Concurrency

Performance Engineering: Theory & Practice

Concurrency

- Definition: Speeding up processing by using multiple tasks executing in parallel
  - Motivation: overcome inherent limitations of serial processing:
    - **Hardware**: CPU clock speed, Instruction Execution rate, IO service time, Network latency, etc.
      - e.g.,
        - Symmetric Multi-processing (SMP)
        - Superscalar; Vector instructions
        - Massively Parallel Processors (MPP)
        - Simultaneous Multi-Threading (SMT): Intel Hyper-Threading (HT)
    - **Software**: HPC, Hadoop, MapReduce
    - **Wetware**: most humans are much more comfortable thinking serially

Concurrency

- Parallel Computation
  - Huge topic in CS, encompassing Hardware, Software, compilers, algorithms, etc.

- How can parallelism be applied to different workloads?

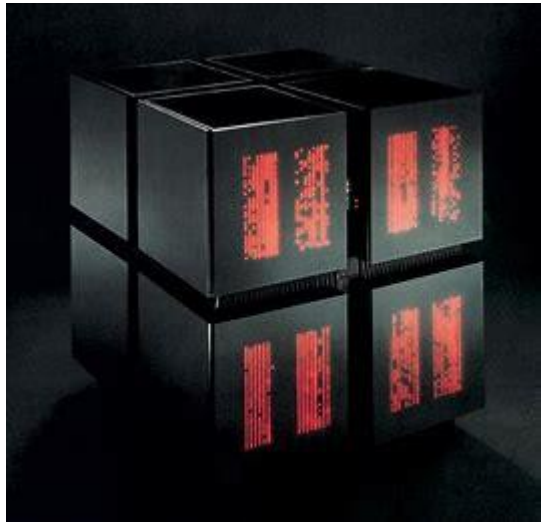
  - inherent parallelism (e.g., vector operations, transaction processing)
  - parallel algorithms (e.g., "Divide & Conquer", SIMD)
  - serial algorithms that (so far) resist parallelism

- What are some of the key performance limitations parallel computing encounters?

SIMD

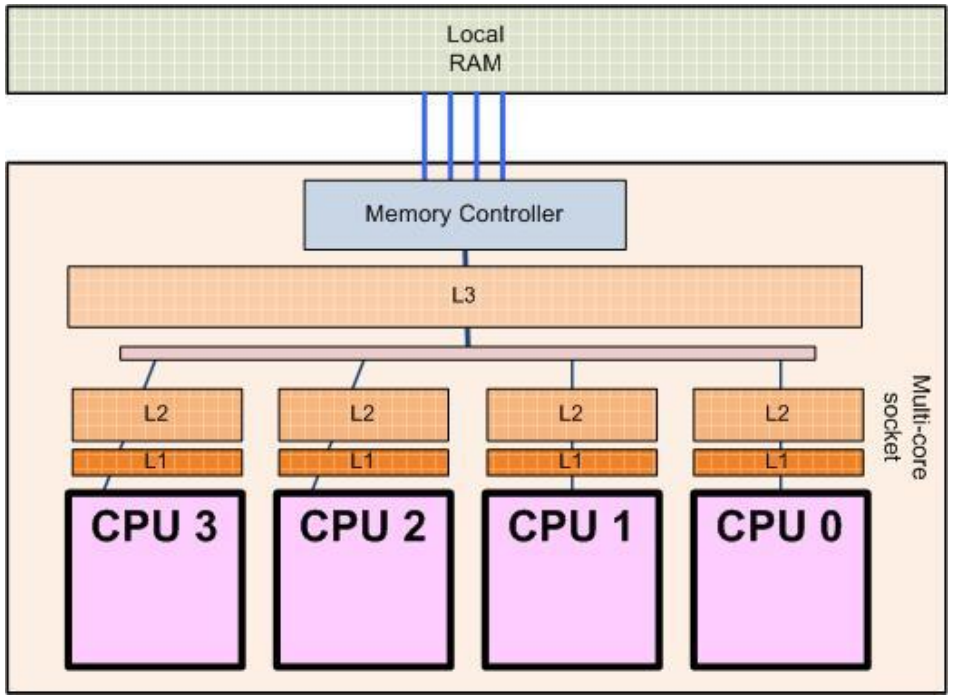
- Vector graphics hardware
  - modern GPUs (e.g., nVidia GeForce, CUDA)

- Explicit parallelism
  - Fork : Join
- "Divide and Conquer" pattern
  - e.g., parallel Search
    - Danny Hillis's Connection Machine demonstration
    - Massively Parallel Processors (MPPs)
  - MapReduce (see Hadoop)



General purpose CPU hardware evolution

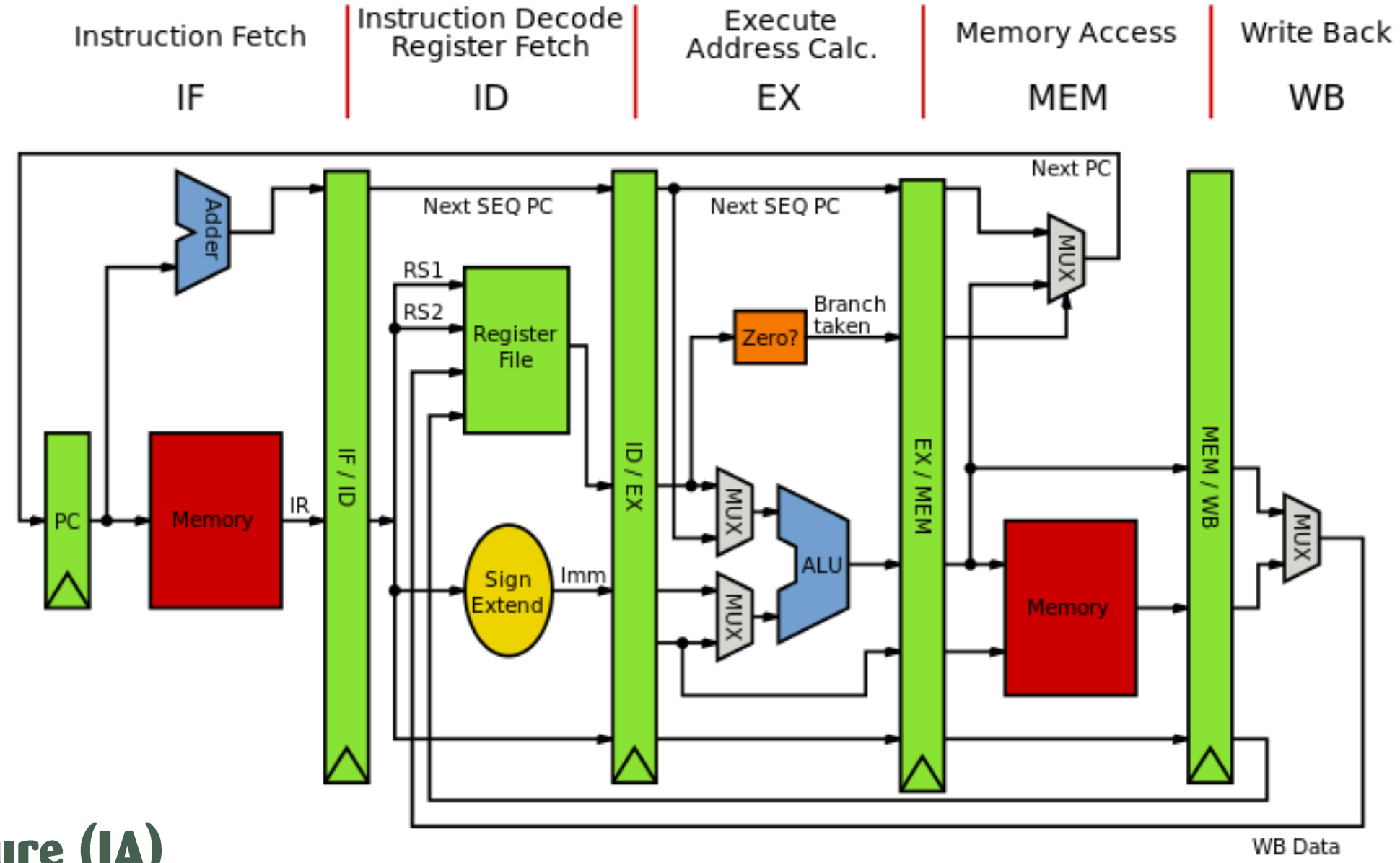
- Symmetric Multiprocessors
  - Multi-Core designs
- Pipelining
- Superscalar
- SMT



- RISC vs CISC

- CISC Intel CPUs transform complex instructions into simple, RISC machine instructions (μops)

Pipelining



Intel Architecture (IA)

Pipelining

- Pipeline stages (Intel)

1. Instruction Fetch
2. Instruction Decode
3. Execute
4. Memory access
5. Write-back

- In theory, execute one pipeline stage per clock
- if successful, complete execution of an instruction each clock!

