

CSCI465/ECEN433

Introduction to Parallel Computing

Spring 2024





OUR VISION

To be a world-class school recognized as one of the top information technology and computer science schools in the region for research, education, and entrepreneurship

OUR MISSION

The school is committed to preparing scientifically and professionally distinguished graduates in many information technology and computer science disciplines. It strives to: strongly contribute to society's prosperity; achieve sustainable development goals; and support the information technology industry through multidisciplinary scientific research, innovation and enhancement of entrepreneurial capabilities



CSCI465/ECEN433 – Introduction to Parallel Computing

Instructor:

Dr. Islam Tharwat

Assistant Professor of Computer Science, Information Technology and
Computer Science School

ihalim@nu.edu.eg

Office hours:

Mon 12:30 PM - 02:30 PM

Wen 08:30 AM - 10:30 AM

Location: (First Floor) 202

TA:

Eng. Bishoy Kamal Sharobim

b.kamal2160@nu.edu.eg

Course Description

Introduction to parallel computing for scientists and engineers. Shared memory parallel architectures and programming, distributed memory, message-passing data-parallel architectures, and programming.

Prerequisite:

PR: CSCI207/ECEN432

Course Aim

From smartphones to multi-core CPUs and GPUs, to the world's largest supercomputers, parallel processing is ubiquitous in modern computing. Upon completing this course, the student will have learned, through appropriate classroom and laboratory experiences, the following.

- A deep understanding of the fundamental principles and engineering trade-offs involved in designing modern parallel computing systems.
- The key machine performance characteristics, and how that affects software design.
- The fundamentals of parallel and distributed programming.
- The parallel programming techniques necessary to effectively utilize these machines.
- The basic algorithmic, programming, and software engineering issues associated with the development of parallel/distributed applications.
- Evaluating parallel programs and comparing the time complexity of parallel and sequential applications.

CSCI465/ECEN433 – Introduction to Parallel Computing

Knowledge and Understanding Skills	
A1	Define parallel computing principles, parallelism models, communication models, and resource limitations.
A2	Describe the fundamental steps for designing and analyzing parallel algorithms.
A3	Identify the essential mathematics relevant to the analysis of parallel algorithms.
Intellectual Skills	
B1	Analyze and improve the performance of parallel applications.
B2	Perform comparisons between parallel architectures, algorithms, methods, techniques...etc.
B3	Develop solutions to handle different communication structures
Professional Skills	
C1	Practice designing different parallel solutions
C2	Design parallel programming applications
C3	Quantify performance of parallel machines
Transferable Skills	
D1	Manage team projects
D2	Evaluate parallel designs based on statements of requirements
D3	Collaborate with project team members to get the term project implemented, documented, and presented.

CSCI465/ECEN433 – Introduction to Parallel Computing

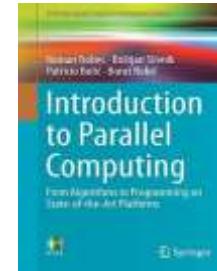
Grading Policy:	
Tutorial and Lab	10%
Quizzes	10%
Assignments	10%
Project	20%
Midterm	20%
Final	30%

CSCI465/ECEN433 – Introduction to Parallel Computing

References

Textbook

- “Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms”, by Roman Trobec, Boštjan Slivnik, Patricio Bulić, Borut Robič, (2018).
- “An Introduction to Parallel Programming”, by Peter S. Pacheco, 2nd edition (2021)
- "Introduction to Parallel Computing", Vipin Kumar, George Karypis, Anshul Gupta, Ananth Grama, Addison-wesley Professional, ISBN: 0201648652 (2003).



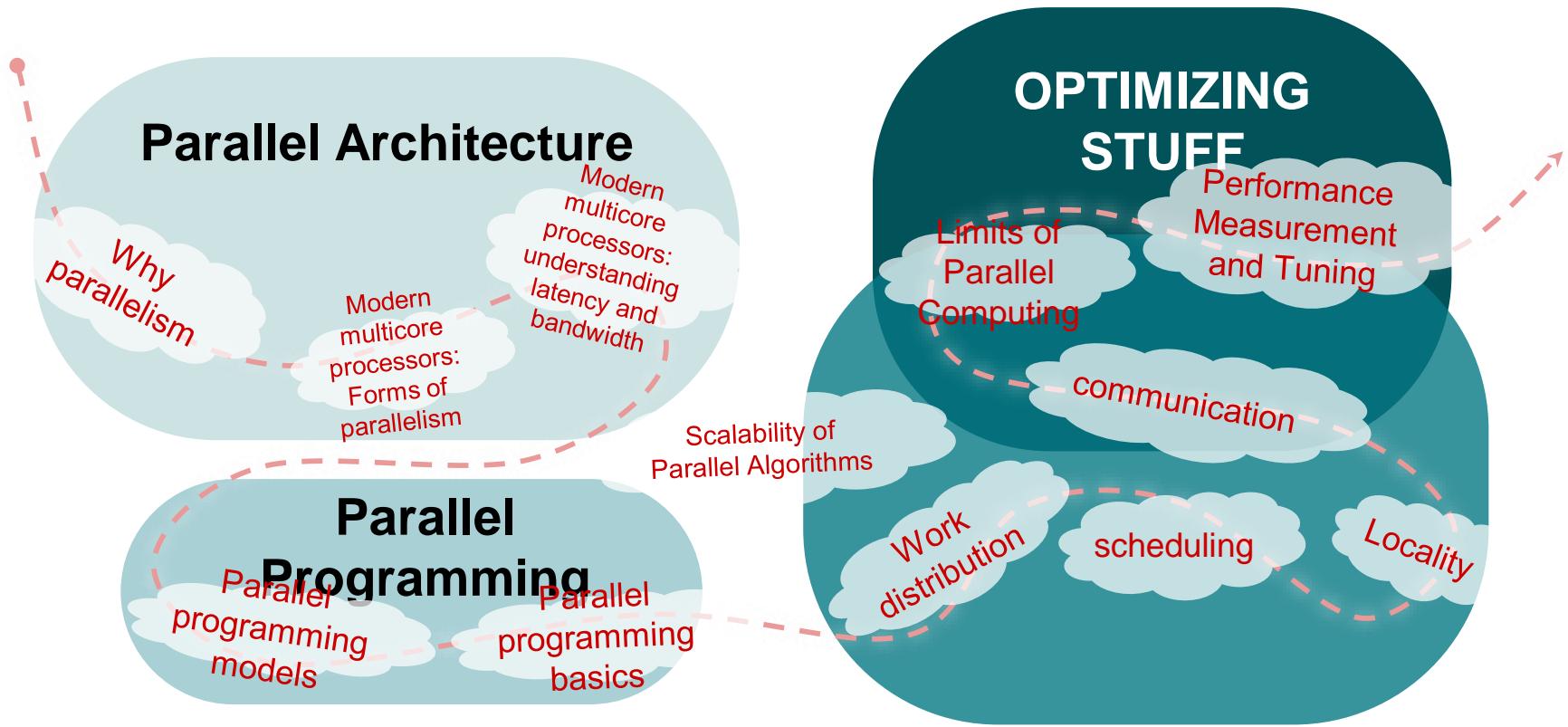
Handouts:

- [Introduction to Parallel Computing Tutorial](#)
- [Parallel Computer Architecture and Programming Carnegie Mellon University](#)

Course Contents

1. Introduction to Parallel Computing: Why Parallelism?
2. Modern multicore processors: Forms of parallelism
3. Modern multicore processors: understanding latency and bandwidth
4. Parallel Programming Abstractions (and their corresponding HW/SW implementations)
5. Parallel programming models (Shared address space, Message passing, and Data parallel)
6. Parallel Programming Basics
7. Performance Optimization: Work distribution and Scheduling
8. Performance Optimization: Locality, communication, and contention
9. Workload-driven performance evaluation
10. Limits of Parallel Computing
11. Scalability of Parallel Algorithms
12. Performance Measurement and Tuning

OUR TOPIC ROADMAP



Lecture (1)

**Why Parallelism?
Why Efficiency?**



A Brief History of Parallel Computing

- Initial Focus (starting in 1970s): “Supercomputers” for Scientific Computing



C.mmp at CMU (1971)
16 PDP-11 processors



Cray XMP (circa 1984)
4 vector processors



Thinking Machines CM-2 (circa 1987)
65,536 1-bit processors +
2048 floating-point co-processors



SGI UV 1000cc-NUMA (today)
4096 processor cores

← Blacklight at the Pittsburgh Supercomputer Center

A Brief History of Parallel Computing

- Initial Focus (starting in 1970s): “Supercomputers” for Scientific Computing
- Another Driving Application (starting in early ‘90s): Databases



Sun Enterprise 10000 (circa 1997)
16 UltraSPARC-II processors



Oracle Supercluster M6-32 (today)
32 SPARC M2 processors

A Brief History of Parallel Computing

- Cloud computing (2000–present)
- Build out massive centers with many, simple processors
 - Connected via LAN technology
- Program using distributed-system models



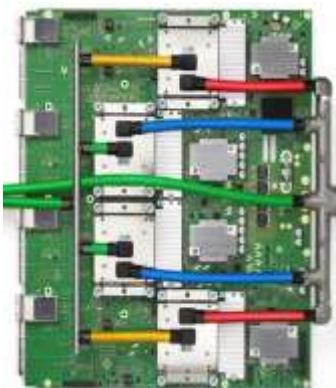
Today: Next-generation ML infrastructure for large Model training

- Current Focus: “Supercomputers” for Machine Learning



TPU v4 Cluster Google (2023)

4096 Chips per Pod

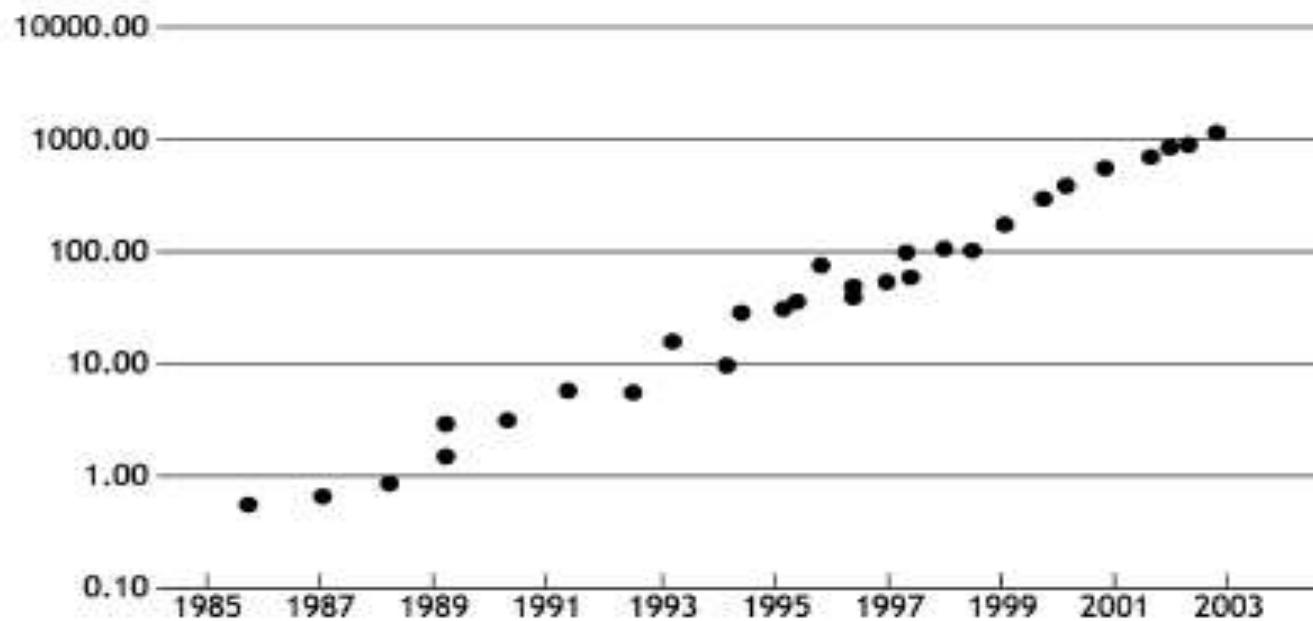


TPU v4 Chip (2023)

275 TeraFlops

Setting Some Context

- Before we continue our multiprocessor story, let's pause to consider:
 - Q: what had been happening with single-processor performance?
- A: since forever, they had been getting exponentially faster
 - Why?

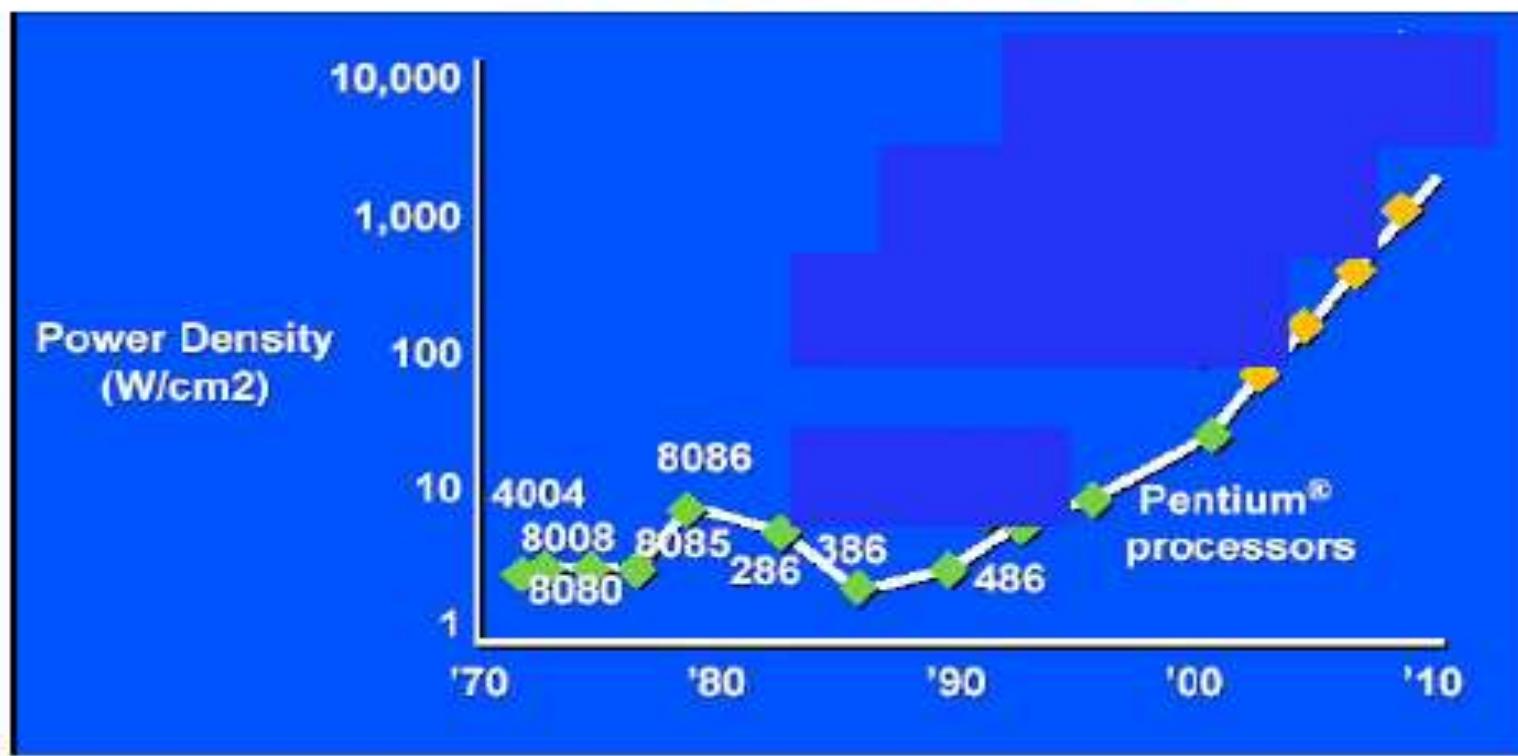


A Brief History of Processor Performance

- Wider data paths
 - 4 bit → 8 bit → 16 bit → 32 bit → 64 bit
- More efficient pipelining
 - e.g., 3.5 Cycles Per Instruction (CPI) → 1.1 CPI
- Exploiting instruction-level parallelism (ILP)
 - “superscalar” processing: e.g., issue up to 4 instructions/cycle
- Faster clock rates
 - e.g., 10 MHz → 200 MHz → 3 GHz
- During the 80s and 90s: large exponential performance gains
 - and then...

