Centro de Estudios de Postgrado

Máster en Profesorado de Enseñanza Secundaria Obligatoria, Bachillerato, Formación Profesional y Enseñanza de Idiomas

UNIVERSIDAD DE JAÉN
Centro de Estudios de Postgrado

Trabajo Fin de Máster

PROGRAMACIÓN Y ARQUITECTURAS PARALELAS

Alumno/a: **Pamos Ureña, Miguel Ángel**

Tutor/a: Prof. D. Rafael J. Segura Sánchez
Dpto: Departamento de Informática

Junio, 2019

Contenido

Índice de figuras 4
1. Resumen 6
1.1. Resumen y palabras clave 6
1.2. Abstract and keywords 6
2. Introducción 7
3. Fundamentación epistemológica 8
3.1. Contextualización de la computación paralela. 8
3.1.1. Arquitectura Von Neumann 9
3.1.2. Taxonomía de Flynn 11
3.1.3. Clasificación Wolfgang Handler 14
3.2. Arquitecturas paralelas 15
3.2.1. Paralelismo en procesadores 18
3.2.1.1. Paralelismo a nivel de instrucción 18
3.2.1.2. Paralelismo a nivel de threads 22
3.2.2. Multiprocesadores y multicomputadores 26
3.2.2.1. Multiprocesadores 27
3.2.2.2. Multicomputadores. Sistemas distribuidos 29
3.3. Programación paralela y distribuida 31
3.3.1. Lenguajes para la programación paralela 32
3.3.1.1. Fortran 32
3.3.1.2. ALGOL 33
3.3.1.3. ADA 34
3.3.1.4. Haskell 36
3.3.1.5. Lenguajes de programación modernos 37
3.3.2. Herramientas de programación paralela 44
3.3.2.1. POSIX Threads. 44
3.3.2.2. MPI 46
3.3.2.3. OpenMP 48

3.3.2.4. CUDA 49
3.3.2.5. OpenCL 53
3.3.3. Algoritmos de programación paralela. Patrones de diseño 55
3.3.3.1. Task Parallelism 55
3.3.3.2. Data Parallelism 56
3.3.3.3. Recursive Parallelism 57
3.3.3.4. Pipeline 59
3.3.3.5. Estructuras soporte 59
3.3.4. Problemática en la concurrencia 60
3.3.4.1. Race Condition 60
3.3.4.2. Deadlock 62
3.3.4.3. Starvation y Livelock 62
3.4. Reflexión sobre el futuro de la programación paralela. 63
4. Proyección didáctica 64
4.1. Título 64
4.2. Justificación 64
4.3. Contextualización curricular. 64
4.4. Contexto del centro y alumnado 64
4.5. Marco legislativo 65
4.6. Competencia general 65
4.7. Orientaciones pedagógicas 66
4.8. Competencias profesionales, personales y sociales 66
4.9. Líneas de actuación 66
4.10. Objetivos Generales 66
4.11. Resultados de aprendizaje 66
4.12. Objetivos didácticos 67
4.13. Temas transversales 67
4.14. Contenidos Conceptuales 67
4.15. Metodología 67
4.15.1. Organización del aula 67

4.15.2. Agrupamientos 68
4.15.3. Actividades de enseñanza-aprendizaje 68
4.15.4. Materiales y recursos 68
4.15.5. Proceso de enseñanza-aprendizaje 69
4.15.6. Cronograma 71
4.16. Evaluación y Recuperación 74
4.16.1. Criterios de evaluación 74
4.16.2. Procedimientos. 74
4.16.3. Instrumentos de evaluación 74
4.16.4. Criterios de Calificación 76
4.16.5. Criterios de recuperación. 78
5. Bibliografía 79

Índice de figuras

Figura 1. Arquitectura Von Neumann 9
Figura 2. Clasificación Flynn 11
Figura 3. SISD 12
Figura 4. MISD 13
Figura 5. SIMD 13
Figura 6. MIMD 14
Figura 7. La ejecución paralela es un caso especial de la ejecución concurrente. 16
Figura 8. Ejecución concurrente, no paralela 17
Figura 9. Ejecución concurrente y paralela 17
Figura 10. Organización de arquitecturas de computadores 18
Figura 11. Ejecución secuencial y ejecución segmentada 19
Figura 12. Procesamiento superescalar de 2 vías 21
Figura 13. Esquema de instrucción VLIW 22
Figura 14. Ejecución multihilo de grano fino 23
Figura 15. Ejecución multihilo de grano grueso 24
Figura 16. Ejecución simultánea con multihilo 25
Figura 17. Instrucciones vectoriales 26
Figura 18. Clasificación Tanenbaum 27
Figura 19. Modelo UMA de un multiprocesador 28
Figura 20. Modelo NUMA de un multiprocesador 29
Figura 21. Multicomputadores 30
Figura 22. Código en ADA que describe dos tareas que se ejecutan concurrentemente. 35
Figura 23. Código en ADA con entradas 35
Figura 24. Explicación del algoritmo Fork/Join basado en divide y vencerás 37
Figura 25. Fork/join en Java. 38
Figura 26. Evolución de la librería STL 42
Figura 27.. Esquema funcionamiento de una transacción 44
Figura 28. Tipos de Pthreads 46

Figura 29. Sintaxis directivas compilador en OpenMP 48
Figura 30. Variables de entorno en OpenMP 49
Figura 31. Rendimiento máximo de CPU y GPU en GigaFlops..... 50
Figura 32. Flujo de procesamiento en CUDA. 51
Figura 33. Grid de bloques de hilos. 52
Figura 34. Ejecución secuencial y declaración de función kernel en CUDA 52
Figura 35. Esquema OpenCL 54
Figura 36. Esquema modelo de ejecución. 54
Figura 37. Comparativa de resultados estimados para la resolución del problema de multiplicación de matrices. 57
Figura 38. Esquema pipeline 59
Figura 39. Algoritmos de soporte y estrategias de paralelismo. 60
Figura 40. Ejemplo de race condition 61
Figura 41. Ejemplo de deadlock. 62

1. Resumen

1.1. Resumen y palabras clave

Este documento desarrolla el Trabajo Fin de Máster dentro del Máster en Profesorado de Educación Secundaria Obligatoria, Bachillerato, Formación Profesional y Enseñanza de idiomas en la especialidad de Informática. El tema a abordar es la programación paralela, así como las arquitecturas que dan soporte a ese paradigma de programación.

En la primera parte del documento se realiza un estudio epistemológico del tema, en el que se profundiza en el tema del trabajo, desde los conceptos más básicos a las tecnologías que se utilizan hoy día. Se realiza un recorrido por arquitecturas, lenguajes, patrones de diseño y problemática en la programación concurrente.

En una segunda parte, se lleva ese estudio teórico del tema a la práctica docente. Se desarrolla una unidad didáctica englobada en el módulo *Programación de Servicios y Procesos*, del *Ciclo Formativo de Grado Superior en Desarrollo de Aplicaciones Multiplataforma.* Se contextualiza el centro, se encuadra en el marco normativo y se establecen los resultados de aprendizaje y criterios de evaluación, y, por último, se desarrolla la metodología docente y la planificación temporal.

Palabras clave: docencia, procesos, servicios, programación, paralelismo, arquitecturas.

1.2. Abstract and keywords

This work contains the Final Master Project within Teaching of Secondary Education, A levels (sixth form), Professional Training (alternative sixth form) and Language Teaching. It is focused on parallel programming, as well as the architectures that support this programming paradigm.

In the first part of the document, an epistemological study is addressed, focused on the different concepts about parallelism, from architectures to languages and patterns design.

In the second one, this theoretical study is applied to teaching. It is developed a teaching unit included in the Process and Services Programming module of the “Ciclo Formativo de Grado Superior” (second level of alternative sixth form) of Development of Multiplatform Applications. It is based on current legal framework. School and students are contextualized. In this part, the methodological and concepts addressed in previous theoretical part are applied.

Keywords: teaching, threading, parallel computation, parallelism, computer architectures.

2. Introducción

Este documento desarrolla el Trabajo Fin de Máster dentro del Máster en Profesorado de Educación Secundaria Obligatoria, Bachillerato, Formación Profesional y Enseñanza de idiomas.

Se divide en dos grandes apartados, la fundamentación epistemológica del tema y la unidad didáctica desarrollada, enmarcada en una asignatura del Ciclo Formativo de Grado Superior en Desarrollo de Aplicaciones Multiplataforma.

En la primera parte, se realiza un recorrido sobre las distintas arquitecturas que existen hoy en día, así como la evolución de las mismas. Además, se desarrollan los distintos tipos de paralelismo que conocemos. Una vez que conocemos las arquitecturas, se detallan los elementos que nos permiten aprovecharlas, desde lenguajes de programación, librerías, APIs, hasta patrones de diseño para implementar software paralelo de forma eficiente y sin fallos.

En la segunda parte del documento se traslada todo ese estudio realizado al entorno docente, en el que se desarrolla la propia unidad didáctica. En esta unidad, además de contextualizarla tanto social, como normativamente, se desarrolla una metodología de aprendizaje para que el alumno adquiera los conocimientos deseados sobre la temática trabajada, detallando, entre otros, los objetivos, el contenido, la planificación y la evaluación de la misma.

3. Fundamentación epistemológica

A lo largo de este apartado se va a profundizar en los conceptos relacionados con la programación paralela y distribuida, las arquitecturas conectadas a esa tipología de programación y patrones de diseño. Se va a realizar un análisis de los inicios de estos paradigmas, un estudio actual y una mirada al futuro de estos conceptos

3.1. Contextualización de la computación paralela.

La principal razón del estudio de la computación paralela es que el mundo que conocemos es un mundo paralelo. Y entiéndase paralelo no como una dimensión desconocida del mundo real, sino como una sucesión de eventos complejos e interrelacionados que ocurren al mismo tiempo, pero dentro de una secuencia temporal. Si comparamos la computación tradicional o en serie con la computación en paralelo, ésta última es mucho más adecuada para reflejar, simular, modelar y comprender los sucesos o fenómenos más complejos del mundo real, tal y como nos sugiere Barney (2019). Estos fenómenos, que el ser humano intenta modelar, son situaciones reales de la vida cotidiana de cada uno de los seres humanos. Imaginemos un atasco en hora punta, la predicción del tiempo en una ciudad, sacar dinero de un cajero, una cadena de montaje de una fábrica, como otros tantos sucesos que nos rodean, no dejan de ser eventos que se ajustan al paralelismo del que estamos hablando.

La computación paralela ha sido el gran acontecimiento en el mundo de la computación en los últimos años, según Voevodin (2018). Hoy en día, cualquier dispositivo que conozcamos: un móvil, una tablet, un ordenador personal, servidores, supercomputadores, clústeres, etc., permiten la ejecución de forma paralela. Esta ejecución en forma paralela no es más que la existencia de diferentes nodos de computación. Estos nodos de computación consisten en varios procesadores, éstos procesadores a su vez tienen varios núcleos, que a su vez tienen varias unidades funcionales independientes. Todo este hardware puede trabajar en paralelo, siempre que ejecuten software que se ajuste a ese paralelismo, con algoritmos específicos para ello.

A continuación, se detallan una serie de arquitecturas y clasificaciones de las mismas, según distintos parámetros: la arquitectura estándar, instrucciones que pueden cargar o datos que pueden leer. Para ello, primeramente, se describe la arquitectura Von Neumann, que supuso un vuelco a la arquitectura de las primeras máquinas que procesaban programas fijos (los que necesitan de una reestructuración de cableado y diseño para cambiar el programa). Esta nueva arquitectura, como se indica a continuación, se basa en el programa almacenado, que posee un conjunto de instrucciones que pueden ser almacenados en memoria. Posteriormente, se muestra al lector la clasificación, denominada taxonomía de Flynn, de las arquitecturas según las

instrucciones y datos que son capaces de procesar un ordenador. Por último, se introducirá la clasificación de los computadores según una notación específica que relaciona los distintos elementos de una CPU.

3.1.1. Arquitectura Von Neumann

La arquitectura de Von Neumann fue publicada por primera vez por John von Neumann en 1945 (Neumann, 1945). Su diseño de arquitectura de computadora consiste en una unidad de control (UC), unidad aritmética y lógica (ALU), unidad de memoria, registros y entradas / salidas. La arquitectura de Von Neumann se basa en el concepto de computadora de programa almacenado, donde los datos de instrucción y de programa se almacenan en la misma memoria. Este diseño todavía se utiliza en la mayoría de las computadoras producidas en la actualidad.

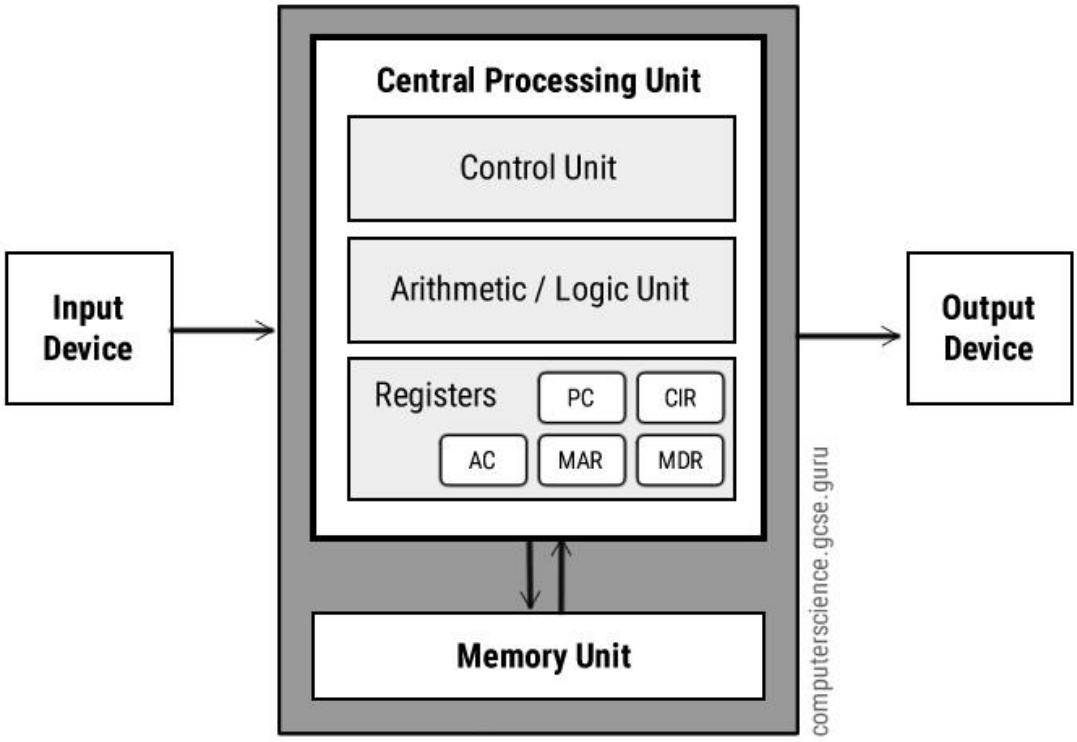

Figura 1. Arquitectura Von Neumann[1]

Unidad Central de Procesamiento (CPU)

La Unidad Central de Procesamiento (CPU) es el circuito electrónico responsable de ejecutar las instrucciones de un programa de ordenador. Normalmente, a esta unidad central de procesamiento se le conoce como procesador. Dentro de esta unidad de procesamiento se encuentra la ALU y la UC, además de una variedad de registros.

[1] https://www.computerscience.gcse.guru/theory/von-neumann-architecture

Registros

Los registros son áreas de almacenamiento de alta velocidad en la CPU. Todos los datos deben almacenarse en un registro antes de que puedan procesarse.

MAR: Registro de direcciones de memoria. Contiene la ubicación de la memoria de los datos a los que se debe acceder

MDR: Registro de datos de memoria. Guarda los datos que se transfieren a la memoria o desde ella

AC: Acumulador. Es el lugar donde se almacenan resultados aritméticos y lógicos intermedios

PC: Contador de programas. Contiene la dirección de la siguiente instrucción a ejecutar

CIR Registro de instrucciones actuales Contiene las instrucciones actuales durante el procesamiento.

Unidad aritmética y lógica (ALU)

La ALU permite que se realicen operaciones aritméticas (sumar, restar, etc.) y lógicas (AND, OR, NOT, etc.).

Unidad de Control (UC)

La unidad de control controla el funcionamiento de la ALU, la memoria y los dispositivos de entrada / salida de la computadora, diciéndoles cómo responder a las instrucciones del programa que acaba de leer e interpretar desde la unidad de memoria. La unidad de control también proporciona las señales de tiempo y control requeridas por otros componentes de la computadora.

Buses

Los buses son los medios por los cuales los datos se transmiten de una parte de una computadora a otra, conectando todos los componentes internos principales a la CPU y la memoria. Un bus de sistema de CPU estándar se compone de un bus de control, bus de datos y bus de dirección.

Bus de direcciones: Transporta las direcciones de datos (pero no los datos) entre el procesador y la memoria.

Bus de datos: Transporta datos entre el procesador, la unidad de memoria y los dispositivos de entrada / salida.

Bus de control: Transporta señales / comandos de control desde la CPU (y señales de estado desde otros dispositivos) para controlar y coordinar todas las actividades dentro de la computadora.

Unidad de memoria

La unidad de memoria consta de RAM, a veces denominada memoria principal o principal. A diferencia de un disco duro (memoria secundaria), esta memoria es rápida y también es accesible directamente por la CPU. RAM se divide en particiones. Cada partición consta de una dirección y su contenido (ambos en forma binaria). La dirección identificará de forma única cada ubicación en la memoria. La carga de datos de la memoria permanente (disco duro), en la memoria temporal (RAM) más rápida y de acceso directo, permite que la CPU funcione mucho más rápido.

3.1.2. Taxonomía de Flynn

Si deseamos clasificar los ordenadores según su arquitectura, probablemente la clasificación de Flynn es la más conocida en el mundo de la computación. Para comprender esta clasificación, primero deberemos entender qué es un flujo de instrucciones y un flujo de datos. Un flujo de instrucciones no es más que un conjunto de instrucciones de forma secuencial que se ejecutan en un único procesador. Por otra parte, un flujo de datos son aquellos que son utilizados por las instrucciones.

Según Flynn(1972), categorizar una arquitectura se basa en el número de instrucciones y datos que puede procesar de forma concurrente.

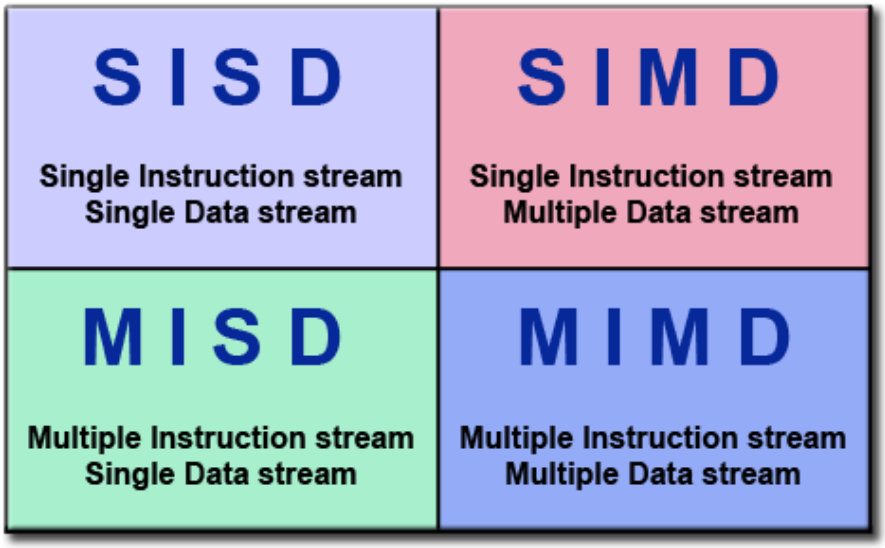

Figura 2. Clasificación Flynn[2]

- SISD (Single Instruction Single Data: Una instrucción-Un dato): en esta categoría se tiene un único flujo de instrucciones y un único flujo de datos. Esto no sería más que una arquitectura en serie, donde en un momento dado, únicamente se está ejecutando una única instrucción. A este tipo de computadores se le conoce como escalares, y siguen la arquitectura definida por Von Neumann. Este tipo de máquinas consta de contador de programa, que es un registro que gestiona la

[2] https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

ejecución en serie del programa. Las instrucciones se van cargando de la memoria y el contador de programa va indicando la siguiente instrucción a procesar, de forma secuencial. Hoy en día las máquinas no presentan esta arquitectura de forma pura, ya que incorporan algún tipo de paralelización, como puede ser la segmentación de instrucciones o la ejecución de dos instrucciones al mismo tiempo, en lo que llamaríamos una arquitectura superescalar.

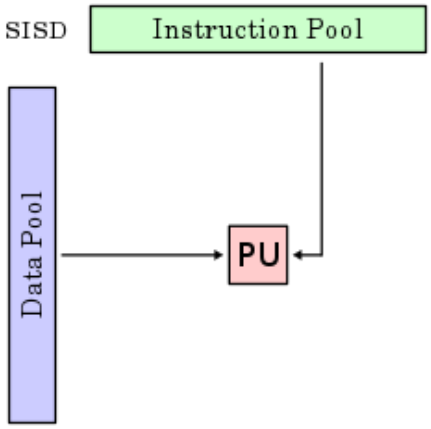

Figura 3. SISD[3]

- MISD (Multiple Instruction Single Data: Multiple instrucciones-Un dato): esta clasificación agrupa aquellos computadores que procesan un flujo múltiple de instrucciones y un único flujo de datos. Dicho de otra forma, existen varias instrucciones que simultáneamente procesan el mismo y único espacio de datos. Este tipo de es muy común, debido a que no es efectivo tener múltiples instrucciones para acceder a un solo flujo de datos. Existen un tipo de máquinas que realizan un procesamiento vectorial a través de una secuencia de etapas, en la que en cada una presenta un resultado intermedio. Este tipo de máquinas se denominan de arquitecturas segmentadas. Se clasifican dentro de esta categoría MISD porque los elementos del vector se pueden considerar que pertenecen a un mismo flujo de datos, y cada una de las etapas, que hemos indicado que producen el resultado intermedio, representan las múltiples instrucciones que se aplican al flujo de datos.

[3] https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

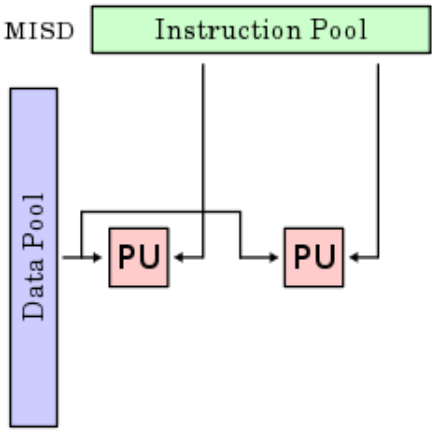

Figura 4. MISD[4]

- SIMD (Single Instruction Multiple Data: Una instrucción-Multiples datos): esta categoría consta de un único flujo de instrucción simple y un flujo de datos múltiple. De este modo, sobre distintos datos se está ejecutando la misma instrucción al mismo tiempo. Así, la unidad de control invoca varias unidades de procesado. Como se ha comentado anteriormente en la clasificación MISD, este tipo de máquinas también soporta el procesamiento vectorial, de forma que se asigna cada elemento del vector a una unidad de procesado diferente, para el procesamiento concurrente.

