Parallelism in Computing

Yi Zhao
Dept. Electrical and Computer Engineering

Parallelism in Computing

Introduction to Parallelism

What is Parallelism in Computing?

- Also known as *Parallel Computing*

- Computations carried out simultaneously
  - Problems typically divided into small ones

Why is Parallelism Needed?

- Single processing unit – sequential
  - Traditional technologies to boosting computing capacity
    - Increasing clock frequency
    - Shrinking transistor dimension
  - Improvement constrained by physical barriers
  - Finite performance of each single unit

- Multiple processing units – parallel
  - No physical limit (still has budget/power constraints)
  - Performance can be easily scaled up

Advantage of Parallelism

- Using parallelism significantly boosts performance.
- Enhancements through exploiting parallelism have been studied extensively in computer design and program developments.

Parallelism Technologies

- Instruction-level
  - Pipelining, branch prediction, dynamic scheduling, speculation
- Data-level
  - Vector architecture, SIMD, GPU, FPGA
  - Multithreading, multiprocessing
- Task-level
  - Multithreading, multiprocessing

Parallelism in Intel DevCloud

Parallelism	Intel DevCloud for the Edge
Instruction-level	• Pipelining – handled by CPU, not visible to developer • Hardware-based scheduling – handled by CPU, not visible to developer
SIMD	• CPU/GPU/VPU/FPGA – supported in inference engine of OpenVINO toolkit • CPU – SIMD intrinsic supported in C/C++ compiler, available to developer
Thread-level	• Supported in inference engine of OpenVINO toolkit • Supported in Python module threading, but doesn't have speedup gain (in CPython implementation, only one thread can execute Python code at a time) • Supported in C/C++
Process-level	• Supported in inference engine of OpenVINO toolkit

Enhancement and Speedup

- A *quantitative* measurement of performance gain in computer/program enhancements is needed.
- *Speedup* is defined as the ratio of performance

$$\text{Speedup} = \frac{\text{Performance using enchacement}}{\text{Performance withou enhancement}}$$

- Alternatively, it's usually calculated from measured execution time

$$\text{Speedup} = \frac{\text{Execution time without enhancement}}{\text{Execution time using enahncement}}$$

Amdahl's Law – Formula

- One partially enhanced program's whole execution time is

$$\text{Time}_{\text{new}} = \text{Time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Frcation}_{\text{enahanced}}}{\text{Speedup}_{\text{enchanced}}} \right)$$

$\text{Fraction}_{\text{enhanced}}$ – Fraction of original execution time that can be enhanced

$\text{Speedup}_{\text{enchaced}}$ – Performance gain on fraction that can be enhanced

- Overall speedup becomes

$$\textbf{Speedup}_{\textbf{overall}} = \frac{\mathbf{1}}{(\mathbf{1} - \textbf{Fraction}_{\textbf{enhanced}}) + \frac{\textbf{Fraction}_{\textbf{enhanced}}}{\textbf{Speedup}_{\textbf{enhanced}}}}$$

Amdahl's Law – Example

- Suppose that one web serving program spends 40% time on computation and 60% waiting on I/O.
- What is the overall speedup if the original processor is replaced with new one that is 10 times faster?
- It's quite clear that

$$\text{Fraction}_{\text{enhanced}} = 0.4$$
$$\text{Speedup}_{\text{enhanced}} = 10$$

- Applying Amdahl's law

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = \frac{1}{0.64} = 1.5625$$

Parallelism in Computing

Multithreading

Thread

- Smallest instruction sequence handled by scheduler
- *Has its own state* and program counter ← like process
- *Shares address space* of a single process with other threads (in the same process) ← different from process
- Easy access to data of other threads (in the same process)

