CS 3006 Parallel and Distributed Computer

Fall 2022

1. Learn about parallel and distributed computer architectures.(1)
2. Implement different parallel and distributed programming paradigms and algorithms using Message-Passing Interface (MPI) and OpenMP.(4)
3. Perform analytical modelling, dependence, and performance analysis of parallel algorithms and programs.(2)
4. Use Hadoop or MapReduce programming model to write bigdata applications.(5)

Week # 3 – Lecture # 7, 8, 9

5th , 7th, 9th September 2022

8th , 10th , 12th Safar ul Muzaffar, 1444

Dr. Nadeem Kafi Khan

Lecture # 7 – Topics (Lab # 2)

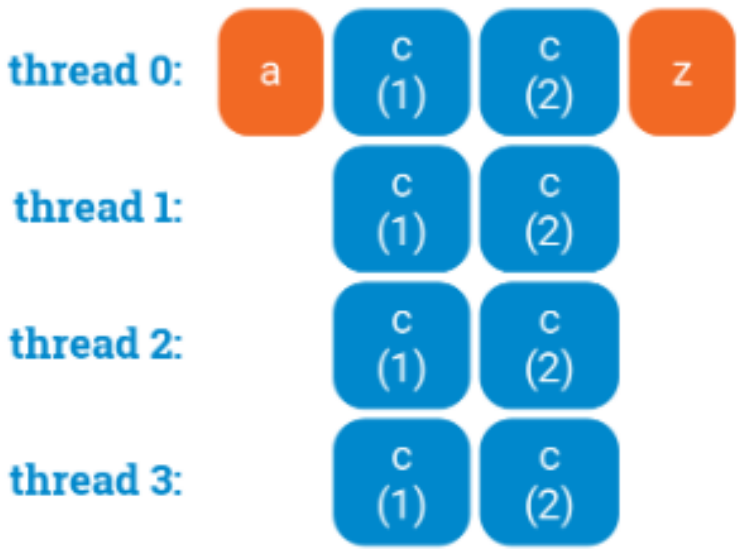
- **OpenMP vs Pthreads**
- **OpenMP Execution**
  - **#pragma omp parallel**
  - **#pragma omp critical**

Basic multithreading construction: parallel regions

Any useful OpenMP program has to use at least one `parallel` region. This is a construction that tells OpenMP that we would like to create multiple threads:

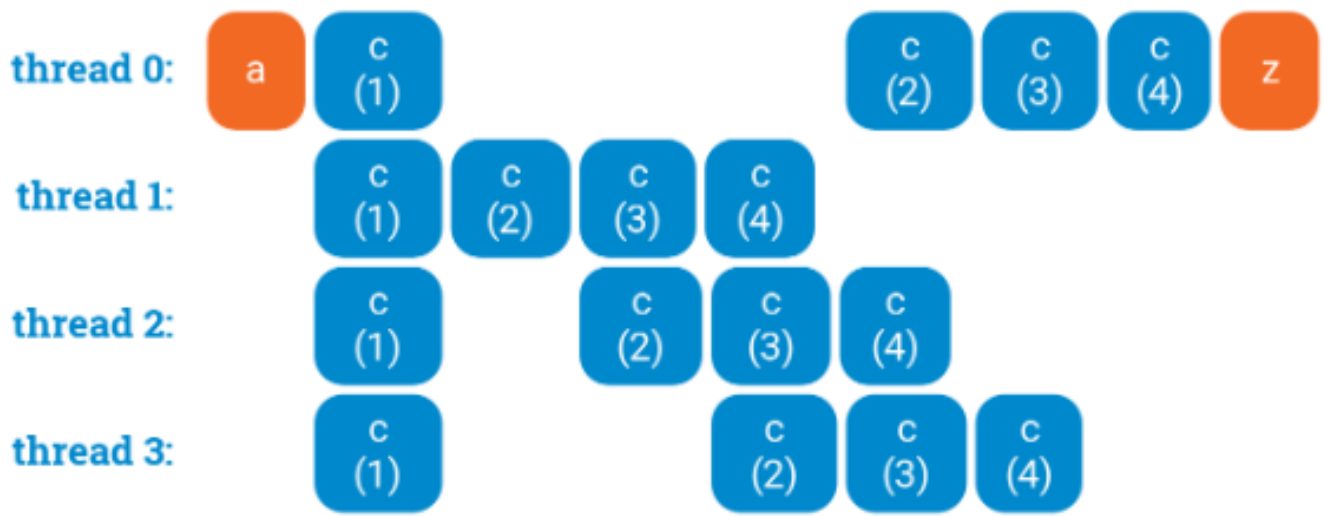
```
a();
#pragma omp parallel
{
    c(1);
    c(2);
}
z();
```

