UNIVERSIDAD TÉCNICA ESTATAL DE QUEVEDO

FACULTAD DE CIENCIAS DE LA INGENIERÍA

ASIGNATURA:

APLICACIONES DISTRIBUIDAS

INTEGRANTES:

CASANOVA MORANTE HÉCTOR

HERRERA SILVA ALEXANDER

ROBALINO CHUEZ BRYAN

TEMA:

COMPUTACIÓN PARALELA

DOCENTE:

GUERRERO ULLOA GLEISTON

TABLA DE CONTENIDO

ILUSTRACIONES

1. Introducción

Debido al auge de la computación en la actualidad se requieren de aplicaciones que presenten un alto rendimiento y ejecución de múltiples procesos en el menor tiempo posible. Esto conllevo a la creación de diversas técnicas que agilicen la ejecución de procesos y actividades del computador, como el trabajo con hilos, tareas, hasta la computación concurrente y paralela.

En este documento se analiza la computación paralela como medio principal para la ejecución de actividades. La programación paralela es un tipo de procesamiento basado en la ejecución de múltiples actividades de forma simultánea; es decir, en procesos concurrentes.

El costo de la computación paralela necesita de los dos ejes presentes en la computación; el hardware y software. Para aplicarla se necesita de un conjunto de procesadores conectados entre sí, mismos que se dividirán las tareas y ejecutarlas simultáneamente para posteriormente agruparlas cuando hayan finalizado.

La lógica aplicada en la computación paralela no es muy diferente al trabajo con hilos y tareas en los sistemas operativos o en los lenguajes de programación. Se crean diversos procesos que son ejecutados en un momento determinado para obtener datos de cada uno de los procesos.

Es posible asociar una computadora paralela a la computación paralela, después de todo una computadora paralela es un dispositivo que cuenta con múltiples procesadores que tienen la capacidad de ejecutar diferentes procesos y trabajar colaborativamente para solucionar un problema.

2. Objetivos

2.1. Objetivo general

- Determinar la importancia de la computación paralela como un proceso necesario en el desarrollo de sistemas computacionales que presentan problemas complejos.

2.2. Objetivos específicos

- Definir las características principales de la computación paralela.
- Demostrar el funcionamiento de la computación paralela mediante modelos, arquitecturas y conjunto de pasos aplicados.
- Diferenciar los conceptos de computación concurrente, distribuida y paralela.

3. ¿Qué es el paralelismo?

El paralelismo es un término que hace referencia a la ejecución de múltiples procesos derivados de uno general [1]. La idea es por qué realizar una tarea que tarda *t* tiempo cuando podrían dividirse sub-tareas que reduzcan el tiempo de ejecución [2].

El paralelismo también llamado computación paralela utiliza múltiples núcleos del procesador ubicados en un computador o en otro mediante conexiones que permitan subdividir el trabajo en pequeños conjuntos que serán asignados a cada núcleo del CPU [3].

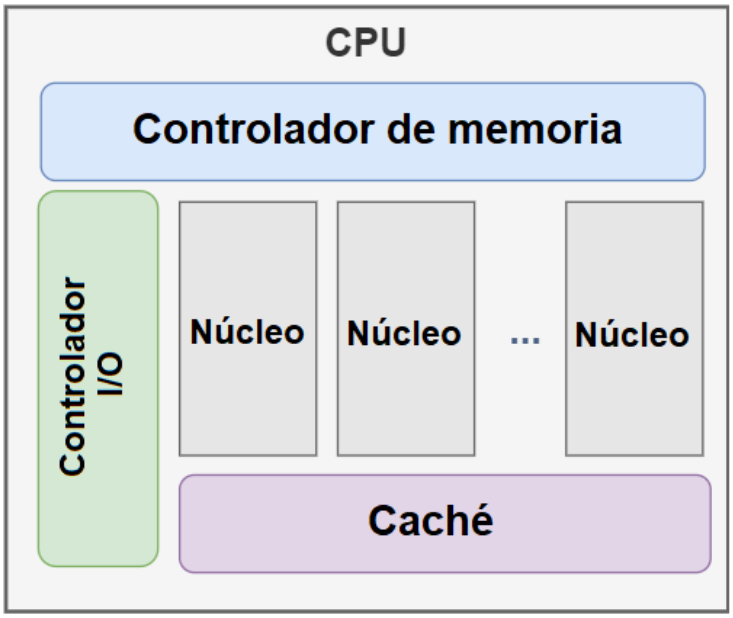


Ilustración 1. Unidad Central de Proceso, Obtenida de: [4].

3.1. El surgimiento de la computación paralelismo

Las mejoras continuas y más relevantes a los computadores fueron presentados desde el año de 1986, optimizando anualmente el rendimiento y velocidad de los procesadores y demás componentes hardware un aproximado del 50% [5].

Sin embargo, a partir del año 2002, el crecimiento porcentual de los procesadores se redujo a 20% anualmente. Este debido a los cambios en la arquitectura utilizada en el diseño de los procesadores [6].

El decremento notado en el desarrollo de los CPU por parte de los fabricantes, llevo a pensar en renovar las arquitecturas de los CPUs [7]. Entonces, los fabricantes

decidieron seleccionar el camino fácil ante tratar de mejorar las operaciones realizadas por el procesador [6] [7].

Es decir; el surgimiento de la computación paralela nace debido al decremento en las mejoras realizadas a los microprocesadores, y el pensamiento de los fabricantes de microprocesadores debido al pensamiento: ¿Por qué utilizar un procesador o CPU monolítico cuando puedo integrar múltiples CPUs en un circuito? [5], [6].

Esto conllevo a una serie de preguntas por parte de los fabricantes, entre ellas:

- ¿Por qué usar varios procesadores cuando uno puede satisfacer las necesidades de los usuarios?
- ¿Cuáles son las limitaciones que evitan el desarrollo y mejora de los procesadores monolíticos?
- ¿La inclusión de múltiples núcleos representa complejidad en el desarrollo de algoritmos?

El surgimiento de la computación paralela no solo surge debido al decremento del avance de los procesadores, si no también, se relaciona con la resolución de problemas y al cumplimiento de la frase “Divide y vencerás” [8]. Para entenderlo, se hace énfasis en los problemas computacionales [9].

3.2. Problemas computacionales

Existen problemas computaciones que se clasifican según su dificultad. La programación paralela al igual que la distribuida busca solucionar problemas que requieren de la gestión de múltiples datos y que la ejecución de un algoritmo dura mucho tiempo [10].

Es decir, la computación paralela apunta a solucionar problemas de tipo P y NP [6].

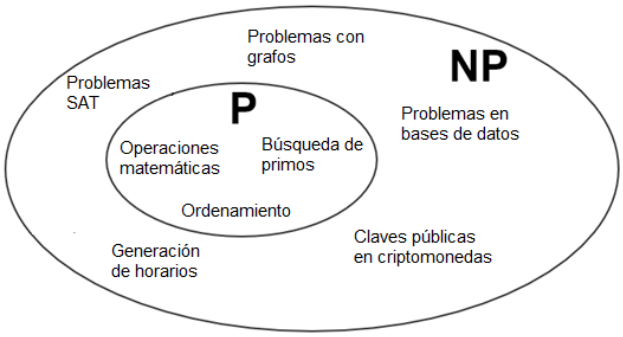


Ilustración 2. Clasificación de problemas Computacionales. Obtenido de: [11].

Entre los principales problemas de tipo NP, se encuentran:

- Problema de satisfacibilidad booleana (SAT o SMT);
- Problema de la mochila (knapsack);
- Problema del ciclo hamiltoniano;
- Problema del vendedor viajero;
- Problema de la clique.

4. Computación paralela

4.1. ¿Qué es la computación paralela?

Las razones fundamentales para adoptar máquinas de cómputo paralelo son reducir el tiempo total de ejecución de una aplicación, abordar problemas más complejos y de mayores dimensiones, así como permitir la ejecución simultanea de tareas concurrentes [6] [12].

Además, existen consideraciones económicas y se han alcanzado límites físicos difíciles de mejorar tecnológicamente en la transmisión de datos y la velocidad de la CPU de los procesadores actuales [12] [13]. Por lo tanto, la única manera de continuar aumentando el rendimiento de las computadoras es mediante la transición del paradigma de cómputo secuencial al paralelo, con múltiples procesadores, núcleos e hilos [14] [15].

La elaboración de programas informáticos para computación paralela resulta más compleja en comparación con los programas secuenciales, ya que requieren considerar de manera eficiente la coordinación en la ejecución de múltiples subtareas y la coherente agrupación de sus diversos resultados [15].

Según la naturaleza de la tarea a realizar, las múltiples tareas de un programa paralelo deben intercambiar información entre sí y establecer algún tipo de comunicación.[16]

A pesar de la gran popularidad de la computación paralela en la actualidad, se esperaría contar con gran información acerca de la misma [17]. Sin embargo, el conocimiento que se tiene acerca de la computación paralela es el mínimo [16].

En la actualidad, existen diferentes tipos de arquitectura, lenguajes de programación y compiladores que buscan adaptar la computación paralela para ser usada de forma

sencilla y permitir la ejecución de múltiples tareas mediante los procesadores del computador [18].

En los últimos años, se ha observado un notable desarrollo en los computadores paralelos, impulsado por avances de diversos campos como la arquitectura, las tecnologías de interconexión y los entornos de programación, incluyendo lenguajes y sistemas, entre otros [6] [12] [19].

Aunque se han propuesto varios esquemas de clasificación, el más ampliamente aceptado es la taxonomía de Flynn, la cual se fundamenta en la organización del flujo de datos y de instrucciones [17], [19].

La taxonomía de Flynn, una de las clasificaciones más reconocidas en el ámbito de la computación paralela, ofrece una perspectiva valiosa al analizar la multiplicidad del flujo de instrucciones y datos en un computador [19]. Esta clasificación se basa en la secuencia de datos sobre los cuales operan estas instrucciones, conocida como flujo de datos [5], [17], [18].

Esta clásica clasificación en el campo de la computación paralela utiliza conceptos familiares de la computación convencional para proponer una taxonomía de arquitecturas de computadores [17]. Aunque es una de las clasificaciones más antiguas, sigue siendo la más conocida hasta la fecha [20].

La idea central se basa en el análisis de los flujos de instrucciones y datos, permitiendo ser simples o múltiples, dando origen a cuatro tipos de máquinas [21]. En esencia, esta clasificación se fundamenta en el número de flujos de instrucciones y datos simultáneos que puede manejar el sistema computacional durante la ejecución de un programa [18].

4.2. Taxonomía de Flynn

La taxonomía de Flynn presenta una clasificación de las arquitecturas implementadas para la computación paralela. La clasificación se basa en dos ejes principales que son el número de instrucciones y el número de datos [17], [18], [21].