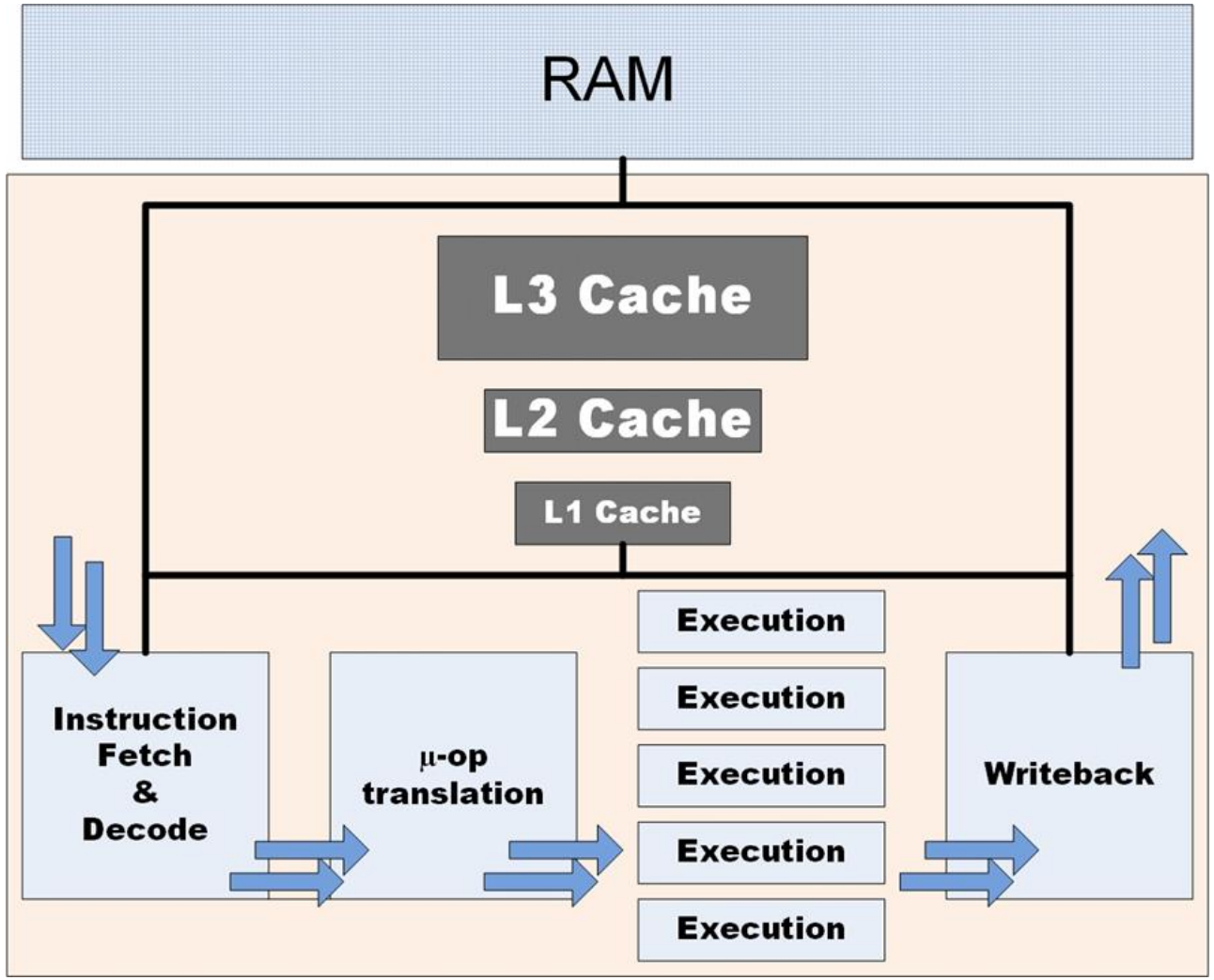
IER = 1/clock

Pipelining

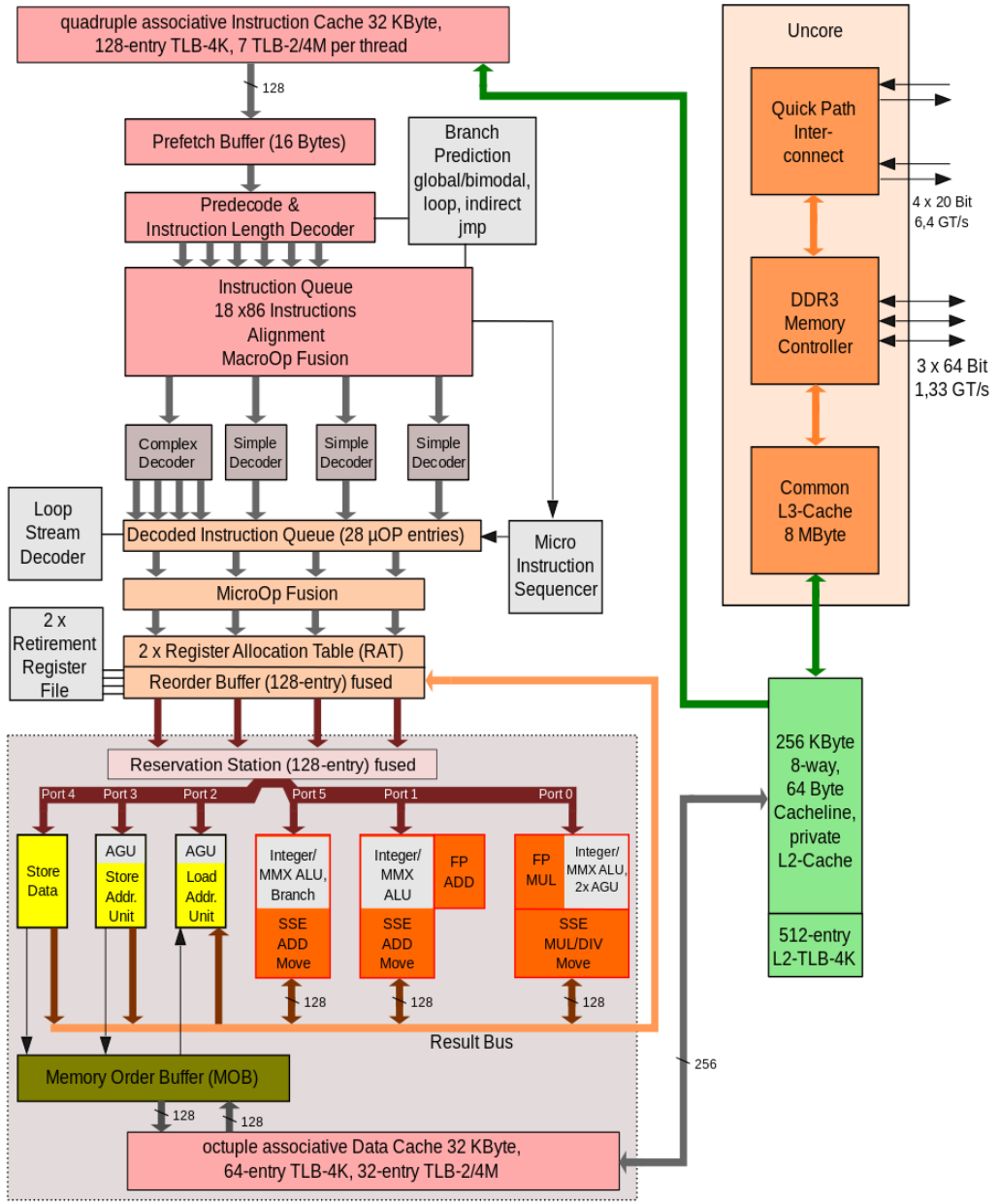
- In theory, IER = 1/clock
- exploits Instruction-level parallelism
- In practice,
  - pipeline **stalls**
  - pipeline **hazards**: a subsequent instruction that relies on a value written/updated by a current instruction
  - mis-predicted branches

	0	1	2	3	4	5	6	7	8	9	10	11
ADD $r0, $r1, $r2	IF	ID	EX	MEM	WB							
SUB $r4, $r0, $r3		IF	*	*	*	ID	EX	MEM	WB			
AND $r5, $r0, $r6			IF	*	*	*	ID	EX	MEM	WB		
OR $r7, $r0, $r8				IF	*	*	*	ID	EX	MEM	WB	
XOR $r9, $r0, $r10					IF	*	*	*	ID	EX	MEM	WB

Pipelining + Parallelism = Superscalar architecture



Intel Nehalem microarchitecture



GT/s: gigatransfers per second

Superscalar architecture = pipelining + parallelism

- Multiple instructions can complete each clock

IER > 1/clock

- Out-of-Order execution
- In-order retirement
- internal pseudo-registers

Simultaneous Multithreading

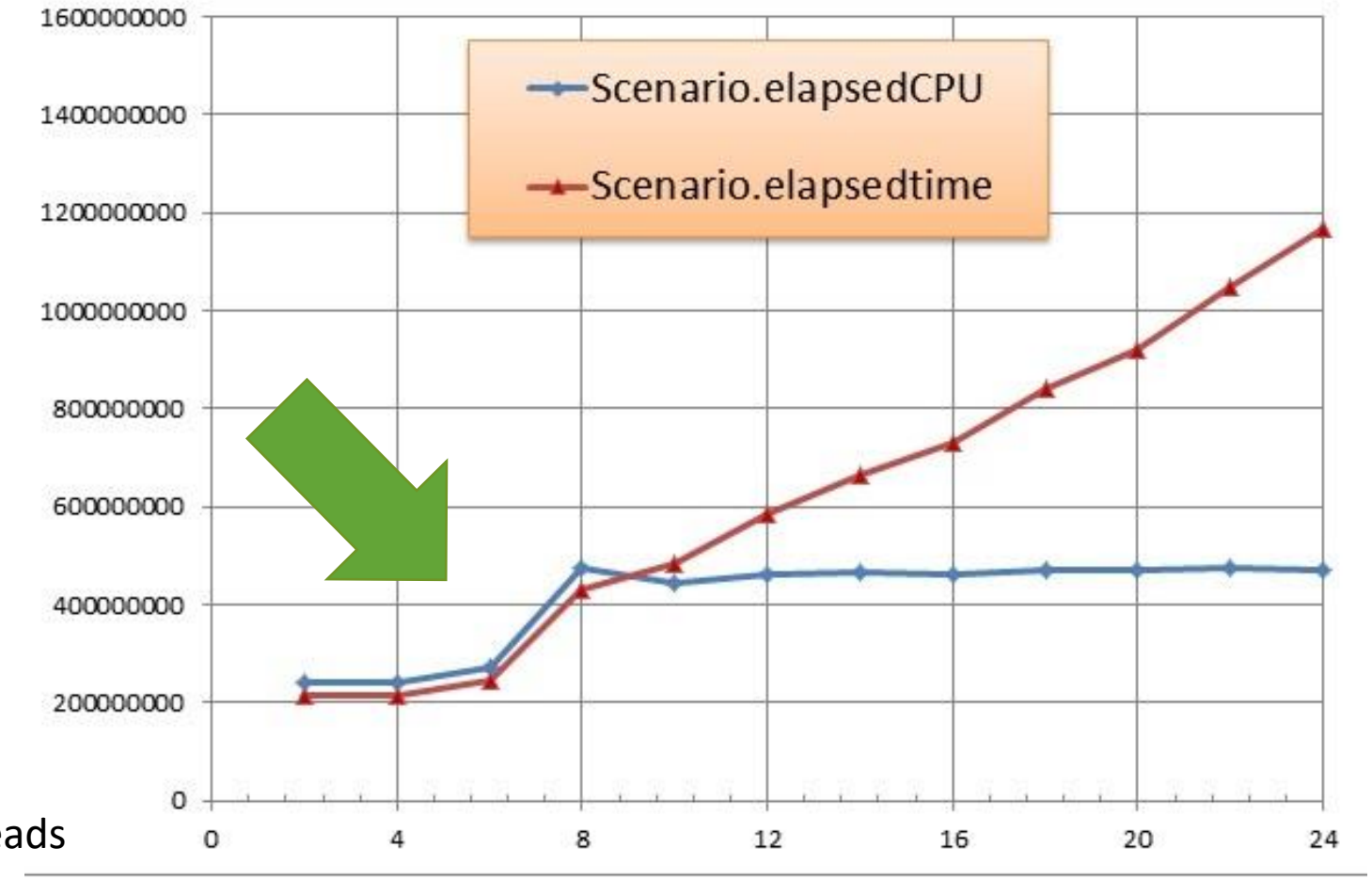
- Overcome the limits of Thread-level parallelism using Two (or more) logical CPUs per physical CPU
  - Multiple instruction execution pipelines per core
  - shared cache, instruction Execution units, etc.
- Looked promising in the Lab:
  - see Susan Eggers, "SMT: Maximizing On-Chip Parallelism," 1995.
- Adopted by Digital and then absorbed into Intel
  - aka, Hyper-Threading (HT)

- Helps many single-user workloads, but contention for shared resources can impact others

Simultaneous Multithreading

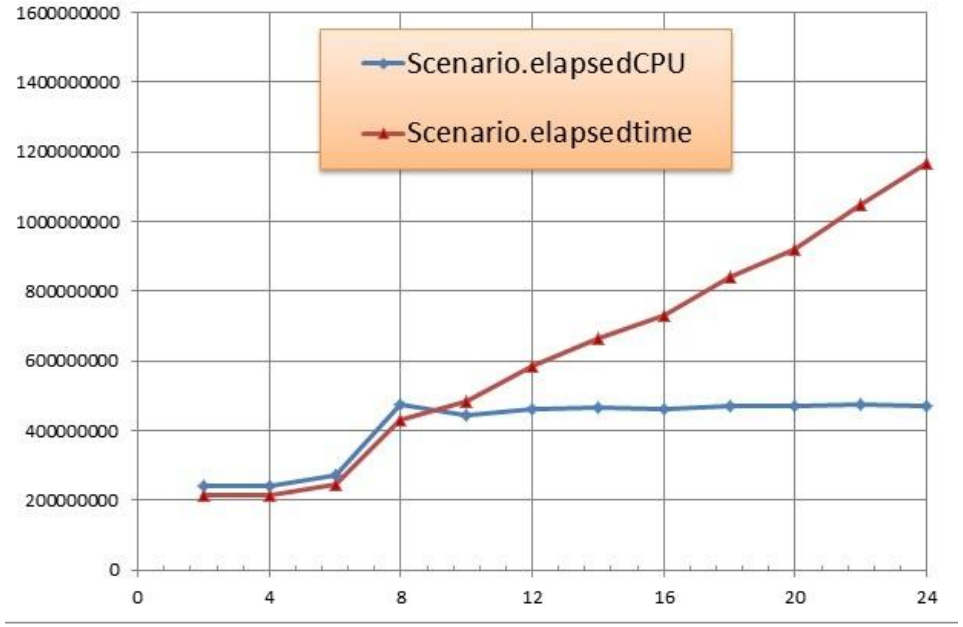
- see https://performancebydesign.blogspot.com/2012/04/mystery-of-scalability-of-cpu-intensive.html