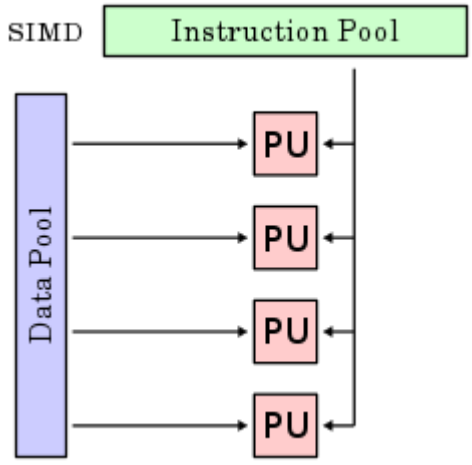

Figura 5. SIMD[5]

- MIMD (Multiple Instruction Multiple Data: Múltiples instrucciones-Múltiples datos): cómo podemos imaginar en este punto, este acrónimo indica la existencia de un flujo de instrucciones múltiple y un flujo de datos múltiple. Las máquinas clasificadas en esta categoría presentan varias unidades de procesamiento, en las cuales se pueden realizar múltiples instrucciones de forma

[4] https://computing.llnl.gov/tutorials/parallel_comp/#Whatis
[5] https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

simultánea sobre distintos datos. Esta categoría es la más compleja de todas, de igual forma que son las que presentan una mayor eficiencia en la ejecución paralela.

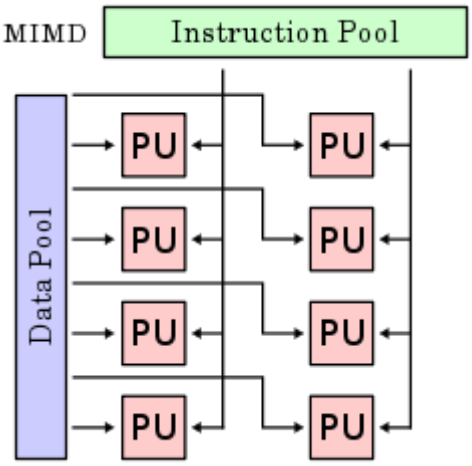

Figura 6. MIMD[6]

3.1.3. Clasificación Wolfgang Handler

En 1977, Wolfgang Handler propuso una notación elaborada para expresar el pipeline y el paralelismo de las computadoras. La clasificación de Handler se basa en tres distintos niveles:

- Unidad de control del procesador (UC)
- Unidad lógica aritmética (ALU)
- Circuito de nivel de bit (BLC).

La UC corresponde a un procesador o CPU, la ALU corresponde a una unidad funcional o un elemento de procesamiento y el BLC corresponden al circuito lógico necesario para realizar operaciones bit a bit en la ALU. Esta clasificación que propuso Handler, utiliza los siguientes tres pares de enteros para describir una computadora:

Computador = (p * p ', a * a', b * b ')

Donde p = número de UC

Donde p '= número de UC que pueden “encauzarse” (pipelining)

Donde a = número de ALU controladas por cada UC

Donde a '= número de ALU que pueden encauzarse

Donde b = número de bits en ALU o elemento de procesamiento (PE)

Donde b '= número de segmentos en todas las ALU o en un solo PE

[6] https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

Las siguientes reglas y operadores se utilizan para mostrar la relación entre varios elementos de la computadora:

- El operador '*' se usa para indicar que las unidades están canalizadas o macro-canalizadas con un flujo de datos corriendo a través de todas las unidades.
- El operador '+' se usa para indicar que las unidades no están canalizadas pero que trabajan en transmisiones independientes de datos.
- El operador 'v' se usa para indicar que el hardware de la computadora puede funcionar en uno de varios modos
- El símbolo '~' se utiliza para indicar un rango de valores para cualquiera de los parámetros.

Los procesadores periféricos se muestran ante el procesador principal utilizando otros tres pares de enteros. Si el valor del segundo elemento de cualquier par es 1, se puede omitir para brevedad.

La clasificación de Handler se explica mejor al mostrar cómo se usan las reglas y los operadores. Por ejemplo, el CDC 6600 tiene un único procesador principal compatible con 10 procesadores de E / S. Una unidad de control coordina una ALU con una longitud de palabra de 60 bits. La ALU tiene 10 unidades funcionales que se puede formar en una tubería. Los 10 procesadores periféricos de E / S pueden funcionar de forma paralela entre sí y la CPU.

La descripción de los 10 procesadores de E / S es:

CDC 6600I / O = (10, 1, 12)

La descripción para el procesador principal es:

CDC 6600principal = (1, 1 * 10, 60)

Se puede considerar que el procesador principal y los procesadores de E / S forman una macro-canalización por lo que el operador '*' se utiliza para combinar las dos estructuras:

CDC 6600 = (procesadores de E / S) * (procesador central = (10, 1, 12) * (1, 1 * 10, 60)

3.2. Arquitecturas paralelas

Una computadora paralela es un conjunto de procesadores que pueden trabajar cooperativamente para resolver un problema computacional. Esta definición es lo suficientemente amplia como para incluir supercomputadoras paralelas que tienen cientos o miles de procesadores, redes de estaciones de trabajo, estaciones de trabajo con múltiples procesadores y sistemas integrados. Las computadoras paralelas son interesantes porque ofrecen la posibilidad de concentrar los recursos computacionales,

ya sean procesadores, memoria o ancho de banda de E / S, en problemas computacionales importantes.

El paralelismo a veces se ha visto como un sub-área de computación rara y exótica, interesante, pero de poca relevancia para el programador promedio. Según Foster (1995), el paralelismo se está volviendo omnipresente, y la programación paralela se está convirtiendo en algo central para la empresa de programación. Hoy en día, esa afirmación de hace más de 20 años es más real que nunca.

Cuando se habla de paralelismo o concurrencia, el lector puede llegar a una confusión con estos términos equivalentes, que a, a menudo, la computación paralela se usa para significar los mismo que la computación concurrente, y viceversa. Y esta equivalencia errónea, es de fácil distinción. Cuando nos referimos a concurrencia se está hablando de la vista abstracta de cómo se ejecutan múltiples flujos de instrucciones a lo largo del tiempo, mientras que cuando se habla de paralelismo, nos centramos en cómo se ejecutan en relación al tiempo.

Figura 7. La ejecución paralela es un caso especial de la ejecución concurrente.[7]

Sottile et al. (2010) hace una distinción entre lo que es concurrencia y paralelismo. El autor define un programa concurrente como uno en el que hay múltiples flujos instrucciones están activas al mismo tiempo. Uno o más de los flujos están disponibles para avanzar en una sola unidad de tiempo. La clave para diferenciar el paralelismo de la concurrencia es el hecho de que, a través del tiempo compartido o la multitarea, se puede dar la ilusión de ejecución simultánea cuando, de hecho, solo un flujo de instrucciones avanza en un momento dado.

[7] Elaboración propia

De otra forma, el autor define un programa paralelo como una instancia de un programa concurrente que se ejecuta en presencia de múltiples unidades de hardware que garantizarán que dos o más flujos de instrucciones progresarán en una sola unidad de tiempo. El factor diferenciador de un programa concurrente que se ejecuta a través de la división de tiempo es que, en un programa paralelo, al menos dos flujos progresan en un momento dado. Esto es más a menudo una consecuencia directa de la presencia de múltiples unidades de hardware que pueden soportar la ejecución simultánea de diferentes flujos.

Estas definiciones se ven de forma más clara en el siguiente gráfico:

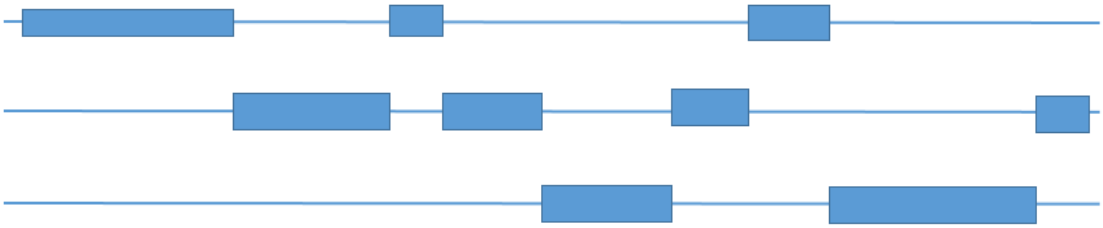

Figura 8. Ejecución concurrente, no paralela[8]

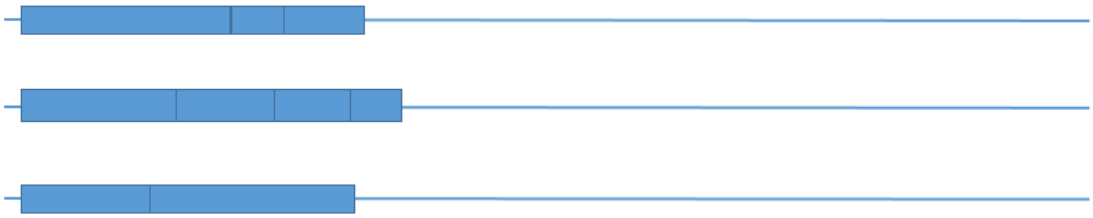

Figura 9. Ejecución concurrente y paralela[9]

Una vez que se ha definido lo que es el paralelismo computacional, se van a detallar los elementos que lo componen, desde los niveles, pasando por el hardware que lo implementa, así como los sistemas para comunicar los procesos paralelos. El siguiente esquema (Tosini, 2015), recoge de manera gráfica la organización de las arquitecturas, incluyendo los sistemas distribuidos.

[8] Sottile, M.J. et al (2010). *Introduction to Concurrency in Programming Languages.*
[9] Sottile, M.J. et al (2010). *Introduction to Concurrency in Programming Languages.*

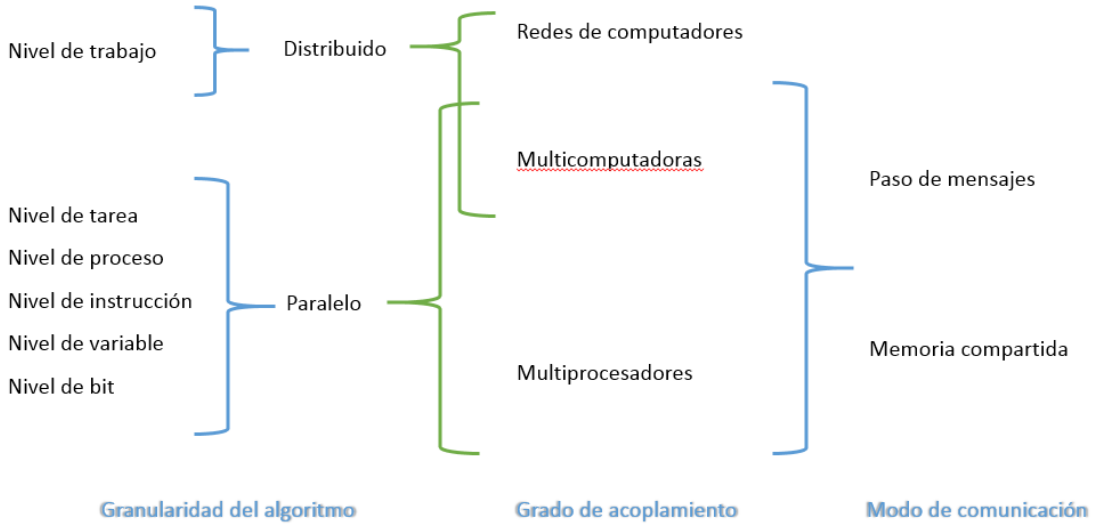

Figura 10. Organización de arquitecturas de computadores[10]

3.2.1. Paralelismo en procesadores

A nivel hardware, se pueden definir 3 niveles de paralelismo: a nivel de instrucción, a nivel de threads y a nivel de datos.

3.2.1.1. Paralelismo a nivel de instrucción

Dentro de los distintos niveles de paralelismos, se va a comentar en primer lugar el paralelismo a nivel de instrucción (ILP). Este tipo de paralelismo representa una técnica en el diseño de arquitecturas de ordenadores, especialmente centrada en procesadores y compiladores. ILP es capaz de tener un rendimiento mayor en la ejecución de un determinado código mediante las operaciones, las cuales individualiza para que se ejecuten en paralelo. Esta técnica no fue determinante hasta la década de los 80 del siglo pasado. En realidad, cuando hablamos de la ILP, significa cuántas instrucciones se pueden ejecutar o emitir a la vez. Cuando hablamos de ILP podemos distinguir tres tipos de procesadores que implementan este tipo de paralelismo: procesadores segmentados, procesadores superescalares y procesador con una palabra de instrucción muy larga, o comúnmente conocidos como VLIW (Very Long Instruction Word).

Procesadores segmentados

Cuando se habla de procesadores segmentados se está haciendo mención a la división en la ejecución de una instrucción en diferentes etapas. Cada etapa se ejecuta, de forma genérica, en un ciclo de reloj del procesador. La nomenclatura y el número de etapas pueden variar de un procesador a otro.

[10] Tosini, M. (2015). *Introducción a las Arquitecturas Paralelas.*

Al segmentar la ejecución de las instrucciones, debemos introducir el concepto de ratio de ejecución. Esta ratio no es más que la media de instrucciones que se ejecutan por cada ciclo de reloj. Con esta técnica de segmentación, se aumenta dicha ratio, gracias a que en un mismo tiempo se solapan las etapas de ejecución de más de una instrucción. Así, el paralelismo lo encontramos al poder ejecutar tantas instrucciones como número de etapas tenga la propia ejecución de las instrucciones, y después de rellenar todas las etapas con tantas instrucciones como etapas haya en la segmentación, conseguiremos finalizar una instrucción por ciclo.

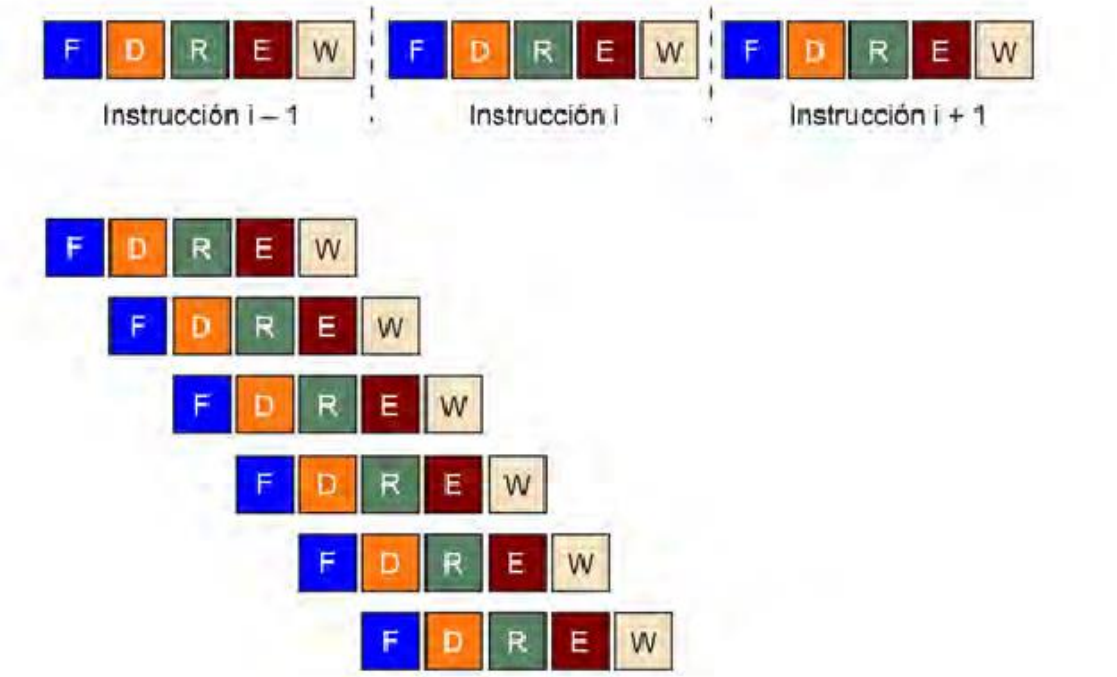

Figura 11. Ejecución secuencial y ejecución segmentada[11]

En la figura anterior se muestran las 5 fases en las que se segmenta una instrucción: F (Instruction Fetch) en la que se extrae la instrucción de la memoria principal, D (Instruction Decode), en la que se decodifica la propia instrucción y los operandos sobre la misma, R (Operand Fetch Unit), en la que se extraen los operandos necesarios para la instrucción, E (Instruction Execution Unit), en lal que se ejecutan las operaciones y W (Writeback), en la que se escriben los datos en los registros o memoria. En la imagen anterior también se muestra la ejecución en una máquina en la que no existe segmentación, por lo que, al ejecutar cada instrucción, no se ejecute la siguiente hasta que no haya terminado. Sin embargo, si se segmenta la ejecución, se puede observar que diferentes instrucciones se pueden estar ejecutando en paralelo; en este caso 5 instrucciones. También observamos que tras tener lleno el pipeline del procesador, acabará una instrucción a cada ciclo.

[11] Jiménez, D. (s,f). *Introducción a las arquitecturas paralelas.* Universitat Oberta de Catalunya

El número de etapas en la que se puede segmentar un procesador es distinto al de otro procesador. De todas las etapas, aquella que sea la que tarde más tiempo en ejecutarse, será la que determine la frecuencia del procesador. De este modo, una técnica que se sigue para aumentar la velocidad de los procesadores es aumentar el número de etapas, a cambió de reducir el tamaño de las mismas. Sin embargo, hay algunos inconvenientes en tener segmentaciones tan profundas. Principalmente, el mayor problema es el de los saltos, en el que se predice el sentido y destino del mismo. Una predicción incorrecta de éste penaliza de tal forma que todas las instrucciones que se han iniciado y no tenían que hacerlo hay que pararlas.

Por tanto, esta penalización es la que determina el número máximo de fases en las que se puede segmentar una instrucción. Para obviar esta penalización y poder aumentar la ratio antes comentada, se debe conseguir que se ejecuten más instrucciones por ciclo de reloj del procesador. Para ello, se modifica el hardware para ejecutar más instrucciones por cada ciclo. Este tipo de procesador se llama superescalar. Estos procesadores se basan en la existencia de varias unidades de cómputo duplicadas, de forma que se puede ejecutar más de una instrucción en el mismo ciclo de reloj.

Procesadores superescalares

Los procesadores superescalares, los que ya hemos indicado que tienen unidades funcionales duplicadas, aparecen en la década de los 80.

La figura de abajo muestra como un procesador puede lanzar varias instrucciones en cada ciclo de reloj, en este caso 2. Con este procesador superescalar de dos vías o pipelines se puede llegar a tener una ratio de ejecución de hasta 2 instrucciones por ciclo. Este adelanto, sin embargo, presenta una serie de problemas que limitan la ratio, las denominadas dependencias. Existen tres que afectan negativamente en la ratio: dependencia de verdad, dependencia de recursos, y dependencia de saltos o de procedimientos/funciones.

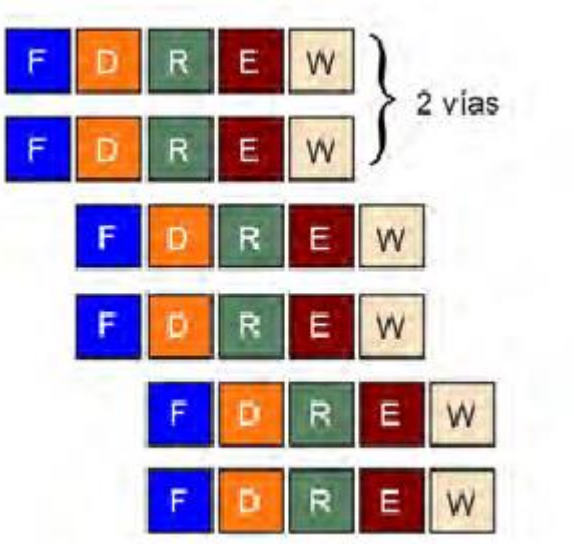

Figura 12. Procesamiento superescalar de 2 vías[12]

Cuando una instrucción en ejecución necesita un operando que en ese momento se está calculando por una instrucción anterior que también está siendo ejecutada, se produce la dependencia de verdad. Estas dependencias se deben resolver antes de realizar el lanzamiento de las dos instrucciones en paralelo para ejecutarse.

Anteriormente se ha comentado anteriormente la existencia de unidades funcionales de un tipo. Si no existen un número adecuado de éstas, se puede limitar el número de instrucciones que pueden lanzarse. A esto se le denomina dependencia de recursos, y viene derivado del hecho de que no existen un número ilimitado de recursos. Por ejemplo, si tenemos un procesador superes calar con dos pipelines, pero una única unidad funcional de coma flotante, no podremos lanzar más de una instrucción que vaya a la unidad funcional.

La última dependencia existente en este tipo de procesadores es la dependencia por salto. Es equivalente a lo que ocurre en el anteriormente citado procesador segmentado. El destino del salto sólo es conocido en el momento de la ejecución, y, por consiguiente, seguir ejecutando instrucciones sin saber cuál es el resultado del salto puede llevar a errores. Para evitar este tipo de errores, se utiliza un método especulativo para intentar predecir el salto y el destino del mismo. En caso de error, se deshace todo lo que se había hecho.

Procesadores VLIW

Estos procesadores se basan en la existencia de instrucciones formadas por varias instrucciones, de ahí el nombre de "largas". De esta forma se evitan las

[12] Jiménez, D. (s,f). *Introducción a las arquitecturas paralelas.* Universitat Oberta de Catalunya

dependencias por salto que se han comentado anteriormente. Normalmente también disponen de instrucciones dedicadas para controlar el flujo de ejecución del programa.

La siguiente figura muestra el esquema de instrucción: una suma, un desplazamiento binario, una instrucción SIMD, una operación de lectura y una de escritura. El compilador debe devolver palabras en cinco instrucciones que se puedan ejecutar en paralelo. En caso que sea imposible obtener estas instrucciones, se rellena con operaciones vacias o "no operation" en inglés.

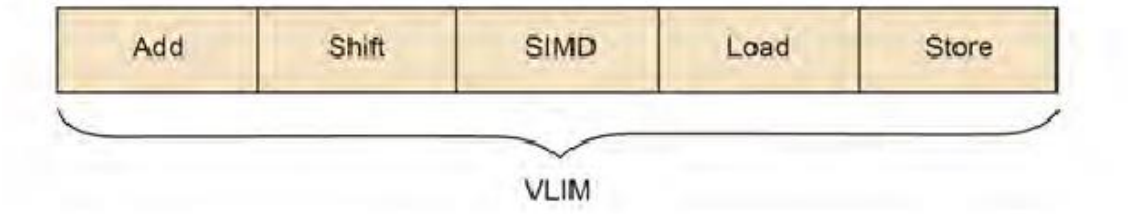

Figura 13. Esquema de instrucción VLIW[13]

3.2.1.2. Paralelismo a nivel de threads

El paralelismo a nivel de instrucción que se ha comentado en el apartado anterior, puede llegar a no ocurrir si existen fallos al acceder a los diferentes niveles de la memoria caché. Esto se debe a que la ejecución de nuevas instrucciones se detiene hasta que el dato, en caché, se extrae. Para resolver este problema, se puede conseguir que el procesador siga ejecutando instrucciones de otro programa o hilo de ejecución, evitando así la situación de bloqueo. A esto se le denomina multithreading, y existen distintos tipos: de grano fino, de grano grueso y simultáneo.

Multithreading de grano fino

El algoritmo Round Robin, comúnmente conocido en el ámbito de la asignación de la CPU a un proceso, se utiliza para ocultar los bloqueos del propio procesador, seleccionando instrucciones de hilos diferentes. Este algoritmo da un número fijo de ciclos a un hilo de ejecución detrás de otro, de forma consecutiva. De esta forma se reduce la posibilidad de que el procesador quede vacío en los ciclos de bloqueo producidos por otro hilo de ejecución.

En la siguiente figura aparecen 4 hilos de ejecución, que se van ejecutando en las diferentes unidades funcionales del procesador superescalar. Por cada ciclo de ejecución, el hilo procesado es distinto.

