Técnicas y Herramientas para Computación de Alto Rendimiento

Francisco Hidrobo

hidrobo@ula.ve

Universidad de Los Andes

Facultad de Ciencias

Contenido

- Introducción
- Mejoras de Rendimiento
- Caracterización y medición de Código
- Optimización y compiladores
- Concurrencia (Paralelismo)
- Tendencias y retos

Introducción

- Aspectos a considerar en Programación
  - Solución Correcta (Algoritmo(s) Correcto(s))
  - Modularidad y Legibilidad
  - Rendimiento (Mejor solución en tiempo)
  - Portabilidad (Se puede implantar en otras plataformas)
  - Código compacto (Programa ejecutable ”pequeño”)
  - . . .
- ¿Qué nos interesa?

Rendimiento

Rendimiento (Conflictos)

- Solución Correcta

  Mejorar un código podría implicar cambios en la solución

- Modularidad y Legibilidad

  La modularidad puede perjudicar el rendimiento (ej. Demasiadas clases o llamadas a subprogramas)

  Mejorar el código puede implicar agregar líneas

- Portabilidad

  El rendimiento puede variar de una máquina a otra.

  Típicamente, un código "óptimo" NO es portable

Mejorar rendimiento: ¿Por qué?

- Mejorar el tiempo de las operaciones más ”pesadas”
  - **Simulación**
- Obtener resultados en un límite de tiempo
  - **Predicción del clima**
- Aumentar el número de operaciones por unidad de tiempo
  - **Buscadores**

Rendimiento (tiempo)

- Tiempo transcurrido desde el inicio hasta el fin de una operación
  - Este tiempo es conocido como: tiempo de ejecución, tiempo del reloj, latencia, etc.
  - Una medida específica: El programa tarda 15.25 segundos en un P4 de 3.2 GHz
  - También puede usarse un tiempo de referencia
- Para realizar la medición se debe usar una máquina ”Dedicada”

Rendimiento (Velocidad)

- Utilizado de modo que el rendimiento puede ser independiente del tamaño de la aplicación
  - Ej. Comprimir un archivo de 1 MB toma 1', uno de 2 MB toma 2'. El rendimiento es el mismo.
- Millones de instrucciones por segundo (MIPS)
  - MIPS = (instrucciones) / ((tiempo)* 10^6)
    = (velocidad del reloj) / (CPI * 10^6)
  - Sin embargo, el conjunto de instrucciones entre arquitecturas no es equivalente
    - Una instrucción CISC = muchas instrucciones RISC
    - Diversas mezclas de instrucciones
    - Mismo programa misma arquitectura

Rendimiento (Velocidad)

- Millones de operaciones punto flotante por segundo (Mflops)
  - Popular (puede ser engañosa)
  - e.j: Se puede tener un número grande de MFlops en un algoritmo muy malo y obtener un pésimo rendimiento de la aplicación.
- Específicos de la aplicación:
  - Millones de *frames renderizados* por segundo
  - Millones de pares de aminoácidos comparados por segundos
  - Millones de solicitudes HTTP servidas por segundo

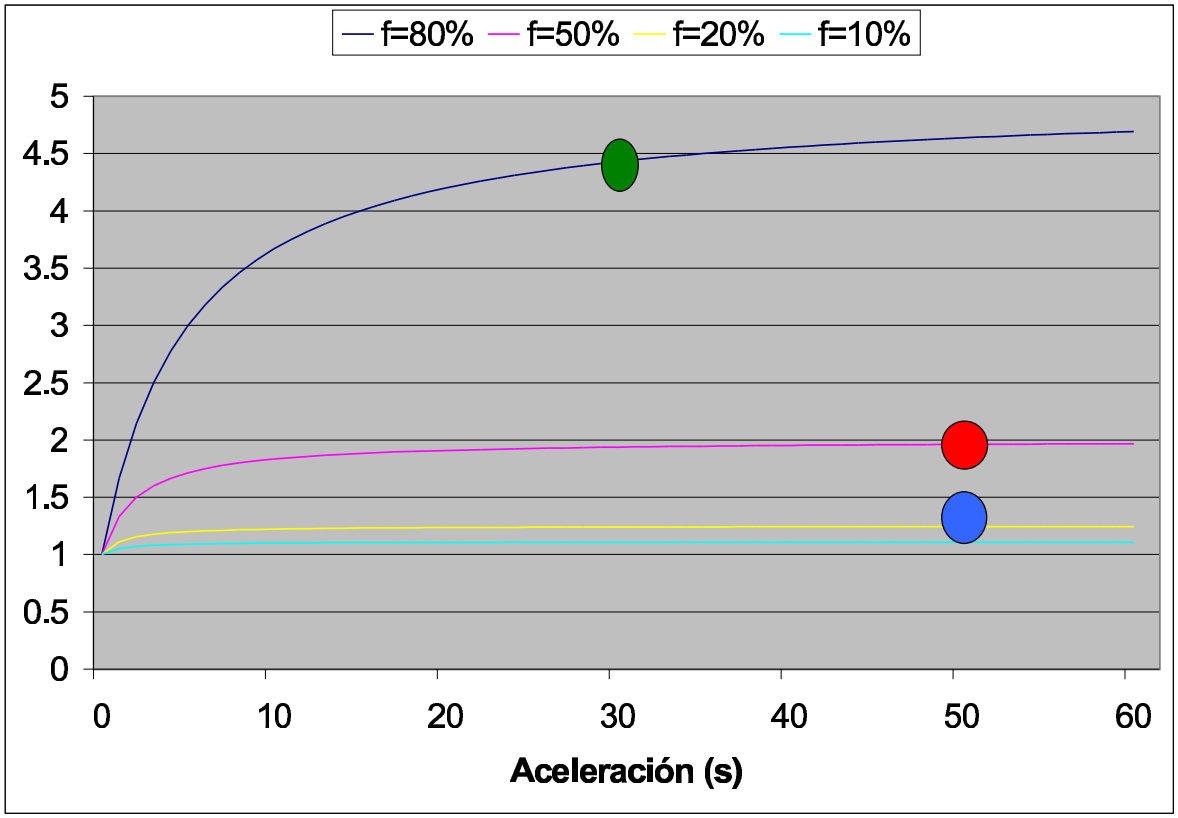
Bajo rendimiento (Razones)