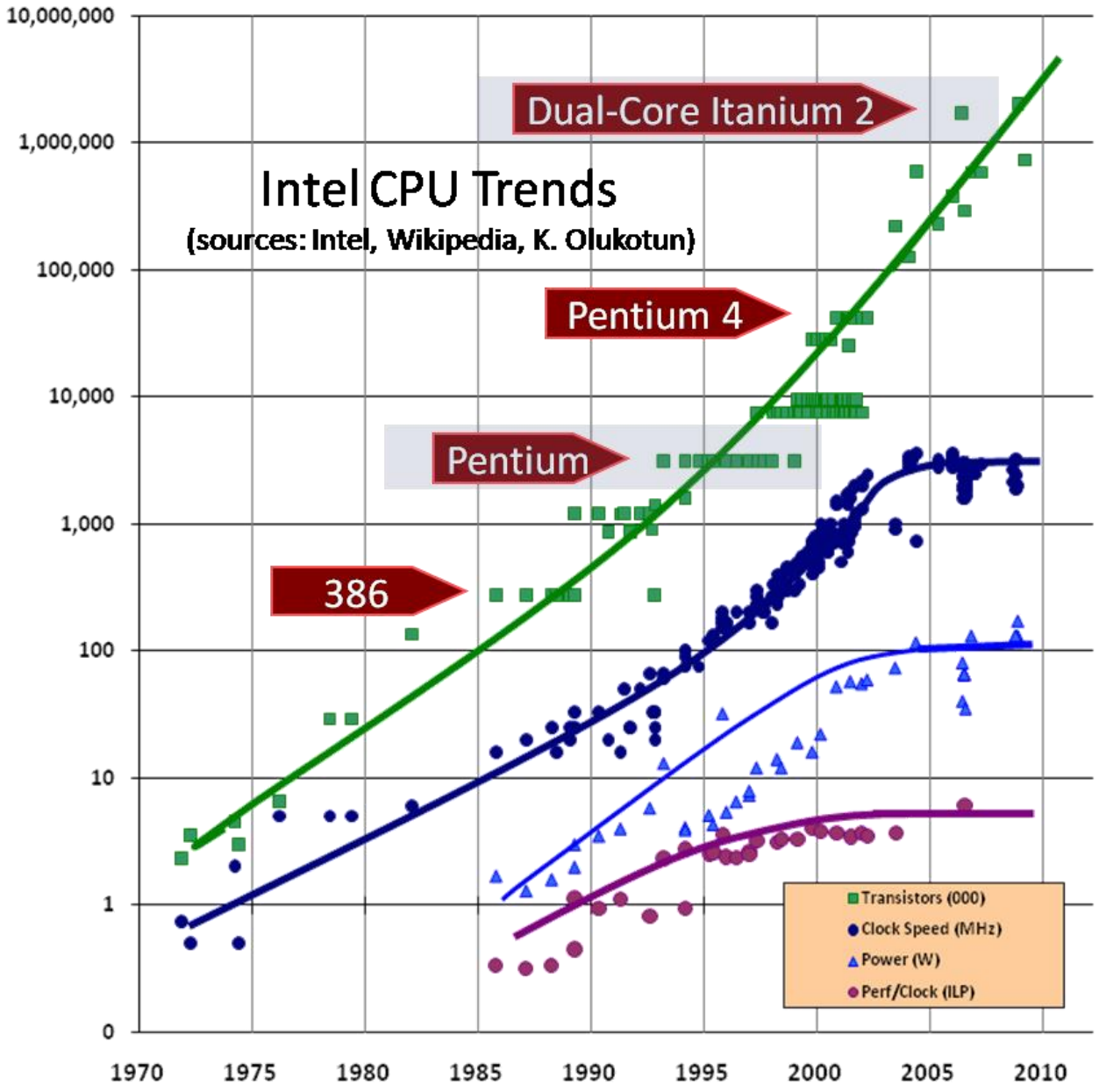
- 4 physical CPUs
- 8 logical CPUs



Simultaneous Multithreading

- Helps many single-user workloads, but contention for shared resources can impact others
  - see https://performancebydesign.blogspot.com/2012/04/mystery-of-scalability-of-cpu-intensive.html
  - Unproductive "Hyper-threading: Off or On" decisions/discussions





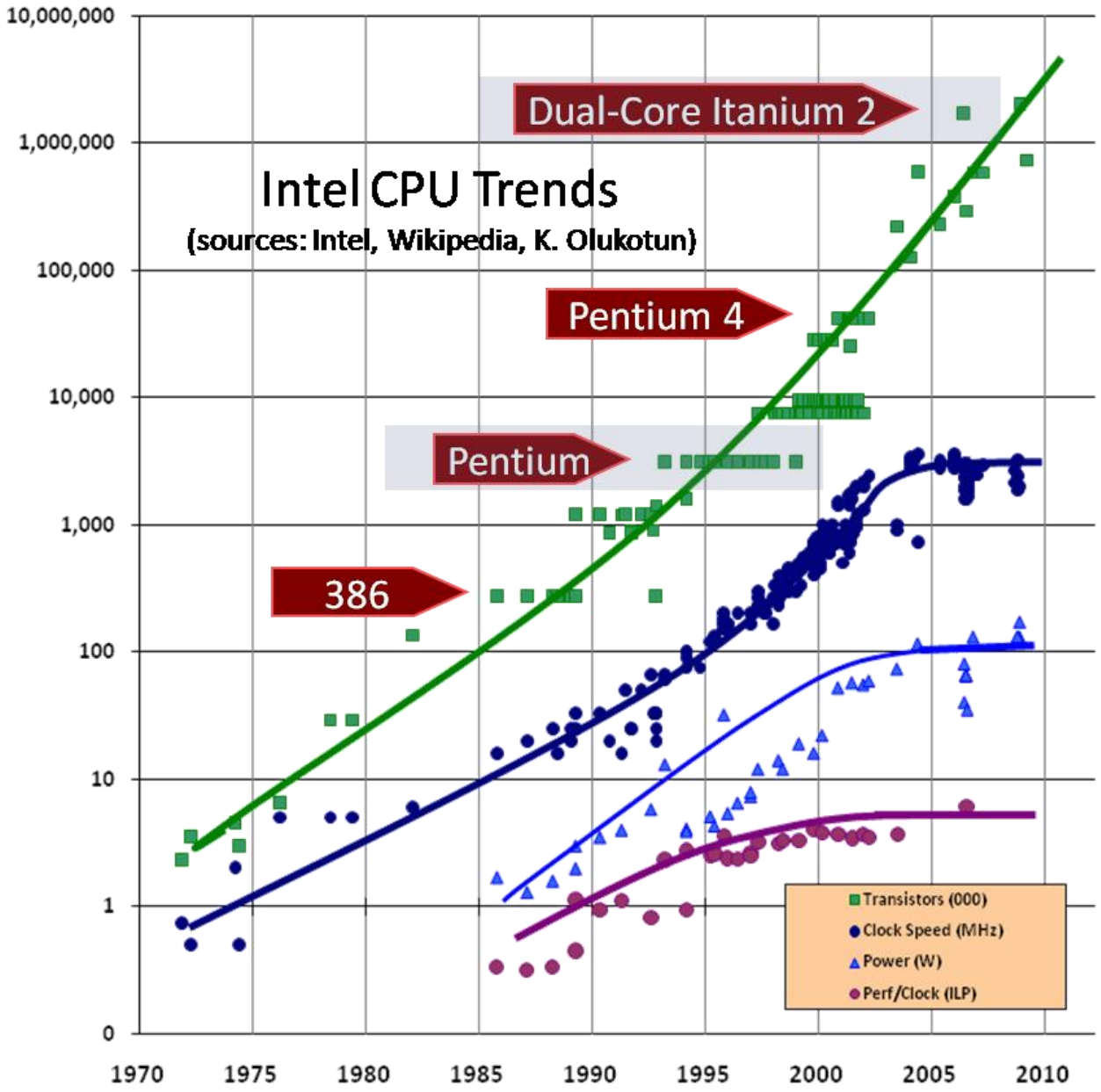
Multi-core

see Herb Sutter, "The Free Lunch Is Over," 2004

Multi-core designs reflect:

- a dearth of good ideas at Intel about what to do with advances in semi-conductor fabrication that increase capacity
- memory performance "wall"
- plus,

$power^3 = clock\ speed$

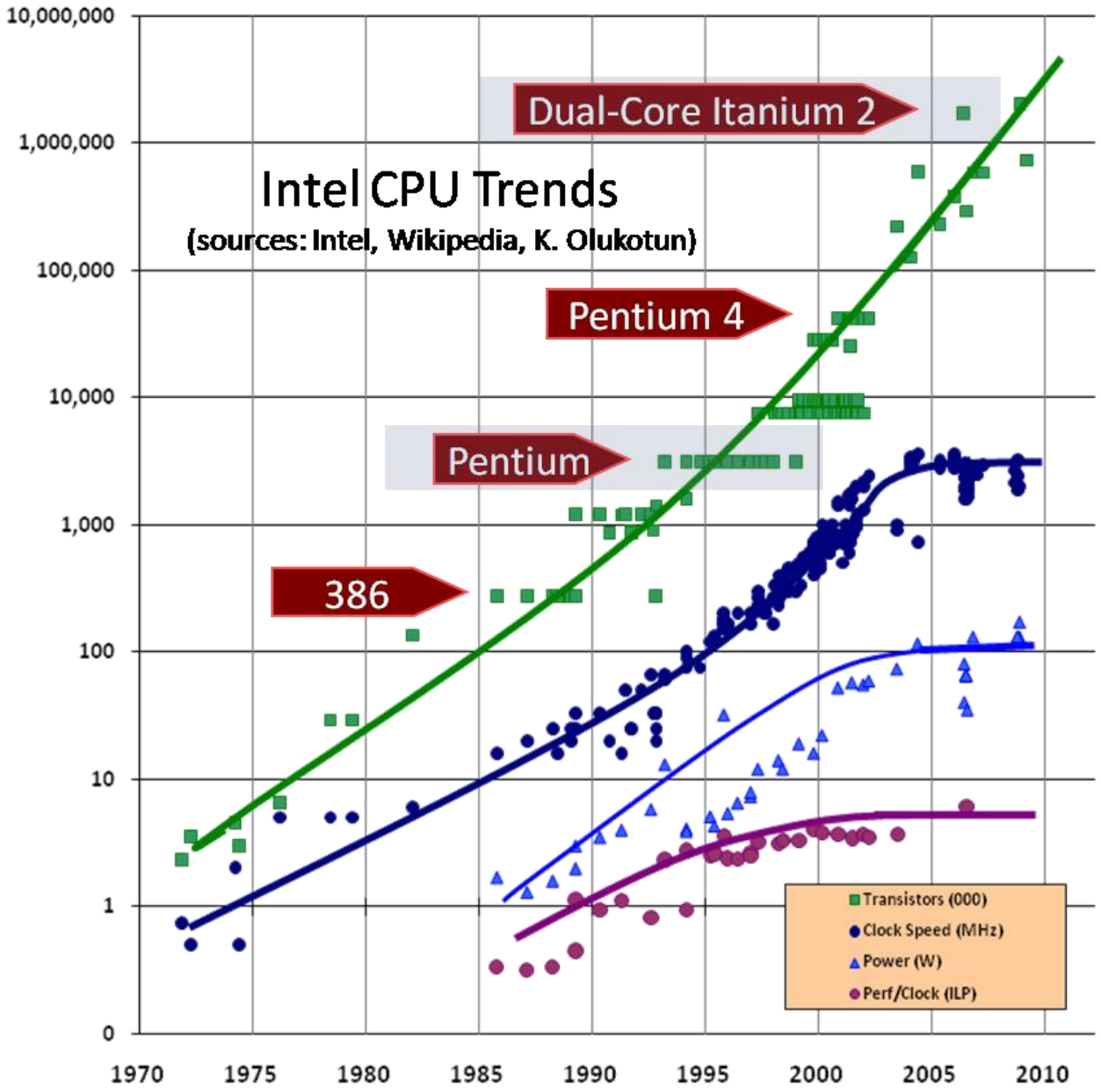


Multi-core

see Herb Sutter, "The Free Lunch Is Over," 2004

Call to Arms:

- CPU clock speed no longer increasing at historical rates
- Instead, more processors per chip
- To speed up processing, figure out how to leverage those additional processors



Multi-core

see Herb Sutter, "The Free Lunch Is Over," 2004

still,

- workloads on single-user devices (phones, tablets, portables) do not warrant having massively parallel processors
- no revolutionary breakthrough in concurrent programming technology/support

Design Patterns for Concurrent programming

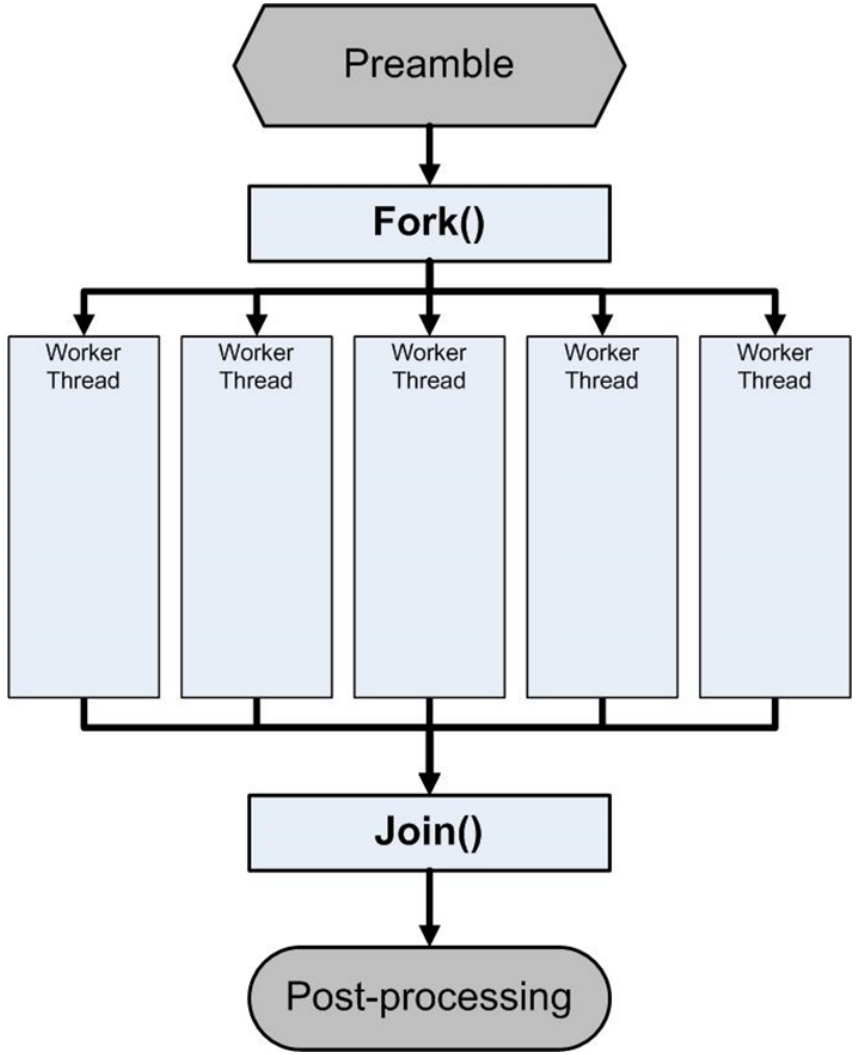
- Parallel.For
- Fork/Join
- Partitioners
- MapReduce (Dean & Ghemawat, see link)
- ProducerConsumer

