Lecture #09 - Parallel Computing and OpenMP

AMath 483/583

Announcements

- Homework #1 Solutions posted within a week
- Homework #2 Updates —> add remote, pull (demo)
    - fixes will be made this morning, announced

Word of Wisdom

- For single-core and parallel computing…

  “Premature optimization is the root of all evil.” —
  Donald Knuth

  (Computer scientist, mathematician, “Father of algorithm analysis.”)

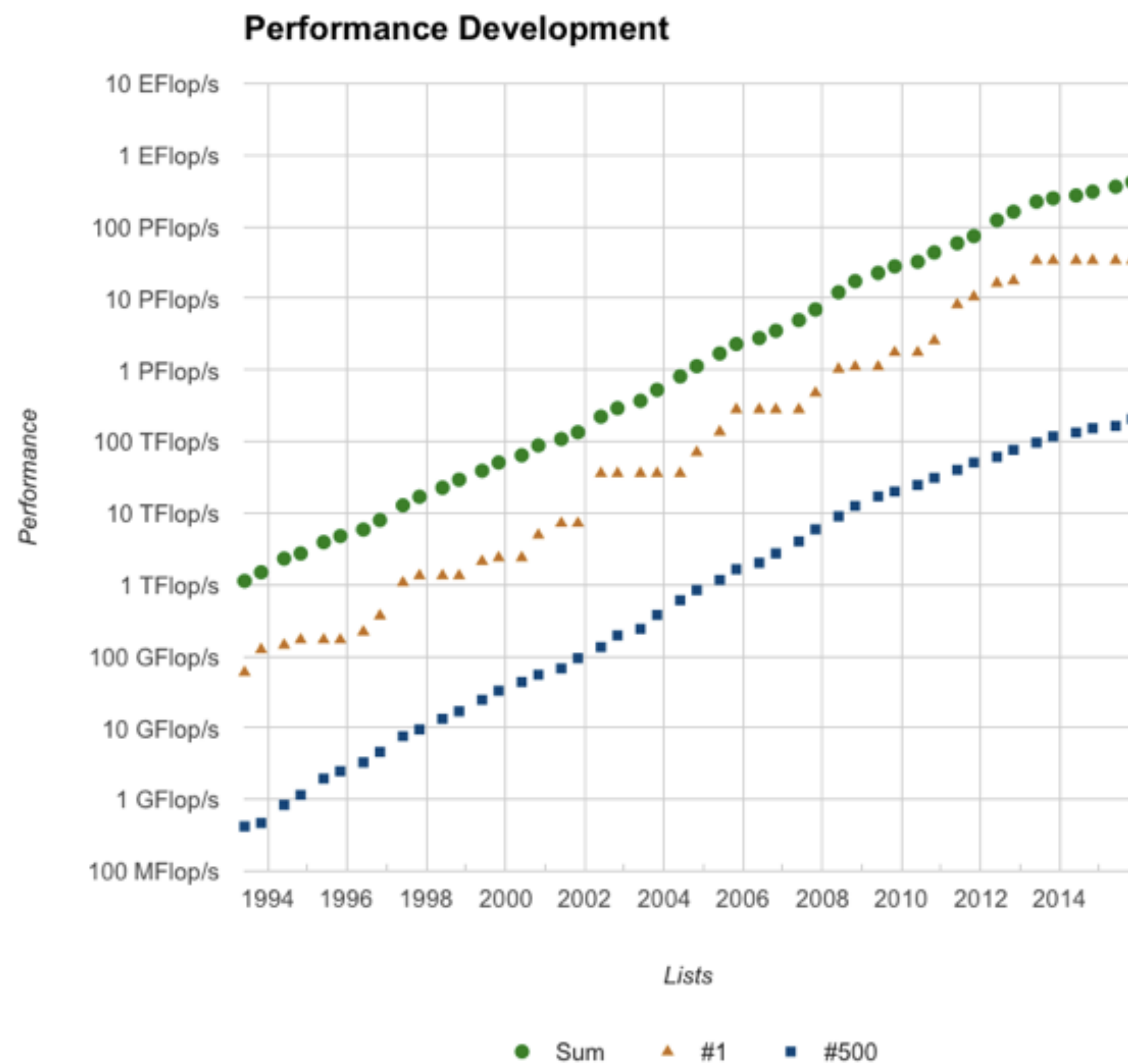
Parallel Computing

- Old Version:

  processor speed doubles every eighteen months

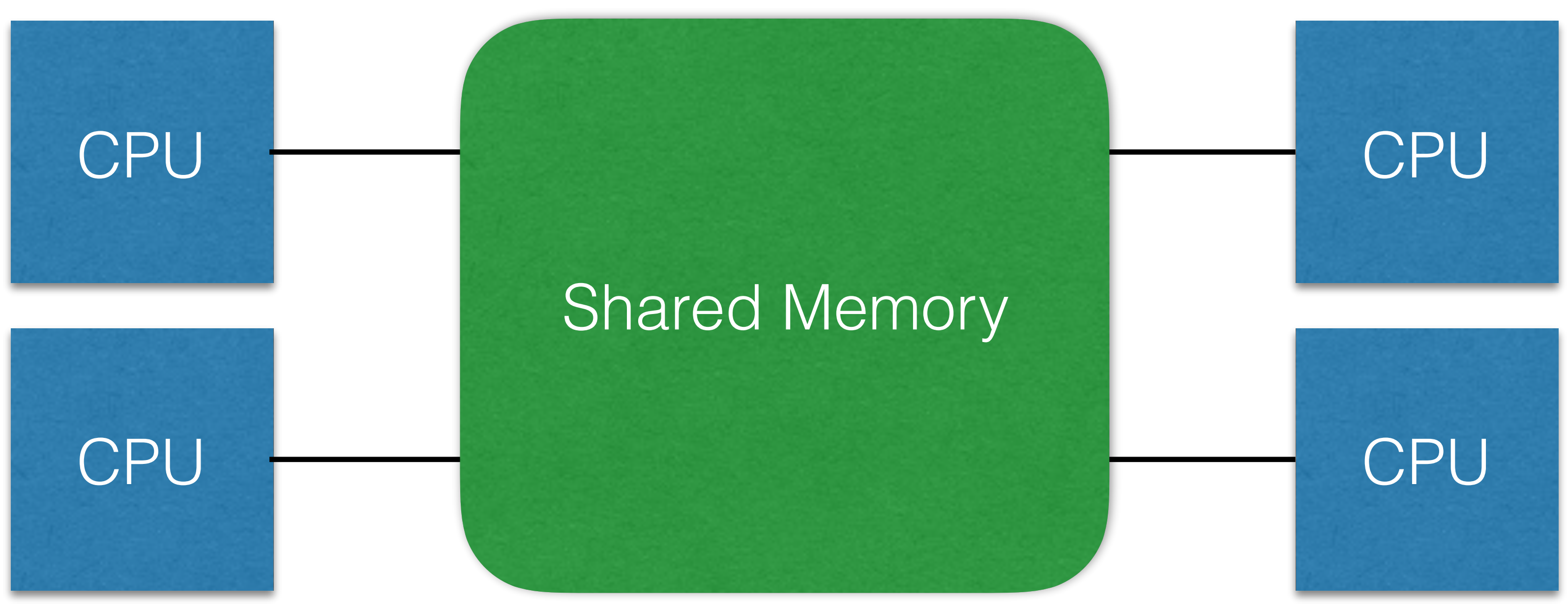
- New Version:

