Chapter 7
Multicores, Multiprocessors, and Clusters
Introduction

- Goal: connecting multiple computers to get higher performance
  - Multiprocessors
  - Scalability, availability, power efficiency
- Job-level (process-level) parallelism
  - High throughput for independent jobs
- Parallel processing program
  - Single program run on multiple processors
- Multicore microprocessors
  - Chips with multiple processors (cores)
Hardware and Software

- Hardware
  - Serial: e.g., Pentium 4
  - Parallel: e.g., quad-core Xeon e5345

- Software
  - Sequential: e.g., matrix multiplication
  - Concurrent: e.g., operating system

- Sequential/concurrent software can run on serial/parallel hardware
  - Challenge: making effective use of parallel hardware
What We’ve Already Covered

- §2.11: Parallelism and Instructions
  - Synchronization
- §3.6: Parallelism and Computer Arithmetic
  - Associativity
- §4.10: Parallelism and Advanced Instruction-Level Parallelism
- §5.8: Parallelism and Memory Hierarchies
  - Cache Coherence
- §6.9: Parallelism and I/O:
  - Redundant Arrays of Inexpensive Disks
Parallel Programming

- Parallel software is the problem
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it’s easier!
- Difficulties
  - Partitioning
  - Coordination
  - Communications overhead
Amdahl’s Law

- Sequential part can limit speedup

Example: 100 processors, 90× speedup?

\[ T_{\text{new}} = \frac{T_{\text{parallelizable}}}{100} + T_{\text{sequential}} \]

\[ \text{Speedup} = \frac{1}{(1 - F_{\text{parallelizable}}) + \frac{F_{\text{parallelizable}}}{100}} = 90 \]

Solving: \( F_{\text{parallelizable}} = 0.999 \)

Need sequential part to be 0.1% of original time
Scaling Example

- Workload: sum of 10 scalars, and $10 \times 10$ matrix sum
  - Speed up from 10 to 100 processors
- Single processor: Time $= (10 + 100) \times t_{\text{add}}$
- 10 processors
  - Time $= 10 \times t_{\text{add}} + 100/10 \times t_{\text{add}} = 20 \times t_{\text{add}}$
  - Speedup $= 110/20 = 5.5$ (55% of potential)
- 100 processors
  - Time $= 10 \times t_{\text{add}} + 100/100 \times t_{\text{add}} = 11 \times t_{\text{add}}$
  - Speedup $= 110/11 = 10$ (10% of potential)
- Assumes load can be balanced across processors
Scaling Example (cont)

- What if matrix size is 100 × 100?
- Single processor: Time = (10 + 10000) × t_{add}
- 10 processors
  - Time = 10 × t_{add} + 10000/10 × t_{add} = 1010 × t_{add}
  - Speedup = 10010/1010 = 9.9 (99% of potential)
- 100 processors
  - Time = 10 × t_{add} + 10000/100 × t_{add} = 110 × t_{add}
  - Speedup = 10010/110 = 91 (91% of potential)
- Assuming load balanced
Strong vs Weak Scaling

- **Strong scaling:** problem size fixed
  - As in example
- **Weak scaling:** problem size proportional to number of processors
  - 10 processors, $10 \times 10$ matrix
    - Time = $20 \times t_{\text{add}}$
  - 100 processors, $32 \times 32$ matrix
    - Time = $10 \times t_{\text{add}} + \frac{1000}{100} \times t_{\text{add}} = 20 \times t_{\text{add}}$
  - Constant performance in this example
Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks
  - Memory access time
    - UMA (uniform) vs. NUMA (nonuniform)
Example: Sum Reduction

- Sum 100,000 numbers on 100 processor UMA
  - Each processor has ID: 0 ≤ Pn ≤ 99
  - Partition 1000 numbers per processor
  - Initial summation on each processor
    \[ \text{sum}[Pn] = 0; \]
    \[ \text{for } (i = 1000*Pn; \]
    \[ \quad i < 1000*(Pn+1); i = i + 1) \]
    \[ \text{sum}[Pn] = \text{sum}[Pn] + A[i]; \]

