Sistemas Multiprocesadores

Arquitectura y Tecnología de Computadores
Universidad de Sevilla
Prof. Saturnino Vicente Díaz

Sistemas Multiprocesadores

1. Introducción
2. Clasificación de los multiprocesadores
3. Redes básicas de interconexión
4. Clusters
5. Programación y aplicaciones

1. Introducción

- Niveles de paralelismo
- Introducción a los sistemas multiprocesadores
- Argumentos a favor de los multiprocesadores
- Argumentos en contra de los multiprocesadores
- Investigación actual

Introducción

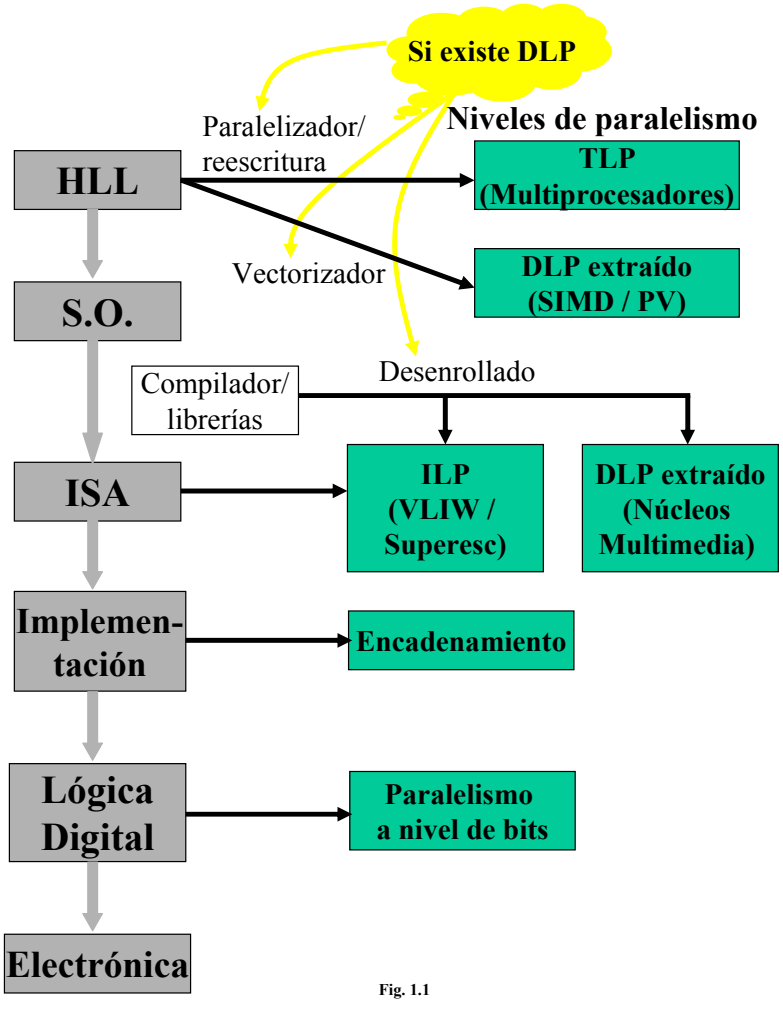


Fig. 1.1

Introducción

- Sistemas multiprocesadores
  - En los sistemas actuales la CPU
    - es un todo monolítico
    - está separada de la memoria principal y de la E/S
    - Alcanza grandes velocidades
  - Vamos a romper con esta concepción para diseñar los multiprocesadores

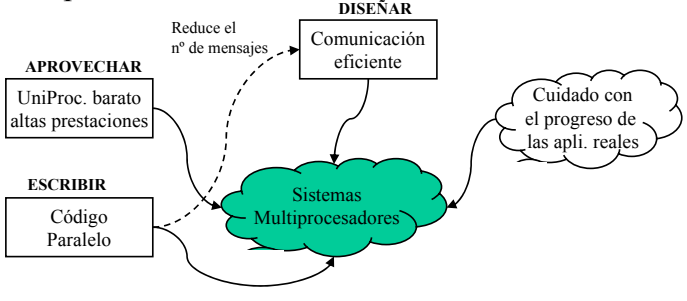


Fig. 1.2

  - Lo que vamos a construir se parece a:

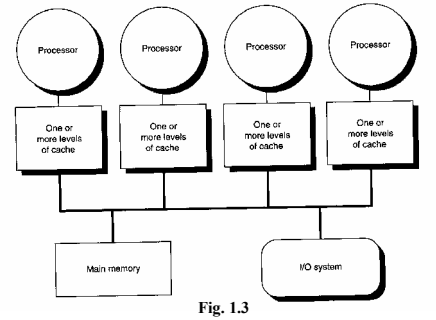


Fig. 1.3

Introducción

  - Idealmente, $t_{ejec,mult} = \frac{t_{ejec,uni}}{N}$, lo cual es imposible
    - La red de interconexión juega un papel importante
      - Comunicación entre procesadores
      - Comunicación entre procesadores y memoria
    - La memoria está separada de los procesadores, pero no totalmente ⇒ Cada procesador tiene su caché local. La forma de organizar la jerarquía de memoria es vital

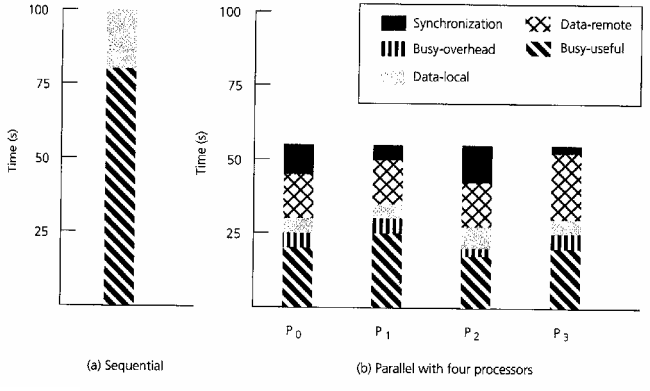


Fig. 1.4 Components of execution time from the perspective of an individual processor

Introducción

- Argumentos a favor de los Multiprocesadores
  - En 15 ó 20 años, el rendimiento de los Uniprocesadores llegará a su fin, debido a dos razones:
    - Incrementar el rendimiento conectando uniprocesadores en paralelo
      - Menos costoso que diseñar un nuevo procesador
      - Más flexible para todo tipo de soft que un nuevo diseño
    - El ritmo de innovación en la endoarquitectura es difícil de mantener indefinidamente
      - Actualmente el incremento en transistores es superior al de rendimiento (nºtran x 4 ; rend. x 3 cada 3 años)
      - Algunos estudios indican el inicio de la saturación en prestaciones de los uniprocesadores

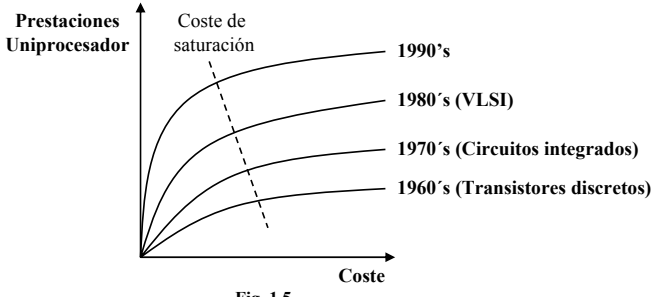


Fig. 1.5

  - Parece que los MPr's son la única forma de aumentar considerablemente el rendimiento a un precio razonable

Introducción

- Argumentos en contra de los Multiprocesadores
  - Cada 2 ó 3 años, alguien anuncia el fin de los UniPr
    - A los 2 ó 3 años se disparan las prestaciones por una nueva idea arquitectónica. Esto es debido a que hay mucho mercado ⇒ mucha inversión ⇒ mucho avance
      - Segmentación
      - Vectorización
      - Superescalaridad
      - HyperThreading, etc.
  - Lentitud en las comunicaciones
    - Latencia a memoria local es del orden de pocos *ns*
    - En MPr, si un Pr necesita un dato remoto ⇒alta latencia, del orden de microsegundos (Ejemplo)
  - Limitación del paralelismo de las aplicaciones
    - La ley de Amdahl nos indica el efecto del paralelismo de las aplicaciones en la aceleración
      - Intuitivamente $t_{ejec,mult} = t_{ejec,uni} / N$
      - Sin embargo, la mejora solo afecta al porcentaje de código que se pueda paralelizar
        - » Dado un código normal, un compilador intenta paralelizar el código, y lo consigue en la mitad del programa

$$A = \frac{1}{(1-F)+\frac{F}{N}} = \frac{1}{\frac{1}{2}+\frac{1}{2*N}} = \frac{2N}{N+1} \to 2$$

        - » Suponer que queremos conseguir una aceleración de 80, con 100 procesadores ¿Qué fracción del programa debe ser paralela?

$$F \approx 99{,}75$$

Introducción

- Investigación actual
  - Software más adaptado a la computación paralela
    - Nuevos algoritmos
      - Se obtendrían mayor rendimiento
    - Nuevos lenguajes paralelos
    - Que los compiladores extraigan el paralelismo implícito en los programas no escritos explícitamente en paralelo
    - Portabilidad de los programas entre distintas plataformas
  - Reducir/ocultar la alta latencia remota
    - Mediante hardware (p.e. cachés de datos compartidos)
    - Mediante software, (p.e. reestructurando los accesos a datos para que sean más locales, ocultar los accesos remotos mediante ejecución de otras partes del código, etc.)
  - Flexibilidad en los multiprocesadores
    - Distintas aplicaciones pueden requerir distintas topología para extraer más paralelismo. Los MPr deberían poder cambiar su estructura (topología, distribución de la memoria, etc) de forma dinámica y fácil, para así, adaptarse a las distintas aplicaciones

2. Clasificación de los multiprocesadores

- Clasificación en función de la organización de la memoria
  - Uniform Memory Access (UMA)
  - NonUuniform Memory Access (NUMA)
  - Message Passing Machines (MPM)
- Emulación de Multiprocesadores
- Mercado actual de los multiprocesadores

Clasificación en función de la organización de la memoria

- La memoria es determinante en un sistema con tanto ancho de banda de memoria (AB_{mem})
  - El AB_{mem} de 1 sólo procesador es muy grande
  - La memoria puede servir para comunicarse entre los procesadores
- UMA (Uniform Memory Access)

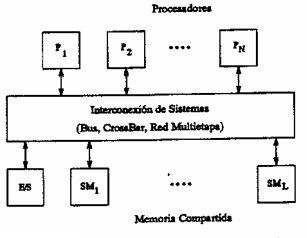


Fig. 2.2. Modelo de multiprocesador UMA

  - Espacio de direcciones de memoria compartido
  - Tiempo de acceso uniforme para toda dirección
  - Hardware bastante simple.
  - El cuello de botella es el acceso a memoria principal, lo cual implica:
    - Grandes cachés en cada procesador
    - El número de procesadores no puede ser muy alto. Hoy en día N≤32.