A Brief History of Parallel Computing

- Initial Focus (starting in 1970s): “Supercomputers” for Scientific Computing
- Another Driving Application (starting in early ‘90s): Databases
- Inflection point in 2004: Intel hits the Power Density Wall



Pat Gelsinger, ISSCC 2001

From the New York Times

Intel's Big Shift After Hitting Technical Wall

The warning came first from a group of hobbyists that tests the speeds of computer chips. This year, the group discovered that the Intel Corporation's newest microprocessor was running slower and hotter than its predecessor.

What they had stumbled upon was a major threat to Intel's longstanding approach to dominating the semiconductor industry - relentlessly raising the clock speed of its chips.

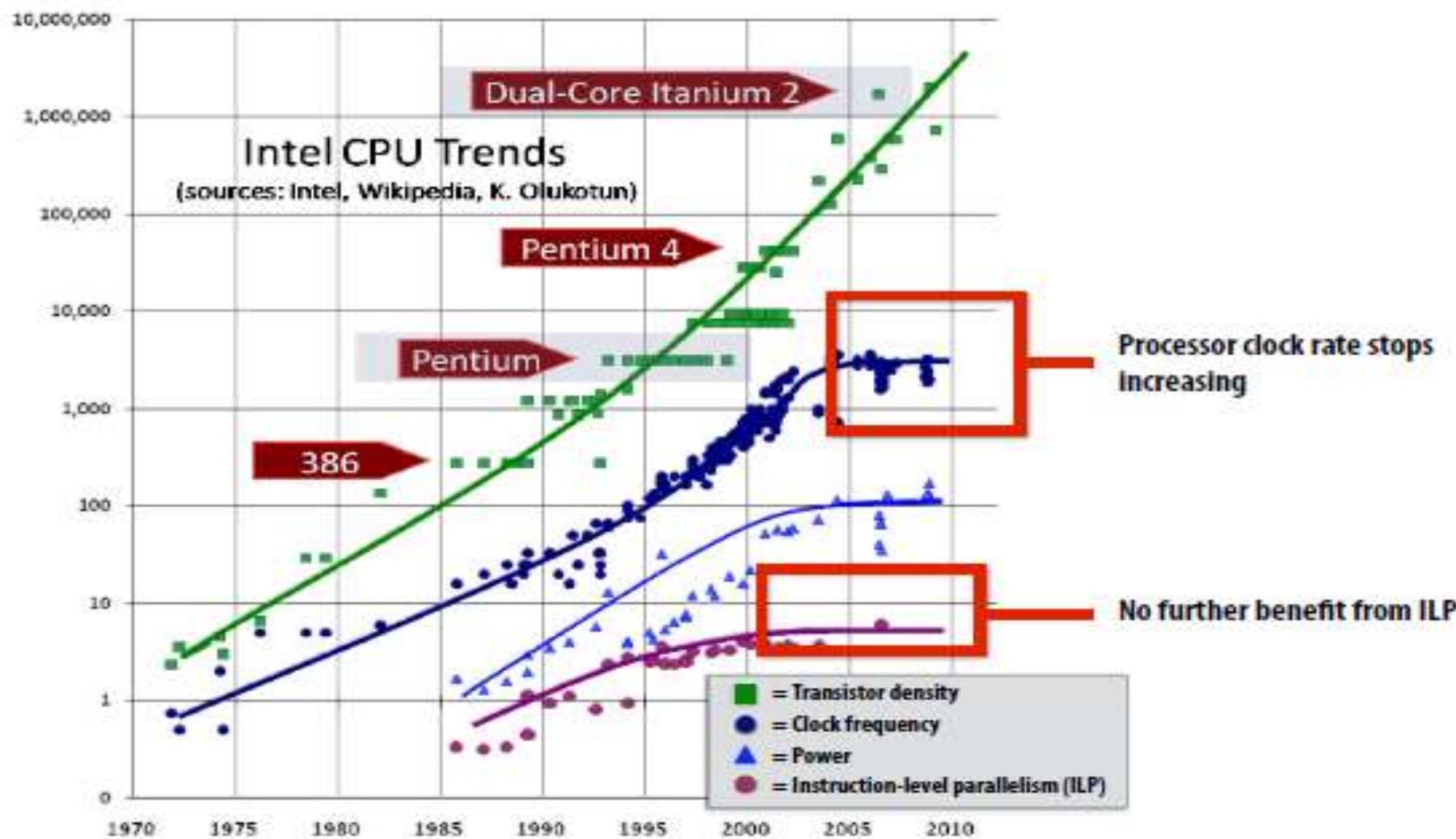
Then two weeks ago, Intel, the world's largest chip maker, publicly acknowledged that it had hit a "thermal wall" on its microprocessor line. As a result, the company is changing its product strategy and disbanding one of its most advanced design groups. Intel also said that it would abandon two advanced chip development projects, code-named Tejas and Jayhawk.

Now, Intel is embarked on a course already adopted by some of its major rivals: obtaining more computing power by stamping multiple processors on a single chip rather than straining to increase the speed of a single processor.

...

John Markoff, New York Times, May 17, 2004

ILP tapped out + end of frequency scaling



Approaches to the serial problem

- Write translation programs that automatically convert serial programs into parallel programs.
 - This is very difficult to do.
 - Success has been limited.
- Rewrite serial programs so that they're parallel.

Programmer's Perspective on Performance

Question: How do you make your program run faster?

Answer before 2004:

- Just wait 6 months, and buy a new machine!
- (Or if you're really obsessed, you can learn about parallelism.)

Answer after 2004:

- You need to write parallel software.



Parallel Machines Today

Examples from Apple's product line:



Mac Pro
12 Intel Xeon E5 cores



iMac
12 Intel Xeon E5 cores



MacBook Pro Retina 15"
4 Intel Core i7 cores



iPad Retina
2 Swift cores

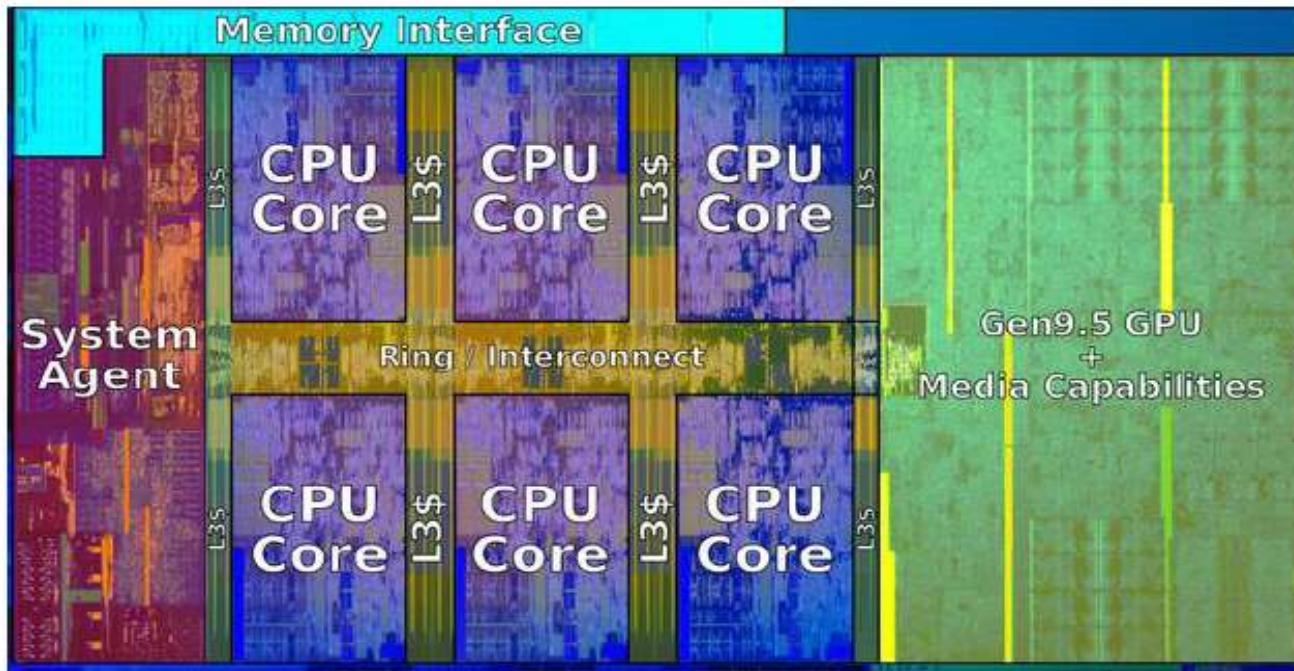


iPhone XS
6 CPU cores
(2 fast +
4 low power)
6 GPU cores

(images from [apple.com](#))

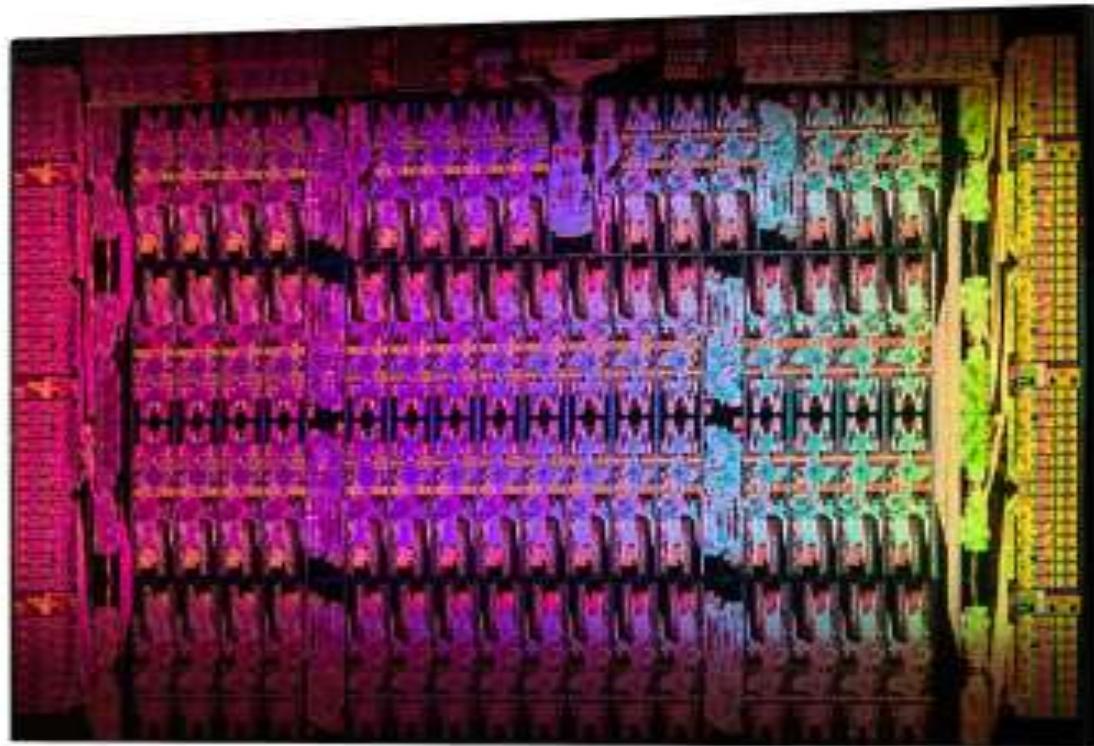
Intel Coffee Lake Core i9 (2019)

6-core CPU + multi-core GPU integrated on one chip



Intel Xeon Phi 7120A “coprocessor”

- 61 “simple” x86 cores (1.3 Ghz, derived from Pentium)
- Targeted as an accelerator for supercomputing applications



NVIDIA GeForce GTX 1660 Ti GPU (2019)

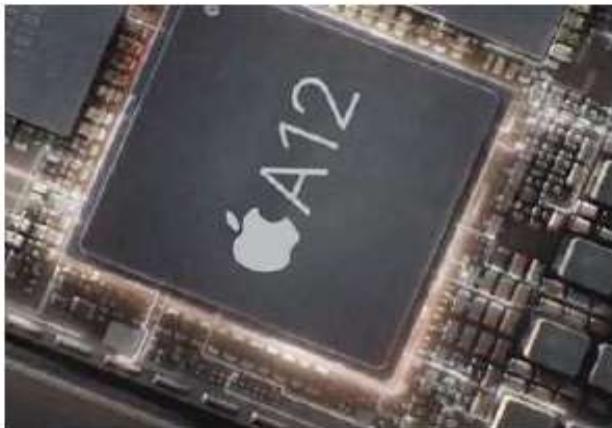
24 major processing blocks

(but much, much more parallelism available... details coming soon)

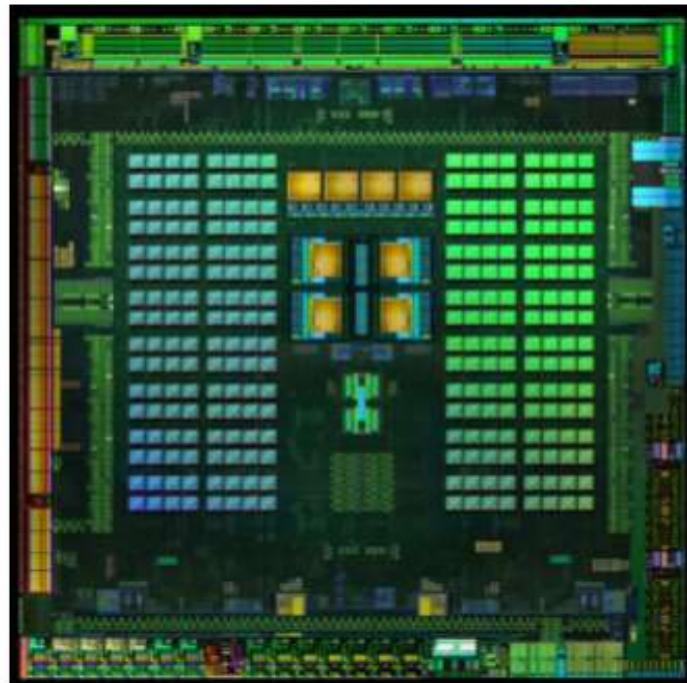


Mobile parallel processing

Power constraints heavily influence design of mobile systems



Apple A12: (in iPhone XR)
4 CPU cores
4 GPU cores
Neural net engine
+ much more



NVIDIA Tegra K1:
Quad-core ARM A57 CPU + 4 ARM A53 CPUs +
NVIDIA GPU + image processor...

Supercomputing

- Today: clusters of multi-core CPUs + GPUs
- Oak Ridge National Laboratory: Summit (#1 supercomputer in world)
 - 4,608 nodes
 - Each with two 22-core CPUs + 6 GPUs
- Oak Ridge National Laboratory: Titan (#2 supercomputer in world)
 - 18,688 x 16 core AMD CPUs + 18,688 NVIDIA K20X GPUs



The Final Frontier: US Has Its First Exascale Supercomputer (**Frontier**)

- (For more on all of that, read *HPCwire's* [Top500 coverage](#).)



What is next?

