CS 426
Parallel Computing

Parallel Computing Platforms

Ozcan Ozturk
http://www.cs.bilkent.edu.tr/~ozturk/cs426/

Slides are adapted from "Introduction to Parallel Computing"
Topic Overview

- Implicit Parallelism: Trends in Microprocessor Architectures
- Limitations of Memory System Performance
- Dichotomy of Parallel Computing Platforms
- Communication Model of Parallel Platforms
- Physical Organization of Parallel Platforms
- Communication Costs in Parallel Machines
- Messaging Cost Models and Routing Mechanisms
- Mapping Techniques
- Case Studies
Scope of Parallelism

- Conventional architectures coarsely comprise of a processor, memory system, and the datapath.
- Each of these components present significant performance bottlenecks.
- Parallelism addresses each of these components in significant ways.
- Different applications utilize different aspects of parallelism - e.g., data intensive applications utilize high aggregate throughput, server applications utilize high aggregate network bandwidth, and scientific applications typically utilize high processing and memory system performance.
- It is important to understand each of these performance bottlenecks.
Implicit Parallelism: Trends in Microprocessor Architectures

- Microprocessor clock speeds have posted impressive gains over the past two decades (two to three orders of magnitude).
- Higher levels of device integration have made available a large number of transistors.
- The question of how best to utilize these resources is an important one.
- Current processors use these resources in multiple functional units and execute multiple instructions in the same cycle.
- The precise manner in which these instructions are selected and executed provides impressive diversity in architectures.
Pipelining and Superscalar Execution

- Pipelining overlaps various stages of instruction execution to achieve performance.
- At a high level of abstraction, an instruction can be executed while the next one is being decoded and the next one is being fetched.
- This is akin to an assembly line for manufacture of cars.
Pipelining and Superscalar Execution

- Pipelining, however, has several limitations.
- The speed of a pipeline is eventually limited by the slowest stage.
- For this reason, conventional processors rely on very deep pipelines (20 stage pipelines in state-of-the-art Pentium processors).
- However, in typical program traces, every 5-6th instruction is a conditional jump! This requires very accurate branch prediction.
- The penalty of a misprediction grows with the depth of the pipeline, since a larger number of instructions will have to be flushed.
Pipelining and Superscalar Execution

- One simple way of alleviating these bottlenecks is to use multiple pipelines.
- The question then becomes one of selecting these instructions.
Superscalar Execution: An Example

Example of a two-way superscalar execution of instructions.

1. load R1, @1000
2. load R2, @1000
3. add R1, @1004
4. add R2, @100C
5. add R1, R2
6. store R1, @2000

(i)   (ii)   (iii)

(a) Three different code fragments for adding a list of four numbers.

<table>
<thead>
<tr>
<th>Instruction cycles</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF: Instruction Fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID: Instruction Decode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OF: Operand Fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E: Instruction Execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB: Write-back</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NA: No Action</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Execution schedule for code fragment (i) above.

(c) Hardware utilization trace for schedule in (b).
Superscalar Execution: An Example

- In the above example, there is some wastage of resources due to data dependencies.
- The example also illustrates that different instruction mixes with identical semantics can take significantly different execution time.
Superscalar Execution

• Scheduling of instructions is determined by a number of factors:
  – True Data Dependency: The result of one operation is an input to the next.
  – Resource Dependency: Two operations require the same resource.
  – Branch Dependency: Scheduling instructions across conditional branch statements cannot be done deterministically a-priori.
  – The scheduler, a piece of hardware looks at a large number of instructions in an instruction queue and selects appropriate number of instructions to execute concurrently based on these factors.
  – The complexity of this hardware is an important constraint on superscalar processors.
Superscalar Execution: Issue Mechanisms

• In-order issue.
• A more aggressive model: dynamic issue.
• Performance of in-order issue is generally limited.
• Due to limited parallelism in typical instruction traces, dependencies, or the inability of the scheduler to extract parallelism, the performance of superscalar processors is eventually limited.
• Conventional microprocessors typically support four-way superscalar execution.
Very Long Instruction Word (VLIW) Processors