- Asumamos que se ha identificado una porción del código que tiene bajo rendimiento
- Existen los denominados ”cuellos de botella”
  - Una aplicación se ejecuta en una plataforma que tiene múltiples componentes: CPU, memoria, sistema operativo, conexión de red, disco duro, tarjeta de vídeo, etc.
  - ”Solución”: Se escoge un componente y se hace más rápido. Si el rendimiento de la aplicación mejora, ese componente era un cuello de botella.

Aceleración

- Se requiere una métrica que permita medir el impacto de la mejora de rendimiento
- Aceleración: cambio del tiempo anterior al nuevo
  - tiempo original = 2h
  - tiempo con mejora = 1h
  - aceleración = 2h / 1h = 2
- Ley de Amdahl: $S = \frac{1}{1-f+f/s}$
- Puede hablarse de desaceleración si la mejora no es beneficiosa

Ley de Amdahl: Gráficamente



Acelerar 50 veces el 20 % del código nos dará una aceleración total de $\approx 1{,}25$

Acelerar 50 veces el 50 % del código nos dará una aceleración total de $\approx 1{,}96$

Acelerar 30 veces el 80 % del código nos dará una aceleración total de $\approx 4{,}41$

Acelerar el 10 % del código es **inútil**

Contenido

- Introducción
- Mejoras de Rendimiento
- Caracterización y medición de Código
- Optimización y compiladores
- Concurrencia (Paralelismo)
- Tendencias y retos

¿Cómo mejorar el rendimiento?

- Opción 1: Comprar hardware más rápido
  - Puedes quedarte corto
  - Las velocidades de los componentes son diferentes. No puedes esperar a que la tecnología mejore y los precios bajen
  - Algunas veces, puedes lograr el mismo efecto mejorando un poco el código

¿Cómo mejorar el rendimiento?

- Opción 2: Cambiar el algoritmo
  - Ejemplo: Buscar un elemento en un vector ordenado
  - Primera solución: búsqueda secuencial
    - fácil
    - resuelve el problema
  - Cuando el rendimiento es importante, cambiar la búsqueda secuencial por búsqueda binaria
    - compleja
    - más rápida para vectores grandes

¿Cómo mejorar el rendimiento?

- Opción 3: Modificar la estructura de datos
  - Ejemplo: Lista enlazada
    - El método lista.numElementos(). Recorre los elementos y va incrementando un contador
    - Aumenta el tiempo de ejecución (*overhead*) si se ejecuta frecuentemente
    - Cambio: agregar el atributo **longitud**. Sumar 1 al agregar y restar 1 al eliminar
    - Ahora lista.numElementos() simplemente devuelva el valor de **longitud**
    - No afecta demasiado a insertar() y eliminar(), solo requiere 4 bytes adicionales

¿Cómo mejorar el rendimiento?

- Opción 4: Modificar la implementación
  - No cambia el espíritu del algoritmo, pero ....
    - cambiar el orden de las instrucciones
  - Cambiar la organización del código
    - eliminar funciones
    - expandir y/o desenrrollar lazos

¿Cómo mejorar el rendimiento?

- Opción 5: Utilizar concurrencia
  - Código MultiHilo en una máquina de un solo procesador para utilizar los recursos de hardware más eficientemente
  - Código MultiHilo en máquinas de multi-CPU/multi-core
  - Código MultiHilo en muchas máquinas individuales, posiblemente multi-CPU/multi-core

Contenido

- Introducción
- Mejoras de Rendimiento
- Caracterización y medición de Código
- Optimización y compiladores
- Concurrencia (Paralelismo)
- Tendencias y retos

Midiendo el rendimiento

- Primera idea: ¡ A mano !. Mirar un reloj, ejecutar el programa y volver a mirar el reloj cuando termine
- Desventajas
  - Muy mala resolución
  - Requiere la atención del usuario
- Afortunadamente, los sistemas operativos proveen maneras de tomar esta medida automáticamente
- En los sistemas basados en UNIX, existe el comando *time*

Usando *time*

- Puedes colocar el comando *time* antes de cualquier otro comando
- Cuando el comando termina, *time* reportará el tiempo de ejecución (puede que más información) del comando dado
- Ejemplo:
  time ./programa

```
real    0m11.770s
user    0m6.796s
sys     0m0.980s
```