- Work Queue : Thread Pool
  - e.g., Windows File Server, SQL Server Thread Pool, IIS Worker process
  - .NET Framework ThreadPool

- see Stephen Toub, Patterns of Parallel Programming

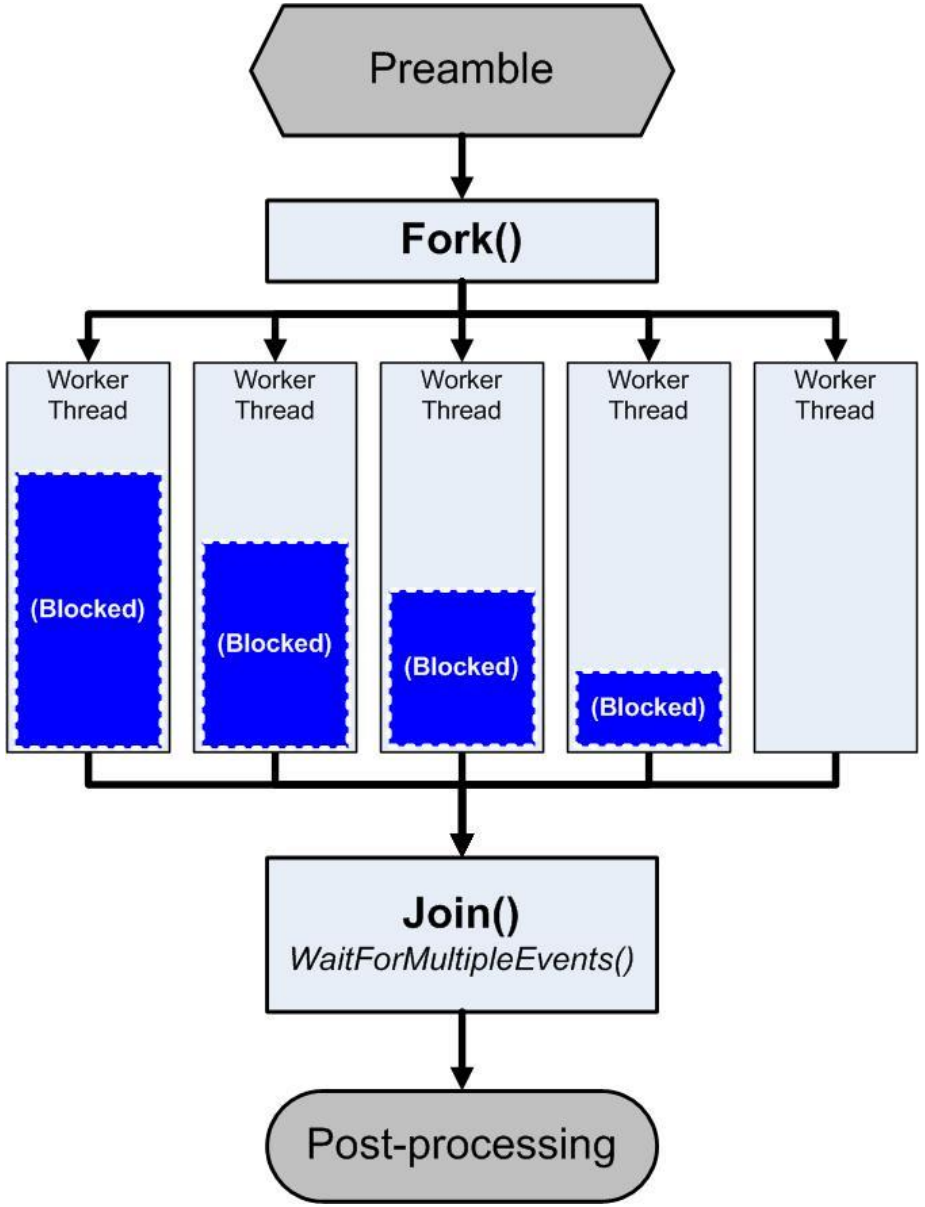
Design Patterns for Concurrent programming

- Parallel.For
  - SIMD –
    - Single Instruction
    - Multiple Data
  - “embarrassingly parallel”
  - see https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-write-a-simple-parallel-foreach-loop
  - runtime library determines how many concurrent tasks to release



Concurrent programming

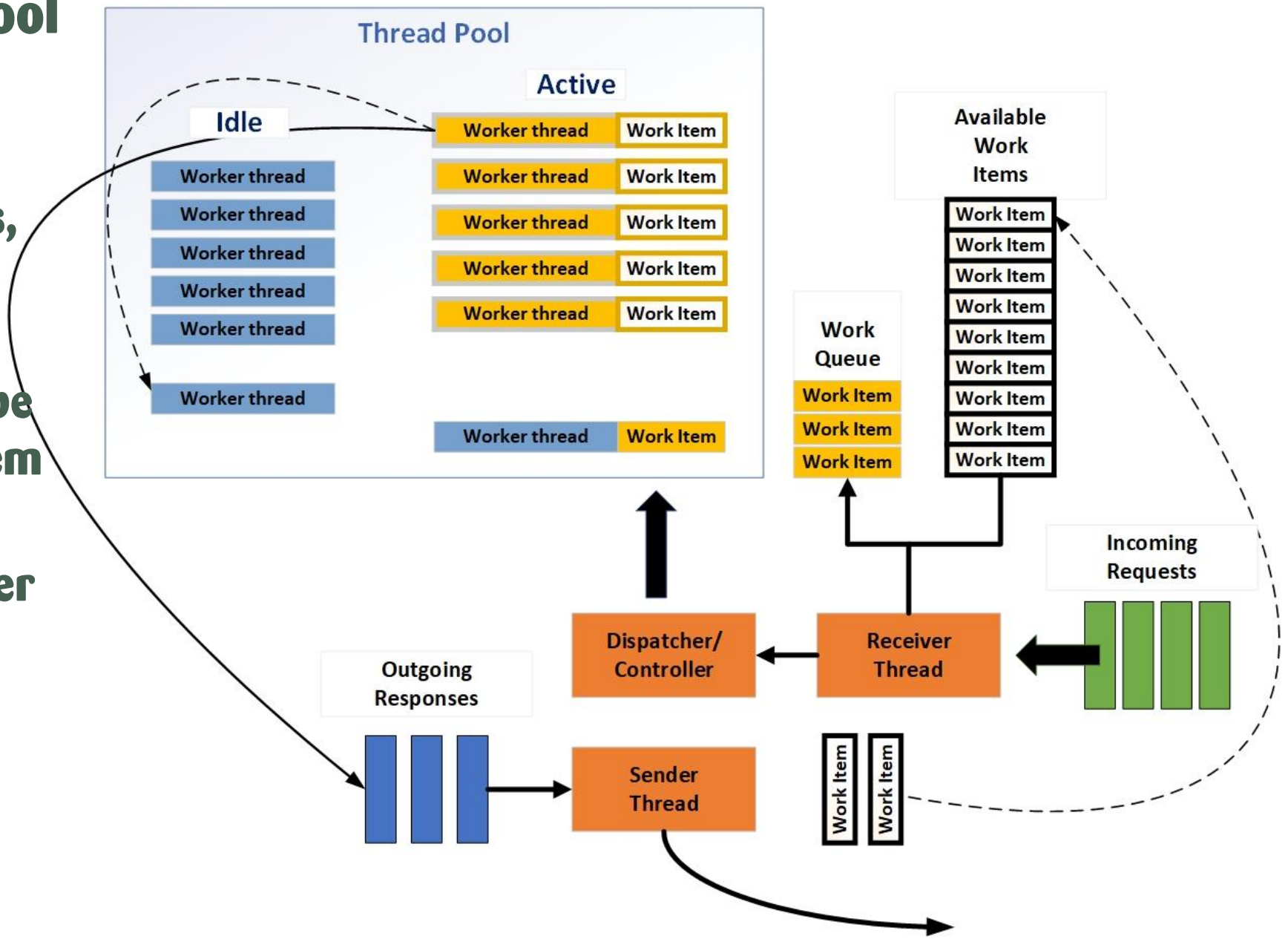
- Parallel.For
  - SIMD –
    - Single Instruction
    - Multiple Data
  - “embarrassingly parallel”
- “Shared Nothing” partitioning/decomposition
- Performance anti-pattern
  - Unbalanced partitioning



Work Queue: Thread pool Design Pattern

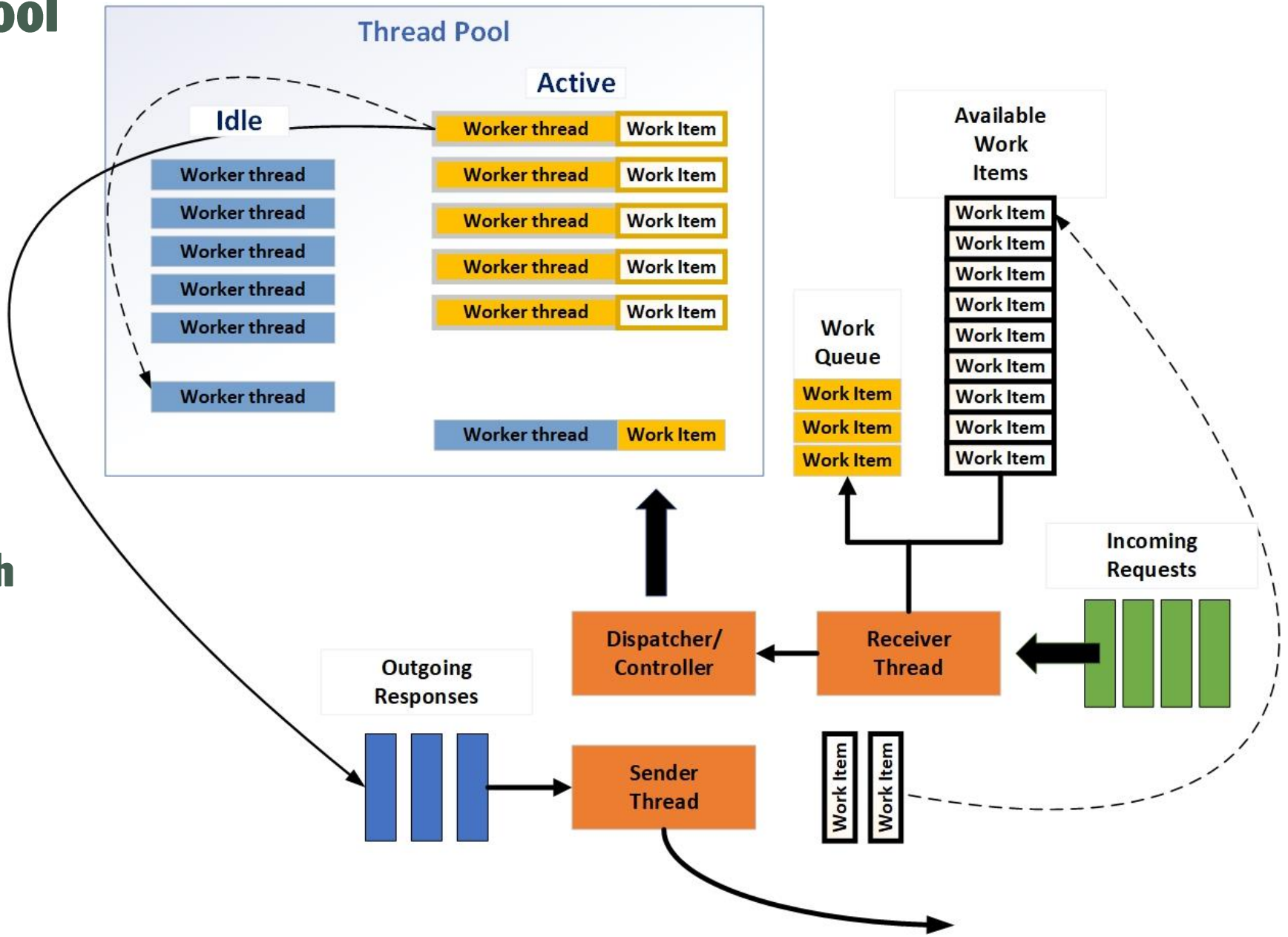
- reusable Worker threads,
  - activated on demand
- reusable Work items
- Any Worker thread can be assigned to any Work item

