ARQUITECTURA DE COMPUTADORES II

AUTORES:
David Expósito Singh
Florin Isaila
Daniel Higuero Alonso-Mardones
Javier García Blas
Borja Bergua Guerra

Área de Arquitectura y Tecnología de Computadores
Departamento de Informática
Universidad Carlos III de Madrid

Julio de 2012

TEMA 3:
MP DE MEMORIA COMPARTIDA (I)

Índice

Índice

3. Programación
   - Comunicación
   - Sincronización
     - Tipos
     - Historia
     - Componentes
     - Implementación
       - `locks`
       - `barriers`

Introducción

- **Definición**:
  - En un **MP de Memoria Compartida** (*Shared-Memory MP*) cualquier procesador puede acceder a cualquier posición de memoria.
- **Ventajas**:
  - La ubicación es "***transparente***".
  - Su "modelo de programación natural" es similar al que se usa en los sistemas monoprocesadores con "*time-sharing*".
  - Pero los procesos se pueden ejecutar en procesadores diferentes.
  - Implican habitualmente una mejora del *throughput*.
- **Mercado:**
  - La gran mayoría de los MP existentes son de Memoria Compartida.
  - Tendencias futuras: los PCs serán MP de Memoria Compartida.

Introducción (II)

- **Uso (habitual):**
  1. Como ‘**servidores**’ (ej., Sun servers).
  2. Como sistemas de cálculo de **programación paralela:**
     - Permiten compartición con tamaño de grano todo lo “fino” que se quiera.
     - La comunicación (acceso ‘convencional’) y la sincronización (acceso ‘atómico’) es siempre a través de variables compartidas.
     - Modelo *load/store*: no necesarios nuevos mecanismos en la arquitectura de los procesadores.
     - “Automatización” del movimiento de datos (caches) y de la gestión de su coherencia.

Introducción (y III)

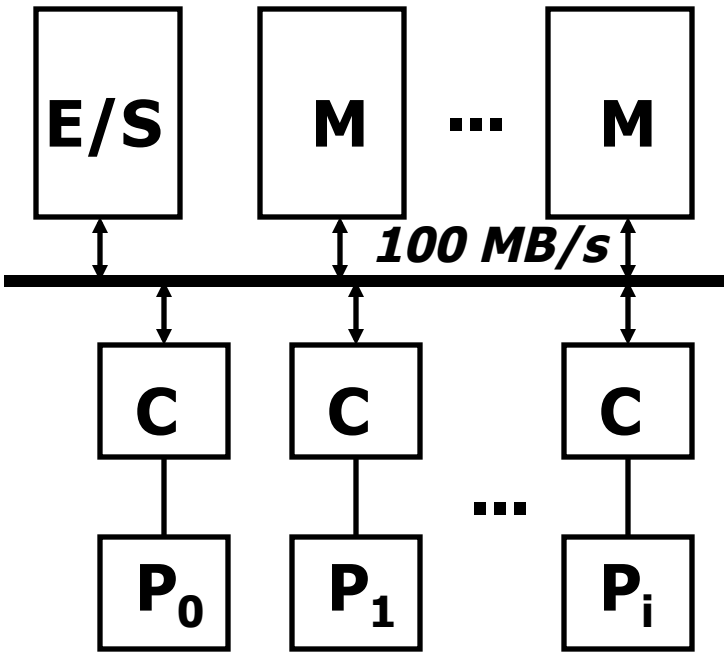
- **Nomenclatura:**
  - Tradicionalmente: *MP de Memoria Compartida*.
  - Hoy en día engloba:
    - **UMA** (*Uniform Memory Access*).
    - **NUMA** (*Non* UMA), también llamados *Distributed Shared Memory* o *Virtual Shared Memory.*

Problemática

- **Implementación:**
  - UMA:
    - Mayoritariamente basado en un BUS.
    - Otra red de interconexión: p.ej., *crossbar*.
  - NUMA:
    - Otra red de interconexión: anillo, hipercubo, toro, etc.

- **'Escalabilidad':**
  - UMA:
    - Basado en un BUS: muy poca: límite 20-30 procesadores
    - Otra red: 106 procesadores: caso extremo *Sun Fire* 15.000.
  - NUMA:
    - Se supone que está garantizada hasta, p.ej., los 2048 procesadores del *Cray T3E*.

Ej. saturación del bus en un UMA



- ¿cuántos procesadores puede soportar?
  - Cada procesador:
    - 50 MIPS
    - Tamaño instrucciones: 4 Bytes.
    - Tamaño datos: 4 Bytes.
    - Tamaño bloque datos: 16 Bytes.
    - 30% de *load/stores*.
  - *Hit ratios*:
    - 98% instrucciones.
    - 95% datos.

Programación

- **Comunicación:**
  - Implícita a través de variables compartidas:
  - ¿Cómo se implementa paso de mensajes, p.ej. MPI, sobre un MP de Memoria Compartida?
- **Sincronización:**
  - Tipos:
    - Exclusión mutua:

      Locks
    - Sincronización de eventos:

      Barriers

Sincronización (I)

- **Historia**
  - Ha habido, y sigue habiendo, mucho debate sobre qué mecanismos debe suministrar la **arquitectura** para soportar la sincronización:

    CONCLUSIÓN:

    flexibilidad vs. rendimiento

  - La mayoría de las arquitecturas emplea operaciones *read-modify-write* atómicas.