	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C Instinct MI250X Slingshot-11, Oak Ridge National Laboratory	8,730,112	1,102.00	1,685.65	21,100
Computer Fugaku - Supercomputer 64FX 48C 2.2GHz, Tefu Project, D. Fujitsu Center for Computational Science	7,630,848	442.01	537.21	29,899
HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X Slingshot-11, HPE CSC	2,220,288	309.10	428.70	6,016
BullSequana XH2000, Xeon 3358 32C 2.6GHz, NVIDIA A100 4B_Quad-rail NVIDIA HDR100, Atos CINECA	1,463,616	174.70	255.75	5,610



From the New York Times

Intel's Big Shift After Hitting Technical Wall

The warning came first from a group of hobbyists that tests the speeds of computer chips. This year, the group discovered that the Intel Corporation's newest microprocessor was running slower and hotter than its predecessor.

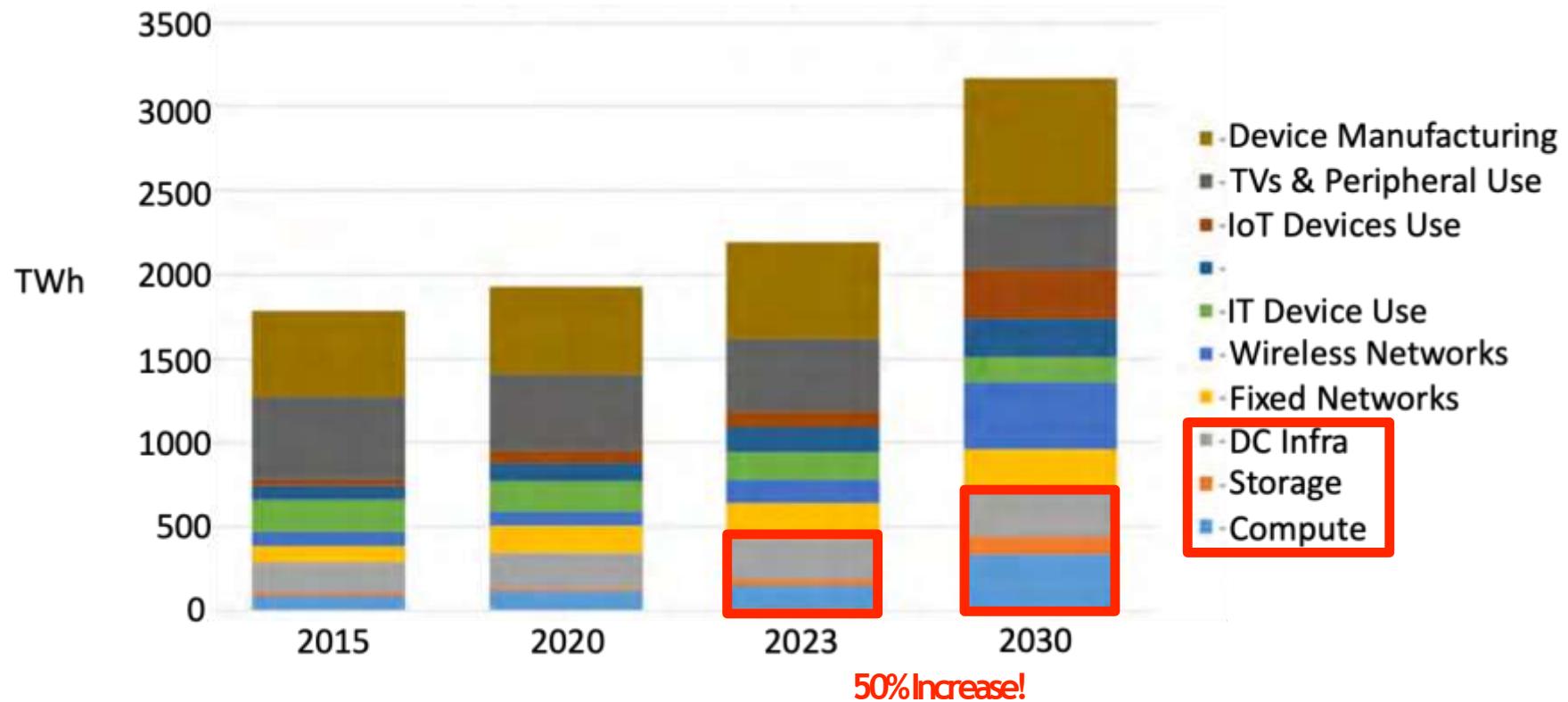
What they had stumbled upon was a major threat to Intel's longstanding approach to dominating the semiconductor industry - relentlessly raising the clock speed of its chips.

Then two weeks ago, Intel, the world's largest chip maker, publicly acknowledged that it had hit a "thermal wall" on its microprocessor line. As a result, the company is changing its product strategy and disbanding one of its most advanced design groups. Intel also said that it would abandon two advanced chip development projects, code-named Tejas and Jayhawk.

Now, Intel is embarked on a course already adopted by some of its major rivals: obtaining more computing power by stamping multiple processors on a single chip rather than straining to increase the speed of a single processor.

... John Markoff, New York Times, May 17, 2004

Power Consumption of Datacenters



What is a parallel computer?

One common definition

A parallel computer is a collection of processing elements that cooperate to solve problems quickly

We care about performance *
We care about efficiency

We're going to use multiple
processors to get it

DEMO 1

(This semester's first parallel program)

Speedup

One major motivation of using parallel processing: achieve a speedup

For a given problem:

$$\text{speedup(using P processors)} = \frac{\text{execution time (using 1 processor)}}{\text{execution time (using P processors)}}$$

Class observations from demo 1

- Communication limited the maximum speedup achieved
 - In the demo, the communication was telling each other the partial sums
- Minimizing the cost of communication improves speedup
 - Moving students (“processors”) closer together (or let them shout)

DEMO 2

(scaling up to four “processors”)

Class observations from demo 2

- **Imbalance in work assignment limited speedup**
 - Some students (“processors”) ran out work to do (went idle), while others were still working on their assigned task
- **Improving the distribution of work improved speedup**

DEMO 3

(massively parallel execution)

Class observations from demo 3

- The problem I just gave you has a significant amount of communication compared to computation
- Communication costs can dominate a parallel computation, severely limiting speedup

What we'll be doing

- Learning to write programs that are explicitly parallel.
- Using the C language.
- Using three different extensions to C.
 - Message-Passing Interface (MPI)
 - OpenMP

Course theme 1:

Parallel computer hardware implementation: how parallel computers work

- Mechanisms used to implement abstractions efficiently
 - Performance characteristics of implementations
 - Design trade-offs: performance vs. convenience vs. cost
- Why do I need to know about hardware?
 - Because the characteristics of the machine really matter
(recall speed of communication issues in earlier demos)
 - Because you care about efficiency and performance
(you are writing parallel programs after all!)

Course theme 2:

Designing and writing parallel programs ... that scale!

- Parallel thinking
 1. Decomposing work into pieces that can safely be performed in parallel
 2. Assigning work to processors
 3. Managing communication/synchronization between the processors so that it does not limit speedup
- Abstractions/mechanisms for performing the above tasks
 - Writing code in popular parallel programming languages

Course theme 3:

Thinking about efficiency

- **FAST != EFFICIENT**
- Just because your program runs faster on a parallel computer, it does not mean it is using the hardware efficiently
 - Is 2x speedup on computer with 10 processors a good result?
- Programmer's perspective: make use of provided machine capabilities
- HW designer's perspective: choosing the right capabilities to put in system (performance/cost, cost = silicon area?, power?, etc.)

Summary

- Today, single-thread performance is improving very slowly
 - To run programs significantly faster, programs must utilize multiple processing elements
 - Which means you need to know how to write parallel code
- Writing parallel programs can be challenging
 - Requires problem partitioning, communication, synchronization
 - Knowledge of machine characteristics is important
- I suspect you will find that modern computers have tremendously more processing power than you might realize, if you just use it!
- Welcome to ECEN433!

Questions

CSCI465/ECEN433

Introduction to Parallel Computing

Spring 2024



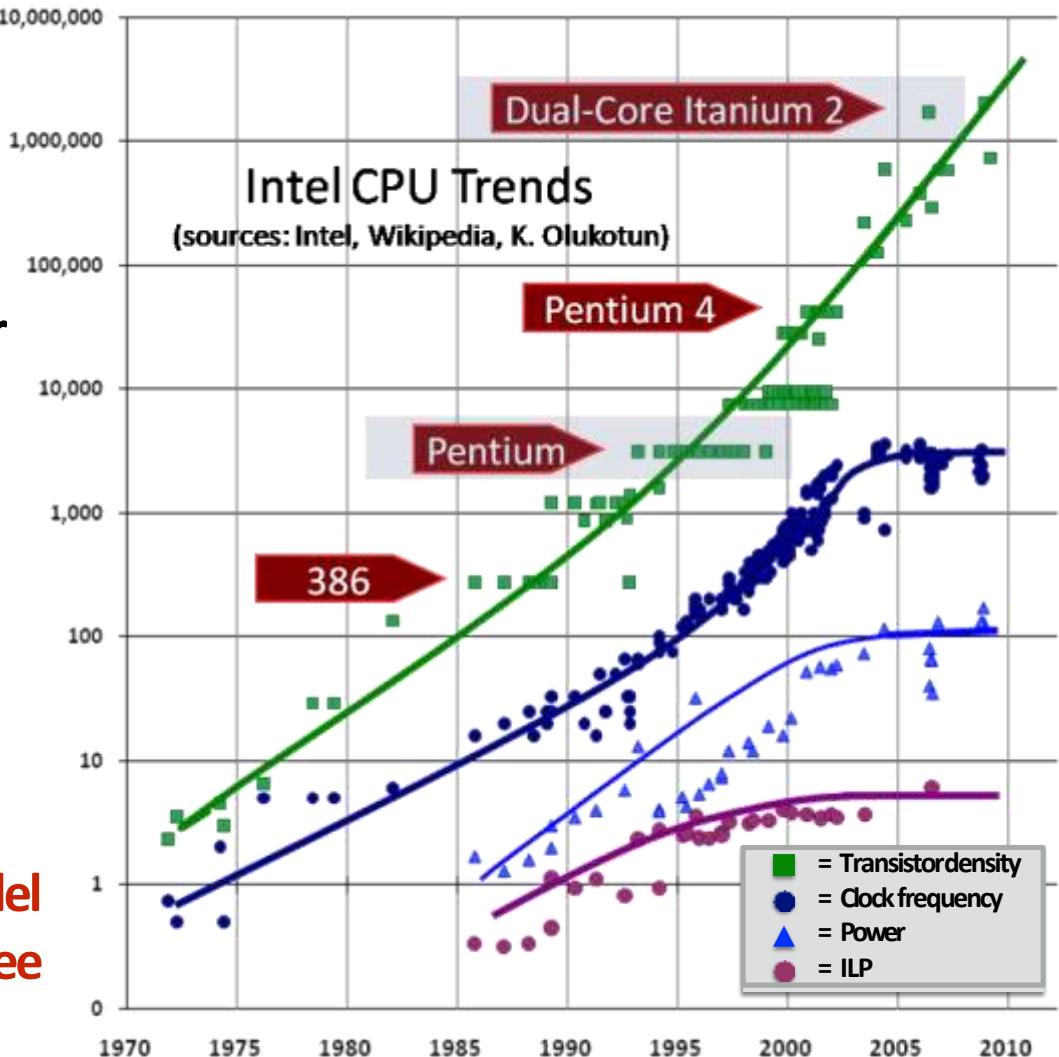
Recap: Single-core performance scaling

The rate of single-instruction stream performance scaling has decreased (almost to zero)

1. Frequency scaling limited by power
2. ILP scaling tapped out

Architects are now building faster processors by adding more execution units that run in parallel.

Software must be written to be parallel to see performance gains. No more free lunch for software developers!



Recap: why parallelism?

- **The answer 15 years ago**
 - To realize performance improvements that exceeded what CPU performance improvements could provide
 - (specifically, in the early 2000's, what clock frequency scaling could provide)
 - Because if you just waited until next year, your code would run faster on the next generation CPU
- **The answer today:**
 - Because it is the primary way to achieve significantly higher application performance for the foreseeable future *

Lecture (2)

A Modern Multi-Core Processor

(Forms of parallelism + understanding latency and bandwidth)



This Lecture

- Today we will talk computer architecture
- Four key concepts about how modern computers work
 - Two concern parallel execution
 - Two concern challenges of accessing memory
- Understanding these architecture basics will help you
 - Understand and optimize the performance of your parallel programs
 - Gain intuition about what workloads might benefit from fast parallel machines

Part 1: parallel execution

What is a program? (from a processor's perspective)

A program is just a list of processor instructions!

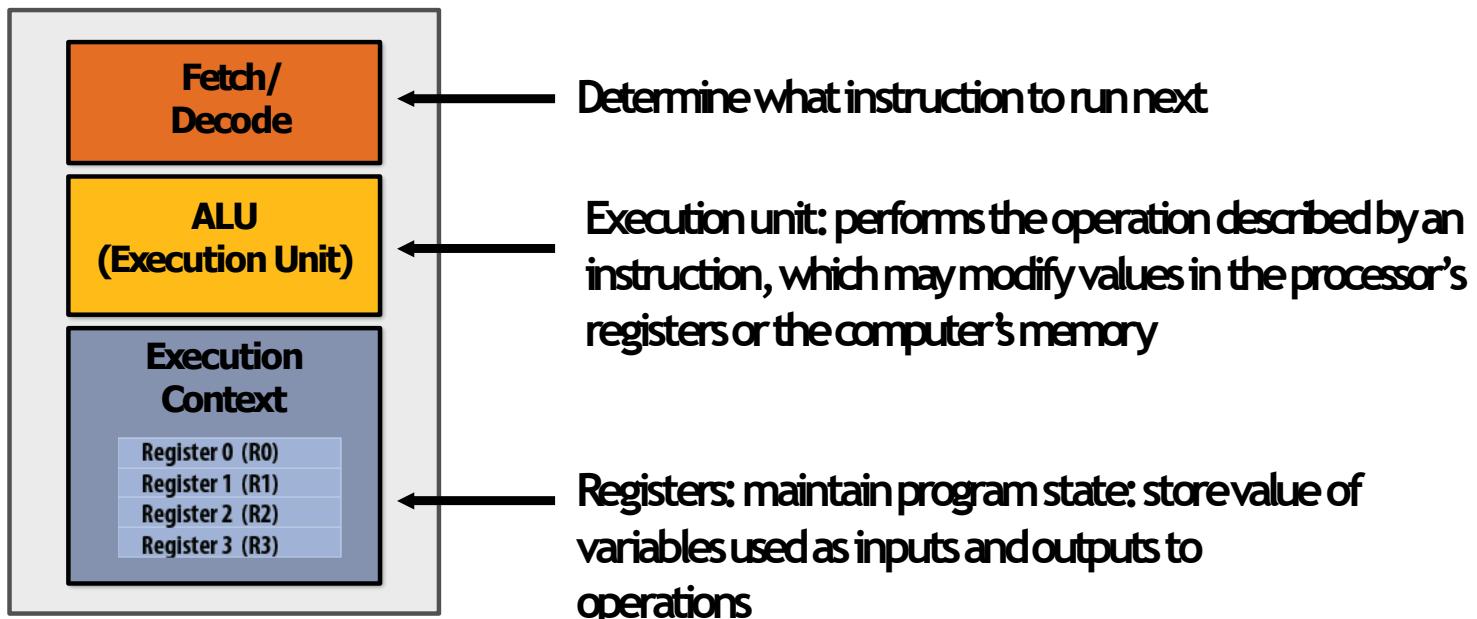
```
int main(int argc, char** argv) {  
  
    int x = 1;  
  
    for (int i=0; i<10; i++) {  
        x = x + x;  
    }  
  
    printf("%d\n", x);  
  
    return 0;  
}
```



```
_main:  
100000f10: pushq   %rbp  
100000f11: movq %rsp, %rbp  
100000f14: subq $32, %rsp  
100000f18: movl $0, -4(%rbp)  
100000f1f: movl %edi, -8(%rbp)  
100000f22: movq %rsi, -16(%rbp)  
100000f26: movl $1, -20(%rbp)  
100000f2d: movl $0, -24(%rbp)  
100000f34: cmpl $10, -24(%rbp)  
100000f38: jge 23 <_main+0x45>  
100000f3e: movl -20(%rbp), %eax  
100000f41: addl -20(%rbp), %eax  
100000f44: movl %eax, -20(%rbp)  
100000f47: movl -24(%rbp), %eax  
100000f4a: addl $1, %eax  
100000f4d: movl %eax, -24(%rbp)  
100000f50: jmp -33 <_main+0x24>  
100000f55: leaq 58(%rip), %rdi  
100000f5c: movl -20(%rbp), %esi  
100000f5f: movb $0, %al  
100000f61: callq 14  
100000f66: xorl %esi, %esi  
100000f68: movl %eax, -28(%rbp)  
100000f6b: movl %esi, %eax  
100000f6d: addq $32, %rsp  
100000f71: popq %rbp  
100000f72: rets
```

A processor executes instructions

A Very Simple Processor



what does a processor do?

