

**GRADO EN INGENIERÍA INFORMÁTICA**

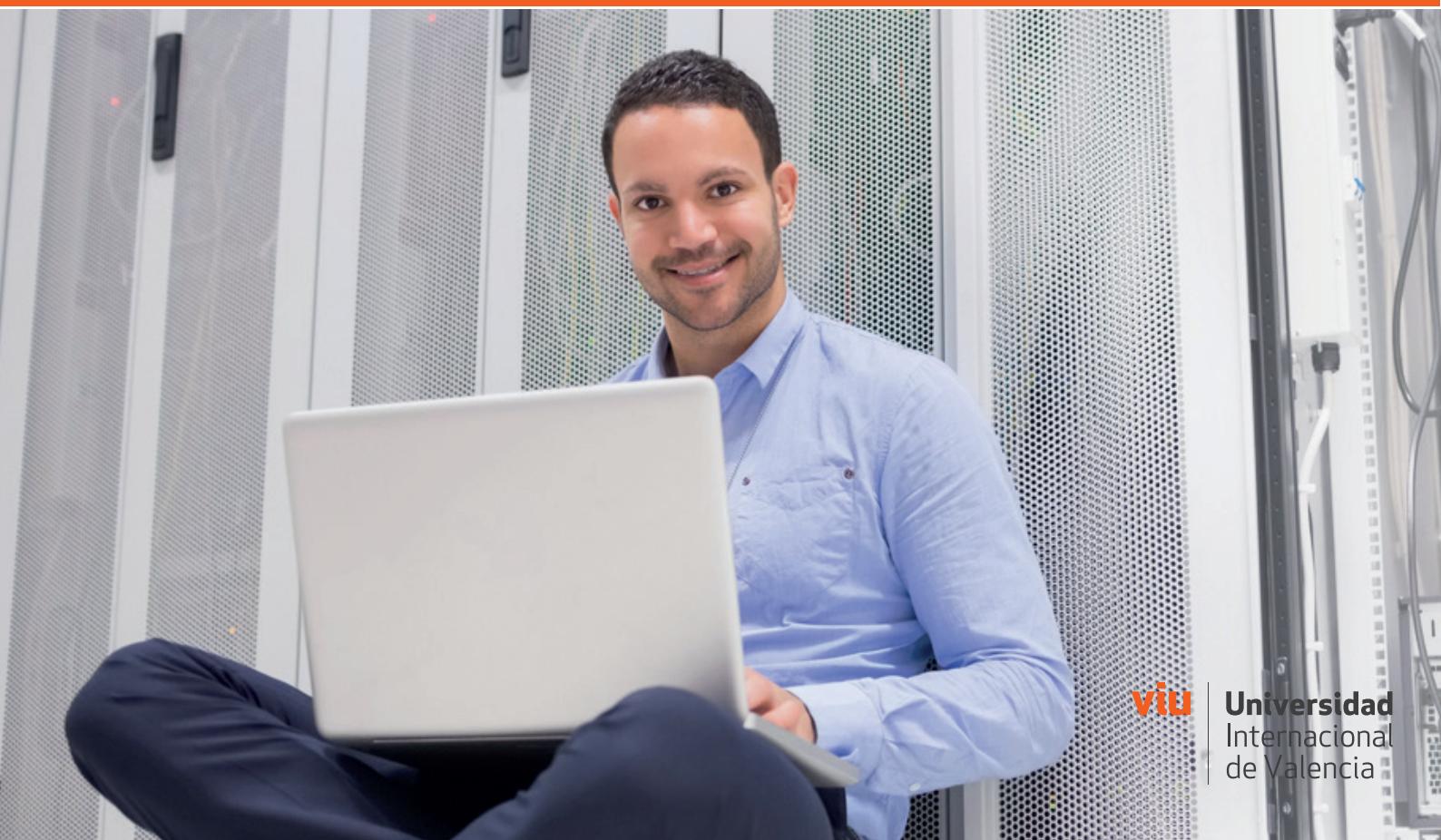
Módulo Común de la Rama de Informática

---

# PARALELISMO

---

**Yudith C. Cardinale Villarreal**





Este material es de uso exclusivo para los alumnos de la VIU. No está permitida la reproducción total o parcial de su contenido ni su tratamiento por cualquier método por aquellas personas que no acrediten su relación con la VIU, sin autorización expresa de la misma.

**Edita**

Universidad Internacional de Valencia

Grado en  
**Ingeniería Informática**

---

**Paralelismo**

Módulo Común de la Rama de Informática  
6 ECTS

---

**Yudith C. Cardinale Villarreal**

## Leyenda

---

**abc** **Glosario**  
Términos cuya definición correspondiente está en el apartado “Glosario”.

 **Enlace de interés**  
Dirección de página web.

---

## Índice

1. NOCIONES FUNDAMENTALES DE PARALELISMO.....	7
1.1. Diferencias entre computación secuencial, concurrente, paralela y distribuida.....	8
1.2. Importancia y objetivos de la programación paralela.....	11
1.3. Definiciones básicas.....	13
2. INTRODUCCIÓN A LAS ARQUITECTURAS PARALELAS .....	19
2.1. Taxonomía de Flynn.....	20
2.2. Clasificación de los sistemas MIMD .....	22
2.2.1. Sistemas MIMD con memoria compartida y conectados con buses .....	25
2.2.2. Sistemas MIMD con memoria compartida y conectados con conmutadores .....	27
2.2.3. Sistemas MIMD con memoria distribuida y conectados con buses .....	28
2.2.4. Sistemas MIMD con memoria distribuida y conectados con conmutadores .....	29
3. TÉCNICAS DE DISEÑO DE PROGRAMAS PARALELOS .....	33
3.1. Modelos de programas paralelos: SPMD, MPMD y maestro/esclavos.....	34
3.2. Diseño Metodológico .....	35
3.2.1. Particionamiento.....	37
3.2.2. Comunicación.....	39
3.2.3. Aglomeración.....	44
3.2.4. Mapeo.....	46
3.3. Solapamiento de comunicación con cómputo.....	50
4. ANÁLISIS DE APLICACIONES PARALELAS.....	51
4.1. Aspectos de rendimiento de las arquitecturas paralelas.....	52
4.1.1. Métricas de desempeño para sistemas de computación.....	52
4.1.2. Consideraciones de efectividad-coste .....	53
4.1.3. Técnicas de análisis de desempeño de sistemas computacionales .....	54
4.1.4. Otras métricas de rendimiento populares para arquitecturas computacionales.....	55
4.1.5. Comparación de dos arquitecturas en base al rendimiento .....	57
4.2. Aspectos de rendimiento de algoritmos paralelos.....	58
4.2.1. Métricas para la evaluación del rendimiento de algoritmos paralelos .....	58
4.2.2. Ley de Amdahl .....	62

---

4.2.3. Ley de Gustafson-Barsis.....	64
4.2.4. Factores que afectan el rendimiento de los programas paralelos.....	65
5. HERRAMIENTAS DE PROGRAMACIÓN PARALELA.....	71
5.1. Modelos de comunicación en programación paralela.....	71
5.1.1. Comunicación directa simétrica y directa asimétrica .....	73
5.1.2. Comunicación indirecta.....	74
5.1.3. Comunicación síncrona y asíncrona.....	74
5.1.4. Comunicación transitoria y persistente .....	75
5.2. Programación paralela con memoria distribuida.....	75
5.2.1. Librerías MPI .....	76
5.2.2. OpenMPI: una implementación de MPI .....	76
5.3. Programación paralela con memoria compartida.....	84
5.3.1. OpenMP .....	84
5.3.2. Combinando MPI y OpenMP.....	94
5.4. Otras técnicas: CUDA y OmpSs.....	95
GLOSARIO.....	97
ENLACES DE INTERÉS .....	101
BIBLIOGRAFÍA.....	103

## 1. Nociones Fundamentales de Paralelismo

Muchos de los problemas en ciencia e ingeniería se resuelven con aplicaciones que requieren de la ejecución de muchas tareas y análisis de gran volumen de datos, lo que conlleva a la exigencia de gran poder de cómputo para ser ejecutadas en un tiempo razonable. En respuesta a esta necesidad, durante la última década se ha presenciado un gran crecimiento en las capacidades y en el rendimiento de los sistemas computacionales. Estos avances se deben a dos tipos de cambios: tecnológicos y arquitecturales. Los cambios tecnológicos han permitido aumentar las capacidades en los **circuitos VLSI**, incrementar las velocidades de los relojes, incrementar la velocidad con la que se realizan las funciones de un circuito. Además, desde el punto de vista tecnológico, se ha producido una gran expansión en el uso de **arquitecturas paralelas** con memoria distribuida, en especial los **clusters de procesadores** personales, que ofrecen este poder de cómputo a través de la interacción de varios procesadores (desde decenas hasta miles). Los cambios arquitecturales se basan fundamentalmente en nuevas organizaciones de los componentes que permiten realizar nuevas funciones o las funciones anteriores con mayor velocidad.

Ambos aspectos, arquitecturales y tecnológicos, están estrechamente ligados entre sí; mejoras tecnológicas promueven cambios arquitecturales y éstos, a su vez, demandan más capacidades a los circuitos. Uno de los aspectos que mejor representan estos cambios es sin lugar a duda la aparición de las arquitecturas paralelas (y de ahí la computación paralela). **Un computador paralelo es una colección de elementos de procesamiento que se comunican y cooperan para resolver problemas grandes de manera rápida.** La **programación paralela** se refiere a la forma de construir las aplicaciones para que trabajen en computadores paralelos y disminuir el tiempo de ejecución en varios órdenes de magnitud y la plataforma computacional donde se ejecuta se conoce

como sistema paralelo o sistema distribuido. De ahí surge la **computación de alto rendimiento** (*HPC – High Performance Computing*), que promueve el **modelo de programación paralela**.

Así, la computación de alto rendimiento implica el uso de los computadores más poderosos del momento (**supercomputadores**) para la resolución de problemas que requieren mucho cómputo y análisis de grandes volúmenes de datos. La programación paralela es la que permite que las aplicaciones de alto rendimiento usen muchos procesadores simultáneamente. En la mayoría de los casos está asociada con computación para investigación científica en áreas como teoría de números, ciencias de la vida, predicción del tiempo, petroquímica (dinámica de fluidos, simulación, modelamiento), biología digital y aplicaciones militares.

La programación paralela involucra muchos aspectos que no se presentan en la programación secuencial. El diseño de un programa paralelo tiene que considerar entre otras cosas: el tipo de arquitectura sobre la cual se va a ejecutar el programa, las necesidades de tiempo y espacio que requiere la aplicación, el modelo de programación paralelo adecuado para implantar la aplicación y la forma de coordinar y comunicar a diferentes procesadores para que resuelvan un problema común.

Antes de entrar en detalle en los tópicos de la programación paralela, se revisarán las diferencias entre programación secuencial, concurrente, paralela y distribuida. Luego se abordarán los conceptos fundamentales de la computación de alto rendimiento o computación paralela. Más adelante se revisarán las arquitecturas paralelas y diferentes herramientas que apoyan el modelo de programación paralela.

## 1.1. Diferencias entre computación secuencial, concurrente, paralela y distribuida

Originalmente, en los inicios de la era computacional, los programas fueron escritos para cómputo secuencial (serial), lo que implica que:

- Son ejecutados en un único computador, usando solo un CPU.
- Los problemas son resueltos por una serie de instrucciones ejecutadas una después de la otra por un solo CPU.
- En un instante de tiempo determinado, solo una instrucción de ese programa secuencial se está ejecutando.

Aún hoy en día, muchos problemas computacionales se resuelven con este tipo de programas. La Figura 1 ilustra la ejecución de un programa secuencial.

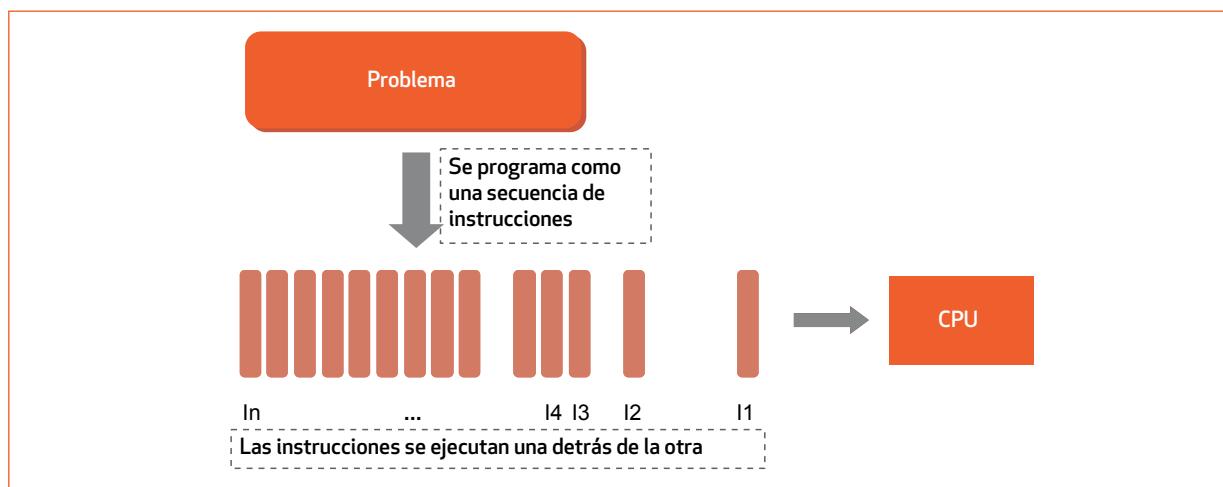


Figura 1. Ejecución de programa secuencial.

Posteriormente, con la evolución y avances de las tecnologías computacionales tanto a nivel de software como hardware, se da paso a la programación concurrente. La concurrencia, o ejecución simultánea de tareas, aparece inicialmente como una necesidad de los sistemas operativos: un solo procesador de gran capacidad debía repartir su tiempo entre muchas actividades (procesos, usuarios) con la idea de reducir su tiempo ocioso. **Cuando un proceso está realizando tareas de acceso a datos (entrada o salida de datos), no utiliza el CPU, por lo tanto, ese tiempo puede ser usado por otro proceso que sí requiere tiempo de CPU.** Esto se traduce en uso eficiente de los recursos, frente a la capacidad de multi-programación.

Los sistemas operativos son clases de programas inherentemente concurrentes porque las actividades que controlan, o reflejan, son en sí mismas concurrentes. Sin embargo, la concurrencia también puede utilizarse como una herramienta de programación para mejorar la eficiencia de los programas. Es decir, que los desarrolladores de programas tienen la capacidad de diseñar e implementar sus algoritmos, programas, aplicaciones o sistemas, usando mecanismos de concurrencia.

Así tenemos que las tareas (procesos o hilos) concurrentes pueden **competir** o **colaborar** entre sí por los recursos del sistema. Por tanto, se requiere que los sistemas de computación provean herramientas para implementar la colaboración y sincronización entre las tareas y resolver los problemas de comunicación y sincronización, que pueden presentarse.

Los beneficios de la concurrencia se ven reflejados en varios aspectos:

- **Compartimiento de recursos:** varios usuarios pueden estar interesados en el compartimiento de recursos lógicos (un archivo compartido) o físicos (accesos concurrentes a dispositivos que así lo permitan).
- **Velocidad de ejecución:** al subdividir un programa en procesos o hilos, éstos se pueden "repartir" entre procesadores o gestionar en un único procesador según su importancia, logrando un mejor desempeño en la ejecución.

- **Mejorar modularidad:** existen algunos problemas cuya solución es más fácil utilizando el enfoque concurrente, pues permite mejor modularidad al dividir las funciones del sistema o aplicación que resuelve el problema, en procesos separados. Algunos ejemplos de ese tipo de problemas son la captura de datos, análisis y actuación, sistemas de tiempo real.
- **Conveniencia:** cada usuario puede tener muchas tareas que pueden trabajar al mismo tiempo. Por ejemplo, un usuario puede editar, compilar e imprimir en paralelo. Además, existen aplicaciones que por naturaleza son concurrentes. Por ejemplo, programas que modelan sistemas físicos con autonomía (simulaciones), sistemas gestores de bases de datos, en el que cada usuario puede ser representado por un proceso.
- **Tecnologías web:** servicios que son capaces de atender varias peticiones concurrentemente; tales como servidores web, servidores de *chat*, servidores de correo electrónico.

Aun cuando el concepto de concurrencia y sus beneficios están claros, hoy en día suele haber confusión entre varios términos que implican ejecución simultánea de procesos: **programación concurrente**, **programación paralela** y **programación distribuida**.

En general, todos estos conceptos tienen en común la noción de ejecución simultánea de tareas: dos o más tareas son concurrentes si, en un momento dado, cada una está en algún punto entre su punto de comienzo y su punto de finalización. A este nivel de conceptualización solo importa el **qué**, independiente de **cómo** se implementa tal simultaneidad y de la arquitectura física donde se ejecutarán los procesos. La Figura 2 muestra un esquema de comparación entre estos tres conceptos.

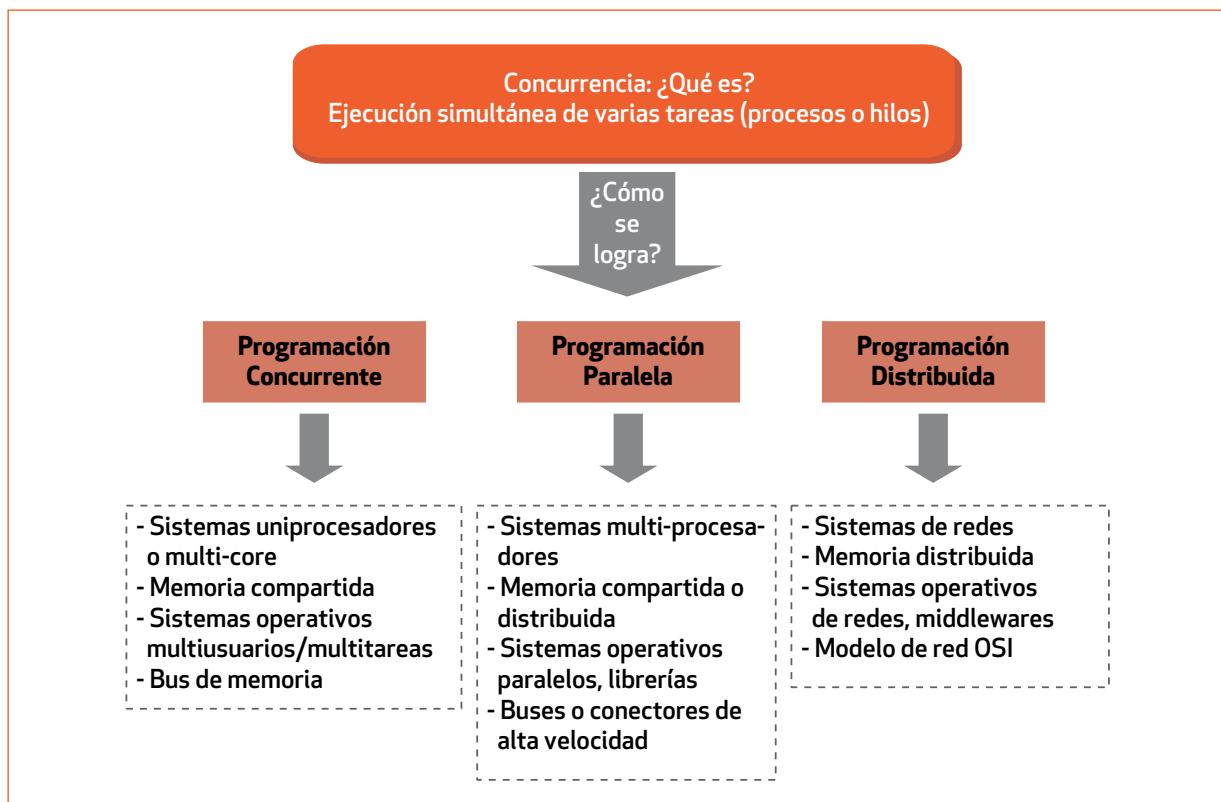


Figura 2. Esquema de comparación entre concurrencia, paralelismo y distribución.

A nivel del **qué**, los tres términos comparten la noción de ejecución simultánea. Se diferencian a nivel del **cómo**:

- **Programación concurrente:** se desarrolla sobre sistemas de computación de un procesador o **procesadores multi-núcleos** (*multi-core*), con memoria compartida y con sistemas operativos multi-usuarios y multi-tareas (como linux, windows, MacOs). Los programas concurrentes son un conjunto de procesos secuenciales que son ejecutados bajo un **parallelismo abstracto**, es decir en un instante de tiempo, un único proceso se está ejecutando y su ejecución es intercalada con la ejecución de otros procesos. La concurrencia se da mediante el uso de esquemas de interrupción que permiten la suspensión de la ejecución de un proceso para que continúe con la ejecución de otro proceso.
- **Programación paralela:** se implementa en sistemas de computación de múltiples procesadores con memoria compartida o distribuida (computadores paralelos, supercomputadores, *cluster* de computación), con sistemas operativos paralelos o librerías especializadas para pase de mensajes. Con la programación paralela es posible que varios procesos se puedan ejecutar al mismo tiempo usando varios procesadores (pocos o muchos). La red de comunicación de estas arquitecturas, por lo general, está conformada por buses o conectores de alta velocidad.
- **Programación distribuida:** se desarrolla en arquitecturas de redes, que implican múltiples procesadores distribuidos, con memoria distribuida, sobre sistemas operativos de redes o *middlewares*. La red de interconexión es basada en la filosofía de las capas de red del modelo OSI (*Open System Interconnection*).

Esta asignatura está enfocada principalmente en la programación paralela. En las siguientes secciones profundizaremos en los conceptos relacionados.

## 1.2. Importancia y objetivos de la programación paralela

La computación paralela implica el uso simultáneo de múltiples recursos computacionales para resolver un problema computacional. Así, un programa paralelo se caracteriza por:

- Su ejecución ocupa varios CPUs simultáneamente.
- Los problemas son resueltos por un conjunto de tareas que se ejecutan simultáneamente en varios CPUs.
- En un instante de tiempo determinado, varias tareas de ese programa paralelo se están ejecutando.

Las plataformas computacionales que soportan la computación paralela pueden incluir:

- Un único computador con múltiples procesadores.
- Un número arbitrario de computadores conectados por una red.
- Una combinación de ambos.

La Figura 3 ilustra la ejecución de un programa paralelo.

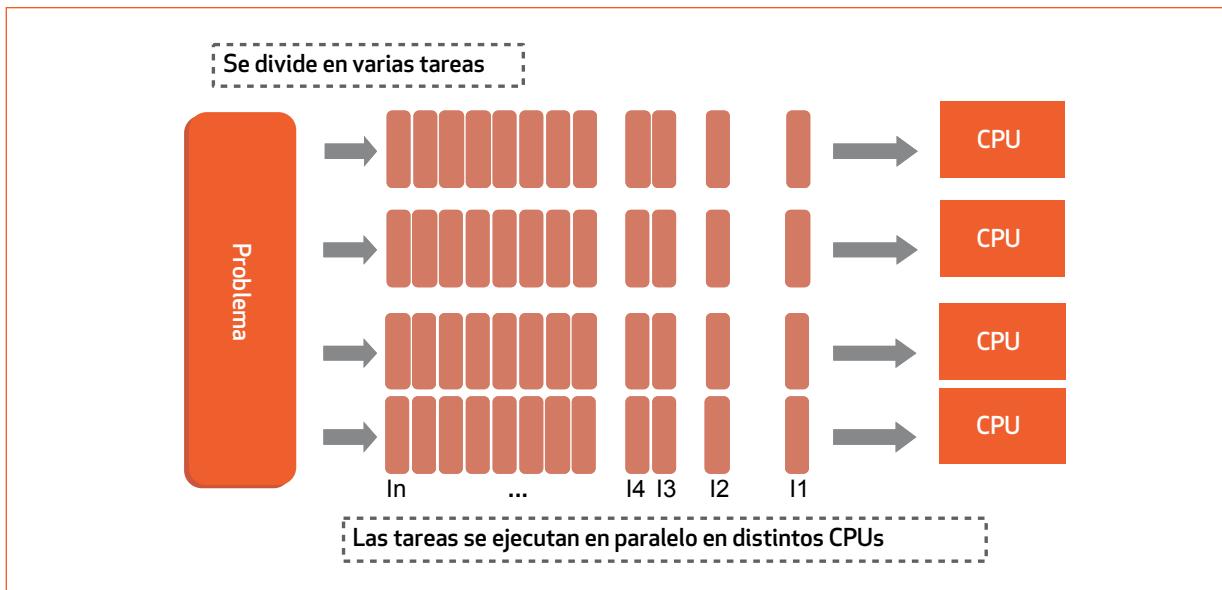


Figura 3. Ejecución de un programa paralelo.

En los últimos años, el uso extensivo de la computación paralela ha estado ligado a varios hechos:

- **Necesidad de mayor poder de cómputo:** independientemente de que la potencia de los procesadores aumente, siempre habrá un límite que dependerá de la tecnología del momento. Para aumentar el poder de cómputo, además de los progresos tecnológicos que permiten aumentar la velocidad de cálculo, se requieren nuevos paradigmas basados en el cálculo paralelo. Es decir, una forma de aumentar la velocidad es usando varios procesadores al mismo tiempo. Así un problema es dividido en partes, cada una de las cuales es ejecutada en paralelo en diferentes procesadores.
- **Una mejor relación coste/rendimiento:** es muy importante disponer de máquinas con excelente relación coste/rendimiento. Normalmente, el mayor poder de cómputo implica costes elevados que hace de estos sistemas paralelos prohibitivos. Una manera de lograr una buena relación coste/rendimiento es hacer cooperar muchos elementos de cálculo de bajo poder computacional y por consiguiente de bajos costes.
- **Algunos problemas son intrínsecamente paralelos:** ciertos problemas son más fáciles de modelar usando el paradigma paralelo por la estructura que se usa para su resolución.

La computación paralela está jugando un papel fundamental en el avance de todas las ciencias, permitiendo mayor poder de cómputo para explorar modelos más realistas. Quizás la limitante mayor en computación paralela y distribuida es que no son ideas fáciles de entender e implementar. Existen diferentes enfoques para aplicar paralelismo. Incluso para ciertas aplicaciones algunos enfoques pueden ser contraproducentes. Además, existen varias arquitecturas, modelos de programación, lenguajes y compiladores compitiendo por conquistar el mercado. Por otro lado, los precios baratos de los computadores personales (PC) con su incremento en el poder computacional los hacen buenos competidores contra los computadores paralelos.

Hay muchas áreas donde el poder computacional de un computador simple no es suficiente para obtener resultados. Los sistemas paralelos y distribuidos pueden producir resultados más rápidos. A continuación, se mencionan algunas áreas donde puede ser aplicada la computación paralela:

- **Procesamiento de imágenes:** este tipo de aplicaciones abarca las transformaciones de imágenes, análisis de imágenes, reconocimiento de patrones, etc. Existen dos aspectos básicos que las hacen apropiadas para el procesamiento paralelo. El primero es la gran cantidad de datos que manejan y el segundo tiene que ver con la velocidad requerida para procesar imágenes. Muchas veces estas aplicaciones tienen fuertes restricciones a nivel de tiempo.
- **Modelado matemático:** los investigadores construyen y prueban modelos con nuevas teorías para describir fenómenos naturales o para resolver problemas de la vida real. Estos modelos matemáticos altamente complejos requieren, generalmente, resolver ecuaciones diferenciales parciales o elementos finitos cuyas soluciones pueden requerir de gran cantidad de poder de cómputo. El modelado matemático consiste en describir entidades en términos de ecuaciones que comprenden variables y constantes.
- **Computación inteligente:** en esta área existen muchos problemas intrínsecamente paralelos que tratan de imitar el comportamiento inteligente de los humanos. En el área de las redes neuronales artificiales, el procesamiento consiste en la interacción de un grupo de neuronas cuyos elementos van describiendo la dinámica del sistema. A nivel de la computación evolutiva existen muchos aspectos paralelos, por ejemplo, el procesamiento paralelo de cada individuo que conforma la población.
- **Manipulación de bases de datos:** en este caso se puede reducir el tiempo de ejecución al asignar segmentos de bases de datos a elementos de procesamiento paralelo.
- **Predicción del estado del tiempo:** este es un buen ejemplo de una aplicación que requiere de mucho poder computacional. La atmósfera es modelada dividiéndola en regiones o celdas de tres dimensiones usando ecuaciones matemáticas complejas. Las condiciones de cada celda (temperatura, humedad, presión, velocidad y dirección del viento, etc.) son calculadas en intervalos de tiempo usando las condiciones existentes en intervalos previos de tiempo. Un aspecto importante en este tipo de aplicación es el número de celdas a usar. Supongamos que queremos dividir la atmósfera terrestre en celdas de  $1.6\text{Km} \times 1.6\text{ Km} \times 1.6\text{ Km}$ , así cada  $16\text{ Km}$  requieren 10 celdas. Para cubrir toda la atmósfera se necesitan aproximadamente  $5 \times 10^8$  celdas. Supongamos que cada cálculo en una celda requiere 200 operaciones de punto flotante. Si queremos predecir el tiempo de los próximos 10 días con intervalos de 10 minutos, se necesitan aproximadamente  $10^3$  pasos y  $10^{14}$  operaciones de punto flotante. Un computador de 100 Mflops tardaría  $10^6$  segundos. Para ejecutarlo en tiempos cortos se necesita de más de 1Tflop.

### 1.3. Definiciones básicas

Antes de abordar los conceptos relacionados con la computación paralela, es importante tener clara la diferencia entre algoritmo, programa, proceso y tarea.

- **Algoritmo:** define una secuencia de pasos para llevar a cabo una actividad o resolver un problema a través de una serie de instrucciones o reglas bien definidas y ordenadas. A partir de un estado inicial y una entrada, se sigue la secuencia de pasos para alcanzar un estado final y obtener una solución o respuesta.
- **Programa:** es una secuencia de instrucciones que llevan a cabo una tarea específica en un computador. Un programa es una entidad pasiva y permanece alojada en disco. Un **programa ejecutable** tiene un formato que el procesador entiende y puede utilizar directamente para ejecutar las instrucciones. Un programa fuente está escrito en un lenguaje de alto nivel (como C, C++, Java, Phyton) en un formato legible para humanos. A partir de un **programa fuente** se derivan los programas ejecutables, a través de un proceso de compilación o interpretación en tiempo de ejecución.
- **Proceso:** es un programa en ejecución. Es un programa cuya ejecución ha empezado, pero aún no ha terminado. Por lo tanto, es una entidad activa reconocida por el sistema operativo y con recursos asociados (memoria principal, tiempo de CPU), datos asociados (variables, *buffers*), contexto de ejecución (toda la información que el CPU necesita para ejecutarlo), registros, prioridades, eventos por los que espera, etc. Otra manera de definir un proceso se relaciona con las capacidades del sistema operativo de asignar recursos y ejecutar tareas. En este contexto, un proceso es una **unidad de despacho**, dado que se considera como una entidad activa que puede ser ejecutada, y una **unidad de asignación de recursos**, dado que el sistema operativo le asigna los recursos necesarios para su ejecución. Uno de los primeros recursos que el sistema operativo asigna a un proceso es su espacio de direcciones en memoria principal. Esto es, el sistema operativo asigna una parte de la memoria principal, a cada proceso que va a iniciar su ejecución. Un programa se convierte en proceso cuando inicia su ejecución.
- **Tarea:** es una secuencia de instrucciones que tiene asociado un conjunto de datos y que pertenece a un proceso. Un proceso se puede descomponer en varias tareas.

A continuación, se presentan otras definiciones básicas relacionadas con la programación paralela:

- **Computación Paralela:** "... *Es el procesamiento de información que enfatiza la manipulación concurrente de elementos de datos pertenecientes a uno o más procesos resolviendo un problema común*". (Quinn y Quinn, 1994).
- **Computador Paralelo:** "*Es un computador con múltiples procesadores capaz de realizar cómputo paralelo*". (Quinn y Quinn, 1994).
- **Procesamiento Paralelo:** es una forma eficaz de procesamiento que favorece la explotación de los sucesos concurrentes en un sistema de computación. Esta concurrencia implica simultaneidad, solapamiento y multiplicidad de recursos. **El paralelismo que involucra solapamiento se denomina temporal porque permite la ejecución concurrente de sucesos sobre los mismos recursos en intervalos intercalados de tiempo. El paralelismo que permite simultaneidad real sobre múltiples recursos al mismo tiempo se denomina paralelismo espacial.** A medida que se fueron evidenciando las necesidades y requerimientos

de mayor poder de cómputo y rendimiento en los sistemas, se fueron implementando una serie de técnicas y mecanismos de mejora del desempeño que introducen algún nivel de paralelismo que han ido evolucionando y consolidándose, permitiendo los niveles de rendimiento con los que se cuenta en los computadores de hoy en día:

- **Multiprogramación y tiempo compartido:** es una técnica de software introducida por los diseñadores de sistemas operativos para implementar paralelismo temporal en la ejecución de múltiples procesos de usuarios y los mismos procesos del sistema operativo, logrando que todos progresen a la vez. La multiprogramación permite tener múltiples procesos en memoria y solo realiza cambio de contexto con los procesos de monitoreo del sistema o cuando el proceso se bloquea en espera de un servicio. El tiempo compartido es más dinámico en el sentido que asigna un *quantum* de tiempo de CPU a cada proceso.
- **Solapamiento de las operaciones CPU y E/S:** se refiere a mecanismos de mejora del sistema de entrada/salida que permite liberar al CPU de las operaciones de entrada/salida, permitiéndole ejecutar otras operaciones. Este mecanismo requiere la introducción de procesadores auxiliares para la transferencia de entrada/salida a memoria: controlador de acceso directo a memoria (controlador DMA por *Direct Memory Access*), canales de E/S y controladores de E/S inteligentes. La Figura 4 ilustra la ejecución simultánea de instrucciones de cómputo de un proceso P2 en el CPU, mientras el controlador DMA de disco ejecuta una operación de lectura de otro proceso P1.