- Now need to add these partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, ...
  - Need to synchronize between reduction steps
Example: Sum Reduction

```c
half = 100;
repeat
    synch();
    if (half%2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[half-1];
        /* Conditional sum needed when half is odd;
           Processor0 gets missing element */
    half = half/2; /* dividing line on who sums */
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```
Message Passing

- Each processor has private physical address space
- Hardware sends/receives messages between processors
Loosely Coupled Clusters

- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system
    - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, …
- High availability, scalable, affordable
- Problems
  - Administration cost (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP
Sum Reduction (Again)

- Sum 100,000 on 100 processors
- First distribute 100 numbers to each
  - The do partial sums
    
    ```
    sum = 0;
    for (i = 0; i<1000; i = i + 1)
      sum = sum + AN[i];
    ```

- Reduction
  - Half the processors send, other half receive and add
  - The quarter send, quarter receive and add, ...
Sum Reduction (Again)

- Given send() and receive() operations

```plaintext
limit = 100; half = 100; /* 100 processors */
repeat
    half = (half+1)/2; /* send vs. receive dividing line */
    if (Pn >= half && Pn < limit)
        send(Pn - half, sum);
    if (Pn < (limit/2))
        sum = sum + receive();
    limit = half; /* upper limit of senders */
until (half == 1); /* exit with final sum */
```

- Send/receive also provide synchronization
- Assumes send/receive take similar time to addition
Grid Computing

- Separate computers interconnected by long-haul networks
  - E.g., Internet connections
  - Work units farmed out, results sent back
- Can make use of idle time on PCs
  - E.g., SETI@home, World Community Grid
Multithreading

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads

- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed

- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn’t hide short stalls (e.g., data hazards)
Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches
Multithreading Example

Chapter 7 — Multicores, Multiprocessors, and Clusters — 20
Future of Multithreading

- Will it survive? In what form?
- Power considerations $\Rightarrow$ simplified microarchitectures
  - Simpler forms of multithreading
- Tolerating cache-miss latency
  - Thread switch may be most effective
- Multiple simple cores might share resources more effectively
Instruction and Data Streams

An alternate classification

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD:</td>
<td>SISD:</td>
<td>SIMD: SSE instructions of x86</td>
</tr>
<tr>
<td>Intel Pentium 4</td>
<td>SIMD:</td>
<td>Intel Xeon e5345</td>
</tr>
<tr>
<td>MISD:</td>
<td>No examples today</td>
<td></td>
</tr>
<tr>
<td>MIMD:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SPMD: Single Program Multiple Data

- A parallel program on a MIMD computer
- Conditional code for different processors
SIMD

- Operate elementwise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications
Vector Processors

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to MIPS
  - $32 \times 64$-element registers (64-bit elements)
  - Vector instructions
    - lv, sv: load/store vector
    - addv.d: add vectors of double
    - addvs.d: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth
Example: DAXPY (Y = a × X + Y)

- Conventional MIPS code
  ```
  l.d  $f0,a($sp)      ;load scalar a
  addiu r4,$s0,#512    ;upper bound of what to load
  loop: l.d  $f2,0($s0) ;load x(i)
         mul.d $f2,$f2,$f0 ;a × x(i)
         l.d  $f4,0($s1)  ;load y(i)
         add.d $f4,$f4,$f2 ;a × x(i) + y(i)
         s.d  $f4,0($s1)  ;store into y(i)
         addiu $s0,$s0,#8 ;increment index to x
         addiu $s1,$s1,#8 ;increment index to y
         subu  $t0,r4,$s0 ;compute bound