Clasificación en función de la organización de la memoria

  - Multiprocesadores muy populares hoy en día:
    - Todos los SMP's (Multiprocesadores simétricos con memoria compartida basado en bus común)
    - bi, quad-Pentium
    - Sun Enterprise 10000 (crossbar)

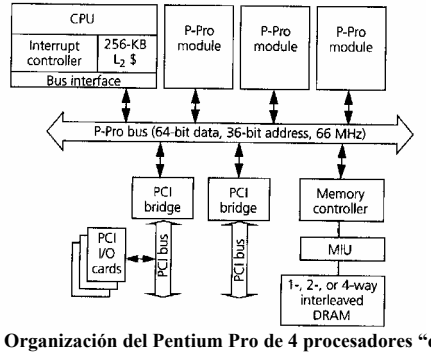


Fig.2.3. Organización del Pentium Pro de 4 procesadores "quad pack"

  - Programación y comunicación
    - Al tener espacio de direcciones de memoria compartida ⇒ Se adapta perfectamente a la programación multihilo (multi-thread)
    - También se puede usar para programación multiprocesos, pero sería más lento. La comunicación entre procesos sería mediante intercambio de variables en la memoria compartida
    - Surge el problema de la coherencia entre cachés ⇒ disponer de Hard adicional para solucionarlo

Clasificación en función de la organización de la memoria

- Problema de coherencia en UMA
  - El overhead por acceder a la memoria compartida es muy grande. Hay que usar cachés
  - Debemos asegurar la coherencia de los datos en dichos dispositivos
  - Cuando usamos un dato privado ⇒ traerlo a caché
  - Si accedemos a un dato compartido, hay que traerlo a caché, pero si otros procesadores también acceden al dato y se lo llevan a su caché, existirán varias copias distintas del mismo dato. Si algún procesador modifica el dato ⇒ ***Problema de coherencia***
- Solución
  - Intentar evitar las incoherencias mediante hard
  - En UMA se usan protocolos de husmeo (snooping)
    - Se pueden utilizar gracias a tener un bus único y compartido de comunicación
    - Los controladores de cachés locales espían el bus para ver si circula algún dato de alguna línea que tenga almacenada en su caché
    - Los cachés deben de estar provistos de 2 puertos de acceso (p.e. replicando la memoria de etiquetas)
      - » Uno para el acceso del procesador
      - » Otro para que el controlador pueda comparar las direcciones que aparecen por el bus
  - Ejemplo: Protocolo de invalidación CB (MSI)

Clasificación en función de la organización de la memoria

- NUMA (NonUniform Memory Access)

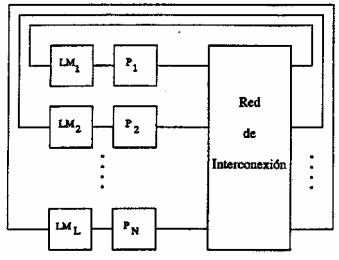


Fig.2.4. Modelo de multiprocesador NUMA con memorias locales compartidas

  - Espacio de direcciones de memoria compartido
  - Tiempo de acceso **no** uniforme para toda dirección. Depende de si accedemos a memoria local (no a caché) o a remota.
  - Debido al incremento de prestaciones del uniprocesador y del AB requerido, ésta es la única solución para sistemas con muchos procesadores
  - t_{acc} pequeño en accesos a memoria local
  - Hard de la red es complejo, necesita redes de interconexión especiales. Existen distintas líneas de comunicación separadas ⇒ mayor AB
  - El problema de la coherencia es más difícil de mantener en hard

Clasificación en función de la organización de la memoria

- Estos multiprocesadores son escalables
  - Escalable: Si crece el número de procesadores N, entonces la memoria crece como orden(N). Esto es gracias a que cada procesador lleva su propia memoria
  - Los procesadores UMA no son escalables, puesto que si aumentamos el número de procesadores, la memoria es la misma.
- Programación y comunicación
  - Comunicación entre procesadores es mediante variables globales
  - Al tener espacio de direcciones de memoria compartida ⇒ Se adapta perfectamente a la programación multihilo (multi-thread)
  - Debe existir cacheo de datos remotos en hard, aunque la coherencia sea compleja
  - Debe existir soporte hard para el acceso remoto
    - Ld local, provoca acceso a caché o memoria local
    - Ld remoto, provoca interrupción I/O a la red
  - Si existe coherencia, el código NUMA es totalmente compatible con UMA. Esto supone una ventaja, puesto que los UMA están muy extendidos y hay muchas aplicaciones desarrolladas. El programador debe tener cuidado con los accesos remotos
- Ejemplos: TC2000, KSR-1, Standford Dash, BBN Butterfly, Cray T3D

Clasificación en función de la organización de la memoria

- Solución al problema de coherencia en NUMA
  - La solución hard es más compleja que en UMA
  - Han existido máquinas con control de coherencia por software (máquinas NC, NonCoherent). CRAY T3D
    - Los datos compartidos se declaran no cacheables
    - Si el compilador/programador detecta que un dato compartido será leído muchas veces, y no escrito, realiza un cacheo por software
      - » Copia el dato en una variable local privada
      - » Trabaja con esta copia hasta que por software sepa que la copia es mala (otro procesador la modifica)
  - Desventajas de las máquinas NC
    - Detectar la posibilidad de cacheo (compilador)
      - Posible en bucles "for" simples
        - » Imposible en accesos a memoria dinámica
        - » Sólo se sabe la dirección en tiempo de ejecución
        - » El compilador no se puede arriesgar
        - » Los declara No Cacheable ⇒ Perder prestaciones
      - Se pierde tiempo en hacer la copia de la variable
    - Cuando se cachean variables remotas de una sola palabra se pierde la ventaja de la localidad de datos
    - Estos dos problemas se acentúan debido a la alta latencia de la red
      - CRAY T3D
        - » tiempo de acceso a caché = 2 ciclos
        - » tiempo de acceso a memoria remota = 150 ciclos
  - Solución Hard
    - Incluir algún medio compartido para el control de la coherencia
      - En los NUMA el nº de procesadores es muy alto
      - Si el nº de procesadores es alto ⇒ saturación del medio
    - Distribuir sistema de coherencia. **Directorio distribuido**

Clasificación en función de la organización de la memoria

- MPM (Message Passing Machines)

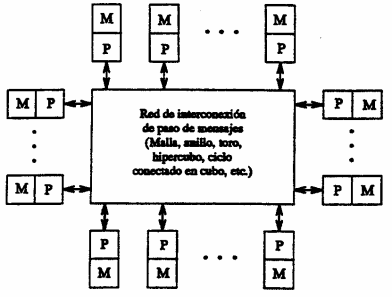


Fig.2.5. Modelo de multiprocesador MPM

- Espacio de direcciones de memoria distribuido
- Son compatibles en hard a los NUMA
- Cada procesador es un computador independiente del resto
- Fácilmente escalable
  - El mecanismo de comunicación es a través de mensajes (igual que la comunicación entre procesos del S.O.). No necesita mecanismos hard ni para controlar la coherencia de cachés (pues son espacios distintos de direcciones), ni para implementar los accesos remotos
  - La red es la parte más compleja de escalar

Clasificación en función de la organización de la memoria

- Programación y comunicación
  - Programación orientada a multiprocesos
  - Comunicación se realiza por mensajes. Existen librerías estándar que:
    - Realizan la mensajería
    - Distribuyen los procesos por la máquinas declaradas del multicomputador
  - Comunicación explícita, lo cual tiene que ser tenido en cuenta por los programadores. Esto obliga a estar pendiente de ella e intentar ocultar la latencia
  - El overhead de comunicación es muy alto, luego interesa enviar datos grandes, p.e. Páginas
- Ejemplos de multiprocesadores MPM
  - Todos los Clusters del mercado
  - Constelaciones
    - Es un cluster
    - Cada nodo es un UMA con uno número de procesadores >= 16
  - IBM SP-2 (CPU: RS6000)
  - Intel/Sandia Option Red (CPU: Pentium Pro)
  - Intel Paragon (CPU: i860)
  - CRAY T3E (CPU: DEC Alpha 21164)
  - CM-5, nCUBE 2, Comic Cube, etc.

Clasificación en función de la organización de la memoria

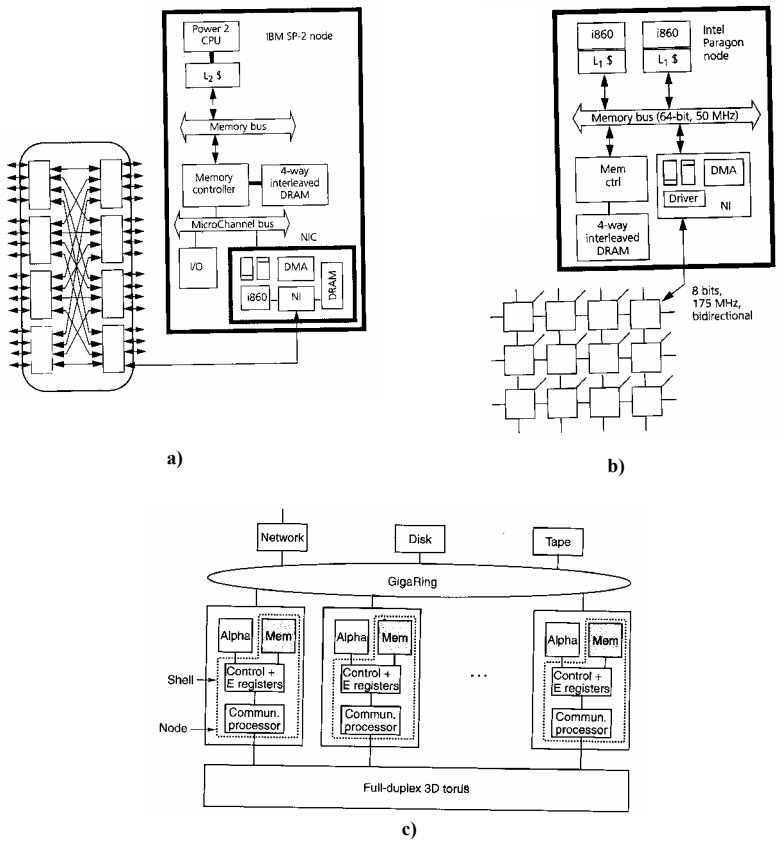


Fig.2.6. Modelos de multiprocesadors MPM. a) IBM SP-2, b) Intel Paragon, c) CRAY T3E

