
Tema 1

Arquitecturas MIMD

Nicolás Calvo Cruz

Dpto. de Arquitectura y Tecnología de los Computadores

Telegram: [@nkalvocruz](https://t.me/nkalvocruz)

email: nkalvocruz@ugr.es

Índice

1. Objetivos
2. Motivación
3. Qué es el High Performance Computing (HPC)
4. Un poco de historia
5. Aplicaciones
6. Clasificación de Arquitecturas y Criterios de Clasificación
7. Enfoques
8. Evaluación de Prestaciones
9. Limitación de prestaciones
10. Ley de Amdahl, Ley de Gustafson
11. Mejorando prestaciones

Objetivos

- Conocer arquitecturas paralelas
- Entender los criterios de clasificación
- Repasar enfoques de diseño de arquitecturas paralelas



Motivación

1 Cada día hay más **necesidades de recursos para computación**, simulación numérica, gráficos, sistemas de predicción y modelado.

2 El conocimiento de las arquitecturas existentes **es una ventaja** a la hora de ganarle la partida a un algoritmo que deseamos acelerar, permitiendo que el software se adapte al hardware de la mejor forma posible.

3 La simulación de grandes sistemas donde los datos son el flujo fundamental para su funcionamiento deben ir **acoplados a la arquitectura**, sacándole así el máximo partido.

4 La simulación numérica permite:

- Disminuir los costes de producción
- Disminuir los costes de construcción de sistemas reales
- Aumentar la productividad disminuyendo el tiempo de desarrollo
- Aumentar la seguridad

3. Qué es el High Performance Computing (HPC)

Es la rama de la **Informática** que estudia la resolución de **problemas computacionalmente costosos** mediante el **máximo aprovechamiento** de los recursos hardware disponibles.

Permite que la comunidad científica e ingenieril solucionen complejos problemas empresariales, industriales y científicos con la ayuda de aplicaciones que requieren un **gran ancho de banda, una red de baja latencia y altas prestaciones** en el sistema computacional donde se resuelven.

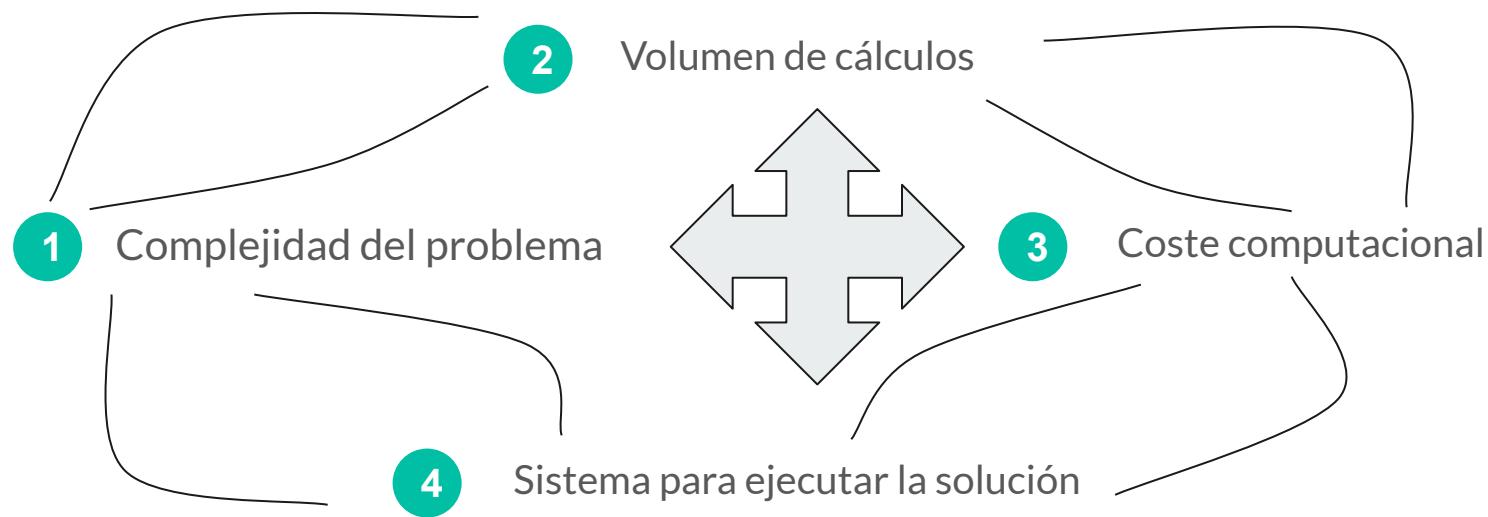


3.1 Factores que deben estar presentes

1. Problema a resolver de forma computacional
2. Gran volumen de cálculos a realizar
3. Gran coste computacional (“**tiempo**”, ¿**energía**?)
4. Sistema con capacidad de paralelización de tareas y capacidad de cómputo suficiente para resolver el problema



3.2 Influencia de los factores



3.3 Cosas a tener en cuenta

HPC va más de **FLOPS** que de **tiempos**, pero no siempre incluye operaciones en punto flotante.

HPC habla de procesos, de hebras/hilos, o de ambos simultáneamente, para **aprovechar toda la máquina**.

HPC para las compañías es más bien qué máquina te venden y qué características tiene que se adapten a lo que quieras hacer.

HPC va más de **optimización y adaptación**. La optimización es hacer algo lo mejor posible, y para eso hay que tener una **comparación**, para **saber si has mejorado o no**.

4. Un poco de Historia

1. Durante los últimos 50 años, hemos aumentado la **frecuencia** de los relojes y **disminuido el coste**
2. Pero esto **ya no es posible**
3. Crecemos en capacidad a diferentes niveles
 - a. Aumentando más cores on-chip (Ley de Moore) (SMP)
 - b. Aumentando el ancho de las unidades funcionales (SIMD)
 - c. Combinando nodos individuales (MIMD)
 - d. Uniendo otras arquitecturas (GPGPU)



4. Un poco de Historia



1. Seymour Cray creó el primer supercomputador, o la primera máquina así llamada, a finales de los 60.
2. El primer supercomputador fue el CDC 6600, que utilizaba una sola CPU con un repertorio de instrucciones RISC y un pipeline capaz de ejecutar una instrucción en cada ciclo.
3. Tenía más prototipos, pero dejó CDC en 1972 y fundó su propia compañía para abandonar los multiprocesadores y dedicarse a los procesadores vectoriales
4. Lideró el mercado durante 5 años, de 1985 a 1990

4.1 Necesidades

1. **Compiladores adaptados a las unidades funcionales que tenga la máquina (32, 64...)**
2. **Uso de instrucciones que aprovechen la arquitectura, como las vectoriales**
3. **Eliminación de dependencias de forma automática**
4. **Distribuir de forma semiautomática los accesos a memoria y minimizarlos** (por eso está indicado para aplicaciones con mucho cálculo)
5. **Evitar comunicaciones innecesarias y optimizarlas** porque la latencia de red perjudica al rendimiento



4.2 Tres niveles ... tres tipos de paralelismo

1. Paralelismo a nivel de comunicación entre los nodos del sistema masivamente paralelo (**Procesos**)
2. Paralelismo a nivel de unidades de ejecución dentro del mismo nodo (**Hebras**)
3. Paralelismo a nivel de datos, utilizando unidades funcionales paralelas para las operaciones que lo permitan (**SIMD**)

4.3 La importancia de fijar el esfuerzo

- Si se desea escalar un programa hasta poder ejecutarlo en unos 10.000 procesadores simultáneamente, y estamos en un punto en el que solo se ejecuta en 500, habrá que optimizar el proceso de escalado a nivel de nodo, ya que las comunicaciones entre nodos afectarán mucho, reduciendo la sincronización, el desbalanceo de carga.
- Si la aplicación escala bien, es decir, pasa de ejecutarse en 5 procesadores a ejecutarse en 100 procesadores sin incrementar en exceso el tiempo de comunicación, entonces nuestro esfuerzo debe centrarse en optimizar el paralelismo dentro de los nodos, reduciendo accesos a memoria, quitando dependencias, etc.



5. Aplicaciones

- Las **aplicaciones** deben estar especialmente **diseñadas para sacar partido de la naturaleza paralela de los sistemas** que las ejecutan.
- Además, los algoritmos deben permitir extraer esa ventaja, y **adaptarse a una máquina que no tenga esas características**, sin más que volverlo a compilar o ejecutarlo con otros parámetros.
- La elección de cualquier parámetro puede afectar a la ejecución.
- La elección del **lenguaje**, y los **esquemas de comunicación** son muy importantes (Lo veremos más a fondo en el tema 2).

5. Aplicaciones

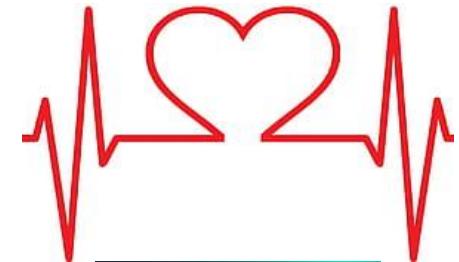
Salud: Aunar tecnología y medicina ha permitido, por ejemplo, digitalizar la simulación de procesos y analizar todos sus datos:

1. Doblado y secuenciación de una proteína
2. La secuenciación del material genético humano para detectar posibles enfermedades
3. Simular el efecto de ciertos compuestos
4. ...



LIVING HEART PROJECT: A CYBER-HEART.

Stanford, California



TEXAS ADVANCED COMPUTING CENTER: IS CANCER WRITTEN IN OUR DNA?

Austin, Texas

RADY CHILDREN'S INSTITUTE FOR GENETIC MEDICINE: SWIFT GENETIC TESTING.

San Diego, California

BOEING 787 Seattle, Washintong DC.

5. Aplicaciones

Ingeniería: La creación de obras de ingeniería es un reto cada vez mayor. Las simulaciones las hacen más seguras, más baratas y acelera su desarrollo.

1. Simulación de aviones
2. Simulaciones de fluidos
3. Simulación de centrales
4. ...



BICICLETAS TREK. Waterloo, Wisconsin

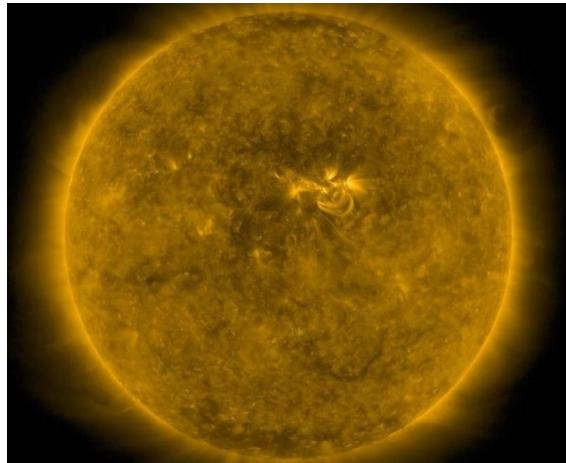


SUPERTRUCK, OPTIMIZACIÓN DE CONSUMOS EN CAMIONES.
EE.UU. Department of Energy.

5. Aplicaciones

Investigación Espacial: El espacio es algo totalmente desconocido, y se investiga en muchos ámbitos:

1. Vida inteligente fuera de la Tierra (*SETI @ home*)
2. Movimientos de meteoritos
3. De dónde viene el universo
4. ...



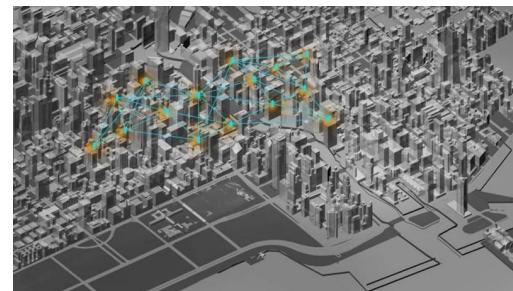
NASA. Washington D.C. .

POLARBEAR, DE DONDE VIENE TODO Berkeley. California

5. Aplicaciones

Redes de Transporte: Las Smart Cities son un objetivo que se conseguirá en no muchos años. Una ciudad es una fuente ingente de datos de movilidad, clima, patrones de tráfico, niveles de ruido... lo que permite:

1. Predecir contaminación
2. Coordinación de transportes
3. Diseñar de controles de semáforos
4. ...



ARRAY OF THINGS

Chicago, EE.UU



PREVISIÓN DE HUMO

Iowa City, EE.UU.



CONSTRUCCIONES SMART

Tianjin, China.

5. Aplicaciones

Economía y Negocio

1. Criptomonedas
2. Búsqueda de clientes
3. Predicciones del mercado financiero
4. Procesos de producción más rápidos y eficientes
5. Ubicación de negocios
6. ...

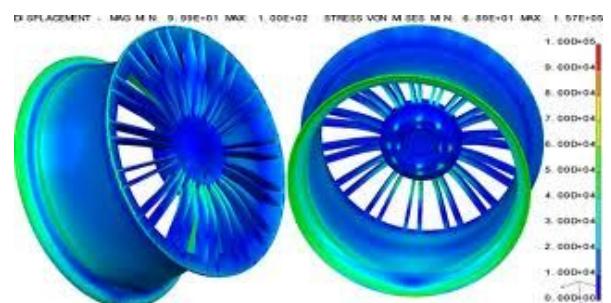
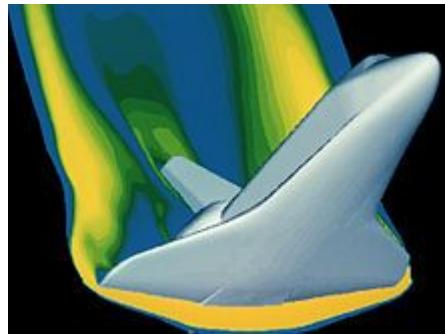
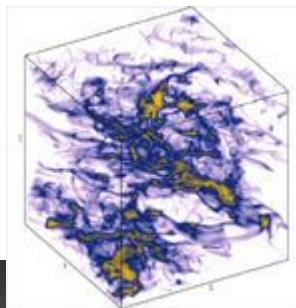
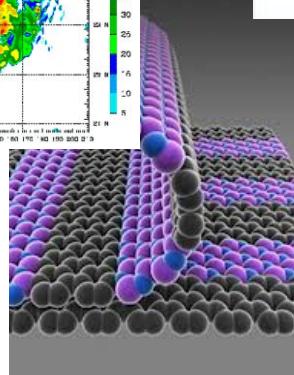
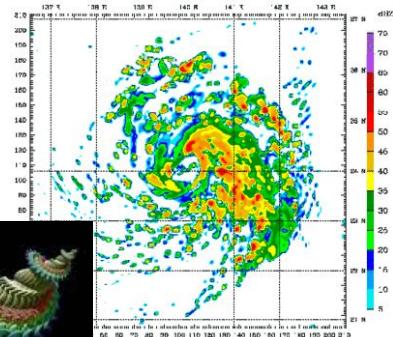
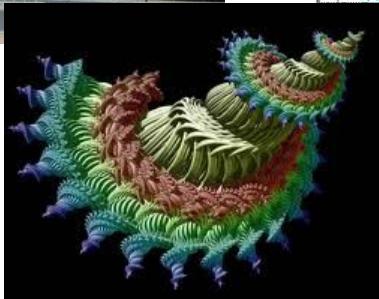


**CRISTALES EFICIENTES Y
SEGUROS** Tokyo, Japón



BITCOINS The Web.

5. Aplicaciones



Índice

1. Objetivos
2. Motivación
3. Qué es el High Performance Computing (HPC)
4. Un poco de historia
5. Aplicaciones
6. Clasificación de Arquitecturas y Criterios de Clasificación
7. Enfoques
8. Evaluación de Prestaciones
9. Limitación de prestaciones
10. Ley de Amdahl, Ley de Gustafson
11. Mejorando prestaciones

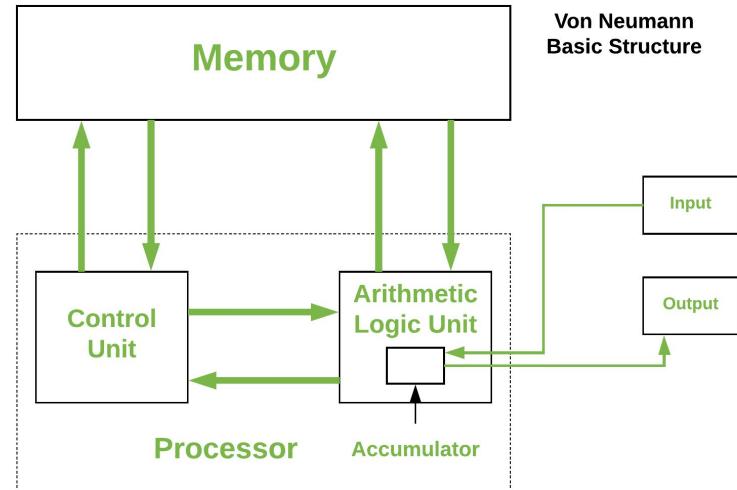
6. Clasificación de Arquitecturas



6. Clasificación de MIMD

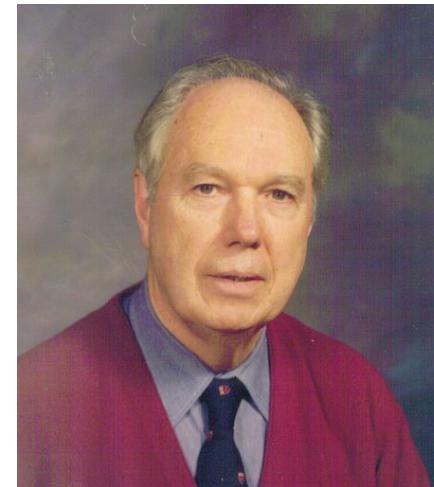
6.1 Conceptos previos

- La máquina de von Neumann
- ¿Cómo funciona?
- ¿Cuál es su problema? ¿Cuellos de botella?
- ¿Modos de aumentar su capacidad?
 - Caching, Jerarquía de memoria
 - Memoria Virtual: Para parecer que se hace más de una cosa a la vez sin gastar tiempo en crear procesos completos cada vez
 - Ejecutar más de una instrucción a la vez, pipelining, envío a ejecución de más de una instrucción, multithreading...
 - Aumentar el número de máquinas y coordinarlas



6. Clasificación de MIMD

6.1 Taxonomía de Flynn

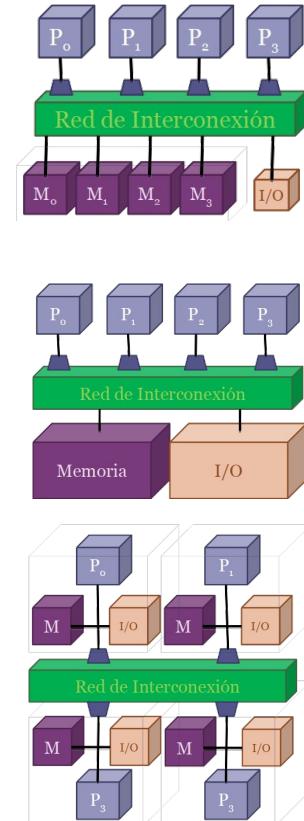
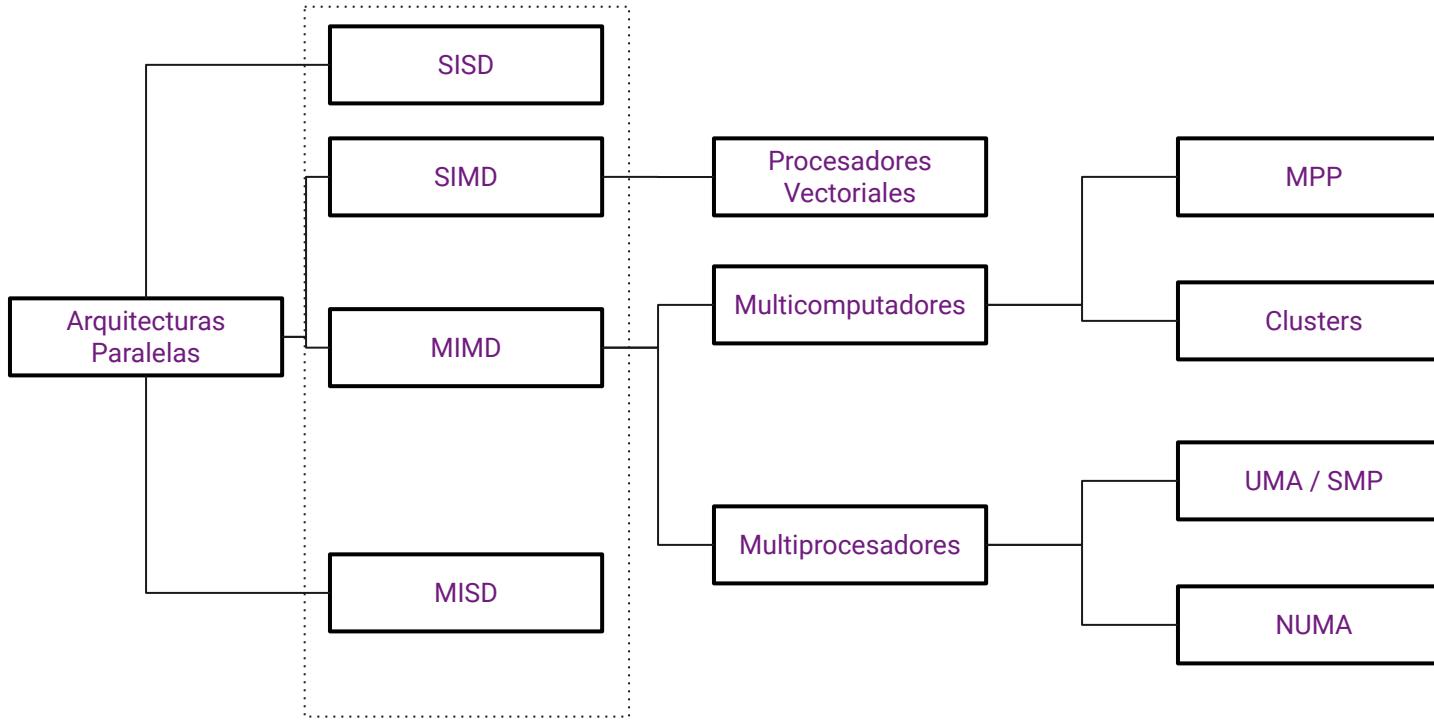


Michael J. Flynn

	Un solo dato usado (SD)	Muchos datos usados (MD)
Una instrucción (SI)	SISD	SIMD
Varias instrucciones (MI)	MISD	MIMD

6. Clasificación de MIMD

6.2 Según Andrew S. Tanenbaum



6. Clasificación de MIMD

6.3 MIMD

1. Normalmente son **máquinas asíncronas**: cada unidad de procesamiento tiene su propio reloj
2. No existe una sola arquitectura posible
 - a. Múltiples unidades de ejecución y control y una sola memoria: **multiprocesadores**
 - b. Múltiples unidades de memoria, y múltiples unidades de ejecución y control: **multicomputadores**
 - c. Enfoques Híbridos
3. Actualmente:
 - a. En una máquina para HPC, **se mezclan todos** los enfoques de paralelización de arquitectura. Se tienen multicomputadores donde cada uno de los nodos a su vez es un multiprocesador, y donde la red que une todos los nodos es una red de altas prestaciones.

6. Clasificación de MIMD

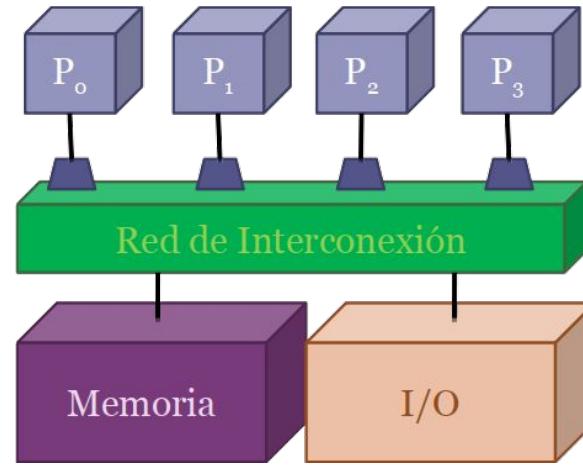
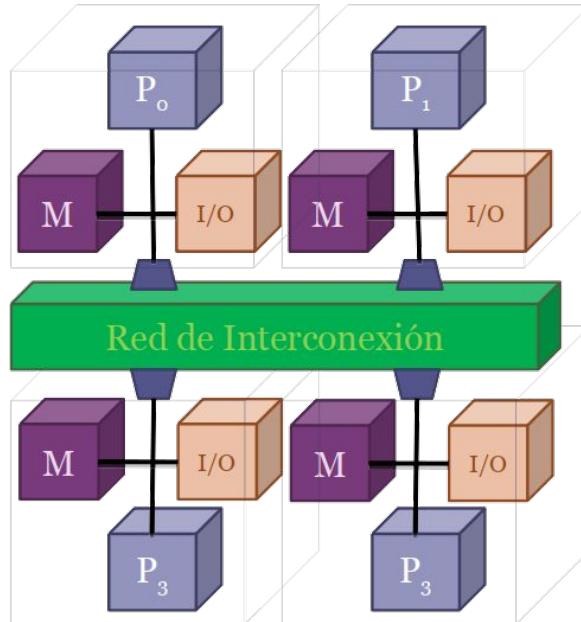
6.4 Criterios de clasificación

1. Criterios temporales: máquinas síncronas o asíncronas
2. Criterios de control: máquinas con un control centralizado o distribuido
3. Criterios relacionados con la distribución física de la memoria:
 - a. Memoria centralizada (Multiprocesadores)
 - b. Memoria distribuida (Multicomputadores)
4. Criterios relacionados con el espacio de direcciones de la memoria:
 - a. Único (Shared Memory Systems)
 - b. Múltiple (Distributed Memory Systems)
5. Criterios relacionados con los tiempos de acceso a memoria:
 - a. Tiempo de acceso a memoria no uniforme (NUMA)
 - b. Tiempo de acceso a memoria uniforme (UMA)
6. Criterios según la red de interconexión
 - a. Máquinas con redes estáticas
 - b. Máquinas con redes dinámicas
7. Criterios comerciales, según precio y prestaciones

6. Clasificación de MIMD

6.4 Criterios de clasificación

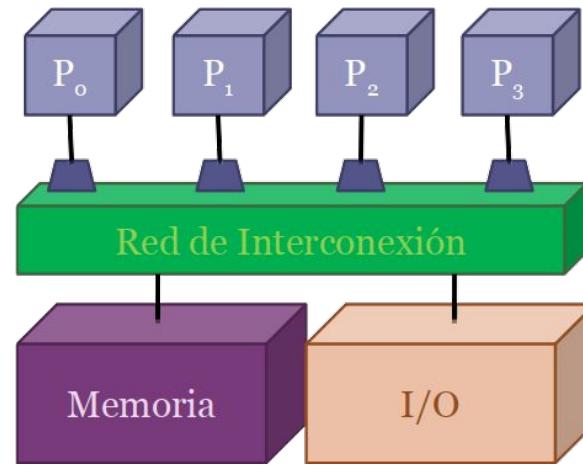
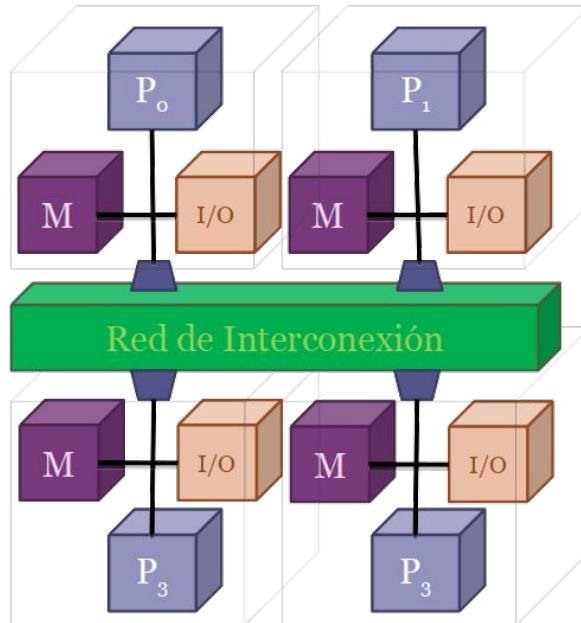
Distribución de la memoria



6. Clasificación de MIMD

6.4 Criterios de clasificación

Espacio de direcciones, ÚNICO O MÚLTIPLE

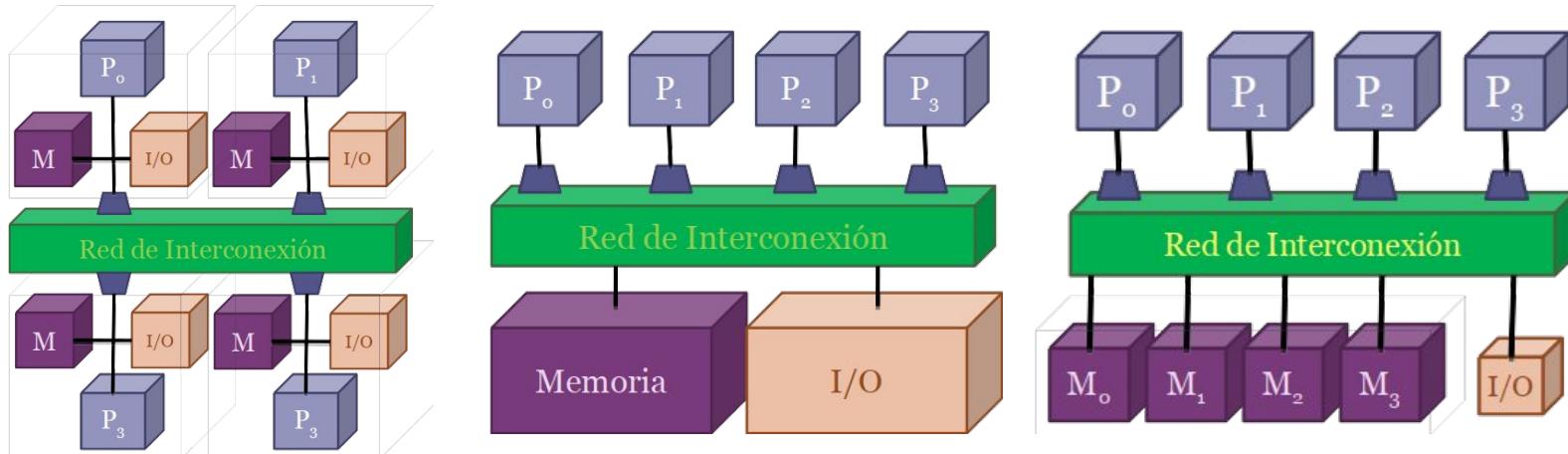


6. Clasificación de MIMD

6.4 Criterios de clasificación

Tiempo de acceso a memoria

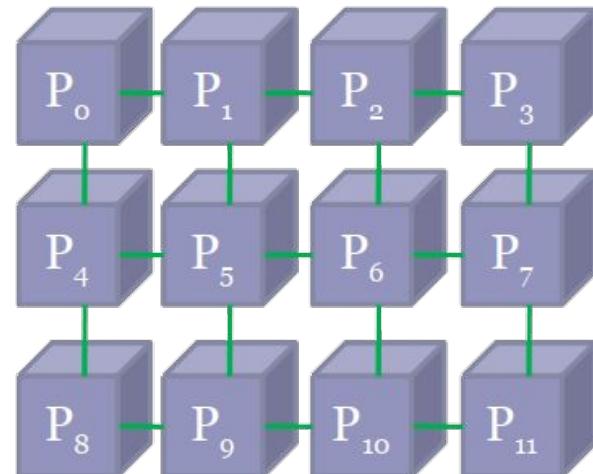
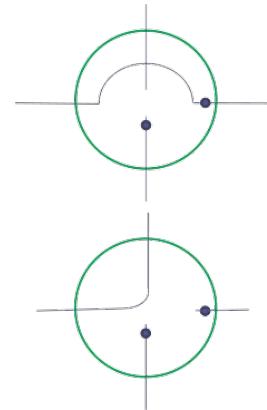
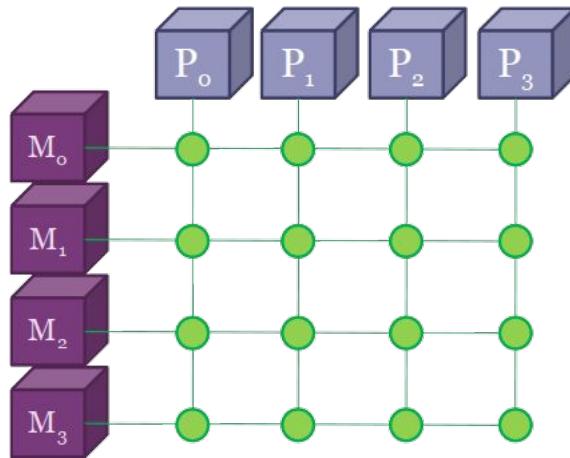
NUMA, UMA, NORMA



6. Clasificación de MIMD

6.4 Criterios de clasificación

Red de interconexión



Indirecta

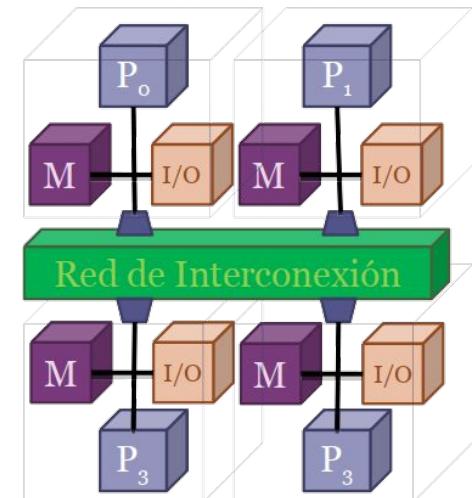
Directa



7. Enfoques

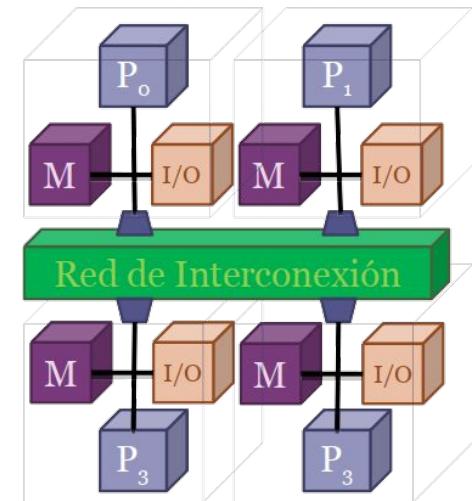
7.1 Multicomputadores

- Tiempo de acceso a memoria no uniforme
- Más escalables o ampliables, la limitación es la red de interconexión
- Comunicación y Sincronización por paso de mensajes
- Programación más compleja: el programador debe distribuir el trabajo y los datos además del código
- Normalmente se programan con enfoque de Paso de Mensajes, ya sean síncronos o asíncronos
- Ventajas:
 - Independencia de las partes paralelas, mantenimiento
 - Escalabilidad
 - Ampliación
- Desventajas:
 - Programación
 - Red de interconexión
 - Sincronización



7.1 Multicomputadores

- Con un solo sistema operativo y un solo reloj: control centralizado
- Con más de un sistema operativo y varios relojes: control distribuido.
- Hoy en día, casi todos los nodos tienen capacidad de cómputo paralelo también, ya sea por el paralelismo a nivel de instrucción, o a nivel de hebra/funcional.
- Herramientas disponibles:
 - Sistemas operativos específicos
 - Compiladores
 - Lenguajes o librerías que permitan el paso de mensajes



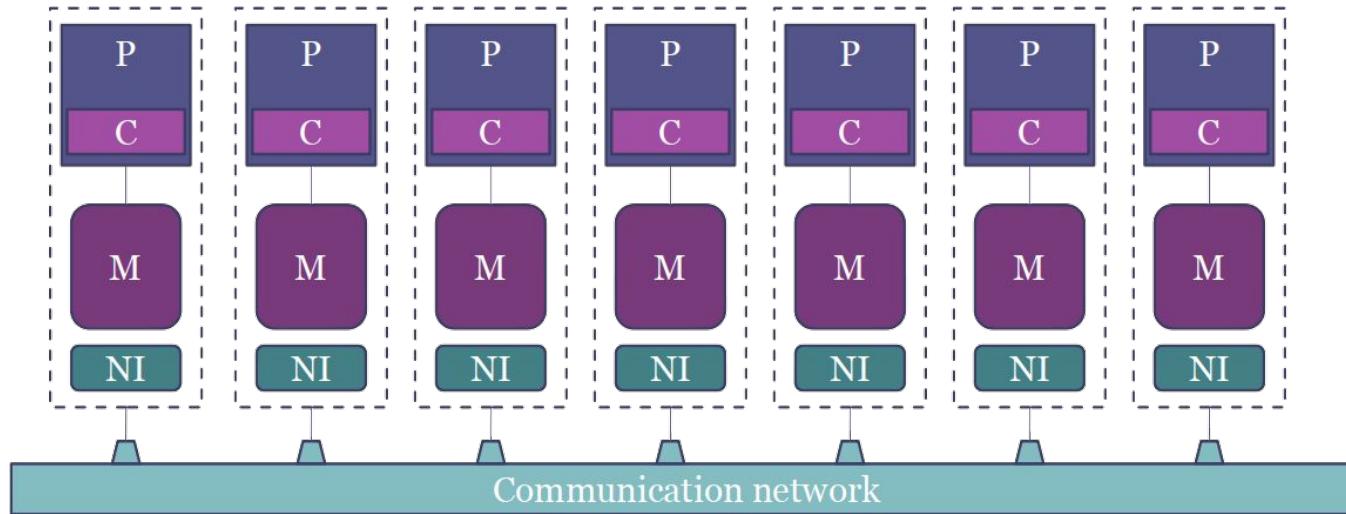
7.1 Multicomputadores

NORMA

- No Remote Memory Access
- Se trata de un multiprocesador con un sistema de direcciones privado para cada unidad de procesamiento
- Los **espacios de memoria no son accesibles entre nodos**
- Su única ventaja es que son **altamente escalables**, más que CC-NUMA o COMA, ya que no tiene que mantener la coherencia de memoria en el sistema completo
- Su **programación es extremadamente complicada**, ya que hay que partir los datos en memorias locales y mantener la consistencia mediante la programación

7.1 Multicomputadores

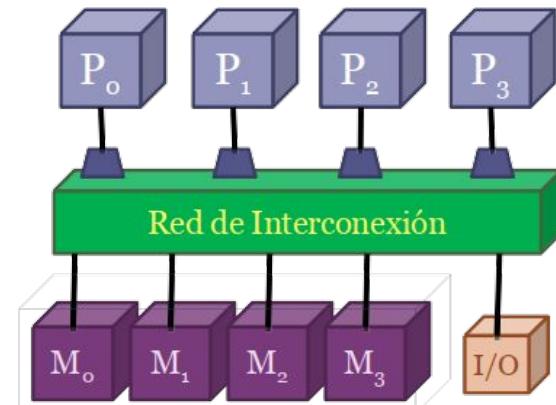
Estructura simplificada NORMA



Visión simplificada de un sistema de memoria distribuida en la que cada nodo de procesamiento tiene un solo nivel de caché, y un solo módulo de memoria. La comunicación se realiza a través de una interfaz de red y la conexión es directa a la red de comunicaciones. Ningún procesador puede acceder a otro módulo de memoria que no sea el suyo, por eso se denominan No Remote Memory Access System (NORMA)

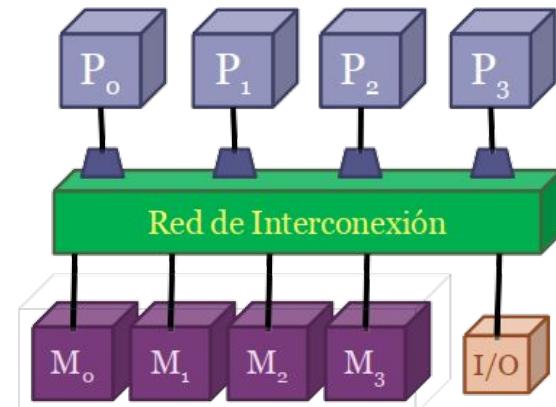
7.2 Multiprocesadores

- Espacio de memoria uniforme
- Aumenta la latencia por la red de interconexión que se sitúa entre las unidades de procesamiento y la memoria
- La sincronización es por hardware usando primitivas de sincronización
- Programación más sencilla
- Herramientas disponibles:
 - Compiladores
 - Lenguajes con capacidad de manejo de hebras
 - ...
- No se distribuye ni código ni datos
- Comunicación a través de variables compartidas



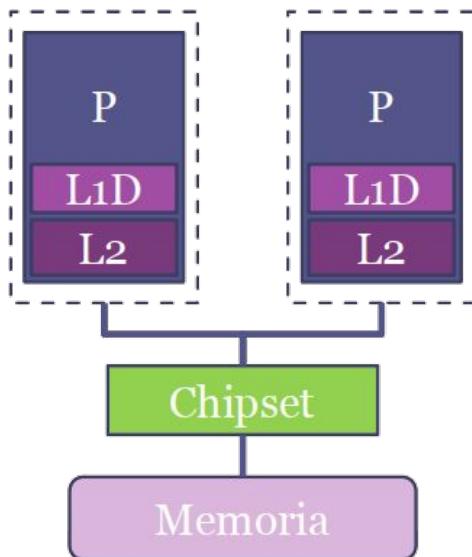
7.2 Multiprocesadores

- Clasificación:
 - UMA
 - SMP
 - ¿COMPUTACIÓN HETEROGÉNEA?
 - NUMA
 - CC-NUMA: Cache Coherent- Non Uniform Memory Access
 - Non CC-NUMA: Non Cache Coherent- Non Uniform Memory Access
 - COMA: Cache Only Memory Access

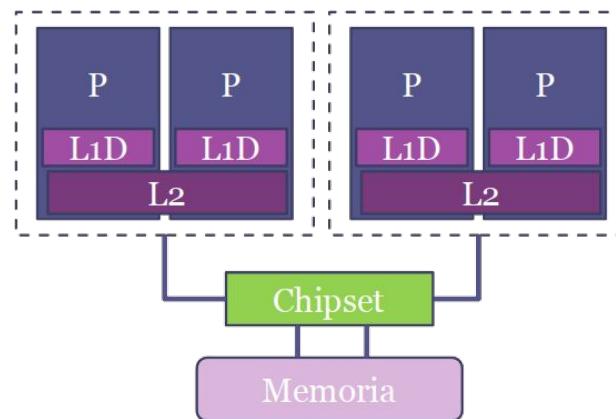


7.2 Multiprocesadores

Esquemas simplificados comunes



Sistema UMA con dos procesadores single-core que comparten un bus (frontside bus -FSB-)

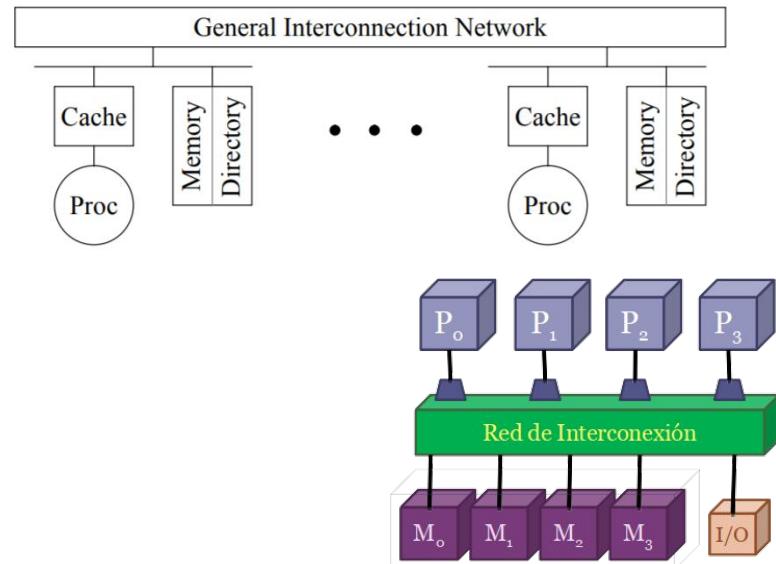


Sistema UMA con dos procesadores dual-core que no comparten la conexión al bus (frontside bus -FSB-)

7.2 Multiprocesadores

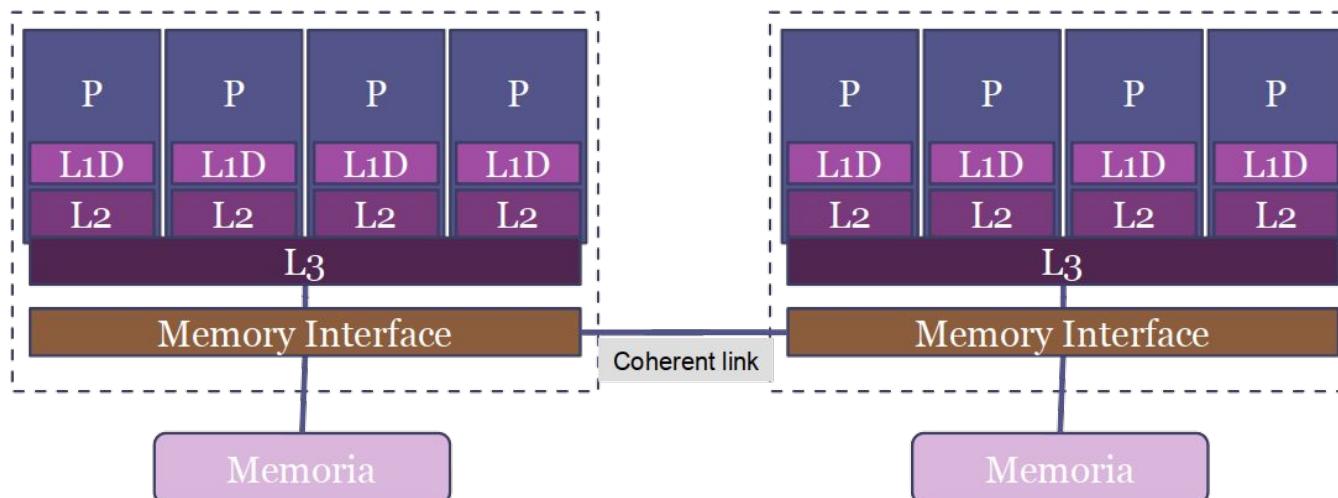
CC-NUMA

- Cada procesador tiene asociado una cache y una porción propia de la memoria compartida
- La coherencia de memoria se realiza con protocolos basados en directorios
- El acceso a cada línea de memoria se produce mediante hardware
- El soporte de acceso lo proporciona el sistema operativo normalmente



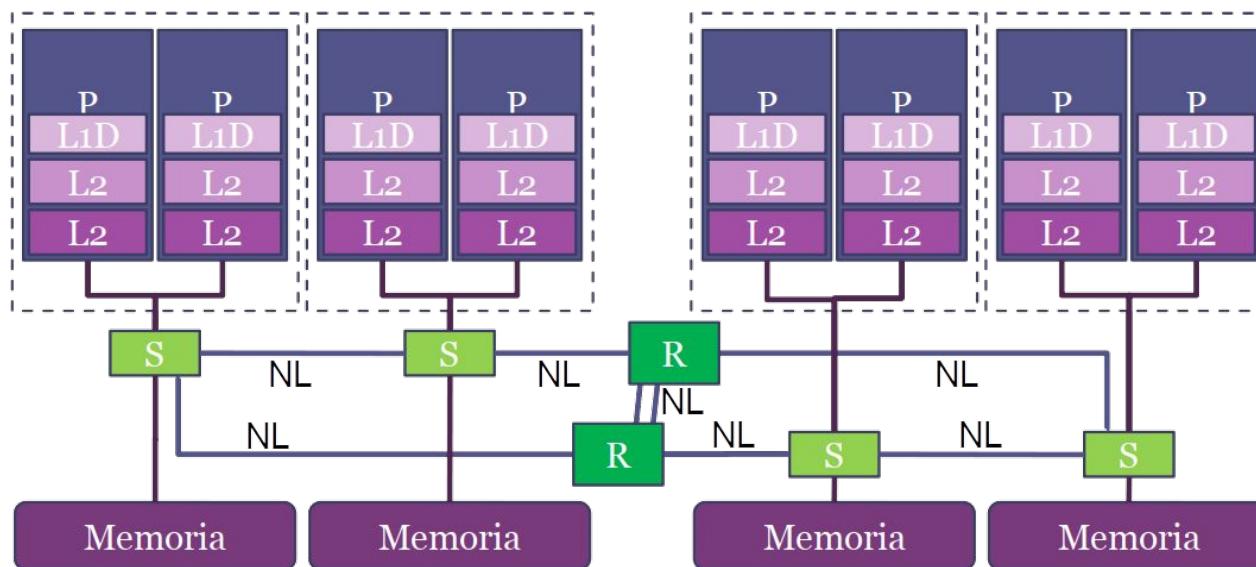
7.2 Multiprocesadores

CC-NUMA



Sistema ccNUMA con dos dominios de localidad (uno por cada socket) y 4 cores cada uno de los dominios

7.2 Multiprocesadores CC-NUMA

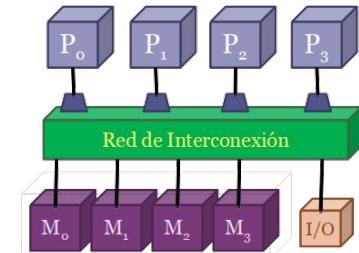


Sistema ccNUMA de SGI Altix con cuatro dominios de localidad, cada uno con un socket (S) y un procesador dual-core en cada dominio. Las conexiones del sistema de memoria son NUMALink (NL) y utilizando dos routers (R). Se usan los accesos a NL solo cuando el acceso a memoria de cada dual-core no es local

7.2 Multiprocesadores

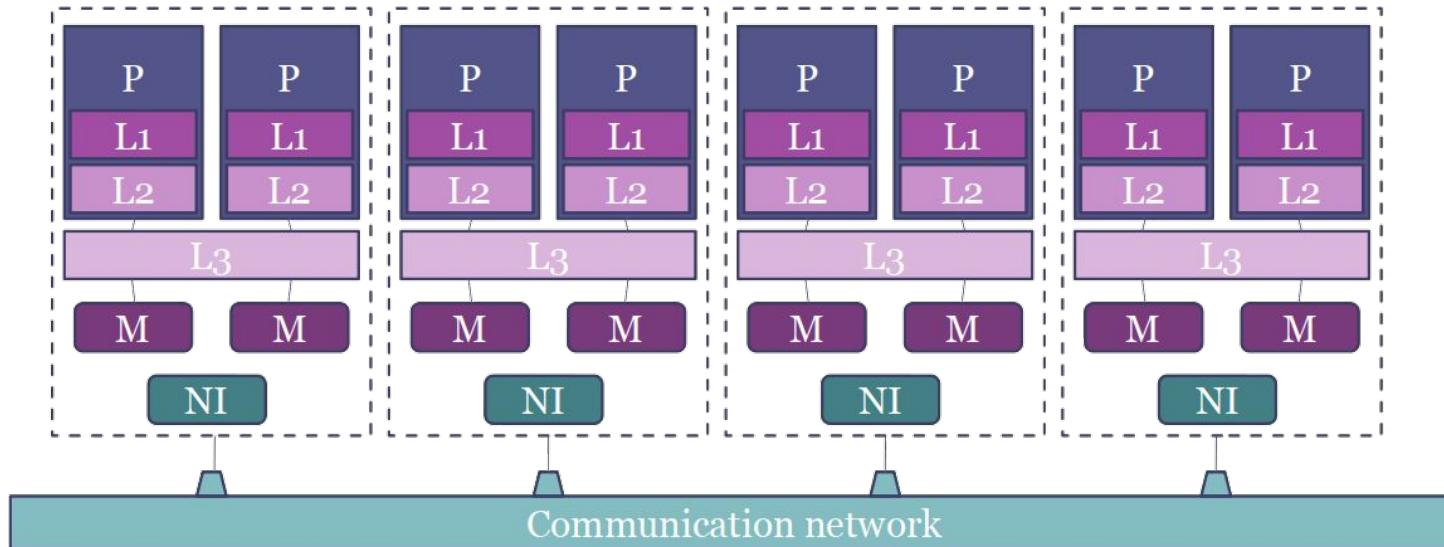
COMA

- Se trata de una máquina CC-NUMA, pero toda la memoria compartida se trata como una **cache gigante**.
- Al gestionar toda la memoria como una caché , cada línea tiene un contenido y un estado
- Cuando un procesador necesita una línea, se busca y se replica en la porción de memoria compartida local de ese proceso (si no estaba ya).
- Esta arquitectura incrementa las posibilidades de encontrar datos en la zona de memoria local de cada procesador
- La jerarquía de memoria es capaz de gestionar y evitar largas latencias de acceso a memoria mediante adelantamientos de lecturas.



7.3 Multiprocesadores y Multicomputadores

Estructura híbrida simplificada



Sistema híbrido con nodos de memoria compartida tipo CC-NUMA. Se mantienen dos sistemas de comunicación, uno a través de los NI y otro a través del sistema de memoria caché de nivel 3. El modelo de programación para estos sistemas incluyen paso de mensajes y memoria compartida.



Gracias.



Trabajos Primera Semana

Grupo 1: BOINC

Grupo 2: BITCOINS Y HPC

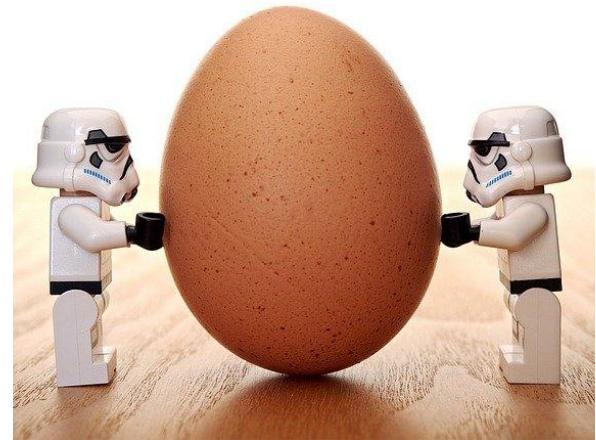
Grupo 3: PROFILING Y TRACING, DIFERENCIAS Y SIMILITUDES

Grupo 4: HERRAMIENTAS PARA HPC CON EJEMPLOS CONCRETOS

Grupo 5: CRITERIOS DE CLASIFICACIÓN Y RANKING DE MÁQUINAS DEL TOP 500

Grupo 6: CRITERIOS DE CLASIFICACIÓN Y RANKING DE MÁQUINAS DEL GREEN 500

Grupo 7: SOLUCIONES COMERCIALES PARA HPC DE LOS PRINCIPALES FABRICANTES



8. Evaluación de Prestaciones

Índice

1. Objetivos
2. Motivación
3. Qué es el High Performance Computing
4. Un poco de historia
5. Aplicaciones
6. Clasificación de Arquitecturas y Criterios de Clasificación
7. Enfoques
8. Evaluación de Prestaciones
9. Limitación de prestaciones
10. Ley de Amdahl, Ley de Gustafson
11. Mejorando prestaciones

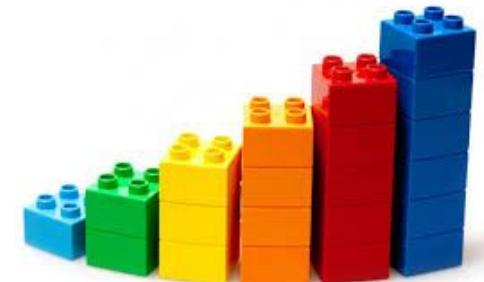
8.1 Medir tiempos

- No siempre interesa el tiempo desde el inicio hasta el final.
- Importa el **tiempo necesario para obtener una solución** a nuestro problema.
- Debemos **comparar con una versión del código secuencial** para obtener medidas de **ganancia o aceleración**.
- La ganancia está relacionada con el **tiempo secuencial y paralelo** de un algoritmo.
- La ganancia determina la **eficiencia**.
- Nuestro objetivo es siempre lograr la **máxima eficiencia**.



8.2 Escalabilidad

- Se dice que **un algoritmo es escalable**, si al aumentar el tamaño del problema que resuelve, la **eficiencia se mantiene constante**.
 - Si aumentamos el tamaño del problema, normalmente tendremos más procesos y la eficiencia suele caer.
- Nuestro objetivo será **buscar una ganancia lineal** (o súper-lineal si es posible) para lograr una **eficiencia constante**.



8.3 Latencia: wall-clock

- Debemos medir el **tiempo de ejecución necesario para llevar a cabo la tarea** que nos interesa.
- Por ejemplo, en una red, la latencia es la cantidad de tiempo que tarda un paquete en desplazarse de un origen a un destino.
- En nuestro caso **la latencia de un programa o tiempo de respuesta es el tiempo que tenemos que esperar para tener un resultado, contando solo el tiempo útil que el procesador ha estado trabajando en nuestro algoritmo.**



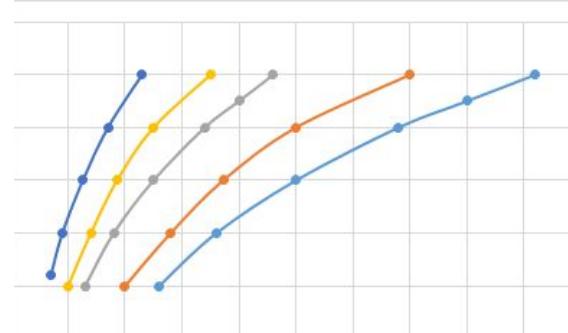
8.4 Energía consumida



- Cada vez se presta más **atención** también a la **energía consumida** en relación al trabajo realizado.
- A menor energía:
 - **Menor gasto en refrigeración** (y más tiempo de vida útil de los circuitos).
 - **Más sostenibilidad**
- El consumo energético en un supercomputador se mide a veces en el consumo por instrucción, el consumo por aplicación...

8.5 Rendimiento (Throughput)

- La potencia es la cantidad de trabajo que una máquina puede hacer por unidad de tiempo.
- Según se varíe la unidad de trabajo (instrucciones, procesos, hilos...), se generan diferentes medidas de rendimiento.
- La más conocida es **FLOPS**: Operaciones en coma flotante por unidad de tiempo. (Y sus múltiplos, como MFLOPS).
- Si se refieren a la **memoria**, será cantidad de **datos** (bytes, MB, GB) **por segundo**.
- Si se refieren a la **red** de interconexión, será la cantidad de **datos** que se pueden enviar por cantidad de tiempo.





8.5 Rendimiento (Throughput)

- El rendimiento se suele medir como la inversa de la latencia, pero esto solo es correcto si nos referimos a la misma cantidad de trabajo.
- Por ejemplo, en un proceso con una latencia de 1 segundo en completar 10^9 FLOPS, sí es correcto decir que su rendimiento es 10^9 FLOPS.
- Pero si hablamos de paquetes enviados a través de una red de interconexión, rendimiento y latencia no son inversamente proporcionales, pues afectan factores como la carga de la red.

8.6 Grado de concurrencia



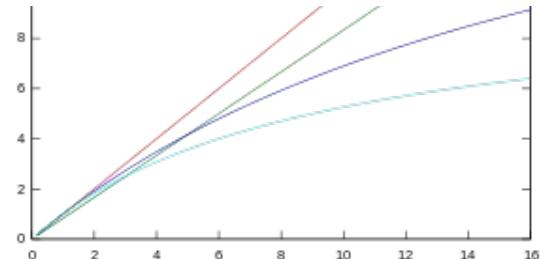
- Grado de concurrencia: **Cantidad de trabajo que se puede hacer de forma simultánea.**
- Las unidades de trabajo pueden ser: instrucciones, hilos, procesos...
- Aumentando la concurrencia se pueden mejorar otras métricas:
 - **Incrementa el rendimiento**, haciendo más trabajo en el mismo tiempo.
 - **Disminuye la latencia**, terminando un trabajo en menos tiempo.
 - **Oculta las latencias** necesarias en puntos de sincronización, en sistemas desbalanceados... pues mientras que unos hilos están ejecutándose, otros pueden estar esperando.