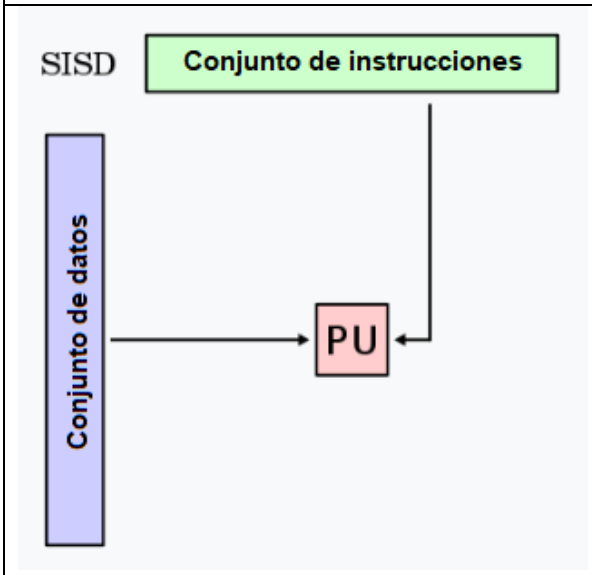
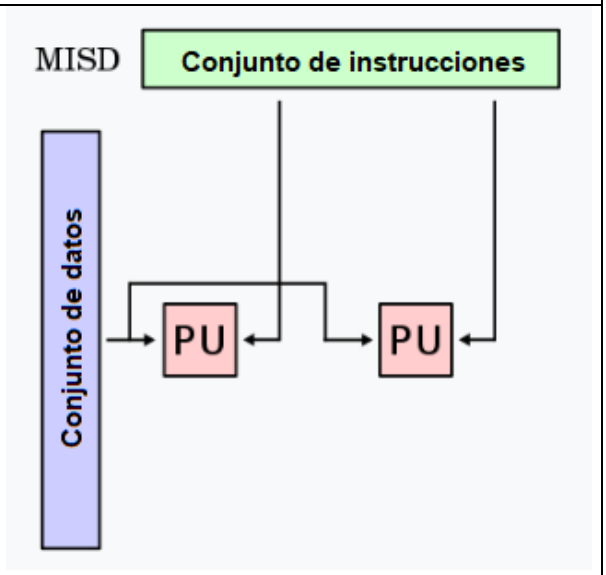
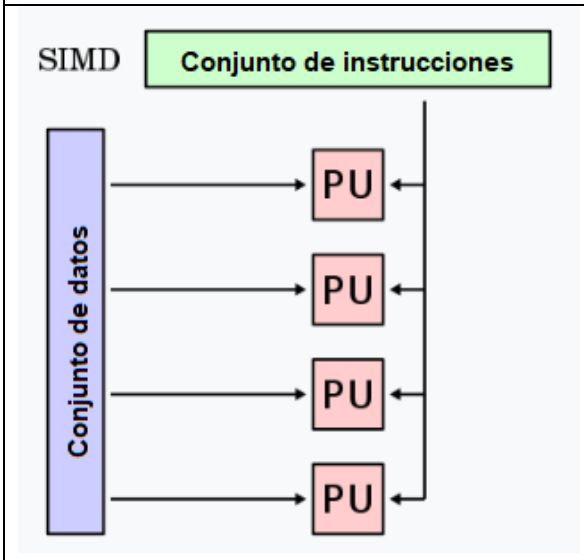
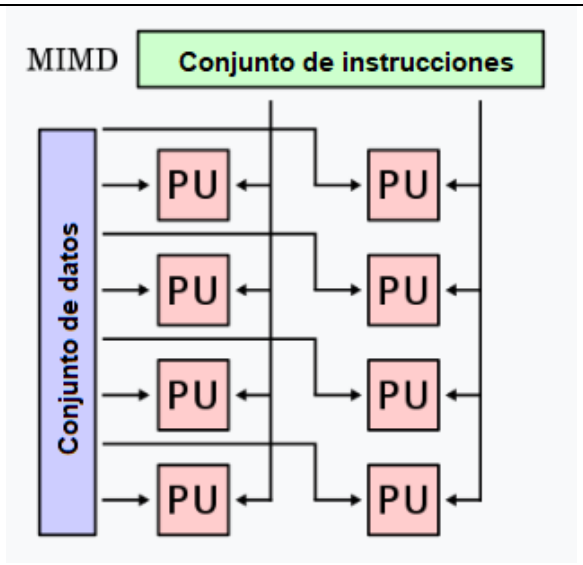
Una instrucción, un dato	Múltiples instrucciones, un dato
	
Una instrucción, múltiples datos	**Múltiples instrucciones, múltiples datos**
	

Tabla 1. Tipos de arquitecturas según la clasificación de Flynn. Obtenida de: https://www.researchgate.net/profile/Gabriel-Freytag-2/publication/325718150/figure/fig3/AS:636675323621378@1528806634560/Figura-5-Taxonomia-de-Flynn.png

4.2.1. **Una instrucción, un dato**

Los computadores SISD (Single instruction stream, single data stream), que constituyen la mayoría de las máquinas seriales, operan con un único procesador central (CPU) que ejecuta una instrucción en un momento específico y, simultáneamente, busca o almacena un dato en otro momento determinado.[22]

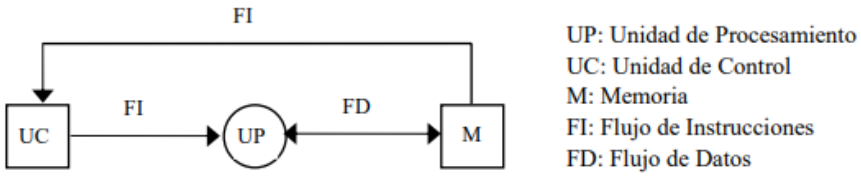


Ilustración 3. Único flujo de instrucciones, Un único flujo de datos. Obtenido de: [18].

4.2.2. **Una instrucción, múltiples datos**

Un controlador básico transmite las instrucciones de manera secuencial a un conjunto de procesadores que operan en un esquema maestro-esclavo [17]. En este caso, las instrucciones se difunden desde la memoria hacia un conjunto de procesadores [23].

Cada procesador actúa como una unidad aritmética-lógica y comparte una única unidad de control [24]. Todos los procesadores ejecutan simultáneamente cada operación recibida, lo que implica que cada uno realiza la misma instrucción en datos diferentes [18].

Los sistemas modernos SIMD (Single instruction stream, multiple data stream) incorporan una CPU adicional para el procesamiento escalar, permitiendo la combinación de operaciones entre el arreglo de procesadores y el procesador secuencial estándar [22]. Esto se debe a que muchas aplicaciones paralelas incluyen fases de código secuencial y paralelo. En consecuencia, la parte secuencial del código se ejecuta en la CPU adicional [24].

Este enfoque implica dos procesos distintos para trasladar datos entre el arreglo de procesadores y el procesador secuencial: en un caso, se difunden los datos desde el procesador secuencial al arreglo de procesadores; en el otro, al mover los datos desde el arreglo de procesadores al procesador secuencial, los datos se reducen [23], [24].

Desde la perspectiva del programador, la ejecución del programa parece ser secuencial, con la diferencia de que las instrucciones se ejecutan de manera múltiple en datos distintos y no una sola vez [17]. Esto hace que, para el usuario, la máquina siga siendo percibida como secuencial desde el punto de vista de la programación [25].

El tipo de plataforma computacional ha evolucionado en respuesta a numerosas aplicaciones científicas e ingenieriles que se benefician de este enfoque, como el

procesamiento de imágenes, la simulación de partículas, métodos de elementos finitos, sistemas moleculares, entre otros [26].

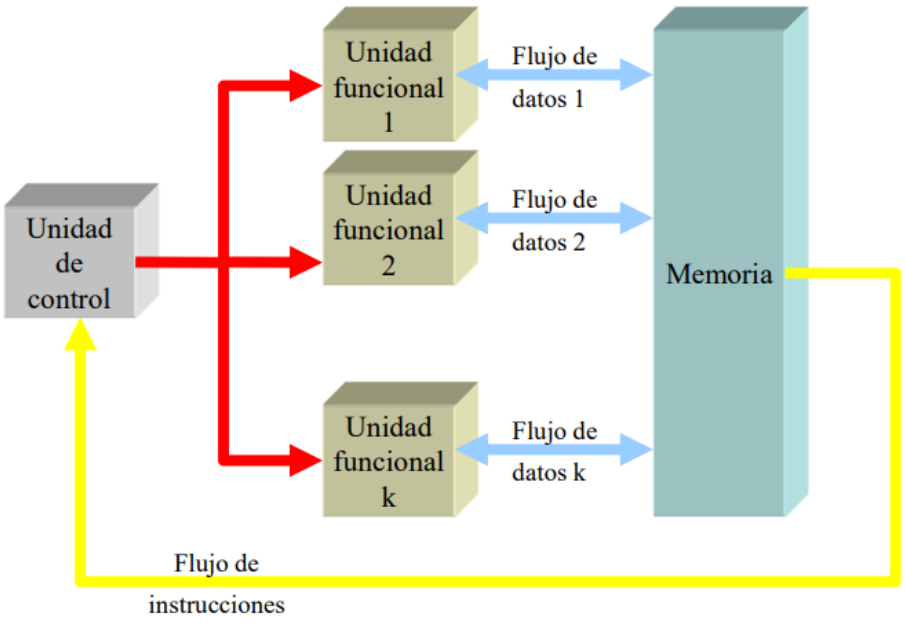


Ilustración 4. Flujo único de instrucciones, flujo múltiple de datos (SIMD). Obtenido de: [26].

4.2.3. **Múltiples instrucciones, un dato**

La clasificación MISD (Multiple instruction stream, single data stream) resulta poco intuitiva, ya que implica n procesadores, cada uno recibiendo una instrucción diferente y operando sobre el mismo flujo de datos, se presentan dos perspectivas del flujo de datos [18].

En FD1, el flujo se desplaza de unidad de procesamiento (UP) a UP, similar a un pipeline, mientras que en FD2, cada UP recibe una copia del flujo de datos [22]. Algunos arquitectos de computadoras han considerado esta estructura como impráctica, y en la actualidad no existen máquinas de este tipo, a pesar de que ciertos sistemas MIMD podrían ser utilizados de esta manera [27].

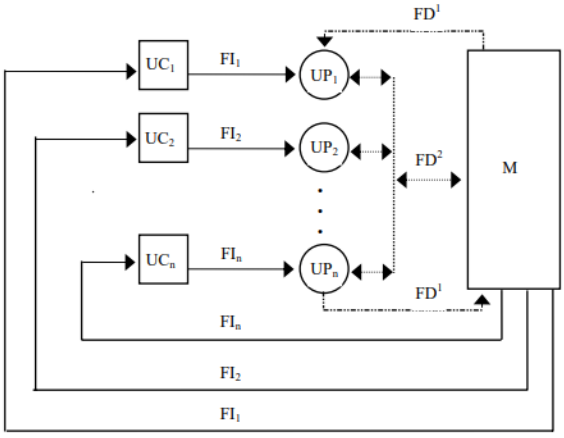


Ilustración 5. Flujo múltiple de instrucciones, flujo único de datos (MISD).Obtenido de: [18].

4.2.4. **Múltiples instrucciones, múltiples datos**

La mayoría de los sistemas multiprocesadores y multicomputadoras pueden ser categorizados dentro de este grupo [18], [22]. Los sistemas MIMD (Multiple instruction stream, multiple data stream) cuentan con múltiples procesadores independientes, y cada uno tiene la capacidad de ejecutar un programa distinto sobre sus propios datos [28].

Además, se puede realizar una subdivisión adicional de los multiprocesadores según la organización de su memoria, ya sea como memoria compartida o memoria distribuida [29].

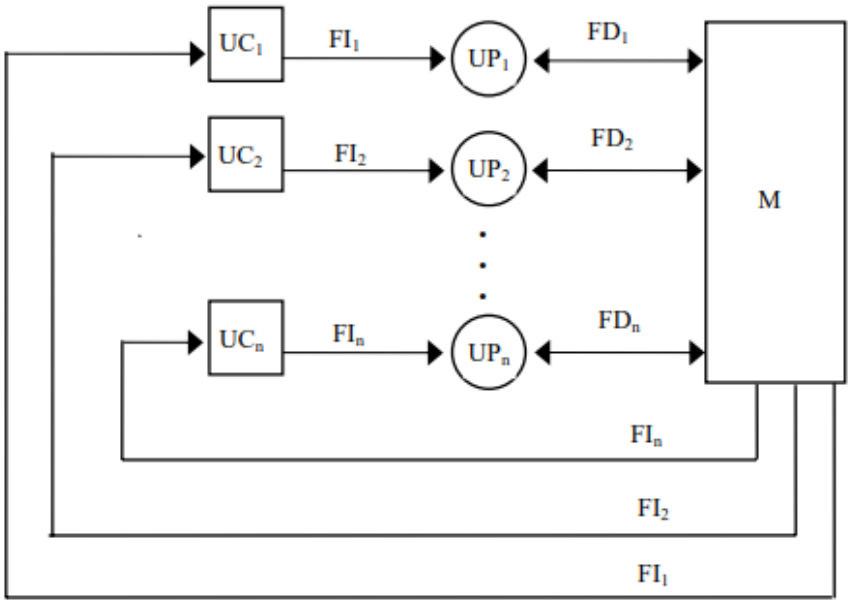


Ilustración 6. Flujo múltiple de instrucciones, flujo múltiple de datos (MIMD). Obtenido de: [18].