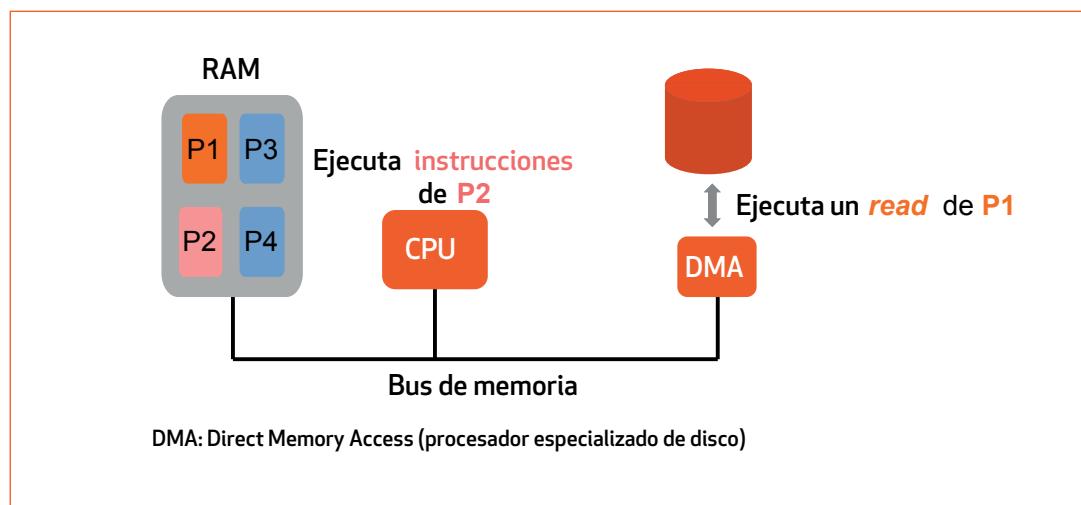


Figura 4. Solapamiento de operaciones de CPU con operaciones de E/S.

- **Jerarquización y equilibrio de ancho de banda:** las diferencias de velocidad entre diferentes componentes del sistema afectan a su desempeño global por los cuellos de botella que los dispositivos más lentos imponen en las transferencias entre ellos. La jerarquización de recursos en diferentes niveles y la introducción de niveles intermedios que reduzcan la brecha de velocidad permiten ir equilibrando el ancho de banda del sistema y acelerar su rendimiento global. En el sistema de memoria se cuenta con la jerarquía de memoria y en los sistemas de entrada y salida con técnicas de compensación con la jerarquización de buses y técnicas de *buffering* en los módulos de entrada/salida.

- **Solapamiento de la ejecución y segmentación encauzada (*pipeline*):** el modelo de ejecución de las máquinas secuenciales puede ser ineficiente en cuanto al nivel de utilización de los componentes internos del CPU, al ocupar todos los recursos en el ciclo de instrucciones completo. Un análisis de las fases y etapas del ciclo permiten identificar diferentes tareas que podrían solaparse. Por ejemplo, la fase *fetch* solo se encarga de traer desde memoria principal al CPU la próxima instrucción a ejecutarse. Finalizada la fase *fetch*, el registro *Program Counter (PC)* no vuelve a utilizarse hasta la próxima fase *fetch*. Entonces, podría considerarse que cuando se complete la tránsito de la **instrucción i**, una parte dedicada del CPU puede ir a traer la siguiente **instrucción i+1**, mientras el resto ejecuta la **instrucción i**. Cada parte o sección del CPU se denomina en este caso etapa o segmento del cauce de ejecución o *pipeline*, porque la salida de cada etapa anterior alimenta la entrada de la siguiente como si estuvieran conectados en una tubería. Por eso la implementación de tal solapamiento de la ejecución en el CPU se denomina segmentación encauzada. El ejemplo de segmentación mostrado en este párrafo solo consta de dos etapas y se le conoce como precarga de instrucciones (*prefetching*). Sin embargo, los sistemas segmentados hoy día suelen tener muchas más etapas de cauce. La técnica de solapamiento de la ejecución permite reducir considerablemente el tiempo de ejecución. Adicionalmente, los sistemas de acceso a memoria también han implementado un modelo de solapamiento en los ciclos de máquina de acceso a memoria, permitiendo reducir grandemente el tiempo de respuesta global del sistema. La Figura 5 muestra un ejemplo de una segmentación encauzada (*pipeline*) de cinco etapas. El CPU es capaz de procesar simultáneamente una instrucción diferente en cada etapa. Con estos mecanismos se logra paralelismo a nivel de instrucciones de un mismo proceso.

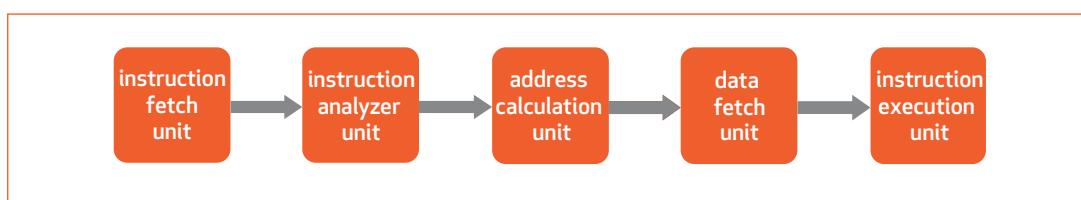


Figura 5. Pipeline de cinco etapas.

- **Sistemas escalares o multiplicidad de unidades funcionales:** la introducción de la segmentación de bajo nivel o *prefetching* de instrucciones, condujo a otra mejora en el rendimiento que consiste en agregar más de una unidad aritmética al procesador a fin de poder acelerar aún más la ejecución. Con esta importante mejora de diseño, podrían tenerse más de una instrucción en ejecución a la vez. Este modelo, sin embargo, introduce requerimientos nuevos también en cuanto a mantener el orden estricto de ejecución de las instrucciones y el control de disponibilidad de los recursos. A los sistemas que implementan esta mejora se les conoce como **sistemas escalares**. La Figura 6 muestra la gráfica de un CPU con cinco unidades aritméticas que pueden resolver operaciones de manera simultánea. La Figura 7 muestra una ilustración de un computador vectorial: las unidades aritméticas pueden realizar, en un solo ciclo, operaciones sobre vectores y producen como resultado otro vector. Con estos mecanismos se logra paralelismo a nivel de instrucciones de un mismo proceso.

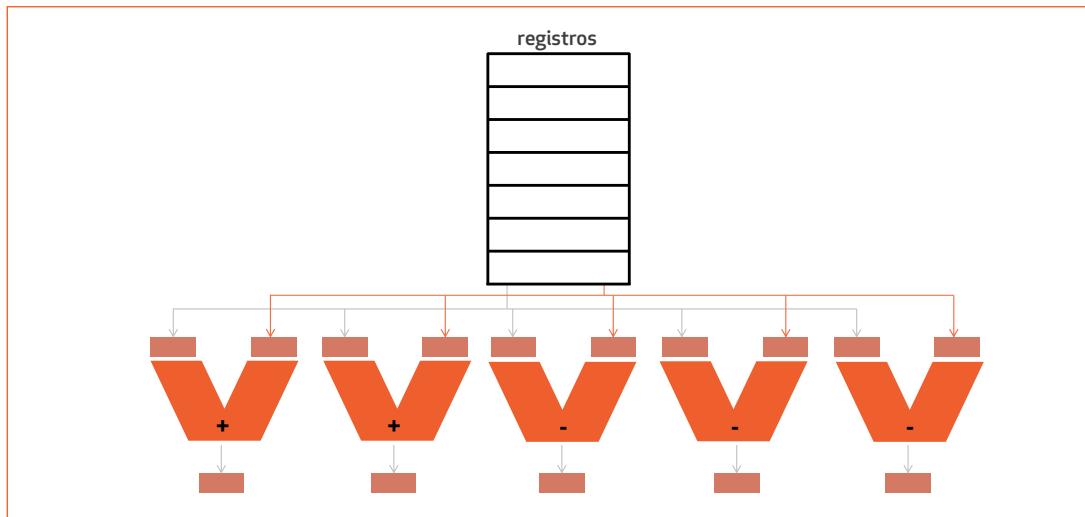


Figura 6. CPU con cinco unidades aritmeticológicas.

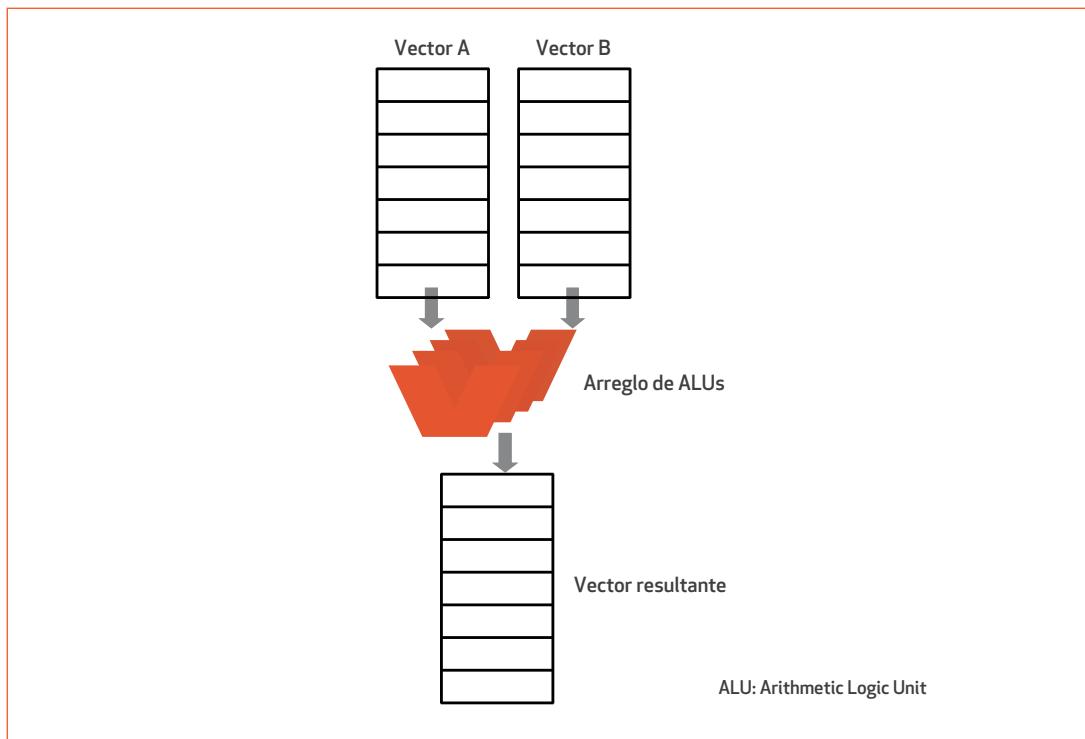


Figura 7. Procesador vectorial.

- **Sistemas paralelos:** las técnicas mencionadas arriba, explotan mayormente el paralelismo temporal. Algunas introducen elementos de paralelismo espacial al incorporar elementos auxiliares de proceso o varias unidades funcionales. Aunque los sistemas escalares introducen un nivel interesante de paralelismo espacial, suele denominarse “sistemas paralelos” solamente a aquellos que explotan el paralelismo efectivamente mediante la multiplicidad de procesadores o elementos de procesamiento simples o complejos que colaboran en la solución de uno o más problemas de forma simultánea.

- **Paralelismo funcional:** en el paralelismo de tipo funcional se encuentran las tareas que pueden ser ejecutadas al mismo tiempo. Un tipo especial de paralelismo funcional es la segmentación encauzada (*pipeline*). Este tipo de paralelismo se puede implementar a nivel de:
  - **Instrucción:** diferentes instrucciones consecutivas se ejecutan en distintas unidades funcionales.
  - **Lazos:** Cada iteración del lazo se ejecuta en una unidad funcional.
  - **Procedimiento:** diferentes procedimientos de un mismo programa se ejecutan en unidades funcionales distintas.
  - **Programa:** diferentes programas de un usuario o de usuarios distintos se ejecutan en unidades funcionales diferentes.
- **Paralelismo de datos:** en este tipo de paralelismo se ejecuta la misma instrucción o función sobre un gran conjunto de datos. Se tienen réplicas del mismo programa trabajando sobre distintas partes de los datos, es decir, se reparten los datos entre las unidades funcionales. El control de este puede ser centralizado o distribuido.

Hasta aquí se han abordado los conceptos principales relacionados a la computación paralela, así como su importancia y uso en escenarios de aplicaciones con requerimientos de gran poder de cómputo sobre gran cantidad de datos.

## 2. Introducción a las arquitecturas paralelas

**Las arquitecturas paralelas se caracterizan por contener múltiples procesadores interconectados.** Existen muchos tipos de arquitecturas paralelas y se distinguen por el tipo de interconexión entre los procesadores y entre los procesadores y la memoria. Esta variedad de hardware no permite una clasificación universal de los sistemas paralelos. Así que se pueden clasificar de acuerdo a distintos aspectos. Si se toma en cuenta la arquitectura de la memoria, los computadores paralelos se pueden clasificar en:

- **Sistemas paralelos con memoria compartida:** estos sistemas poseen múltiples procesadores que acceden a toda la memoria disponible como un espacio global de direcciones. Éstos a su vez se pueden dividir en dos grandes grupos basados en el tiempo de acceso a dicha memoria:
  - **Acceso a Memoria Uniforme (UMA** por sus siglas en inglés de *Uniform Memory Access*). Los sistemas UMA proveen un tiempo de acceso a memoria constante; es decir que el tiempo de acceso a memoria es igual desde cualquier CPU, sin importar la dirección de memoria que se acceda;
  - **Acceso a Memoria No Uniforme (NUMA** por sus siglas en inglés de *Non-Uniform Memory Access*). En estos sistemas NUMA, la memoria está organizada de manera jerárquica, de manera que el tiempo de acceso a la memoria es variable, depende de la dirección a la cual se acceda en un momento determinado.

- **Sistemas paralelos con memoria distribuida:** estos sistemas poseen múltiples procesadores, pero cada uno de ellos solo puede acceder a su memoria local y no existe un espacio de direcciones globales entre los procesadores.

Otra forma de clasificar los sistemas paralelos es de acuerdo con el número de procesadores que lo integran. Aquellos sistemas con cientos de procesadores se denominan **Supercomputadores**; a los que poseen miles de procesadores se les conoce como **Masivamente Paralelos**. Además, se puede hacer referencia a la capacidad de los procesadores que lo integran, como de *Larga Escala* o de *Baja Escala*. Esto depende del tamaño del procesador. Un sistema paralelo integrado por PCs se considera generalmente como un sistema de *Baja Escala*.

Se han propuesto diferentes criterios para clasificar las arquitecturas paralelas, según diferentes características y niveles de paralelismo. La más popularizada y usada en la literatura es la taxonomía de Flynn (1996), que clasifica los computadores secuenciales y paralelos según cómo se despacha el flujo de instrucciones a cada procesador. Así, se tiene:

- **Sistemas SISD (Single Instruction Single Data)** o **Sistemas SIMD (Single Instruction Multiple Data)**, si los procesadores ejecutan la misma instrucción en el mismo instante de tiempo.
- **Sistemas MISD (Multiple Instruction Single Data)** o **Sistemas MIMD (Multiple Instruction Multiple Data)**, si cada procesador ejecuta instrucciones diferentes.

## 2.1. Taxonomía de Flynn

La taxonomía de Flynn (1996) distingue la arquitectura computacional de los multiprocesadores según cómo éstos pueden ser clasificados a través de dimensiones independientes de **Instrucciones** y **Datos**. Cada una de estas dimensiones puede tener solo uno de dos posibles estados: *Single* o *Multiple*. Así, tenemos cuatro posibles clasificaciones de acuerdo con Flynn (1996):

- **Single Instruction, Single Data (SISD):**
  - En esta categoría se encuentran los computadores seriales (no-paralelos).
  - *Single Instruction:* implica que solo un flujo de instrucciones está actuando en el CPU durante cualquier ciclo de reloj.
  - *Single Data:* implica que solo un flujo de datos está siendo usado como entrada durante cualquier ciclo de reloj.
  - La ejecución del flujo de instrucciones es determinística.
  - Ejemplos de sistemas SISD: las PCs y estaciones de trabajo que poseen un único CPU.
- **Single Instruction, Multiple Data (SIMD):**
  - Esta categoría considera un tipo particular de computador paralelo.

- *Single Instruction*: implica que todas las unidades de procesamiento ejecutan el mismo flujo de instrucciones en cualquier ciclo de reloj.
- *Multiple Data*: significa que cada unidad de procesamiento puede operar en datos diferentes.
- Conveniente para problemas caracterizados por un alto grado de regularidad, como, por ejemplo, procesamiento de imágenes.
- Existen dos variedades de sistemas SIMD: arreglo de procesadores y procesadores vectoriales.
- Ejemplos de arreglo de procesadores: Connection Machine CM-2, Maspar MP-1, MP-2.
- Ejemplos de computadores vectoriales: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820.
- Los procesadores vectoriales surgen frente a la necesidad de procesamiento de vectores y matrices. A finales de los 50 e inicios de los 60, se incrementó la demanda de procesamiento automatizado para cálculos científicos. Estos requieren procesar vectores y matrices, a menudo de gran tamaño. Un gran inconveniente surgió con los modelos secuenciales de la época, pues las arquitecturas y lenguajes de programación secuenciales tienen un desempeño muy pobre en este tipo de operaciones que regularmente se simulan mediante ciclos, obteniendo tiempos de respuesta prohibitivos para la mayoría de las aplicaciones requeridas.

Un ejemplo claro es la multiplicación de matrices que involucra tres ciclos anidados, con lo cual cada incremento de las dimensiones de las matrices multiplica los ciclos por dos factores, lo que resulta en un algoritmo exponencial del orden  $O(n^3)$ . La necesidad impulsó a los diseñadores de computadores a tratar de interpretar de mejor manera el comportamiento de estas estructuras de datos. Para ello se consideró útil hacer un análisis de las características del procesamiento de vectores/matrices y sus exigencias. El resultado fueron los sistemas vectoriales, un conjunto de máquinas de alto coste, desempeño y una enorme capacidad de cálculo (que varía de máquina a máquina) que se puede asignar a la clasificación SIMD de la taxonomía de Flynn (aunque algunos autores señalan que solo las arquitecturas de arreglo de procesadores son SIMD). Dado que estas arquitecturas se diseñaron para optimizar las operaciones sobre vectores y matrices, no todas las aplicaciones se benefician de su poder computacional. Las características de las aplicaciones ideales para usar este tipo de sistemas vectoriales son:

- Trabajan con una colección o arreglo de datos.
- Los datos se almacenan en celdas contiguas en la memoria (vectores o matrices).
- Presentan independencia de datos (operación en elemento  $[i, j]$  no depende de resultado en  $[i-1, j-1]$ ).

- Se ejecuta una misma operación sobre todo el conjunto de datos.
- **Multiple Instruction, Single Data (MISD):**
  - Existen pocas clases de este tipo de computadores.
  - *Multiple Instruction*: implica que cada unidad de procesamiento ejecuta un flujo de instrucciones diferente en cualquier ciclo de reloj.
  - *Single Data*: significa que todas las unidades de procesamiento operan sobre los mismos datos.
  - Algunos ejemplos de aplicaciones que se beneficiarían de este tipo de sistemas son:
    - Filtros de múltiple frecuencia operando en una única señal.
    - Múltiples algoritmos de criptografía actuando en un mensaje codificado.
- **Multiple Instruction, Multiple Data (MIMD):**
  - Actualmente, es el tipo más común de computador paralelo.
  - *Multiple Instruction*: implica que cada procesador puede ejecutar un flujo de instrucciones diferente.
  - *Multiple Data*: implica que cada procesador puede estar trabajando con un flujo de datos diferente.
  - Ejemplos de sistemas MIMD: supercomputadores actuales, redes de computadores, computación en malla (*grid computing*), multi-procesadores SMP (*symmetric multiprocessing*) - incluyendo algunos tipos de PCs.

Dado que la mayoría de arquitecturas computacionales existentes actualmente pertenecen a la clasificación MIMD, en la literatura se ha propuesto una subclasiación de los sistemas MIMD (Tanenbaum, 2017).

## 2.2. Clasificación de los sistemas MIMD

Los sistemas MIMD predominan actualmente como las plataformas de hardware para los sistemas paralelos, sistemas de redes y sistemas distribuidos. De acuerdo con el soporte de la memoria se clasifican en *Sistemas Fuertemente Acoplados* o de Memoria Compartida (también llamados **multiprocesadores**) y *Sistemas Débilmente Acoplados* de Memoria Distribuida (también llamados **multicomputadores**). Tanto los multiprocesadores como los multicomputadores, pueden interconectarse a través de dos esquemas diferentes de interconexión: **buses** o **conmutadores**.

Independientemente del sistema de interconexión, los *Sistemas Fuertemente Acoplados* (de Memoria Compartida o multiprocesadores) son más fáciles de construir que los *Sistemas Débilmente Acoplados* (de Memoria Distribuida o multicomputadores), pero presentan limitaciones respecto a la cantidad

de procesadores concurrentes, dado que por lo general la interconexión se caracteriza por presentar bajo ancho de banda y alta latencia.

- **Sistemas Fuertemente Acoplados (de Memoria Compartida o multiprocesadores)**

Los sistemas MIMD de memoria compartida tienen en común las siguientes características generales:

- Todos los procesadores pueden acceder a la memoria como un espacio de direccionamiento global.
- Múltiples procesadores operan independientemente, pero comparten los mismos recursos de memoria.
- Los cambios en una localización de memoria realizados por un procesador son visibles a todos los otros procesadores.
- La sincronización es obtenida controlando la escritura y lectura a la memoria.

Los sistemas MIMD de memoria compartida, a su vez se pueden dividir en dos clases principales, basadas en los tiempos de acceso a memoria:

- **De Acceso Uniforme a Memoria (UMA por *Uniform Memory Access*)**, cuyas características principales son:
  - Son sistemas representados por multiprocesadores simétricos (SMP, por *Symmetric Multiprocessor machines*).
  - Todos los procesadores son idénticos.
  - Los tiempos de acceso a la memoria son constantes desde cualquiera de los procesadores que integran el sistema.
  - Aseguran **coherencia de la memoria caché**, por lo que algunas veces son llamados CC-UMA (por *Cache Coherent UMA*). Coherencia de caché significa que, si un procesador actualiza una posición en la memoria compartida, todos los otros procesadores conocen acerca de la actualización. La coherencia de caché se resuelve a nivel de *hardware*.
- **De Acceso No Uniforme a Memoria (NUMA por *Non Uniform Memory Access*)**, caracterizados principalmente por:
  - El conjunto de procesadores que conforman estos sistemas, por lo general enlazan físicamente dos o más SMPs.
  - Un SMP puede acceder directamente a la memoria de otro SMP.
  - Los tiempos de acceso a la memoria no son constantes. No todos los procesadores tienen igual tiempo de acceso a toda la memoria.

- El acceso a la memoria es lento.
- Si la coherencia de caché es mantenida a nivel de hardware, son llamados CC-NUMA (por *Cache Coherent NUMA*).

Los sistemas MIMD de memoria compartida presentan las siguientes ventajas:

- El espacio de direccionamiento global provee una facilidad de programación al usuario.
- El intercambio de datos entre las tareas es rápido y uniforme debido a la proximidad de la memoria a los CPUs.

Sin embargo, poseen limitaciones que se convierten en desventajas, tales como:

- Pérdida de escalabilidad entre la memoria y CPUs; agregar CPUs puede aumentar el tráfico para el acceso a la memoria compartida desde los CPUs.
- El programador es el responsable de construir la sincronización que asegura un correcto acceso a la memoria global.
- Es costoso diseñar y producir máquinas de memoria compartida con un gran número de procesadores.

- ***Sistemas Débilmente Acoplados (de Memoria Distribuida o multicomputadores)***

Por su parte, los sistemas MIMD de memoria distribuida presentan las siguientes características comunes:

- Requieren una red de comunicación para conectar la memoria a los procesadores.
- Múltiples procesadores operan independientemente pero cada uno tiene su propia memoria.
- Los datos son compartidos a través de una red de comunicación usando pase de mensajes.
- El usuario es el responsable de la sincronización usando pase de mensajes.

Las principales ventajas que presentan estos sistemas son:

- La memoria es escalable al número de procesadores. Un aumento de los procesadores implica aumento del tamaño de la memoria y del ancho de banda.
- Cada procesador puede rápidamente acceder a su propia memoria sin interferencia.

Las principales desventajas de los sistemas de memoria distribuida son:

- Dificultad para asignar estructuras de datos existentes a esta organización de memoria.
- El usuario es el responsable de enviar y recibir datos a través de los procesadores.

Los computadores de memoria compartida y de memoria distribuida pueden ser construidos conectando procesadores y memoria usando una variedad de redes de interconexión. Estas redes pueden ser clasificadas como estáticas y dinámicas. Las **redes estáticas** consisten en enlaces de comunicación punto a punto entre los procesadores, por ejemplo, buses. También se les conoce como redes **directas**. Las **redes dinámicas** son construidas con conmutadores (*switches*) y enlaces de comunicación; también se les conoce como **indirectas**. Son usadas generalmente para computadores que trabajan con envío de mensajes. Así, según el soporte de la memoria principal y la red de interconexión los sistemas MIMD se clasifican en (ver Figura 8):

- Sistemas MIMD con memoria compartida, conectados por buses.
- Sistemas MIMD con memoria compartida, conectados por conmutadores.
- Sistemas MIMD con memoria distribuida, conectados por buses.
- Sistemas MIMD con memoria distribuida, conectados por conmutadores.

A continuación, se describe cada clasificación.

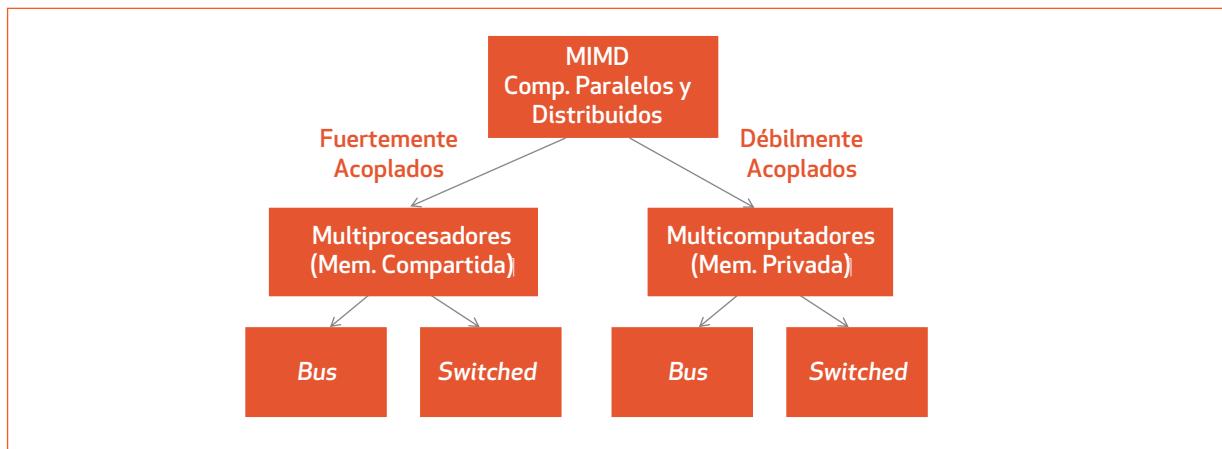


Figura 8. Clasificación de los sistemas MIMD.

## 2.2.1. Sistemas MIMD con memoria compartida y conectados con buses

Estos sistemas son fuertemente acoplados por la existencia de una única memoria común. Para que los procesadores puedan compartir la memoria, se requiere una estructura de interconexión fuertemente acoplada. La manera más sencilla de interconectar varios CPUs con una memoria común es mediante un **bus común**, que implica una conexión estática.

El problema que se presenta en este tipo de interconexión es que la memoria se convierte en cuello de botella, requiriendo gran ancho de banda en el acceso a la memoria y limitando la cantidad máxima de procesadores concurrentes contendientes por el bus. Para solucionar este problema se proponen varias soluciones:

- Agregar memoria local o memoria caché a cada CPU; el problema con esta solución es mantener la coherencia de las memorias cachés con la memoria principal.

- Agregar memoria local o memoria caché a cada CPU con políticas de escritura: escritura a través de la memoria caché (*write through cache*) y monitoreo del caché (*snoopy cache*).

La Figura 9 ilustra una arquitectura de multiprocesadores conectados a través de un bus común a la única memoria principal compartida.

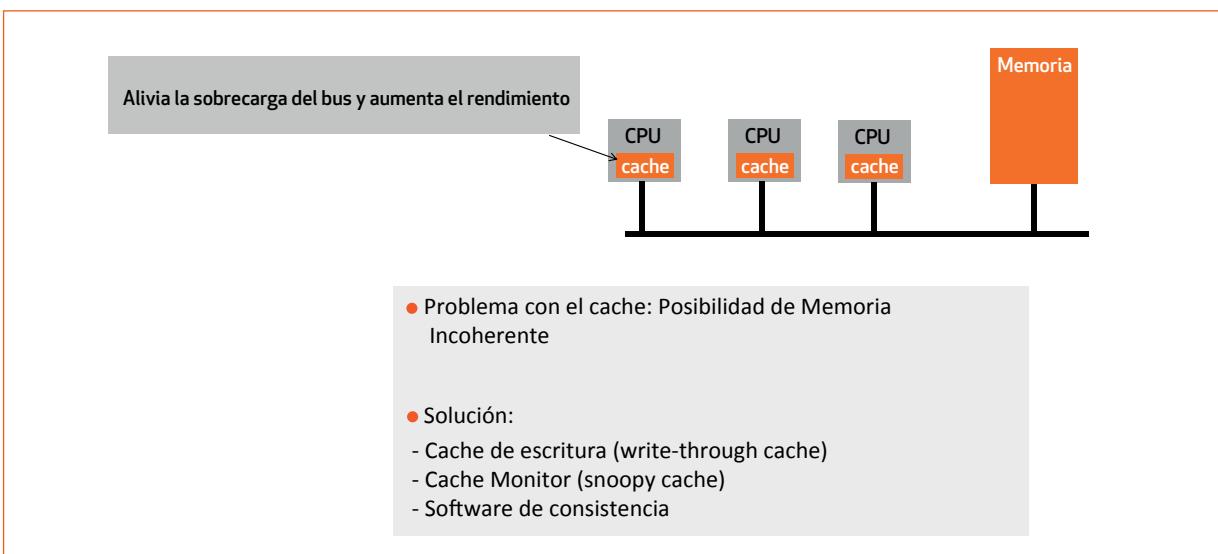


Figura 9. Multiprocesadores conectados con un bus común

Con el uso de memorias cachés se logra un poco más de escalabilidad, pero sigue siendo limitado, en especial para aplicaciones paralelas o concurrentes que comparten muchos datos y los modifican frecuentemente, además de los problemas de coherencia de cachés que se deben resolver.

- **Coherencia de cachés**

En sistemas de bus común, el tráfico y el cuello de botella que implica que el bus sea de uso exclusivo (solo un procesador puede acceder a la memoria en un momento dado), se puede aliviar significativamente con el uso de cachés. La concurrencia impone altos requisitos a estas cachés. El principal de ellos es resolver el problema de la coherencia de caché. Imaginemos el siguiente escenario: varios procesadores ejecutan hilos de un **código reentrante** que comparte algunas variables globales. Estas variables son accedidas y actualizadas por todos los procesadores. Cada vez que un procesador actualice, los demás deben tener noción de ello, pues las copias quedan incoherentes entre sí.

En sistemas de bus común, la coherencia de cachés suele resolverse a nivel de *hardware* con políticas de escritura: escritura a través de la memoria caché (*write through cache*) y monitoreo del caché (*snoopy cache*).

La política de escritura a través de la memoria caché consiste en que cada vez que un dato alojado en la memoria caché de un procesador se modifica, automáticamente dicho valor se actualiza en la memoria principal, a través del bus. Por su parte, la política de monitoreo del caché consiste en que los procesadores permanecen monitoreando el bus y si detectan una actualización en memoria de un dato que está en su caché, este dato será invalidado; así

cuando ese procesador quiera acceder a dicho dato, lo encontrará invalidado, por lo tanto, tendrá que cargarlo desde la memoria principal, donde está la actualización más reciente.

## 2.2.2. Sistemas MIMD con memoria compartida y conectados con conmutadores

El uso de una memoria compartida de gran tamaño, organizada en bancos o módulos, también puede aprovecharse para “paralelizar” el acceso a la memoria mediante entrelazamiento, permitiendo un acceso “encauzado” a la memoria. También pueden crearse puertos a cada módulo con lo cual podría accederse en paralelo a diferentes direcciones de la misma memoria principal o mejor aún, múltiples accesos al mismo módulo.

