cnd];
26          }
27          else{ //no se hallaron candidatos coincidentes
28              F[pos] = 0;
29              pos++;
30          }
31      }while( pos <= strlen(P) );
32  }
33
34  void busquedaKMP(unsigned char *T, unsigned char *P, int *F, int *ocurrencia){
35      int k = 0, i = 0; //Variables para examinar T y P respectivamente
36      int j = 0;
37      if( strlen(T) >= strlen(P) ){
38          precalcular_tablaKMP(P, F);
39          while( (k+i) < strlen(T) ){
40              if( P[i] == T[k+i] ){
41                  if( i == (strlen(P)-1) ){
42                      ocurrencia[j] = k;  // Ocurrencia
43                      j++;
44                  }
45                  i++;
46              }
47              else{
48                  k = k + i - F[i];
49                  if( i > 0)
50                      i = F[i];
51              }
52          }
53      }
54      ocurrencia[j] = -1;  //Final del texto
55  }
```

```
56
57  int main(int argc, char *argv[]){
58      FILE *texto;
59      unsigned char T0[100000], T1[10000]; //Texto total y texto relativo
60      unsigned char P[50]; //cadena a buscar
61      int ocurrencia[100];
62      int F[50]; //Array de fallos
63      int i = 0;
64
65      texto = fopen("silmarillion", "r");
66      while( !feof(texto) ){
67          fgets(T1, 10000, texto);
68          if( !feof(texto) )
69              strcat(T0, T1); //concatenar todos los párrafos en T0
70      }
71
72      normalizar(T0);
73      printf("\n%s", T0);
74      fclose(texto);
75
76      printf("\nIntroduzca cadenas a buscar:\n");
77      gets(P);
78
79      busquedaKMP( T0, P, F, ocurrencia );
80
81      if(ocurrencia[0] != -1){
82          printf("Ocurrencias:");
83          while(ocurrencia[i] != -1){
84              printf("\n%d", ocurrencia[i]);
85              i++;
86          }
87      }
88
89      return 0;
90  }
```

Rabin-Karp (C++)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <math.h>
5   #include <time.h>
6   #include <ctype.h>
7   #include "CpuTimer.h"
8
9   //Función que obtiene el valor Hash de la cadena
10  int valor(char *t, int m, int d){
11      int i, v = 0;
12      for( i = 0; i < m; i++ )
13          v = ( d*v + t[i] );
14      return v;
15  }
16
17  //Función de búsqueda
18  void busqueda_RK( char *T, char *P ){
19      const int d = 50; //cardinalidad del conjunto
20      const int q = 33554393; //número primo grande que cabe en 32 bits
21      int n = strlen(T); //Se obtiene la longitud del texto
22      int m = strlen(P); //Se obtiene la longitud de la cadena a buscar
23      int p, t, i;
24
25      p = valor(P, m, d) % q; //Valor hash del patrón
26
27      //Se busca ese valor en el texto
28      for( i = 0; i <= (n-m); i++ ){
29          if( i < (n-m) )
30              t = valor(&T[i], m, d) % q;
31          if( p == t )
32              printf("%d\n", i);
33
34      }
35  }
36
37  //Función que convierte a minúsculas
38  void normalizar( char *T){
39      for( ; *T; ++T)
40          *T = tolower(*T);
41  }
42
43  int main(int argc, char *argv[]){
44      FILE *texto; //Archivo del cual se lee el texto
45      char *T0 = (char *)malloc(5000000 * sizeof(char)); //Texto total