  number of cores doubles every eighteen months



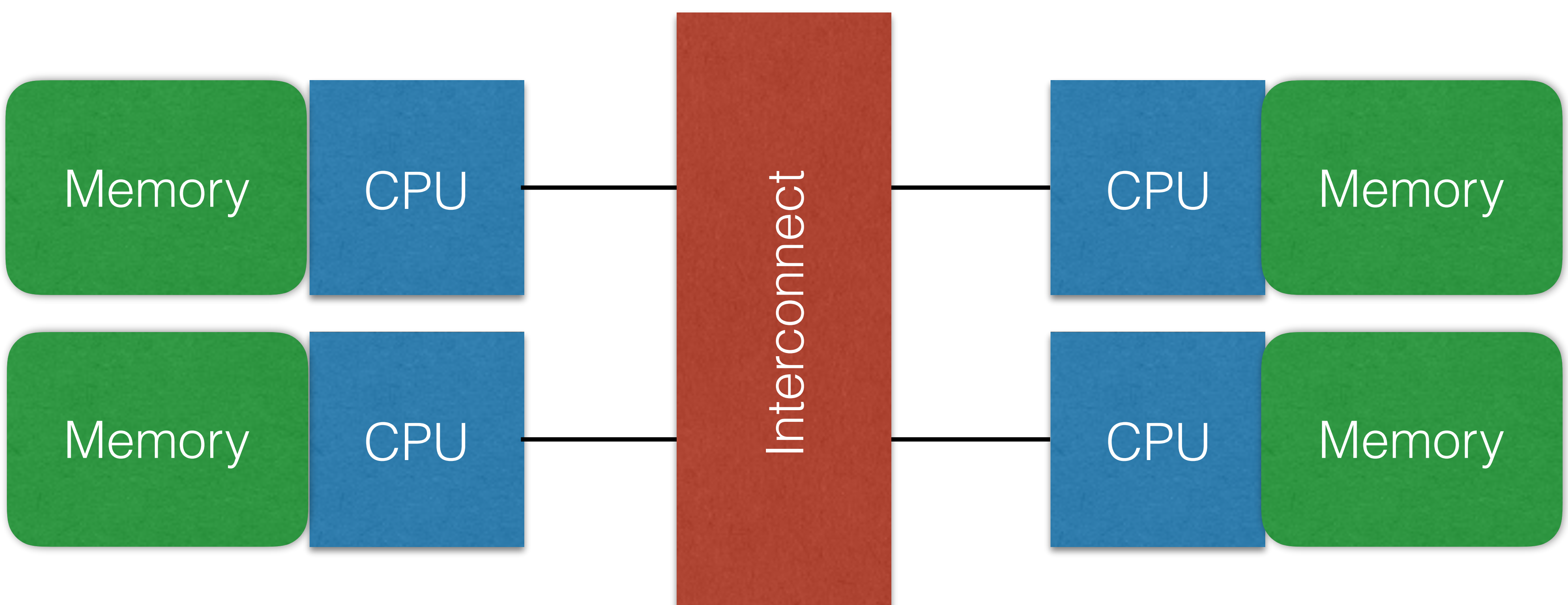
Shared Memory

- All processors / nodes have access to same memory
    - e.g. L3 cache (in most chips) and RAM
    - nodes on massively parallel machines
    - implicit data communication



Distributed Memory

- Each processor / node has its own memory pool
  - non-example: L1 and L2 cache on most processors (*coherence*)
  - nodes on massively parallel machines
  - explicit data communication