Otra manera de interconectar los CPUs con la memoria, si ésta es modular, consiste en emplear una estructura de interconexión en forma de una malla de conmutación de circuitos. La Figura 10 muestra un ejemplo de esta red de interconexión: multiprocesadores conectados a diferentes módulos de la memoria principal, con conmutadores en topología de malla. Existen buses de conexión de cada procesador a cada módulo de memoria. En las intersecciones de las barras o buses, existen conmutadores capaces de unir una barra horizontal con una vertical creando un bus particular entre el procesador y el módulo. En cualquier instante cualquier procesador puede adquirir acceso a un módulo que no esté ocupado por otro procesador. En el caso de que dos procesadores deseen acceder al mismo módulo al mismo tiempo, éstos se bloquearán mutuamente. En ese caso uno tendrá que reintentar luego. Para lograr este objetivo, se implementan políticas de arbitraje que definen qué procesador tiene prioridad en caso de bloqueo. El problema con este tipo de conexiones es el alto número de conmutadores que puede representar, en general si hay  $N$  procesadores, se requieren  $N^2$  conmutadores (por ejemplo, para 32 procesadores  $32^2=1024$  conmutadores). Esto hace que sean arquitecturas muy costosas y poco escalables.

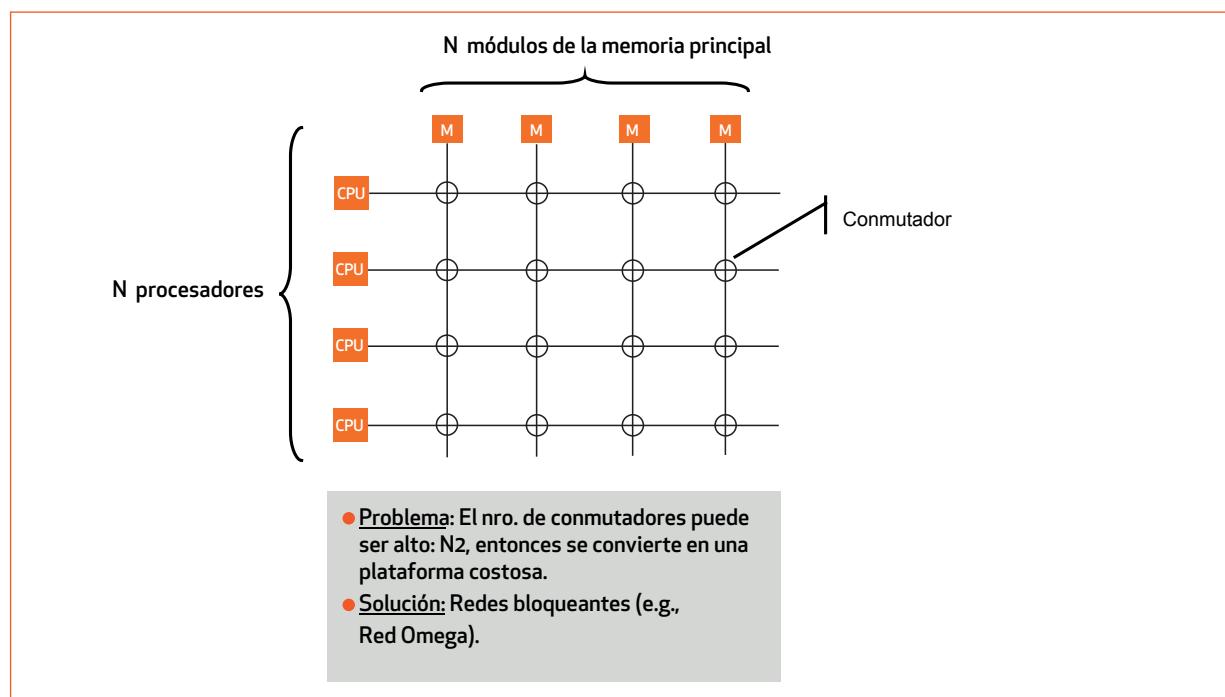


Figura 10. Multiprocesadores conectados con conmutadores en topología de malla.

Para reducir el número de commutadores, se proponen redes de interconexión en multi-etapas. Por ejemplo, la Figura 11 muestra una Red Omega, un tipo de red multi-etapa. Con  $N$  procesadores, en cada etapa se requieren  $N/2$  commutadores y se establecen  $\log_2 N$  etapas. Por lo tanto, el número total de commutadores se reduce a  $N/2 * \log_2 N$  (por ejemplo, para 32 procesadores se requieren  $32/2 * \log_2 32 = 80$  commutadores). Otras redes más complejas permiten caminos alternativos a módulos que no han sido ocupados. En cada red multi-etapa, existen parámetros que se pueden calcular para poder definir de antemano: cantidad de commutadores, etapas y enlaces necesarios, de acuerdo con la cantidad de CPUs y módulos de memoria.

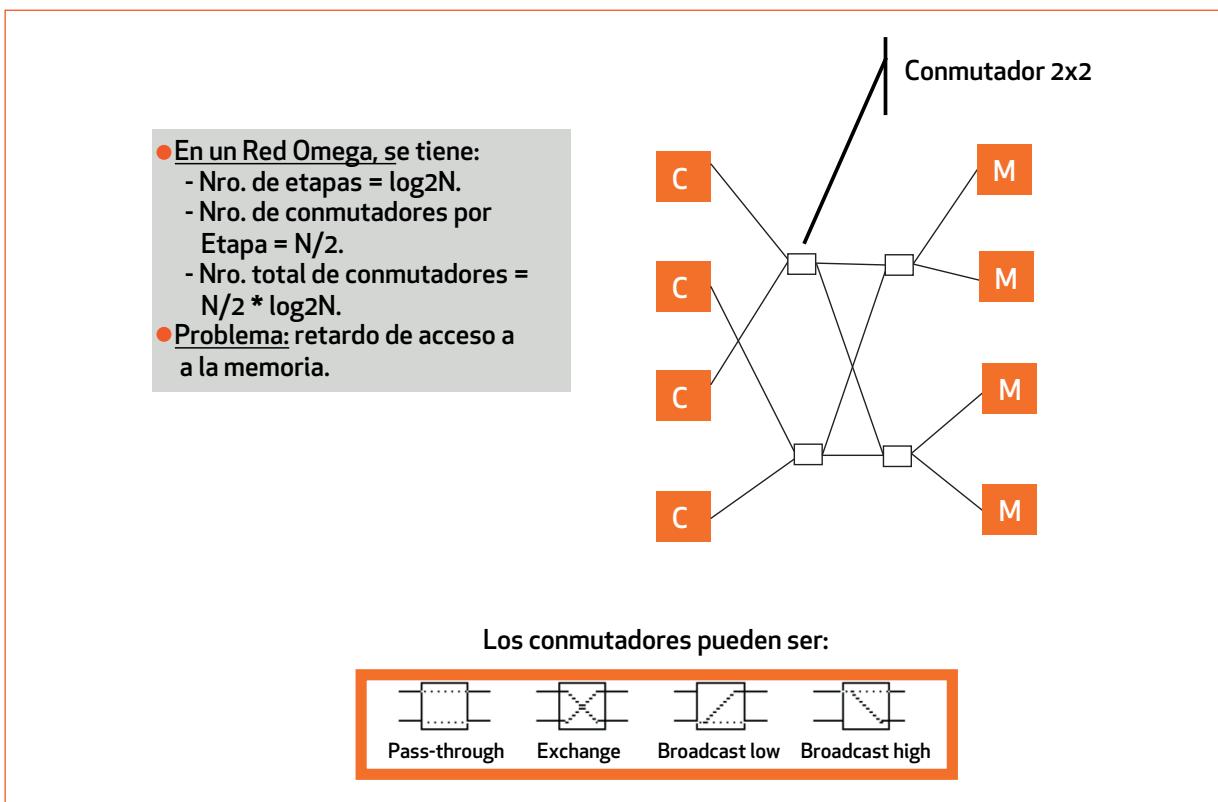


Figura 11. Red Omega con commutadores en multi-etapas.

Las estructuras de interconexión anteriores son redes estáticas y son relativamente simples. Las redes de interconexión deben garantizar el acceso único a un módulo en cualquier momento (arbitraje) para evitar condiciones de carrera y por tanto se diseñan como redes de múltiples etapas. Los nodos de estas redes son dispositivos de commutación denominados commutadores de intercambio. En la parte inferior de la Figura 11 se muestran distintos tipos de commutadores de intercambio. Estos permiten en cualquier momento enrutar cualquiera de sus entradas a cualquiera de sus salidas que no se encuentre ocupada. Otra vez, se debe definir la prioridad de cada entrada. Se emplea un bus de control de 1 bit asociado a cada entrada, que indica la salida a la que se quiere commutar.

### 2.2.3. Sistemas MIMD con memoria distribuida y conectados con buses

En la literatura, no se considera esta clasificación como válida. En los sistemas MIMD, con un bus común, los procesadores comparten una memoria principal que acceden a través del bus. Esto corresponde a la categoría explicada en la Sección 2.2.1.

## 2.2.4. Sistemas MIMD con memoria distribuida y conectados con conmutadores

En los sistemas MIMD de memoria distribuida, cada procesador tiene su propia memoria local, individual. Ningún procesador tiene acceso ni conocimiento sobre las memorias de los otros procesadores. Así que, para compartir los datos, se deben usar mecanismos de pase de mensaje entre los procesadores. Dado que no existe una memoria compartida, los problemas de contención y retardo en el acceso no están presentes en estos sistemas. Es decir, ahora se tiene que resolver la comunicación entre procesadores y no la comunicación entre procesadores y memoria.

Para la comunicación entre procesadores no resulta económicamente factible conectar un gran número de procesadores directamente con cada uno de los otros procesadores. Una forma de evitar esta multitud de conexiones directas es conectar cada procesador solo a unos pocos. Sin embargo, estas topologías pueden resultar ineficientes, dado el tiempo requerido de pasar un mensaje de un procesador a otro, hasta llegar al procesador destino. La cantidad de tiempo requerido por los procesadores para ejecutar el enrutamiento de un simple mensaje puede ser sustancial. A continuación, se presentan varios sistemas MIMD de memoria distribuida con distintos esquemas de interconexión, que representan redes tanto estáticas como dinámicas y que intentan reducir los tiempos de enrutamiento.

- **Redes LAN y WAN**

Se trata de estaciones de trabajo y otros recursos (como impresoras, faxes) conectados a una topología de red dinámica compartida utilizando un protocolo determinado. El término LAN (*Local Area Network*) alude a una red -a veces llamada subred- instalada en una misma sala, oficina o edificio. Con la autorización adecuada, se puede acceder a los recursos disponibles en la LAN, desde cualquier otro nodo también conectado a la LAN. Por su parte, una red de área ancha o WAN (*Wide Area Network*) es una colección de LAN interconectadas. Las WAN pueden extenderse a ciudades, estados, países o continentes. Las redes que comprenden una WAN utilizan encaminadores (*routers*) para dirigir sus paquetes al destino apropiado. Los encaminadores son dispositivos hardware que enlazan diferentes redes dinámicas para proporcionar el camino más eficiente para la transmisión de datos. La Figura 12 presenta un ejemplo de red LAN y un ejemplo de red WAN.

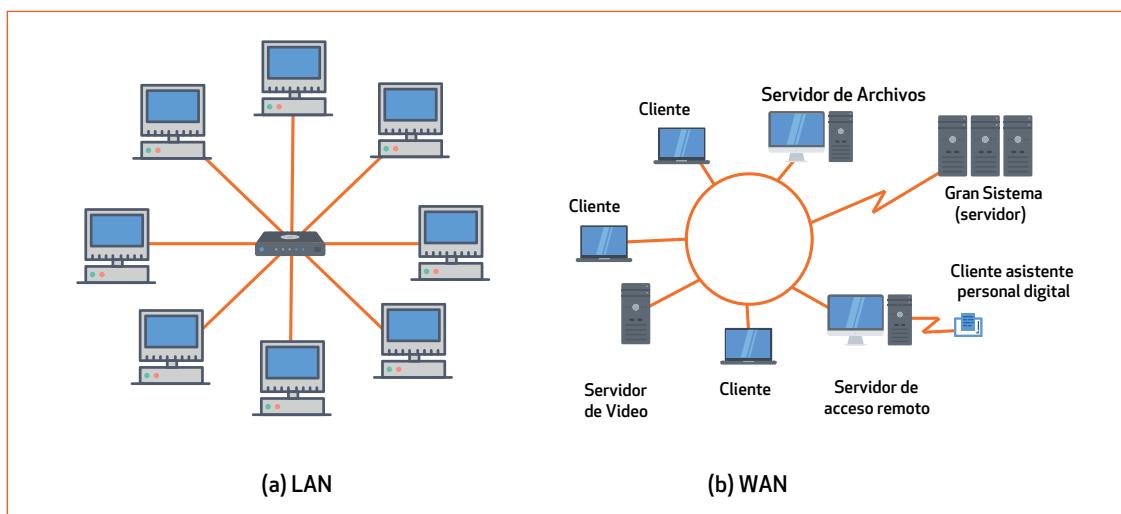


Figura 12. (a) Red LAN. (b) Red WAN. Adaptado de [http://dominiopublico.com/intranets/lan\\_wan.php](http://dominiopublico.com/intranets/lan_wan.php)

- **Redes de interconexión en malla**

En un sistema MIMD de memoria distribuida con una red de interconexión en malla, los procesadores se organizan en una matriz de dos dimensiones. Cada procesador se conecta a sus cuatro vecinos inmediatos (ver Figura 13(a)). En algunos casos, el último elemento de una fila se conecta con el primero de la siguiente fila, formando una red en malla toroidal (ver Figura 13(b)). En ambos casos las redes son estáticas. La Figura 13 muestra en ejemplo de malla plana y malla toroidal con 16 procesadores (cada procesador tiene su memoria local).

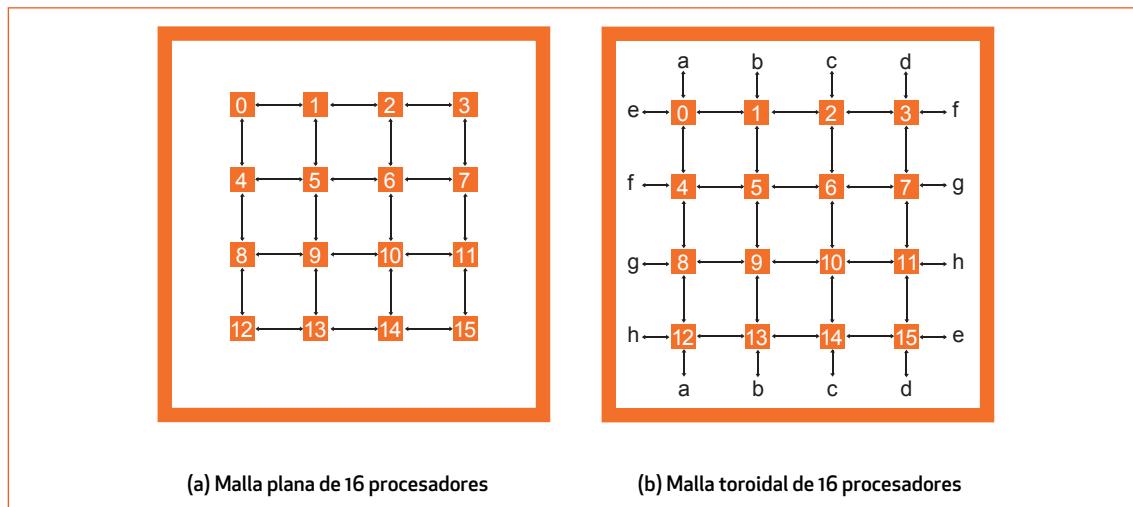


Figura 13. (a) Conexión en malla plana. (b) Conexión en malla toroidal.

- **Redes de interconexión en hipercubo**

Este tipo de red es normalmente usado en sistemas MIMD de memoria distribuida y en algunos sistemas SIMD. Es una red de comutación de paquetes. En este caso, así como en el caso de malla, los nodos de la red no son comutadores sino CPUs con su memoria local e inclusive sistemas de E/S propios. Las redes hipercubo consisten en  $N$  procesadores, donde  $N$  debe ser expresado en potencia de dos. Es decir,  $N=2^m$  procesadores. Esos procesadores forman los vértices de cuadrados interconectados, formando una red multidimensional con dos nodos en cada dimensión.

El hipercubo es una red eficiente y escalable, aunque presenta algunos retos al diseñador, como la asignación de direcciones para los CPUs. Las direcciones se asignan en forma de cadenas de bits. Si  $N=2^m$ ,  $m$  representa la cantidad de bits requeridos para asignar las direcciones a cada procesador en la red. Un principio establece que dos nodos contiguos solo diferencian su dirección uno del otro en 1 bit. Esto permite establecer algoritmos de asignación de direcciones sencillos, así como algoritmos de enrutamiento de paquetes o mensajes. La Figura 14 muestra la construcción escalable de hipercubos de 1 a 4 dimensiones. Un hipercubo 3D, se representa con un cubo de 8 nodos y 12 arcos. Un hipercubo 4D, se crea duplicando dos hipercubos 3D y adicionando un bit más significativo. El nuevo bit deberá ser "0" para uno de los hipercubos 3D y "1" para el otro. Luego las esquinas de ambos hipercubos son conectadas para crear el cubo de una dimensión más. Este método se usa para crear hipercubos de cualquier dimensión.

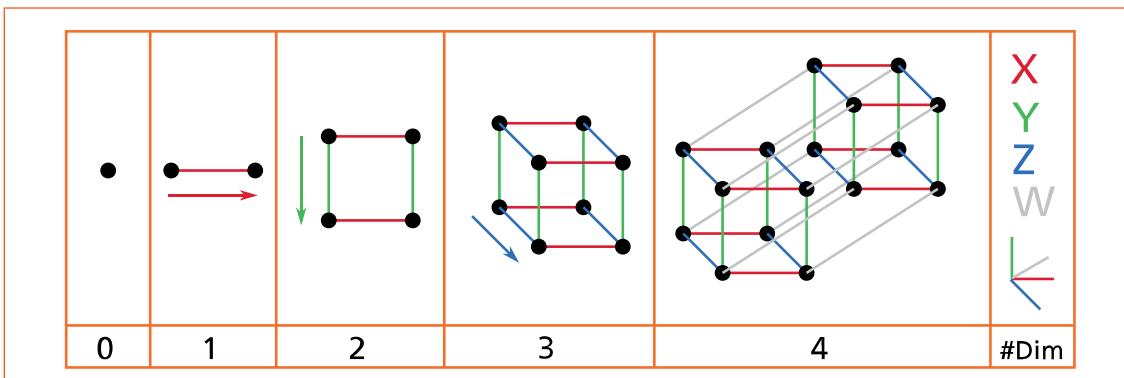


Figura 14. Hipercubos de 1 a 4 dimensiones. Recuperado de <https://en.wikipedia.org/wiki/Hypercube>

**El diámetro de un sistema se refiere al mínimo número de pasos que toma enviar un mensaje de un procesador a otro.** Así, por ejemplo, el diámetro de un hipercubo 2D es 2 ( $2^2$  procesadores en total), de un hipercubo 3D es 3 ( $2^3$  procesadores en total), y así sucesivamente.

Una ventaja de las redes de interconexión en malla sobre los hipercubos es que la malla no requiere configurarse en potencia de dos. Así en un hipercubo es posible que haya más procesadores de los realmente requieren las aplicaciones. Una desventaja es que el diámetro de la malla es mayor que la de un hipercubo para sistemas de más de 4 procesadores.

Algunos ejemplos de arquitecturas MIMD con redes hipercubos son:

- nCUBE Hypercube
- Intel Hypercube
- TMC CM-5
- IBM SP1, SP2
- Intel Paragon
- **Clusters y MMPs (Massively Parallel Processors)**

Otro tipo de plataformas MIMD de memoria distribuida que abundan actualmente son los *cluster* y los procesadores masivamente paralelos (**MPP** por *Massively Parallel Processors*). Los MPPs son computadores paralelos con cientos y miles de procesadores. Los *cluster* están conformados por computadores genéricos unidos por una red de alta velocidad y bajo retardo, vistas como un solo computador. Los *clusters* y MPPs dominan la lista TOP500 que mantiene los 500 computadores más rápidos del mundo. La lista se actualiza dos veces por año.

 **Top500**  
<https://www.top500.org/>

En particular los *clusters* se han popularizado debido al gran beneficio económico que representan. Los *clusters* se construyen con procesadores genéricos de fácil adquisición, que, conectados a través de redes de interconexión genéricas y dinámicas, ofrecen una gran

capacidad agregada de cómputo y almacenamiento, mejoran el desempeño, escalabilidad, disponibilidad, re-uso, además de que existen muchas herramientas libres y de código abierto disponibles para el uso, construcción, monitoreo y administración de los *clusters*. El uso de los *clusters* se ha venido popularizando desde sus inicios en los años 90, cuyo objetivo original fue sustituir las máquinas paralelas y supercomputadores que representaban altos costes, para cómputo de alto rendimiento. Actualmente, se utilizan como plataformas para objetivos distintos:

- **Cluster para balance de carga:** se usan para compartir la carga computacional entre múltiples tareas que pueden o no estar relacionadas, se distribuye la carga de trabajo de manera eficiente entre los procesadores; son usados tanto en ambientes científicos y académicos, como en empresas comerciales.
- **Cluster de alta disponibilidad:** proveen redundancia de datos y servicios (sistemas de respaldo) y actúan como granjas de servidores para soportar grandes cantidades de peticiones por parte de millones de clientes (servicios de Google, de Amazon, etc.).
- **Cluster de alto desempeño:** para la ejecución de programas paralelos, principalmente usados en ambientes científicos y académicos.

A pesar de sus beneficios, los *clusters* presentan limitaciones relacionadas con el consumo de potencia energética que requieren y la adecuación particular del espacio físico. Esto limita la agregación, escalabilidad y mantenimiento. Se habla de potencia/superficie. Así, para el ámbito de computación de alto desempeño, en el que las aplicaciones requieren cada vez más poder de cómputo, se plantea la solución de agregar recursos distribuidos geográficamente a través de redes de alta velocidad. Esto se conoce como **computación ubicua**. En esta clasificación entra la computación voluntaria, los *grids* computacionales y la computación en la nube.

Todas las arquitecturas explicadas en este tema son capaces de soportar programas paralelos. Dependiendo de la aplicación que se desea parallelizar y de la arquitectura sobre la cual se pretende ejecutar dicha aplicación, los diseñadores deberán tomar consideraciones particulares. Sin embargo, existen pautas generales que se deben seguir en cualquier escenario, como se verá en el siguiente tema.

### 3. Técnicas de diseño de programas paralelos

Existen diferentes niveles de granularidad que se pueden alcanzar con las técnicas paralelas. En general se distinguen los siguientes cuatro niveles de granularidad:

- **Nivel de tareas (Job level):** a este nivel se le denomina de **granularidad gruesa**, pues se trata de la administración de diferentes procesos, independientes o no, que son asignados a diferentes procesadores. Incluso se pueden asignar hilos de ejecución de un mismo proceso a diferentes procesadores, si se trata de una arquitectura de memoria compartida. Este nivel de granularidad es manejado por el sistema operativo, quien controla la asignación de los procesos o los hilos de ejecución a los diferentes procesadores.
- **Nivel de Programa (Program level):** este nivel de granularidad se logra cuando diferentes procesadores pueden ejecutar diferentes secciones de un mismo programa en paralelo. Dependiendo del tamaño de las secciones del programa que se ejecutan en paralelo, se obtienen diferentes subniveles de granularidad fina, media y gruesa.
  - Si las secciones que se ejecutan en paralelo son tramos o grupos de instrucciones, se llama **granularidad fina** y es un nivel de paralelismo que lo puede definir el compilador, el programador a bajo nivel o el sistema operativo.
  - Si las secciones en paralelo son diferentes iteraciones de un ciclo sobre datos independientes (como matrices, por ejemplo), se trata de **granularidad media** y puede ser definido a

nivel del compilador o a nivel del lenguaje de programación con instrucciones explícitas que indican que se ejecutarán en paralelo.

- Si las partes que se ejecutan en paralelo son rutinas completas, se habla de **granularidad gruesa** y debe ser definida por el programador con instrucciones o librerías explícitas que indican que se ejecutarán en paralelo.

Es importante notar que el nivel de granularidad a nivel de programa, frecuente implica tener datos compartidos.

- **Nivel de instrucciones (Instruction level):** este nivel es más íntimo a la organización del procesador y ocurre en el caso específico de los procesadores encauzados (*pipelines*), donde la ejecución de varias instrucciones se efectúa en diferentes etapas del procesador de forma solapada en el tiempo. Se considera granularidad muy fina.
- **Nivel de Aritmética y bits (Bit level):** en este nivel, el paralelismo se verifica a nivel aritmético, es decir, cuando se tienen varios procesadores trabajando en paralelo sobre diferentes secciones de bits de los datos en sí (especie de SIMD a nivel de bits). Este tipo de procesadores se compone de enlaces de varios bits. Es solo responsabilidad del *hardware* y del sistema operativo explotar este nivel de granularidad, que se considera el nivel más fino de granularidad.

Desde el punto de vista de los diseñadores y programadores de aplicaciones paralelas, el interés es explotar granularidad a nivel de programa. El diseño de tales algoritmos paralelos no es una tarea fácil que pueda ser explicada en una serie de pasos sencillos. Por el contrario, requiere de **gran creatividad** por parte del diseñador o programador. Sin embargo, un enfoque metodológico puede facilitar esta tarea. Foster (1995), propone una metodología de diseño de programas paralelos que maximiza el rango de posibles opciones de parallelización de un problema, provee mecanismos para evaluar las alternativas consideradas y reduce el esfuerzo de probar varias opciones. Este enfoque metodológico da lugar a la creatividad y a la intuición del diseñador/programador para producir buenos algoritmos paralelos.

### 3.1. Modelos de programas paralelos: SPMD, MPMD y maestro/esclavos

Anteriormente se describieron las diferentes clasificaciones de las arquitecturas de *hardware*, dependiendo del flujo de instrucciones y el flujo de datos que se pueden asignar a los diferentes procesadores que conforman la plataforma de hardware. Éstas son SISD, SIMD, MISD y MIMD. En el área de computación de alto rendimiento surgen dos términos que erróneamente se asocian a esas clasificaciones. Se trata de los términos **SPMD** (por *Single Process Multiple Data*) y **MPMD** (por *Multiple Process Multiple Data*). Mientras SISD, SIMD, MISD y MIMD, representan una clasificación de plataformas de hardware, SPMD y MPMD tratan de **técnicas para desarrollar programas paralelos**. Ambos modelos de programación paralelos se pueden ejecutar en arquitecturas MIMD y se popularizaron desde la aparición en 2006, de la lista TOP500 predominada por arquitecturas MIMD. Adicionalmente, ambos modelos de programas se pueden estructurar con un **enfoque maestro-esclavos** o un **enfoque de procesos “amigos”**.

En el patrón maestro-esclavo uno de los procesos juega el papel de *maestro* y controla la actividad de un grupo de procesos *esclavos*, asignando el trabajo que se ha de realizar en paralelo y ocupándose también de las operaciones de E/S. Dado un problema de grandes dimensiones, se divide y se distribuye sobre los *esclavos* disponibles. Cada *esclavo* calcula los resultados parciales de manera independiente y se los envía al *maestro*, así permite el procesamiento en paralelo, pues cada *esclavo* trabaja independientemente de los demás. Las tareas del *maestro* consisten en distribuir trabajo a los componentes *esclavos*, iniciar la ejecución de los *esclavos*, recolectar los resultados parciales de los *esclavos* y generar el resultado final a partir de estos resultados parciales de los diferentes *esclavos*.

Por su parte, en el patrón de grupo de procesos amigos, no existe un proceso coordinador que controle las tareas de los demás. En contraste, todos los procesos tienen autonomía, conocen las tareas que deben realizar sobre los datos que posee y eventualmente se comunican con los demás procesos para colaborar u obtener datos complementarios que requieran.

**SPMD es un modelo de programación paralelo que asume que múltiples procesadores cooperantes ejecutan el mismo programa sobre un conjunto de datos diferentes.** Se ha convertido en el modelo de programación paralelo predominante en el mundo de la computación de alto desempeño. El modelo SPMD fue propuesto por Darema (1998) en el contexto del proyecto *Parallel Processor Prototype* (RP3) de IBM. SPMD puede combinarse con el enfoque maestro-esclavo o de procesos de amigos.

**MPMD es un modelo de programación paralelo que asume que múltiples procesadores cooperantes ejecutan programas diferentes sobre un conjunto de datos diferentes.** Típicamente sigue un modelo maestro-esclavos, en el que existe un programa coordinador, que genera los datos para los demás programas. Cada esclavo ejecuta el programa que el maestro le indica. Estos otros programas retornan sus resultados directamente al coordinador.

### 3.2. Diseño Metodológico

Un problema computacional puede tener varias opciones de parallelización que dependen de la versión secuencial que se está considerando, del hardware disponible para su ejecución y del nivel de concurrencia/paralelismo deseado. El enfoque de diseño metodológico propuesto por Foster (1995) sugiere considerar los aspectos independientes del *hardware* (tales como el nivel de paralelismo deseado) en las primeras etapas de diseño y dejar para las etapas finales los aspectos dependientes de la arquitectura. Foster (1995) propone cuatro etapas de diseño:

- **Particionamiento:** esta etapa consiste en reconocer el cómputo y los datos que se pueden descomponer en pequeñas tareas, sin considerar aspectos de hardware como el número de nodos de la plataforma paralela en la que se ejecutará.
- **Comunicación:** en esta etapa se identifica la comunicación requerida para sincronizar y coordinar la ejecución de las tareas, además se definen las estructuras y algoritmos de comunicación apropiados.

- **Aglomeración:** para mejorar el desempeño y los costes de implementación, en esta etapa se evalúan las tareas y las estructuras de comunicación definidas en las etapas anteriores, y de ser necesario las pequeñas tareas se agrupan en tareas más grandes.
- **Mapeo:** esta etapa consiste en asignar cada tarea o grupo de tareas a un procesador, de manera de maximizar la utilización de los nodos y minimizar los costes de comunicación; puede ser especificado de manera estática o determinado en tiempo de ejecución usando algoritmos de balance de carga.

Esta metodología es llamada **PCAM** por sus siglas en inglés de *Partitioning, Communication, Agglomeration, y Mapping*. La Figura 15 muestra el esquema de las etapas del modelo PCAM.

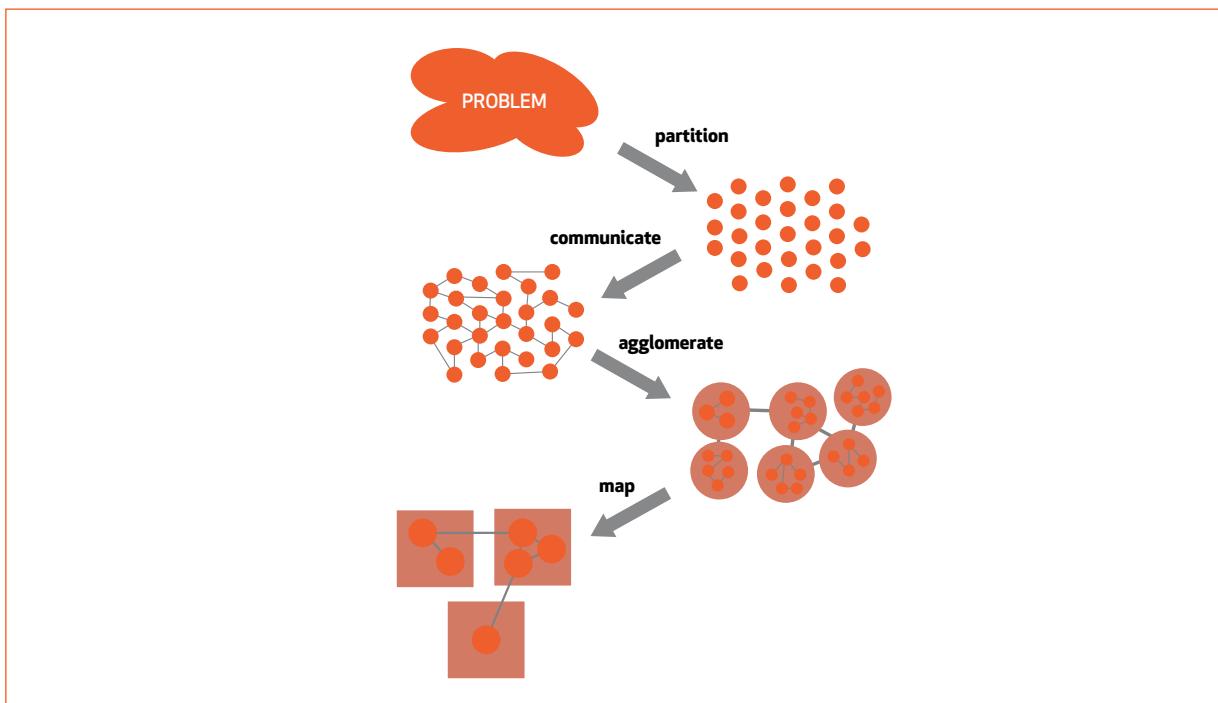


Figura 15. Diseño metodológico de programas paralelos. Recuperado de Foster (1995).