[13] Jiménez, D. (s,f). *Introducción a las arquitecturas paralelas.* Universitat Oberta de Catalunya

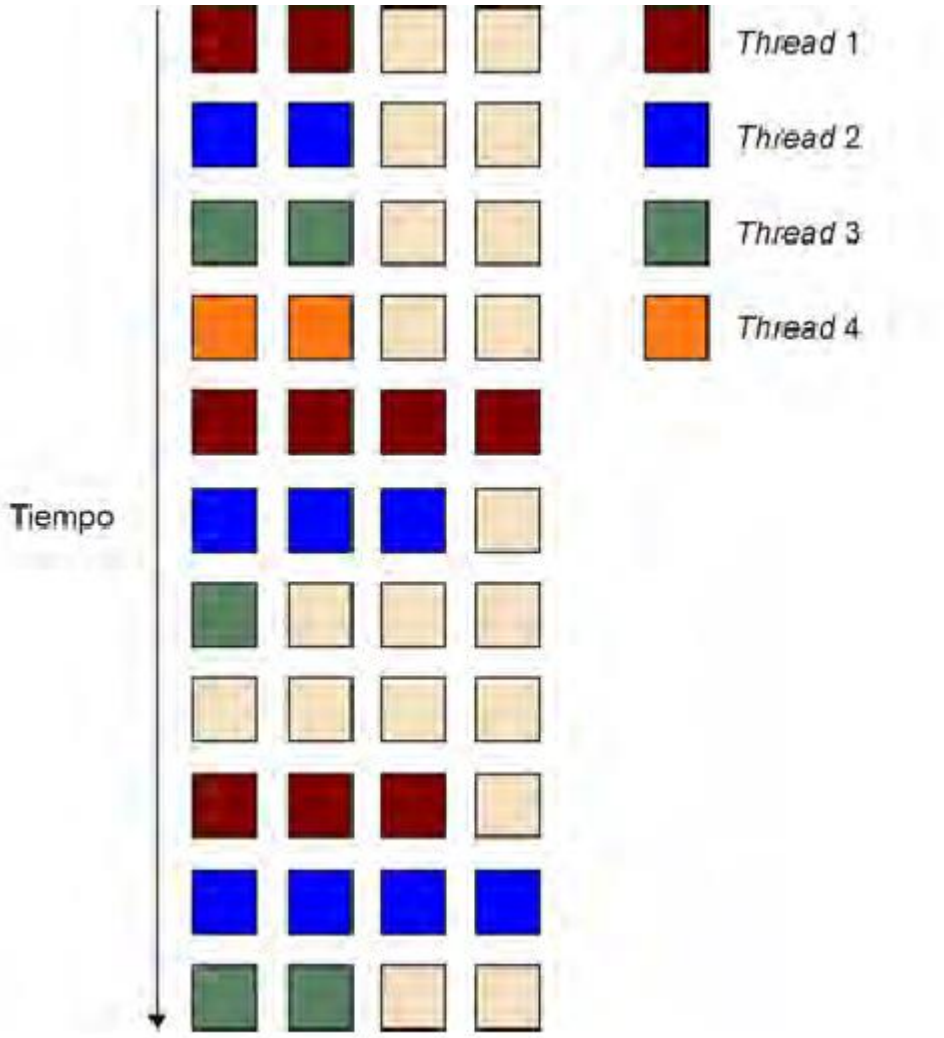

Figura 14. Ejecución multihilo de grano fino[14]

Este tipo de paralelismo necesita de un conjunto de registros asociados para poder realizar la ejecución de los hilos de forma independiente. De este modo, cada instrucción sabe qué registro usar. Por consiguiente, el hardware de la máquina será determinante para poder ejecutar un número mayor o menor de hilos a la vez.

Multithreading de grano grueso

Siguiendo la filosofía del multihilo, esta técnica difiere en que es posible ejecutar más de una instrucción de un mismo hilo en ciclos consecutivos. La siguiente figura detalla de forma gráfica.

[14] Jiménez, D. (s,f). *Introducción a las arquitecturas paralelas.* Universitat Oberta de Catalunya

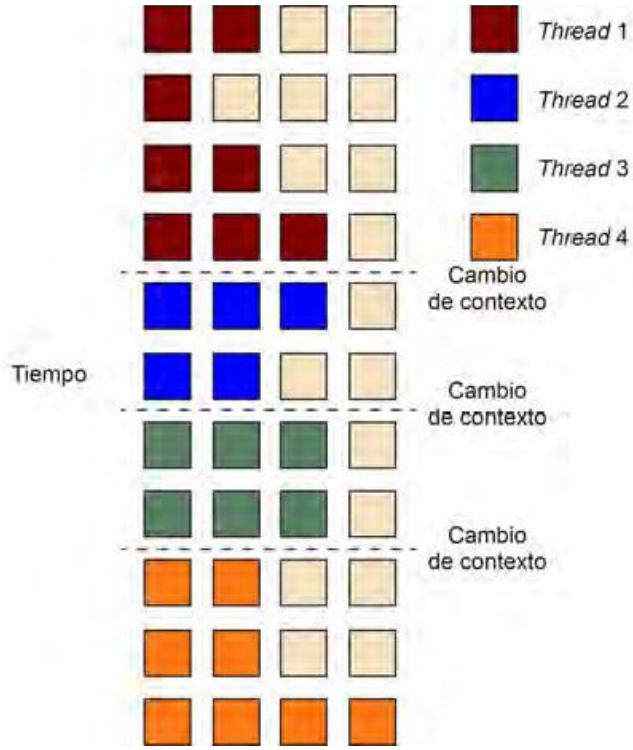

Figura 15. Ejecución multihilo de grano grueso[15]

En este caso, la ejecución de un hilo continúa hasta que se produce un bloqueo de los que ya hemos comentado (fallo de memoria, salto, datos erróneos). De esta forma de mantener la ejecución más de un ciclo a un hilo son es necesario tener tantos hilos activos. Sin embargo, el hecho de que el cambio de hilo se produzca cuando haya un bloqueo, los ciclos que se necesiten para darse cuenta del bloqueo y cambiar de hilo se perderán. En cualquier caso, si el número de hilos activos es suficiente, se mejora el rendimiento que con la técnica del grano fino.

Con este tipo de multithreading de grano grueso puede también tener más sentido vaciar el pipeline cada vez que cambiemos de thread. De esta forma no tenemos que tener la información sobre qué banco de registros se tiene que usar a lo largo de toda la ejecución de una instrucción.

Simultaneous Multithreading

Esta técnica reduce el desaprovechamiento, permitiendo que dos hilos diferentes puedan ejecutarse en el mismo ciclo de reloj. Se puede considerar como un refinamiento del grano grueso, de tal forma que, si en un ciclo del procesador una instrucción de un thread se bloquea, una instrucción de otro thread se puede usar para mantener el procesador y todas sus unidades funcionales ocupadas. También es posible que una instrucción de un thread pudiera quedar bloqueada porque hay un número

[15] Jiménez, D. (s,f). *Introducción a las arquitecturas paralelas.* Universitat Oberta de Catalunya

limitado de unidades funcionales de un tipo. En este caso también se podría coger una instrucción de otro thread.

En la siguiente figura se muestra la ejecución de 4 hilos en un sistema simultáneo de multihilo. Como se puede ver, en cada ciclo de ejecución puede haber instrucciones de diferentes threads.

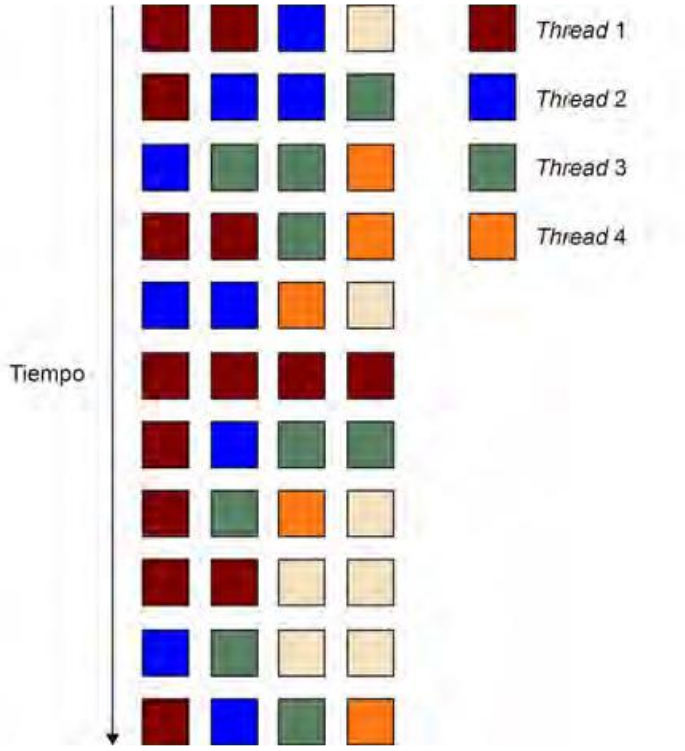

Figura 16. Ejecución simultánea con multihilo[16]

De cara a los sistemas operativos, un procesador que presente esta aproximación, es visto como un procesador con dos núcleos, que comparten caché y memoria. En cambio, cuando nos referimos al hardware, hay que conocer qué recursos se comparten y cómo hay que gestionarlos. Intel contemplaba cuatro estrategias diferentes.

3.2.1.3. Paralelismo a nivel de datos

Después de comentar los paralelismos a nivel de instrucciones y de threads, queda por detallar el paralelismo a nivel de datos. Este paradigma realiza una división del flujo de datos de un programa, asignando a cada unidad central de procesamiento uno de éstos subconjuntos. En cada procesador se realizan los mismos cálculos que en los otros procesadores, pero sobre el subconjunto de datos que le ha sido asignado. Se podría decir, de forma simple, que se dividen los datos y se repite el procesamiento. Esta acción, de forma ideal, tiene como resultado una aceleración sobre el cómputo del programa. Está especialmente indicada para operaciones sobre vectores y matrices, ya

[16] Jiménez, D. (s,f). *Introducción a las arquitecturas paralelas.* Universitat Oberta de Catalunya

que en este tipo de estructuras de datos se suele aplicar la misma operación sobre los elementos de éstas.

Para poder ejecutar este tipo de instrucciones, el hardware de este tipo de máquinas (SIMD) debe incluir registros más grandes, unos buses de memoria que puedan acceder a los datos del tamaño de los registros y unidades funcionales que permitan las operaciones con más de un dato de forma simultánea.

El tamaño del registro se divide en bloques de elementos dependiendo la instrucción. En la figura siguiente, se muestra una distribución del tamaño del registro según la semántica de la instrucción. Las operaciones que se aplican a cada elemento sólo afectan a esos elementos, a no ser que la semántica de la instrucción permita que se opere entre dos elementos consecutivos de un mismo registro. En la parte de la derecha se muestra una operación de suma sobre 2 registros vectoriales, en el que cada uno existen 4 elementos de tipo entero que ocupan 32bits.

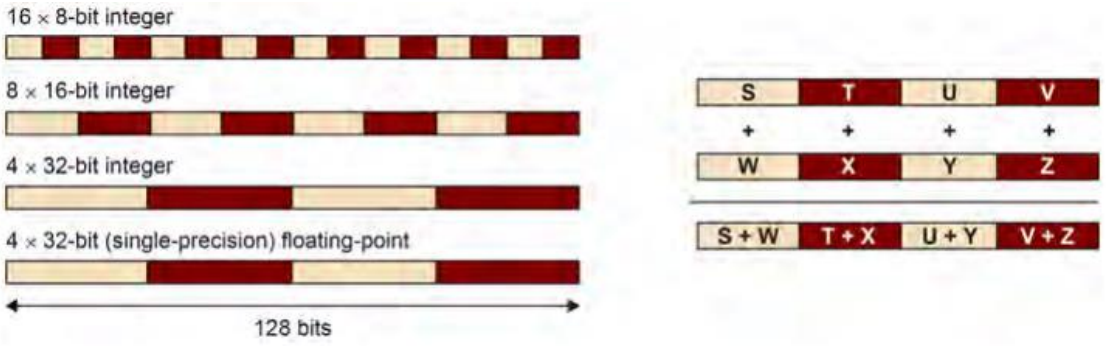

Figura 17. Instrucciones vectoriales[17]

La evolución de este tipo de paralelismo ha sido progresiva, apareciendo nuevas unidades funcionales de tipo SIMD. Entre estas unidades funcionales cabe destacar, las MMX de Intel, que operaban únicamente sobre enteros de 32 bits, no pudiendo realizar operaciones sobre flotantes. Otras unidades a destacar son las extensiones SSE, con la incorporación de flotantes y el aumento del número de bits.

3.2.2. Multiprocesadores y multicomputadores

En este apartado se van a introducir las diferencias entre los multiprocesadores y los multicomputadores. Según Tanenbaum (2006), la clasificación en la que se pueden dividir las distintas arquitecturas se pueden esquematizar en la siguiente figura. En ella, se puede ver como los multicomputadores están basados en memoria distribuida, mientras que los multiprocesadores se basan en memoria compartida.

[17] Jiménez, D. (s,f). *Introducción a las arquitecturas paralelas.* Universitat Oberta de Catalunya

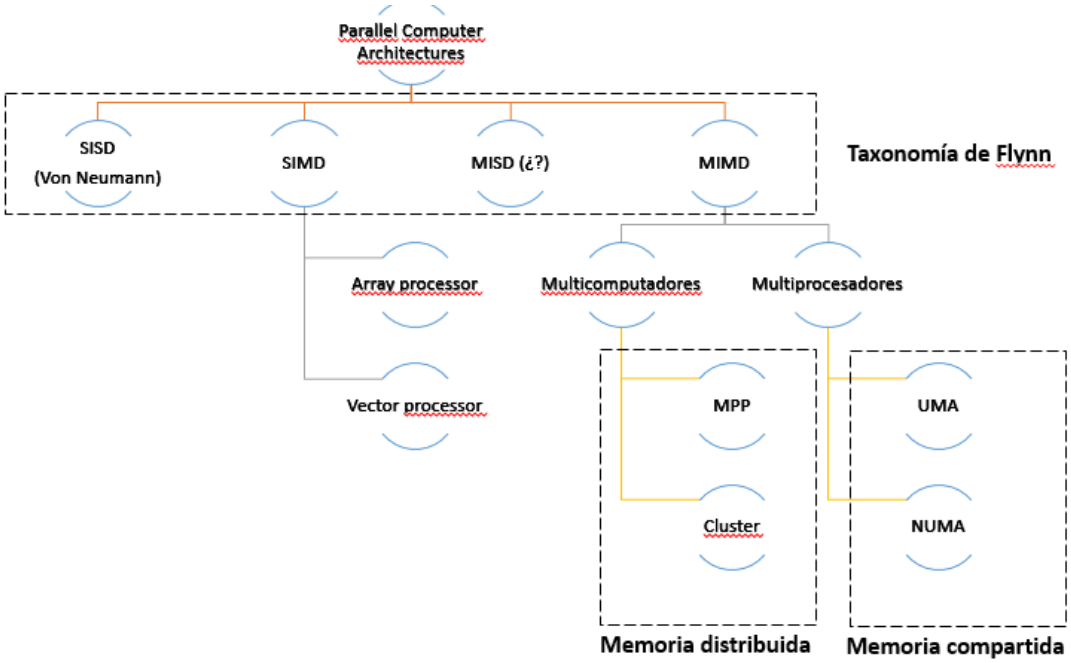

Figura 18. Clasificación Tanenbaum[18]

3.2.2.1. Multiprocesadores

Un multiprocesador se entiende como una máquina que trabaja de forma paralela compuesta por varios procesadores que están interconectados entre sí y que comparten una misma memoria. Cada procesador puede configurarse para que ejecute una determinada parte de un programa o incluso de varios programas a la vez. Los procesadores se pueden configurar para que ejecute cada uno una parte de un programa o varios programas al mismo tiempo. De forma general, un multiprocesador está formado por n procesadores y m módulos de memoria. La red de interconexión conecta cada procesador a un subconjunto de los módulos de memoria.

Al compartir diferentes módulos de memoria, y varios procesadores acceden a un mismo módulo, a los multiprocesadores se les puede denominar sistema de memoria compartida. Particularizando la forma en que comparten la memoria, se puede hacer una clasificación de los multiprocesadores:

UMA (Uniform Memory Access):

En este modelo, de memoria de acceso uniforme, todos los procesadores del sistema comparten la misma memoria, de forma uniforme. Dicho de otro modo, los procesadores tienen el mismo tiempo de acceso a todas las palabras de memoria. Cada procesador puede tener su cache privada, y los periféricos son también compartidos de alguna manera.

[18] Tanenbaum, A. Organización de computadores. Un enfoque estructurado

Como estos recursos se comparten en un alto grado, a estos sistemas se les denomina fuertemente acoplados. Existe un bus común que se utiliza como red de conexión entre los recursos.

Si los periféricos son accedidos de igual forma por los procesadores, se denomina que es un sistema simétrico. Así, todos los procesadores poseen la misma capacidad para ejecutar los programas. Por el contrario, en los multiprocesadores llamados asimétricos, existe únicamente un subconjunto del total de los procesadores que tienen la capacidad de ejecutar los programas. A los que tienen esa capacidad se les denomina maestros y a los demás, se les llama procesadores adheridos. La siguiente figura muestra el modelo de memoria de acceso uniforme de un multiprocesador.

Si además existe la coherencia de caché, a estos sistemas se les llama CC-UMA (*Cache-Coherent Uniform Memory Access*).

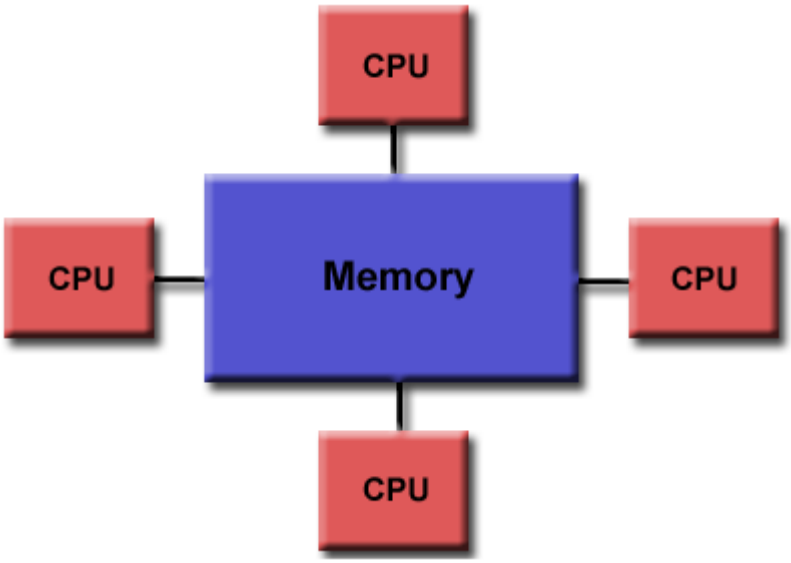

Figura 19. Modelo UMA de un multiprocesador[19]

NUMA (Non-Uniform Memory Access):

Al contrario que el caso anterior, un multiprocesador de tipo NUMA es un sistema de memoria compartida donde el tiempo de acceso varía según el lugar donde se encuentre localizado el acceso. En estos sistemas la memoria es compartida pero local a cada procesador. Existen los llamados clústers, que no son más que agrupaciones de sistemas que se comunican a través de otra red de interconexión y que puede incluir una memoria compartida a nivel global.

La principal ventaja de estos sistemas es que el acceso a memoria es más rápido que en los sistemas UMA. La memoria que se utiliza por cada uno de los procesos que

[19] https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

son ejecutados por cada procesador se intenta que sea lo más cercana posible, es decir, en la memoria de cada procesador y así disminuir el tiempo

Al igual que hay sistemas de tipo CC-UMA, también existe el modelo de acceso a memoria no uniforme con coherencia de caché CC-NUMA (*Cache-Coherent Non-Uniform Memory Access*) que consiste en memoria compartida distribuida y directorios de cache.

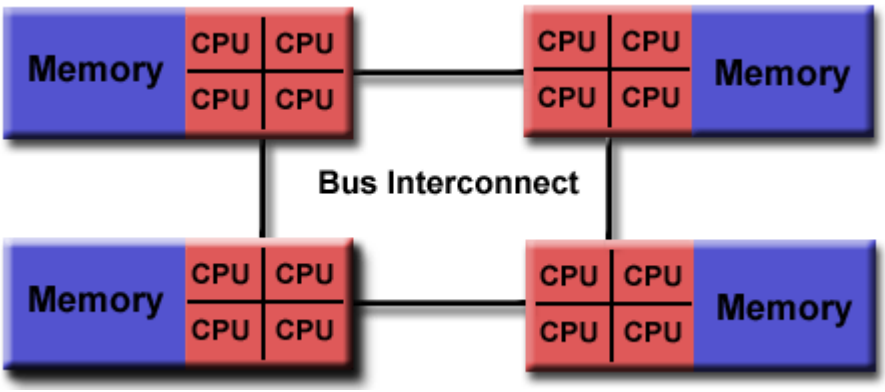

Figura 20. Modelo NUMA de un multiprocesador[20]

Para finalizar este apartado referente a los multiprocesadores, se van a indicar una serie de ventajas y desventajas de éstos:

Ventajas:

- El espacio de direcciones global proporciona una perspectiva de programación fácil de usar para la memoria
- El intercambio de datos entre tareas es rápido y uniforme debido a la proximidad de la memoria a las CPU

Desventajas:

- La principal desventaja es la falta de escalabilidad entre la memoria y las CPU. Agregar más CPU puede aumentar el tráfico geométricamente en la ruta de la CPU y la memoria compartida, y para los sistemas coherentes de caché, aumentar el tráfico asociado con la administración de memoria / caché.
- Responsabilidad del programador para las construcciones de sincronización que aseguran el acceso "correcto" de la memoria global.

3.2.2.2. Multicomputadores. Sistemas distribuidos

A diferencia de lo visto en el apartado anterior, un multicomputador presenta procesadores que tienen su propia memoria local. La memoria del sistema se encuentra distribuida entre todos los procesadores y, por lo tanto, cada procesador únicamente tiene acceso a las direcciones de memoria de forma local. Si se desea acceder a

[20] https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

posiciones de memoria de otros procesadores, se utiliza la técnica del paso de mensajes. Por lo tanto, este acceso local y privado a la memoria es lo que diferencia los multicomputadores de los multiprocesadores.

Cuando hay que transmitir datos entre los procesadores se utiliza una red de conexión, en la cual se conectan unos procesadores con otros. La transferencia de unos procesadores a otros se realiza por tanto por múltiples transferencias entre procesadores conectados dependiendo de cómo esté establecida la red.

A estos sistemas se les denomina sistemas distribuidos, por el hecho que la memoria está distribuida entre los diferentes elementos del proceso. Esto no debe llevar a confusión, ya que puede haber sistemas que tengan memoria distribuida pero no compartida, en cuyo caso no estaríamos hablando de multicomputadores. Al contrario que los multiprocesadores, a estos sistemas se les denomina débilmente acoplados, ya que los elementos trabajan de forma independiente unos de otros.

Al igual que los sistemas de memoria compartida, los sistemas de memoria distribuida varían ampliamente, pero comparten una característica común. Los sistemas de memoria distribuida requieren una red de comunicación para conectar la memoria entre procesadores.

Los procesadores tienen su propia memoria local. Las direcciones de memoria en un procesador no se asignan a otro procesador, por lo que no existe un concepto de espacio de direcciones global en todos los procesadores.