- The hardware cost and complexity of the superscalar scheduler is a major consideration in processor design.
- To address this issue, VLIW processors rely on compile time analysis to identify instructions that can be executed concurrently.
  - “dumb” hardware
  - “intelligent” compiler
- This concept was used with some commercial success in the Multiflow Trace machine (circa 1984).
- Variants of this concept are employed in the Intel IA64 processors.
- Transmeta processor
Very Long Instruction Word (VLIW) Processors: Considerations

• Compilers, however, do not have runtime information such as cache misses. Scheduling is, therefore, inherently conservative.
• Branch and memory prediction is more difficult.
• VLIW performance is highly dependent on the compiler. A number of techniques such as loop unrolling, speculative execution, branch prediction are critical.
• Typical VLIW processors are limited to 4-way to 8-way parallelism.
Limitations of Memory System Performance

- Memory system, and not processor speed, is often the bottleneck for many applications.
- Memory system performance is largely captured by two parameters, latency and bandwidth.
- Latency is the time from the issue of a memory request to the time the data is available at the processor.
- Bandwidth is the rate at which data can be pumped to the processor by the memory system.
Memory System Performance: Bandwidth and Latency

• It is very important to understand the difference between latency and bandwidth.

• Consider the example of a fire-hose. If the water comes out of the hose two seconds after the hydrant is turned on, the latency of the system is two seconds.

• Once the water starts flowing, if the hydrant delivers water at the rate of 5 gallons/second, the bandwidth of the system is 5 gallons/second.

• If you want immediate response from the hydrant, it is important to reduce latency.

• If you want to fight big fires, you want high bandwidth.
Memory Latency: An Example

- Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor has two multiply-add units and is capable of executing four instructions in each cycle of 1 ns. The following observations follow:
  - The peak processor rating is 4 GFLOPS.
  - Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made, the processor must wait 100 cycles before it can process the data.
Memory Latency: An Example

• On the above architecture, consider the problem of computing a dot-product of two vectors.
  – A dot-product computation performs one multiply-add on a single pair of vector elements, i.e., each floating point operation requires one data fetch.
  – It follows that the peak speed of this computation is limited to one floating point operation every 100 ns, or a speed of 10 MFLOPS, a very small fraction of the peak processor rating!
Improving Effective Memory Latency Using Caches

• Caches are small and fast memory elements between the processor and DRAM.
• This memory acts as a low-latency high-bandwidth storage.
• If a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache.
• The fraction of data references satisfied by the cache is called the cache *hit ratio* of the computation on the system.
• Cache hit ratio achieved by a code on a memory system often determines its performance.
Impact of Caches: Example

Consider the architecture from the previous example. In this case, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle. We use this setup to multiply two matrices $A$ and $B$ of dimensions $32 \times 32$. We have carefully chosen these numbers so that the cache is large enough to store matrices $A$ and $B$, as well as the result matrix $C$. 
Impact of Caches: Example (continued)

The following observations can be made about the problem:

- Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately 200 µs.
- Multiplying two $n \times n$ matrices takes $2n^3$ operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or 16 µs) at four instructions per cycle.
- The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., 200 + 16 µs.
- This corresponds to a peak computation rate of 64K/216 or 303 MFLOPS.
Impact of Caches

• Repeated references to the same data item correspond to temporal locality.

• In our example, we had $O(n^2)$ data accesses and $O(n^3)$ computation. This asymptotic difference makes the above example particularly desirable for caches.

• Data reuse is critical for cache performance.
Impact of Memory Bandwidth

• Memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units.
• Memory bandwidth can be improved by increasing the size of memory blocks.
• The underlying system takes $l$ time units (where $l$ is the latency of the system) to deliver $b$ units of data (where $b$ is the block size).
Impact of Memory Bandwidth: Example

• Consider the same setup as before, except in this case, the block size is 4 words instead of 1 word. We repeat the dot-product computation in this scenario:
  – Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in 200 cycles.
  – This is because a single memory access fetches four consecutive words in the vector.
  – Therefore, two accesses can fetch four elements of each of the vectors. This corresponds to a FLOP every 25 ns, for a peak speed of 40 MFLOPS.
Impact of Memory Bandwidth