Antes de entrar en detalle en la metodología a seguir para diseñar algoritmos paralelos es importante que el diseñador/programador reflexione sobre los siguientes aspectos:

- Asegurarse de entender bien el problema y dominar correctamente el código secuencial que se desea parallelizar.
- Identificar los puntos álgidos del programa:
  - Partes del programa que consumen la mayor cantidad de tiempo computacional (ejecución de gran trabajo computacional); esas serán las partes más apropiadas a parallelizar.
  - Se debe hacer uso de **herramientas de perfilamiento** que ayuden a identificar tales puntos álgidos del programa.

- Identificar los cuellos de botella en el programa:
  - Algunas secciones del código pueden ser desproporcionalmente lentas.
  - Siempre es posible reestructurar el código para minimizar los cuellos de botella.
- Algunas veces es posible identificar diferentes algoritmos computacionales que presentan mejores propiedades de escalabilidad.

Una vez el diseñador/programador tenga claros estos puntos, deberá pasar al diseño de la versión paralela del algoritmo, siguiendo una metodología. Ahora veremos en detalle cada etapa de la metodología PCAM.

### 3.2.1. Particionamiento

La etapa de particionamiento persigue exponer las mejores oportunidades para la ejecución paralela. Así, la idea es **definir un gran número de pequeñas tareas, lo que conlleva a una descomposición de granularidad fina, que a su vez provee la mayor flexibilidad en términos de algoritmos paralelos potenciales**. Una buena descomposición, divide el problema en pequeñas piezas de cómputo y de datos sobre los cuales el cómputo es realizado. Existen diferentes estrategias de particionamiento:

- **Descomposición por dominio:** en esta etapa de particionamiento, normalmente los programadores se enfocan en dividir los datos asociados al problema en pequeñas y adecuadas piezas, preferiblemente de igual tamaño, y luego se concentran en asociar los datos con el cómputo. Si una operación requiere datos de otras tareas, se requiere comunicación. Los programadores se deben concentrar en las estructuras de datos más grandes y más frecuentemente accedidas. La Figura 16 ilustra varios ejemplos de descomposición por dominio.

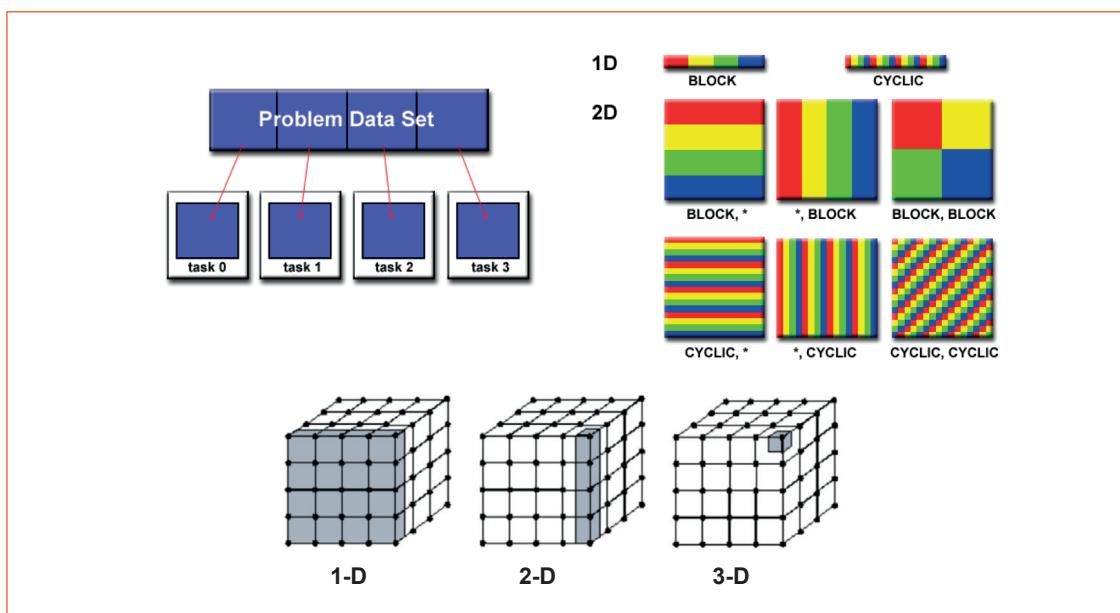
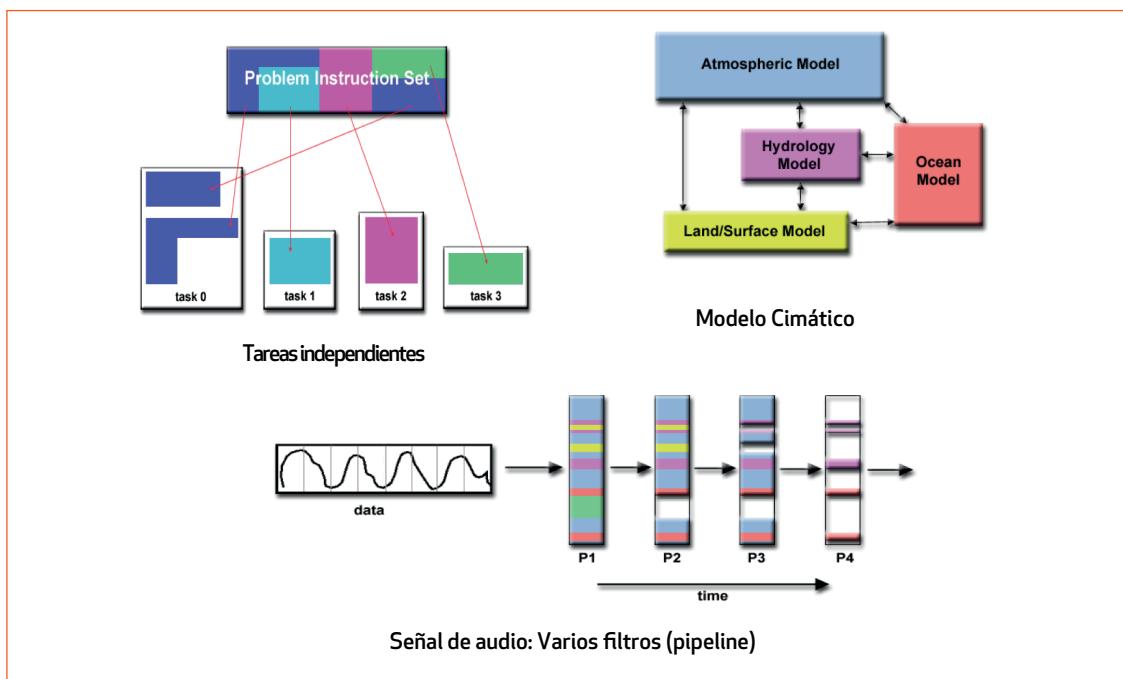


Figura 16. Ejemplos de descomposición por dominio. Adaptado de Foster (1995).

- **Descomposición funcional:** un enfoque alternativo es, primero descomponer el cómputo en piezas y luego determinar cómo asociar los datos con el cómputo. Si se comparten datos entre las tareas, se requiere comunicación entre ellas. Se puede lograr diferentes descomposiciones funcionales:
  - **Tareas independientes:** una descomposición funcional particular consiste en subdividir el cómputo en tareas que no son dependientes entre sí. Esto se logra en **algoritmos embarazosamente paralelos**. Es decir, problemas que presentan tareas que por naturaleza son independientes y paralelas.
  - **Paralelismo de arreglos:** identificar las tareas que se aplican simultáneamente a vectores, matrices u otros arreglos.
  - **Divide y conquista:** implica dividir el problema recursivamente en árboles, tales como jerarquía de subproblemas.
  - **Encausamiento (pipelining):** otra posible manera de dividir las tareas es en secuencia de estados independientes, de manera que diferentes tareas puedan estar en diferentes estados de una secuencia.

La Figura 17 ilustra varios ejemplos de descomposición funcional.



Todas estas técnicas son complementarias y pueden ser aplicadas a diferentes componentes de un mismo problema para obtener algoritmos paralelos alternativos.

En esta primera etapa del diseño, se debe evitar la replicación del cómputo y de los datos. Es decir, se deben definir tareas que partitionan cómputo y datos en conjuntos disjuntos. Este aspecto se

revisa en las próximas etapas, si se detecta que con la replicación se reducen los requerimientos de comunicación.

Para finalizar la etapa de particionamiento, es necesario revisar las siguientes propiedades deseables:

- ¿Se logra la máxima ejecución concurrente posible de las tareas resultantes?
- ¿Hay muchas más tareas que procesadores disponibles?
- ¿Se incrementa el número de tareas (en lugar del tamaño de cada tarea) a medida que se aumenta el tamaño del problema?
- ¿Es razonablemente uniforme el tamaño de las tareas?
- ¿Se evita la redundancia en cómputo y datos?

Si la respuesta a todas estas preguntas es afirmativa, se habrá logrado una buena partición del problema.

### 3.2.2. Comunicación

Las tareas generadas por el particionamiento en la etapa anterior se deben ejecutar de manera paralela, pero, en general, no de manera independiente. Es posible que algunas tareas requieran datos generados por otras tareas para completar el cómputo que les corresponde realizar. Dado que cada tarea almacena de manera local sus datos, éstos deben ser transferidos entre las tareas que los requieren. En esta segunda etapa, se debe establecer el flujo de comunicación necesario.

Para el caso de descomposición por dominio, **los requerimientos de comunicación pueden ser difíciles de determinar**. La descomposición por dominio implica el particionamiento de las estructuras de datos en subconjuntos disjuntos y luego se asocian las operaciones a cada conjunto de datos. Algunas de estas operaciones requieren datos mantenidos o producidos por otras tareas, por lo tanto, se requiere de comunicación para manejar la transferencia de dichos datos. Se requiere entonces, organizar esta comunicación de manera eficiente.

En contraste, **los requerimientos de comunicación en algoritmos paralelos obtenidos por descomposición funcional son generalmente identificados de manera directa**: la comunicación corresponde al flujo de datos entre las tareas.

Independientemente de la forma cómo se llevó a cabo el particionamiento, los modelos de comunicación son determinados por las dependencias de datos entre tareas. Estos modelos de comunicación pueden ser:

- **Local o global:** en la comunicación local, cada tarea se comunica con un conjunto pequeño de otras tareas, por ejemplo con sus tareas vecinas (e.g., modelo de grupo de amigos); en contraste, la comunicación global requiere que cada tarea se comunique con muchas tareas (e.g., modelo maestro-esclavos); en ambos casos es importante que el esquema de comunicación que se defina, permita la comunicación concurrente entre diferentes tareas; la

Figura 18 muestra un ejemplo de comunicación local y un ejemplo de comunicación global; en el caso de la comunicación local mostrado en la Figura 18(a) es posible que varias tareas se comuniquen concurrentemente con sus vecinas sin interferir con las otras comunicaciones, mientras que el caso de la comunicación global mostrado en la Figura 18(b) no permite tal concurrencia: si una tarea se está comunicando con todas las demás, ninguna otra podrá proceder con su comunicación.

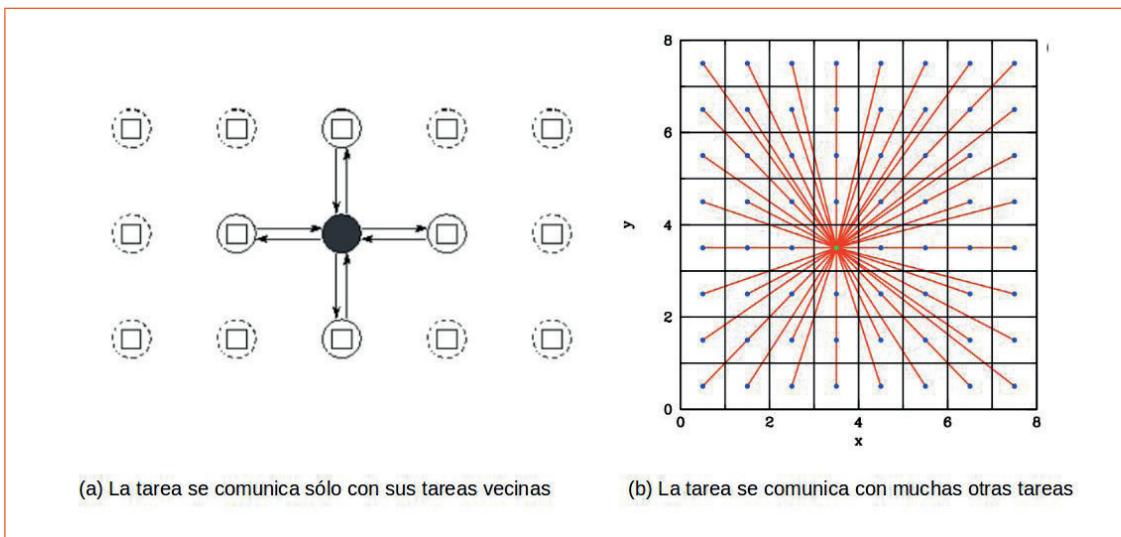


Figura 18. (a) Ejemplo de comunicación local; (b) Ejemplo de comunicación global. Adaptado de Foster (1995).

- **Estructurada o no estructurada:** en la comunicación estructurada, una tarea y sus vecinas forman una estructura regular, tal como un árbol o una malla; por su lado las redes de comunicación no estructuradas pueden ser representadas por grafos arbitrarios. Los dos ejemplos mostrados en la Figura 18 representan esquemas de comunicación estructurados. La Figura 19 muestra dos esquemas de comunicación no estructurados, cada punto en las figuras representa una tarea y los arcos representan necesidades de comunicación.

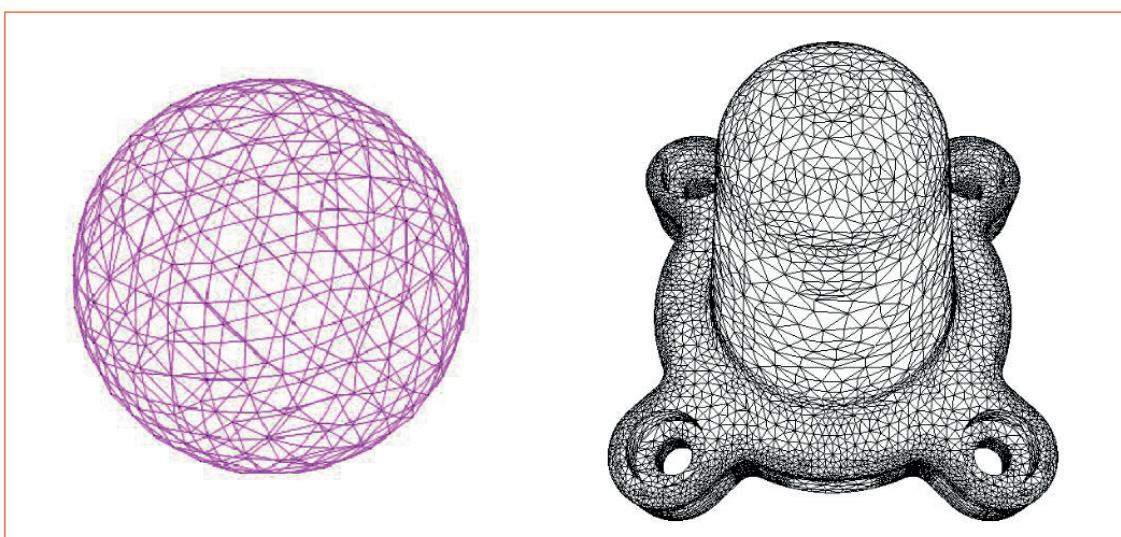


Figura 19. Ejemplos de esquemas de comunicación no estructurados. Adaptado de Foster (1995).

- **Estática y dinámica:** en la comunicación estática, la identidad de los modelos de comunicación no cambia en el tiempo; en contraste, la identidad de los patrones de comunicación de las estructuras de comunicación dinámica puede ser determinada por los datos computados en tiempo de ejecución y puede resultar altamente dinámica.
- **Síncrona o asíncrona:** en la comunicación síncrona, las tareas que envían y reciben los datos, se deben ejecutar de manera coordinada como pares productor/consumidor cooperando en operaciones de transferencia de datos; también es conocida como **comunicación bloqueante**. La tarea A envía un mensaje o datos y debe esperar hasta que la tarea B lo reciba, para poder continuar su ejecución. Por el contrario, la comunicación asíncrona implica que la tarea que recibe los datos lo hace sin cooperación de la tarea que los envía; también se le llama **comunicación no bloqueante**. Una vez una tarea A envía un mensaje, puede continuar su ejecución sin esperar a que la tarea B lo reciba; este tipo de comunicación requiere cuidado para evitar las condiciones de carrera e interbloqueos.

### Ejemplo de diversos esquemas de comunicación

Con el siguiente ejemplo se ilustrarán diferentes esquemas de comunicación para un mismo problema. Considere que se desea paralelizar una **operación de reducción**, es decir una operación que reduce  $N$  valores distribuidos en  $N$  tareas, usando un operador asociativo tal como la adición:

$$S = \sum_{i=0}^{N-1} X_i$$

Supongamos que se tienen  $N=8$  valores y tareas. A continuación se proponen tres modelos de comunicación para la versión paralela de este problema (ver Figura 20):

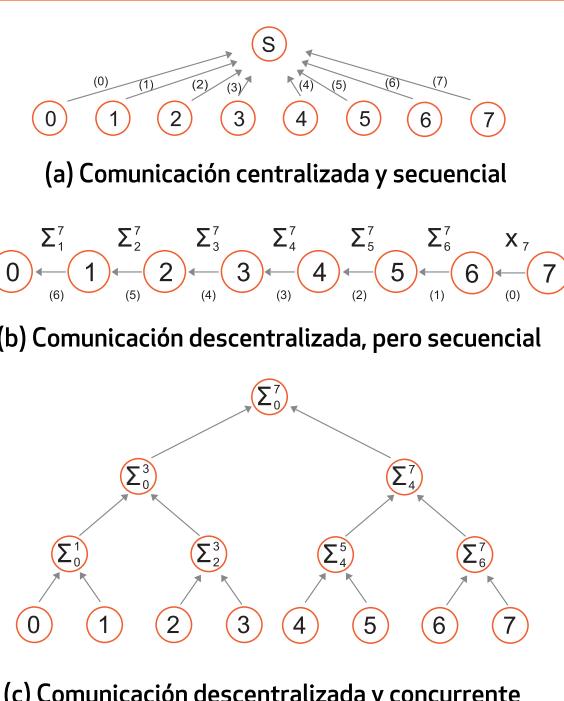


Figura 20. Varias estrategias de comunicación para el problema de sumatoria. Adaptado de Foster (1995).

- La primera estrategia de comunicación que muestra la Figura 20(a) implica un coordinador que recibe los valores de las demás tareas y realiza la sumatoria (representa un modelo **maestro-esclavos**). En este caso, dado que el coordinador está envuelto en cada operación de comunicación, solo una tarea a la vez se puede comunicar con el coordinador, convirtiendo el esquema de comunicación en un **modelo centralizado** (no se distribuye el cómputo ni la comunicación) y **secuencial** (no permite operaciones de comunicación ni cómputo concurrentes). Para el caso de  $N=8$ , se requieren 8 operaciones de comunicación. Además, si todas las tareas requieren la suma, el coordinador deberá enviar el resultado a cada tarea, en cuyo caso se requieren  $2N=16$  operaciones de comunicación. Es decir que la complejidad del algoritmo se convierte en  $O(N)$ . ¡Es una versión paralela muy pobre!
- Una segunda opción de modelo de comunicación para este problema se muestra en la Figura 20(b). Se trata de conectar las  $N$  tareas en un arreglo unidimensional (grupo de tareas amigas). La tarea  $N-1$  envía su valor a su tarea vecina  $N-2$ , la tarea  $N-2$  suma su valor con el recibido de la tarea  $N-1$  y lo pasa a su tarea vecina  $N-3$ , y así sucesivamente. Al final del arreglo, la tarea 0 recibe la suma parcial y suma su valor local para obtener el resultado final. Este **modelo de comunicación es descentralizado** (distribuye las operaciones de cómputo y de comunicación), pero aún sigue siendo **secuencial** (no hay comunicación ni cómputo concurrente, cada tarea debe esperar por el resultado parcial de su vecina de la derecha). Se requieren  $N-1=7$  operaciones de comunicación para que la tarea 0 obtenga el resultado final y  $2(N-1)=14$  operaciones de comunicación si todas las tareas requieren el resultado. Aun cuando se reduce un poco la cantidad de operaciones de comunicación, la complejidad del algoritmo sigue siendo de  $O(N)$ .
- La tercera opción de modelo de comunicación se muestra en la Figura 20(c). Aplicando el método de **divide y conquista**, las tareas se organizan en un árbol binario (grupo de tareas amigas). La idea es dividir el problema en dos subproblemas más pequeños de más o menos de igual tamaño (sumar  $N/2$  números), que pueden ejecutarse de manera concurrente:

$$\sum_{i=0}^{N-1} x_i = \sum_{i=0}^{N/2-1} x_i + \sum_{i=0}^{N-1} x_i$$

El problema se sigue subdividiendo recursivamente para formar el árbol mostrado en la Figura 20(c). Así, el **esquema de comunicación es descentralizado** (no hay coordinador que participe en todas las operaciones de cómputo y de comunicación) y **concurrente** (se distribuyen las operaciones de cómputo y de comunicación, por ejemplo, los pares de tareas 0–1, 2–3, 4–5 y 6–7, pueden ejecutar sus operaciones de cómputo y comunicación concurrentemente, en el nivel más bajo del árbol). Para el caso de  $N=8$ , se requieren de  $\log_2 N=3$  pasos de comunicación. Si es necesario que todas las tareas tengan el resultado, el algoritmo completo requiere  $2(\log_2 N) = 6$  operaciones de comunicación. La complejidad del algoritmo se reduce a  $O(\log_2 N)$ . El resultado es un algoritmo paralelo eficiente con una estructura de comunicación regular y local, en el que cada tarea se comunica con un conjunto pequeño de vecinos.

## Latencia y ancho de banda

Un aspecto importante relacionado a la comunicación es el **coste** (*overhead*) que pueda representar. Ese coste se mide en términos de **latencia y ancho de banda**.

- **La latencia** mide el tiempo que toma enviar el mensaje más pequeño (de un bit) de un nodo A a otro nodo B.
- **El ancho de banda** representa la cantidad de datos que pueden ser comunicados por unidad de tiempo.

Algunos de los factores que se deben considerar para reducir el coste de comunicación son:

- Si se envían muchos mensajes pequeños, causará que la latencia domine el coste de la comunicación: **es mejor empaquetar muchos mensajes pequeños en un solo mensaje grande.**
- **Mientras menos información se requiera transmitir, menor serán los tiempos de comunicación requeridos.**
- **Generalmente es más conveniente hacer que toda la comunicación necesaria ocurra al mismo tiempo.**

Una vez definido el particionamiento y la comunicación, se deben revisar las siguientes propiedades deseables para evaluar la eficiencia, efectividad y conveniencia de la comunicación:

- ¿Todas las tareas realizan aproximadamente el mismo número de operaciones de comunicación? ¿La comunicación es balanceada y razonablemente uniforme? Si la comunicación resulta no balanceada, puede representar problemas de escalabilidad.
- ¿La comunicación es altamente localizada (se realiza entre tareas vecinas)? Siempre es preferible definir esquemas de comunicación local ante los patrones de comunicación global para asegurar mejor desempeño y escalabilidad.
- ¿Los recursos de red son usados concurrentemente? ¿Es posible que las operaciones de comunicación procedan concurrentemente?
- ¿La comunicación no inhibe la concurrencia entre las tareas? ¿Se minimiza la frecuencia y volumen de la comunicación? ¿Se solapa lo máximo posible la comunicación con el cómputo?

Si estas preguntas tienen una respuesta afirmativa, se habrá logrado un esquema de comunicación eficiente y escalable.

Las primeras dos etapas de diseño, particionamiento y comunicación, logran paralelizar el problema. Sin embargo, el diseño obtenido hasta este punto probablemente no se adapta a la máquina real sobre la cual se ejecutará la solución paralela. Por ejemplo, si el número de tareas excede por mucho al número de procesadores disponibles, el desempeño será fuertemente afectado por la forma cómo

sean asignadas dichas tareas a los procesadores. Así, es importante conocer o decidir qué tipo de arquitectura se usará para ejecutar el algoritmo paralelo:

- ¿Es un multiprocesador centralizado o un multicomputador?
- ¿Qué modelos de comunicación son soportados?
- ¿Cómo se deben combinar las tareas de manera de asignarlas de manera efectiva a los procesadores?

Estos aspectos se abordan en la siguiente etapa del enfoque metodológico de diseño, en la que entra en juego los aspectos relacionados a la arquitectura de hardware que se usará.

### 3.2.3. Aglomeración

El particionamiento del problema a nivel de granularidad muy fina generalmente no resulta en un algoritmo paralelo eficiente, dado que se requiere mucha comunicación de datos. Así, la aglomeración es necesaria para lograr **localidad de datos y buen desempeño**.

**La etapa de aglomeración consiste en agrupar muchas tareas de grano fino** (tareas primitivas), **en menos tareas más grandes de granularidad mediana**. La aglomeración debe tomar en cuenta los detalles del problema, para generar un algoritmo paralelo con buenas propiedades de escalabilidad y eficiencia. Otro aspecto que se evalúa en la etapa de aglomeración es la posibilidad de replicar cómputo y datos.

La aglomeración persigue los siguientes objetivos:

- **Reducir los costes de comunicación:** generalmente se logra creando en cada procesador una aglomeración de tareas que se comunican (se pueden definir grupos de tareas enviadoras y receptoras), así se elimina la comunicación entre ellas, ya que los datos están locales en el procesador. Es decir, se reduce el número de operaciones de comunicación y el tamaño de las comunicaciones. En la Figura 21(a) se muestra una malla de  $8 \times 8 = 64$  datos. Si cada dato se asigna a una tarea, actualizar 16 datos requerirá de  $16 \times 4$  operaciones de comunicación. Cada tarea envía un mensaje con un dato a sus 4 tareas vecinas. Si se aglomeran submatrices de  $4 \times 4$  para cada tarea, se reduce el número de tareas, como lo muestra la Figura 21(b). En este caso, actualizar 16 datos requerirá de solo 4 operaciones de comunicación. Cada tarea envía un mensaje con 4 datos a sus 4 tareas vecinas. Se ve claro con este ejemplo, cómo aumentando la granularidad, se reducen las operaciones de comunicación.

Otra estrategia que puede ayudar a reducir la comunicación es **replicando tanto cómputo como datos. Queda de tarea para el alumno conseguir dos ejemplos en el que la replicación de datos y cómputo mejora el desempeño de un algoritmo paralelo.**

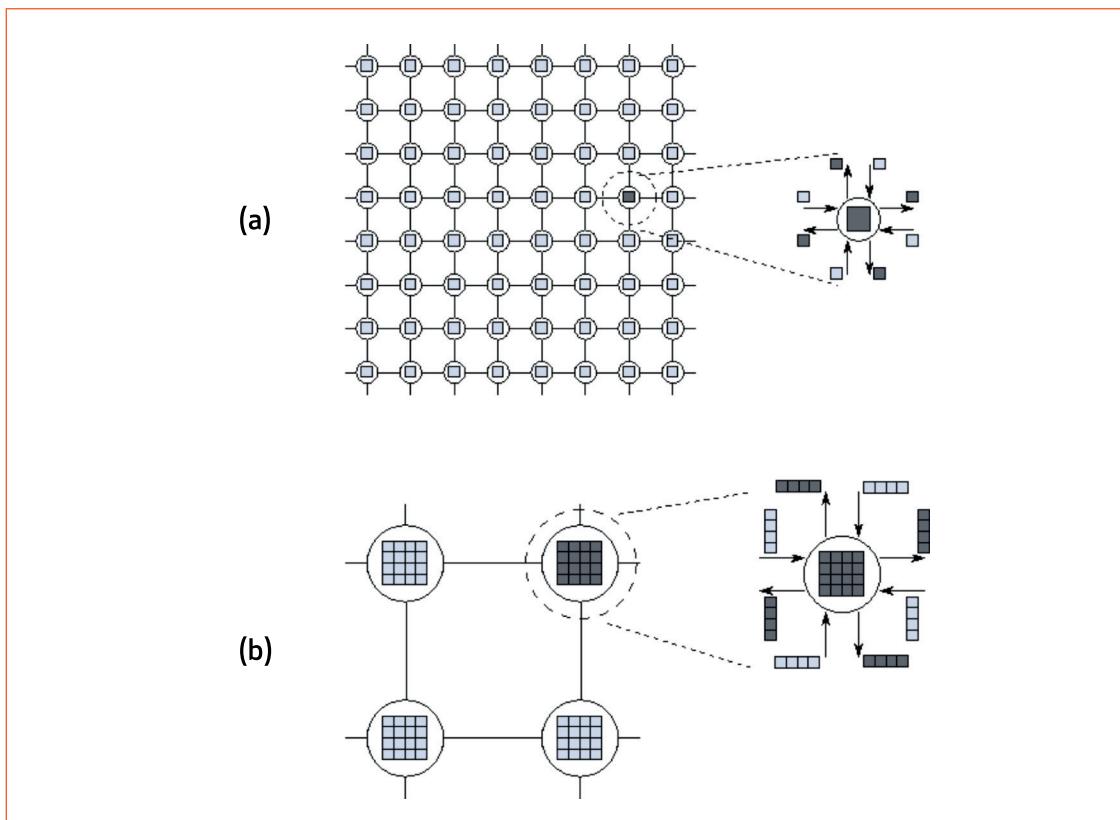


Figura 21. Aglomeración de tareas. Adaptado de Foster (1995).

- **Mejorar el desempeño de los algoritmos paralelos**, dado que se reducen los costes de comunicación y se incrementa la granularidad del cómputo. La comunicación siempre es más lenta que el cómputo.
- **Mantener la escalabilidad del programa**. Con el siguiente ejemplo se ilustra cómo la escalabilidad y flexibilidad del algoritmo es afectada por la decisión de aglomeración: supongamos que se está manipulando una matriz 3D de tamaño  $8 \times 128 \times 256$  y la arquitectura sobre la cual se ejecutará el programa es un multiprocesador centralizado con 4 procesadores, si se aglomera la 2da y 3era dimensión, ¿se puede ejecutar sobre esa arquitectura? La respuesta es sí, porque se tendrían 4 tareas que son responsables de una submatriz de  $2 \times 128 \times 256$ , cada una. Si luego se tiene la misma arquitectura, pero con 8 CPUs, ¿el diseño previo de aglomeración sigue siendo válido? De nuevo la respuesta es afirmativa porque se tendrán 8 tareas con una submatriz de  $1 \times 128 \times 256$ , cada una. Ahora, ¿qué pasa si se tienen más de 8 procesadores? ¿Sigue siendo válido el mismo diseño de aglomeración? En este caso es claro que no, se debe cambiar el diseño de aglomeración para que la escalabilidad no esté limitada a solo 8 procesadores. El nuevo diseño de aglomeración debería ser capaz de funcionar con un número mayor de procesadores.
- **Simplificar la programación**, es decir reducir los costes de ingeniería de software. Entendiendo que la ingeniería de software estudia las técnicas para asegurar que los proyectos grandes y complejos se finalicen a tiempo y bajo el presupuesto establecido, una técnica es buscar un algoritmo secuencial disponible para realizar la parallelización a partir de éste, en lugar de comenzar la programación paralela desde cero.

Para evaluar la calidad de la aglomeración, es necesario revisar los siguientes aspectos:

- ¿Se redujeron los costes de comunicación mediante el incremento de la localidad del algoritmo paralelo?
- Si se realizó una replicación de cómputo, ¿los cómputos replicados consumen menos tiempo que la comunicación que sustituyeron? ¿sigue siendo eficiente para un amplio rango de tamaño del problema y número de procesadores?
- Si se replicaron datos, ¿la replicación de los datos no afecta la escalabilidad?
- Las tareas aglomeradas ¿tienen costes de comunicación y cómputo similares?
- ¿El número de tareas escala (se incrementa) con el tamaño del problema?
- ¿El número de tareas está acorde con la plataforma sobre la cual se ejecutará el programa paralelo sin limitar la escalabilidad?
- ¿Se puede reducir el número de tareas aún más sin comprometer la escalabilidad, el balance de carga y los costes de la ingeniería de software?
- Si se está parallelizando un algoritmo secuencial existente, ¿la diferencia (*tradeoff*) entre los costes de programar desde cero y los costes de modificaciones del código es razonable?

Nuevamente, si son positivas las respuestas a todas estas interrogantes, es altamente probable que la estrategia de aglomeración resulte conveniente y eficiente.

### 3.2.4. Mapeo

La última etapa del enfoque metodológico para diseñar programas paralelos corresponde a la **asignación de tareas a procesadores**, llamada mapeo. **Las tareas se deben asignar a los procesadores físicos para que se ejecuten**. Esta asignación se puede realizar de diferentes maneras, incluyendo asignar una tarea por procesador o múltiples tareas por procesador. Lo importante es que la semántica del programa no se vea afectada ni dependa del número de procesadores o de una estrategia de mapeo particular. Además, es importante tener presente que el rendimiento del algoritmo se ve impactado por la estrategia de asignación de tareas a procesadores pues puede afectar el nivel de concurrencia, el balance de carga, los patrones de comunicación, etc. Así, **el objetivo primordial del mapeo es minimizar el tiempo total de ejecución del algoritmo paralelo**.