         bne   $t0,$zero,loop ;check if done
  ```

- Vector MIPS code
  ```
  l.d  $f0,a($sp) ;load scalar a
  lv   $v1,0($s0) ;load vector x
  mulvs.d $v2,$v1,$f0 ;vector-scalar multiply
  lv   $v3,0($s1) ;load vector y
  addv.d $v4,$v2,$v3 ;add y to product
  sv   $v4,0($s1) ;store the result
  ```
Vector vs. Scalar

- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
  - Better match with compiler technology
History of GPUs

- Early video cards
  - Frame buffer memory with address generation for video output

- 3D graphics processing
  - Originally high-end computers (e.g., SGI)
  - Moore’s Law $\Rightarrow$ lower cost, higher density
  - 3D graphics cards for PCs and game consoles

- Graphics Processing Units
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, rasterization
Graphics in the System

Chapter 7 — Multicores, Multiprocessors, and Clusters — 28
GPU Architectures

- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth

- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code

- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)
Example: NVIDIA Tesla

Chapter 7 — Multicores, Multiprocessors, and Clusters — 30
Example: NVIDIA Tesla

- Streaming Processors
  - Single-precision FP and integer units
  - Each SP is fine-grained multithreaded
- Warp: group of 32 threads
  - Executed in parallel, SIMD style
    - 8 SPs
      - × 4 clock cycles
  - Hardware contexts for 24 warps
    - Registers, PCs, …
Classifying GPUs

- Don’t fit nicely into SIMD/MIMD model
  - Conditional execution in a thread allows an illusion of MIMD
    - But with performance degradation
    - Need to write general purpose code with care

<table>
<thead>
<tr>
<th></th>
<th>Static: Discovered at Compile Time</th>
<th>Dynamic: Discovered at Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction-Level</td>
<td>VLIW</td>
<td>Superscalar</td>
</tr>
<tr>
<td>Parallelism</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data-Level Parallelism</td>
<td>SIMD or Vector</td>
<td>Tesla Multiprocessor</td>
</tr>
</tbody>
</table>

Chapter 7 — Multicores, Multiprocessors, and Clusters — 32
Interconnection Networks

- Network topologies
  - Arrangements of processors, switches, and links

- Bus
- Ring
- 2D Mesh
- N-cube (N = 3)
- Fully connected

Chapter 7 — Multicores, Multiprocessors, and Clusters — 33
Multistage Networks

a. Crossbar

b. Omega network

c. Omega network switch box

Chapter 7 — Multicores, Multiprocessors, and Clusters — 34
Network Characteristics

- **Performance**
  - Latency per message (unloaded network)
  - Throughput
    - Link bandwidth
    - Total network bandwidth
    - Bisection bandwidth
  - Congestion delays (depending on traffic)
- **Cost**
- **Power**
- **Routability in silicon**
Parallel Benchmarks

- Linpack: matrix linear algebra
- SPECrate: parallel run of SPEC CPU programs
  - Job-level parallelism
- SPLASH: Stanford Parallel Applications for Shared Memory
  - Mix of kernels and applications, strong scaling
- NAS (NASA Advanced Supercomputing) suite
  - Computational fluid dynamics kernels
- PARSEC (Princeton Application Repository for Shared Memory Computers) suite
  - Multithreaded applications using Pthreads and OpenMP
Code or Applications?

- Traditional benchmarks
  - Fixed code and data sets
- Parallel programming is evolving
  - Should algorithms, programming languages, and tools be part of the system?
  - Compare systems, provided they implement a given application
  - E.g., Linpack, Berkeley Design Patterns
- Would foster innovation in approaches to parallelism
Modeling Performance

- Assume performance metric of interest is achievable GFLOPs/sec
  - Measured using computational kernels from Berkeley Design Patterns
- Arithmetic intensity of a kernel
  - FLOPs per byte of memory accessed
- For a given computer, determine
  - Peak GFLOPS (from data sheet)
  - Peak memory bytes/sec (using Stream benchmark)
Roofline Diagram

Attainable GPLOPs/sec