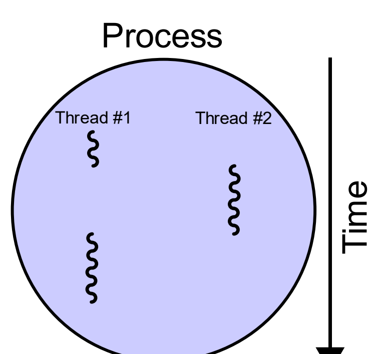
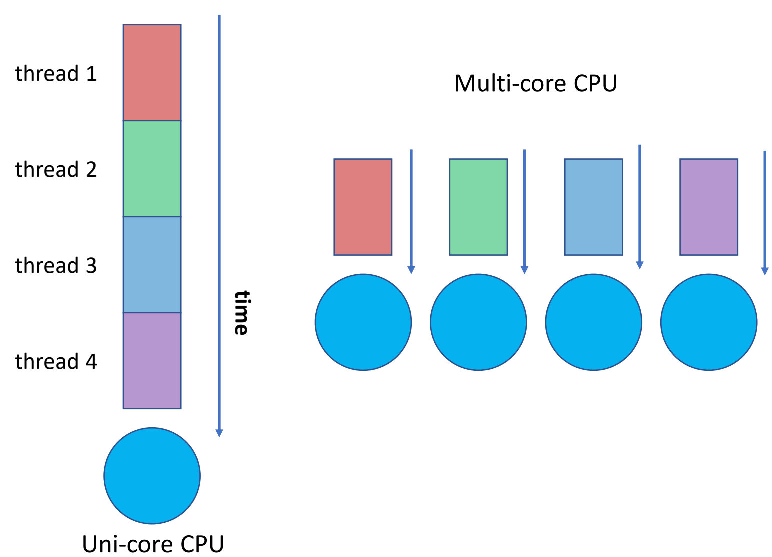


Image from Wikipedia

Multithreading

- Multithreading
  - Allowing multiple threads to share a processor without intervening process switch
    - Thread switch is much faster than process switch
  - A primary technique for exposing more parallelism to the hardware
    - Boosting instruction/data/task-level performance

Multithreading – Uni-Core vs. Multi-Core



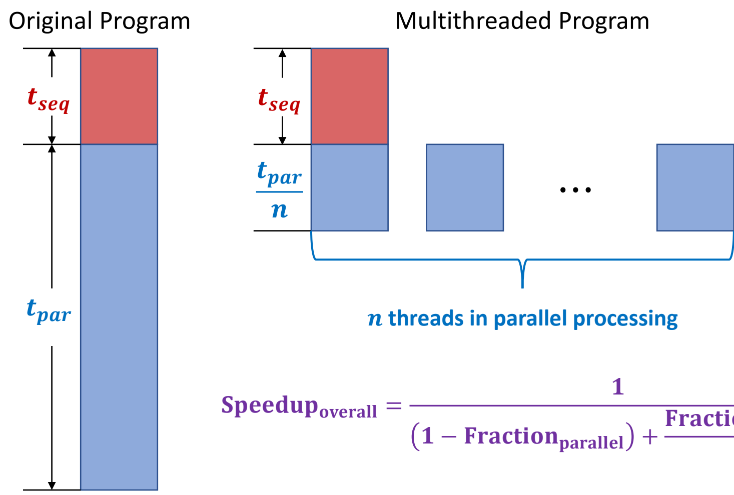
Multithreading in Parallelism

- Data-level
  - Each thread conducting same operations on one subset of the same data
  - e.g. UHD image filtering
    - Image divided into slices/tiles, each handled by one thread
- Task-level
  - Each thread conducting different operations on same or different data
  - e.g. Zoom meeting
    - Video capturing, compressing, and transmission handled by different threads

Multithreading – Data-level

- Original execution time divided into two parts:
  - Sequential – not to be enhanced through multithreading
    - Only one (main) thread processing
    - E.g. file I/O, thread creation, …
  - Parallel – to be enhanced through multithreading
    - Work distributed to threads working simultaneously
    - E.g. Array addition, matrix multiplication, …