Algunas consideraciones importantes para realizar el mapeo son:

- Los canales de comunicación entre tareas pueden o no corresponder a las conexiones físicas de la red de interconexión entre los procesadores.
- Las tareas que se pueden ejecutar concurrentemente se deben asignar a procesadores diferentes.
- Las tareas que se comunican frecuentemente se deberían asignar al mismo procesador.

Con estas consideraciones el mapeo debe lograr, al igual que la aglomeración, maximizar la concurrencia, minimizar la comunicación y mantener el balance de carga.

Con varias de las estrategias de descomposición por dominio, la etapa de aglomeración logra decrementar el número de tareas de granularidad media a exactamente el número de procesadores. En estos casos el mapeo es directo: una tarea por procesador. En otros casos, se tienen más tareas que procesadores, por lo tanto, el mapeo debe garantizar **balance de carga** entre los procesadores.

El balance de carga se refiere a mantener los procesadores con una carga de trabajo aproximadamente similar. Si no se asegura balance de carga, algunos procesadores se pueden quedar ociosos mientras otros están sobrecargados de trabajo. Para problemas con descomposición funcional o diseño maestro/esclavos, el balance de carga puede significar un reto significativo. La Figura 22, muestra un ejemplo de un grafo que establece los requerimientos de comunicación de tareas (Figura 22(a)) y el mapeo realizado en 3 procesadores (Figura 22(b)). La comunicación entre los procesadores no es equitativa: P1 tiene más carga de comunicación. Además, si todas las tareas requieren aproximadamente la misma cantidad de tiempo y cada CPU tiene la misma capacidad, este mapeo causará que el procesador P1 tenga el doble de la carga de los procesadores P0 y P2.

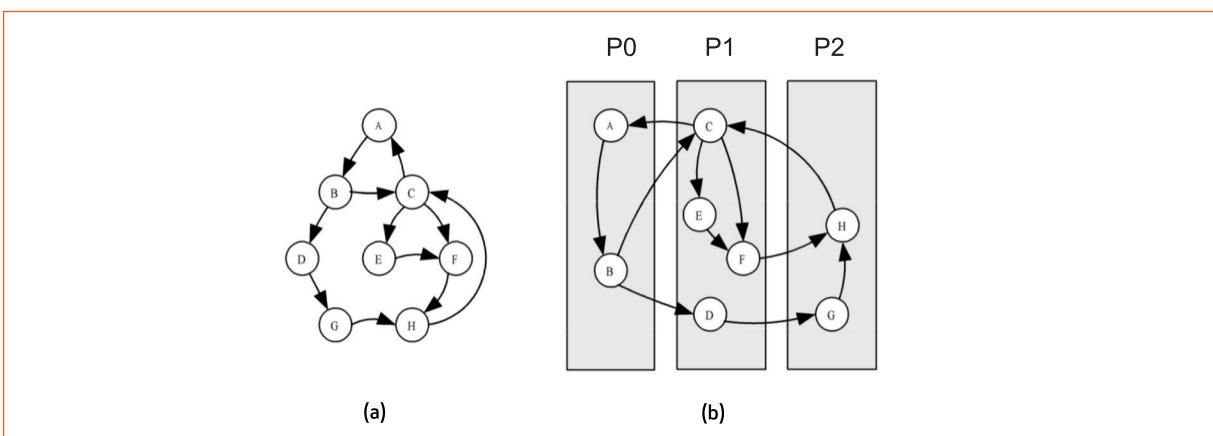


Figura 22. Ejemplo de mapeo con mal balance de carga.

El problema del balance de carga no es una tarea fácil de resolver. Existen algoritmos sencillos de distribución de trabajo que no necesariamente aseguran balance de carga y algoritmos complejos de distribución estática y dinámica, para asegurar un balance de carga pero son costosos en tiempo. Se conoce que el problema de encontrar un mapeo óptimo es  $NP$ -completo. Así que normalmente se basan en heurísticas aplicadas manualmente o por el sistema operativo. Lo importante es asegurar que la relación utilización del procesador y la comunicación sea adecuada. Por ejemplo, con  $P$  procesadores y  $N$  tareas, si se asignan las  $N$  tareas a un solo procesador, se tendrá que la comunicación es 0, pero la utilización de los procesadores es  $1/P$ .

Quinn (1994), sugiere el siguiente algoritmo para realizar el mapeo:

- Si el **número** de tareas es **estático**
  - Para **comunicación estructurada**:
    - Con **tiempo de cómputo** por tarea **constante**:
      - Aglomerar las tareas para minimizar las comunicaciones.
      - Crear una tarea por procesador.
    - Con **tiempo de cómputo** por tarea **variable**:
      - Asignar las tareas a los procesadores de manera cíclica.
  - Para **comunicación no-estructurada**:
    - Usar un algoritmo de balance de carga estático.
- Si el **número de tareas** es **dinámico**
  - Para **comunicación frecuente** entre tareas:
    - Usar un algoritmo de balance de carga dinámico.
  - Con muchas tareas cortas, **sin comunicación interna**:
    - Usar un algoritmo de planificación de tareas dinámico.

Para complementar el algoritmo propuesto por Quinn (1998), se establecen otras estrategias de mapeo. Si se tienen  $N$  tareas y  $P$  procesadores enumerados consecutivamente, se pueden realizar las siguientes asignaciones (la Figura 23 muestra las estrategias):

- **Mapeo por bloques:** un bloque de  $P$  tareas consecutivas se asigna a cada procesador sucesivamente (Figura 23(a)).
- **Mapeo cíclico:** la tarea  $i$  es asignada al procesador  $i \bmod P$  (Figura 23(b)).
- **Mapeo reflexivo:** similar al mapeo cíclico excepto que las tareas son asignadas en orden inverso en pasos alternados (Figura 23(c)).
- **Mapeo por bloque-cíclico y por bloque-reflexivo:** los bloques de tareas son asignados a los procesadores en orden cíclico o reflexivo (Figura 23(b)).

Para dimensiones más grandes de matrices, estas estrategias de mapeo pueden aplicarse en cada dimensión.

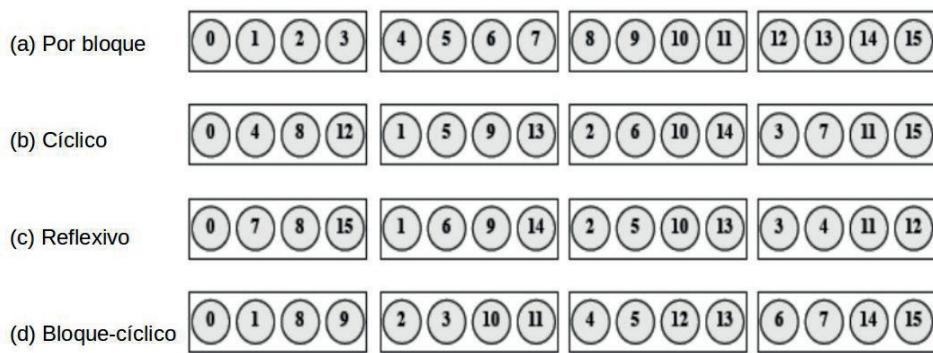


Figura 23. Estrategias de Mapeo.

Todas estas estrategias de mapeo son asignaciones estáticas (no varían durante la ejecución del algoritmo), y son apropiadas cuando el número y tamaño de las tareas es constante. Si el número y tamaño de las tareas varía durante la ejecución del algoritmo, es más aconsejable realizar un mapeo dinámico. Es decir, reasignar las tareas a los procesadores durante la ejecución para mantener un balance de carga razonable. Para que esta estrategia sea beneficiosa, la ganancia en el balance de carga debe ser mayor al coste de mover las tareas y sus datos entre los procesadores. El balance de carga dinámico normalmente se basa en intercambio de información de la carga de trabajo, así que este intercambio de información debe hacerse de manera eficiente para que no impacte en el desempeño del algoritmo.

Para evaluar la calidad del mapeo realizado, es necesario revisar las siguientes preguntas:

- Si se consideró un diseño SPMD, ¿se ha considerado un algoritmo basado en creación y eliminación de tareas dinámico?
- Si se consideró un diseño basado en creación y eliminación de tareas dinámico, ¿se ha considerado un algoritmo SPMD?
- Si se usó un esquema de balance de carga centralizado, ¿se verificó que el coordinador no represente un cuello de botella?
- Si se usó un esquema de balance de carga dinámico, ¿se evaluó el coste relativo de las diferentes estrategias?
- Si se usaron métodos probabilísticos o cílicos, ¿se tiene un número suficientemente grande de tareas para asegurar un balance de carga razonable?

Una estrategia de asignación bien evaluada será el resultado de responder afirmativamente estas preguntas.

### 3.3. Solapamiento de comunicación con cómputo

Una buena estrategia de diseño siguiendo los cuatro pasos explicados en la sección anterior, que presente un buen desempeño, debe asegurar que la comunicación se solape con el cómputo. Esto es, permitir que el cómputo avance, mientras las operaciones de comunicación se llevan a cabo. Si el cómputo de un determinado valor por parte de una tarea no depende de todas las operaciones de comunicación, no es necesario que la tarea se bloquee hasta que todas las operaciones de comunicación finalicen. Dado que el coste de comunicación es mayor que el coste de cómputo es preferible iniciar la comunicación enviando todos los mensajes primero y luego actualizar los valores internos, mientras se esperan por los valores de las tareas vecinas. Gran parte de la actualización de los datos se puede realizar en lugar de que la tarea esté esperando por mensajes.

El solapamiento de comunicación con cómputo es posible si se dispone de primitivas de comunicación no bloqueantes.

A continuación se exponen algunas consideraciones generales. Uno de los retos de la computación paralela es que la estrategia de paralelización más eficiente para cada problema requiere de una única solución. Vale la pena invertir un poco de tiempo considerando algoritmos alternativos para encontrar uno óptimo, en lugar de solo implementar lo primero que se viene a la mente. Es necesario considerar el tiempo requerido para implementar una versión paralela determinada: se puede usar un método menos eficiente, si la implementación es mucho más fácil; siempre se podrá mejorar el esquema de paralelización más tarde; solo hay que enfocarse en hacer el código paralelo primero.

**Tiempo es el factor más importante al elegir una estrategia de paralelización: ¡Nuestro tiempo!**

## 4. Análisis de aplicaciones paralelas

En el mundo de la computación de alto desempeño, la evaluación del rendimiento es uno de los tópicos que merece especial atención. La computación de alto desempeño implica el uso de los computadores más rápidos para resolver problemas más rápidos. El poder valorar el desempeño de un sistema de cómputo es necesario, en diferentes escenarios, para poder tomar decisiones. Por ejemplo, se podría determinar si un sistema existente es apto o cubre las necesidades de poder de cómputo de una aplicación específica. O se podría determinar si se requiere actualizar o ampliar sus capacidades.

En este contexto, es necesario medir el desempeño tanto de la plataforma de *hardware* como de las aplicaciones paralelas, para fines de análisis y comparación. A nivel de diseño o de actualización de un sistema, es importante poder predecir el rendimiento que tendrá el resultado final y así poder determinar si éste será adecuado para el objetivo previsto. De este análisis se podrán tomar decisiones acerca de la viabilidad del proyecto. Así mismo, es indispensable poder comparar rendimientos de diferentes opciones al momento de seleccionar el mejor sistema de cómputo. La evaluación del desempeño permite analizar el rendimiento de un sistema y tomar acciones para su perfeccionamiento.

Así, para poder analizar el desempeño de un sistema de computación y analizarlo y compararlo respecto a otro, se requiere definir y medir su rendimiento. Pero, ¿qué se entiende por rendimiento? ¿En base a qué parámetros se puede expresar o medir el rendimiento? ¿Cómo se puede establecer un mecanismo que permita comparar dos sistemas de computación?

Para cuantificar el rendimiento, es imprescindible determinar los factores que influyen en el desempeño del equipo de cómputo o de la aplicación paralela y así definir una expresión (un modelo matemático) que caracterice este rendimiento y que permita tanto evaluarlo como predecirlo. Esos factores que se representan en tal modelo matemático dependen de lo que los usuarios deseen medir para determinar el rendimiento del equipo computacional o el algoritmo paralelo. Para un usuario individual que está ejecutando un único programa, el computador con mejor rendimiento es aquel que complete la ejecución de su programa en menor tiempo. Sin embargo, para el administrador de un centro de cómputos, que tiene múltiples tareas que realizar a la vez, la plataforma de mejor rendimiento es la que realice más tareas en menor tiempo. Como elemento común, sin embargo, se evidencia que la medida del rendimiento del computador es **el tiempo**. El computador que ejecute los programas en menor tiempo es la que tiene mejor rendimiento.

Por la parte de las aplicaciones, un algoritmo secuencial es evaluado por su tiempo de ejecución como función del tamaño del problema. El comportamiento asintótico del tiempo de ejecución es idéntico en cualquier plataforma secuencial. En cambio, el tiempo de ejecución de un programa paralelo depende del tamaño del problema, del número de procesadores y de ciertos parámetros de comunicación de la plataforma. Es por ello que los algoritmos paralelos deben ser evaluados y analizados teniendo en cuenta también la plataforma. Además, en el mundo paralelo, adicional al tiempo de ejecución como medida de rendimiento, también se encuentran otras métricas como la aceleración (*speedup*) y la eficiencia (*efficiency*).

Por lo tanto, la percepción de rendimiento puede cambiar dependiendo de la situación. Se usará entonces el término de **rendimiento** como una medida de lo “**bien**” que un sistema, o los componentes que lo constituyen, lleva a cabo las tareas asignadas.

## 4.1. Aspectos de rendimiento de las arquitecturas paralelas

Para evaluar el comportamiento de un computador, se requiere obtener medidas que permitan analizar y comparar dicho comportamiento. Se denomina medida al valor obtenido mediante un instrumento de medición confiable. Una medida **proporciona una indicación cuantitativa de extensión, cantidad, dimensiones, capacidad y tamaño de algunos atributos de un proceso o producto**.

La medida del rendimiento de interés en el computador es **el tiempo**. Sin embargo, se pueden identificar diferentes medidas de tiempo. Por ejemplo, el tiempo de ejecución de programas por parte del CPU, puede descomponerse en tiempos de usuario (tiempo en que el CPU ejecuta los programas de los usuarios) y el tiempo del sistema operativo. También se pueden identificar otros tiempos que están ligados con los otros componentes del sistema de cómputo: el tiempo requerido para realizar intercambio de datos con la memoria o con diferentes dispositivos de E/S que tienen muy diversas velocidades.

### 4.1.1. Métricas de desempeño para sistemas de computación

Para poder evaluar el desempeño de un sistema de cómputo y poder comparar dos sistemas en función de su rendimiento, se requiere establecer métricas que permitan estandarizar las medidas

que se emplearán para tales fines. Las métricas establecen un criterio estandarizado para evaluar el desempeño de un sistema de cómputo de forma general, de manera que mediante la medida de tal métrica se puedan comparar dos sistemas disímiles. Todas las métricas de desempeño están basadas en el comportamiento del sistema durante el tiempo, dado que el tiempo es la medida básica de rendimiento. Existen tres clases de métricas que pueden denominarse externas, pues pueden percibirse por un usuario o entidad externa al sistema medido:

- **Latencia o tiempo de respuesta:** es una medida del tiempo que el sistema tarda en producir resultados. Esta métrica puede representar diferentes conceptos dependiendo del contexto. Por ejemplo, en la evaluación del desempeño del CPU, se tienen los ciclos de reloj requeridos para completar un programa, el ancho del pulso o período del reloj del CPU y el tiempo total de ejecución de un programa; en cambio, en la evaluación del rendimiento de la memoria, se tiene el tiempo de acceso a una dirección de memoria.
- **Productividad:** la productividad (su término en inglés es *throughput*) como métrica de rendimiento, es la cantidad de trabajos o tareas completadas por unidad de tiempo. A diferencia del tiempo de ejecución que mide directamente el rendimiento del CPU, la productividad depende de diferentes factores externos y circunstanciales (como acceso a disco, algoritmo de planificación) y mide un rendimiento más global del sistema.
- **Disponibilidad:** mide cuánto tiempo un sistema se mantiene en operación normal del tiempo total requerido.

Existe una cuarta métrica de tipo interna, que se percibe desde dentro del sistema. Es la métrica de **utilización**. Esta última es de vital importancia para entender el sistema y predecir su desempeño en condiciones específicas.

- **Utilización:** es el fragmento de tiempo que un componente del sistema (CPU, disco, memoria, etc.) está activo para su utilización o contribuye al trabajo. La utilización se mide en el rango entre 0 y 1 o como porcentaje. La productividad máxima de un sistema se alcanza cuando el componente más ocupado logra una utilización de 1 ó 100%.

Aunque las métricas permiten contar con un criterio de comparación genérico, muchas veces es difícil definir métricas que sean efectivas 100% o estén libres de errores en los procedimientos o casos de aplicación.

#### 4.1.2. Consideraciones de efectividad-coste

En la evaluación de computadores, sobre todo con fines económicos o financieros, también se incluyen consideraciones de costes. Normalmente interesa el coste de producción o adquisición, la **tasa de retorno de la inversión** y el **coste total de pertenencia**.

La historia muestra proyectos costosos, como Cray, donde se busca a ultranza lograr el mayor poder de cálculo con el mejor rendimiento. En estos casos, el coste no se consideró como una métrica de decisión para la implementación del proyecto. En el otro extremo se tiene a proyectos genéricos como soluciones SOHO, donde el menor precio posible es la meta.



**SOHO soluciones**  
<https://sohosoluciones.com/>

Aunque el rendimiento sí cuenta, se puede renunciar a ciertos beneficios en función de obtener un equipo de bajo coste para un mercado con baja exigencia. Quizás el mayor trabajo para el arquitecto o diseñador de arquitecturas de computación consiste en alcanzar el mayor rendimiento posible al menor coste.

#### 4.1.3. Técnicas de análisis de desempeño de sistemas computacionales

Las técnicas de análisis del desempeño de la ejecución de computadores se pueden clasificar en cuatro grandes bloques:

- Modelado analítico
- Modelado por simulación
- Medición de la ejecución (*benchmarking*)
- Modelado híbrido

Las técnicas analíticas y de simulación requieren de la construcción de un modelo: una representación abstracta del sistema real. Un modelo analítico de la ejecución es una estructura matemática, mientras que un modelo por simulación requiere del soporte de un programa de computador especializado (un simulador). La mayor parte del arte en análisis de la ejecución queda en seleccionar a un buen modelo que integre los aspectos más sobresalientes del sistema, eliminando toda la masa de detalles no pertinentes que puedan ocultar lo esencial del mismo.

La técnica, de medición de la ejecución, no usa modelos, pero en cambio se sustenta en la observación directa y en la recolección de valores del sistema de interés, o un sistema similar. Esta técnica incluye la elección de métricas y sus medidas a emplear para caracterizar el sistema y el uso de *benchmarks* o programas que sirvan como carga de trabajo e instrumento de medición. Ninguna técnica es mejor en todos los casos. Los modelos analíticos y de simulación predominan en escenarios donde no se cuenta con el sistema vivo o un prototipo del mismo. Cuando se quiere determinar el desempeño de un sistema existente, la técnica de medición de la ejecución es lo más indicado.

##### Elección de programas *benchmarks*

Los programas *benchmarks* permiten determinar valores o medidas para evaluar, según algunas métricas, un computador con una carga de trabajo determinada. Para esto se necesita establecer la carga de trabajo idónea. Esta carga de trabajo está compuesta por uno o más programas tipo (*benchmarks*) que permiten establecer una marca de comparación de este computador con otros. La elección de estos programas debe hacerse de forma cuidadosa de acuerdo a la actividad para la que está pensada el computador. Existen cuatro tipos de *benchmarks* según las tendencias que se han venido imponiendo:

- **Aplicaciones o programas reales:** consiste en usar de carga de trabajo la aplicación principal a emplear en el equipo específico o programas tipo predefinidos como: compiladores, procesadores de texto y aplicaciones de diseño asistido por computador (CAD por *Computer Assisted Design*).
- **Núcleos o kernels:** emplea extracciones de secciones de código importante de programas reales usadas para evaluar rendimiento, por ejemplo: Livermore Loop y LinPack.



#### Livermore Loop

<http://www.roylongbottom.org.uk/livermore%20loops%20results.htm>



#### Linpack

<https://www.top500.org/project/linpack/>

- **Benchmarks reducidos:** emplean rutinas pequeñas (10 a 100 líneas) con resultados conocidos. Se introducen y ejecutan fácilmente en computadores para medir su desempeño; por ejemplo: QuickSort, Puzzle.
- **Benchmarks sintéticos:** son análogos a la filosofía de los núcleos. Se crean mezclas de diferentes rutinas de programas reales o inventados por el diseñador del *benchmark*. Determinan frecuencia media de operaciones y acceso a operandos en diferentes escenarios de cómputo. No calculan algo que un usuario pueda utilizar, solo pretenden determinar un perfil medio de ejecución que sirva de referencia para comparar con otras máquinas. Ejemplos de estos *benchmarks* son: WhetStone y Dhrystone.



#### WhetStone

<http://www.roylongbottom.org.uk/whetstone.htm>



#### DhryStone

<http://www.roylongbottom.org.uk/dhystone%20results.htm>

**Queda de tarea para el alumno investigar los programas de *benchmark* más usados para determinar el TOP500.**

#### 4.1.4. Otras métricas de rendimiento populares para arquitecturas computacionales

En el intento de crear una métrica estándar de rendimiento que permita comparar dos computadores cualesquiera, se han propuesto una serie de medidas de rendimiento populares. Este es un esfuerzo enorme porque la meta es ambiciosa y difícil de alcanzar. Y aunque, como resultado de los mismos se han propuesto una serie de métricas simples que se han empleado ampliamente, éstas tienen un contexto de aplicación limitado y son susceptibles a errores de utilización.

- **MIPS (Million Instructions Per Second):** es una métrica de productividad muy difundida que representa la **velocidad o frecuencia de ejecución de instrucciones** y se expresa en millones de instrucciones por segundo. En términos del tiempo de ejecución, la velocidad de ejecución de instrucciones equivale a la cantidad total de instrucciones ejecutadas sobre el tiempo total

de ejecución. Se emplea el múltiplo millones (dividiendo entre 106) para obtener el resultado en un número más legible. Así,

$$\text{MIPS} = N/t * 10^6$$

Donde  $N$  es el número total de instrucciones y  $t$  es el tiempo total de ejecutar esas  $N$  instrucciones.

Como MIPS es una frecuencia, expresa el rendimiento en proporción inversa al tiempo, lo cual es adecuado porque mayores MIPS indican máquinas más rápidas. El problema con el uso de MIPS consiste en que, si se emplea como método comparativo, se debe tener cuidado que las máquinas tengan características similares. Por ejemplo, un programa que emplea emulación de las operaciones de punto flotante puede resultar con más MIPS que una máquina que emplea una Unidad de Punto Flotante (**FPU** por *Floating Point Unit*), aunque el tiempo de ejecución del programa sea menor en la segunda, con lo cual el criterio MIPS conduce a un resultado equivocado. De manera similar, si el repertorio de instrucciones de ambas máquinas difiere, es posible que el resultado también sea equivocado.

- **MIPS relativos:** para atacar el problema de dos máquinas con repertorios de instrucciones muy diferentes, se propuso emplear la métrica de MIPS relativos que se refiere a los MIPS de la máquina que se desea evaluar en comparación con una máquina de referencia. Se mide el tiempo de ejecución de un programa de prueba en la máquina a evaluar y se divide entre el tiempo de ejecución del mismo programa en la máquina de referencia y finalmente se multiplica por los MIPS de la máquina de referencia, lo que permite comparar dos máquinas respecto a una tercera:

$$\text{MIPS}_{\text{Relativos}} = t_{\text{Referencia}} / t_{\text{Evaluada}} * \text{MIPS}_{\text{Referencia}}$$

Donde  $t_{\text{Referencia}}$  es el tiempo de ejecución en la máquina referencia,  $t_{\text{Evaluada}}$  es el tiempo de ejecución en la máquina a ser evaluada y  $\text{MIPS}_{\text{Referencia}}$  representa los MIPS de la máquina referencia.

Un problema de la métrica de MIPS es que evalúa el desempeño solo para un programa específico. De hecho, se han demostrado resultados equívocos con tipos diferentes de programas. Adicionalmente, la dificultad de obtener una máquina de referencia adecuada con la misma carga de los sistemas operativos y compiladores hace casi impráctico este método hoy día.

- **MFLOPS (Million Floating Point Operations Per Second):** para enfrentar las deficiencias de MIPS, respecto a las diferencias en las operaciones de punto flotante, que son extensivamente usados en los ámbitos de computación científica, se propuso emplear los MFLOPS o Millones de Operaciones en Punto Flotante por segundo. Los megaflops se obtienen del cociente entre el total de operaciones en punto flotante de un programa sobre el tiempo de ejecución por 106:

$$\text{MFLOPS} = O_{\text{punto\_flotante}} / t * 10^6$$

Donde  $O_{\text{punto\_flotante}}$  es la cantidad de operaciones punto-flotante y  $t$  el tiempo de ejecutar esas operaciones.

Aunque está restringido a las operaciones en punto flotante, los MFLOPS son más consistentes que los MIPS por cuanto la cantidad de operaciones no cambia de máquina a máquina. La problemática que se presenta, sin embargo, es que los conjuntos de operaciones en punto flotante disponibles de máquina a máquina no son estándares y por tanto algunas operaciones se deben implementar en función de otras. Adicionalmente, existen operaciones en punto flotante que son muy rápidas y otras muy lentas.

- **MFLOPS normalizados:** para la solución a los problemas de MFLOPS se ha propuesto el uso de los MFLOPS normalizados. Este método concede pesos relativos a cada tipo de operación en punto flotante y permite además considerar las operaciones enteras. Al asignar pesos mayores a las operaciones complejas se puede normalizar la diferencia de velocidades permitiendo comparar dos máquinas de manera más equitativa:

$$\text{MFLOPS}_{\text{Normalizados}} = (\sum O_i^* \text{peso\_relativo}_i) / (t * 10^6)$$

Donde  $O_i$  es la cantidad de operaciones de tipo  $i$  y  $\text{peso\_relativo}_i$  es el peso de ese tipo de operación  $i$ .

El problema de los MFLOPS normalizados, consiste en que evalúa el rendimiento para ese programa tipo de prueba (*benchmark*) específico y que, aunque representa una opción para comparar dos computadores, no representa el rendimiento de la máquina.

#### 4.1.5. Comparación de dos arquitecturas en base al rendimiento

Dado que la medida de rendimiento es el tiempo, independientemente de cómo se haya obtenido la métrica que lo caracteriza, se establece que a menor tiempo de ejecución se tiene mayor rendimiento. Es decir, el rendimiento ( $R$ ) de un computador es inversamente proporcional al tiempo de ejecución ( $t$ ) que presenta para una carga definida. Por tanto, se puede definir rendimiento como:

$$R = 1/t$$

Con esta medida ya se puede comparar dos computadores. Sean  $R_A$  el rendimiento de una máquina A y  $R_B$  el de la máquina B, para una misma carga, se puede decir que:

$$R_A > R_B \Leftrightarrow 1/t_A > 1/t_B \Leftrightarrow t_B > t_A$$

Se puede cuantificar la relación entre las dos máquinas, para expresar qué tanto más rinde la máquina A sobre la máquina B con la siguiente relación:

$$n = R_A / R_B$$

Donde  $n$  representa la cantidad de veces que A tiene mayor rendimiento que B. Dependiendo del contexto,  $n$  se conoce como *ganancia* ( $G$ ), si A es un reemplazo de B, o como *aceleración* ( $A$ ) o mejora de rendimiento, si A es una evolución de la misma arquitectura de B.

Por ejemplo, si la máquina A ejecuta un programa en 10s y la máquina B en 15s, ¿qué tanto mayor es el rendimiento de la máquina A sobre el de la máquina B?

$$n = R_A / R_B = (1/t_A) / (1/t_B) = t_B/t_A = 15s/10s = 1,5$$

Es decir, el rendimiento de A es 1,5 veces el de B. Esta relación es adecuada como base de comparación respecto a la unidad, sin embargo, a veces se desea saber en cuánto supera una a la otra, es decir, el incremento porcentual de rendimiento:

$$n\% = ((R_A - R_B) / R_B) * 100\%$$

De donde se deduce que A es 50% más rápida que B. **Queda como tarea para el alumno demostrarlo.**

Una vez definidas las métricas o características básicas que intervienen en la definición del rendimiento, se podrá aplicar esta metodología para comparar los rendimientos de dos computadores mediante su tiempo de ejecución.

La siguiente sección evalúa las métricas que definen el rendimiento relacionado a una aplicación paralela y las diferentes técnicas para obtener las respectivas medidas.

## 4.2. Aspectos de rendimiento de algoritmos paralelos

Al igual que la evaluación del rendimiento de arquitecturas o computadores, la evaluación del desempeño de programas paralelos consiste en obtener medidas que caractericen tal rendimiento. Desde el punto de vista de aplicaciones paralelas, no es sencillo definir qué es rendimiento, es un concepto complejo y polifacético. Por ejemplo, la tarea de los ingenieros de software es diseñar e implementar programas que satisfagan los requerimientos del usuario relacionados con correctitud y rendimiento, mientras que en el ámbito de computación de alto rendimiento se busca reducir el tiempo de ejecución de un programa particular.

La utilización de recursos disponibles y la capacidad de utilizar mayor poder de cómputo para resolver problemas más complejos o de mayor dimensión, son las características más deseables para aplicaciones paralelas. Así, el tiempo de ejecución es la medida tradicionalmente utilizada para evaluar la eficiencia computacional de un programa. Es una medida útil para evaluar el esfuerzo computacional requerido para resolver un problema a través de una solución paralela y además es simple de obtener, aun cuando hay varios factores que influyen en el tiempo de ejecución de una aplicación paralela, como la transmisión de datos entre procesos, el número de procesadores y el almacenamiento de datos en dispositivos.

### 4.2.1. Métricas para la evaluación del rendimiento de algoritmos paralelos

En esta sección se estudiarán las principales métricas que caracterizan el rendimiento de una aplicación paralela.

- **Tiempo total de ejecución**

El modelo de evaluación de desempeño de programas paralelos más usado considera que el **tiempo total de ejecución es el tiempo que transcurre desde el inicio del primer proceso hasta la finalización del último proceso paralelo**. Cada proceso que compone la aplicación paralela mide su tiempo de ejecución en función de tres estados: tiempo de procesamiento o cómputo efectivo ( $T_{Comp}$ ), tiempo de comunicación ( $T_{Comm}$ ) y tiempo ocioso ( $T_{idle}$ ):

$$T = T_{Comp} + T_{Comm} + T_{idle}$$

Si se tienen  $P$  tareas ejecutándose en  $P$  procesadores, el tiempo total de la aplicación paralela se puede aproximar con el siguiente modelo:

$$T = 1/P \left( \sum_{i=1}^P T_{Comp}^i + \sum_{i=1}^P T_{Comm}^i + \sum_{i=1}^P T_{idle}^i \right)$$

- El **tiempo de cómputo** ( $T_{Comp}$ ) corresponde al cómputo que debe ser realizado en la versión paralela y depende de la complejidad y dimensión del problema (tamaño del problema), del número de tareas y procesadores utilizados y de las características de los elementos de procesamiento (hardware, heterogeneidad, no dedicación).
- El **tiempo de comunicación** ( $T_{comm}$ ) depende de la localidad de los procesos y los datos (comunicación inter e intra procesador y uso de canales de comunicación). Generalmente, los procesadores que trabajan sobre un problema paralelo requerirán comunicarse entre sí. Incluye el tiempo empleado en el envío y recepción de mensajes. Dado que la comunicación intra procesador no incurre en el uso de los canales de comunicación, no se toma en cuenta para el cálculo del tiempo de comunicación. El coste de la comunicación inter procesadores, se define como el tiempo necesario para el establecimiento de la conexión (**latencia**) y tiempo de transferencia ( $T_{TR}$ ) de información (dados por el ancho de banda del canal físico). Para enviar un mensaje de  $L$  bits, se requiere un tiempo de:

$$T_{mensaje}(L) = \text{latencia} + L * T_{TR}$$

Donde  $T_{TR}$  es el tiempo de transferencia de un bit.

- El **tiempo ocioso** (**Tidle**) es consecuencia del no determinismo en la ejecución. Los procesos pueden estar ociosos en algunos instantes de tiempo debido a un desbalance en la carga de trabajo asignado a cada proceso, a la ausencia de recursos de cómputo disponibles, a la ausencia de datos sobre los cuales debe operar (necesidad de sincronizaciones) o a la existencia de trozos de código que deben ser ejecutados de forma secuencial por un único procesador. Este tiempo es difícil de determinar pues depende del orden en la que se realizan las operaciones. Lo ideal es reducirlo lo más que se pueda, a través de técnicas de balance de carga o rediseñar el programa para distribuir más adecuadamente los datos.