Midiendo porciones del código

Función *gettimeofday()* (C estándar) y rutina *CPU_Time* (Fortran)

- ¿Cómo medir porciones del código?
  1. Llamar a *gettimeofday()* al inicio de la sección
  2. Llamar a *gettimeofday()* al final de la sección
  3. Calcular la diferencia entre la cantidad de microsegundos

Caracterizando el código

- Podemos medir el rendimiento en porciones del código
- La ley de Amdahl nos dice: ”mejora las partes del código que más tardan”
- Si escribo el código puedo tener alguna idea de cual es la porción que más tarda (la más importante)
- ¿Qué pasa si obtengo un gran pedazo de código que ha escrito otra(s) persona(s)?
  - Primera opción: entender el código e insertar llamadas a *gettimeofday* en todas partes (Complicado!!!)
  - Segunda opción: usar un ***profiler***

¿Qué es un *profiler*?

- Es una herramienta para monitorear la ejecución de un programa y reportar el tiempo de sus funciones
- Sirve para identificar las funciones más costosas
- Pasos para hacer *profiling* :
  1. Compilar el código para obtener perfiles
  2. Ejecutar el programa
  3. Identificar la función más costosa
  4. Optimizar esa función
     - Mejorarla
     - Llamarla con menos frecuencia
  5. Regrese a 1 si cree que puede mejorar otra función

El *gprof* de UNIX

- Compilar el código con *gcc* agregando la opción *-pg*
- Ejecute el programa generado
- Ejecute *gprof* con el nombre de su programa
- Ejemplo:
  - *gcc -pg programa.c -o programa*
  - *./programa*
  - *gprof programa > programa.gprof*
- El archivo de salida contiene la información (el perfil del programa)

Herramienta *kprof*

- Es una herramienta gráfica para visualizar archivos generados con *gprof* (también soporta otros *profilers*)
- Usando *kprof*
  - *gcc -pg programa.c -o programa*
  - *./program*
  - *gprof -b programa > programa.gprof*
  - *kprof -f programa.gprof*
- Ayuda: *kprof ––help*

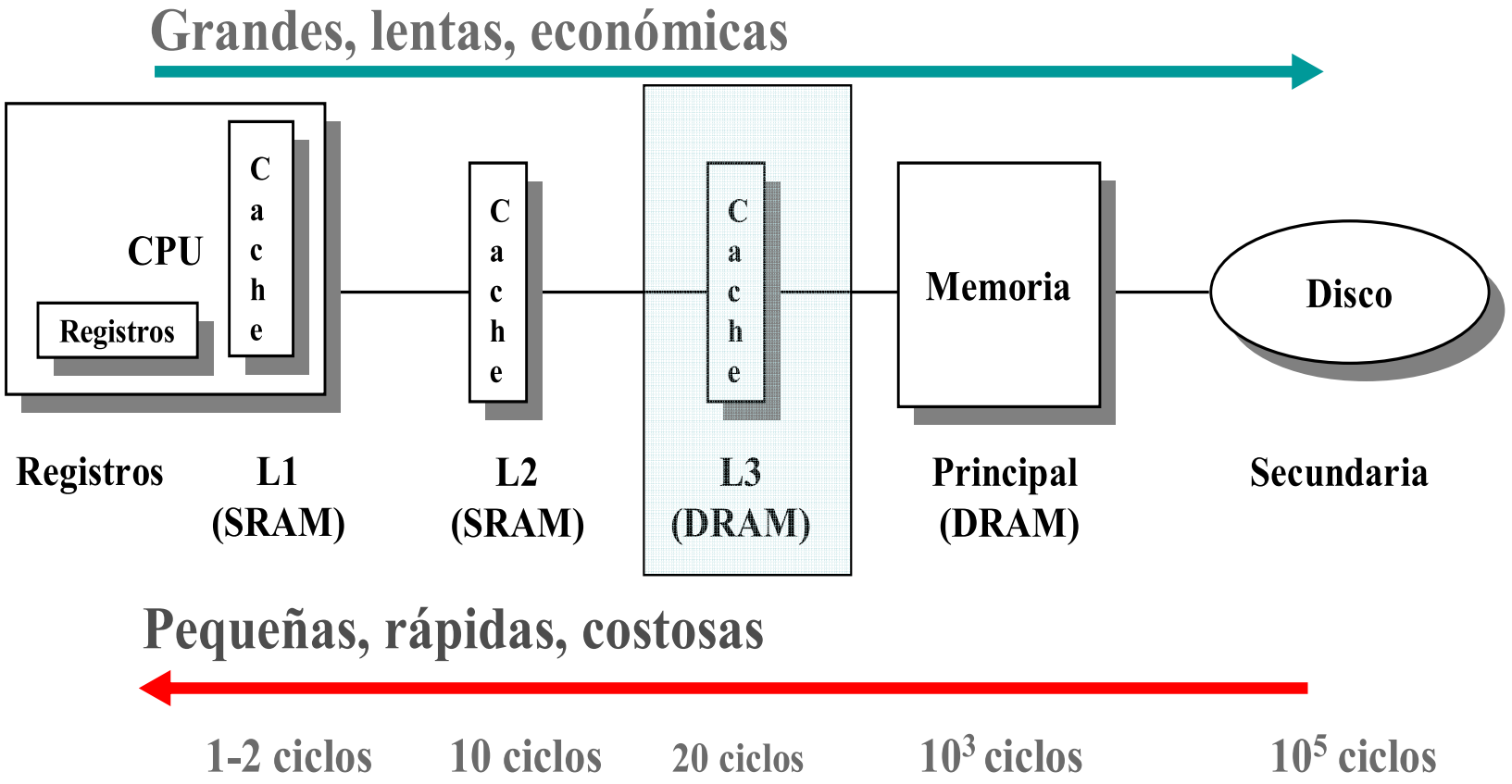
¿Cómo utilizar un *profiler* ?