4.3. Ventajas y desventajas de la computación paralela

4.3.1. Ventajas

- **Reducción de tiempo**

La reducción de tiempo se considera una ventaja significativa debido a la capacidad de realizar múltiples tareas simultáneamente [30].

- **Se pueden solucionar problemas muy grandes que no es posible resolver mediante programación secuencial.**

Supera las limitaciones de la programación secuencial al permitir la resolución simultánea de tareas, lo que la hace especialmente adecuada para enfrentar problemas de gran envergadura que requieren un poder de procesamiento sustancial [31].

- **Se mejora la eficiencia al dividir el problema en tareas más pequeñas.**

La eficiencia en la computación paralela se mejora al dividir un problema en tareas más pequeñas, lo que aprovecha el paralelismo, facilita la distribución equitativa de la carga de trabajo, mejora la escalabilidad y permite abordar problemas complejos de manera más efectiva [32].

4.3.2. Desventaja

- **Encontrar la solución a un problema tiene una complejidad mayor.**

La complejidad adicional a menudo está relacionada con la necesidad de adaptar y diseñar adecuadamente el software para aprovechar al máximo el paralelismo sin introducir problemas de sincronización o rendimiento [33].

- **Requiere el uso de más recursos de la máquina**.

Implementar un sistema paralelo o desarrollar programas que aprovechen el paralelismo puede exigir más recursos hardware y, en algunos casos, software, en comparación con enfoques secuenciales [34]. Esto incluye componentes como procesadores adicionales, memoria, interconexiones de red más avanzadas y posiblemente mayor capacidad de almacenamiento [35].

- **Pueden surgir problemas de sincronización entre los diversos procesos en ejecución.**

Al ejecutar múltiples procesos o subprocesos de manera simultánea, puede haber situaciones en las que estos necesiten coordinarse y sincronizarse adecuadamente para evitar resultados inesperados o errores [36]. La sincronización se refiere a la coordinación temporal de las operaciones entre diferentes partes de un sistema paralelo [37].

4.4. Herramientas utilizadas para la computación paralela

En la actualidad existen algunas herramientas que permiten realizar aplicaciones utilizando múltiples núcleos o múltiples procesadores comunicándolos entre sí [6]. Estás herramientas son las siguientes:

4.4.1. Interfaz de transmisión de mensajes (MPI)

MPI es una herramienta que se basa en el uso de los núcleos del procesador, accediendo mediante variables a la cantidad de núcleos disponibles para realizar procesos [66].

Las atributos accesibles son: get_size() y get_rank(), los cuales permiten obtener el número total de procesos del dispositivo o comunicador, y los identificadores correspondientes a cada uno de los núcleos o procesos [66].

4.4.2. CUDA

CUDA es una plataforma disponible para las tarjetas gráficas (GNU) de nvidia, estas permiten la computación paralela, el cual incluye un compilador y un conjunto de herramientas que optimiza el rendimiento en las aplicaciones [68].

También incluye el procesamiento con la CPU; es decir, permite una computación mixta entre la CPU y GPU según sea requerido [68]. Sin embargo, las tarjetas gráficas presentadas por envidia en la actualidad presentan un mejor rendimiento a los procesadores genéricos [67], [68].

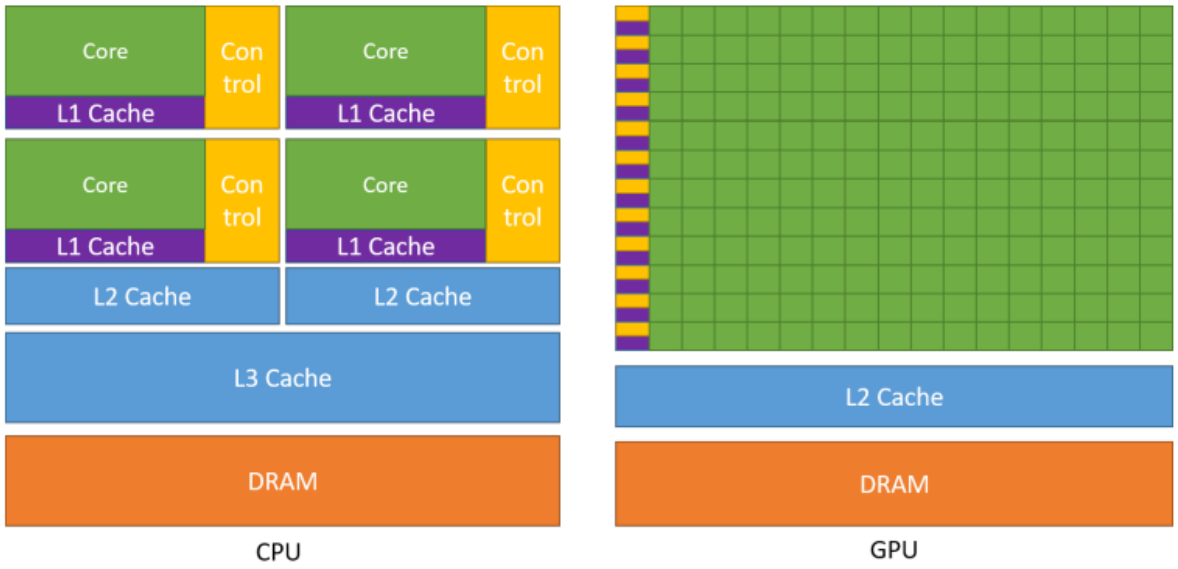


Ilustración 7.Uso de CPU y GPU presentado en la plataforma de programación CUDA. Obtenido de: [68].

4.4.3. OTRAS HERRAMIENTAS

Si bien existen diferentes herramientas que pueden ser utilizadas (diferentes a las anteriormente mencionadas), existen librerías o dependencias que son aplicables en los lenguajes de programación para hacer uso de la computación paralela [6].

Las herramientas presentadas a continuación, son un recopilatorio de aquellas librerías que pueden ser incorporadas en Python para el uso de múltiples procesadores o núcleos del mismo:

- Multiprocessing;
- Concurrent.features;
- Threading;
- Asyncio;
- Ray;
- Dask;
- etc.

4.4.4. OTRAS HERRAMIENTAS

Existen diversos modelos que a lo largo de los años se han ido proponiendo para hacer factible una computación paralela de propósito general, hasta la fecha no existe un modelo único que fundamente el desarrollo de la computación paralela [38].

4.4.5. El modelo PRAM (1978)

El modelo PRAM (Parallel Random Access Machine) es ampliamente reconocido y utilizado en la historia para analizar la complejidad de los algoritmos paralelos [39]. Este modelo asume la existencia de un conjunto ilimitado de procesadores, cada uno con una pequeña memoria local [40].

Estos procesadores ejecutan instrucciones de manera sincronizada y tienen acceso a una memoria compartida [41]. En cada unidad de tiempo, cada procesador tiene la capacidad de realizar cualquier subconjunto de las siguientes operaciones:

- Leer dos valores de la memoria global.
- Realizar una de las operaciones básicas sobre los dos valores leídos.

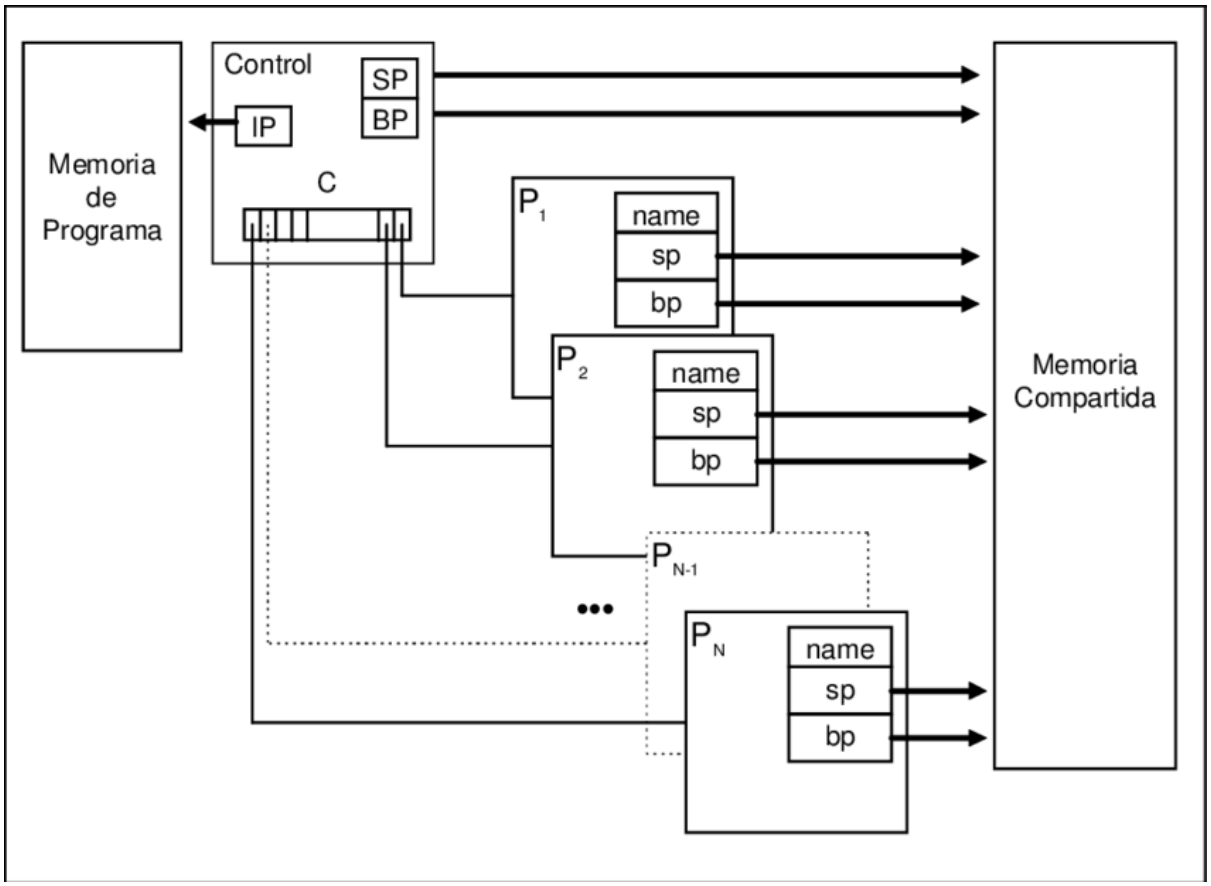


Ilustración 8. Modelo PRAM. Obtenido de: [42].

4.4.6. El modelo BSP (1990)

El modelo BSP(Bulk Synchronous Parallel) Computadora abstracta paralela síncrona masiva es uno de los fundamentos principales para la computación paralela [43]. Fue propuesta por Valiant en 1990 con la intención de establecer un modelo más realista que el PRAM, está compuesta por conjuntos de pares procesador-memoria, este

modelo utiliza cuatro parámetros para describir las características de una computadora paralela [44].

- P es el número de procesadores.
- S es la velocidad de computación de cada procesador.
- L es el tiempo necesario para realizar una sincronización entre los procesadores.
- G es la razón entre la capacidad total de computación por segundo y la capacidad total de comunicación por segundo.