It runs programs!

Execute instruction referenced
by the program counter (PC)
(executing the instruction will modify
machine state: contents of registers,
memory, CPU state, etc.)

Move to next instruction ...

Then execute it ...



And so on ...

```
401200: xor    %eax,%eax
401202: xor    %r9d,%r9d
401205: jmp    401222 <_Z12verifyResultPiS_ii+0x42>
401207: nopw
40120e:
401210: mov    0x4(%rdi,%rax,1),%r10d
401215: add    $0x4,%rax
401219: mov    (%r11,%rax,1),%r8d
40121d: cmp    %r8d,%r10d
401220: jne    401248 <_Z12verifyResultPiS_ii+0x68>
401222: add    $0x1,%r9d
401226: cmp    %edx,%r9d
401229: jne    401210 <_Z12verifyResultPiS_ii+0x30>
40122b: add    $0x1,%esi
40122e: add    %rbx,%rdi
401231: add    %rbx,%r11
401234: cmp    %ecx,%esi
401236: jne    4011f1 <_Z12verifyResultPiS_ii+0x11>
401238: mov    $0x1,%eax
40123d: pop    %rbx
40123e: retq
40123f: xor    %r9d,%r9d
401242: nopw
401248: mov    %r10d,%ecx
40124b: mov    %r9d,%edx
40124e: mov    $0x401be8,%edi
401253: xor    %eax,%eax
401255: callq  400a40 <printf@plt>
40125a: xor    %eax,%eax
40125c: pop    %rbx
40125d: retq
40125e: mov    $0x1,%eax
401263: retq
```

Instruction level parallelism (ILP)

- Processors did in fact leverage parallel execution to make programs run faster, it was just invisible to the programmer

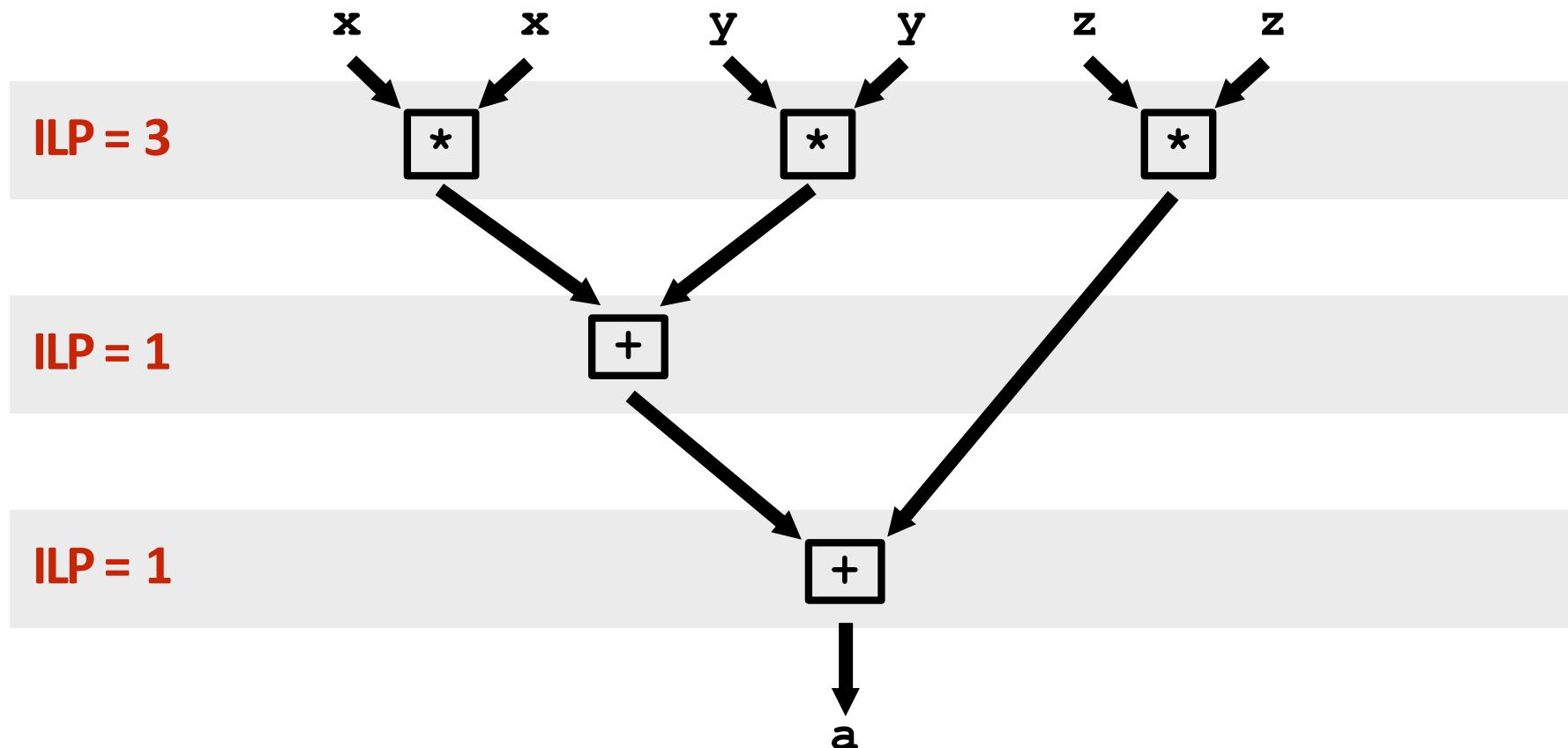
- Instruction level parallelism (ILP)

- Idea: Instructions must appear to be executed in program order. BUT independent instructions can be executed simultaneously by a processor without impacting program correctness
- Superscalar execution: processor dynamically finds independent instructions in an instruction sequence and executes them in parallel



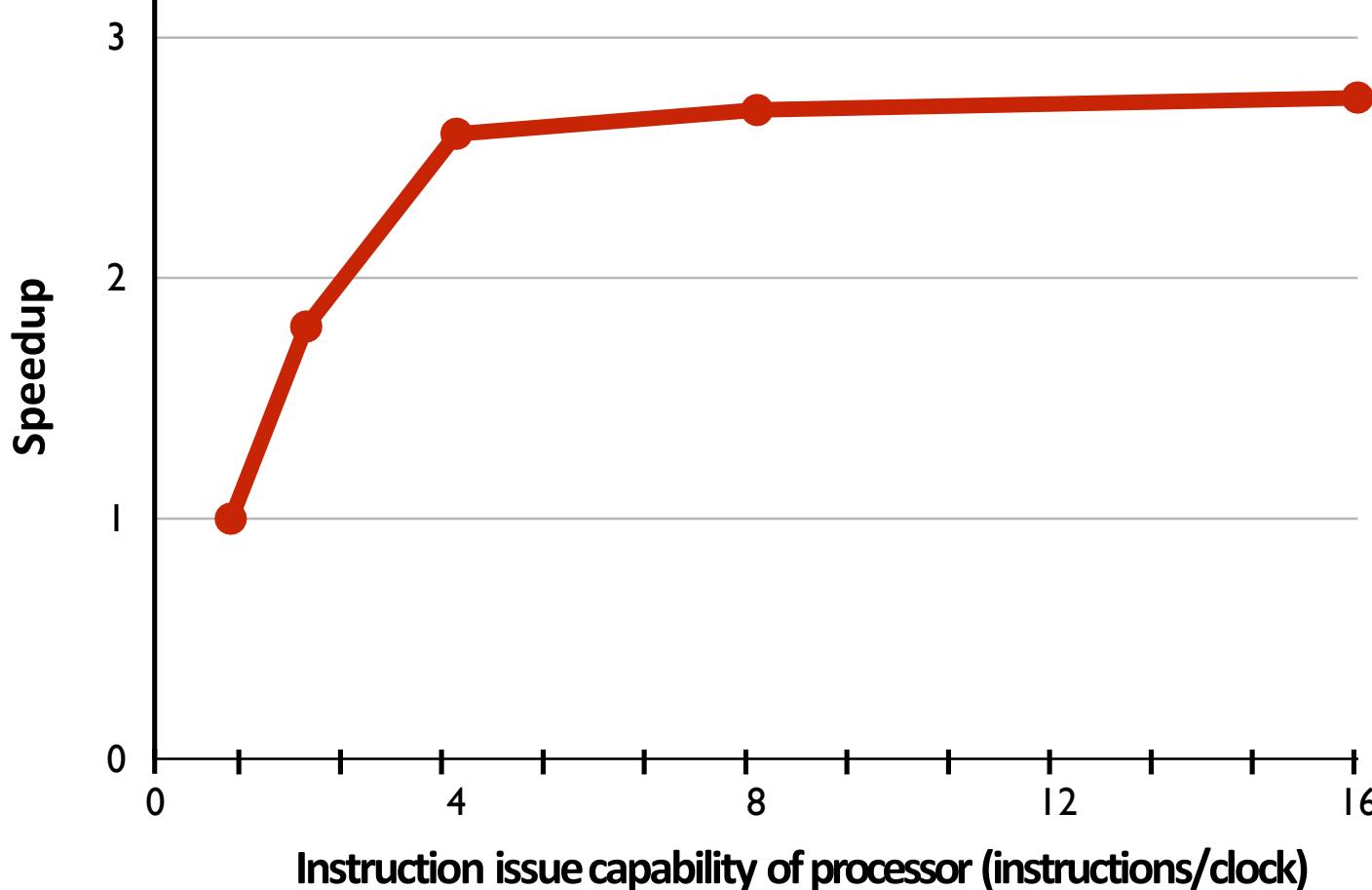
ILP example

$$a = x*x + y*y + z*z$$



Diminishing returns of superscalar execution

Most available ILP is exploited by a processor capable of issuing four instructions per clock
(Little performance benefit from building a processor that can issue more)



Example program

Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$
for each element of an array of N floating-point numbers

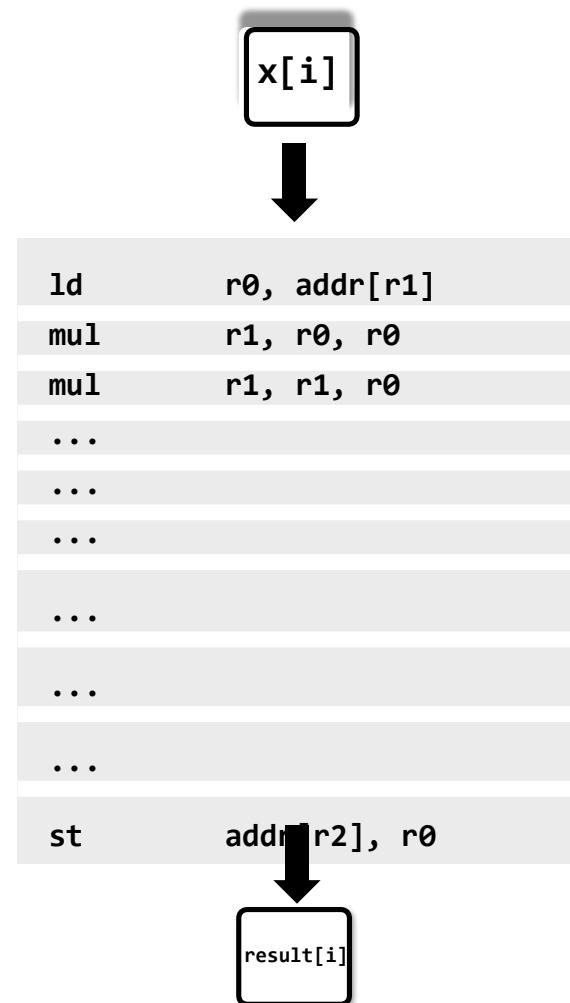
```
void sinx(int N, int terms, float* x, float* result)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

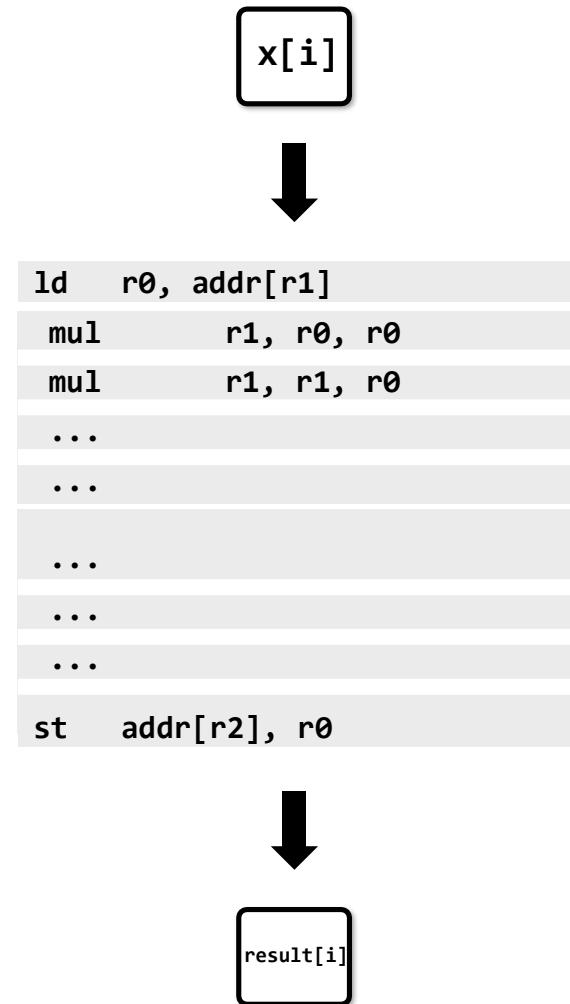
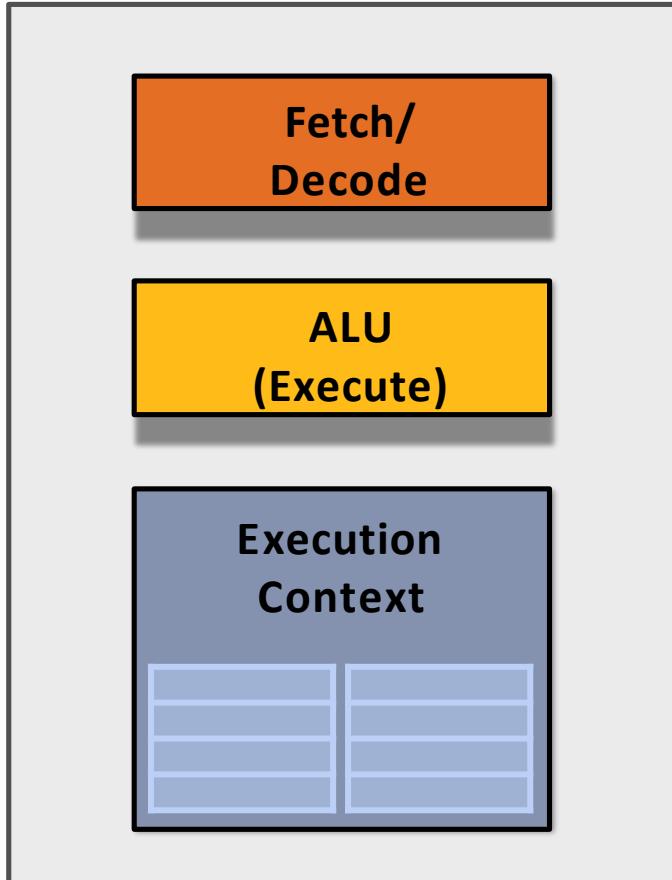
        result[i] = value;
    }
}
```