1. Obtenga la salida del *profiler*
2. Identifique la(s) función(es) más costosas
3. Intente optimizar esa(s) función(es)
   - Elimine los cuellos de botella
   - Reduzca el número de llamadas
4. Repita el paso anterior hasta que no encuentre mejoras

¿Dónde mirar primero?

- En los accesos a Memoria
- Problema típico: no se usa adecuadamente la *cache*
  - La cache hace creer que se trabaja con una memoria grande y barata, con los tiempos de acceso de una memoria costosa.
  - Usando cache se aprovecha la localidad de los accesos a memoria; especificamente la localidad espacial y la localidad temporal.
  - ¿Por qué es importante la cache?
    R.- Porque el código puede ser reorganizado para aprovechar la localidad

Jerarquía de memoria



Conocer la jerarquía de memoria

- ¿Para qué?.
  Si el programador conoce detalles de la jerarquía podría mejorar rendimiento
  - Si sabemos que la cache L2 es de 256 KB, podríamos descomponer el problema en subproblemas que trabajen con 256 KB de datos
  - Si sabemos que la línea de cache es de 32 bytes, podríamos contar los fallos de cache y calcular algunos parámetros para la operación óptima

Contenido

- Introducción
- Mejoras de Rendimiento
- Caracterización y medición de Código
- Optimización y compiladores
- Concurrencia (Paralelismo)
- Tendencias y retos

Técnicas de Optimización

1. Identificación de constantes (Invariantes en el lazo)

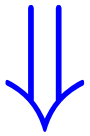
```
for (kk=0;kk<N;kk++)
    C[ii][jj] += A[ii][kk] * B[kk][jj];
```

```
suma = 0.0;
for (kk=0;kk<N;kk++)
    suma += A[ii][kk] * B[kk][jj];
C[ii][jj] = suma;
```

Técnicas de Optimización

2. Acceso a arreglos por referencias a apuntadores

```
for (jj=0;jj<N;jj++)
    mat[ii][jj]=2;      // 2*N sumas y N multiplicaciones
```

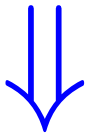


```
double *aptr = &(mat[ii][0]);      // 2 sumas y 1 multiplicación
for (jj=0;jj<N;jj++) {
    *aptr= 2;
    aptr++;      // N sumas
}
```

Técnicas de Optimización

3. Desenrrollado de lazos

```
for (ii=0; ii<N; ii++)
    vec[ii] = ii;
```

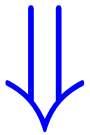


```
ii=0;
do {
    vec[ii] = ii;ii++;
    vec[ii] = ii;ii++;
    vec[ii] = ii;ii++;
    vec[ii] = ii;ii++;
} while (ii <N); / / (Menos comparaciones)
```

Técnicas de Optimización

4. Movimiento de código

```
suma = 0;
for (ii = 0; ii <NN/kk; ii++)
    suma += ii;
```

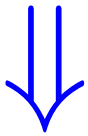


```
suma = 0;
div = NN/kk;
for (ii = 0; ii <div; ii++)
    suma += ii;
```

Técnicas de Optimización

5. Eliminación del llamado de funciones

```
suma=0.0;
for (ii=0;ii<N;ii++)
    suma += cubo(ii);
. . .
float cubo(x)
{ return (x*x*x); }
```

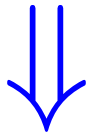


```
suma=0.0;
for (ii=0;ii<N;ii++)
    suma += ii*ii*ii;
```

Otras técnicas de Optimización

6. Eliminación de expresiones comunes

```
area = pi*r*r;
peri = 2*pi*r;
```



```
tmp = pi*r;
area = tmp*r;
peri = 2*tmp;
```

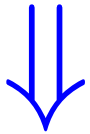
Otras técnicas de Optimización

7. Eliminación de código inútil

```
y = 10;
:
y = m*x + b;
```

Parece obvio; pero podría estar oculto

```
int x = 0;
...
#ifdef MAXVAL
    x = MAXVAL*2;
#endif
```

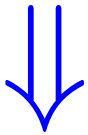


```
:
y = m*x + b;
```

Otras técnicas de Optimización

8. Reducción a la fuerza

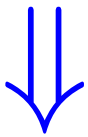
```
v = k*3;
```



```
v = k+k+k;
```

9. Propagación de constantes

```
int velocidad = 3;
eficiencia = 100* velocidad / procs;
def = eficiencia * 2;
```



```
def = 600 / procs;
```

¿Entonces que hacemos?

- Se han presentado algunas técnicas de optimización
- Debemos tener presente que existen algunas otras
- ¿Qué podemos hacer?
  Podríamos aplicar todas estas técnicas a nuestro código; pero podemos obtener un código difícil de leer y depurar
- Afortunadamente, los compiladores nos ayudan mucho
- Con algo de suerte, el compilador puede hacer una buena parte del trabajo de optimización.