- **Aceleración (speedup)**

La medición de rendimiento en ambientes paralelos es más compleja por el interés de conocer cuánto más rápido se ejecuta una aplicación en un computador paralelo. Es decir, es importante conocer cuál es el beneficio obtenido cuando se usa paralelismo y cuál es la **aceleración** que resulta por el uso de dicho paralelismo. Así, la **aceleración (A)** es una medida de la mejora de rendimiento de una aplicación al aumentar la cantidad de procesadores (comparado con el rendimiento al utilizar un solo procesador). Puede ser definida como la relación entre el tiempo de ejecución secuencial y el tiempo de ejecución paralelo, de la siguiente manera:

$$A = \frac{\text{Tiempo de Ejecución Secuencial}}{\text{Tiempo de Ejecución Paralelo}}$$

Existen diversas formas de expresar la aceleración que dependen de la forma en que se defina lo qué es el **Tiempo de Ejecución Secuencial y Paralelo**:

- **Aceleración Relativa:** para obtener la **aceleración relativa**, el *Tiempo de Ejecución Secuencial* usado es el tiempo total de ejecución del programa paralelo cuando éste es ejecutado sobre un único procesador del computador paralelo. Por lo tanto, la *aceleración relativa (A<sub>relat</sub>)* de un programa paralelo cuando se resuelve una instancia I de tamaño N usando P procesadores es:

$$A_{\text{relat}}(I,P) = \frac{T(I,1)}{T(I,P)}$$

Donde  $A_{\text{relat}}(I,P)$  significa aceleración relativa de la instancia I de un programa paralelo en P procesadores,  $T(I,1)$  es el tiempo de ejecución de la instancia I en 1 procesador y  $T(I,P)$  es el tiempo de ejecución de la instancia I en P procesadores.

- **Aceleración Real o Absoluta.** Para obtener la **aceleración real**, el *Tiempo de Ejecución Secuencial* usado es el tiempo del mejor programa secuencial para resolver el problema. Por lo tanto, la *aceleración real (A<sub>real</sub>)* de un programa paralelo cuando se resuelve una instancia I de tamaño N usando P procesadores es:

$$A_{\text{real}}(I,P) = \frac{T'}{T(I,P)}$$

Donde  $A_{\text{real}}(I,P)$  significa aceleración real o absoluta de la instancia I de un programa paralelo en P procesadores,  $T'$  es el tiempo de ejecución de la mejor versión secuencial de la instancia I y  $T(I,P)$  es el tiempo de ejecución de la instancia I en P procesadores.

La aceleración relativa es la más frecuentemente utilizada en la práctica para evaluar la mejora de desempeño (considerando el tiempo de ejecución) de programas paralelos. La aceleración real es difícil de calcular, porque no es sencillo conocer el mejor algoritmo serial que resuelve un problema determinado. La aceleración real o absoluta considera el tiempo de un algoritmo secuencial (el mejor existente o conocido) para la comparación, por lo tanto, evalúa la mejora de desempeño al utilizar técnicas de programación paralela. Mientras que la aceleración relativa,

toma en cuenta el tiempo de ejecución de un algoritmo paralelo en un único procesador, por lo que da una medida de cuán paralelizable o escalable resulta el algoritmo paralelo utilizado. Por eso, también se le llama **nivel de paralelización del algoritmo**.

La situación ideal es lograr una aceleración lineal; es decir, al utilizar  $P$  procesadores se obtiene una mejora de factor  $P$ . Sin embargo, la realidad indica que es habitual obtener una aceleración sublineal, lo que indica que utilizar  $P$  procesadores no garantiza una mejora de factor  $P$ . La aceleración sublineal puede ser causada por código que se debe ejecutar secuencialmente, por la necesidad de sincronizaciones o comunicaciones, por los tiempos de intercambio de datos, por los tiempos ociosos, por la existencia de cuellos de botella en el acceso a recursos de hardware, etc. Estos factores pueden incluso producir que el uso de más procesadores sea contraproducente para el rendimiento de la aplicación.

En ciertos casos es posible obtener valores de aceleración superlineal, si se aprovechan ciertas características especiales del problema o del hardware disponible. Una aceleración mayor que  $P$  es posible solo si cada elemento de procesamiento consume menos del tiempo secuencial entre  $P$  ( $T_s/P$ ) resolviendo el problema. Una razón de esto es que la versión paralela realiza menos trabajo que el correspondiente algoritmo secuencial. Puede deberse también a factores relacionados con los recursos que utiliza.

La Figura 24 presenta una gráfica que muestra la aceleración lineal, sublineal y superlineal.

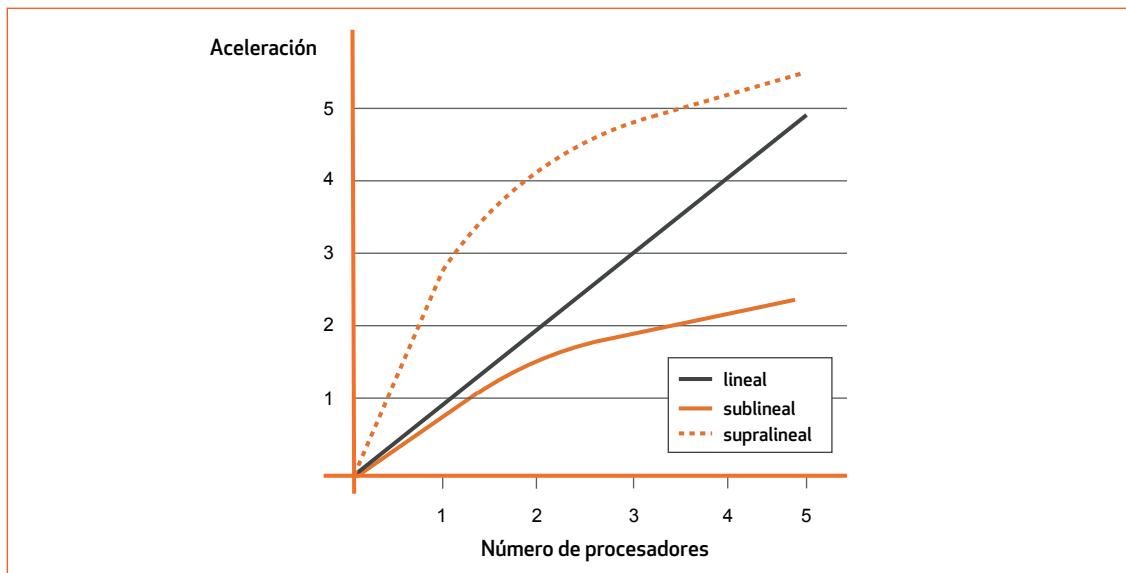


Figura 24. Aceleración lineal, sublineal y supralineal.

Considere el problema de un algoritmo *bubblesort* (ordenamiento burbuja) paralelo. Suponga que el tiempo secuencial es 150s y el tiempo paralelo para una implementación eficiente es de 40s. La aceleración es de  $150/40 = 3.75$ . ¿Es esto un buen logro? ¿Qué pasa si el *quicksort* secuencial (mejor algoritmo secuencial) toma solo 30s? En este caso la aceleración es de solo 0.75.

**Tarea para los alumnos: ¿Qué significa una aceleración igual a 2? Analice el ejemplo anterior.**

- **Eficiencia**

Se puede definir la **eficiencia (E)** como la relación entre la aceleración (**A**) y el número de procesadores (**P**) usados en la ejecución:

$$E = A/P$$

Corresponde a un valor normalizado de la aceleración (entre 0 y 1), respecto a la cantidad de procesadores utilizados. Es una medida relativa que permite la comparación de desempeño en diferentes entornos de computación paralela. Si se obtienen valores de eficiencia cercanos a uno, se identificarán situaciones casi ideales de mejora de rendimiento.

**Ejemplo:** Si el mejor algoritmo secuencial para resolver un problema I requiere 8s para ejecutar en uno de los nodos de un computador paralelo homogéneo y 2s al utilizar 5 procesadores, se tiene:

- Aceleración real:  $A_{Real}(I,5) = 8s/2s = 4$
- Eficiencia, toma en cuenta la cantidad de procesadores utilizados:
  - $E = 4/5 = 0,8$ .
  - Aceleración relativa (o nivel de paralelización), toma en cuenta al mismo algoritmo paralelo para la medición de tiempos. Suponiendo que el algoritmo paralelo toma ventajas de la estructura del problema y ejecuta en un único procesador en 6s, se tendrá:  $A_{Relat}(I,5) = 6s/2s = 3$ .

- **Escalabilidad**

Otro término importante es el de **escalabilidad**. Constituye una de las principales características deseables de los algoritmos paralelos y distribuidos. **La escalabilidad se refiere al estudio del cambio en las medidas de rendimiento de un sistema cuando una o varias características del sistema son variadas.** Se usa ya sea para medir la capacidad de mejorar el desempeño al utilizar recursos de cómputo adicionales (más procesadores) para la ejecución de aplicaciones paralelas o para resolver instancias más complejas de un determinado problema. Un sistema es escalable si mantiene constante la eficiencia al aumentar el número de procesadores y aumentando también el tamaño del problema.

### 4.2.2. Ley de Amdahl

Otro aspecto importante a considerar es que todo programa paralelo tiene una parte secuencial que eventualmente limita la aceleración que se puede alcanzar en una plataforma paralela. En 1967 Amdahl, G. expuso que la parte serial de un programa determina una cota inferior para el tiempo de ejecución, aun cuando se utilicen al máximo técnicas de paralelismo. Observe la Figura 25, el componente pintado en rojo denota la parte del código que no puede ser paralelizado, y que se ejecutará en tiempo TS, por lo tanto, en la versión paralela se ejecutará en ese mismo tiempo TS. La

parte pintada en verde denota la parte del algoritmo que se puede parallelizar y se ejecutará con tiempo  $T_p$  en la versión secuencial y tiempo  $T_p/P$  si se ejecuta en  $P$  procesadores.

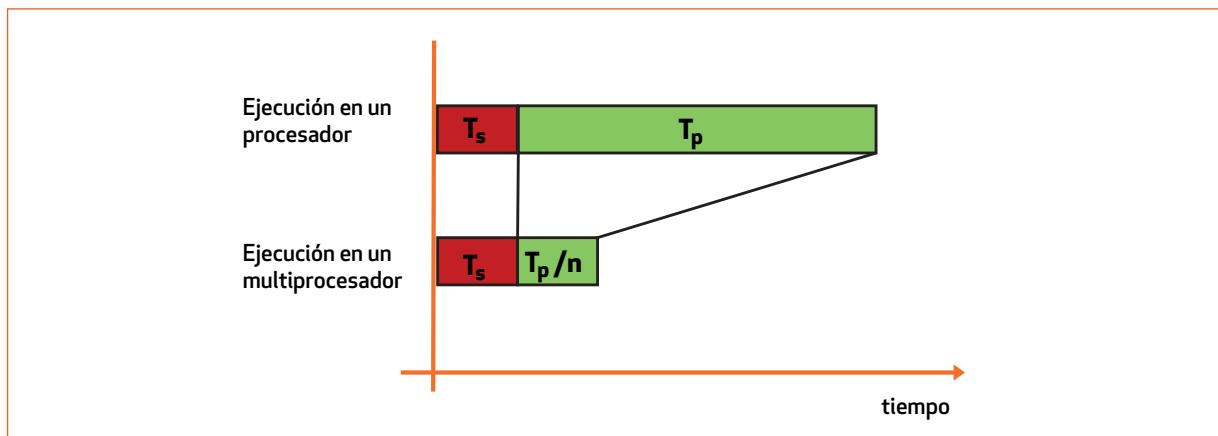


Figura 25. Relación de código secuencial con código paralelizable.

Si  $P$  es la fracción de un programa que es paralelizable, y  $S = 1 - P$  es la parte secuencial restante, entonces la aceleración (**A**) al trabajar con  $N$  procesadores estará acotada por:

$$A \leq 1/(S + P/N)$$

El valor asintótico para  $A$  es  $1/S$  cuando  $N \rightarrow \infty$ .

**Ejemplo 1:** Si solo el 50% de un programa es paralelizable,  $S = P = 1/2$ , se tiene que:

$$A \leq 1 / (1/2, + 1/2, /N) = 2/(1 + 1/N)$$

Que es  $\leq 2$  (valor asintótico:  $A = 1/S = 1/(1/2) = 2$ ), independientemente de  $N$  (cantidad de procesadores utilizados).

**Ejemplo 2:** Si un programa tiene un 5% de componente secuencial entonces la aceleración máxima que se puede alcanzar es de 20. **Queda como tarea para el alumno demostrarlo.**

La ley de Amdahl tiene varias implicaciones:

- Para una carga dada, la aceleración máxima tiene una cota superior de  $1/S$  (valor asintótico). Al aumentar  $S$ , la aceleración decrecerá proporcionalmente;
- Para alcanzar buenas aceleraciones, es importante reducir  $S$ .

En la Figura 26 se puede observar el comportamiento de la aceleración en función de la fracción de código que no puede ser paralelizado. Se observa que aun usando una gran cantidad de procesadores (1024), con un porcentaje bajo de código serial se obtienen aceleraciones muy por debajo del número de procesadores usado. Por ejemplo, si la aplicación paralela tiene solo un 1% de código serial, la aceleración máxima que puede lograrse es de 91, es decir que con 1024 procesadores dicha aplicación puede ejecutar 91 veces más rápido que la aplicación serial.

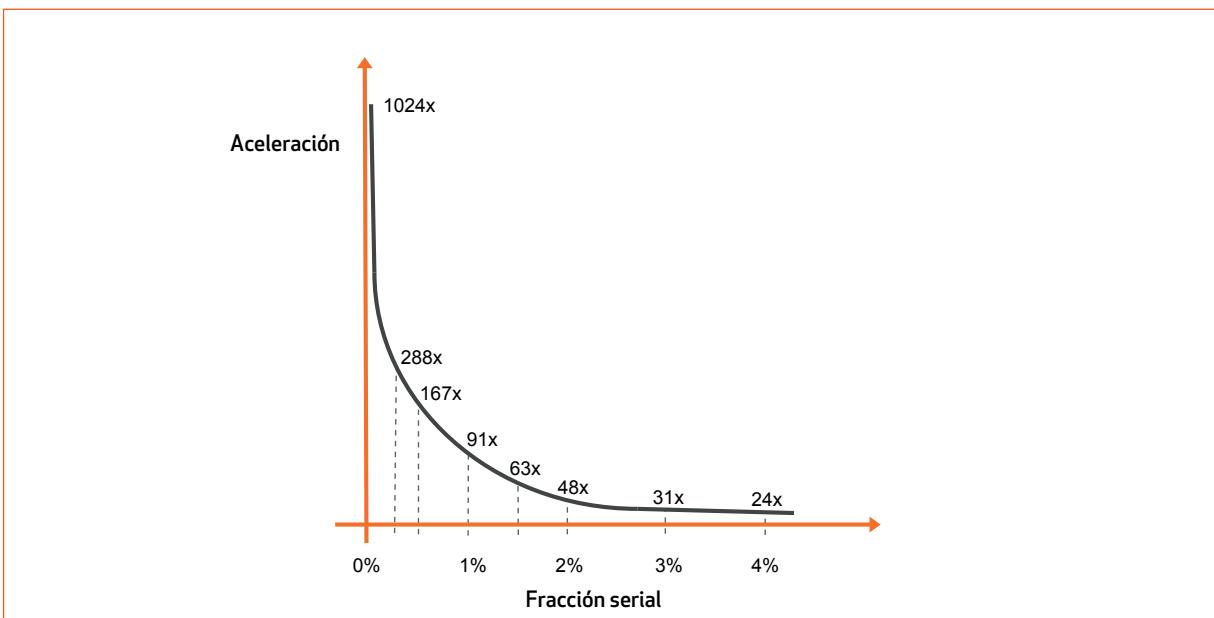


Figura 26. Aceleración según la ley de Amdahl. Adaptado de <http://albert-jan.yzelman.net/education/parco14/amdahl.html>.

En los tiempos iniciales de la computación paralela, se creyó que el efecto capturado por la ley de Amdahl limitaría la utilidad de la computación paralela a un grupo pequeño de aplicaciones. Sin embargo, ciertas experiencias prácticas han demostrado que esta forma de ver los programas con una parte estrictamente secuencial tiene poca relevancia en los problemas reales.

Para entender esto, veamos qué sucede en un problema no relacionado con la computación. Suponga que 999 obreros de 1000 que están trabajando en la construcción de una autopista están ociosos cuando uno de los obreros completa una tarea secuencial. Este hecho se puede ver como una falla en el manejo del proyecto y no como un atributo del problema en sí. Si colocar el cemento en un punto de la autopista es un cuello de botella, se puede argumentar que la construcción de la autopista se puede hacer en varios puntos distintos al mismo tiempo. Haciendo esto se puede introducir alguna inefficiencia pues a lo mejor los camiones tienen que viajar distancias más largas para llegar al punto de trabajo, pero esto llevará a realizar el proyecto en menos tiempo. Algo similar se puede aplicar en el caso de las soluciones paralelas de un problema en específico.

La ley de Amdahl puede ser relevante cuando un programa secuencial es parallelizado de forma incremental. De esta forma, con una herramienta de perfilamiento, se evalúa el programa secuencial para identificar los componentes con mayor demanda computacional. Estos componentes son adaptados para una ejecución paralela uno por vez hasta que se logre un rendimiento aceptable.

### 4.2.3. Ley de Gustafson-Barsis

A finales de la década de los 80, Gustafson, J. (1988) y su equipo de investigación en Sandia Nacional Laboratorios se encontraban realizando investigación relacionada con el uso de Procesamiento Masivamente Paralelo y notaron que existía un alto nivel de escepticismo por el uso de estos sistemas paralelos debido a la ley de Amdahl. Según esta ley, aún en problemas con una fracción secuencial de trabajo muy pequeña, la aceleración máxima alcanzable para un número infinito de procesadores era de solamente  $1/S$ . Con los resultados de las investigaciones que realizaron en un sistema con 1024

procesadores, demostraron que las suposiciones de la ley de Amdahl no son apropiadas para el caso de paralelismo masivo.

El problema está en que generalmente no se toma un problema de tamaño fijo y se ejecuta variando el número de procesadores. Esto se hace solo para fines académicos y de investigación. Generalmente, el tamaño del problema crece al aumentar el número de procesadores. Cuando se dispone de un procesador más poderoso, el problema se expande para hacer uso de ese poder computacional. Esta visión más optimista de la ley de Amdahl propone que el tamaño (o la complejidad) del problema crece con el número de procesadores a usar y la parte secuencial del programa se mantiene constante.

Así, asumiendo que el tamaño (o la complejidad) del problema ( $z$ ) crece con el número de procesadores a usar ( $N$ ), la parte secuencial del programa ( $S$ ) se mantiene constante y la parte paralela del programa ( $P$ ) crece según el tamaño del problema ( $z^*P, z^{2*}P$ , etc.), entonces las fracciones de tiempo de ejecución de las partes secuencial ( $T_s$ ) y paralela ( $T_p$ ) del programa resultan:

$$T_s = S / (S + z^*P)$$

$$T_p = z^*P / (S + z^*P)$$

Del razonamiento anterior es posible deducir que la aceleración para  $N$  procesadores ( $A(N)$ ):

$$A(N) \leq N * (1 + z^*(P/S)) / (N + z^*(P/S))$$

Se concluye que dado un número de recursos de cómputo  $N$ , existirán problemas de complejidad  $z$  cuyas partes serial y paralela cumplan que  $z^*(P/S) \leq N$  y entonces la cota teórica corresponde a la aceleración lineal  $A(N) \leq N$ .

Por lo tanto, la ley de Gustafson rescata el procesamiento paralelo que no era favorecido por la ley de Amdahl: *la razón para utilizar un número mayor de procesador debe ser resolver problemas más grandes o más complejos, y no para resolver más rápido un problema de tamaño fijo.*

#### 4.2.4. Factores que afectan el rendimiento de los programas paralelos

**Si se usan dos procesadores, ¿cuáles son las razones por las cuales el programa no se ejecuta dos veces más rápido? La respuesta es directa: porque existe una serie de factores que degradan el rendimiento del programa.** Algunos de esos factores están relacionados con tiempos: tiempo de comunicación y tiempo ocioso. Como se ha visto anteriormente, un programa secuencial no presenta estos tiempos, sin embargo, un algoritmo paralelo por lo general está diseñado para que las tareas cooperen para alcanzar un objetivo común. Esta cooperación, normalmente se expresa en transferencias de datos y sincronizaciones. En la Figura 27 se muestra el perfil de ejecución de un programa paralelo hipotético sobre 8 procesadores. El perfilador muestra tiempo ocupado en la ejecución propiamente dicha, en comunicación y ocioso. Note que todos los procesos no finalizan al mismo tiempo, que algunos ejecutan más cómputo que otros y que los tiempos de comunicación y ocio también son diferentes entre los procesos.

La **granularidad** consiste en la cantidad de cómputo con relación a la comunicación. En **granularidad fina**, las tareas individuales son relativamente pequeñas en término de tiempo de ejecución.

La comunicación entre los procesadores es frecuente. En cambio, en **granularidad gruesa**, la comunicación entre los procesadores es poco frecuente y se realiza después de largos períodos de ejecución. Sin embargo, la granularidad gruesa puede reducir el nivel de paralelismo y limitar la escalabilidad de la aplicación paralela.

Un buen diseño de una solución paralela debe lograr un compromiso entre el grado de paralelismo obtenido y el sobretiempo (*overhead*) introducido por las necesidades de sincronización y comunicación. Esto se logra con técnicas de aglomeración y una estrategia de mapeo eficiente. Un buen nivel de granularidad debe asegurar la menor cantidad de comunicación y ocio posible.

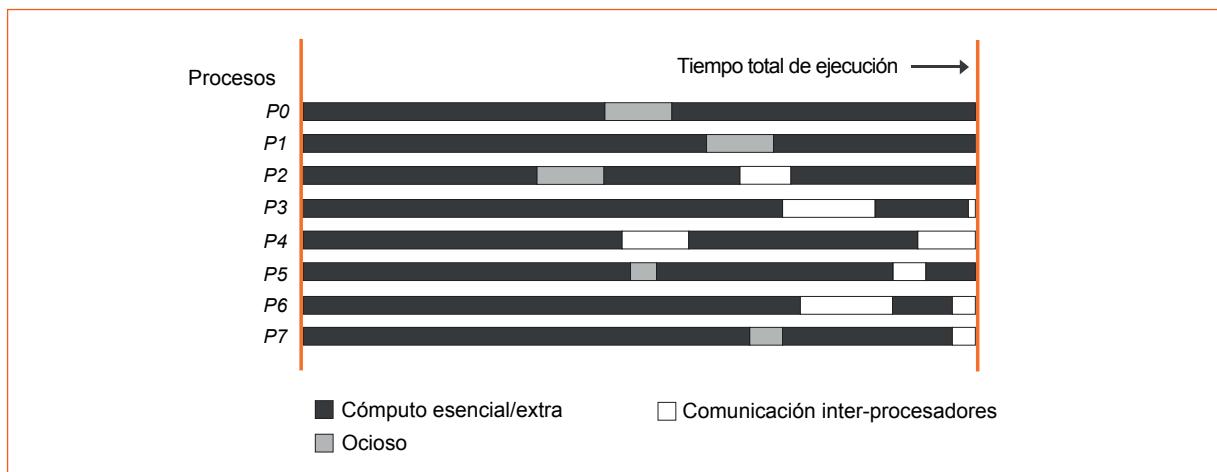


Figura 27. Perfil de un programa paralelo hipotético.

Otro factor determinante en el rendimiento de un programa paralelo está relacionado con la escalabilidad. Un sistema es escalable si es posible mantener la eficiencia (**E**) constante incrementando simultáneamente el tamaño del problema (**z**) y el número de procesadores (**N**). Es necesario entonces evaluar la escalabilidad de un programa paralelo para determinar si es necesario rediseñar la solución. Una forma de evaluar la escalabilidad es mediante pruebas de desempeño o analizando los modelos que representan la eficiencia y la aceleración.

### ¿Cómo se mide el tiempo de ejecución de un programa paralelo?

En el caso de un programa secuencial, el tiempo de ejecución es el tiempo que transcurre desde que se inicia la ejecución hasta que finaliza. En el caso de un programa paralelo, es el tiempo transcurrido desde que comienza su ejecución hasta el momento en que el último procesador finaliza su ejecución. Se ilustrará cómo medir el tiempo de ejecución de programas paralelos con varios ejemplos.

**Ejemplo 1.** Considere el problema de sumar  $N$  números usando  $N$  elementos de procesamiento. Si  $N$  es potencia de dos, se puede realizar esta operación en  $\log(N)$  pasos propagando sumas parciales a lo largo de un árbol lógico de procesadores (vea Figura 20 del Capítulo 3). En la Figura 28 se muestran los pasos de comunicación necesarios para el caso de  $N=16$ . Como ya hemos visto, el tiempo de este algoritmo paralelo es de orden  $O(\log(N))$  ( $T_p = O(\log(N))$ ) y se conoce que el algoritmo secuencial es de orden  $O(N)$  ( $T_s = O(N)$ ). Así la aceleración real ( $A_{Real} = T_s/T_p$ ), dada que se conoce el tiempo del algoritmo secuencial, es de orden  $O(N/\log(N))$  ( $A_{Real} = O(N/\log(N))$ ).

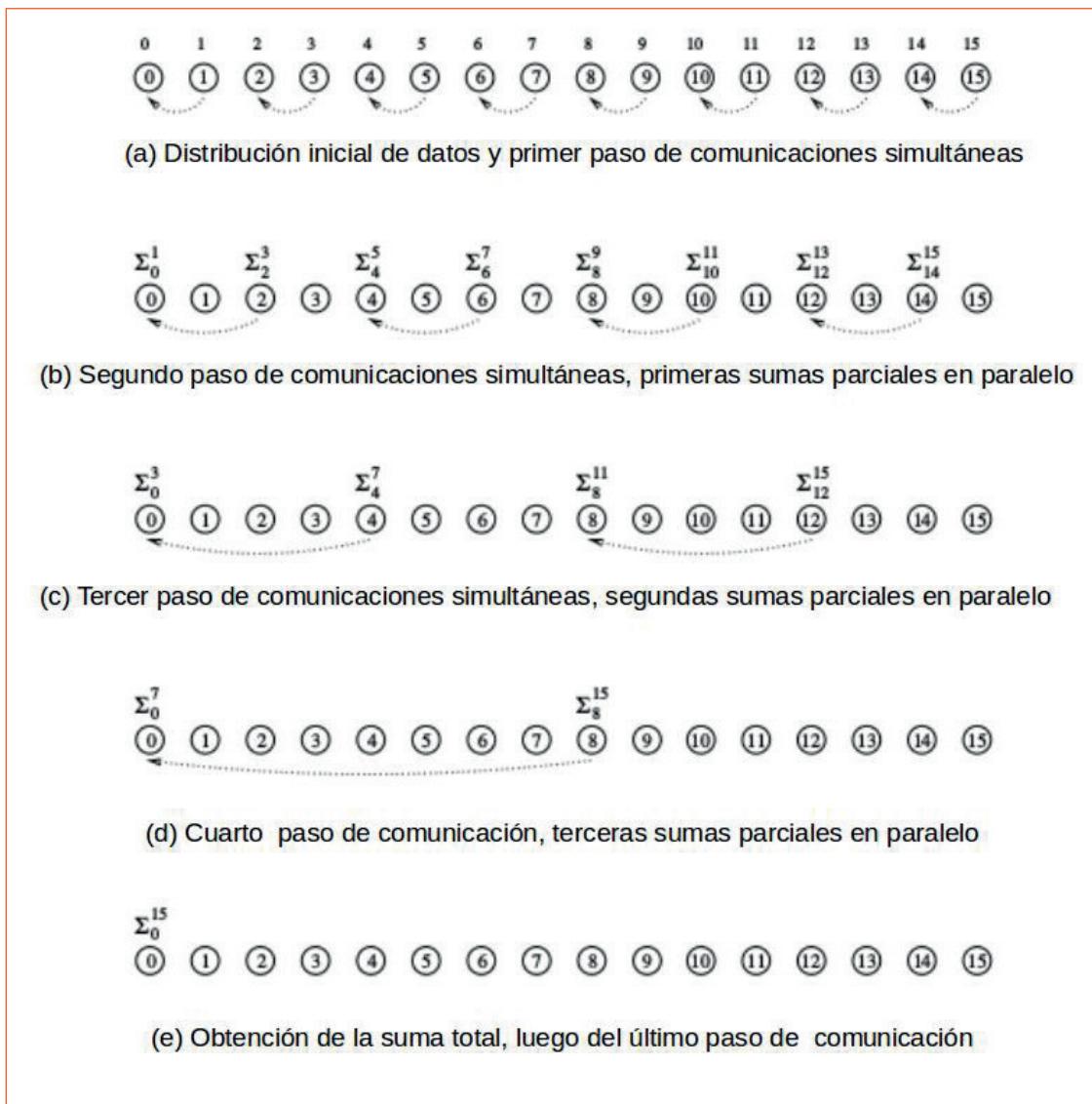


Figura 28. Pasos de comunicación de sumar  $N$  números en  $N$  procesadores ( $N=16$ ).

**Ejemplo 2: Suma de  $N$  números en un hipercubo de  $P$  procesos.** A cada proceso se le asigna  $N/P$  números. Cada proceso calcula la suma parcial de sus  $N/P$  números. Por lo tanto, efectúan  $N/P - 1$  operaciones de suma. Cada proceso tiene  $\log(P)$  vecinos con quienes se comunica e intercambia los resultados locales. Después de intercambiar resultados, actualiza la suma. La Figura 29 ilustra un hipercubo de  $P=8$  procesadores y muestra las comunicaciones simultáneas en los pasos (1), (2) y (3). Los tiempos de cómputo ( $T_{\text{comp}}$ ), comunicación ( $T_{\text{comm}}$ ) y paralelo ( $T_p$ ), son los siguientes:

$$T_{\text{comp}} = t_c * (N/P - 1) + t_c * \log(P)$$

Donde  $t_c$  es el tiempo medio de cómputo por punto.

$$T_{\text{comm}} = (I + TTR) * \log(P)$$

$$TP = T_{\text{comp}} + T_{\text{comm}} = t_c * (N/P - 1) + (t_c + I + T_{\text{TR}}) * \log(P)$$

Queda como tarea para el alumno interpretar esos modelos y calcular la aceleración y la eficiencia.

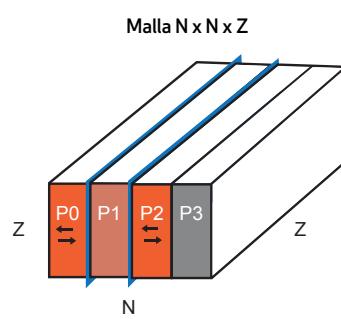


Figura 29. Suma en un hipercubo de  $P=8$  procesadores.

**Ejemplo 3: Eficiencia del algoritmo de diferencias finitas.** Dado el problema de diferencias finitas en una malla de  $N \times N \times Z$  elementos (puntos), se descompone la malla en la dimensión horizontal como se muestra en la Figura 30.

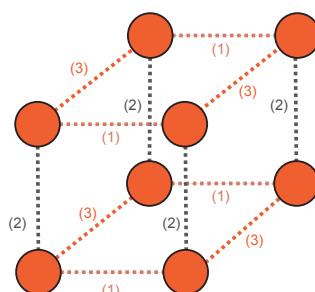


Figura 30. División de la malla de  $N \times N \times Z$  puntos entre  $P=4$  procesadores. Recuperada de [http://www.laminfo.com/blog/archivos/\\_Teoria\\_2\\_Programacion\\_Concurrente.pdf](http://www.laminfo.com/blog/archivos/_Teoria_2_Programacion_Concurrente.pdf)

Cada tarea trabaja sobre una submalla con  $N \times N/P \times Z$  puntos. Cada tarea lleva a cabo el mismo cómputo sobre cada punto asignado en cada paso de iteración del algoritmo. El tiempo de cómputo ( $T_{\text{comp}}$ ) para cada paso es:

$$T_{\text{comp}} = t_c * N^2 * Z/P$$

Donde  $t_c$  es el tiempo medio de cómputo por punto. Cada tarea intercambia  $2*N*Z$  puntos con dos vecinos, por lo tanto, cada tarea envía dos mensajes con  $2*N*Z$  puntos por mensaje. Así, si la latencia se representa con  $l$  y el tiempo de transferencia como  $T_{\text{TR}}$ , el tiempo de comunicación ( $T_{\text{comm}}$ ) se puede expresar como:

$$T_{\text{comm}} = 2(l + T_{\text{TR}} * 2 * N * Z)$$

El tiempo total de la aplicación paralela ( $T_p$ ) viene dado por:

$$T_p = T_{\text{comp}} + T_{\text{comm}} = t_c * N^2 * Z/P + 2*l + T_{\text{TR}} * 4 * N * Z$$

La aceleración relativa ( $A_{relat} = T_1 / T_p$ ), ya que se puede calcular el tiempo en 1 procesador, viene dado por:

$$A_{relat} = T_1 / T_p = (t_c * N^2 * Z) / (t_c * N^2 * Z/P + 2*I + T_{TR} * 4 * N * Z)$$

y la eficiencia ( $E = A/P$ ) por:

$$E = A / P = (t_c * N^2 * Z) / (t_c * N^2 * Z + 2*I*P + T_{TR} * 4 * N * Z * P)$$

Estos modelos de rendimiento pueden ser usados para explorar y refinar el diseño de un algoritmo paralelo. Con ellos se puede realizar un análisis cualitativo de rendimiento. Por ejemplo, a partir de las ecuaciones de  $T_p$  y  $E$  se puede emitir las siguientes observaciones del algoritmo de diferencias finitas:

- La eficiencia baja al incrementar el número de procesadores ( $P$ ) y el coste de comunicación ( $I$  y  $T_{TR}$ ).
- La eficiencia sube al incrementar el tamaño del problema ( $N$  y  $Z$ ) y el tiempo de cómputo de cada punto ( $t_c$ ).
- TP baja al incrementar  $P$  pero está acotado inferiormente por el coste de intercambiar trozos de la matriz.