46      char T1[100000];  //Texto relativo
47      char P[50]; //Cadena a buscar
48
49      //Se lee el archivo
50      texto = fopen("texto", "r");
51      while( !feof(texto) ){
52          fgets(T1, 100000, texto);
53          if( !feof(texto) )
54              strcat(T0, T1); //concatenar todos los párrafos en T0
55      }
56
```

```
57      fclose(texto);
58      normalizar(T0);
59
60      printf("\nIntroduzca cadenas a buscar:\n");
61      gets(P);
62
63      //Se inicia el reloj
64      CpuTimer timer;
65      timer.Start();
66      busqueda_RK( T0, P);
67      //Se detiene el reloj
68      timer.Stop();
69      printf( "Tiempo de ejecucion:  %f ms\n", timer.Elapsed() );
70
71      return 0;
72  }
```

1.3 Algoritmo genético (C)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4
5   #define POBLACION 16
6   #define ATTRIBS 16
7   #define GENERACIONES 1000
8
9   //Función para imprimir
10  void imprimir( char *str, int b[][ATTRIBS], int k, int l ){
11      int i, j;
12      printf("%s%d  ", str, k);
13      for( i = 0; i < POBLACION/l; i++){
14          printf(" [");
15          for( j = 0; j < ATTRIBS; j++){
16              if( j == ATTRIBS-1 )
17                  printf(" %d]", b[i][j]);
18              else
19                  printf(" %d,", b[i][j]);
20          }
21      }
22      printf("\n\n");
23  }
24
25  //Funciçon que convierte los atributos binarios a decimales
26  int binToDec( int *b ){
27      int i, j = 1;
28      long k = 0;
29      for( i = ATTRIBS-1; i >= 0; i--, j*=2 )
30          k += j * b[i];
31      return k;
32  }
33
34  //Función evaluada; en esta se busca el máximo relativo
35  long long fun( long a ){
36      return (long long) -a*a*a*a + 50000*a*a*a + 20000*a*a - 18*a + 5;
37  }
38
```

```
39  int main(){
40      int b[POBLACION][ATTRIBS], ganadores[POBLACION/2][ATTRIBS], temp[POBLACION][ATTRIBS];
41      int i, j, k, l, n, gen;
42      time_t t;
43  
44      srand( (unsigned) time( &t ) );
45  
46      //Se crea la poblacion inicial
47      for( i = 0; i < POBLACION; i++)
48          for( j = 0; j < ATTRIBS; j++)
49              b[i][j] = rand() % 2;
50  
51      //Se producen nuevas generaciones
52      for( i = 0; i < GENERACIONES; i++){
53          j = 0;
54          //Imprimimos la población actual (opcional)
55          imprimir( "Poblacion", b, i, 1);
56          //Se seleccionan los ganadores de la población actual en el arreglo de ganadores
57          while( j < POBLACION){
58              if( fun( binToDec( b[2*j] ) ) > fun( binToDec( b[2*j+1] ) ) )
59                  for( k = 0; k < ATTRIBS; k++)
60                      ganadores[j][k] = b[2*j][k];
61              else
62                  for( k = 0; k < ATTRIBS; k++)
63                      ganadores[j][k] = b[2*j+1][k];
64              j++;
65          }
66          //Se imprime a los ganadores de la generaciòn actual (opcional)
67          imprimir("Ganadores", ganadores, i, 2);
68  
69          //Combinaciones de ganadores para una nueva generacion
70          for( n = 0; n < POBLACION/2; n++ ){
71              //Seleccionamos un ganador diferente a n
72              do{
73                  k = rand() % (POBLACION/2);
74              }while( k == n );