¿Qué hacen los compiladores?

- Todos los compiladores actuales realizan alguna optimización automática cuando generan código
- Los compiladores ofrecen diferentes niveles:
  -O0: Ninguna optimización (siempre se hace algo)
  -O1, -O2, ... -OX
- A más alto nivel de optimización más probable es que el depurador tenga problemas con el código
  Si va a depurar use -O0 (en algunos compiladores -g significa -O0)
- Debe tomar en cuenta que los niveles de optimización alto pudieran tener efectos sobre los resultados

¿Qué hacen los compiladores?

- Eliminación de llamadas a funciones
- Asignación de variables a registros (no es fácil)
- Eliminación de código inútil
- Simplificación algebraica
- Movimiento de código fuera de los lazos
- Propagación de constantes
- Simplificación del flujo de control
- Planificación y reordenamiento de instrucciones
- Reducción a la fuerza (usando apuntadores)
- Desenrrollado de lazos y encauzamiento de software
- Eliminación de almacenamiento inútil
- otras · · ·

Limitaciones

- El comportamiento que puede ser obvio para el programador podría ser transformado por los lenguajes y los estilos de codificación (ej. Rangos de datos más limitados que los sugeridos por el tipo)
- La mayor parte del análisis es interno a los procedimientos. Analizar el programa completo sería demasiado costoso
- El análisis se realiza con información estática. Es imposible que el compilador pueda predecir las entradas tomadas en tiempo de ejecución.
- Si hay dudas, el compilador debe ser conservador, no puede cambiar el programa para ciertas circunstancias; aunque esas circunstancias sean poco probables.

¿Cómo obtener mejor rendimiento?

- Se han presentado algunas técnicas para optimizar el código
- Sabemos que los compiladores hacen una buena cantidad de cosas
- ¿Esto significa que los programadores no tienen nada de que preocuparse?
  R.- Desafortunadamente, NO

Recomendaciones

Escribir un buen código significa trabajar en conjunto con el compilador

1. Optimizar las cosas que sabemos que el compilador no podrá hacer
   Por ejemplo, trabajar con subgrupo de datos (por bloques) debe ser hecho a mano; pero el compilador podría encontrar el mejor orden para lazos anidados (i, j)
2. Escribir el código para permitir que el compilador pueda hacer sus optimizaciones. Esto es, debemos eliminar los **bloqueadores** de optimización

Contenido

- Introducción
- Mejoras de Rendimiento
- Caracterización y medición de Código
- Optimización y compiladores
- Concurrencia (Paralelismo)
- Tendencias y retos

Concurrencia y paralelismo

- Concurrencia ⇒ múltiples **tareas** al mismo **tiempo**
- Es común en la computación (a nivel de hardware):
  - Cargar datos de memoria mientras suma dos registros
  - Sumar dos pares de registros simultáneamente
  - Recibir datos de la red mientras escribe a disco
  - Sistemas de doble procesador
  - *Clusters* de estaciones de trabajo
- La concurrencia puede ser:
  - Real: Las cosas pasan de verdad al **mismo tiempo**
  - Simulada: Se cambia muy rápido de una actividad a otra, **parece** que pasan al mismo tiempo

Medidas de rendimiento paralelo

- La manera más simple de medir que tan bueno es un programa paralelo es calcular su aceleración
- El programa tarda un tiempo T_1 en 1 procesador
- El programa tarda un tiempo T_p en p procesadores
- La aceleración se define como:

$$S(p) = \frac{T_1}{T_p}$$

Eficiencia Paralela

- Se define como: $E_f = S(p)/p$
- El valor esperado es 1, a menos que tengamos aceleración superlineal
- Se usa para medir que tanto se aprovechan los procesadores
  - Si aumentando el número de procesadores en factor de 10, la aceleración se aumenta en un factor de 2, quizás no vale la pena el "gasto"; la eficiencia se dividirá entre 5.
  - Los recursos para usar paralelismo suelen ser costosos (tiempo de máquina)

Modelos de programación paralela

- 2 modelos
  - Espacio de memoria compartida
  - Intercambio de mensajes
- Parelelismo Híbrido
- Se relacionan con la arquitectura subyacente
  - Ciertas arquitecturas son concebidas para ciertos modelos de programación
  - Todos los modelos pueden ser implantados/emulados en más de un tipo de arquitectura
  - En realidad, todos pueden emularse en un computador de un solo procesador (computador no paralelo)

Modelo de Programación Paralela

- Control
  - ¿Cómo se crea el paralelismo?
  - ¿Cómo se hace el flujo de control entre las instrucciones?
  - ¿Cómo se sincronizan los diferentes hilos (hebras) de control?

Modelo de Programación Paralela

- Nombramiento
  - ¿Cúales datos son privados?
  - ¿Cúales datos son compartidos?
  - Si los datos son compartidos, ¿Cómo se comparten?
    - ¿A través de la memoria?
    - ¿Por comunicación explícita?
- Operaciones
  - ¿Cúales son las operaciones básicas?