8.7 Peak Performance



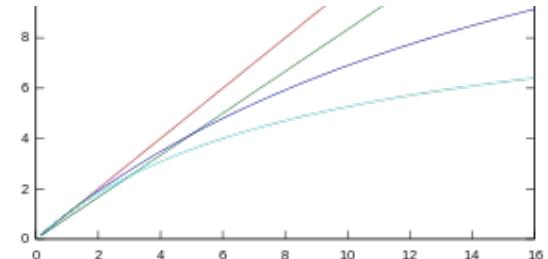
- El Rendimiento Pico o *Peak Performance* es la cantidad máxima de trabajo que puede realizar un sistema **por unidad de tiempo**, el máximo rendimiento.
- La diferencia entre ese máximo y el rendimiento de nuestro algoritmo, determina la eficiencia.
- Por ejemplo, la eficiencia en términos de *wall-clock* es la fracción de tiempo que un algoritmo ha estado realmente ejecutándose de forma útil. La eficiencia según el rendimiento de una máquina será una medida de cuánto tiempo útil ha estado ocupada.
- El rendimiento máximo es siempre un límite superior (a veces inalcanzable).

8.8 Ganancia y escalabilidad



- En HPC, normalmente se dispone de gran cantidad de recursos de computación. Si el algoritmo es escalable, el aumento de recursos no reducirá la eficiencia.
- El incremento de prestaciones según los recursos de computación utilizados en una ejecución, se llama ganancia o aceleración (*Speedup*).
- En otros términos, la ganancia es la relación entre la latencia antes de añadir un recurso y la latencia después de añadirlo.
- La ganancia debe ser siempre mayor que uno si el rendimiento aumenta.

8.8 Ganancia y escalabilidad



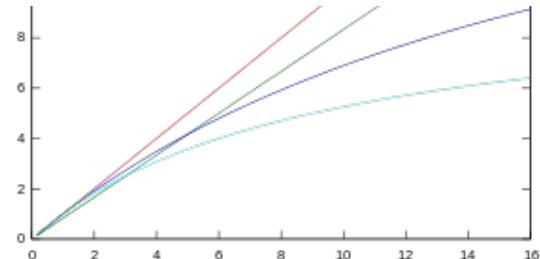
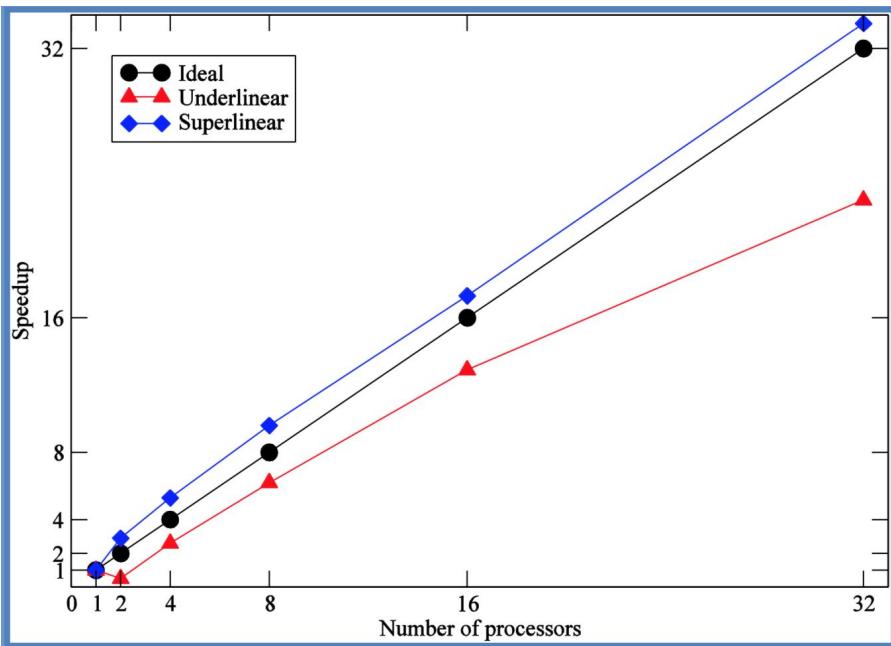
- La ganancia o aceleración (*speedup*) de un programa paralelo sobre N unidades de ejecución con respecto a su equivalente secuencial se calcula a partir de los tiempos de ejecución secuencial, T_p , y paralelo, T_s (en esa configuración):

$$S_N = \frac{T_s}{T_p}$$

- A partir de la aceleración, podemos calcular la eficiencia obtenida al usar N unidades de ejecución, E_N :

$$E_N = \frac{S_N}{N}$$

8.8 Ganancia y escalabilidad



¿Qué podría causar una aceleración superlineal?

- El efecto de la **caché** y la **RAM**
- El **algoritmo** (si la distribución de tareas permite evitar un número significativo de ellas, como en un Branch & Bound)



8.9 Profiling

- Es el proceso de **analizar un programa en ejecución** (análisis dinámico) **para medir** distintas magnitudes de interés sobre el mismo, como la cantidad de memoria usada, el número y tipo de instrucciones ejecutadas, o el tiempo usado por sus distintas funciones.
- Ejemplos de profiler son el **Intel VTune Profiler** o el Matlab Profiler.



8.9 Profiling

- Podemos distinguir los siguientes tipos de profilers:
 - **Según el resultado:**
 - Planos/Simples: Sólo dan el tiempo consumido por cada llamada
 - De grafo de llamadas: Además de tiempos, muestran el flujo de llamadas
 - Para conjunto de entradas: Estudian cómo varía el rendimiento de un programa para un conjunto de entradas.
 - **Según la granularidad** de los datos (tamaño de las muestras del profiler):
 - Basados en eventos: Registran la frecuencia de sucesos como interrupciones o llamadas a funciones
 - Estadísticos: En lugar de seguir toda la ejecución, presta atención en ciertos momentos y los extrapolan.

8.10 Tracing

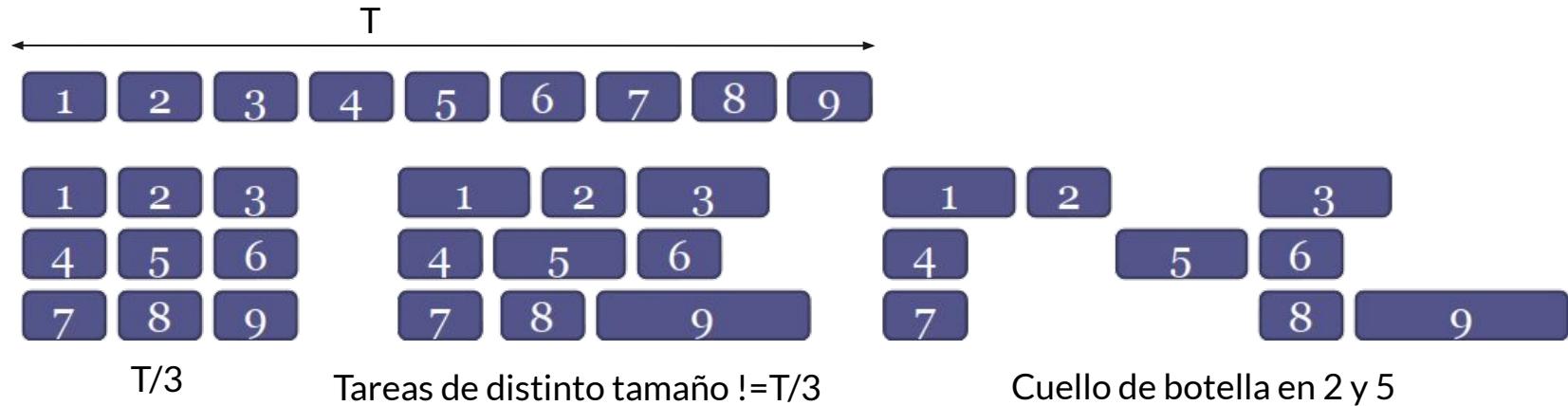
- El *tracing* es un **tipo de registro de eventos para obtener información precisa y a bajo nivel sobre la ejecución de un programa**. Se presta especial atención a los mensajes que se manda durante la ejecución para detectar errores en el **orden de aparición en los eventos**.
- Mientras que el profiling se centra más en detectar problemas de rendimiento (cuellos de botella, funciones mal diseñadas, desbalanceo de cargas... => **tiempos**), el tracing se enfoca en identificar problemas en la lógica o diseño (**orden y desencadenantes**).
- Un ejemplo de Tracer es VampirTrace

9. Limitación de prestaciones



9.1 Factores limitantes

Dado un trabajo T que tarda un tiempo TS en secuencial, y un conjunto de N unidades de procesamiento, al paralelizar T esperaríamos que el tiempo fuera TS / N . Sin embargo, esto difícilmente ocurrirá, pues al dividir el trabajo en tareas, no todas tardan lo mismo, y unas dependen de otras. Además, la propia paralelización implica tareas nuevas (*overhead*).



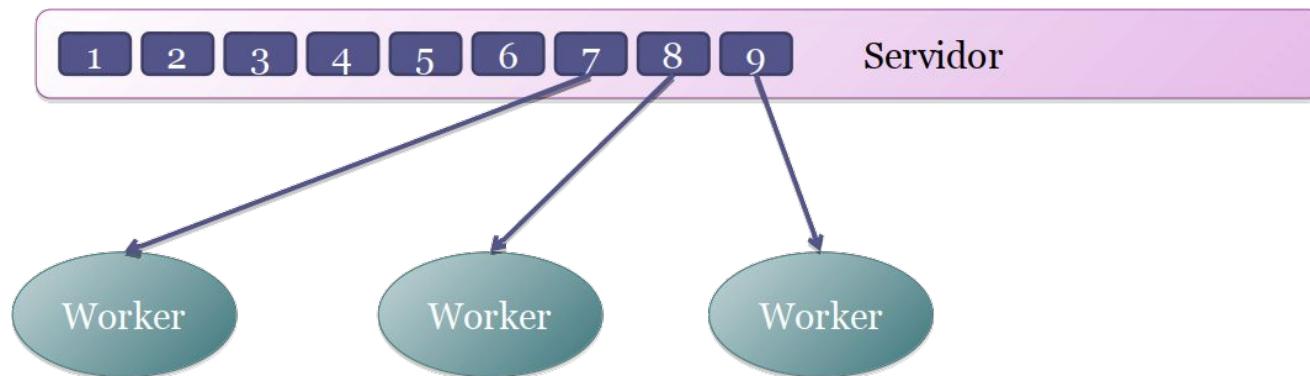
9.1 Factores que limitan

- Limitaciones por dependencias del algoritmo.
- Limitaciones por sincronizaciones.
- Limitaciones por cuellos de botella por el acceso a recursos compartidos.
- *Startup Overheads.*



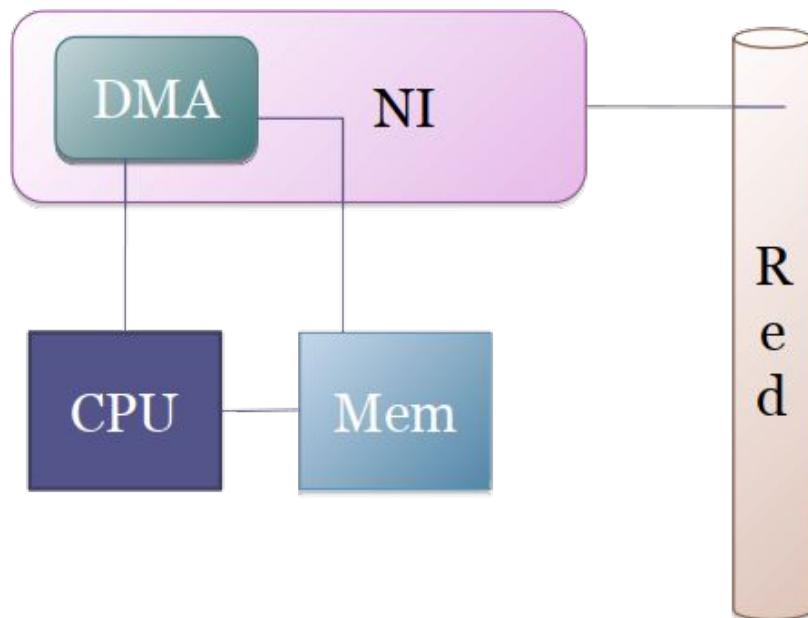
9.2 Limitaciones del algoritmo

- Ejemplo de reparto inicial de tareas entre unidades de procesamiento:

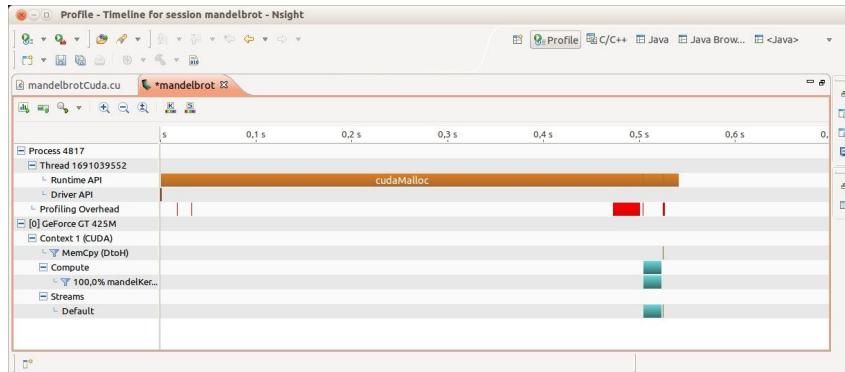


9.3 Acceso a recursos compartidos

- Ejemplo de comunicaciones por paso de mensajes con una sola interfaz de red:
- Estar accediendo siempre a la misma zona de la memoria (lo que implica sincronización y fallos de caché...)
- ...

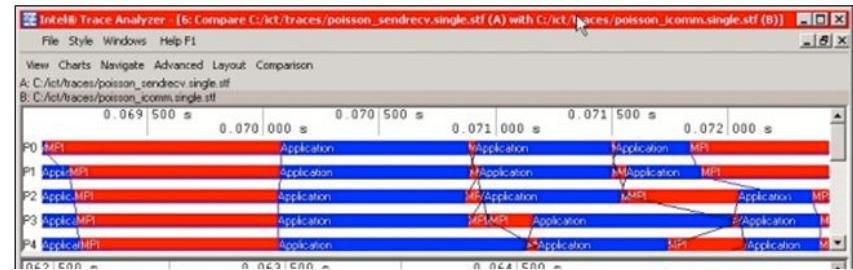


9.4 Startup Overheads



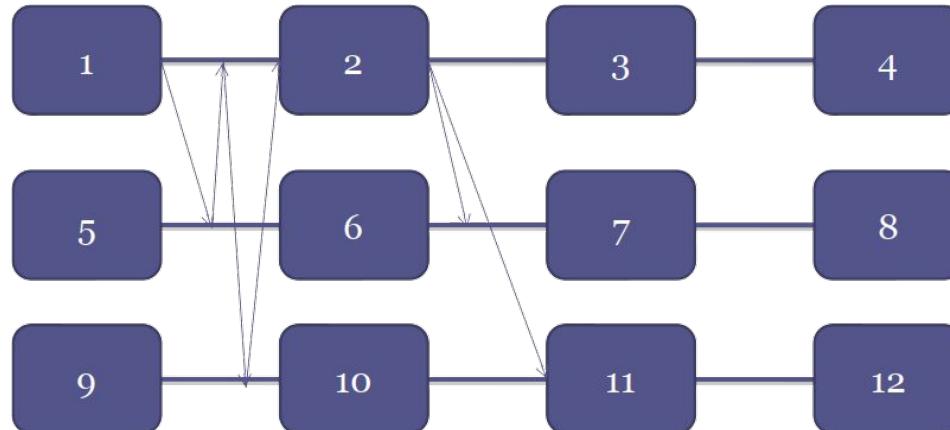
Profiling para CUDA

Profiling para MPI



9.5 Comunicaciones

- A cierto nivel, las comunicaciones siempre son secuenciales:



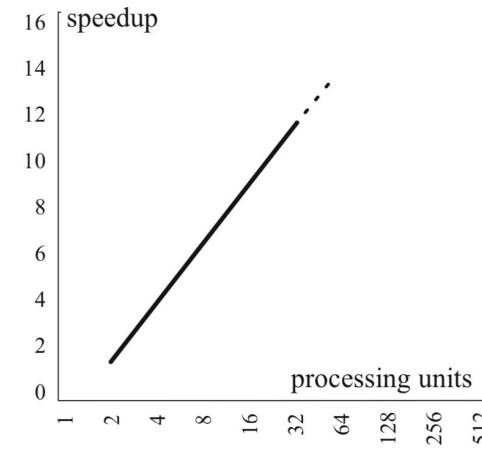
10. Ley de Amdahl y Ley de Gustafson



10.1 Ley de Amdahl

- Si doblamos el número de procesadores disponibles para un trabajo dado, esperamos que el tiempo sea la mitad. Si volvemos a doblar los procesadores, confiamos en que el tiempo anterior vuelva a ser la mitad del anterior... Es decir, **esperamos una aceleración lineal con el número de procesadores:**

+ Lo que esperamos:



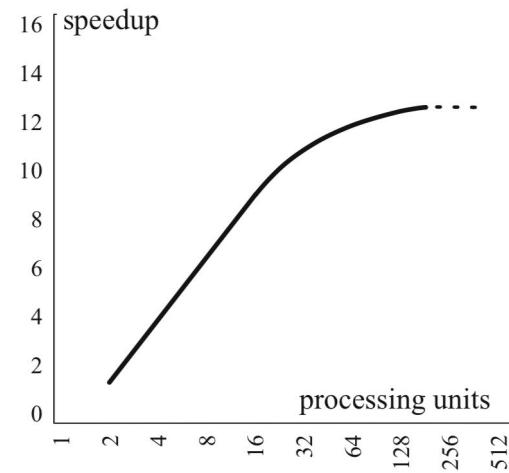
10.1 Ley de Amdahl

- Sin embargo, pocas veces veremos ese comportamiento. En su lugar, veremos una aceleración casi lineal con pocos procesadores y plana a partir de cierto número:

+ Lo que obtenemos:

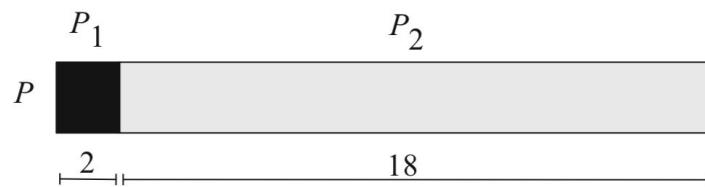


¿Por qué?



10.1 Ley de Amdahl

- Imaginemos un programa P con 2 partes, P_1 y P_2 tal que $P = P_1 \cup P_2$
 - P_1 es una tarea secuencial pura, como analizar un directorio y hacer un listado de archivos. (2 min)
 - P_2 es una tarea paralelizable, como aplicar una función sobre cada archivo. (18 min)
- En secuencial, el tiempo de P puede mostrarse como la suma del tiempo de cada parte:



10.1 Ley de Amdahl

- Sólo P_2 puede beneficiarse de añadir procesadores, por lo que, aunque ese coste fuera prácticamente cero (logrando muy buena aceleración), **el tiempo de P nunca va a ser menor que el de la parte secuencial de P_1 .**
- Además, la propia P_2 no tiene por qué ser totalmente paralelizable, manteniendo entonces una parte secuencial.

=> La aceleración a la que podemos aspirar al mejorar una parte de un sistema está limitada por el impacto temporal de dicha parte mejorada. Por tanto, nuestra situación sería bastante peor si los tiempos de P_1 y P_2 fueran al revés (y deberíamos cambiar el enfoque).

10.1 Ley de Amdahl

- En general, podemos ver todo programa P como divisible en dos grandes partes P_1 y P_2 , siendo la primera totalmente secuencial (no “ve” más de un procesador), y la segunda susceptible de paralelizarse. El tiempo de ejecución en paralelo (y la consiguiente aceleración) se ve limitado por P_1 y las partes secuenciales que quedan en P_2 .
- En este contexto, vamos a intentar calcular la aceleración formalmente sobre un programa $P = P_1 \cup P_2$. Su tiempo de ejecución secuencial, T_{seq} , viene dado por:

$$T_{\text{seq}}(P) = T_{\text{seq}}(P_1) + T_{\text{seq}}(P_2)$$

10.1 Ley de Amdahl

- Si empleamos más de un procesador para ejecutar P, su parte P_2 se verá acelerada por un factor $s > 1$ (mientras que P_1 seguirá igual). De esta forma, el tiempo de ejecución paralelo de P, T_{par} , será:

$$T_{par}(P) = T_{seq}(P_1) + T_{par}(P_2) = T_{seq}(P_1) + \frac{1}{s} T_{seq}(P_2)$$

- La aceleración de P, $S(P)$, puede calcularse entonces como:

$$S(P) = \frac{T_{seq}(P)}{T_{par}(P)} = \frac{T_{seq}(P_1) + T_{seq}(P_2)}{T_{seq}(P_1) + \frac{1}{s} T_{seq}(P_2)}$$

10.1 Ley de Amdahl

- Llegados a este punto, podemos normalizar el tiempo según el tiempo secuencial de P como $T_{seq}(P) = T_{seq}(P_1) + T_{seq}(P_2) = 1$ (podemos verlo también como un 100%).
- Además, podemos expresar la parte paralelizable, P_2 , en términos de P_1 , como su complementaria: $P_2 = 1 - P_1$. Teniendo esto en cuenta sobre la expresión de aceleración previa:

$$S(P) = \frac{1}{T_{seq}(P_1) + \frac{1 - T_{seq}(P_1)}{s}}$$

10.1 Ley de Amdahl

- Podemos además reescribir la expresión anterior llamando seq al tiempo de la parte secuencial pura, $seq = T_{seq}(P_1)$ y N al número de procesadores empleado ($s=N$) y con el que asumimos una aceleración s . Esta es la formulación que se suele ver:

$$S(P) = \frac{1}{seq + \frac{1 - seq}{N}}$$

Ley de Amdahl, 1967

- De aquí se deduce la parte secuencial limita la aceleración aunque $N=>\infty$: $S(P) < \frac{1}{seq}$

10.1 Ley de Amdahl

- Por consiguiente, la aceleración tiende a $1/\text{seq}$ para infinitos procesadores y llega un punto en el que se deja de obtener aceleración (y **la eficiencia tiende a cero**).
- Sólo en el caso ideal cuando $\text{seq}=0$ se tiene que $S = N$ (y $E = 1$).
- Por ejemplo, supongamos que el 70% de un programa puede acelerarse mediante parallelización y disponemos de 16 procesadores:

$$S = \frac{1}{0.3 + \frac{0.7}{16}} = 2.91$$

Para 32 procesadores, $S = 3.11$. Para 64 procesadores, $S = 3.22$. Para 128, $S = 3.27\dots$:-)

10.1 Ley de Amdahl

- Es importante tener en cuenta que únicamente estamos hablando de aumentar procesadores, por lo que la carga de trabajo (el coste computacional de P) se asume fija en este modelo.
- En este contexto, se dice que un algoritmo es fuertemente escalable (escalabilidad fuerte) si se puede acelerar mediante la adición de procesadores.

10.2 Ley de Gustafson

- La Ley de Amdahl da una visión pesimista (exigente) de la utilidad del procesamiento paralelo, pues se centra en cómo la parte secuencial fija limita la máxima aceleración.
- Sin embargo, **no considera** que, al disponer de más recursos computacionales, podamos plantearnos abordar problemas más grandes (**variando la carga**), situación que puede darse:
 - Una estimación de Pi indefinidamente precisa
 - Un algoritmo genético con una población más grande...
 - ...
- Metáfora de la conducción: https://es.wikipedia.org/wiki/Ley_de_Gustafson

10.2 Ley de Gustafson

- De hecho, muchas veces nos plantearemos **fijar el tiempo** más que el tamaño **computacional** del problema (entrada):
 - *¿Cuánto tiempo podemos permitirnos esperar un resultado lo más preciso posible?*
- Pensemos en una situación alternativa a la anterior en la que **fijamos el tiempo** y pretendemos **maximizar el trabajo** que hacemos con N procesadores. Descomponiendo el proceso P en su parte secuencial, **seq**, y la paralelizable, **par**:
 - El tiempo en paralelo será: seq + par = 1 (Normalización)
 - Sin paralelismo, el trabajo realizado sería seq + par*N

$$S(P) = \frac{T_{seq}}{T_{par}} = \frac{seq + par * N}{1}$$

10.2 Ley de Gustafson

- En este contexto de carga variable y tiempo fijo se define la **Ley de Gustafson**:

$$S(P) = \text{seq} + N^*\text{par} = \text{seq} + N^*(1-\text{seq})$$

Ley de Gustafson, 1988

- Por tanto, aunque el código tenga una parte secuencial, **la aceleración** no tiene por qué estar limitada por ella, y **puede ser proporcional a N**.
- Se dice que un algoritmo es **débilmente escalable** (escalabilidad débil) si se puede aumentar su aceleración incrementando el tamaño del problema.

10.3 Corolario

- Ambas leyes son complementarias.
- Debe aplicarse una u otra **según el contexto** en el que nos encontremos:
 - No siempre tiene sentido optar por un problema más grande.
 - A veces necesitamos reducir el tiempo que esperamos “a toda costa” (latencia).
- Otras veces sí: tenemos que hacer lo máximo posible en un tiempo fijado...

10.3 Corolario

- De hecho, ambas leyes se pueden aunar usando una variable auxiliar, $\alpha (\alpha > 0)$:

$$S(P) = \frac{seq + parN^\alpha}{seq + parN^{\alpha-1}} = \frac{seq + (1 - seq)N^\alpha}{seq + (1 - seq)N^{\alpha-1}}$$

Si $\alpha=0$: Ley de Amdahl

$$\frac{seq + parN^0}{seq + \frac{par}{N}} = \frac{1}{seq + \frac{(1 - seq)}{N}}$$

Si $0 < \alpha < 1$ y $N \gg 1$:

$$S(P) = 1 + \frac{par}{seq} N^\alpha$$

Barrera de Amdahl superada:
S(P) proporcional a N

Si $\alpha=1$: Ley de Gustafson

$$\frac{seq + (1 - seq)N}{seq + (1 - seq)N^0} = seq + (1 - seq)N$$

10.3 Corolario

- En cualquier caso, hay que tener en cuenta que se definen en **condiciones ideales** (como igualar el tiempo paralelo al secuencial dividido por el número de procesadores). Sin embargo, al **parallelizar** se esperan:

- Coste de creación/inicialización
- Comunicación
- Sincronización
- Desbalanceo de carga



OVERHEAD, que se reduce maximizando la relación cómputo/comunicación, evitando sincronismo y balanceando la carga

10.4 Ejercicio

Un programa se ejecuta en 40 segundos en un sistema con múltiples procesadores. Un 20% del tiempo ha usado 4 de los procesadores y un 60% del tiempo ha utilizado 3. El 20% restante su ejecución ha sido en un único procesador. Asumimos que la carga se balancea de forma equitativa en todo momento y obviamos el *overhead*. ¿Cuánto tiempo tardaría en ejecutarse en un solo procesador? ¿Qué aceleración estamos obteniendo? ¿Y qué eficiencia?

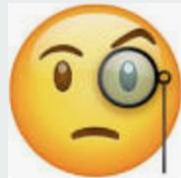
11 Mejorando prestaciones

Cosas básicas que deberíamos hacer y no siempre hacemos

- Compilar con opciones de optimización -O2, -O3...
- No utilizar variables innecesarias o tenerlas declaradas desde el principio para usarlas solo en una parte pequeña del código.
- Las variables globales admiten tamaños mayores que las dinámicas o las locales
- Usar operaciones vectoriales
- Desenrollar bucles
- Generar iteraciones en bucles independientes

11 Mejorando prestaciones

Cosas básicas que deberíamos hacer y no siempre hacemos



- Utilizar funciones inline (Inlining)
- Alineación de variables en memoria (Aliasing)
- Utilización de registros para datos muy utilizados (register)
- Utilizar variables de precisión adecuada
- Optimización de accesos a memoria evitando fallos de página

En cualquier caso, no olvidemos estas dos máximas:

- *First, make it work. Then, make it fast.*
- *Premature optimization is the root of evil!*

11 Mejorando prestaciones

No repetir trabajo

```
1 Logical::: FLAG
2 FLAG = false
3 Do i=1, N{
4     If(complex(A(i)) < Threshold){
5         FLAG = true
6     }
7 }
```

```
1 Logical::: FLAG
2 FLAG = false
3 Do i=1, N{
4     If(complex(A(i)) < Threshold){
5         FLAG = true
6         Exit()
7     }
8 }
```

11 Mejorando prestaciones

Optimizar/aproximar cálculos, y evitar llamadas a función

`Y= Pow(x,2);`

Problema

`Integer::iL,iR,iU,iO,iS,iN`

`Double precision::edelz,tt`

`Loop`

`...`

`edelz=iL+iR+iU+iO+iS+iN`

`BF=0.5d0*(1.d0+THNH(edelz/tt));`

`Y= x*x;`

Solución

```
Double precision, dimension(-6:6)::tanh_table
Integer::iL,iR,iU,iO,iS,iN
Double precision::tt
do i=-6,6
    tanh_table(i) = 0.5d0*(1.d0+THNH(dble(i)/tt));
enddo
Loop
...
BF=tanh_table(iL+iR+iU+iO+iS+iN);
```

- Si el conjunto de variables activas es mayor que la caché de mayor nivel => fallos de página
- Si nos basta 1 byte, no usar variables que ocupen 4 (especialmente si la máquina direcciona así)...
- Si es un cuadrado, podemos hacerlo simplemente multiplicando, y evitamos una función externa
- Senos, cosenos, tangentes... según el caso podemos usar aproximaciones (p.e.j. polinomios de Taylor) y/o pre-calcular y almacenar el rango que necesitamos para ahorrar cálculos.

11 Mejorando prestaciones

Almacenar cálculos recurrentes

```
Do i=1,N  
    A(i)=A(i)+s+r*sin(x)  
enddo
```

```
tmp=s+r*sin(x)  
Do i=1,N  
    A(i)=A(i)+tmp  
enddo
```

11 Mejorando prestaciones

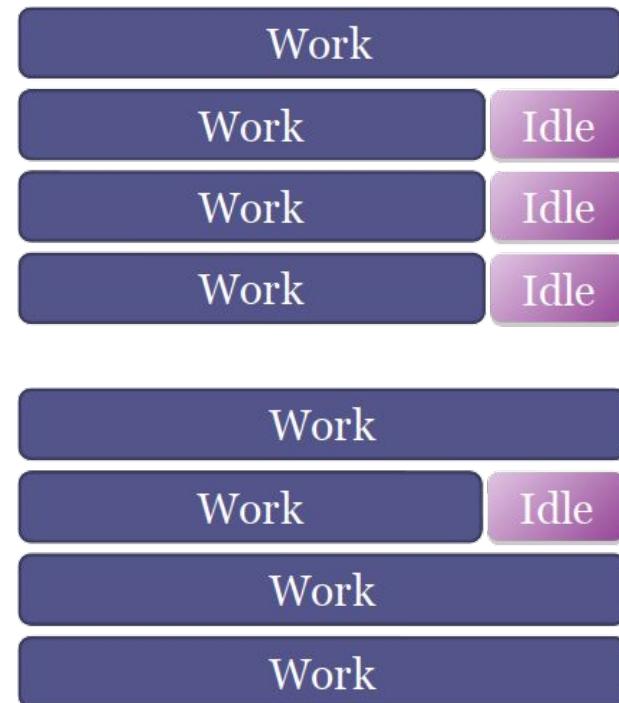
```
do j=1,N
    do i=1,N
        if(i>=j) then
            sign=1;
        else if(i<j)then
            sign=-1;
        else
            sign=0;
        endif
        C(j)=C(j)+sign*A(i,j)*B(i)
    enddo
enddo
```

```
do j=1,N
    do i=j+1,N
        C(j)=C(j)+A(i,j)*B(i)
    enddo
enddo
do j=1,N
    do i=1,j-1
        C(j)=C(j)-A(i,j)*B(i)
    enddo
enddo
```

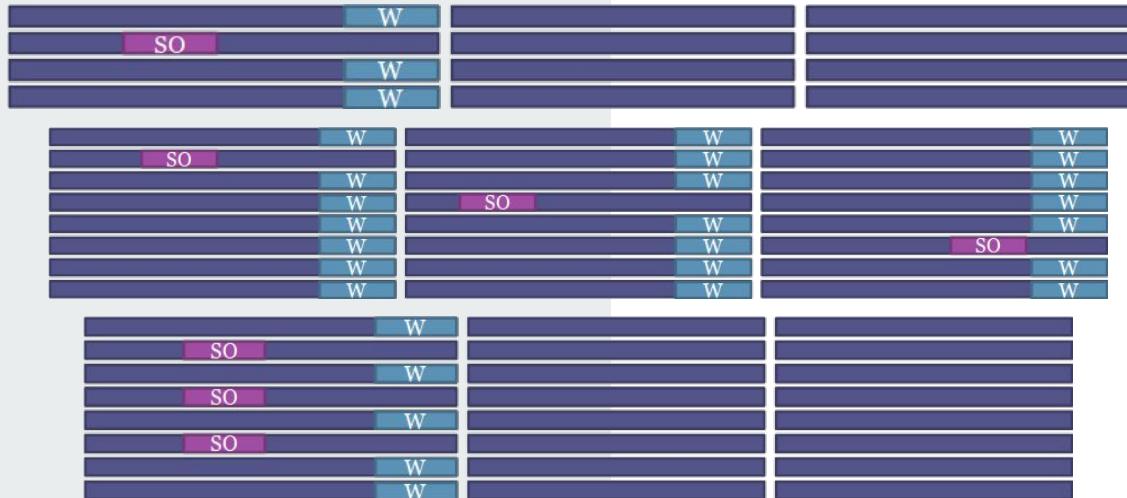
11 Mejorando prestaciones

Profiling y Tracing para buscar

- Cuellos de botella
- Pérdidas de tiempo en sincronización
- Sobrecarga en las comunicaciones
- ...



11 Mejorando prestaciones



- Si la cantidad de trabajo a parallelizar es dinámica, lo que puede dar lugar a desbalanceos
- La granularidad puede ser inadecuada
- Las unidades funcionales específicas, como multiplicadores, están ocupadas
- Acciones del SO (Jitter)



Gracias.





Tema 2

Modelos de programación paralela adaptados a la arquitectura

Nicolás Calvo Cruz

Dpto. de Arquitectura y Tecnología de los Computadores

@ncalvocruz

ncalvocruz@ugr.es

Objetivos

- Aprender a **encontrar** puntos para paralelizar un algoritmo
- Aprender a **descomponer** un algoritmo en tareas
- Aprender a identificar y respetar **dependencias** entre tareas
- **Agrupar** las tareas que se pueden realizar en paralelo
- Aplicar revisiones de **diseño**



Motivación

- 1 Abordar **problemas** cada vez más grandes.
- 2 Obtener (mejores) soluciones en un **tiempo razonable**.
- 3 Ilustrar diferentes **enfoques** de la bibliografía para **parallelizar** algoritmos tradicionales.

Índice

1. Introducción
2. ¿Cómo encontrar concurrencia?
 - a. Encontrar tareas
 - b. Agrupar tareas
 - c. Buscar patrones de comunicación
3. Patrones de los principales algoritmos paralelos
4. Estructuras de algoritmos más comunes
5. Qué hacer con las estructuras de datos
6. Ejemplos prácticos de algoritmos para arquitecturas de memoria compartida



1. Introducción

1. Introducción

Los pasos básicos para parallelizar un algoritmo son:

1. Encontrar la **conurrencia** del algoritmo de partida.
2. Decidir la **estructura** del algoritmo.
3. Dar **soporte a las estructuras de datos** necesarias para la implementación.
4. **Implementación** según la metodología deseada.



1. Introducción

Los pasos a seguir y cómo desarrollarlos, dependen del Dominio de problema.



2. Cómo encontrar concurrencia



2. Cómo encontrar concurrencia

Primeras cuestiones

- ¿Es paralelizar la opción adecuada? (P. Ej., Búsqueda Binaria)
- Control del Dominio del Problema
- Identificación de las partes más costosas computacionalmente (**Profiling**)
- Pasos a seguir para encontrar concurrencia
 - Descomposición del problema
 - Análisis de dependencias
 - Evaluación del diseño elegido

2. Cómo encontrar concurrencia

Pasos a seguir

Descomposición del Problema:

1. Descomposición de datos
2. Descomposición de tareas
3. Descomposición por flujo

Flexibilidad

Eficiencia

Simplicidad

Análisis de Dependencias

1. Agrupar tareas
2. Ordenar tareas
3. Compartir datos

Evaluación de la solución

2. Cómo encontrar concurrencia

1. Descomposición del problema según sus datos



2. Cómo encontrar concurrencia

1. Descomposición del problema según sus datos

- La descomposición de datos **requiere** que el algoritmo use **una estructura de datos divisible** entre unidades de ejecución (hilos/procesos) concurrentes.
- Paralelizar en este caso es hacer que **cada unidad de ejecución trabaje con una parte** de los datos.
- Debe haber, al menos, tantas porciones como unidades de ejecución.
- Este enfoque sólo es eficiente si las **operaciones sobre la estructura de datos** son relativamente **independientes**.

2. Cómo encontrar concurrencia

1. Descomposición del problema según sus datos

- Este enfoque es adecuado si:
 - La mayor parte computacional del algoritmo se dedica a gestionar la estructura de datos (matrices, vectores, grafos...)
 - Las operaciones en cada parte son iguales y se pueden hacer independientemente.
- A veces, la descomposición de datos, genera una descomposición de tareas.

2. Cómo encontrar concurrencia

1. Descomposición del problema según sus tareas



2. Cómo encontrar concurrencia

1. Descomposición del problema según sus tareas

- El algoritmo se ve como un flujo de instrucciones que podrían ejecutarse simultáneamente y que condicionarán el proceso general del algoritmo.
- La pregunta es: ¿qué operaciones se pueden realizar de forma simultánea e independiente?
- En un equipo, esta fase se aborda de forma individual para obtener varias opciones posibles y elegir la mejor.

2. Cómo encontrar concurrencia

1. Descomposición del problema según sus tareas

- Las tareas podrían ser:
 - Llamadas a función (functional decomposition)
 - Iteraciones de bucles (loop-split)
 - Tareas que reparten datos a las diferentes unidades de ejecución y los actualizan
- Debe cumplirse:
 - Que compense crear y destruir (gestionar) esa tarea
 - Que sean tareas que se puedan adaptar al crecimiento de las unidades de ejecución, siendo siempre mayor o igual a ese número (escalabilidad)

2. Cómo encontrar concurrencia

1. Descomposición por flujo de datos



2. Cómo encontrar concurrencia

1. Descomposición por flujo de datos

- No siempre se destaca, pero como **condiciona el siguiente paso** (la elección de la estructura del algoritmo), **hay que tenerlo en cuenta** en esta primera fase también.
- Es la opción **adecuada cuando el flujo de los datos es el que ordena los grupos de tareas estableciendo un orden entre ellas**.
- Es el enfoque que **se sigue en todos los procesadores comerciales** en el pipeline, **gestionando la paralelización de instrucciones**.
- Es el enfoque **menos utilizado**, aunque **hay problemas que lo requieren**, como los **algoritmos de compresión**.

2. Cómo encontrar concurrencia

1. Descomposición del problema: Consideraciones

- Flexibilidad: la partición de tareas o datos debe poder adaptarse a un número variable de unidades de ejecución.
- Eficiencia: al decidir sobre partición de tareas y datos, hay que pensar en la máquina escogida, la **granularidad**, y el **rendimiento** que queremos obtener.
- Simplicidad: el punto de partida debe ser el **más sencillo posible**, así las tareas deben ser las más básicas y la partición de datos la más pequeña posible para cada hebra o proceso.

2. Cómo encontrar concurrencia

1. Descomposición del problema: Ejemplos



2. Cómo encontrar concurrencia

1. Descomposición del problema: Ejemplos

Imágenes médicas PET

- La *Positron Emission Tomography* (PET) produce **imágenes** que muestran cómo se propaga una sustancia radioactiva por el paciente para diagnosticar enfermedades.
- Las imágenes son de baja calidad, y el cuerpo del paciente falsea su intensidad.
- Para solucionarlo: Se modela un cuerpo humano con el que corregir las imágenes crudas. Cada zona del cuerpo emitirá una radiación y se simulará la trayectoria de cada partícula.
- Este proceso se puede **parallelizar con partición de datos o de tareas...**

2. Cómo encontrar concurrencia

1. Descomposición del problema: Ejemplos

Imágenes médicas PET

- Paralelización de datos:
 - Se reparte el cuerpo en partes de forma que la simulación de las trayectorias que atraviesan cada porción se asignan a una unidad de ejecución.
- Paralelización de tareas:
 - Las unidades de ejecución realizan la simulación de trayectorias completas de partículas, que son independientes entre sí.
- ¿Pros y contras de cada una?

2. Cómo encontrar concurrencia

1. Descomposición del problema: Ejemplos

Imágenes médicas PET

- Paralelización de datos: La estructura de datos principal es el **modelo del cuerpo**.
 - Debe **partirse en segmentos**.
 - Los segmentos se pueden agrupar según el número de bloques deseado.
 - No hay dependencias de datos porque la estructura es de sólo lectura.
- La partición de tareas asociadas a la partición de datos será la simulación de las trayectorias que se emitan en cada uno de los segmentos creados. **Si llega al límite** del segmento, ha de **pasarse al siguiente**:
 - En **Memoria Compartida** pasar una trayectoria de una zona a otra implicará sincronización.
 - En **Memoria Distribuida** habrá que trasladar trayectorias antes y durante...

2. Cómo encontrar concurrencia

1. Descomposición del problema: Ejemplos

Imágenes médicas PET

- Paralelización de tareas:
 - Se selecciona un punto del cuerpo y se golpea con una partícula radioactiva.
 - Se sigue la trayectoria de dicha partícula.
 - Este proceso se repite con miles de partículas.
- Esta descomposición genera un **alto grado de paralelismo**, lo que es bueno para su implementación en cualquier máquina paralela:
 - En **Memoria Compartida** se almacena en memoria el modelo del cuerpo, y los hilos acceden a él **sin sincronización**.
 - En **Memoria Distribuida**, el coste de hacer llegar a todos los nodos el modelo del cuerpo puede ser problemático.

2. Cómo encontrar concurrencia

1. Descomposición del problema: Ejemplos

Producto de matrices

$$\begin{matrix} A & * & B & = & C \\ \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} & * & \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix} & = & \begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{pmatrix} \end{matrix}$$

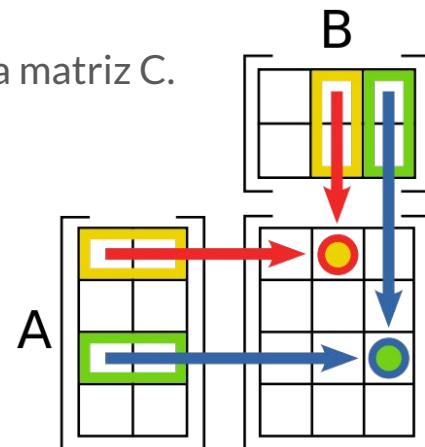
$$c_{1,1} = (a_{1,1} * b_{1,1} + a_{1,2} * b_{2,1} + a_{1,3} * b_{3,1}) \\ (\dots)$$

$$c_{3,3} = (a_{3,1} * b_{1,3} + a_{3,2} * b_{2,3} + a_{3,3} * b_{3,3})$$

2. Cómo encontrar concurrencia

1. Descomposición del problema: Ejemplos Producto de matrices

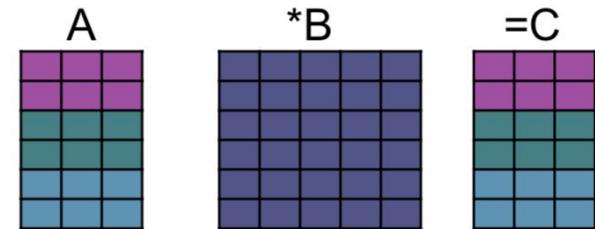
- Paralelización de datos:
 - Se parte la matriz A en bloques de filas y la matriz B en bloques de columnas
- Paralelización de tareas:
 - Cada tarea será generar un elemento completo de la matriz C.
- ¿Pros y contras?



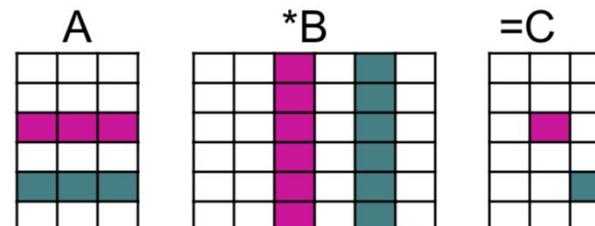
2. Cómo encontrar concurrencia

1. Descomposición del problema: Ejemplos Producto de matrices

- Existen diversas alternativas de reparto, como:
 - Descomponer las matrices A y C en grupos de filas:



- Descomponer la matriz C y, a partir de esta división, partir A y B:



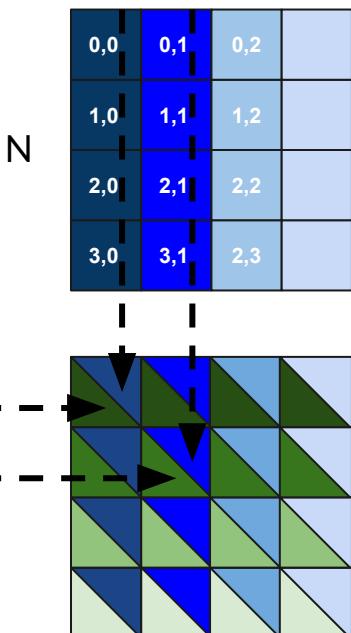
2. Cómo encontrar concurrencia

Producto de matrices

Cada componente se usa dos veces en cada Tile (2x2)

Tiling:

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

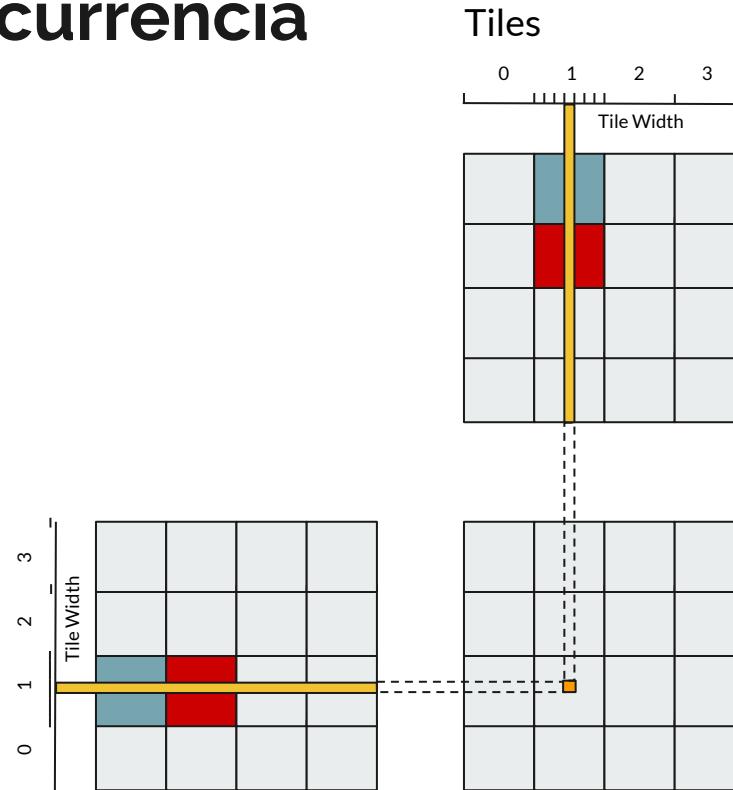


R(0,0)	R(1,0)	R(0,1)	R(1,1)
$M(0,0)*N(0,0)$	$M(1,0)*N(0,0)$	$M(0,0)*N(0,1)$	$M(1,0)*N(0,1)$
$M(0,1)*N(1,0)$	$M(1,1)*N(1,0)$	$M(0,1)*N(1,1)$	$M(1,1)*N(1,1)$
$M(0,2)*N(2,0)$	$M(1,2)*N(2,0)$	$M(0,2)*N(2,1)$	$M(1,2)*N(2,1)$
$M(0,3)*N(3,0)$	$M(1,3)*N(3,0)$	$M(0,3)*N(3,1)$	$M(1,3)*N(3,1)$

2. Cómo encontrar concurrencia

Producto de matrices

- Escalamos el problema y suponemos que cada cuadrado es una matriz de 4×4 , así estas matrices M y N serán de 16×16 .
- Se van calculando *tiles* parciales que luego se van sumando a los siguientes conforme aumenta el tile por el que recorremos la matriz
- ¿Partición por datos?
- ¿Partición por tareas?



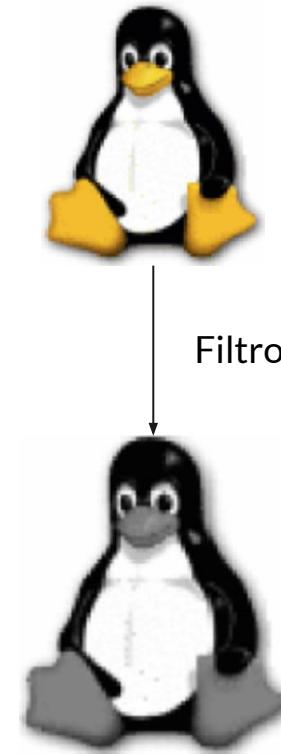
2. Cómo encontrar concurrencia

Procesamiento de imágenes

- Aplicación de un filtro a una imagen:

```
do j = 1 , Numero_Pixels_en_j  
    do i = 1, Numero_Pixels_en_i  
        Color = Imagen (i,j)  
        Imagen (i,j) = f(i,j, Color )  
    enddo  
enddo
```

- La aplicación de un filtro a un pixel no depende de los vecinos, es un **problema embarazosamente paralelo**.



2. Cómo encontrar concurrencia

Procesamiento de imágenes

- Descomposición equitativa de la imagen (datos):

```
j1 = Mi_Primera_Columna
```

```
j2 = Mi_Ultima_Columna
```

```
do j = j1 , j2
```

```
    do i = 1, Numero_Pixels_en_i
```

```
        Color = Imagen (i,j)
```

```
        Imagen (i,j) = f(i,j, Color )
```

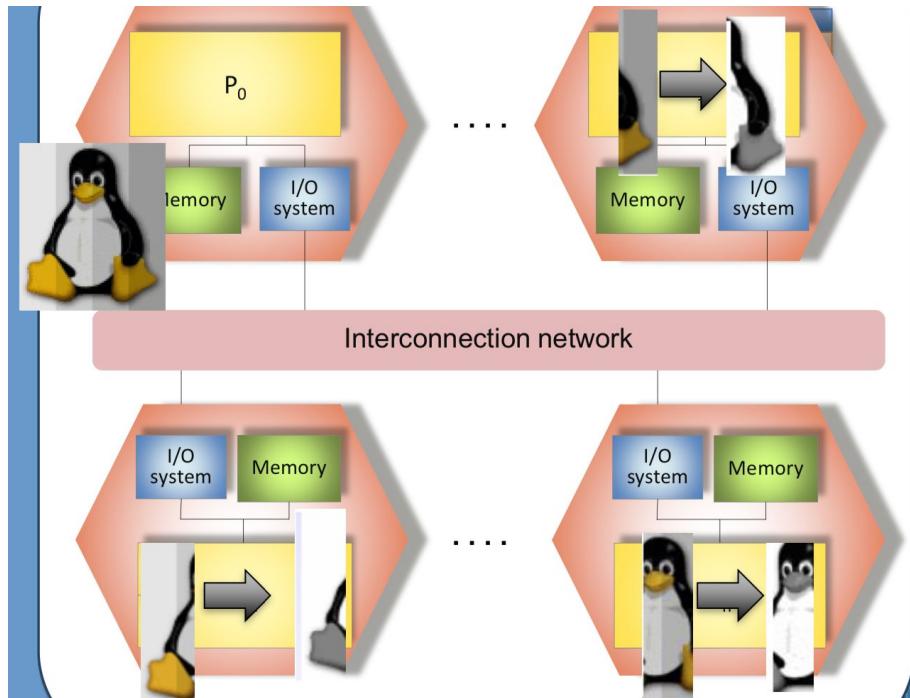
```
    enddo
```

```
enddo
```



2. Cómo encontrar concurrencia

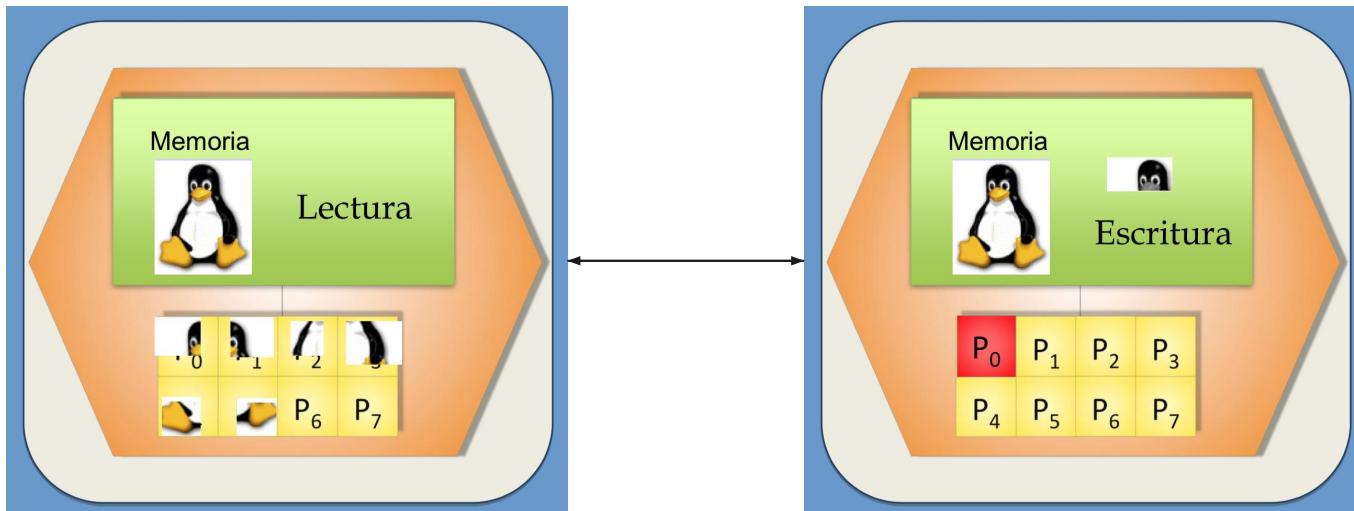
Procesamiento de imágenes



2. Cómo encontrar concurrencia

Procesamiento de imágenes

- Descomposición de tareas:
 - Cada tarea es el cálculo de un píxel resultado.



2. Cómo encontrar concurrencia Pasos a seguir (Recordatorio)

Descomposición del Problema

1. Descomposición de datos
2. Descomposición de tareas
3. Descomposición por flujo

Flexibilidad

Eficiencia

Simplicidad

Análisis de Dependencias

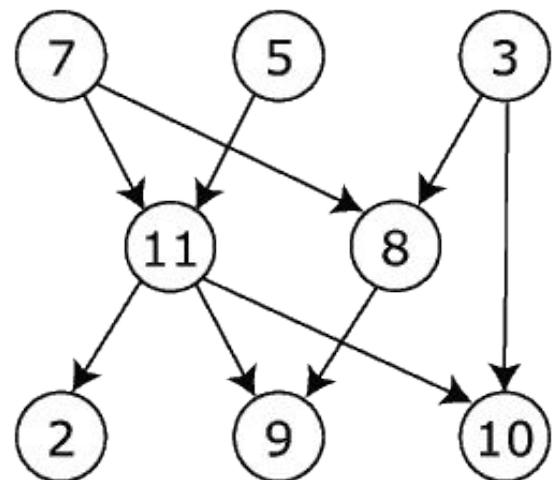
1. Agrupar tareas
2. Ordenar tareas
3. Compartir datos

Evaluación de la solución

2. Cómo encontrar concurrencia

2. Análisis de dependencias

- En esta fase hay que gestionar dependencias entre las partes de la **estructura de datos** que hayamos dividido o entre **las tareas** encontradas.
- Los pasos son:
 - a. **Agrupar** intentando minimizar dependencias
 - b. **Ordenar** los grupos de tareas estableciendo prioridades
 - c. **Buscar patrones** de comunicación



2. Cómo encontrar concurrencia

2. Análisis de dependencias: agrupar tareas

- ❑ La agrupación de tareas permite identificar grupos que pueden ejecutarse simultáneamente y establecer un orden parcial según restricciones comunes.
 - ❑ Todas las tareas dentro de un grupo tienen que ser independientes entre sí
- La agrupación puede ser:
- Jerarquía
 - Restricciones comunes

2. Cómo encontrar concurrencia:

2. Análisis de dependencias:

Producto de matrices (Agrupar tareas)

Tarea: Generar un valor de la matriz resultado, Generar el sumatorio de los resultados parciales;?

Agrupación: Todas pueden ser independientes y ejecutarse simultáneamente

Simplicidad / Granularidad: Podría crecer generando más de un valor por cada proceso o hebra

Eficiencia: Se podría ajustar al número de unidades de ejecución siendo igual al número de tiles. Se podría hacer una partición donde se reutilizara cada dato más de una vez

2. Cómo encontrar concurrencia:

2. Análisis de dependencias:

Procesado de imágenes

(Agrupar tareas)

Tarea: Calcular el píxel resultante de aplicar un filtro.

Agrupación: Todas pueden ser independientes y ejecutarse simultáneamente

Simplicidad / Granularidad: Podría crecer cómo se planifique el acceso a la imagen.

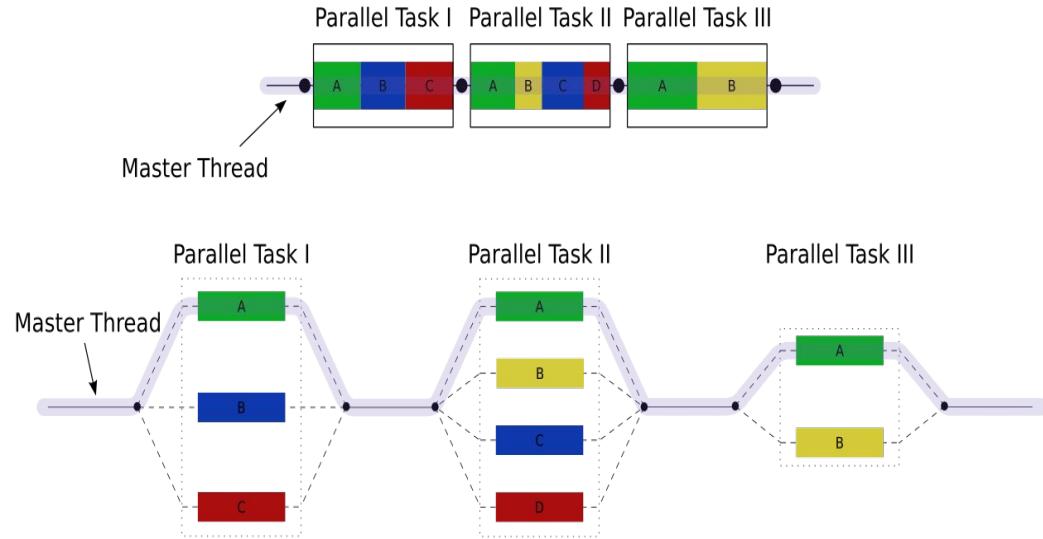
Eficiencia: Se podría ajustar al número de unidades de ejecución generando tantos grupos de tareas como unidades de ejecución.