Compile program

```
void sinx(int N, int terms, float* x, float*
result)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] *
x[i];    int denom = 6;      // 3!
        int sign = -1;
        for (int j=1; j<=terms; j++)
        {
            value += sign * numer /
denom;    numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[i] = value;
    }
}
```

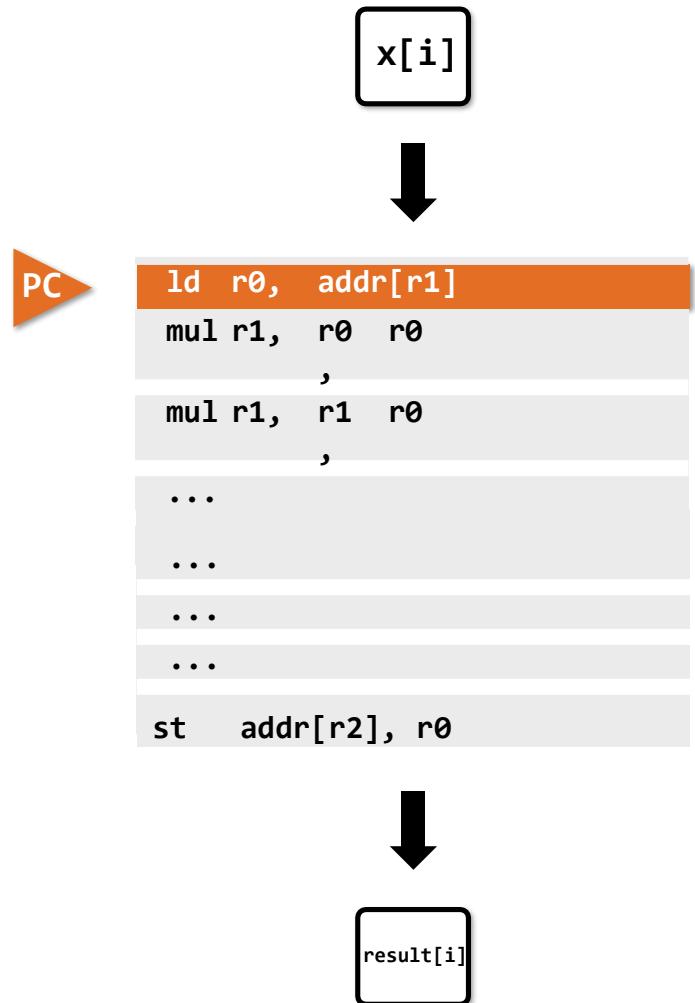
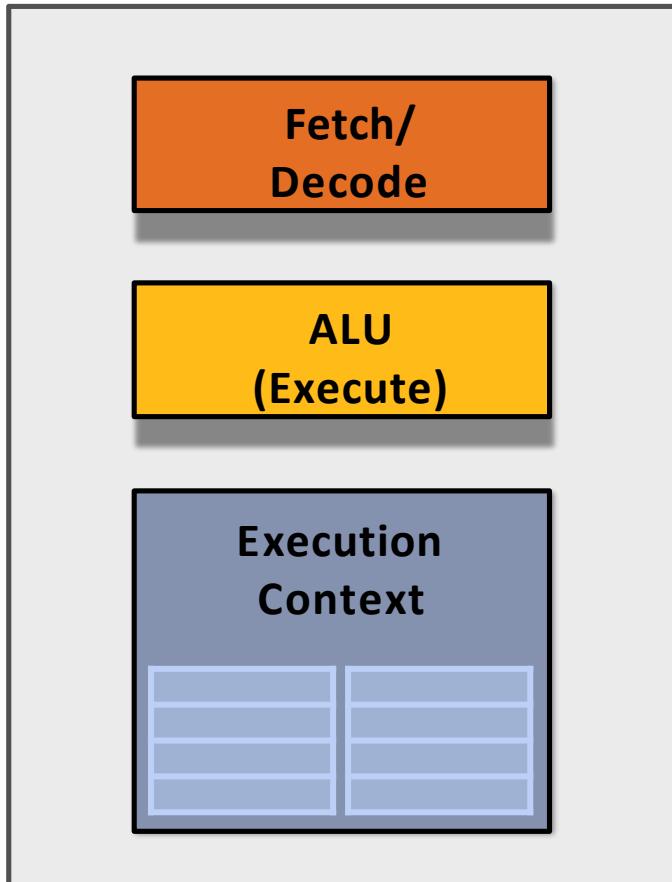


Execute program



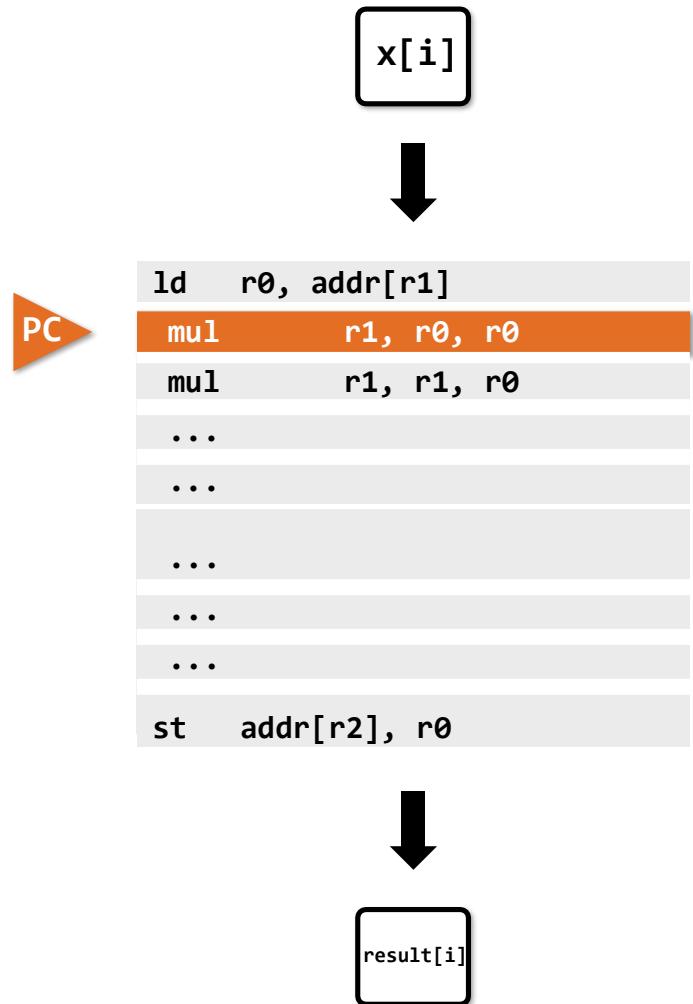
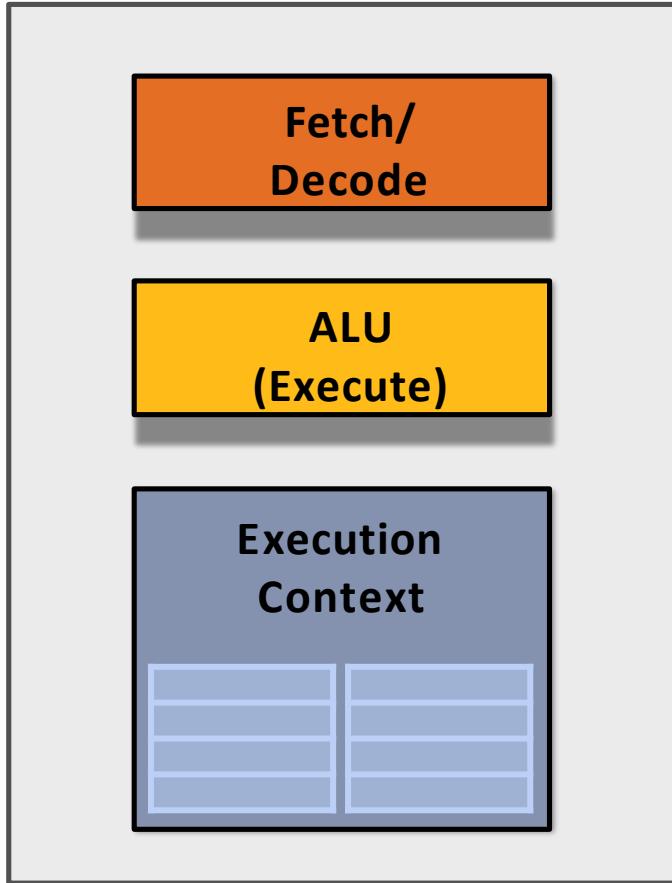
Execute program

My very simple processor: executes one instruction per clock



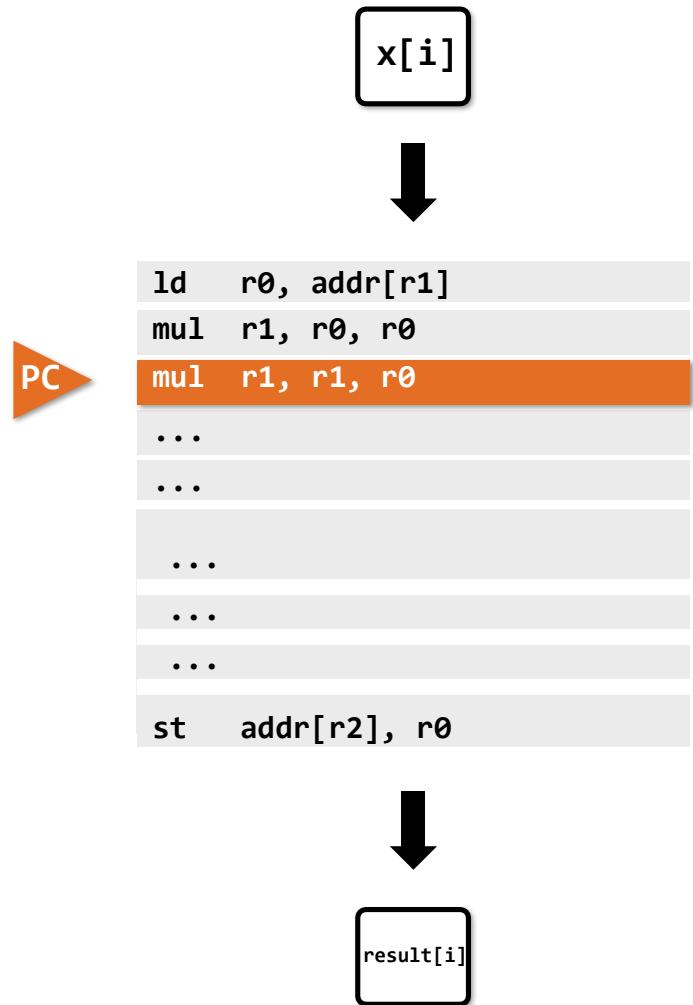
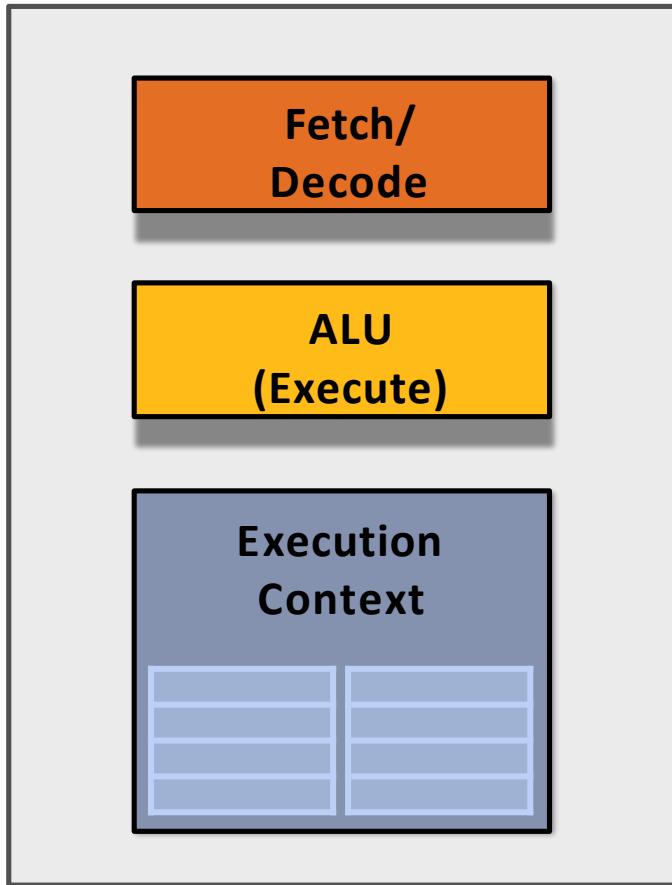
Execute program

My very simple processor: executes one instruction per clock



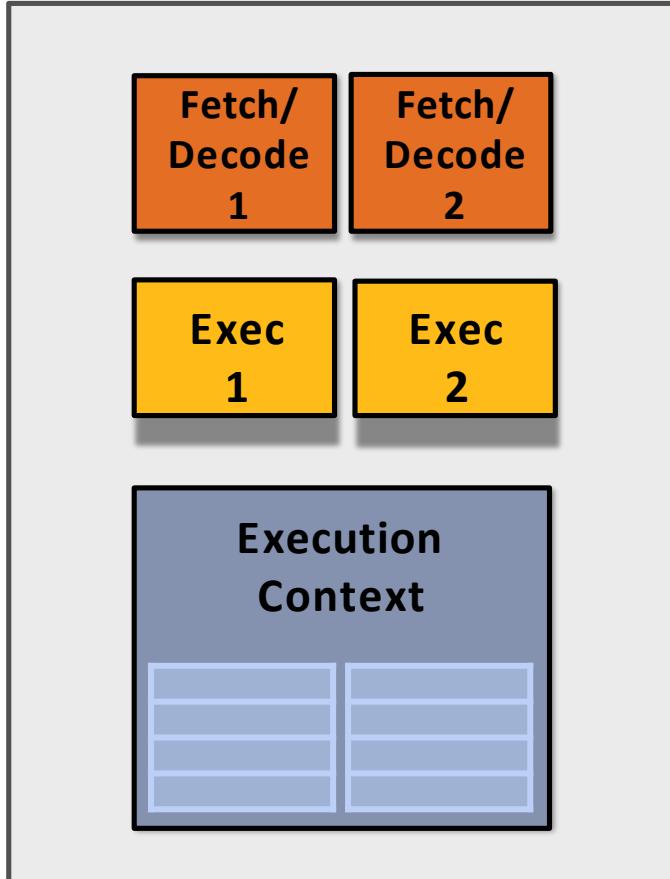
Execute program

My very simple processor: executes one instruction per clock



Superscalar processor

Recall from last class: instruction level parallelism (ILP) Decode and execute two instructions per clock (if possible)