75              //Combinacion entre los ganadores n y k
76              gen = rand() % (ATTRIBS-2) + 1;
77              for( l = 0; l < gen; l++ )
78                  temp[2*n][l] = ganadores[n][l];
79              for( ; l < ATTRIBS; l++ )
80                  temp[2*n][l] = ganadores[k][l];
81  
82              gen = rand() % (ATTRIBS-2) + 1;
83              for( l = 0; l < gen; l++ )
84                  temp[2*n+1][l] = ganadores[k][l];
85              for( ; l < ATTRIBS; l++ )
86                  temp[2*n+1][l] = ganadores[n][l];
87          }
88  
89          //Reemplazar la poblacion anterior con la nueva
90          for( k = 0; k < POBLACION; k++)
91              for( l = 0; l < ATTRIBS; l++)
92                  b[k][l] = temp[k][l];
93      }
94      return 0;
95  }
96  
```

1.4 Algoritmos para generar números primos

Por fuerza bruta (C++)

```
1    #include <stdio.h>
2    #include <vector>
3    #include <stdlib.h>
4
5    #define N 100
6
7    int main(  ){
8        //Arreglo para guardar primos
9        int *a = ( int* )malloc( N/2 * sizeof( int ) );
10       int i, j;
11
12       //Se inicia el arrelo a ceros
13       for( i = 0; i < N/2; i++)
14           a[i] = 0;
15
16       //Se almacena el primer primo
17       a[0] = 2 ;
18
19
20       for( i = 3; i < N; i++ )
21           for( j = 0; j < N/2; j++ ){
22               //Si se encuentra un divisor, no es primo
23               if( i % a[j] == 0 )
24                   break;
25               //Si se alcanza el final de la lista, es primo
26               if( a[j+1] == 0 ){
27                   a[j+1] = i;
28                   break;
29               }
30           }
31
32       //Se imprime la lista (opcional)
33       i = 0;
34       while( a[i] != 0 ){
35           printf("%d\t   %d\n", a[i], i);
36           i++;
37       }
38   }
39
```

Criba de Eratóstenes (C++)

```
1  #include <stdio.h>
2  #include "CpuTimer.h"
3  #include <stdlib.h>
4
5  #define MARK 1
6  #define UNMARK 0
7  #define N 100000000
8
9  void prueba( int *a ){
10     int i, k;
11
12     //Inicia el reloj
13     CpuTimer timer;
14     timer.Start();
15
16     //Se recorre desde el 2 hasta la raíz de N
17     for( k = 2; k*k <= N; k++ ){
18         if( a[k] == UNMARK ){
19             i = k;
20             //Se marcan los múltiplos del primo
21             while( i * k < N){
22                 a[i*k] = MARK;
23                 i++;
24             }
25         }
26     }
27
28     //Se detiene el reloj
29     timer.Stop();
30     printf("Tiempo CPU: %f ms\n", timer.Elapsed());
31 }
32
33 int main(){
34     //Arreglo donde se marcan los primos
35     int *a = (int*)malloc( N * sizeof(int) );
36     int i;
37
38     //Se marcan a 0 todos los elementos
39     for( i = 0; i < N; i++)
40         a[i] = UNMARK;
41
42     prueba( a );
43     free( a );
44
45     return 0;
46 }
47
```

Anexo 2: Códigos de programas paralelos

2.1 Algoritmo casillero-shell (C++ con OpenMP)

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include <time.h>
4   #include <vector>
5   #include <assert.h>
6   #include <omp.h>
7   #include <string>
8   #include <iostream>
9
10  #define N 1000000
11  #define R 10000
12  #define B 1000
13  #define H 8