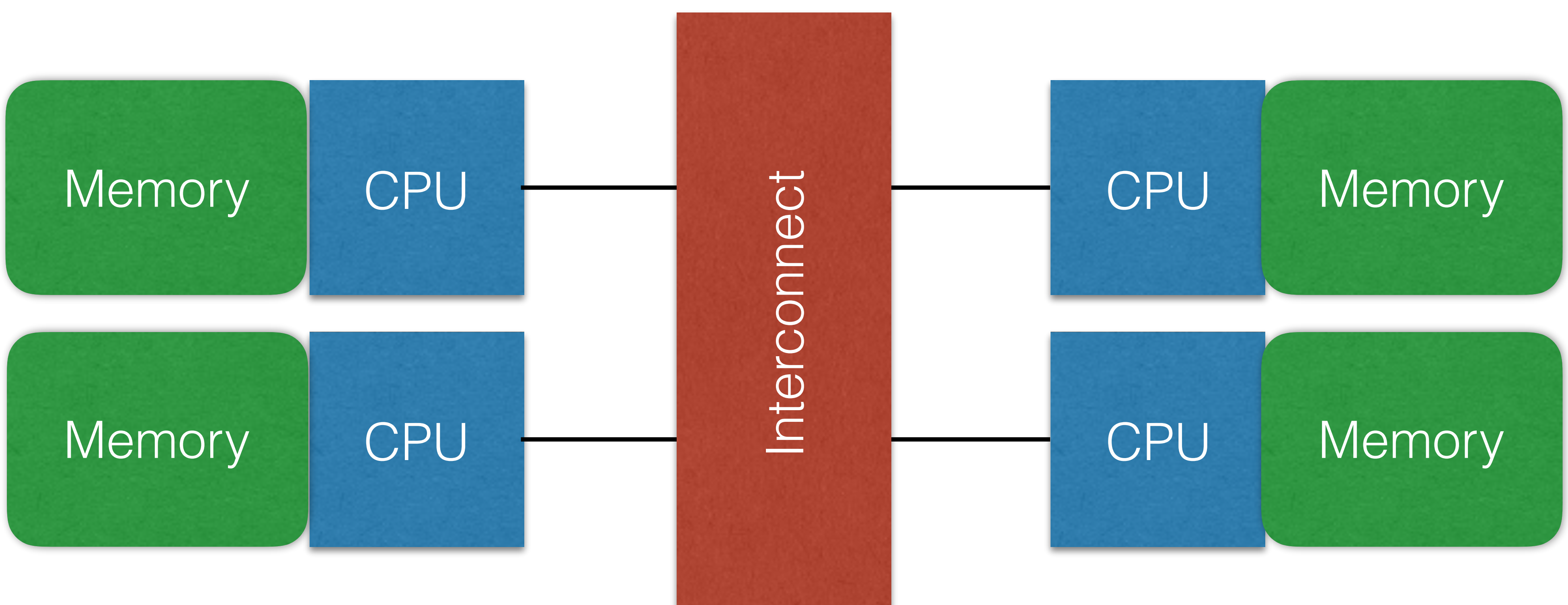


Distributed Memory

- Each proc
  - non-exa
  - nodes c
  - explicit data communication

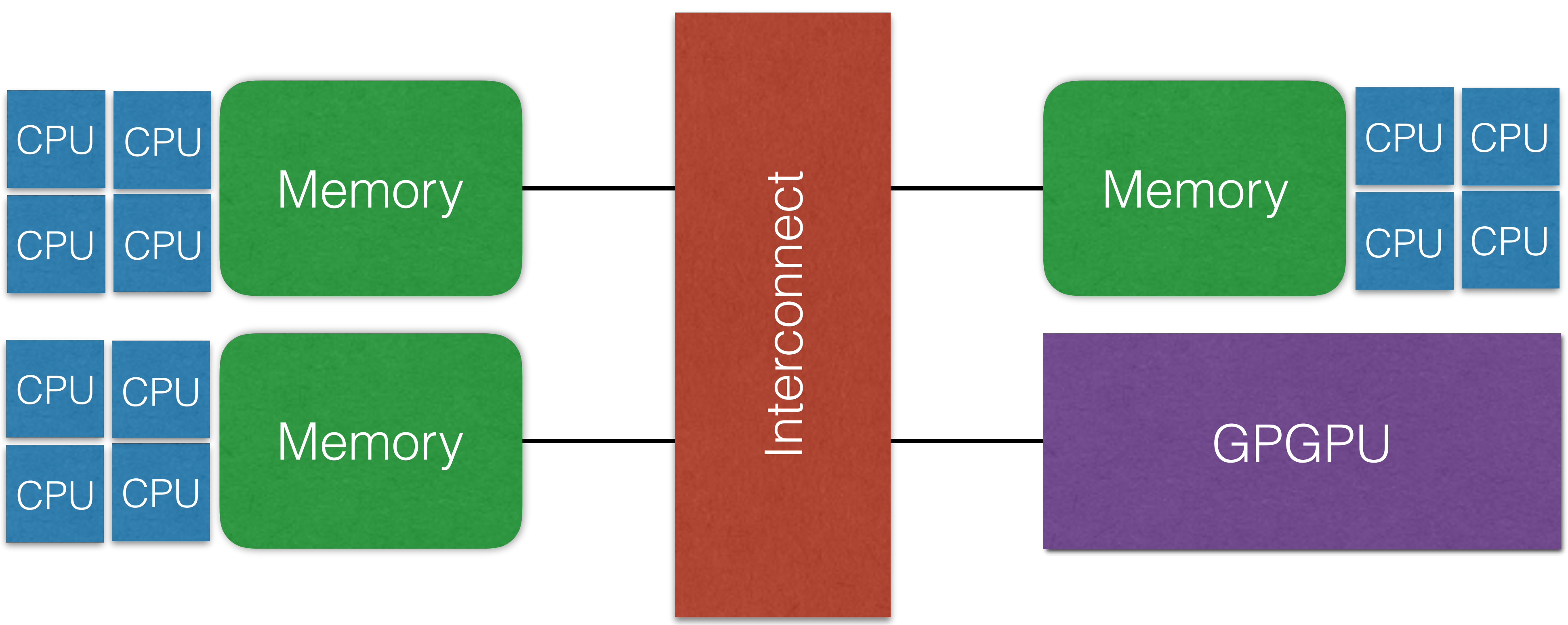
The interconnect is very slow!
(Relative to CPU <—> Shmem communication)

CPU —> RAM = 100 ns
CPU —> CPU = 10,000+ ns



Multi-Hardware Systems

- Combined systems of shared + distributed memory
- Dedicated hardware:
  - SIMD / Vector Processors
  - GPGPUs

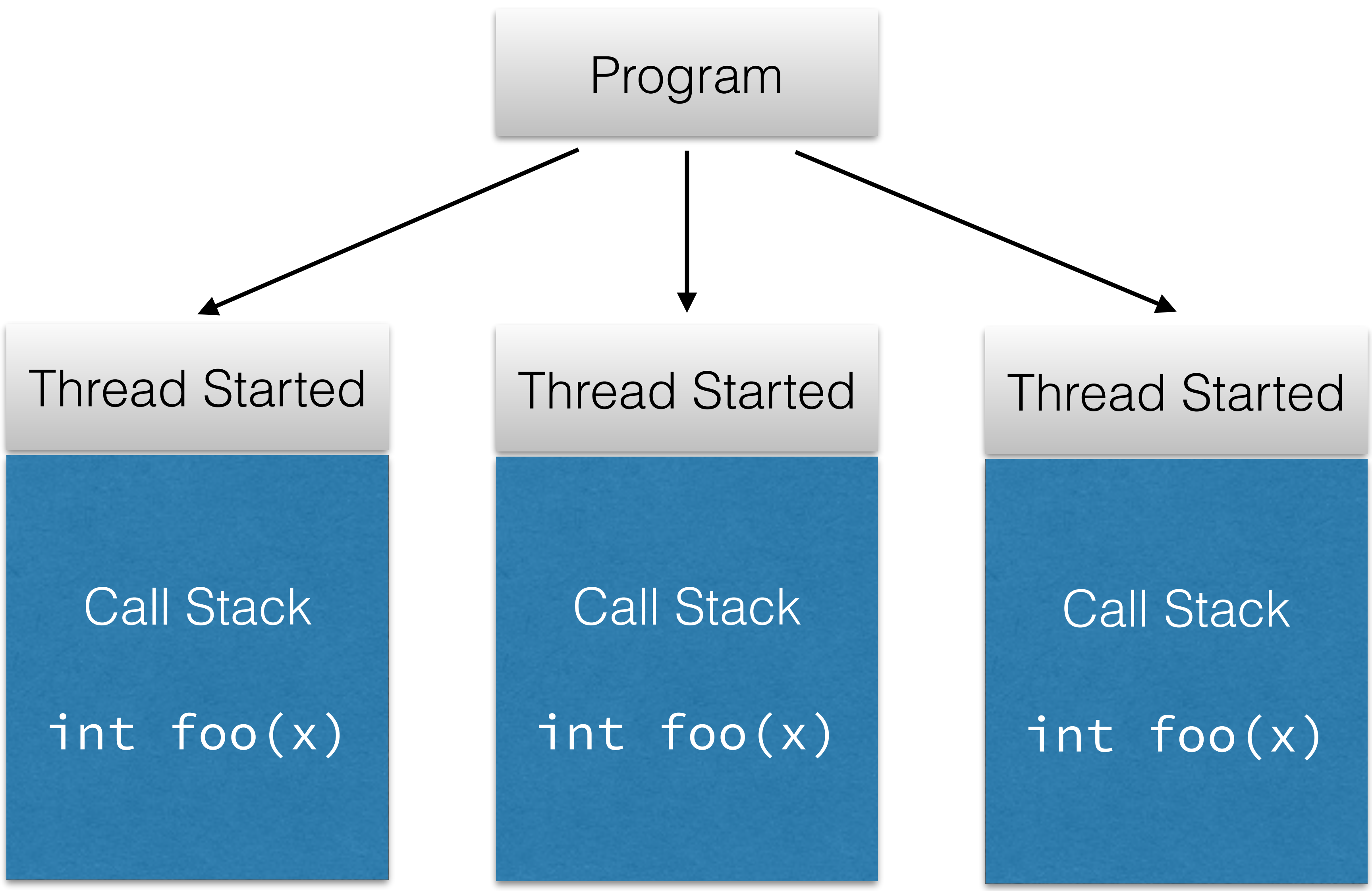


Threads

- *“Sequence of instructions with both thread-specific data and access to shared address space.”*
    - each thread has own call stack and local vars
    - each thread can access shared data
- Spawned and destroyed throughout a program / computation.

