Introduction to Parallel Computing

Overview of the Talk

- **What is Parallel Computing?**
- **Applications**
- **A brief history**
- **Latency vs. Bandwidth**
- **Harnessing Multi-cores**
- **Challenges of Parallel Programming**
- **Why Parallel Programming is difficult**

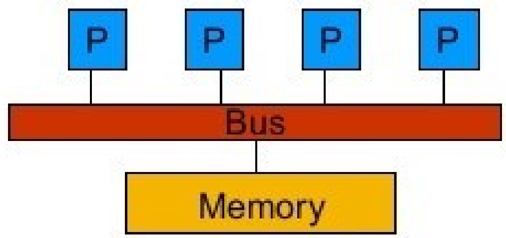
What is Parallel Computing?

simultaneous use of multiple compute resources to solve a computational problem

What is Parallel Computing?

Parallel Computing vs. Distributed Computing

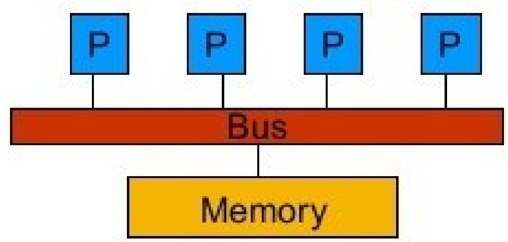
- **Parallel Computing** refers to a model in which the computation is divided among several processors sharing the same memory.



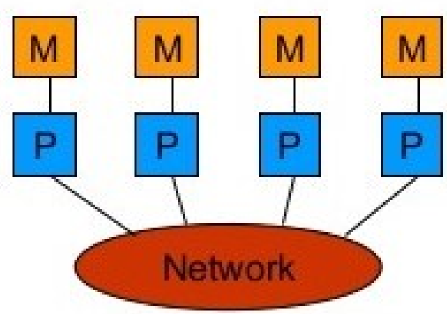
What is Parallel Computing?

Parallel Computing vs. Distributed Computing

- **Parallel Computing** refers to a model in which the computation is divided among several processors sharing the same memory.



- **Distributed Computing** refers to the model in which each processor has its own private memory. Information is exchanged by passing messages between the processors.



Applications

- Complex problems require more computing power
  - Climate Modeling (weather forecasting)
  - Geophysics simulation (earthquake/tsunami prediction)
  - Structure or flow simulation (crash test)
  - Large data analysis (LHC)
  - Military applications (crypto analysis)

First Software Crisis: 1960s-70s

- **Problem:** Assembly Language Programming
  - Computers could handle larger more complex programs
  - Needed to get Abstraction and Portability without losing Performance
- **Solution:** High-level languages for von-Neumann machines
  - FORTRAN and C Provided “common machine language” for uniprocessors
  - Single memory image. Single flow of control

Second Software Crisis: 80s and '90s

Problem:

- Inability to build and maintain complex and robust applications requiring multi-million lines of code developed by hundreds of programmers
  - Needed to get Composability, Malleability and Maintainability
  - High-performance was not an issue → left to Moore's Law
- Programming in the Large Vs Programming in the Small
  - Coordination, discipline of software system management.
  - software management needed to focus from software requirements to realization and at the same time manage migration of systems to new technology or new requirements.

Solution:

- Object Oriented Programming - C++, C# and Java
- Better tools
- Better software engineering methodology
- Design patterns, specification, testing, code reviews

Emerging Software Crisis: 2002 – 20??

Problem: Sequential performance is left behind by Moore's law

– Needed continuous and reasonable performance improvements
  - To support new features to support larger datasets
  - While sustaining portability, malleability and maintainability without unduly increasing complexity faced by the programmer

– Critical to keep-up with the current rate of evolution in software

General-purpose unicores stopped historic performance scaling

- Power consumption → Power Wall

General-purpose unicores stopped historic performance scaling

- Power consumption → Power Wall
- DRAM access latency → Memory Wall

Memory Latency

Example: Consider a processor operating at GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor is capable of executing four instructions in each cycle of 1 ns.

Observations:

- The peak processor rating is 4 GFLOPS
- Since the memory latency is equal to100 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.

