CS525 Parallel Computing

© Xiyuan Chen

Spring Term, 2024

Information

- Course website: https://www.cs.purdue.edu/homes/ayg/CS525/

Contents

1 Application properties of parallelism

- Simulations. e.g., weather forecasting, molecular dynamics, etc.
- Learning & Data Analysis. e.g., machine learning, data mining, etc.

2 Processor

Jobs of a processor: 1. fetch instructions. 2. decode instructions. 3. execute instructions. 4. store results.

Most of the time, the instructions are of few types: e.g. MOVE, ADD, LD, STD

2.1 Pipeline Parallelism

Basic idea: an instruction can be executed while the next one is being decoded and the next one is being fetched.

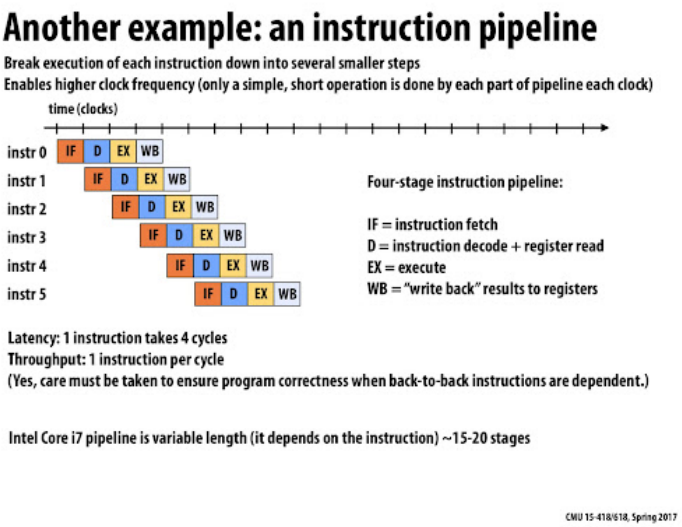

Figure 1: Pipeline Parallelism

Limitations:

- **Bottleneck**: the slowest stage determines the speed of the pipeline. We solve by 1. decpmosing the slowest stage into multiple stages. 2. Use multiple pipelines (Superscalar Excution, see below).
- **Conditional jumps**: the pipeline must be flushed when a branch is encountered. We solve by branch prediction.

2.2 Superscalar Execution

Having multiple excution units (pipelines) to execute multiple instructions at the same time.

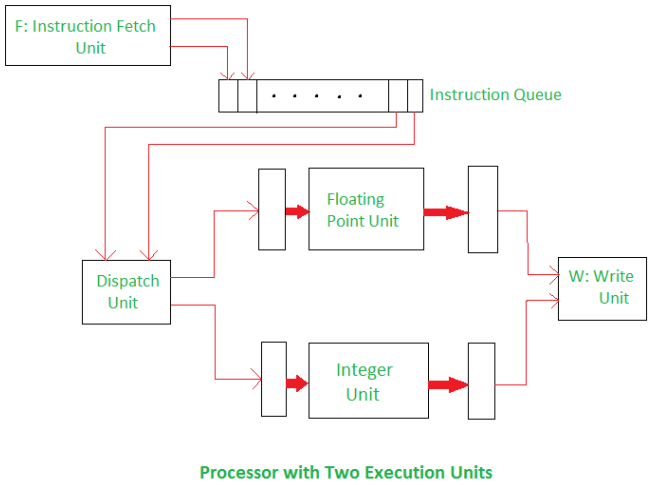

Figure 2: Superscalar Execution Example with 2 Units

2.2.1 Difference between pipelining and superscalar

- pipelining focuses on breaking down an instruction into sequential stages, whereas superscalar emphasizes on executing multiple instructions at the same time.
- pipelining typically has one execution unit, whereas superscalar has multiple execution units.
- superscalar is more parallel than pipelining.

2.2.2 Scheduler

See video here.

How to determine which instructions should run in parallel and schedule instructions to multiple pipelines? We consider:

- **Data Dependencies**: the result of one operation is an input to the next.
- **Control Dependencies**: the next operation depends on the result of a conditional branch.
- **Resource Dependencies**: the next operation requires the resource (hardware such as AMU) as the current one (all hardwares are occupied).

We also **unroll the loops** to expose more parallelism.