14  #define P 10
15
16  int * arreglo(int tam, int rango, char tipo);
17  void imprimeArr(int *a,int tam);
18  int ** conVtoA(std::vector<int> *a);
19  int * tamVec(std::vector<int> *a);
20  void shellsort(int *a,int tam);
21  int * conArr2to1(int **c, int *t);
22  void prueba(double * tiempo);
23
24  int main(){
25      double t[P][2];
26      FILE *fp;
27      fp = fopen ( "prueba_8h_1000b.txt", "w" );
28      fprintf(fp,"Hilos=%d  Tamaño=%d  Bloques=%d\n",H,N,B);
29      fprintf(fp,"Serial\t\tParalelo\n");
30      for(int i=0;i<P;i++)
31          prueba(t[i]);
32
33      for(int i=0;i<P;i++){
34          fprintf(fp,"%lf\t%lf\n",t[i][0],t[i][1]);
35      }
36
37      double p1=0,p2=0;
38      for(int i=0;i<P;i++){
39          p1+=t[i][0];
40          p2+=t[i][1];
41      }
42      fprintf(fp,"Promedio:\n");
43      fprintf(fp,"%lf\t%lf\n",p1/P,p2/P);
44
45      fclose(fp);
46  }
47
```

```
48  void prueba(double * tiempo){
49      int *a,*b;
50      double st,sp;
51      std::vector<int> tc1[B],tc2[B];
52      a = arreglo(N, R, 'r');
53  
54      //Llenado de los casilleros
55      for (int i = 0; i < N; i++){
56          tc1[(int)(a[i] / (R/B))].push_back(a[i]);
57          tc2[(int)(a[i] / (R/B))].push_back(a[i]);
58      }
59  
60      int **c1=conVtoA(tc1);
61      int **c2=conVtoA(tc2);
62      int *t=tamVec(tc1);
63  
64      st=omp_get_wtime();
65      for(int i=0;i<B;i++)
66          shellsort(c2[i],t[i]);
67  
68      a=conArr2to1(c2,t);
69      sp=omp_get_wtime();
70  
71      tiempo[0]=sp-st;
72  
73      st=omp_get_wtime();
74      omp_set_num_threads(H);
75      #pragma omp parallel
76      {
77      #pragma omp for
78      for(int i=0;i<B;i++)
79          shellsort(c1[i],t[i]);
80      }
81      b=conArr2to1(c1,t);
82      sp=omp_get_wtime();
83  
84      tiempo[1]=sp-st;
85  
86      for(int i=0;i<N;i++){
87          assert(a[i]==b[i]);
88      }
89  }
90  
91  //Convierte una matriz (doble apuntador) a un arreglo (un apuntador)
92  int * conArr2to1(int **c, int *t){
93      int *tmp = new int[N];
94      int ind = 0;
95      for (int i = 0; i < B; i++){
96          for (int j = 0; j < t[i]; j++){
97              tmp[ind + j] = c[i][j];
98          }
99          ind += t[i];
100     }
101     return tmp;
102 }
103 
```

```
104  //Generador de arreglos aleatorios, 1 o 0
105  int * arreglo(int tam, int rango, char tipo){
106      int *tmp,i;
107      tmp = new int[tam];
108
109      if (tipo == 'z'){
110          for (i = 0; i < tam; i++){
111              tmp[i] = 0;
112          }
113      }
114      else if (tipo == 'o'){
115          for (i = 0; i < tam; i++){
116              tmp[i] = 1;
117          }
118      }
119      else if (tipo == 'r'){
120          srand(time(NULL));
121          for (i = 0; i < tam; i++){
122              tmp[i] = rand()%rango;
123          }
124      }
125      else{
126          printf("Error en el tipo");
127      }
128      return tmp;
129
130  }
131
132  //Imprime un arreglo
133  void imprimeArr(int *a,int tam){
134      int i;
135      for (i = 0; i < tam; i++){
136          printf("%i ", a[i]);
137      }
138      printf("\n");
139  }
140
141  //Convierte un arreglo de vectores a una matriz