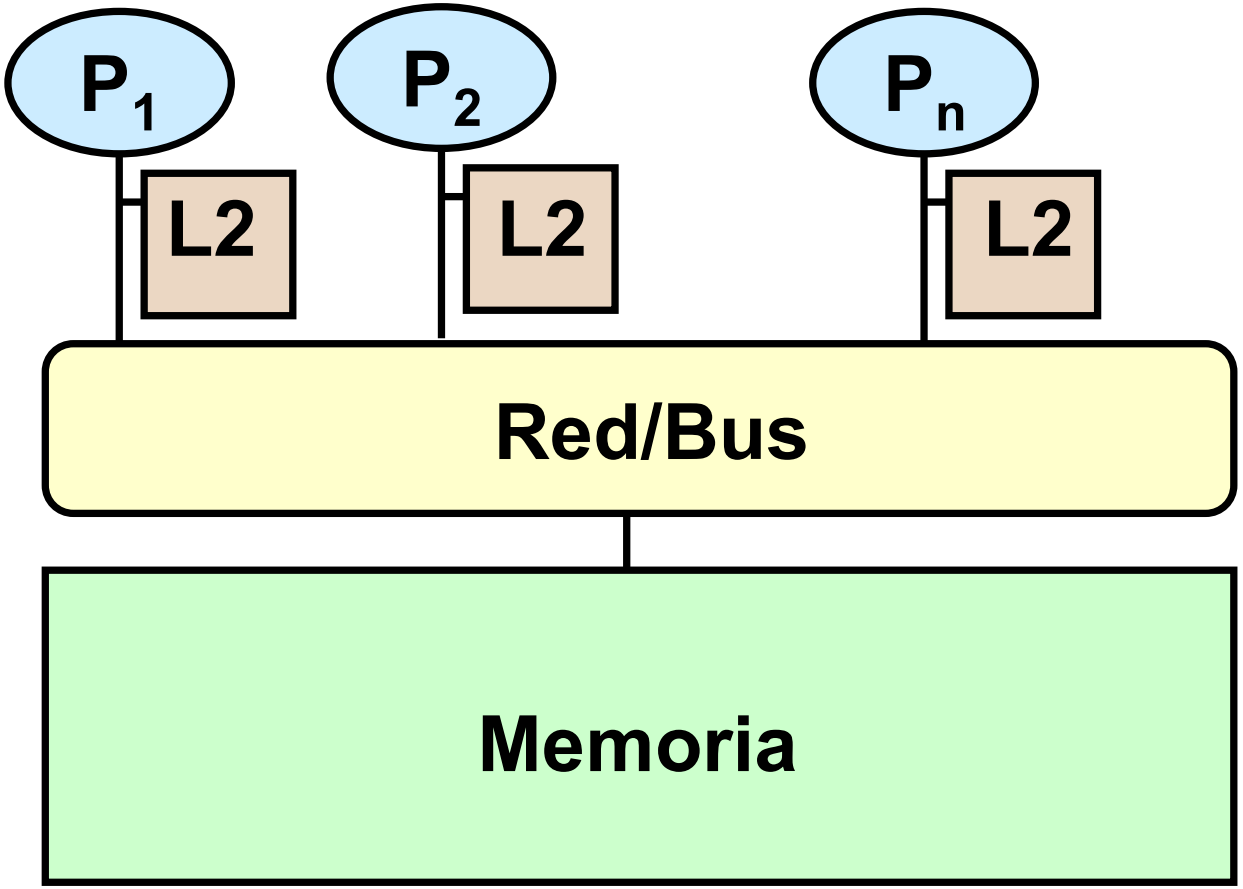
    ¿Cúal es la API?
  - ¿Cúales operaciones son atómicas?

Espacio de Memoria Compartida

- El programa está formado por una colección de **Hilos**
- Cada hilo puede tener variables locales (sus propias variables)
- Colectivamente todos los hilos pueden compartir variables (variables globales)
- Deben existir mecanismos para controlar el acceso a memoria compartida

Modelo subyacente de la máquina

Máquina de Memoria Compartida



Pthreads

- Un API estándar de POSIX (IEEE 1003.1c) para la creación y sincronización de hilos
  - El API especifica el comportamiento del estándar
  - Opciones de implantación en manos de los desarrolladores
  - Las implantaciones varían de sistema a sistema, unas mejores que otras
- Común en todos los sistemas UNIX
- También existe para Win32
- Enfocada en la portabilidad

OpenMP

- Objetivo: Hacer la programación en memoria compartida fácil (al menos más fácil que con Pthread)
- ¿Cómo?
  - Una biblioteca con funciones simples
  - Definiendo unas pocas directivas (en C: pragmas, en Fortran: C$OMP)
    - Una forma de extender el lenguaje
    - Permiten dar pistas/información al compilador
  - Un compilador