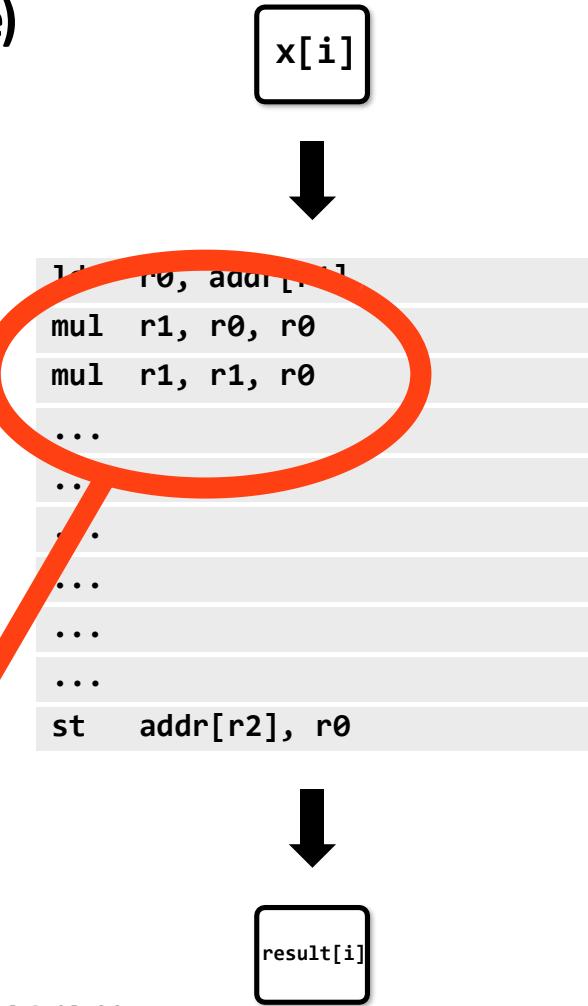
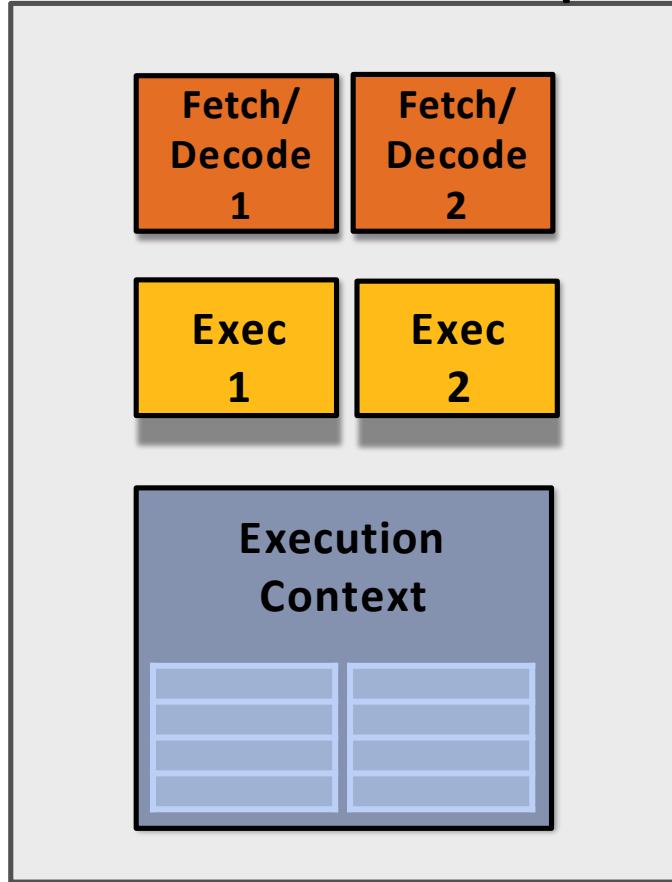


Superscalar execution: processor automatically finds independent instructions in an instruction sequence and can execute them in parallel on multiple execution units.

Superscalar processor

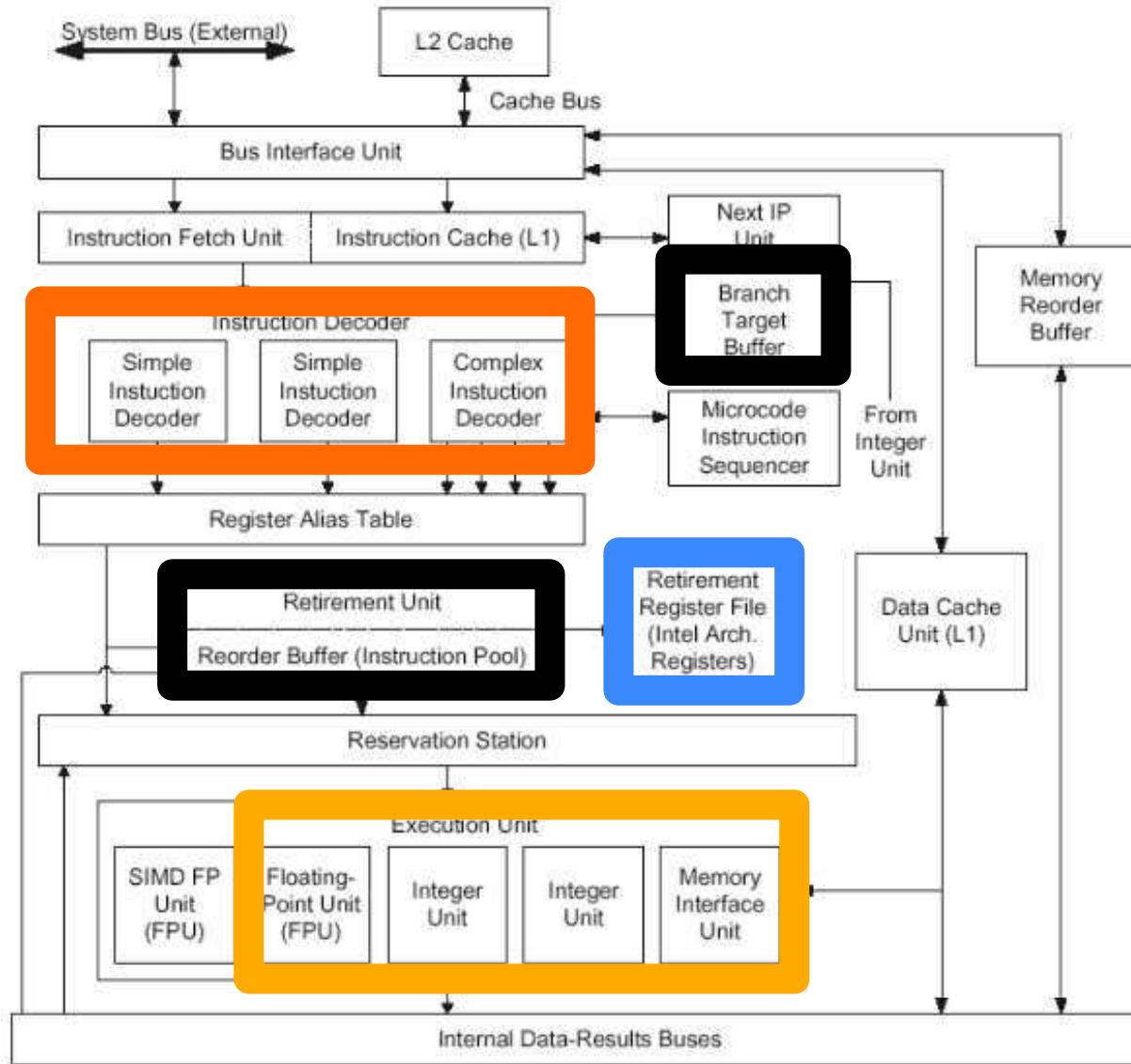
Recall from last class: instruction level parallelism (ILP)

Decode and execute two instructions per clock (if possible)



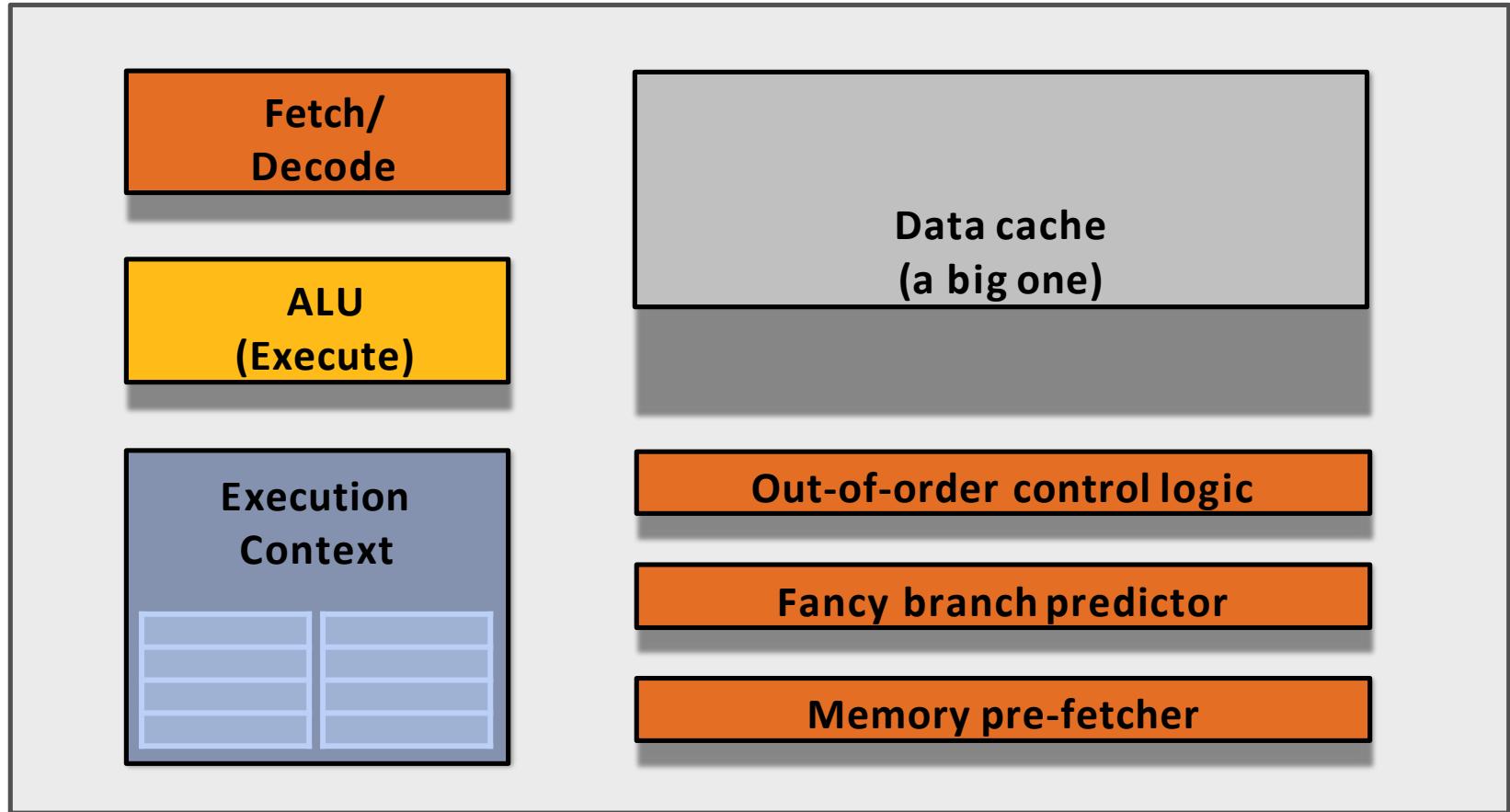
Note: No ILP exists in this region of the program

Aside: Pentium 4



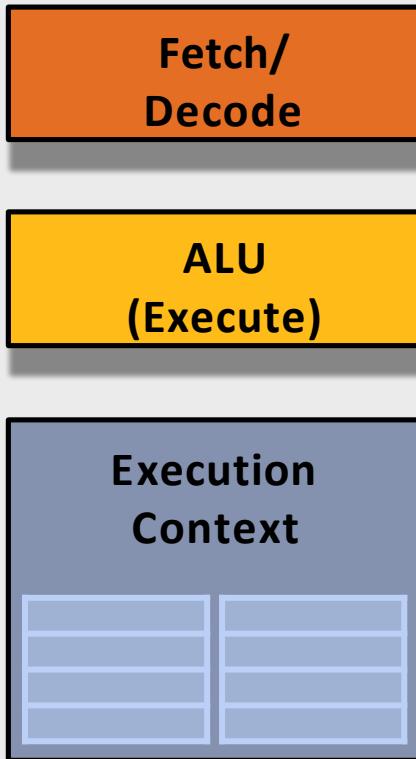
Processor: pre multi-core era

Majority of chip transistors used to perform operations
that help a single instruction stream run fast



More transistors = larger cache, smarter out-of-order logic, smarter branch predictor,
etc. (Also: more transistors → smaller transistors → higher clock frequencies)

Processor: multi-core era

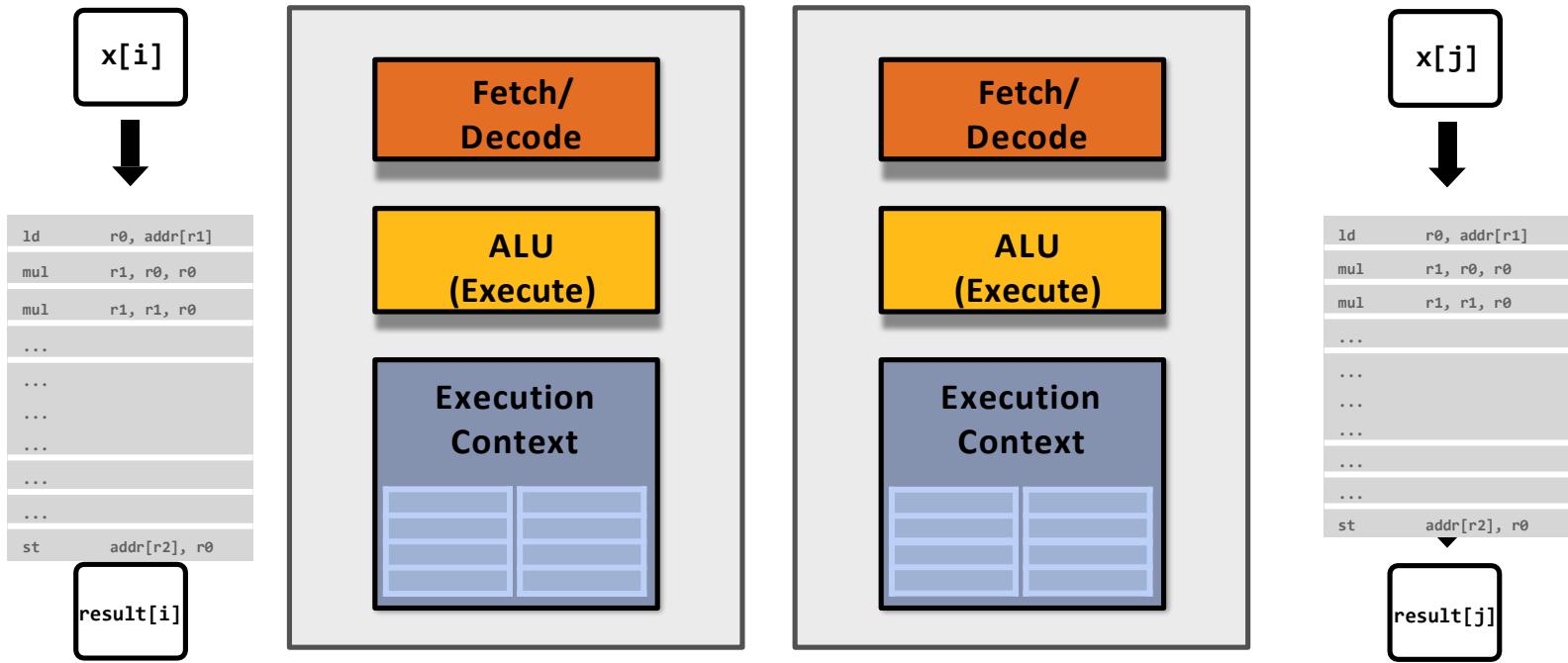


Idea #1:

Use increasing transistor count to add more cores to the processor

Rather than use transistors to increase sophistication of processor logic that accelerates a single instruction stream (e.g., out-of-order and speculative operations)

Twocores: compute two elements in parallel



Simpler cores: each core is slower at running a single instruction stream than our original “fancy” core (e.g., 25% slower)

But there are now two cores: $2 \times 0.75 = 1.5$ (potential for speedup!)