= Max ( Peak Memory BW × Arithmetic Intensity, Peak FP Performance )
Comparing Systems

- Example: Opteron X2 vs. Opteron X4
  - 2-core vs. 4-core, 2× FP performance/core, 2.2GHz vs. 2.3GHz
  - Same memory system

To get higher performance on X4 than X2
- Need high arithmetic intensity
- Or working set must fit in X4’s 2MB L-3 cache
Optimizing Performance

- Optimize FP performance
  - Balance adds & multiplies
  - Improve superscalar ILP and use of SIMD instructions
- Optimize memory usage
  - Software prefetch
    - Avoid load stalls
  - Memory affinity
    - Avoid non-local data accesses
Optimizing Performance

- Choice of optimization depends on arithmetic intensity of code

- Arithmetic intensity is not always fixed
  - May scale with problem size
  - Caching reduces memory accesses
    - Increases arithmetic intensity
Four Example Systems

2 × quad-core
Intel Xeon e5345
(Clovertown)

2 × quad-core
AMD Opteron X4 2356
(Barcelona)
Four Example Systems

2 × oct-core
Sun UltraSPARC
T2 5140 (Niagara 2)

2 × oct-core
IBM Cell QS20

Chapter 7 — Multicores, Multiprocessors, and Clusters — 44
And Their Rooflines

- Kernels
  - SpMV (left)
  - LBHMD (right)
- Some optimizations change arithmetic intensity
- x86 systems have higher peak GFLOPs
  - But harder to achieve, given memory bandwidth

Chapter 7 — Multicores, Multiprocessors, and Clusters — 45
Performance on SpMV

- Sparse matrix/vector multiply
  - Irregular memory accesses, memory bound
- Arithmetic intensity
  - 0.166 before memory optimization, 0.25 after

- Xeon vs. Opteron
  - Similar peak FLOPS
  - Xeon limited by shared FSBs and chipset
- UltraSPARC/Cell vs. x86
  - 20 – 30 vs. 75 peak GFLOPs
  - More cores and memory bandwidth
Performance on LBMHD

- Fluid dynamics: structured grid over time steps
  - Each point: 75 FP read/write, 1300 FP ops
- Arithmetic intensity
  - 0.70 before optimization, 1.07 after

- Opteron vs. UltraSPARC
  - More powerful cores, not limited by memory bandwidth
- Xeon vs. others
  - Still suffers from memory bottlenecks
Achieving Performance

- Compare naïve vs. optimized code
  - If naïve code performs well, it’s easier to write high performance code for the system

<table>
<thead>
<tr>
<th>System</th>
<th>Kernel</th>
<th>Naïve GFLOPs/sec</th>
<th>Optimized GFLOPs/sec</th>
<th>Naïve as % of optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon</td>
<td>SpMV</td>
<td>1.0</td>
<td>1.5</td>
<td>64%</td>
</tr>
<tr>
<td></td>
<td>LBMHD</td>
<td>4.6</td>
<td>5.6</td>
<td>82%</td>
</tr>
<tr>
<td>AMD Opteron X4</td>
<td>SpMV</td>
<td>1.4</td>
<td>3.6</td>
<td>38%</td>
</tr>
<tr>
<td></td>
<td>LBMHD</td>
<td>7.1</td>
<td>14.1</td>
<td>50%</td>
</tr>
<tr>
<td>Sun UltraSPARC T2</td>
<td>SpMV</td>
<td>3.5</td>
<td>4.1</td>
<td>86%</td>
</tr>
<tr>
<td></td>
<td>LBMHD</td>
<td>9.7</td>
<td>10.5</td>
<td>93%</td>
</tr>
<tr>
<td>IBM Cell QS20</td>
<td>SpMV</td>
<td>Naïve code not feasible</td>
<td>6.4</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>LBMHD</td>
<td>Naïve code not feasible</td>
<td>16.7</td>
<td>0%</td>
</tr>
</tbody>
</table>
Fallacies

- Amdahl’s Law doesn’t apply to parallel computers
  - Since we can achieve linear speedup
  - But only on applications with weak scaling
- Peak performance tracks observed performance
  - Marketers like this approach!
  - But compare Xeon with others in example
  - Need to be aware of bottlenecks
Pitfalls

- Not developing the software to take account of a multiprocessor architecture
  - Example: using a single lock for a shared composite resource
    - Serializes accesses, even if they could be done in parallel
    - Use finer-granularity locking
Concluding Remarks

- Goal: higher performance by using multiple processors

- Difficulties
  - Developing parallel software
  - Devising appropriate architectures

- Many reasons for optimism
  - Changing software and application environment
  - Chip-level multiprocessors with lower latency, higher bandwidth interconnect

- An ongoing challenge for computer architects!