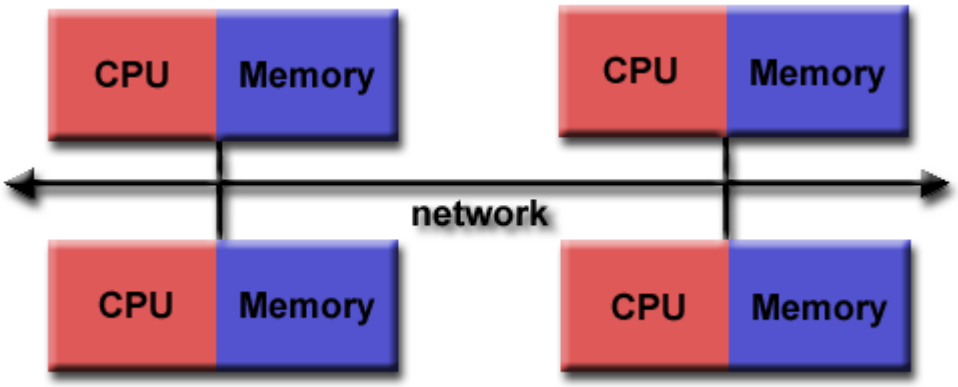

Figura 21. Multicomputadores[21]

Debido a que cada procesador tiene su propia memoria local, opera de manera independiente. Los cambios que realiza en su memoria local no tienen efecto en la memoria de otros procesadores. Por lo tanto, el concepto de coherencia de caché no se aplica.

[21] https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

Cuando un procesador necesita acceder a los datos en otro procesador, generalmente es tarea del programador definir explícitamente cómo y cuándo se comunican los datos. La sincronización entre tareas es responsabilidad del programador.

El tipo de red utilizada para la transferencia de datos varía ampliamente, aunque puede ser tan simple como Ethernet.

Al igual que se ha hecho con los multiprocesadores, se va a detallar una serie de ventajas y desventajas de este tipo de arquitectura.

Ventajas:

- La memoria es escalable con el número de procesadores. Aumenta la cantidad de procesadores y el tamaño de la memoria aumenta proporcionalmente.
- Cada procesador puede acceder rápidamente a su propia memoria sin interferencias y sin los gastos generales incurridos al tratar de mantener la coherencia de caché global.
- Rentabilidad: puede utilizar productos básicos, procesadores estándar y redes.

Desventajas:

- El programador es responsable de muchos de los detalles asociados con la comunicación de datos entre procesadores.
- Puede ser difícil asignar las estructuras de datos existentes, basadas en la memoria global, a esta organización de memoria.
- Tiempos de acceso a la memoria no uniformes: los datos que residen en un nodo remoto tardan más en acceder a los datos locales del nodo

3.3. Programación paralela y distribuida

La concurrencia es un área central de la informática que ha existido durante mucho tiempo en las áreas de computación científica y de alto rendimiento y en el diseño de sistemas operativos, bases de datos, sistemas distribuidos e interfaces de usuario. Se ha convertido rápidamente en un tema relevante para una audiencia más amplia de programadores con el advenimiento de las arquitecturas paralelas en los sistemas de consumo. La presencia de procesadores multinúcleo en computadoras de escritorio y portátiles, potentes coprocesadores de unidades de procesamiento de gráficos programables (GPU) y hardware especializado como el procesador Cell, ha convertido el tema de la concurrencia como un problema clave de programación en todas las áreas de computación. Dada la proliferación de los recursos informáticos que dependen de la concurrencia para lograr el rendimiento, los programadores deben aprender a programarlos adecuadamente para aprovechar al máximo su potencial.

Acompañando a esta proliferación de soporte de hardware para el paralelismo, existe una proliferación de herramientas de lenguaje para la programación de estos dispositivos. Los principales fabricantes admiten las técnicas existentes, como las interfaces de subprocesamiento estandarizadas y las anotaciones de idiomas como OpenMP. Los fabricantes de procesadores gráficos, como NVidia, han introducido el lenguaje Compute Unified Devide Architecture (CUDA) para GPU, mientras que IBM ha creado bibliotecas para controlar el procesador Cell que se encuentra en la plataforma de juegos Playstation y en varias máquinas de tipo servidor. El grupo Khronos, una organización de estándares de la industria responsable de OpenGL, definió recientemente un lenguaje para programar plataformas heterogéneas compuestas por CPU, GPU y otros procesadores llamados OpenCL. Este lenguaje es compatible con los modelos de programación de datos paralelos que son familiares para los programadores de CUDA, pero, además, OpenCL incluye los modelos de programación de tareas paralelas que se usan comúnmente en las CPU.

3.3.1. Lenguajes para la programación paralela

Los lenguajes de programación han sufrido un gran cambio en los últimos años, debido a las necesidades de los desarrolladores. Estas necesidades se han ido centrando en el incremento de la complejidad del software desarrollado, y en los continuos cambios del hardware sobre el que se ejecutan esos programas.

Los cambios en la complejidad requieren que los lenguajes ayuden a los desarrolladores a controlar la estructura del código y analizar varios tipos de propiedades de corrección del código que son importantes.

En este apartado se va a realizar un recorrido sobre los principales lenguajes de programación y su relación con la computación paralela.

3.3.1.1. Fortran

Comenzar por este lenguaje no significa que sea el más relevante en cuanto a la programación paralela, sino que, cronológicamente, es el primero que tuvo una gran aceptación por los desarrolladores.

Este lenguaje, cuyo acrónimo viene del sistema de traducción de fórmulas matemáticas de IBM, fue desarrollado en los años 50. Este lenguaje se asemeja mucho al lenguaje ordinario de las matemáticas (Backus et al., 1956). Fue desarrollado para la máquina 704 EDPM de IBM, la cual aceptaba un programa escrito en ese lenguaje y que producía un programa objeto en su propio lenguaje máquina. Fortran ha sido utilizado desde sus inicios y durante más de medio siglo en distintas disciplinas, como predicción con modelos numéricos del tiempo, análisis de elementos finitos, dinámica de fluidos computacional, física computacional, entre otras. Según Loh (2010), es el lenguaje ideal para la computación de alto rendimiento.

Fortran proporcionó algunas abstracciones clave que influyeron en los lenguajes posteriores. Incluía la noción de una variable desacoplada de su dirección física en la memoria principal. Fortran también proporcionó construcciones de flujo de control para ramificación y bucle, desacoplando de manera similar la estructura del código de las direcciones reales que el contador del programa cargaba durante la ejecución. Al utilizar estos, los programadores no tenían la responsabilidad de realizar un seguimiento manual de los detalles de diseño del programa real y del almacén de datos. En su lugar, podrían programar basándose en nombres y etiquetas que correspondían a partes significativas de su programa. La generación automatizada de código de flujo de control permitió que la complejidad de los bucles aumentara sin cargar al programador con la gestión de los espaguetis del flujo de control que resulta de un programa suficientemente complejo.

Las primeras versiones de Fortran también introdujeron el concepto de subrutina, un bloque de código al que se podía llamar durante la ejecución de un programa y se pasaban los datos a través de parámetros. Este fue un avance importante sobre la simple definición de bloques de código que podrían ejecutarse mediante un salto y retorno sin formato. La abstracción de la subrutina y sus parámetros permitieron al compilador ocultar el mecanismo por el cual los datos se pusieron a disposición de la subrutina, gestionando el contador del programa para hacer una transición entre la parte que llama y la parte llamada.

Fortran no fue el primer idioma en adoptar construcciones de concurrencia en su estándar oficial. De hecho, hasta 1990 no se estandarizó una versión de Fortran que incluía características paralelas de datos. Las revisiones posteriores del lenguaje en 1995 y 2008 han adoptado características adicionales específicamente para la concurrencia. Por otro lado, las extensiones de Fortran para el paralelismo han existido durante la mayor parte de la vida del lenguaje. (Sottile et al., 2010)

3.3.1.2. ALGOL

Este lenguaje de programación, cuyo nombre proviene de las iniciales de Algorithmic Language obtuvo gran popularidad en las universidades en la década de los 60, pero no llego a ser un referente para el desarrollo de software comercial. Sin embargo, tuvo una gran influencia en otros lenguajes que sí tuvieron una gran difusión, como Pascal, C y Ada.

La evolución más allá de Fortran se produjo por muchas razones, siendo la más dominante la necesidad de controlar programas cada vez más complejos y el deseo de crear lenguajes más propicios para una compilación automatizada eficiente. Unos años después de la introducción de Fortran, se inventó un nuevo lenguaje llamado ALGOL que llevó la noción de programación estructurada al panorama de los lenguajes de programación, según Backus et al. (1960). La programación estructurada tenía como

objetivo manejar códigos que se volvieron innecesariamente complejos de administrar a medida que se hacían más grandes. Este lenguaje, en su origen, introdujo la familiar estructura *if-then-else* y el conocido bucle *for-loop*.

De manera sorprendente, las siguientes versiones, más pulidas, sobre la década de los 60, abordaron las deficiencias de la versión original. Los diseñadores de Algol en su versión 68, decidieron abordar directamente el problema de expresar paralelismos dentro de los programas en esta versión revisada del lenguaje. En Algol 68, se agregó el concepto de *cláusula colateral* para llamar partes de secuencias de instrucciones que se podrían reordenar o ejecutar simultáneamente sin efectos perjudiciales en la salida del programa. Además, el lenguaje proporcionó mecanismos de sincronización a través de variables del tipo de datos *sema* correspondientes a los semáforos de Dijkstra. La concurrencia se convirtió en un tema de interés para la comunidad informática durante la década de 1960, a medida que los límites en la utilización y el rendimiento de los sistemas por lotes de proceso único se hacían evidentes, especialmente con una creciente brecha entre el rendimiento de los elementos computacionales y el almacenamiento de datos, como la memoria y la cinta. (Sottile et al., 2010)

Las variables de modo *sema* pueden utilizarse mediante los operadores *up* y *down* que incrementan y disminuyen la variable. El compilador debe ser notificado explícitamente cuando las variables *sema* están presentes dentro de una cláusula colateral, encapsulando la cláusula dentro de una cláusula paralela a través del símbolo *par*. El concepto de *cobegin* que se encuentra en lenguajes posteriores subsiguientes se remonta al *par-begin* introducido en Algol 68

3.3.1.3. ADA

El lenguaje ADA fue desarrollado en el Departamento de Defensa de los EEUU en los años 70 del siglo pasado. La idea original del lenguaje era la de construir software crítico para el desarrollo militar y comercial. Está basado en un gran número de lenguajes de finales de los años 50 y mediados de los años 70. Su principal influencia es el lenguaje Pascal.

Desde el punto de vista de la concurrencia, Ada es un lenguaje que incluye construcciones que facilitan la programación concurrente en forma de tareas con cualquier tipo de sistema de memoria compartida o paso de mensajes.

Como se ha comentado en el párrafo anterior, el elemento principal de Ada con respecto al paralelismo es la tarea. Una tarea representa un hilo independiente de control que ejecuta de forma concurrente otras tareas. Según Ali y Pinho (2011), el modelo de tarea de Ada es adecuado para sistemas concurrentes donde cada tarea se asigna a una aplicación concurrente, que puede cancelar, suspender o reanudar su ejecución según sean los requisitos del sistema.

```
1   procedure Get_Dressed is
2       task Put_On_Pants is
3       ...
4       end;
5
6       task body Pun_On_Pants is
7       ...
8       end;
9
10      task Put_On_Shirt is
11      ...
12      end;
13
14      task body Put_On_Shirt is
15      ...
16      end;
17
18  begin
19      Put_On_Shoes;
20  end Get_Dressed;
```

Figura 22. Código en ADA que describe dos tareas que se ejecutan concurrentemente.[22]

El método principal para las tareas que se comunican entre sí en Ada es a través de lo que se conoce como cita. Una tarea de Ada proporciona el área conocida como entradas, que son similares a las especificaciones de procedimientos, aunque están restringidas a ocurrir dentro de declaraciones de objetos protegidos o de tareas. Una tarea proporciona el cuerpo para una entrada en lo que se conoce como declaraciones de aceptación.

```
1   task Container is
2       entry PUT (X: in INTEGER);
3       entry GET (X: out INTEGER);
4
5   begin
6       loop
7           accept PUT(X: in INTEGER) do
8               V := X;
9           end;
10          accept GET(X: out INTEGER) do
11              X := V;
12          end;
13      end loop;
14  end Container;
```

Figura 23. Código en ADA con entradas[23]

[22] Sottile, M.J. et al (2010). *Introduction to Concurrency in Programming Languages*
[23] John G. P. Barnes. An overview of Ada. Software-Practice and Experience, 10:851-887, 1980.

El código anterior es una tarea que representa un contenedor que almacena un entero y que espera un elemento cuando está vacío y espera otra tarea que lo devuelva cuando esté lleno.

3.3.1.4. Haskell

Concurrent Haskell es el nombre colectivo de los elementos que Haskell proporciona para la programación con múltiples hilos de control. A diferencia de la programación paralela, donde el objetivo es hacer que el programa se ejecute más rápido mediante el uso de más CPU, el objetivo en la programación concurrente suele ser escribir un programa con múltiples interacciones, según Marlow (2013).

En muchas áreas de aplicación hoy en día es necesario algún tipo de concurrencia. Una aplicación típica a la que se puede enfrentar un usuario tendrá una interfaz que debe permanecer receptiva mientras la aplicación está descargando datos de la red o calculando algunos resultados. A menudo, estas aplicaciones pueden estar interactuando con múltiples servidores a través de la red al mismo tiempo; un navegador web, por ejemplo, tendrá muchas conexiones simultáneas abiertas a los sitios que el usuario está navegando, mientras que siempre mantiene una interfaz de usuario con la que se puede interactuar. Las aplicaciones del lado del servidor también necesitan concurrencia para administrar múltiples interacciones de clientes simultáneamente.

Haskell considera que la concurrencia es una abstracción útil porque permite que cada interacción se programe por separado, lo que resulta en una mayor modularidad. Las abstracciones no deben ser demasiado costosas porque entonces no se usarían; por lo tanto, GHC proporciona subprocesos ligeros para que la concurrencia pueda usarse para una amplia gama de aplicaciones.

La filosofía de Haskell es proporcionar un conjunto de características muy simples pero generales que puede utilizar para crear una funcionalidad de nivel superior. Entonces, si bien la funcionalidad incorporada puede parecer bastante escasa, en la práctica es lo suficientemente general como para implementar abstracciones elaboradas. Además, debido a que estas abstracciones no están integradas, puede tomar sus propias decisiones sobre qué modelo de programación adoptar, o programar hasta las interfaces de bajo nivel para mejorar el rendimiento.

Por lo tanto, para aprender Haskell, podemos comenzar desde las interfaces de bajo nivel y luego explorar cómo combinarlas y construir capas superiores para crear abstracciones de alto nivel. El objetivo es que al construir las implementaciones de abstracciones de nivel superior utilizando las funciones de bajo nivel, las abstracciones de nivel superior serán más accesibles.

Un programa de Haskell puro puede parecer tener muchas opciones para implementar paralelismo automático. Dada la falta de efectos secundarios, puede parecer que podemos evaluar cada subexpresión de un programa Haskell en paralelo. En la práctica, esto no funciona bien porque crea demasiados artefactos pequeños de trabajo que no pueden ser programado eficientemente y el paralelismo está limitado por las dependencias de datos en el programa fuente. Haskell proporciona un mecanismo para permitir al usuario controlar la granularidad de paralelismo indicando qué cálculos pueden realizarse de manera útil de forma paralela. Esto se hace usando funciones del módulo Control Parallel, según Peyton y Singh (2019).

3.3.1.5. Lenguajes de programación modernos

Java

A principios de siglo, apareció un framework de JAVA para resolver problemas de paralelismo a través de la división de problemas de forma recursiva, repartiendo entre subtareas que resuelven en paralelo y esperando su finalización para posteriormente componer el resultado. Este framework fue diseñado por Lea (2000), y se llama Fork/Join Framework.

Según el autor, el paralelismo que se puede implementar con este framework se encuentra entre las técnicas de diseño más simples y efectivas para obtener un buen rendimiento. Los algoritmos fork/join, bifurcación y unión en castellano, son versiones paralelas del conocido algoritmo divide y vencerás.

```
1  Result solve(Problem problem) {
2      if (problem is pequeño)
3          resolver directamente el problema
4      else {
5          dividir el programa en partes independientes
6          crear nuevas subtareas para cada parte (fork)
7          unir todas las subtareas (join)
8          componer resultado final de los subresultados
9      }
10 }
```

Figura 24. Explicación del algoritmo Fork/Join basado en divide y vencerás[24]

La operación fork de bifurcación inicia una nueva subtarea paralela. La operación join hace que la tarea actual no continúe hasta que se ha completado la subtarea bifurcada. Los algoritmos fork/join, como otros algoritmos basados en divide y vencerás, suelen ser algoritmos recursivos, en los que subdivide de forma repetida las subtareas hasta que son suficientemente pequeñas como para poder resolverlas.

[24] A Java Fork/Join Framework. Doug Lea (2010)

Actualmente, la librería *java.util.concurrent* contiene todo lo relativo a paralelismo del lenguaje. Podemos encontrar *fork/join*, además de otras interfaces y clases, como pueden ser los semáforos o cerrojos.

Para utilizar el algoritmo *fork/join* se utilizan las clases *ForkJoinTask* y *ForkJoinPool*. La clase referida al join, *ForkJoinPool*, es la que llama a la tarea de tipo *ForkJoinTask*. Si deseamos hacer múltiples llamadas, con subtareas en paralelo de forma recursiva, se instancia una tarea de la clase *RecursiveAction*, que extiende de *ForkJoinTask*.

Según Roales (2015), la clase *RecursiveAction* es la encargada de lanzar la tarea que vamos a paralelizar, mediante el método *compute*() de la misma. Para implementar el paralelismo de una tarea, se instancia esta clase recursiva y se sobrecarga el método *compute*() con la tarea principal. Además, existe una diferencia entre el código de la tarea secuencial y el código que contenga el método *compute*(); mientras la tarea secuencial consiste en un método que se llame a sí mismo de forma recursiva, el método *compute*() creará de forma recursiva nuevas instancias de su misma clase, correspondiendo cada nueva instancia a una subtarea distinta.

El siguiente diagrama, extraído de Fernández (2012), resume el funcionamiento del algoritmo fork/join.

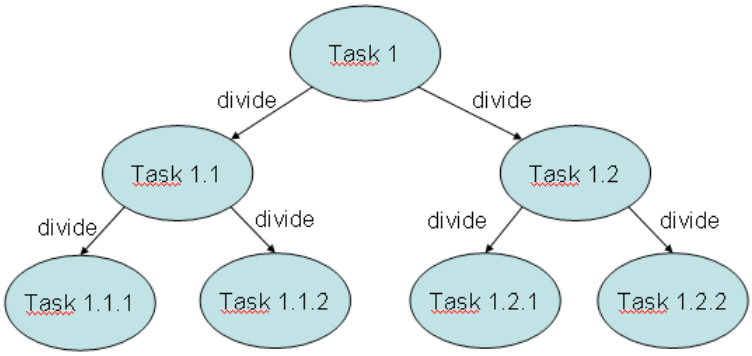

Figura 25. Fork/join en Java.

Además de estas clases para la implementación del framework *fork/join*, el paquete *java.util.concurrent* contiene otras muchas herramientas relativas a la creación de aplicaciones concurrentes. A continuación, se describen los principales componentes de este paquete, según Baeldung (2019).

- *Executor*: es una interfaz que representa un objeto que ejecuta las tareas proporcionadas.
- *ExecutorService*: es una solución completa para el procesamiento asíncrono. Administra una cola en memoria y programa las tareas enviadas en función de la disponibilidad de subprocesos.

- *ScheduledExecutorService*: es una interfaz similar a *ExecutorService*, pero puede realizar tareas periódicamente. Los métodos de *Executor* y *ExecutorService* están programados in situ sin introducir ningún retraso artificial.
- *Future*: se utiliza para representar el resultado de una operación asíncrona.
- *CountDownLatch*: introducido en el JDK5 es una clase de utilidad que bloquea un conjunto de subprocesos hasta que se completa alguna operación.
- *CyclicBarrier*: funciona casi igual que *CountDownLatch*, excepto que podemos reutilizarlo. A diferencia de *CountDownLatch*, permite que varios subprocesos se esperen entre sí utilizando el método *await* () (conocido como condición de barrera) antes de invocar la tarea final.
- *Semaphore*: El semáforo se usa para bloquear el acceso a nivel de subproceso a alguna parte del recurso físico o lógico. Un semáforo contiene un conjunto de permisos; Cada vez que un hilo intenta ingresar a la sección crítica, debe verificar el semáforo si hay un permiso disponible o no
- *ThreadFactory*: actúa como un grupo de hilos (no existente) que crea un nuevo hilo a pedido. Elimina la necesidad de una gran cantidad de codificación repetitiva para implementar mecanismos de creación de subprocesos eficientes.
- *BlockingQueue*: en la programación asíncrona, uno de los patrones de integración más comunes es el patrón productor-consumidor. El paquete *java.util.concurrent* viene con una estructura de datos conocida como *BlockingQueue*, que puede ser muy útil en estos escenarios asíncronos
- *DelayQueue*: es una cola de bloqueo de elementos de tamaño infinito donde solo se puede extraer un elemento si se completa el tiempo de caducidad (conocido como retraso definido por el usuario)
- *Locks*: es una utilidad para evitar que otros subprocesos accedan a un determinado segmento de código, aparte del subproceso que lo ejecuta actualmente. La principal diferencia entre un bloqueo y un bloque sincronizado es que el bloque sincronizado está completamente contenido en un método; sin embargo, podemos tener las operaciones *lock*() y *unlock*() de la API en métodos separados
- *Phaser*: es una solución más flexible que *CyclicBarrier* y *CountDownLatch*, se utiliza para actuar como una barrera reutilizable en la que el número dinámico de subprocesos debe esperar antes de continuar con la ejecución. Podemos coordinar varias fases de ejecución, reutilizando una instancia de *Phaser* para cada fase del programa

Python

El lenguaje Python, creado por Guido Van Rossum, es un lenguaje multi-paradigma y multipropósito. Ha sido ampliamente aceptado en todo el mundo debido a su poderosa sencillez y fácil mantenimiento. Existe una amplia gama de módulos para

hacer facilitar su uso. Dentro de la programación paralela, Python tiene módulos incorporados y externos que simplifican la implementación.

Según Palach (2014), Python presenta diferentes herramientas para poder implementar el paralelismo. Entre ellas se encuentran los módulos threading, multiprocessing, parallel y celery. A continuación se introducen estos módulos de forma breve, pudiendo encontrar información más extendida en la propia documentación de Python[25].

- *threading:* es el módulo de subprocesos de Python, que ofrece una capa de abstracción al módulo *_thread*, que es un módulo de nivel inferior. Proporciona funciones que ayudan al programador en la tarea de desarrollar sistemas paralelos basados en hilos.
- *multiprocessing:* este módulo tiene como objetivo proporcionar una API simple para el uso del paralelismo basado en procesos. Este módulo es similar al módulo threading, que simplifica alternancias entre los procesos sin mayores dificultades. El enfoque que está basado en procesos es muy popular dentro de la comunidad de usuarios de Python ya que es una alternativa a responder preguntas sobre el uso de subprocesos enlazados a la CPU.
- *parallel:* este módulo es externo a Python y ofrece una API enriquecida para la creación de paralelos y sistemas distribuidos que hacen uso del enfoque de procesos. Este módulo pretende ser ligero y fácil de instalar[26], y se integra con otros programas de Python. Entre algunas de las características, podemos destacar las siguientes:
    - Detección automática de la configuración óptima.
    - El hecho de que una serie de procesos de trabajo se pueden cambiar durante el tiempo de ejecución
    - Balance de carga dinámico
    - Tolerancia a fallos
    - Auto-descubrimiento de recursos computacionales.
- *Celery:* es un excelente módulo de Python que se utiliza para crear sistemas distribuidos y tiene una excelente documentación. Hace uso de al menos tres tipos diferentes de enfoque para ejecutar tareas en forma concurrente: multiprocesamiento, *eventlet* y *gevent*

Existe un mecanismo, llamado GIL (*Global Interpreter Lock*), que se utiliza en la implementación de Python estándar, conocida como *CPython*, para evitar bytecodes que se ejecutan simultáneamente por diferentes hilos. La existencia de GIL en Python es una razón para la discusión entre los usuarios de este lenguaje. GIL fue diseñado para

[25] https://docs.python.org/3/library/concurrency.html
[26] Web de descarga del modulo: http://parallelpython.com

proteger la memoria interna utilizada por el intérprete de *CPython*, que no implementa mecanismos de sincronización para el acceso concurrente por hilos. En cualquier caso, GIL resulta en un problema cuando decidimos utilizar hilos, y estos tienden a estar vinculados a la CPU. Los hilos de E / S, por ejemplo, están fuera del alcance de GIL.