Clasificación en función de la organización de la memoria

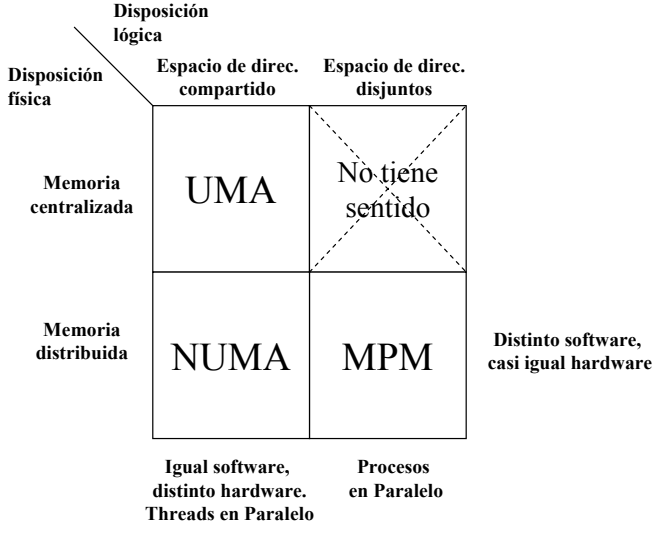


Fig.2.7. Resumen de la clasificación según la organización de la memoria

Emulación de Multiprocesadores

- Emulación de multiprocesadores
  - Se puede emular un MPM en un NUMA o viceversa. El hard es muy parecido
  - El MPM se emula fácilmente con un NUMA. La emulación de paso de mensajes es sencilla si se tiene memoria compartida (escribir/leer en una determinada dirección de memoria)
  - El NUMA es más difícil de emular con un MPM
    - Cualquier Ld/St requiere un overhead tremendo
      - Llamar al S.O. para traducir la dirección
      - Comprobar protecciones
      - Crear el mensaje
    - No tiene sentido enviar 1 solo byte o palabra, sino muchos datos de una sola vez, para ello podemos limitar la compartición de datos solo a páginas de memoria virtual ⇒ Tamaño de datos comunicados es muy grande y, por tanto, el overhead por byte es pequeño

Mercado actual

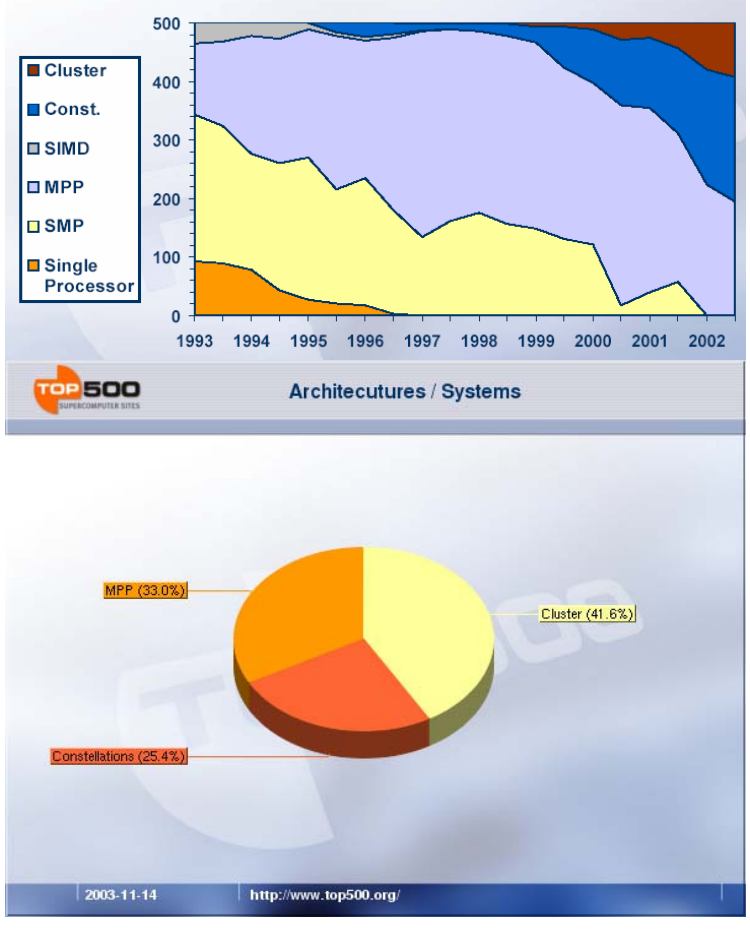
- Mercado actual de los multiprocesadores
  - Los UMA se han introducido con gran éxito en el mercado (memoria centralizada)
    - Fáciles de contruir
      - Basados en bus común
      - Hard especial para mantener la coherencia
    - Los microprocesadores comerciales estándar llevan soporte para UMA
    - Bien caracterizados y documentados
    - Se adaptan bien a sistemas multiusuario
    - Limitación: nº de procesadores que soporta
    - AB solicitado por cada procesador crece y el bus común tiende a saturarse cada vez más
    - Se adaptan bien a los cluster y son utilizados para construir las constelaciones.
  - Tendencias hacia memoria distribuida
    - Escalables
    - Los NUMA son compatibles en soft con los UMA, luego puede aprovechar el soft ya desarrollado
      - El único problema es que el hard para mantener la coherencia de los cachés es más complejo, pero en los nuevos procesadores (Alpha 21364) ya se incluye dicho hard

Mercado actual

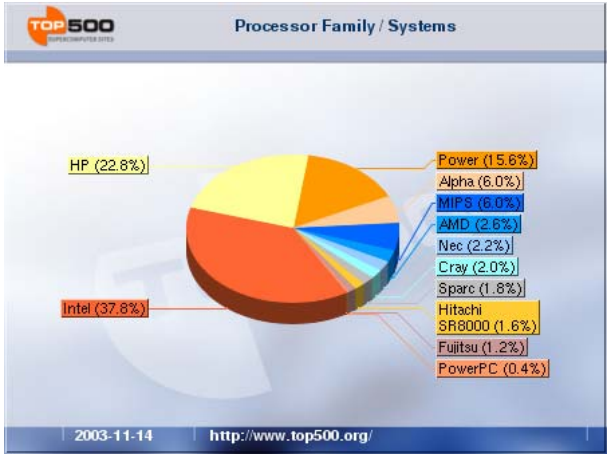
- Los supercomputadores se fabrican con cientos, o miles de microprocesadores comerciales (MPP, Massively Parallel Processors). Existen gran interes en ellos, pero:
  - El mercado no es muy grande comparado con UMA
  - Las herramientas de programación no están maduras: Lenguajes de programación, compiladores, etc
  - No está claro el mecanismo de comunicación, tipo NUMA (mem. Compartida), tipo MPM (mensajes), o híbridos (sólo acceso remoto compartido a una página entera)
- Los MPM se están expandiendo rápidamente, sólo necesitan:
  - Una librería (p.e. PVM, MPI)
  - Una red, que puede ser la Ethernet estándar

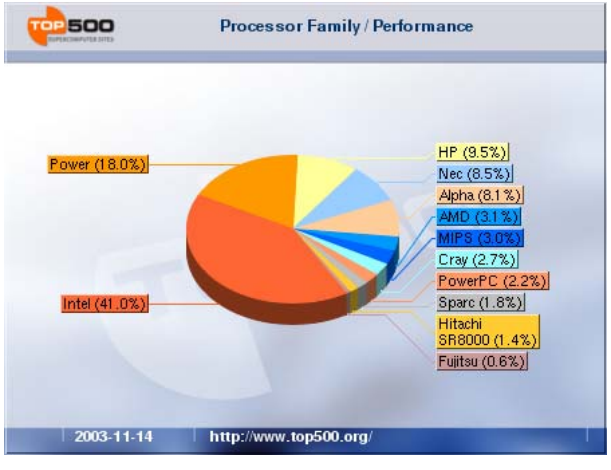
CLUSTERS

Mercado actual

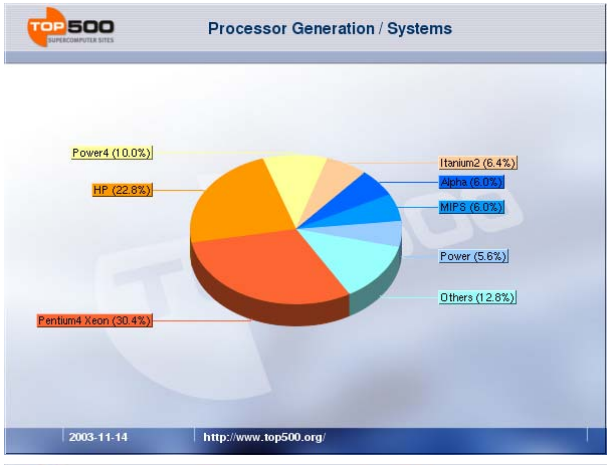


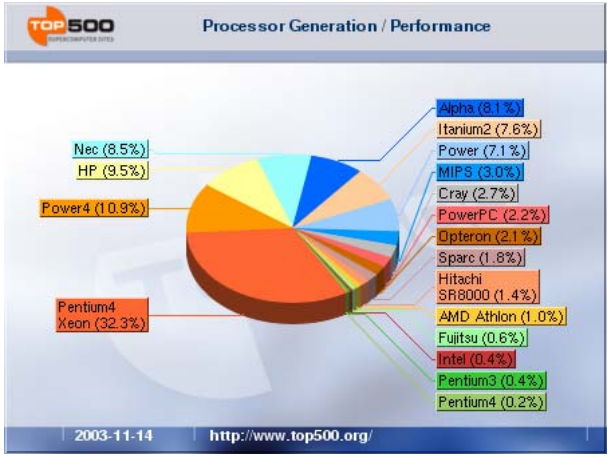
Mercado actual





Mercado actual



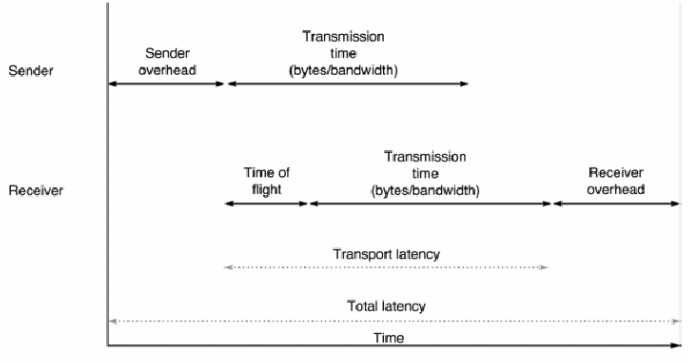


3. Redes básicas de interconexión

- Diseño de redes de interconexión
  - Consideraciones en el diseño de redes
  - Estructura y funcionamiento
  - Espacio de diseño
- Topología de redes
  - Redes estáticas (directas)
  - Redes dinámicas (indirectas)
- Mecanismos de rutado en redes estáticas

Diseño de redes de interconexión

- Prestaciones de la red
  - Múltiples procesadores → Múltiples procesos cuya sincronización y comunicación se realiza mediante el paso de mensajes y el acceso a variables compartidas a través de la red
  - Los accesos a la red dependen de:
    - Ancho de banda
    - Latencia
    - Sobrecarga del emisor y/o receptor