If we execute the above program, the timeline of the execution might look e.g. like this (in the figure, time goes from left to right):



Whenever we modify a shared resource, we must take care of proper synchronization between the threads. The simplest synchronization primitive is a critical section. A critical section ensures that **at most one thread is executing code that is inside the critical section**. For example, here only one thread is running `c(2)` at any point of time:

```
a();
#pragma omp parallel
{
    c(1);
    #pragma omp critical
    {
        c(2);
    }
    c(3);
    c(4);
}
z();
```



Lecture # 8 - Topics

- Implicit parallelism (chapter # 2)
  - Pipelines
    - Hazards (Data, Naming, Control)
    - How to mitigate Hazards?
  - Superscalar Execution (Super-pipelines)
    - In-order execution
    - Out-of-order execution
    - Branch prediction
  - Very Large Instruction Word (VLIW)
    - Role of compiler
    - Comparison with Pipelining

Implicit Parallelism: Trends in Microprocessor Architectures

- Microprocessor clock speeds have posted impressive gains over the past two decades (two to three orders of magnitude). - 2003
- 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 similar 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 large number of stages (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

- 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.

Lecture # 9 - Topics

- Limitations of Memory System Performance
  - Latency and Bandwidth
- Improving Effective Memory Latency Using Caches
- Impact of Caches
  - Temporal Locality
  - Spatial Locality
- Memory Latency Example (Numerical)
  - Without cache (accumulation of dot product results)
  - With a 32KB cache (Matric multiplication)

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

- What is 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.

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.
  - Avg. Memory Access time = (cache hit time) + (cache miss time)

Impact of Caches

- **Temporal locality** correspond to repeated references to the same data item.

- 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 Caches

- **Spatial locality** (also termed data locality) refers to the use of data elements within relatively close storage locations.

- **Sequential locality,** a special case of spatial locality, occurs when data elements are arranged and accessed linearly, such as traversing the elements in a one-dimensional array.

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 (FLOating Point Instructions per second)

  - 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.**

Kilo 10^3, Mega 10^6, Giga 10^9, Tera 10^{12}, Peta 10^{15}, Exa 10^{18}, Zetta 10^{21}, Yotta 10^{24},
Milli 10^{-3}, Micro 10^{-6}, Nano 10^{-9}, Peco 10^{-12}, Femto 10^{-15},

Memory Latency: An Example

- On the above architecture, consider the problem of computing a dot-product of two vectors.

```
int dotProduct(int vect_A[], int vect_B[]) {

    int product = 0;
     // Loop for calculate dot product
    for (int i = 0; i < n; i++)
        product = product + vect_A[i] * vect_B[i];
    return product;
}
```

  - 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!

```
LD R1, [R2]          ; 100ns
LD R3, [R4]          ; 100ns
MADD R5, R1, R2      ; 1ns
```

2/200ns= 10 MFLOPS

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 × 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.

```
for(i=0;i<r;i++) {
  for(j=0;j<c;j++) {
    for(k=0;k<c;k++) {
      mul[i][j]+=a[i][k]*b[k][j];
    }}}
```

O (N^3)

Action items:

1. Draw processor, cache and memory
2. Draw Cache of size 32 K
3. How three matrices of size 32 x 32 fits into this cache?
4. How cache data is reused during these calculations?

Impact of Caches: Example (continued)

- The following observations can be made about the problem:
  - Fetching the two matrices (n x n = 32 x 32) into the cache corresponds to fetching 2K words, which takes approximately **200 µs**. **HOW?**
  - 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. **See slide in the end of this slide deck.**

$$n^2(n + (n-1)) = 2n^3 - n^2 = O(n^3).$$

  - 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.

A **thirty-fold improvement** over the previous example. However, it is still **less than 10% of the peak processor performance**. By placing a small cache memory, we are able to improve processor utilization considerably.

Complexity of Matric Multiplication

Say you have two square matrices A and B. Computing element a_{ij} of AB requires taking the dot product of row i in A and column j in B. Computing the dot product requires n multiplications and $n-1$ additions. Since there are n^2 elements, the dot product must be computed n^2 times.

Thus the total number of operations is $n^2(n+(n-1)) = 2n^3 - n^2 = O(n^3)$.