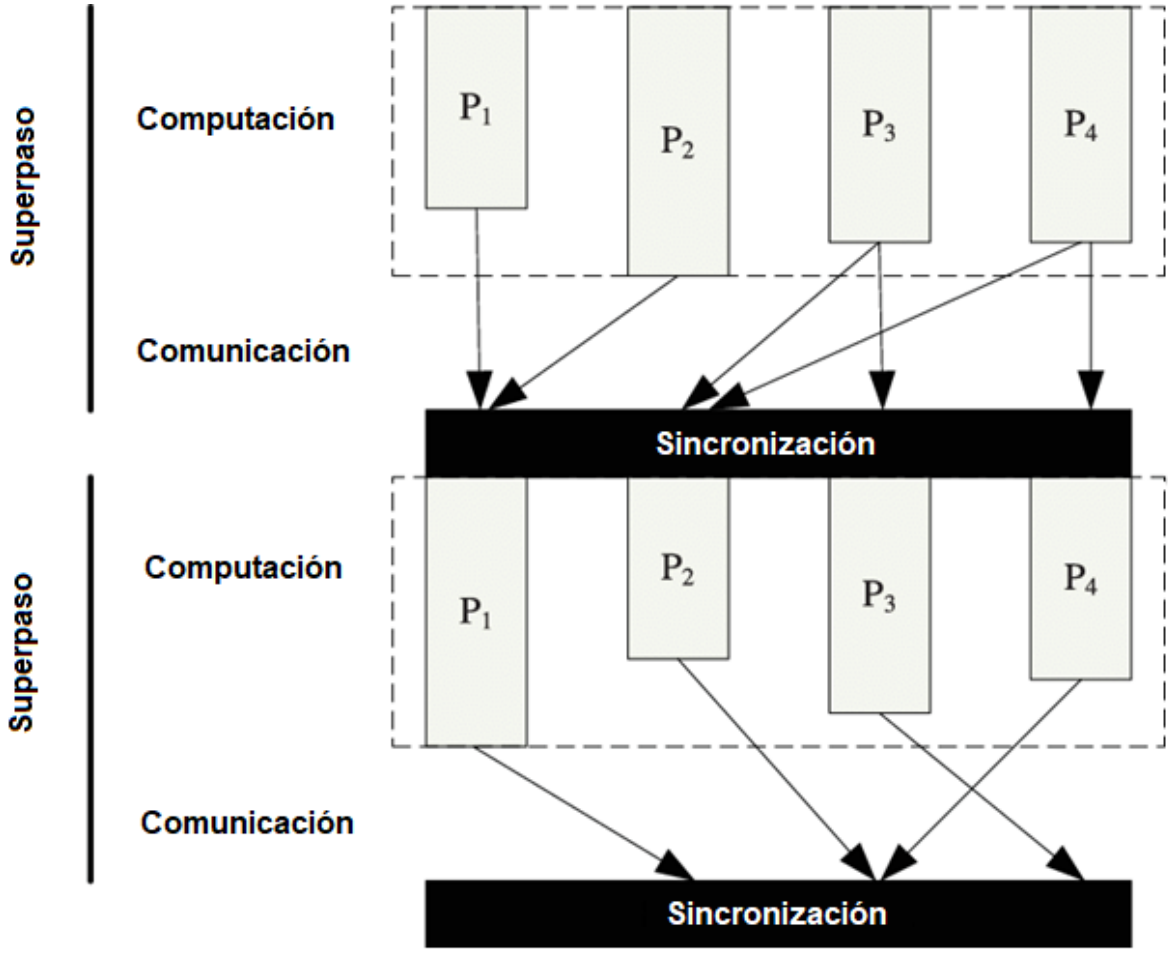


Ilustración 9. Modelo BSP. Obtenido de: [45].

4.4.7. El modelo LOGP (1993)

El modelo LogP, propuesto por Culler, Karp, Patterson en los últimos años, intenta reflejar las tendencias tecnológicas de los computadores paralelos: multiprocesador con memoria distribuida, y pretende ser útil en el desarrollo de algoritmos paralelos portables y eficientes [46].

Está caracterizado por cuatro parámetros que modelan el ancho de banda de comunicación, el tiempo de comunicación y la eficiencia de simultanear comunicación y cálculo [47]:

- P es el número de procesadores.
- L es una cota superior de la latencia que se produce al realizar una comunicación punto a punto de un mensaje corto, desde un procesador origen a otro destino.
- O (overhead) se define como el tiempo en que los procesadores están ocupados en la trasmisión o recepción de un mensaje y no pueden realizar ninguna otra tarea.
- G (gap) es el mínimo intervalo de tiempo requerido en un procesador entre dos envíos, o recepciones consecutivas, su recíproco corresponde al ancho de banda disponible por cada procesador.

La elección de estos parámetros pretende recoger de forma precisa las características del funcionamiento de las máquinas reales, esto hace que la utilización del modelo para el desarrollo y análisis de algoritmos no sea sencilla [48].

En Bilardi, Herley, Pietracaprina, Pucci y Spirakis realizan una comparación cuantitativa entre los modelos LogP y BSP señalan que ambos modelos pueden simularse eficientemente entre sí por lo que es preferible el modelo BSP debido a su mayor simplicidad [47], [48].

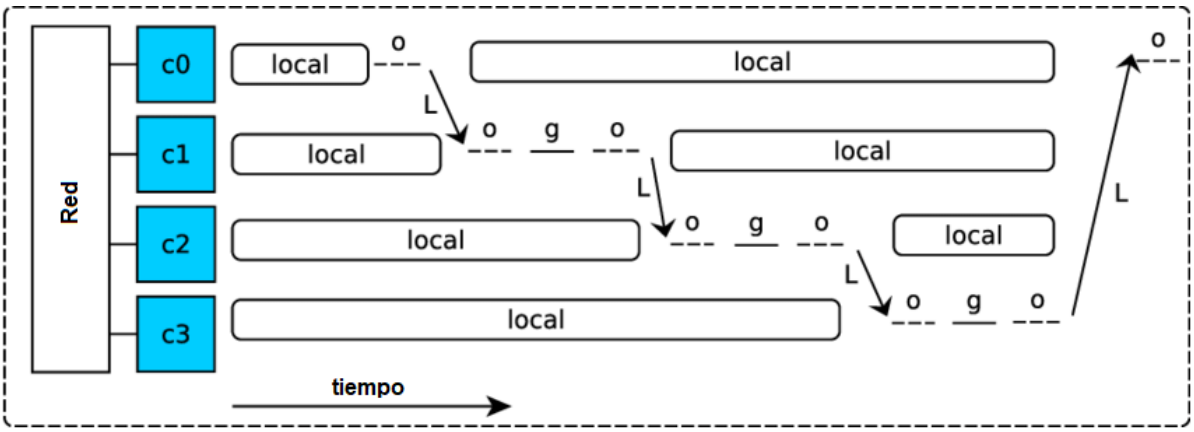


Ilustración 10. Modelo LOGP. Obtenido de: [48].

4.4.8. Datos en Paralelo

Dentro del campo de la computación paralela escalable, el paradigma data-parallel constituye otra alternativa importante a la construcción de programas [47]. Existe un

alto número de lenguajes de programación y elegantes teorías desarrolladas en torno al mismo [49].

Más que una base teórica para el análisis de la complejidad de los algoritmos paralelos es un modelo de programación que se basa en la distribución de los datos entre los distintos procesadores y en la generación automática por parte del compilador de la comunicación necesaria para llevar a cabo los cálculos, resulta particularmente apropiado para problemas en los que la localización es crucial [49].

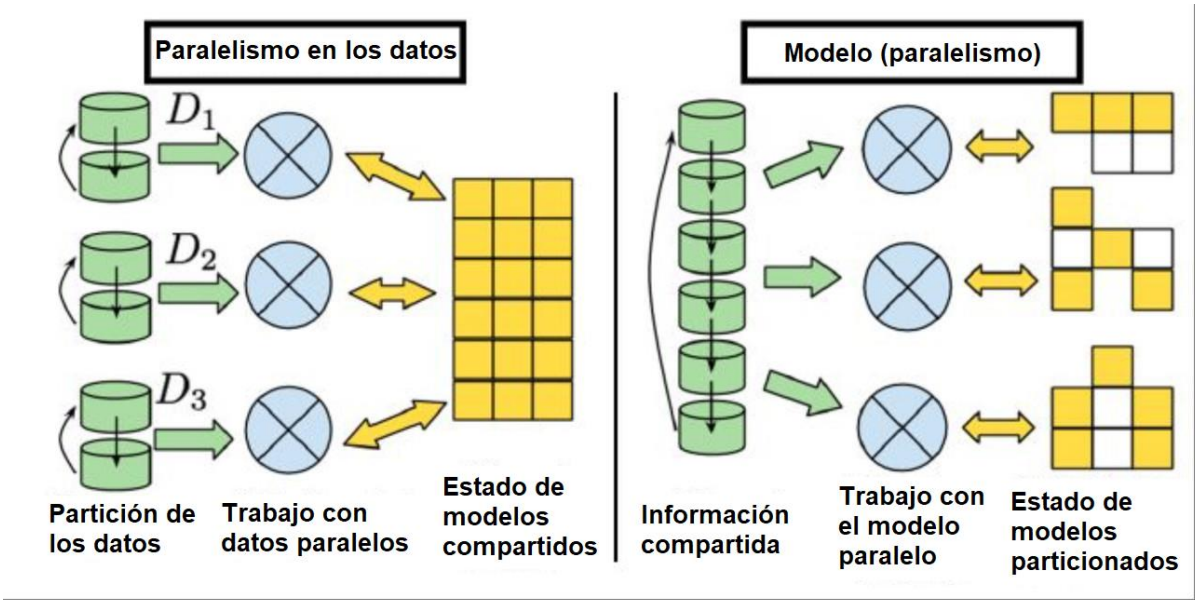


Ilustración 11. Datos en Paralelo. Obtenido de: [50].

4.4.9. Modelos de paso de mensajes (1989) (1992)

Los modelos de paso de mensajes se fundamentan en las primitivas de comunicación entre procesadores mediante operaciones de envío y recepción [53]. Carecen de un modelo analítico simple para analizar o prever el costo de los algoritmos paralelos, siendo difíciles de utilizar y con programas que presentan desafíos en el proceso de depuración [51]. Dos de los modelos más comúnmente empleados son PVM (Parallel Virtual Machine) y MPI (Message Passing Interface).[52]

La biblioteca de paso de mensajes PVM posibilita que una red heterogénea de computadoras se perciba como un recurso único: una máquina virtual. Su desarrollo se inició en 1989 en el Oak Ridge National Laboratory (ORNL) en Tennessee, EE. UU [53].

En los últimos años, se ha implementado para la mayoría de los sistemas multiprocesadores, convirtiéndose prácticamente en el método estándar para escribir programas paralelos en sistemas de memoria distribuida. Esto ha logrado el objetivo de generar códigos paralelos portables [54].

El propósito principal de MPI es establecer un estándar que facilite la creación de programas paralelos portátiles mediante el uso de paso de mensajes [55]. En su concepción, que comenzó en 1992, se consideraron las características de diversos sistemas de paso de mensajes ya existentes [52]. MPI posibilita la compilación de bibliotecas por separado y ha sido diseñado con la capacidad de ejecutarse en la mayoría de las plataformas paralelas.

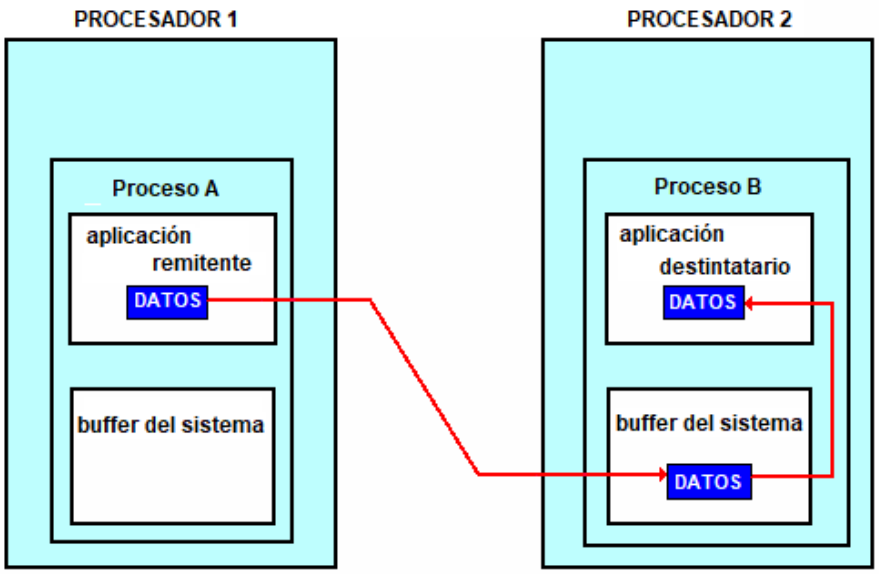


Ilustración 12. Modelo Paso de Mensajes. Obtenido de: [56].