Diseño de redes de interconexión

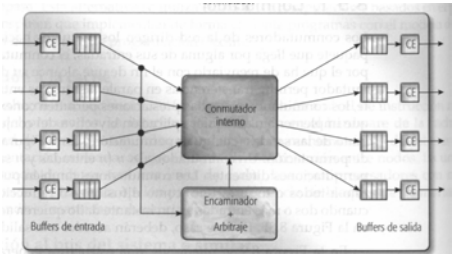
- Escalabilidad
  - Al añadir un nuevo procesador, los anchos de banda de la memoria, la E/S y la red se incrementan proporcionalmente
- Fácil expandibilidad
  - Permitir la adición de un pequeño número de nodos.
- Posibilidad de particionar
  - La red debe de poder ser partida en pequeños subsistemas funcionales.
    - Los trabajos de unos usuarios no deben afectar al resto
    - Razones de seguridad
- Simplicidad
  - A mayor simplicidad, normalmente, mayores frecuencias de reloj y mayores prestaciones. Además son más fáciles de comprender.
- Longitud de los enlaces
  - El tipo de multiprocesador impondrá la longitud y el tipo de material de los enlaces

Diseño de redes de interconexión

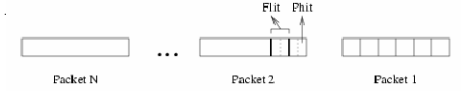
- Restricciones físicas
  - Disposición de los cables en un área limitada
  - Número de pines por chip (o placa) dedicados a canales de comunicación
  - Posibilidad de empaquetar los cables
- Fiabilidad y fácil reparación
  - Tolerancia a fallos
  - Diseño modular: Facilidades en la reparación y en la actualización
  - Reconfiguración de la red en caso de fallo o eliminación de algún nodo
- Cargas de trabajo esperadas
  - Si conocemos a priori el tipo de aplicaciones a ejecutar podremos optimizar la red en función de las condiciones del tráfico de esa aplicación.
    - Patrones de comunicación
    - Tamaño de los mensajes
    - Carga de la red, etc.
  - En caso contrario, tendremos que hacer una red robusta que tenga unas buenas prestaciones en un amplio rango de condiciones del tráfico
- Restricciones en el coste
  - Normalmente, la mejor red es también la más cara. Tenemos que encontrar un equilibrio entre coste y prestaciones.

Diseño de redes de interconexión

- Estructura y funcionamiento
  - Conmutador
    - Dirigen los paquetes hacia el nodo destino.
    - Generalmente, permiten transferencias en paralelo entre sus entradas y salidas.
    - Suelen tener algún medio para el almacenamiento de la información de tránsito. Este almacenamiento puede situarse tanto en las entradas como en las salidas.
    - Estructura
      - Conmutador interno: La mayor parte de las implementaciones de altas prestaciones utilizan un conmutador de barras cruzadas
      - Encaminador: Basándose en la información contenida en la cabecera del paquete entrante, decide por que salida ha de enviarse.
      - Arbitraje: Resuelve mediante una circuitería de control los conflictos que se producen entre paquetes entrantes que deben enviarse por la misma salida.



  - Enlaces y canales
    - Ancho de banda depende del ciclo de reloj de la red y del número de líneas del enlace.
    - Pueden ser unidireccionales o bidireccionales (full ó half-duplex)
    - **Phit** (PHysical unIT, unidad física)
      - Unidad de información que se transfiere por un enlace en un ciclo de red.
    - **Flit** (FLow control unIT, unidad de control de flujo)
      - Unidad mínima de información que puede atravesar un canal entre conmutadores, sujeta a sincronización.



Diseño de redes de interconexión

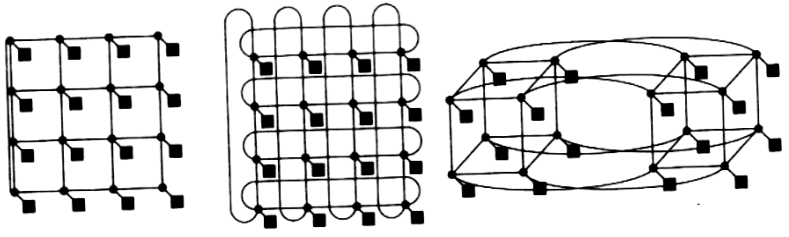
- Espacio de diseño
  - Topología
    - Estructura de interconexión física de la red
  - Algoritmo de encaminamiento
    - Determina el camino a seguir por un paquete desde el nodo fuente al nodo destino.
  - Estrategia de conmutación
    - Determina *cómo* los datos de un paquete atraviesan el camino hacia el destino.
  - Mecanismo de control de flujo
    - Determina *cuándo* las unidades de información se desplazan a lo largo del camino hacia el destino.

Topología de redes

- Redes estáticas (directas)
  - Cada conmutador está asociado a un nodo de cómputo.
  - La red queda conformada por conexiones directas punto a punto estáticas entre nodos.
  - (Def) *Distancia mínima* entre dos conmutadores en una red estática es el número mínimo de enlaces que los separan.
  - Podemos clasificar estas redes en dos grupos
    - Redes Estrictamente Ortogonales
      - Cada nodo tiene al menos un enlace de salida para cada dimensión.
      - Mallas: Intel Parangon, MIT J-Machine …
      - Toros: KSR, iWarp, Cray T3D/T3E, …
      - Hipercubos: nCube, Intel iPSC, …
    - Redes no ortogonales
      - Árbol
      - Estrella

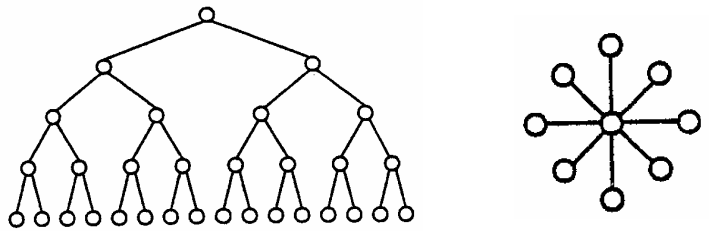
Topología de redes

Redes estrictamente ortogonales



Malla 2-D **Toro 2-D** **Hipercubo 4-D**

Redes no ortogonales



Árbol **Estrella**

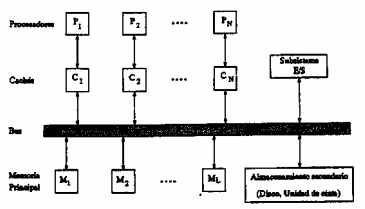
Topología de redes

- Redes dinámicas (indirectas)
  - Conmutadores
    - Están fuera de los nodos terminales (procesadores, memoria)
    - Pueden variar dinámicamente los nodos de entrada y salida que comunican.
    - Pueden estar conectados a varios nodos terminales o a otros conmutadores.
  - Podemos clasificar estas redes en tres grupos
    - Redes de medio compartido (buses)
      - Conmutador interno basado en un medio compartido por las entradas y salidas
      - Necesitamos una lógica de arbitraje.
      - Bus único. Ethernet, Token Bus, Token Ring, …
      - Bus Backplane. SGI POWERpath-2, Sun SDBus,…
    - Redes de barras cruzadas
      - Permite que cualquier fuente se conecte a cualquier destino libre.
      - Admite cualquier aplicación biyectiva entre entradas y salidas.
      - Cray X/Y-MP, Myrinet

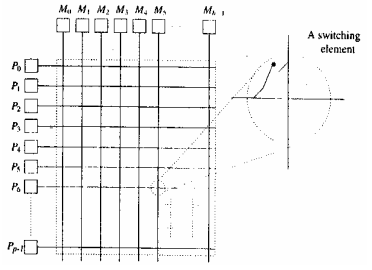
Topología de redes

- Redes multietapa
  - Múltiples conmutadores organizados en varias etapas.
  - Conexión entre fuente y destino se realiza a través de conmutadores de diferentes etapas
- Clasificación de redes multietapa
  - Redes bloqueantes.
    - » No es posible siempre establecer una nueva conexión entre un par fuente/destino libres, debido a conflictos con las conexiones activas.
    - » Red omega, red mariposa, red cubo multietapa, red baraje, red línea base,...
    - » NEC Cenju-3, IBM RP3, IBM SP, …
  - Redes no bloqueantes
    - » Cualquier fuente puede conectarse con cualquier destino libre sin afectar a las conexiones activas.
    - » Redes de Clos.
  - Redes reconfigurables
    - » Cualquier fuente puede conectarse con cualquier destino libre, pero puede ser necesario reconfigurar la red para mantener las conexiones activas.
    - » Red de Benes.

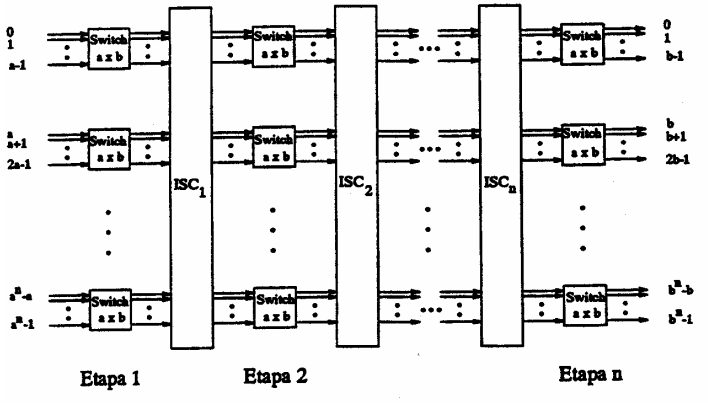
Topología de redes



Red bus común

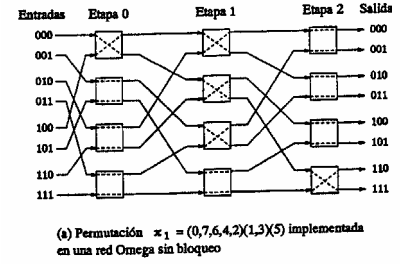


Red Barras cruzadas (Crossbar)

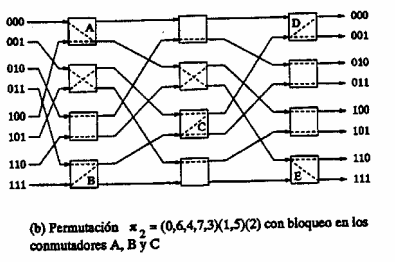


Redes Multietapa

Topología de redes

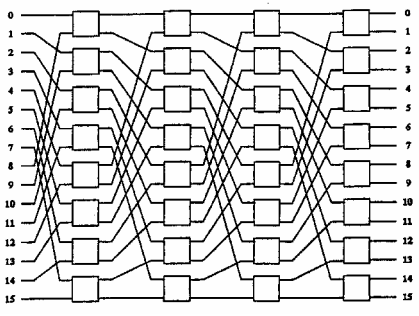


(a) Permutación $\pi_1 = (0,7,6,4,2)(1,3)(5)$ implementada en una red Omega sin bloqueo

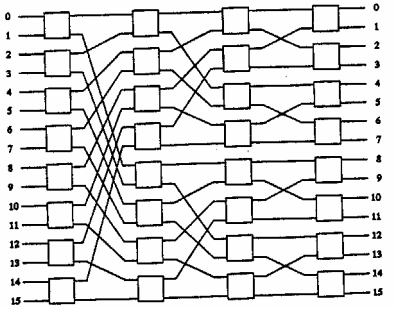


(b) Permutación $\pi_2 = (0,6,4,7,3)(1,5)(2)$ con bloqueo en los conmutadores A, B y C