Estas observaciones proveen visiones interesantes de las características del algoritmo. También es necesario obtener resultados cuantitativos que requieren que se sustituyan ciertos parámetros en el modelo de rendimiento por valores específicos de la arquitectura en la que se ejecutará. Estos valores se obtienen por medio de estudios empíricos. Una vez incluidos estos valores al modelo, éste puede ser usado para contestar interrogantes como:

- ¿Cumple el algoritmo con restricciones de diseño (tiempo de ejecución, requerimientos de memoria) en la arquitectura paralela a usar?
- ¿Cómo se adapta el algoritmo? ¿Se adapta a aumentos en el tamaño del problema y en el número de procesadores?
- ¿Cómo se comporta este algoritmo con respecto a otros algoritmos para el mismo problema?

La evaluación empírica se puede apoyar en herramientas ofrecidas por el sistema de comunicación usado para la implementación de la solución paralela, ya sea a nivel del sistema operativo, de librerías o del propio lenguaje de programación.



## 5. Herramientas de programación paralela

### 5.1. Modelos de comunicación en programación paralela

Los procesos cooperantes pueden seguir distintos modelos de comunicación que dependen de la arquitectura donde se ejecuten. Los modelos de comunicación más usados son:

- **Por memoria compartida:** se trata de mecanismos usados en sistemas centralizados (e.g., pipes, semáforos, mutex, señales, hilos) o **sistemas paralelos** con memoria compartida (e.g., hilos, OpenMP).
- **Cliente-servidor:** usado en redes y sistemas distribuidos (e.g., sockets, RPC, RMI).
- **Pase de mensajes:** se usa en sistemas centralizados, distribuidos y paralelos (e.g., MPI, CUDA).
- **Comunicación basada en streams:** se usa en sistemas distribuidos en escenarios en los que el tiempo y orden de los mensajes es vital (transferencia de audio y vídeo).
- **Comunicación en grupo:** se usa en sistemas centralizados, distribuidos y paralelos, para enviar mensajes entre múltiples procesos (e.g., *multicast, broadcast*).

Note que, en el mundo de la computación de alto rendimiento, los modelos de comunicación que se pueden usar son modelos de memoria compartida, modelos de pase de mensaje y modelos de comunicación en grupo.

El enfoque de comunicación de memoria compartida en el marco de la programación paralela tiene algunas semejanzas con la programación concurrente:

- Se basa en el uso de múltiples hilos.
- Es el subsistema de comunicación, librería o sistema operativo quien realiza la distribución de los datos y cómputo y controla la comunicación.
- Requiere de mecanismos de control de sincronizaciones y exclusión mutua.
- Funciona solo en arquitecturas de una única memoria.

El enfoque de comunicación de pase de mensajes tiene las siguientes características:

- Funciona tanto en arquitecturas paralelas de memoria compartida como distribuida.
- Se puede usar combinado con el modelo de memoria compartida.
- Es soportado por llamadas al sistema operativo o a través de librerías especiales.
- Las primitivas principales de comunicación son:
  - **send(destino, mensaje, long\_mensaje):** para enviar un **mensaje** de tamaño **long\_mensaje** a un proceso **destino**;
  - **receive(fuente, mensaje, &long\_mensaje):** para recibir un **mensaje** de tamaño **long\_mensaje** de un proceso **fuente**.

La comunicación en grupo consiste en usar primitivas de comunicación que permitan enviar/recibir mensajes a/desde múltiples procesos.

Independientemente del modelo de comunicación usado, la transferencia de datos y mensajes se realiza a través de un enlace físico, tal como memoria, buses, cable coaxial, microondas, fibra óptica, etc. Sin embargo, a nivel de los procesos no se accede directamente a tales medios físicos; por el contrario, se definen enlaces lógicos que el sistema operativo traduce en accesos a los enlaces físicos correspondientes. Así, los enlaces lógicos son provistos a nivel de *software* y sirven para administrar la comunicación. Dependiendo del tipo de enlace lógico usado, hay diferentes aspectos importantes que los particularizan: ¿Cómo se establecen? ¿A cuántos procesos está asociado? ¿Cuántos enlaces son posibles entre cada par de procesos? ¿Cuál es la capacidad del enlace y tamaño del mensaje soportado? ¿Es bidireccional o unidireccional? Dependiendo de cómo son definidos los enlaces lógicos, se tienen diferentes implementaciones de los *send* y *receive*, a saber: comunicación directa o indirecta, comunicación simétrica o asimétrica, comunicación síncrona o asíncrona y comunicación transitoria o persistente. Veamos las combinaciones más posibles.

### 5.1.1. Comunicación directa simétrica y directa asimétrica

En la comunicación **directa**, se nombran explícitamente a los procesos destino y fuente, es decir no hay entidades intermediarias durante la comunicación. Los procesos envíador y receptor deben conocer la identidad del proceso destino o fuente, respectivamente, y colocarlo explícitamente en las primitivas de comunicación. El enlace lógico se establece automáticamente y se asocia solo con dos procesos. Entre cada par de procesos existe solo un enlace. Para este tipo de comunicación, los enlaces pueden ser unidireccionales o bidireccionales.

La comunicación **directa** puede ser **simétrica** si se establece solo entre dos procesos, es decir comunicación uno-a-uno. La comunicación **directa asimétrica** permite comunicar más de 2 procesos al estilo uno-a-muchos. La Figura 31 ilustra varios ejemplos de comunicación **directa asimétrica**.

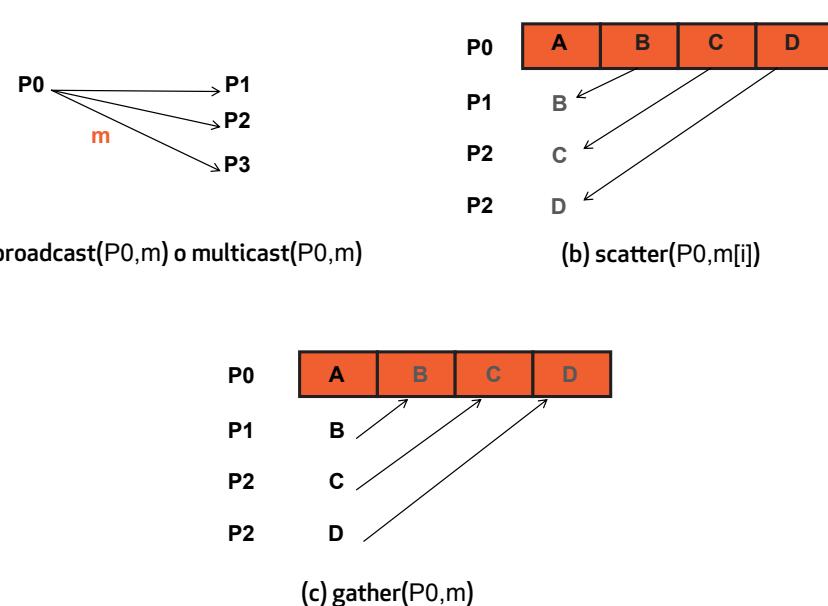


Figura 31. Ejemplos de comunicación directa asimétrica.

### 5.1.2. Comunicación indirecta

En la comunicación indirecta, un enlace lógico se establece entre un par de procesos solo si comparten un buzón (*mailbox*), un puerto (*socket*) o un *pipe*. Por lo tanto, un enlace se puede asociar con más de un par de procesos y entre cada par de procesos puede haber más de un enlace lógico. Los enlaces pueden ser unidireccionales o bidireccionales. La Figura 32 muestra varios ejemplos de comunicación indirecta.

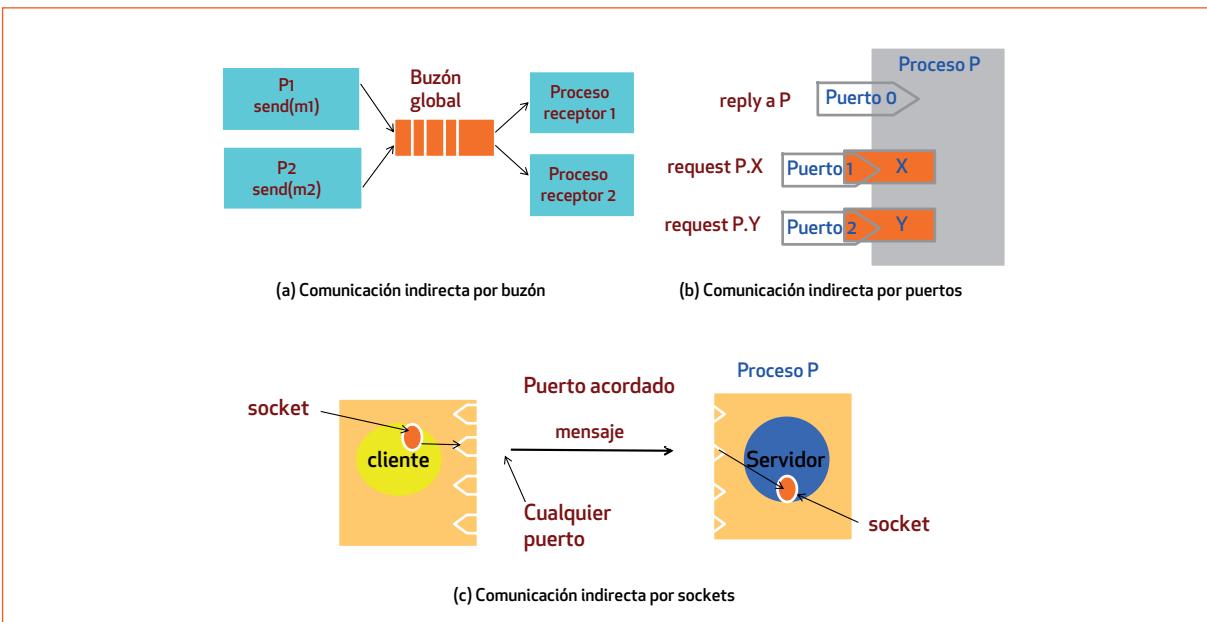


Figura 32. Ejemplos de comunicación indirecta.

### 5.1.3. Comunicación síncrona y asíncrona

La implementación de las primitivas *send* y *receive* pueden ser implementadas como **síncronas** (bloqueantes) o **asíncronas** (no bloqueantes):

- Un **send con bloqueo** espera o se suspende hasta que el receptor haya recibido.
- Un **send sin bloqueo** continúa aunque el receptor no haya recibido, retorna el control al llamador inmediatamente antes que el mensaje sea enviado.
- Un **receive con bloqueo** espera hasta que el enviador haya enviado.
- Un **receive sin bloqueo** continúa aunque el enviador no haya enviado, le indica al kernel un buffer donde se dejará el mensaje y la llamada retorna inmediatamente.

La Figura 33 esquematiza un *send* síncrono y un *send* asíncrono.

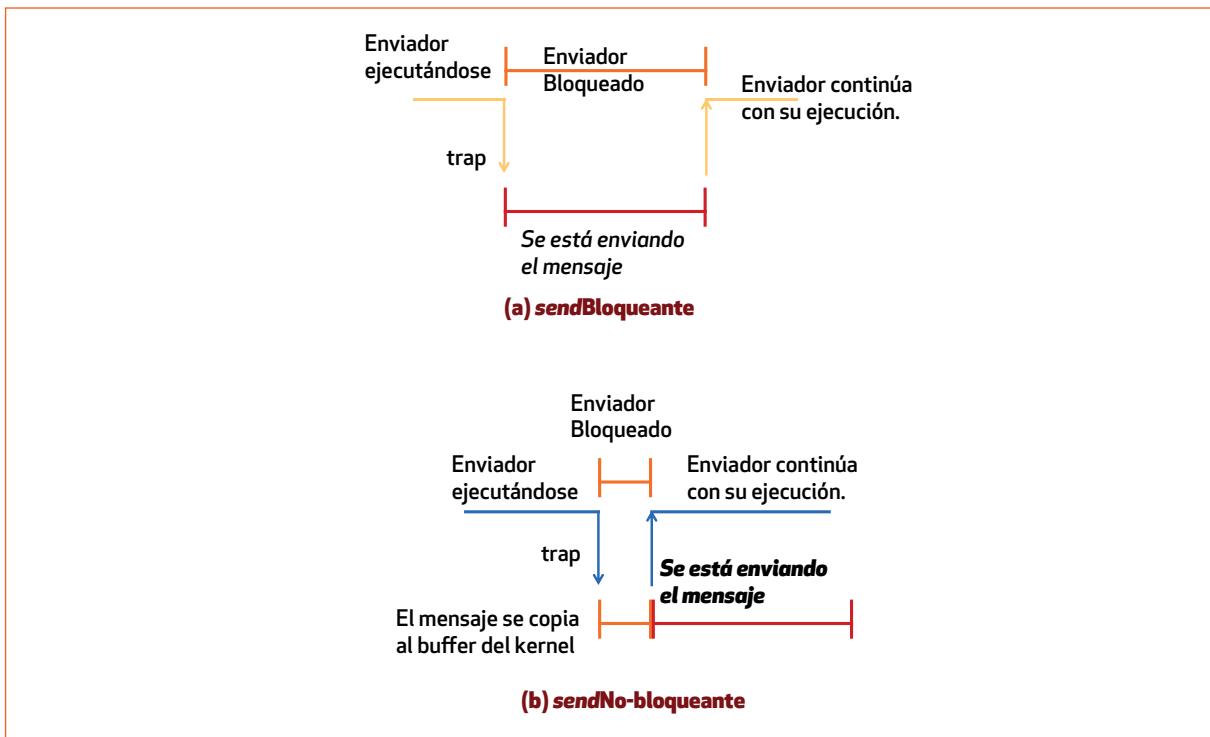


Figura 33. (a) send síncrono; (b) send asíncrono.

#### 5.1.4. Comunicación transitoria y persistente

En la comunicación transitoria los mensajes son almacenados por el sistema de comunicaciones solo mientras los procesos envíador y receptor se están ejecutando. Si alguno de los dos procesos falla, el mensaje es eliminado. Típicamente son servicios de comunicación a nivel de transporte. La comunicación **directa** permite comunicación **transitoria** en la cual los mensajes son mantenidos por el subsistema de comunicación solo mientras se ejecutan envíador y receptor.

Por su lado, en la comunicación **persistent**, los mensajes son almacenados por el sistema de comunicaciones hasta que el receptor los tome. No requiere la ejecución simultánea de envíador y receptor. La comunicación **indirecta** permite comunicación **persistent**: el mensaje se puede mantener en el subsistema de comunicaciones aun cuando ninguno de los procesos enviadores/receptores estén activos.

Existen diferentes herramientas, librerías y lenguajes que soportan la programación paralela y proveen medios para la evaluación del rendimiento. Veremos a continuación varias de las más populares en el mundo de la computación de alto rendimiento.

#### 5.2. Programación paralela con memoria distribuida

La computación de alto rendimiento ha sido dominada desde sus inicios por el paradigma paralelo de memoria distribuida, dado que permite distribuir el trabajo entre varias unidades de cómputo con su propio espacio de memoria privada. Por lo tanto, el modelo de comunicación usado en este paradigma requiere de mecanismos de **pase de mensaje** entre los procesos. Este modelo se

popularizó aún más con las arquitecturas multinúcleos, multiprocesadores y el uso de *cluster* de computación. En respuesta a este avance, en 1992 un grupo de investigadores de la academia y la industria decidió diseñar un sistema de pase de mensajes estandarizado y portable, llamado **MPI** por *Message Passing Interface* (Gropp, W. et al., 1994). Desde entonces, MPI ha sido el estándar *de-facto* para la comunicación entre procesos en sistemas de memoria distribuida y así el sistema de programación dominante en el mundo de la computación de alto rendimiento.

### 5.2.1. Librerías MPI

El estándar MPI ha sido la base de muchos proyectos de desarrollo. Algunos proyectos de implementación de MPI se han enfocado en la portabilidad, ofreciendo implementaciones capaces de ejecutarse en diferentes plataformas, redes y sistemas operativos; así como sobre diferentes lenguajes de programación (Fortran 77, HPF, F90, C y Java). Otros proyectos, atendiendo a los requerimientos de clientes específicos, proveen implementaciones de MPI para redes específicas como Myrinet (Seitz, C., 1994), InfiniBand (InfiniBand Trade Association et al., 2000) y Quadrics (Petrini, F. et al., 2001). A continuación, se estudiarán las características principales del estándar MPI.

Dada la popularidad del estándar MPI, existen muchas implementaciones disponibles libres y de dominio público, soportadas en los lenguajes de programación más populares actualmente (Fortran, C, Java, Phyton). Las más usadas son MPI-LAM, MPICH y OpenMPI. En esta asignatura usaremos OpenMPI. Independientemente de la implementación del estándar MPI que se use, las características principales de los APIs no varían:



- Proveen soporte de librerías de comunicación con facilidades para desarrollar programas paralelos.
- Permite modelar programas paralelos con el enfoque maestro-esclavos.
- Permite modelar topologías lógicas.
- Soporta el modelo SPMD (*Single Process Multiple Data*).
- Incluye facilidades de comunicación uno-a-uno (punto-a-punto), comunicación en grupo (colectiva), definición de grupo de procesos, definición de topologías de procesos, interfaz de perfilamiento y operaciones de memoria compartida con soporte de hilos.

### 5.2.2. OpenMPI: una implementación de MPI

#### Funciones básicas de MPI

Con el siguiente ejemplo, se explicarán las funciones básicas de MPI, así como la forma de compilar y ejecutar programas paralelos, suponiendo que se usa OpenMPI como la implementación del estándar MPI:

```
/* Programa mpi1.c */  
#include "mpi.h"  
#define N 10  
int main(int argc, char **argv) {  
    int myId, numprocess, i;  
    //Inicializaciones  
    MPI_Init(&argc,&argv);  
    MPI_COMM_SIZE(MPI_COMM_WORLD,&numprocess);  
    MPI_COMM_RANK(MPI_COMM_WORLD,&myId);  
    if (myId==0) {  
        printf("Soy el proceso %d",myId);  
        for (i=0;i<N;i++)  
            MPI_Send(&i,1,MPI_INT,1,0, MPI_COMM_WORLD);  
    } else {  
        printf("Soy el proceso %d",myId);  
        for (i=0;i<N;i++)  
            MPI_Receive(&i,1,MPI_INT,0,0, MPI_COMM_WORLD, &status);  
    }  
    MPI_Finalize();  
}
```

## Compilación y ejecución

Un programa que usa la librería MPI, debe compilarse usando el *script* disponible para ello. En OpenMPI es:

**mpicc**: se usa para compilar y enlazar programas en C que hagan uso de MPI. Puede utilizarse con las mismas opciones que el compilador de C/C++ usual, por ejemplo:

```
$ mpicc -c mpi1.c  
$ mpicc -o mpi1 mpi1.o
```

o simplemente:

```
$ mpicc mpi1.c -o mpi1
```

Para la ejecución también se usa un *script* especial. En todas las implementaciones de MPI se denomina **mpirun**. La forma más usual de ejecutar un programa MPI es con la orden:

```
$ mpirun -np 2 mpi1
```

El argumento **-np** sirve para indicar cuántos procesos ejecutarán el programa **mpi1**. En este caso, OpenMPI lanzará dos procesos **mpi1**. Como no se indica nada más en la orden de ejecución, OpenMPI iniciará dichos procesos en la máquina local. Para tener un mayor control sobre qué proceso se lanza en cada máquina, se puede especificar en la orden de ejecución, el conjunto de máquinas que se usarán. En un *cluster* de computadoras, la forma más simple es especificando una lista de máquinas en un archivo de texto que podrá llamarse **machines.txt**. La asignación de procesos se hará por orden: el proceso número 0 a la primera máquina de la lista, el proceso 1 a la segunda máquina y así sucesivamente. Si hay más procesos que máquinas, se continúa la asignación empezando otra vez por el principio de la lista. La llamada a **mpirun** ejecuta una copia del programa en cada uno de los procesadores especificados.

La ejecución de un programa OpenMPI asumiendo un archivo local de máquinas sigue el formato:

```
mpirun [-nolocal] -machinefile <maquinas> -np <num_procesos> <programa>
```

Por ejemplo, si el archivo de máquinas **machines.txt**, contiene las siguientes líneas:

```
159.90.9.187  
159.90.9.188  
159.90.9.189  
159.90.9.190
```

y se ejecuta el programa **mpi1** sobre 2 procesos con la sentencia:

```
$ mpirun -machinefile machines4 -np 4 mpi1
```

Esto significa que la primera réplica del programa ejemplo (el proceso 0) se ejecutará sobre la máquina 159.90.9.187 y la segunda (el proceso 1) se lanzará en la máquina

```
159.90.9.188.
```

Si se incluye el modificador **-nolocal**, no se asignará ninguna réplica a la máquina local (desde la que se llama a **mpirun**), aunque esté en la lista de máquinas.

Todas las funciones de MPI empiezan por **MPI\_** y deben incluirse en el programa a través de la directiva: **#include "mpi.h"**

Todas las funciones de MPI del programa deberán estar comprendidas entre las líneas **MPI\_Init(&argc, &argv);** (inicializa el proceso en su respectivo procesador) y **MPI\_Finalize();** (finaliza el proceso).

En MPI, los procesos que se comunican deben pertenecer a un grupo o **comunicador**. Por defecto se puede utilizar **MPI\_COMM\_WORLD**, en cuyo caso el grupo de procesos es el conjunto de procesos

lanzados conjuntamente para resolver un problema. Para conocer cuántos procesos se iniciaron con el programa paralelo se usa la función:

```
MPI_COMM_SIZE(MPI_COMM_WORLD,&numprocess);
```

En el momento de la ejecución de cada réplica, **mpirun** asigna a cada proceso un identificador diferente (rango de los procesos) comenzando desde 0 y en forma consecutiva. Así, los diferentes procesos podrán ejecutar diferentes sentencias de acuerdo a su identificador o rango. Esto se logra con las sentencias:

```
if (my_rank != 0) {  
    ...  
} else { ... }
```

Esto se corresponde con el modelo SPMD. Cada proceso conoce su rango con la instrucción:

```
MPI_Comm_rank(MPI_COMM_WORLD, &myId);
```

### Comunicación uno-a-uno bloqueante

El pase de mensajes entre procesos se logra mediante las funciones bloqueantes (síncronas):

```
int MPI_Send (void *buffer, int count, MPI_Datatype type, int destino, int tag,  
MPI_Comm comunicador )  
  
int MPI_Recv (void *buffer, int count, MPI_Datatype type, int origen, int tag,  
MPI_Comm comunicador , MPI_Status *estado)
```

Los primeros argumentos de estas funciones son inmediatos:

- Un puntero a los datos (**void \*buffer**) a enviar o al *buffer* donde recibir (**&i** en el programa **mpi1.c**, tanto para *send* como *receive*).
- La longitud de los datos (**int count**) a enviar (**MPI\_Send**) y la longitud máxima de los datos a recibir (**MPI\_Recv**), en **mpi1.c** es **1** para *send* y *receive*.
- El tipo de los datos (**MPI\_Datatype type**), en el programa **mpi1.c** es **MPI\_INT**. MPI proporciona una serie de tipos predefinidos que comienzan todos por la cadena **MPI\_**. La Figura 34 muestra la tabla de tipos básicos de MPI y el tipo correspondiente en C.
- El identificador o rango del proceso al que se le envían los datos (**int destino**); (en **MPI\_Send** es **1** en el programa **mpi1.c**) y del que se esperan los datos **int fuente**) (en **MPI\_Recv** es **0** en el programa **mpi1.c**); es decir que en el programa **mpi1.c**, el proceso 0, envía al proceso 1 y en

contraparte, el proceso 1 recibe del 0. En el caso de **MPI\_Recv**, se puede especificar la constante **MPI\_ANY\_SOURCE**, que indica que puede recibir de cualquier proceso.

- El argumento **int tag** consiste en una etiqueta entera que se añade al mensaje que se pasa entre los procesos. Se pone una marca al mensaje que puede ser usada a voluntad del programador (por ejemplo, para diferenciar mensajes de distintos tipos). Hay un comodín llamado **MPI\_ANY\_TAG** para **MPI\_Recv**. En **mpi1.c**, el tag es **0** para *send* y *receive*.
- El siguiente argumento es lo que se denomina el comunicador (**MPI\_Comm comunicador**). Permite crear grupos de procesos que intercambian información. El programa **mpi1.c** usa el comunicador predefinido **MPI\_COMM\_WORLD**, que incluye a todos los procesos que están en ejecución.
- El último argumento de **MPI\_Recv** (**MPI\_Status \*estado**) es un puntero a una estructura que contiene información sobre el mensaje. Por ejemplo, en caso de que se use **MPI\_ANY\_TAG** en una llamada a **MPI\_Recv**, se puede identificar el proceso que ha enviado el mensaje examinando **status -> MPI\_SOURCE**.

Tipo de dato en MPI	Correspondiente Tipo en C
<b>MPI_CHAR</b>	<code>signed char</code>
<b>MPI_SHORT</b>	<code>signed short int</code>
<b>MPI_INT</b>	<code>signed int</code>
<b>MPI_LONG</b>	<code>signed long int</code>
<b>MPI_UNSIGNED_CHAR</b>	<code>unsigned char</code>
<b>MPI_UNSIGNED_SHORT</b>	<code>unsigned short int</code>
<b>MPI_UNSIGNED</b>	<code>unsigned int</code>
<b>MPI_UNSIGNED_LONG</b>	<code>unsigned long int</code>
<b>MPI_FLOAT</b>	<code>float</code>
<b>MPI_DOUBLE</b>	<code>double</code>
<b>MPI_LONG_DOUBLE</b>	<code>long double</code>
<b>MPI_BYTE</b>	
<b>MPI_PACKED</b>	

Figura 34. Tipos de datos básicos de MPI y su correspondiente tipo en lenguaje C.

## Comunicación colectiva

Las funciones colectivas en MPI, son invocadas por todos los procesos en un comunicador. Dos de las funciones de comunicación en grupo más usadas en MPI son:

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,  
MPI_Comm comm )
```

Distribuye datos de un proceso (el proceso maestro con ID = 0) a todos los demás procesos que pertenecen al mismo comunicador.

```
MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,  
MPI_Op op, int root, MPI_Comm comm)
```

Combina datos desde todos los procesos que pertenecen a un comunicador y retorna el resultado final a un solo proceso.

El siguiente ejemplo ilustra el uso de estas dos operaciones.

```
/* Programa mpi2.c */  
#include "mpi.h"  
#include <math.h>  
int main(int argc, char **argv) {  
    int myId, numprocess, n, i, rc;  
    double mypi, pi, h, sum, x, a;  
    //Inicializaciones  
    MPI_Init(&argc,&argv);  
    MPI_COMM_SIZE(MPI_COMM_WORLD,&numprocess);  
    MPI_COMM_RANK(MPI_COMM_WORLD,&myId);  
    while (1) {  
        if (myId==0) {  
            printf("Introduzca el nro. de intervalos");  
            scanf("%d",&n);  
        }  
        MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);  
        if (n==0)  
            break;  
        else {  
            for (i=myId+1;i <= n;i+=numprocess) {  
                realizar calculos de h y sum  
            }  
            mypi=h*sum;  
            MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
```

```

if (myid==0)
    printf("pi es aprox. %.16f",pi);
} //else
}//while
MPI_Finalize();
}

```

La instrucción **MPI\_Bcast(&n,1,MPI\_INT,0,MPI\_COMM\_WORLD)**; del ejemplo **mpi2.c**, indica que el proceso **0** enviará el valor de n (**&n**), que es un **1 MPI\_INT**, a todos los demás procesos que pertenecen al comunicador por defecto **MPI\_COMM\_WORLD**.

Mientras que la instrucción **MPI\_Reduce(&mypi,&pi,1,MPI\_DOUBLE,MPI\_SUM,0,MPI\_COMM\_WORLD)**; indica que el proceso **0** recibirá en la variable **&pi** la suma (**MPI\_SUM**) de todos los valores parciales (**&mypi**), de todos los demás procesos que pertenecen al comunicador por defecto **MPI\_COMM\_WORLD**.

Las operaciones *reduce* definidas en MPI incluyen:

- **MPI\_MAX**: retorna el elemento máximo.
- **MPI\_MIN**: retorna el elemento mínimo.
- **MPI\_SUM**: retorna la suma de los elementos.
- **MPI\_PROD**: retorna la multiplicación de los elementos.
- **MPI\_LAND**: retorna el *and* lógico aplicado a todos los elementos.
- **MPI\_LOR**: retorna el *or* lógico aplicado a todos los elementos.
- **MPI\_BAND**: retorna el *and* a nivel de bits (*bitwise and*) aplicado a todos los elementos.
- **MPI\_BOR**: retorna el *or* a nivel de bits (*bitwise or*) aplicado a todos los elementos.

Otras operaciones colectivas ofrecidas por MPI son:

- **MPI\_Barrier( )**: bloquea los procesos hasta que todos lo invocan.
- **MPI\_Gather( )**: recibe valores de un grupo de procesos.
- **MPI\_Scatter( )**: distribuye un *buffer* en partes a un grupo de procesos.
- **MPI\_Alltoall( )**: envía datos de todos los procesos a todos.
- **MPI\_Reduce\_scatter( )**: combina valores de todos los procesos y distribuye.
- **MPI\_Scan( )**: reducción prefija (0,...,i-1 a i).

## Send y Receive no-bloqueantes (asíncronos)

Para evitar problemas de interbloqueo o poder solapar comunicación con cómputo, MPI ofrece primitivas send y receive no-bloqueantes (asíncronas):

- **MPI\_Isend(buf, count, datatype, dest, tag, comm, request):** envío no-bloqueante.
- **MPI\_Irecv(buf, count, datatype, source, tag, comm, request):** recepción no-bloqueante. El parámetro **request** se usa para saber si la operación de envío correspondiente ya se realizó.
- **MPI\_Wait( ):** espera hasta que se complete la operación de envío correspondiente.
- **MPI\_Test( ):** devuelve un *flag* diciendo si la operación de envío correspondiente se ha completado.

## Medida de rendimiento

Para medir el tiempo de ejecución de una aplicación, MPI ofrece varias alternativas: usando rutina de librería llamada **MPI\_WTIME** o usando la librería **MPE**.

**MPI\_WTIME** retorna la hora de ese momento. La librería **MPE** es una herramienta de perfilamiento que provee un amplio rango de funcionalidades para medir el rendimiento de programas MPI. Incluye funcionalidades para recolectar información sobre la ejecución de los programas y almacenarla en archivos *logs* tanto para realizar visualizaciones *post-mortum* como animación en tiempo real, a través de un sistema gráfico de ventanas.

El uso de **MPI\_WTIME** se muestra en el siguiente ejemplo. **Queda de tarea para el alumno investigar sobre MPE y verificar si se puede usar con OpenMPI.**

```
/* Programa mpi2T.c */
#include "mpi.h"
#include <math.h>
int main(int argc, char **argv) {
    int myld, numprocess, n, i, rc;
    double mypi, pi, h, sum, x, a;
    double t1, t2;
    //Inicializaciones
    MPI_Init(&argc,&argv);
    MPI_COMM_SIZE(MPI_COMM_WORLD,&numprocess);
    MPI_COMM_RANK(MPI_COMM_WORLD,&myld);
    t1= MPI_Wtime();
    while (1) {
        if (myld==0) {
            printf("Introduzca el nro. de intervalos");
            scanf("%d",&n);
        }
    }
}
```