4.5. Computación paralela vs computación concurrente

Antes de introducir en las diferencias y similaridades existentes entre la computación paralela y concurrente, se detallará que es la computación concurrente y cuáles son sus características.

4.5.1. Computación concurrente

La concurrencia es la capacidad de un sistema para procesar más de una tarea o un hilo de ejecución(proceso) al mismo tiempo [6], [57]. Una de las características en la computación concurrente es que sí puede darse en un sistema mono-procesador o

monolítico [57], [58]. Hay concurrencia si los procesos “conviven” en el mismo instante de tiempo [58].

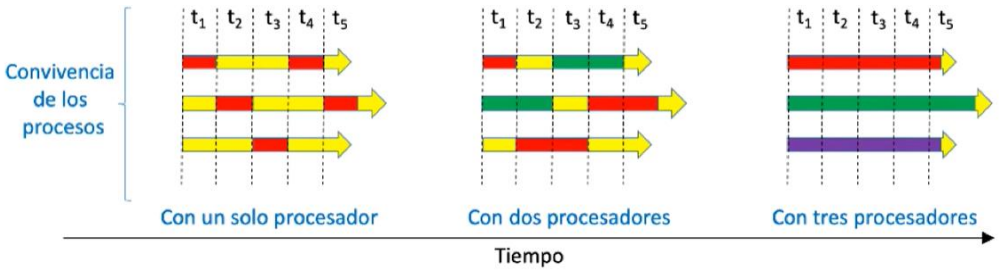


Ilustración 13. Convivencia de los procesos. Obtenido de: [60].

4.5.2. Paralelismo frente a concurrencia

El paralelismo es la capacidad de un sistema para ejecutar más de un hilo de ejecución(proceso) al mismo tiempo [60]. Es decir, si varios procesos se ejecutan al mismo tiempo (son paralelos), eso implica que conviven a la vez. Por tanto, son concurrentes [61].

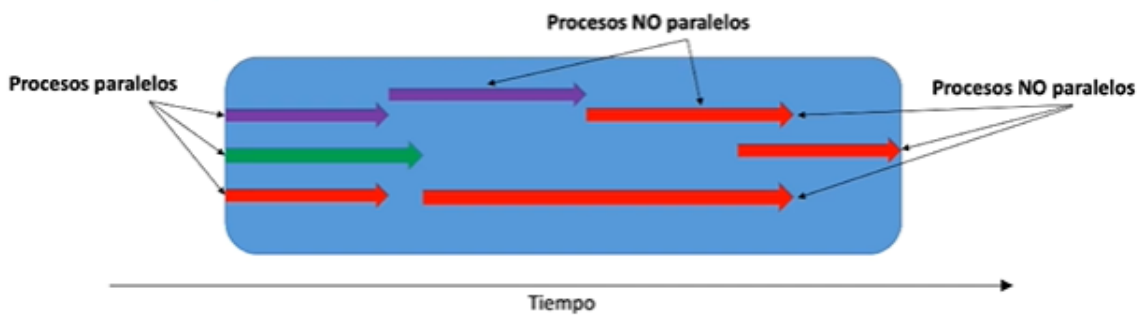


Ilustración 14. Procesos Paralelos. Obtenido de: [58]

.1.1.1. La concurrencia no implica el paralelismo

Si varios procesos conviven a la vez (son concurrentes), esto no implica que deben ejecutarse a la vez [62]. Por tanto, no necesariamente son paralelos [62].

La concurrencia y el paralelismo son dos conceptos diferentes, aunque muy relacionados. Concurrencia implica que varios procesos o tareas coexisten peor no necesariamente se ejecutan al mismo tiempo. Normalmente se desconoce el orden en el que se ejecutan [62].

Paralelismo implica que varios procesos o tareas se ejecutan al mismo tiempo. Normalmente se desconoce el orden en el que se ejecutan. Paralelismo implica concurrencia, pero concurrencia no implica paralelismo [64].

En conclusión, la computación paralela al igual que la concurrente presentan la capacidad de ejecutar múltiples procesos de forma simultánea [62] [64]. La gran diferencia es que la programación paralela utiliza múltiples procesadores o núcleos de la computadora. Mientras que la programación concurrente utiliza los hilos dentro de cada núcleo del procesador y su uso depende del SO [62].

La comparación gráfica del tipo de computación concurrente y paralela se puede visualizar en la siguiente ilustración:

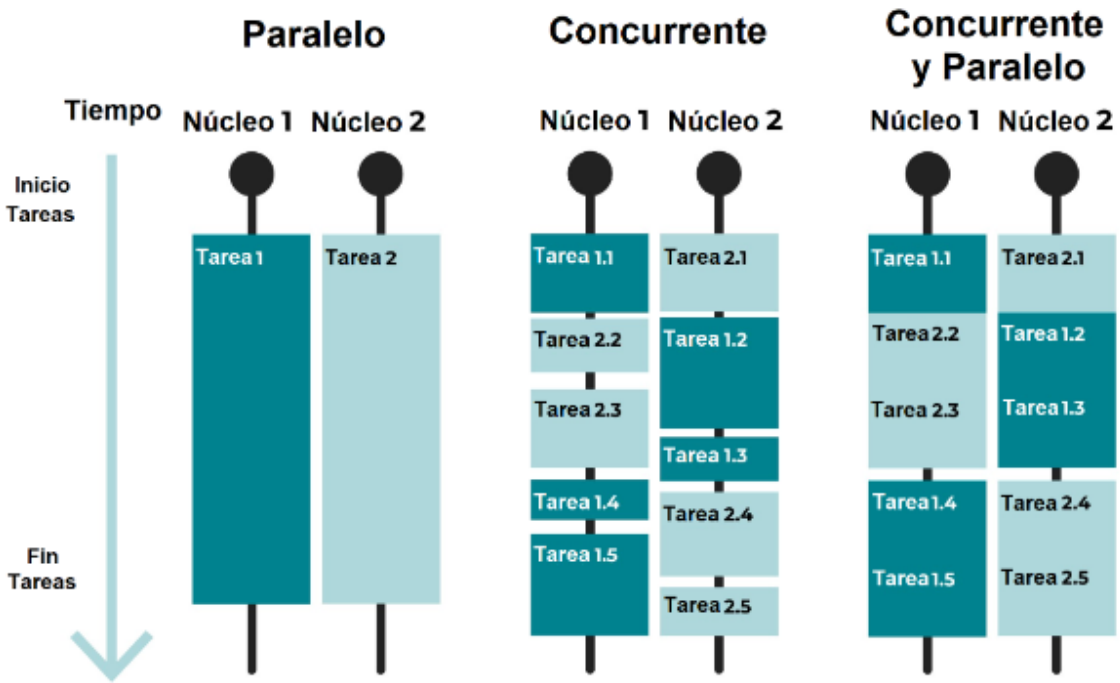


Ilustración 15. Comparación entre computación concurrente y paralela. Obtenida de: https://devopedia.org/images/article/339/6574.1623908671.jpg

4.6. Computación paralela vs computación distribuida

4.6.1. Computación distribuida

El objetivo es gestionar las actividades y procesos requeridos evitando una sobrecarga a un solo computador [65] [66]. Para una correcta implementación y uso de la computación distribuida se requiere del diseño de una arquitectura de red [6], [12], [65].

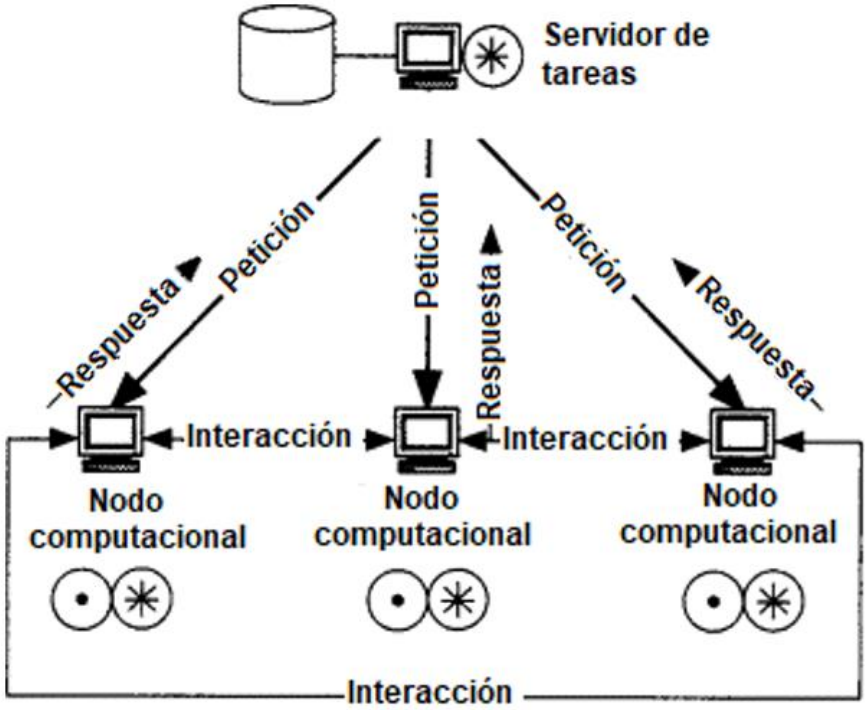


Ilustración 14. Arquitectura de red para la computación distribuida. Obtenida de: [66].

4.6.2. La computación distribuida implica al paralelismo

La computación distribuida presenta características y definiciones similares al paralelismo [6], [12]. La principal diferencia de la computación distribuida frente a la paralela es la utilización de múltiples computadores para solucionar un problema [6], [12].

La comparación entre estos tipos de computación, pueden visualizarse en la siguiente ilustración:

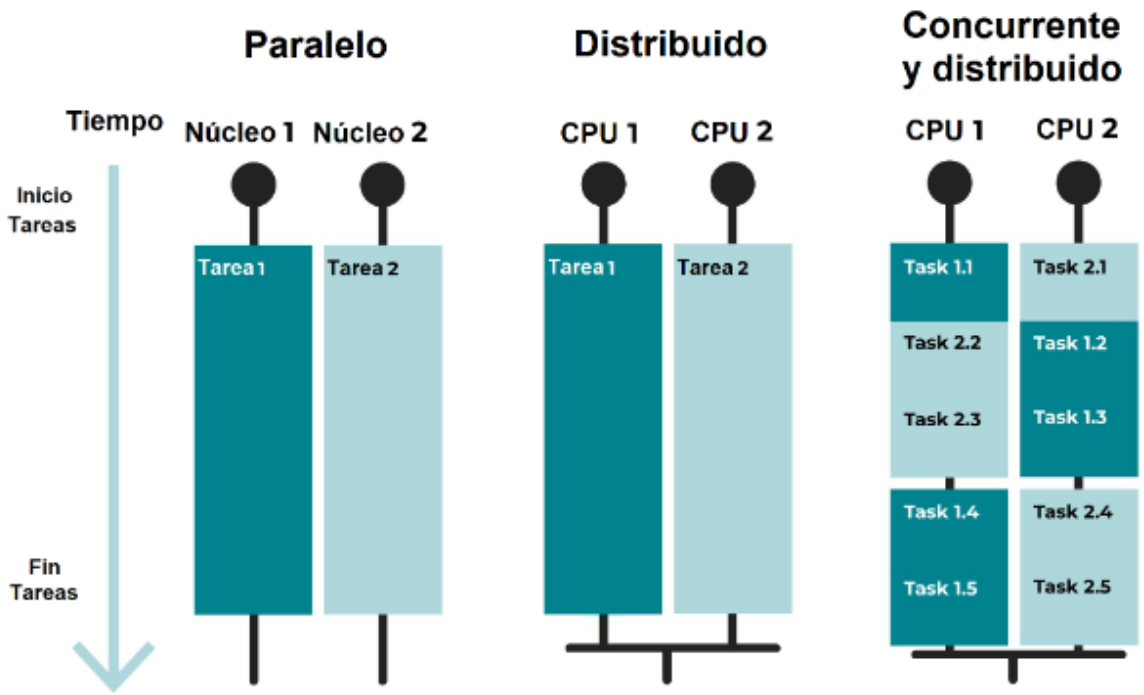


Ilustración 16. Comparación entre computación distribuida y paralela. Realizada por: Autores del documento. Modificada a partir de la Ilustración 13.