Ejemplo de red omega (bloqueante)



Red Omega 16x16



Construcción recursiva de una red baseline

Mecanismos de rutado

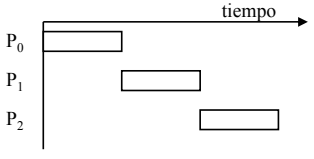
- Mecanismos de rutado en redes estáticas
  - Estos mecanismos determinan el camino que debe tomar un mensaje para atravesar la red y llegar a su destino
  - Toman como datos de entrada
    - Nodos fuente y destino
    - Información sobre el estado de la red
  - Generan una o más rutas
  - Los podemos clasificar en función del tipo de ruta que generan
    - En función de la longitud de la ruta
      - » Mínimos: Siempre selecciona una de las rutas más cortas. No tiene en cuenta congestiones de la red
      - » No mínimos: Pueden suministrar la ruta más larga pero para intentar evitar las congestiones de la red
    - En función del uso de la información sobre el estado de la red
      - » Deterministas: Determinan una única ruta, basándose sólo en la fuente y el destino (siempre va por el mismo camino). No tienen en cuenta la información sobre el estado de la red. Puede provocar un uso desigual de los recursos y provocar congestiones (los mínimos serían de este tipo)
      - » Adaptativos: Usan la información sobre el estado de la red para determinar la ruta del mensaje. Detectan congestiones e intentan evitarlas

Mecanismos de rutado

- Coste de la comunicación en redes estáticas
  - La latencia de comunicación (t_{com}) es uno de los parámetros más importantes a tener en cuenta.
    - Tiempo en preparar el mensaje para su transmisión
    - Tiempo que dicho mensaje tarda en atravesar la red
  - Usaremos los siguientes parámetros:
    - m : Tamaño, en palabras de red, del mensaje completo
    - l : Nº de enlaces que debe atravesar el mensaje
    - t_s (Tiempo de inicio): Tiempo que tarda el procesador emisor en preparar el mensaje
      - » Empaquetar el mensaje (añadir cabecera, información de corrección de errores, etc.)
      - » Cálculo de la ruta (si la red no sabe hacerlo)
      - » Tiempo en establecer una interfase entre el procesador local y el router.
    - t_h (Tiempo por salto): Tiempo tomado por la cabecera del mensaje en viajar entre dos procesadores conectados directamente en la red
      - » Tiempo que tarda el algoritmo de rutado en ejecutarse
      - » Tiempo en atravesar el conmutador
    - t_w (Tiempo de transferencia por palabra): Tiempo que tarda una palabra en atravesar el enlace.
      - » Si el ancho de banda del canal es r palabras/segundos, cada palabra tarda $t_w = 1/r$ segundos en atravesar el enlace
    - Podríamos tener en cuenta otro tiempo (t_r), que sería igual que (t_s), pero en el receptor. Nosotros vamos a obviarlo

Mecanismos de rutado

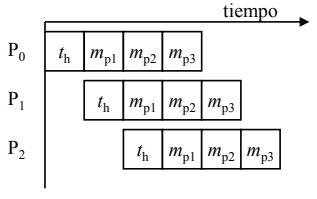
- Dos factores que influyen claramente en el t_{com}
  - Topología de la red
  - Técnicas de conmutación
- Vamos a estudiar dos técnicas de conmutación usadas frecuentemente en computadores paralelos. Las técnicas de rutado que utilizan dichas técnicas son llamadas:
  - Conmutación de paquete o reenvío (Store-and-forward)
  - Fragmentado o encadenado (Cut-trough)
- Conmutación de paquete o Reenvío (Store-and-forward)
  - Cada nodo almacena todo el mensaje entero antes de reenviarlo al siguiente nodo



  - $t_{com} = t_s + l(t_h + m * t_w)$
  - En caso de que exista congestión (bloqueo) hay que sumarle un tiempo de espera (t_e). La congestión se puede producir en el *spool* de FIFO del conmutador, o en espera para que se libere un nodo
- Fragmentado o encadenado (Cut-trough)
  - Encadena los envíos entre los distintos enlaces.
  - Por ejemplo, se divide el mensaje en paquetes de tamaño m_p, de forma que su transferencia sea igual a t_h, es decir, $m_{p*} t_w = t_h$

Mecanismos de rutado

  - Para este envío encadenado se suele hacer, por simplicidad, un túnel (wormhole) desde el origen al destino, que queda abierto durante toda la transmisión (lo cual puede provocar congestión)
    - » El primer paquete lleva información del destino y del número de paquetes que le siguen
    - » El conmutador queda "abierto" contando los paquetes hasta que pase el último
  - Con esto se simplifican los conmutadores, ya que no necesitan grandes buffers, ni inteligencia de rutado



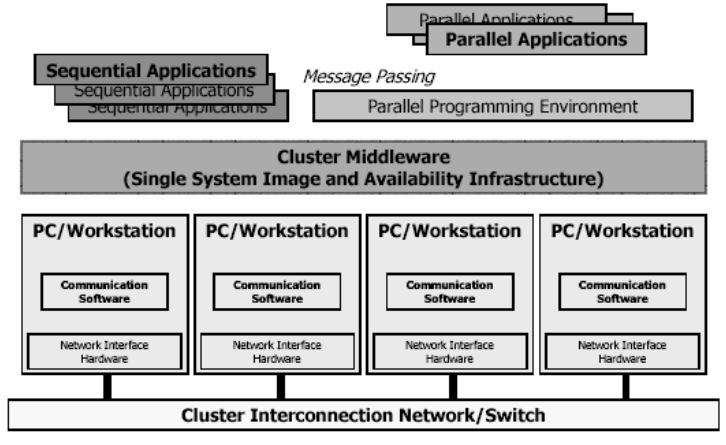
$$t_{com} = t_s + l * t_h + m * t_w$$
$$m * t_w = (m_{p1} + m_{p2} + m_{p3}) * t_w$$

4. Clusters

- Arquitectura de un cluster
  - Introducción
  - Componentes de un cluster
  - Ventajas e inconvenientes
- Tipos de clusters
  - Clusters de alto rendimiento
  - Clusters de balanceo de carga
  - Clusters de alta disponibilidad
- Redes para clusters
  - Redes Ethernet
  - Myrinet
  - InfiniBand
- Ejemplo de Cluster: Google

Arquitectura de Clusters

- Definición: Agrupación de computadores independientes e interconectados que colaboran en una tarea.
- El Cluster es un tipo más de arquitectura paralela distribuida (MPM), pero con una característica especial: cada computador puede utilizarse de forma independiente.



Arquitectura de un cluster

Arquitectura de Clusters

- Componentes de un cluster
  - Nodos de computación
    - Son múltiples y pueden encontrarse encapsulados en un único computador o estar físicamente separados.
    - Los nodos pueden ser PC's, workstation o SMP's
  - Middleware
    - SSI (Single System Image)
      - Ofrece al usuario una visión unificada de todos los recursos del sistema
      - Se define mediante hardware o software y puede implementarse a tres niveles:
        - » Nivel hardware: Se ve el cluster como un sistema de memoria compartida distribuida.
        - » Nivel kernel S.O.: Ofrece alto rendimiento a las aplicaciones secuenciales y paralelas. Por ejemplo, gang-scheduling para programas paralelos, identificación de recursos sin uso, acceso global a recursos, migración de procesos (balanceo de carga), etc.
        - » Nivel aplicación: Software de planificación y gestión de recursos, herramientas de gestión del sistema, sistemas de ficheros paralelos, etc.
    - Disponibilidad
      - Infraestructura de alta disponibilidad, que incluye servicios de checkpointing, recuperación tras fallo, tolerancia a fallos, etc.

Arquitectura de Clusters

- Red de interconexión
  - Suele ser una red de altas prestaciones: Myrinet, Infiniband, Gigabit Ethernet, etc.
- Sistema operativo
  - Pueden utilizarse la mayoría de los S.O. del mercado: UNIX, Linux, W2k, WXP, etc.
- Protocolos rápidos de comunicación
  - Active message, Fast messages, VIA, etc.
- Entornos y herramientas de programación paralela
  - Programación: MPI, PVM, OpenMP, DSM (Threadmarks, Linda), etc.
  - Depuradores paralelos: TotalView
  - Análisis de rendimiento: VT (IBM SP), MPE, Pablo, Vampir, etc.
  - Administración: Parmon.
- Aplicaciones
  - Aplicaciones paralelas o distribuidas, secuenciales.

Arquitectura de Clusters

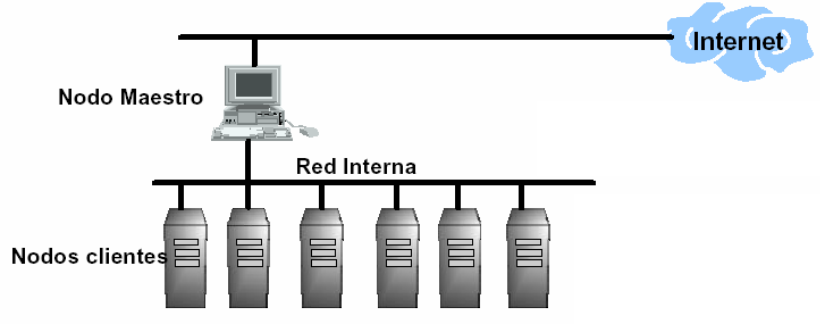
- Ventajas
  - Relación coste/prestaciones.
    - Uso de componentes comerciales (nodos y red)
    - Escaso coste de integración.
  - Flexibilidad
    - Hardware común: red, procesador, etc
    - Software de dominio público (Linux, PVM, MPI, etc)
  - Alta disponibilidad
    - El uso de hardware común con un coste ajustado, permite la réplica de componentes.
    - Existe redundancia natural, cada nodo posee sus propios componentes internos (bus, memoria, procesador, etc).
  - Escalabilidad
    - Permite agregar nuevos componentes para aumentar las prestaciones sin necesidad de eliminar elementos ya existentes
  - Incorporación de tecnología punta
  - Heterogeneidad
- Desventajas
  - Software
  - Problemas de administración y gestión
  - Memoria físicamente distribuida ➔ utilización menos eficiente
  - Varias copias del sistema operativo
  - La red es el cuello de botella del sistema

Tipos de Clusters

- Clasificaremos los clusters en tres tipos básicos:
  - Clusters de alto rendimiento
  - Clusters de balanceo de carga
  - Clusters de alta disponibilidad
- Existen clusters que combinan varios de los tipos anteriores.
- Clusters de alto rendimiento (HPC)
  - Utilizado en aplicaciones que realizan cálculos intensivos: Simulaciones científicas, modelos meteorológicos, renderización de gráficos, etc.
  - Pretenden sustituir a los grandes y costosos supercomputadores.
  - Ejemplo: Clusters Beowulf
    - Supercomputador virtual paralelo basado en una agrupación de computadores con hardware común, y con sistema operativo y aplicaciones libres.