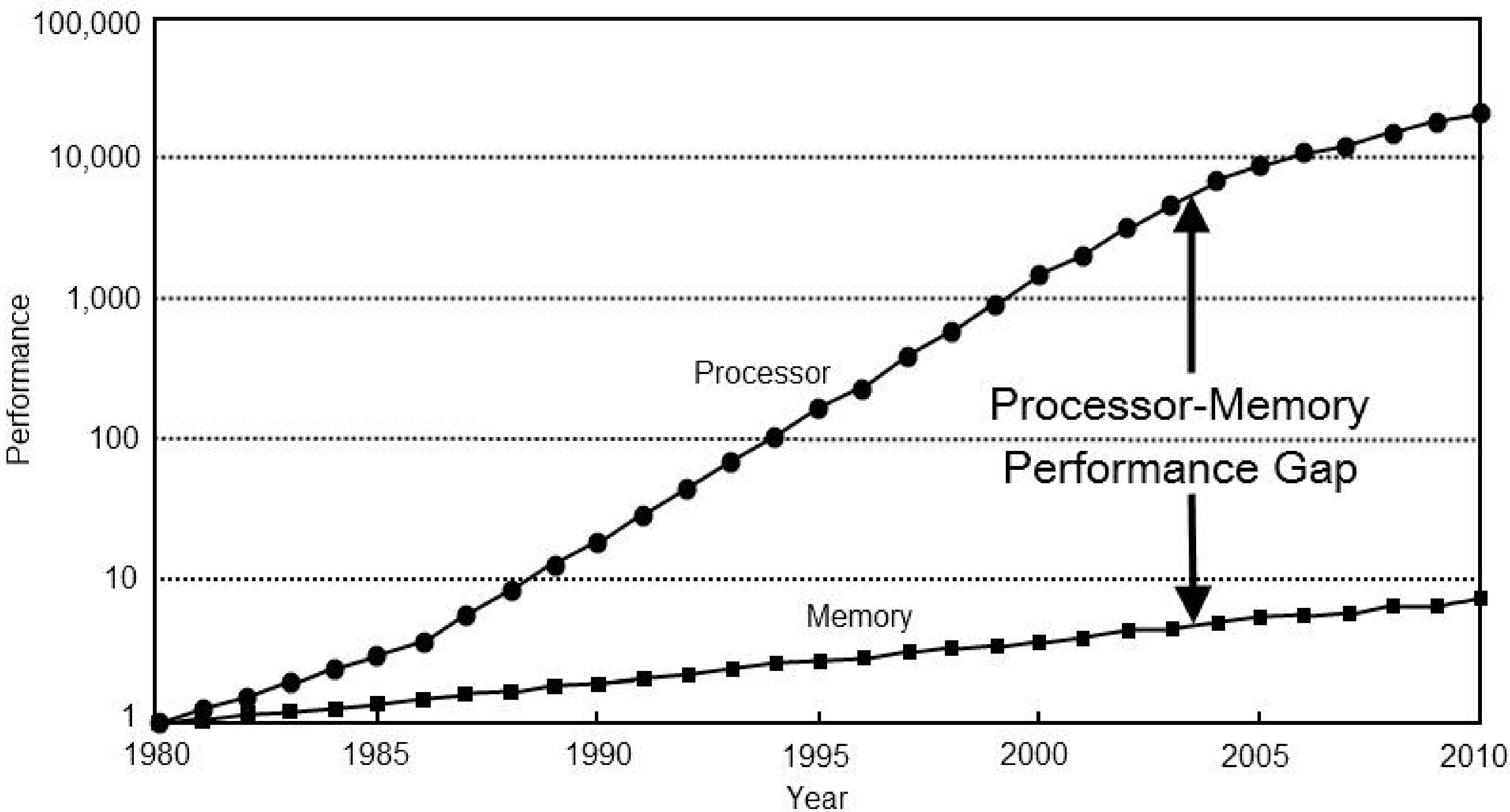
Memory Latency

Example: Consider a processor operating at GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor is capable of executing four instructions in each cycle of 1 ns.

