Challenges to Obtaining Good Parallel Processing Performance

• Outline:
  • **Coverage** or extent of parallelism in algorithm
    - Amdahl’s Law
  • **Granularity** of partitioning among processors
    - Communication cost and load balancing
  • **Locality** of computation and communication
    - Communication between processors or between processors and their memories

• **Coverage:** The Parallel Processing Challenge of Finding Enough Parallelism

  **Amdahl’s Law:**
  
  - The parallel speedup of any program is limited by the time needed for any sequential portions of the program to be completed.

  - For example, if a program runs for ten hours on a single core and has a sequential (nonparallelizable) portion that takes one hour, then no matter how many cores are devoted to the program, it will never go faster than one hour.
• Amdahl’s law: If $s$ is the execution time for inherently sequential computations, the speedup is limited by

$$speedup(p) = \frac{time(1)}{time(p)} = \frac{time(1)}{s + (parallel\_time(p))} \leq \frac{time(1)}{s}$$

• If 20% of the sequential execution time is in sequential regions, the speedup is limited to 5 independent of the number of processors.

Even if parallel part speeds up perfectly, performance is limited by sequential part

![Diagram showing time breakdown and speedup calculation]

- Speedup = old running time / new running time
  - = 100 seconds / 60 seconds
  - = 1.67
  - (parallel version is 1.67 times faster)
**Granularity: The Parallel Processing Challenge of Overhead caused by Parallelism**

- Given enough parallel work, this is the biggest barrier to getting desired speedup.

- Parallelism overheads include:
  - Cost of starting a thread or process
  - Cost of communicating shared data
  - Cost of synchronizing

- Each of these can cost several milliseconds (=millions of flops) on some systems.

- Tradeoff: Algorithm needs sufficiently large units of work to run fast in parallel (i.e. large granularity), but not so large that there is not enough parallel work.

**I/O Time vs. CPU Time**

- Input/Output Time includes both the Memory System and Bus/Network System.

- The rate of improvement of I/O is much slower than that of the CPU.

<table>
<thead>
<tr>
<th>Year</th>
<th>CPU time</th>
<th>I/O time</th>
<th>Elapsed time</th>
<th>% I/O time</th>
</tr>
</thead>
<tbody>
<tr>
<td>now</td>
<td>90s</td>
<td>10s</td>
<td>100s</td>
<td>10%</td>
</tr>
<tr>
<td>+2</td>
<td>45s</td>
<td>10s</td>
<td>55s</td>
<td>18%</td>
</tr>
<tr>
<td>+4</td>
<td>23s</td>
<td>10s</td>
<td>33s</td>
<td>31%</td>
</tr>
<tr>
<td>+6</td>
<td>11s</td>
<td>10s</td>
<td>21s</td>
<td>47%</td>
</tr>
</tbody>
</table>
• Exponentially growing gaps are occurring between:
  o Floating point time (CPU processing speed) and
  o Memory BandWidth (Transmission Speed of Memory) and
  o Memory Latency (Startup Time of Memory Transmission)

Floating Point Time << 1/Memory Bandwidth << Memory Latency Time

<table>
<thead>
<tr>
<th></th>
<th>Annual increase</th>
<th>Typical value in 2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-chip floating-point performance</td>
<td>59%</td>
<td>4 GFLOP/s</td>
</tr>
<tr>
<td>Memory bus bandwidth</td>
<td>23%</td>
<td>1 GWord/s = 0.25 word/flop</td>
</tr>
<tr>
<td>Memory latency</td>
<td>5.5%</td>
<td>70 ns = 280 FP ops = 70 loads</td>
</tr>
</tbody>
</table>
Exponentially growing gaps are also occurring between:

- Floating point time (CPU processing speed) and
- Network BandWidth (Transmission Speed of Network) and
- Network Latency (Startup Time of Network Transmission)

Floating Point Time $<< \frac{1}{\text{Network Bandwidth}} << \text{Network Latency Time}$

<table>
<thead>
<tr>
<th></th>
<th>Annual increase</th>
<th>Typical value in 2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Bandwidth</td>
<td>26%</td>
<td>65 MWord/s $= 0.03$ word/flop</td>
</tr>
<tr>
<td>Network latency</td>
<td>15%</td>
<td>5 ms $= 20K$ FP ops</td>
</tr>
</tbody>
</table>

Note that for both Memory and Network, Latency (not bandwidth) is the weaker link.

- This means that it is better to use Larger Chunk Sizes (Larger Granularity)
- Better to Retrieve (from Memory) or Transmit (over the Network) a small number of large blocks, rather than a large number of small blocks.
• However, there is a Tradeoff between using larger Granularity and Locality
  o CPU Performance improves much faster than RAM Memory Performance

  ![Graph showing Performance over time for CPU and DRAM]

  “Moore’s Law”
  Processor-Memory Performance Gap: (grows 50% / year)
  CPU 60%/yr.
  DRAM 7%/yr.

  o So Memory Hierarchies are Used to Provide Cost-Performance Effectiveness
    o Small Memories are Fast, but Expensive; Large Memories are Cheap, but Slow
Locality (location of the data in the Mem Hierarchy) Substantially Impacts Performance

- Keeping active Working Set in upper levels improves performance
  - But this means we need to use finer granularity (many smaller blocks)

Example: Intel Pentium4
- L1 cache: 3 cycles = 1.64 ns for a 1.83 GHz CPU = 12 calculations
- L2 cache: 14 cycles = 7.65 ns for a 1.83 GHz CPU = 56 calculations
- RAM: 48 cycles = 26.2 ns for a 1.83 GHz CPU = 192 calculations
In parallel programming, communication considerations have the same importance as single core optimizations!
• Tiling can be used to Partition Task such that Memory Hierarchy is better Leveraged

• Challenge: Tradeoff in Granularity Size
  - From a BandWidth vs. Latency Point of View with Memory and Network:
    ➔ Want Larger Blocks because Latency is Slower than Bandwidth
  - From a Memory Locality Point of View:
    ➔ Want Smaller Blocks that will fit into Fastest (Smallest) Memory in Hierarchy
      Reduces Mem Access Times & Can make possible SuperLinear Speedup
Partitioning Should also Strive to Load Balance Tasks onto the Processors

The primary sources of inefficiency in parallel codes:

- Poor single processor performance
  - Typically in the memory system
- Too much parallelism overhead
  - Thread creation, synchronization, communication
- Load imbalance
  - Different amounts of work across processors
    - Computation and communication
    - Different speeds (or available resources) for the processors
      - Possibly due to load on the machine
- How to recognize load imbalance
  - Time spent at synchronization is high and is uneven across processors, but not always so simple ...

Load Imbalance is the Time that some processors in the system are idle due to:
  - Insufficient Parallelism
  - Unequal Size Tasks

Load Imbalance Exacerbates Synchronization Overhead
  - Slowest (Longest) Task or Processor holds up all other Tasks or Processors
Improving Real Performance

Peak Performance grows exponentially, a la Moore’s Law
- In 1990’s, peak performance increased 100x; in 2000’s, it will increase 1000x

But efficiency (the performance relative to the hardware peak) has declined
- was 40-50% on the vector supercomputers of 1990s
- now as little as 5-10% on parallel supercomputers of today

Close the gap through ...
- Mathematical methods and algorithms that achieve high performance on a single processor and scale to thousands of processors
- More efficient programming models and tools for massively parallel supercomputers
Much of the Performance is from Parallelism

- Bit-Level Parallelism
- Instruction-Level Parallelism
- Thread-Level Parallelism?