142  int ** conVtoA(std::vector<int> *a){
143      int **tmp = new int*[B];
144      for (int i = 0; i < B; i++){
145          tmp[i] = new int[a[i].size()];
146          std::copy(a[i].begin(),a[i].end(),tmp[i]);
147      }
148      return tmp;
149  }
150
151  //Obtiene el tamaño de cada vector en un arreglo de vectores
152  int * tamVec(std::vector<int> *a){
153      int *tmp = new int[B];
154      for (int i = 0; i < B; i++){
155          tmp[i] = a[i].size();
156      }
157      return tmp;
158  }
159
```

```
160   //Shellsort serial
161   void shellsort(int *a,int tam){
162       for (int k = tam / 2; k > 0; k /= 2){
163           for (int i = k; i < tam; i++){
164               for (int j = i - k; j >= 0 && a[j] > a[j + k]; j -= k) {
165                   int temp = a[j];
166                   a[j] = a[j + k];
167                   a[j + k] = temp;
168               }
169           }
170       }
171   }
172
```

2.2 Algoritmo Rabin-Karp (CUDA C/C++)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <ctype.h>
4   #include "GpuTimer.h"
5
6   #define d 50 //cardinalidad del conjunto
7   #define q 33554393 //número primo grande que cabe en 32 bits
8   #define tam_ocurr 1024
9
10  //Función que calcula el valor Hash de las cadenas
11  __host__ __device__ int valor(char *t, int m ){
12      int i, v = 0;
13      for( i = 0; i < m; i++ )
14          v = ( d*v + t[i] );
15      return v;
16  }
17
18  //Función kernel, ejecutada de forma secuencial por cada hilo
19  __global__ void kernel_RK( char *d_T, char *d_P, int *d_n, int *d_m,
20                          int *d_ocurrencia, int *d_p, int *d_i){
21      //Se obitne el identificador del hilo
22      int tid = blockIdx.x * blockDim.x + threadIdx.x;
23      //Se calcula el límite inferior de la búsqueda del hilo
24      int inf = *d_n * tid / (gridDim.x * blockDim.x) - 1;
25      //Se calcula el límite superior de la búsqueda del hilo
26      int sup = *d_n * (tid + 1) / (gridDim.x * blockDim.x) + 1;
27      //Se obtiene el valor Hash de la primer subcadena del texto
28      int t = valor( &d_T[inf], *d_m ) % q;
29
30      //Cada hilo busca en un intervalo diferente
31      while( inf < (sup + *d_m)){
32          //Si se encuentra el mismo valor Hash
33          if( t == *d_p && *d_i < tam_ocurr){
34              //Operaciones atómicas para marcar la ocurrencia
35              atomicExch( &d_ocurrencia[ *d_i ], inf );
36              atomicAdd( d_i, 1 );
37          }
38          //Se avanza a la siguiente subcadena
39          inf++;
40          //Se calcula el valor Hash de la siguiente subcadena
41          t = valor( &d_T[inf], *d_m ) % q;
42      }
43  }
44
```

```
45  void dispositivo( char *T, char *P, int *ocurrencia, int *n, int *m, int *p,
46                    int *i, int BLOQUES, int HILOS){
47      //Apuntadores en el dispositivo
48      char *d_T, *d_P;
49      int *d_n, *d_m, *d_ocurrencia, *d_p, *d_i;
50
51      // Asignamos memoria en el GPU
52      cudaMalloc( (void**)&d_T, *n * sizeof(char) );
53      cudaMalloc( (void**)&d_P, *m * sizeof(char) );
54      cudaMalloc( (void**)&d_n, sizeof(int) );
55      cudaMalloc( (void**)&d_m, sizeof(int) );
56      cudaMalloc( (void**)&d_p, sizeof(int) );