Según el propio creador del lenguaje, eliminar este elemento GIL no es sencillo[27], y anima a la comunidad a desarrollar una rama del lenguaje en la que no contenga este elemento.

Además de estos elementos de programación paralela, existe PyCUDA, que permite acceder a la API de computación paralela CUDA de NVIDIA, la cual se comentará en siguientes apartados de este documento. Entre las características de PyCUDA se encuentran, según la propia web de NVidia[28]:

- Mapas de todo CUDA en Python.
- Permite la generación de códigos en tiempo de ejecución para códigos flexibles, rápidos y ajustados automáticamente.
- Robustez añadida: gestión automática de la vida útil de los objetos, comprobación automática de errores
- Conveniencia adicional: viene con álgebra lineal en GPU, reducción y escaneo listos para usar. Paquetes adicionales para FFT y LAPACK disponibles.
- Rápido.
- Documentación completa y útil.

C/C++

C y C++ no trabajan con la concurrencia directamente en su lenguaje, aunque existen diferentes librerías disponibles para ello. Principalmente, la conocida STL, *Standard Template Library*, que tiene más de 100 algoritmos para buscar, contar y manipular rangos y sus elementos. Con C++17, 69 de ellas están sobrecargadas y algunas nuevas se agregan. El algoritmo sobrecargado y nuevo se puede invocar con una llamada política de ejecución. Al utilizar la política de ejecución, puede especificar si el algoritmo debe ejecutarse de forma secuencial, paralela o paralela y vectorizada.

[27] https://www.artima.com/weblogs/viewpost.jsp?thread=214235
[28] https://developer.nvidia.com/pycuda

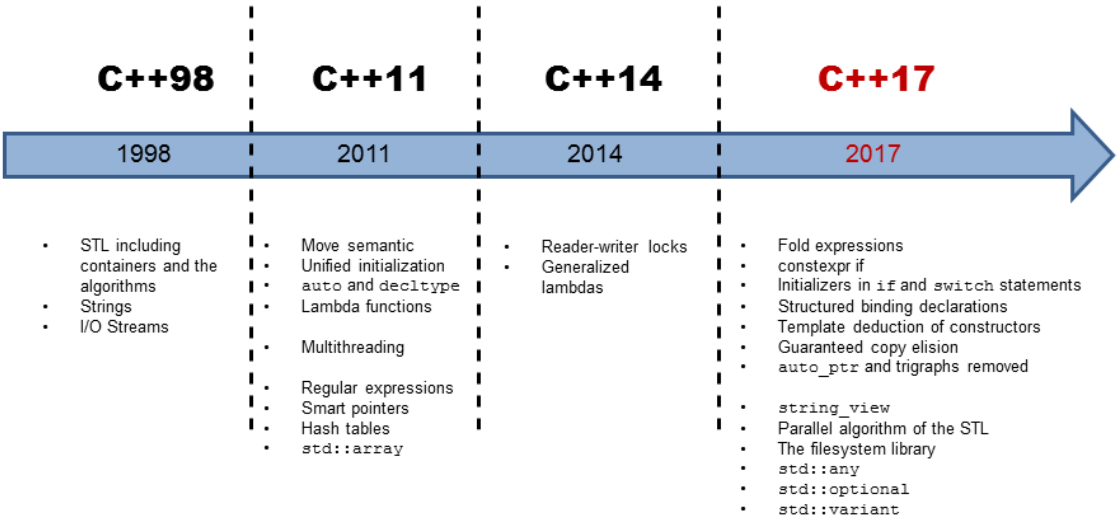

Figura 26. Evolución de la librería STL[29]

Antes de C ++ 11, C ++ no tenía mucho soporte para el paralelismo. Esto no significa que no fuese posible iniciar, controlar, detener y sincronizar subprocesos, pero fue es necesario utilizar bibliotecas específicas del sistema operativo porque los subprocesos estaban inherentemente relacionados con el sistema operativo. (Galowicz, 2017).

Con C ++ 11, llegó *std::thread*, que permite el control básico de subprocesos portátiles en todos los sistemas operativos. Para sincronizar hilos, C++11 también introdujo clases de exclusión mutua. Además de eso, *std::condition_variable* permitía la notificación flexible de eventos entre hilos.

Algunas otras implementaciones nuevas que fueron interesantes son *std::async* y *std::future*. Con la primera, se podía envolver una función normal en llamadas *std:async* para ejecutarlas asíncronamente en background. Esto devuelve objetos de tipo *std::future* , que prometen contener el resultado de la función más adelante, por lo que mientras se puede hacer otra cosa mientras se espera el resultado.

Otra mejora realmente enorme de la STL son las políticas de ejecución, las cuales pueden ser añadidas a los 69 algoritmos ya existentes. Esta adición significa que solo podemos agregar un único argumento de la política de ejecución a las llamadas algoritmo estándar existentes en nuestro antiguo código y obtener paralelismo sin reescrituras complejas.

La versión C++17 comprende una extensión realmente importante para el paralelismo: políticas de ejecución para algoritmos estándar. Se ampliaron sesenta y nueve algoritmos para aceptar políticas de ejecución, para poder lanzar en paralelo en múltiples núcleos, e incluso con vectorización habilitada.

[29] https://www.modernescpp.com/index.php/c-17-new-algorithm-of-the-standard-template-library

Para el usuario, esto significa que, si ya usamos algoritmos STL en nuestro código, obtenemos una mejora de paralelismo de forma "gratuita". Podemos fácilmente dar a nuestras aplicaciones paralelismo, simplemente agregando un solo argumento de política de ejecución a nuestras llamadas de algoritmo STL existentes.

Las características más reseñables de esta versión de STL para C++17, son las siguientes:

- Establecer un programa con un estado *sleep* por un determinado tiempo.
- Iniciar y detener hilos
- Realizar bloqueo compartido de forma segura con *std::unique_lock* y *std::shared_lock*
- Evitar deadlocks con *std::scoped_lock*
- Sincronizar de forma concurrente con el uso de *std::cout*
- Posponer de forma segura la inicialización con *std::call_once*
- Ejecutar tareas en segundo plano usando *std::async*
- Implementar el algoritmo *producer/consumer* con *std::condition_variable*
- Paralelizar el renderizador ASCII Mandelbrot usando *std::async*
- Implementar una pequeña librería de paralelismo automático con *std::future*

En un apartado posterior de este documento se hablará de OpenMP como un framework para la programación multiproceso. Esta API puede ser utilizada en programas escritos en C y C++, entre otros.

SQL

SQL es un lenguaje distinto a los comentados hasta ahora, ya que está optimizado para una clase específica de aplicación (consultas de base de datos). Debido al hecho de que las bases de datos a menudo son accedidas por más de un cliente con posibilidad de acceso concurrente, SQL incluye un mecanismo para tratar la concurrencia. En SQL, la concurrencia se aborda mediante el uso de transacciones. El uso de transacciones permite que los bloques de código SQL se ejecuten especulativamente sin modificar los contenidos de la base de datos directamente. Los cambios realizados dentro de una transacción se escriben en un registro que está asociado con la transacción. Cuando se completa el bloque de SQL dentro de la transacción, la transacción se termina con una declaración de confirmación (COMMIT). Esta declaración intenta confirmar, o escribir, los resultados de la transacción en la base de datos para el almacenamiento permanente.

El poder de una transacción es que los bloques de código SQL ejecutados dentro de la transacción lo hacen sin adquirir bloqueos, lo que limita el grado de bloqueo de otros clientes durante su ejecución. Cuando la transacción está terminada y lista para comprometerse, es posible ver si otro cliente permite alguna operación que haga que

los resultados de la transacción sean inválidos. Si esto ocurre, la transacción puede abortarse limpiamente sin que sea necesario reparar la base de datos debido al uso del registro de transacciones durante su ejecución.

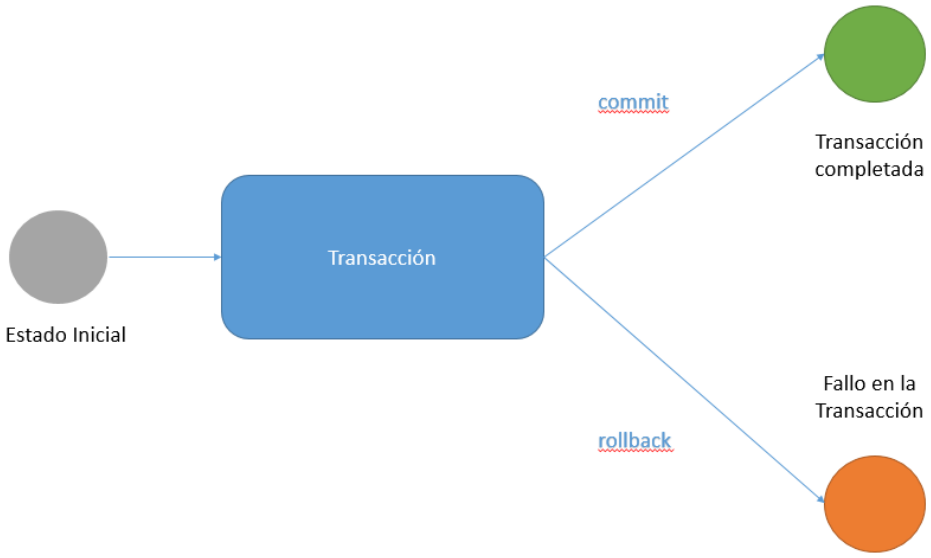

Figura 27.. Esquema funcionamiento de una transacción[30]

Los siguientes comandos se utilizan para controlar las transacciones en SQL:

- *COMMIT*: guardar los cambios de una transacción.
- *ROLLBACK*: deshace los cambios de una transacción.
- *SAVEPOINT*: crea puntos dentro de los grupos de transacciones en los que poder hacer un *rollback*.

3.3.2. Herramientas de programación paralela

Una vez que se han visto los principales lenguajes de programación y cómo pueden implementar soluciones paralelas, hay que detallar la existencia de distintas librerías, frameworks, api's y herramientas que pueden ser empleadas en distintos lenguajes para implementar código que se ejecute de forma paralela. En los siguientes subapartados se describen algunas de las más importantes.

3.3.2.1. POSIX Threads.

Según Butenhof (1997), los "POSIX es, técnicamente, la API especificado por el internacional estándar formal POSIX 1003.1c-1995". Esta norma fue aprobada por el IEEE en junio de 1995. Una nueva edición de POSIX 1003.1. llamado ISO / IEC 9945-1: 1996 (ANSI /IEEE Std 1003.1, Edición de 1996) está disponible en el IEEE. Este nuevo

[30] Elaboración propia

documento incluye 1003. lb-1993 (tiempo real). 1003. lc -1995 (hilos), y 1003.1i-1995 (correcciones a 1003. lb -1993).

Los subprocesos POSIX (Pthreads en adelante) es un estándar para la programación con subprocesos, y define un conjunto de tipos, funciones y constantes al estilo C.

De forma más general, los subprocesos son una forma en que un programa puede generar unidades de procesamiento concurrentes que luego pueden ser delegadas por el sistema operativo a múltiples núcleos de procesamiento. Claramente, la ventaja de un programa multiproceso (uno que usa múltiples subprocesos que están asignados a múltiples núcleos de procesamiento) es que puede lograr grandes aceleraciones, ya que todos los núcleos de su CPU (y todos los de la CPU si tiene más de uno) se usan al mismo tiempo.

Los tres aspectos esenciales de un sistema de *threads* son: el contexto de ejecución, la programación y la sincronización. Cuando se evalúa un sistema de este tipo, o se comparan dos sistemas, hay que categorizar las características en capacidades que soportan contextos de ejecución, programación y sincronización.

Con Pthreads, se crea un contexto de ejecución llamando a *pthread_create*. La creación de un hilo también programa el hilo para su ejecución, y lo hará llamando a una "función de inicio de hilo" que se especificó. Pthreads permite especificar los parámetros de programación al momento de crear el subproceso o más tarde, cuando el hilo se está ejecutando. Un hilo normalmente termina cuando se llama a *pthread_exit,* o devuelve desde la función de inicio de hilo, aunque existen otras posiblidades.

El modelo de sincronización de Pthreads básico utiliza *mutexes* para protección y variables de condición para la comunicación. También se pueden usar otros mecanismos de sincronización como semáforos, tuberías y colas de mensajes. Un *mutex* permite a un hilo bloquear los datos compartidos mientras se usa, para que otros hilos no puedan interferir accidentalmente. Una variable de condición permite que un hilo espere a que se compartan los datos compartidos para alcanzar algún estado deseado (como "cola no vacía" o "recurso disponible").

Los distintos tipos de Pthreads que existen son:

Type	Description
pthread_t	thread identifier
pthread_mutex_t	mutex
pthread_cond_t	condition variable
pthread_key_t	"access key" for thread-specific data
pthread_attr_t	thread attributes object
pthread_mutexattr_t	mutex attributes object
pthread_condattr_t	condition variable attributes object
pthread_once_t	"one time initialization" control context

Figura 28. Tipos de Pthreads[31]

3.3.2.2. MPI

El estándar de interfaz de paso de mensajes (MPI) es una librería estándar de paso de mensajes basada en el consenso del Foro MPI[32], que cuenta con más de 40 organizaciones participantes, incluidos proveedores, investigadores, desarrolladores de bibliotecas de software y usuarios. El objetivo de la interfaz de paso de mensajes es establecer un estándar portátil, eficiente y flexible para el paso de mensajes que se utilizará ampliamente para escribir programas de paso de mensajes. Como tal, MPI es la primera biblioteca de paso de mensajes estandarizada, independiente del proveedor. Las ventajas de desarrollar software de paso de mensajes utilizando MPI coinciden con los objetivos de diseño de portabilidad, eficiencia y flexibilidad. MPI no es un estándar IEEE o ISO, pero, de hecho, se ha convertido en el "estándar de la industria" para escribir programas de paso de mensajes en plataformas HPC, según Barney (2019).

MPI es una especificación para los desarrolladores y usuarios de bibliotecas de paso de mensajes. Por sí misma, no es una biblioteca, sino la especificación de lo que debería ser esa biblioteca. Aborda principalmente el modelo de programación paralela de paso de mensajes: los datos se mueven del espacio de direcciones de un proceso al de otro proceso a través de operaciones cooperativas en cada proceso. Dicho de forma simple, el objetivo de la interfaz de paso de mensajes es proporcionar un estándar ampliamente utilizado para escribir programas de paso de mensajes. La interfaz intenta ser: práctica, portátil, eficiente y flexible.

El estándar MPI ha pasado por varias revisiones, siendo la versión más reciente MPI-3.x Las especificaciones de la interfaz se han definido para los enlaces de lenguaje C y Fortran90: os enlaces C ++ de MPI-1 se eliminan en MPI-3 y también proporciona soporte para las características de Fortran 2003 y 2008.

[31] Programming with POSIX Threads. David R. Butenhof
[32] https://www.mpi-forum.org/

Actualmente se están recogiendo propuestas de ideas para la nueva versión MPI 4.0, y desde el foro de trabajo se está trabajando y proponiendo, de forma especial, las siguientes características:

- Extensiones para soportar mejor los modelos de programación híbridos.
- Soporte para tolerancia a fallas en aplicaciones MPI.
- Colecciones persistentes
- Comunicación unilateral RMA

Las implementaciones reales de la biblioteca MPI difieren en qué versión y características del estándar MPI son compatibles. Los desarrolladores deberán estar conscientes de esto.

Las principales razones por las que usar MPI son:

- **Estandarización**: MPI es la única biblioteca de paso de mensajes que puede considerarse estándar. Es compatible con prácticamente todas las plataformas HPC. Prácticamente, ha reemplazado todas las bibliotecas de paso de mensajes anteriores.
- **Portabilidad**: hay poca o ninguna necesidad de modificar su código fuente cuando transfiere su aplicación a una plataforma diferente que admite (y cumple con) el estándar MPI.
- **Rendimiento**: las implementaciones de los proveedores deben poder explotar las características de hardware nativas para optimizar el rendimiento. Cualquier implementación es libre de desarrollar algoritmos optimizados.
- **Funcionalidad**: hay más de 430 rutinas definidas en MPI-3, que incluye la mayoría de las que están en MPI-2 y MPI-1. La mayoría de los programas MPI se pueden escribir con una docena o menos de rutinas.
- **Disponibilidad**: una gran variedad de implementaciones está disponible, tanto para proveedores como para dominio público.

Existen distintas implementaciones del estándar MPI, que pueden diferir entre ellos. Dependiendo de distintas cuestiones como qué versión de MPI soporta, si están todas las funcionalidades, si se añaden las nuevas funcionalidades, cómo compilar o cómo lanzar aplicaciones, se debe elegir entre una u otra implementación. A continuación, se van a indicar las implementaciones más conocidas de MPI:

- *MVAPICH*: está desarrollada por el Network-Basec Computing Lab de la Universidad Estatal de Ohio. Está disponible en todos los clústeres Linux de LC. Existe una versión, MVAPICH2 que implementa MPI por defecto y realizan *threading* de forma segura.
- *OPENMPI*: es una implementación MPI de código abierto, segura para subprocesos, desarrollada y respaldada por un consorcio de socios académicos,

de investigación y de la industria. También está disponible en todos los clústers de LC Linux. Sin embargo, primero hay que cargar el módulo deseado.

3.3.2.3. OpenMP

Según Román (2018), "OpenMP es un Sistema de programación de memoria compartida basado en directivas de compilación, es decir, el programador inserta directivas en el código secuencial para indicar al compilador cómo ha de realizar la paralelización, además de incorporar algunas llamadas a funciones específicas".

OpenMP es una interfaz de programación de aplicaciones (API) estándar de la industria para la programación paralela. El lenguaje, creado a finales de los años 90, ha evolucionado continuamente y actualmente se encuentra en el versión 5.0[33].

Esta API implementa una forma de paralelismo de memoria compartida de múltiples subprocesos. Utiliza un conjunto de directivas de compilación (estas declaraciones se agregan al código fuente origianl) que se incorporan en tiempo de compilación para generar una versión de código múltiple de su subproceso. A difrenciar de Pthreads que hace la programación de múltiples subprocesos de forma "manual", OpenMP, al ser una API de nivel superior, automatiza ese proceso. OpenMP se encarga de muchos de los detalles de bajo nivel que normalmente el desarrollador tendría que implementar si utilizase PThreads.

La API consta, tal y como describe Barney (2019), consta de tres componentes principales:

- **Directivas del compilador**: aparecen como comentarios en el código fuente y son ignoradas por los compiladores a menos que se indique lo contrario, generalmente especificando el indicador de compilador apropiado. Las directivas de compilador tienen la siguiente sintaxis:

sentinel directive-name [clause, ...]

For example:

Fortran	`!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)`
C/C++	`#pragma omp parallel default(shared) private(beta,pi)`

Figura 29. Sintaxis directivas compilador en OpenMP[34]

- **Rutinas de la librería de tiempo de ejecución**: estas rutinas se utilizan para una variedad de propósitos: configuración y consulta del número de hilos; consultar el identificador único de un hilo (ID de hilo), el identificador del padre de un hilo,

[33] https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
[34] https://computing.llnl.gov/tutorials/openMP

el tamaño del equipo de hilo; configuración y consulta de la función de subprocesos dinámicos; preguntar si en una región paralela, y en qué nivel; configuración y consulta de paralelismo anidado; configuración, inicialización y terminación de bloqueos y bloqueos anidados; consultar tiempo y resolución de reloj de pared.

- **Variables de entorno**: proporciona varias variables de entorno para controlar la ejecución de código paralelo en tiempo de ejecución. Estas variables de entorno se pueden usar para controlar cosas como: configuración del número de hilos; especificar cómo se dividen las interacciones de bucle; enlazar hilos a procesadores; habilitar y deshabilitar el paralelismo anidado, configurando de los niveles máximos de paralelismo anidado; habilitar y deshabilitar hilos dinámicos; configurar el tamaño de pila de hilos y configurar la política de espera de hilo. La configuración de las variables de entorno de OpenMP se realiza de la misma manera que establece otras variables de entorno, y depende de qué shell utilice.

csh/tcsh	`setenv OMP_NUM_THREADS 8`
sh/bash	`export OMP_NUM_THREADS=8`

Figura 30. Variables de entorno en OpenMP[35]

En la especificación de la última versión disponible, la 5.0 de 2018, se detallan los distintos conceptos y elementos de OpenMP. En este documento se explica que esta API usa el model fork-join de ejecución paralela.

3.3.2.4. CUDA

La definición de CUDA, recogida en la web de NVidia, nos dice que es una plataforma de computación paralela y un modelo de programación desarrollado por NVIDIA para computación general en unidades de procesamiento gráfico (GPU). Con CUDA, los desarrolladores pueden acelerar drásticamente las aplicaciones informáticas aprovechando el poder de las GPU. El nombre viene de las siglas de *Compute Unified Device Architecture.*

En 2007, NVIDIA vio la oportunidad de incorporar las GPU a la corriente principal al agregar una herramienta fácil de usar, una API, que llamó CUDA. Si nos fijamos en la potencia computacional relativa entre GPU y CPU, se puede ver una distanciación en la potencia computacional a partir de 2009, cuando las GPU superan la barrera de los 1000 Gigaflops, como se ve en la siguiente figura, según Cook (2013). Esto es impulsado por la introducción de hardware masivamente paralelo.

[35] https://computing.llnl.gov/tutorials/openMP

Vemos que las GPU NVIDIA dan un salto de 300 gigaflops desde la arquitectura G200 a Fermi, casi un 30% de mejora en el rendimiento. En comparación, el salto de Intel desde su arquitectura de 2 núcleos, la arquitectura Nehalem ve solo una pequeña mejora. Las GPU no están diseñadas para el flujo de ejecución en serie y solo logran su máximo rendimiento cuando se utiliza plenamente de forma paralela.

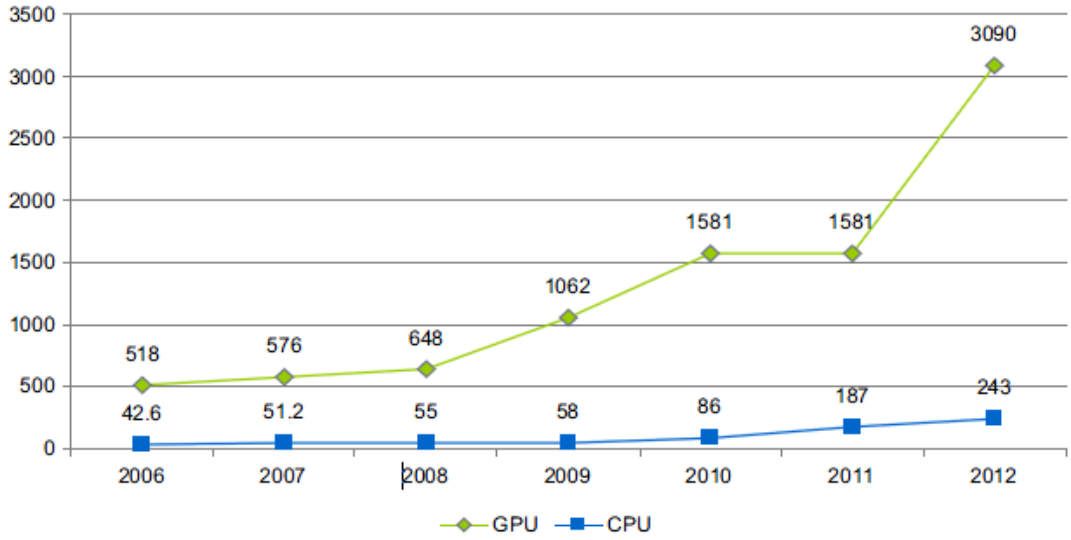

Figura 31. Rendimiento máximo de CPU y GPU en GigaFlops[36]

En las aplicaciones aceleradas por GPU, la parte secuencial de la carga de trabajo se ejecuta en la CPU, que está optimizada para el rendimiento de un solo subproceso, mientras que la parte de uso intensivo de cómputo se ejecuta en miles de núcleos de GPU en paralelo. Cuando se utiliza CUDA, los desarrolladores programan en lenguajes populares como C, C ++, Fortran, Python y MATLAB y expresan el paralelismo a través de extensiones en forma de algunas palabras clave básicas.