4.7. Computación concurrente vs paralela vs distribuida

Mediante el estudio de los tipos de computación, se puede concluir en que la computación paralela pertenece a la concurrente y a su vez la distribuida pertenece a la paralela. Esto debido a que cualquier proceso que involucre la ejecución de múltiples procesos de forma simultánea se considera concurrente.

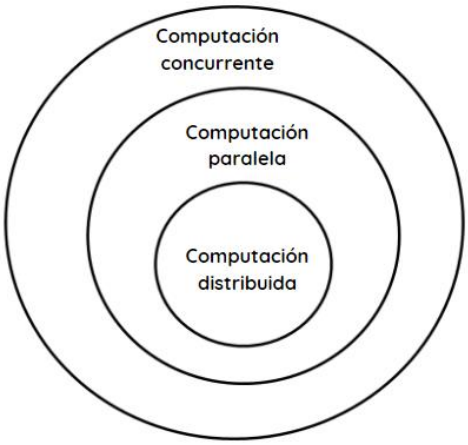


Ilustración 17. Implicación en tipos de computación concurrente, paralela y distribuida. Realizada por: Autores del documento.

Y la programación distribuida implica a la paralela, puesto que se basan en los mismos principios y características con la diferencia de la utilización de procesadores

ubicados en diferentes computadores y lugares. Finalmente, se demuestra el funcionamiento de cada tipo de computación basado en sus características en la siguiente ilustración:

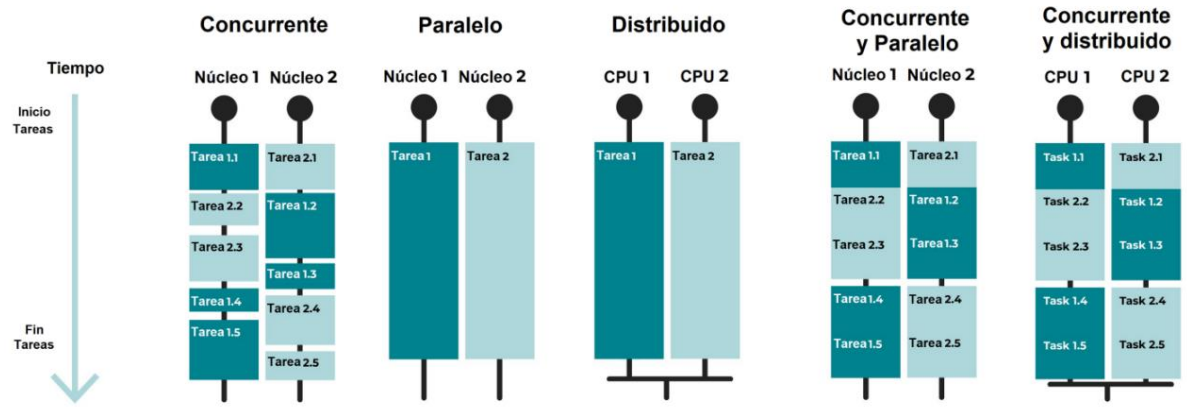


Ilustración 18. Comparación entre computación concurrente, paralela y distribuida. Realizada por: Autores del documento. Modificada a partir de la Ilustración 13.

5. Conclusión

La computación paralela es una buena práctica ya que se justifica por su capacidad de reducir el tiempo total de ejecución, abordar problemas complejos y permite la ejecución simultanea de tareas.

Actualmente, las limitaciones tecnológicas en la transmisión de datos y la velocidad de los procesadores actuales, la transición al cómputo paralelo se convierte en la única manera de mejorar el rendimiento de las computadoras. Sin embargo, la programación para computación paralela es más compleja, requiriendo coordinación eficiente y comunicación entre múltiples subtareas.

6. Referencias

[1] A. D. Kshemkalyani and M. Singhal, “Distributed Computing: Principles, Algorithms, and Systems,” 2008.

[2] Zankoya Zaxo, Duhok Polytechnic University, IEEE Computational Intelligence Society. Iraq Chapter., IEEE Communications Society. Iraq Chapter., and Institute of Electrical and Electronics Engineers, *International Conference on Advanced Science and Engineering : book of abstracts & program book : October 9-11, 2018*. 2018.

[3] D. Kirk and W. Hwu, *Programming massively parallel processors : a hands-on approach*. Morgan Kaufmann Publishers, 2010.

[4] Vinicius Fulber-Garcia, “Differences Between Core and CPU | Baeldung on Computer Science.” Accessed: Dec. 19, 2023. [Online]. Available: https://www.baeldung.com/cs/core-vs-cpu

[5] A. D. Kshemkalyani and M. Singhal, “Distributed Computing: Principles, Algorithms, and Systems,” 2008.

[6] D. Kirk and W. Hwu, *Programming massively parallel processors : a hands-on approach*. Morgan Kaufmann Publishers, 2010.

[7] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, “A survey on parallel computing and its applications in data-parallel problems using GPU architectures,” *Commun Comput Phys*, vol. 15, no. 2, pp. 285–329, 2014, doi: 10.4208/CICP.110113.010813A.

[8] México DF, “INSTITUTO POLITÉCNICO NACIONAL UNIDAD PROFESIONAL INTERDISCIPLINARIA DE INGENIERIA Y CIENCIAS SOCIALES Y ADMINISTRATIVAS Sección de Estudios de Posgrado e investigación P R E S E N T A JESÚS ANTONIO ALVAREZ CEDILLO,” 2006.

[9] M. A. Socorro, “Transdisciplinariedad: Una Mirada desde la Educación Universitaria,” *Revista Scientific*, vol. 3, no. 10, pp. 278–289, Nov. 2018, doi: 10.29394/SCIENTIFIC.ISSN.2542-2987.2018.3.10.15.278-289.

[10] David Orellana Martın, “Desarrollo de nuevas tecnicas a traves de modelos de computación bioinspirados,” 2019.

[11] Husnain Saeed, “The Search for Efficient Algorithms: P vs. NP and Heuristics | by Husnain Saeed | Medium.” Accessed: Dec. 19, 2023. [Online]. Available: https://husnainsaeed000.medium.com/the-search-for-efficient-algorithms-p-vs-np-and-heuristics-8c828772b0e3

[12] OSCAR FRANCISCO PEREDO ANDRADE, “IMPLEMENTACION DE UN METODO DE PROGRAMACIÓN SEMIDEFINIDA USANDO COMPUTACION PARALELA,” 2010.

[13] J. María Fernández, “Computación paralela y clústeres de cálculo (Parallel computing and computer clusters),” 2014. [Online]. Available: https://www.researchgate.net/publication/282606459

[14] C. Bischof, Jülich Supercomputing Centre, and ParCo 2007 Jülich, *Parallel computing: architectures, algorithms and applications [proceedings of the International Conference ParCo 2007, Jülich Supercomputing Centre]*. 2014.

[15] K. Asanovic *et al.*, “A view of the parallel computing landscape,” *Commun ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009, doi: 10.1145/1562764.1562783.

[16] U. Nacional de La Matanza, B. Aires, M. A. Daniel Giulianelli, I. A. Rocío Rodríguez, and I. M. Pablo Vera, “ADAPTACION DE PROCODI PARA COMPUTACION PARALELA,” 2008.

[17] J. Aguilar, “Introducción a la Computación Paralela,” 2004. [Online]. Available: https://www.researchgate.net/publication/267367623

[18] H. Hoeger, “Introducción a la Computación Paralela,” 2006.

[19] D. Padua, “Encyclopedia of Parallel Computing,” 2011, Accessed: Dec. 20, 2023. [Online]. Available: www.springer.com

[20] Zamora-Gomez-Antonio, “Computación Paralela,” 2001.

[21] S. Juliana, M. Niño, and S. Roa Prada, “Aplicaciones de la Computación Paralela por medio de Clusters Propuesta de Investigación,” 2021, Accessed: Dec. 20, 2023. [Online]. Available: http://www.dccia.ua.es/cp/index.htm

[22] B. Barney and L. Computing, “Overview Concepts and Terminology Parallel Computer Memory Architectures Parallel Programming Models Designing

Parallel Programs Introduction to Parallel Computing," 2007, Accessed: Dec. 20, 2023. [Online]. Available: https://computing.llnl.gov/tutorials/parallel_comp/

[23] R. Melhem, "Introduction to parallel computing," 2017.

[24] A. Polze and P. Tröger, "Parallel Programming Concepts! ! Parallel Computing Hardware / ! Models / Connection Networks," 2001.

[25] J. M. Cámara, "PROGRAMACIÓN DE ARQUITECTURAS PARALELAS," 2016.

[26] mtosini, "001 Introduccion a las arquitecturas Paralelas [Modo de compatibilidad]," 2015.

[27] S. Nesmachnow, "COMPUTACIÓN DE ALTA PERFORMANCE Curso 2022 Centro de Cálculo," 2022.

[28] V. E. Sonzogni and S. Fe, "Cálculo Científico con Computadoras Paralelas".

[29] M. De La Programací On Paralela, D. Giménez, and G. Giménez, "Paradigmas de Programací on Paralela Entornos de Programací on Paralela Introduccí on a la Computací on Paralela," 2018.

[30] J. María Fernández, "Computación paralela y clústeres de cálculo (Parallel computing and computer clusters)," 2018.

[31] C. Ernemann, V. Hamscher, U. Schwiegelshohn, R. Yahyapour, and A. Streit, "On Advantages of Grid Computing for Parallel Job Scheduling," 2002.

[32] H. J. Siegel, L. Wang, V. P. Roychowdhury, and M. Tan, "Computing with Heterogeneous Parallel Machines: Advantages and Challenges," 2004.

[33] V. Dilip Thoke, "THEORY OF DISTRIBUTED COMPUTING AND PARALLEL PROCESSING WITH ITS APPLICATIONS, ADVANTAGES AND DISADVANTAGES," 2015.

[34] R. S. Yadava, "Review of the Parallel Programming and its Challenges on the Multicore Processors," *Asian Journal of Computer Science and Technology*, vol. 4, no. 1, pp. 8–13, 2015.

[35] S. D. L. Martins, C. C. Ribeiro, and N. Rodriguez, "Parallel Computing Environments," 2001.

[36] P. Krastev, “INTRODUCTION TO PARALLEL COMPUTING,” 2016.

[37] Mike Flynn, “Parallel Processing SSC-0114 ARQUITETURA DE COMPUTADORES,” 2017.

[38] Robert van Engelen, “Parallel Programming Models HPC Prof. Robert van Engelen,” 2017.

[39] Ōyō Butsuri Gakkai., Denshi Jōhō Tsūshin Gakkai (Japan), IEEE Electron Devices Society. Japan Chapter., and IEEE Electron Devices Society. Kansai Chapter., *2008 International Conference on Simulation of Semiconductor Processes and Devices : SISPAD 2008 : September 9-11, 2008, Yumoto Fujiya Hotel, Hakone, Japan*. IEEE Xplore, 2008.

[40] Y. Sun, Y. Gu, B. Su, and X. Wang, “The research on parallel algorithm of A-order in binary tree based on PRAM model,” in *Proceedings of the 1st International Workshop on Education Technology and Computer Science, ETCS 2009*, 2009, pp. 349–353. doi: 10.1109/ETCS.2009.605.

[41] G. Kim *et al.*, “Adjustable voltage dependent switching characteristics of PRAM for low voltage programming of multi-level resistances.”

[42] Coromoto Leon, “(PDF) Diseño e implementación de lenguajes orientados al modelo PRAM.” Accessed: Dec. 20, 2023. [Online]. Available: https://www.researchgate.net/publication/39379494_Diseno_e_implementacion_de_lenguajes_orientados_al_modelo_PRAM

[43] C. C. L. Wang and D. Manocha, “Efficient boundary extraction of BSP solids based on clipping operations,” *IEEE Trans Vis Comput Graph*, vol. 19, no. 1, pp. 16–29, 2013, doi: 10.1109/TVCG.2012.104.