Tipos de Clusters

- Utiliza una red privada de alta velocidad
- Los nodos están dedicados exclusivamente al cluster
- Existe un nodo maestro que se encarga de acceder, compilar y manejar las aplicaciones a ejecutar. Normalmente es el único con salida al exterior.
- Sistema operativo libre: GNU/Linux, FreeBSD, NetBSD
- Utiliza aplicaciones GNU: compiladores, herramientas de programación, y librerías de comunicaciones estándar (MPI, PVM, OpenMOSIX, etc.)



Tipos de Clusters

- Librerías de comunicaciones estándar en clusters HPC
  - MPI (Message Passing Interface)
    - Librería para paso de mensaje, propuesto como estándar por un comité de vendedores, implementadores y usuarios.
    - Colección de funciones que oculten detalles de bajo nivel, tanto software como hardware
    - Diseñado para obtener un alto rendimiento tanto en máquinas paralelas como en clusters
    - La unidad básica de paralelismo son los procesos independientes
    - Tienen espacios de memoria independientes
    - Intercambio de datos y sincronización mediante paso de mensajes
    - A cada proceso se le asigna un identificador interno propio
    - Proporciona una funcionalidad flexible, incluyendo diferentes formas de comunicación, rutinas especiales para comunicaciones "colectivas" y la habilidad de usar tipos de datos y topologías definidas por el usuario dentro de las comunicaciones.
    - Programación en Fortran y C. En la revisión del estándar (MPI-2) se soporta C++ y Fortran 90

Tipos de Clusters

– Ejemplo de programa escrito en C, usando MPI, y su salida por pantalla cuando se ejecuta en un cluster de 10 nodos.

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char *argv[])
{
        int err, mi_rango;
        err = MPI_Init (&argc, &argv);
        err = MPI_Comm_rank (MPI_COMM_WORLD, &mi_rango);
        printf ("Hola mundo!, desde el nodo %d\n", mi_rango);
        err = MPI_Finalize();
}

$ mpirun –np 10 ./ejemplo_mpi
Hola mundo!, desde el nodo 0
Hola mundo!, desde el nodo 3
Hola mundo!, desde el nodo 8
Hola mundo!, desde el nodo 4
Hola mundo!, desde el nodo 2
Hola mundo!, desde el nodo 5
Hola mundo!, desde el nodo 7
Hola mundo!, desde el nodo 6
Hola mundo!, desde el nodo 9
Hola mundo!, desde el nodo 1
```

Tipos de Clusters

- PVM (Paralell Virtual Machine)
  - Objetivo: Coordinar una red de ordenadores para que se puedan usar cooperativamente en la resolución de problemas de cálculo computacional paralelo
  - Existe un conjunto de nodos (hosts) susceptibles de poder ser usados en tareas de computación, definido por el usuario.
  - El usuario selecciona el conjunto de máquinas donde se ejecutarán la tarea de computación paralela. Dicho conjunto puede alterarse en tiempo de ejecución, añadiendo o quitando máquinas, esto es importante para la tolerancia a fallos
  - Acceso semitransparente al hardware: Podemos aprovechar las características de cada nodo para ejecutar determinados cálculos.
  - Computación basada en el proceso.
  - Modelo de paso de mensajes explícito.
  - Soporte para arquitecturas heterogéneas, en términos de máquinas, redes y aplicaciones.
  - Soporta los lenguajes C, C++ y Fortran.

Tipos de Clusters

```
Labatc73:~$ ./hola
Yo soy la tarea iniciadora t40015
Mensaje desde t40016: Hola, mundo desde labatc74.cluster-atc.net
```

```
// Tarea iniciadora "hola.c" en PVM
#include "pvm3.h"
void main (void)
{
  int cc, tid, msgtag;
  char buf[100];

  printf("Yo soy la tarea iniciadora t%x\n", pvm_mytid());
  cc = pvm_spawn ("hola_otro", (char**)0, 0, "", 1, &tid);
  if (cc == 1)
  {
    msgtag = 1;
    pvm_recv(tid, msgtag);
    pvm_upkstr(buf);
    printf("Mensaje desde t%x: %s\n", tid, buf);
  }
  else
    printf("No puedo ejecutar hola_otro\n");
  pvm_exit();
}
```

```
// Tarea esclava "hola_otro.c" en PVM
#include "pvm3.h"
#include <string.h>
void main (void)
{
  int ptid, msgtag;
  char buf[100];

