Parallel Processing

Ch 17  [Sta10]

Taxonomy
SMP
Cache Coherence – MESI Protocol
NUMA and CC-NUMA
Vector Computation
Parallel Processor Architectures
Flynn’s taxonomy from 1972

Processor Organizations

- Single Instruction, Single Data Stream (SISD)
- Single Instruction, Multiple Data Stream (SIMD)
- Multiple Instruction, Single Data Stream (MISD)
- Multiple Instruction, Multiple Data Stream (MIMD)

Uniprocessor

- Vector Processor
- Array Processor

Shared Memory (tightly coupled)

Distributed Memory (loosely coupled)

Clusters

Symmetric Multiprocessor (SMP)

Nonuniform Memory Access (NUMA)

(Sta10 Fig 17.1)
Processor Organization Structures

(a) SISD

CU → PU → DS → MU

(b) SIMD (with distributed memory)

CU → PU₁ → DS → LM₁
   ...  
   CU → PUₙ → DS → LMₙ

(c) MIMD (with shared memory)

CU₁ → PU₁ → DS → Shared Memory
   ...  
   CUₙ → PUₙ → DS → Shared Memory

(d) MIMD (with distributed memory)

CU₁ → PU₁ → DS → LM₁
   ...  
   CUₙ → PUₙ → DS → LMₙ

CU = control unit  SISD = single instruction, 
IS = instruction stream  single data stream
PU = processing unit  SIMD = single instruction, 
DS = data stream  multiple data stream
MU = memory unit  MIMD = multiple instruction, 
LM = local memory  multiple data stream

(Sta10 Fig 17.2)
Parallel Processor Architectures

- Single instruction, single data stream – **SISD**
  - Uniprocessor

- Single instruction, multiple data stream – **SIMD**
  - Vector and array processors
  - Single machine instruction controls simultaneous execution
  - Each instruction executed on different set of data by different processors

- Multiple instruction, single data stream – **MISD**
  - Sequence of data transmitted to set of processors
  - Each processor executes different instruction sequence
  - Not used

- Multiple instruction, multiple data stream - **MIMD**
  - Set of processors simultaneously execute different instruction sequences on different sets of data
  - SMPs, clusters and NUMA systems
Multiple instruction, multiple data stream - MIMD

- Differences in processor communication
- Symmetric Multiprocessor (SMP)
  - Tightly coupled – communication via shared memory
  - Share single memory or pool, shared bus to access memory
  - Memory access time of a given memory location is approximately the same for each processor
- Non-uniform memory access (NUMA)
  - Tightly coupled – communication via shared memory
  - Access times to different regions of memory may differ
- Clusters
  - Loosely coupled – no shared memory
  - Communication via fixed path or network connections
  - Collection of independent uniprocessors or SMPs
SMP – Symmetric Multiprocessor

- Two or more similar processors of comparable capacity
- All processors can perform the same functions (hence symmetric)
- Connected by a bus or other internal connection
- Share same memory and I/O
- I/O access to same devices through same or different channels
- Memory access time is approximately the same for each processor
- System controlled by integrated operating system
  - providing interaction between processors
  - Interaction at job, task, file and data element levels
SMP – Advantages

- **Performance**
  - Only if some work can be done in parallel

- **Availability**
  - More processors to do the same functions
  - Failure of a single processor does not halt the system

- **Incremental growth**
  - Increase performance by adding additional processors
  - Is there a limit on this? Bus becomes serious bottleneck?