Sincronización (II)

- Ejemplos:
  - IBM 370: ***compare&swap*** atómico para multiprogramación.
  - Intel x86: cualquier instrucción (de acceso a Memoria) puede llevar el **prefijo *LOCK*** (acceso atómico)
  - Los SPARC (RISC): operaciones atómicas registro-memoria: ***swap, compare&swap***
  - MIPS, PowerPC, Alpha: no accesos atómicos sino pareja de instrucciones ***load-locked / store-conditional***
  - En general, mucho debate y discusión: qué implementar en el *hardware*, cómo implementarlo, etc.

Sincronización (III)

- **Componentes de un evento de comunicación:**
  1. Método de adquisición / liberación (*release*):
     - Adquirir derecho a la sincronización: p.ej., entrar en la región crítica.
     - "Liberar" la sincronización.
  2. Método de espera:
     - Con bloqueo:
       - El proceso en espera puede pasar a hacer otras cosas: "replanificado" o bien suspendido, pasa a la cola de "ready".
       - Mucho *overhead.*
     - Espera activa: *spin-lock*
       - Los procesos en espera muestrean repetidamente una variable hasta que su valor cambia.
       - Produce tráfico extra en la red.
       - Adecuado si:
         - El coste de "replanificado" es mayor que el tiempo de espera.
         - Los recursos del procesador no se necesitan para otros procesos.

Sincronización (IV): implementación

- DECISIONES:
  - Velocidad vs. Flexibilidad.
  - ¿Qué implementar en *hardware?*: normalmente, sólo operaciones atómicas.
  - Un mismo punto de sincronización se puede acceder de diferentes maneras a lo largo del tiempo:
    - Grado de "contención" (nº de procesos "compitiendo").
    - Latencia baja vs. *throughput* alto.
    - Diferentes algoritmos más adecuados para cada caso.
  - *Software-Hardware*:
    - Los mecanismos suministrados por el *hardware* determinan lo que los algoritmos pueden usar.
    - A la hora de diseñar el *hardware* se tiene en cuenta lo que el `software` demanda.
  - Evaluación: como en todo, con programas de prueba.

Sincronización (V): implementación

- **LOCKs**:
  - Primera versión:

```
lock: ld    .R1, /dir_cerrojo
      cmp   .R1, #0
      bnz   $lock
      st    #1, /dir_cerrojo
      ret

unlock:  st   #0, /dir_cerrojo
      ret
```

Sincronización (VI): implementación

- **LOCKs**:
  - Segunda versión: con *test&set (t&s)*

```
   lock:          t&s   .R1, /
dir_cerrojo
               bnz   $lock
               ret

   unlock:     st    #0, /dir_cerrojo
               ret
```

    - Consideraciones de rendimiento: tráfico, invalidaciones, hambruna,...
    - Variantes: t&s con *backoff*, *test&test&set*

Sincronización (VII)

Test and Test and Set:

```
A:    while (LOAD(lock) = 1) do
                nothing;
      if (TEST&SET(lock) = 0)
                {
                  región crítica;
                }
      else goto A;
```

Versión mejorada:

```
      repeat
           while (LOAD(lock) = 1) do nothing;
      until (TEST&SET(lock) = 0))
```

+ : hace el "spin" en la cache

– : sigue habiendo mucho tráfico si hay mucho procesos ejecutando el *t&s*

Sincronización (VIII)

Test and Set con *Backoff:*

- Cuando hay fallo, tarda "un poco" antes de reintentarlo:
  - retardo constante.
  - retardo exponencial.

\+ : mucho menos tráfico.

– : el retardo exponencial puede producir hambruna en caso de mucha "contención": los recién llegados, menos espera: y sin embargo, el retardo exponencial produce los mejores resultados en la práctica.

Sincronización (IX)

- Tercera versión: con load-locked + store-conditional (RISCs)

  `ll`: lee la variable en un registro

  `sc`: intenta almacenar la variable en la posición de Memoria si y sólo si ningún otro procesador ha escrito en la variable desde que ejecutó `ll`.

```
lock:     ll    .R1, /dir_cerrojo /* LL dir a R1  */
          bnz   .R1, $lock        /* ¿cerrado?    */
          sc    .R2, /dir_cerrojo /* SC R2 en dir */
          beqz  $lock             /* si fallo, again*/
          ret
unlock:   st #0, /dir_cerrojo
          ret
```

otras versiones: *ticket lock, array-based queing*: ambas logran FIFO.

Sincronización (y X)

- ***Barriers:***
  - Se pueden, y suelen, implementar en software utilizando LOCKS, contadores e instrucciones atómicas.
  - Las **implementaciones en *hardware*** no son habituales hoy en día, al menos en las máquinas basadas en buses:
    - Líneas adicionales en el bus.
    - Útiles cuando las barreras son globales y frecuentes.
    - Difícil de implementar cuando sólo implica a un subconjunto de los procesadores: peor cuando hay varios procesos por procesador.
    - Difícil cambiar dinámicamente el número e identidad de los procesos.
  - **Implementaciones (*software*):**
    - Centralizado:
      - Árbol de combinación: disminuye el grado de "contienda".
      - Con diseminación.
      - Con "*tournament*".