Methods to schedule instructions

- **In-order Issuing**: issue instructions **in the order**. When there is a dependency issue, stall the pipeline.
- **Out-of-order Issuing (Dynamic Scheduling)**: find all the instructions in the prefetch queue that are ready to execute and issue them.

2.2.3 Limitations of Superscalar

1. Need to resolve dependencies in sub nano-seconds.
2. Has only limited scope since the instruction prefetch queue is small.

2.2.4 Very Long Instruction Word Processor (VLIW)

Move the complexity of scheduling to the compiler. The compiler will schedule the instructions and pack them into a single long instruction. The processor will then execute the packed instruction.

Limitations of VLIW: Unable to see the runtime information in the compiler. (e.g., memory access latency)

3 Memory

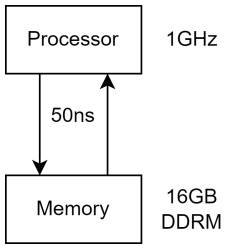

Figure 3: Memory-Processor Architecture

3.1 Latency and Bandwidth

- **Latency**: The time it takes for the processor to start / time to get data back from memory.
- **Bandwidth**: Once the processor starts, how fast can it execute / number of words that got transferred per unit time.

> In memory, to increase bandwidth, when we fetch a word, we also get a parcel of words next to it.
>
> e.g. if latency is 50 cycles and fetch speed is 1 word per cycle, then the first word is fetched at 50ns, and the rest of the words are fetched at 1ns. If the memory is set to fetch 4 words, then we have $50 + 1 + 1 + 1$ throughput $\rightarrow 4$ words$/53$ ns bandwidth.

3.2 Caches

To address the latency and bandwidth bottleneck, we use caches — fast memory near the processor.

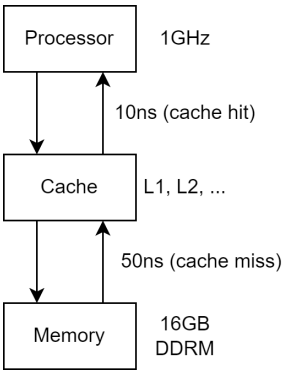

Figure 4: Memory-Processor Architecture with Cache

3.2.1 Mapping data from memory to cache

Cache is a small memory, so we need to map data from a large storage to a small storage.

Direct Map Cache

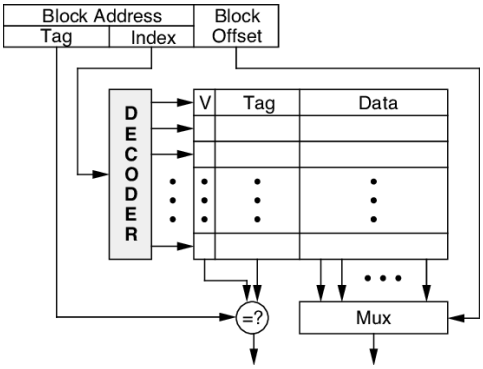

Figure 5: Direct Map Cache

Step 1. Take the lower bits of the address and use them as the index to the cache. e.g. if the cache has 2^{13} entries, then the lower 13 bits of the address will be used as the index to the cache. i.e. $c = m\%2^{13}$.

Step 2. Determine if the data is already in the cache by comparing the upper bits of the address (tag). If all match, then the data is in the cache. Otherwise, the data is not in the cache, and we update the upper bits and write in the data.

Issue: what should the replacement policy be? Theoratically, we should replace the least recently used (LRU) data. How? Set-associated cache $\downarrow$.

Set-Associated Cache Approximating LRU (not exactly the same). Treat the cache as n independent sub-sized caches. Each memory address now maps to n cache entries. For each word in cache, we have a time tag to indicate when it was last used. When we need to replace a word, we replace the old one. When searching for a word, we search n entries and return the one that matches.

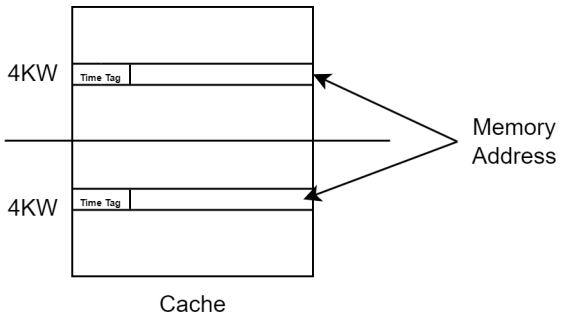

Figure 6: Two-way Set-Associated Cache

3.3 Spatiality and Locality

3.3.1 Dot product

Video here. When access is spatical localized and cache access is in bulk, we get good performance.