Multithreading – Ideal Speedup



$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{n}}$$

Multithreading – Actual Speedup

- The actual multithreading speedup is impacted by lots real-world factors.
- Thread overhead
  - Context switching, synchronization, …
- Load balance
  - Work distribution
- Resource availability
  - CPU, I/O, memory, …

Instruction Pipelining – Principle

- Each instruction's execution is divided to multiple stages
- Multiple instructions are overlapped in execution
- If perfectly balanced

$$\text{time per instruction on pipelined} = \frac{\text{time per instruction on unpipelined}}{\text{number of pipe stages}}$$

Instruction Pipelining – Example

Pipelining on 5-stages RISC

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

IF – Instruction fetch from memory

ID – Instruction decode/register fetch cycle

EX – Execution/effective address cycle

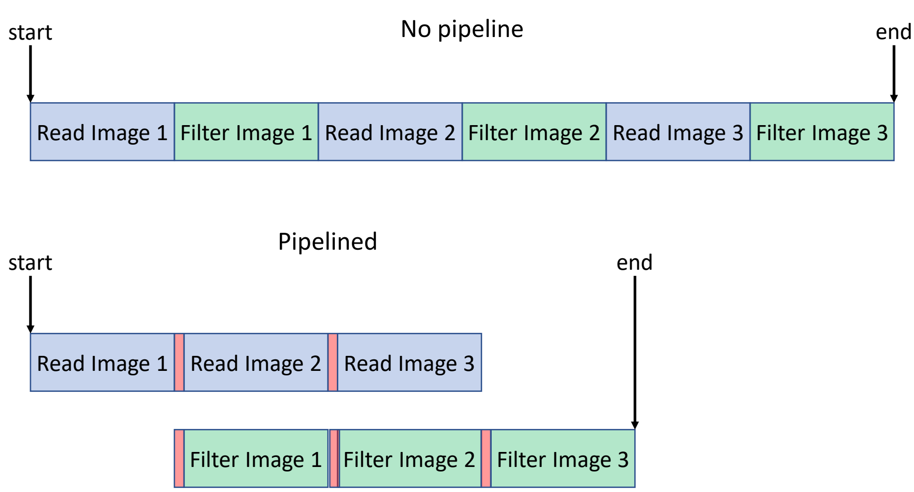
MEM – Memory access (read/write data using effective address)

WB – Write-back result to register

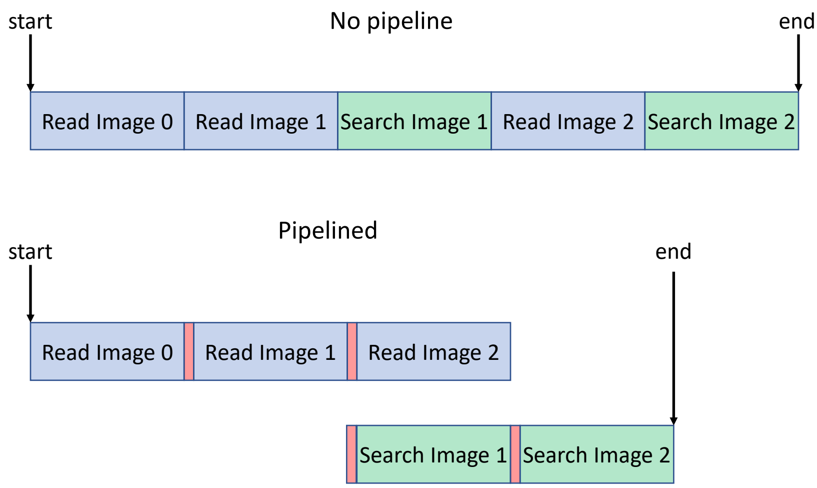
Multithreading – Task-level Pipelining

- Similar to the concept of instruction pipelining
- Each task's execution is divided into multiple stages (or steps)
- Multiple tasks can be scheduled to executed in an overlapping fashion