El kit de herramientas CUDA de NVIDIA ofrece todo lo que necesita para desarrollar aplicaciones aceleradas por GPU. El kit de herramientas CUDA incluye bibliotecas aceleradas por GPU, un compilador, herramientas de desarrollo y el tiempo de ejecución CUDA.

CUDA se puede utilizar en conjunto tanto con OpenMP como con MPI. También hay una versión de directiva similar a OpenMP de CUDA (OpenACC) eso puede ser algo más fácil para aquellos familiarizados con OpenMP.

[36] Shane Cook. Cuda Programming. A developer's guide to parallel computing with GPUs

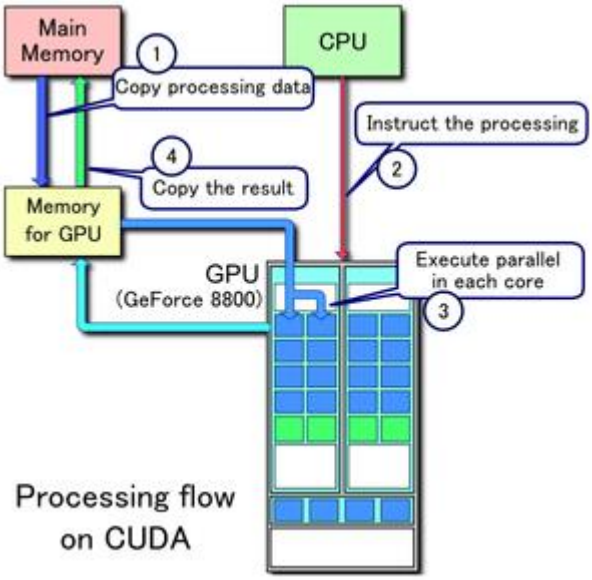

Figura 32. Flujo de procesamiento en CUDA.[37]

Tener un gran ancho de banda e las GPU permite que CUDA optimice el paralelismo en aplicaciones que requieran de un gran coste computacional, al tener que realizar muchos accesos a la memoria, lo que podría provocar un cuello de botella.

El modelo de programación de CUDA se basa en tres elementos clave: la jerarquía de grupos de hilos, las memorias compartidas y las barreras de sincronización. Para estructurar este modelo se define un grid, en el cual existen una serie de bloques de hilos, con un máximo de 512. Éstos, están identificados de manera única, y se accede con una variable que lo identifica. Esta variable es la que permite repartir el trabajo entre diferentes hilos. Tiene tres elementos, que se ajustan a las dimensiones de bloques de hilos. Al igual que los hilos, los bloques se identifican mediante *blockIdx* (en este caso con dos componentes x e y). Otro parámetro útil es *blockDim*, para acceder al tamaño de bloque.

[37] https://es.wikipedia.org/wiki/CUDA

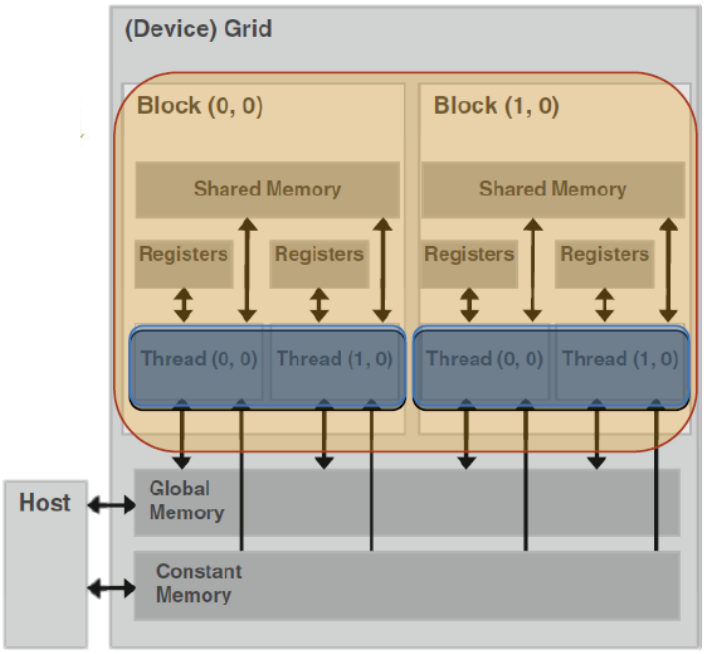

Figura 33. Grid de bloques de hilos[38].

Un ejemplo de la potencia de CUDA mediante las funciones *kernel*. Imaginemos un trozo de código que ejecute un bucle X veces. Este bucle itera esas X veces. Con CUDA, se puede trasladar a X hilos, donde cada uno ejecuta una iteración. Para ello, se crea una función *kernel*, que es una función que se ejecuta sólo en la GPU y no se puede ejecutar directamente en la CPU.

```
1   #Código original que ejecuta un bucle 128 veces
2   void some_func(void){
3       int i;
4
5       for (i=0;i<128;i++){
6           a[i] = b[i] * c[i];
7       }
8   }
9
10
11  #Código con CUDA que crea una función kernel para cada iteración
12  __global__ void some_kernel_func(int * const a, const int * const b, const int * const c)
13  {
14  a[i] = b[i] * c[i];
15  }
```

Figura 34. Ejecución secuencial y declaración de función kernel en CUDA[39]

La función del *kernel* de la GPU, conceptualmente, parece idéntica al cuerpo del bucle, pero con la estructura del bucle eliminada. Se puede observar que se ha perdido el bucle y la variable de control del mismo, i. Además, existe un prefijo *__global__* añadido a la función, que le indica al compilador que genere código de GPU y no código de CPU al compilar esta función, y que haga que el código de GPU sea visible globalmente desde dentro de la CPU. La CPU y la GPU tienen espacios de memoria

[38] Cuda Zone. Nvidia Web.
[39] Shane Cook. Cuda Programming. A developer's guide to parallel computing with GPUs

separados, lo que significa que no puede acceder a los parámetros de la CPU en el código de la GPU y viceversa. Como consecuencia, las matrices globales a, b y c en el nivel de la CPU ya no son visibles en el nivel de GPU. Hay que declarar un espacio de memoria en la GPU, copiar las matrices desde la CPU y pasar los punteros de función del kernel al espacio de memoria de la GPU para poder leer y escribir.

3.3.2.5. OpenCL

El objetivo de OpenCL, según Tay (2013), es desarrollar un estándar multiplataforma, para la programación paralela de procesadores modernos que se encuentran en las ordenadores personales, servidores, y dispositivos móviles. Este esfuerzo lo realiza "The Khronos Group", con la participación de empresas como Intel, ARM, AMD, NVIDIA, QUALCOMM, Apple y muchos otros. OpenCL permite que el software se escriba una vez y luego se ejecute en los dispositivos que lo soporten. De esta manera es similar a Java, esto tiene beneficios porque el desarrollo de software en estos dispositivos ahora tiene un enfoque uniforme, y OpenCL hace esto exponiendo el hardware a través de varias estructuras de datos, y estas estructuras interactúan con el hardware a través de APIs. Hoy en día, OpenCL es compatible con CPU que incluye x86s, ARM y PowerPC y GPU de AMD, Intel y NVIDIA.

Cuando los desarrolladores están implementando una solución para OpenCL, necesitarían descomponer un problema en diferentes subtareas, que se pueden ejecutar simultáneamente, asignándolas a unidades de procesamiento en un entorno paralelo de ejecución. Por otro lado, hay tareas que no se pueden ejecutar de manera concurrente e incluso posiblemente interdependientes.

Para entender bien el funcionamiento de OpenCL, es necesario profundizar y conocer los componentes que forma su arquitectura:

- Modelo de plataforma: una plataforma es en realidad un host que está conectado a uno o más dispositivos OpenCL. Cada dispositivo comprende posiblemente unidades de cálculo múltiples (CU) que puede descomponerse en uno o posiblemente múltiples elementos de procesamiento, y es en los elementos de procesamiento donde se ejecutará el cálculo.

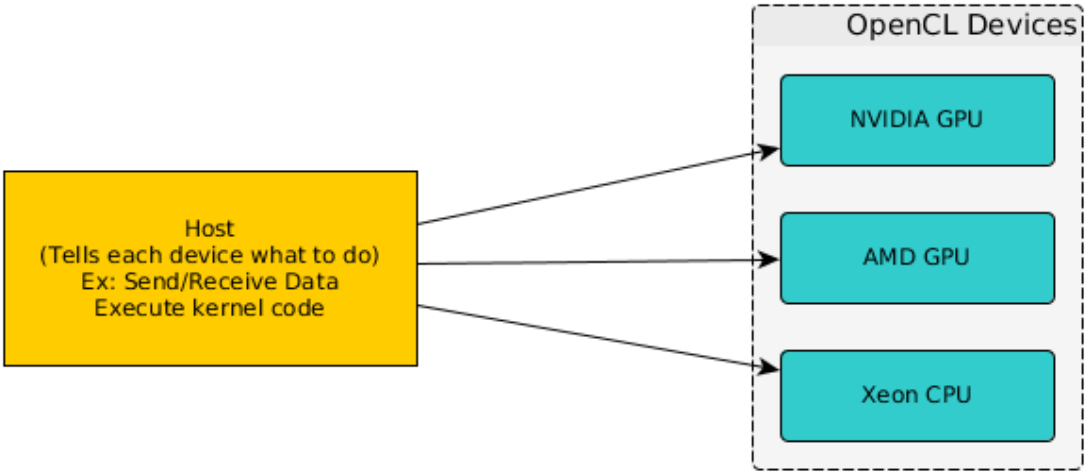

Figura 35. Esquema OpenCL[40]

- Modelo de ejecución: la ejecución de un programa OpenCL es tal que el programa anfitrión se ejecutaría en el host, y es el programa host el que envía los *kernels* para ejecutarse en uno o más dispositivos OpenCL en esa plataforma. Cuando se envía un *kernel* para su ejecución, se define un espacio de índice tal que un el elemento de trabajo se crea una instancia para ejecutar cada punto en ese espacio. Un elemento de trabajo sería identificado por su ID global y ejecuta el mismo código como se expresa en el núcleo. Los elementos de trabajo se agrupan en grupos de trabajo y cada grupo de trabajo recibe una identificación, comúnmente conocida como su ID de grupo de trabajo, y son los elementos de trabajo del grupo de trabajo los que son ejecutados concurrentemente en los PE de una sola UC.

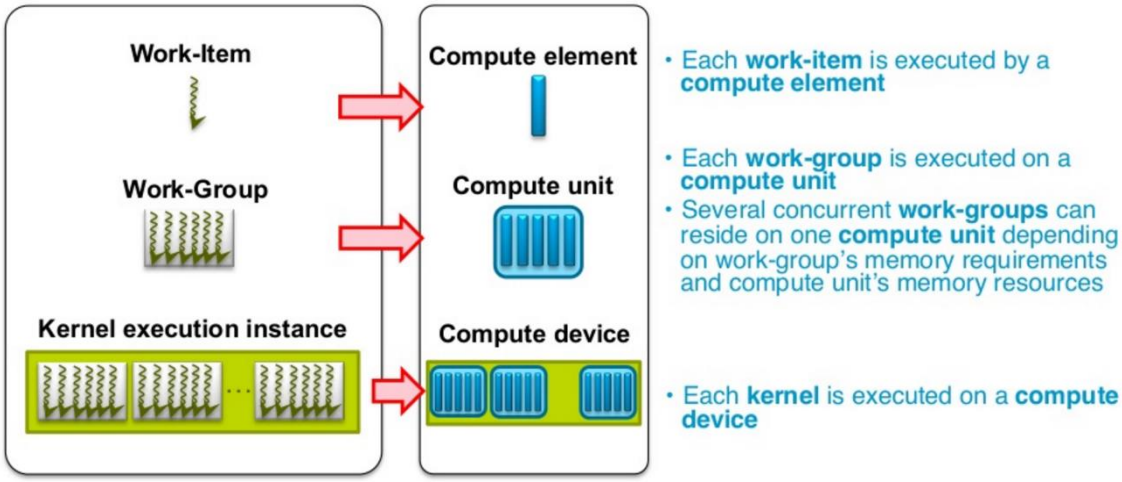

Figura 36. Esquema modelo de ejecución[41].

[40] https://leonardoaraujosantos.gitbooks.io/opencl/chapter1.html
[41] https://leonardoaraujosantos.gitbooks.io/opencl/chapter1.html

- Modelo de memoria: cuando el kernel se ejecuta, en realidad es el elemento de trabajo que está ejecutando su instancia del código kernel. Por lo tanto, el elemento de trabajo necesita leer y escribir los datos de la memoria y cada elemento de trabajo tiene acceso a cuatro tipos de memorias: global, constante, local y privada. Estas memorias varían de tamaño y accesibilidades, donde la memoria global tiene el tamaño más grande y es más accesible para los elementos de trabajo, mientras que la memoria privada es posiblemente la más restrictiva en el sentido de que es privada para el elemento de trabajo. La memoria constante es una memoria de solo lectura en la que se almacenan objetos inmutables y puede ser compartida con todos los elementos de trabajo. La memoria local solo está disponible para todos los elementos de trabajo que se ejecutan en el grupo de trabajo y se lleva a cabo por cada unidad de cómputo.

3.3.3. Algoritmos de programación paralela. Patrones de diseño

A la hora de desarrollar programas con partes paralelas, es conveniente conocer los distintos patrones de diseño que se adaptan a esta situación. Según Sottile et al. (2010), los principales patrones de diseño paralelo son los relacionados con lo siguiente: paralelismo de tarea (*task parallelism*), paralelismo de datos (*data parallelism*), paralelismo recursivo (*recursive parallelism*) y *pipelining*. En este apartado se van a detallar brevemente cada uno de estos patrones de diseño para implementaciones paralelas.

3.3.3.1. Task Parallelism

El patrón de paralelismo de tareas es uno de los patrones de diseño más comunes en computación en paralelo con aplicaciones que incluyen ray-tracing, muestreo de grandes parámetros en simulaciones, o procesamiento de grandes colecciones de datos distintos.

El paralelismo de tareas está muy cerca de la forma en que alcanzamos la concurrencia en el mundo real. Si tenemos, por ejemplo, diez mil piezas que revisar, podemos contratar 100 trabajadores para que cada uno revise 100 piezas. Si todos los trabajadores son igual de eficientes y mantenemos bajos los gastos generales, podemos esperar acelerar el proceso 100 veces. Este es un ejemplo clásico del patrón paralelo de tareas.

Este patrón se basa en la tarea. Una tarea es una secuencia de sentencias conectadas lógicamente en un programa. Lo complicado es definir las tareas y administrarlas para que se ejecuten de manera efectiva y el programa que las agrupa, de forma correcta y eficiente. Los elementos esenciales de la solución a un problema de paralelismo de tareas son:

- Definir el conjunto de tareas que conforman nuestro algoritmo.

- Gestionar dependencias entre las tareas.
- Planificar su ejecución para que la carga computacional se equilibre uniformemente entre los elementos de procesamiento.
- Detectar la condición de finalización y parar el cálculo.

Cuando la tarea es completamente independiente, el problema se llama *embarrassingly parallel*. En este caso, el programador simplemente define las tareas y las programa para su ejecución. Las técnicas son bien conocidas por crear algoritmos que equilibran automáticamente la carga entre los elementos de procesamiento para problemas "embarazosamente paralelos".

Cuando existen las dependencias, la aplicación de este patrón es más complicada. A menudo, las dependencias son separables y pueden ser extraídas de las tareas convirtiendo el problema en un problema independiente. Un método común para tratar las dependencias separables es usar la replicación de datos seguida por un proceso de recombinación global. Una vez que todas las tareas están completas, las copias locales de las estructuras de datos se combinan para producir el resultado final. En esencia, el problema se convierte en un patrón "*embarrassingly parallel*".

Las aplicaciones más complejas del patrón paralelo de tareas se combinan cuando las dependencias no son separables. En tales casos, las actualizaciones simultáneas deben gestionarse explícitamente mediante un patrón de estructura de datos compartidos o mediante protocolos explícitos de exclusión mutua. A medida que las dependencias se vuelven más complejas, el patrón se vuelve más difícil de aplicar y, en algún punto, un patrón paralelo de datos o descomposición geométrica se convierte en un patrón más natural para trabajar.

3.3.3.2. Data Parallelism

Los algoritmos paralelos fundamentalmente definen tareas que se ejecutan concurrentemente. En algunos problemas, la forma más natural de expresas esas tareas es en términos de datos que son modificados en el transcurso de la ejecución. En estos problemas, los datos son descompuestos en componentes relativamente independientes y las tareas son definidas como actualizaciones concurrentes sobre esos datos.

El término "*data parallelism*" describe un amplio rango de algoritmos paralelos. En un extremo, los datos son definidos en términos de arrays que son distribuidos sobre sistemas paralelos y las tareas son definidas como aplicaciones concurrentes con un único flujo de instrucciones que trabajan sobre los elementos de los arrays. En otros casos, los datos son descompuestos en grandes bloques de estructuras más complejas y las tareas son definidas como actualizaciones sobre esos bloques con variaciones en el procesamiento de un bloque a otro.

El tema común en todos los algoritmos paralelos de datos es un espacio de índice abstracto que se define y distribuye entre los elementos computacionales dentro de un sistema. Los datos se alinean con este espacio de índice y se producen las tareas como una secuencia de instrucciones para cada punto en el espacio de índice.

El caso de estudio que mejor refleja este tipo de paralelismos es el de la multiplicación de matrices. Si tenemos dos matrices, A y B de dimensiones MxN y NxP, la matriz resultante de la multiplicación es una matriz C de dimensiones MxP, que resulta de multiplicar dos vectores, uno de cada matriz. En una ejecución secuencial, esto conllevaría a utilizar un bucle triple, con la consecuente pérdida de eficiencia conforme las matrices sean más grandes.

Una aproximación, detallada por Gergel (2012), a la resolución de este problema es la división en bloques de filas y columnas. La matriz A se descompone en filas y la matriz B en columnas. Cada subtarea contiene una fila de matriz A y una columna de la matriz B. En cada iteración, cada subtarea realiza el cálculo del producto interno de su fila y columna, dando como resultado el elemento correspondiente en la matriz C. Luego, cada subtarea i, 0<= i < n (n es número de subtareas), transmite su columna de matriz B para la subtarea con el número (i + 1) mod n. Después de todas las iteraciones del algoritmo, todas las columnas de la matriz B llegan desde cada subtarea, una tras otra. En caso de que el número de procesadores p sea menor que el número de subtareas básicas n, los cálculos se pueden agregar de tal manera que ada procesador ejecutaría varios productos internos de las filas de la matriz A y matrices de columnas B. En este caso después de la finalización del cálculo, cada subtarea básica agregada determina varias filas de la matriz de resultados C. En tales condiciones, la matriz inicial A se descompone en p bloques horizontales, filas, y la matriz B se descompone en p bloques verticales, columnas.

Matrix Size	2 processors		4 processors		8 processors	
	Model	Experiment	Model	Experiment	Model	Experiment
500	0,8243	0,3758	0,4313	0,1535	0,2353	0,0968
1000	6,51822	5,4427	3,3349	2,2628	1,7436	0,6998
1500	21,9137	20,9503	11,1270	11,0804	5,7340	5,1766
2000	51,8429	45,7436	26,2236	21,6001	13,4144	9,4127
2500	101,1377	99,5097	51,0408	56,9203	25,9928	18,3303
3000	174,6301	171,9232	87,9946	111,9642	44,6772	45,5482

Figura 37. Comparativa de resultados estimados para la resolución del problema de multiplicación de matrices.[42]

3.3.3.3. Recursive Parallelism

Este patrón de diseño distribuye los datos de forma que éstos presentan alguna característica de estructura de datos recursiva, como, por ejemplo, los grafos o los árboles.

[42] https://homes.cs.washington.edu/~djg/msr_russia2012/gergel3.pdf

La recursividad es uno de las técnicas algorítmicas fundamentales en el mundo de la informática. El concepto básico de recursividad es que una función se llama a sí misma. La evaluación de la función recursiva se crea al componer los resultados de las llamadas a sí mismo, utilizando refinamientos sucesivos de sus datos de entrada en cada paso. Este refinamiento puede abarcar desde dividir los datos en partes más pequeñas, atravesar una estructura de datos similar como una lista vinculada, o realizar un cálculo numérico en las entradas. Finalmente, la función debe tener una condición para detener el proceso repetido de llamada para que las llamadas puedan volver a la función que la llamó de manera original con el resultado final. El ejemplo clásico es el de calcular los números de Fibonnaci.

La recursividad, como todas las invocaciones de funciones, está íntimamente ligada a la pila de registros de activación. En el caso de funciones recursivas que se ejecutan de forma concurrente, existe el problema de la situación en la pila de ejecución. Para resolver esta problemática existen dos aproximaciones, la primera se basa en la generación de nuevas áreas para cada llamada, y la segunda se basa en la redefinición de la estructura de la pila.

Además de esta problemática con la pila de ejecución para administrar las llamadas a las funciones, también hay que considerar otro problema que surge cuando se usa la concurrencia en algoritmos recursivos, y es, la modificación de algún estado compartido por alguna de las funciones que son llamadas. Si esto ocurre, se debe asegurar que los datos estén protegidos mediante primitivas de control de concurrencia para evitar que estas operaciones entre en conflicto entre sí. Esta necesidad de control de concurrencia no es diferente a otros contextos de programación.

Los algoritmos "*divide y vencerás"* son muy comunes en la resolución de problemas. La premisa básica de este enfoque es que un problema difícil se puede resolver al dividirlo en subproblemas más simples que se pueden resolver de manera independiente, y sus resultados se combinan para forma la solución al problema original. Hay una adaptación natural a la computación concurrente aquí, ya que la independencia de los subproblemas implica que puedan resolverse al mismo tiempo sin conflicto.

Diferentes problemas que pueden ser resueltos mediante el diseño de algoritmos siguiente este patrón son los algoritmos de ordenación como *mergesort* o *quicksort*, o la resolución de sudokus. Ambos problemas se adaptan a las características deseadas de división en tareas más pequeñas que pueden ser resueltas de forma concurrente e independiente.