- **Scaling**
  - Different computers can have different number of processors
  - Vendors can offer range of products based on number of processors
Multiprogramming vs multiprocessing (Moniajo)

(a) Interleaving (multiprogramming, one processor)

(b) Interleaving and overlapping (multiprocessing; multiple processors)

Blocked

Running

(Sta10 Fig 17.3)
Multiprocessor Organization

- Processors
  - Two or more
  - Self-contained
  - Additionally, may have private memory and/or I/O channels
- Multiport memory
  - Shared memory
  - Simultaneous access to separate blocks
- Interconnection
  - Time shared bus (most common)
Example: SMP Organization

(No multicore)

(Sta10 Fig 17.5)
Time-shared bus

Advantages

- Simplicity
  - Addressing, arbitration and time-sharing logic same as in uniprocessor system
- Flexibility
  - Expand by attaching more processors to the bus
- Reliability
  - Bus is passive, failure of attached device should not cause failure of the whole

Disadvantages

- Performance limited by bus cycle time
- Each processor should have local cache
  - Reduce number of bus accesses
- Leads to problems with cache coherence
  - Solved in hardware - see later
New Requirements to Operating System

- Simultaneous concurrent processes
  - Reentrant OS routines (only local data, no conc. problems)
  - OS data structure synchronization – avoid deadlocks etc.

- Scheduling
  - On SMP any processor may execute scheduler at any time

- Synchronization
  - Controlled access to shared resources

- Memory management
  - Use parallel access options

- Reliability and fault tolerance
  - Graceful degradation in the face of single processor failure
**IBM z990**

**Multiprocessor Structure**

Dual-core processor chip
- CISC superscalar
- 256-kB L1 instruction and a 256-kB L1 data cache
- Point-to-point conn. to L2

L2 cache, each 32 MB
- Clusters of five
- Each cluster supports eight processors and access to entire main memory space

**System control element (SCE)** (one of the L2 caches)
- Arbitrates system comm.
- Maintains cache coherence

**Memory card**
- Each 32 GB, Maximum 8
- Interconnect to MSC via synchronous memory interfaces (SMIs)

**Memory bus adapter (MBA)**
- Interface to I/O channels, go directly to L2 cache
Cache and Data Consistency

- Multiple processors with their own caches
  - Multiple copies of same data in different caches
  - Concurrent modification of the same data
- Could result in an inconsistent view of memory
  - Inconsistency – the values in caches are different
- Write back policy
  - Write first to local cache and only later to memory
- Write through policy
  - The value is written to memory when changed
  - Other caches must monitor memory traffic
- Solution: maintain cache coherence
  - Keep recently used variables in appropriate cache(s), while maintaining the consistency of shared variables!
Software Solutions for Cache Coherence

- Compiler and operating system deal with problem
- Overhead transferred to compile time
- Design complexity transferred from hardware to software
- However, software tends to make conservative decisions
  - Inefficient cache utilization - do not cache shared variables
- Analyze code to determine safe periods for caching shared variables
Hardware Solutions for Cache Coherence

- Dynamic recognition of potential problems at run time
- More efficient use of cache, transparent to programmer
- Directory protocols
  - Collect and maintain information about copies of data in cache
  - Directory stored in main memory
  - Requests are checked against directory
  - Creates central bottleneck
  - Effective in large scale systems with complex interconnections
- Snoopy protocols
  - Distribute cache coherence responsibility to all cache controllers
  - Cache recognizes that a line is shared
  - Updates announced to other caches
  - Suited to bus based multiprocessor
Snoopy Cache Protocols

- **Write-Invalidate**
  - Multiple readers, one writer
  - Write request invalidates that line in all other caches
  - Writing processor gains exclusive (cheap) access until line required by another processor
  - Used in Pentium II and PowerPC systems
  - State of every line marked as modified, exclusive, shared or invalid (MESI)

- **Write-Update**
  - Multiple readers and writers
  - Updated word is distributed to all other processors

- Some systems use an adaptive mixture of both solutions
MESI Protocol

- Four states (two bits per tag)
  - **Modified**: modified cache line (only in this cache)
  - **Exclusive**: only in this cache, but the same as memory
  - **Shared**: same as memory, may be in other caches
  - **Invalid**: line does not contain valid data