57      cudaMalloc( (void**)&d_ocurrencia, tam_ocurr * sizeof(int) );
58      cudaMalloc( (void**)&d_i, sizeof(int) );
59
60      // Copia memoria del CPU al GPU
61      cudaMemcpy( d_T, T, *n * sizeof(char), cudaMemcpyHostToDevice );
62      cudaMemcpy( d_P, P, *m * sizeof(char), cudaMemcpyHostToDevice );
63      cudaMemcpy( d_n, n, sizeof(int), cudaMemcpyHostToDevice );
64      cudaMemcpy( d_m, m, sizeof(int), cudaMemcpyHostToDevice );
65      cudaMemcpy( d_p, p, sizeof(int), cudaMemcpyHostToDevice );
66      cudaMemcpy( d_ocurrencia, ocurrencia, tam_ocurr*sizeof(int), cudaMemcpyHostToDevice );
67      cudaMemcpy( d_i, i, sizeof(int), cudaMemcpyHostToDevice );
68
69      //Se inicializa el reloj
70      GpuTimer timer;
71      timer.Start();
72      //Llamada al kernel, con la configuración ingresada en consola
73      kernel_RK<<< BLOQUES, HILOS >>>( d_T, d_P, d_n, d_m, d_ocurrencia, d_p, d_i );
74      //Se detiene el reloj
75      timer.Stop();
76      printf( "Tiempo de ejecucion:  %f ms\n", timer.Elapsed() );
77
78      //Copia del arreglo de ocurrencias GPU al CPU
79      cudaMemcpy( ocurrencia, d_ocurrencia, tam_ocurr*sizeof(int), cudaMemcpyDeviceToHost );
80
81      // Liberamos memoria del GPU
82      cudaFree( d_T );
83      cudaFree( d_P );
84      cudaFree( d_n );
85      cudaFree( d_p );
86      cudaFree( d_m );
87      cudaFree( d_i );
88      cudaFree( d_ocurrencia );
89  }
90
91  //Función que convierte a minúsculas
92  void normalizar( char *T){
93      for( ; *T; ++T)
94          *T = tolower(*T);
95  }
96
```

.

```
97   //Se limpian el arreglo ocurrencia y se asignan valores
98   void inicializar( int *ocurrencia, char *T, char *P, int *n, int *m, int *p){
99       int i;
100      for(i = 0; i < tam_ocurr; i++)
101          ocurrencia[i] = -1;
102      *n = strlen(T);
103      *m = strlen(P);
104      *p = valor(P, *m) % q;
105  }
106
107  int main(int argc, char *argv[]){
108      FILE *texto;
109      char *T0 = (char *)malloc(4000000 * sizeof(char));
110      char T1[100000]; //Texto total y texto relativo
111      char P[50]; //cadena a buscar
112      int ocurrencia[tam_ocurr], n, m, p;
113      int i = 0, j = 0;
114
115      //Se lee el texto
116      texto = fopen("texto", "r");
117      while( !feof(texto) ){
118          fgets(T1, 100000, texto);
119          if( !feof(texto) )
120              strcat(T0, T1); //concatenar todos los párrafos en T0
121      }
122      fclose(texto);
123      normalizar(T0);
124
125      //Se puede introducir la cadena a buscar desde consola
126      if(argc != 4 ){
127          printf("Introduzca cadena(s) a buscar:   ");
128          gets(P);
129      }
130      else{
131          strcpy(P, argv[3]);
132          printf("Cadena a buscar:  %s\n", P);
133      }
134
135      //La configuracón de ejecución se especifica en consola
136      printf("Configuracion de ejecucion: Bloque(s): %d  Hilo(s): %d\n",
137              atoi(argv[1]), atoi(argv[2]));
138      inicializar(ocurrencia, T0, P, &n, &m, &p);
139      dispositivo(T0, P, ocurrencia, &n, &m, &p, &j, atoi(argv[1]), atoi(argv[2]) ) ;
140
141      //Se imprime el lugar dentro del texto donde hubo ocurrencias.