• It is important to note that increasing block size does not change latency of the system.
• Physically, the scenario illustrated here can be viewed as a wide data bus (4 words or 128 bits) connected to multiple memory banks.
• In practice, such wide buses are expensive to construct.
• In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved.
Impact of Memory Bandwidth: Example

Consider the following code fragment:

```c
for (i = 0; i < 1000; i++)
    column_sum[i] = 0.0;
for (j = 0; j < 1000; j++)
    column_sum[i] += b[j][i];
```

The code fragment sums columns of the matrix b into a vector `column_sum`. 
Impact of Memory Bandwidth: Example

- The vector `column_sum` is small and easily fits into the cache.
- The matrix `b` is accessed in a column order.
- The strided access results in very poor performance.

Multiplying a matrix with a vector: (a) multiplying column-by-column, keeping a running sum; (b) computing each element of the result as a dot product of a row of the matrix with the vector.
Impact of Memory Bandwidth: Example

We can fix the above code as follows:

```c
for (i = 0; i < 1000; i++)
    column_sum[i] = 0.0;
for (j = 0; j < 1000; j++)
    for (i = 0; i < 1000; i++)
        column_sum[i] += b[j][i];
```

In this case, the matrix is traversed in a row-order and performance can be expected to be significantly better.
Memory System Performance: Summary

The series of examples presented in this section illustrate the following concepts:

- Exploiting spatial and temporal locality in applications is critical for amortizing memory latency and increasing effective memory bandwidth.
- The ratio of the number of operations to number of memory accesses is a good indicator of anticipated tolerance to memory bandwidth.
- Memory layouts and organizing computation appropriately can make a significant impact on the spatial and temporal locality.
Alternate Approaches for Hiding Memory Latency

- Consider the problem of browsing the web on a very slow network connection. We deal with the problem in one of three possible ways:
  - we anticipate which pages we are going to browse ahead of time and issue requests for them in advance;
  - we open multiple browsers and access different pages in each browser, thus while we are waiting for one page to load, we could be reading others; or
  - we access a whole bunch of pages in one go - amortizing the latency across various accesses.
Multithreading for Latency Hiding

A thread is a single stream of control in the flow of a program. We illustrate threads with a simple example:

```
for (i = 0; i < n; i++)
    c[i] = dot_product(get_row(a, i), b);
```

Each dot-product is independent of the other, and therefore represents a concurrent unit of execution. We can safely rewrite the above code segment as:

```
for (i = 0; i < n; i++)
    c[i] = create_thread(dot_product, get_row(a, i), b);
```
Prefetching for Latency Hiding

- Misses on loads cause programs to stall.
- Why not advance the loads so that by the time the data is actually needed, it is already there!
- The only drawback is that you might need more space to store advanced loads.
- However, if the advanced loads are overwritten, we are no worse than before!
Tradeoffs of Multithreading and Prefetching

- Bandwidth requirements of a multithreaded system may increase very significantly because of the smaller cache residency of each thread.
- Multithreaded systems become bandwidth bound instead of latency bound.
- Multithreading and prefetching only address the latency problem and may often exacerbate the bandwidth problem.
- Multithreading and prefetching also require significantly more hardware resources in the form of storage.
Dichotomy of Parallel Computing Platforms

• An explicitly parallel program must specify concurrency and interaction between concurrent subtasks.
• The former is sometimes also referred to as the control structure and the latter as the communication model.
Control Structure of Parallel Programs

• Parallelism can be expressed at various levels of granularity - from instruction level to processes.
• Between these extremes exist a range of models, along with corresponding architectural support.
Control Structure of Parallel Programs

- Processing units in parallel computers either operate under the centralized control of a single control unit or work independently.
- If there is a single control unit that dispatches the same instruction to various processors (that work on different data), the model is referred to as single instruction stream, multiple data stream (SIMD).
- If each processor has its own control unit, each processor can execute different instructions on different data items. This model is called multiple instruction stream, multiple data stream (MIMD).