2. Cómo encontrar concurrencia:

2. Análisis de dependencias (Ordenar tareas)

- Lo único a tener en cuenta en la ordenación de las tareas es **respetar las dependencias** y poder identificar posibles **puntos de sincronización obligatoria**.



2. Cómo encontrar concurrencia:

2. Análisis de dependencias: patrones de comunicación

- Esta fase consiste en identificar qué datos necesita cada una de nuestras tareas.
- Tengamos una paralelización por datos o por tareas, **todos los casos necesitarán un conjunto de datos con los que operar** y en eso debemos centrarnos.
- Los datos que cada tarea maneja se denominan *task-local* o *local-data*.

2. Cómo encontrar concurrencia:

2. Análisis de dependencias: patrones de comunicación

- Situaciones posibles:
 - Un conjunto de tareas que comparten el **mismo conjunto de datos**
 - Un conjunto de tareas que actualizan la **misma estructura de datos en colaboración**.
 - La misma situación anterior, pero con **dependencias entre los diferentes sectores por vecindades, etc.**

2. Cómo encontrar concurrencia:

2. Análisis de dependencias: patrones de comunicación

- Pasos a seguir:
 - Analizar qué datos o qué porción de datos necesita cada tarea y proponer la partición.
 - Identificar las operaciones que cada tarea realizará sobre sus datos y clasificar las estructuras de datos según:
 - Read only: Son de sólo lectura
 - Local: Será una porción de datos que sólo actualiza una tarea y no hay acceso compartido.
 - Read-Write: Si más de una tarea lee y escribe en la estructura, hay que establecer sincronización en los accesos de escritura

2. Cómo encontrar concurrencia:

2. Análisis de dependencias: patrones de comunicación

- Tipos de estructuras *Read-Write*:
 - Accumulative: Para realizar reducciones o acumulaciones de resultados parciales. En este caso, cada tarea posee una copia local de la estructura y opera con ella, hasta que hay que hacer la recopilación final en una sola estructura global.
 - Multiple-Read/Single-Write: Todas las tareas leen la estructura pero cada una escribe solo en su parte.

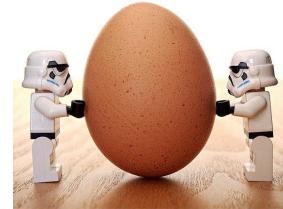
2. Cómo encontrar concurrencia: 3. Evaluación de la solución

- En este paso se revisarán las decisiones tomadas buscando posibles problemas que no se hayan tenido en cuenta, como intercambio de datos, datos en los bordes que se tendrán que duplicar, etc.
- Los puntos a abordar son:
 - Asegurarse de que cada tarea se puede realizar de forma simultánea.
 - Que hemos identificado correctamente los datos que pueden ser locales a cada tarea.
 - Que se satisfacen todas las dependencias temporales establecidas.

2. Cómo encontrar concurrencia: 3. Evaluación de la solución

- Siguiendo con las **comprobaciones**:
 - Que el enfoque de **partición** es adecuado para la máquina donde se ejecutará.
 - Que hemos respetado las reglas de **simplicidad, eficiencia y flexibilidad**.
 - Y ya es hora de plantearse otras cosas como:
 - ¿Las tareas son **regulares**? ¿realmente necesitamos **sincronización**?
¿Podríamos agrupar tareas de otra forma?

Trabajo para la próxima semana



Cada grupo debe escoger un problema computacionalmente costoso, explicarlo, y analizar cómo se podría paralelizar con una división por datos y otra por tareas.

Pensad en las estructuras de datos necesarias y de qué tipo sería su acceso.

=> Un miembro del grupo presentará su trabajo la próxima semana. Habrá que entregar tanto la presentación como una transcripción de lo que se dice (incluyendo las fuentes consultadas).

Importante: Poneos de acuerdo entre grupos para no escoger el mismo problema.



Gracias.





Tema 2

Modelos de programación paralela adaptados a la arquitectura

Nicolás Calvo Cruz

Dpto. de Arquitectura y Tecnología de los Computadores

@ncalvocruz

ncalvocruz@ugr.es



Índice

1. Introducción
2. ¿Cómo encontrar concurrencia?
 - a. Encontrar tareas
 - b. Agrupar tareas
 - c. Buscar patrones de comunicación
3. Patrones de los principales algoritmos paralelos
4. Estructuras de algoritmos más comunes
5. Qué hacer con las estructuras de datos
6. Ejemplos prácticos de algoritmos para arquitecturas de memoria compartida

3. Patrones de los principales algoritmos paralelos

3. Patrones de los principales algoritmos paralelos

Búsqueda de Concurrencia

Datos

Tareas

Flujo

Lineal

Recursiva

Lineal

Recursiva

Regular

Irregular

Geometric
Decompo
sition

Recursive
Data

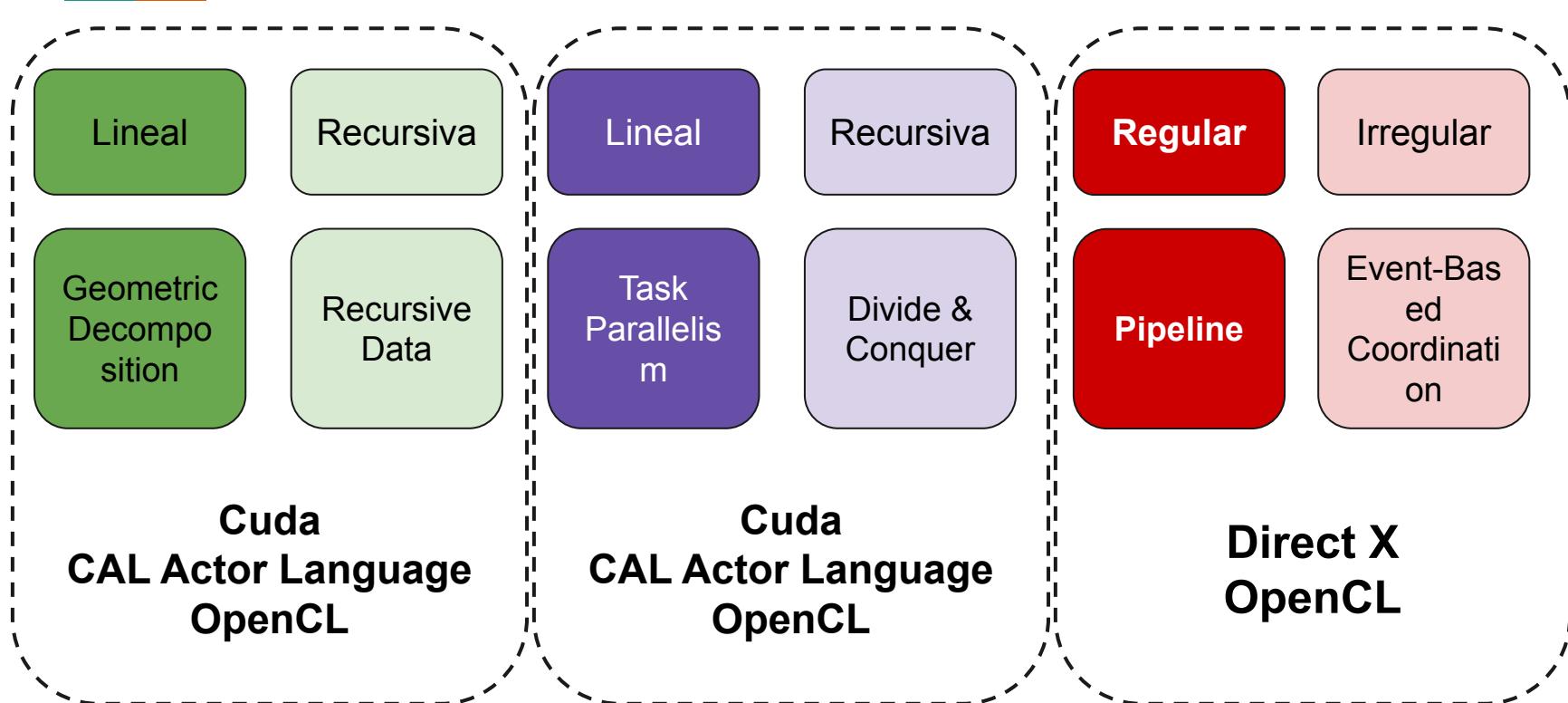
Task
Parallelis
m

Divide &
Conquer

Pipeline

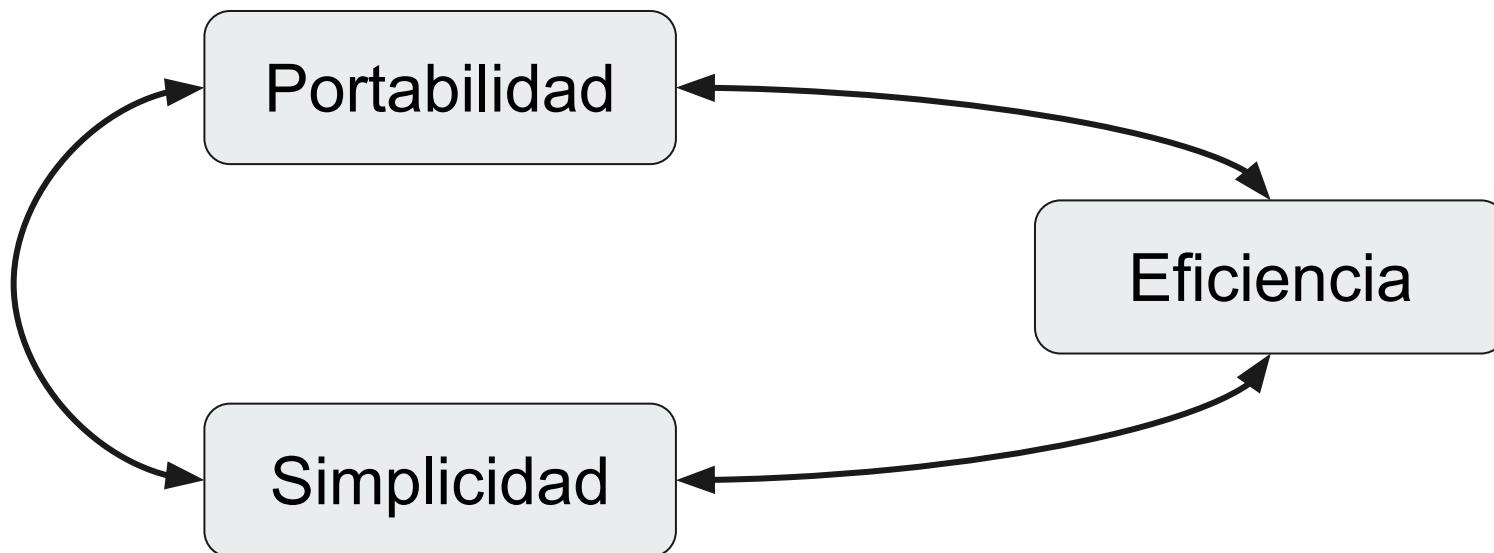
Event-Bas
ed
Coordinati
on

3. Patrones de los principales algoritmos paralelos



3. Patrones de los principales algoritmos paralelos

¿Cómo elegimos la mejor estructura?



3. Patrones de los principales algoritmos paralelos

¿Cómo elegimos la mejor estructura?

Orden de magnitud
de UE necesarias

¿Seguimos
siendo
flexibles?

¿Cuánto cuesta la
comunicación?

¿Qué
lenguajes
tenemos
disponibles?

3. Patrones de los principales algoritmos paralelos

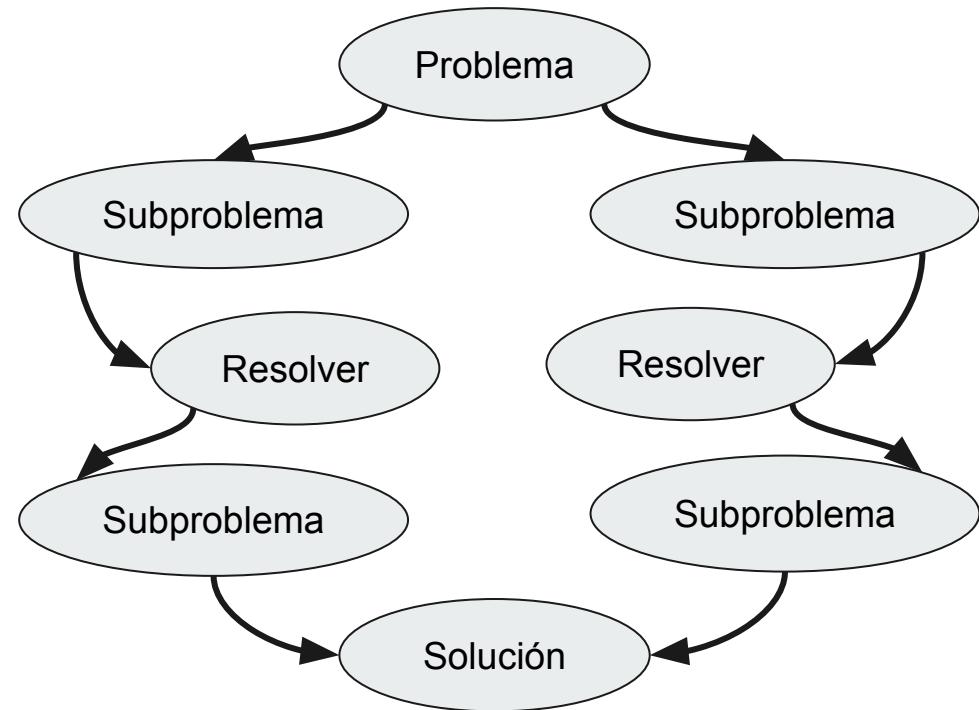
3.1 Descomposición por tareas

- Las tareas pueden ser:
 - Independientes (Ejemplo de procesado de imágenes)
 - Dependientes
 - Porque acceden a una estructura común
 - Porque tienen que coordinarse por mensajes
- Si las tareas se pueden organizar de forma recursiva: **Divide & Conquer**
- Si las tareas no son recursivas: **parallelismo de tareas general** (hebras o procesos) (*Task Parallelism*)

3. Patrones de los principales algoritmos paralelos

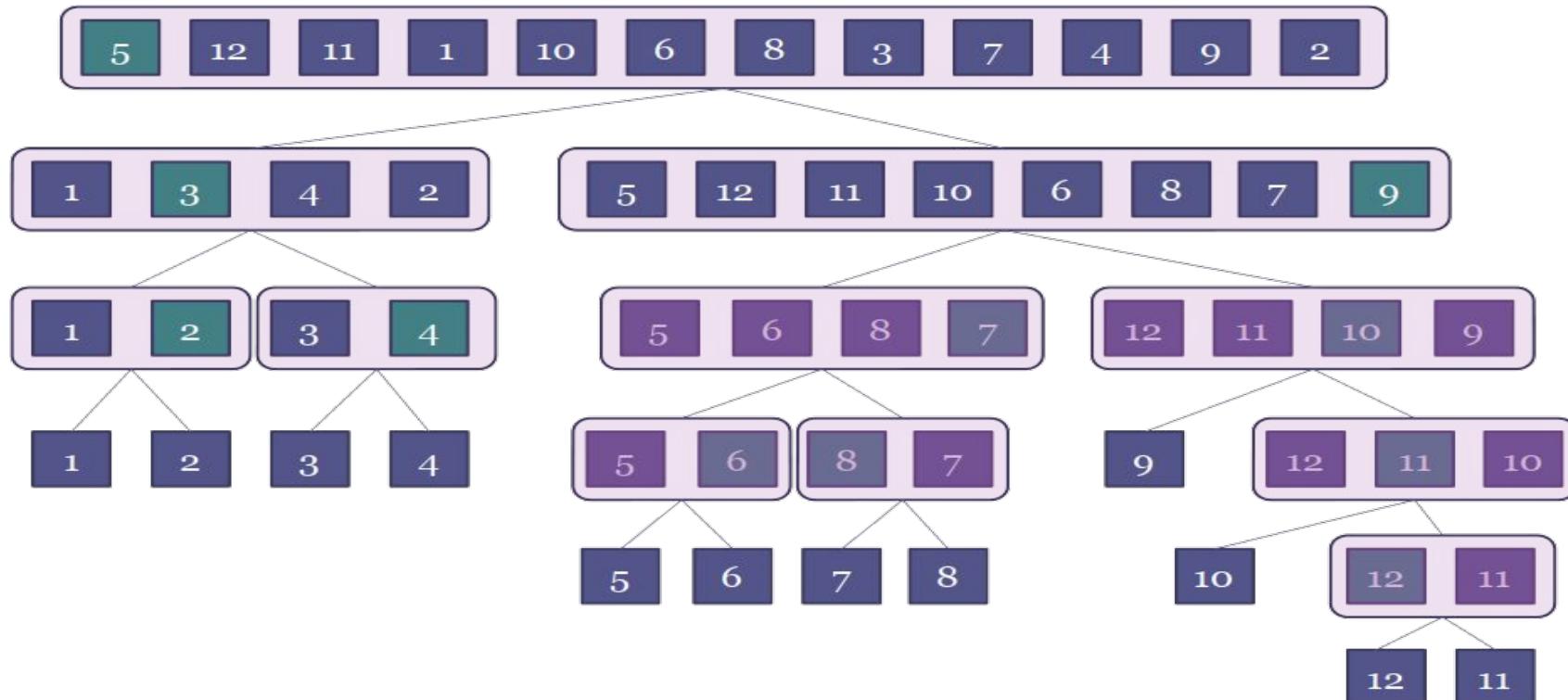
3.1 Descomposición por tareas

- Divide & Conquer
 - Las tareas no tienen porque ser de igual tamaño
 - Normalmente necesita balanceo de carga



3. Patrones de los principales algoritmos paralelos

3.1 Descomposición por tareas: Quicksort



3. Patrones de los principales algoritmos paralelos

3.2 Descomposición por datos

- Si la estructura se divide siguiendo dimensiones:
 - Algoritmos con una **descomposición geométrica** (Vectores, matrices) (Ej: Multiplicación de matrices)
- Si se divide de forma recursiva:
 - Algoritmos con un esquema recursivo (**Recursive Data**) (Para árboles, grafos, listas,...) (Ej: Encontrar la raíz)

3. Patrones de los principales algoritmos paralelos

3.3 Descomposición por flujo de datos

- Adecuada cuando el **flujo de los datos es el que ordena los grupos de tareas** definiendo un orden entre ellas
- Si el flujo es regular y estático y suele ser de una sola dirección-> **Pipeline**. Ejemplo compresión JPEG
- Si no: -> **coordinación basada en eventos**. Ejemplo, gestión de las peticiones del servidor web.
 - Monitorización de las peticiones
 - Clasificación
 - Envío para su tratamiento

3. Patrones de los principales algoritmos paralelos

3.4 Propuesta de análisis

N-Body problem

http://en.wikipedia.org/wiki/N-body_problem

(Video:

<https://www.youtube.com/watch?v=Cn2Z8bZDuyw>)

```
Array or Real :: atoms(3,N)
Array of Real :: velocities (3,N)
Array or Real :: forces (3,N)
Array or List :: neighbors(N)
Loop over time steps
    vibrational_forces(N, atoms, forces)
    rotational_forces(N, atoms, forces)
    neighbor_list(N, atoms, neighbors)
    non_bonded_forces(N, atoms, neighbors, forces)
    update_atoms_pos_and_vel(N, atoms, velocities,
    forces)
    physical_properties(...)

endloop
```



Índice

1. Introducción
2. ¿Cómo encontrar concurrencia?
 - a. Encontrar tareas
 - b. Agrupar tareas
 - c. Buscar patrones de comunicación
3. Patrones de los principales algoritmos paralelos
4. Estructuras de algoritmos más comunes
5. Qué hacer con las estructuras de datos
6. Ejemplos prácticos de algoritmos para arquitecturas de memoria compartida

4. Estructuras de algoritmos más comunes

SPMD: Single Program, Múltiple Data

Todas las hebras/procesos se derivan del mismo código en el que, en última instancia, se acaba operando sobre distintos datos.

Master/Worker: Un proceso organiza el reparto y la recolección de tareas y todos los demás suelen hacer la misma tarea o especializarse en algún tipo de ellas, y se crean y se destruyen de forma dinámica en la mayoría de los casos.

Loop Parallelism: Consiste en la división de iteraciones de bucles en hebras diferentes que normalmente son independientes o se pueden transformar de alguna forma para que lo sean.

Fork/Join: Cada unidad de ejecución se divide tantas veces como haga falta en hijos que luego se reúnen de nuevo cuando el trabajo está realizado.

4. Estructuras de algoritmos más comunes

	Task Parallelism	Divide & Conquer	Descomp. Geométrica	Recursive Data	Pipeline	Event-Based Coordination
SPMD	****	***	****	**	***	**
Loop Parallelism	****	**	***			
Master/Worker	****	**	*	*	*	*
Fork/Join	**	****	**		****	****

4. Estructuras de algoritmos más comunes

	OpenMP	MPI	Java	CUDA (OpenCL)
SPMD	***	****	**	***
Loop Parallelism	****	*	***	*
Master/Worker	**	***	***	*
Fork/Join	***		****	**

4. Estructuras de algoritmos más comunes

1. SPMD

Pasos:

- Inicializaciones
- Obtener identificaciones únicas
- Reparto de trabajo
- Ejecución del trabajo
- Recolección de resultados
- Finalizaciones

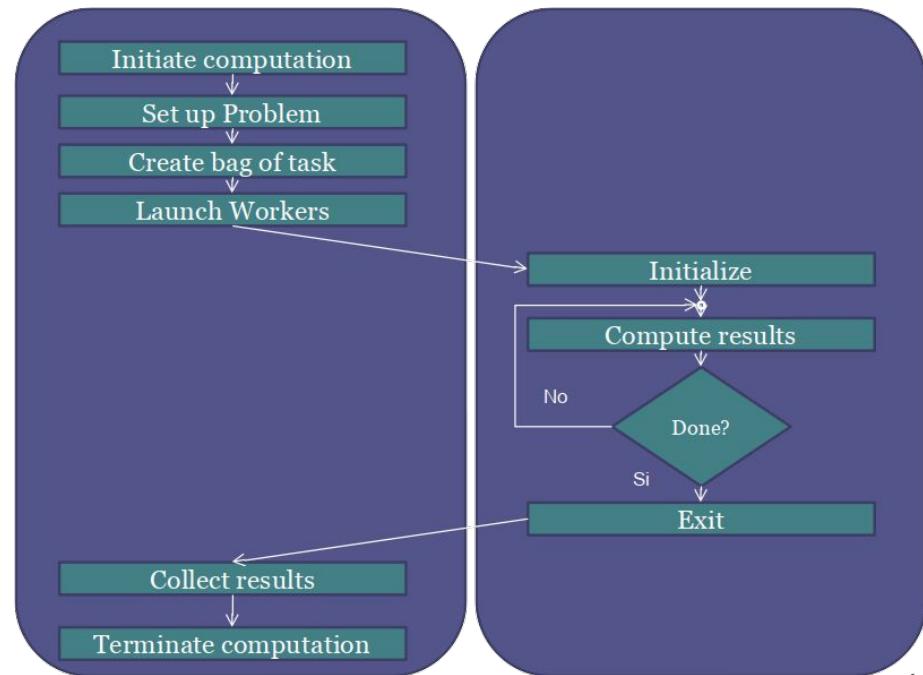
Ejemplos

4. Estructuras de algoritmos más comunes

2. Maestro - Trabajador

Características:

- Tareas de tamaños diferentes o no predecibles o con dependencias y sin comunicación necesaria entre tareas
- Se necesita balanceo de carga y tiene que estar adaptado a la máquina
- El grueso del trabajo no es un solo bucle
- Las estructuras de computación también pueden ser heterogéneas



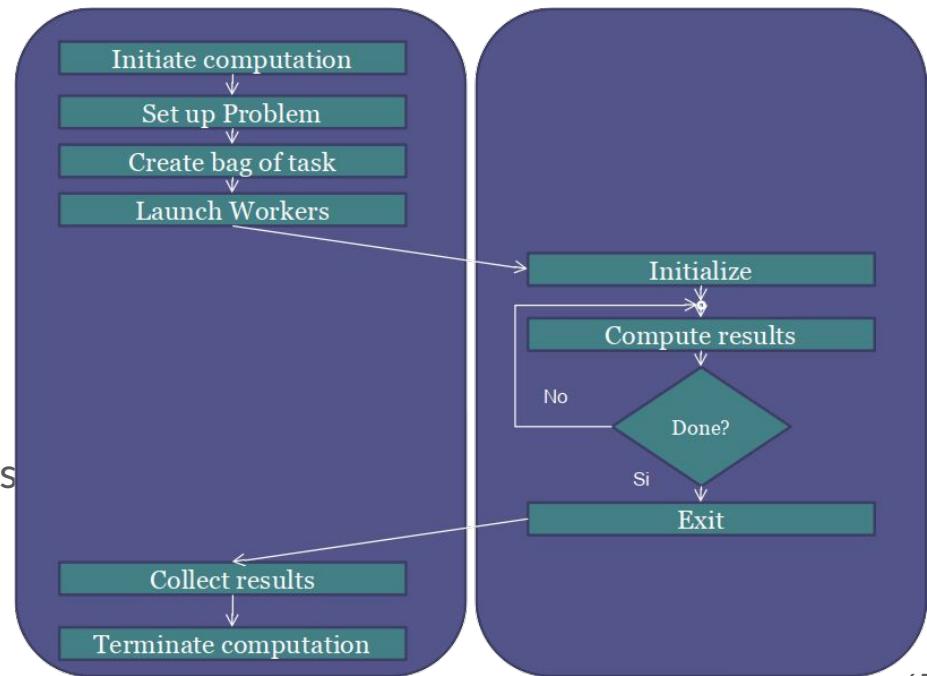
4. Estructuras de algoritmos más comunes

2. Maestro - Trabajador

Problemas:

- Comprobación de finalización.
- Asegurar la robustez incluso si los workers caen
- Implementación de cola de tareas pendientes
- Se puede necesitar jerarquización con colas de tareas también en workers.

Variaciones: El Maestro también es worker



4. Estructuras de algoritmos más comunes

3. Loop Parallelism

Pasos:

- Seleccionar bucles candidatos (bottlenecks con profilers)
- Eliminación de dependencias entre iteraciones sin afectar al resultado final
- Paralelización de bucle
- Optimización de la planificación de iteraciones entre unidades de ejecución

Problemas:

- El principal problema es el acceso simultáneo a posiciones de memoria contiguas de varias hebras
- Sincronización de hebras cuando sea necesaria por operaciones críticas
- Gestión de datos compartidos que podrían no serlo para simplificar

4. Estructuras de algoritmos más comunes

3. Loop Parallelism

Trucos:

- Unir bucles anidados
- Unir bucles adyacentes
- Invertir el orden de los bucles si es posible
- Desenrollar bucles para generar menos hebras y más cantidad de trabajo por hebra

4. Estructuras de algoritmos más comunes

4. Fork/Join

- Se suele usar solo con Java o usando el estándar POSIX, pero es más difícil de controlar.
- Las tareas se relacionan mediante jerarquía.
- El mapeo de trabajo se realiza de dos formas
 - Forma directa : 1 tarea => 1 hebra o 1 proceso
 - Forma indirecta: se asignan tareas a procesadores lo que ya implica un reparto de las hebras o de los procesos, pero los procesos y las hebras no se crean y destruyen durante toda la ejecución.

4. Estructuras de algoritmos más comunes

4. Fork/Join

- Ejemplos:
 - Implementar un patrón de maestro-trabajador con Fork/Join
 - Implementar un patrón de Loop-parallel con Fork/Join



Índice

1. Introducción
2. ¿Cómo encontrar concurrencia?
 - a. Encontrar tareas
 - b. Agrupar tareas
 - c. Buscar patrones de comunicación
3. Patrones de los principales algoritmos paralelos
4. Estructuras de algoritmos más comunes
5. Qué hacer con las estructuras de datos
6. Ejemplos prácticos de algoritmos para arquitecturas de memoria compartida

Estructuras de datos en algoritmos paralelos

Tipos de implementaciones

Las estructuras de datos que son compartidas entre varias unidades de ejecución, hebras o procesos, pueden ser de tres tipos:

- **Shared Data**: hay que mantener coherencia y eso afecta al rendimiento, además de secuenciar los accesos
- **Shared Queue**: Se implementa con colas de tipos abstractos que mantienen corrección en los datos incluso con varias hebras o procesos. Normalmente mediante cerrojos y/o barreras
- **Distributed Array**: vector que se divide entre las tareas o procesos según los datos que necesite cada uno y que no se accede a posiciones por más de una tarea o proceso.

Estructuras de datos en algoritmos paralelos

Puntos clave

Los puntos a tener en cuenta son:

- Si la estructura es compartida o no es necesario, porque nos ahorraremos la sincronización
- Si cada tarea escribe en una parte y las demás tareas solo leen, tampoco es muy problemático.

Como reglas generales:

- El resultado debe ser el mismo ante accesos múltiples o no.
- Los accesos a estructuras deben ser mínimos para evitar secuencializar las lecturas.
- Se podría limitar el número de accesos, por ejemplo en las sesiones de una BBDD.

Estructuras de datos en algoritmos paralelos

Pasos a seguir

- Asegurate de que la estructura es necesaria
- Define un tipo de dato abstracto (ADT) donde tengas bien definidas las operaciones básicas que se pueden realizar sobre dicho tipo.
- Evalúa la posibilidad de implementar un protocolo de acceso a la estructura que podrá ser:
 - Acceso One-At-A-Time:
 - Bloquear con mecanismos de cerrojos y hebras.
 - Pedir acceso al proceso que tiene el acceso mediante un mensaje.
- Establecer procesos o hebras que solo hagan una tarea, o leer del ADT o escribir en ella, facilitando la programación.
- Reducir las secciones críticas al mínimo.



Gracias.





Tema 2

Modelos de programación paralela adaptados a la arquitectura

Nicolás Calvo Cruz

Dpto. de Arquitectura y Tecnología de los Computadores

@ncalvocruz

ncalvocruz@ugr.es

Objetivos

- Aprender a **encontrar** puntos para paralelizar un algoritmo
- Aprender a **descomponer** un algoritmo en tareas
- Aprender a identificar y respetar **dependencias** entre tareas
- **Agrupar** las tareas que se pueden realizar en paralelo
- Aplicar revisiones de **diseño**



Motivación

- 1 Abordar **problemas** cada vez más grandes.
- 2 Obtener (mejores) soluciones en un **tiempo razonable**.
- 3 Ilustrar diferentes **enfoques** de la bibliografía para **parallelizar** algoritmos tradicionales.

Índice

1. Introducción
2. ¿Cómo encontrar concurrencia?
 - a. Encontrar tareas
 - b. Agrupar tareas
 - c. Buscar patrones de comunicación
3. Patrones de los principales algoritmos paralelos
4. Estructuras de algoritmos más comunes
5. Qué hacer con las estructuras de datos
6. Ejemplos prácticos de algoritmos para arquitecturas de memoria compartida

6. Ejemplos prácticos de algoritmos para arquitecturas de memoria compartida

4. Estructuras de algoritmos más comunes Recordatorio

	Task Parallelism	Divide & Conquer	Descomposición Geométrica	Recursive Data	Pipeline	Event-Based Coordination
SPMD	****	***	****	**	***	**
Loop Parallelism	****	**	***			
Master/Worker	****	**	*	*	*	*
Fork/Join	**	****	**		****	****

4. Estructuras de algoritmos más comunes (Recordatorio)

	OpenMP	MPI	Java	CUDA (OpenCL)
SPMD	***	****	**	***
Loop Parallelism	****	*	***	*
Master/Worker	**	***	***	*
Fork/Join	***		****	**

6. Ejemplos prácticos: SPMD - Map-Reduction

Map: fase en la que el trabajo se divide en partes y se gestiona independientemente.

Reduction: fase en la que se agrupan los datos en un solo. La combinación se hace con una operación que tiene elemento neutro y cumple las propiedades asociativa y conmutativa, como la suma, la multiplicación...

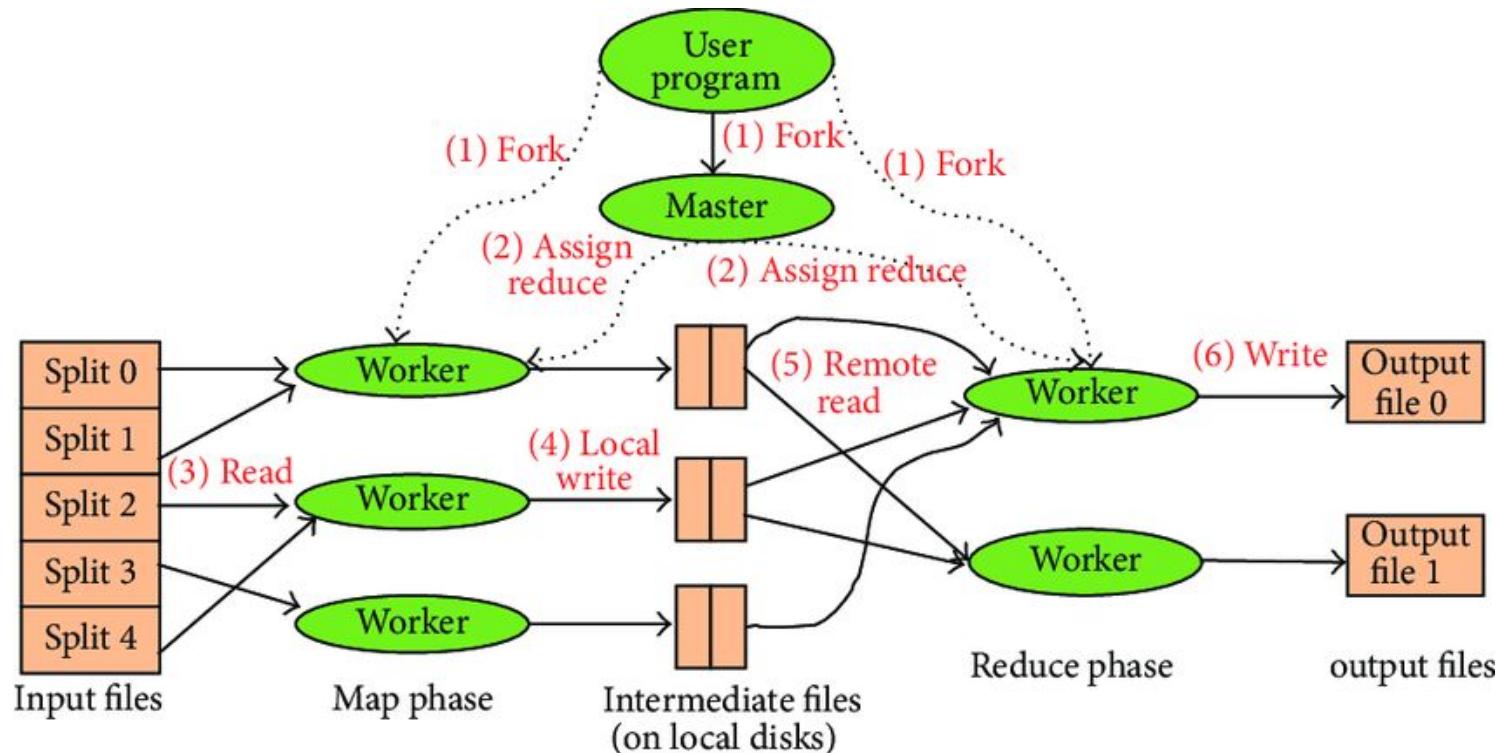
¿Donde se usa?

- Map-Reduce es el ejemplo más famoso, primero se mapea la información y luego se reduce. ([Contar palabras en Google](#))
- Calcular frecuencias ([Histograma](#))
- En el mundo real, es como funciona cualquier competición deportiva ([bracket de competición](#))

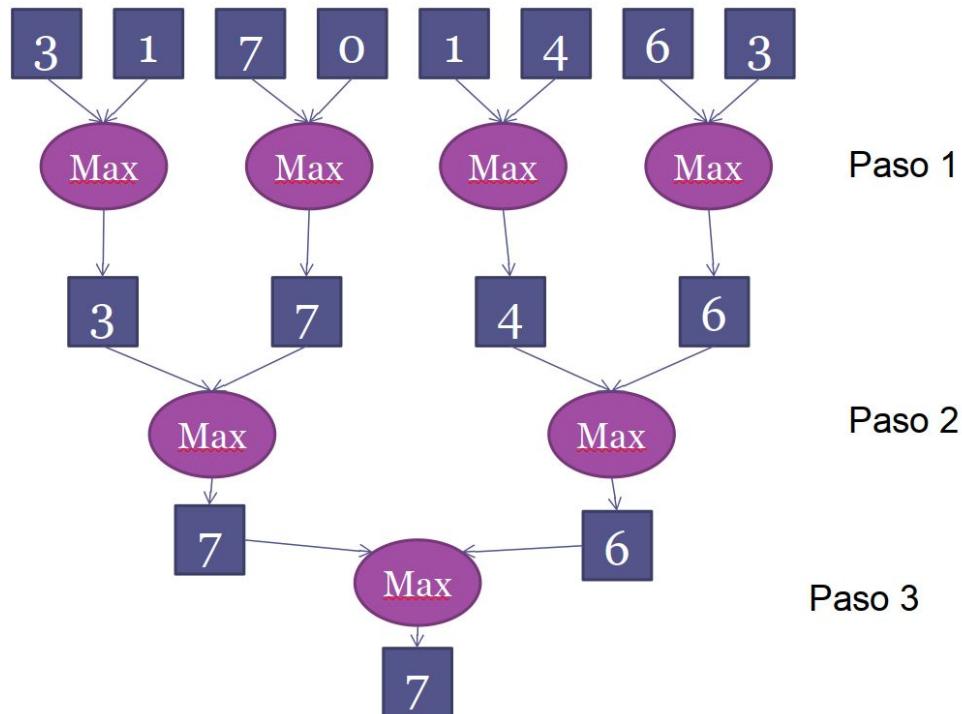
¿Problemas?

- Los volúmenes de datos.
- Las operaciones atómicas que hay que llevar a cabo.

6. Ejemplos prácticos: SPMD - Reduction



6. Ejemplos prácticos: Reduction



6. Ejemplos prácticos: SPMD - Reduction

Imaginemos que hay distintos grupos (bloques) de unidades de ejecución (hilos), con la memoria compartida a nivel de bloque (como procesos MPI que lanzan hilos... o como en una GPU).

<u>Bloque 0:</u>	Hilo 0	Hilo 1	Hilo 2	Hilo 3
<u>Bloque 1:</u>	Hilo 0	Hilo 1	Hilo 2	Hilo 3
<u>Bloque 2:</u>	Hilo 0	Hilo 1	Hilo 2	Hilo 3
<u>Bloque 3:</u>	Hilo 0	Hilo 1	Hilo 2	Hilo 3

6. Ejemplos prácticos: SPMD - Reduction

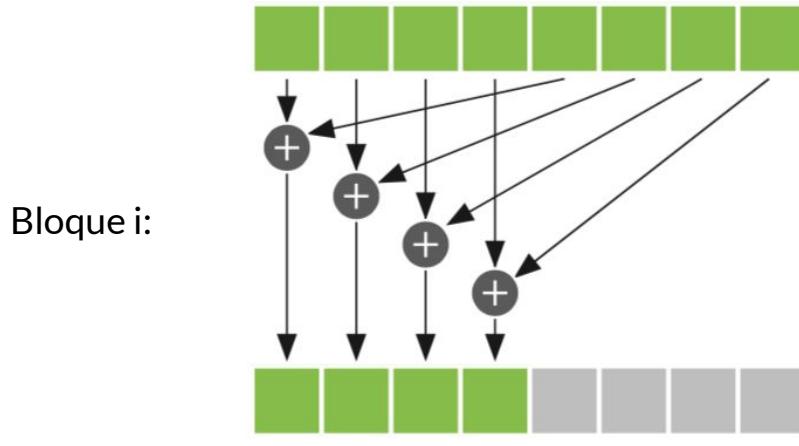
Imaginemos que hay distintos grupos (bloques) de unidades de ejecución (hilos), con la memoria compartida a nivel de bloque (como procesos MPI que lanzan hilos... o como en una GPU).

Tenemos que tener en cuenta:

- ❑ Dónde almacenar los **resultados parciales**:
 - ❑ En la primera **mitad del vector de datos**
 - ❑ En otra **estructura de datos**
 - ❑ Es importante **adecuar la estructura** a los conjuntos de hilos que tengamos.
- ❑ Controlar los hilo que ya no tienen datos a los que acceder.

6. Ejemplos prácticos: SPMD - Reduction

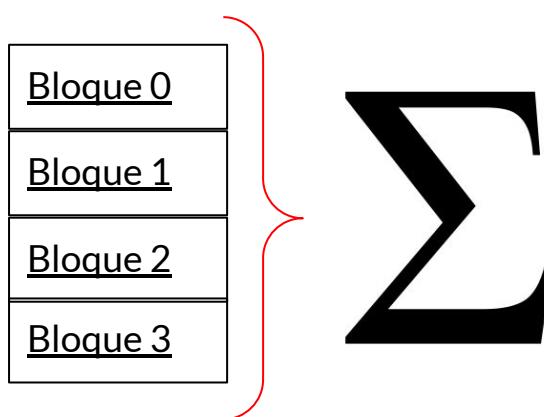
- ❑ Pensemos en una suma, donde cada bloque de hilos tiene un buffer con una posición para cada hilo:



```
calcularMiElemento();  
barrier();  
int focus = myThreadID;  
int i = blockSize/2;  
while (i != 0) {  
    if (focus < i)  
        buffer[focus] += buffer[i + focus];  
    barrier();  
    i /= 2;  
}
```

6. Ejemplos prácticos: SPMD - Reduction

- ❑ Pensemos en una suma, donde cada bloque de hilos tiene un buffer con una posición para cada hilo:



```
barrierBloque();
int resultadoFinal = 0;
for (i in bloques) {
    resultadoFinal += bloques[i].buffer[0]
}
```

6. Ejemplos prácticos: SPMD - Reduction

- Si esta reducción o suma acumulada la hicéramos **en secuencial**: El **tiempo** sería **proporcional a la longitud del vector**.
- Aprovechando los múltiples hilos... el tiempo será proporcional al logaritmo de la longitud del vector:
 - Primero, cada hilo combina dos registros en uno, necesitando la mitad de pasos que elementos.
 - Se repite con la **mitad** que queda, tardando entonces la **mitad de la mitad...**
- Tardaremos entonces $\log_2(\text{hilosEnBloque})$... Por ejemplo, con **256 hilos** (y elementos), quedan **sumados tras 8 iteraciones**.

- Así se plasma esta idea en CUDA:

```
__global__ void productoEscalar( float *a, float *b, float *c ){
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;
    float temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
    cache[cacheIndex] = temp;
    // Sincronizar hilos de este bloque
    __syncthreads();
    // threadsPerBlock debe ser potencia de 2
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }
    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
```

6. Ejemplos prácticos: SPMD - Reduction

Finalmente, en la CPU:

```
(...)
cudaMemcpy( partial_c, dev_partial_c, sizeof(float)*blocksPerGrid, cudaMemcpyDeviceToHost );

c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}
(...)
```

6. Ejemplos prácticos: SPMD-scan

- El algoritmo **Prefix Sum (Scan)** es usado en asignación de tareas en procesadores masivamente paralelos y en asignación de recursos para hilos.
- Se utiliza también para pasar de código secuencial a paralelo, eliminando la limitación de la escalabilidad por las secuencias de código paralelo.

```
for(j=1;j<n;j++)  
out[j]=out[j-1]+f(j);
```



```
forall(j){  
    temp[j]=f[j] // se hace en paralelo  
}  
scan(out, temp) // se puede hacer en paralelo
```

- Es uno de los patrones de computación paralela más usado y da pistas para resolver otros problemas similares.

6. Ejemplos prácticos: SPMD-scan

Formalmente, el problema se puede definir así, en su variante inclusiva:

- Sea un vector X de tamaño N : $[X_0, X_1, \dots, X_{N-1}]$

Se desea **calcular la secuencia** que relaciona todos los X_i con una función f :

$X_0, f(X_0, X_1), f(X_0, X_1, X_2), \dots f(X_0 \dots X_{N-1})$

Por ejemplo, si f es la operación suma y $X = [3, 1, 7, 0, 4, 1, 6]$

La solución será: [3, 3+1, 3+1+7, 3+1+7+0, 3+1+7+0+4, 3+1+7+0+4+1, 3+1+7+0+4+1+6]

= [3, 4, 11, 11, 15, 16, 22]

(Nota: En definición exclusiva, X_i no se coge para $f(X_i)$)

6. Ejemplos prácticos: SPMD-scan

- Supongamos que vamos a repartir **100 trozos de tarta entre 10 personas**, y cada una pide la siguiente cantidad: [3,5,2,7,28,4,3,0,8,1].
- Uno repartiría tarta de forma constante y pararía cuando cada persona tuviera lo pedido, es decir, cortar 1 porción y darla, otra porción y darla y así sucesivamente...
¿Pero cuánta dejar por si viene alguien más?
- Una **opción (secuencial)** sería cortar primero 3, luego los 5... y así sucesivamente.
- Otra: calculamos **scan inclusivo**, obteniendo [3, 8, 10, 17, 45, 49, 52, 52, 60, 61] y sabiendo así por donde cortar la tarta dejando **100-61** porciones y cómo marcar las grandes zonas de tarta rápidamente.

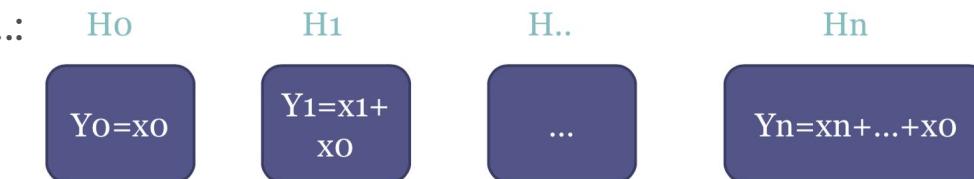
6. Ejemplos prácticos: SPMD-scan

- En secuencial: Hacer $Y_0 = X_0; Y_1 = X_0 + X_1; Y_2 = X_0 + X_1 + X_2 \dots$:

```
y[0]=x[0];  
for(i=1;i<n;i++) y[i]=y[i-1]+x[i];
```

Eficiencia de orden N, O(N)

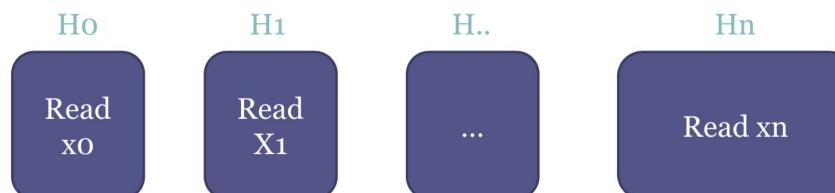
- Una primera (y descuidada) versión paralela crearía un hilo para calcular cada elemento de salida...:



- Pero claro, las **iteraciones finales** son **más costosas**... La latencia vendrá dada por el último hilo.

6. Ejemplos prácticos: SPMD-scan

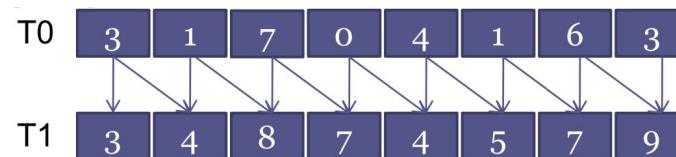
- Una segunda versión paralela, conceptualmente mucho más afinada, haría que el **1^{er}** hilo calculara los elementos **0** y **N-1**, que el **2º** se ocupara del **1** y el **N-2**, que el **3º** hiciera el **2** y **N-3**... Esto nos balancea perfectamente las operaciones... pero no tiene en cuenta los accesos a memoria y su escalabilidad llega hasta **N/2**.
- Otra alternativa (GPU): La estructura de datos es de lectura y escritura -> cada hilo debe acceder a una parte de la estructura. El primer paso será:



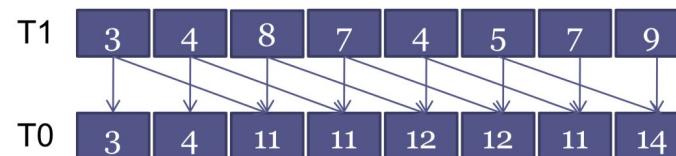
- Tras esto, todos los elementos estarán en la memoria compartida entre hilos.

6. Ejemplos prácticos: SPMD-scan

- Seguidamente: Iterar $\log(n)$ y sumar elementos que distan un determinado paso y que aumentará al doble en la siguiente iteración y alternando T0 y T1.



Distancia de suma = 1

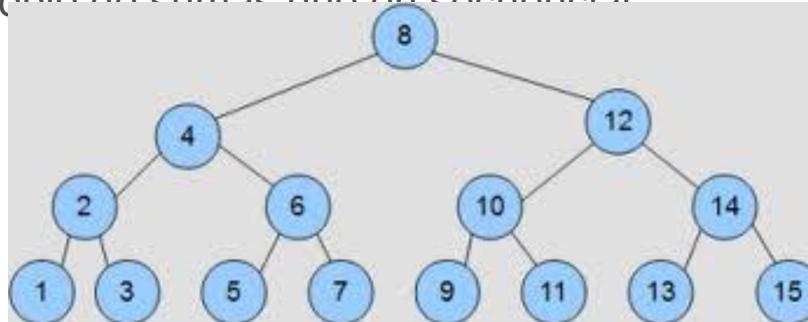


Distancia de suma = 2

(...)

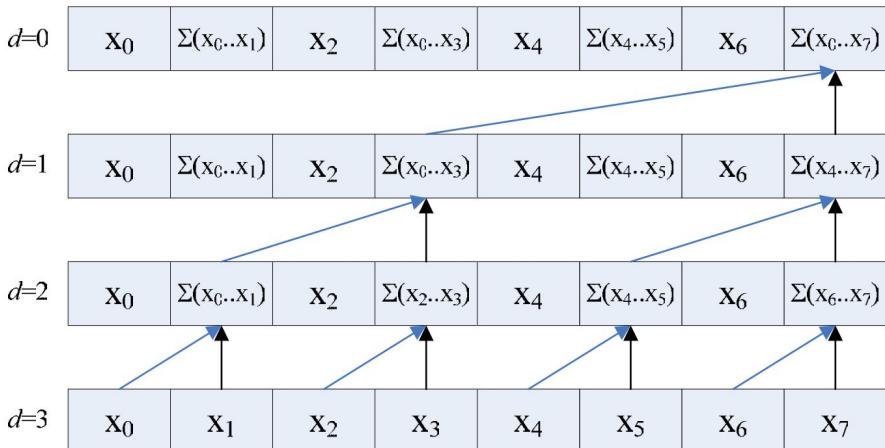
6. Ejemplos prácticos: SPMD-scan

- Según el tamaño de los bloques (hilos), tienen que **sincronizarse** por bloque en cada iteración. Además, se hacen **$O(n*\log(n))$ sumas**. Por ejemplo para 256 elementos, la secuencial hace 255 sumas y la paralela $(256*\log(256)-(255)=1793)$ sumas.
- Solución: **Árboles** balanceados => Transformar el vector de números en un árbol B y aprovecharnos de las propiedades del árbol para implementar el SCAN. Así sólo se hace el doble de sumas que en secuencial

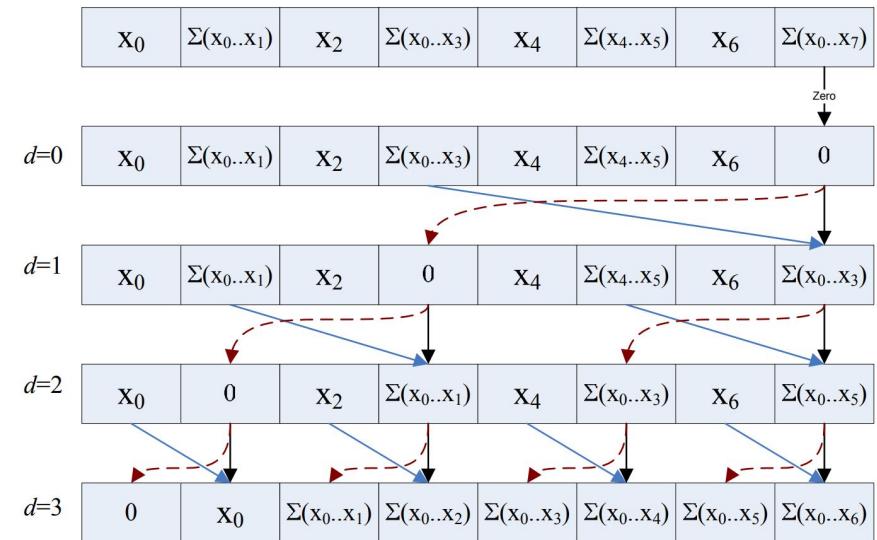


6. Ejemplos Prácticos: SPMD-scan

Up-Sweep



Down-Sweep



Este enfoque tiene conflictos por los accesos a memoria en los arrays compartidos por cada bloque de hebras, porque en algunos casos las hebras acceden a posiciones contiguas de memoria

6. Ejemplos Prácticos: SPMD-scan

Enfocamos el problema como dos fases:

Fase de reducción (desde hojas hasta raíz) calculando algunas sumas parciales que son necesarias para la fase 2 (Fase Up-Sweep)

Fase de barrido de subida, (Down-Sweep) donde se ponen algunos elementos del array a 0 inicializando con el último elemento del array que se pone a 0 desde el mismo programa

```

for  $d := 0$  to  $\log_2 n - 1$  do
    for  $k$  from 0 to  $n - 1$  by  $2^{d+1}$  in parallel do
         $x[k + 2^{d+1} - 1] := x[k + 2^d - 1] + x[k + 2^{d+1} - 1]$ 
    
```

```

 $x[n - 1] := 0$ 
for  $d := \log_2 n$  down to 0 do
    for  $k$  from 0 to  $n - 1$  by  $2^{d+1}$  in parallel do
         $t := x[k + 2^d - 1]$ 
         $x[k + 2^d - 1] := x[k + 2^{d+1} - 1]$ 
         $x[k + 2^{d+1} - 1] := t + x[k + 2^{d+1} - 1]$ 
    
```

- Más detalles de este tipo de problema en: <https://slideplayer.com/slide/14460027/>

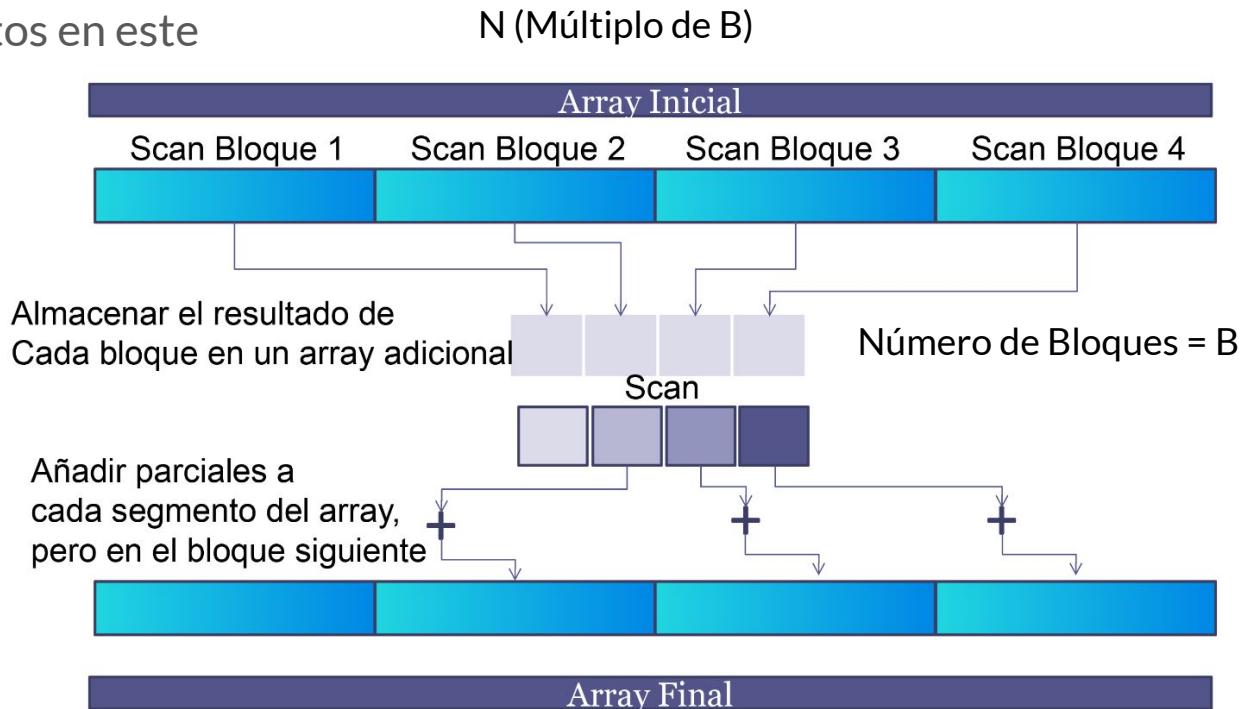
```
y[0]=x[0];
for(i=1;i<n;i++) y[i]=y[i-1]+x[i];
```

6. Ejemplos prácticos: scan

- Y si no caben los datos en este enfoque lineal...

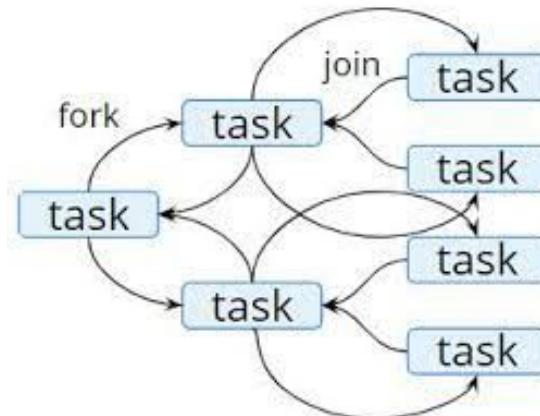
$B = \text{Número elementos calculados en un bloque}$
 $\text{Bloques} = N/B$

Hilos = $B/2$



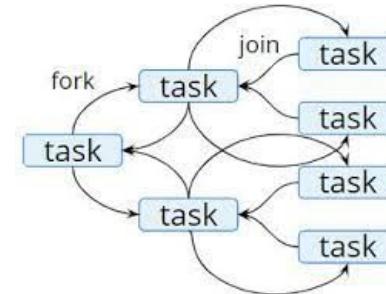
6. Ejemplos prácticos: fork-join

El esquema principal de la estructura Fork-Join es:



6. Ejemplos prácticos: fork-join

Veamos el framework de Java Fork/Join dentro del paquete `java.util.concurrent`.