142      if(ocurrencia[0] != -1){
143          printf("Ocurrencias:\n");
144          while( i < tam_ocurr ){
145              if( ocurrencia[i] != -1)
146                  printf("%d\t\t", ocurrencia[i]);
147              i++;
148          }
149      }
150      else
151          printf("No hubo ocurrencias\n");
152      return 0;
153  }
154
```

2.3 Algoritmo genético (C con OpenMP)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4   #include <omp.h>
5
6   #define POBLACION 256
7   #define ATTRIBS 16
8   #define GENERACIONES 20000
9   #define HILOS 8
10
11  //Función para imprimir
12  void imprimir( char *str, char b[][ATTRIBS], int k, int l ){
13      int i, j;
14      printf("%s%d  ", str, k);
15      for( i = 0; i < POBLACION/l; i++){
16          printf(" [");
17          for( j = 0; j < ATTRIBS; j++){
18              if( j == ATTRIBS-1 )
19                  printf(" %d]", b[i][j]);
20              else
21                  printf(" %d,", b[i][j]);
22          }
23      }
24      printf("\n\n");
25  }
26
27  //Función que convierte los valores binarios a decimal
28  int binToDec( char *b ){
29      int i, j = 1;
30      long k = 0;
31      for( i = ATTRIBS-1; i >= 0; i--, j*=2 )
32          k += j * b[i];
33      return k;
34  }
35
36  //Función de la cual se busca el máximo relativo
37  long long fun( long a ){
38      return (long long) -a*a*a*a + 50000*a*a*a + 20000*a*a - 18*a + 5;
39  }
40
41  int main(){
42      char b[POBLACION][ATTRIBS], ganadores[POBLACION/2][ATTRIBS], temp[POBLACION][ATTRIBS];
43      int i, j, k, l, n, gen;
44      time_t t;
45      double start, stop;
46
47      srand( (unsigned) time( &t ) );
48
49      //Cantidad de hilos e iniciamos reloj
50      omp_set_num_threads( HILOS );
51      start = omp_get_wtime();
52
53      //Se crea la poblacion inicial
54      for( i = 0; i < POBLACION; i++)
55          for( j = 0; j < ATTRIBS; j++)
56              b[i][j] = rand() % 2;
57
```

```
58      //Se producen nuevas generaciones
59      for( i = 0; i < GENERACIONES; i++){
60          //impresiòn de población actual (opcional)
61          //imprimir( "Poblacion", b, i, 1);
62          //Se seleccionan los ganadores
63          #pragma omp parallel for firstprivate(b) lastprivate(b) private(k)
64          for( j = 0; j < POBLACION; j++ ){
65              if( fun( binToDec( b[2*j] ) ) > fun( binToDec( b[2*j+1] ) ) )
66                  for( k = 0; k < ATTRIBS; k++)
67                      ganadores[j][k] = b[2*j][k];
68              else
69                  for( k = 0; k < ATTRIBS; k++)
70                      ganadores[j][k] = b[2*j+1][k];
71          }
72          //impresiòn de ganadores de la generación actual (opcional)
73          //imprimir("Ganadores", ganadores, i, 2);
74
75          //Combinaciones de ganadores para una nueva generacion
76          #pragma omp parallel for firstprivate(ganadores) lastprivate(temp) private(gen, k, l)
77          for( n = 0; n < POBLACION/2; n++ ){
78              //Seleccionamos un ganador diferente al n
79              do{
80                  k = rand() % (POBLACION/2);
81              }while( k == n );
82              //Combinacion entre los ganadores n y k
83              gen = rand() % (ATTRIBS-2) + 1;
84              for( l = 0; l < gen; l++ )
85                  temp[2*n][l] = ganadores[n][l];
86              for( ; l < ATTRIBS; l++ )
87                  temp[2*n][l] = ganadores[k][l];
88
89              gen = rand() % (ATTRIBS-2) + 1;
90              for( l = 0; l < gen; l++ )
91                  temp[2*n+1][l] = ganadores[k][l];
92              for( ; l < ATTRIBS; l++ )
93                  temp[2*n+1][l] = ganadores[n][l];
94          }
95
96          //Reemplazar la poblacion anterior con la nueva
97          for( k = 0; k < POBLACION; k++)
98              for( l = 0; l < ATTRIBS; l++)
99                  b[k][l] = temp[k][l];
100     }
101
102     //Imprimir la solución y detener el reloj
103     imprimir("Ganadores", ganadores, i, 2);
104     stop = omp_get_wtime();
105     printf("Tiempo paralelo: %f\n", stop - start);
106
107     return 0;
108 }
109
```

2.4 Algoritmo Criba de Eratóstenes (CUDA C/C++)

```
1   #include <stdio.h>
2   #include <assert.h>
3   #include "Error.h"
4   #include "GpuTimer.h"
5   #include "CpuTimer.h"
6
7   #define MARK 1
8   #define UNMARK 0
9   #define N 100000000
10  #define BLOQUES 64
11  #define HILOS 512
12
13  const int BYTES = N * sizeof(int);
14
15  //Kernel ejecutado secuencialmente por cada hilo
16  __global__ void kernelCriba(int k, int *d_a){
17      //Se obtiene el identificador del hilo
18      int i = threadIdx.x + blockIdx.x*blockDim.x;
19
20      //Si el elemento está desmarcado...
