Parallel Programming Platforms

Chapter#2

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 itensive 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 -Performance

- ► 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- Limitations

- ► 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.

Superscalar Execution: An Example

Example of a two-way superscalar execution of instructions.

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 the simpler model, instructions can be issued only in the order in which they are encountered. That is, if the second instruction cannot be issued because it has a data dependency with the first, only one instruction is issued in the cycle. This is called *in-order* issue.

- In a more aggressive model, instructions can be issued out of order. In this case, if the second instruction has data dependencies with the first, but the third instruction does not, the first and third instructions can be co-scheduled. This is also called dynamic issue.

- Performance of in-order issue is generally limited.

Superscalar Execution: Efficiency Considerations

► Not all functional units can be kept busy at all times.

► If during a cycle, no functional units are utilized, this is referred to as vertical waste.

► If during a cycle, only some of the functional units are utilized, this is referred to as horizontal waste.

► 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 issues, VLIW processors rely on compile time analysis to identify and bundle together instructions that can be executed concurrently.

- These instructions are packed and dispatched together, and thus the name very long instruction word.

- 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.

Very Long Instruction Word (VLIW) Processors: Considerations

- ► Issue hardware is simpler.

- ► Compiler has a bigger context from which to select co-scheduled instructions.

- ► 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.

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 (continued)

- The following observations can be made about the problem:

  - **Given:**

  - 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.

  - **Solution :**

  - 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 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).

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.

- The first approach is called *prefetching*, the second *multithreading*, and the third one corresponds to spatial locality in accessing memory words.

Multithreading for Latency Hiding (thread execution of m*m

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);
```

Multithreading for Latency Hiding: Example

- In the code, the first instance of this function accesses a pair of vector elements and waits for them.

- In the meantime, the second instance of this function can access two other vector elements in the next cycle, and so on.

- After l units of time, where l is the latency of the memory system, the first function instance gets the requested data from memory and can perform the required computation.

- In the next cycle, the data items for the next function instance arrive, and so on. In this way, in every clock cycle, we can perform a computation.

Multithreading for Latency Hiding

- The execution schedule in the previous example is predicated upon two assumptions: the memory system is capable of servicing multiple outstanding requests, and the processor is capable of switching threads at every cycle.

- It also requires the program to have an explicit specification of concurrency in the form of threads.

- Machines such as the HEP and Tera rely on multithreaded processors that can switch the context of execution in every cycle. Consequently, they are able to hide latency effectively.

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.

Interconnection Networks for Parallel Computers

- Interconnection networks carry data between processors and to memory.

- Interconnects are made of switches and links (wires, fiber).

- Interconnects are classified as static or dynamic.

- Static networks consist of point-to-point communication links among processing nodes and are also referred to as *direct* networks.

- Dynamic networks are built using switches and communication links. Dynamic networks are also referred to as *indirect* networks.

Static and Dynamic Interconnection Networks

Classification of interconnection networks: (a) a static network; and (b) a dynamic network.

Interconnection Networks

- ► Switches map a fixed number of inputs to outputs.

- ► The total number of ports on a switch is the *degree* of the switch.

- ► The cost of a switch grows as the square of the degree of the switch, the peripheral hardware linearly as the degree, and the packaging costs linearly as the number of pins.

Interconnection Networks: Network Interfaces

- ► Processors talk to the network via a network interface.

- ► The network interface may hang off the I/O bus or the memory bus.

- ► In a physical sense, this distinguishes a cluster from a tightly coupled multicomputer.

- ► The relative speeds of the I/O and memory buses impact the performance of the network.

Network Topologies: Buses

- Some of the simplest and earliest parallel machines used buses.

- All processors access a common bus for exchanging data.

- The distance between any two nodes is $O(1)$ in a bus. The bus also provides a convenient broadcast media.

- However, the bandwidth of the shared bus is a major bottleneck.

- Typical bus based machines are limited to dozens of nodes. Sun Enterprise servers and Intel Pentium based shared-bus multiprocessors are examples of such architectures.

Network Topologies: Buses

Bus-based interconnects (a) with no local caches; (b) with local memory/caches.

Since much of the data accessed by processors is local to the processor, a local memory can improve the performance of bus-based machines.

Network Topologies: Crossbar

- *Crossbar networks* allow any processor in the system to connect to any other processor or memory unit so that many processors can communicate simultaneously without contention

Network Topologies: Crossbars

A crossbar network uses an *p×m* grid of switches to connect *p* inputs to m outputs in a non-blocking manner.

A completely non-blocking crossbar network connecting *p* processors to b memory banks.

Message Passing Costs in Parallel Computers

► The total time to transfer a message over a network comprises of the following:

  ► *Startup time* (t_s): Time spent at sending and receiving nodes (executing the routing algorithm, programming routers, etc.).

  ► *Per-hop time* (t_h): This time is a function of number of hops and includes factors such as switch latencies, network delays, etc.

  ► *Per-word transfer time* (t_w): This time includes all overheads that are determined by the length of the message. This includes bandwidth of links, error checking and correction, etc.