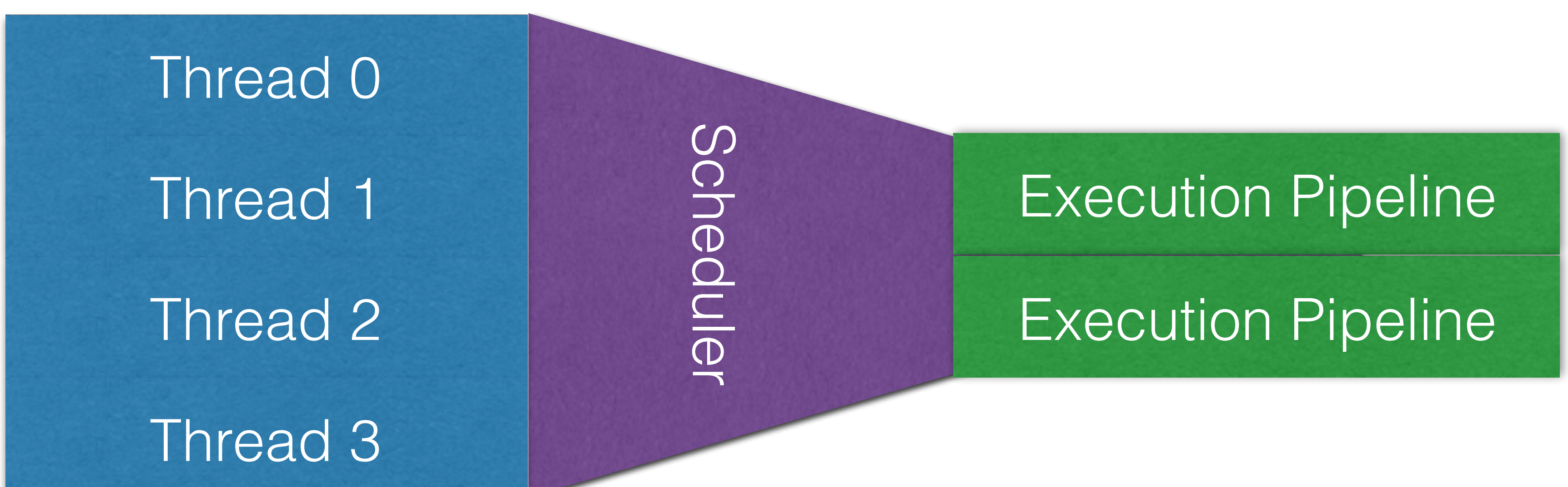
Threads

- Each thread has its own call stack (even when running same code)



Threads

- Software Threads — spawned by program
- Hardware Threads — simult. instruction pipelines offered by CPU
  - “scheduler” feeds software threads into hardware
  - schedule chooses which instruction to execute next (major area of research)



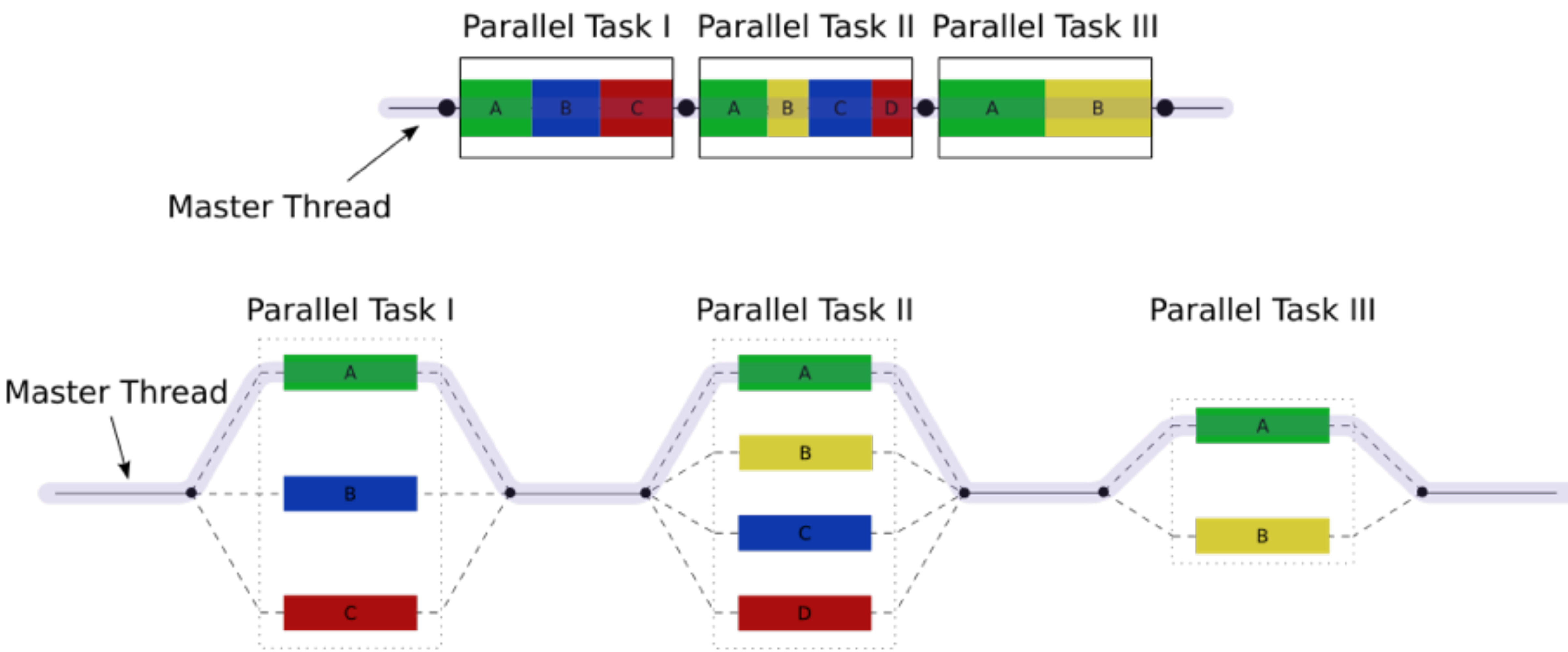
Parallel Work

- "Embarrassingly parallel (link)" — each thread works completely independently
  - `vec_add` — each thread adds across a component: `out[i] = v[i] + w[i]`
  - `mat_mul` — each thread computes `Cij`
  - `gradient_descent` — each thread tries a different initial guess `x0`

Parallel Work

- Some situations more complicated:
  - `vec_norm` — how do you parallelize summation across N elements?
  - Problem: multiple threads writing to same result
- <u>Need to break parallelism to "*synchronize*" threads.</u>