21      if( d_a[k] == UNMARK ){
22          //... se marcan sus múltiplos
23          while( i * k < N ){
24              if( i >= k )
25                  d_a[i*k] = MARK;
26              //Se avanza a otro múltiplo
27              i+=blockDim.x*gridDim.x;
28          }
29      }
30  }
31
32  void device( int *h_a ){
33      //Apuntador al GPU
34      int *d_a;
35      int k;
36
37      //Se asigna y copia memoria del CPU al GPU
38      HANDLER_ERROR_ERR( cudaMalloc( &d_a, BYTES ) );
39      HANDLER_ERROR_ERR( cudaMemcpy( d_a, h_a, BYTES, cudaMemcpyHostToDevice));
40
41      //Se inicializa el reloj
42      GpuTimer timer;
43      timer.Start();
44
45      //Se itera desde el 2 hasta la raíz de N; se llama al kernel
46      for( k = 2; k*k <= N; k++)
47          kernelCriba<<<BLOQUES, HILOS >>>(k, d_a);
48      //Se detiene el reloj
49      timer.Stop();
50      printf("Tiempo GPU: %f ms\n", timer.Elapsed());
51
52      //Se copia el arreglo resultado y se libera memoria
53      HANDLER_ERROR_ERR( cudaMemcpy( h_a, d_a, BYTES, cudaMemcpyDeviceToHost));
54      HANDLER_ERROR_ERR( cudaFree( d_a ) );
55  }
56
```

```
57  void prueba( int *h_a ){
58      //Arreglo auxiliar para realizar la prueba secuencial
59      int *h_prueba = (int*)malloc( BYTES );
60      int i, k;
61
62      //Se inicializan a 0 todos los elementos
63      for( i = 0; i < N; i++)
64          h_prueba[i] = UNMARK;
65
66      //Se inicializa el reloj
67      CpuTimer timer;
68      timer.Start();
69
70      //se itera desde 2 hasta la raíz de N
71      for( k = 2; k*k <= N; k++ ){
72          //Si el elemento está desmarcado...
73          if( h_prueba[k] == UNMARK ){
74              i = k;
75              //... se marcan sus múltiplos
76              while( i * k < N){
77                  h_prueba[i*k] = MARK;
78                  i++;
79              }
80          }
81      }
82
83      //Se detiene el reloj
84      timer.Stop();
85      printf("Tiempo CPU: %f ms\n", timer.Elapsed());
86
87      //Se comparan los resultados e imprimen (opcional)
88      for( i = 2; i < N; i++ ){
89          assert( h_a[i] == h_prueba[i] );
90          //if( h_a[i] == UNMARK )
91              //printf("%d \n", i);
92      }
93      printf("Ejecucion Correcta :D\n");
94  }
95
96  int main(){
97      //Arreglo para marcar los primos
98      int *h_a = (int*)malloc( BYTES );
99      int i;
100
101     //Se inicializan a 0 todos los elementos
102     for( i = 0; i < N; i++)
103         h_a[i] = UNMARK;
104
105     device( h_a );
106     prueba( h_a );
107     free( h_a );
108
109     return 0;
110 }
111
```