Observations:

- The peak processor rating is 4 GFLOPS
- Since the memory latency is equal to100 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



General-purpose unicores stopped historic performance scaling

- Power consumption → Power Wall
- DRAM access latency → Memory Wall
- instruction-level parallelism → ILP Wall

Power Wall + Memory Wall + ILP Wall = Brick Wall

- End of uniprocessors and faster clock rates

Power Wall + Memory Wall + ILP Wall = Brick Wall

- End of uniprocessors and faster clock rates

↓

Multicore Architectures

- "New" Moore's law is 2x processors or "cores" per socket every 2 years

Latency vs Bandwidth

Harnessing Multi-cores

- Having an n-core computer should result in n-times increase in performance

Harnessing Multi-cores

- Having an n-core computer should result in n-times increase in performance

Problem

Not possible - communication and synchronization costs

Harnessing Multi-cores

- Having an n-core computer should result in n-times increase in performance

Problem

Not possible - communication and synchronization costs

Solution

Utilize as much parallelism as possible

Harnessing Multi-cores

Different forms of parallelism :

1) **Instruction-level Parallelism**

- number of instructions that can be executed in parallel
- Superscalar architectures exploit ILP

⬇

consists of multiple computational units, that allow multiple instructions to be issued in the same clock cycle.

Harnessing Multi-cores

Different forms of parallelism :

2) Data-level Parallelism

- the same instruction is performed on different data in parallel.
- GPUs exploit DLP.
- CUDA - a parallel programming framework that supports DLP.

Harnessing Multi-cores

Different forms of parallelism :

3) **Thread-level Parallelism**

- Multiple threads are executed in parallel
- Multiprocessors/ Multi-core architectures exploit TLP

Challenges of Parallel Programming

i) Concurrency

i) Concurrency

Concurrency

- Different problems inherently have differing amounts of concurrency

Why do we care?

- A larger sequential part implies reduced performance
- Amdahl’s law: this relation is not linear…

i) Concurrency- Amdahl's law

$$\text{Speedup} = \frac{\text{OldExecutionTime}}{\text{NewExecutionTime}}$$

$$= \frac{1}{1 - p + \frac{p}{n}}$$

i) Concurrency- Amdahl's law

Parallel fraction

$$\text{Speedup} = \frac{1}{1 - p + \frac{\boxed{p}}{n}}$$

i) Concurrency- Amdahl's law

Sequential fraction

Parallel fraction

$$\text{Speedup} = \frac{1}{\boxed{1-p} + \frac{\boxed{p}}{n}}$$

i) Concurrency- Amdahl's law

Sequential fraction

Parallel fraction

$$\text{Speedup} = \frac{1}{\boxed{1-p} + \frac{\boxed{p}}{\boxed{n}}}$$

Number of processors

i) Concurrency- Amdahl's law

Ten processors
60% concurrent, 40% sequential
How close to 10-fold speedup?

i) Concurrency- Amdahl's law

Ten processors
60% concurrent, 40% sequential
How close to 10-fold speedup?

$$\text{Speedup}=2.17=\frac{1}{1-0.6+\frac{0.6}{10}}$$

i) Concurrency- Amdahl's law

Ten processors
80% concurrent, 20% sequential
How close to 10-fold speedup?

i) Concurrency- Amdahl's law

Ten processors
80% concurrent, 20% sequential
How close to 10-fold speedup?

$$\text{Speedup}=3.57=\frac{1}{1-0.8+\frac{0.8}{10}}$$

i) Concurrency- Amdahl's law

Ten processors
90% concurrent, 10% sequential
How close to 10-fold speedup?

i) Concurrency- Amdahl's law

Ten processors
90% concurrent, 10% sequential
How close to 10-fold speedup?

$$\text{Speedup}=5.26=\frac{1}{1-0.9+\frac{0.9}{10}}$$

i) Concurrency- Amdahl's law

Ten processors
99% concurrent, 01% sequential
How close to 10-fold speedup?

i) Concurrency- Amdahl's law

Ten processors
99% concurrent, 01% sequential
How close to 10-fold speedup?