3.3.3.4. Pipeline

El diseño *pipeline* es una traducción directa del concepto de la línea de ensamblaje al software. Los elementos clave de la solución son:

- El problema original se descompone en varias etapas especializadas.
- Las etapas están conectadas por un flujo de datos. Por lo general, este flujo es una "tubería" recta, pero se puede utilizar la entrada en abanico (varias etapas que se alimentan en una sola etapa) y la salida en abanico (una etapa que alimenta varias etapas).
- Cada entrada de elemento de datos al pipeline genera un patrón de flujo de datos que pasa a través de todas las etapas en un orden fijo. Los datos fluyen en un extremo del pipeline y fluyen hacia el otro extremo, al igual que con una línea de ensamblaje de fabricación.
- Aparte de los datos para controlar el flujo del cálculo, las etapas son sin estado, es decir, el estado del cálculo está contenido en los datos que fluyen entre las etapas

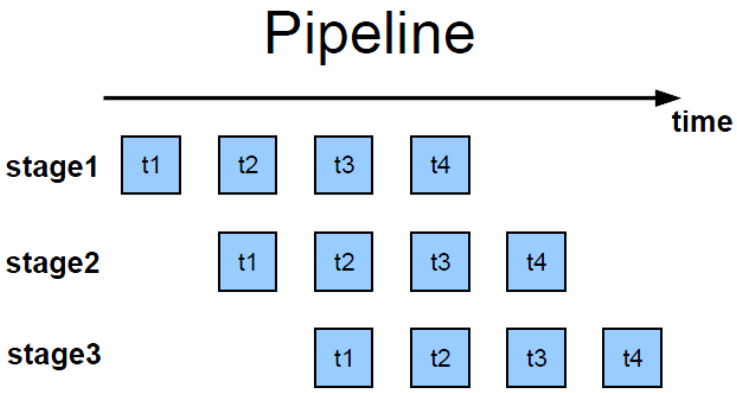

Figura 38. Esquema pipeline[43]

3.3.3.5. Estructuras soporte

Una vez que hemos detallado los distintos patrones de diseño que se adaptan al paradigma de la programación paralela, se van a comentar las estructuras que dan soporte a estos patrones de diseño:

- *Master-worker*: un *master* gestiona una colección de tareas y programa la ejecución en una colección de *workers.* Los *workers* solicitan más tareas conforme van terminando las que tienen asignadas, dando un balanceo de carga automático entre los *workers.*
- *Loop-level:* las colecciones de tareas se definen como iteraciones de uno o más bucles. Los bucles están divididos en una colección de elementos para procesar

[43] Patterns for Parallel Programming. Mattson, T. 2014

las tareas de forma concurrente. Este patrón de diseño también se usa mucho con los patrones de diseño de datos paralelos.

- *Fork-join*: esta estructura se ha visto en el lenguaje de programación Java, y se basa en dividir las tareas en diferentes hilos de ejecución.

La siguiente tabla muestra la relación entre el algoritmo de soporte y la estrategia de paralelismo:

	Task Paralellism	Divide and Conquer	Geometric Decomposition	Recursive Data	Pipeline	Event-Based Coordination
SPMD	****	***	****	**	***	**
Loop Parallelism	****	**	***			
Master/Worker	****	**	*	*	*	*
Fork/Join	**	****	**		****	****

Figura 39. Algoritmos de soporte y estrategias de paralelismo.[44]

3.3.4. Problemática en la concurrencia

Al diseñar programas que se ejecutarán simultáneamente, o librerías que serán utilizadas por programas potencialmente concurrentes, se debe considerar el impacto que esto tendrá en que el código sea correcto. Los problemas de corrección aquí son aquellos que deben considerarse cada vez que se está diseñando un programa concurrente. A diferencia de los programas secuenciales en los que muchos errores frecuentes se deben a errores en las condiciones de flujo de control o a una lógica errónea, los errores en los programas concurrentes surgen de un código que puede parecer correcto en un sentido secuencial, pero conduce a resultados incorrectos debido a las interacciones entre hilos de ejecución paralelos.

3.3.4.1. Race Condition

Una *race condition* o “condición de carrera" se puede definir como "un comportamiento anómalo debido a una dependencia crítica inesperada en el tiempo relativo de los eventos" según Bosworth (2014). Las condiciones de carrera generalmente involucran uno o más procesos que acceden a un recurso compartido (como un archivo o variable), donde este acceso múltiple no se ha controlado adecuadamente.

De manera más ejemplarizante, se puede decir que este tipo de problemas surgen cuando varias tareas quieren leer y escribir un dato cuyo resultado depende del orden en cómo se han ejecutado. Para resolver esta problemática hay que implementar mecanismos que sincronicen los accesos a esos datos.

En general, los procesos no se ejecutan atómicamente; otro proceso puede interrumpirlo esencialmente entre dos instrucciones. Si el proceso de un programa seguro no está preparado para estas interrupciones, otro proceso puede interferir con

[44] Introducción a las arquitecturas paralelas. Daniel Jiménez-González. UOC.

el proceso del programa seguro. Cualquier par de operaciones en un programa seguro aún debe funcionar correctamente si se ejecutan entre ellas cantidades arbitrarias de código de otro proceso.

Los problemas del tipo *race condition* se pueden dividir teóricamente en dos categorías:

- Interferencia causada por procesos no confiables. Algunas taxonomías de seguridad llaman a este problema condición `` secuencia '' o `` no atómica ''. Estas son condiciones causadas por procesos que ejecutan otros programas diferentes, que deslizan otras acciones entre los pasos del programa seguro. Estos otros programas pueden ser invocados por un atacante específicamente para causar el problema. Este libro llamará a estos problemas de secuenciación.
- Interferencia causada por procesos confiables (desde el punto de vista del programa seguro). Algunas taxonomías llaman a estas condiciones de bloqueo, bloqueo o bloqueo. Estas son condiciones causadas por procesos que ejecutan el programa `` mismo ''. Dado que estos diferentes procesos pueden tener los privilegios "iguales", si no se controlan adecuadamente, pueden interferir entre sí de una manera que otros programas no pueden. A veces este tipo de interferencia puede ser explotada. A estos problemas se les suele llamar problemas de bloque.

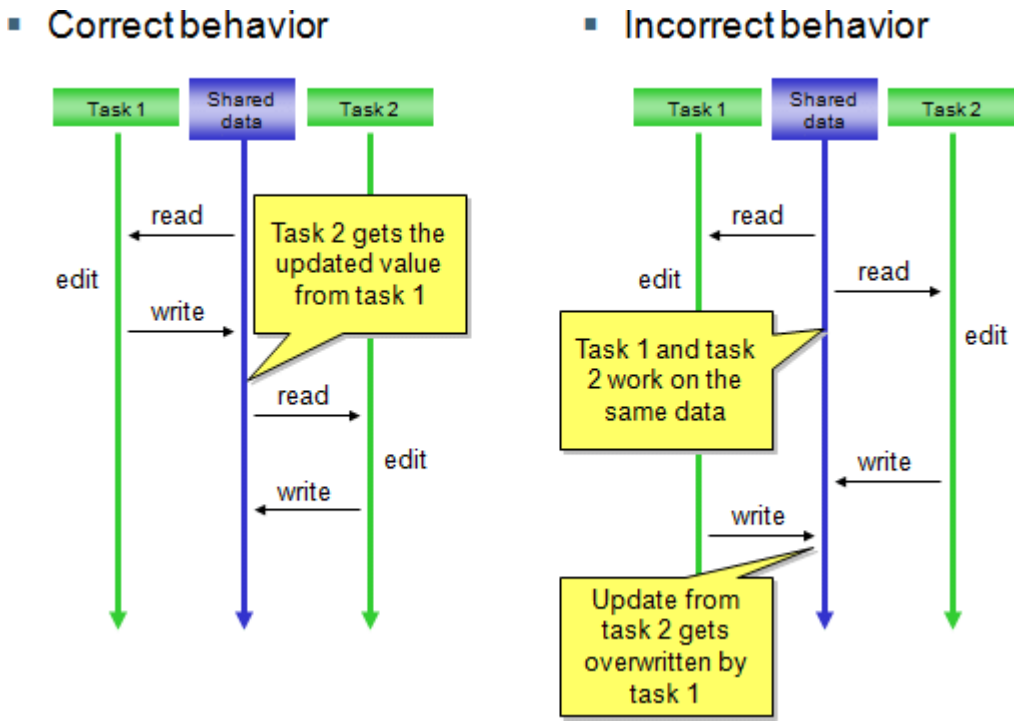

Figura 40. Ejemplo de race condition[45]

[45] https://www.researchgate.net/figure/Race-condition_fig2_316989907

3.3.4.2. Deadlock

El *deadlock* o condición de bloque es uno de los problemas más comunes en la programación paralela. Ocurre cuando varias tareas no pueden seguir con su ejecución normal porque unas están a la espera de la finalización de las otras.

Para evitar este tipo de situaciones de bloqueo es necesario realiza una reestructuración u ordenación de las sincronizaciones que están provocando esta situación.

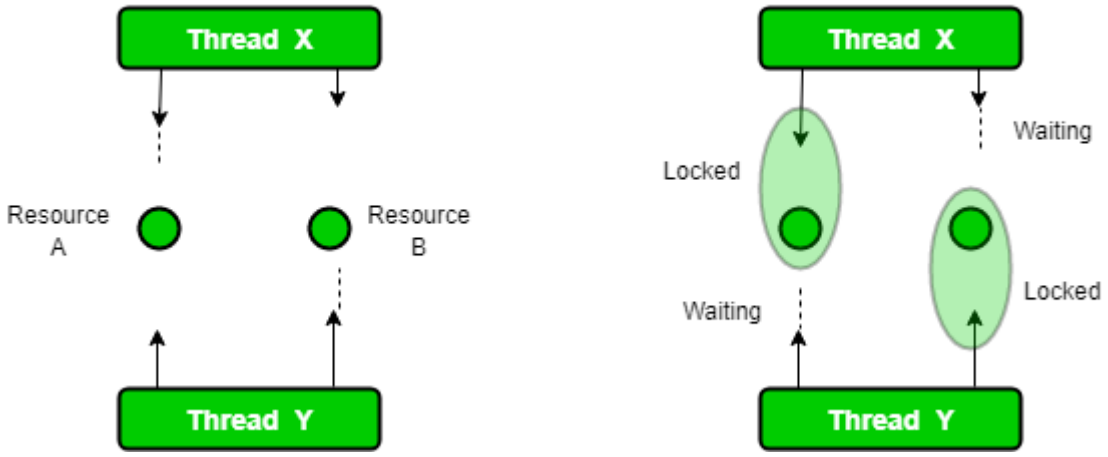

Figura 41. Ejemplo de deadlock[46].

3.3.4.3. Starvation y Livelock

Starvation y *livelock* son problemas mucho menos comunes que el interbloqueo (*deadlock*), pero siguen siendo problemas que es probable que todos los diseñadores de software concurrentes encuentren. (Java Tutorials, 2017)

Starvation: describe una situación en la que un subproceso no puede obtener acceso regular a los recursos compartidos y no puede avanzar. Esto sucede cuando los recursos compartidos no están disponibles durante largos períodos de tiempo por subprocesos "codiciosos". Por ejemplo, supongamos que un objeto proporciona un método sincronizado que a menudo toma mucho tiempo para volver. Si un hilo invoca este método con frecuencia, a menudo se bloquearán otros hilos que también necesiten acceso frecuente y sincronizado al mismo objeto.

Livelock: un hilo a menudo actúa en respuesta a la acción de otro hilo. Si la acción del otro subproceso también es una respuesta a la acción de otro subproceso, entonces puede resultar *livelock*. Al igual que con el interbloqueo, los subprocesos bloqueados no pueden avanzar más. Sin embargo, los subprocesos no están bloqueados, simplemente están demasiado ocupados respondiéndose entre sí para reanudar el trabajo. Esto es comparable a dos personas que intentan cruzarse en un corredor: A se mueve a su

[46] https://www.geeksforgeeks.org/deadlock-in-java-multithreading/

izquierda para dejar pasar a B, mientras que B se mueve a su derecha para dejar pasar a A. Al ver que todavía se están bloqueando, A se mueve a su derecha, mientras que B se mueve a su izquierda.

3.4. Reflexión sobre el futuro de la programación paralela.

Hablar de futuro cuando nos referimos al paralelismo en la computación es hablar de presente. Hoy en día, en cualquier dispositivo que manejemos dispone de los elementos necesarios para el paralelismo, tanto a nivel software como a nivel hardware.

El progresivo aumento de las prestaciones del hardware que se encuentra en los dispositivos, permite que la programación paralela sea hoy día más necesaria que nunca, y, por consiguiente, los futuros desarrolladores de sistemas deben conocer las principales características de la misma y las herramientas que están a su disposición, desde lenguajes específicos a librerías de programación concurrente para lenguajes generales.

Muestra de este aumento de prestaciones son las tarjetas gráficas. En el mercado de hoy en día podemos encontrar tarjetas como la Titan-RTX[47] que tienen unos procesadores con más de 4000 núcleos CUDA. Esto nos debe hacer reflexionar hacia dónde se dirige la computación paralela, y la importancia de conocer estas arquitecturas y los lenguajes para realizar un aprovechamiento óptimo de estos recursos hardware.

Además, es un indicativo de la importancia de la programación paralela, que todos los lenguajes de propósito general implementan librerías o recursos para la ejecución en paralelo de los algoritmos que los desarrolladores deseen implementar.

[47] https://www.nvidia.com/es-es/titan/titan-rtx/

4. Proyección didáctica

4.1. Título

Procesos y programación paralela

4.2. Justificación

Esta unidad didáctica se desarrolla después de que el alumno haya adquirido los conocimientos del módulo (0485) Programación del primer curso del Ciclo formativo de Grado Superior en Desarrollo de Aplicaciones Multiplataforma. Su existencia es importante, pues una vez que se ha aprendido los elementos del desarrollo software, es imprescindible para el alumnado conocer los principios de la programación paralela. Además, el contenido abordará conceptos y mecanismos relacionados con la computación paralela que serán de uso común cuando nuestros alumnos sean futuros profesionales.

4.3. Contextualización curricular.

- Etapa: Ciclo Formativo de Grado Superior. Desarrollo de Aplicaciones Multiplataforma
- Módulo: (0490) Programación de Servicios y Procesos
- Curso: 2
- Horas totales: 63. Horas semanales: 3
- Trimestre: Primero
- Número de sesiones: 13 sesiones de 1 horas
- Temporalización número de horas: 13 horas

4.4. Contexto del centro y alumnado

Esta unidad didáctica se contextualiza en el IES Fernando III de Martos. Martos es un municipio rico que presenta un importante dinamismo socioeconómico, de carácter eminentemente agrícola, cubierto por olivares en un 77% de su territorio. Se encuentra a 25 km de la capital de Jaén y emplazado en la comarca denominada Campiña Sur. Es el mayor productor olivarero del mundo, y en la actualidad tiene fábricas derivadas de su agricultura, del sector de iluminación del automóvil e industrias auxiliares de éstas. Martos se sitúa entre la Campiña de Jaén y la zona noroccidental de las Sierras Sub-béticas. La población aproximada es algo más de 24000 habitantes; el término tiene una extensión de 259,16 km2 y la ciudad se asienta en la falda de La Peña. El asentamiento urbano de Martos se realiza históricamente en la ladera de la Peña. La Peña es un promontorio natural rocoso, de forma troncocónica, última estribación de Sierra Mágina, que alcanza una altitud de 1003 m. (Proyecto de Centro)

Parte de la zona de influencia del centro corresponde a la parte alta del núcleo urbano. Está situada en la ladera de La Peña y habitada en su mayoría por personas

asalariadas y de escasos recursos económicos. Es el casco histórico del pueblo, que se va despoblando por los inconvenientes que presenta su habitabilidad.

En la Educación Secundaria Obligatoria, el IES "Fernando III" tiene adscritos para 1º de ESO los siguientes CEIP: Tucci y San Amador. En 3º de ESO se incorporan alumnos del CEIP "Fernando IV" de Monte Lope Álvarez y del CEIP "Santiago Apóstol" de Santiago de Calatrava. Las realidades sociales, económicas, culturales y laborales de los cuatro colectivos de alumnos y alumnas procedentes de estos centros son bien distintas, aunque lógicamente con características comunes, propias de la sociedad del entorno en el que nos movemos.

Una parte considerable de la población que corresponde con la zona de influencia del Centro posee un nivel cultural y socioeconómico bastante bajo. La población que se sitúa en las áreas de influencia donde está situado el Colegio Tucci, que es el otro Centro del que se reciben alumnos tiene un nivel socioeconómico y cultural medio y medio-alto. La zona de expansión recoge la población con un mayor índice socioeconómico y cultural, y en ella se sitúan ciudadanos con una aceptable cualificación profesional, empresarios, titulados universitarios, funcionarios, etc.

4.5. Marco legislativo

La normativa que regula el título donde se encuadra la programación es la siguiente, tanto a nivel nacional como autonómico:

Real Decreto 450/2010, de 16 de abril, por el que se establece el título de Técnico Superior en Desarrollo de Aplicaciones Multiplataforma y se fijan sus enseñanzas mínimas[48].

Orden de 16 de junio de 2011, por la que se desarrolla el currículo correspondiente al título de Técnico Superior en Desarrollo de Aplicaciones Multiplataforma[49].

4.6. Competencia general

Según el artículo 4 del Real Decreto 450/2010, la competencia general de este título consiste en desarrollar, implantar, documentar y mantener aplicaciones informáticas multiplataforma, utilizando tecnologías y entornos de desarrollo específicos, garantizando el acceso a los datos de forma segura y cumpliendo los criterios de «usabilidad» y calidad exigidas en los estándares establecidos.

[48] https://www.boe.es/boe/dias/2010/05/20/pdfs/BOE-A-2010-8067.pdf
[49] https://www.juntadeandalucia.es/boja/2011/142/20

4.7. Orientaciones pedagógicas

Este módulo profesional contiene parte de la formación necesaria para desempeñar la función de desarrollo de aplicaciones seguras en red.

La función de desarrollo de aplicaciones seguras en red incluye aspectos como:

- La utilización de las capacidades ofrecidas por el sistema operativo para la gestión de procesos e hilos.
- La programación de aplicaciones compuestas por varios procesos e hilos.
- El desarrollo de aplicaciones con capacidades para comunicarse y ofrecer servicios a través de una red.
- La utilización de mecanismos de seguridad en el desarrollo de aplicaciones.

4.8. Competencias profesionales, personales y sociales

La formación del módulo contribuye a alcanzar las competencias profesionales, personales y sociales de este título que se relacionan a continuación:

n) Desarrollar aplicaciones multiproceso y multihilo empleando librerías y técnicas de programación específicas.

w) Mantener el espíritu de innovación y actualización en el ámbito de su trabajo para adaptarse a los cambios tecnológicos y organizativos de su entorno profesional.

4.9. Líneas de actuación

Las líneas de actuación en el proceso de enseñanza-aprendizaje que permiten alcanzar los objetivos del módulo versarán sobre:

- La gestión de procesos e hilos y los mecanismos de comunicación y sincronización entre ellos.
- El desarrollo de programas compuestos por varios procesos e hilos.

4.10. Objetivos Generales

De conformidad con lo establecido en el artículo 9 del Real Decreto 450/2010, de 16 de abril, por el que se establece el título de Técnico Superior en Desarrollo de Aplicaciones Multiplataforma y se fijan sus enseñanzas mínimas, la formación del módulo contribuye a alcanzar los objetivos generales de este ciclo formativo que se relacionan a continuación:

ñ) Analizar y aplicar técnicas y librerías de programación, evaluando su funcionalidad para desarrollar aplicaciones multiproceso y multihilo.

4.11. Resultados de aprendizaje

1. Desarrolla aplicaciones compuestas por varios procesos reconociendo y aplicando principios de programación paralela

4.12. Objetivos didácticos

Además de los objetivos generales recogidos en la normativa, en esta unidad didáctica se pretenden alcanzar los siguientes objetivos extra:

- Conocer las principales arquitecturas de computadores
- Conocer y distinguir conceptos como proceso, servicio e hilo de ejecución
- Gestionar los procesos en ejecución en un sistema operativo
- Conocer y utilizar clases para lanzar y gestionar procesos

4.13. Temas transversales

En esta unidad didáctica se va a tratar como eje transversal la concienciación del entorno social, cultural y medioambiental que rodea al alumnado del centro. Para ello, se plantea una actividad en la que se conectarán los contenidos de la unidad con la gestión del pesaje de aceitunas de una cooperativa agrícola.

Además, se realizarán actividades encaminadas a fomentar el respeto al compañero y el trabajo en equipo mediante actividades de trabajo colaborativo.

4.14. Contenidos Conceptuales

Los contenidos básicos que se tratan en esta unidad didáctica son:

- Ejecutables. Procesos. Servicios. Problemas asociados a recursos compartidos.
- Estados de un proceso. Planificación de procesos por el sistema operativo.
- Hilos.
- Programación paralela.
- Creación de procesos.
- Gestión de procesos.
- Depuración y documentación.

Además de estos contenidos básicos recogidos en la Orden de 16 de junio de 2011, se tratarán estos otros contenidos:

- Arquitecturas de computadores: Von Neumann y Taxonomía de Flynn. (Antes de entrar en el contenido que regula la norma, se introducirán conceptos básicos de arquitecturas de computadores).

4.15. Metodología

En los siguientes subapartados se describe la metodología empleada en la unidad didáctica, desde la organización del aula, las actividades a desarrollar, pasando por la temporalización y los criterios de evaluación y calificación.

4.15.1. Organización del aula

Las clases se impartirán en el laboratorio de Informática del IES. La disposición de los ordenadores será en forma de U, donde el profesor podrá controlar las pantallas

de todos los ordenadores del aula. La pizarra digital estará al lado del profesor, por lo que el mismo tendrá contacto visual con todos los alumnos mientras explica. Si no hay suficientes ordenadores para los alumnos, podrán agruparse en grupos de 2 personas por ordenador.

4.15.2. Agrupamientos

Para llevar a cabo nuestra metodología se podrán realizar distintos agrupamientos de alumnos:

- Individual
- Parejas
- Grupos de 5 personas
- Grupo completo

Esta UD está pensada para grupos de 20 alumnos en total

4.15.3. Actividades de enseñanza-aprendizaje

La metodología seguida para para conseguir un aprendizaje significativo por parte del alumno/a se basará en distintas técnicas de aprendizaje:

- Actividades de exposición de contenidos
- Actividades de repaso: se utilizará una pequeña parte del tiempo de cada sesión para repasar el contenido visto en la sesión anterior.
- Actividades prácticas
- Actividades complementarias.
- Actividades de exposición de conocimientos por parte del alumnado.
- Actividades de evaluación.

4.15.4. Materiales y recursos

- Apuntes elaborados por el profesor y disponibles en la plataforma virtual del curso
- Ordenadores
- Conexión a internet:
  - Búsqueda de información, consulta plataforma virtual.
- Pizarra digital o proyector
- Medio transporte
- Software:
  - Kahoot
  - Máquina virtual con distribución Linux
  - Entorno de desarrollo: Netbeans con JDK Java instalado en la máquina virtual Linux.

4.15.5. Proceso de enseñanza-aprendizaje

Sesión 1. Semana 1

Sesión introductoria del tema. Se explicarán los conceptos básicos sobre arquitecturas, especialmente la arquitectura Von Neumann. Otros conceptos a explicar serán los procesos en un sistema operativo, los hilos de ejecución, los servicios y los ficheros ejecutables.

Se utilizará la plataforma docente para proporcionar al alumnado los apuntes para seguir la clase, con imágenes y ejemplos de cada concepto.

Sesión 2. Semana 1

En la primera parte de la sesión se realizará una actividad de repaso para consolidar los conceptos explicados en la sesión anterior. Posteriormente se realizará una actividad de expositiva de contenidos, abarcando contenidos referentes a la gestión de procesos en sistemas operativos y al envío de señales a los procesos. El material estará disponible en la plataforma docente para la descarga por parte del alumnado.

Sesión 3. Semana 1

Como en la sesión anterior, en la primera parte de la sesión se realizará una actividad de repaso para consolidar los conceptos explicados en la sesión anterior. Después se realizará una actividad de exposición de contenidos referidos a programación paralela y comunicación entre procesos. De igual modo que en anteriores sesiones, el material estará previamente disponible en la plataforma docente del módulo para su descarga, consulta y seguimiento en clase.

Sesión 4. Semana 2

Reparto de trabajos. Se crearán grupos de 5 alumnos, y se propondrán diferentes temas a elegir, sobre los que desarrollar un trabajo. Una vez que se ha realizado el reparto, se realizará una actividad expositiva cuyo contenido versará sobre la creación hilos en JAVA utilizando clase Thread e interfaz Runnable. Los alumnos podrán seguir los ejercicios que el profesor va explicando, en grupos de 2, para favorecer el aprendizaje colaborativo.