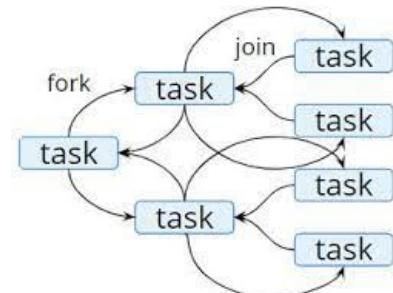
Las dos clases principales son:

1. **ForkJoinTask**: clase que nos permite crear bifurcaciones y uniones de tareas con dos subclases principales:
 - **RecursiveAction** : se utiliza para las tareas que no devuelven resultados
 - **RecursiveTask** : se utiliza para las tareas que sí devuelven resultados
2. **ForkJoinPool**: que mediante ForkJoinTask la ajustamos para crear una pila de tareas a gestionar.

La forma de gestionar los hilos es mediante una cola de tareas que cada hilo mantiene y que si se queda vacía se podrían recopilar tareas de otras colas para ejecutarlas (*work stealing*).

6. Ejemplos prácticos: fork-join

```
public static void main(String[] args) throws Exception {  
    // Toma de tiempos e inicialización de la pila de tareas.  
    //  
    long start=System.currentTimeMillis();  
    ForkJoinPool pool=new ForkJoinPool();  
    Future<Integer>future=pool.submit(new ForkJoin(1,1000000));  
    long end=System.currentTimeMillis();  
    System.out.println("-----");  
    System.out.println("Resultado Paralelo:"+future.get()+" Tiempo"+(end-start));  
  
    // Comparación con la ejecución secuencial  
    long start1=System.currentTimeMillis();  
    int sum=getadd(1,1000000);  
    long end1=System.currentTimeMillis();  
    System.out.println("Resultado Secuencial:"+sum+" Tiempo "+(end1-start1));  
}
```



```

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;
import java.util.concurrent.RecursiveTask;

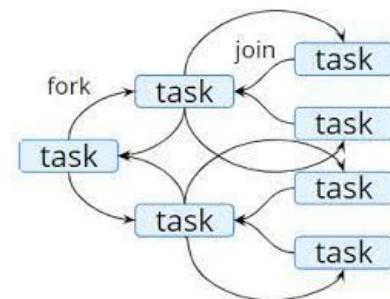
//Esquema del marco de ForkJoin (calcule la suma desde comenzar ... terminar, como 1 + 2 + 3 ... + 100)

public class ForkJoin extends RecursiveTask<Integer>{
    private static final long serialVersionUID = 28980553733573142L;
    private int begin; // Calcular el punto de partida de la suma acumulativa
    private int end; // Calcular el punto final de la suma acumulativa
    public ForkJoin(int begin, int end) { super(); this.begin = begin; this.end = end; }

    @Override
    protected Integer compute() {
        int sum=0;
        if(end-begin<=1) { // La tarea se divide si supone un cierto tamaño
            for(int i=begin;i<=end;i++) sum+=i; // Calcula la pequeña suma
        }else { // Split
            ForkJoin d1= new ForkJoin(begin, (begin+end)/2);
            ForkJoin d2= new ForkJoin((begin+end)/2+1,end);
            // Tarea de ejecución dividida
            d1.fork();
            d2.fork();
            Integer a=d1.join();
            Integer b=d2.join();
            sum=a+b;
        }
        return sum;
    }
    // Resultado de ejecución de subprocesso único
    public static int getadd(int begin,int end) {
        int sum=0;
        for(int i=begin;i!=end+1;i++) { sum+=i; }
        return sum;
    }
}

```

6. Ejemplos prácticos: fork-join



- Y otro ejemplo que ya vimos al hablar de recursividad:
<https://stackoverflow.com/questions/22583012/parallel-algorithm-to-produce-all-possible-sequences-of-a-set>

6. Ejemplos prácticos: master/worker

Hay una entidad por encima de las demás y que puede controlar el flujo general del trabajo.

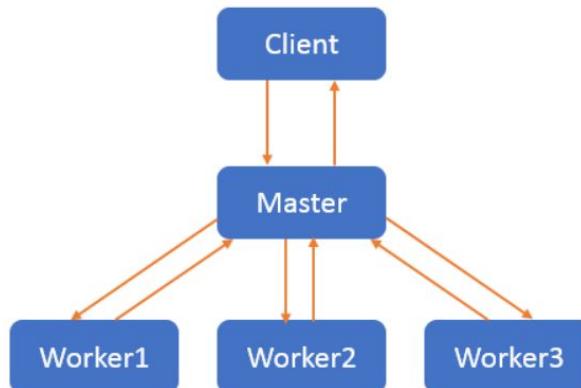
La principal diferencia con fork/Join es que la cantidad de trabajo no tiene por qué estar fijada desde el principio.

Existen distintas variantes de este esquema. Por ejemplo:

- Cada Worker tendrá su propia cola de tareas a las que el Master le va introduciendo trabajo.
- Los hilos Worker se crean desde el principio y permanecen hasta el final aunque no existan tareas pendientes en sus colas.
- Es el Maestro el que marca la creación y la destrucción de los Workers.

6. Ejemplos prácticos: master/worker

El esquema principal de la estructura Master/Worker es:

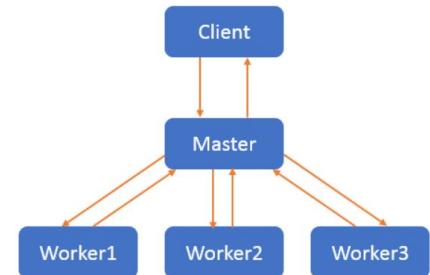


6. Ejemplos prácticos: master/worker

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

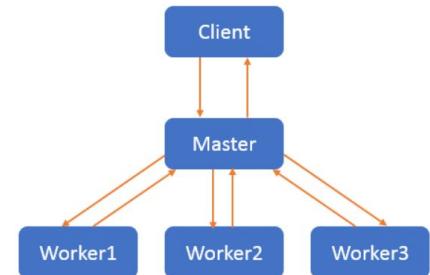
#define NUM_WORKERS 4

int works[NUM_WORKERS];
pthread_mutex_t mutex[NUM_WORKERS];
pthread_cond_t cond[NUM_WORKERS];
```



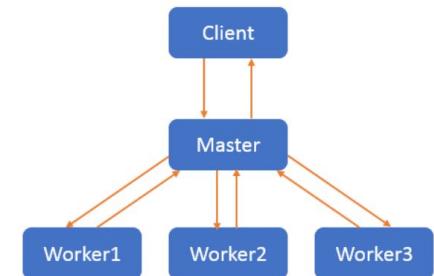
6. Ejemplos prácticos: master/worker

```
void* worker(void* param){  
    long int myID = (long int) param;  
    int myTask;  
    while(1){  
        pthread_mutex_lock(&(mutex[myID]));  
        while(works[myID]<0){  
            pthread_cond_wait(&(cond[myID]), &(mutex[myID]));  
        }  
        myTask = works[myID];  
        if(myTask!=0){  
            sleep(myTask);  
            printf("Hilo [%d]: %d^2 = %d\n", myID, myTask, myTask*myTask);  
            works[myID] = -1;  
        }else{  
            pthread_mutex_unlock(&(mutex[myID]));  
            break;  
        }  
        pthread_mutex_unlock(&(mutex[myID]));  
    }  
    return 0;  
}
```



6. Ejemplos prácticos: master/worker

```
int main(void){  
    pthread_t workers[NUM_WORKERS];  
    for(long int i = 0; i<NUM_WORKERS; i++){  
        pthread_mutex_init(&(mutex[i]), 0);  
        pthread_cond_init(&(cond[i]), 0);  
        works[i] = -1;  
    }  
    for(long int i = 0; i<NUM_WORKERS; i++){  
        pthread_create(&(workers[i]), 0, worker, (void*) i);  
    }  
    (...)
```



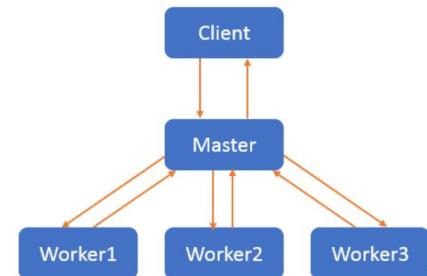
(...)

//BUCLE DE GESTIÓN DE TRABAJO <MASTER>:

```

int newTask = 1;
int focus = 0;
while(newTask){
    printf("Introduce un elemento >0 para elevarlo al cuadrado:\n");
    scanf("%d", &newTask);
    if(newTask){
        while(1){
            if(!pthread_mutex_trylock(&(mutex[focus]))){
                works[focus] = newTask;
                pthread_cond_signal(&(cond[focus]));
                pthread_mutex_unlock(&(mutex[focus]));
                focus = (focus + 1) % NUM_WORKERS;
                break;
            }else{
                focus = (focus + 1) % NUM_WORKERS;
            }
        }
    }else{//Indicamos salida:
        for(long int i = 0; i<NUM_WORKERS; i++){
            pthread_mutex_lock(&(mutex[i]));
            works[i] = 0;
            pthread_cond_signal(&(cond[i]));
            pthread_mutex_unlock(&(mutex[i]));
        }
    }
}
(...)
```

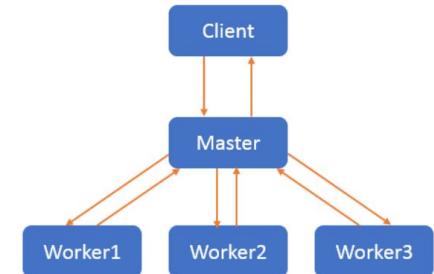
6. Ejemplos prácticos: master/worker

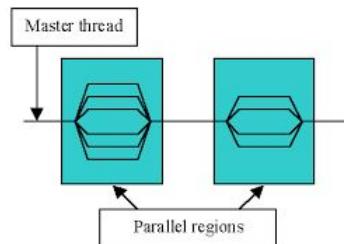


6. Ejemplos prácticos: master/worker

```
(...)
for(long int i = 0; i<NUM_WORKERS; i++){
    pthread_join(workers[i], 0);
}
for(long int i = 0; i<NUM_WORKERS; i++){
    pthread_mutex_destroy(&(mutex[i]));
    pthread_cond_destroy(&(cond[i]));
}
return 0;
}
```

```
|MBP-de-Nicolas:Desktop nccruz$ ./mW
Introduce un elemento >0 para elevarlo al cuadrado:
10
Introduce un elemento >0 para elevarlo al cuadrado:
4
Introduce un elemento >0 para elevarlo al cuadrado:
5
Introduce un elemento >0 para elevarlo al cuadrado:
2
Introduce un elemento >0 para elevarlo al cuadrado:
Hilo [3]: 2^2 = 4
Hilo [1]: 4^2 = 16
Hilo [2]: 5^2 = 25
Hilo [0]: 10^2 = 100
```





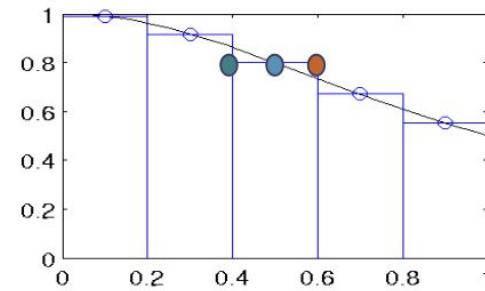
6. Ejemplos prácticos: Loop Parallelism

- El ámbito de la paralelización se centra en **repartir iteraciones de bucles entre distintas unidades de ejecución**.
- El estándar **OpenMP**, una API para programación en memoria compartida sobre C, C++ y Fortran, es uno de los ejemplos por excelencia orientados (aunque no limitados) a esta estrategia:
 - Define directivas de compilación que se traducen automáticamente en código paralelo para la plataforma.
 - Define también funciones con las que consultar ID's de hilo, total de hilos, variables de estado... y tipos de datos (por ejemplo, `omp_lock_t`).
- Podemos encontrar **implementaciones** de este modelo de paralelismo en otros entornos, como ***parfor*** en Matlab o **Parallel.For** en C#.

6. Ejemplos prácticos: Loop Parallelism

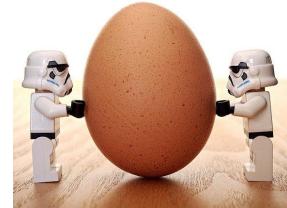
```
double piRectangles(int intervals){  
    double width = 1.0/intervals;  
    double sum = 0.0, x;  
    #pragma omp parallel for reduction(+:sum) private(x)  
    for(int i = 0; i<intervals; i++){  
        x = (i + 0.5)*width;  
        sum += 4.0/(1.0 + x*x);  
    }  
    return sum*width;  
}
```

```
nicolas@miriam-pc:~/Documentos/Docencia/UGR_ACAP/MiPr1$ gcc -o pi_omp omp_pi.c -fopenmp  
nicolas@miriam-pc:~/Documentos/Docencia/UGR_ACAP/MiPr1$ time ./pi_omp 10000000  
PI por integración de círculo [10000000 intervalos] = 3.141593  
real    0m0,025s  
user    0m0,182s  
sys     0m0,000s
```



$$\int_0^1 \frac{1}{1+x^2} dx$$

Trabajo para la próxima semana



Cada grupo debe explicar y presentar un mini-ejemplo funcional de las siguientes funciones colectivas de MPI:

- Grupo 1: MPI_Barrier
- Grupo 2: MPI_Reduce
- Grupo 3: MPI_Bcast
- Grupo 4: MPI_Gather
- Grupo 5: MPI_Allgather
- Grupo 6: MPI_Scatter
- Grupo 7: MPI_Scatterv



Gracias.





Tema 2

Modelos de programación paralela adaptados a la arquitectura

Nicolás Calvo Cruz

Dpto. de Arquitectura y Tecnología de los Computadores

@ncalvocruz

ncalvocruz@ugr.es

Objetivos

- Aprender a **encontrar** puntos para paralelizar un algoritmo
- Aprender a **descomponer** un algoritmo en tareas
- Aprender a identificar y respetar **dependencias** entre tareas
- **Agrupar** las tareas que se pueden realizar en paralelo
- Aplicar revisiones de **diseño**



Motivación

- 1 Abordar **problemas** cada vez más grandes.
- 2 Obtener (mejores) soluciones en un **tiempo razonable**.
- 3 Ilustrar diferentes **enfoques** de la bibliografía para **parallelizar** algoritmos tradicionales.

Índice

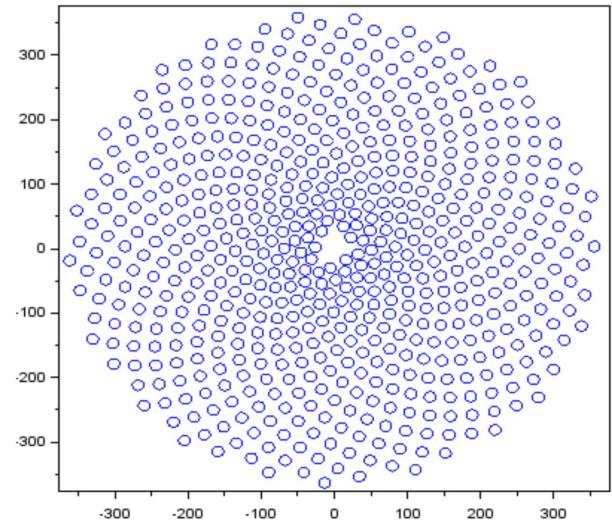
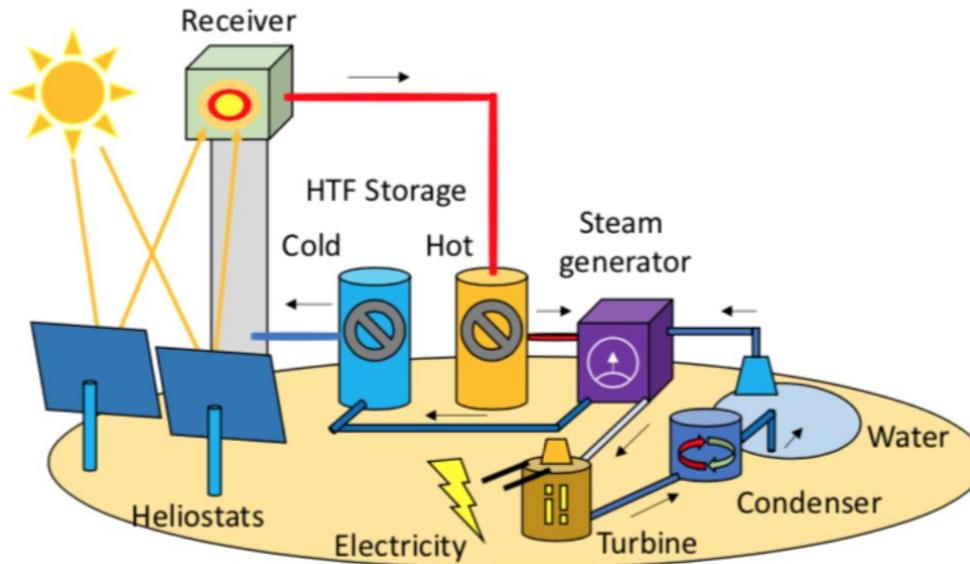
1. Introducción
2. ¿Cómo encontrar concurrencia?
 - a. Encontrar tareas
 - b. Agrupar tareas
 - c. Buscar patrones de comunicación
3. Patrones de los principales algoritmos paralelos
4. Estructuras de algoritmos más comunes
5. Qué hacer con las estructuras de datos
6. Ejemplos prácticos de algoritmos paralelos

6. Ejemplos prácticos de algoritmos paralelos

6. Ejemplos prácticos: Simular un campo de heliostatos

Simular un campo de heliostatos (I)

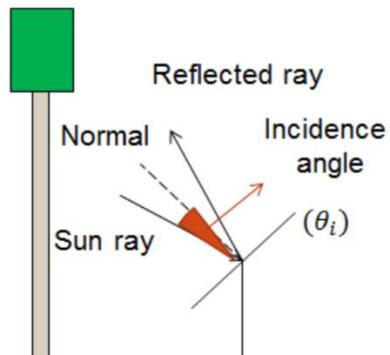
- 50% de coste
- 40% de pérdida energética



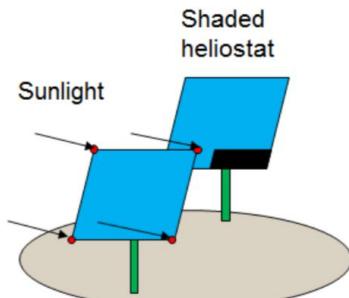
6. Ejemplos prácticos: Simular un campo de helióstatos

Simular un campo de helióstatos (II)

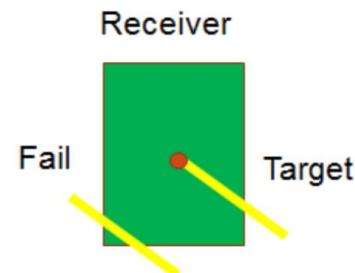
$$\eta = \eta_{cos} \cdot \eta_{sb} \cdot \eta_{itc} \cdot \eta_{aa} \cdot \eta_{ref}$$



Factor coseno



Bloqueo y sombreado



Factor de intercepción

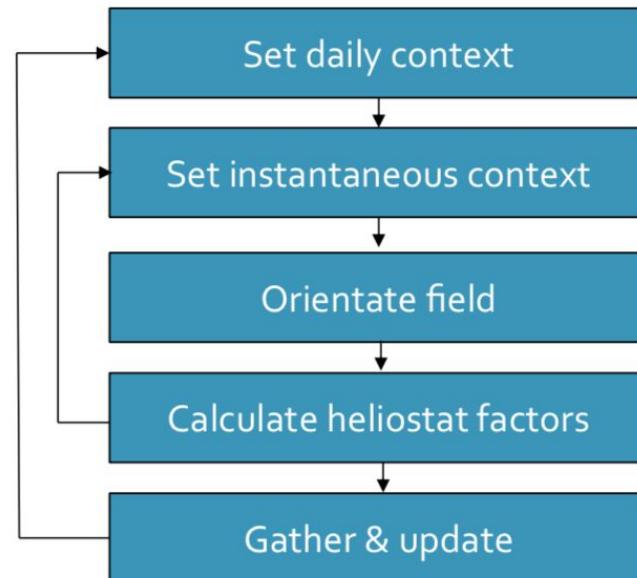


Atenuación atmosférica

6. Ejemplos prácticos: Simular un campo de helióstatos

Simular un campo de helióstatos (III)

El proceso de simulación se puede resumir así:



6. Ejemplos prácticos: Simular un campo de helióstatos

Simular un campo de helióstatos (IV)

Este proceso se puede implementar con éxito bajo un modelo **fork-join** en el que un conjunto de hilos se encarga de realizar el cálculo **supeditado** a donde se necesite (optimizador?).

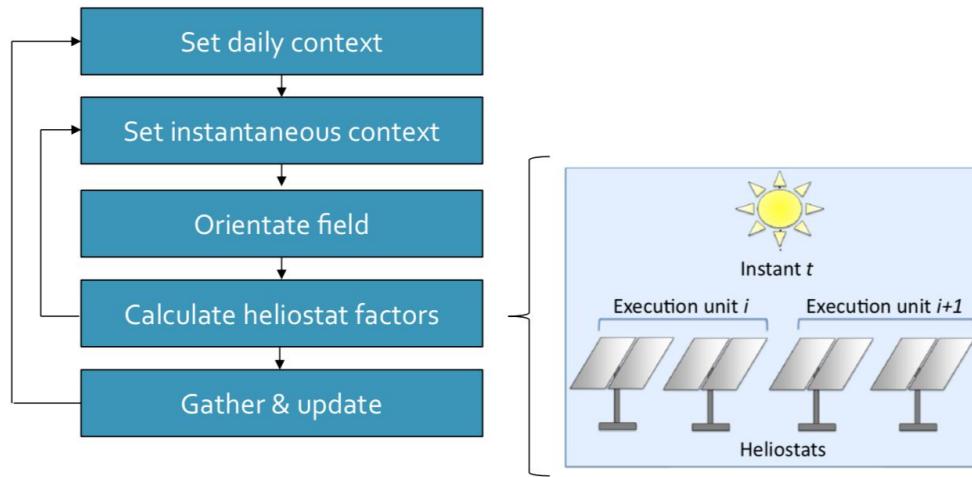
Encontramos 3 “entidades”:

- Helióstatos
- Instantes
- Conjuntos de instantes (p.ej. días)

6. Ejemplos prácticos: Simular un campo de helióstatos

Simular un campo de helióstatos (V)

División por helióstatos:

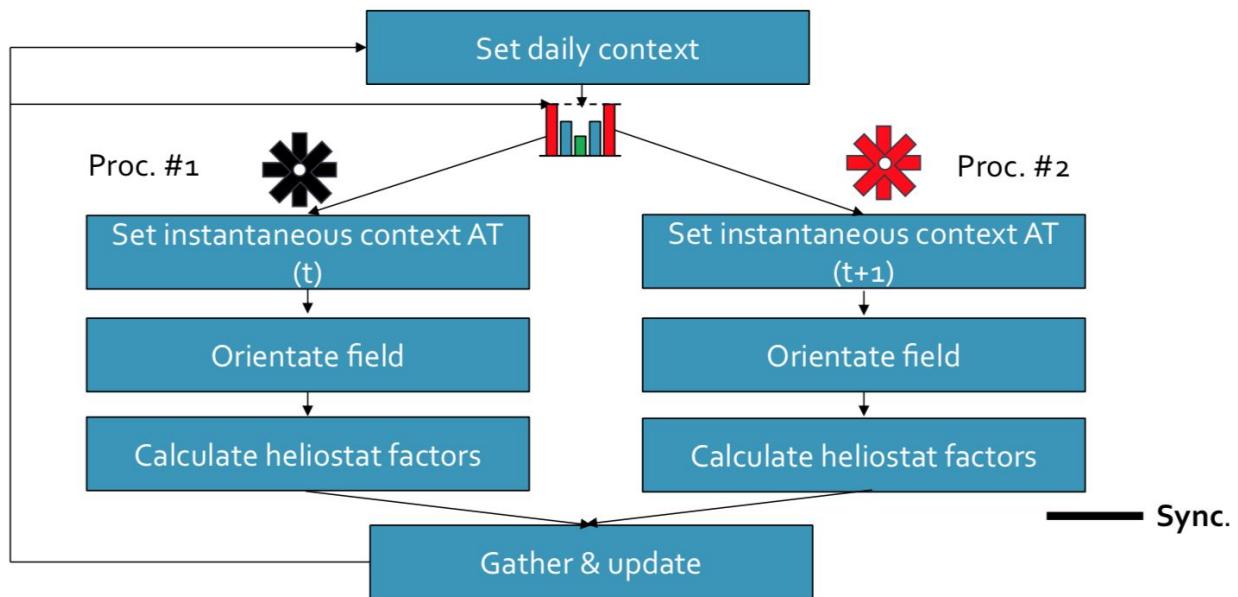


Más simple...
y más overhead

6. Ejemplos prácticos: Simular un campo de helióstatos

Simular un campo de helióstatos (VI)

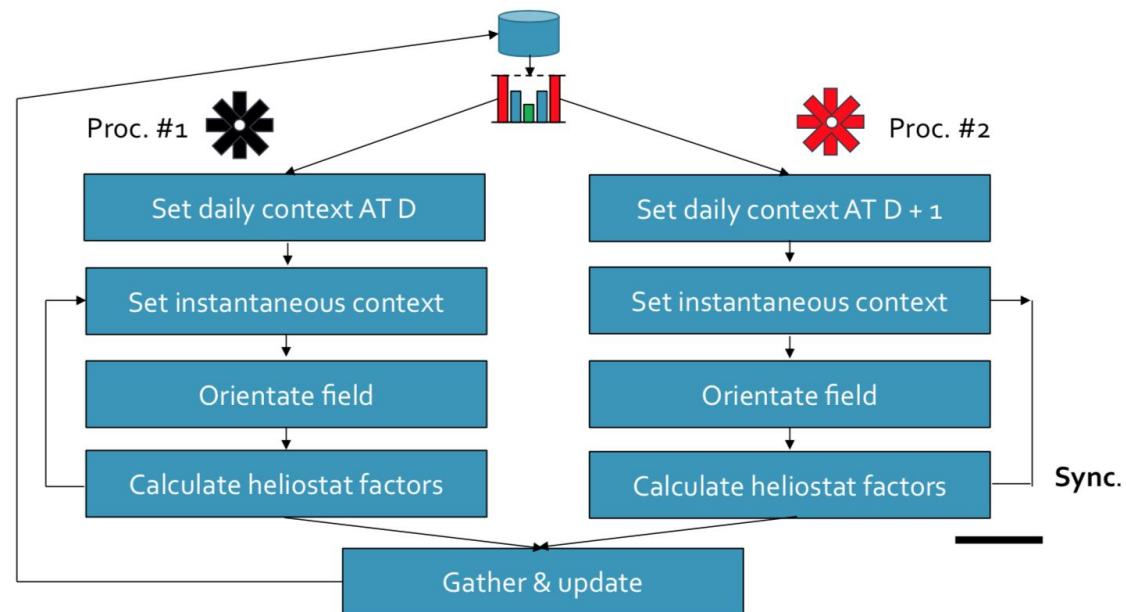
División por instantes:



6. Ejemplos prácticos: Simular un campo de helióstatos

Simular un campo de helióstatos (VII)

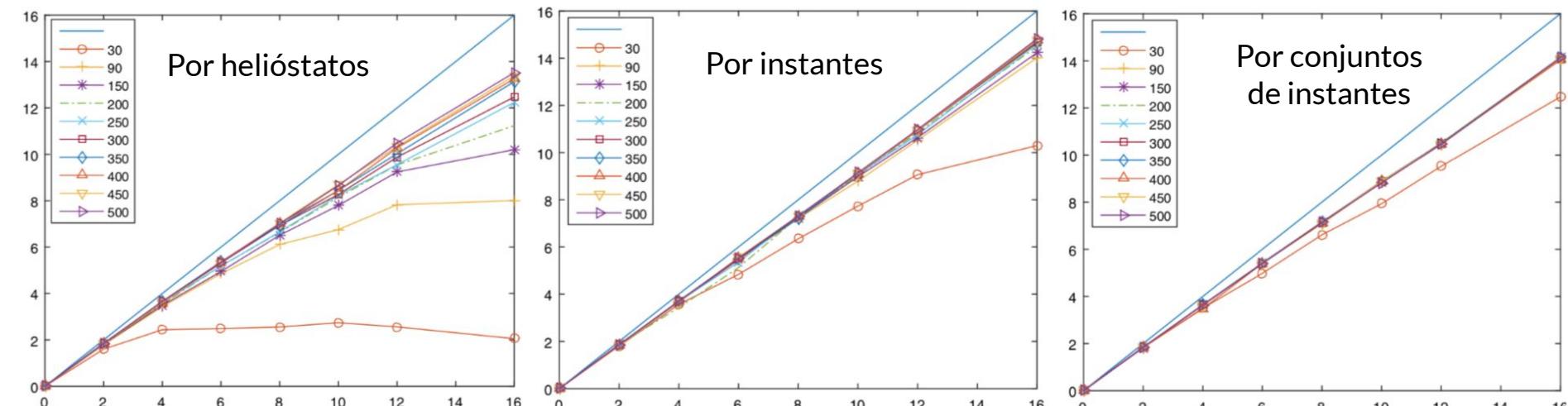
División por conjuntos de instantes:



Más complejo
y más tamaño de grano

6. Ejemplos prácticos: Simular un campo de helióstatos

Simular un campo de helióstatos (VIII)



6. Ejemplos prácticos: Optimizadores paralelos

Optimización continua (I)

Búsqueda de los extremos de una función continua. Esta función f , conocida como “función objetivo”, modela algún aspecto de la realidad (coste de distribución, eficiencia de producción...) y depende de una serie (N) de variables:

$$f : \mathbb{R}^N \rightarrow \mathbb{R}$$

Si f es un coste, nos interesan sus mínimos (**minimización**). Si es una eficiencia, nos interesan sus máximos (**maximización**)... Realmente cambia poco, pues minimizar f es igual que maximizar $-f$.

(Este tipo de problemas también se denominan **Programación Matemática**)

6. Ejemplos prácticos: Optimizadores paralelos

Optimización continua (II)

Podemos tener además una serie de **restricciones** que cumplir, y que nos acotan las soluciones válidas.

Las más restricciones más simples, conocidas como “de caja”, limita el rango de cada variable entre un límite inferior, L, y uno superior, U:

$$f : \mathbb{R}^N \rightarrow \mathbb{R}$$

queda limitada al espacio vectorial definido por $[L_1, U_1] \times \dots \times [L_N, U_N]$

6. Ejemplos prácticos: Optimizadores paralelos

Optimización continua (III)

Formalmente, podríamos definir así un problema de optimización (minimización) continua con restricciones de caja:

$$\underset{x}{\text{minimize}} \quad f(x)$$

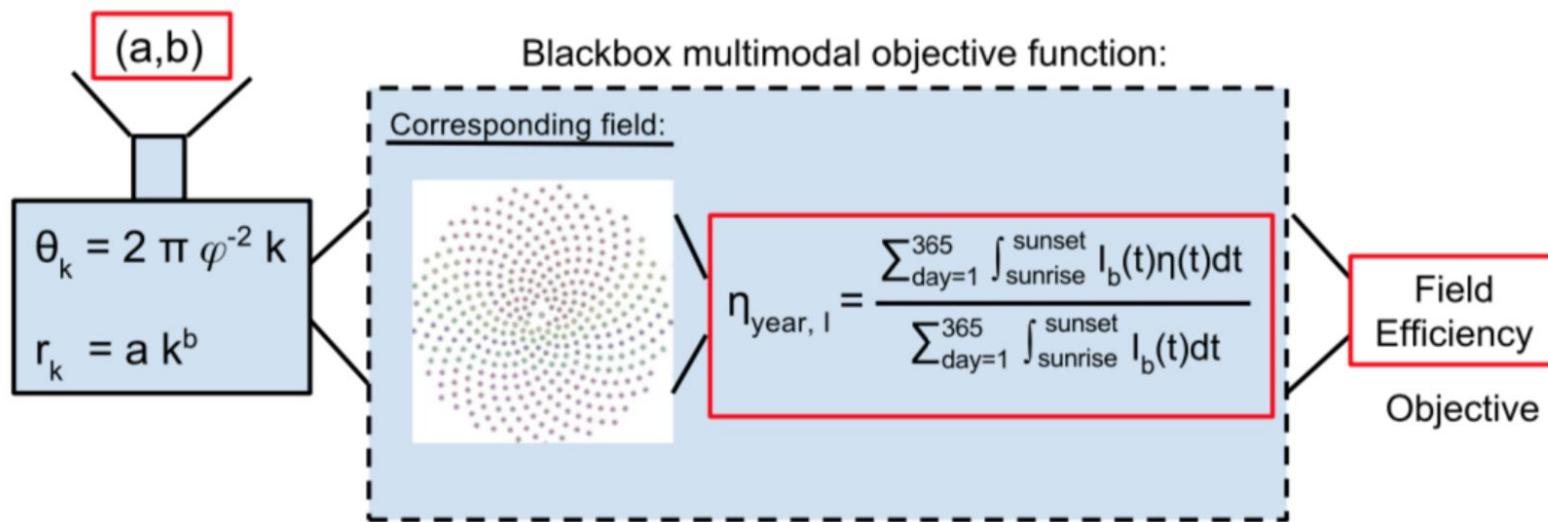
$$\text{subject to} \quad L_i \leq x_i \leq U_i, \quad i = 1, \dots, N$$

Si f tiene una forma analítica concreta y propiedades como linealidad y convexidad, estos problemas se pueden resolver fácilmente por métodos analíticos exactos. Sin embargo, esto no siempre ocurre: a veces sólo podemos evaluar (black-box), y no siempre es rápido.

6. Ejemplos prácticos: Optimizadores paralelos

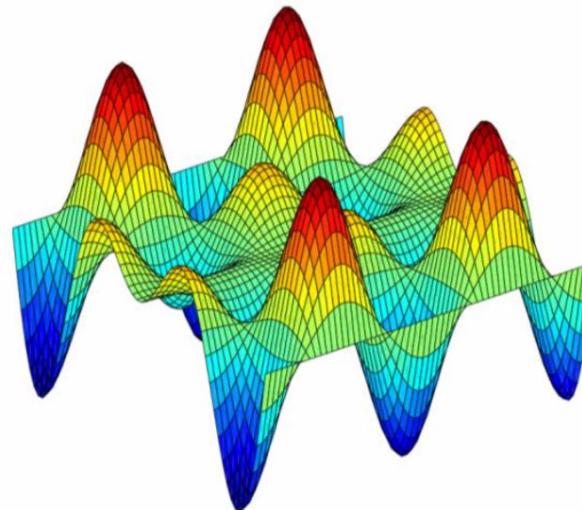
Optimización continua (IV)

Under optimization



6. Ejemplos prácticos: Optimizadores paralelos

Optimización continua (V)



Sin propiedades matemáticas que explotar, se aplican **heurísticas y meta-heurísticas**, que son **estrategias de resolución que** se sabe que **obtienen buenos resultados**, aunque no siempre se pueda demostrar su optimalidad.

6. Ejemplos prácticos: Optimizadores paralelos

Optimización continua (VI)

- Muchos de estos métodos se apoyan fuertemente en el uso de aleatoriedad (métodos estocásticos / Monte-Carlo) y en maximizar la capacidad de evaluación de soluciones por unidad de tiempo:
 - El **HPC** puede jugar un papel clave en este sentido, encajando muy bien con el modelo de la Ley de Gustafson (tiempo fijo).
- En este contexto destacan los métodos basados en poblaciones, como algoritmos genéticos (**evolutivos**) y los de optimización de conjunto de partículas (**inteligencia colectiva**).

6. Ejemplos prácticos: Optimizadores paralelos

Búsqueda Aleatoria Pura

- Sencilla
- **Muy paralelizable**
- Eficaz para problemas de pocas dimensiones

(Versión modificada para registrar múltiples puntos)

```
function [solutions, cycles] = ParExplorer(func, nVars, mode, killSwitchName)
    solutions = []; % As many rows as solutions
    cycles = 0;

    pool = gcp;
    numThreads = pool.NumWorkers; disp(['Using: ', num2str(numThreads), ' threads']);

    buffer = zeros(numThreads, nVars+1); % Where to store partial results

    while ~exist(killSwitchName, 'file') % Iterate indefinitely
        parfor i = 1:numThreads
            point = rand(1, nVars);
            val = func(point);
            buffer(i,:) = [point, val];
        end

        solutions = [solutions; buffer]; %#ok<AGROW>
        cycles = cycles + numThreads;
    end

    solutions = unique(solutions, 'rows');
    [~, indices] = sort(solutions(:, end), mode);
    solutions = solutions(indices, :); % Sort the output

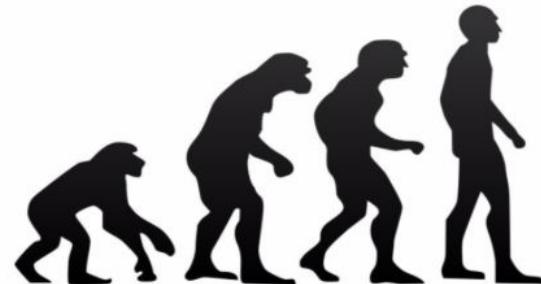
    delete(pool);
end
```

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmos evolutivos (I)

Gestionan un conjunto (población) de soluciones candidatas (individuos) y simulan su interacción y evolución siguiendo el modelo de Darwin.

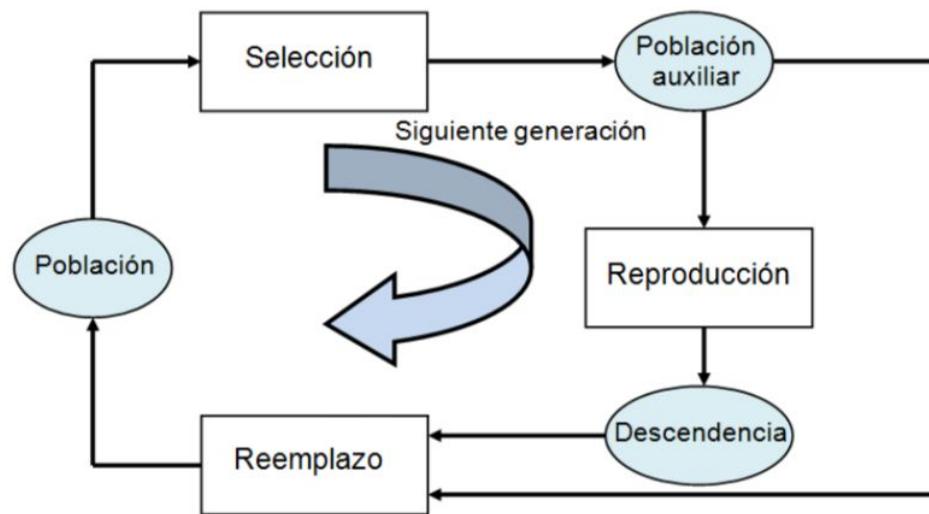
Punto	Valor
(0.1, ..., 0.75)	19
(...)	(...)
(0.9, ..., 1.0)	10



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmos evolutivos (II)

Ciclo básico de un algoritmo evolutivo (aunque realmente son muy modificables):

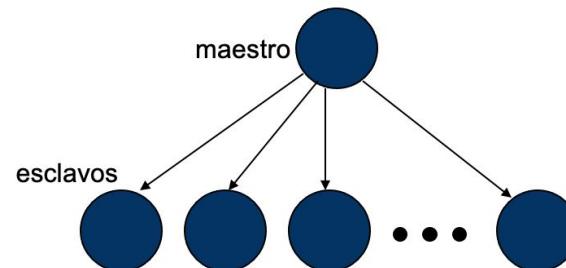


6. Ejemplos prácticos: Optimizadores paralelos

Algoritmos evolutivos (III)

El método más simple de parallelizar un algoritmo evolutivo es la “**parallelización global**”:

- Sólo hay una población, bajo responsabilidad final del hilo/proceso principal, pero la evaluación de individuos se paralleliza explícitamente, lo que sigue un **modelo maestro-esclavo**:



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmos evolutivos (IV)

El método más simple de paralelizar un algoritmo evolutivo es la “**paralelización global**”:

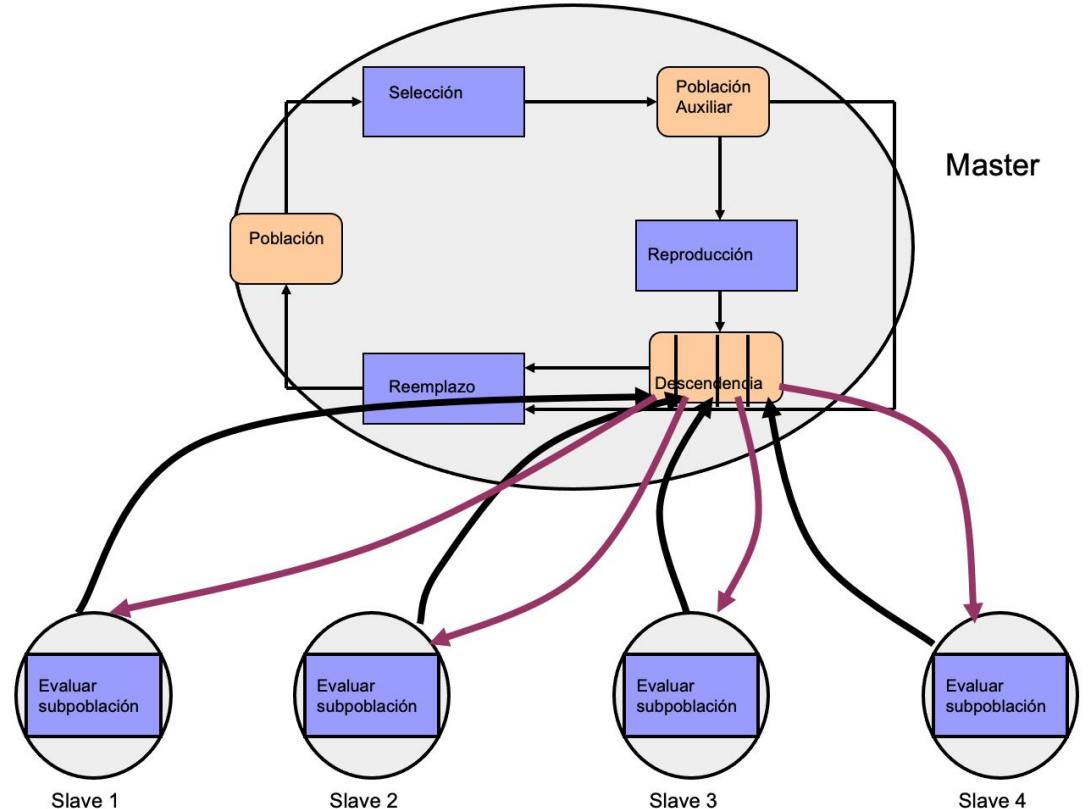
- En memoria distribuida: La comunicación sólo ocurre cuando el procesador **recibe** los individuos a evaluar y cuando **devuelve** los resultados.
- En memoria compartida: No hay comunicación explícita, pero sí necesidad de **puntos de sincronismo** (barreras y mutex).

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmos evolutivos (V)

- En memoria distribuida:
 - Más escalabilidad

Grano Grueso:
Menor ratio trabajo / comunicación



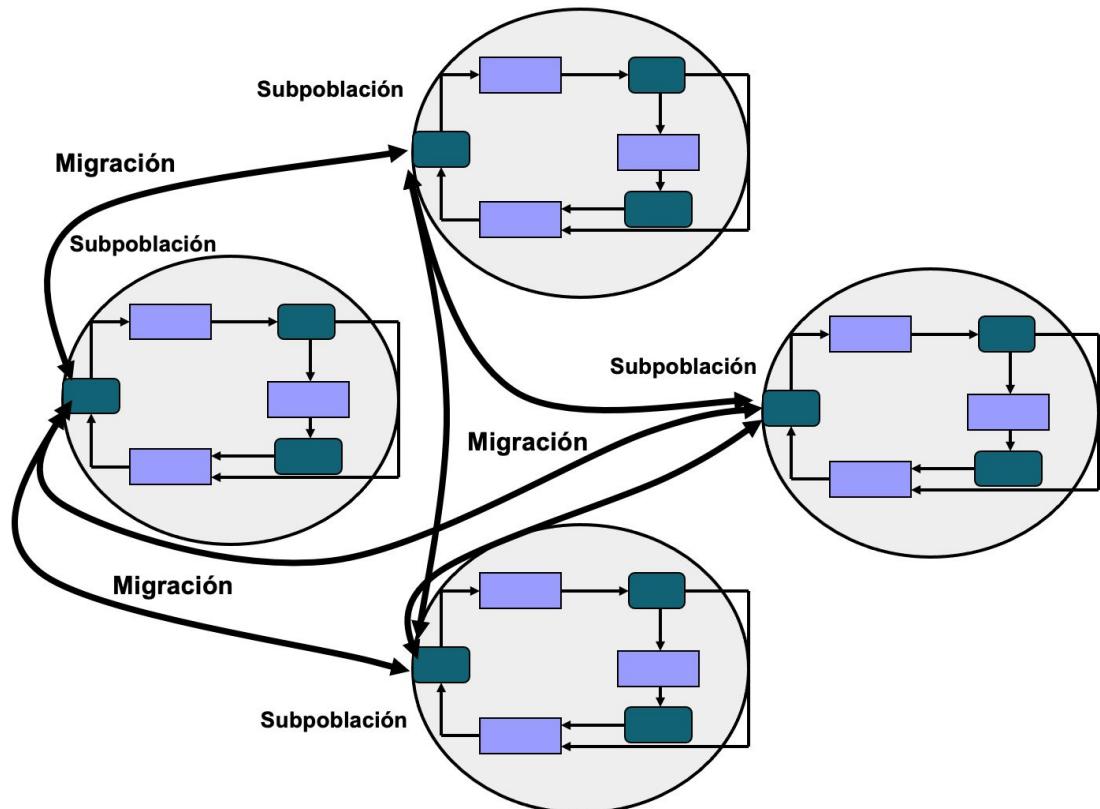
6. Ejemplos prácticos: Optimizadores paralelos

Algoritmos evolutivos (VI)

- En memoria distribuida:
 - Más escalabilidad

Grano Grueso:

Menos interacción... y
aún más escalabilidad
(para el algoritmo, no la
máquina)



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmos evolutivos (VII)

- En memoria compartida:
 - Menos overhead



- 1.- Create pool of **threads**
- 2.- Create **population** of solutions (**Knowledge?**)
- 3.- For $i = 1$ to **cycles**
- 4.- **Select** progenitors
- 5.- **Generate** descendants
- 6.- **Mutate** descendants
- 7.- **Replace** individuals
- 8.- **Return** the best solution

6. Ejemplos prácticos: Optimizadores paralelos

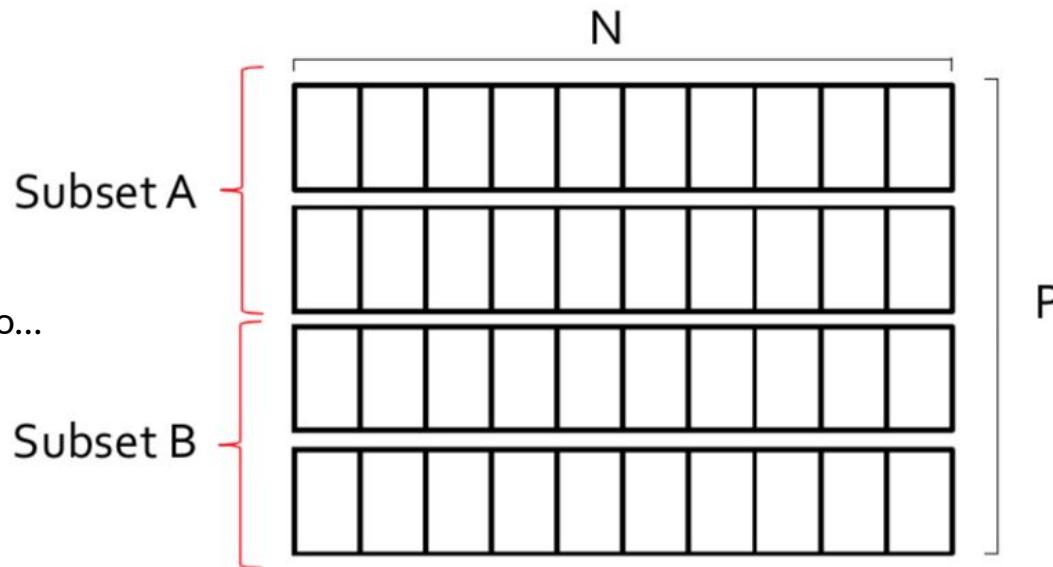
Algoritmos evolutivos (VIII)

- En memoria compartida:

- Menos overhead

Balanceo de carga estático...

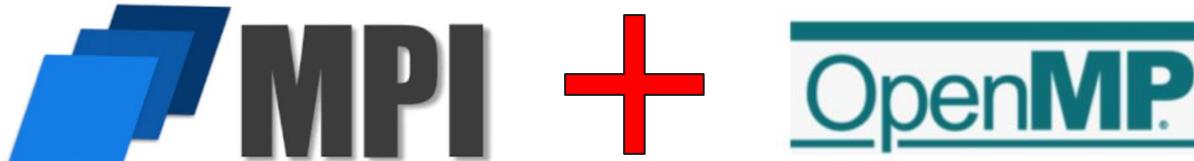
O dinámico:



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmos evolutivos (IX)

- También podemos combinar ambas estrategias:
 - Memoria Distribuida (gestión principal) & Memoria Compartida (auxiliar)



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Genético (I)

- Veamos un ejemplo de algoritmo genético en memoria compartida:

```
T* population = 0; T* new_population = 0; T* descendants = 0;//NULL  
this->getMemory(popSize, numPairs, &population, &new_population, &descendants);  
T best_solution;//DEFAULT->power = -inf <minimum possible>  
  
if(threads>numPairs)  
    threads = numPairs;//IT WOULD NOT MAKE SENSE TO LAUNCH MORE THREADS THAN PAIRS TO BE PROCESSED  
MersenneTwister RNG(seed);
```

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Genético (II) (Balanceo de carga estático)

```
#pragma omp parallel num_threads(threads)//Every variable created inside the parallel region is thread-local
{
    T local_result;//DEFAULT->power = -inf <minimum possible> To update it as a thread without critical

    uint32_t mySeed = 0;//Where to save the seed for my new local RNG
    #pragma omp critical(RAND) //rand() is not thread-safe, but we will force every thread to get its own RNG at t
    {
        mySeed = RNG(); //Using the initial MersenneTwister to create a seed for every local one
    }
    MersenneTwister randGen(mySeed);
    #ifdef _OPENMP //This is simply to allow non-OpenMP compilation(! -fopenmp)
        int rank = omp_get_thread_num(); //I would like to know what my number of thread is
    #else
        int rank = 0; threads = 1; //I am the only thread
    #endif
    int start = 0, end = 0, pairStart = 0, pairEnd = 0; //Let's compute my region of activity as a thread <POPULATI
    computeThreadLoad(popSize, threads, rank, start, end); //Population
    computeThreadLoad(numPairs, threads, rank, pairStart, pairEnd); //Descendants

    int local_best = initializePopulation(context, randGen, population, start, end);
    if(population[local_best].isBetterThan(local_result))
        local_result = population[local_best];
}
```



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Genético (III) (Balanceo de carga estático)

```
for(int i = 2*pairStart; i<(2*pairEnd); i=i+2){//Generating every pair:  
    this->selectProgenitors(randGen, popSize, population, tournSize, prog_A, prog_B);
```

```
#pragma omp barrier // We need to know that the whole population has been initialized before trying to operate with it (s  
for(int i = 0; i<numCycles; i++){//Evolutionary loop:  
    local_best = pairAndReproduce(context, randGen, population, popSize, pairStart, pairEnd, tournSize, descendants);  
    if(descendants[local_best].isBetterThan(local_result))  
        local_result = descendants[local_best];  
  
    local_best = mutate(context, randGen, descendants, pairStart, pairEnd, mutProb, perBitMutProb);  
    if(descendants[local_best].isBetterThan(local_result))  
        local_result = descendants[local_best];  
  
#pragma omp barrier //Before replacing and swapping, all threads must achieve this point (and single does not pro  
#pragma omp single  
{  
    this->replace(randGen, population, popSize, descendants, numPairs, tournSize, elite, new_population);// T  
    this->swap(&population, &new_population);  
} //Implicit BARRIER at the end of the single block -> Keep all threads at the same cycle  
}
```

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Genético (IV) (Balanceo de carga dinámico)

```
#pragma omp parallel num_threads(threads)//Every variable created inside the parallel region is thread-local
{
    T local_result;//DEFAULT->power = -inf <minimum possible> To update it as a thread without critical

    uint32_t mySeed = 0;//Where to save the seed for my new local RNG
    #pragma omp critical(RAND) //rand() is not thread-safe, but we will force every thread to get its own RNG
    {
        mySeed = RNG();//Using the initial MersenneTwister to create a seed for every local one
    }
    MersenneTwister randGen(mySeed);
    #ifdef _OPENMP //This is simply to allow non-OpenMP compilation(! -fopenmp)
        int rank = omp_get_thread_num(); //I would like to know what my number of thread is
    #else
        int rank = 0; threads = 1;//I am the only thread
                                //Include here knowledge-based solutions distribution)
    #endif

    int local_best = initializePopulation(context, randGen, population, popSize);//Parallel no-wait
    if(population[local_best].isBetterThan(local_result))
        local_result = population[local_best];
}

#pragma omp for schedule(dynamic) nowait//Think in this
for(int i = 0; i<popSize; i++){//If we inject the first
    population[i].randomSolution(randGen, context);}
```



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Genético (V) (Balanceo de carga dinámico)

```
#pragma omp for schedule(dynamic) //DO NOT ADD nowait (we cannot try to mutate descendants in advance
for(int i = 0; i<(2*numPairs); i=i+2){//Generating every pair:
    this->selectProgenitors(randGen, popSize, population, tournSize, prog_A, prog_B);
```

```
#pragma omp barrier // We need to know that the whole population has been initialized before trying to operate with it
for(int i = 0; i<numCycles; i++){//Evolutionary loop:
    local_best = this->pairAndReproduce(context, randGen, population, popSize, tournSize, descendants, numPairs);
    if(descendants[local_best].isBetterThan(local_result))
        local_result = descendants[local_best];
    //There is a barrier at the end of pairAndReproduce because we cannot know if descendants re-assigned for mutation
    local_best = mutate(context, randGen, descendants, numPairs, mutProb, perBitMutProb);//Parallel no-wait
    if(descendants[local_best].isBetterThan(local_result))
        local_result = descendants[local_best];

#pragma omp barrier //Before replacing and swapping, all threads must achieve this point (and single does not
#pragma omp single
{
    this->replace(randGen, population, popSize, descendants, numPairs, tournSize, elite, new_population);
    this->swap(&population, &new_population);
}//Implicit BARRIER at the end of the single block -> Keep all threads at the same cycle
}
```

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Genético (VI)

```
#pragma omp critical
{
    if(local_result.isBetterThan(best_solution))
        best_solution = local_result;
}
//Implicit barrier at the end of a parallel region <JOIN>