But our program expresses no parallelism

```
void sinx(int N, int terms, float* x, float*  
result)  
{  
    for (int i=0; i<N; i++)  
    {  
        float value = x[i];  
        float numer = x[i] * x[i] *  
            x[i];    int denom = 6;      // 3!  
        int sign = -1;  
        for (int j=1; j<=terms; j++)  
        {  
            value += sign * numer /  
                denom;    numer *= x[i] * x[i];  
            denom *= (2*j+2) * (2*j+3);  
            sign *= -1;  
        }  
  
        result[i] = value;  
    }  
}
```

This program, compiled with gcc will run as one thread on one of the processor cores.

If each of the simpler processor cores was 25% slower than the original single complicated one, our program now runs 25% slower.:-)

Expressing parallelism using pthreads

```
typedef struct {
    int N;
    int terms;  float*
    x;  float* result;
} my_args;

void parallel_sinx(int N, int terms, float* x, float* result)
{
    pthread_t thread_id;
    my_args args;

    args.N = N/2;  args.terms
    = terms;  args.x = x;
    args.result = result;

    pthread_create(&thread_id, NULL, my_thread_start, &args); // launch thread
    sinx(N -args.N, terms, x + args.N, result + args.N); // do work
    pthread_join(thread_id, NULL);
}

void my_thread_start(void* thread_arg)
{
    my_args* thread_args = (my_args*)thread_arg;
    sinx(args--N, args-->terms, args-->x, args-->result); // do work
}
```

```
void sinx(int N, int terms, float* x, float* result)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] *
        x[i];  int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer /
            denom  numer *= x[i] * x[i];
            denom *= (2*j+2) *
            (2*j+3);  sign *= -1;
        }

        result[i] = value;
    }
}
```

Data-parallel expression

(in our fictitious data-parallel language)

```
void sinx(int N, int terms, float* x, float* result)
{
    // declare independent loop iterations
    forall (int i from 0 to N-1)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

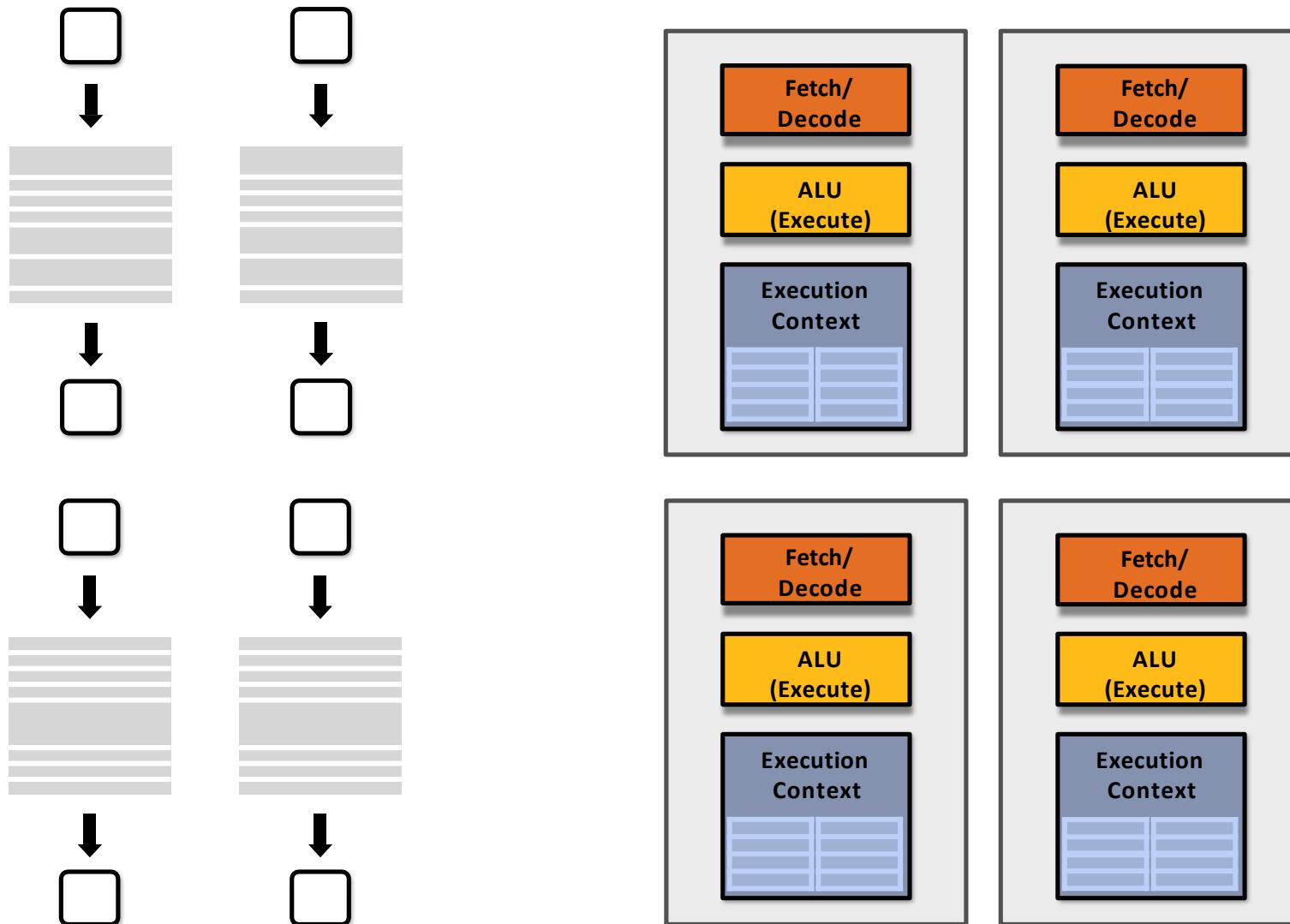
        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

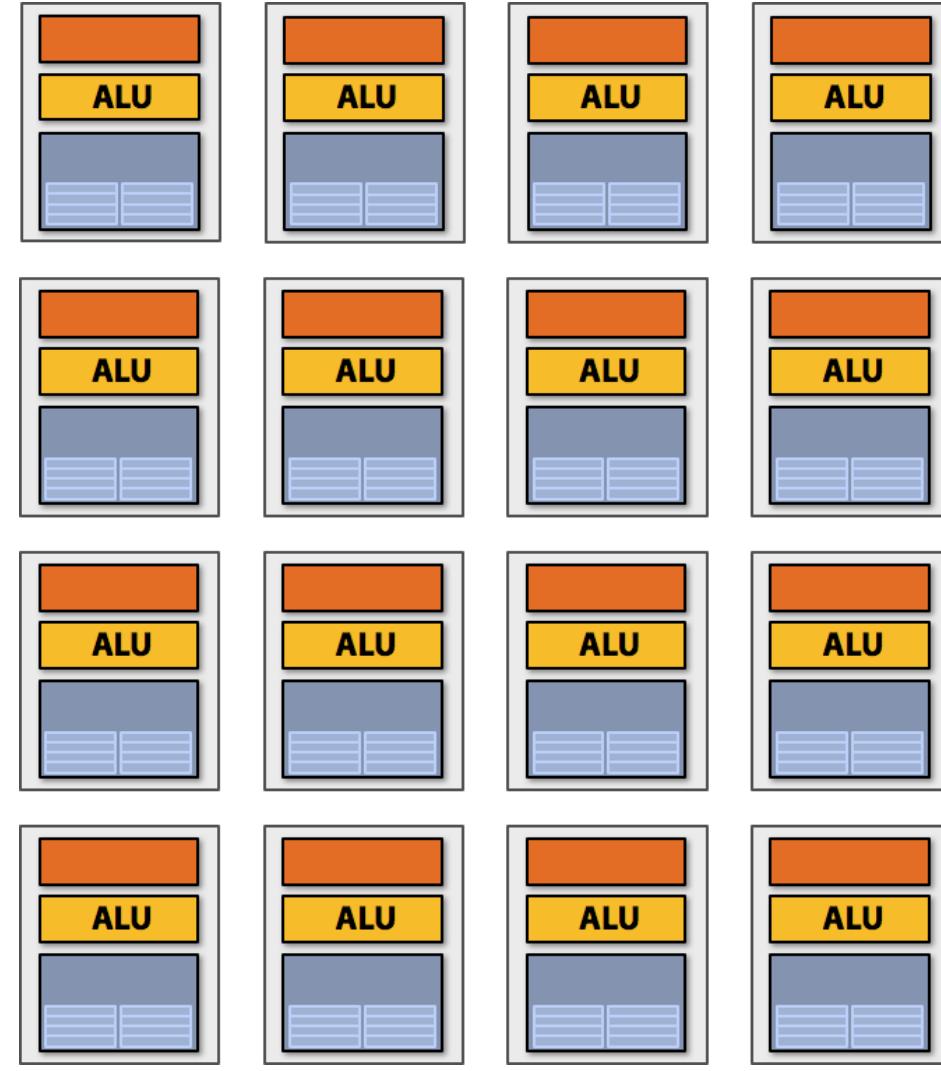
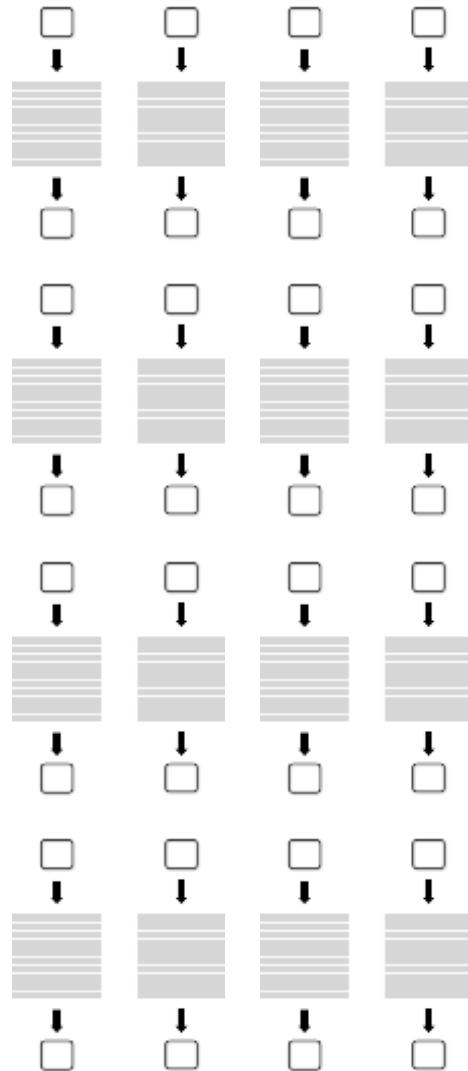
Loop iterations declared by the programmer to be independent

With this information, you could imagine how a compiler might automatically generate parallel threaded code

Four cores: compute four elements in parallel



Sixteen cores: compute sixteen elements in parallel



Sixteen cores, sixteen simultaneous instruction streams

Data-parallel expression

(in our fictitious data-parallel language)

```
void sinx(int N, int terms, float* x, float* result)
{
    // declare independent loop iterations
    forall (int i from 0 to N-1)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

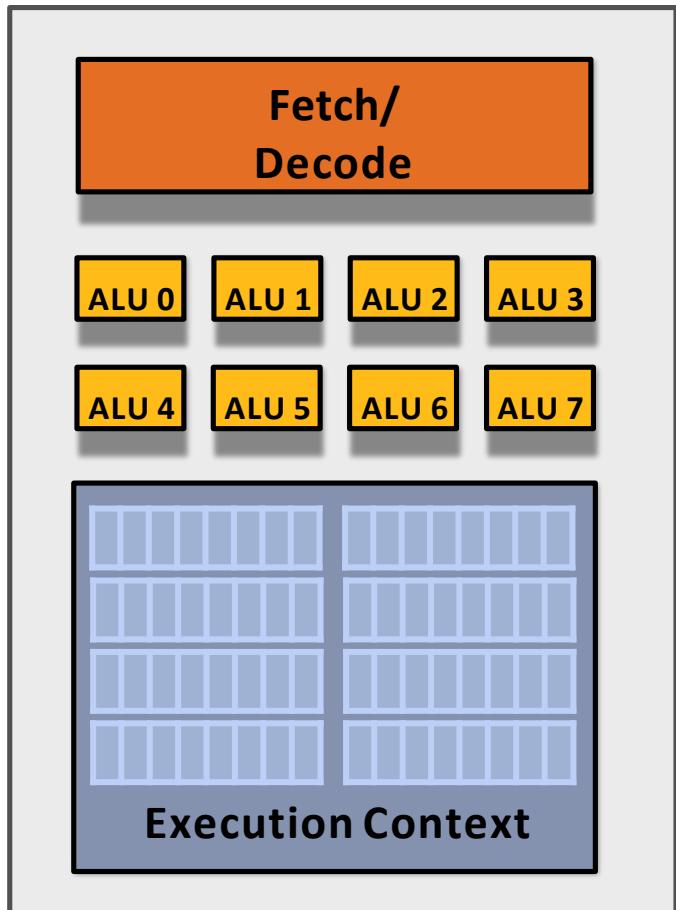
        result[i] = value;
    }
}
```

Another interesting property of this code:

Parallelism is across iterations of the loop.

All the iterations of the loop do the same thing: evaluate the sine of a single input number

Add ALUs to increase compute capability



Idea #2:

Amortize cost/complexity of managing an instruction stream across many ALUs

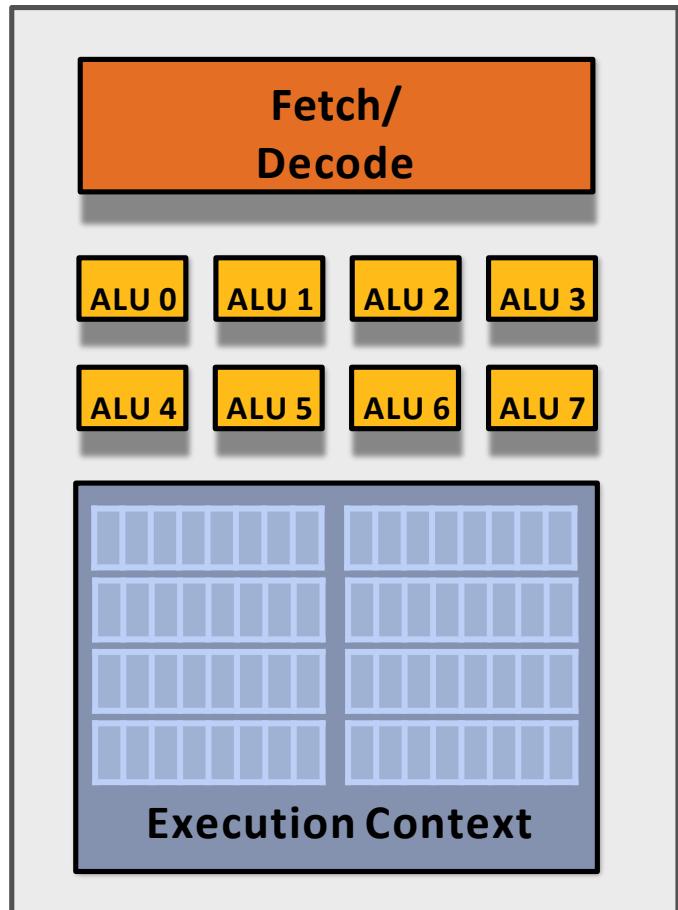
SIMD processing

Single instruction, multiple data

Same instruction broadcast to all ALUs

Executed in parallel on all ALUs

Add ALUs to increase compute capability



```
ld  r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
...
...
...
...
...
...
...
...
st  addr[r2], r0
```

Recall original compiled program:
Instruction stream processes one array element
at a time using scalar instructions on scalar
registers (e.g., 32-bit floats)

Scalar program