Sesión 5. Semana 2

En esta sesión se organiza una visita a la Universidad de Jaén, concretamente al Centro de Estudios Avanzados en Tecnologías de la Información y de la Comunicación, CEATIC. Allí, previa conformidad y puesta en común con los responsables, se mostrará al alumnado el clúster con los diferentes nodos de cómputo destinados a supercomputación. Se pedirá a los responsables del mismo que muestren al alumnado la gestión de los nodos, así como aplicaciones que corran de forma concurrente en dichos nodos. Se pasará al alumnado un cuestionario sobre lo visto durante la visita, que

deberán cumplimentar durante la misma y posteriormente en casa. Entrega disponible hasta el día de la última sesión en el entorno virtual de aprendizaje del módulo.

Sesión 6. Semana 2

Repaso de los conceptos vistos en las sesiones 3 y 4. Una vez realizado el repaso, se llevará a cabo una exposición de contendidos, concretamente la sincronización de procesos mediante semáforos. Los alumnos podrán seguir los ejercicios que el profesor va explicando en su ordenador, en grupos de 2, para favorecer el aprendizaje colaborativo. Antes de finalizar esta sesión se informará a los alumnos de la existencia en la plataforma virtual una relación de problemas para su resolución en casa, relacionados con la materia que se ve en esta sesión 6, junto con lo que se verá en las sesiones 7 y 8, todos ellos de sincronización de procesos. Esta batería de problemas se entregará hasta el día anterior al de la penúltima sesión de la UD, la 11, mediante la plataforma de docencia virtual del módulo, con vista a su corrección en dicha sesión 11.

Sesión 7. Semana 3

Esta sesión se divide en dos partes, la primera, será una continuación de la anterior sesión, siguiendo con la sincronización de procesos mediante semáforos y utilizando la misma metodología (seguimiento por parejas de ejercicios guiados por el profesor).

La segunda parte, de igual forma que en la sesión anterior y la primera parte de ésta, se llevará a cabo una actividad de desarrollo de contenido, siguiendo con la sincronización de procesos, en este caso mediante tuberías (pipeline). Los alumnos podrán seguir los ejercicios que el profesor va explicando en su ordenador, en grupos de 2, para favorecer el aprendizaje colaborativo.

Sesión 8. Semana 3

Esta sesión octava, se termina de ver la sincronización de procesos mediante tuberías, utilizando la misma metodología que en las 2 sesiones anteriores, es decir, ejercicios guiados por el profesor, de forma grupal, en concreto, por parejas.

Sesión 9. Semana 3

Repaso de lo visto en las sesiones 6 ,7 y 8 sobre sincronización de procesos (semáforos y pipeline, 5 minutos de repaso para cada concepto). A continuación, se desarrolla una actividad de exposición de contenidos, donde se verá cómo depurar y documentar código utilizando JavaDoc. Se proporcionarán recomendaciones a la hora de depurar código, y se comentará el uso de Synchronized y excepciones en Java.

Sesión 10. Semana 4

Actividad de contenido transversal. Esta sesión se dedicará a plantear la resolución de un problema en el que los alumnos deben simular un proceso de la vida

real, en este caso, de la vida agrícola, que tanta importancia tiene en el contexto socio-económico del centro. Para ello, los alumnos deberán simular el proceso de pesaje de las aceitunas que los olivareros lleven a una cooperativa. Por un lado, deberán simular una cantidad X de tractores, cada uno con una cantidad distinta de kg de aceituna. La cooperativa tiene una báscula de pesaje, en la que, de forma secuencial, van pasando los tractores y pesando su carga. Los alumnos deberán programar tanto la simulación del proceso secuencial como una variante en la que, lugar de una única báscula de pesaje, existan Y básculas, y simular el paralelismo de pesaje de la cooperativa mediante la programación de la solución de este supuesto. La solución a esta actividad se entregará hasta el día de la última sesión de esta UD.

Sesión 11. Semana 4

Actividad expositiva por parte del alumnado de los trabajos propuestos en la sesión 4. Contarán con 10 minutos por grupo. Grupo de 5 alumnos. Al final de la exposición se realizará una pregunta a cada miembro del grupo sobre el trabajo expuesto para comprobar su participación en el mismo. En total, al ser 4 grupos se destinarán 15 minutos a cada uno entre exposición y cuestiones breves.

Sesión 12. Semana 4

Corrección de los ejercicios propuestos en la sesión 6 de cara a la prueba escrita de la última sesión. Se corregirán por parte del alumnado, participando de forma activa en la resolución de los mismos, de forma que se pueda crear un debate entre el alumnado para encontrar la solución óptima a cada problema. Servirán para asentar los conocimientos vistos y de cara a la prueba de evaluación de la sesión siguiente.

Sesión 13. Semana 5

Última sesión de evaluación de conocimientos adquiridos en la UD. Se desarrollará una prueba escrita, con distintos tipos de preguntas, desde tipo test a cuestiones prácticas. Se ajustará la dificultad de la prueba al tiempo disponible

4.15.6. Cronograma

Proceso de Enseñanza-Aprendizaje				
Actividad	**Tiempo**	**Sesión**	**Agrupamiento**	**Recursos**
1.1. Expositiva: Desarrollo de contenidos: - Conceptos básicos de arquitectura, proceso, hilo, servicio y ejecutable	60’	1	Individual	Ordenador Proyector Acceso a Internet Entorno virtual de aprendizaje

2.1. Repaso: Repaso de los contenidos vistos en la sesión 1	5’	2	Individual	Kahoot
2.2. Expositiva: desarrollo de contenidos: - Gestión de procesos en S.O. - Envío de señales a los procesos.	55’	2	Individual	Ordenador Proyector Acceso a Internet S.O. Linux Entorno virtual de aprendizaje
3.1. Repaso: Repaso de los contenidos vistos en la sesión 2	5’	3	Individual	Kahoot
3.2. Expositiva: desarrollo de contenidos. - Programación paralela: conceptos, diferencia con distribuida y concurrente. - Comunicación entre procesos: memoria compartida y paso de mensajes	55’	3	Individual	Ordenador Proyector Acceso a Internet S.O. Linux Entorno virtual de aprendizaje
4.1. Actividad grupal: Formación de grupos y elección de tema propuesto para realizar un trabajo	10’	4	Grupos 5	Propuestas de temas sobre los que realizar el trabajo.
4.2. Expositivita: desarrollo de contenidos y prácticas guiadas - Realización de ejercicios guiados por el profesor sobre creación de hilos en JAVA	50’	4	Parejas	Ordenador Proyector Entorno virtual de aprendizaje JDK Java Netbeans
5.1. Visita centro de computación de altas prestaciones en la UJA. Se realizará una visita al CEATIC. Previo a la visita los alumnos tendrán disponible un cuestionario sobre la visita.	60’	5	Grupal. Todos los alumnos	Transporte al centro Cuestionario disponible en el entorno virtual de aprendizaje
6.1. Repaso Repaso de los contenidos vistos en las sesiones 3 y 4	10’	6	Individual	Kahoot
6.2. Expositivita: desarrollo de contenidos y prácticas guiadas - Sincronización de procesos con semáforos. Realización de ejercicios guiados por el profesor.	45’	6	Parejas	Ordenador Proyector Entorno virtual de aprendizaje JDK Java Netbeans
6.3. Actividad informativa Poner en conocimiento del alumnado de la existencia de una batería de problemas a resolver en casa, del contenido visto hasta el momento.	5’	6	Individual	Batería de problemas disponible en el entorno virtual de aprendizaje

7.1. Expositivita: desarrollo de contenidos y prácticas guiadas - Continuación con la sincronización de procesos con semáforos. Realización de ejercicios guiados por el profesor.	40’	7	Parejas	Ordenador Proyector Entorno virtual de aprendizaje JDK Java Netbeans
7.2. Expositivita: desarrollo de contenidos y prácticas guiadas - Sincronización de procesos con tuberías. Realización de ejercicios guiados por el profesor.	20’	7	Parejas	Ordenador Proyector Entorno virtual de aprendizaje JDK Java Netbeans
8.1. Expositivita: desarrollo de contenidos y prácticas guiadas - Continuación con la sincronización de procesos con tuberías. Realización de ejercicios guiados por el profesor.	60’	8	Parejas	Ordenador Proyector Entorno virtual de aprendizaje JDK Java Netbeans
9.1. Repaso Repaso de los contenidos vistos en las sesiones 6, 7 y 8	10’	9	Individual	Kahoot
9.2. Expositivita: desarrollo de contenidos y prácticas guiadas - Depuración de código de programación paralela. - Uso de Synchronized y Excepciones en JAVA	50’	9	Parejas	Ordenador Proyector Entorno virtual de aprendizaje JDK Java Netbeans
10.1. Actividad práctica. **Tema transversal:** simulación de pesaje en cooperativa agrícola	60’	10	Individual	Ordenador Proyector Entorno virtual de aprendizaje JDK Java Netbeans
11.1. Expositivas por parte del alumnado. Cada grupo expondrá el trabajo elegido durante un máximo de 10 minutos. Posteriormente el profesor dispondrá de otros 5 minutos para preguntas.	60’	11	Grupos de 5 alumnos	Proyector
12.1. Actividad práctica Corrección de la batería de ejercicios propuesta en la sesión 6.3.	60’	12	Individual	Proyector Ordenador
13.1. Realización de una prueba sobre los conocimientos de la UD	60’	13	Individual	Examen en papel

4.16. Evaluación y Recuperación

En los siguientes subapartados se detallan los criterios de evaluación de la UD, los procedimientos e instrumentos usados para la evaluación, así como los criterios de calificación y recuperación que deben conocer los alumnos para la superación de la UD.

4.16.1. Criterios de evaluación

a) Se han reconocido las características de la programación concurrente y sus ámbitos de aplicación.

b) Se han identificado las diferencias entre programación paralela y programación distribuida, sus ventajas e inconvenientes.

c) Se han analizado las características de los procesos y de su ejecución por el sistema operativo.

d) Se han caracterizado los hilos de ejecución y descrito su relación con los procesos.

e) Se han utilizado clases para programar aplicaciones que crean subprocesos.

f) Se han utilizado mecanismos para sincronizar y obtener el valor devuelto por los subprocesos iniciados.

g) Se han desarrollado aplicaciones que gestionen y utilicen procesos para la ejecución de varias tareas en paralelo.

h) Se han depurado y documentado las aplicaciones desarrolladas.

4.16.2. Procedimientos.

Los procedimientos utilizados para la evaluación de la UD son:

- La actitud ante la asignatura y el comportamiento con los compañeros.
- Participación en las actividades de repaso.
- Realización de actividades prácticas propuestas en las sesiones 6.3 y 10.1
- Participación en la actividad complementaria de visita al centro de investigación de la universidad. Cuestionario con preguntas relacionadas con la visita
- Trabajo en grupo: se deberá realizar la actividad de la sesión 4.1 y exponerla en la sesión 11.1.
- La realización de un ejercicio de evaluación individual al finalizar la UD.

4.16.3. Instrumentos de evaluación.

Los instrumentos que se utilizarán para la evaluación de la UD son:

- El cuaderno del profesor, donde se recogerán anotaciones sobre la asistencia, puntualidad, comportamiento y participación en las actividades de repaso y participación en la corrección de problemas de la sesión 12.1

- Desarrollo de un trabajo sobre la temática de la UD, de forma grupal y exposición de la solución frente a los compañeros. Se evaluará mediante una rúbrica. Cada alumno deberá entregar su trabajo en su espacio de la plataforma virtual

No entrega el trabajo en la plataforma virtual	Entrega el trabajo en la plataforma virtual	Expone parte del trabajo, pero de forma imprecisa	Expone parte del trabajo, de forma clara y precisa	Contesta correctamente a la cuestión planteada por el profesor	Puntuación máxima (suma de los apartados anteriores)
0 puntos	5 puntos	7 puntos	15 puntos	10 puntos	Máximo 30 puntos

- Resolución de la batería de ejercicios propuestos en la sesión 6.3. Cada ejercicio se evaluará mediante una rúbrica. Se propondrán 4 ejercicios en total. Todos ellos tendrán el mismo peso.

No entrega el ejercicio o lo entrega fuera de plazo.	El ejercicio contiene errores graves que imposibilitan la consecución del resultado	El ejercicio contiene errores leves que imposibilitan la consecución del resultado	El ejercicio no contiene errores y se alcanza el resultado pedido.
0 puntos	1.5 puntos	3 puntos	5 puntos

- Resolución de la actividad de contenido transversal. Esta actividad de la sesión 10.1 se evaluará con una rúbrica, dando más peso a la parte de la solución paralelo que a la parte de la solución de forma secuencia. Puntuación máxima de la actividad: 10 puntos.

No entrega el ejercicio o lo entrega fuera de plazo.	La solución aportada a la parte de programación secuencial contiene errores que imposibilita la consecución del resultado deseado	La solución aportada a la parte de programación secuencial no contiene errores y se alcanza el resultado correcto.	La solución aportada a la parte de programación paralela contiene errores que imposibilita la consecución del resultado deseado	La solución aportada a la parte de programación paralela no contiene errores y se alcanza el resultado correcto.
0 puntos	1’5 puntos	4 puntos	2,5 puntos	6 puntos

- Cuestionario sobre la actividad complementaria. Este cuestionario versará sobre la visita realizada a la universidad de Jaén. Constará de 10 preguntas cortas, cada una de ellas con valor de 1 punto.

- Cuestionario de evaluación: consistirá en una batería de 20 preguntas con 4 respuestas múltiples, en las que un acierto contará 0.25 puntos y un fallo restará 0.06 puntos. Estas preguntas múltiples pueden ser de contenido teórico y de contenido práctico. Además, habrá 3 preguntas cortas cada, que versarán sobre bloques de contenido vistos en la unidad didáctica: arquitecturas, hilos, procesos, paralelismo, comunicación entre procesos, sincronización y depuración. Por cada pregunta corta bien contestada tendrá un valor de 1 punto. La máxima puntuación del cuestionario será: (20 * 0.25) + (3 * 1) = 8 puntos

4.16.4. Criterios de Calificación

La calificación final de esta UD se ajustará a los siguientes criterios de calificación:

a) **Actitud y comportamiento**: 10% de la nota de la UD.
    1. Asistencia: el alumno deberá asistir al menos al 50% de las sesiones de la UD para poder evaluarse según estos criterios. Si no es así, la calificación de la UD se regirá por los criterios de recuperación que posteriormente se detallan.
    2. Puntualidad: se partirán de 13 puntos en este apartado, 1 por cada sesión, y se restará 1 puntos por cada retraso en el acceso al aula.
    3. Comportamiento: se partirán de 10 puntos y se restará 2 puntos por cada actitud inadecuada con los compañeros o el profesor y 1 punto por maltratar el material del laboratorio.
    4. Refuerzo: se evaluará la participación en las actividades de repaso. Las actividades de las sesiones 2.1 y 3.1 tendrán 5 preguntas, y las de las sesiones 6.1 y 9.1 tendrán 8 y 9 preguntas respectivamente, debido a que versarán sobre más contenido Se utilizará Kahoot, En total serán 27 preguntas en Kahoot. Por cada pregunta que se responda se obtendrá 1 puntos (es indiferente que estén bien o mal, se califica la participación). En total podrá obtener en este apartado 27 puntos.

    La suma de este apartado será la suma de los puntos de puntualidad, comportamiento y refuerzo (máximo 50 puntos). Esta puntuación se divide entre 5 para que corresponda al % asignado a este criterio.

b) **Entrega de batería de problemas:** 20% de la nota de la UD.
    1. Se evaluarán los 4 problemas propuestos según la rúbrica anterior. Cada ejercicio tendrá un valor máximo de 5 puntos.

    La suma de este apartado es la suma de las puntuaciones de los 4 ejercicios, con un máximo de 20 puntos que se corresponde al % asignado a este criterio

c) **Entrega de cuestionario sobre la visita a la universidad:** 10% de la nota de la UD.

1. El alumno que no asista a la visita no podrá contestar al cuestionario, y por lo tanto no se le evaluará este apartado.
2. Para aquellos que lo entreguen, se evaluarán las 10 preguntas planteadas con 1 punto como máximo cada una.

La suma de este apartado es la suma de las puntuaciones de cada pregunta, con un máximo de 10 puntos, lo que se corresponde con el % asignado a este criterio

d) **Actividad de carácter transversal:** 10% de la nota de la UD.
   1. Se evaluará la solución entregada al ejercicio planteado como actividad transversal, cuya puntuación máxima es 10 puntos según la rúbrica de evaluación antes descrita.

   El valor de este apartado es el valor de la rúbrica de evaluación, con un máximo de 10 puntos, que se corresponde con el % asignado a este criterio.

e) **Trabajo en grupo:** 10% de la nota de la UD.
   1. Se evaluará el trabajo grupal entregado y expuesto por los alumnos. Tendrá un valor máximo de 30 puntos, según la rúbrica de evaluación descrita anteriormente.

   El valor de este apartado es el valor otorgado por la rúbrica dividido entre 3, para ajustarse al % asignado a este criterio de evaluación. 30 puntos máximo / 3 = 10 puntos, que se corresponde con dicho %.

f) **Examen sobre los contenidos de la UD:** 40% de la nota de la UD.
   1. Consiste en la resolución de un cuestionario tipo test sobre el contenido de la materia y la resolución de 3 problemas cortos. La puntuación máxima del cuestionario es de 8 puntos, según se describe en los instrumentos de evaluación. Las preguntas de tipo test tendrán un valor máximo de 5 puntos (20 preguntas a 0.25 puntos cada una), y los problemas cortos tendrán cada uno un valor de 1 punto.

   El valor de este apartado es la puntuación final del examen (máximo 8 puntos) multiplicado por 5, para ajustarse al % asignado a este criterio de calificación.

La nota final de la UD es la suma de las siguientes: Actitud y comportamiento (10%) + Batería de problemas (20%) + Cuestionario actividad complementaria (10%) + Ejercicio de contenido transversal (10%) + Trabajo en grupo (10%) + Examen (40%)

4.16.5. Criterios de recuperación.

Para aquellos alumnos que no superen la UD según los criterios de calificación ordinarios de la misma, se establecen unos criterios de recuperación que son los siguientes:

a) Deberán entregar una nueva batería de problemas similar a la presentada en la sesión 6.3, con 4 problemas diferentes. Se evaluará mediante la misma rúbrica que los ejercicios ordinarios, es decir, cada uno con un máximo de 5 puntos. Esto supone un máximo de 20 puntos, que será el porcentaje de este criterio. (20%).
b) Examen sobre los contenidos: 80%. Serán 40 preguntas tipo test con 2 puntos por pregunta y penalización de 0.5 por respuesta errónea.
c) Si el alumno obtiene 5 puntos sobre 10 con estos criterios de recuperación, obtendrá un 5 en la nota final de la UD. Si el alumno obtiene un 10 en la recuperación, obtendrá un 7.5 en la UD. Se llevará la puntuación final obtenida sobre el rango 5 - 10 a un rango 5 - 7.5. Es decir, esta puntuación de 7.5 será la puntuación máxima posible en la recuperación de la UD

5. Bibliografía

- Ali, H.I. y L.M. Pinho. (2011). A parallel programming model for ada. *ACM SIGAda Ada Letters, 31(3),* 19-26. doi: 10.1145/2070336.2070350
- Backus, J.W. et al. (1956). *The Fortran Automatic Coding System For The IBM 740 EDPM.* Nueva York: Applied Science Division And Programming Research Dept.
- Backus, J.W. et al. (1960). Revised Report on the Algorithmic Language Algol 60. *Communications of the ACM*, 3(5), 299-314. doi: 10.1145/367236.367262
- Baeldung. (2019). *Overview of the java.util.concurrent*. Recuperado de : https://www.baeldung.com/java-util-concurrent (última consulta: 14/6/2019)
- Barney B. (2019). Introduction to Parallel Computing. *Lawrence Livermore National Laboratory*. Recuperado de https://computing.llnl.gov/tutorials/parallel_comp/#Whatis (última consulta: 17/6/2019)
- Barney, B. (2019). Message Passing Interface (MPI). *Lawrence Livermore National Laboratory*. Recuperado de: https://computing.llnl.gov/tutorials/mpi/ (última consulta: 16/6/2019)
- Barney, B. (2019). OpenMP. *Lawrence Livermore National Laboratory.* Recuperado de: https://computing.llnl.gov/tutorials/openMP (última consulta: 16/6/2019)
- Butenhof, D.R. (1997). *Programming with POSIX Threads*. Addison Wesley
- Bosworth, S. et al. (2014*). Computer Security Handbook. Sixth Edition.* John Willey & Sons
- Cook, S. (2013). *Cuda Programming. A developer's guide to parallel computing with GPUs.* Waltham: Elsevier
- Cuda Zone. *NVidia*. Recuperado de: https://developer.nvidia.com/cuda-zone (última consulta: 16/6/2019)
- Fernández, J. (2012). *Java 7 Concurrency Cookbook*. Birmingham: Packt Publishing.
- Flynn, M.J. (1972). Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computer,* 21, 948
- Foster, I. (1995). *Designing and Building Parallel Programs (Online).* Addison-Wesley. Recuperado de: https://www.mcs.anl.gov/~itf/dbpp/ (última consulta: 10/6/2019)
- Galowicz, J. (2017). *C++17 STL Cookbook.* Birmingham: Packt Publishing
- Gergel, V. (2012). *Introduction to Parallel Programming*. Microsoft Research. Summer School on Concurrency
- Lea, D. (2000). A Java fork/join framework. *Proceedings of the ACM 2000 conference on Java Grande*, 36-43. doi: 10.1145/337449.337465

- Loh, E. (2010). The Ideal HPC Programming Language. *ACMQueue, 8(6).* Recuperado de: https://queue.acm.org/detail.cfm?id=1820518 (última consulta: 12/6/2019)
- Marlow, S. (2013). *Parallel and Concurrent Programming in Haskell*. O'Reilly Media
- Palach, J. (2014). *Parallel Programming with Python*. Birmingham: Packt Publishing
- Peyton, S. y S. Singh. Satnam. (2019). A Tutorial on Parallel and Concurrent Programming. *Haskell Lecture Notes from Advanced Functional Programming Summer School 2008 (to appear)*
- Proyecto de Centro. IES Fernando III, Martos
- Roales, N. (2015). *El paralelismo en Java y el framework Fork/Join.* Recuperado de: https://www.adictosaltrabajo.com/2015/08/21/el-paralelismo-en-java-y-el-framework-forkjoin/ (última consulta: 14/6/2019)
- Román, J.E. et al. (2018). *Ejercicios de programación paralela con OpenMP y MPI.* Valencia: Universitat Politécnica de Valencia
- Sottile, M.J. et al (2010). *Introduction to Concurrency in Programming Languages.* Boca Raton: Chapman & Hall/CRC
- Starvation and Livelock. (2017). *The Java Tutorials.* Recuperado de: https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html (última consulta: 26/6/2019)
- Tanenbaum, A.S., (2006). *Organización de Computadoras. Un enfoque práctico*. Pearson Educación. 4ª edición
- Tay, R. (2013). *OpenCL Parallel Programming Development Cookbook.* Packt Publishing
- Tosini, M. (2015). *Introducción a las Arquitecturas Paralelas.* Recuperado de: http://www.exa.unicen.edu.ar/catedras/arqui2/arqui2/filminas/Introduccion%20a%20las%20arquitecturas%20Paralelas.pdf (última consulta: 12/06/2019)
- Voevodin V. (2018). What Do We Need to Know About Parallel Algorithms and Their Efficient Implementation? En: Prasad S., Gupta A., Rosenberg A., Sussman A., Weems C. (ed.) *Topics in Parallel and Distributed Computing*, 23-58. Springer, Cham. doi: https://doi.org/10.1007/978-3-319-93109-8_2
- Von Nuemann, J. *Von Neumann Architecture*. Recuperado de https://www.computerscience.gcse.guru/theory/von-neumann-architecture (última consulta: 09/06/2019)