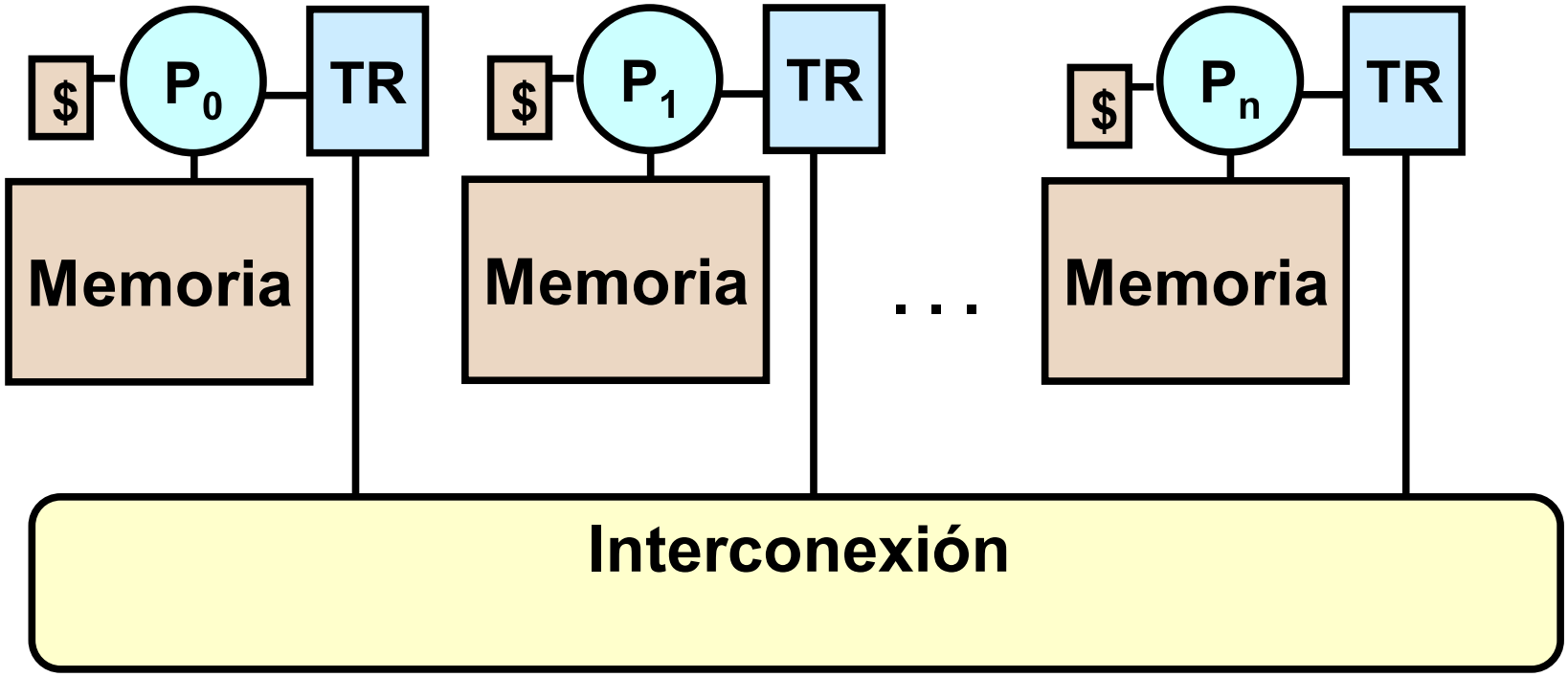
Intercambio de Mensajes

- El programa consiste de un grupo de procesos
  - No comparten espacio de memoria
  - La terminología de **hilos** y **procesos** proviene del mundo de sistema operativos

    bajo UNIX, los **procesos** no comparten memoria (a pesar que existen mecanismos para hacerlo), mientras que los **hilos** si lo hacen
- Los procesos se comunican y sincronizan mediante la transferencia de datos (Intercambiando mensajes)
  - ENVIAR (*SEND*): dato a un proceso particular
  - RECIBIR (*RECEIVE*): dato de un proceso particular o de cualquier proceso

Modelo subyacente de la máquina

Máquina de Memoria Distribuida



MPI

- MPI (1992) se ha convertido en un estándar de facto con los siguientes objetivos:
  - Portabilidad del código fuente
  - Desarrollo de implantaciones eficientes
  - Soporte para plataformas heterogéneas
- MPI no es:
  - Un lenguaje de programación
  - una implantación (da recomendaciones sobre cómo implantar)
- Opera con C, C++, Fortran y Java