Parallel Work



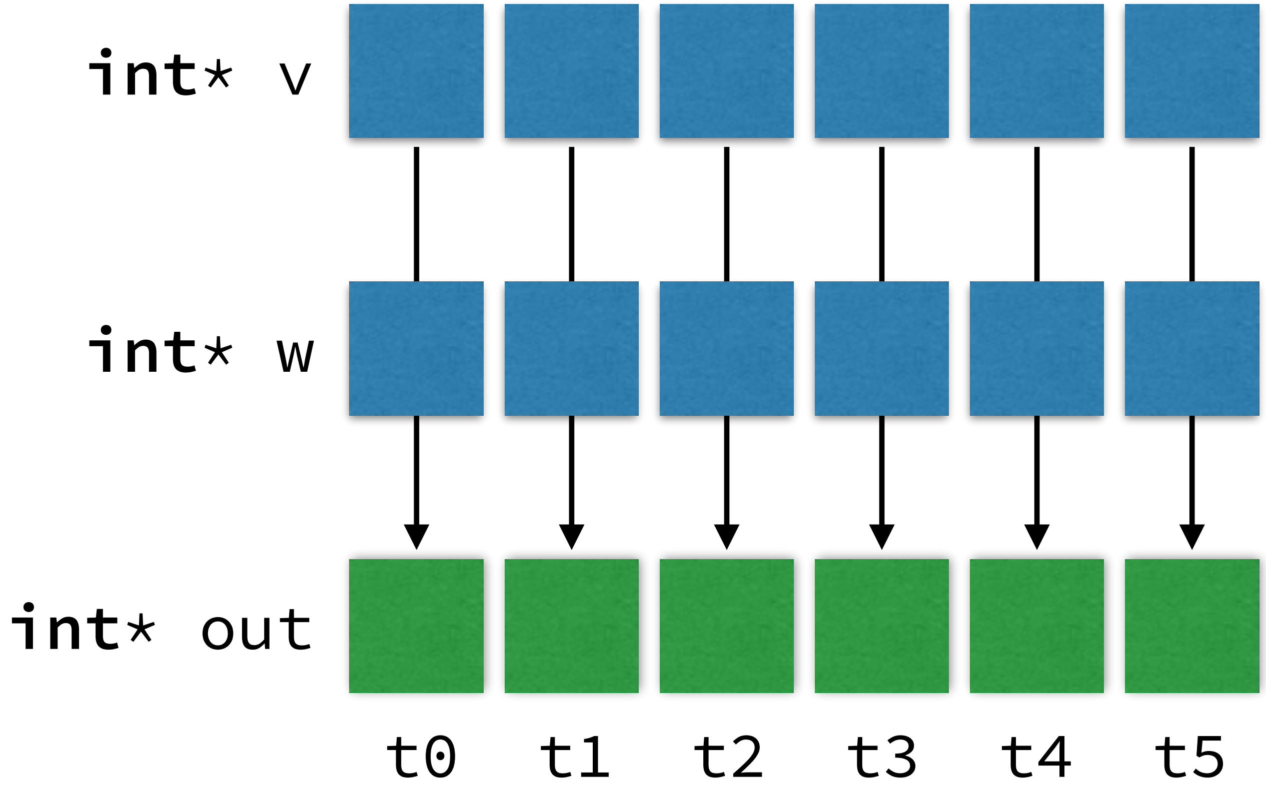
source: OpenMP wiki

OpenMP

- Shared memory multiprocessing in C/C++/FORTAN
    - easy to use (difficult to debug)
    - wraps PThreads
- POSIX Threads (link)
    - standardized C thread programming library
    - UNIX standard since 1995

Threads Issues

- Multiple threads have access to same data — very deep subject
- **Simple Situation**: `vec_add` — each thread computes the sum at each index, data is independent / disjoint



Threads Issues

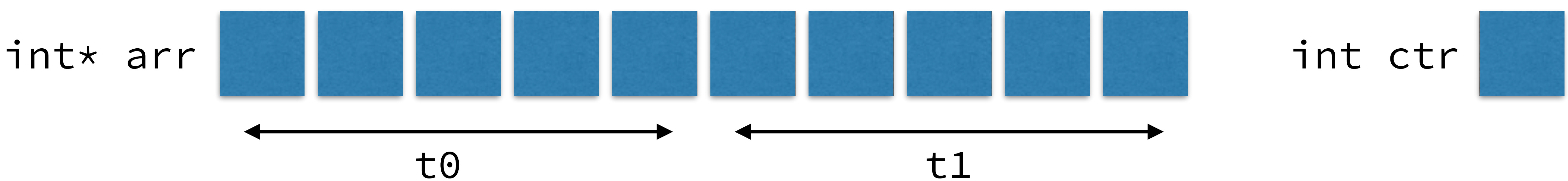
- **Complex Situation**: `count` — determine the number of elements in an array greater than x


```
for (int i=0; i<length; ++i)
  if (arr[i] < x)
    ctr += 1;
```

- Three step process:
  - read `counter`, add one, write to `counter`

Threads Issues

- **Complex Situation**: `count` — determine the number of elements in an array greater than x



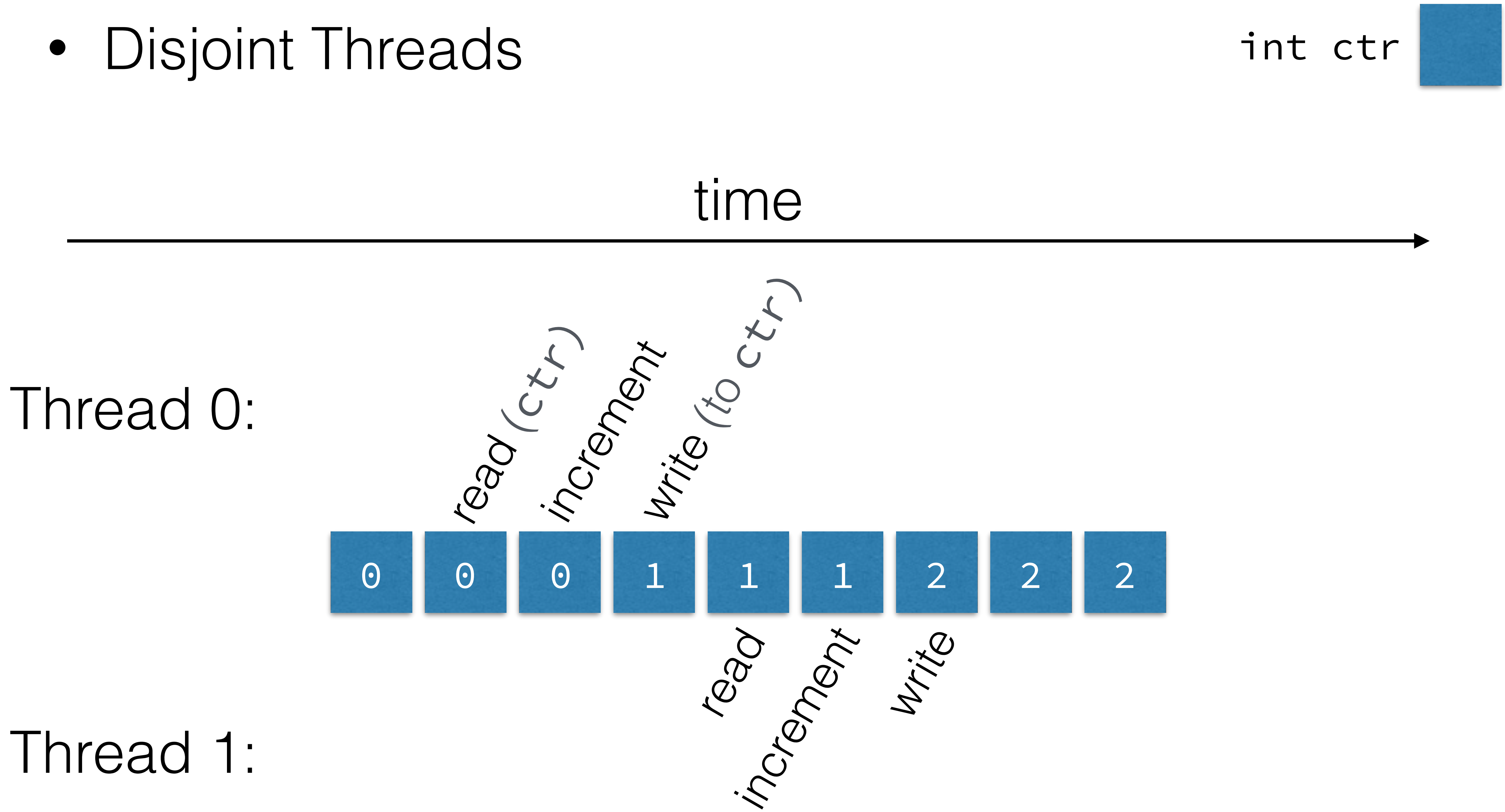

```
// Executed by thread 0
for (int i=0; i<length/2; ++i)
{
  if (arr[i] < x)
    ctr += 1;
}
// Executed by thread 1
for (int i=length/2; i<length; ++i)
{
  if (arr[i] < x)
    ctr += 1;
}
```