```

MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
if (n==0)
    break;
else {
    for (i=myId+1;i <= n;i+=numprocess) {
        realizar calculos de h y sum
    }
    mypi=h*sum;
    MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
    if (myid==0)
        printf("pi es aprox. %.16f",pi);
    } //else
} //while
t2= MPI_Wtime();
printf("El tiempo total fue: %.16f",t2-t1);
MPI_Finalize();
}

```

## 5.3. Programación paralela con memoria compartida

Aun cuando el paradigma de programación paralela con memoria distribuida es el más usado en el ámbito de la computación de alto rendimiento, existen herramientas que permiten la programación paralela en arquitecturas de memoria compartida, por ejemplo en arquitecturas multinúcleos. Tal es el caso de OpenMP (*Open Multi-Parallelism*) que se describirá a continuación.

### 5.3.1. OpenMP

OpenMP es una especificación, un API, para implementaciones portables de paralelismo en FORTRAN y C/C++, a través del cual se puede expresar paralelismo de memoria compartida, basado en múltiples hilos.



**OpenMP**

<http://openmp.org/>

Las características principales de OpenMP son:

- **Combina código serial y paralelo en un solo código fuente, en base al modelo llamado *fork-join*:** a partir de un único hilo de ejecución se crean  $N$  hilos paralelos (*fork* o división). Estos hilos terminan su ejecución conjuntamente en un punto posterior llamado punto de reencuentro o *join*. En ese momento, la ejecución del programa continúa de nuevo con un solo hilo hasta un nuevo punto de *fork* y así sucesivamente. Este modelo es mostrado en la Figura 35.

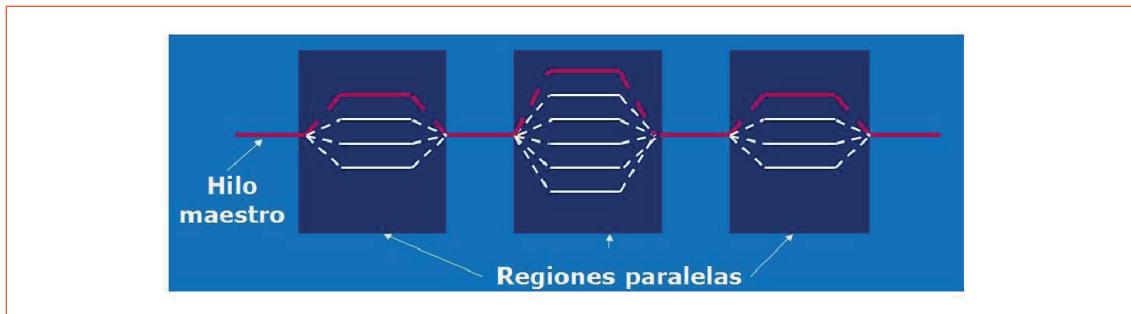


Figura 35. Modelo fork-join de OpenMP.

- **Maneja el paralelismo basado en hilos, pero de manera transparente al programador.** Es decir, que el manejo de hilos se hace de manera implícita, el programador no tiene que preocuparse de ese aspecto, ni de la creación y eliminación de los hilos, ni de la sincronización entre ellos.
- **Soporta el modelo de paralelismo de datos y paralelismo Incremental.**
- **El código es portable**, ya que OpenMP encapsula las llamadas a rutinas de manejo de hilos específicas de cada plataforma.
- **No usa el pase de mensajes**, lo que minimiza la aparición de errores asociados a las transmisiones.
- **Es altamente escalable**, el rendimiento suele mejorar cuando se ejecuta sobre un número mayor de procesadores.
- Por la forma como se diseñó, **el código paralelo se mantiene igual al secuencial**, lo que permite ejecuciones tanto en uno como en muchos procesadores sin necesidad de cambiar el programa ni recompilar.

Sin embargo, OpenMP presenta ciertas limitaciones:

- **No tiene un manejo de errores eficiente**, lo que dificulta el proceso de depuración (*debugging*).
- **No posee a la fecha un mecanismo de control sobre los detalles de implementación de los hilos.** Por un lado, esto simplifica la semántica, pero impide hacer ajustes específicos de desempeño.
- **No tiene soporte para operaciones de bajo nivel para el manejo de memoria compartida.**
- **El tiempo de arranque de los hilos es alto comparado con la ejecución de un hilo secuencial**, por lo que en ocasiones (si el tiempo de ejecución del código es muy bajo) la implementación OpenMP resulta más ineficiente.
- **No está diseñado para trabajar con múltiples máquinas en un ambiente de memoria distribuida**, dado que su implementación está basada en el manejo de hilos.

Los tres principales componentes del API de OpenMP son:

- **Directivas de compilador:** una directiva es una línea especial de código fuente que tiene significado solo para el compilador; una directiva indica al compilador que realice algo. Las directivas de OpenMP son:
  - En Fortran: !\$OMP
  - En C/C++: #pragma omp

Esto significa que las directivas de OpenMP son ignoradas si el código es compilado como un programa secuencial regular en Fortran/C/C++.

- **Funciones de librería:** son las rutinas que pueden ser invocadas desde el programa para que realicen una función específica.
- **Variables de entorno:** son variables de ambiente que deben definirse antes de la ejecución del programa y que los hilos podrán acceder. Una de las variables de entorno más importante es OMP\_NUM\_THREADS que indica el número de hilos que se crearán al iniciarse una sección paralela (un punto *fork*). Un ejemplo de cómo inicializarla es con la siguiente orden en el *shell* de Linux:

**\$ export OMP\_NUM\_THREADS=8.**

A continuación, se describen las principales directivas de OpenMP.

## Definiendo paralelismo con OpenMP

Observe el siguiente programa:

```
/* Programa e1.c */
#include "stdio.h"
#include "stdlib.h"
#include "omp.h"
int main()
{
#pragma omp parallel
  printf("Hola mundo\n");
  exit (0);
}
```

La directiva **#include "omp.h"** de C, indica la inclusión de la librería OpenMP. Sin esta directiva, las directivas de OpenMP no serán reconocidas por el compilador. La directiva **#pragma omp parallel**, indica el comienzo de una sección paralela del código (*fork*).

Se establece la variable de ambiente:

```
$ export OMP_NUM_THREADS=8.
```

Se compila el programa:

```
$ gcc -fopenmp e1.c -o e1
```

En general la compilación tiene la forma:

```
gcc -fopenmp {source.c} opciones -o ejecutable
```

Luego se ejecuta:

```
$ ./e1
```

En este caso se iniciarán 8 hilos de acuerdo a la variable de ambiente **OMP\_NUM\_THREADS** definida.

También se puede especificar el número de hilos desde el programa como muestra el siguiente ejemplo:

```
/* Programa e2.c */  
#include "stdio.h"  
#include "stdlib.h"  
#include "omp.h"  
int main()  
{  
    int cantidad_hilos=6;  
    omp_set_num_threads(cantidad_hilos);  
    #pragma omp parallel  
    printf("Hola mundo\n");  
    exit (0);  
}
```

La llamada a la función de librería **omp\_set\_num\_threads(cantidad\_hilos)**; indica el número de hilos que se crearán en el punto *fork* definido por **#pragma omp parallel**. En este caso se ignora la variable de ambiente **OMP\_NUM\_THREADS**. Al ejecutar este programa, seis hilos dirán “Hola mundo”.

Otra manera de indicar el número de hilos, ignorando la variable de ambiente, es con la directiva:

```
#omp parallel num_threads (N)
```

## Manejo de variables compartidas y privadas con OpenMP

Las variables que hayan sido declaradas *antes* del inicio de la sección paralela serán *compartidas* entre todos los hilos de ejecución. Note el siguiente programa:

```
/* Programa e3.c */
#include "stdio.h"
#include "stdlib.h"
#include "omp.h"
int main() {
    int comp=0; //Variable global a todos los hilos
    # pragma omp parallel
    {
        int priv=0; //Variable privada en cada hilo
        priv++;
        comp++;
        printf("Hola mundo! Valor priv = %d; valor compartido = %d\n",priv, comp);
    }
    exit (0);
}
```

La variable **comp** es compartida entre los hilos durante la sección paralela, mientras que la variable **priv** es privada para cada hilo. Así, invariablemente la impresión de los hilos dará como resultado 1 para todas las variables **priv**, mientras que el resultado de las variables **comp** podría ser imprevisible. **Queda como tarea para el alumno responder a qué se debe este comportamiento.**

Con OpenMP se pueden declarar de manera explícita como privadas a compartidas las variables que se acceden en las regiones paralelas:

```
#pragma omp parallel private(x0,y0) {
...
x0 = xarray[mi_pos];
y0 = yarray[mi_pos];
f[mi_pos] = foo1(x0,y0);
...
}
```

En este ejemplo x0 y y0 son variables privadas a cada hilo, tomadas de los arreglos compartidos xarray y yarray, que son usados para calcular alguna variable que luego se almacena en el arreglo compartido f. También es posible especificar que variables son compartidas:

```
#pragma omp parallel private(x0,y0) shared(xarray,yarray,f)
...
x0 = xarray[mi_pos];
```

```
y0 = yarray[mi_pos];  
f[mi_pos] = foo1(x0,y0);  
...  
}
```

## Paralelización de ciclo con OpenMP

En las regiones paralelas, todos los hilos ejecutan el mismo código, sin embargo, OpenMP provee directivas que indican que el trabajo se va a dividir entre los hilos, de manera que los datos se reparten entre los hilos, en lugar de replicar. Esto es el paradigma de paralelismo de datos, que **es adecuado principalmente en aquellos problemas donde se desea aplicar la misma función o conjunto de instrucciones a diferentes partes (disjuntas) de un universo de datos**. Este tipo de problema es el más común cuando se trata de ejecutar instrucciones sobre elementos de arreglos o sobre listas numeradas del tipo:

```
for (i=inicio;i<fin;i++)  
    x[i] = función(a[i])
```

Es responsabilidad del programador asegurar que las iteraciones de un ciclo paralelo sean independientes para que tenga sentido la paralelización.

Dado que el número de elementos a operar en un ciclo es conocido y que normalmente estas operaciones se implementan con la instrucción **for**, los diseñadores de OpenMP implementaron una directiva especial para las construcciones **for (DO en Fortran) paralelas** de los lenguajes de programación. De esta forma el programador no tiene que cambiar la semántica de sus programas, ya que OpenMP se encargará de dividir los índices y asignarlos a los distintos hilos. En el siguiente ejemplo:

```
#define max 300  
double vec[max];  
#pragma omp parallel for  
for (i =0; i <max; ++i) {  
    vec[ i ] = generate(i) ;  
}
```

La directiva **#pragma omp parallel for**, implica que el ciclo **for** se paraleliza y automáticamente las iteraciones se dividen equitativamente entre los hilos.

La cláusula **if** en la directiva **parallel for**, permite decidir si la paralelización se realiza de acuerdo a una condición. En el siguiente ejemplo, el cálculo paralelo de **pi** se hace únicamente si el número de iteraciones es mayor a 5000, ya que para valores menores se considera que el trabajo de crear y gestionar los hilos es mayor que la ganancia de tiempo.

```
#pragma omp parallel for if ( n > 5000 )
for ( i= 0 ; i< n ; i++ ) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

## Secciones críticas

Observe el siguiente ejemplo de suma sobre una variable compartida (**sum**):

```
float producto(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for (int i=0; i<N; i++)
        { sum += a[i] * b[i]; }
    return sum;
}
```

Este código funciona correctamente en el caso secuencial, sin embargo, al ejecutarse de forma paralela, es posible que frecuentemente los valores resultantes sean erróneos. ¿De dónde surge el error? Al analizar el código de forma exhaustiva, se observa que en la línea: **sum += a[i] \* b[i];** no hay garantía que mientras uno de los hilos actualice la variable **sum**, que es compartida, otro no esté alterándola. De hecho, si *N* es grande y hay muchos hilos, la probabilidad de que tales interferencias ocurran es muy alta. Para evitar este tipo de casos, una idea correcta es hacer que en el momento de la actualización, **un solo hilo pueda acceder a la variable sum.**

OpenMP provee una cláusula que permite el acceso a bloques de código a uno solo de los hilos. Este bloque de código se denomina **sección crítica** y puede ser accedida solamente por uno de los hilos en cualquier momento, aunque otras partes del código pueden seguirse ejecutando de forma concurrente. El siguiente ejemplo ilustra la implementación correcta del código anterior definiendo una sección crítica:

```
float producto(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for (int i=0; i<N; i++) {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

Ahora solo un hilo a la vez podrá acceder a sum, evitando condiciones de carrera. Otro ejemplo:

```
#pragma omp parallel
for (int i =0;i<max;++i) {
    x = f ( i );
    #pragma omp critical
    g(x);
}
```

En este código, las llamadas a *f()* se realizan en paralelo, pero solamente un hilo puede entrar a la vez en *g()*.

### Operaciones de reducción

OpenMP permite tener variables privadas que hagan acumulaciones parciales y, al final de la sección paralela, sumar todos los valores parciales en una simple variable totalizadora, mediante la directiva: **reduction(oprador:lista-de-variables)**.

Con esta directiva, se indica al compilador que una variable (o un conjunto de ellas) se usará de forma privada en cada uno de los hilos y será solamente al final de los cálculos que los resultados parciales obtenidos en los hilos se agruparán en un solo resultado global. Esta agrupación puede ser una suma (como en el caso del producto interno) o también otros operadores aritméticos y lógicos. Observe el siguiente código:

```
float producto(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for (int i=0; i<N; i++)
        { sum += a[i] * b[i];}
    return sum;
}
```

Dentro del bloque de trabajo en paralelo se crea una copia privada de **sum** y se inicializa de acuerdo con el elemento neutro de cada operación (0 para el caso de la suma). Estas copias son actualizadas localmente por los hilos según el cálculo que se realice; en este caso cada hilo sumará el subvector que le correspondió de acuerdo a la división de la directiva **parallel for**. Al final del bloque paralelo, las copias locales se combinan de acuerdo al operador (**+:sum**, en el ejemplo) y se actualiza la variable compartida original (**sum**).

### Sincronización con maestro

En el ejemplo siguiente, la directiva **master** marca un bloque que solamente se ejecuta en el hilo maestro:

```
#pragma omp parallel {
```

```
f(); // En todos los hilos
#pragma omp master {
    g(); // Solamente en maestro
    h(); // Solamente en maestro
}
i(); // En todos los hilos
}
```

## Balance de carga

Cuando el problema se desconoce, el tiempo que tomará la ejecución de las tareas y la distribución de dichos tiempos puede generar que unos hilos tomen cargas más pesadas (lentas) que otros. Una forma de contrarrestar este problema es haciendo agendas de trabajo (*scheduling*) diferentes de acuerdo con los tiempos de ejecución. Para ofrecer balance de carga, OpenMP ofrece la directiva:

**schedule (<tipo>[, <tamaño>] )**

Que puede ser:

- **schedule(static)** o **schedule(static, n)**: planifica bloques de iteraciones equitativos (o de tamaño n) para cada hilo.
- **schedule(dynamic)** o **schedule(dynamic, K)**: cada hilo toma un bloque de 1 o K iteraciones, respectivamente, de una cola hasta que se han procesado todas.
- **schedule(guided)** o **schedule(guided, K)**: cada hilo toma un bloque de iteraciones hasta que se han procesado todas. Se comienza con un tamaño de bloque grande y se va reduciendo exponencialmente hasta llegar a un tamaño K.
- **schedule(runtime)**: se usa lo indicado en **OMP\_SCHEDULE** o por la rutina de librería en tiempo de ejecución.

¿Cuándo usar cada una de estas opciones de planificación?

- **STATIC**: cuando todas las tareas a ejecutar son de complejidad similar. En este caso, no se pierde tiempo en la planificación y la ejecución será eficiente.
- **DYNAMIC**: cuando las tareas tienen tiempos impredecibles, cuando la complejidad en tiempo es desconocida y además es altamente variable.
- **GUIDED**: es útil cuando las cargas de tiempo son variables, pero no demasiado, y se quiere minimizar el coste de planificación. Particularmente, cuando N es demasiado grande (>106).

En el siguiente ejemplo, cada hilo probará 8 números (8 iteraciones) y tomará siempre muestras de tamaño 8 hasta que se termine la ejecución. Sin embargo, todos los hilos tomarán la misma cantidad de muestras.

```
#pragma omp parallel for schedule (static, 8)
for( int i = start; i <= end; i += 2 ) {
    if ( TestForPrime(i) )
        gPrimesFound++;
}
```

Si se cambia la cláusula por **dynamic**, los hilos irán tomando grupos de 8 iteraciones. En la medida que cada hilo vaya terminando, va tomando otros 8 valores para probar. Es posible que al final de la ejecución, un hilo haya ejecutado muchas más iteraciones que otros, sin embargo, los tiempos serán relativamente uniformes entre todos:

```
#pragma omp parallel for schedule (dynamic, 8)
for ( int i = start; i <= end; i += 2 ) {
    if ( TestForPrime(i) )
        gPrimesFound++;
}
```

## Medida de rendimiento

Para medir el tiempo de ejecución de una aplicación, OpenMP ofrece una rutina de librería llamada **omp\_get\_wtime()** que retorna la hora de ese momento y se usa como muestra el siguiente ejemplo:

```
int main() {
    double *A, sum;
    double startTime, endTime, runtime;
    int flag = 0;
    A = (double *)malloc(N*sizeof(double));
    startTime= omp_get_wtime();
    #pragma omp parallel for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
    endTime = omp_get_wtime();
    runtime = endTime - startTime;
    printf(" In %lf seconds, The sum is %lf \n",runtime,sum);
}
```

Se toma el tiempo de inicio y el tiempo final; luego el tiempo consumido es la resta de ambos tiempos.

## Otras rutinas de librería de OpenMP

- **omp\_set\_num\_threads:** fija el número de hilos para el proceso que la invoca e ignora la variable de ambiente **OMP\_NUM\_THREADS**.
- **omp\_get\_num\_threads:** devuelve el número de hilos en ejecución en un momento determinado.
- **omp\_get\_max\_threads:** devuelve el número máximo de hilos que lanzará el programa en las zonas paralelas.
- **omp\_get\_thread\_num:** devuelve el identificador del hilo dentro de la sección de código paralelo (automáticamente a cada hilo se le asigna un identificador entre 0 y **omp\_get\_num\_threads()** – 1). Este valor puede ser utilizado para diferenciar tareas entre los procesadores.

### 5.3.2. Combinando MPI y OpenMP

Las arquitecturas modernas permiten combinar los modelos de cómputo de memoria compartida y distribuida por ejemplo si se tienen múltiples procesadores cada uno con múltiples núcleos. Como lo muestra la Figura 36.

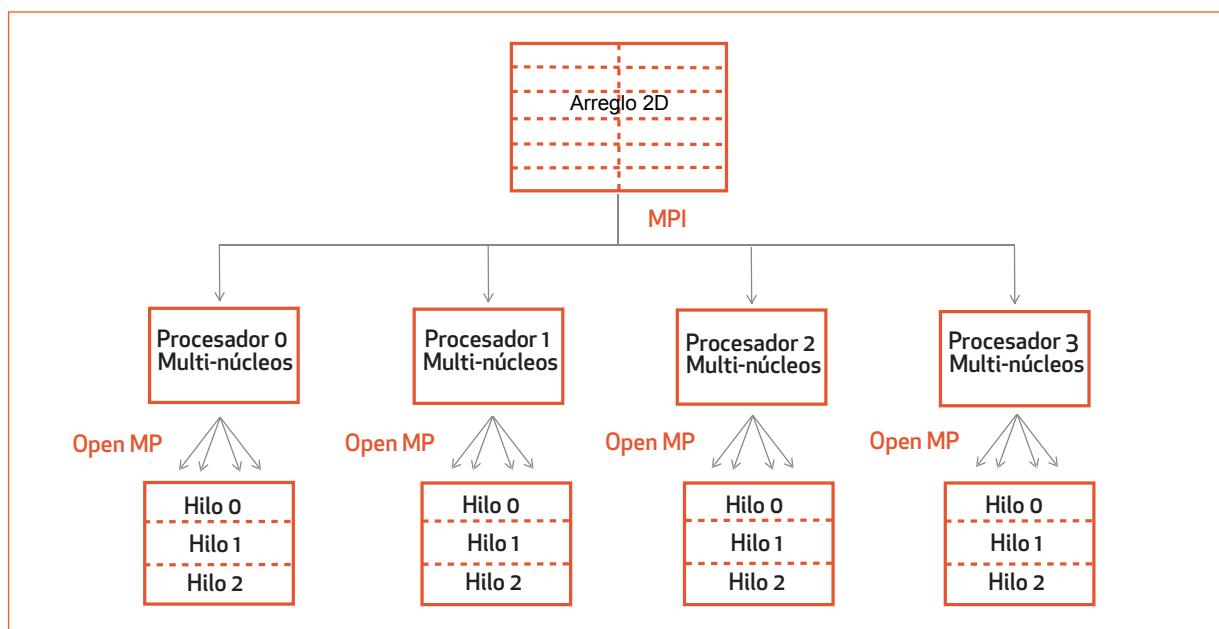


Figura 36. Combinación de MPI y OpenMP.

```

/* Programa e04.c */
#include <mpi.h>
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc,char **argv){
    int i; int nodo,numnodos;
  
```

```
int tam=32;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&nodo);
MPI_Comm_size(MPI_COMM_WORLD,&numnodos);
MPI_Bcast(&tam, 1, MPI_INT, 0, MPI_COMM_WORLD);
#pragma omp parallel
    printf("Soy el hilo %d de %d hilos dentro del procesador %d de %d procesadores\n",
        omp_get_thread_num(),omp_get_num_threads(),nodo,numnodos);
MPI_Finalize();
}
```

Para compilar el programa e07.c, se hace con mpicc y se agrega la opción de OpenMP **-fopenmp**:

```
$ mpicc -fopenmp e07.c -o e7
```

Para ejecutarlo se usa mpirun:

```
$ mpirun --disable-hostname-propagation --machinefile ./maquinas.txt -np 8 ./e7
```

La Figura 37 muestra una salida si se ejecuta en 4 procesadores. Note que en cada procesador se crean dos hilos.

```
$ mpirun --disable-hostname-propagation --machinefile ./maquinas.txt -np 4 ./e7
Soy el hilo 0 de 2 hilos dentro del procesador 0 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 0 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 1 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 1 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 2 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 2 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 3 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 3 de 8 procesadores
$
```

Figura 37. Ejecución de un programa con procesos MPI e hilos OpenMP.

## 5.4. Otras técnicas: CUDA y OmpSs

Existen otras técnicas que combinan, extienden o se inspiran en MPI u OpenMP para ofrecer herramientas de paralelización en escenarios particulares. Tales son los casos de **CUDA** y **OmpSs**.

Araíz de la tendencia creciente de combinar procesamiento entre CPU y GPUs (unidad de procesamiento gráfico) para aumentar el rendimiento de las aplicaciones, la empresa NVIDIA desarrolló **CUDA**, una arquitectura de cálculo paralelo que ahora se incluye en las GPUs, disponible para los desarrolladores aplicaciones de alto rendimiento, especialmente en áreas de procesamiento de videos e imágenes,

biología y química computacional, simulación de dinámica de fluidos, reconstrucción de imágenes, análisis sísmico, trazado de rayos, entre otras. Esto hace que los GPUs no actúan solo como procesadores gráficos, sino como procesador paralelo de propósito general accesible para cualquier aplicación.

La plataforma de cálculo paralelo CUDA proporciona unas cuantas extensiones de C y C++ que permiten implementar el paralelismo en el procesamiento de tareas y datos con diferentes niveles de granularidad. El programador puede expresar ese paralelismo mediante diferentes lenguajes de alto nivel como C, C++ y Fortran o mediante estándares abiertos como las directivas de OpenACC. Los desarrolladores disponen de una gama completa de herramientas y soluciones del ecosistema de CUDA.

**OmpSs** es una extensión de OpenMP para soportar paralelismo asíncrono y heterogeneidad (dispositivos como GPUs, FPGAs). Para lograr esto, OmpSs agrega nuevas directivas a las directivas de OpenMP, que pueden ser entendidas para extender otros APIs como CUDA y OpenCL. El paralelismo asíncrono es posible en OmpSs a través del uso de dependencias de datos entre las diferentes tareas del programa. Para soportar heterogeneidad, se introduce la cláusula **target construct**.

**Queda como tarea para el alumno investigar estas dos herramientas: más detalle de la descripción general, las APIs y ejemplos de programación.**

## Glosario

### Algoritmos embarazosamente paralelos

Son algoritmos que implementan problemas conformados por tareas que por naturaleza son independientes y paralelas. Implica entonces que hay muy poca dependencias o transmisión de información entre las distintas tareas y que no hay necesidad de coordinación o es muy ligera. Es el tipo de algoritmo paralelo más sencillo de programar, y el más efectivo. Sin embargo, no todos los problemas son susceptibles de este esquema. Algunos ejemplos de este tipo de algoritmos son: simulaciones, fórmulas que actúan de forma independiente sobre una matriz de puntos que se pueden dividir en regiones, cálculo de trayectorias independientes.

### Circuitos VLSI (*Very Large Scale Integration*)

Son circuitos integrados compuestos por millones de transistores y otros componentes en un chip de silicio muy pequeño (2 cm<sup>2</sup>). VLSI comenzó a usarse en los años 70, como parte de las tecnologías de semiconductores y comunicación que se estaban desarrollando para el momento. Los primeros chips contenían solo un transistor cada uno. A medida que la tecnología de fabricación fue avanzando, se añadieron más y más transistores y, debido a ello, más y más funciones fueron integradas en un mismo chip. El microprocesador es un dispositivo VLSI.

### Clusters de procesadores

Es una plataforma de cómputo compuesta por múltiples computadores o procesadores conectados a través de una red, preferiblemente de alta velocidad, de tal forma que sean visto como un único computador, más potente que un PC común. El cómputo con *clusters* surge como resultado de la convergencia de varias tendencias actuales que incluyen la disponibilidad de microprocesadores económicos de alto rendimiento y redes de alta velocidad, el desarrollo de herramientas de software para cómputo distribuido de alto rendimiento, así como la creciente necesidad de potencia computacional para aplicaciones que la requieran.

### Código reentrante

Un programa o subrutina se dice que es reentrant si puede ser interrumpido en medio de su ejecución y volver a invocarse de forma segura ("reentrar") antes de que las invocaciones anteriores completen su ejecución. La interrupción puede ser causada por una acción interna como un salto o llamada, o por una acción externa como una interrupción de hardware o software. Una vez que la invocación reentrante completa, las invocaciones anteriores reanudarán su ejecución de forma correcta.

### Coherencia de la memoria caché

La coherencia de caché hace referencia a la integridad y consistencia de los datos compartidos entre procesos, que pueden estar replicados en las cachés locales de los procesadores, además de la memoria principal. La coherencia debe asegurar que todas las réplicas de los datos en caché sean consistentes con la última actualización de dicho dato, independientemente del procesador que originó la última modificación. En sistemas multiprocesadores, la coherencia e caché se logra a través del *hardware*, con técnicas de escritura a través del caché (*write through cache*) y monitoreo del caché (*snoopy cache*).

### Coste total de pertenencia

Se refiere a un estimado del coste asociado, ya sea directo o indirecto, en la compra de tecnología, es decir, el coste total de disponer o tener habilitada determinada tecnología. Este coste, normalmente es usado para decidir la compra de equipos o programas informáticos, pues ayuda a los usuarios y a los gestores empresariales a determinar tanto los costes como los beneficios, relacionados con un producto o sistema.

### Herramientas de perfilamiento

Las herramientas de perfilamiento permiten recopilar información sobre el comportamiento de un programa durante su ejecución. Al realizar el perfilado, se miden el tiempo de ejecución y el número de llamadas de funciones y líneas individuales en el código del programa. Con esta herramienta, el programador puede encontrar las secciones más lentas del código y optimizarlas.

### Multicomputadores

Son un tipo especial de plataforma de cómputo conformado por múltiples computadores clásicos. La memoria es privada (es decir, cada procesador tiene un mapa de direcciones propio que no es accesible directamente a los demás). La comunicación entre procesadores es por pase de mensajes a través de una red de interconexión. Algunos ejemplos de multi computadores son: una red LAN o WAN, un *cluster* de computadores.

### Multiprocesadores

Son un tipo especial de plataforma de cómputo conformado por múltiples procesadores que pueden tener memoria privada (es decir, cada procesador tiene un mapa de direcciones propio que no es accesible directamente a los demás) o compartir una memoria principal común. La comunicación entre procesadores es por pase de mensajes a través de una red de interconexión, si es de memoria distribuida o a través del bus que conecta los procesadores a la memoria. Las PCs con múltiples núcleos o procesadores son multiprocesadores, algunos supercomputadores pueden tener este tipo de arquitecturas.

### Procesadores multi núcleos

Un procesador multi núcleos es aquel que combina dos o más microprocesadores independientes en un solo circuito integrado. Un dispositivo de doble núcleo contiene solamente dos microprocesadores independientes. En general, los microprocesadores multi núcleos permiten que un dispositivo computacional exhiba una cierta forma del paralelismo a nivel de hilos, sin incluir múltiples microprocesadores en paquetes físicos separados.

### Programa ejecutable

Un programa o código ejecutable está conformado por un conjunto de instrucciones de bajo nivel que son generadas por un compilador o interpretador y que son interpretadas por el CPU. Un código ejecutable está listo para ser ejecutado por el CPU.

### Programa fuente

Un programa o código fuente está conformado por un conjunto de instrucciones de alto nivel, escritas en un lenguaje de programación determinado, elegido por el programador, tal como: Basic, C, C++, C#, Java, Perl, Python, PHP. El código fuente no es directamente ejecutable por el computador, sino que debe ser traducido a un código ejecutable o código binario, que podrá ser interpretado por el CPU. Para esta traducción se usan los llamados compiladores, ensambladores, intérpretes y otros sistemas de traducción.

### Programación paralela

Implica el uso de múltiples recursos computacionales para resolver un problema computacional. Se distingue de la computación secuencial en que varias operaciones pueden ocurrir simultáneamente. El problema se divide en partes discretas que se pueden resolver simultáneamente. Cada parte se descompone en una serie de instrucciones. Las instrucciones de cada parte se ejecutan simultáneamente en diferentes procesadores. Se emplea un mecanismo global de control/coordinación.

### Supercomputadores

Un supercomputador es un ordenador con capacidades de cálculo superiores a las computadoras comunes y de escritorio y es usado con fines específicos, especialmente en el ámbito científico. Hoy día el término de supercomputador está siendo reemplazados por computadora de alto rendimiento, dada las dimensiones que han adquirido recientemente. En noviembre de 2018, el supercomputador más rápido del mundo estaba formado por más de dos millones de procesadores.

### Tasa de retorno de la inversión

Es la tasa de interés o rentabilidad que ofrece una inversión. Es decir, es el porcentaje de beneficio o pérdida que tendrá una inversión para las cantidades que no se han retirado del proyecto. La tasa de retorno de la inversión (TRI) puede utilizarse como indicador de la rentabilidad de un proyecto: a mayor TRI, mayor rentabilidad; así, se utiliza como uno de los criterios para decidir sobre la aceptación o rechazo de un proyecto de inversión.



## Enlaces de interés

### Dhrystone Benchmark History and Results

**Autores:** Roy Longbottom's PC Benchmark Collection

En este enlace encontrará una descripción detallada de este *benchmark* y ejemplos de su aplicación y uso en distintas arquitecturas con fines comparativos.

<http://www.roylongbottom.org.uk/dhrystone%20results.htm>

### HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers

**Autores:** A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary

Otro *benchmark* popular para comparar el rendimiento de computadores. En este enlace encontrará una descripción general de este *benchmark*, ejemplos de su aplicación y uso en distintas arquitecturas con fines comparativos, documentación y mucha más información; así como los programas que lo conforman. Intente bajarlo y mida el rendimiento de su computador.

<https://www.top500.org/project/linpack/>

### Livermore Loops Benchmark Results On PCs

**Autores:** Roy Longbottom's PC Benchmark Collection

En este enlace encontrará una descripción detallada de este *benchmark* y ejemplos de su aplicación y uso en distintas arquitecturas con fines comparativos.

<http://www.roylongbottom.org.uk/livermore%20loops%20results.htm>

### Open MP

Este enlace es el sitio oficial del proyecto Open MP. A través de este enlace encontrará extensa documentación acerca de la librería, tutoriales y ejemplos de uso.

<http://openmp.org/>

### Open MPI: Open Source High Performance Computing

Este enlace es el sitio oficial del proyecto OpenMPI. A través de este enlace encontrará extensa documentación acerca de la librería, tutoriales y ejemplos de uso.

<http://www.open-mpi.org/>

**TOP500: The List.**

**Autor: Jack Dongarra, Top500.org**

Desde 1993, el proyecto TOP500, mantiene el *ranking* de las 500 supercomputadoras más rápidos del mundo. Se actualiza dos veces por año: en junio y noviembre. En el enlace encontrará la lista TOP500 con sus descripciones y detalles de la arquitectura y software que utiliza, así como el tipo de aplicaciones que se ejecutan en tales computadoras.

<https://www.top500.org/>

**Whetstone Benchmark History and Results**

**Autores: Roy Longbottom's PC Benchmark Collection**

En este enlace encontrará una descripción detallada de este *benchmark* y ejemplos de su aplicación y uso en distintas arquitecturas con fines comparativos.

<http://www.roylongbottom.org.uk/whetstone.htm>

## Bibliografía

### Referencias bibliográficas

- Darema, F., George, D. A., Norton, V. A., Pfister, G. F. (1988). *A single-program multiple-data computational model for EPEX/FORTRAN*. Parallel Computing. 7 (1): 11–24. doi:10.1016/0167-8191(88)90094-4.
- Flynn, M. J. (1966). *Very high-speed computing systems*. Proceedings of the IEEE, 54(12), pp. 1901-1909.
- Foster, I. (1995). *Designing and Building Parallel Programs*, (Vol. 78). Addison Wesley Publishing Company.
- Gropp, W. , Lusk, E. & Skjellum, A (1994). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Tre MIT Press.
- Gustafson, J. L. (1988). *Reevaluating Amdahl's Law*. Communications of the ACM. 31 (5): 532–3. doi:10.1145/42411.42415.
- InfiniBand Trade Association et al. (2000). *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association.
- Petrini, F., Feng, W.-C., Hoisie, A., Coll, S. & Frachtenberg, E. (2001). *The Quadrics network (QsNet): high-performance clustering technology*. In Hot Interconnects 9, pages 125–130.
- Quinn, M. J., & Quinn, M. J. (1994). *Parallel computing: theory and practice* (Vol. 2). New York: McGraw-Hill.
- Seitz, C. (1994). *Myrinet—a gigabit-per-second local-area network*. In Hot Interconnects II, Symposium Record, pages 161–180.
- Tanenbaum, A & Steen, M. van (2017). *Distributed Systems Principles and Paradigms*. Tercera Edición. Prentice-Hall.

# Agradecimientos

## **Autora**

Yudith C. Cardinale Villarreal



viu  
.es