Multithreading – Pipelined Image Filtering



Multithreading – Pipelined Motion Search



Parallelism in Computing

Multithreaded Programming in C++

C++ Support on Multithreading

- Historically, C/C++ didn't have native support on multithreaded programming in their language standards since the very beginning.
- Implementations to support multithreading are mostly *OS/vendor dependent*.
  - i.e. Windows and Linux provide different functions/tools in programming multithreaded applications.
- Since **C++11**, native support on multithreaded programming has been provided.
  - thread, mutex, chrono

C++ Support – `thread`

- The **`std::thread`** class provides support to create individual threads of execution.

```
void foo()                    // ← thread function
{
  // doing something
}

int main()
{
  …
  std::thread child(foo);   // ← create/start thread child
  child.join();             // ← wait child to complete
  …
}
```

C++ Support – thread function

- Thread function – return value ignored
  - May have parameters
  - Thread terminates once returned from it

```
void bar(int N)                      // has parameter
{
  for (int i = 0; i < N; i++)  printf(" %d", i);
}

int main()
{
  …
  std::thread child(bar, 100);  // ← argument is 100
  child.join();            // ← wait child to complete
  …
}
```


C++ Support – `mutex`

- The **`std::mutex`** class provides support on *mutual exclusion* to avoid data race and enable concurrent execution of critical sections.

```
void produce()
{
  // produce one item
  std::unique_lock<std::mutex> lock(mtxReadyItem);
  // add one item to ready queue and increase count
}

void consume()
{
  std::unique_lock<std::mutex> lock(mtxReadyItem);
  // get one item from ready queue and decrease count
  // use obtained item
}
```

C++ Support – chrono

- Old functions such as `time()` are not very precise.
- The **`std::chrono`** sub-namespace contains high-precision clock support.

```
void foo()
{
  using std::chrono::high_resolution_clock;
  using std::chrono::duration;
  auto t1 = high_resolution_clock::now(); // ← start
  // doing something
  auto t2 = high_resolution_clock::now(); // ← end
  duration<double> etSec = t2 - t1;  // ← duration in s
  duration<double, std::micro> et = t2 - t1;
  double etMic = et.count();         // ← duration in us
}
```


C++ Support – sleep_for

- The **`sleep_for()`** function in `std::this_thread` sub-namespace locks for *at least* specified duration.
- Very helpful in allowing one idle (waiting) thread to yield execution to other busy threads.

```
void foo()
{
  using namespace std::chrono_literals;
  while (!done) {
    if (!ready) {
      std::this_thread::sleep_for(100us);  // 100 us
      continue;
    // do something
  }
}
```

Programming Activity 1

- Write one multithreaded program
  - One child thread print 1, 2, 3, .., 10
  - Another child thread print A, B, C, D, …, N
  - Main thread wait them to finish using join()
  - Run this program and observe its output.
  - Run it again and can you notice the difference?

Parallelism in Computing

Challenges in Multithreading

Multithreading – Execution Order

- Within-thread – *deterministic*
  - Same as single threaded ← sequential
- Between-thread – ***not deterministic***
  - Impacted by scheduling and actual CPU load
  - e.g. two threads A and B started from main thread
    - Either A or B could begin to run first

Multithreading – Data Race

- Race condition
  - Two threads attempting to *update* the same data at the same time
  - Difficult to reproduce

```
void produce()
{
  …
  ++numItems;  // ← increase count, may lead to data race
}

void consume()
{
  …
  --numItems;  // ← decrease count, may lead to data race
}
```

Programming Activity 2

- Write one multithreaded program to find the maximum value in one array
  - Using two child threads
  - Avoid data race

Programming Activity 3

- Rewrite the program you developed in Programming Activity 2 so that the number of child threads can be flexible and controlled by command-line argument
  - Use ArrayAdd4 as reference
  - Avoid data race