`Thread 0`: read counter, increment, write to counter

`Thread 1`: read counter, increment, write to counter

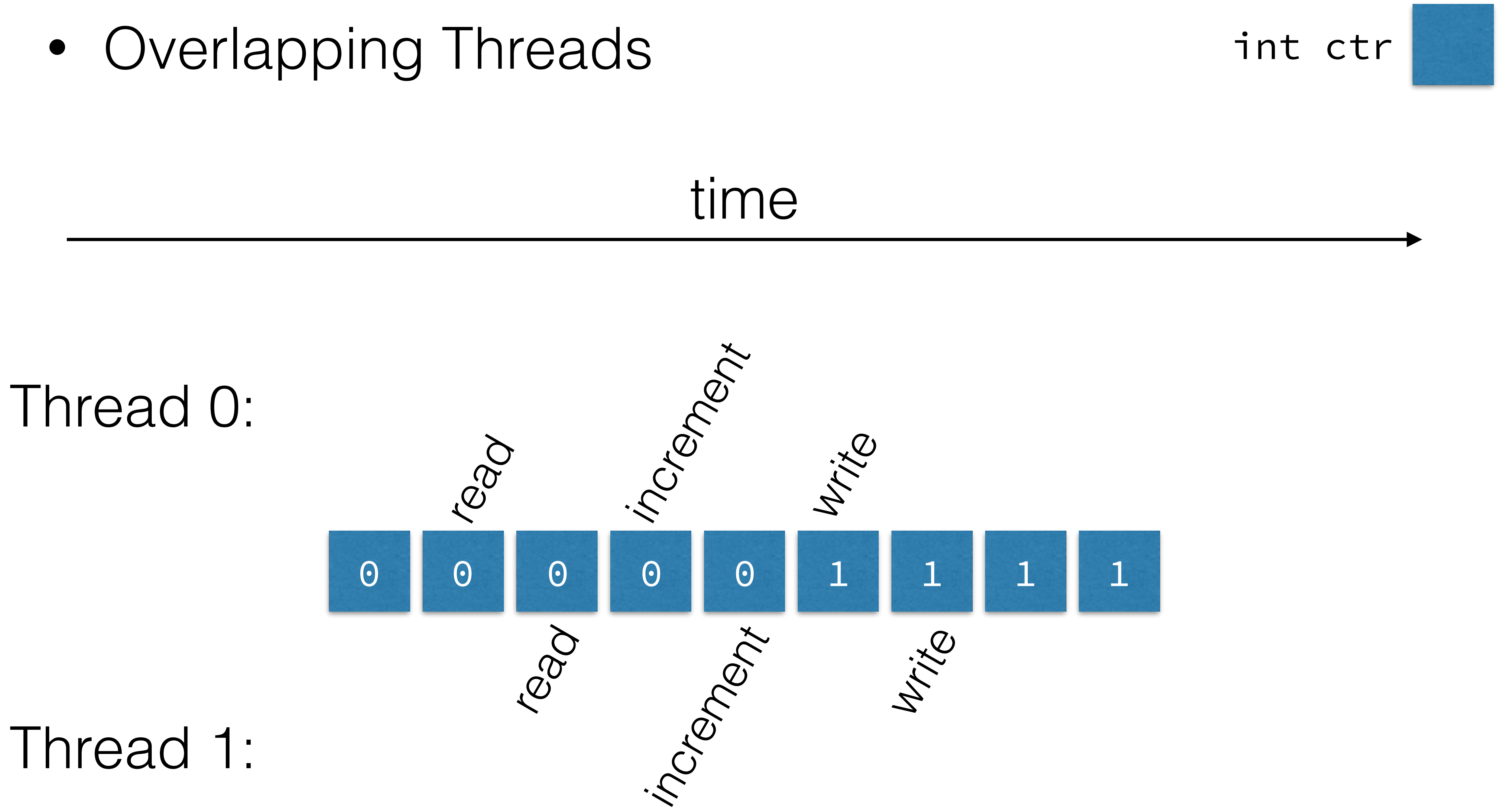
Threads Issues

- Disjoint Threads



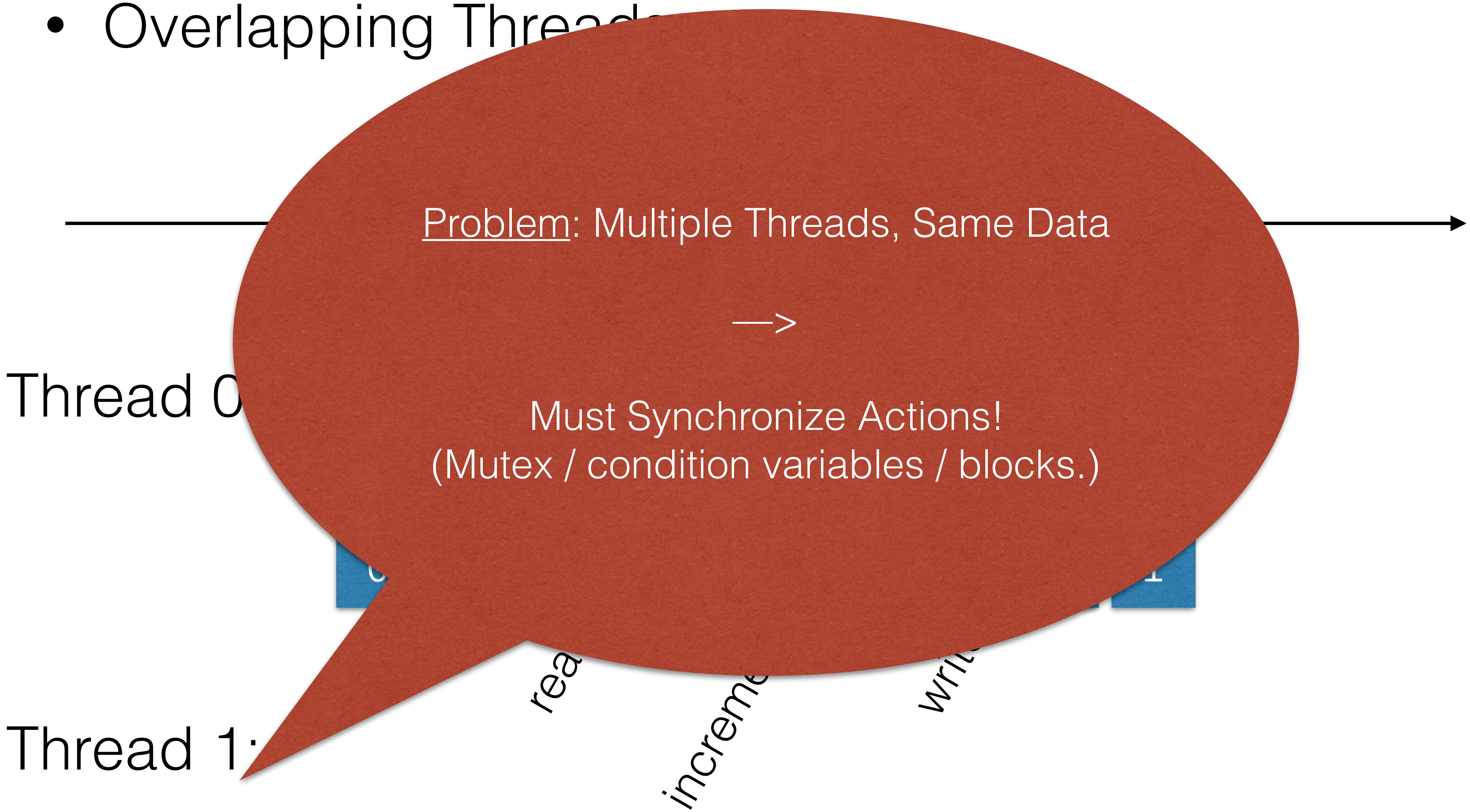
Threads Issues

- Overlapping Threads

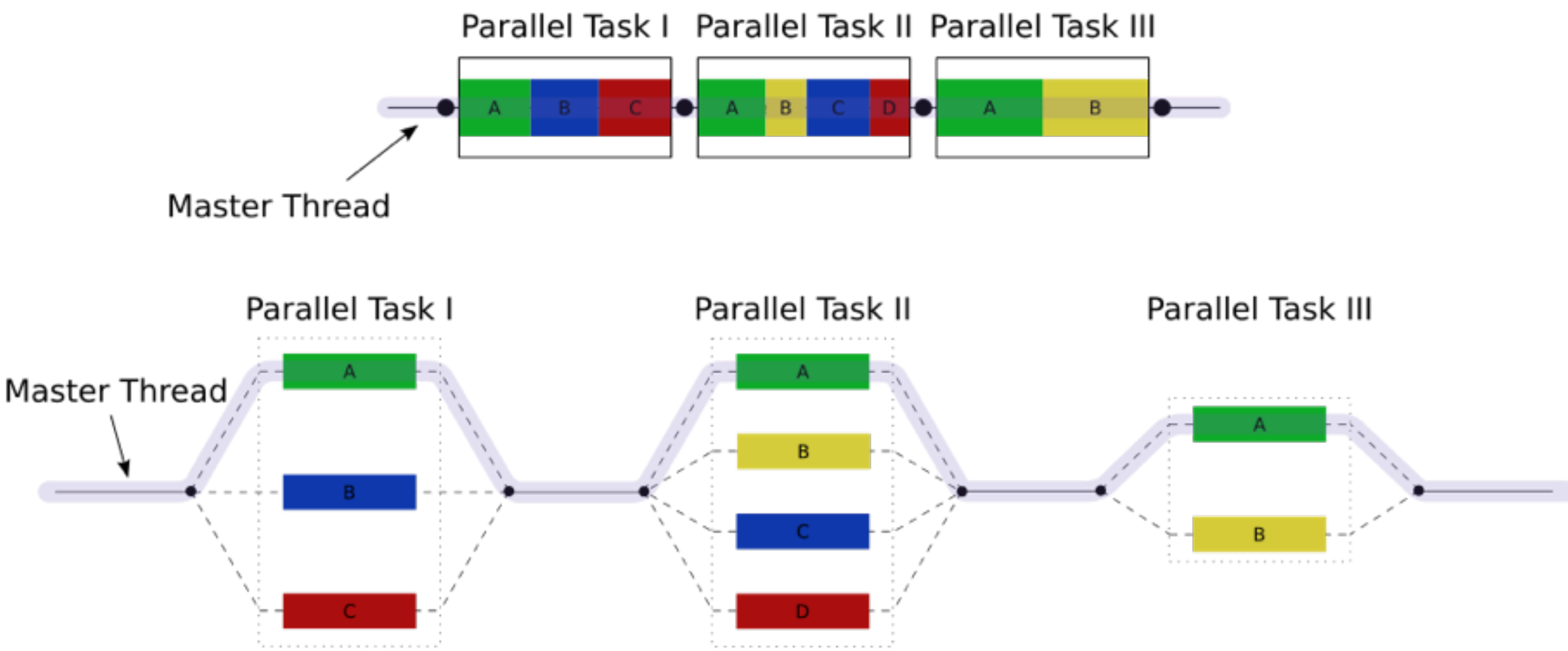


Threads Issues

- Overlapping Threa



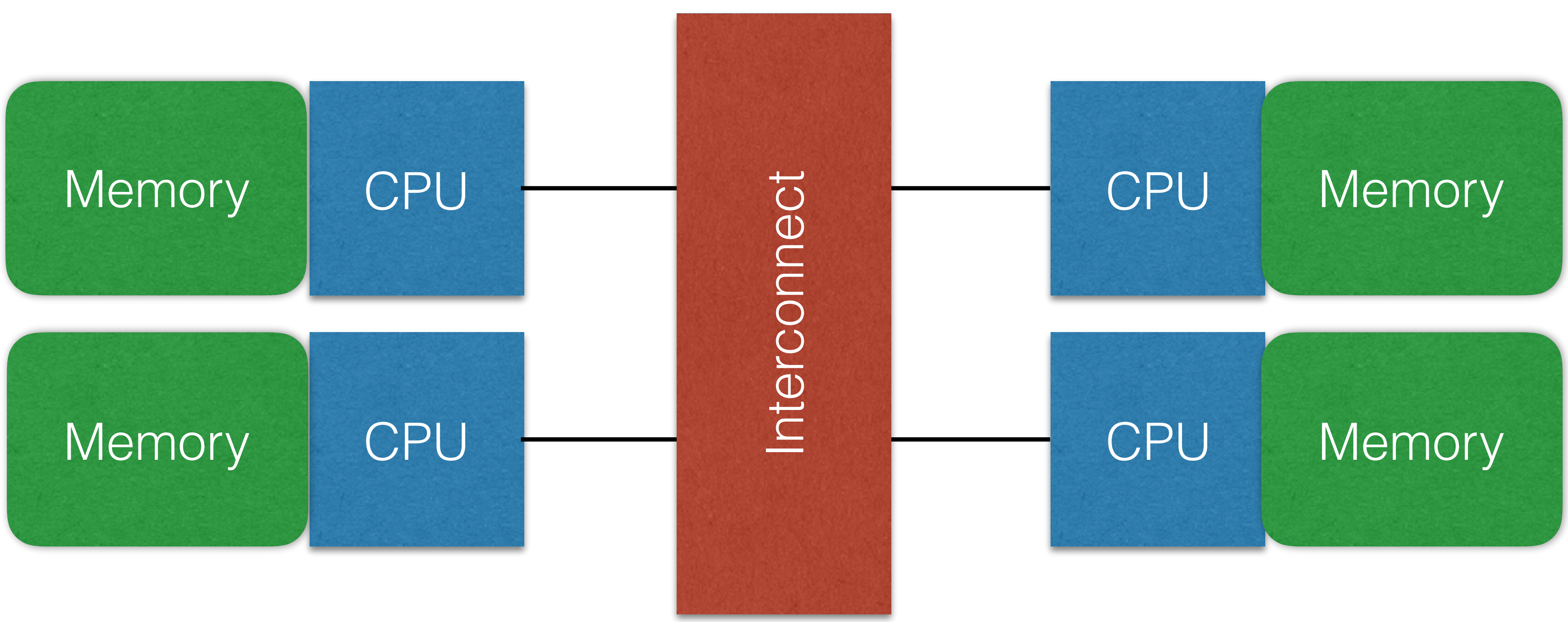
Thread Issues



source: OpenMP wiki

Processes

- *"Thread with its own address space." (i.e. thread's data is private)*
- Thousands+ processes running on computer
  - (I can give this talk, move my mouse cursor, receive (many) emails…)
- Process model used in distributed memory systems:



MPI

- "Message Passing Interface" for C/C++/FORTRAN
- Processes communicate by explicitly passing messages to each other
  - high communication overhead —> maximize local data processing before sending info
- "Coarse-grain parallelism"
- Uses sockets / TCP over network

Amdahl's Law

- Only part of a computation can be parallelized

  Suppose 50% of computation is inherently sequential and other 50% is parallelizable.

- **Question**: given N processors how much faster can this computation be run?

Amdahl's Law

- Only part of a computation can be parallelized

  Suppose 50% of computation is inherently sequential and other 50% is parallelizable.

- **Question**: given N processors how much faster can this computation be run?