<table>
<thead>
<tr>
<th></th>
<th><strong>M</strong> Modified</th>
<th><strong>E</strong> Exclusive</th>
<th><strong>S</strong> Shared</th>
<th><strong>I</strong> Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>This cache line valid?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>The memory copy is...</td>
<td>out of date</td>
<td>valid</td>
<td>valid</td>
<td>—</td>
</tr>
<tr>
<td>Copies exist in other caches?</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>A write to this line...</td>
<td>does not go to bus</td>
<td>does not go to bus</td>
<td><strong>goes to bus and updates cache</strong></td>
<td><strong>goes directly to bus</strong></td>
</tr>
</tbody>
</table>
MESI State Transition Diagram

(a) Line in cache at initiating processor

- Invalid → Shared (RMS)
- Shared → Invalid (RH)
- Modified → Exclusive (WH)
- Exclusive → Modified (WH)

(b) Line in snooping cache

- Invalid → Shared (SHW)
- Shared → Invalid (SHW)
- Modified → Exclusive (SHR)
- Exclusive → Modified (SHR)

<table>
<thead>
<tr>
<th>RS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RH</td>
<td>Read hit</td>
</tr>
<tr>
<td>RMS</td>
<td>Read miss, shared</td>
</tr>
<tr>
<td>RME</td>
<td>Read miss, exclusive</td>
</tr>
<tr>
<td>WH</td>
<td>Write hit</td>
</tr>
<tr>
<td>WM</td>
<td>Write miss</td>
</tr>
<tr>
<td>SHR</td>
<td>Snoop hit on read</td>
</tr>
<tr>
<td>SHW</td>
<td>Snoop hit on write or</td>
</tr>
<tr>
<td></td>
<td>read-with-intent-to-modify</td>
</tr>
<tr>
<td></td>
<td>Dirty line copyback</td>
</tr>
<tr>
<td></td>
<td>Invalidate transaction</td>
</tr>
<tr>
<td></td>
<td>Read-with-intent-to-modify</td>
</tr>
<tr>
<td></td>
<td>Cache line fill</td>
</tr>
</tbody>
</table>

(Sta10 Fig 17.6)
MESI Protocol – state transitions

- **Read Miss** – generates SHR (snoop hit on read) to others
  - Not in any cache – simply read
  - Exclusive in some cache – SHR: exclusive ‘owner’ indicates sharing and changes the state of its own cache line to shared
  - Shared in some caches – SHR: each signals about the sharing
  - Modified on some cache – SHR: memory read blocked, the modified content comes to memory and this cache from the other cache, which also changes the state of that line to shared

- **Read Hit**

- **Write Miss** – generates SHW (snoop on writes) to others

- **Write Hit**
Multithreading

- Process
  - Resources
  - Scheduling
    - Process switch
  - 1 or more executable threads
    - Private stack, shared resources (e.g., memory)
    - Thread switch
      - Kernel level threads are OS controlled
      - User level threads are process controlled

- Thread level parallelism
Multithreading Types

- Interleaved multithreading
  - “Fine-grained multithreading”,
  - Separate hw context (registers) for each thread
  - Alternate thread execution after each clock cycle

- Blocked multithreading
  - “Coarse grained multithreading”
  - Switch thread when previous one is blocked (e.g., cache miss)

- Simultaneous multithreading (SMT)
  - Intel “hyperthreading”
  - Instructions from multiple threads in superscalar processor
  - Many contexts (register sets) at use simultaneously

- Chip multiprocessing
  - “Multicore”
  - Many complete cpu’s on one chip, not necessarily symmetric!
  - cpu’s (cores) may share some L2 or L3 cache
Multithreading Types

(e) interleaved multithreading superscalar

(f) blocked multithreading superscalar

(g) VLIW

(h) interleaved multithreading VLIW

(i) blocked multithreading VLIW

(j) simultaneous multithreading (SMT)

(k) chip multiprocessor (multicore)