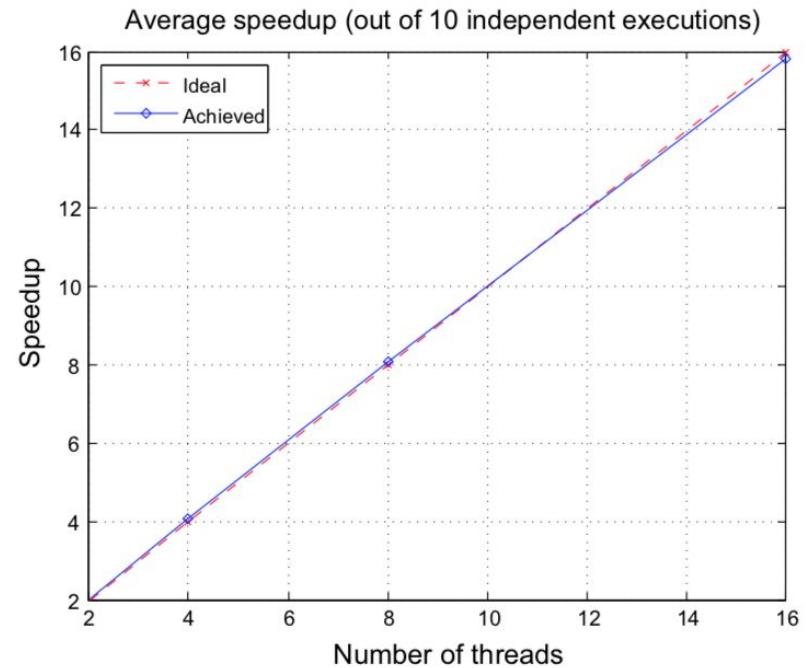
this->freeMemory(population, new_population, descendants);
return best_solution;
}
```

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Genético (VII)

Estático:

El dinámico quedaba algo por debajo pues el desbalanceo solución buena/mala duraba poco



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (I)

Un **algoritmo memético** es un optimizador basado en **poblaciones** en el que los individuos adquieren también un rol activo de auto-mejora (**búsqueda local**).

Un ejemplo es **UEGO**, “*Universal Evolutionary Global Optimizer*”, que define un procedimiento general de gestión de soluciones candidatas, y uno seleccionable de búsqueda local.

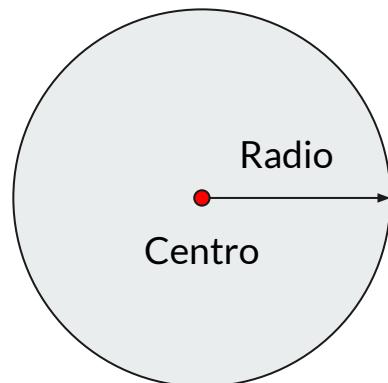
Es **multimodal**: Compatible con la identificación de múltiples óptimos

Se trata de un método **aplicado con éxito** en la búsqueda de fármacos, plegado de proteínas, diseño de campos de helióstatos, ajuste de modelos neuronales...

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (II)

Cada solución candidata o individuo es una “especie” para UEGO, y representa a una región del espacio de búsqueda:

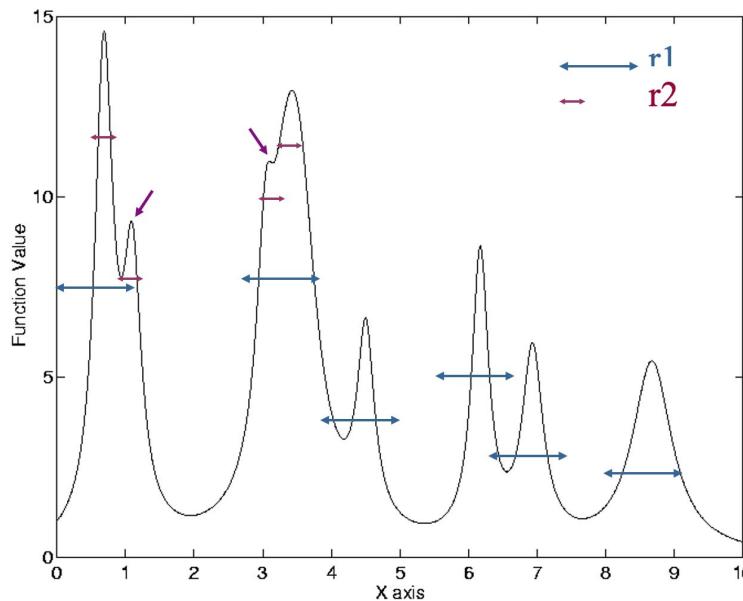


Cada especie se define por su centro y su radio de acción.

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (III)

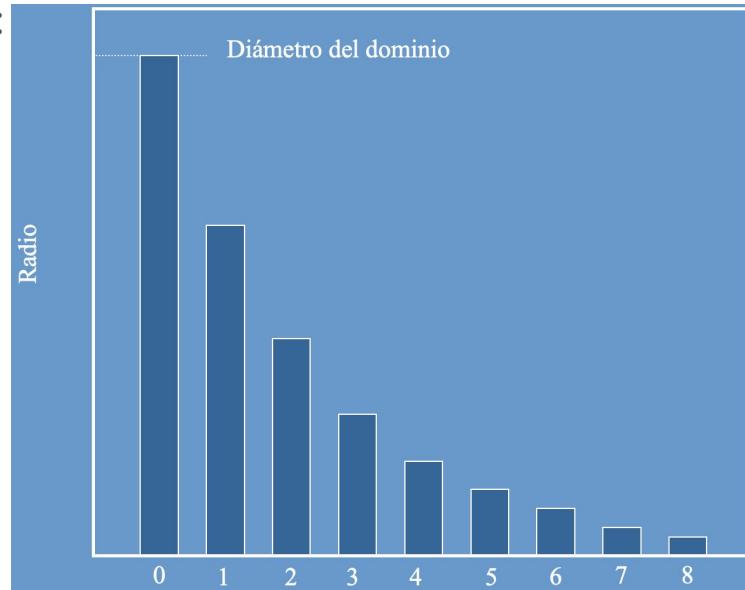
Las especies se distribuyen por el espacio, y los radios de las nuevas se van reduciendo (*cooling* o “enfriamiento”):



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (III)

Las especies se distribuyen por el espacio, y los radios de las nuevas se van reduciendo (*cooling* o “enfriamiento”):



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (IV)

Parámetros de **entrada**:

- N: Número máximo de evaluaciones
- I: Número de niveles (iteraciones)
- M: Número máximo de especies
- r: Radio mínimo (último nivel)

Información calculada para el nivel i:

- R_i : Radio del nivel i
- new_i : Máximo de evaluaciones para crear especies
- n_i : Máximo de evaluaciones para optimización local

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (V)

1. Inicializar lista de especies
2. Optimizar especies (n_1)
3. Desde $i=2$ hasta l
 - a. Determinar R_i , new_i , n_i
 - b. Crear especies (new_i) % Presupuesto por especie: $new_i / size(pop)$
 - c. Fundir especies (R_i)
 - d. Eliminar especies (M)
 - e. Optimizar especies (n_i) % Presupuesto por especie: n_i / M
 - f. Fundir especies (R_i)

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (VI)

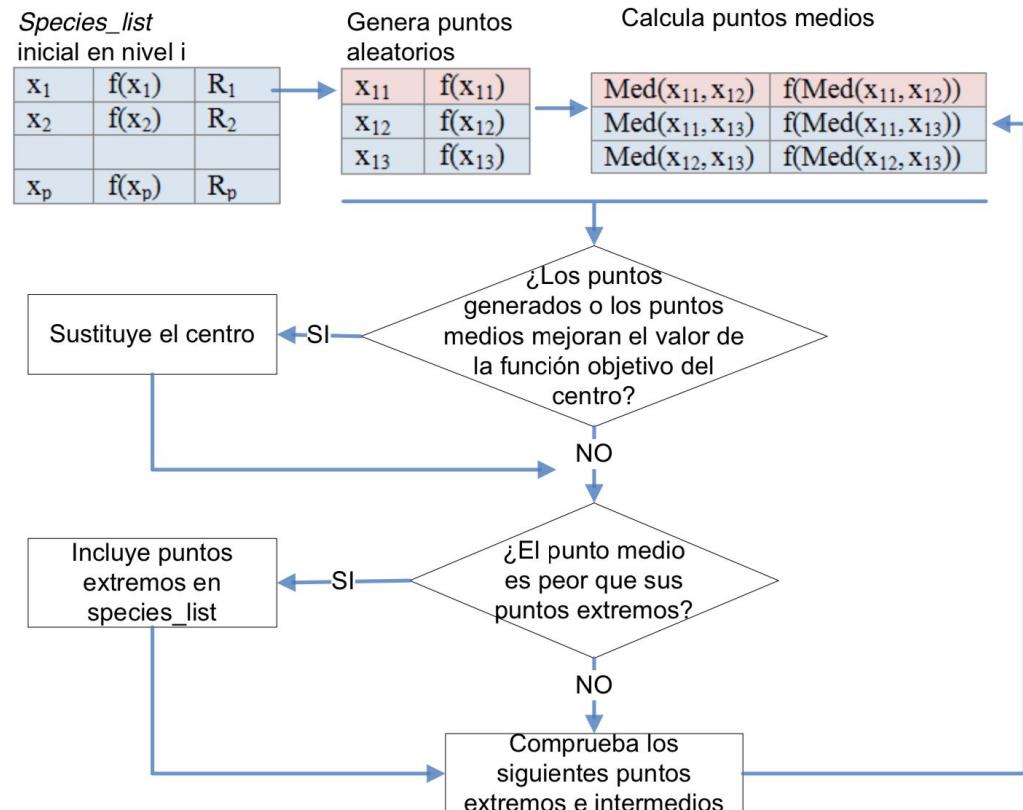
1. Inicializar lista de especies
2. Optimizar especies (n_1)
3. Desde $i=2$ hasta l
 - a. Determinar R_i , new_i , n_i
 - b. Crear especies (new_i)
 - c. Fundir especies (R_i)
 - d. Eliminar especies (M)
 - e. Optimizar especies (n_i)
 - f. Fundir especies (R_i)



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (VII)

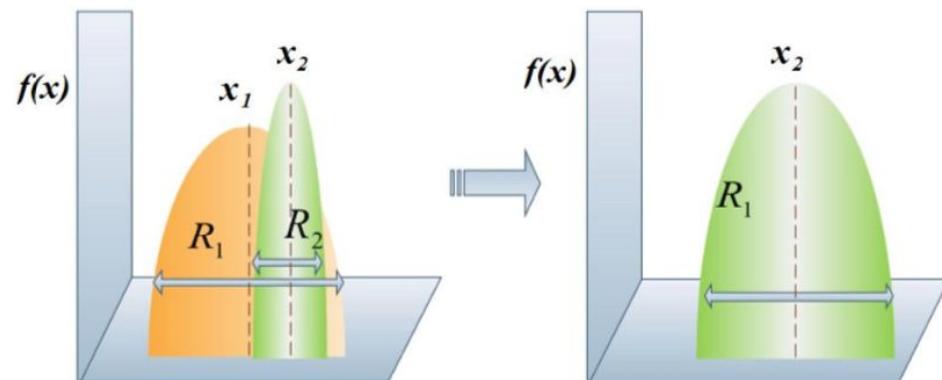
1. Inicializar lista de especies
2. Optimizar especies (n_1)
3. Desde $i=2$ hasta l
 - a. Determinar R_i , new_i , n_i
 - b. **Crear especies** (new_i)
 - c. Fundir especies (R_i)
 - d. Eliminar especies (M)
 - e. Optimizar especies (n_i)
 - f. Fundir especies (R_i)



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (VIII)

1. Inicializar lista de especies
2. Optimizar especies (n_1)
3. Desde $i=2$ hasta l
 - a. Determinar R_i , new_i , n_i
 - b. Crear especies (new_i)
 - c. **Fundir especies (R_i)**
 - d. Eliminar especies (M)
 - e. Optimizar especies (n_i)
 - f. **Fundir especies (R_i)**



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (IX)

1. Inicializar lista de especies
2. Optimizar especies (n_1)
3. Desde $i=2$ hasta l
 - a. Determinar R_i , new_i , n_i
 - b. Crear especies (new_i)
 - c. Fundir especies (R_i)
 - d. **Eliminar especies (M)**
 - e. Optimizar especies (n_i)
 - f. Fundir especies (R_i)



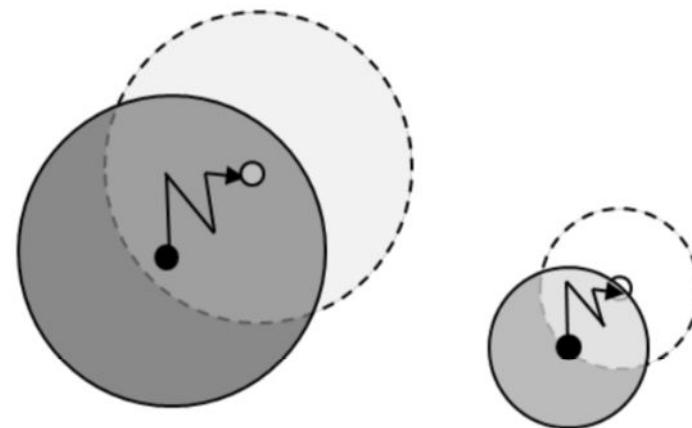
Si la lista es más grande de lo permitido, se elimina el excedente (empezando por los de mayor nivel (o menor radio)).

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (IX)

1. Inicializar lista de especies
2. **Optimizar especies (n_1)**
3. Desde $i=2$ hasta l
 - a. Determinar R_i , new_i , n_i
 - b. Crear especies (new_i)
 - c. Fundir especies (R_i)
 - d. Eliminar especies (M)
 - e. **Optimizar especies (n_i)**
 - f. Fundir especies (R_i)

Se lanza un optimizador local dentro de cada especie. Cuando encuentra un punto mejor que el centro, éste pasa a ser el nuevo centro y la especie se mueve.
Se suele usar **SASS**.



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (X)

Se suele usar **SASS**:

```

Parámetros de entrada: Scnt, Fcnt, ex, ct,  $\sigma_{ub}$ ,  $\sigma_{lb}$ , MaxItc
Seleccionar aleatoriamente  $x_0$  #inicializar vector búsqueda
 $b_0=0$  #inicializar vector parcial
 $k=1$ , scnt=0, fcnt=0,  $\sigma_0=1$ 
MIENTRAS  $k < \text{MaxItc}$  HACER:
    SI scnt> Scnt ENTONCES  $\sigma_k=ex \cdot \sigma_{k-1}$ 
    SI fcnt> Fcnt ENTONCES  $\sigma_k=ct \cdot \sigma_{k-1}$ 
    SI  $\sigma_{k-1} < \sigma_{lb}$  ENTONCES  $\sigma_k= \sigma_{lb}$ 
         $\xi_k=N(b_k, \sigma_k I)$  # generar vector aleatorio
        # gaussiano multivariado
    SI  $\Phi(x_k + \xi_k) < \Phi(x_k)$  ENTONCES
         $x_{k+1}=x_k + \xi_k$ 
         $b_{k+1}=0.4\xi_k+0.2b_k$ 
        scnt=scnt+1, fcnt=0
    SI NO
        SI  $\Phi(x_k - \xi_k) < \Phi(x_k) < \Phi(x_k + \xi_k)$  ENTONCES
             $x_{k+1}=x_k - \xi_k$ 
             $b_{k+1}=b_k-0.4\xi_k$ 
            scnt=scnt+1, fcnt=0
        SI NO
             $x_{k+1}=x_k$ 
             $b_{k+1}=0.5b_k$ 
            fcnt=fcnt+1, scnt=0
    k=k+1

```

Ascenso de colinas
estocástico adaptativo

6. Ejemplos prácticos: Optimizadores paralelos

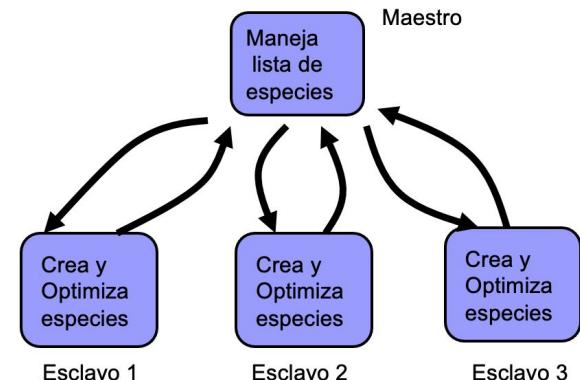
Algoritmo Memético (XI)

Paralelizando UEGO:

- **Estrategia Maestro-Esclavo con modelo global**
- El maestro maneja la lista de especies que distribuye entre los esclavos
- Los esclavos crean y optimizan las especies recibidas

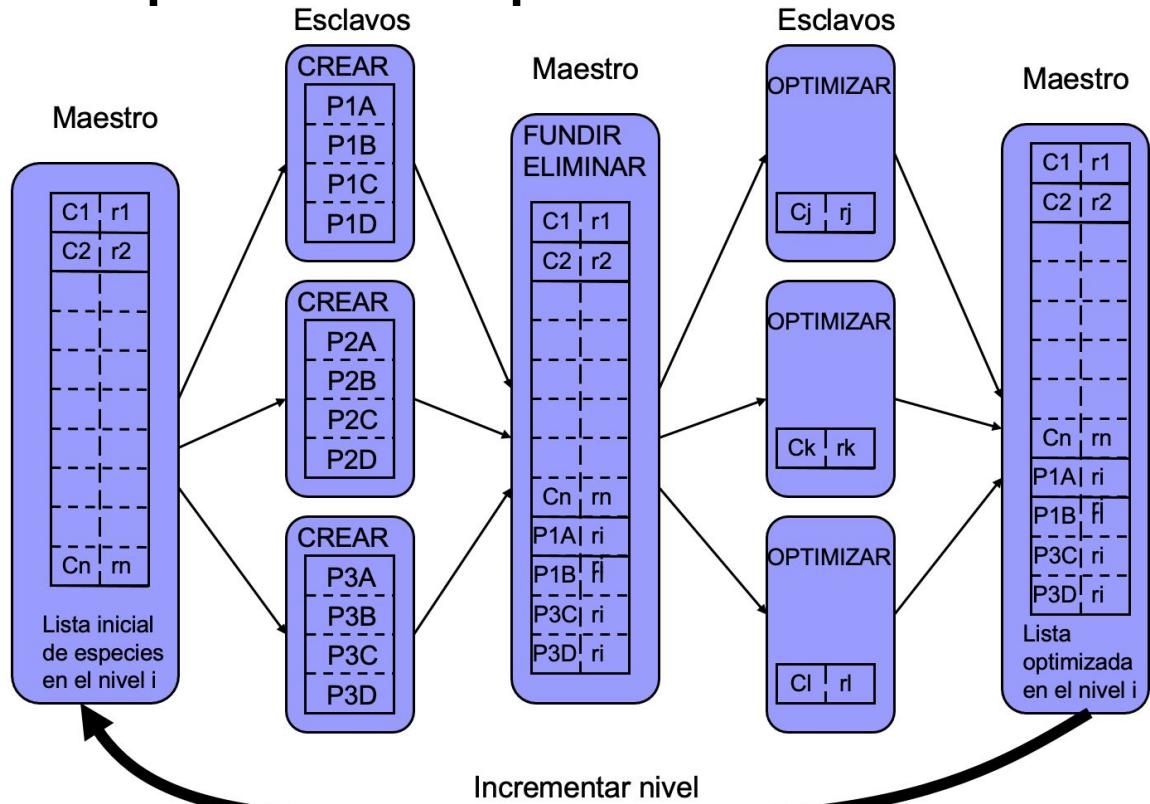
Dos implementaciones:

- Síncrona (PSUEGO)
- Asíncrona (PAUEGO)



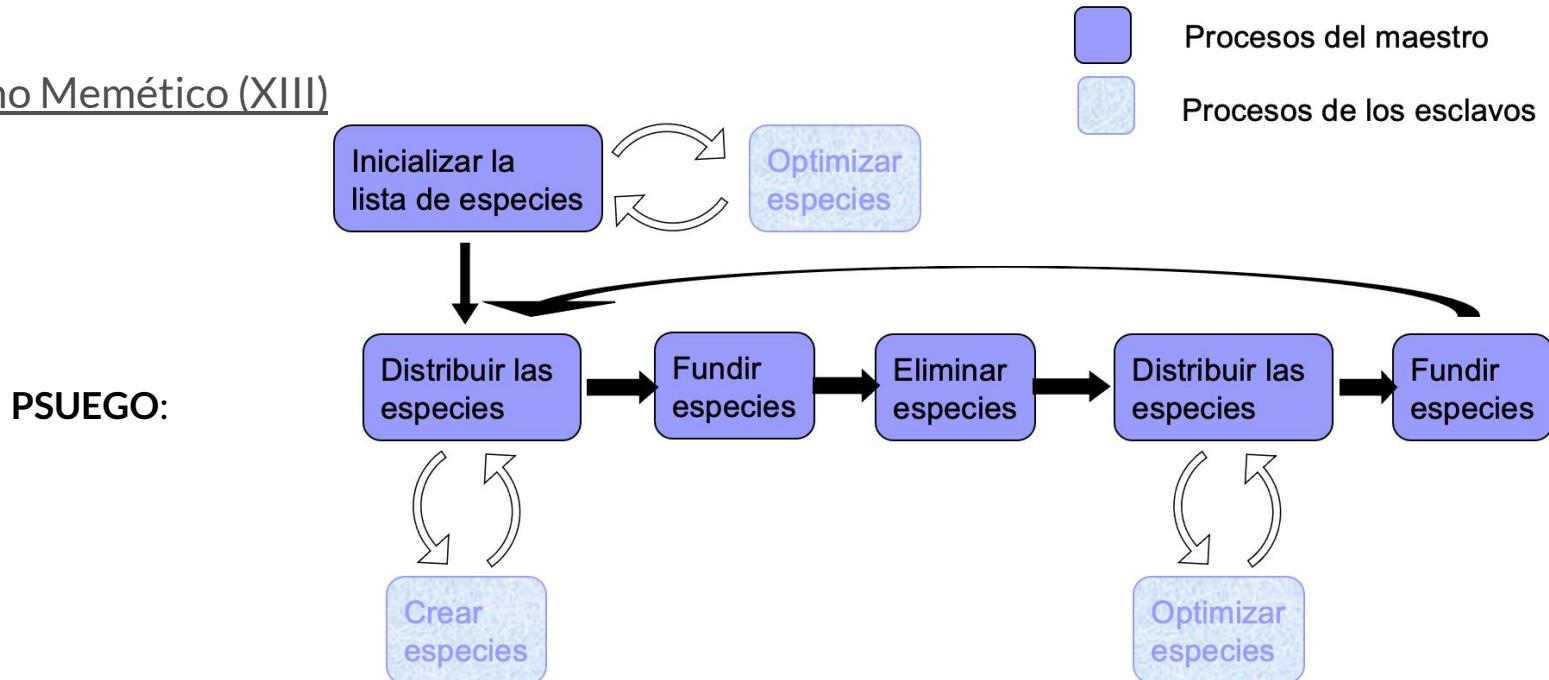
6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (XII)



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (XIII)



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (XIV)

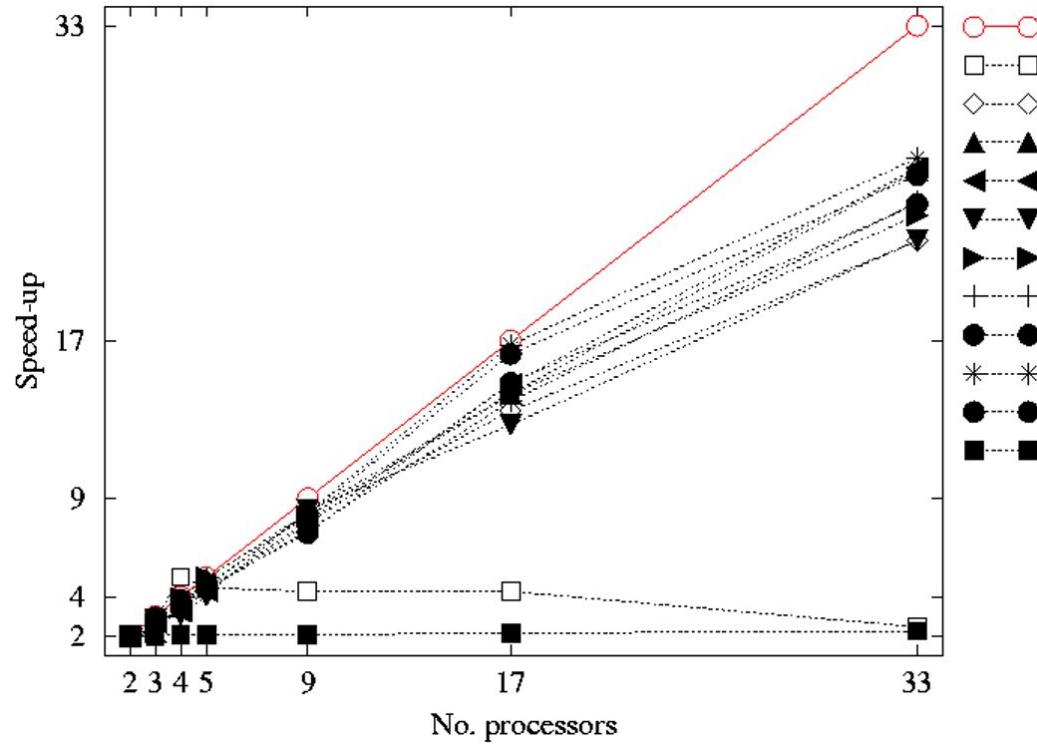
PSUEGO:

Función	% Tiempo en espera							
	NP=2	NP=3	NP=4	NP=5	NP=9	NP=17	NP=33	
F1	49	34	26	20	18	23	36	
F2	49	34	25	20	20	28	26	
F3	49	36	27	21	18	21	38	
F4	49	34	26	21	19	23	36	
F5	49	35	28	21	18	21	36	
F6	49	38	25	16	19	28	38	
F7	49	36	27	21	18	17	35	
F8	49	34	26	20	20	29	36	
F9	49	33	26	20	19	16	35	
F10	50	34	28	21	18	26	36	
F11	49	34	26	21	18	21	33	

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (XV)

PSUEGO:



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (XVI)

Problemas de PSUEGO:

- Puntos de sincronismo tras las etapas de creación y optimización
- El maestro permanece prácticamente parado durante la creación y la optimización
- La carga computacional no está bien balanceada

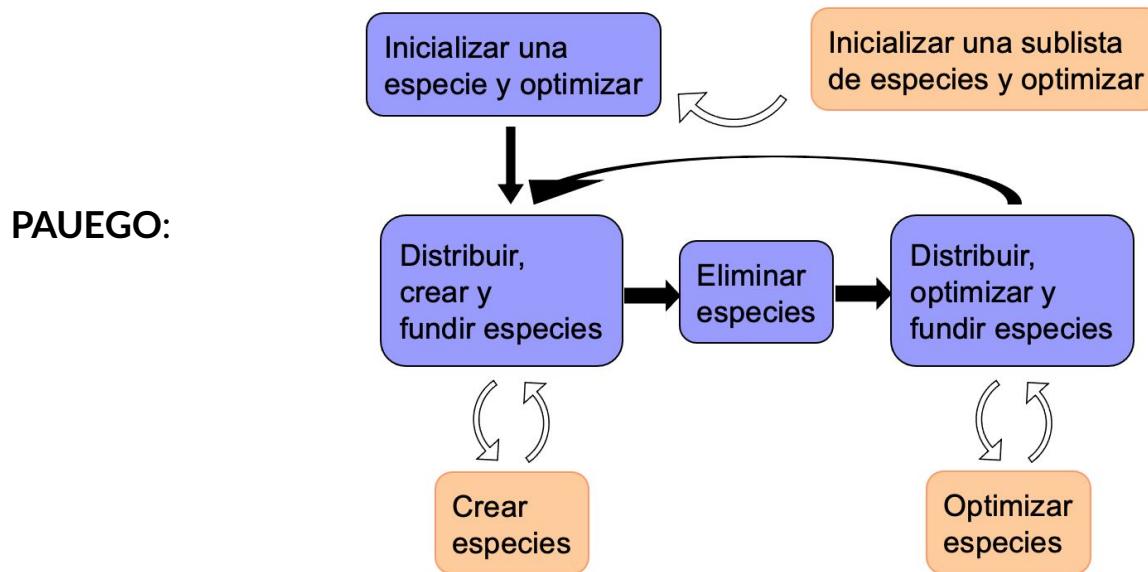
PAUEGO busca:

- Balancear mejor la carga: el maestro también participa en la creación y optimización
- Reducir esperas: el maestro comienza a fundir especies en cuanto empieza a recibirlas

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (XVII)

- Procesos del maestro
- Procesos de los esclavos



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (XVIII)

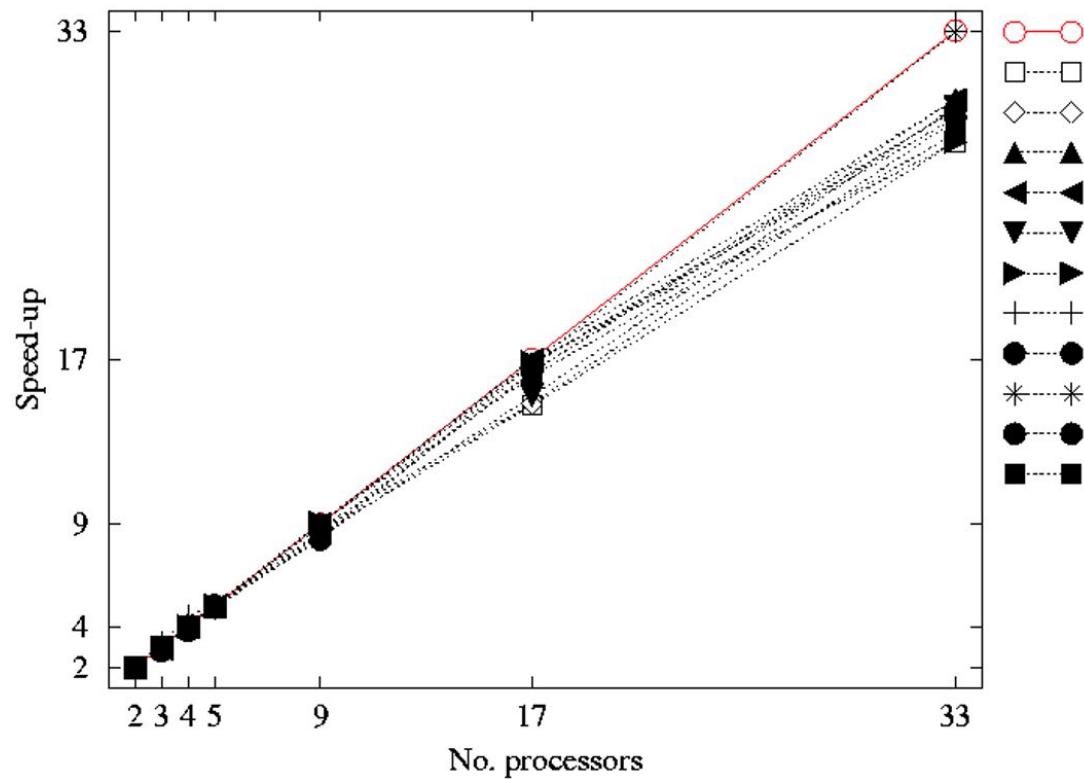
PAUEGO:

Función	% Tiempo en espera						
	NP=2	NP=3	NP=4	NP=5	NP=9	NP=17	NP=33
F1	0	1	1	2	2	3	4
F2	0	0	1	1	1	1	2
F3	0	0	1	2	2	2	4
F4	0	0	0	1	1	2	2
F5	0	1	1	0	1	1	2
F6	1	1	1	2	2	3	4
F7	0	1	1	0	1	1	2
F8	0	0	1	1	0	1	2
F9	0	0	1	1	0	1	1
F10	0	0	1	1	2	2	3
F11	0	0	1	0	1	2	2

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (XIX)

PAUEGO:



6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (XX)

- El número de evaluaciones de la función en el proceso de optimización varía para cada especie, de ahí que resulte muy difícil balancear la carga.
- En **PSUEGO**, el tiempo dedicado a las **comunicaciones** es de alrededor de un **40% del tiempo** de ejecución.
- En **PAUEGO**, este tiempo se reduce al **15%** del tiempo de ejecución.

¿Y una paralelización sobre GPU?

- Es complejo: las estructuras internas del algoritmo han de modificarse.

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (XXI)

UEGO sobre GPU:

- Prescindir de listas enlazadas y **apostar por matrices**
- **Desenrollar bucles** (aunque no demasiado: uso de registros, demasiadas tareas para el planificador...)

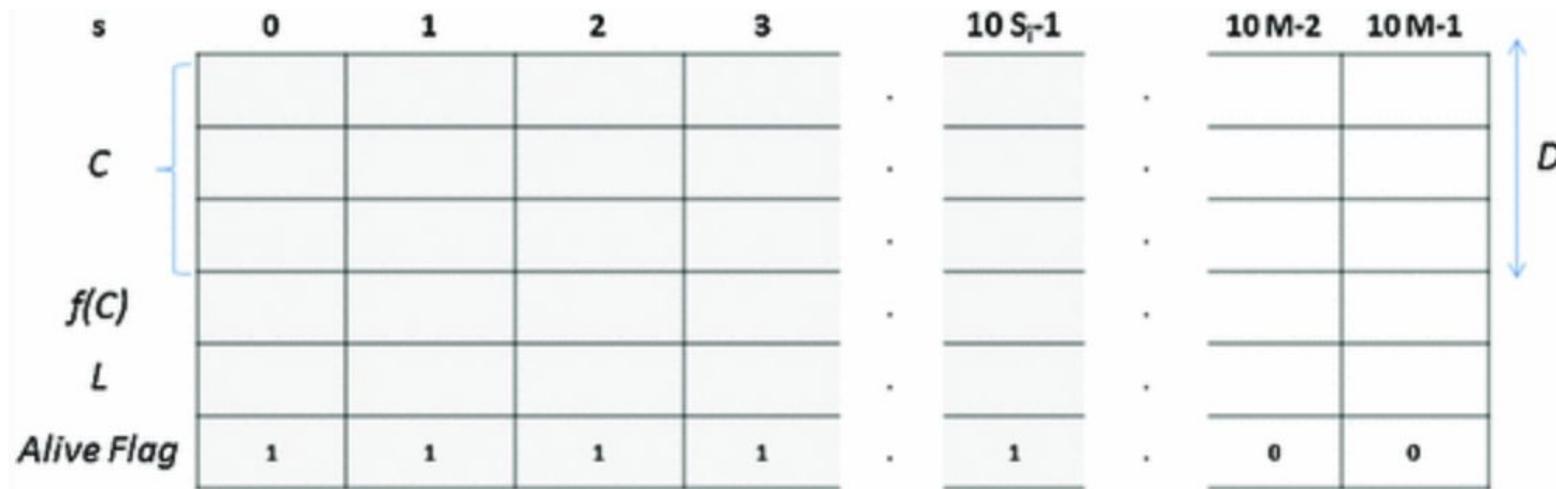
```
/* Antes del desenrollado */
for (i = 0; i < N; ++i) {
    c[i] = a[i] + b[i];
}
```

```
/* Despues del desenrollado */
for (i = 0; i < N - (4 - 1); i += 4) {
    c[i] = a[i] + b[i];
    c[i+1] = a[i+1] + b[i+1];
    c[i+2] = a[i+2] + b[i+2];
}
```

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (XXII)

Máximo reservado directamente
UEGO sobre GPU:

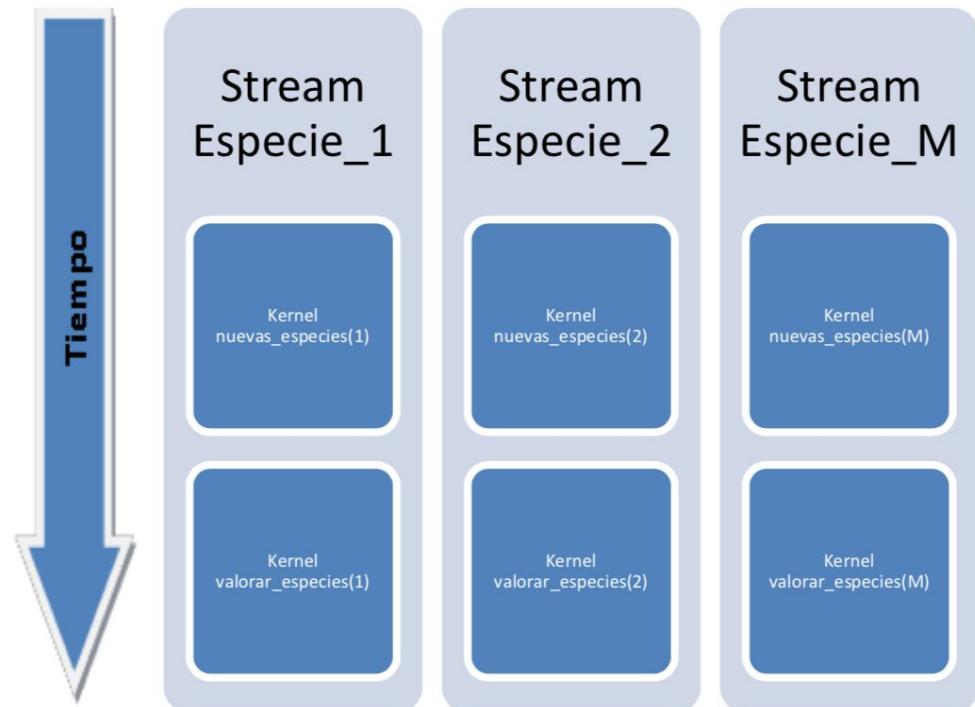


6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (XXIII)

UEGO sobre GPU:

```
streams = (cudaStream_t*)  
malloc(sizeof(cudaStream_t)*longitud_lista);  
for (int st = 0; st < longitud_lista; ++st) {  
    cudaStreamCreate (&streams[st]);  
}  
for (int s = 0; s < longitud_lista; ++s) {  
    nuevas_especies<<<(budget+(hilos-1))/hilos,  
        hilos,0,streams[s]>>>  
    (devStates, d_puntos, s, i+1, budget, D, M, d_nivel);  
    valorar_especies<<<(budget+(hilos-1))/hilos,  
        hilos,0,streams[s]>>>  
    (s, budget, M, d_valor, d_nivel, d_orden);  
}
```



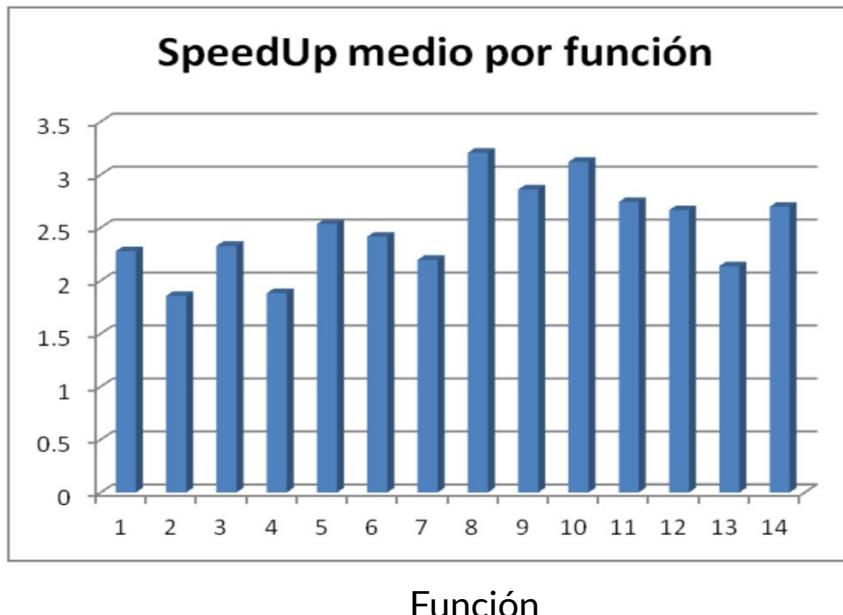
Y también un optimizador local más paralelizable que SASS!

6. Ejemplos prácticos: Optimizadores paralelos

Algoritmo Memético (XXIV)

UEGO sobre GPU:

Aceleración

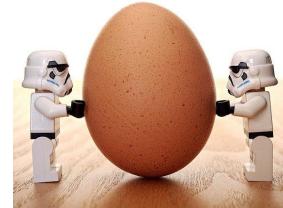


La aceleración es baja comparada con la multi-CPU:

- ¿Funciones objetivo más costosas?
- ¿Mayores poblaciones?

...

Trabajo para la próxima semana



Cada grupo debe buscar y exponer una aplicación/problema en el que la computación basada en GPU's resulte beneficiosa y superior al paralelismo sobre CPU.

=> Un miembro del grupo presentará su trabajo la próxima semana. Habrá que entregar tanto la presentación como una transcripción de lo que se dice (incluyendo las fuentes consultadas).

Importante: Poneos de acuerdo entre grupos para no escoger el mismo caso.



—

Gracias.



Tema 3

Redes de Área de Sistema

Nicolás Calvo Cruz

Dpto. de Arquitectura y Tecnología de los Computadores

@nkalvocruz

nkalvocruz@ugr.es

Motivación

- Redes de **comunicación en computadores paralelos**.
- Hoy en día se están **sustituyendo los buses por redes** con conexiones **punto a punto** a todos los niveles:
 - Interno al chip
 - A nivel de tarjeta y placa
 - A nivel de chasis o caja
 - LAN y Router IP
- Conocer los algoritmos de **encaminamiento** y la **infraestructura** permite mejorar las **prestaciones**.

Objetivos

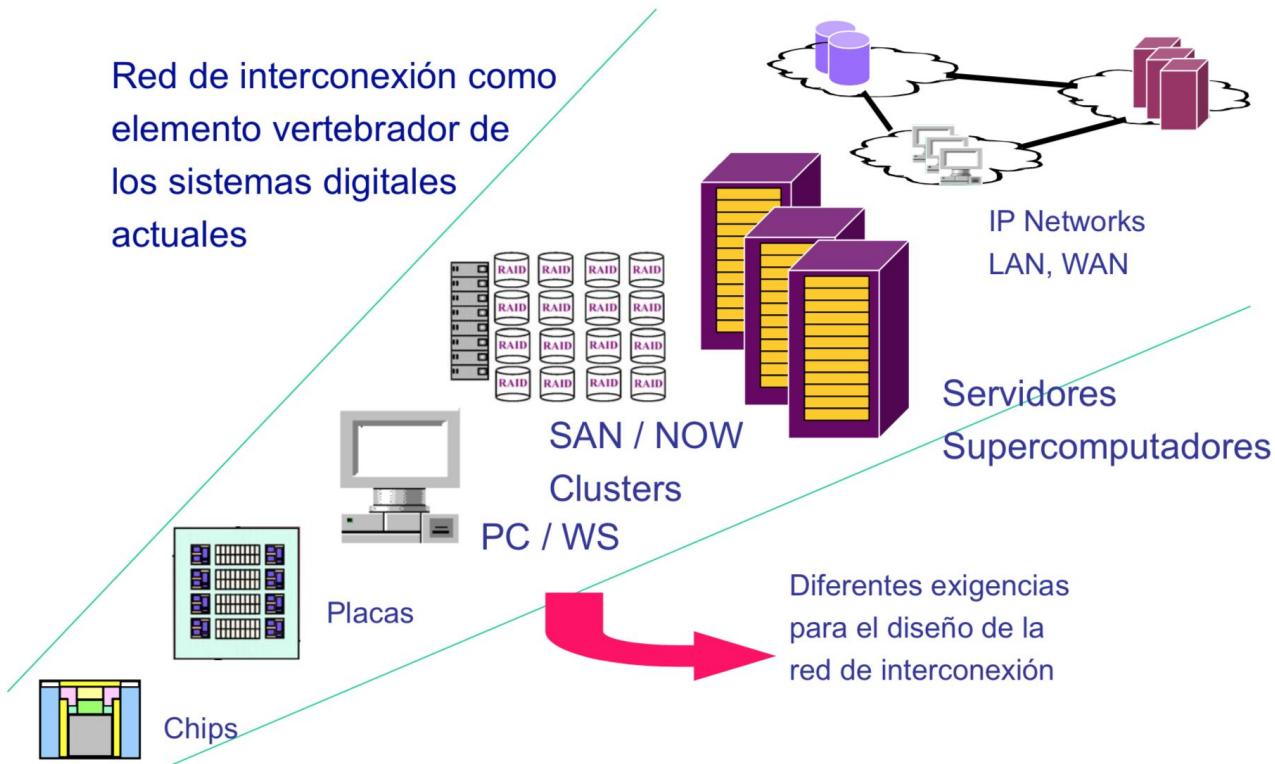
- Distinguir entre redes de altas prestaciones y redes estándar.
- Conocer la estructura general de un conmutador.
- Estudiar las topologías y nomenclaturas de las redes de altas prestaciones.
- Estudiar los algoritmos de encaminamiento.



Índice

1. Clasificación de Sistemas de Comunicación
2. Propiedades
3. Diseñar una Red
4. Prestaciones
5. Enrutamiento
6. Técnicas de conmutación
7. Ejemplo

1. Clasificación de Sistemas de Comunicación (I)

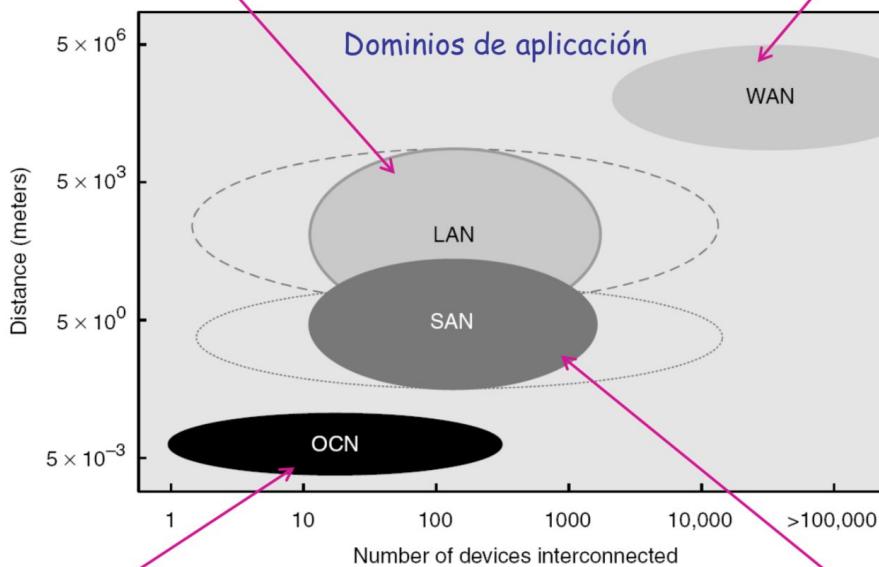


1. Clasificación de Sistemas de Comunicación (II)

Clases	Nº Nodos y Distancia	Utilización	Desarrollo	Ejemplos
Diseñadas a medida	pocos-cientos-miles decenas o cientos de metros	Multiprocesadores y Multicomputadores	Arquitecturas de Altas prestaciones	Cray's Aries Mellanox's Infiniband network interconnect Infiniband, Myricom
SAN: System Area Network	decenas-cientos-miles distancias desde decenas a cientos de metros	Conecta computadores en una sala/habitación	Redes a medida y LAN	
LAN	cientos-miles decenas de Km	Conecta computadores en un edificio o campus	Estaciones de Trabajo	Estándares: Fast Ethernet, Gigabit Ethernet
WAN	miles kilómetros	Conecta computadores a nivel mundial	Telecomunicaciones	Estándares: ATM

1. Clasificación de Sistemas de Comunicación (III)

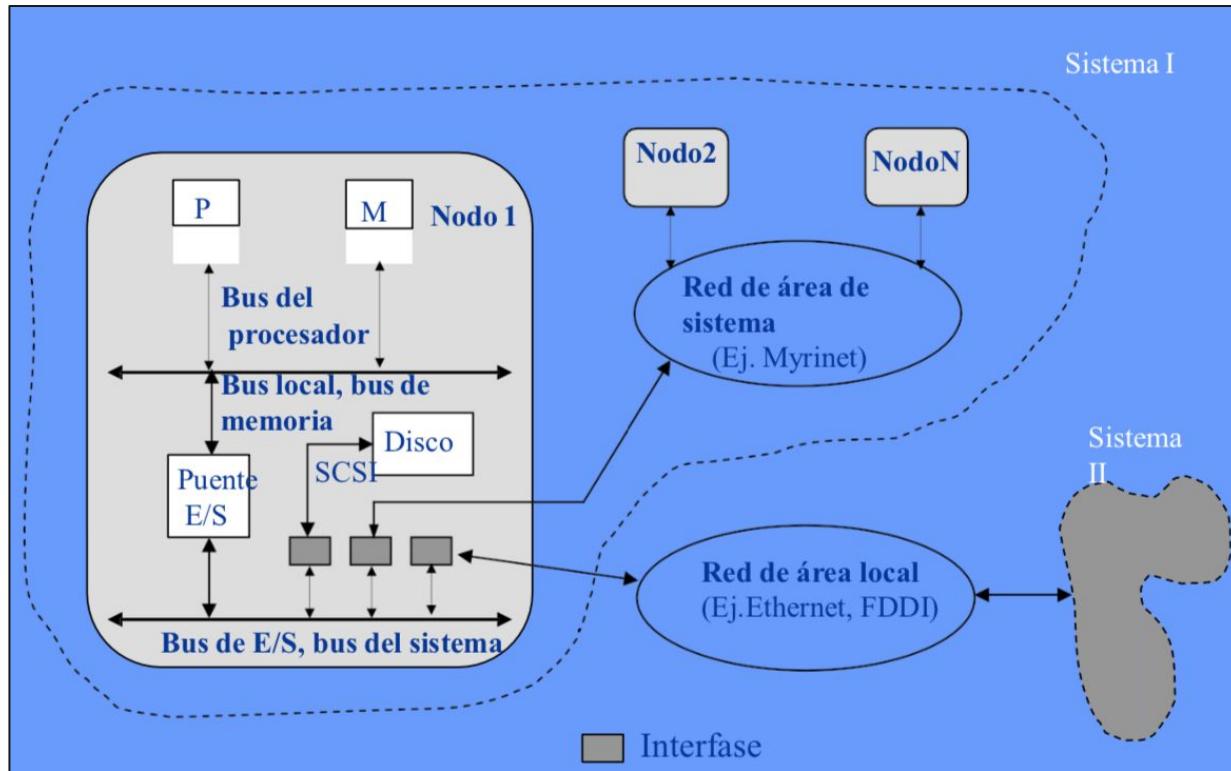
- Local Area Network (LAN). Ej: Ethernet (hasta 10 Gbps sobre una distancia de 40 Km). Tb Clusters de PCs
- Wide Area Network (WAN). Conexión a escala global. Ej: ATM



- Network-on-Chip (NoC, OCN). Ej: AMBA de ARM, SCCC de Intel, TILE-GX de Tilera
- System/Storage Area Network (SAN). Comunicación proc-mem-discos. Ej: Infiniband (hasta 200 Gbps sobre una distancia de 300 m)

1. Clasificación de Sistemas de Comunicación (IV)

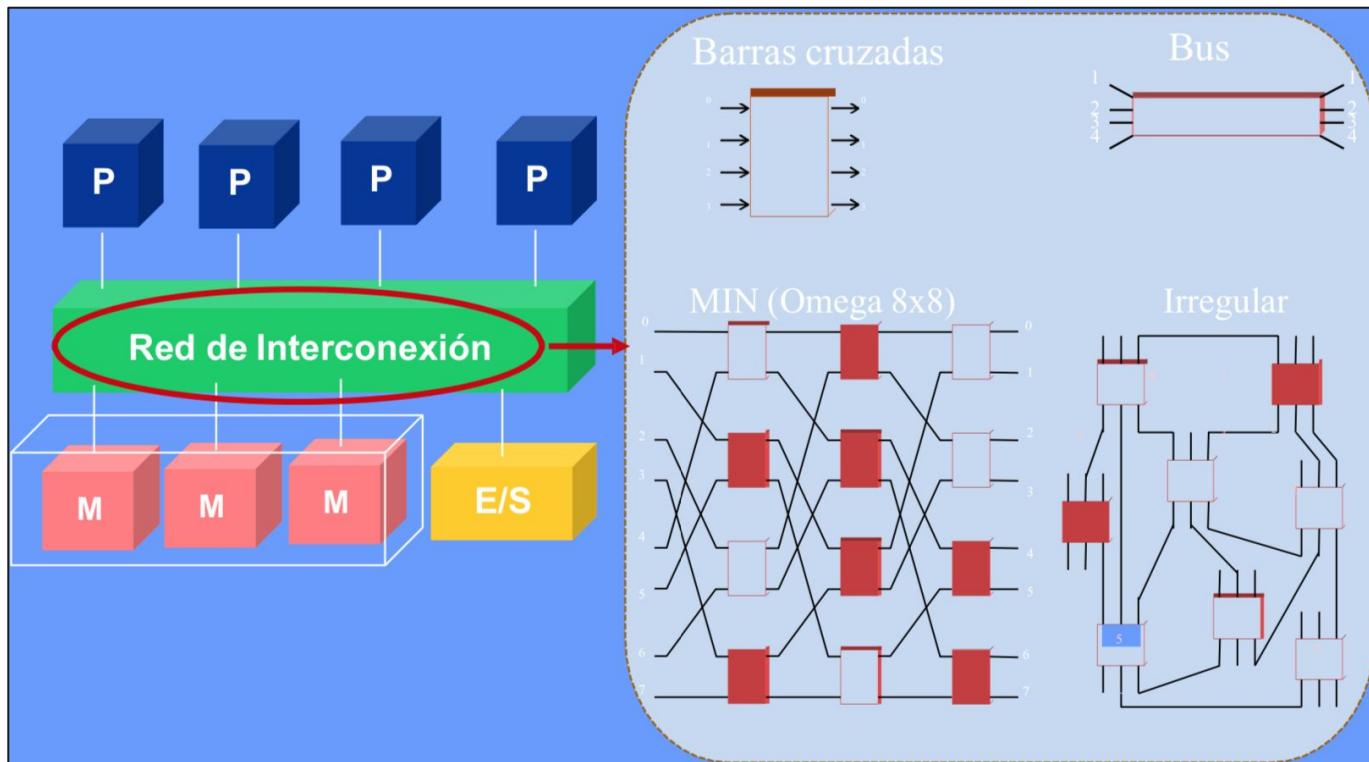
Se relacionan:



1. Clasificación de Sistemas de Comunicación (V)

En HPC:

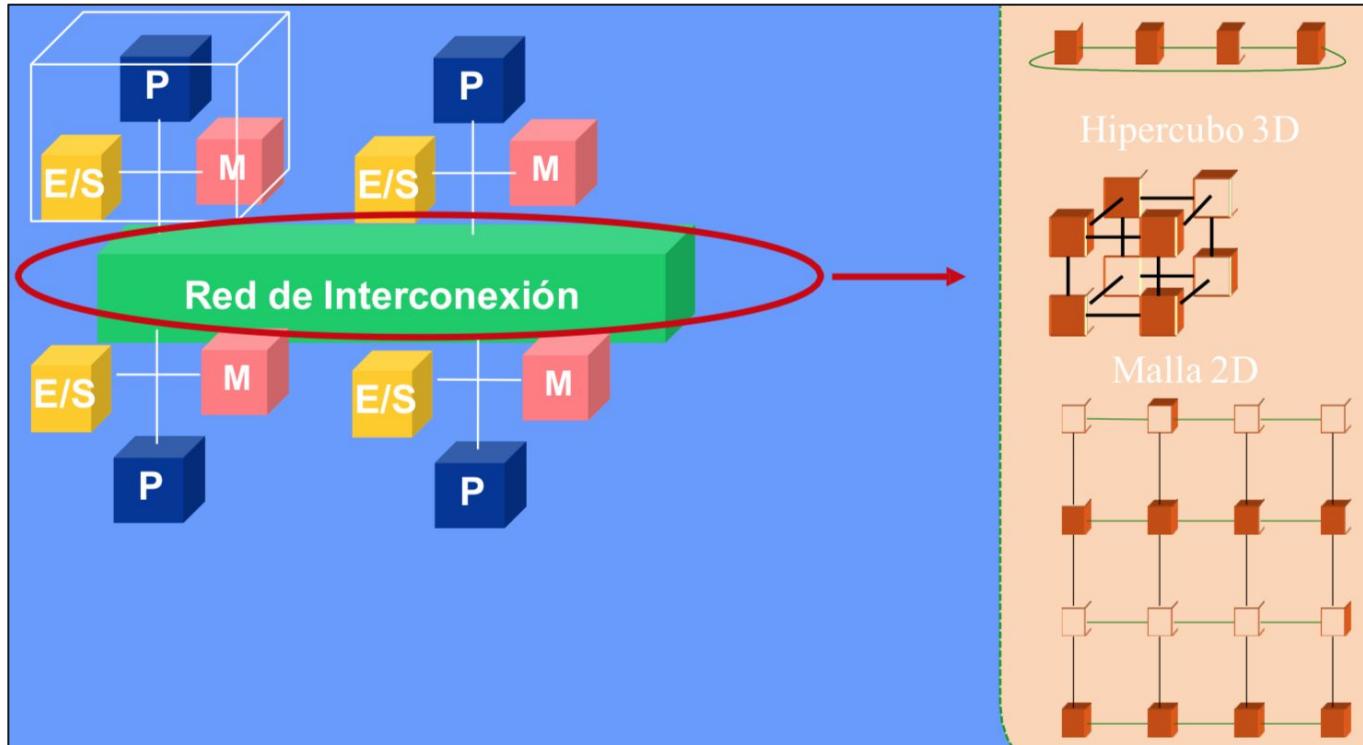
Multiprocesador
de memoria
centralizada



1. Clasificación de Sistemas de Comunicación (VI)

En HPC:

Multicomputador



2. Propiedades

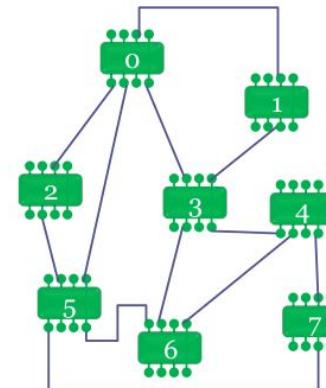
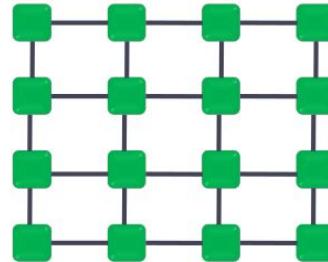
1. Funcionalidad
2. Topología
3. Características:
 - a. Diámetro de una red
 - b. Ancho de la bisección
 - c. Latencia
 - d. Productividad
 - e. Escalabilidad
 - f. Grado de los nodos
 - g. Niveles de servicio
 - h. Calidad de servicio
 - i. Alta disponibilidad
 - j. Tolerancia a fallos
 - k. Fiabilidad
 - l. Remote Direct Memory Access

2. Propiedades: Funcionalidad

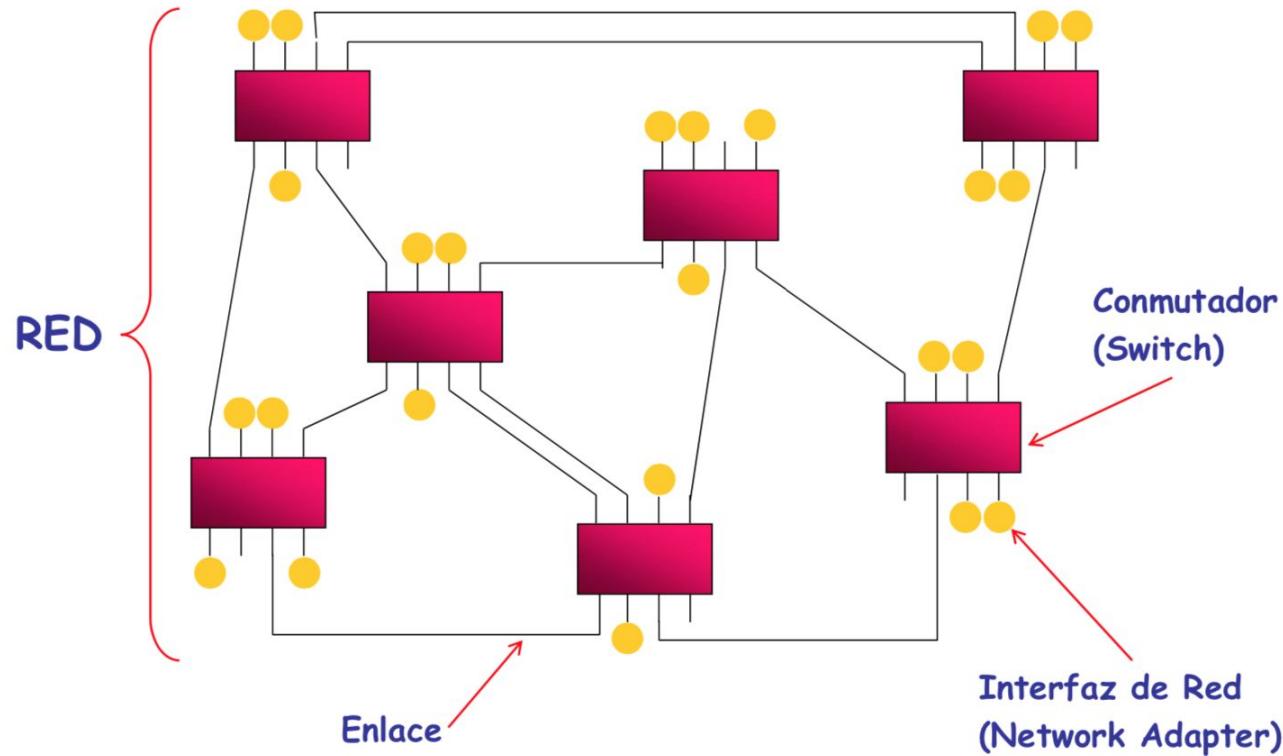
- La Funcionalidad de una red de comunicación **define su uso**.
- Según su funcionalidad encontramos:
 - Redes para **almacenamiento**
 - Redes para **computación**
 - Redes de **administración**

2. Propiedades: Topología (I)

- La topología es la estructura de la interconexión física de la red.
- Se puede modelar mediante un grafo en el que los vértices son conmutadores o interfaces de red (NI) y las aristas son conexiones.

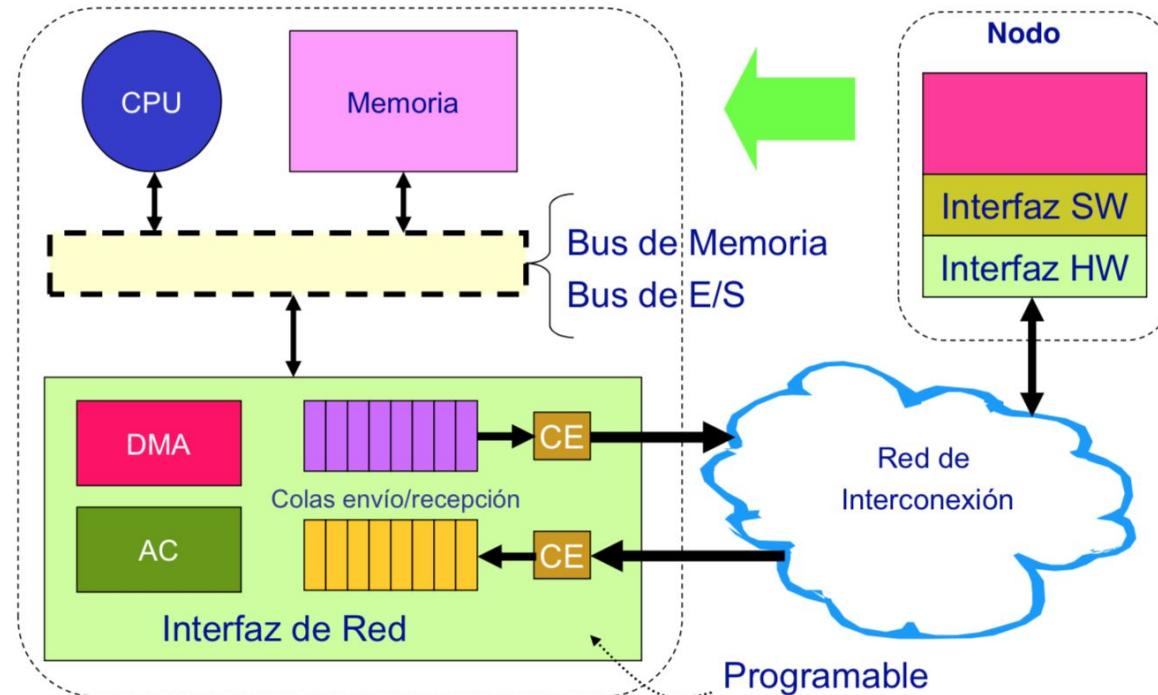


2. Propiedades: Topología (II)



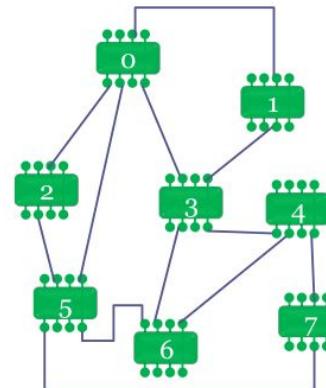
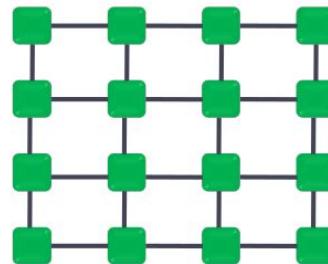
2. Propiedades: Topología (III)

Esquema básico de un interfaz de red



2. Propiedades: Topología (IV)

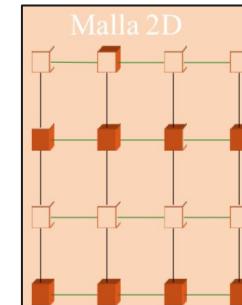
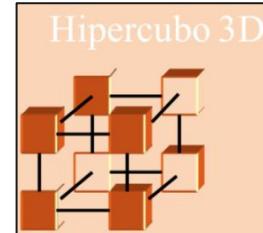
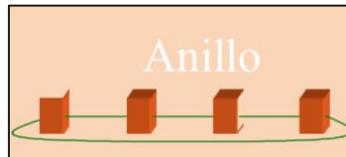
- Clasificación de redes según la topología de los enlaces:
 - Regulares: Son las redes en las que los enlaces siguen un patrón regular.
 - Irregulares: Son las redes en las que los enlaces no siguen un patrón regular.

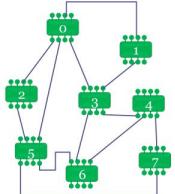


2. Propiedades: Topología (V)

- Clasificación de redes según la rigidez de los enlaces:

- Estáticas (o Directas): El conmutador (switch) está en la Interfaz de Red (NI) y los enlaces son rígidos, no se pueden adaptar. Se subdividen en:
 - Estrictamente ortogonales (anillos, mallas, toros, hipercubos...)
 - No ortogonales (árboles, estrella)





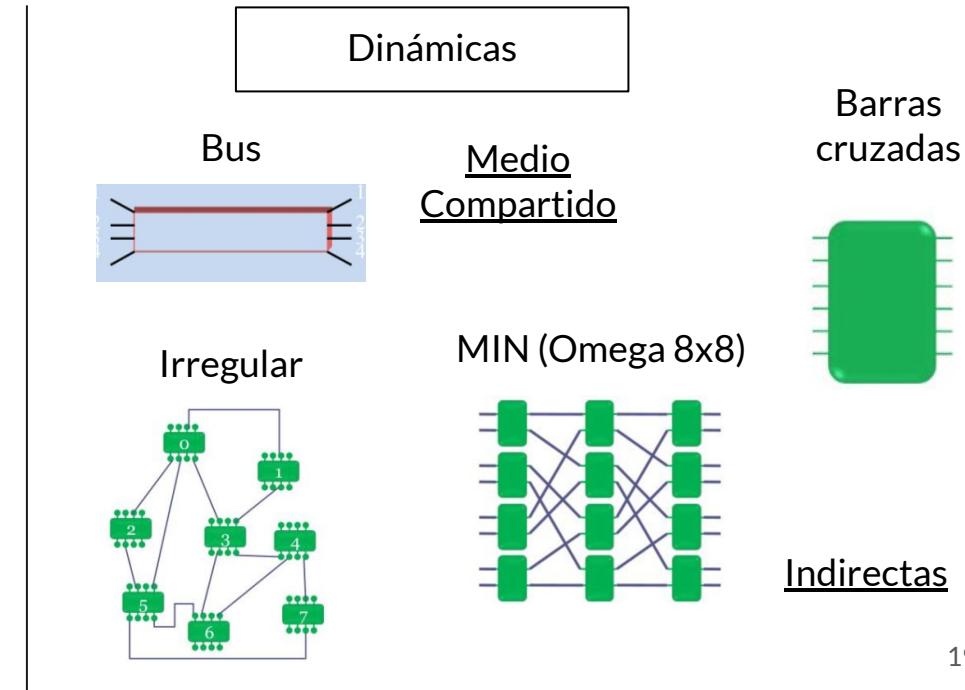
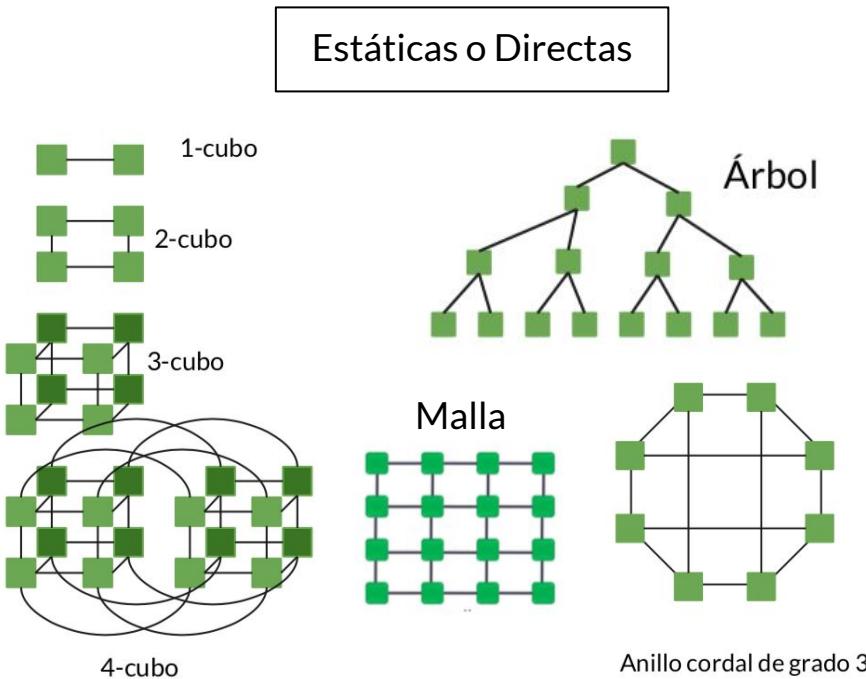
2. Propiedades: Topología (VI)

- Clasificación de redes **según la rigidez de los enlaces:**
 - **Dinámicas:** La topología no es rígida y se puede adaptar a las circunstancias. La adaptación se puede llevar a cabo con commutadores:
 - Medio compartido (buses)
 - Medio no compartido (o Indirectas)
 - Irregulares
 - Barras Cruzadas
 - Multietapa (bloqueantes, no bloqueantes, reconfigurables)



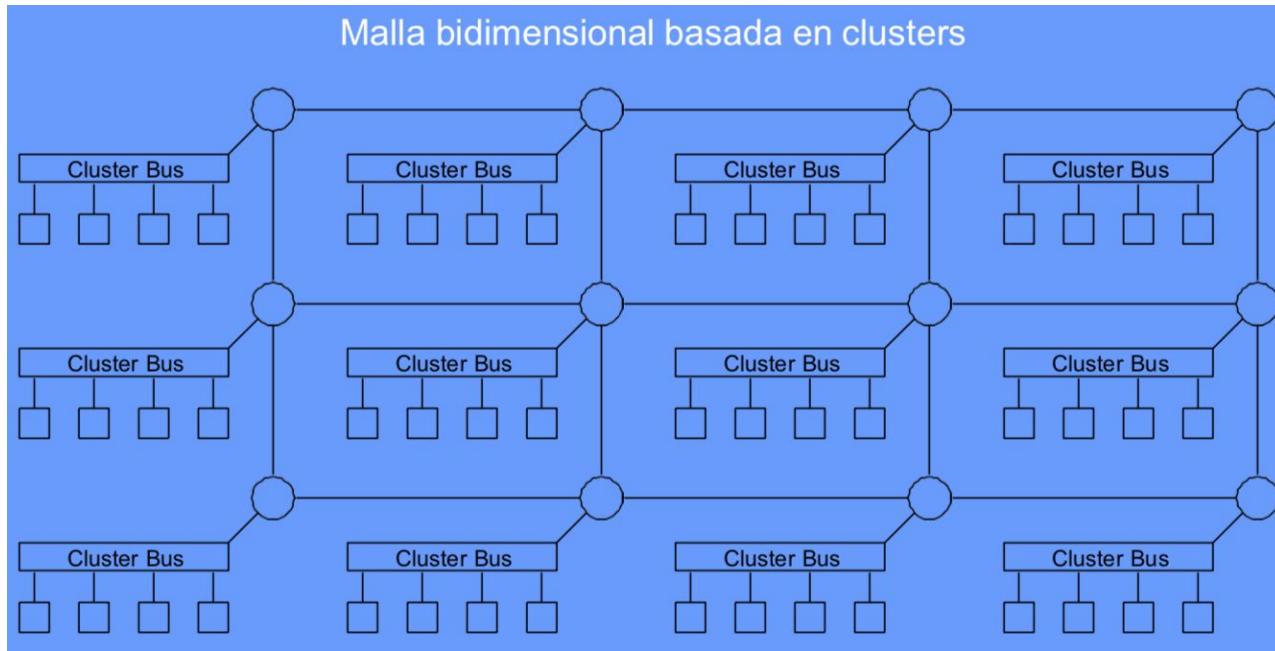
2. Propiedades: Topología (VII)

- Clasificación de redes según la rigidez de los enlaces:



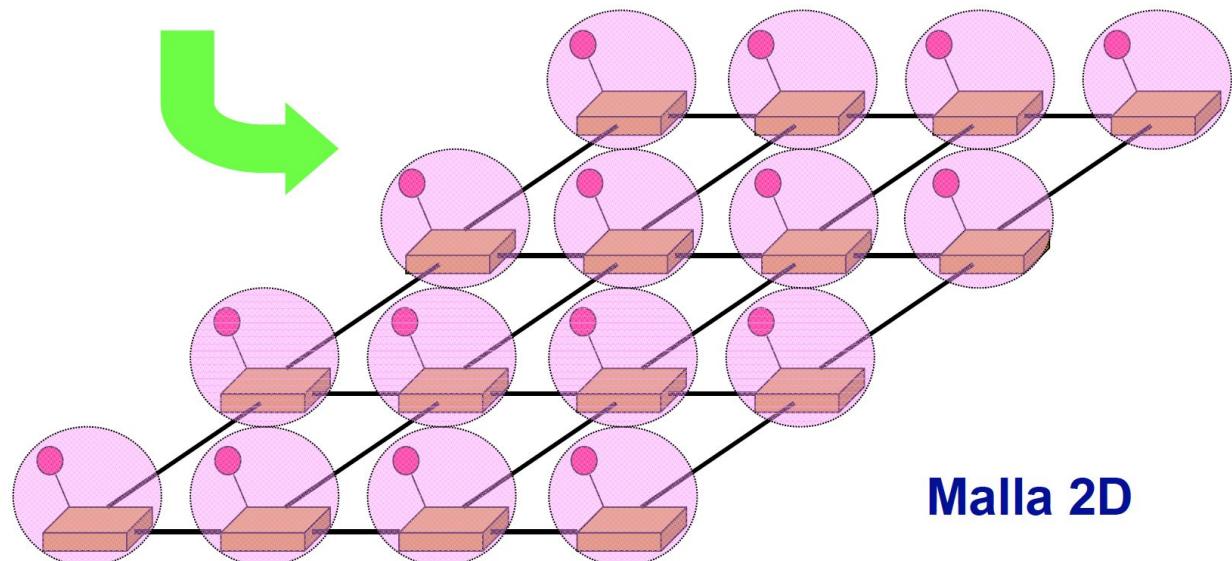
2. Propiedades: Topología (VIII)

- Clasificación de redes según la rigidez de los enlaces:



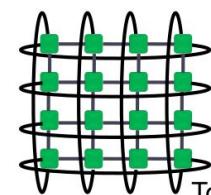
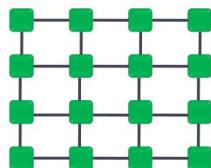
2. Propiedades: Topología (IX)

- Una red directa equivale a una red indirecta donde a cada conmutador se conecta un único nodo:



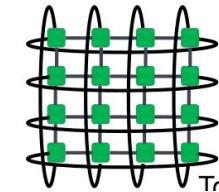
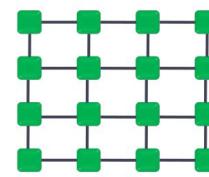
2.2 Topología: Redes estrictamente ortogonales (I)

- Se denomina red estática de dimensión n y $K_{n-1} * ... * K_1 * K_0$ a una red en la que los conmutadores tienen k nodos en la dimensión n-1, k nodos en la dimensión n-2,..., k nodos en la dimensión 1 y k nodos en la dimensión 0.
- Se dice que es estrictamente ortogonal si:
 - Cada conmutador tiene al menos un enlace en cada dimensión.
 - Cada enlace se mueve en una única dimensión.
 - Nodos representables en un espacio n-dimensional ortogonal.



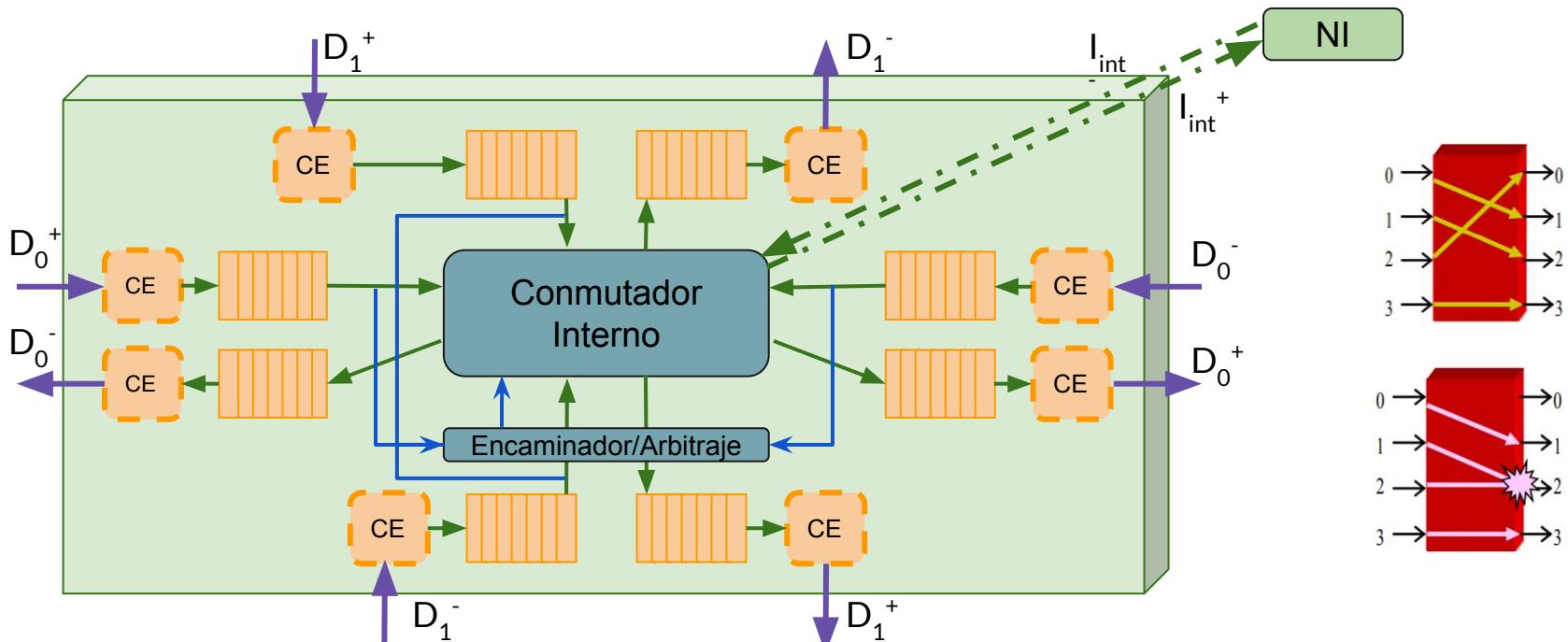
2.2 Topología: Redes estrictamente ortogonales (II)

- Los nodos se numeran como puntos en un espacio vectorial en base k_i , según la dimensión i a la que pertenezca la numeración.
 - La identificación del nodo A será a_{n-1}, \dots, a_1, a_0
- Los enlaces unen nodos cuya distancia es 1. Llevarán un signo que indica el sentido del enlace, positivo, + o negativo, -.

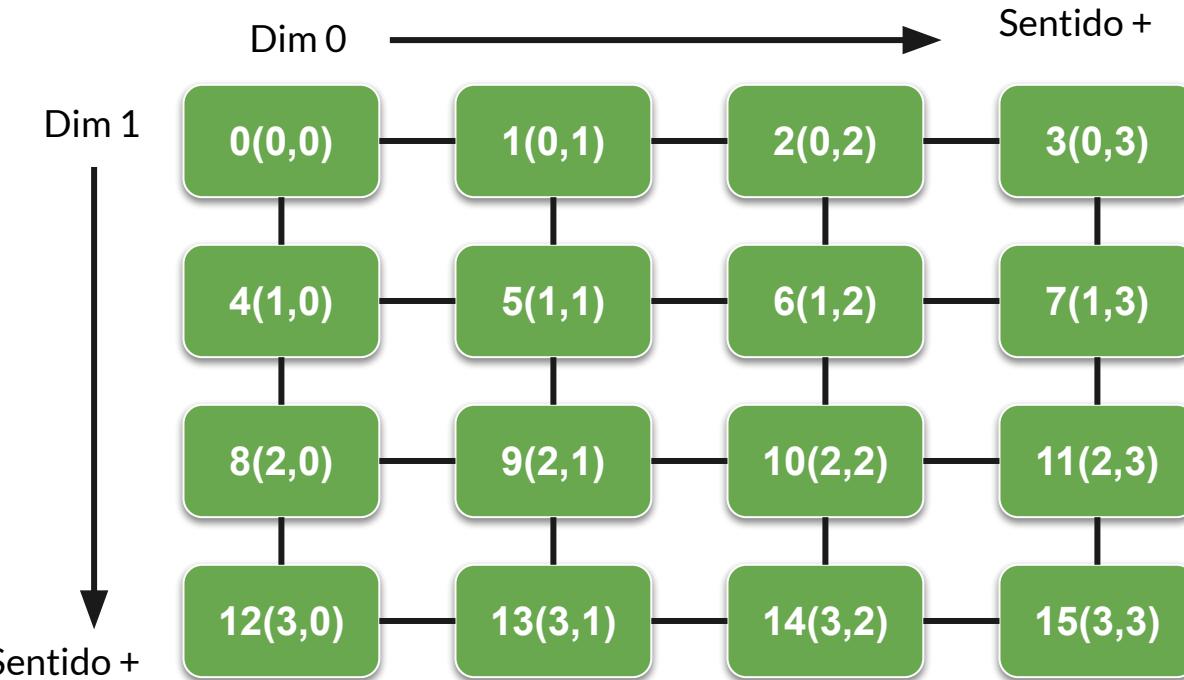


2.2 Topología: Redes estrictamente ortogonales (III)

- Conectores, conmutadores / switches.
- Ej: Conmutador para una red directa estrictamente ortogonal de dimensión 2



2.2 Topología: Redes estrictamente ortogonales (IV)



**Malla de dim. 2 y
base 4 (4x4 nodos)**

$$A = (a_1, a_0)$$

$$a_1, a_0 \in \{0, 1, 2, 3\}$$

Dist. mínima, dm, entre F y D:

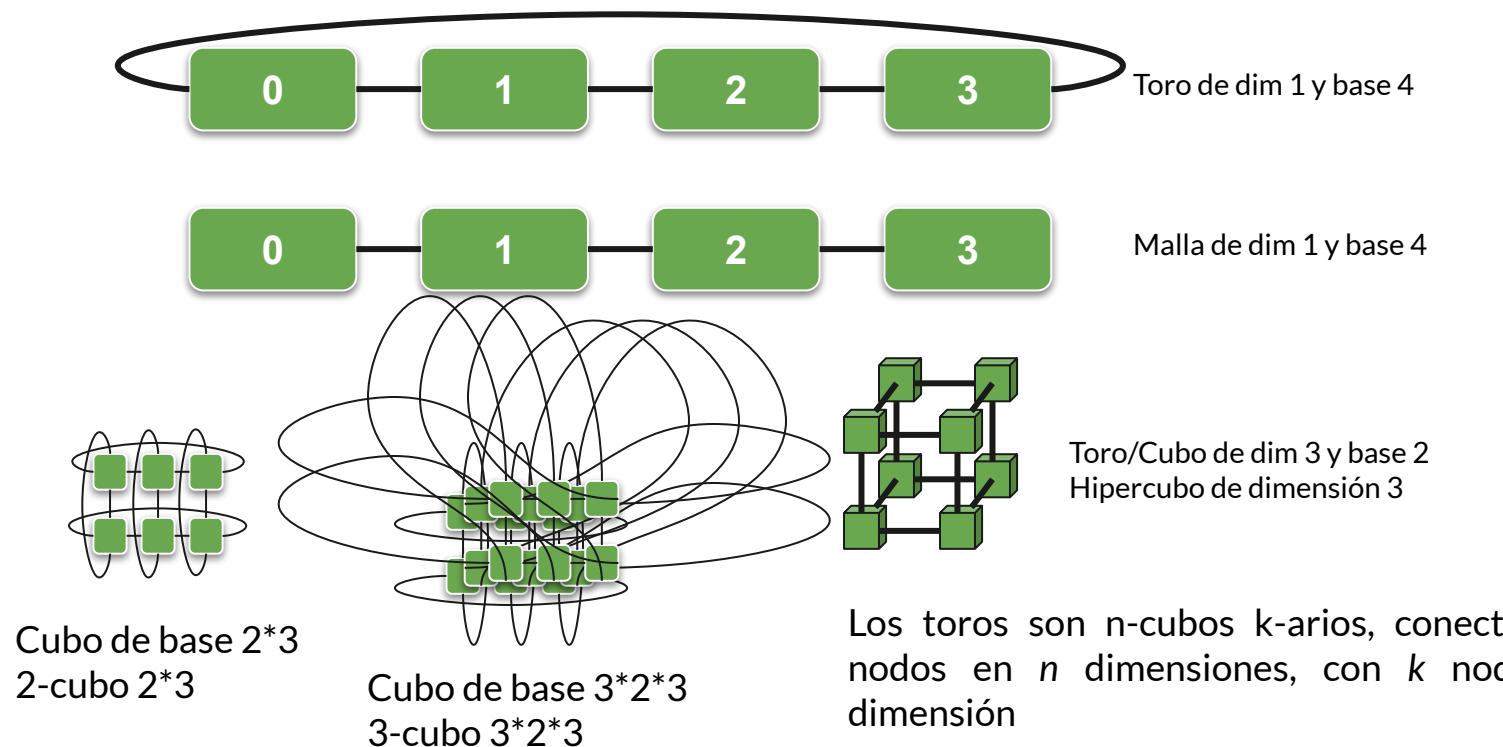
$$dm(F, D) = \sum_{i=0}^{n-1} |d_i - f_i|$$

$$F = 2$$

$$D = 9$$

$$dm(2,9) = |1-2| + |2+0| = 1+2 = 3$$

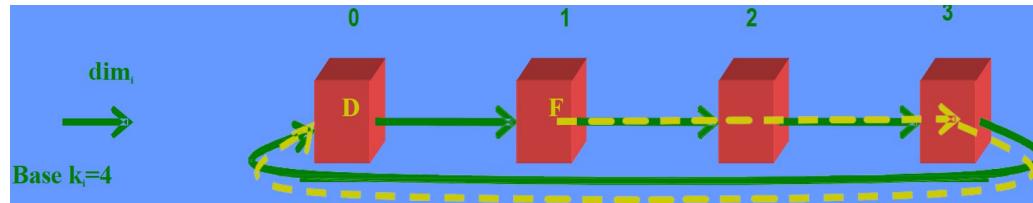
2.2 Topología: Redes estrictamente ortogonales (V)



2.2 Topología: Redes estrictamente ortogonales (VI)

Distancia mínima en un toro unidireccional:

$$dm(F, D) = \sum_{i=0}^{n-1} (d_i - f_i) \bmod k$$



$$dm(1, 0) = (0 - 1) \bmod 4 = 3$$

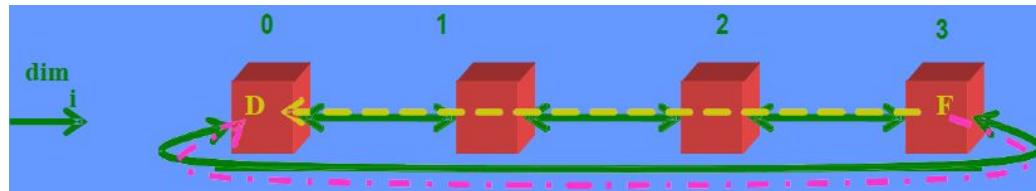
2.2 Topología: Redes estrictamente ortogonales (VII)

Distancia mínima en un toro bidireccional:

$$\text{offset}_i^+ = (d_i - f_i) \% k_i$$

$$\text{offset}_i^- = (f_i - d_i) \% k_i$$

$$\text{Dist}(F, D) = \sum_{i=0}^{n-1} \min(\text{offset}_i^+, \text{offset}_i^-)$$



$$\begin{aligned}\text{offset}^+(F, D) &= (0-3) \% 4 = 1 \\ \text{offset}^-(F, D) &= (3-0) \% 4 = 3\end{aligned}$$

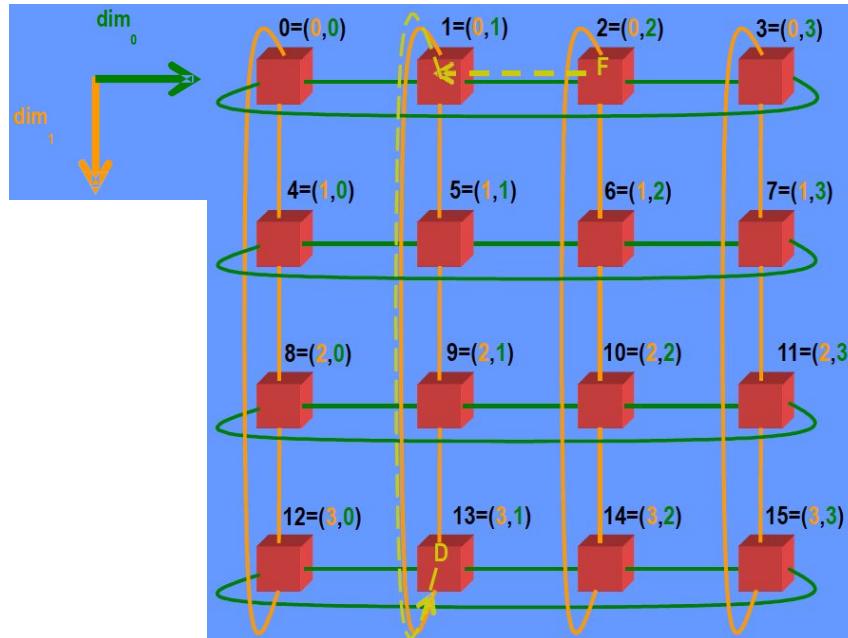
$$\text{Dist}(F, D) = \min(1, 3) = 1$$

2.2 Topología: Redes estrictamente ortogonales (VIII)

Distancia mínima en un toro bidireccional:

2-cubo 4-ario:

Toro de dimensión 2
y base 4
(4x4 nodos)



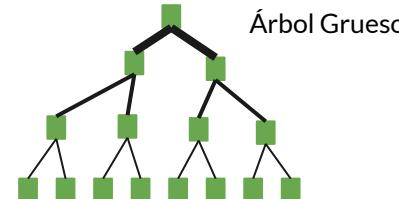
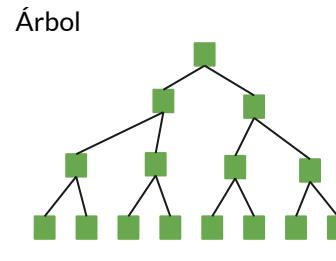
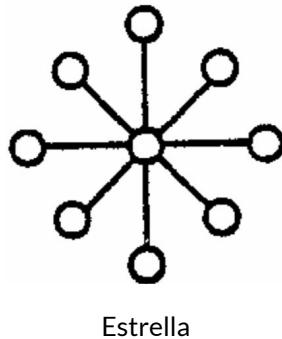
$$A = (a_1, a_0)$$

$$a_1, a_0 \in \{0, 1, 2, 3\}$$

$$dm(2, 13) = \min(3, 1) + \min(3, 1) = 2$$

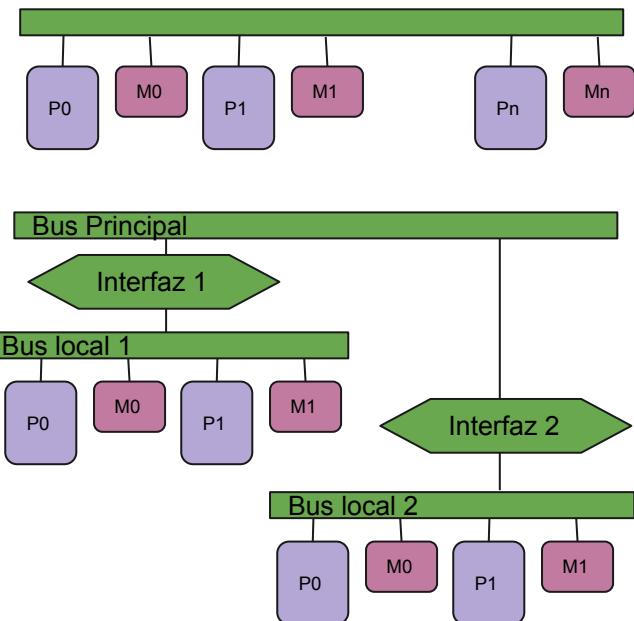
2.2 Topología: Redes no estrictamente ortogonales

- No poseen un único enlace conectando cada dimensión.
- Incluye árboles y estrellas.



2.2 Topología: Redes dinámicas

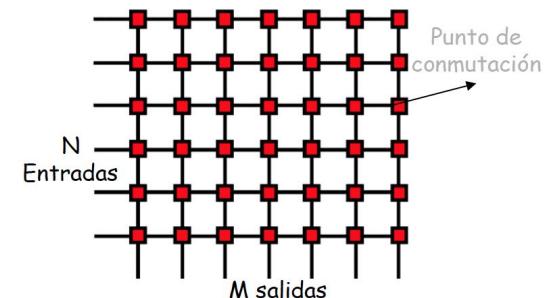
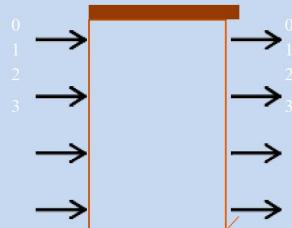
- Incluye los buses, las redes de barras cruzadas y las redes multietapa
- Buses: Red que permite conectar un número variable de componentes que se acogen todos a las mismas normas.
 - Se llama de **medio compartido** porque en el medio sólo puede existir **una comunicación por instante**.
 - En caso de conflicto hay que decidir quién tiene prioridad, según las normas que se controla el árbitro del bus, que será la circuitería que decide quién y en qué orden acceden al bus.
 - Es la red con **menor coste, y peores prestaciones**
 - Se pueden incrementar sus prestaciones, estableciendo **jerarquías de buses**



2.2 Topología: Redes de medio no compartido (I)

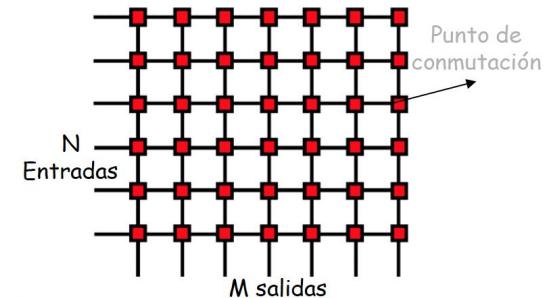
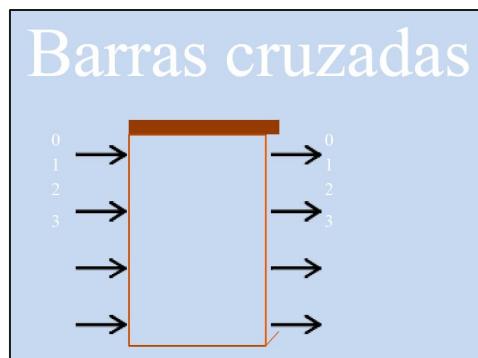
- Redes de barras cruzadas (Crossbar Networks) de n entradas por m salidas ($n*m$)
 - Todos los elementos están conectados mediante un **comutador de líneas cruzadas** que permite **conexión entre todas las entradas y salidas** (totalmente conectada).
 - La conexión en cada cruce de líneas es **configurable**.
 - Se usa para **conectar procesadores y módulos de memoria**: no se puede acceder desde un procesador a dos módulos de memoria simultáneamente.

Barras cruzadas



2.2 Topología: Redes de medio no compartido (II)

- Redes de barras cruzadas (Crossbar Networks) de n entradas por m salidas ($n*m$)
 - También se pueden usar para conectar procesadores entre sí.
 - Son redes **no bloqueantes**: El conjunto de entradas puede conectarse con el conjunto de salidas simultáneamente en cualquier permutación
 - Son fácilmente **escalables**, aunque muy **costosas** y solo son viables en dimensiones pequeñas



2.2 Topología: Redes de medio no compartido (III)

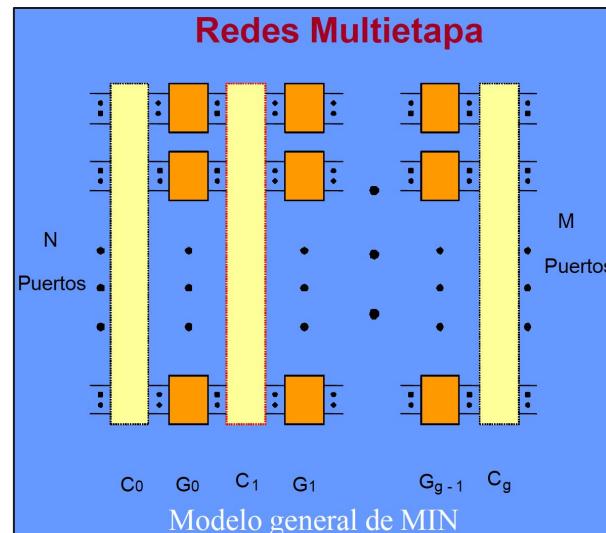
- Redes multietapa (Redes MIN) de n entradas por m salidas ($n*m$):
 - *Multistage Interconnection Network*
 - Están formadas por un conjunto de comutadores y enlaces que se conectan siguiendo una **función de conexión** particular con ciertas propiedades.
 - La función de conexión debe permitir **conexiones desde todas las entradas a todas las salidas simultáneamente** sin repetir ninguna entrada ni salida. Se construyen con permutaciones de los bits que caracterizan cada entrada.

Estados de un comutador 2x2:



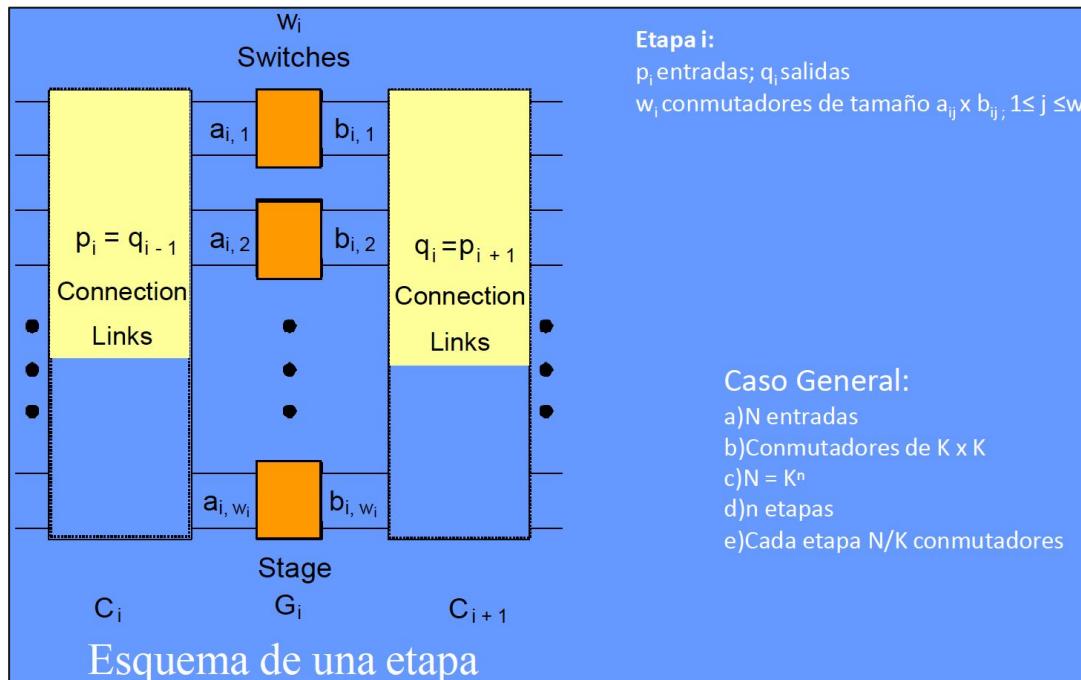
2.2 Topología: Redes de medio no compartido (IV)

- Redes multietapa (Redes MIN) de n entradas por m salidas (n^*m):
 - Las funciones más conocidas es la de baraje perfecto (*perfect shuffle*) y la función de la permutación mariposa (*butterfly*)
 - Los conmutadores de la red permiten una conexión simultánea cruzada, paralela o de difusión.



2.2 Topología: Redes de medio no compartido (V)

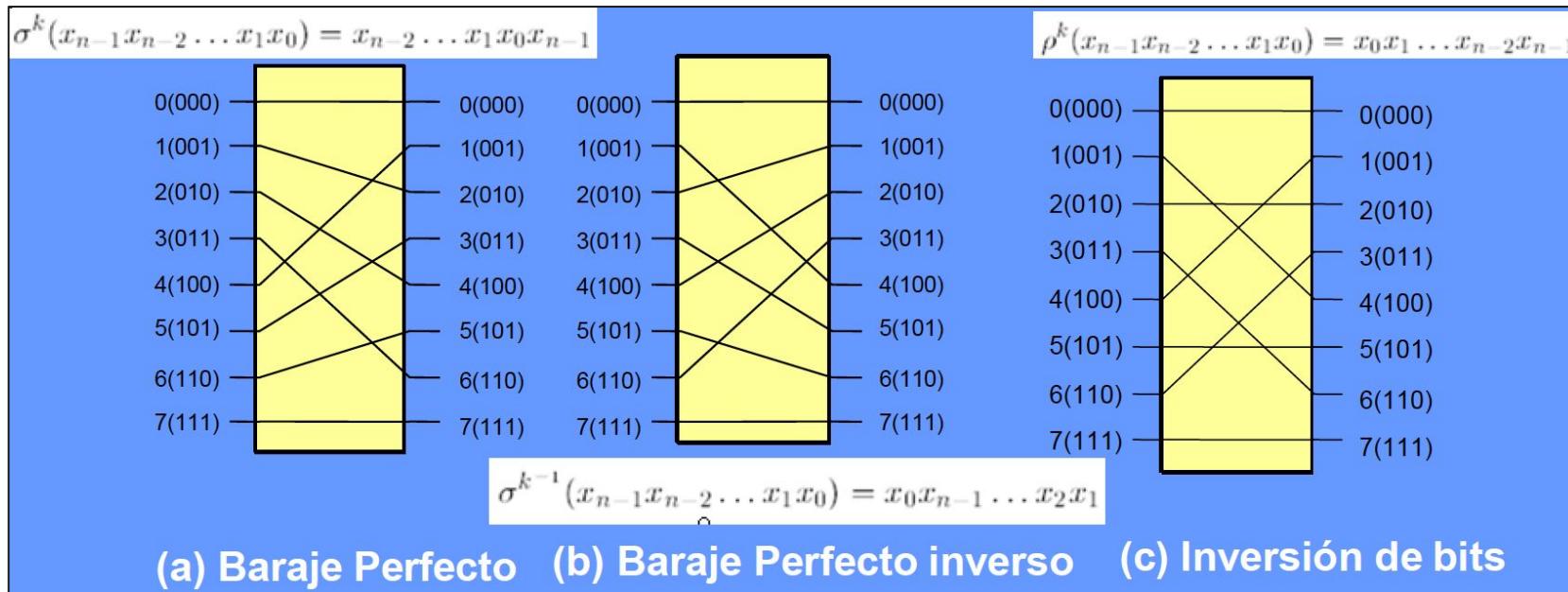
- Redes multietapa (Redes MIN) de n entradas por m salidas (n^*m):



2.2 Topología: Redes de medio no compartido (VI)

- Redes multietapa (Redes MIN) de n entradas por m salidas (n^*m):

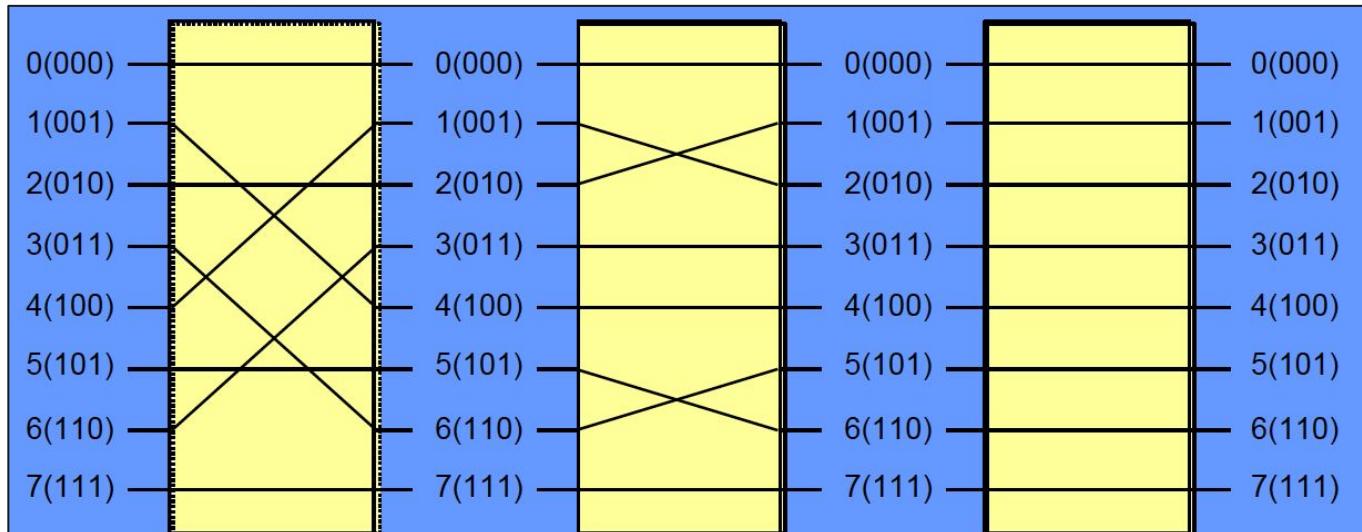
Esquemas de conexión:



2.2 Topología: Redes de medio no compartido (VII)

- Redes multietapa (Redes MIN) de n entradas por m salidas (n^*m):

Esquemas de conexión:



a) Mariposa segunda

b) Mariposa segunda

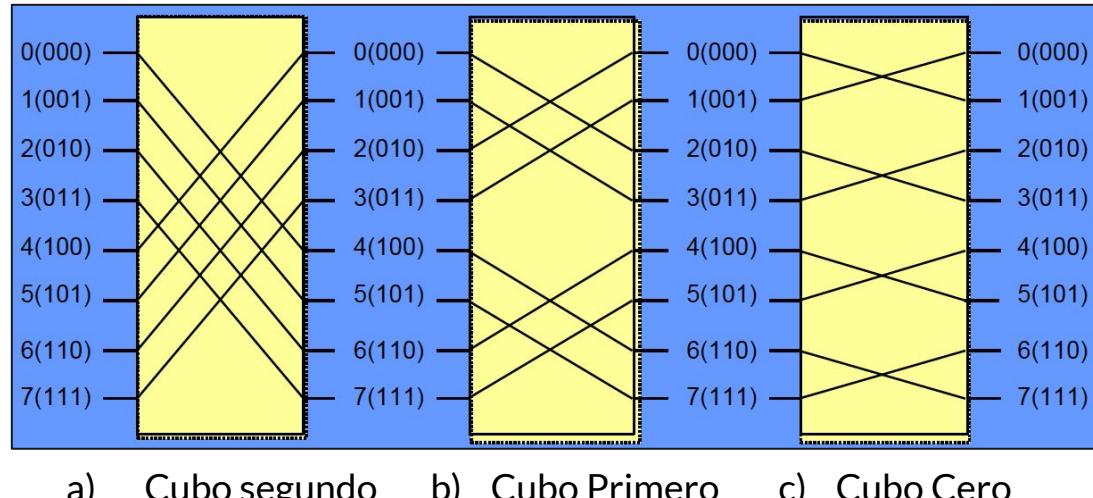
b) Mariposa cero

$$\beta_i^k(x_{n-1} \dots x_{i+1} x_i x_{i-1} \dots x_1 x_0) = x_{n-1} \dots x_{i+1} x_0 x_{i-1} \dots x_1 x_i$$

2.2 Topología: Redes de medio no compartido (IX)

- Redes multietapa (Redes MIN) de n entradas por m salidas (n^*m):

Esquemas de conexión:



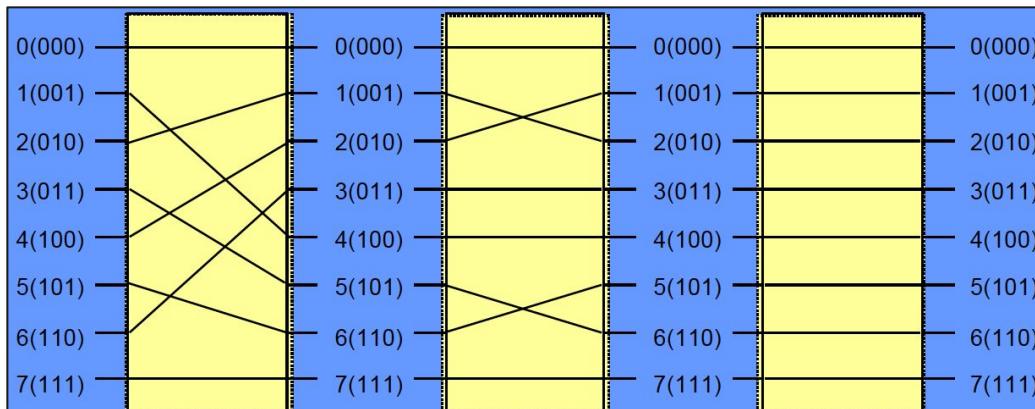
$$E_i(x_{n-1} \dots x_{i+1} x_i x_{i-1} \dots x_0) = x_{n-1} \dots x_{i+1} \bar{x}_i x_{i-1} \dots x_0$$

Conexión cubo

2.2 Topología: Redes de medio no compartido (X)

- Redes multietapa (Redes MIN) de n entradas por m salidas (n^*m):

Esquemas de conexión:



a) Línea base
segunda

b) Línea base
primera

b) Línea base
cero

$$\delta_i^k(x_{n-1} \dots x_{i+1} x_i x_{i-1} \dots x_1 x_0) = x_{n-1} \dots x_{i+1} x_0 x_i x_{i-1} \dots x_1$$

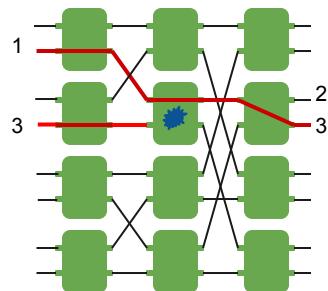
Conexión de línea base

2.2 Topología: Redes de medio no compartido (VI)

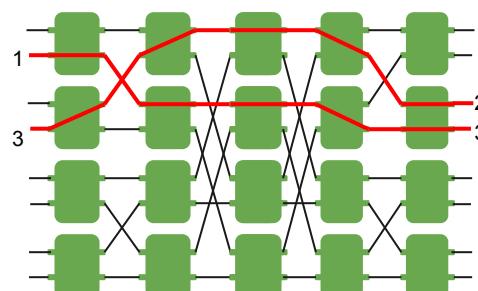
- Según disponibilidad de caminos:
 - Bloqueantes: Hay rutas que impiden la conmutación de otras
 - No bloqueantes: Todas las rutas se pueden simultanear
 - Reconfigurables

- Según el tipo de canales y conexiones:
 - Unidireccionales
 - Bidireccionales

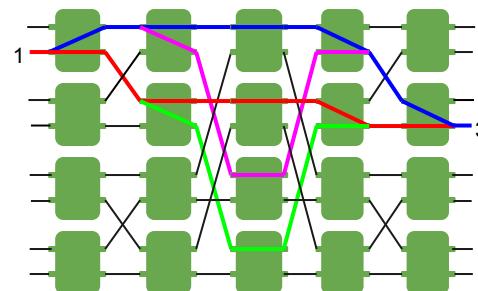
2.2 Topología: Redes de medio no compartido (VII)



Red de Mariposa

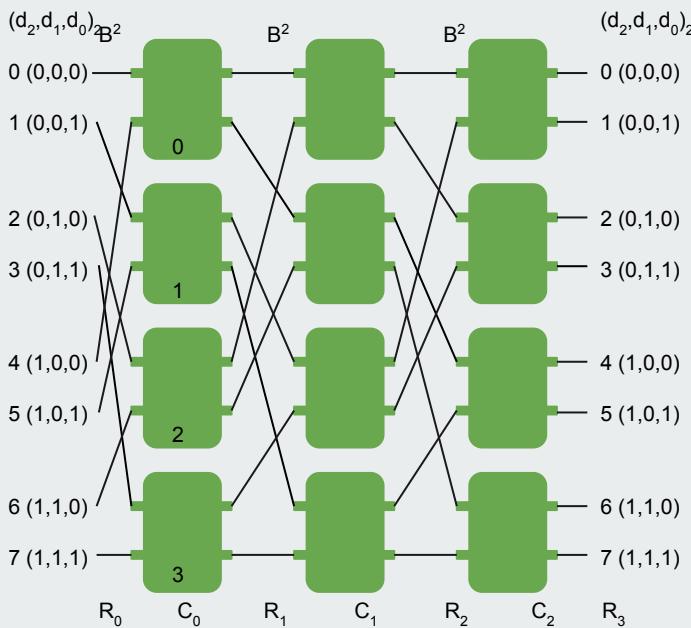


Red de Benes



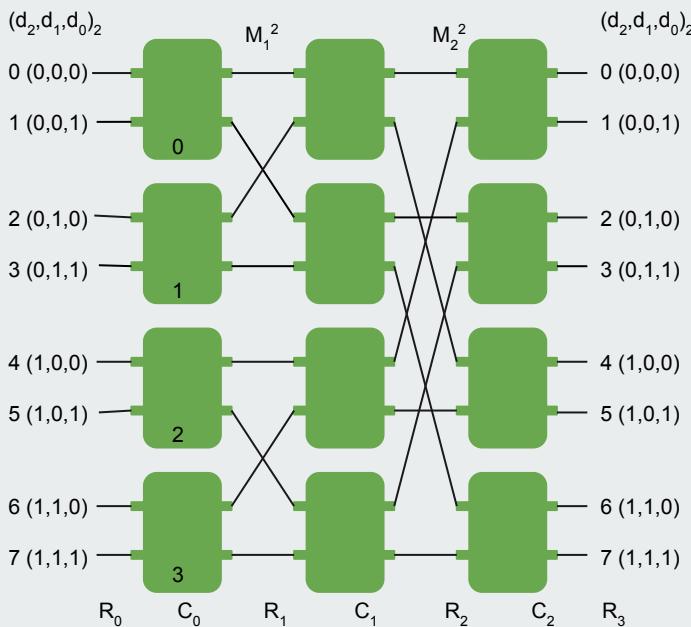
Red de Benes

2.2 Topología: Red omega



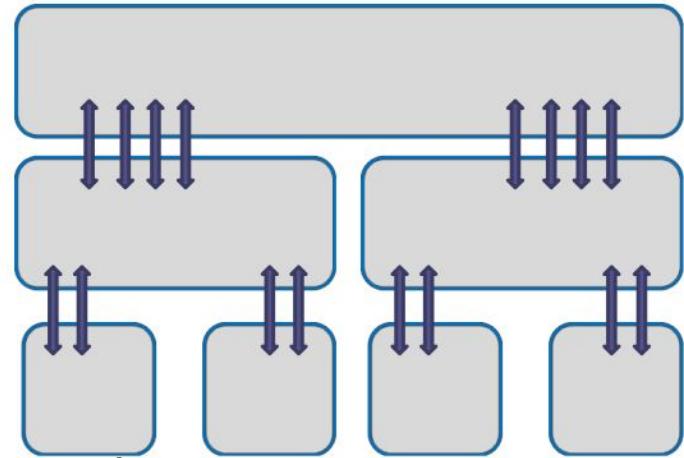
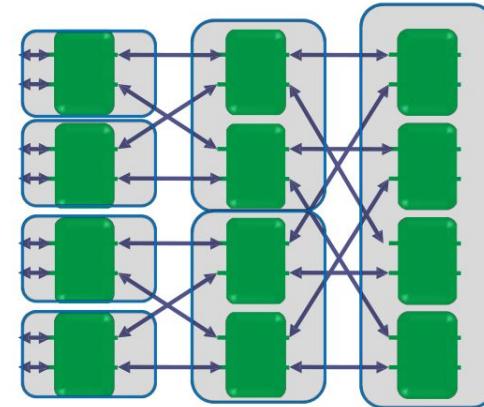
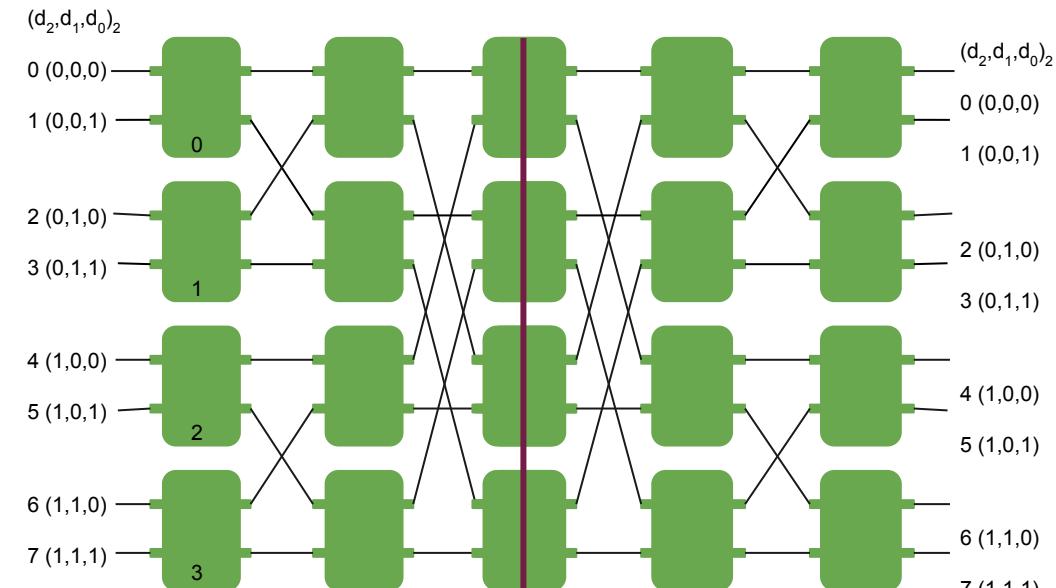
- Red de $K^n * K^n$ ($8 * 8 = 2^3 * 2^3$), $n = 3, k=2$
 - Número de etapas, n , 3. Se nombran con una C y en base n , C_i
 - Conmutadores de $k*k$ ($2*2$)
 - k^{n-1} conmutadores/etapa (2^2)
 - Subred R_i ($i=0, \dots, n-1$); baraje- k perfecto (**baraje-2**)
 - $B^k((f_{n-1}, f_{n-2}, \dots, f_1, f_0)_k) = (f_{n-2}, \dots, f_1, f_0, f_{n-1})_k$
 - $B^k((f_2, f_1, f_0)_2) = (f_1, f_0, f_2)_2$

2.2 Topología: Red mariposa



- Red de $K^n * K^n$ ($8*8=2^3*2^3$), $n = 3, k=2$
 - Número de etapas, n , 3. Se nombran con una C y en base n , C_i
 - Conmutadores de $k*k$ ($2*2$)
 - k^{n-1} conmutadores/etapa (2^2)
 - Subred R_i ($i=0,...,n-1$); Mariposa M_i^k
 - $M_i^k((f_{n-1}, f_{n-2}, ..., f_i, ..., f_1, f_0)_k) = (f_{n-1}, f_{n-2}, ..., f_0, ..., f_1, f_i)_k$
 - $M_2^2((f_2, f_1, f_0)_2) = (f_0, f_1, f_2)_2$
 - La red Mariposa puede ser unidireccional o bidireccional
- Mariposa bidireccional, equivale a una Red de benes reconfigurable y también a una red de árbol grueso

2.2 Topología: Red mariposa



2. Propiedades: Características

1. Diámetro: Máxima longitud (nº de enlaces atravesados) de entre los caminos óptimos (más cortos). **Diámetro grande implica poca habilidad de comunicación** entre nodos. Se buscan diámetros pequeños.

2. Ancho de bisección (b): Mínimo de enlaces a cortar para dividir a la red en dos mitades similares incluyendo el mismo número de commutadores y de nodos. Si cada enlace es capaz de transportar w bits, el ancho de banda de la bisección será bw .

2. Propiedades: Características

3. Latencia: Retraso máximo producido por la comunicación de un mensaje pequeño entre dos nodos cualesquiera de una red. También se le denomina contención y está relacionado con los tiempos de espera producidos durante el transporte de los datos.

4. Productividad: N° total de paquetes de información que una red puede transportar por unidad de tiempo. Hay que tener cuidado con los puntos calientes (hot spot) que son los nodos o enlaces donde se concentra la mayor parte del tráfico de una red.

2. Propiedades: características

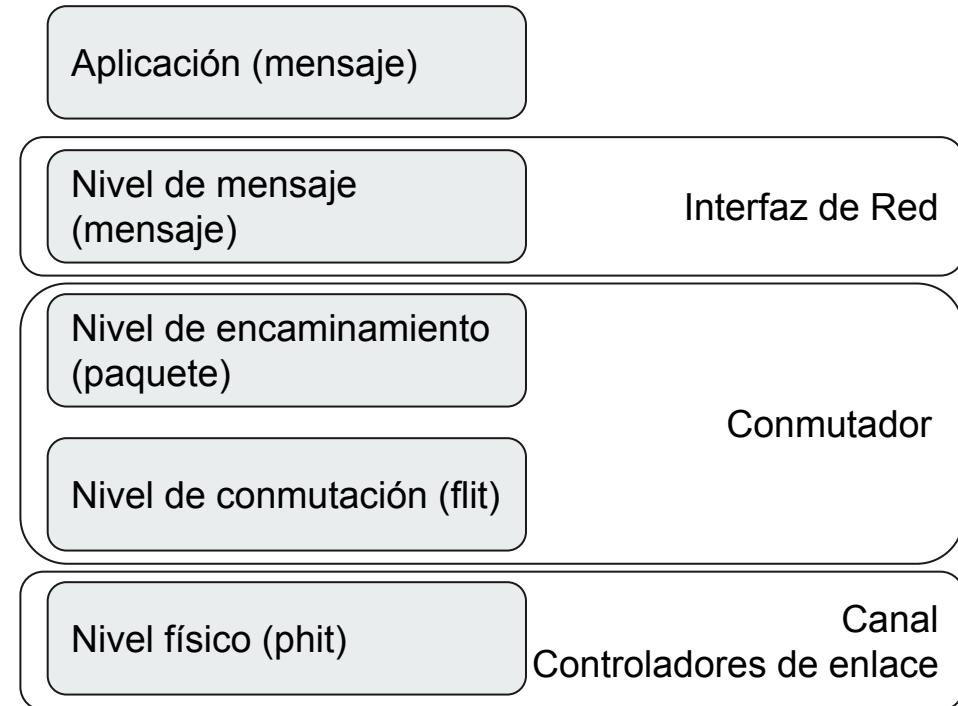
5. Escalabilidad: Facilidad con la que una red puede extenderse sin que sus prestaciones se vean afectadas.
6. Conectividad: Una red es totalmente conectada si existe una conexión directa entre cualquier par de nodos.
7. Grado de los nodos: Número de enlaces que tiene un nodo conectado con otros nodos. Se puede hablar de grado de entrada (GE), enlaces entrantes desde otros nodos y grado de salida (GS). En este caso el grado es GE+GS. Si el grado de todos los nodos es el mismo, la red es regular

2. Propiedades: características

8. Niveles de servicio: Nivel al que se trata la red:

- Físico
- De conmutación
- De encaminamiento
- De mensaje
- De aplicación

Según el nivel de servicio, la unidad de comunicación cambia.



2. Propiedades: características

9. Calidad del servicio (QoS):

- Definición 1: Efecto global de las **prestaciones** de un servicio que determinan el grado de **satisfacción** de quien la utiliza.
- Definición 2: Conjunto de **requisitos del servicio** que debe cumplir una red en el transporte de flujo.

La calidad del servicio se mejora realizando una buena **gestión de la congestión**, teniendo un bajo nivel de retardo, un alto rendimiento y un coste del servicio justo.

2. Propiedades: características

9. Calidad del servicio (QoS): Algunas de las medidas que se usan para determinar la calidad del servicio son:

- Disponibilidad: Tiempo mínimo para asegurar que una red estará en funcionamiento (99,99%)
- Ancho de banda: El mínimo ancho de banda que el operador garantiza (2Mbps)
- Pérdida de paquetes: El número máximo de paquetes perdidos, siempre que el volumen de comunicaciones no exceda de cierto valor (1,1%)
- Round Trip Delay: Retardo de ida y vuelta medio en los paquetes (80ms)
- Jitter: Fluctuación producida en el retardo de ida y vuelta (20ms) que retarda normalmente la formación de mensajes, por el retardo de los diferentes paquetes en los que se divide.

2. Propiedades: características

9. Calidad del servicio (QoS): Qué influye en la calidad del servicio
 - Algoritmos de encaminamiento
 - Controladores de enlace
 - Políticas de prioridades
 - Políticas de desbloqueos
 - Políticas de desestimación de paquetes
 - Cantidad y número de almacenamientos intermedios para unidades de transmisión
 - ...

2. Propiedades: características

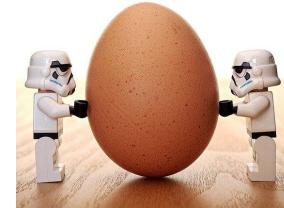
10. Alta disponibilidad: Es una medida del **porcentaje del tiempo** que una red está **operativa** por mes o por año, o por día...
11. Tolerancia a Fallos: Es la **capacidad** que tiene una red de seguir prestando servicio a pesar de que alguna de sus partes no esté operativa. **Enrutamientos adaptativos**. Una red tolerante a fallos, tiene que ofrecer caminos alternativos entre cada par origen-destino.
12. Fiabilidad: Es el **grado de seguridad** que podemos tener de que una transmisión llegará al destino bien, es decir, no se perderá, no llegarán los datos corruptos ...

2. Propiedades: características

13. Remote Direct Memory Access (RDMA): Establece qué tipo de acceso a memoria no local se le permite a cada uno de los nodos que están conectados a la red, ya sea a través de un procesador o del mismo sistema de gestión de la memoria de cada nodo.

- Este acceso puede ser de lectura o de lectura/escritura, permitiendo en muchos casos la realización de operaciones atómicas.
- Si la red no permite este acceso, la alternativa es el uso de mensajes.
- Esta característica permite, por ejemplo, el acceso desde una GPU a la memoria de una CPU.

Trabajo para la próxima semana



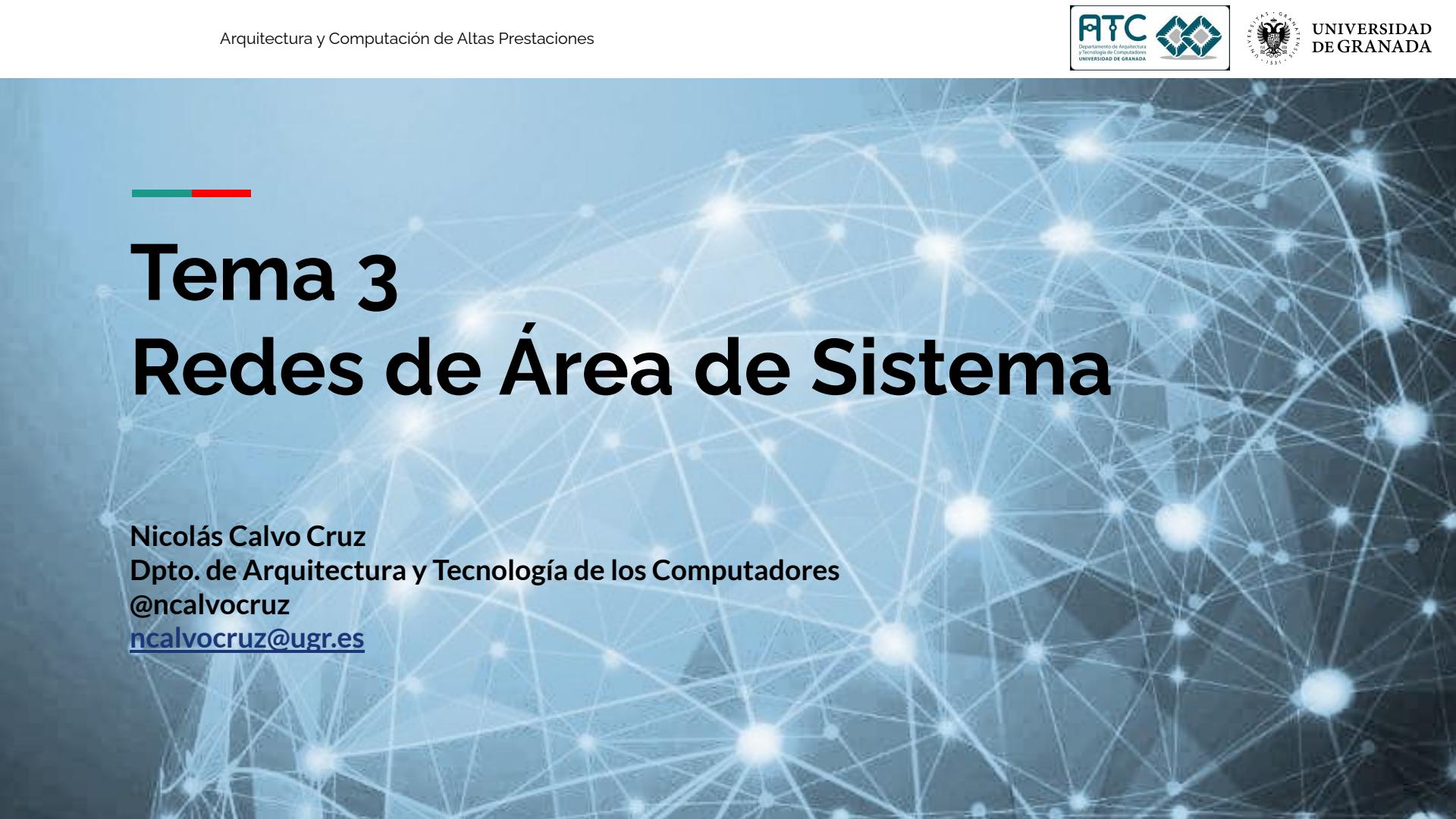
Buscar información de la siguiente red HPC de un equipo Top500:

- Grupo 1: Tofu Interconnect D
- Grupo 2: Infiniband EDR
- Grupo 3: Sunway
- Grupo 4: TH Express-2
- Grupo 5: Cray/HPE
- Grupo 6: Mellanox HDR Infiniband
- Grupo 7: Dual-rail Mellanox EDR Infiniband

=> Un miembro del grupo presentará su trabajo la próxima semana. Habrá que entregar tanto la presentación como una transcripción de lo que se dice (incluyendo las fuentes).



Gracias.

A large, semi-transparent network graph composed of numerous small white dots (nodes) connected by thin white lines (edges), forming a complex web against a dark blue background.

Tema 3

Redes de Área de Sistema

Nicolás Calvo Cruz

Dpto. de Arquitectura y Tecnología de los Computadores

@nkalvocruz

nkalvocruz@ugr.es

Motivación

- Redes de **comunicación en computadores paralelos**.
- Hoy en día se están **sustituyendo los buses por redes** con conexiones **punto a punto** a todos los niveles:
 - Interno al chip
 - A nivel de tarjeta y placa
 - A nivel de chasis o caja
 - LAN y Router IP
- Conocer los algoritmos de **encaminamiento** y la **infraestructura** permite mejorar las **prestaciones**.

Objetivos

- Distinguir entre redes de altas prestaciones y redes estándar.
- Conocer la estructura general de un conmutador.
- Estudiar las topologías y nomenclaturas de las redes de altas prestaciones.
- Estudiar los algoritmos de encaminamiento.



Índice

1. Clasificación Sistemas de Comunicación
2. Propiedades
- 3. Diseñar una Red**
4. Prestaciones
5. Enrutamiento
6. Técnicas de conmutación
7. Ejemplo

A close-up photograph of several clear plastic network cables (RJ45) against a blue background. The background is filled with numerous small, glowing white dots of varying sizes, creating a digital or data flow effect.

3. Diseñar una red

Qué tener en cuenta para diseñar una red SAN

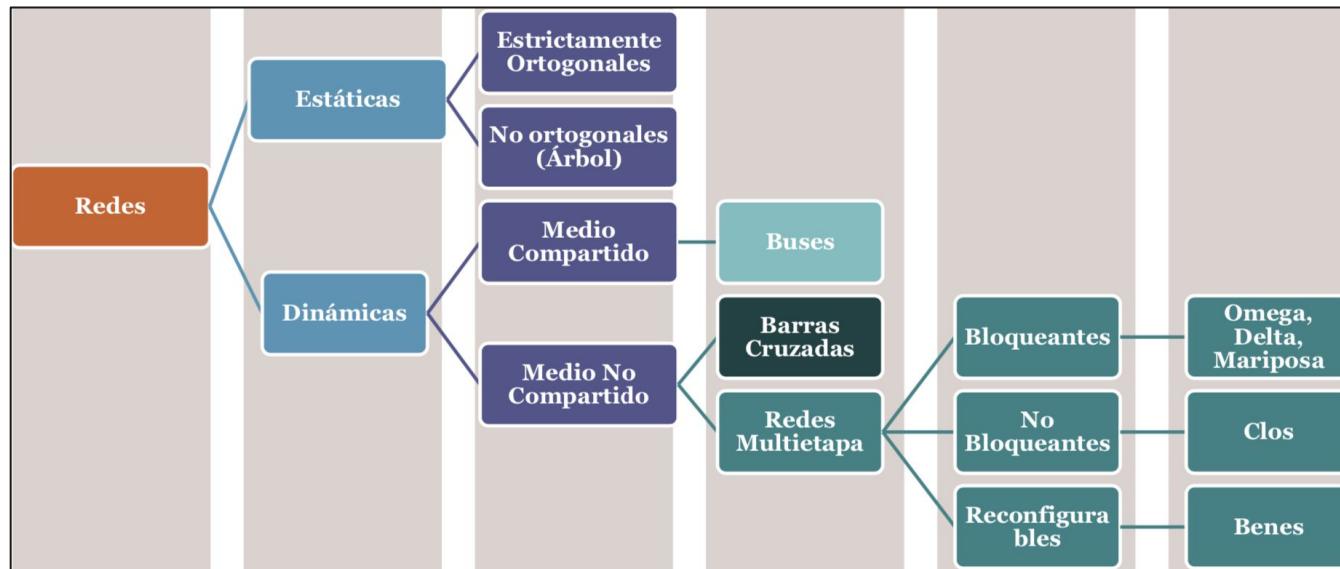
3. Diseñar una red

Hay **cuatro puntos clave** al diseñar una red que marcan su funcionamiento y prestaciones:

1. **Topología**
2. Estrategia de **comutación**
3. Mecanismos de control de **flujo**
4. Algoritmo de **encaminamiento**

3. Diseñar una red: Topología

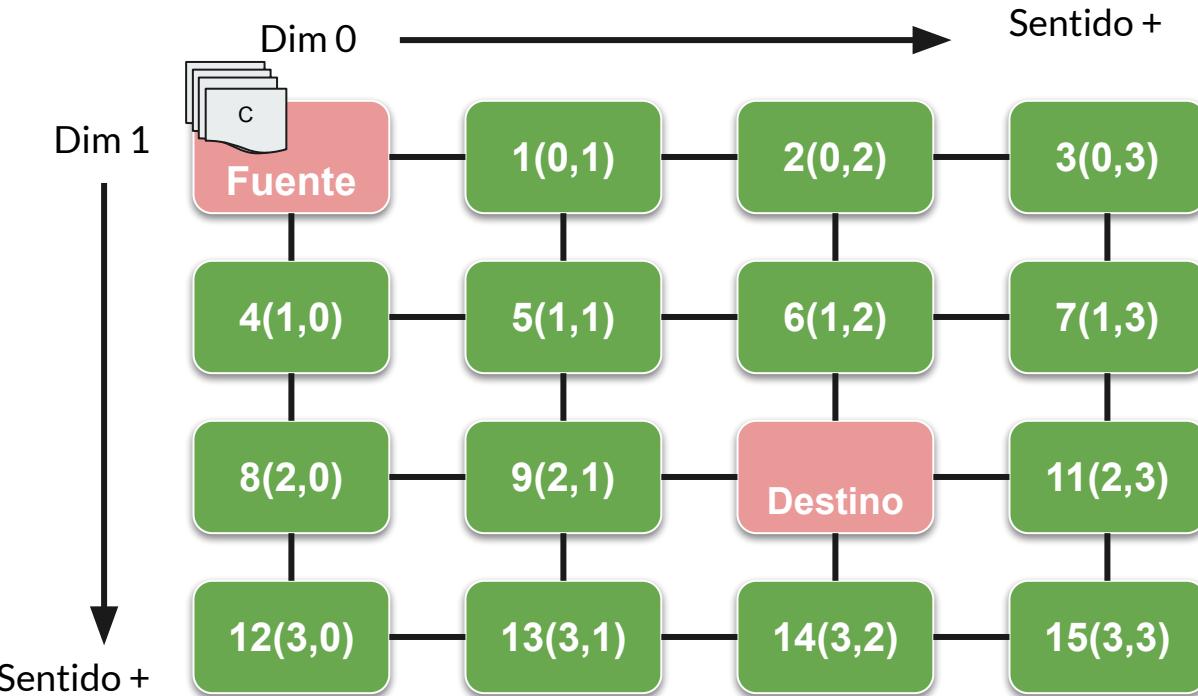
- La topología determinará muchas de las prestaciones de la red.



3. Diseñar una red: Estrategia de conmutación

- Determina cómo viaja la información de un nodo fuente a otro destino:
 - Estableciendo una conexión dedicada (**Conmutación de circuitos**)
 - Agrupada saltando de conmutador en conmutador (**Almacenamiento y Reenvío**)
 - Estirándose a lo largo del camino (**Vermiforme**)
 - Variantes/combinaciones de las anteriores (**Virtual Cut-Through**)

3. Diseñar una red: Estrategia de conmutación



3. Diseñar una red: Control de flujo

- Determina para una unidad de información (de cualquier nivel):
 - Cuándo se mueve entre los diferentes almacenamientos del sistema de comunicación.
 - Cómo y cuándo se asignan los recursos de esos almacenamientos para continuar el transporte de los datos en la red.
- Según el nivel en el que estemos, las unidades de información que deben avanzar varían.

3. Diseñar una red: Control de flujo

- Las unidades de información para las que se establece un control de flujo son:
 - **Phit (Physical Transport Unit):**
 - Unidad más pequeña (nivel más bajo) en la red cuya transferencia sólo se acepta si hay espacio en el destino.
 - Es la cantidad de información transferible en un solo ciclo entre dos elementos de la red.
 - Se transportan en orden entre nodos.
 - La velocidad de transferencia en este nivel se mide en phits/seg.

3. Diseñar una red: Control de flujo

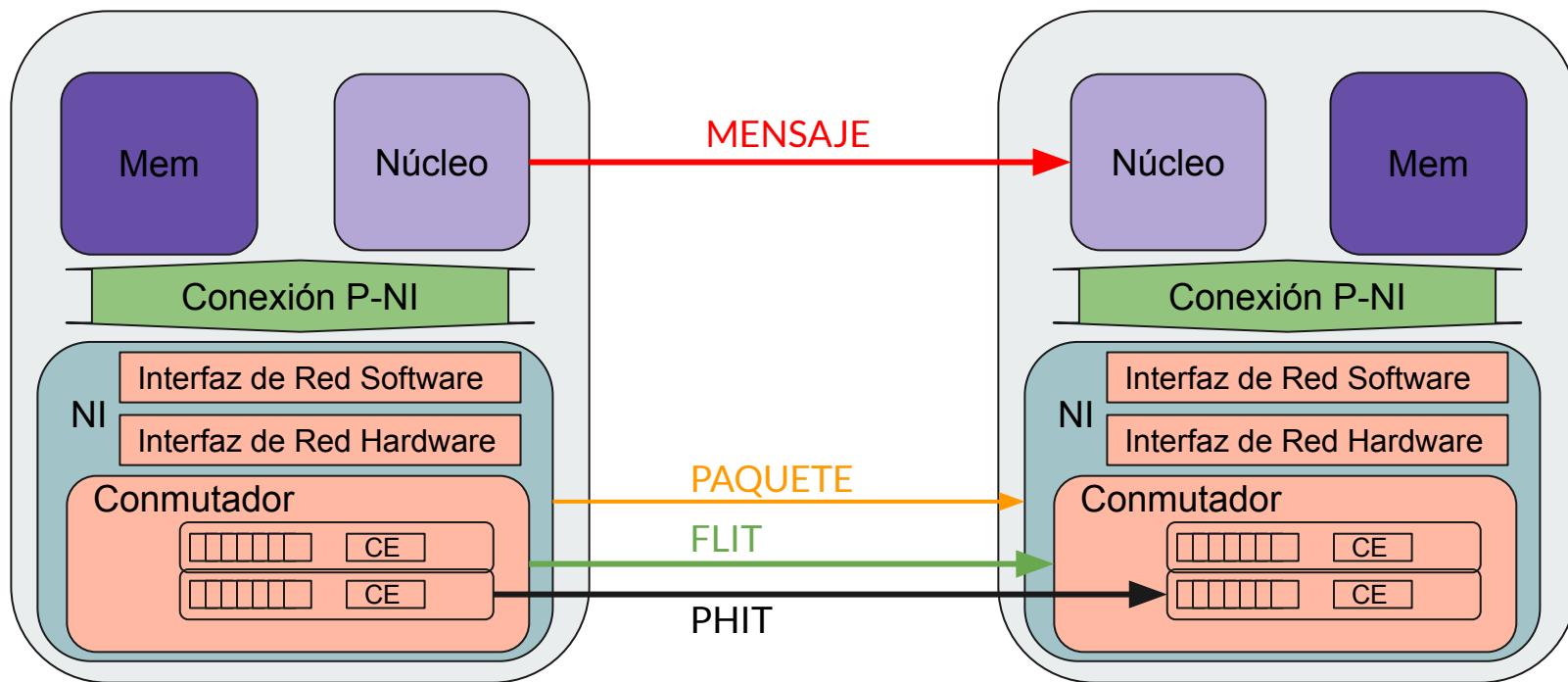
- Las unidades de información para las que se establece un control de flujo son:
 - **Flit (Flow Control Unit):**
 - Mínima cantidad de información que se transfiere entre elementos del sistema de comunicación a nivel de control de flujo.
 - Incluye uno o varios phits.
 - Un flit no se enruta: todos los flits de un paquete siguen la misma ruta (que definirá el paquete, que contiene la dirección de destino).
 - Los flits de un paquete han de transportarse en orden.

3. Diseñar una red: Control de flujo

- Las unidades de información para las que se establece un control de flujo son:
 - Paquete:
 - Unidad de información **siguiente al Flit**, formada mínimo por **1 de estos**.
 - Tiene por una **cabecera**, con la información de destino, una zona de **datos**, con el contenido del paquete y una zona **final** de comprobación.
 - Son las **unidades entre interfaces de red**, y **no se envían en orden**.
 - Mensaje: Conjunto de paquetes. Unidad de control de flujo a nivel de aplicación

Cabecera	Datos	Fin
----------	-------	-----

3. Diseñar una red: Control de flujo



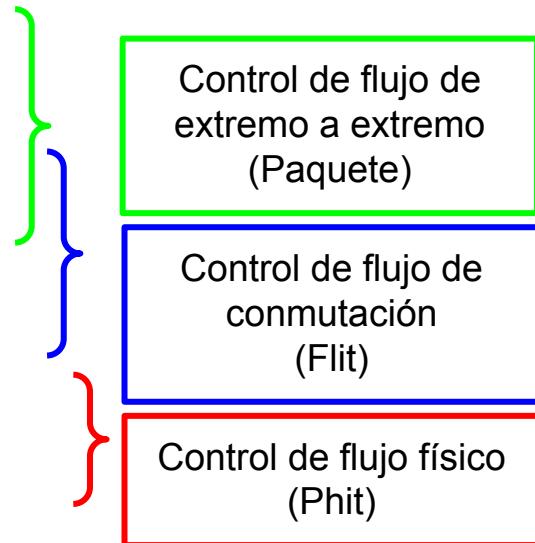
3. Diseñar una red: Control de flujo

Unidad	Componentes	Control de flujo
Mensaje	APP - APP	Control de flujo del programa
Paquete	NI - NI	Control de flujo de extremo a extremo
Flit	C - C	Control de flujo de conmutación
Phit	CE - CE	Control de flujo físico

3. Diseñar una red: Control de flujo

¿Qué tareas tiene exactamente el control de flujo?

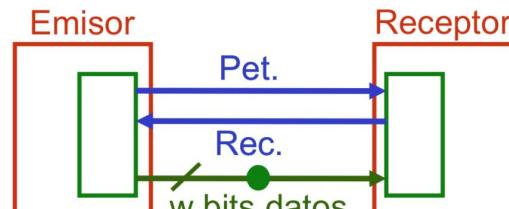
1. **Garantizar las prestaciones mínimas de la red**
2. **Asegurar recepción sin errores**
3. **Garantizar el almacenamiento en destino de las unidades**
4. **Arbitrar entre unidades que quieren acceder a la vez al mismo recurso**
5. **Asegurar que la unidad de información llega al destino sin solaparse con otras**



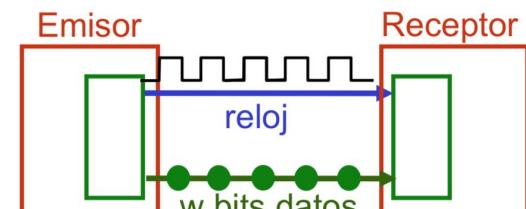
3. Diseñar una red: Control de flujo

Control de flujo físico

- Garantiza que los phits se transfieran por un enlace llegando al destino **sin solapamiento**. Hay 2 alternativas:
 - Síncrono
 - Asíncrono



Asíncrona
Líneas cortas

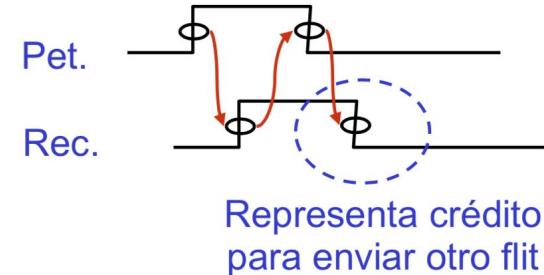


Síncrona
Líneas largas

3. Diseñar una red: Control de flujo

Control de flujo de conmutación

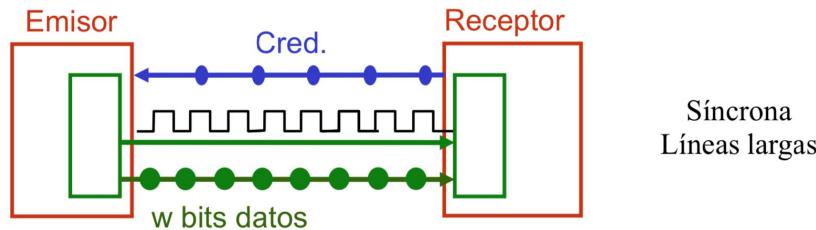
- Transfiere flits dentro de conmutadores: arbitra entre entradas que se quieren conectar simultáneamente a la misma salida.
- Transfiere flits entre conmutadores garantizando almacenamiento en el destino para todo el flit:
 - Con **flit = phit**, probable en líneas cortas: (a) con transferencia asíncrona se puede aprovechar la señal de reconocimiento para indicar disponibilidad de almacenamiento.
 - (b) con transferencia síncrona se necesita una señal de control (rec) desde el receptor al emisor que informe si hay o hubo espacio para un flit



3. Diseñar una red: Control de flujo

Control de flujo de conmutación

- Con **flit \neq phit**, probable en líneas largas: como la transferencia es síncrona se necesita **información de control desde el receptor al emisor que informe de que hay espacio para almacenar un flit (crédito)**.

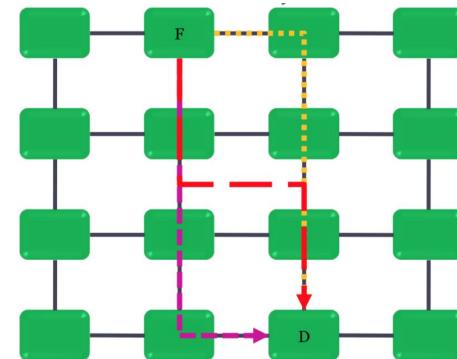


- El emisor puede tener un **contador de créditos asociado al canal**. Este se decrementa en cada envío y se incrementa con cada crédito recibido.
- **Si se multiplexan las señales de control con datos** (enlaces estrechos), **cada señal de reconocimiento puede representar un crédito para n flits**.
- Se pueden utilizar dos tipos de señales, una para parar el envío (STOP) y otra para activarla (GO)

3. Diseñar una red: Algoritmos de encaminamiento

- Debe realizar 2 funciones básicas:
 - **Función de Encaminar:** Determinar la ruta (o rutas) para hacer una operación de comunicación entre:
 - Una fuente y un destino (1-1)
 - Varias fuentes y un destino (M-1)
 - Varios destinos y una fuente (1-M).

Generará un conjunto de caminos candidatos.



- **Función de Selección:** Escoger entre los caminos encontrados cuál seguir para cada paquete.

3. Diseñar una red: Algoritmos de encaminamiento

- El algoritmo de encaminamiento (o enrutamiento) es clave porque:
 - Debe **balancear el tráfico**, incluso ante problemas. A mejor balanceo, mejores prestaciones.
 - Debe **evitar interbloqueos** entre comunicaciones siempre que se pueda.
 - El **camino debe ser lo más corto posible**, manteniendo prestaciones.
 - Debe ser adaptativo para **sobreponerse a fallos físicos de la red**.

3. Diseñar una red: Algoritmos de encaminamiento

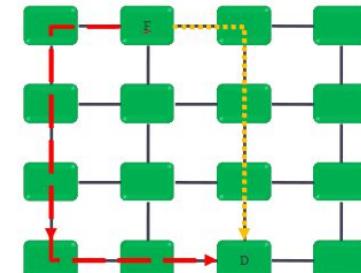
Clasificación:

- Según la selección de rutas:
 - **Deterministas:** Entre el conjunto de rutas posibles, se elige siempre la misma.
 - **No deterministas:** La selección del camino varía. Se pueden hacer 3 cosas:
 - No tomar información de la red (**Selección Aleatoria**)
 - Tomar información de parte de la red para decidir (**Parcialmente Adaptativo**)
 - Tomar información de toda la red para decidir (**Totalmente Adaptativo**)

3. Diseñar una red: Algoritmos de encaminamiento

Clasificación:

- Según la longitud del camino:
 - **Mínimo:** Entre las rutas posibles, se eligen entre las de menor distancia al destino (Algoritmo provechoso)
 - **No mínimo:** Se puede elegir una ruta no mínima (Algoritmo mal-enrutado)



3. Diseñar una red: Algoritmos de encaminamiento

Clasificación:

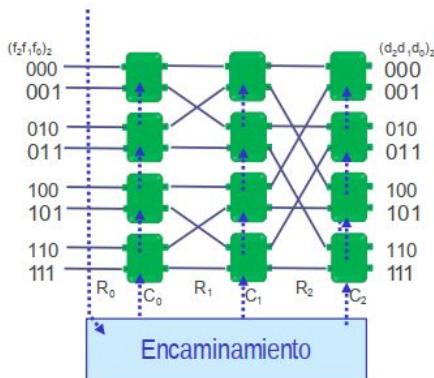
- Según la posibilidad de volver hacia atrás:
 - **Progresivo:** Un paquete no se enrutará por el mismo camino que ha llegado, siempre disminuyendo la distancia.
 - **Con retroceso (*backtraking*):** El camino por el que ha llegado también se tiene en cuenta en la función de selección.

3. Diseñar una red: Algoritmos de encaminamiento

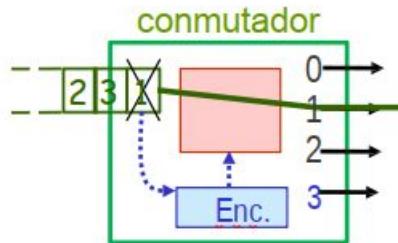
- Hay otros criterios de clasificación, pero no incluyen directamente la función de selección. Se pueden catalogar según:
 - Su funcionalidad (Si es para 1-1, 1-M o M-1)
 - Dónde se hace la función de selección (centralizado, distribuido, multifase o en fuente)
 - Su tipo de implementación (con tabla o sin tabla)
 - Los canales seleccionados, estableciendo siempre un conjunto de canales candidatos o sólo uno posible.

3. Diseñar una red: Algoritmos de encaminamiento

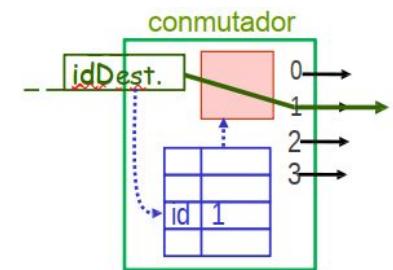
- Hay otros criterios de clasificación, pero no incluyen directamente la función de selección...



Enc. Centralizado: circuitería común genera las señales de control.



Enc. Fuente: los canales a utilizar se fijan en el fuente.



Enc. Distribuido-Tablas: cada comutador obtiene el canal de salida a utilizar. Ej.: consultando una tabla de encaminamiento.

Índice

1. Clasificación Sistemas de Comunicación
2. Propiedades
3. Diseñar una Red
- 4. Prestaciones**
5. Enrutamiento
6. Técnicas de conmutación
7. Ejemplo



—

4. Prestaciones en una red

Medidas de rendimiento en una red SAN

4. Prestaciones

- Las **prestaciones** de una red se miden **según distintas variables** cuyo significado depende también de cómo y a qué nivel se midan:
 - **Coste:** Cantidad de **dinero** que costará la red.
 - **Latencia:** Indica cuánto **tiempo** tardará un **mensaje pequeño** ($L=T(0)$).
 - **Productividad:** Bits por unidad de tiempo transportables desde una fuente a un destino.
 - **Tolerancia a fallos**
- Hay **otras**, como la escalabilidad, calidad del servicio... **que vimos en las propiedades de las redes**.

4. Prestaciones: Tipos de prestaciones

- Las prestaciones se pueden clasificar en 2 grupos:
 - **Prestaciones extremo-a-extremo:** Se miden las prestaciones del envío de mensajes entre dos nodos de la red cualesquiera, X e Y.
 - **Globales:** Se miden a nivel de toda la red con múltiples transferencias simultáneas. La medición se hace inyectando tráfico en la red siguiendo alguna función de distribución que permita representar una cierta situación real.

4. Prestaciones: Prestaciones Extremo-a-Extremo

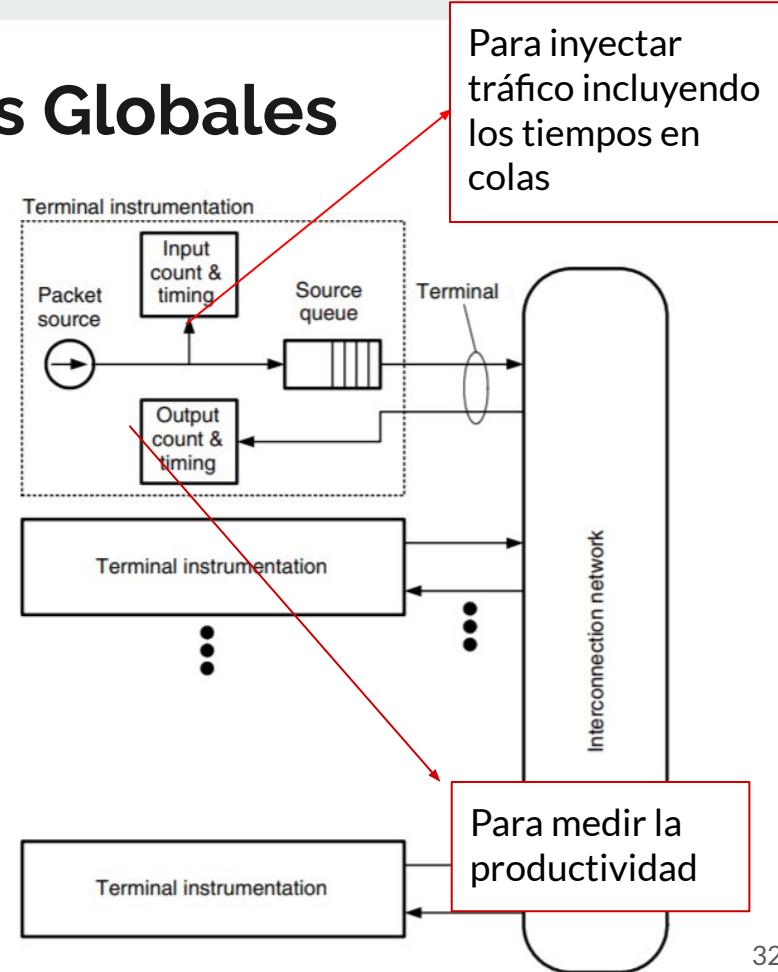
- Tienen en cuenta software y hardware:
 - Ancho de banda teórico o capacidad del canal (Mb/s)
 - Latencia total o tiempo de comunicación, T.
 - Latencia, $L=T(0)$: tiempo que supone la transferencia de un mensaje de tamaño pequeño (0, 1 byte, 8 bytes).
 - Productividad o ancho de banda efectivo, P. (MB/s). $P(m)=m/T(m)$
 - Ancho de banda asintótico, B: productividad máxima.
(Ejemplo Micro-benchmark: test ping-pong)
- Aproximación del tiempo de comunicación a partir de la latencia y el ancho de banda asintótico: $T(m) = L + m / B$

4. Prestaciones: Prestaciones Globales

El sistema de medida debe poder generar tráfico según una distribución deseada:

- Uniforme: Mensajes a cualquier salida de la red con la misma probabilidad.
- Normal: A cualquier salida pero más probablemente a aquellas más cercanas.
- No uniforme: La probabilidad de enviar a un destino depende de la frecuencia con la que se escoge.

Distintos tamaños y regularidad concreta.



4. Prestaciones: Prestaciones Globales

Se toman en distintas situaciones:

- **Sin tráfico previo** para ver cómo evoluciona la red al ir generando el tráfico de forma automática.
- **En equilibrio (steady-state)**. Una red está en equilibrio cuando todos sus almacenamientos intermedios (colas) no aumentan ni disminuyen rápido. Estas medidas se hacen en 3 fases: **warm-up, measurement y drain**.

4. Prestaciones: Prestaciones Globales

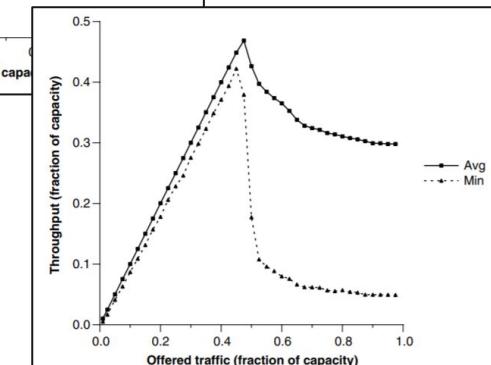
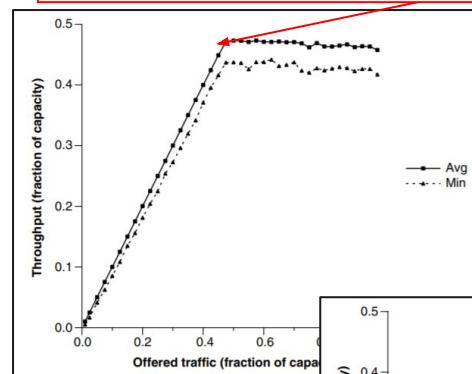
- (...) Estas medidas se hacen en **3 fases**:
 - Warm-up: Se inyecta el tráfico necesario para poner la red en equilibrio.
 - Measurement: Se ejecutan tantos ciclos de inyección de paquetes como se necesite. Estos van etiquetados con marca de tiempo.
 - Drain: Se van eliminando paquetes hasta comprobar que se han recibido todos (midiendo su tiempo).

Las mediciones de la latencia consideran todos los paquetes que llegan a destino en las dos últimas fases, pero no los enviados en las fases warm-up ni drain.

4. Prestaciones: Productividad (Throughput)

- Se mide contando todos los paquetes generados desde cualquier entrada a cualquier salida de la red en un periodo de tiempo.
- También se llama “Tráfico aceptado o productividad aceptada” en contraposición al “Tráfico ofrecido” que es la tasa de paquetes injectados.

Punto de saturación, representa el ancho de banda global para un patrón de comunicación dado.



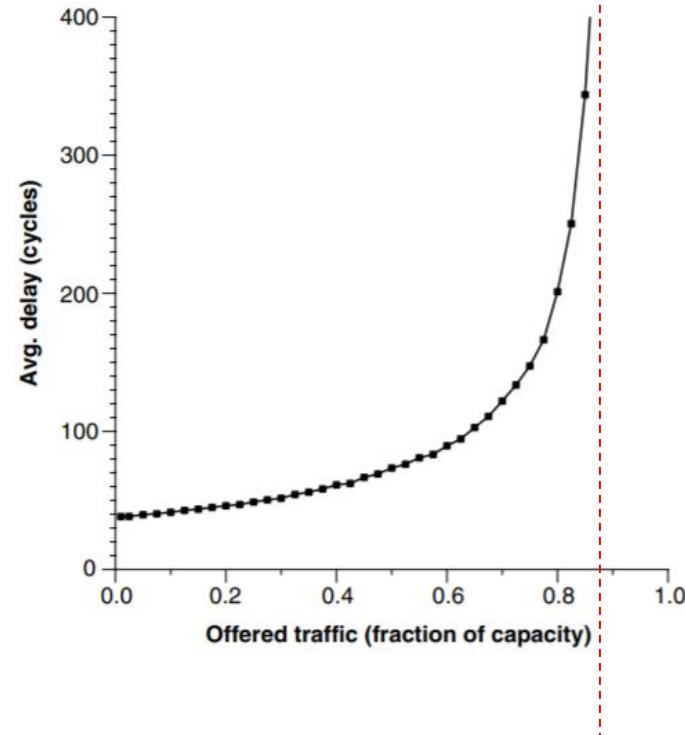
4. Prestaciones: Latencia

- **Latencia total de comunicación:** Tiempo total empleado en el envío de un mensaje de tamaño M desde que un procesador da la orden hasta que el procesador de destino acaba el procesamiento de lo recibido.
- **Latencia de transporte:** Tiempo que pasa desde que se inyecta en la red el primer bit del paquete hasta que el último bit llega al interfaz de destino. **Debe incluirse el tiempo de encaminamiento** (tiempo desde que la cabecera se inyecta en la red hasta que llega al destino).
- **Latencia de red observada** es el tiempo que transcurre desde que un nodo se desentiende del envío, hasta que el destino se da por enterado.

4. Prestaciones: Latencia

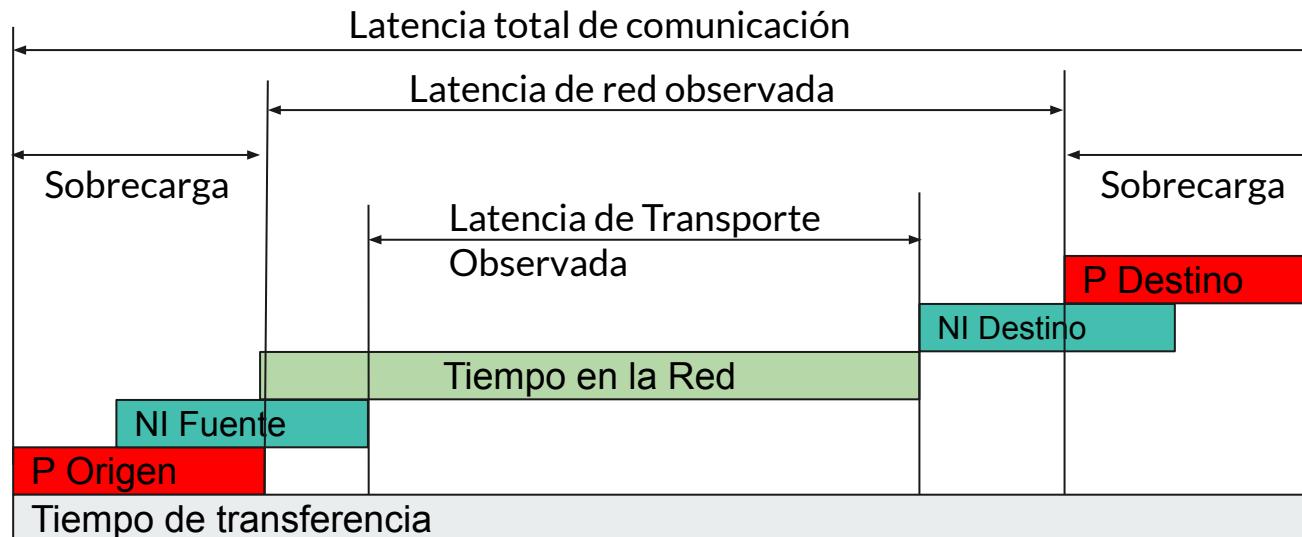
- La latencia o tiempo medio transcurrido en una comunicación en el sistema bajo un patrón de comunicación dado, se puede ver en función del tráfico ofrecido.
- A más tráfico más latencia, hasta que se alcanza el punto de saturación

Punto de saturación, representa el ancho de banda global para un patrón de comunicación dado.



4. Prestaciones: Conceptos básicos

- Tipos de latencia, relación con la comunicación y sobrecarga:



4. Prestaciones: Latencia y ancho de banda

- Latencia media: latencia de todos los paquetes dividida entre el número de paquetes.
- Productividad global o productividad aceptada: Bits/s que llegan por todas las salidas dividido entre el número de salidas (bits por segundo y nodo).
- Productividad solicitada o aplicada: Bits/s generados por todas las entradas dividido entre el número de entradas (bits por segundo y nodo).
- Productividad máxima o ancho de banda asintótico: Ancho de banda que se puede aprovechar de forma efectiva para una distribución de destinos dada (punto de saturación de la red).

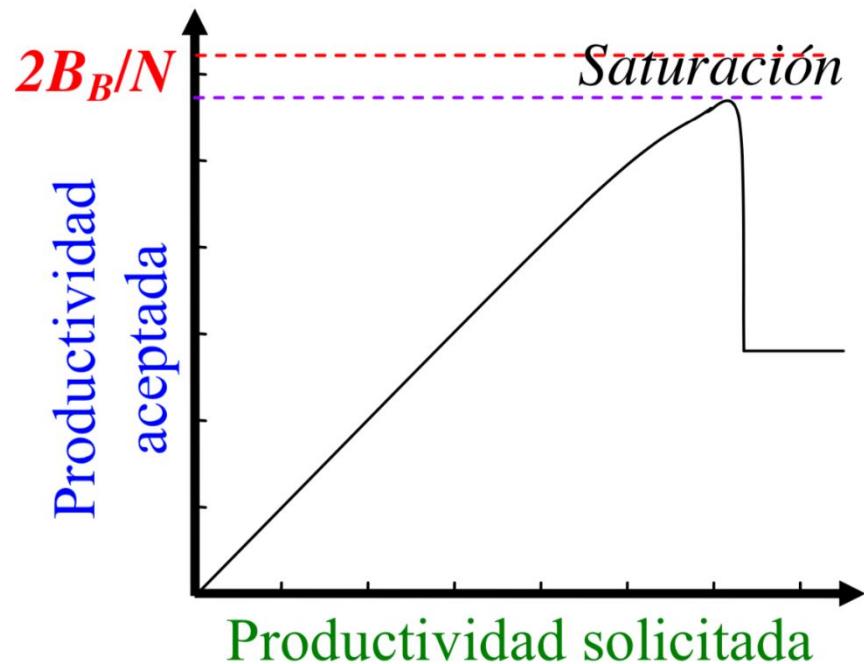
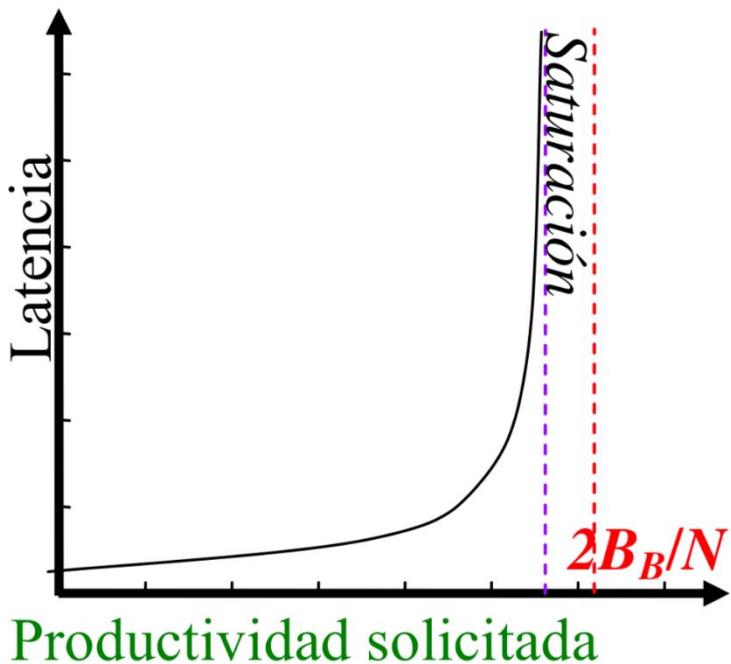
4. Prestaciones: Curiosidades de la latencia

- El punto de saturación de una red se puede aproximar por dos veces el ancho de banda de la bisección de la red (B_b) dividido por el número de nodos, (N):

$$\text{Punto de Saturación} = 2 \frac{B_b}{N}$$

- B_b es el ancho de banda existente en los canales □ a cortar en el ancho de la bisección. Representa la cantidad de datos que la red es capaz de enviar de una mitad a otra por unidad de tiempo.

4. Prestaciones: Curiosidades de la latencia



4. Prestaciones: Latencia

- La latencia se calcula como:

$$\text{Latencia total}(T) = \text{Latencia de la cabecera} (T_h) + \text{Latencia del cuerpo} \left(\frac{L}{b} \right)$$

- Siendo T_h el tiempo necesario para que la cabecera del mensaje llegue a su destino, L la longitud del mensaje y b el ancho de banda del canal por el que se ha enrutado.
- La latencia de la cabecera está formada por los factores, T_r y T_w , que son el tiempo de enrutamiento y el tiempo de transporte, respectivamente.

4. Prestaciones: Latencia

- Todos los cálculos se suponen en ausencia de contención, es decir sin que el paquete tenga que pararse para continuar:

$$\text{Latencia de la cabecera} \left(T_h \right) = F(N, Tr, Tw)$$

- La función que marca la latencia de cabecera será una función en la que intervienen el número de nodos a atravesar, N, el tiempo de enrutamiento de la cabecera al atravesar todos los nodos, T_r , y el tiempo del transporte físico de los bits, T_w .

4. Prestaciones: Latencia

- Si tomamos un caso base en el que el mínimo número de saltos que hay que dar para llegar desde un origen a un destino dado es H_{\min} y el tiempo de enrutamiento en cada salto es t_r .
- Y si tomamos que el tiempo de transporte, T_w , como el número de enlaces a atravesar como mínimo, D_{\min} , dividido por la velocidad de envío en ese enlace, v , ...entonces podemos afirmar que el tiempo mínimo de latencia es T_0 :

$$\text{Latencia de enrutamiento } (T_r) = H_{\min} t_r$$

$$\text{Latencia de transporte } (T_w) = \frac{D_{\min}}{v}$$

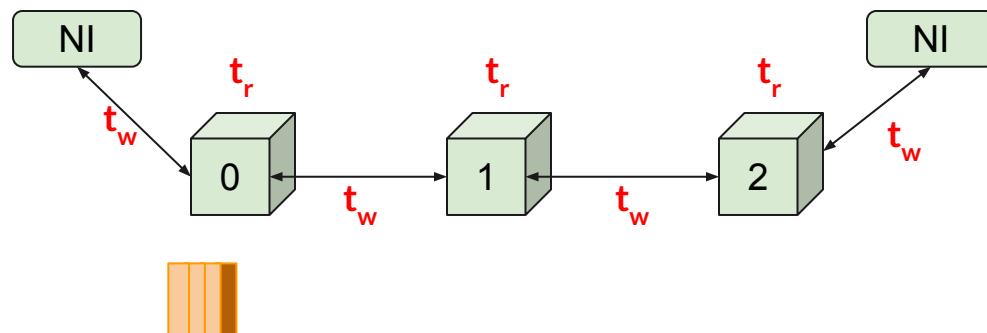
$$T_0 = H_{\min} t_r + \frac{D_{\min}}{v} + \frac{L}{b}$$

4. Prestaciones: Latencia

- T_0 es el tiempo mínimo en transferir un mensaje en nuestra red en ausencia de contención y con una carga mínima.
- Si introducimos un entorno más real, con contenciones, aparecerá otro término denominado T_c , que representara el tiempo que el paquete está retenido en los diferentes buffers de almacenamiento intermedio.
- H_{\min} , D_{\min} y b , son parámetros que vienen marcados por la topología.

4. Prestaciones: Latencia, Ejemplo

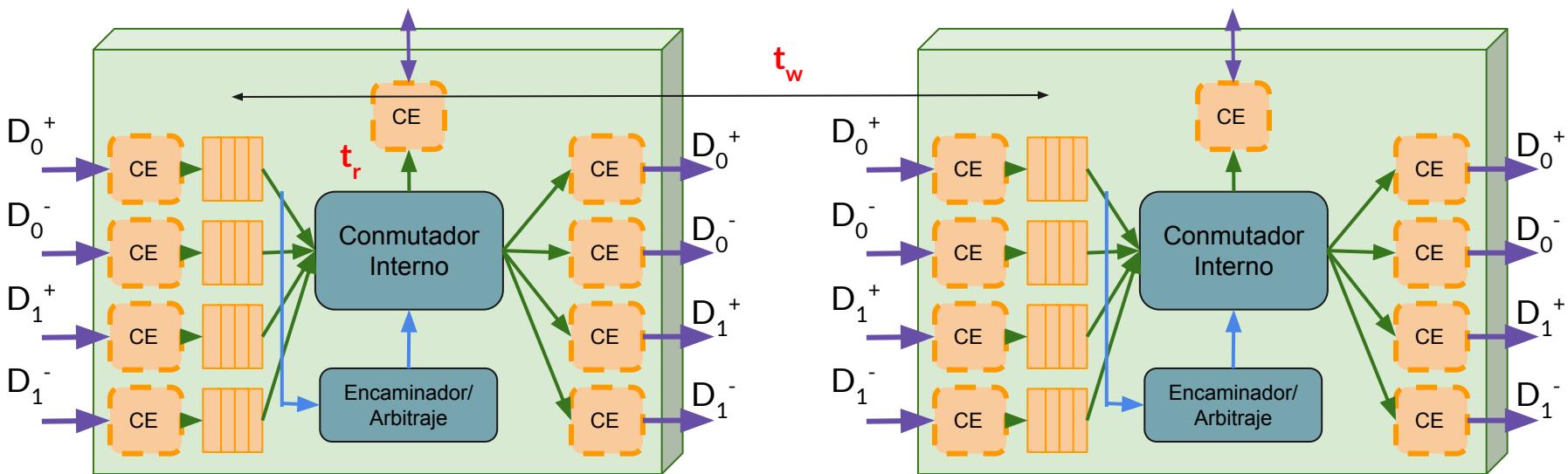
- 1 flit = 1 phit = w bits.
- Cabecera = 1 flit
- Longitud del mensaje, L , $L/w = 3$ flits
- D : Número de parejas comutador-enlace a atravesar desde el origen hasta el destino.
- Comutadores sin buffer a la salida
- Entorno sin contenciones



Paquete con 1
cabecera y 3 flits

El tiempo mínimo de la cabecera será $4*t_w + 3*t_r$
La distancia entre el nodo 0 y el 2 será 2

4. Prestaciones: Latencia, Ejemplo





—
Gracias.

Tema 3

Redes de Área de Sistema

Nicolás Calvo Cruz

Dpto. de Arquitectura y Tecnología de los Computadores

@nkalvocruz

nkalvocruz@ugr.es

Motivación

- Redes de **comunicación en computadores paralelos**.
- Hoy en día se están **sustituyendo los buses por redes** con conexiones **punto a punto** a todos los niveles:
 - Interno al chip
 - A nivel de tarjeta y placa
 - A nivel de chasis o caja
 - LAN y Router IP
- Conocer los algoritmos de **encaminamiento** y la **infraestructura** permite mejorar las **prestaciones**.

Objetivos

- Distinguir entre redes de altas prestaciones y redes estándar.
- Conocer la estructura general de un conmutador.
- Estudiar las topologías y nomenclaturas de las redes de altas prestaciones.
- Estudiar los algoritmos de encaminamiento.



Índice

1. Clasificación Sistemas de Comunicación
2. Propiedades
3. Diseñar una Red
4. Prestaciones
5. Enrutamiento
6. Técnicas de commutación
7. Ejemplo

5. Enrutamiento

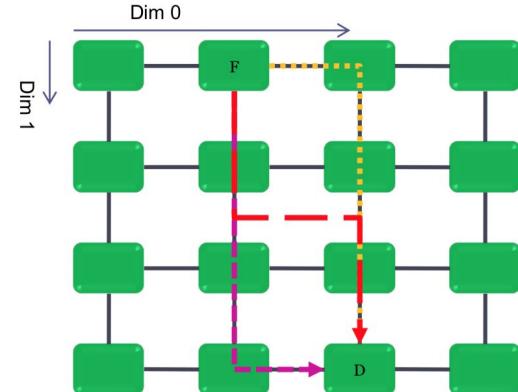
Algoritmos de enrutamiento

5. Enrutamiento

¿Qué hace un algoritmo de enrutamiento?

Un algoritmo de enrutamiento o encaminamiento ha de cumplir 2 funciones:

- Función de encaminamiento: Identificar qué rutas se pueden seguir en la red para llegar desde un cierto origen a un destino.
- Función de selección: Escoger, para cada paquete, qué camino seguir.



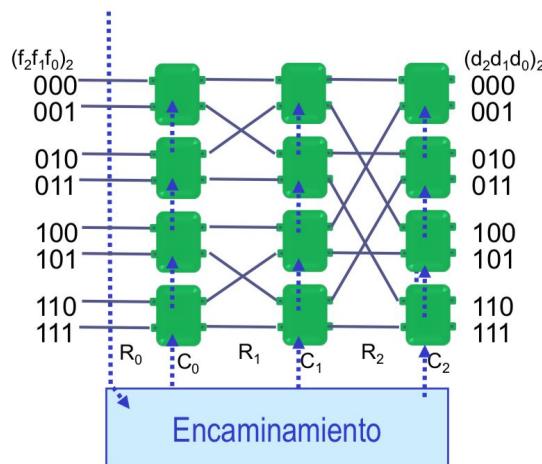
5. Enrutamiento

Diseño y clasificación según:

- Funcionalidad
- Decisión de encaminamiento
 - Centralizado
 - Fuente
 - Distribuido
 - Multifase
- Implementación
 - Con tabla de consulta
 - Sin tabla de consulta
- Selección del camino
 - Deterministas
 - Inconscientes
 - Adaptativos
- Canales candidatos y caminos alternativos (función de encaminamiento)

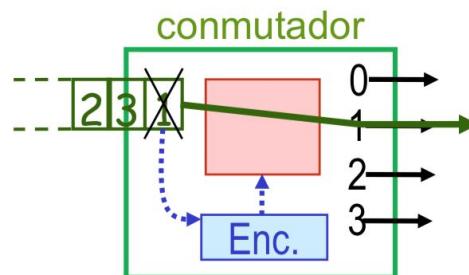
5. Enrutamiento

Decisión de enrutamiento:



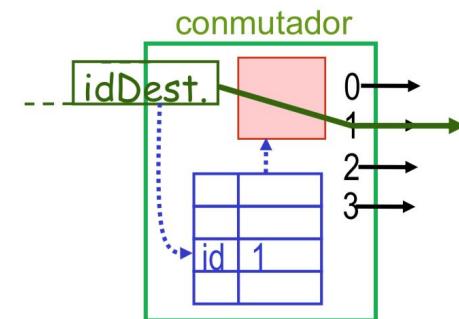
Centralizado

Circuitería común genera las señales de control



Fuente

Los canales escogidos se fijan en la fuente



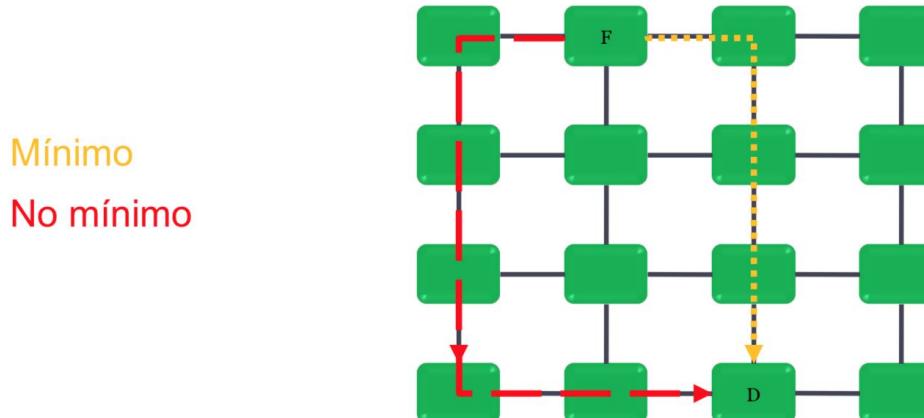
Distribuido

Cada comutador decide localmente (p.ej. consultando una tabla)

5. Enrutamiento

Selección del camino:

- Adaptativo progresivo o con retroceso (*backtraking*)
- Provechoso/mínimo o mal-enrutado/no-mínimo
- Completamente adaptativo o parcialmente adaptativo



5. Enrutamiento

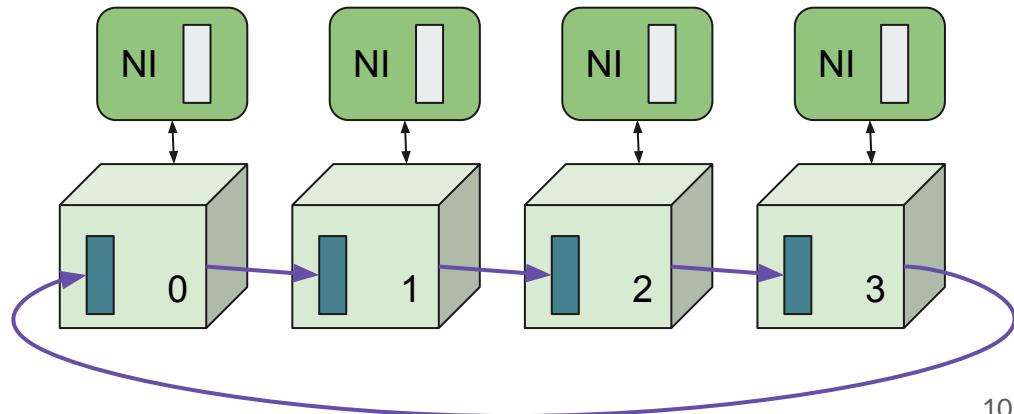
Enrutamiento en toro unidireccional: Distribuido, sin tablas, determinista

Entrada: Coordenada del nodo actual A y del nodo destino D

Salida: Canal de salida seleccionado (C, I)

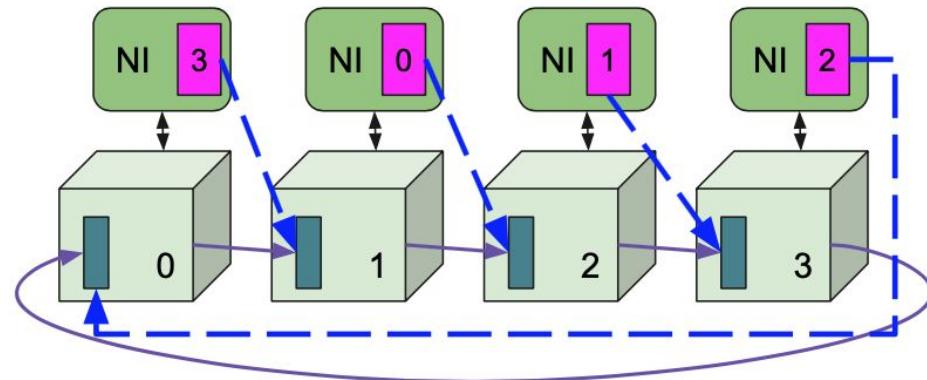
Proceso:

```
if (A==D) then Canal = I  
else Canal = C
```



5. Enrutamiento: Interbloqueo

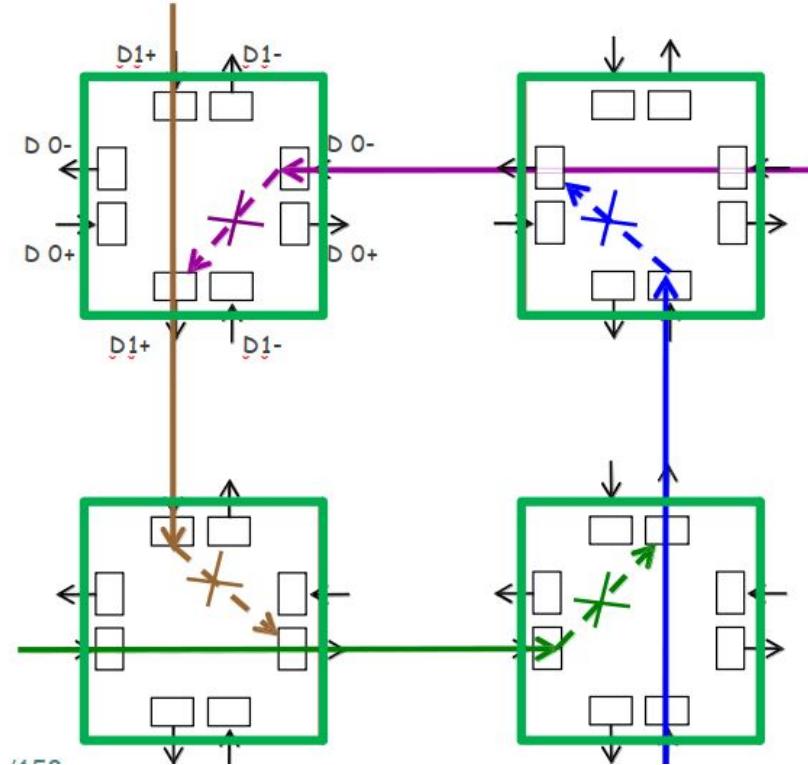
- Conjunto de paquetes bloqueados a la espera de que algún hueco (**recurso**) quede libre.
- Depende de:
 - **Topología:** Si no hay ciclos no hay riesgo de interbloqueo
 - **Algoritmo de enrutamiento:** Evitando ciclos, no habrá interbloqueos



5. Enrutamiento: Interbloqueo

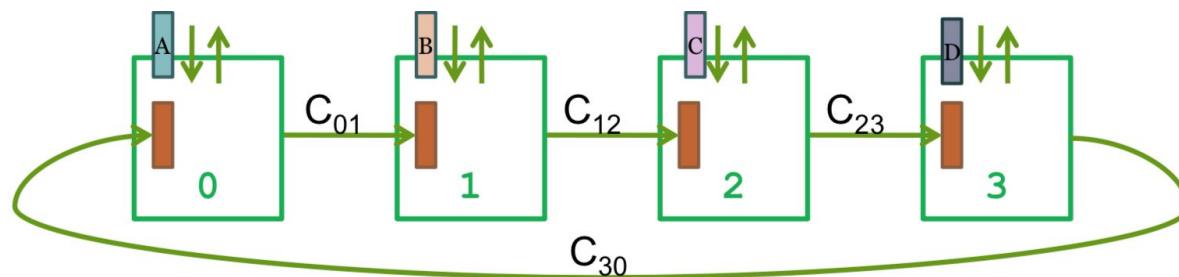
- Siempre que haya **ciclos** en la topología, puede haber **interbloqueos**.

Malla 2D:



5. Enrutamiento: Interbloqueo

- Grafo de dependencias: Grafo que incluye la lógica del algoritmo de enrutamiento y la topología de red para ver si sería posible un interbloqueo.



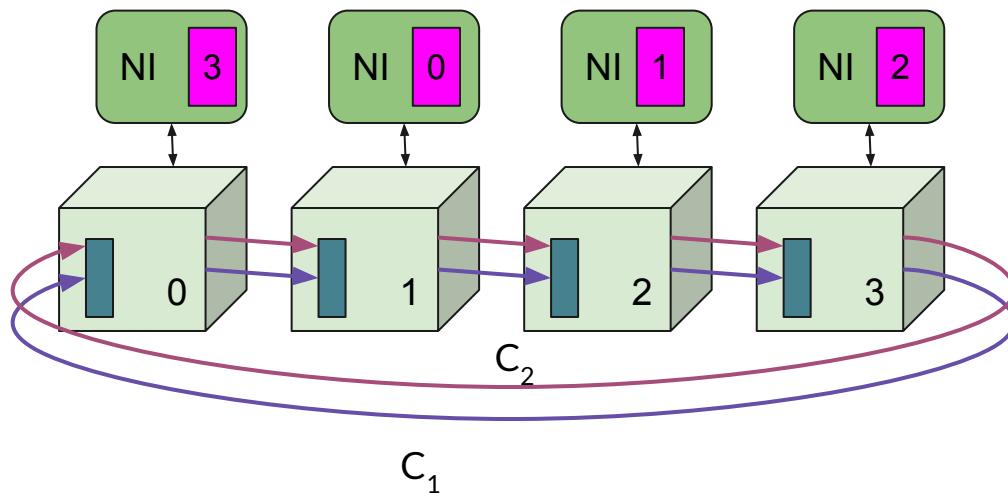
- Hay una dependencia entre el canal C1 y el C2 si un paquete que viene por el canal C1, puede encaminarse por el canal C2.

5. Enrutamiento: Interbloqueo

- Tratamiento de Interbloqueos:
 - Se pueden permitir, pero hay que **detectarlos y eliminarlos**.
- Detección:
 - Contador de saltos / tiempo
- Eliminación:
 - Suprimiendo paquetes del ciclo
 - Usando un camino alternativo
 - Introduciendo **canales virtuales**

5. Enrutamiento: Interbloqueo

Canales virtuales:



Entrada: Nodo actual, A y Nodo destino, D

Salida: Canal (C, I)

Proceso:

If ($D > A$) then $\text{Canal} = C_2$;

If ($D < A$) then $\text{Canal} = C_1$;

If ($D = A$) then $\text{Canal} = I$;

5. Enrutamiento

Objetivos:

- Balancear la **carga** evitando paradas y retrasos (Throughput)
- Minimizar los **tiempos** de enrutamiento (Latencia)
- Minimizar los **saltos** (Latencia)
- Evitar **ciclos** e interbloqueos (Latencia y Throughput)

Todos los objetivos se contradicen de alguna forma: **priorizar**.

5. Enrutamiento

Opciones:

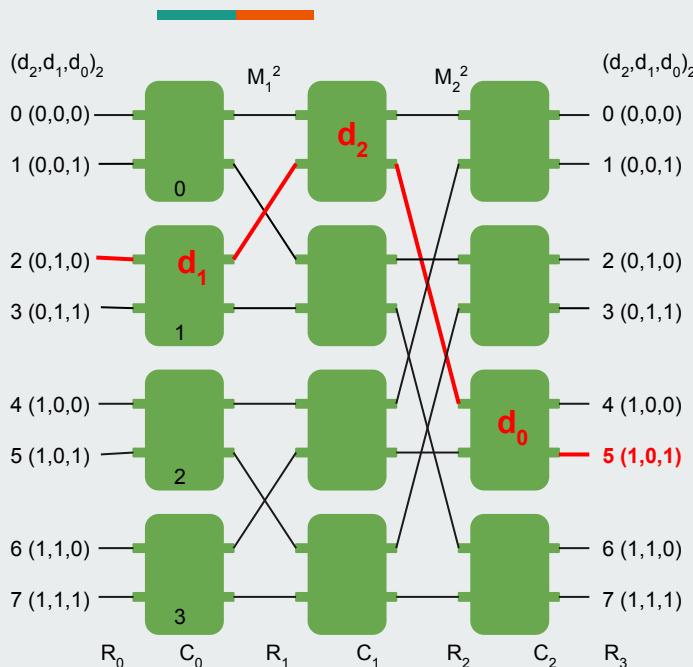
- 1) Ignorar el tráfico y usar algoritmos inconscientes, que dan mal balanceo de carga, retrasos o interbloqueos. **Algoritmos inconscientes y/o deterministas.**
- 2) Tener en cuenta el tráfico y que el algoritmo balancee la carga aunque tarde más en enrutar y aumente la latencia media. **Algoritmos adaptativos**

La solución deberá buscar un **compromiso** entre extremos.

5. Enrutamiento: Alternativas

- 1) Greedy: Se escoge el camino más corto (en cualquier sentido). Si hay empate, se elige aleatoriamente.
- 2) Aleatorio uniforme: Se elige de forma uniformemente aleatoria en sentido horario o contrario.
- 3) Aleatorio no uniforme: Se le da cierta probabilidad al camino más corto, pero el otro también es posible (ponderación).
- 4) Adaptativo: Enviar el paquete por el sentido menos cargado según el recuento de paquetes en ese sentido en los últimos T ciclos de reloj.

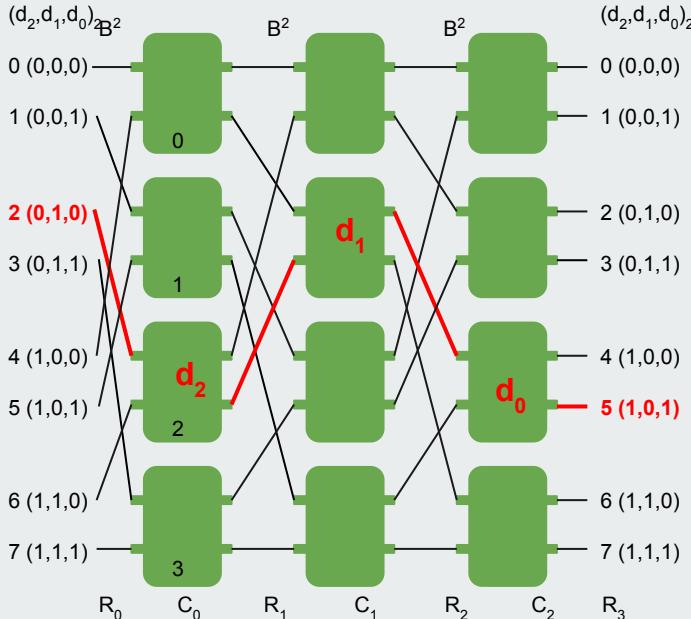
5. Enrutamiento determinista en redes mariposa (unidireccionales) Enrutamiento-TAG



- Cada uno de los dígitos de la dirección de destino se utiliza para seleccionar el puerto de salida de cada etapa de conmutación de la red.
- El bit d_i controla la etapa $i-1$ y el bit d_0 la etapa $n-1$
- Ejemplo: destino 101 (nodo 5)
 - El $d_2 = 1$ controla la etapa $2-1=1$
 - El $d_1 = 0$ controla la etapa 0
 - El $d_0 = 1$ controla la etapa 2
- Ejemplo envío desde 2 hasta 5.
- Este algoritmo no provoca ciclos porque la topología no tiene ciclos.

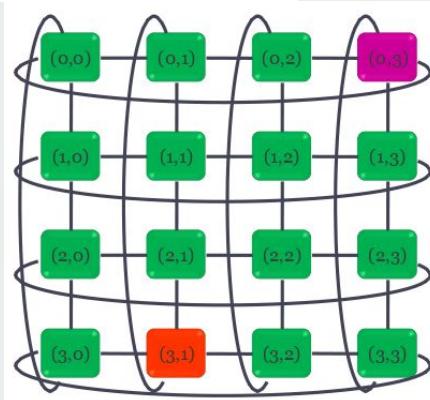
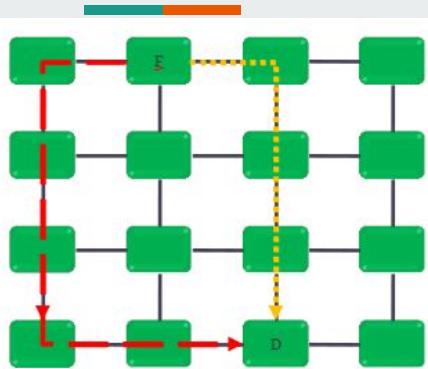
5. Enrutamiento determinista en redes omega (unidireccionales)

Enrutamiento-TAG



- Cada uno de los dígitos de la dirección de destino se utiliza para seleccionar el puerto de salida de cada etapa de conmutación de la red.
- Red omega: El bit d_i controla la etapa $n-1-i$
- Ejemplo: destino 101 (nodo 5)
 - El $d_2 = 1$ controla la etapa $3-1-2=0$
 - El $d_1 = 0$ controla la etapa $3-1-1=1$
 - El $d_0 = 1$ controla la etapa $3-1-0=2$
- Ejemplo envío desde 2 hasta 5.
- Este algoritmo no provoca ciclos porque la topología no tiene ciclos.

5. Enrutamiento en redes estrictamente ortogonales



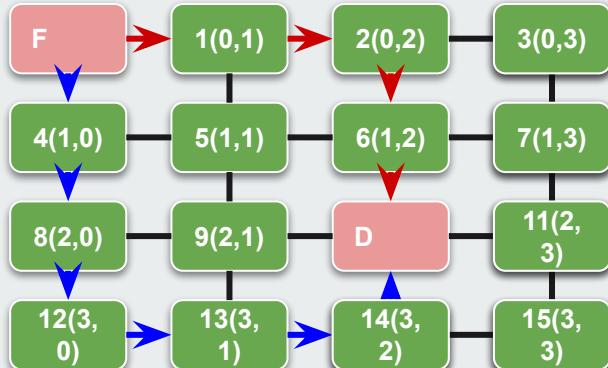
- Encaminamiento ordenado por dimensión creciente o decreciente:
 - Con implementación algorítmica igual para todas las entradas
 - Con implementación algorítmica diferente para todas las entradas
 - Algoritmo del intervalo, con implementación por tablas
- **Creciente:** los paquetes se enrutan siguiendo primero los enlaces de la dimensión menor hacia enlaces de la dimensión mayor.
- **Decreciente:** los paquetes se enrutan siguiendo primero los enlaces de la dimensión mayor a la menor.

5. Enrutamiento: Ordenado por Dimensión

Características:

- Envío hasta un destino $D=(d_{n-1}, d_{n-2}, \dots, d_2, d_1, d_0)_k$
- Proceso:
 - Calcular las distancias hasta el nodo D.
 - Encaminar anulando las distancias en cada dimensión según un orden creciente o decreciente
- Implementación:
 - Distribuido, sin tabla, determinista (XY)
 - Fuente, sin tabla, determinista (street sign)
 - Distribuido con tabla, determinista (Intervalo)
- Interbloqueos: la lógica del algoritmo **evita los ciclos en mallas e hipercubos**. Para redes toro se necesitan **canales virtuales** para eliminar los ciclos.

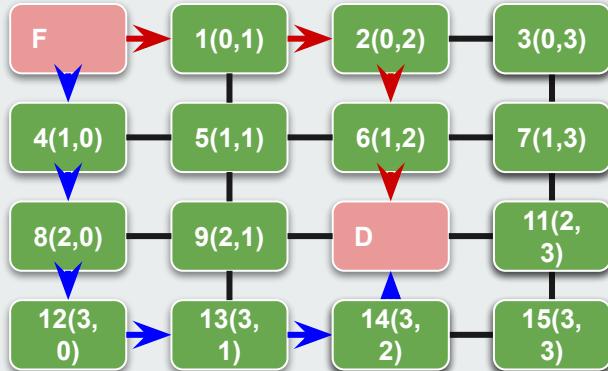
5. Enrutamiento: Ordenado por Dimensión



- Distribuido-sin tabla-determinista, algoritmo común a todas las entradas. **Sigue orden creciente**, se anula la distancia en la dimensión 0 y después en la dimensión 1.
- Entrada: Nodo actual, $A(a_1, a_0)$, Nodo destino $D(d_1, d_0)$.
- Salida: Canal de salida, $CS = D_0^+, D_0^-, D_1^+, D_1^-$, I
- Procedimiento:

```
dist0=d0-a0;  
dist1=d1-a1;  
if(dist0<0) cs=d0-;  
if(dist0>0) cs=d0+;  
if(dist0==0 && dist1<0) cs=d1-;  
if(dist0==0 && dist1>0) cs=d1+;  
if(dist0==0 && dist1==0) cs = I;
```

5. Enrutamiento: Ordenado por Dimensión



- Distribuido-sin tabla-determinista, algoritmo diferente para cada entrada. **Sigue orden creciente**, se anula la distancia en la dimensión 0 y después en la dimensión 1.
- Entrada: Nodo actual, $A(a_1, a_0)$, Nodo destino $D(d_1, d_0)$.
- Salida: Canal de salida, $CS = D_0^+, D_0^-, D_1^+, D_1^-, I$
- Procedimiento:

Canal D0-

Entrada: distancias (dist1, dist0)

Salida: Canal cs=(D0-, D1-, D1+, I).

Procedimiento:

```
if ( dist0 ≠ 0 ) { cs = D0- ; dist0--; }
```

```
if ( dist0=0 & dist1<0 )
```

```
{ cs = D1- ; dist1--; }
```

```
if ( dist0=0 & dist1>0 )
```

```
{ cs = D1+ ; dist1--; }
```

```
if ( dist0=0 & dist1=0 ) cs = I ;
```

Canal D1+

Entrada: distancias (dist1)

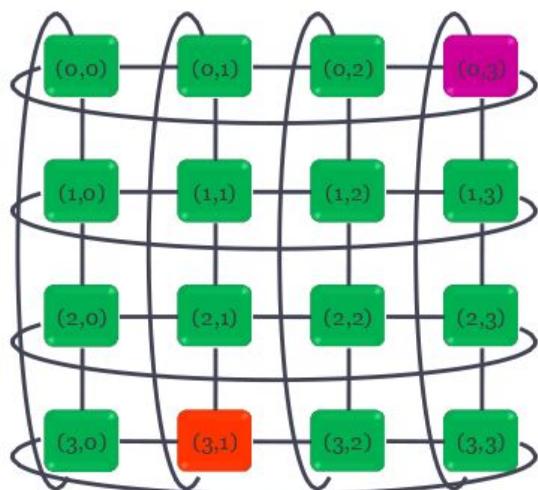
Salida: Canal cs=(D1+, I).

Procedimiento:

```
if ( dist1 ≠ 0 ) {cs = D1+ ; dist1--; }
```

```
if ( dist1=0 ) cs = I ;
```

5. Enrutamiento: Ordenado por Dimensión



- Distribuido-sin tabla-determinista, algoritmo igual para cada entrada. Sigue orden creciente, se anula la distancia en la dimensión 0 y después en la dimensión 1.
- Entrada: Nodo actual, $A(a_1, a_0)$, Nodo destino $D(d_1, d_0)$.
- Salida: Canal de salida, $CS = D_0^+, D_0^-, D_1^+, D_1^-, I$
- Procedimiento:

Entrada: Actual $A = (a_1, a_0)_k$, $D = (d_1, d_0)_k$

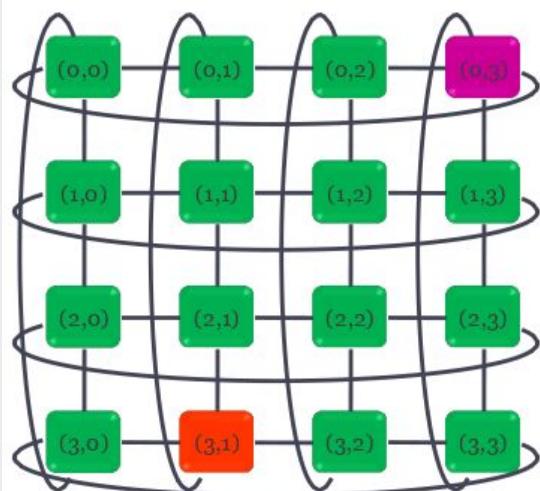
Salida: Canal $cs=(D0+, D0-, D1-, D1+, I)$.

Procedimiento:

```
dist0+=(d - a )0 mod 4; dist0-=(a - d )0 mod 4;  
dist1+=(d1 - a1) mod 4; dist1-=(a1 - d1) mod 4;  
if ( dist0≠0 && dist0+>=dist0- ) cs = D0- ;  
if ( dist0≠0 && dist0+<dist0- ) cs = D0+ ;  
if ( dist0==0){  
    if ( dist1≠0 && dist1+>=dist1- ) cs = D1- ;  
    if ( dist1≠0 && dist1+<dist1- ) cs = D1+ ;  
    if ( dist1==0) cs = I;  
}
```

- Interbloqueos: NO está libre de interbloqueos

5. Enrutamiento: Ordenado por Dimensión



- Si incluimos los canales virtuales:

Entrada: Actual $A = (a_1, a_0)_k$, $D = (d_1, d_0)_k$

Salida: Canal $cs = (D0^{+1}, D0^{-1}, D1^{-1}, D1^{+1}, D0^{+2}, D0^{-2}, D1^{-2}, D1^{+2}, l)$.

Procedimiento:

```
dist0+ = (d0 - a0) mod 4; dist0- = (a0 - d0) mod 4;
```

```
dist1+ = (d1 - a1) mod 4; dist1- = (a1 - d1) mod 4;
```

```
if ( dist0 ≠ 0 && dist0+ >= dist0- )
```

```
    if (d0 > a0) cs = D0-1 else cs = D0-2;
```

```
if ( dist0 ≠ 0 && dist0+ < dist0- )
```

```
    if (d0 < a0) cs = D0+1 else cs = D0+2;
```

```
if ( dist0 == 0 ) {
```

```
    if ( dist1 ≠ 0 && dist1+ >= dist1- )
```

```
        if (d1 < a1) cs = D1-1 else cs = D1-2;
```

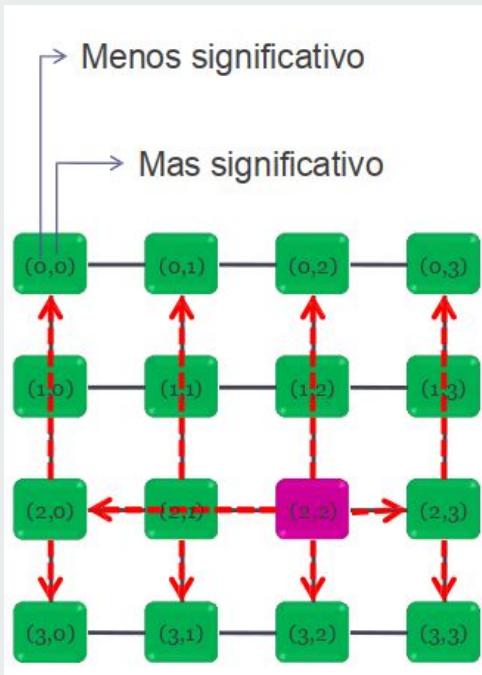
```
        if ( dist1 ≠ 0 && dist1+ < dist1- )
```

```
            if (d1 < a1) cs = D1+1 else cs = D1+2;
```

```
            if ( dist1 == 0 ) cs = l;
```

```
}
```

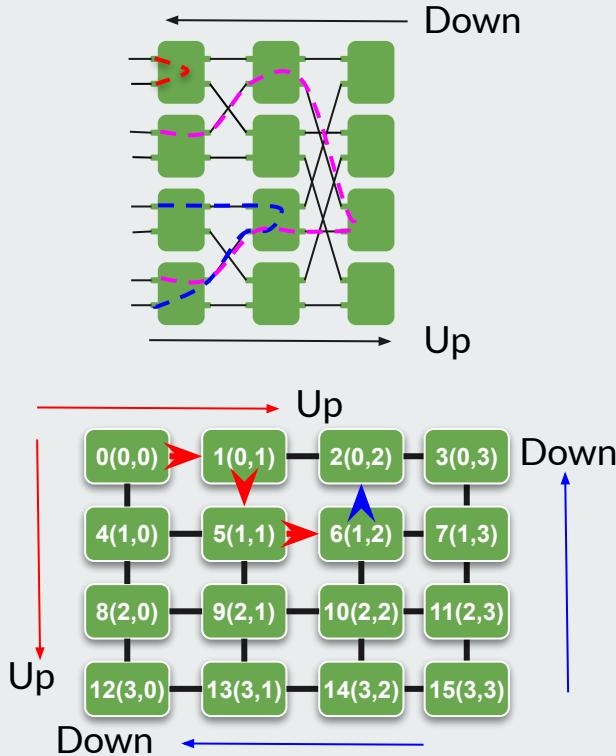
5. Enrutamiento: Ordenado por Dimensión



- Algoritmo del Intervalo: Distribuido, con tabla, algoritmo común a todas las entradas. orden creciente
- Se determina para cada nodo, la tabla de envíos indicando para cada enlace el intervalo de nodos a los que da acceso.

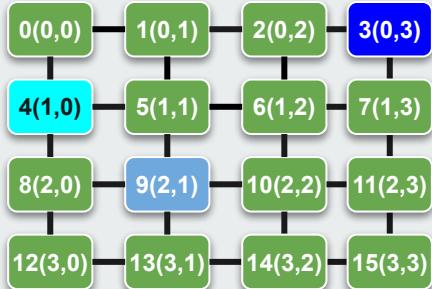
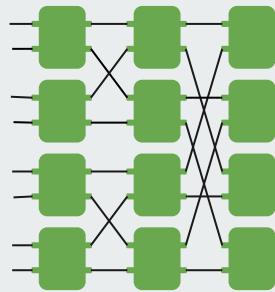
Canal	Intervalo
D0+	12-15
D0-	0-7
D1+	11-11
D1-	8-9
I	10-10

5. Enrutamiento: Up-Down



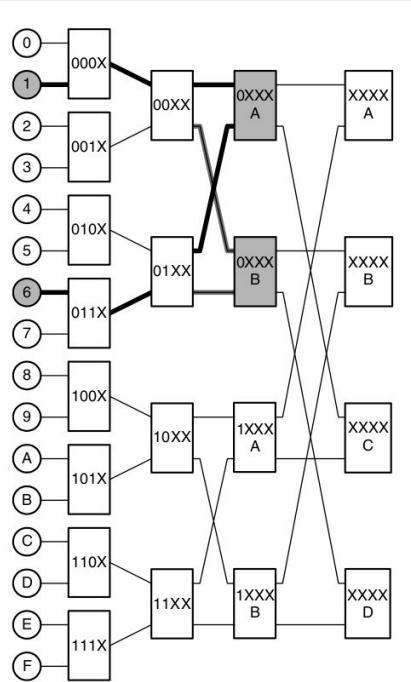
- **Algoritmo Up-Down**
- Se aplica a redes ortogonales y multietapa
- Implementación: Adaptativo o inconsciente.
- Interbloqueos: Evita interbloqueos si la topología no tiene ciclos. Si los tiene, evita los ciclos tomando solo enlaces Up o Down en orden, pero no alternándolos.
- Procedimiento:
 - Se asigna un sentido up o down a todos los enlaces siguiendo siempre el mismo patrón:
 - Redes ortogonales, Up (+) y Down (-)
 - Redes dinámicas bidireccionales Up (raíz) y down (hojas). Ofrece caminos alternativos
 - Se eligen canales **Up hasta que no haya alternativa y se comienzan a coger canales Down**.
- Implementación: Puede ser **adaptativo o parcialmente adaptativo, mínimo o no mínimo**.

5. Enrutamiento: Valiant



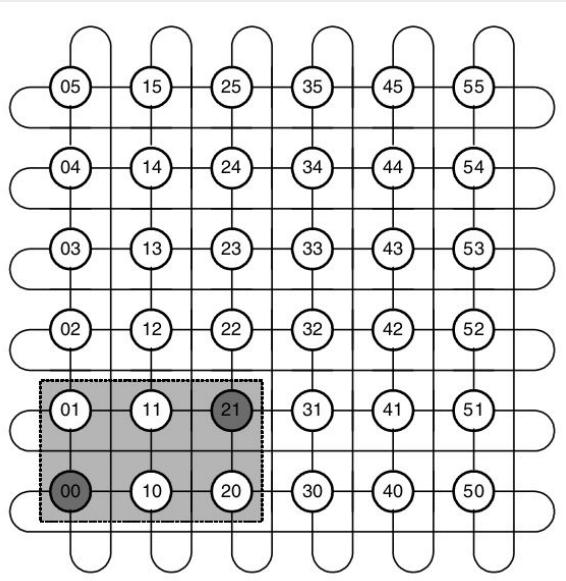
- **Algoritmo De Valiant**
- Se aplica a redes **ortogonales** y multietapa
- Es **inconsciente** y balancea la carga muy bien.
Inconvenientes: **incrementa la latencia media** y produce una ocupación de canales que **puede saturar** la red.
- Procedimiento: Tiene **2 fases**:
 - a. Se elige desde el nodo origen un paso intermedio cualquiera, elegido de forma aleatoria o con algún criterio.
 - b. Desde ese nodo intermedio elegido, se enruta hacia el destino final.
- Implementación: Puede ser **mínimo** o no **mínimo**

5. Enrutamiento: Valiant mínimo

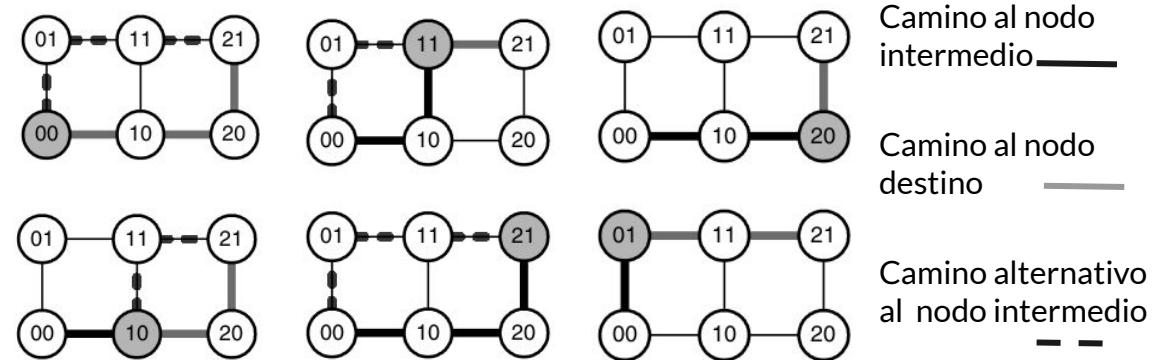


- Valiant, también se puede implementar un algoritmo **inconsciente** y que sea mínimo.
- Las máscaras de cada conmutador fijan el **camino** a seguir en la selección de caminos.
- Se selecciona como nodo **intermedio** algún conmutador previo al destino y desde ahí hasta el destino.
- Ej: Para ir **desde el nodo 1 al 6 (011X)**, se **selecciona** entre los conmutadores que están **antes** en el árbol y que son comunes al 1 y al 6, en este caso el A(0XXX) o el B(0XXX) y ya **desde ahí al destino** usando los bits de la dirección del nodo destino.

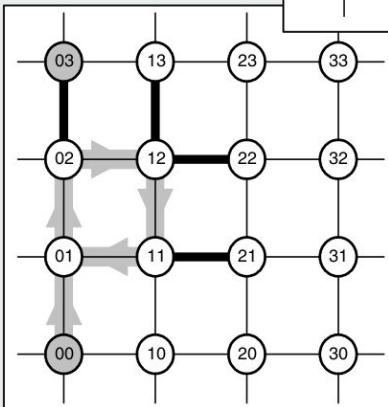
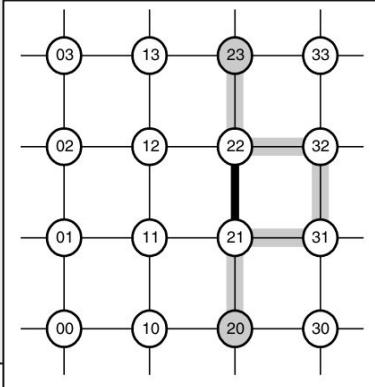
5. Enrutamiento: Valiant mínimo



- Para el caso de redes ortogonales:
 - Se elige el nodo **intermedio** de forma que pertenezca al conjunto de nodos encerrados en el cuadrado mínimo que incluye al nodo **origen** y al nodo **destino**. Para ello se calculan las distancias en cada dimensión entre origen y destino y la suma de dichas distancias será la dimensión de la subred entre la que hay que elegir el nodo intermedio.
 - Se enruta hacia ese nodo seleccionado y después hacia el nodo destino. Todas las opciones serán **caminos mínimos** posibles.



5. Enrutamiento: Adaptativo



- Cualquiera de estos últimos algoritmos se puede implementar de forma adaptativa si, para seleccionar de entre los caminos alternativos, se tiene en cuenta el estado de la red.
- Adaptación global, si tenemos información global de la red
- Adaptación local, si se utiliza sólo información del entorno más cercano a donde se está enrutando.
- Criterios que se pueden consultar para saber el estado de la red
 - Ocupación de los buffers de salida
 - Ocupación media en los últimos T ciclos de los buffers de salida
 - Backpressure, que es el reflejo que pasado un tiempo se nota en los buffers anteriores a un enlace saturado. Si los buffers son grandes, esta presión tarda en notarse. Si los buffers son pequeños, se nota pronto hacia atrás.
- Ejemplo de algoritmo totalmente adaptativo: Para cada paquete, se calcula el camino mínimo a su destino y si el enlace tiene una ocupación menor a un umbral, se enruta en ese salto por ahí. Si no, se envía por el canal más vacío, camino mínimo o no. Podría incluso volverse con retroceso y ciclos.

Índice

- 1. Clasificación Sistemas de Comunicación**
- 2. Propiedades**
- 3. Diseñar una Red**
- 4. Prestaciones**
- 5. Enrutamiento**
- 6. Técnicas de commutación**
- 7. Ejemplos**

6. Técnicas de comunicación

6. Técnicas de conmutación

- Determina **cómo la información viaja** de un nodo fuente a otro destino de la red.
 - Almacenamiento y Reenvío:
 - Usado inicialmente en multicomputadores y redes WAN
 - Vermiforme (*Wormhole*):
 - Inicialmente destinado a multicomputadores
 - Virtual Cut-Through:
 - Propuesto para cualquier tipo de red
 - Conmutación de circuitos:
 - Originalmente propuesto en redes de telefonía

6. Técnicas de conmutación: Almacenamiento y reenvío

En una red con **comutación de almacenamiento y reenvío o *de paquetes***:

- El conmutador espera a recibir (almacenar) la unidad de transferencia entre interfaces (paquete) antes de enrutar, y tras esto, se reenvía.
- En todo momento el paquete ocupa, como mucho, un canal de la red.
- El almacenamiento del conmutador debe permitir guardar todo el paquete.
- Buffer y enlace se asignan a nivel de paquete.

6. Técnicas de conmutación: Vermiforme (*Wormhole*)

En una red con **comutación vermiforme** o *wormhole*:

- El **enrutamiento** del conmutador se ejecuta al recibir la cabecera de la unidad de transferencia entre interfaces, **sin esperar al resto**, que seguirán a la cabecera en orden, según vayan llegando.
- En cada momento, **un paquete** se está transfiriendo por **múltiples canales en paralelo**. El camino se divide en **etapas**, de forma que los flits de un mismo paquete pueden estar ocupando simultáneamente varias etapas, que se separan mediante almacenamientos. Más etapas, más recursos usados por paquete y menor latencia.
- **Buffer y enlace** se asignan a nivel de flit.

6. Técnicas de conmutación: Virtual Cut-through

En una red con conmutación *Virtual Cut-Through*:

- El enrutamiento es en cuanto llega la cabecera. La unidad de transferencia entre interfaces puede romperse (*cut-through*).
- Si no hay bloqueo, un paquete puede ir en paralelo por varios canales. Aunque si lo hay, **debe poder almacenarse en el conmutador**.
- Si los mensajes no tienen un tamaño máximo, se deben dividir en unidades más pequeñas: Segmenta el camino en etapas enviando por el camino segmentado de un paquete como en vermiforme, aunque asignando **buffer** y **enlaces a nivel de paquete** como en almacenamiento y reenvío.

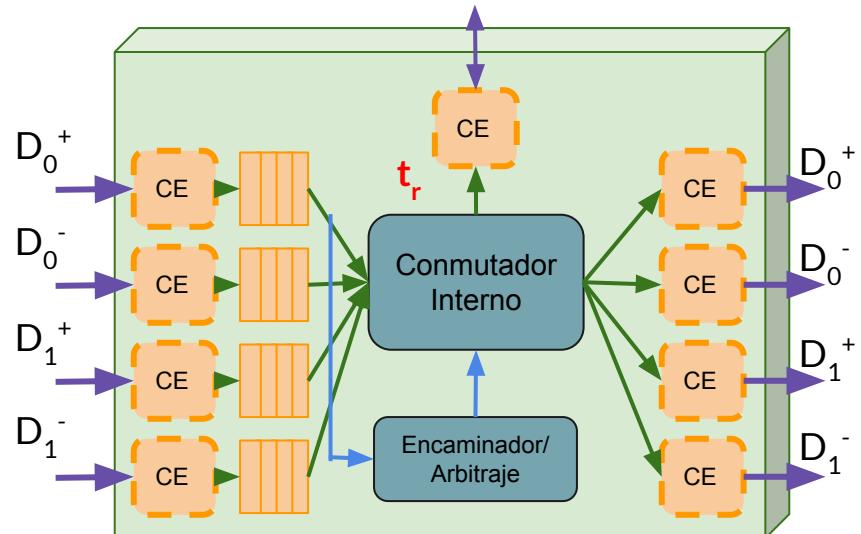
6. Técnicas de conmutación: Circuitos

La **comutación de circuitos** se hereda de los sistemas telefónicos clásicos:

- Reserva el camino entre interfaces que se usa en la transferencia:
 - Completo: Se manda una **sonda** (flit cabecera) que va reservando, y al llegar a destino, éste envía un ACK. Con el ACK en fuente, comienza un envío por canal de comunicación continuo (sin almacenamientos).
 - Segmentado/Etapas: El envío entre extremos es como en conmutación vermiforme, haciendo que los flits de un paquete ocupen varias etapas a la vez. El almacenamiento asociado a cada canal en los conmutadores es de un flit.
- La cola del paquete va liberando el camino reservado.

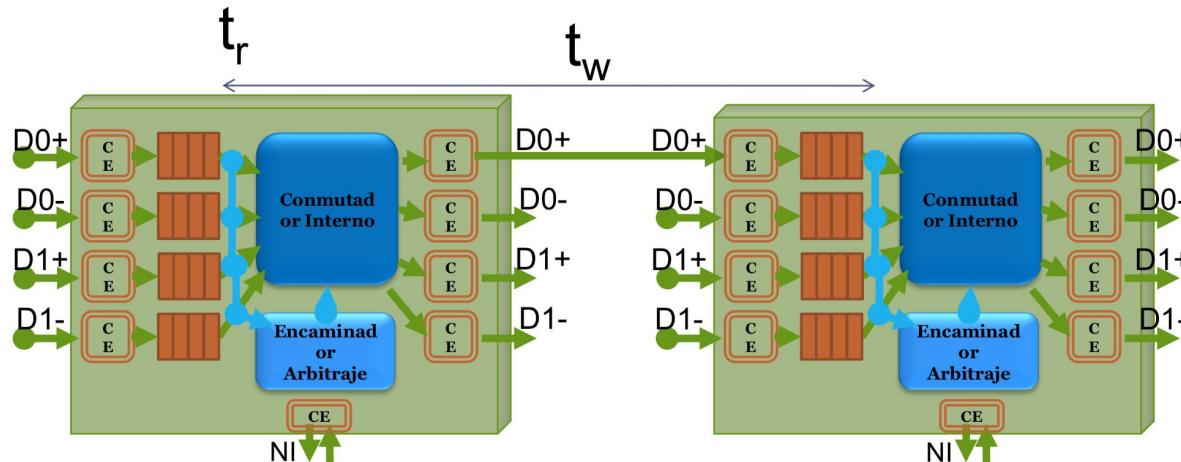
6. Técnicas de conmutación. Ejemplo práctico

- Situación base:
 - 1 flit = 1 phit = w bits
 - Tamaño de la cabecera = 1 flit
 - Tamaño de los datos L, $L/w = 3$ flits
 - Usaremos conmutadores sin buffer a la salida
 - D = número de parejas conmutador-enlace que unen fuente y destino.
 - Asumimos siempre un camino sin retenciones

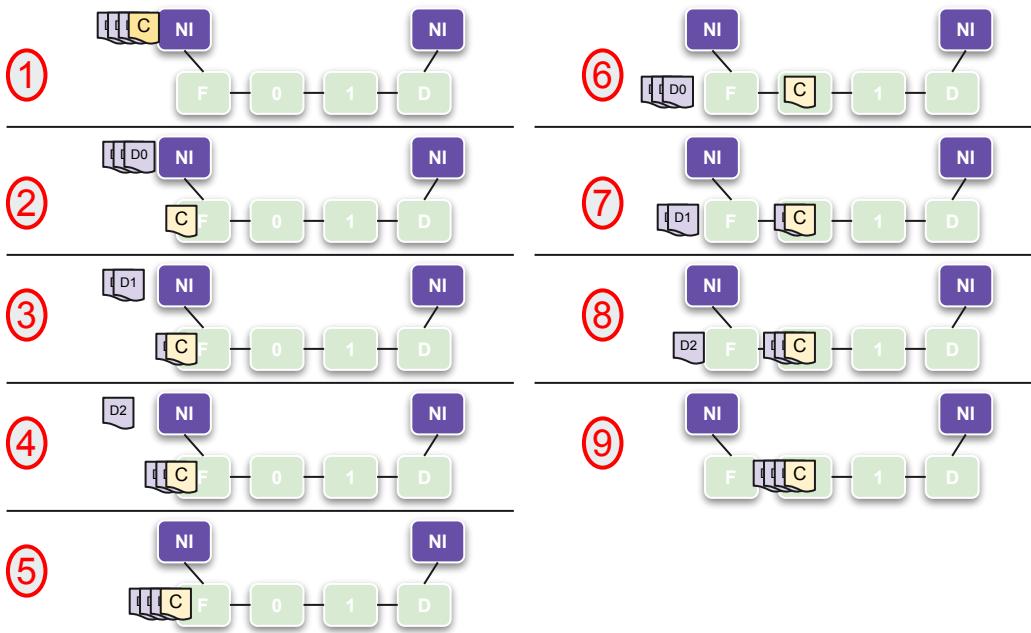


6. Técnicas de conmutación. Ejemplo práctico

- t_w : tiempo necesario para transportar w bits en una etapa conmutador-enlace.
- t_r : tiempo de enrutamiento.



6. Técnicas de conmutación. Almacenamiento y Reenvío



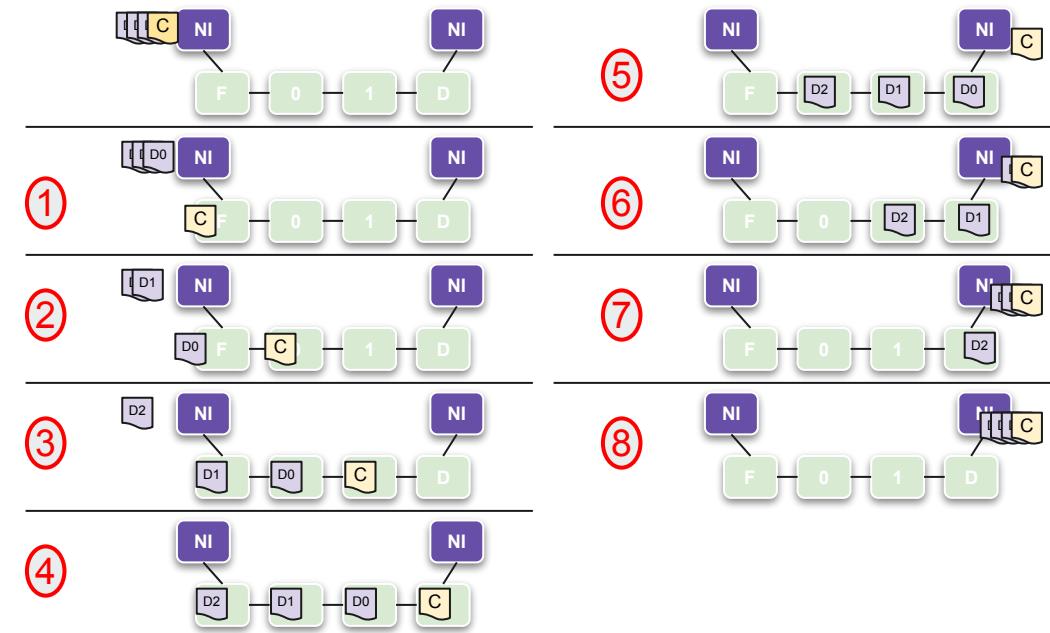
$$t_{AR} = D \left[t_r + t_w \left(\left\lceil \frac{L}{w} \right\rceil + 1 \right) \right] + t_w \left(\left\lceil \frac{L}{w} \right\rceil + 1 \right)$$

T	R	T
1		
2		t_w
3		t_w
4		t_w
5		t_w
6	t_r	t_w
7		t_w
8		t_w
9		t_w

Un paquete ocupará sólo un enlace, por lo que los retrasos provocados no serán nada más que los que haya esperando en ese enlace. La incidencia en el ancho de banda global es mínima. Almacenamientos de como mínimo lo que ocupe un paquete.

6. Técnicas de conmutación. Vermiforme (Wormhole)

$$t_V = t_c + t_r = D \left[t_r + t_w \right] + t_w \left(\frac{L}{W} \right)$$



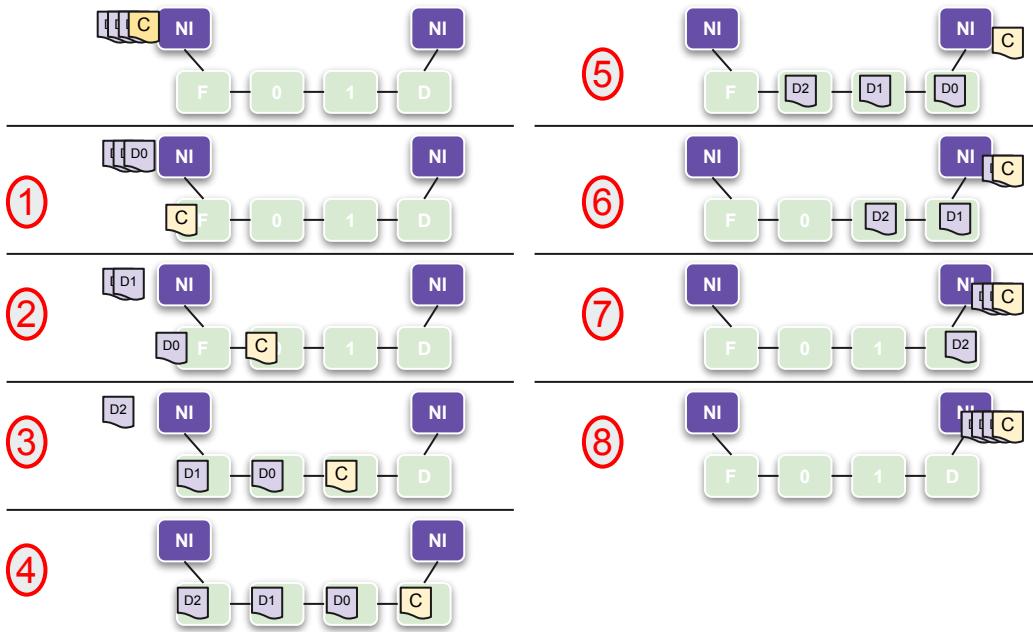
T	R	T
1		t_w
2	t_r	t_w
3	t_r	t_w
4	t_r	t_w
5	t_r	t_w
6		t_w
7		t_w
8		t_w

Un paquete ocupará tantos enlaces como necesite por su longitud, por lo que los retrasos afectarán a varios comutadores.

Almacenamientos de como mínimo lo que ocupe un flit.
 Si el paquete se retrasa, se queda extendido.
 La incidencia en el ancho de banda global es significativa.

6. Técnicas de conmutación. Virtual Cut-Through

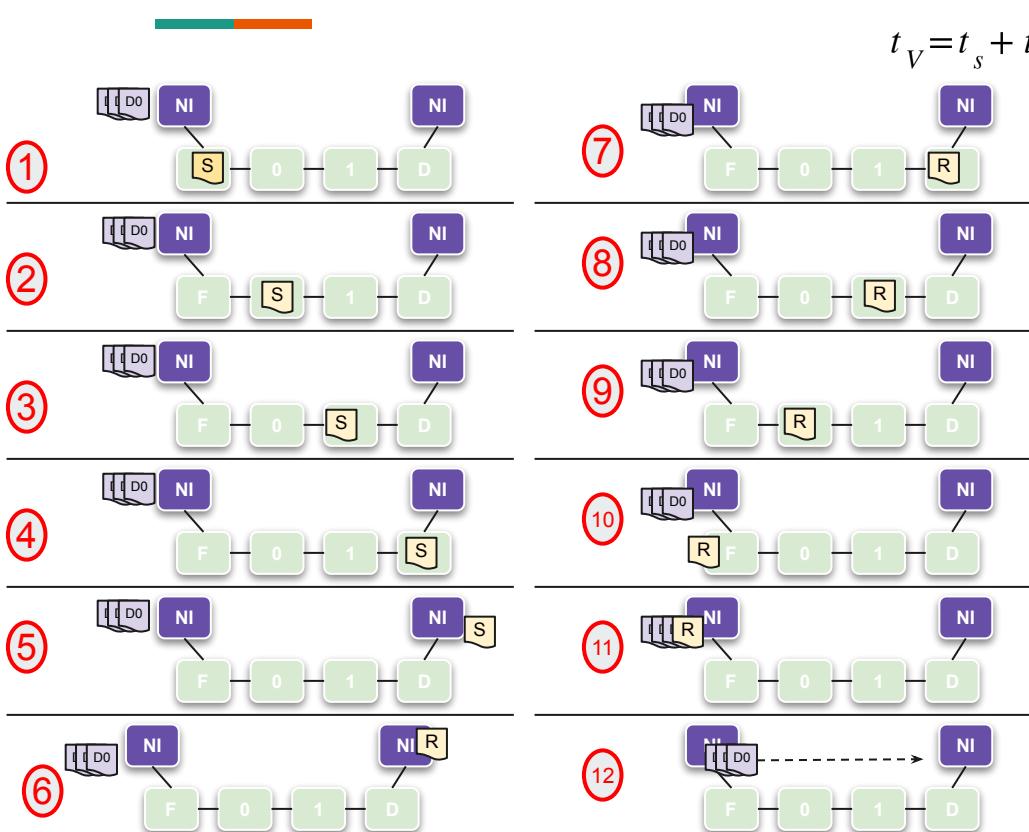
$$t_V = t_c + t_r = D \left[t_r + t_w \right] + t_w \left(\frac{L}{W} \right)$$



T	R	T
1		t_w
2	t_r	t_w
3	t_r	t_w
4	t_r	t_w
5	t_r	t_w
6		t_w
7		t_w
8		t_w

Se comporta igual que Vermiforme, pero en caso de bloqueo o parada, el paquete se agrupa, dejando los enlaces libres. La incidencia en el ancho de banda global es menor que en Vermiforme, y comparable a Almacenamiento y Reenvío.

6. Técnicas de conmutación. Com. de circuitos



$$t_V = t_s + t_r + t_d = D[t_r + t_w] + t_w + D \cdot t_w + t_w + B_{Canal} \left(\frac{L}{W} \right)$$

T	R	T
1		t_w
2	t_r	t_w
3	t_r	t_w
4	t_r	t_w
5	t_r	t_w
6		t_w
7		t_w
8		t_w
9		t_w
10		t_w
11		t_w
12	$B_{Canal} * 3$	

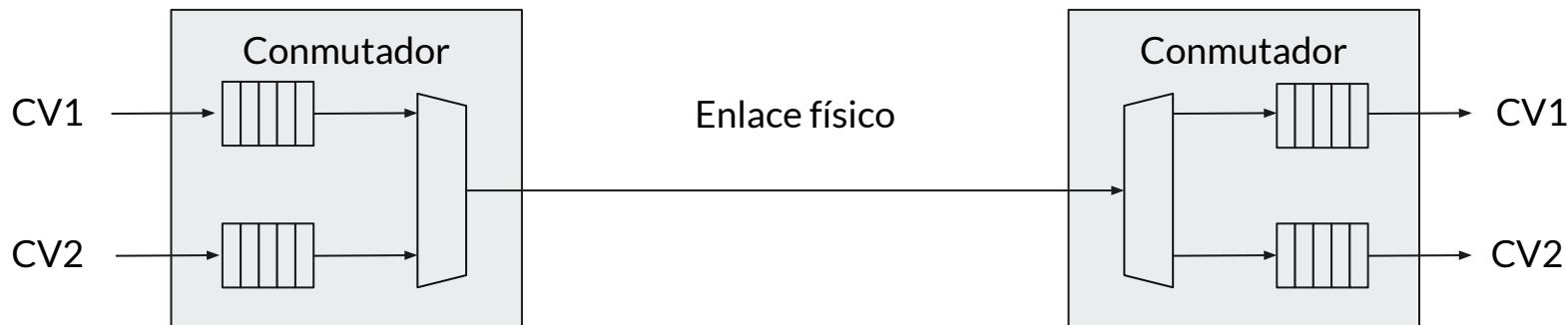
Se establece un **canal** y se usa todo el ancho de banda disponible.

El almacenamiento mínimo requerido es el **tamaño de la sonda**.

Su impacto en el ancho de banda global afecta a todos los canales reservados desde que se envía el reconocimiento hasta la llegada de todos los datos

6. Comutación: Canales virtuales

- Si se llena el buffer de un enlace de entrada, el canal está ocupado aunque no transfiera. Este problema se evita asociando varios **canales virtuales** a un enlace físico, pudiendo así **multiplexar en el tiempo el uso del canal**.
- Los canales comparten enlace a nivel de flit: Los flits de distintos paquetes se envían por el enlace mezclados.



6. Comutación: Canales virtuales

- ¿2 canales virtuales sobre uno físico? Velocidad = velocidad/2 ...
- En una red con conmutación vermiforme se reduce la probabilidad de:
 - Bloqueo de un paquete, aumentando el ancho de banda.
 - Paquetes pequeños esperen a paquetes grandes.
- Usar canales virtuales independiza la asignación de enlace de la asignación de almacenamiento.
- Si se bloquea un canal virtual, el enlace físico no se inutiliza, lo usan los otros. De esta forma, aumenta el tráfico que puede circular.

6. Comutación: Canales virtuales

- Usar canales virtuales necesita aumentar el hardware asociado a cada enlace, pues un canal virtual necesita:
 - Almacenamiento
 - Señales de control de flujo
 - Hardware de multiplexado y demultiplexado
 - Arbitraje virtuales - físico
- Los canales virtuales mejoran el ancho de banda y la latencia global, pero hasta un límite, pues se añade **retardo** (arbitraje, multiplexores...), que **depende del número de canales virtuales**.

Índice

- 1. Clasificación Sistemas de Comunicación**
- 2. Propiedades**
- 3. Diseñar una Red**
- 4. Prestaciones**
- 5. Enrutamiento**
- 6. Técnicas de commutación**
- 7. Ejemplos**

Ejemplos para lectura

- Louis Jenkins. [Networks for High-Performance Computing](#)
- Jesus Escudero-Sahuquillo & Co. [Efficient and Cost-Effective Hybrid Congestion Control for HPC Interconnection Networks](#)
- Francisco J. Andújar-Muñoz &Co. [N-Dimensional Twin Torus Topology](#)
- Benito M. & Co. [Analysis and improvement of Valiant routing in low-diameter networks.](#)

—
Gracias.