$$\text{Speedup}=9.17=\frac{1}{1-0.99+\frac{0.99}{10}}$$

i) Concurrency- Amdahl's law

Summary

Making good use of our multiple processors (cores) means

- Finding ways to effectively parallelize our code
- Minimize sequential parts
- Reduce idle time in which threads **wait**
- The % that is not easy to make concurrent yet may have a large impact on overall speedup

ii) Load Balancing

Load Balancing

Dividing work between the processors as evenly as possible to minimize idle time on each processor

- Minimize idle time on each processor
- reduce the total execution time

ii) Load Balancing: Primality Testing

Challenge

Print primes from 1 to 10^{10}

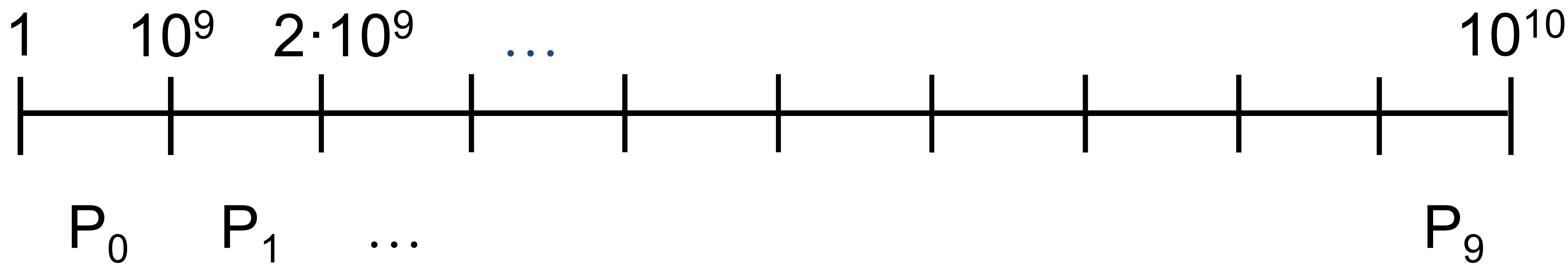
Given

- Ten-processor multiprocessor
- One thread per processor

Goal

Get ten-fold speedup (or close)

ii) Load Balancing: Primality Testing



Split the work evenly

Each thread tests range of 10^9

ii) Load Balancing: Primality Testing

```
void primePrint {
  int i = ThreadID.get(); // IDs in {0..9}
  for (j = i*10^9+1, j<(i+1)*10^9; j++) {
    if (isPrime(j))
      print(j);
  }
}
```

ii) Load Balancing: Primality Testing

Issues

- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict
- Need *dynamic* load balancing

ii) Load Balancing

TA Problem

iii) Synchronization

Current programming practice – Locks, conditions(monitors) for synchronization.

problems:

- Difficult to program
- Granularity
- Deadlocks
- Not composable - correct fragments may fail when combined.

iv) Communication

- Locality
- Data Distribution

Why parallel programming is difficult?

1) Finding parallelism
   - Difficult when the task isn’t fully parallelizable

Why parallel programming is difficult?

1) Finding parallelism
   - Difficult when the task isn’t fully parallelizable

2) Debugging
   - Difficult since bugs may not be reproducible on every run

Why parallel programming is difficult?

Example: X=X+1;

Assembly code:

T0		T1	
A	Load X, R0	D	Load X, R1
B	Increment R0	E	Increment R1
C	Store R0, X	F	Store R1, X

Executions:

ABCDEF: X=2 [Correct]

DEAFBC: X=1

ADBCEF: X=1

Why parallel programming is difficult?

3) Scalability

- Single threaded applications benefit from increasing performance on single cores.
- Multi-threaded applications may not benefit from increased number of cores, and may even slow down in worst case.
- Applications have to be scalable.

References

- *Nir Shavit, Maurice Herlihy, The art of multiprocessor programming*
- *Saman Amarasinghe, 6.189 Multicore Programming Primer, January (IAP) 2007, MIT*
- *Blaise Barney, Introduction to Parallel Computing, Lawrence Livermore National Laboratory*

THANK YOU