Fig 17.8 [Sta10]
Clusters

- Cluster is a group of interconnected nodes
  - Each node is a whole computer
  - Nodes work together as unified resource
  - Illusion of being one machine
  - Commonly for server applications
- Benefits:
  - Scalability
  - High availability
  - Load balancing
  - Superior price/performance

Example: blades in one or more chassis
blade computer

TKTL cluster “Ukko” (Sta10 Fig 17.12)

15 chassis
240 blades

Each blade has 2 cpus’s with 4-cores, each one with 2-hyperthreads

Example: blades in one or more chassis blade (kehikon korttipalvelin)
Cluster Middleware

1. Unified image to user
   - Single system image
2. Single point of entry
3. Single file hierarchy
4. Single control point
5. Single virtual networking
6. Single memory space
7. Single job management system
8. Single user interface
9. Single I/O space
10. Single process space
11. Checkpointing
12. Process migration
What is NUMA?

- **SMP**
  - Identical processors with uniform memory access (UMA) to shared memory
    - All processors can access all parts of the memory
    - Identical access time all memory regions for all processors

- **Clusters**
  - Interconnected computers with NO shared memory

- **NUMA Non-Uniform Memory Access**
  - All processors can access all parts of the memory
  - Access times to different regions are different for different processors
  - Cache-Coherent NUMA (CC-NUMA) maintains cache coherence among caches of various processors
  - Maintain transparent system wide memory
CC-NUMA Organization

- Independent SMP nodes
- Single addressable memory
- Unique system wide address
- Cache coherence based on directory

(Sta10 Fig 17.13)
CC-NUMA Memory Access

- Each processor has local L1 & L2 cache and main memory
- Nodes connected by some networking facility
- Each processor sees single addressable memory space

Memory request order:
- L1 cache (local to processor)
- L2 cache (local to processor)
- Main memory (local to node)
- Remote memory (in other nodes)
  - Delivered to requesting (local to processor) cache
  - Needs to maintain cache coherence with other processor’s caches

- Automatic and transparent
Vector Computation

- SIMD instructions
  - One dimensional data
- When needed?
- When used in HLL code?
  - Vector lengths determined by application
- How to implement in HW?
  - Vector length determined by HW
Matrix Multiplication with Vectors

Matrix multiplication: $C = A \times B$

- Compiler technology for vector machines is outside course objectives
- Vector register length varies
  - 8 64-bit FP registers?

```
serial
  do 100 i = 1,n
    do 100 j = 1, n
      c(i,j) = 0.0
      do 100 k = 1,n
        c(i,j) = c(i,j) + a(i,k) + b(k,j)
      100 continue
```

```
vector
  do 100 i = 1,n
    c(i,j) = 0.0 (j=1,n)
    do 100 k = 1,n
      c(i,j) = c(i,j) + a(i,k) + b(k,j) (j=1,n)
    100 continue
```

```
... vload vr1, a[k] ; 8 values at a time?
vload vr2, b[k]
vadd vr3, vr1, vr2
....
vstore vr3, c[k]
```
Vector Operation Implementation

- Usually vector ops from registers, not from memory

(a) Pipelined ALU (Sta10 Fig 17.16)

(b) Parallel ALUs
**Vector Pipeline Implementation**

- **Chaining**
  - Vector multiply-add in one vector op
  - \( C = sc \times A + B \)

(Sta10 Fig 17.17)
Parallel Processing Summary

- Paralle processing classification
- SMP, NUMA, CC-NUMA
- SMP: Shared memory becomes bottleneck with many processors
- CC-NUMA
  - Performance suffers if too much remote memory access
  - Need **good temporal and spatial locality of software** with
    - L1 & L2 cache design to reduce all memory access
    - Virtual memory management move pages to nodes that use them most
  - Not truly transparent memory access
    - Page allocation, process allocation and load balancing changes needed
- Vector instructions
Review Questions / Kertauskysymyksiä

- Cache coherence and MESI protocol
- Differences and similarities of SMP, NUMA and cluster
- When would you use vector operations?