  ptid = pvm_parent();
  strcpy (buf, "Hola, mundo desde ");
  gethostname(buf + strlen(buf), 64);
  msgtag = 1;
  pvm_initsend (PvmDataDefault);
  pvm_pkstr (buf);
  pvm_send (ptid, msgtag);
  pvm_exit(),
}
```

Tipos de Clusters

- MOSIX
  - Sistema de administración que hace que un cluster de servidores y estaciones de trabajo Linux x86 funciona casi como un sistema SMP.
  - Simplicidad de uso y programación.
  - Prestaciones casi óptimas. Por ejemplo, cuando se crea un proceso, MOSIX lo asignará al nodo menos ocupado, para aumentar las prestaciones.
  - El kernel de MOSIX consiste en algoritmos de administración adaptativos que monitorizan y automáticamente responden a los requerimientos de recursos de todos los procesos frente a los recursos disponible en todo el cluster.
  - Se presenta como un parche al núcleo de Linux, más unas cuantas utilidades de usuario para gestionar el cluster.
  - Dispone de un sistema de ficheros distribuidos: MFS (MOSIX File System). Esto permite el uso de ficheros compartidos por todos los nodos del cluster.
  - Permite trasladar todo el conjunto de trabajo de un proceso (código y datos) desde un nodo del cluster a otro. En MOSIX, *fork()* puede reasignar el proceso recién creado a otra CPU en otro nodo dentro del cluster.
  - No establece ningún mecanismo propio de mensajería entro procesos. Puede trabajar con cualquiera de las librerías de paso de mensajes (PVM, MPI). Además puede usar los mecanismos de comunicación estándares (tuberías, tuberías con nombre, sockets UNIX y sockets TCP/IP) para compartir datos.
  - La gestión del cluster se realiza leyendo y escribiendo en archivos especiales usando la interfaz */proc* del sistema.

Tipos de Clusters

- Clusters de alta disponibilidad (HAC)
  - Intenta mantener en todo momento la prestación del servicio, encubriendo los fallos que se puedan producir.
  - Requisitos de un HAC
    - Fiabilidad: Tiempo durante el cual un sistema puede operar sin pararse (MTTF)
    - Disponibilidad: Porcentaje del tiempo en el cual el sistema está disponible para el usuario.
    - Facilidad de Mantenimiento: Indica la facilidad de mantener el sistema en condiciones de operación (reparaciones, actualizaciones, etc) tanto a nivel hardware como software. El tiempo medio durante el cual un sistema está averiado es MTTR.

    Disponibilidad = MTTF / (MTTF + MTTR)

  - Tolerancia a fallos → Redundancia (menor MTTR)
    - Redundancia en el hardware: Replicación de componentes
    - Redundancia en el software: Administración del hardware redundante para asegurar el correcto funcionamiento en caso de la caída de algún elemento
    - Redundancia temporal: Repetición de la ejecución de un conjunto de instrucciones para asegurar el funcionamiento correcto en caso de que ocurra un fallo

Tipos de Clusters

- Posibles configuraciones
  - Activo-Pasivo: Las aplicaciones se ejecutan sobre un conjunto de nodos (activos), mientras que los restantes actúan como backups redundantes de los servicios ofrecidos.
  - Activo-Activo: Todos los nodos actúan como servidores activos de una o más aplicaciones y potencialmente como backups para las aplicaciones que se ejecutan en otros nodos.
  - En cualquier caso, el fallo de un nodo, provoca la migración de las aplicaciones que ejecutaba, a otro nodo del sistema. Si la migración es automática se denomina *failover*, si es manual *switchover*.
- Servicios típicos en los HAC
  - Bases de datos, sistemas de ficheros, servidores de correo y servidores Web.
  - En general, tareas críticas que no puedan interrumpir el servicio

Tipos de Clusters

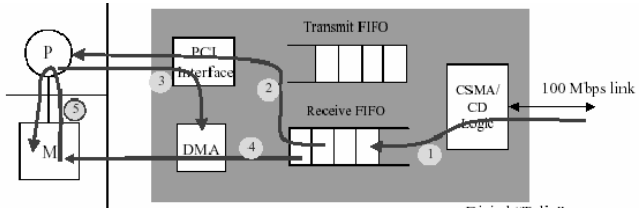
- Clusters de balanceo de carga (LBC)
  - Últimamente, ha crecido mucho la carga de trabajo en los servidores de servicios, especialmente en los servidores Web.
    - Utilización de máquinas de altas prestaciones
      - A largo plazo, el crecimiento de la carga de trabajo dejará la máquina obsoleta.
    - Utilización de tecnología clustering
      - La solución al problema del crecimiento de la carga de trabajo, pasaría por añadir más servidores
  - Balanceo de carga por DNS. Se asigna un mismo nombre a distintas direcciones IP y se realiza un round-robin a nivel de DNS entre ellas.
    - En el caso ideal la carga se repartirá equitativamente
    - Problemas:
      - Los clientes "cachean" los datos del DNS.
      - Las cargas de trabajo pueden ser muy distintas
  - Una solución mejor es utilizar un balanceador de carga para distribuir ésta entre los servidores. En este caso el balanceo queda a nivel de conexión. El balanceo de carga se puede hacer a dos niveles:
    - A nivel de aplicación
    - A nivel IP

Tipos de Clusters

- Linux Virtual Server (LVS)
  - Balanceo a nivel IP.
  - Parches y aplicaciones de mantenimiento y gestión, a integrar sobre GNU/Linux.
  - Permite la construcción de clusters de alta disponibilidad y de balanceo de carga.
  - Todo el sistema aparece como un único servidor virtual.
  - Existe un nodo director que controla el resto de nodos del sistema. En este nodo director corre Linux, parcheado para incluir el código ipvs.
  - El nodo director se encarga de dirigir la carga de trabajo, en función de un conjunto de reglas, al resto de nodos del sistema.
  - Cuando un cliente solicita un servicio al servidor virtual
    - El director escoge un servidor real para que se lo ofrezca.
    - El director debe mantener la comunicación entre el cliente y el servidor real. Esta asociación, durará lo que dure la conexión tcp establecida.
    - Cuando se solicite una nueva conexión, el director escogerá de nuevo un servidor real que puede ser distinto al anterior.
  - Los servidores reales pueden ser extraídos del LVS para su actualización o reparación, y devueltos al cluster sin interrumpir ningún servicio.
  - El sistema es fácilmente escalable.

Redes para Clusters

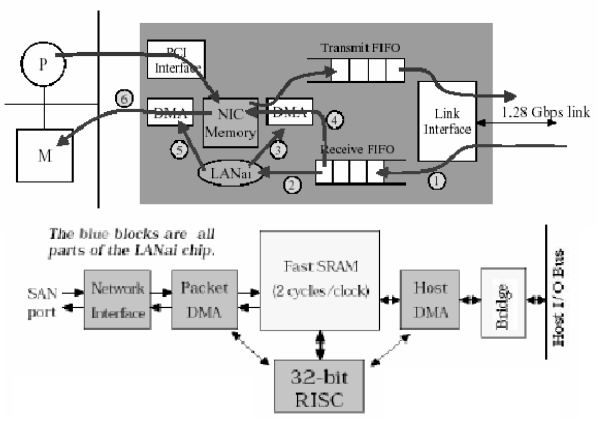
- Redes Ethernet
  - Ethernet
    - LAN introducida en 1982 y más utilizada en los años 90
    - Ancho de banda a 10 Mbits/seg
    - No muy adecuada para clusters debido a su bajo ancho de banda
    - Basada en el concepto de dominios de colisión
  - Fast Ethernet
    - LAN introducida en 1994 y más utilizada actualmente
    - Ancho de banda a 100 Mbits/seg
    - Mediante el uso de conmutadores y hubs se pueden definir varios dominios de colisión separados



    - El S.O. interviene en la introducción y extracción de los mensajes, vía interrupciones
    - La latencia aplicación-aplicación depende del driver y del API
      - TCP/IP aprox. 150µs
      - U-Net, MVIA aprox. 50 µs
  - Gigabit Ethernet
    - LAN introducida en 1998
    - Ancho de banda a 1 Gbits/seg
    - Basado en conmutadores punto-a-punto rápidos

Redes para Clusters

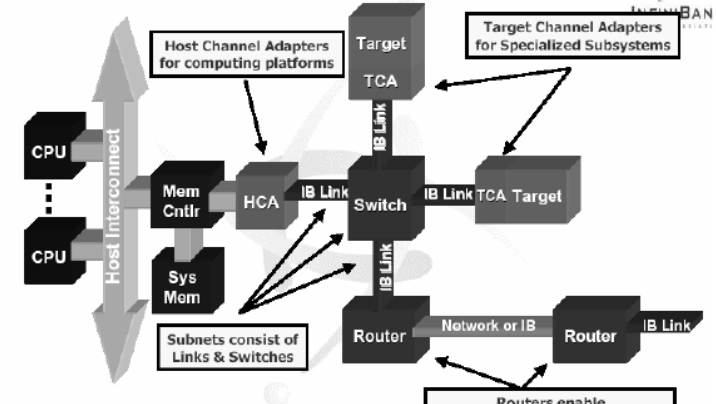
- Myrinet
  - Red diseñada por Myricom
  - Válida para LAN y SAN (System-Area Network)
  - Ancho de banda de 1,28 Gbits/seg, con conmutación wormhole



  - La responsabilidad se reparte entre el procesador del nodo y LANai
    - LANai es un dispositivo de comunicación programable que ofrece un interfaz a Myrinet.
    - LANai se encarga de las interrupciones originadas por los mensajes
      - DMA directa entre cola FIFO y memoria NIC, y entre memoria NIC y memoria del nodo
    - La memoria NIC puede asignarse al espacio lógico de los procesos
    - La latencia aplicación-aplicación es de aprox. 9µs.

Redes para Clusters

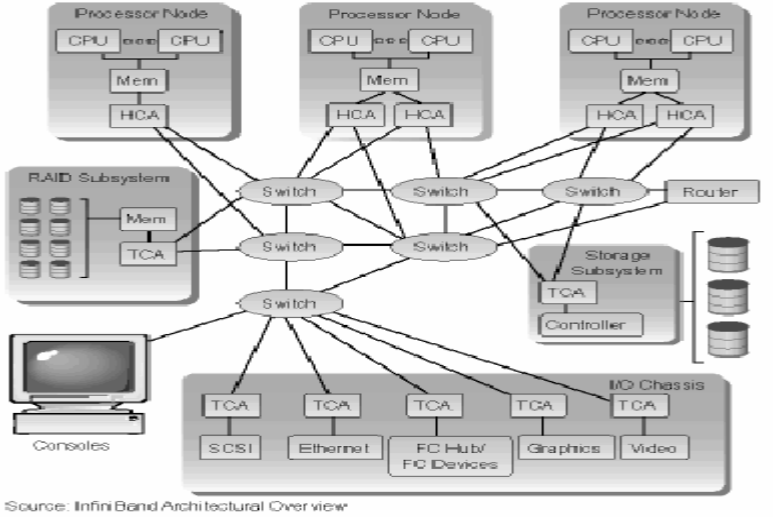
- InfiniBand
  - Nuevo estándar que define un nuevo sistema de interconexión a alta velocidad punto a punto basado en switches.
  - Diseñado para conectar los nodos de procesado y los dispositivos de E/S, para formar una red de área de sistema (SAN)
  - Rompe con el modelo de E/S basada en transacciones locales a través de buses, y apuesta por un modelo basado en el paso remoto de mensajes a través de canales.
  - Arquitectura independiente del sistema operativo y del procesador del equipo.
  - Soportada por un consorcio de las empresas más importantes en el campo: IBM, Sun, HP-Compaq, Intel, Microsoft, Dell, etc.



Arquitectura de InfiniBand

Redes para Clusters

- Características de la red Infiniband
  - No hay bus E/S
  - Todos los sistemas se interconectan mediante adaptadores HCA o TCA
  - La red permite múltiples transferencias de datos paquetizados
  - Permite RDMA (Remote Memory Access Read or Write)
  - Implica modificaciones en el software del sistema



Source: InfiniBand Architectural Overview

Ejemplo de Cluster: Google (i)

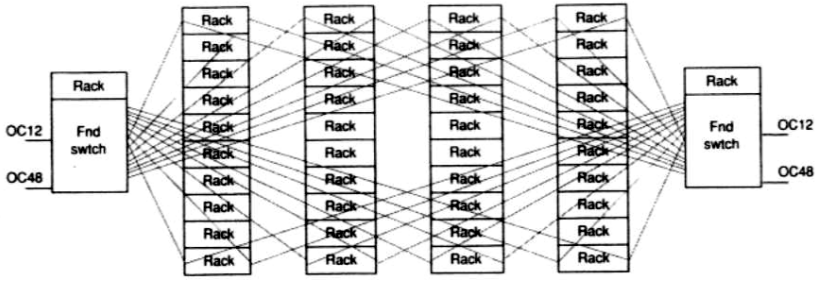
- Funcionamiento ante una petición
  - Selección del servidor Google que atenderá la petición (balanceo de carga basado en DNS entre varios clusters)
  - Búsqueda en un índice invertido que empareja cada palabra de búsqueda con una lista de identificadores de documentos
  - Los servidores de índice intersectan las diferentes listas y calculan un valor de relevancia para cada documento referenciado (establece el orden de los resultados)
  - Los servidores de documentos extraen información básica (título, URL, sumario, etc.) de los documentos a partir de los identificadores anteriores (ya ordenados)
- Características
  - Paralelismo multinivel
    - A nivel de petición: Se resuelven en paralelo múltiples peticiones independientes.
    - A nivel de término de búsqueda: Cada término de búsqueda puede chequearse en paralelo en los servidores de índice.
    - A nivel de índice: El índice de los documentos está dividido en piezas que comprenden una selección elegida aleatoriamente de documentos.
  - Principios del diseño de Google
    - **Fiabilidad hardware**: Existe redundancia a diferentes niveles, como alimentación eléctrica, discos magnéticos en array (RAID), componentes de alta calidad, etc.
    - **Alta disponibilidad y servicio**: Mediante la replicación masiva de los servicios críticos
    - **Preferencia en precio/rendimiento frente a rendimiento pico**: Múltiples servidores hardware de bajo coste configurados en cluster.
    - **PCs convencionales**: Uso masivo de PCs convencionales frente a hardware especializado en alto rendimiento.

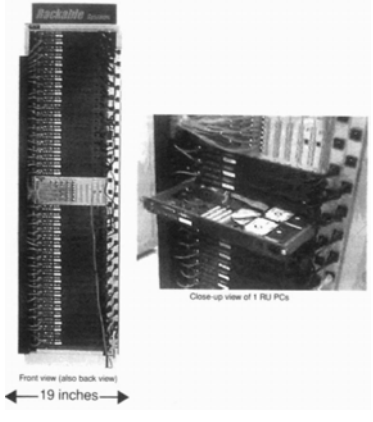
Ejemplo de Cluster: Google (ii)

(Datos de Diciembre de 2000)

- Se realizan alrededor de 1.000 consultas por segundo
- Cada página Web es visitada una vez al mes para actualizar las tablas de índices
- El objetivo es que la búsqueda tarde menos de 0.5 seg (incluido los retrasos de red)
- Utiliza más de 6000 procesadores y 12.000 discos (1 petabyte)
- Por fiabilidad dispone de tres *sites* redundantes: dos en Silicon Valley y uno en Virginia (sólo en uno se rastrea la Web para generar las tablas de índices)
- Para la conexión a Internet se utiliza una conexión OC48 (2488Mbits/seg)
- Dispone de conexiones extra (OC12, 622Mbits/seg) para hacer el sistema más robusto
- Dispone de dos switchs (Foundry, 128 x 1Gbit/s Ethernet) por fiabilidad
- Un *site* dispone de 40 *racks* y cada *rack* contiene 80 PCs
- Cada PC está compuesto por:
  - Un procesador desde un Pentium Celeron (533Mhz) a un Pentium III (800MHz)
  - Dos discos duros entre 40 y 80Gb
  - 256Mb de SDRAM
  - Sistema operativo Linux Red Hat
- Cada *rack* tiene dos líneas Ethernet de 1Gbit/s conectadas a ambos *switches*

Ejemplo de Cluster: Google (iii)





5. Programación y aplicaciones

- Modos de programación
  - Paralelismo de datos
  - Paralelismo de tareas o funciones
- Herramientas para generar programas paralelos
  - Bibliotecas de funciones para la programación
  - Lenguajes paralelos y directivas del compilador
  - Compiladores paralelos
- Estilos de programación
  - Paso de mensajes
  - Variables compartidas
  - Paralelismo de datos
- Estructuras de programas paralelos
- Fases de la programación paralela
- Ejemplo: Paralelizar el cálculo del número *pi*

Programación y aplicaciones

- Modos de progamación
  - Paralelismo de datos (SPMD, *Single-Program Multiple-Data*)
    - Los códigos que se ejecutan en paralelo se obtienen compilando el mismo programa.
    - Cada copia trabaja con un conjunto de datos distintos y sobre un procesador diferente.
    - Es el modo de programación más utilizado
    - Recomendable en sistemas masivamente paralelos
      - Difícil encontrar cientos de unidades de código distintas
    - Grado de paralelismo muy alto, pero no está presente en todas las aplicaciones.
  - Paralelismo de tareas o funciones (MPMD, *Multiple-Programs Multiple-Data*)
    - Los códigos que se ejecutan en paralelo se obtienen compilando programas independientes.
    - La aplicación a ejecutar se divide en unidades independientes.
    - Cada unidad trabaja con un conjunto de datos y se asigna a un procesador distinto.
    - Inherente en todas las aplicaciones.
    - Generalmente tiene un grado de paralelismo bajo.

Programación y aplicaciones

- Herramientas para generar programas paralelos
  - Bibliotecas de funciones para la programación
    - El programador utiliza un lenguaje secuencial (C o Fortran), para escribir el cuerpo de los procesos y las hebras. También, usando el lenguaje secuencial, distribuye las tareas entre los procesos.
    - El programador, utilizando las funciones de la biblioteca:
      - Crea y gestiona los procesos.
      - Implementa la comunicación y sincronización.
      - Incluso puede elegir la distribución de los procesos entre los procesadores (siempre que lo permitan las funciones).
    - Ventajas
      - Los programadores están familiarizados con los lenguajes secuenciales.
      - Las bibliotecas están disponibles para todos los sistemas paralelos.
      - Las bibliotecas están más cercanas al hardware y dan al programador un control a bajo nivel
      - Podemos usar las bibliotecas tanto para programar con hebras como con procesos
    - Ejemplos
      - MPI, PVM, OpenMP y Pthread.
  - Lenguajes paralelos y directivas del compilador
    - Utilizamos lenguajes paralelos o lenguajes secuenciales con directivas del compilador, para generar los programas paralelos.
    - Los lenguajes paralelos utilizan:
      - Construcciones propias del lenguaje, como p.e. FORALL para el paralelismo de datos. Estas construcciones, además de distribuir la carga de trabajo, pueden crear y terminar procesos.
      - Directivas del compilador para especificar las máquinas virtuales de los procesadores, o como se asignan los datos a dichas máquinas.
      - Funciones de biblioteca, que implementan en paralelo algunas operaciones típicas.
    - Ventajas: Facilidad de escritura y claridad. Son más cortos.
  - Compiladores paralelos
    - Un compilador, a partir de un código secuencial, extrae automáticamente el paralelismo a nivel de bucle (de datos) y a nivel de función (de tareas, en esto no es muy eficiente).

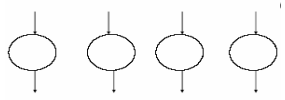
Programación y aplicaciones

- Estilos de programación
  - Paso de mensajes
    - La comunicación entre procesos se produce mediante mensajes de envío y recepción de datos.
    - Las transmisiones pueden ser síncronas o asíncronas
    - Normalmente se usa el envío **no** bloqueante y la recepción bloqueante.
    - PVM, MPI.
  - Variables compartidas
    - La comunicación entre procesos se realiza mediante el acceso a variables compartidas.
    - Lenguajes de programación: Ada 95 o Java.
    - Bibliotecas de funciones: Pthread (POSIX-Thread) u OpenMP.
    - Lenguaje secuencial + directivas: OpenMP.
  - Paralelismo de datos
    - Se aprovecha el paralelismo inherente a aplicaciones en las que los datos se organizan en estructuras (vectores o matrices).
    - El compilador paraleliza las construcciones del programador.
      - Bucles, operandos tipo vector o matriz, distribución de la carga entre los elementos de procesado.
    - C*, lenguaje muy ligado a procesadores matriciales
    - Fortran 90 (basado en Fortran 77) permite usar operaciones sobre vectores y matrices.
    - HPF (ampliación de Fortran 90).
      - Comunicaciones implícitas [A(I)=A(I-1)], paralelizar bucles (FORALL), distribuir los datos entre los procesadores (PROCESSORS, DISTRIBUTE), funciones de reducción (a partir de un vector obtenemos un escalar), funciones de reordenación de datos en un vector, etc.

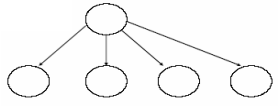
Programación y aplicaciones

- Estructuras de programas paralelos

Paralelismo ideal

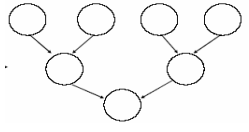


Maestro-Esclavo



Segmentación

Divide y vencerás



- Fases de la programación paralela
  - Descomposición funcional: Identificar las funciones que debe realizar la aplicación y las relaciones entre ellas.
  - Partición
    - Distribución de las funciones en procesos y esquema de comunicación entre procesos.
    - Objetivo: maximizar el grado de paralelismo y minimizar la comunicación entre procesos.
  - Localización
    - Los procesos se asignan a procesadores del sistema
    - Objetivo: ajustar los patrones de comunicación a la topología del sistema.
  - Escribir el código paralelo
  - Escalado: Determina el tamaño óptimo del sistema en función de algún parámetro de entrada.

Programación y aplicaciones

- Ejemplo: Paralelizar el cálculo del número *pi*.
  - Objetivo: Calcular el número *pi* mediante integración numérica, teniendo en cuenta que la integral en el intervalo [0,1] de la derivada del arco tangente de x es π/4.

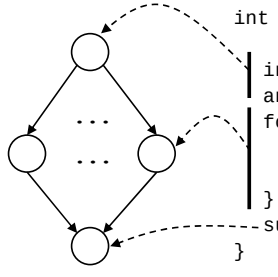
$$\left.\begin{array}{l} arctg'(x) = \dfrac{1}{1+x^2} \\ arctg(1) = \dfrac{\pi}{4} \\ arctg(0) = 0 \end{array}\right\} \Rightarrow \int_0^1 \frac{1}{1+x^2} = arctg(x)\Big|_0^1 = \frac{\pi}{4} - 0$$

  - Vamos a basarnos en una versión secuencial. En ella se aproxima el área en cada subintervalo utilizando rectángulos en los que la altura es el valor de la derivada del arco tangente en el punto medio.

Programación y aplicaciones

Versión secuencial

Grafo de dependencias entre tareas