- Pattern used in multi-user server applications
  - Windows File Server
  - SQL Server
  - IIS



Work Queue: Thread pool Design Pattern

- Tuning parameters:
  - Max threads
  - Max Work Items
- Instrumentation:
  - Active threads
  - Work Items queue length
  - Work Item shortages
  - Completion rate
  - Service Time
  - Response Time



Working with the .NET Framework ThreadPool Object

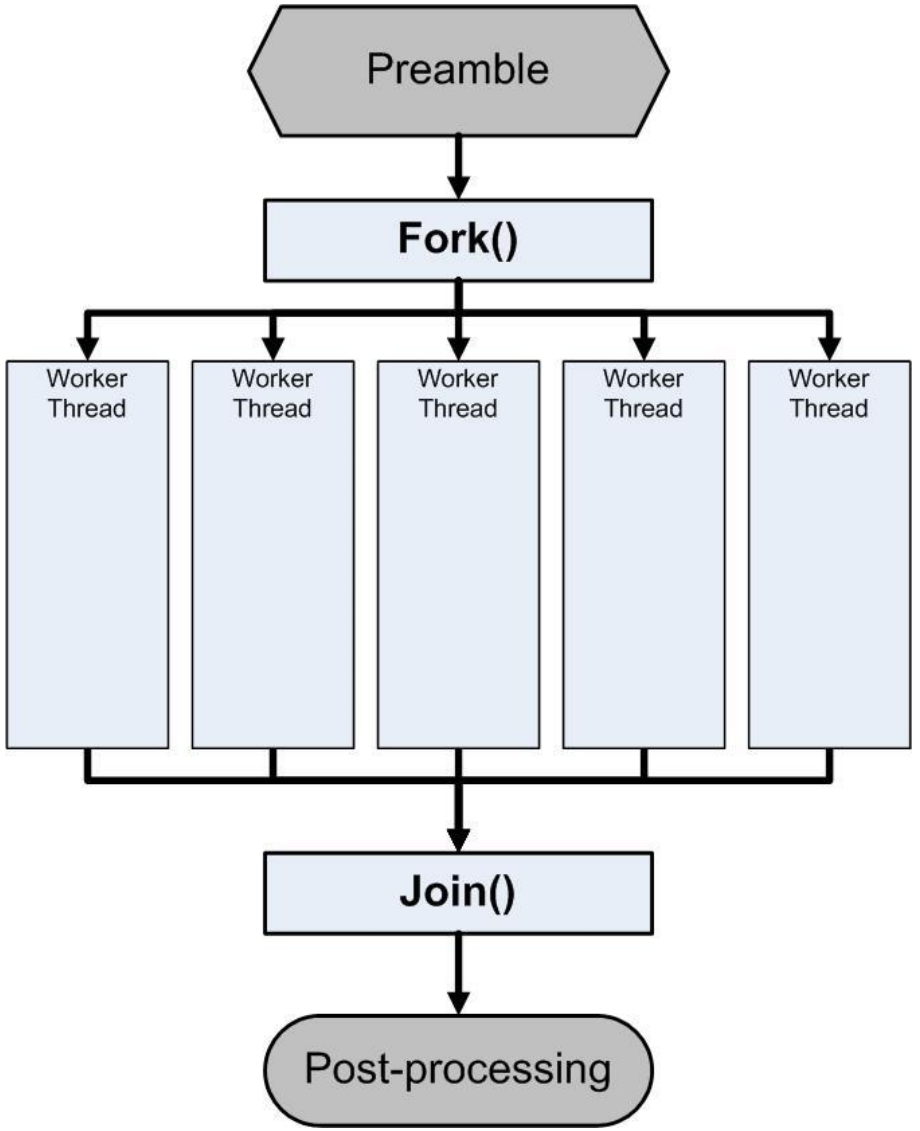
- **ThreadPool.QueueUserWorkItem() Method**
  - **ex:** `ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadProc), taskInfo);`

- **Requires:**
  1. **a class that wraps the Work Item parameter list**
  2. **a delegate that executes in the Worker Thread**
  3. **a Scheduler/Dispatcher**
  4. **an Event handler that runs when each Worker Thread completes:**

```
if Queue<WorkItem>.IsEmpty()
else
  Dispatcher.DispatchNext()
```

The Limits of Parallel Programming

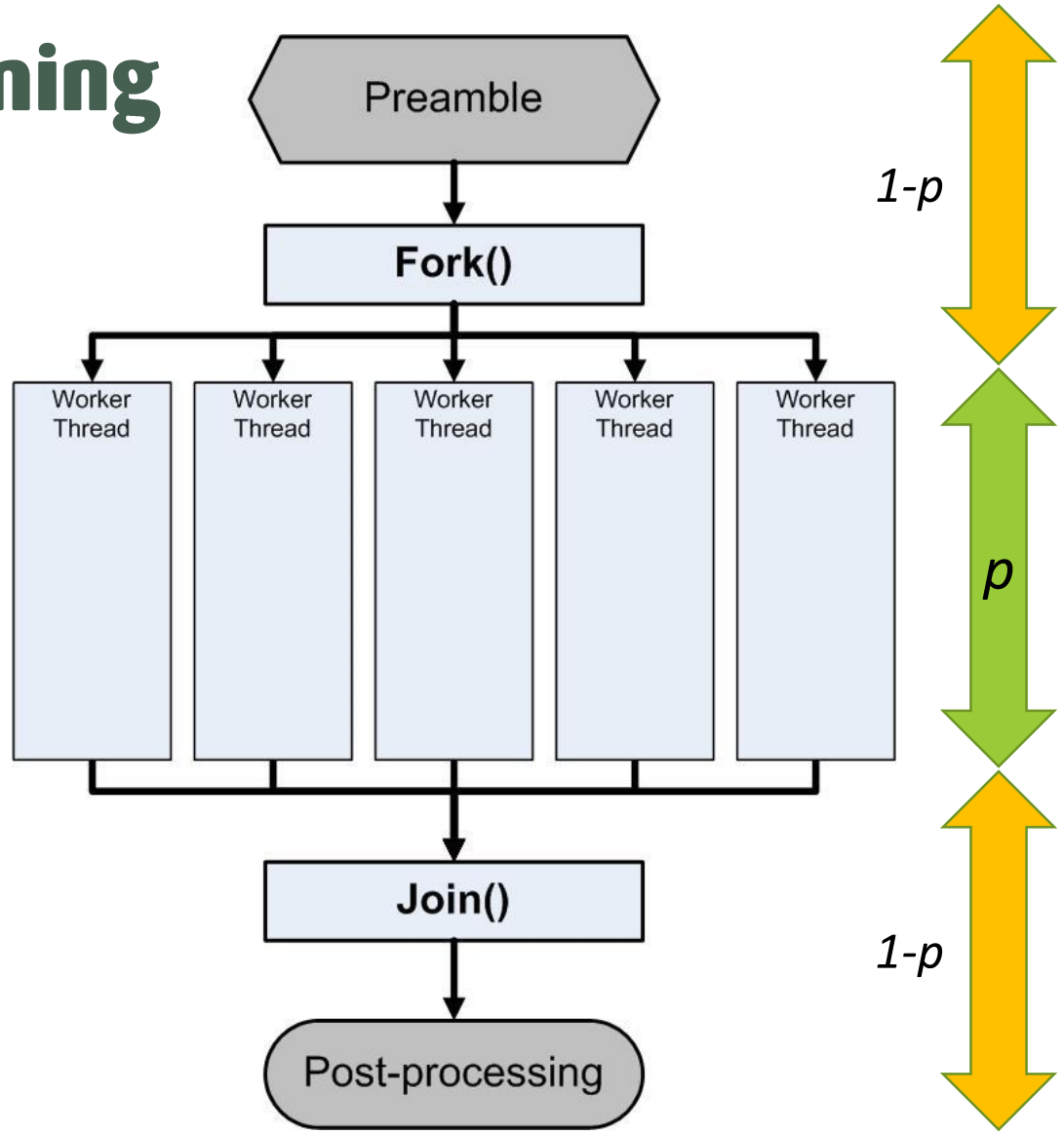
- No reliable method exists to generate a ***parallel*** program from its ***serial*** solution
  - Divide and Conquer pattern
  - SIMD
  - MIMD
- ***non-determinative*** execution
  - difficult to reproduce the ***exact*** sequence of processing events that resulted in an error
  - race conditions, etc.
- A little parallelism may help, but expect diminishing returns from adding more parallel threads



The Limits of Parallel Programming

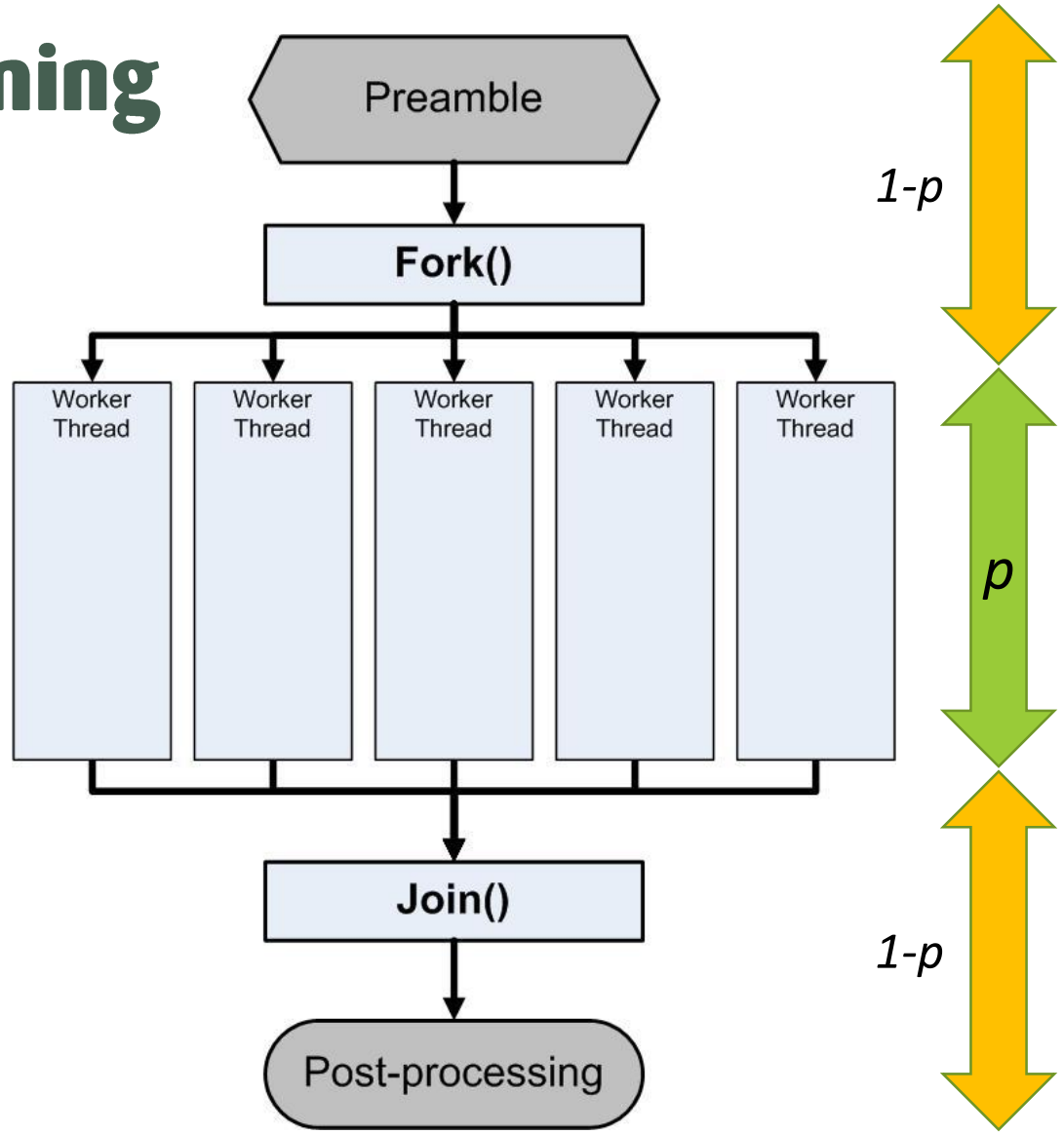
- **Amdahl's Law**
  - ***p*** **is the proportion of the program that can be parallelized**
  - **leaving *(1-p)*, the proportion that runs serially**
  - **Amdahl's Law observes that, regardless of the degree of parallelism, this program cannot execute faster than**

$$s = 1 - p$$



The Limits of Parallel Programming

- Gustafson's counter-argument
  - More optimistic stance to the actual performance of a parallel program