[44] Q. Luo, Jiang. Yi, Chen. Bin, and IEEE Technology Management Council., *2009 ISECS International Colloquium on Computing, Communication, Control, and Management : Sanya, China, August 8-9, 2009*. IEEE, 2009.

[45] J. Gao, P. Liu, X. Kang, L. Zhang, and J. Wang, “PRS: Parallel relaxation simulation for massive graphs,” *Computer Journal*, vol. 59, no. 6, pp. 848–860, Jun. 2016, doi: 10.1093/comjnl/bxu159.

[46] Institute of Electrical and Electronics Engineers, *2018 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon) : 3-4 Oct. 2018.* 2019.

[47] T. Groves, S. K. Gutierrez, and D. Arnold, “A LogP extension for modeling tree aggregation networks,” in *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, Institute of Electrical and Electronics Engineers Inc., Oct. 2015, pp. 666–673. doi: 10.1109/CLUSTER.2015.117.

[48] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, “A survey on parallel computing and its applications in data-parallel problems using GPU architectures,” *Communications in Computational Physics*, vol. 15, no. 2. Global Science Press, pp. 285–329, 2014. doi: 10.4208/cicp.110113.010813a.

[49] IEEE Signal Processing Society and IEEE Circuits and Systems Society, *SiPS 2018 : proceedings of the IEEE International Workshop on Signal Processing Systems : 21-24 October 2018, Cape Town, South Africa*. 2018.

[50] George Williamsv, “Scaling Deep Learning: Highlights From The Startup.ml Workshop.” Accessed: Dec. 20, 2023. [Online]. Available: https://www.linkedin.com/pulse/scaling-deep-learning-highlights-from-startupml-george-williams/

[51] J. L. Nicolas Dilley, “An Empirical Study of Messaging Passing Concurrency in Go Projects,” 2019.

[52] Fernando Pérez Costoya – José Mª Peña Sánchez Mª de los Santos Pérez Hernández, “Comunicación basado en huecos,” 2018.

[53] J. M. Alonso, “Programación de aplicaciones paralelas con MPI (Message Passing Interface)”.

[54] “D A C CEPBA Modelos de Programación Paralela”.

[55] F. A. Bermejo, “EDITORIAL UNIVERSITAT POLITÈCNICA DE VALÈNCIA,” 2018.

[56] Sena Kılıçarslan, “C# Memory Management - Part 1. In this article, I want to mention how… | by Sena Kılıçarslan | C# Programming | Medium.” Accessed:

Dec. 20, 2023. [Online]. Available: https://medium.com/c-programming/c-memory-management-part-1-c03741c24e4b

[57] Fernando Cores, “SISTEMAS CONCURRENTES Y PARALELOS,” 2015.

[58] Dr. Marcelo Naiouf, “Tópicos de programación Concurrente y Paralela Clase 1-Conceptos básicos,” 2015.

[59] M. Rossainz López, “Programación Concurrente y Paralela,” 2019.

[60] Jeisson Hidalgo Céspedes, “Programación Paralela y Concurrente,” 2019.

[61] I. Rodero, C. Francesc, and G. Bernat, “Programación y computación paralelas.”

[62] J. María Fernández, “Computación paralela y clústeres de cálculo (Parallel computing and computer clusters),” 2014.

[63] D. Padua, “Encyclopedia of Parallel Computing,” 2011.

[64] N. Vervliet, L. De Lathauwer, “Data Fusion Methodology and Applications”, sciencedirect, 2019.

[65] N. Vervliet, L. De Lathauwer, “Numerical Optimization-Based Algorithms for Data Fusion”, Elsevier, 2019.

[66] Janusz Kowali, “MPI: The Complete Reference”, The MTI Press, 1996.

[67] C. R, J. M, P.L, “Introducción a la programación en CUDA”, 2016.

[68] Nvidia, “Cuda C++ Programming Guide”, CUDA, 2023.

7. Anexos

El material/recursos utilizados junto a los scripts, se encuentra en el siguiente repositorio de github:

https://github.com/Hecmi/ComputacionParalela

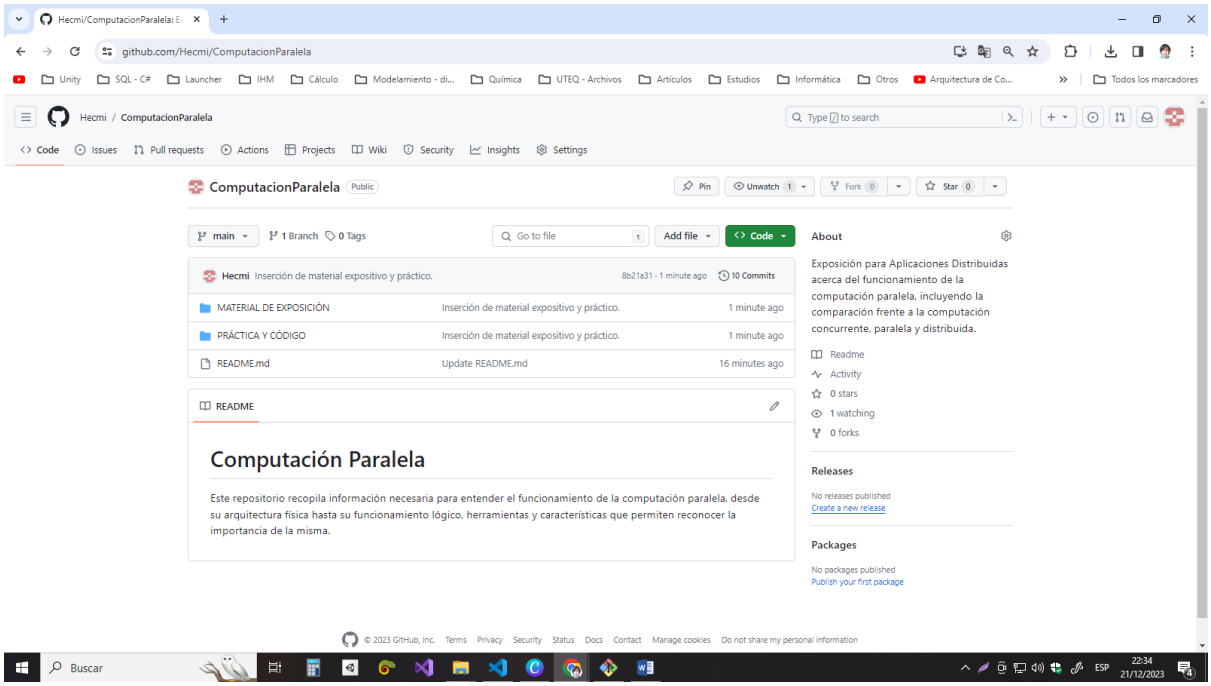


Ilustración 19. Anexo de repositorio en Github con los recursos expositivos y prácticos. Realizado por: Autores del documento.

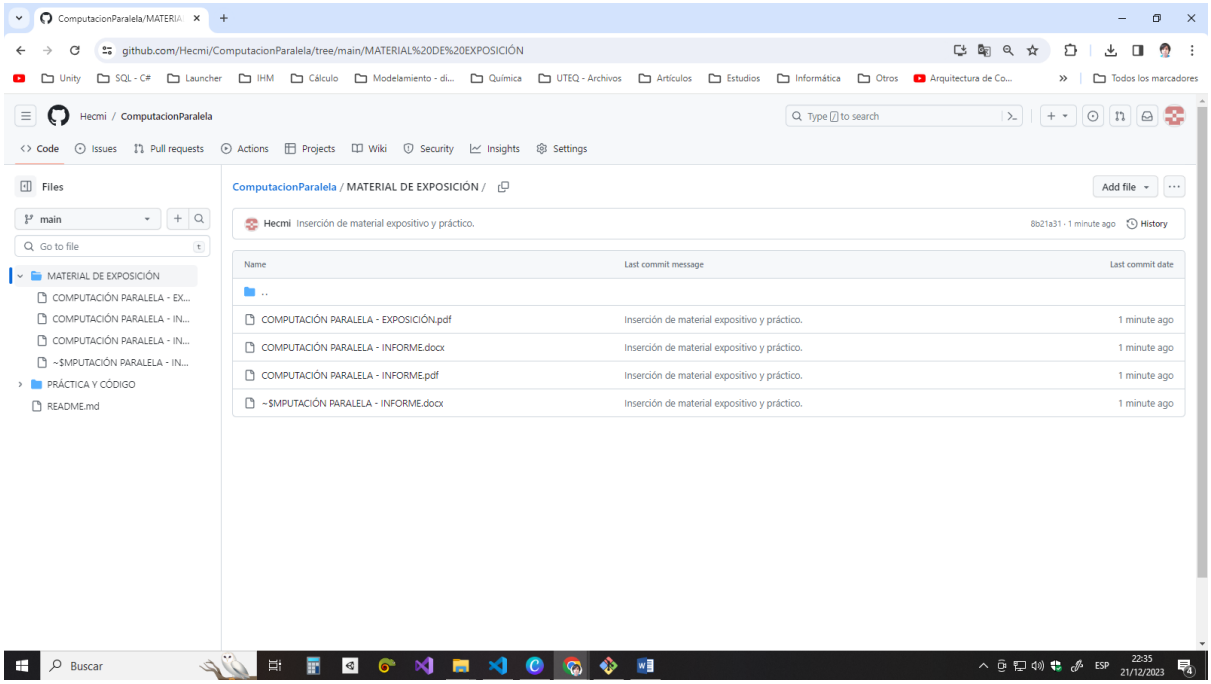


Ilustración 20. Anexo de repositorio en Github. Material expositivo. Realizado por: Autores del documento.

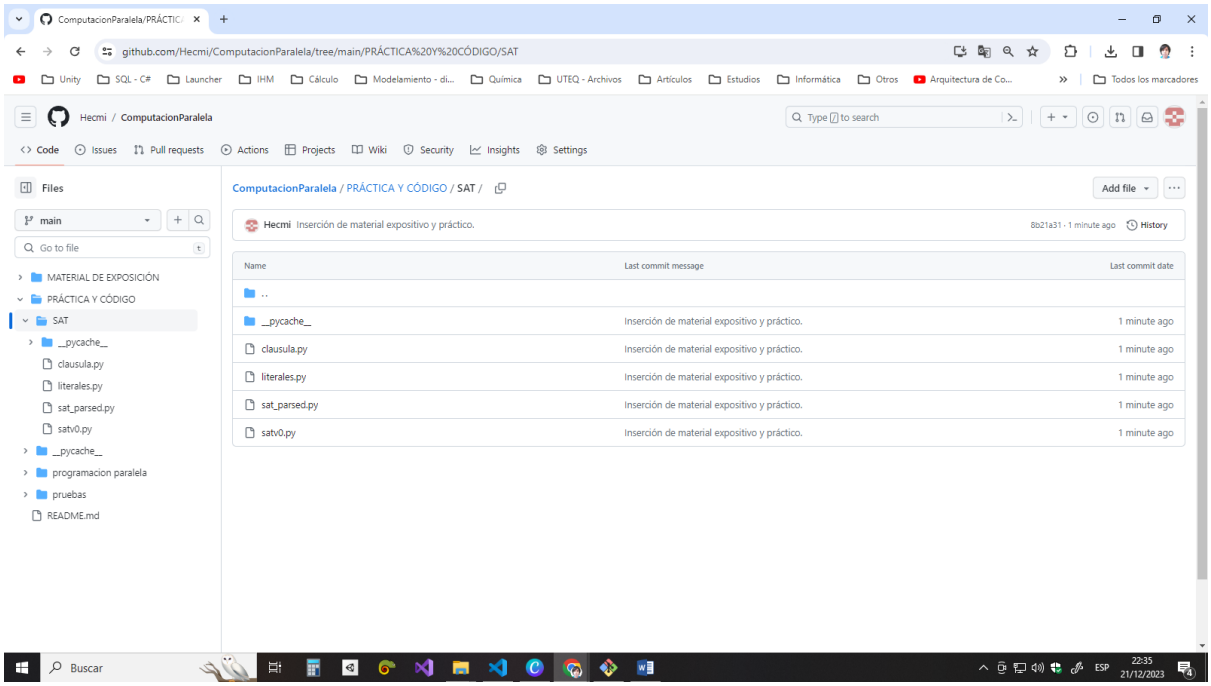


Ilustración 21. Anexo de repositorio en Github del material práctico, solucionador de problema NP para fórmulas de satisfacibilidad booleana. Realizado por: Autores del documento.