```
Main (int argc, char **argv){
double ancho, x, sum;
int intervalos, i;

intervalos = atoi(argv[1]);
ancho = 1.0/(double) intervalos;
for (i=0; i<intervalos;i++){
     x = (i+0.5)*ancho;
     sum=sum + 4.0/(1.0 + x*x);
}
sum*=ancho;
}
```

  - Asignamos tareas a procesos o hebras.
    - Suponemos que tenemos un cluster Beowulf, con 8 PCs.
    - En cada procesador se ejecutará un proceso.
    - La carga de trabajo se distribuirá por igual entre los procesadores.
    - Realizaremos una planificación estática de las tareas.
  - Escribimos el código paralelo usando MPI (paso de mensajes).
    - Utilizaremos parelelismo de datos.
    - Partiendo de la versión secuencial obtenemos el código SPMD paralelo.

Programación y aplicaciones

Versión SPMD paralelo

Incluir librerías

```
#include <mpi.h>
Main (int argc, char **argv)
{
```

Declarar variables

```
double ancho, x, sum,     tsum=0;
int intervalos, i,        nproc,iproc;
```

Crear procesos

```
if(MPI_Init(&argc, &argv)=MPI_SUCCESS) exit(1);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
MPI_Comm_rank(MPI_COMM_WORLD, &iproc);

intervalos = atoi(argv[1]);
ancho = 1.0/(double) intervalos;
```

Asignar/localizar

```
for (i=iproc; i<intervalos;i+=nproc){
     x = (i+0.5)*ancho;
     sum=sum + 4.0/(1.0 + x*x);
}
sum*=ancho;
```

Comunicar/ sincronizar

```
MPI_Reduce(&sum,&tsum,1,MPI_DOUBLE,MPI_SUM,0,
           MPI_COMM_WORLD);
```

Terminar procesos

```
MPI_Finalize();
}
```

Programación y aplicaciones

- Crear y terminar procesos
  - *MPI_Init()*: Crea el proceso que lo ejecuta dentro del mundo MPI, es decir, dentro del grupo de procesos denominado MPI_COMM_WORLD. Una vez que se ejecuta esta función se pueden utilizar el resto de funciones MPI.
  - *MPI_Finalice()*: Se debe llamar antes de que un proceso creado en MPI acabe su ejecución.
  - *MPI_Comm_size(MPI_COMM_WORLD, &nproc)*: Con esta función se pregunta a MPI el número de procesos creados en el grupo MPI_COMM_WORLD, se devuelve en *nproc*.
  - *MPI_Comm_rank(MPI_COMM_WORLD, &iproc)*: Con esta función se devuelve al proceso su identificador, *iproc*, dentro del grupo.
- Asignar tareas a procesos y localizar paralelismo.
  - Se reparten las iteraciones del bucle entre los diferentes procesos MPI de forma explícita. Se utiliza un turno rotatorio.
- Comunicación y sincronización.
  - Los procesos se comunican y sincronizan para sumar las áreas parciales calculadas por los procesos.
  - Usamos MPI_Reduce. Esta es una operación cooperativa realizada entre todos los miembros del comunicador y se obtiene un único resultado final.
  - Los procesos envían al proceso 0 el contenido de su variable local *sum*, los contenidos de *sum* se suman y se guardan en el proceso 0 en la variable *tsum*.

Programación y aplicaciones

Versión OpenMP variables compartidas

Incluir librerías

```
#include <omp.h>
```

Declarar variables y constantes

```
#define NUM_THREADS 4

Main (int argc, char **argv)
{
Register double ancho, x;
int intervalos, i;

   intervalos = atoi(argv[1]);
   ancho = 1.0/(double) intervalos;
```

Crear/terminar

```
   omp_set_num_threads(NUM_THREADS)
   #pragma omp parallel
```

Comunicar/Sincronizar
Asignar/localizar

```
   #pragma  omp  for  reduction(+:sum)  private(x)
                schedule(dynamic)

   for (i=0; i<intervalos;i++){
        x = (i+0.5)*ancho;
        sum=sum + 4.0/(1.0 + x*x);
   }
   sum*=ancho;
}
```

Programación y aplicaciones

- Escribimos el código paralelo usando OpenMP con directivas (variables compartidas).
  - Crear y terminar hebras
    - *omp_set_num_threads()*: Función OpenMP que modifica en tiempo de ejecución el número de hebras que se pueden utilizar en paralelo; para ello sobrescribe el contenido de la variable OMP_NUM_THREADS, que contiene dicho número.
    - *#pragma omp parallel: esta directiva crea un conjunto de hebras; el número será el valor que tenga OMP_NUM_THREADS menos uno, ya que también interviene en el cálculo el padre. El código de las hebras creadas es el que sigue a esta directiva, puede ser una única sentencia o varias.*
  - Asignar tareas a procesos y localizar paralelismo.
    - *#pragma omp for*: La directiva *for* identifica o localiza un conjunto de trabajo compartido iterativo paralelizable y especifica que se distribuyan las iteraciones del bucle *for* entre las hebras del grupo cuya creación se ha especificado con una directiva *parallel*. Esta directiva obliga a que todas las hebras se sincronicen al terminar el ciclo.
      - » *schedule(dynamic):* Utilizamos una distribución dinámica de las hebras.
      - » *private(x):* Hace que cada hebra tenga una copia privada de la variable x, es decir, esta variable no será compartida.
      - » *reduction(+:sum):* Hace que cada hebra utilice dentro del ciclo una variable *sum* local, y que, una vez terminada la ejecución de todas las hebras, se sumen todas las variables locales *sum* y el resultado pase a una variable compartida *sum*.
  - Comunicación y sincronización.
    - Utilizamos una variable compartida llamada *sum* para almacenar el resultado final definida por reduction(+:sum).