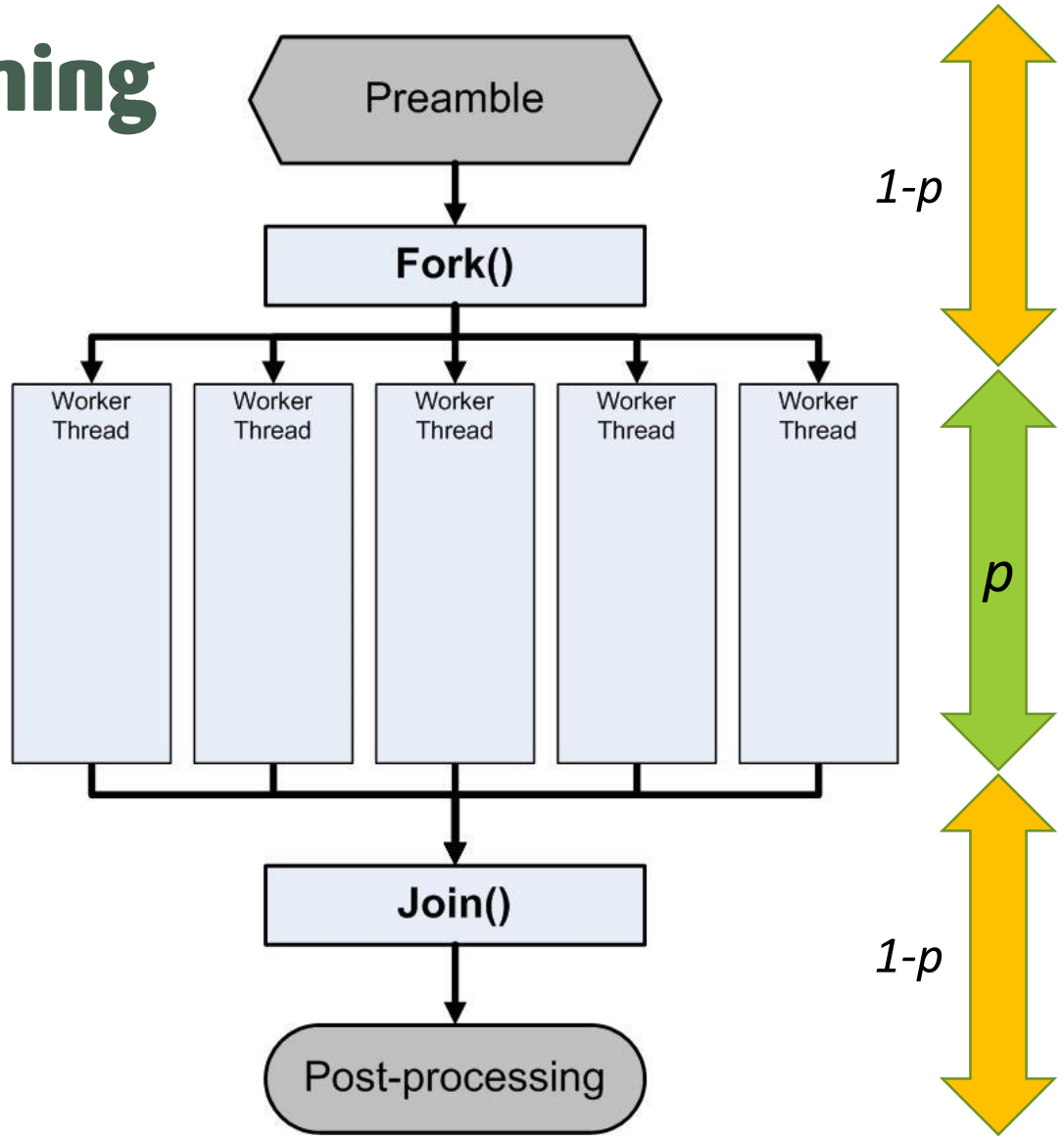
One does not take a fixed-size problem and run it on various numbers of processors except when doing academic research; in practice, the problem size scales with the number of processors. When given a more powerful processor, the problem generally expands to make use of the increased facilities. Users have control over such things as grid resolution, number of timesteps, difference operator complexity, and other parameters that are usually adjusted to allow the program to be run in some desired amount of time. Hence, it may be most realistic to assume that run time, not problem size, is constant.

-- Gustafson, "Reevaluating Amdahl's Law," 1988

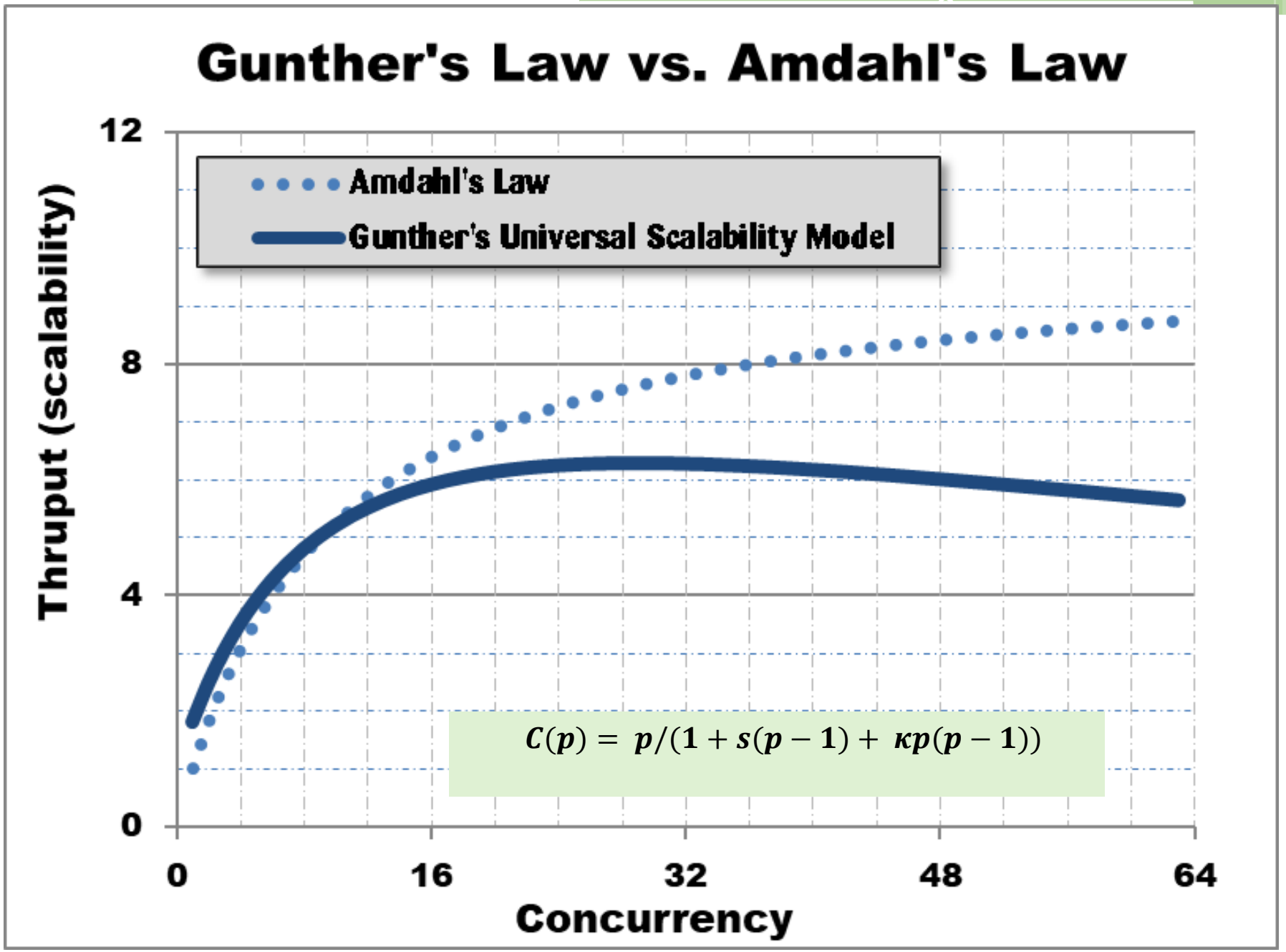
The Limits of Parallel Programming

- **Gunther's Law**
  - Attempts to model the actual performance of a parallel program
  - using ***K***, for coherency delays

$$C(p) = p/(1 + s(p-1) + \kappa p(p-1))$$



Gunther's Law



Model Parameters:

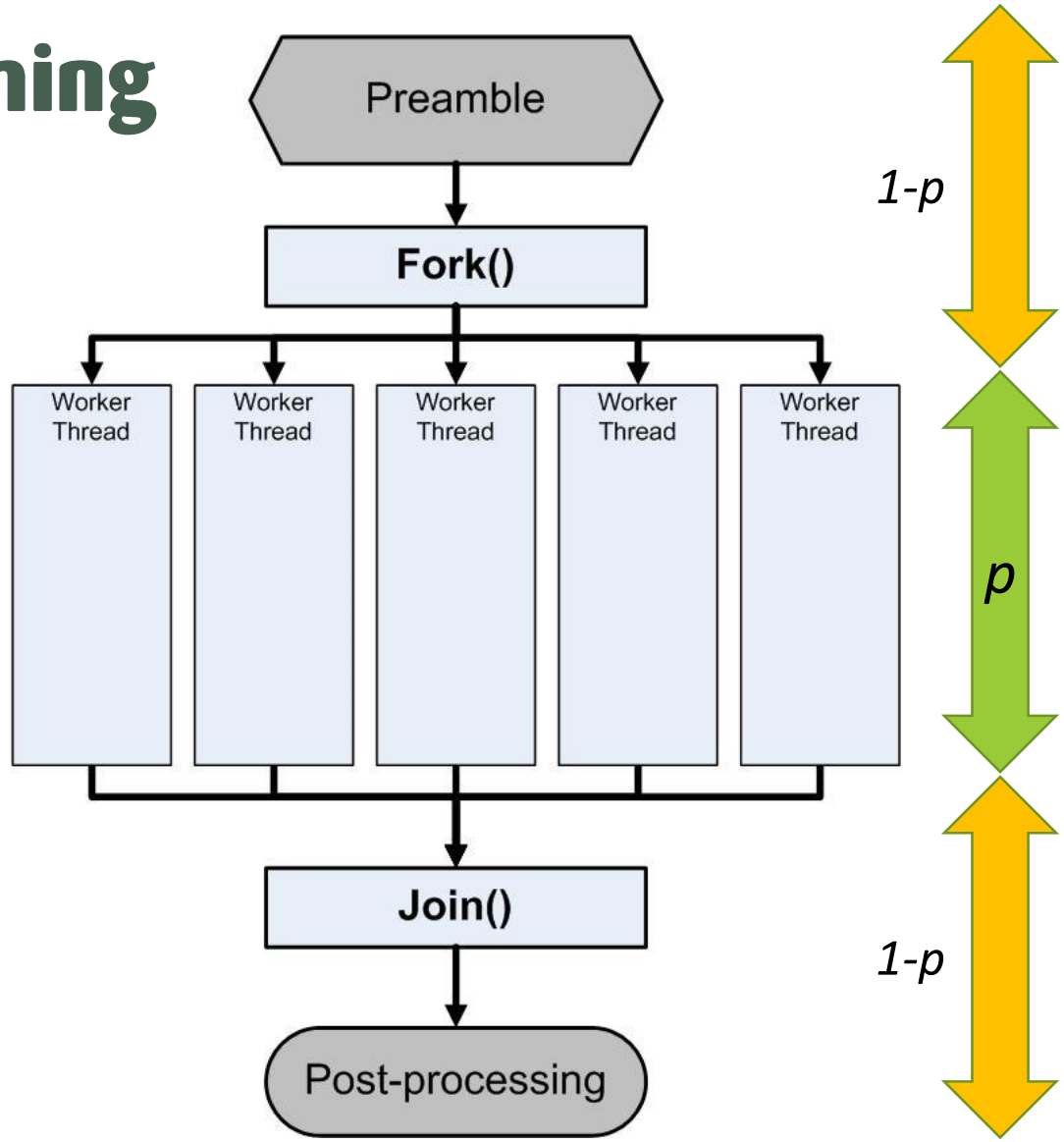
serial portion = 10%
K = 0.001

The Limits of Parallel Programming

- **Gunther's Law**
  - ***K***, represents ***coherency*** delays

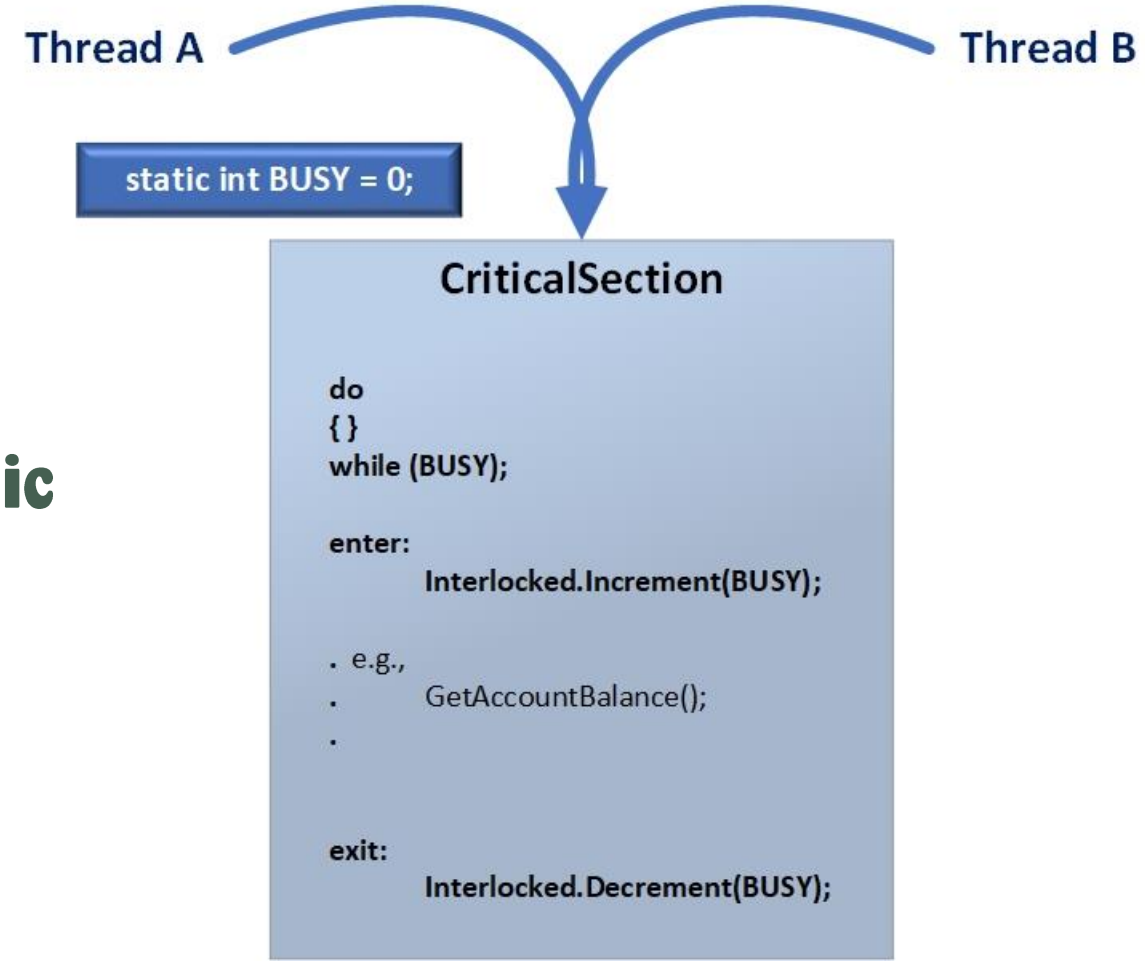
$$C(p) = p/(1 + s(p-1) + \kappa p(p-1))$$

  - queueing delays associated with accessing critical sections
  - slowdown in hardware execution due to cache coherence delays
  - overheads (spinning up additional threads, Fork, Join methods, etc.)



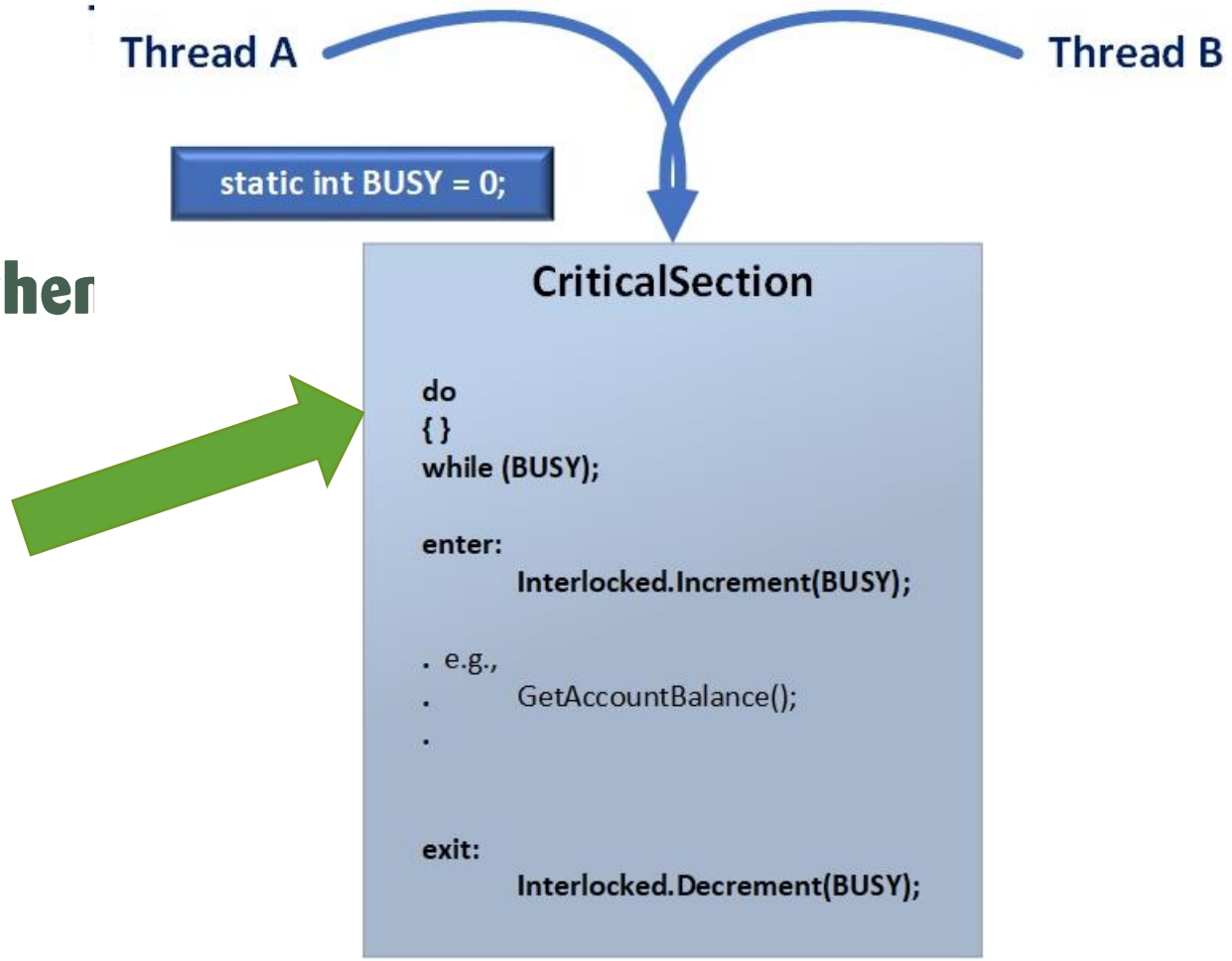
Concurrent Programming concepts

- **Critical Sections**
  - block of ***shared*** code that only one thread at a time can execute
  - mutual exclusion
  - commonly implemented using atomic instructions on Locks
    - Test and Set
    - Compare and Swap
    - XCHG (x86 instruction)



Concurrent Programming concepts

- **Critical Sections Best Practice:**
  - Keep the body of a mutual exclusion, critical section very concise
  - What should the blocked Thread do when the critical section is occupied?
    - Spin
      - short duration : M/D/1 (Busy Waits)
      - kernel threads (pre-emptible?)
    - Queue
      - longer waits
      - indeterminate waits



Concurrent Programming concepts

- see `lock` Statement (C# reference)
  - link
- see also
  Monitor class;
  Interlocked Operations
  etc.
- potential for **deadlocks**!

```
class Account
{
  decimal balance;
  private Object thisLock = new Object();
  public void Withdraw(decimal amount)
  {
       lock (thisLock)
       { if (amount > balance)
          {
             throw new Exception
                ("Insufficient funds");
          }
          balance -= amount;
       }
    }
}
```

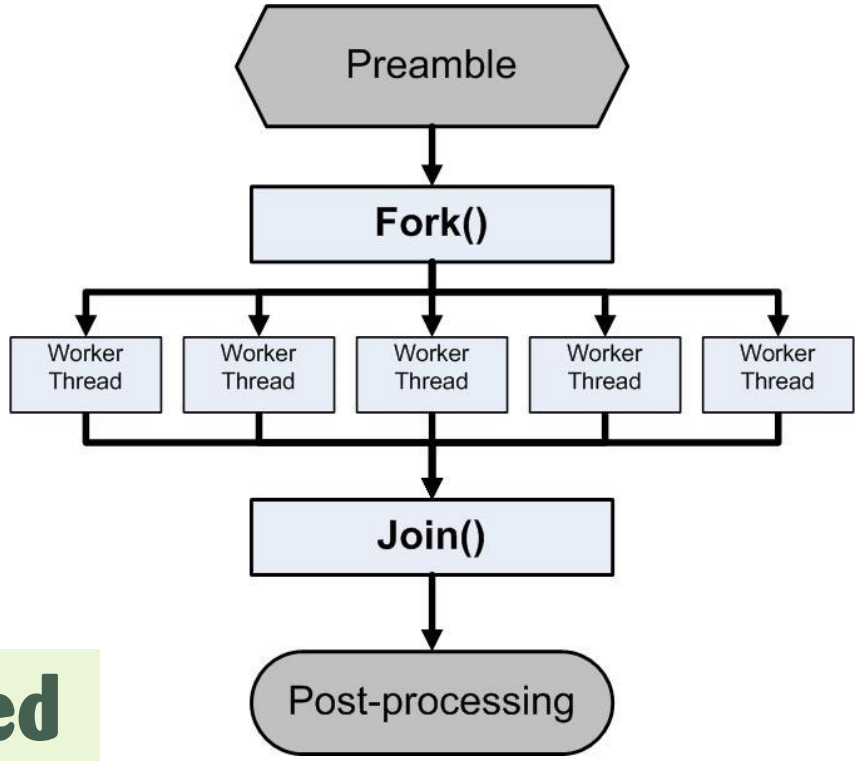
Concurrent Programming concepts

- see `Monitor` class
  - link
- User mode threads
  - spinlocks?
  - critical section code is subject to **preemption** and **blocking** by lower level routines

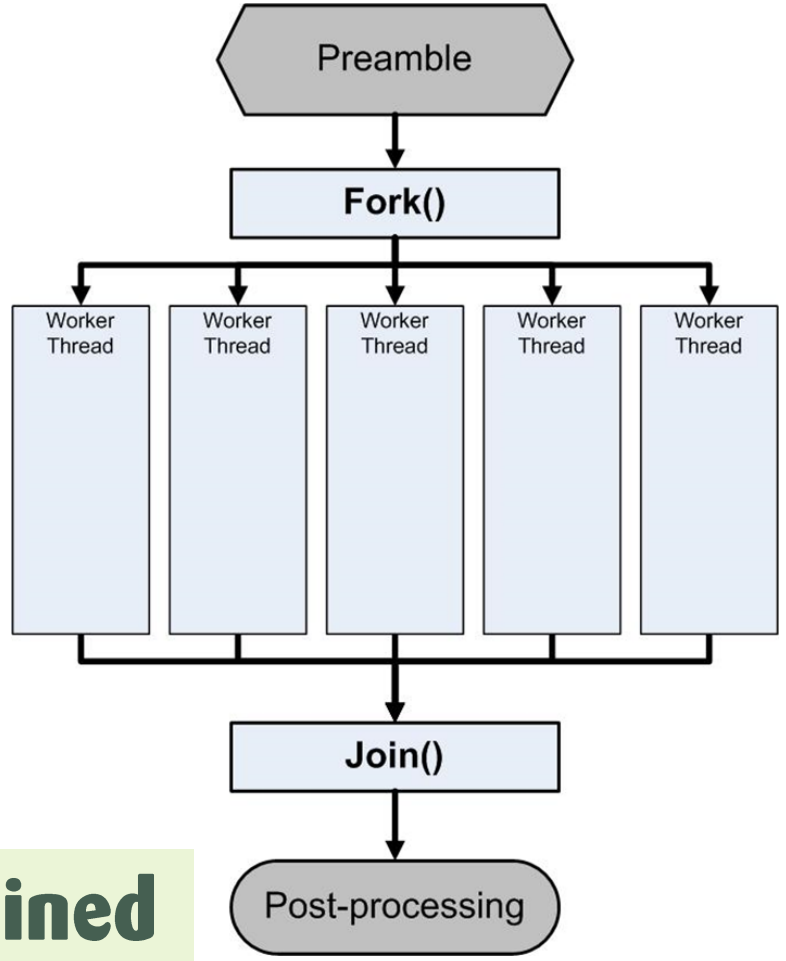
```
var lockObj = new Object();
var timeout = TimeSpan.FromMilliseconds(500);

if (Monitor.TryEnter(lockObj, timeout))
 {
       try
       {
           // The critical section.
       }
       finally
       {
          // Ensure that the lock is released.
           Monitor.Exit(lockObj);
       }
}
else { // The lock was not acquired. }
```