Código

Adicionalmente al enlace del repositorio, se adjunta una sección del código que será demostrado en la práctica. Como fue definido en el material presentado la computación paralela apunta a solucionar problemas P y NP de forma rápido. Así, se demostrará con un script para resolver problemas de satisfacibilidad booleana.

```
from clausula import csClausula
from literales import csLiteral

class solver:
    def __init__(self):
        self.literales_base = []
        self.clausulas_base = []

        self.literales = []
        self.clausulas = []

        self.backbones = []

        self.nivel_decision = 0
        self.cantidad_clausulas = 0
        self.clausulas_satisfechas = 0

        self.literales_asignados = 0

        #Variables globales para el proceso de backtrack:
        self.clausula_erronea = None

    def get_formula(self, ruta_formula):
        """
        Función que recibe la ruta donde se encuentra la fórmula a evaluar por el
        SAT Solver.
        Además, se encarga de preparar el ambiente; clausulas y literales necesarios
        para la evaluación
        """
```

```
        #Abrir el archivo del problema
        ruta_problema = open(ruta_formula, "r")
        lineas = ruta_problema.readlines()

        literales_temp = []

        #Obtener las clausulas que son equivalentes a cada fila del archivo
        for linea in lineas:
            sep_linea = linea.split()
            n_clausula = csClausula()

            for item in sep_linea:
                #c = comentarios, p = información del archivo
                if(item == "c" or item == "p"): break

                #Para facilitar la resolución más adeltante transformarlo a entero
                lit = int(item)

                #Añadir el literal a la claúsula y al resumen de los literales en fórmula
                n_clausula.literales.append(lit)
                if not abs(lit) in literales_temp:
                    literales_temp.append(abs(lit))

            self.clausulas.append(n_clausula)

        self.cantidad_clausulas = len(self.clausulas) - 1

        self.literales_base = literales_temp
        self.clausulas_base = self.clausulas

        self.transformar_literales(literales_temp)
        self.get_backbones()

    def transformar_literales(self, lista_literales):
        """
        Transformar los literales ingresados en valores secuenciales.
        Por ejemplo, -123 = |123| = 1, 98 = 2...
        """
        lista_transformada = []
        cantidad_literales = len(lista_literales) + 1

        for i in range(1, cantidad_literales):
            literal = csLiteral(i)
            lista_transformada.append(i)
            self.literales.append(literal)

        for i in range(len(self.clausulas)):
            clausula = self.clausulas[i]
            clausula = self.transformar_clausula(clausula, lista_literales, lista_transformada)

    def transformar_clausula(self, clausula, literal_base, literal_transformado):
        """
        Una vez se adaptan los literales de forma secuencial para el acceso rápido.
        Modificar la cláusula con la misma finalidad.
        """
        for i in range (len(clausula.literales)):
            literal_clausula = clausula.literales[i]

            for j in range(len(literal_base)):
                literal_normal = literal_base[j]
                literal_modificado = literal_transformado[j]

                if abs(literal_clausula) == literal_normal:
                    if literal_clausula < 0:
                        clausula.literales[i] = literal_modificado * -1
                    else:
                        clausula.literales[i] = literal_modificado
                    break

        return clausula

    def get_backbones(self):
        """
        Sub-implementación para la selección de los mejores literales
```

```
    durante la etapa de selección de variable o literal.
    """
    for clausula in self.clausulas:
        for literal in clausula.literales:
            self.literales[abs(literal) - 1].presencia += 1

            if len(clausula.literales) == 1:
                print(self.literales[abs(literal) - 1].valor)
                self.literales[abs(literal) - 1].es_unitario = True
            else:
                self.literales[abs(literal) - 1].es_unitario = False

def ordenar_literales(self):
    """
    Ordenar los literales basado en los backbones y si son unitarios
    (aparecen en una claúsula como única variable)
    """
    # literales_sin_asignacion = []

    # #Obtener los literales que no han sido asignado un valor
    # for literal in self.literales:
    #     if literal.asignacion == 0: literales_sin_asignacion.append(literal)

    # literales_sin_asignacion = self.ordernar_por_presencia(literales_sin_asignacion);

    # self.backbones = literales_sin_asignacion

def ordernar_por_presencia(self, literales):
    for i in range(len(literales)):
        for j in range(i + 1, len(literales)):
            if literales[i].presencia <= literales[j].presencia:
                temp = literales[i]
                literales[i] = literales[j]
                literales[j] = temp

    return literales

def verificar_literales_asignados(self):
    return self.literales_asignados == len(self.literales)

def pick_literal(self, nivel_decision):
    """
    Seleccionar el literal basado en técnicas de decisiones
    Se aplicará backbones.
    """
    for i in range(len(self.literales)):
        literal = self.literales[i]
        if literal.asignacion == 0:
            self.literales[i].asignacion = 1
            self.literales[i].nivel_de_decision = nivel_decision
            self.literales_asignados += 1
            break


def predecir_literal(self, literal):
    return 1 if literal > 0 else -1

def propagacion_unitaria(self, nivel_decision):
    literal_deducido = True

    while literal_deducido:
        literal_deducido = False

        for i in range(len(self.clausulas)):
            clausula = self.clausulas[i]

            ultimo_literal_sin_asignar = -1

            variables_sin_asignar = 0
            variables_falsificadas = 0

            clausula_satisfacida = False

            for literal in clausula.literales:
```

```
                    #Verificar si la clausula ya está satisfecha

                    lit = self.literales[abs(literal) - 1]

                    #Si el resultado es positivo la clausula fue satisfacida
                    if lit.asignacion * literal > 0:
                        clausula_satisfacida = True
                        break

                    #Sí el valor de la asignación es 0 no ha sido asignada
                    if lit.asignacion == 0:
                        ultimo_literal_sin_asignar = literal
                        variables_sin_asignar += 1

                    if lit.asignacion * literal < 0:
                        variables_falsificadas += 1

                    if variables_falsificadas == len(clausula.literales):
                        self.clausula_erronea = clausula
                        return False

                #Si falta exactamente 1 por asignar y la clausula no ha sido satisfecha
                #entonces es posible deducir el valor de esa variable faltante para cumplirla.
                if variables_sin_asignar == 1 and not clausula_satisfacida:
                    self.literales_asignados += 1
                    valor = self.predecir_literal(ultimo_literal_sin_asignar)
                    self.literales[abs(ultimo_literal_sin_asignar) - 1].asignacion = valor
                    self.literales[abs(ultimo_literal_sin_asignar) - 1].nivel_de_decision =
nivel_decision
                    self.literales[abs(ultimo_literal_sin_asignar) - 1].deducido_para_clausula = i
                    literal_deducido = True

        return True

    def backtrack(self, nivel_decision):
        #Obtener la clausula que causa el error.
        literales_en_mismo_nd = 0
        nueva_clausula = self.clausula_erronea
        print("CLAUSULA CONFLICTO: ", nueva_clausula.literales)
        self.print_literales()

        #Mientras la cantidad de literales en la clausula que se está adaptando
        #sea exactamente > 1 entonces, se debe repetir el proceso de verificación.
        while True:
            clausula_anidar = []
            literales_en_mismo_nd = 0

            #Sí la cantidad de literales en el mismo nivel de decisión es 1 entonces se
            #encontró la nueva clausula.

            for literal in nueva_clausula.literales:
                lit = self.literales[abs(literal) - 1]

                if lit.nivel_de_decision == nivel_decision: literales_en_mismo_nd += 1

                if lit.deducido_para_clausula != -1:
                    clausula_anidar = self.clausulas[lit.deducido_para_clausula]

            if literales_en_mismo_nd < 2: break
            print("ERROR EN  ", nueva_clausula.literales)
            print("ANIDAR ", clausula_anidar.literales)
            nueva_clausula.literales = self.reducir_clausula(nueva_clausula, clausula_anidar)
            print("REDUCIDA  ", nueva_clausula.literales)
            print(nueva_clausula.literales, literales_en_mismo_nd)


        #Obtener el menor nivel de decisión presente en la clausula aprendida
        backtrack_nivel_decision = nivel_decision
        nivel_decision_literales = []

        for literal in nueva_clausula.literales:
            lit = self.literales[abs(literal) - 1]
            nivel_decision_literales.append(lit.nivel_de_decision)
```

```
        #Recorrer la clausula aprendida y obtener el menor nivel de decisión
        for i in range(len(nivel_decision_literales)):
            for j in range(i + 1, len(nivel_decision_literales)):
                if nivel_decision_literales[i] <= nivel_decision_literales[j]
and  nivel_decision_literales[i] > -1:
                    backtrack_nivel_decision = nivel_decision_literales[i]

        #Reiniciar las acciones realizadas hasta el menor nivel de decisión encontrado.
        for literal in self.literales:
            if self.literales[literal.valor - 1].nivel_de_decision == backtrack_nivel_decision:
                self.literales[literal.valor - 1].asignacion = 0
                self.literales[literal.valor - 1].deducido_para_clausula = -1
                self.literales[literal.valor - 1].nivel_de_decision = -1

                self.literales_asignados -= 1

        self.clausulas.append(nueva_clausula)
        print("N_CLAUSULA ", nueva_clausula.literales)
        return backtrack_nivel_decision

    def solve(self, ruta_formula):
        """
        Método principal, encargado de recibir la ruta del problema para su respectiva
        conversión, adaptación y resolución.
        """
        self.get_formula(ruta_formula)

        if not self.propagacion_unitaria(self.nivel_decision):
            return False

        while not self.verificar_literales_asignados():
            #self.print_literales()
            self.nivel_decision += 1
            self.pick_literal(-1)

            if not self.propagacion_unitaria(self.nivel_decision):
                self.nivel_decision = self.backtrack(self.nivel_decision)

                print("BACK TO ", self.nivel_decision)
                self.propagacion_unitaria(self.nivel_decision)
                if self.nivel_decision < 0: return False

        return True

    def remover_literales_repetidos(self, clausula):
        clausula_depurada = []
        for i in clausula:
            if i not in clausula_depurada:
                clausula_depurada.append(i)
        return clausula_depurada

    def reducir_clausula(self, primera_clausula, segunda_clausula):
        #Hacer un vector general con todos los datos
        literales = primera_clausula.literales + segunda_clausula.literales

        literales = self.remover_literales_repetidos(literales)

        while(True):
            length = len(literales)
            found = False
            to_delete = -1
            for i in range(length):
                for j in range(i + 1, length):
                    if(literales[i] == -literales[j] and i != j):
                        found = True
                        to_delete = literales[i]
                        break
                if(found): break

            if(found):
                literales.remove(to_delete)
                literales.remove(-to_delete)
            else: break
```

```
        return literales

    def print_clausulas(self):
        for i in range(len(self.clausulas)):
            c = self.clausulas[i]
            for l in c.literales:
                print("CLAUSULA INDICE ", i, ' LITERAL ' , l)

    def print_literales(self):
        for i in range(len(self.literales)):
            c = self.literales[i]
            cb = self.literales_base[i]
            print("LITERAL: " , c.valor, " -> ", cb,  " LD: ", c.nivel_de_decision, " PRESENTE EN: ",
c.presencia, " ES UNITARIO: ", c.es_unitario, " ASIGNACION: ", c.asignacion, " POR CLAUSULA: ",
self.clausulas[c.deducido_para_clausula].literales if c.deducido_para_clausula >= 0 else 0)

    def print_literales_i(self, literales):
        for i in range(len(literales)):
            c = literales[i]
            print("LITERAL: " , c.valor,  " PRESENTE EN: ", c.presencia, " ES UNITARIO: ",
c.es_unitario)


#Inicializar una variable para ejecutar el verificador SAT.
solv =solver()
#Ejecutar el solucionador, indicando la ruta para la verificación de la fórmula o problema booleano.
resultado = solv.solve('./pruebas/uf250-02.cnf')

#Imprimir los literales para visualizar el resultado
solv.print_literales()
print("SAT" if resultado else "UNSAT")
```