Shared Memory and Distributed Multiprocessing

Bhanu Kapoor, Ph.D.
The Saylor Foundation
Issue with Parallelism

• 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?
  – \( T_{\text{new}} = \frac{T_{\text{parallelizable}}}{100} + T_{\text{sequential}} \)

  – 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 = $(10 + 100) \times t_{add}$
• 10 processors
  – Time = $10 \times t_{add} + \frac{100}{10} \times t_{add} = 20 \times t_{add}$
  – Speedup = $\frac{110}{20} = 5.5$ (55% of potential)
• 100 processors
  – Time = $10 \times t_{add} + \frac{100}{100} \times t_{add} = 11 \times t_{add}$
  – Speedup = $\frac{110}{11} = 10$ (10% of potential)
• Assumes load can be balanced across processors
Scaling Example (cont)

- What if matrix size is $100 \times 100$?
- Single processor: $\text{Time} = (10 + 10000) \times t_{\text{add}}$
- 10 processors
  - $\text{Time} = 10 \times t_{\text{add}} + \frac{10000}{10} \times t_{\text{add}} = 1010 \times t_{\text{add}}$
  - Speedup = $\frac{10010}{1010} = 9.9$ (99% of potential)
- 100 processors
  - $\text{Time} = 10 \times t_{\text{add}} + \frac{10000}{100} \times t_{\text{add}} = 110 \times t_{\text{add}}$
  - Speedup = $\frac{10010}{110} = 91$ (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 × 10 matrix
    • Time = 20 × t_{add}
  – 100 processors, 32 × 32 matrix
    • Time = 10 × t_{add} + 1000/100 × t_{add} = 20 × t_{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
    \[
    \text{sum}[\text{Pn}] = 0; \\
    \text{for } (i = 1000*\text{Pn}; \\
    \quad i < 1000*(\text{Pn}+1); i = i + 1) \\
    \text{sum}[\text{Pn}] = \text{sum}[\text{Pn}] + A[i];
    \]

• 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

```
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 1000 numbers to each
  – The do partial sums
    
    \[
    \text{sum} = 0;
    \text{for (i = 0; i<1000; i = i + 1)} \quad \text{sum} = \text{sum} + \text{AN[i]};
    \]

• 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

  ```
  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
Cache Coherence Problem

• Suppose two CPU cores share a physical address space
  – Write-through caches

<table>
<thead>
<tr>
<th>Time step</th>
<th>Event</th>
<th>CPU A’s cache</th>
<th>CPU B’s cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>CPU A writes 1 to X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Coherence Defined

- Informally: Reads return most recently written value
- Formally:
  - P writes X; P reads X (no intervening writes) \(\Rightarrow\) read returns written value
  - \(\text{P}_1\) writes X; \(\text{P}_2\) reads X (sufficiently later) \(\Rightarrow\) read returns written value
    - c.f. CPU B reading X after step 3 in example
  - \(\text{P}_1\) writes X, \(\text{P}_2\) writes X \(\Rightarrow\) all processors see writes in the same order
    - End up with the same final value for X
Memory Consistency

• When are writes seen by other processors
  – “Seen” means a read returns the written value
  – Can’t be instantaneously

• Assumptions
  – A write completes only when all processors have seen it
  – A processor does not reorder writes with other accesses

• Consequence
  – P writes X then writes Y
    $\Rightarrow$ all processors that see new Y also see new X
  – Processors can reorder reads, but not writes
Multiprocessor Caches

(a) Multiprocessor cache architecture
Multiprocessor Caches

• Caches provide [for shared items]
  – Migration
  – Replication

• Migration Reduces
  – Latency
  – Bandwidth demands

• Replication reduces
  – Latency
  – Contention for a read of shared item