The Limits of Parallel Programming

- Fine-grained vs. coarse-grained parallelism



Fine-grained

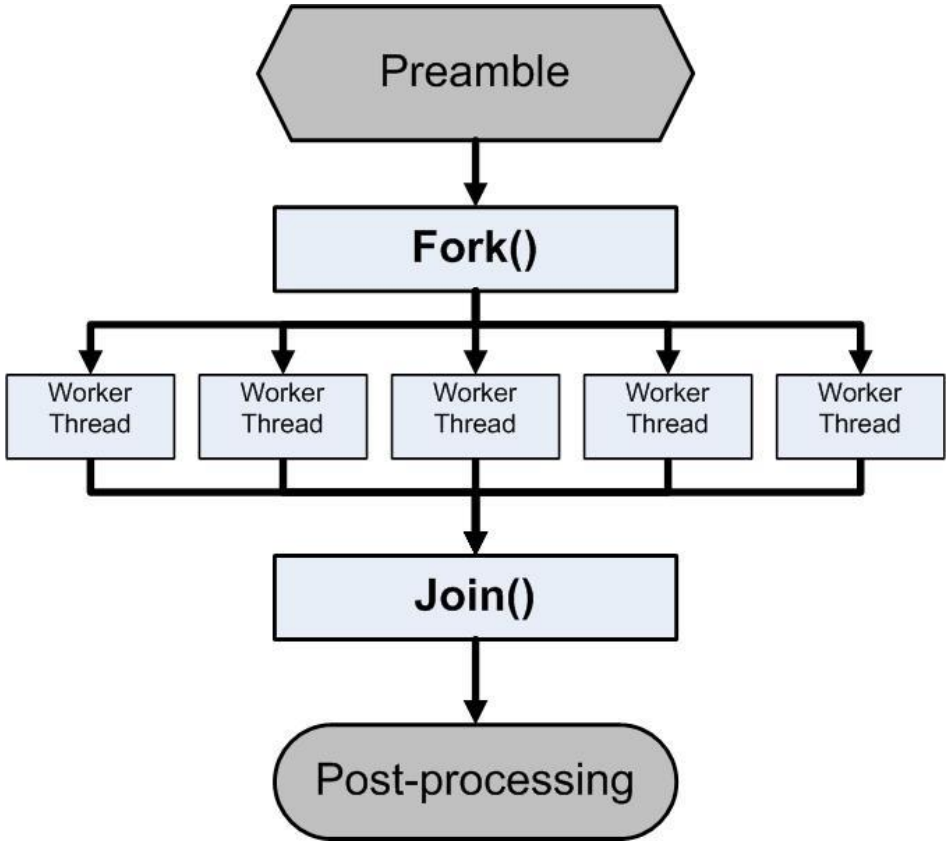


Course-grained

The Limits of Parallel Programming

- "Fine-grained" parallelism is distinguished from "coarse-grained" empirically
  - Coarse-grained parallelism applies when the amount of work each parallel task does ***exceeds*** the "overhead" associated with scheduling and executing additional threads

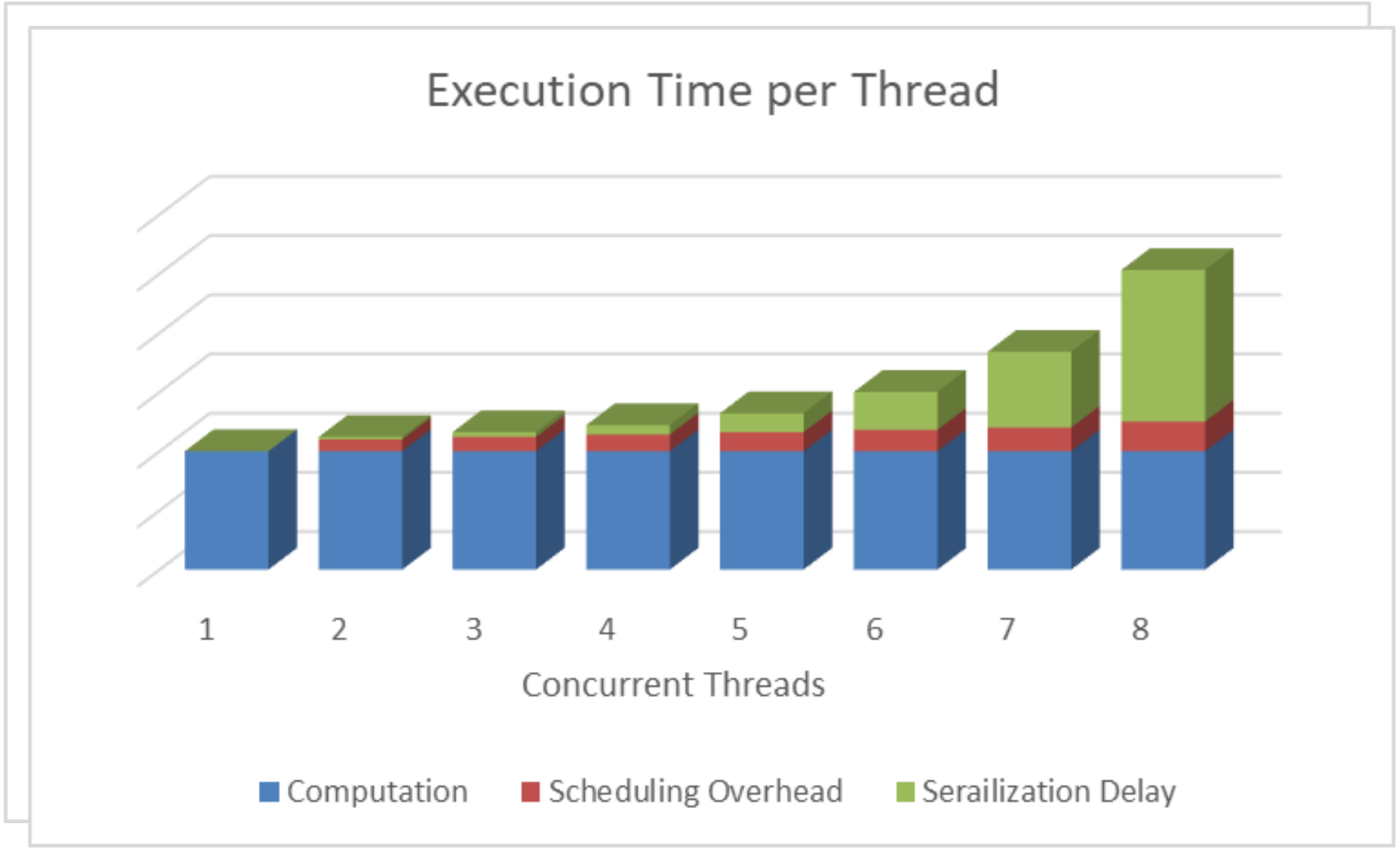
- Anti-pattern:
  - Overhead also includes ***lock serialization*** delays, which tend to increase with the number of contending tasks



The Limits of Parallel Programming

- Serialization delays tend to increase with the number of contending tasks:

- Instrumentation
  - some higher-level counters
  - Traces



The Limits of Parallel Programming

- "Fine-grained" vs. "coarse-grained" parallelism
  - Synchronization delays tend to increase with the number of contending tasks
    - Excessive contention requires finer-grained locking
      - replace global locks with local locks
        - lock a unique data row ID
        - lock a partition
  - e.g., consider the One Writer : multiple Readers pattern
    - uses a counting semaphore
  - Complex locking schemes can have nasty side effects

The Limits of Parallel Programming

- "Fine-grained" vs. "coarse-grained" parallelism
  - **Optimistic locking** works well when the probability of lock contention is low
    - Repeatable Read

      1. Don't acquire the Lock initially
      2. Read
      3. Update
      4. repeatable Read OK? following Update
      5. otherwise, backout the change and try again

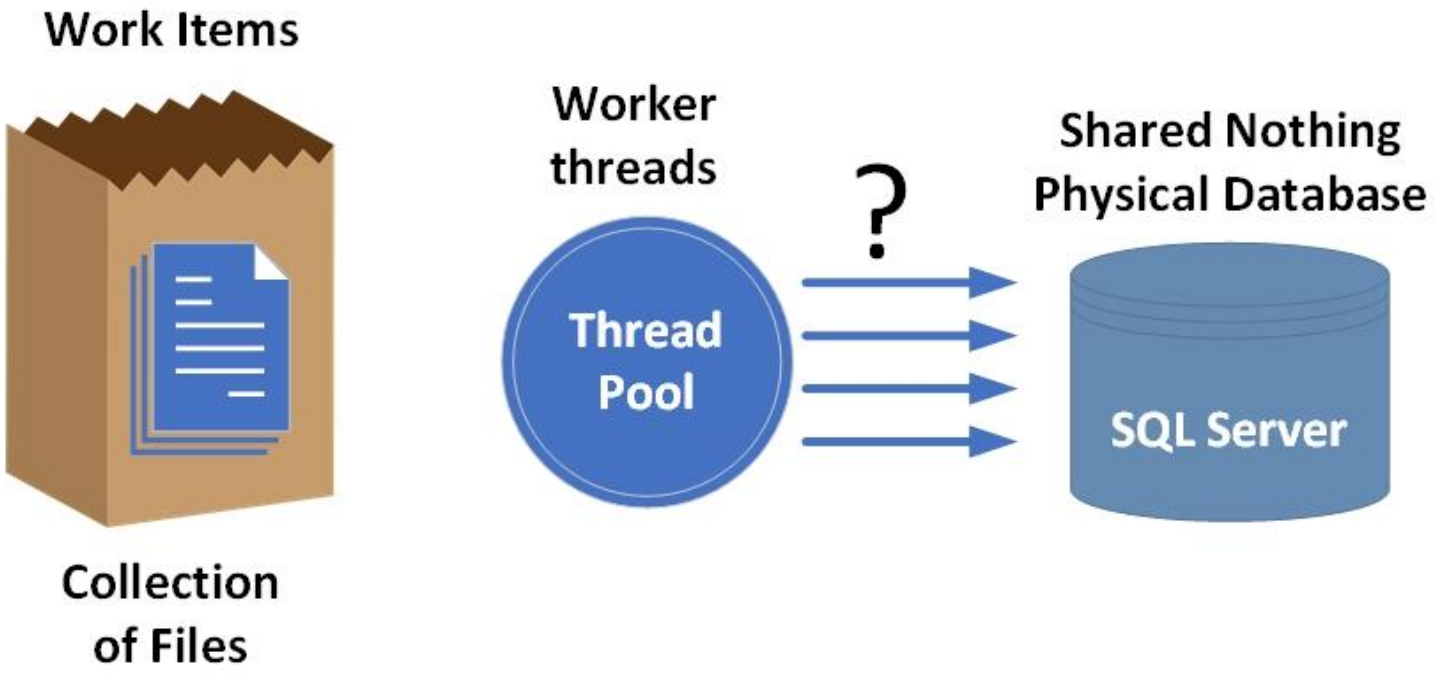
    - More lock contention requires finer-grained locking

The Limits of Parallel Programming

- **Shared Nothing** scalability:
  - requires partitioning the problem space so no resources are shared across concurrently executing threads

- SIMD Example:

What is the optimal number of database update threads that should be released?



Questions

?

References

- Herb Sutter, "The Free Lunch Is Over," Dr. Dobbs Computer Journal, 2004
- Susan Eggers, et. al., "Simultaneous Multi-Threading: Maximizing On-Chip Parallelism," 1995.