3.3.2 Matrix multiplication

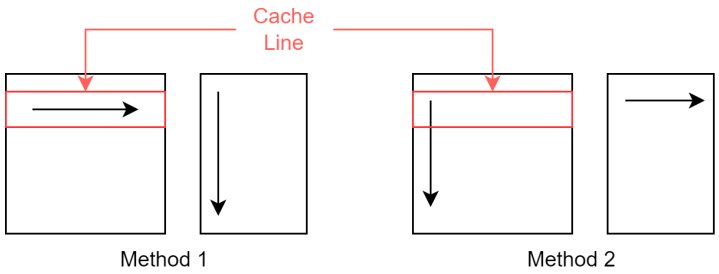

Figure 7: Two Methods of Multiplication

If the array is laid out in **row major**, method 1 is better since the first matrix in method 2 wastes cache line every time because it iterates in column.

Subdividing matrix Even we ensure that the data layout is aligned properly like described above. Our program sitll cannot run as efficient as we expect because **the cache is too small to contain a large matrix** and therefore we end up accessing the same part of the matrix again and again.

Solution: Subdividing matrix into small matrices so that it can fit in the cache.

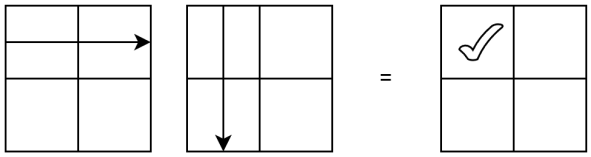

Figure 8: Break Matrix into 4 Submatrices

3.4 Use concurrency to hide lantency

Create **multiple threads** to hide the latency of memory access. e.g. when we are waiting for data from memory in a thread, we can do some computation in another thread. But we need to be aware of the **memory bandwidth** and **processor context**. If memeory bandwidth and contexts are enough, then we can create more threads and can hide more latency.

4 Parallel Platforms

Design concepts of parallel platforms (e.g., multi-core processors):

- Control Structure: synchronization, how to execute codes.
- Communication Structure: share data, how to communicate.

4.1 Control Structure

Simple Instruction Multiple Data (SIMD) (Data Parallelism), execute the same instruction on multiple different data.

e.g. Image processing: applying convolution operation (filters) repeatedly on different pixels of an image.

Multiple Instruction Multiple Data (MIMD) execute different instructions on different data.

Single Program Multiple Data (SPMD) execute the same program on different data.

4.2 Communication Structure

Shared Memory All processors share the same memory.

Approach 1: Just use a shared memory and all processors can access it.

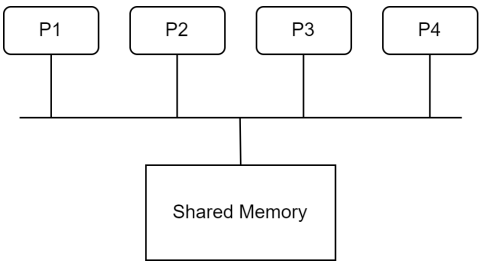

Figure 9: Naive Shared Memory Design

Issue: Every processor will have its own cache. When one processor writes to the memory, other processors' caches are not updated.

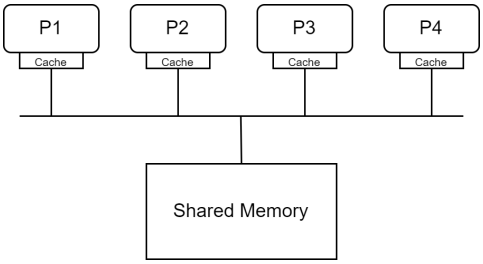

Figure 10: Naive Shared Memory Design with Cache

Approach 2: Every processor has its own memory section and we somehow interconnect them.

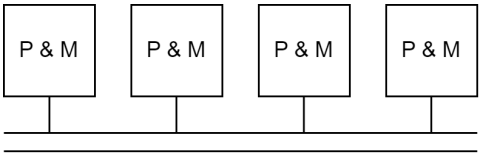

Figure 11: Naive Shared Memory Design with Individual Memory