- **Answer**: at most x2 faster *(indep. of N!)*

  - (if parallel part reduced to zero time)

Amdahl's Law

- Only part of a computation can be parallelized

  *Suppose **10%** of computation is inherently sequential and other **90%** is parallelizable.*

- **Question**: given N processors how much faster can this computation be run?

Amdahl's Law

- Only part of a computation can be parallelized

  *Suppose **<u>10%</u>** of computation is inherently sequential and other **<u>90%</u>** is parallelizable.*

- **Question**: given N processors how much faster can this computation be run?

- **Answer**: at most x10 faster *(indep. of N!)*

  - (sequential part is still taking 1/10 of time)

Amdahl's Law

Suppose 1/S of the computation is inherently sequential and other (1 - 1/S) can be parallelized.

- At most factor of S speedup
- Let T_S be time required in sequential (single-core) machine to run a computation

$$T_S = (1/S)T_S + (1 - 1/S)T_S$$

Amdahl's Law

- Now run on P processors
- Let T_P be time required on parallel machine. Then T_P is at least:

$$T_P = (1/S)T_S + \frac{(1-1/S)}{P}T_S$$

- Note:

$$\lim_{P\to\infty} T_P = (1/S)T_S$$

Amdahl's Law

- *Suppose 1/S of the computation is inherently sequential, then*

$$T_P = (1/S)T_S + \frac{(1-1/S)}{P}T_S$$

where T_S is the original sequential time and T_P is the time taken across P processors

Speedup

- **Speedup** = T_S/T_P
  - Typically, speedup << P
- *Amdahl's law does not account for overhead costs*:
  - starting / destroying processes and threads,thread communication, …

$$T_P = (1/S)T_S + (1 - 1/S)T_S + T_{\text{overhead}}$$

Scaling

- Some algorithms scale better than others as P increases…
  - embarrassingly parallelizable algorithms
  - algorithms with low / batched communication between threads
  - few blocking calls
- Let N = problem size (solve N x N linear system, simulate N particles in space, etc.)

Strong Scaling

- *“How does the algorithm perform as the number of processors P increases for fixed problem size N?"*

Any algorithm will eventually break down. (Consider P > N.)

Weak Scaling

- *"How does the algorithm perform when the problem size increases with the number of processors?"*

e.g. will doubling the number of processors allow us to solve a problem twice as large in the same time?

Weak Scaling

- *"Twice as large"*
- Example: Solving N x N linear system with Gaussian elimination (LU-factorization) = $\mathcal{O}(N^3)$
  - double the problem size = increase N by a factor of $2^{1/3} \approx 1.26$
  - e.g. 100 x 100 —> 126 x 126

Weak Scaling

- *“Twice*
- Examp with Gauss $= \mathcal{O}(N^3)$
  - doul e N by a factor of 2^1
  - e.g.

Warning!

$\mathcal{O}(N^3)$ refers to number of multiplications.

Memory accesses are much more expensive. How does that change with N?

Lecture #10 - OpenMP

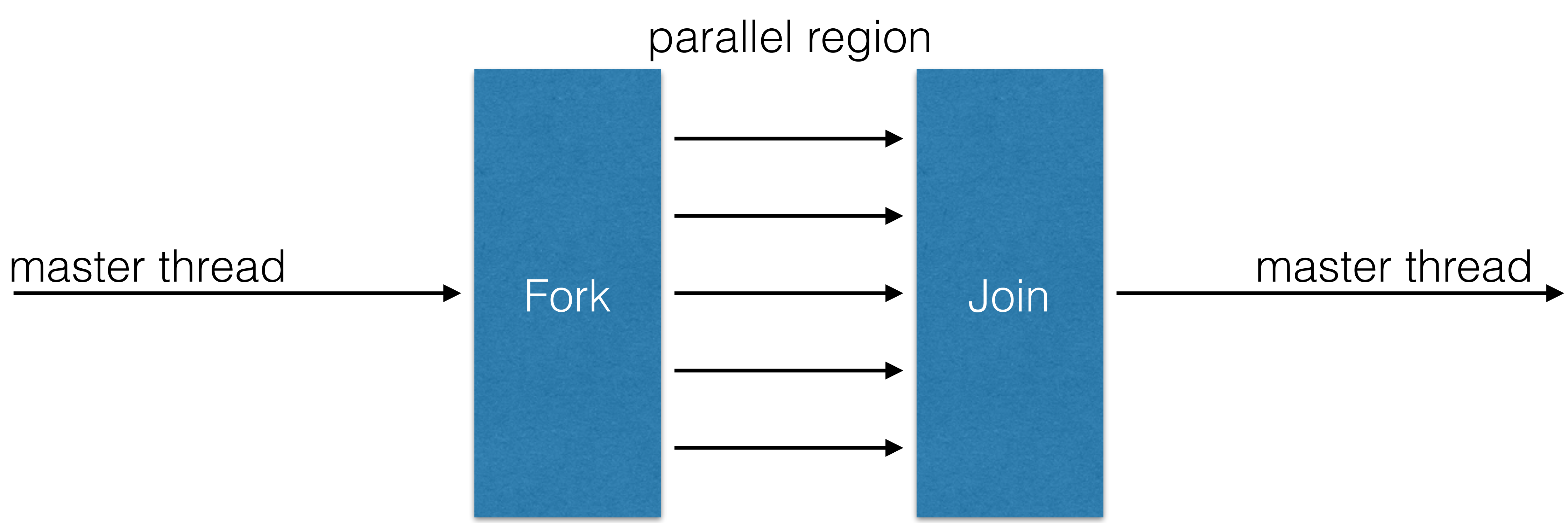
AMath 483/583

OpenMP Resources

- http://www.openmp.org/wp/resources
  - Tim Mattson's (Intel) "Introduction to OpenMP" (YouTube video series)
  - Lawrence Livermore National Labs "Introduction to OpenMP" (Tutorial + Reference)
  - + many more
  - *(theme of this class: lectures are a starting point, online resources are abundant!)*

OpenMP - Basic Idea

- “Fork - Join” model
  - begin with single thread (master thread)
  - FORK: master thread creates team of parallel threads
  - JOIN: when threads complete action they synchronize and terminate



OpenMP - Basic Idea

- Compiler directives — how to fork / join
  - how to assign *parallel regions* to threads
  - what data is *private* (local) to each thread
- Compiler **generates** multithreaded code
- Dependencies:
  - remove them OR explicitly synchronize

OpenMP Compiler Directives

- Within C code:

```
#pragma omp [directivename] [options]
```

- Compile C code containing OpenMP directives:

```
$ gcc -fopenmp …
```

OpenMP Compiler Directives

```
#pragma omp [directive] [clause ...]
                        if (scalar_expression)
                        private (list)
                        shared (list)
                        default (shared || none)
                        firstprivate (list)
                        reduction (operator: list)
                        copyin (list)
                        num_threads (integer-expression)
```

OpenMP Compiler Directives

- Examples

```
#pragma omp parallel [clause]
{
  // block of parallel code
}

#pragma omp parallel for [clause]
for (…)
{
    // for loop body
}

#pragma omp barrier
// wait for all threads to arrive here before proceeding
```

Hello World

- Each thread prints “hello, world”

```
#include “omp.h”
int main()
{
   #pragma omp parallel
   {
      int id = omp_get_thread_num();
      printf(“Hello(%d),”, id);
      printf(“world(%d)”, id);
   }
}
```

OpenMP header file

Parallel region with default number of threads

Runtime library function to get current thread #

End of parallel region

Demo

OpenMP Hello World