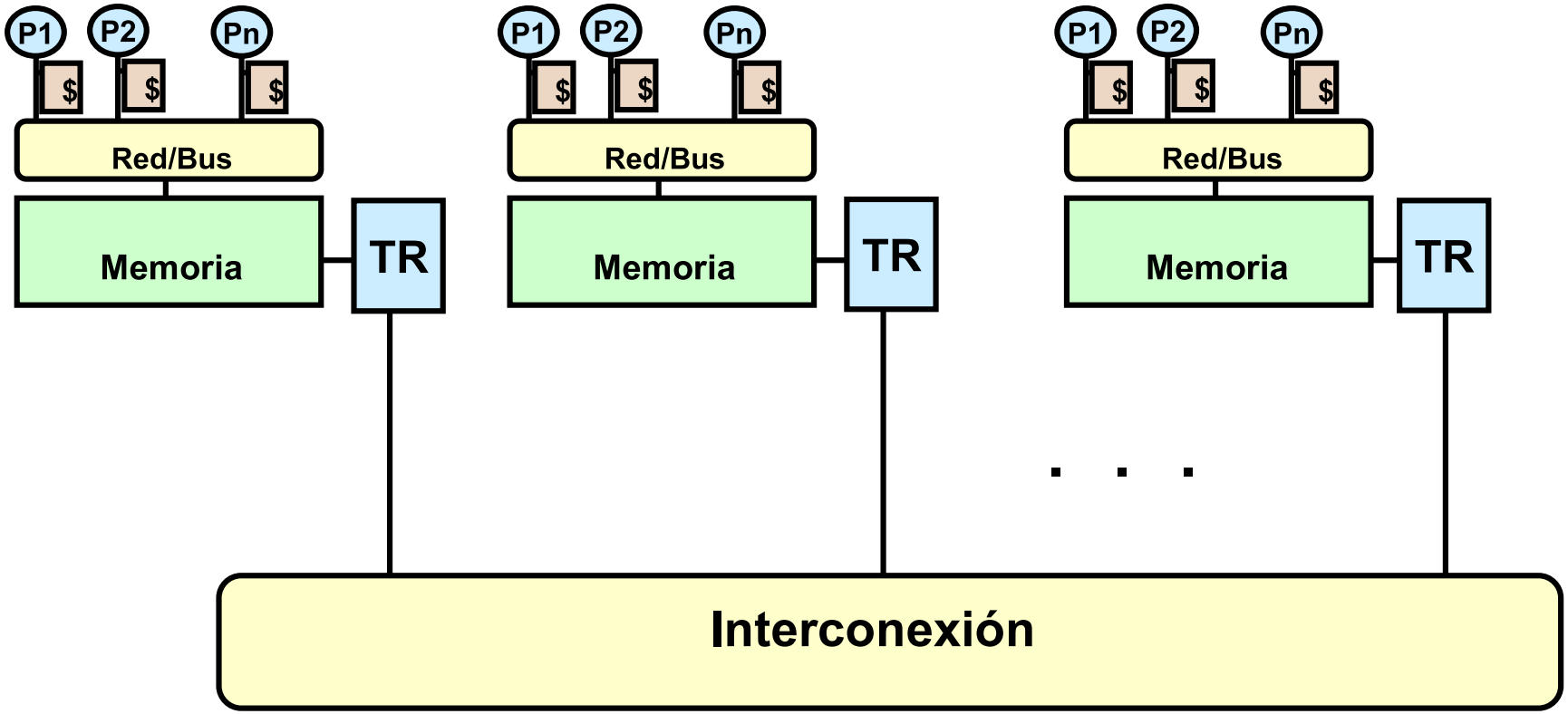
  http://www-unix.mcs.anl.gov/mpi/
- Implantaciones:
  - Bien adoptado por marcas comerciales (*vendors*)
  - Implantaciones de libre distribución: MPICH, LAM, CHIMP/MPI
  - Trabajos en tolerancia a fallos: MPICH-V, FT-MPI, MPIFT, etc.

Paralelismo Híbrido

- Es una mezcla de los modelos anteriores
- El programa está formado por un número de procesos (que no comparten memoria)
- Cada proceso está formado por hilos (con memoria compartida)
- Los procesos se sincronizan por mensajes
- Los hilos internos de cada proceso se sincronizan por variables compartidas
- Combina las ventajas de ambos modelos
- También combina las posibilidades de errores

Modelo subyacente de la máquina

Arquitectura Híbrida



Contenido

- Introducción
- Mejoras de Rendimiento
- Caracterización y medición de Código
- Optimización y compiladores
- Concurrencia (Paralelismo)
- Tendencias y retos

Mejoras Tecnológicas

- Necesitamos:
  - Muchos Gflops
  - Muchos GBytes
- Los componentes son más rápidos, más grandes (mayor capacidad) y más baratos
- ¿Podemos esperar a que la tecnología cubra todas las necesidades?

  Algunas aplicaciones no pueden esperar a que esto ocurra (si es que llega a ocurrir)

Clusters, Constellations, MPPs

- Actualmente, sólo hay tres categorías en el Top500

  www.top500.org

- Todos son máquinas con memoria físicamente distribuida que puede ser direccionada globalmente (lógicamente compartida)
- Cada nodo/procesador tiene su propia memoria y cache pero no puede acceder directamente a la memoria de otro Cada nodo puede ser un Multiprocesador Simétrico (SMP)
- Cada nodo tiene una interfaz de red para realizar todas las comunicaciones y sincronizaciones
- *GRID Computing*

Herramientas

- OpenMP
  (http://www.compunity.org/resources/freeware/index.php)
- MPI
  (http://www.lam-mpi.org/mpi/implementations/shortlist.php)
- HPF
  (http://dmoz.org/Computers/Parallel_Computing/Programming/Languages/High_Performance_Fortran/)
- PBLAS
  (http://www.netlib.org/scalapack/pblas_qref.html)
- Atlas
  (http://math-atlas.sourceforge.net/)
- ScaLAPACK
  (http://netlib2.cs.utk.edu/scalapack/)
- FreeMat
  (http://freemat.sourceforge.net/)

Areas de aplicación de HPC

- Ciencia
  - Modelado climático Global
  - Modelado astrofísico
  - Biología: Genoma, diseño de drogas, análisis de proteínas
  - Química Computacional
  - Ciencias de los materiales y nanociencia
- Ingeniería
  - Simulación de choques
  - Diseño de semiconductores
  - Modelos estructurales y terremotos
  - Dinámica de fluidos
  - Combustibles
- Economía
  - Modelos financieros y económicos
  - Procesamiento de transacciones, servidores WEB, buscadores

Retos

- Infraestructura
  Máquinas, redes, laboratorios, etc.
- Gerencia y Administración
  Manejo y operación
- Formación
  Ciencias Computacionales
- Colaboración
  Grupos multidisciplinarios