```
void sinx(int N, int terms, float* x, float* result)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

Original compiled program:

Processes one array element using scalar instructions on scalar registers (e.g., 32-bit floats)

ld	r0, addr[r1]
mul	r1, r0, r0
mul	r1, r1, r0
...	...
...	...
...	...
...	...
...	...
...	...
st	addr[r2], r0

Vector program (using AVX intrinsics)

```
#include <immintrin.h>

void sinx(int N, int terms, float* x, float* result)
{
    float three_fact =
        6;          //      3!
    for (int i=0; i<N;
         i+=8)

    {
        __m256 origx = _mm256_load_ps(&x[i]);  __m256 value =
        origx;
        __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));  __m256 denom =
        _mm256_broadcast_ss(&three_fact);
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            // value += sign * numer / denom
            __m256 tmp = _mm256_div_ps(_mm256_mul_ps(_mm256_set1ps(sign), numer), denom);  value =
            _mm256_add_ps(value, tmp);

            numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
            denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) * (2*j+3)));  sign *= -1;
        }
        _mm256_store_ps(&result[i], value);
    }
}
```

Intrinsics available to C programmers

Vector program (using AVX intrinsics)

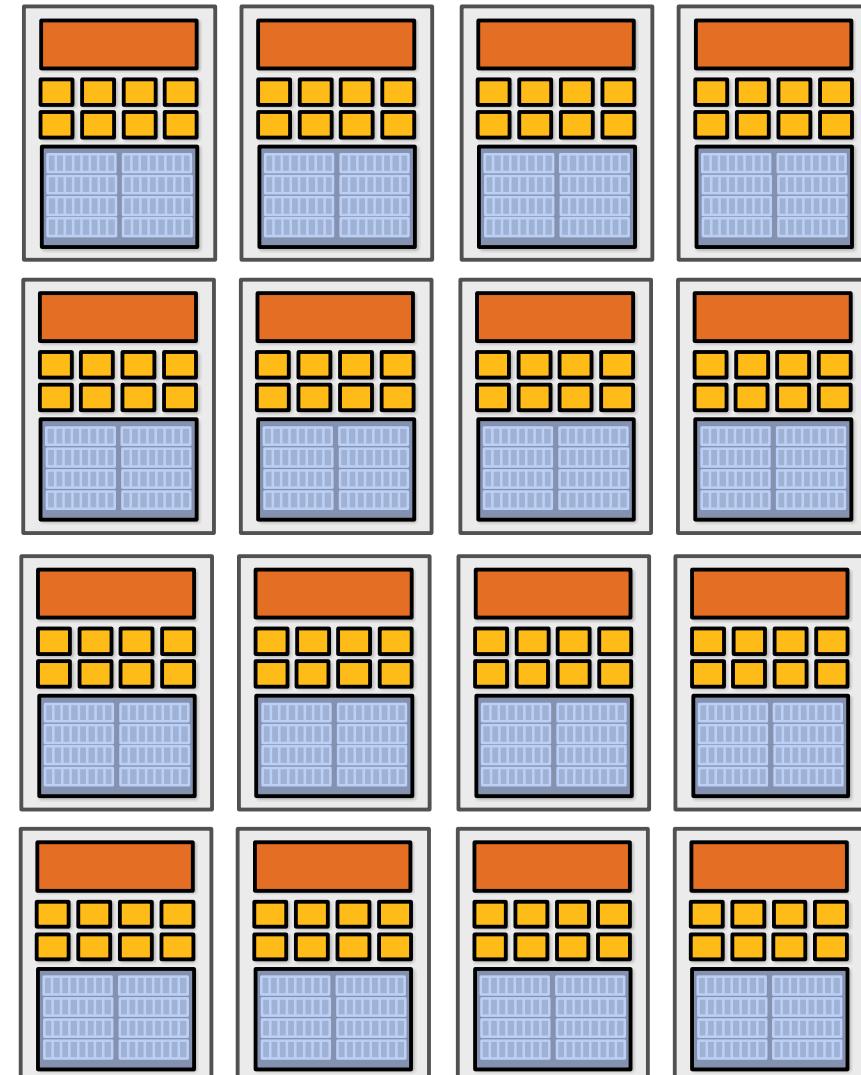
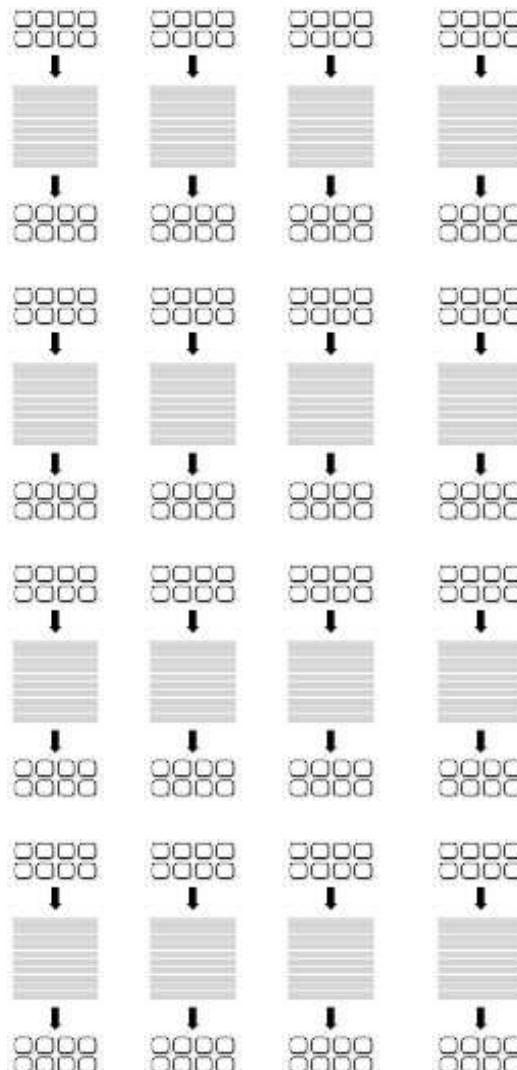
```
#include <immintrin.h>
void sinx(int N, int terms, float* x, float*
sinx)
{ float three_fact = 6; // 3!
  for (int i=0; i<N; i+=8)
  {
    __m256 origx = _mm256_load_ps(&x[i]);
    __m256 value = origx;
    __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx,
origx));
    __m256 denom =
    _mm256_broadcast_ss(&three_fact);  int sign = -
1;
    for (int j=1; j<=terms; j++)
    {
      // value += sign * numer / denom
      __m256 tmp =
      _mm256_div_ps(_mm256_mul_ps(_mm256_broadcast_ss(sign),numer),denom);  value =
      _mm256_add_ps(value, tmp);
      numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
      denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) *
(2*j+3)));  sign *= -1;
    }
    _mm256_store_ps(&sinx[i], value);
  }
}
```

vloadps	xmm0,
addr[r1]	
vmulps	xmm1,
xmm0, xmm0	
...	
...	
...	
...	
...	
...	
vstoreps	xmm2, xmm0

Compiled program:

Processes eight array elements simultaneously using vector instructions on 256-bit vector registers

16 SIMDcores: 128 elements in parallel



16 cores, 128 ALUs, 16 simultaneous instruction streams

Data-parallel expression

(in our fictitious data-parallel language)

```
void sinx(int N, int terms, float* x, float* result)
{
    // declare independent loop iterations
    forall (int i from 0 to N-1)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

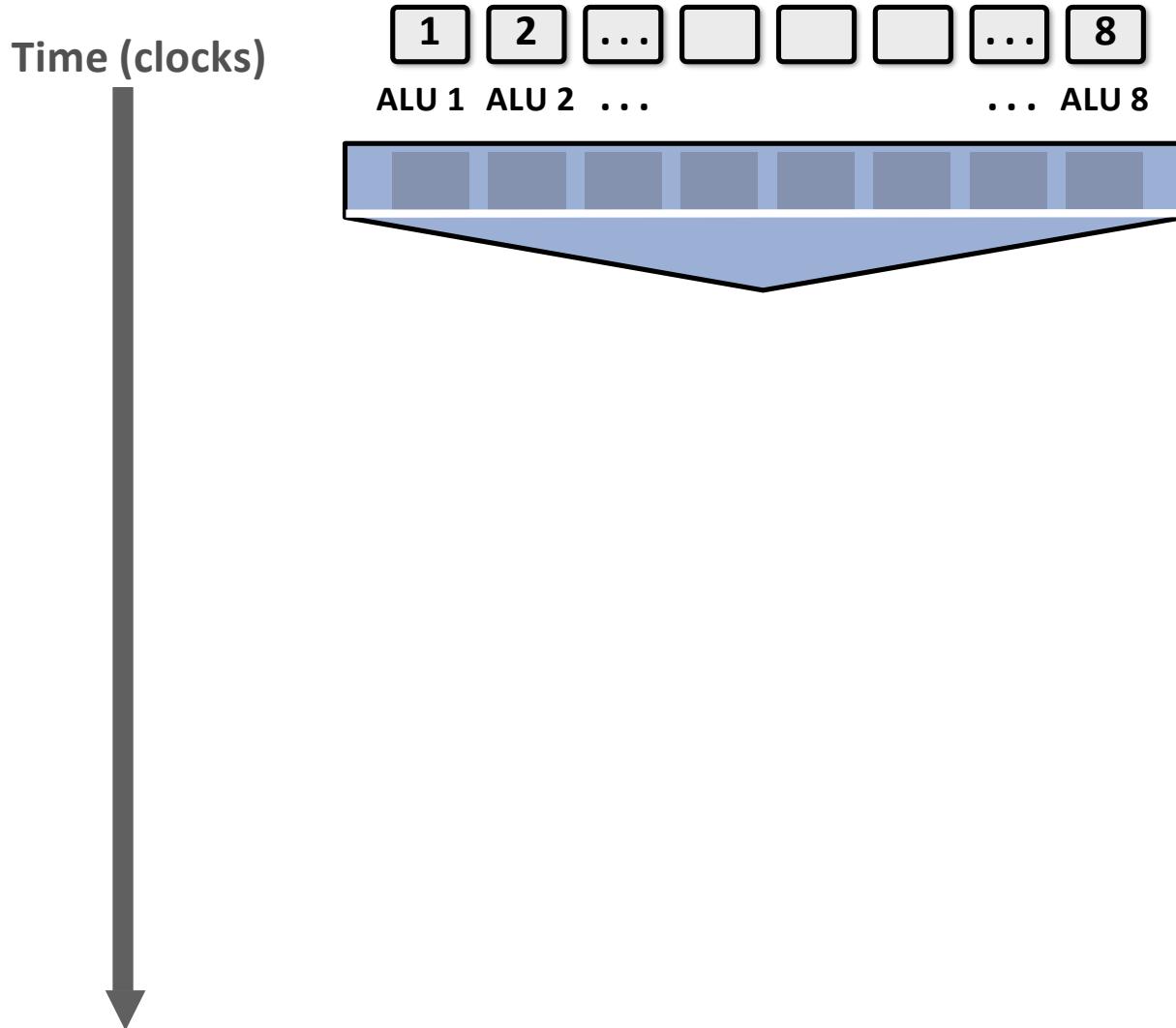
        for (int j=1; j<=terms; j++)
        {
            value += sign * numer /
                denom;
            numer *= x[i] * x[i];
            sign *= -1;
            denom *= (2*j+2) * (2*j+3);
        }

        result[i] = value;
    }
}
```

Compiler understands loop iterations are independent, and that same loop body will be executed on a large number of data elements.

Abstraction facilitates automatic generation of both multi-core parallel code, and vector instructions to make use of SIMD processing capabilities within a core.

What about conditional execution?



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

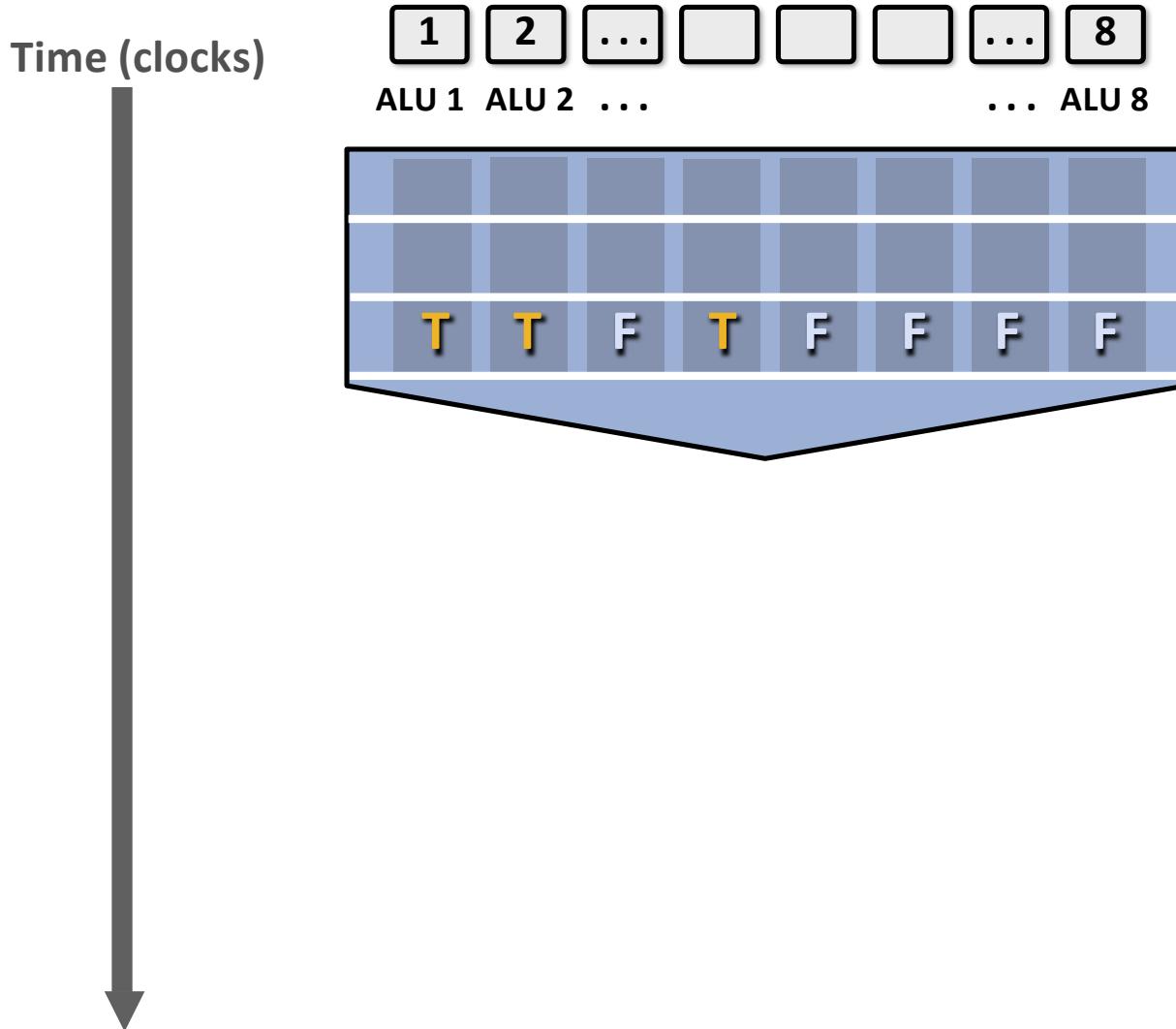
<unconditional code>

```
float x = A[i];  
  
if (x > 0) {  
    float tmp = exp(x,5.f);  
    tmp *= kMyConst1;  
    x = tmp + kMyConst2;  
} else {  
    float tmp = kMyConst1;  
    x = 2.f * tmp;  
}
```

<resume unconditional code>

```
result[i] = x;
```

What about conditional execution?



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

<unconditional code>

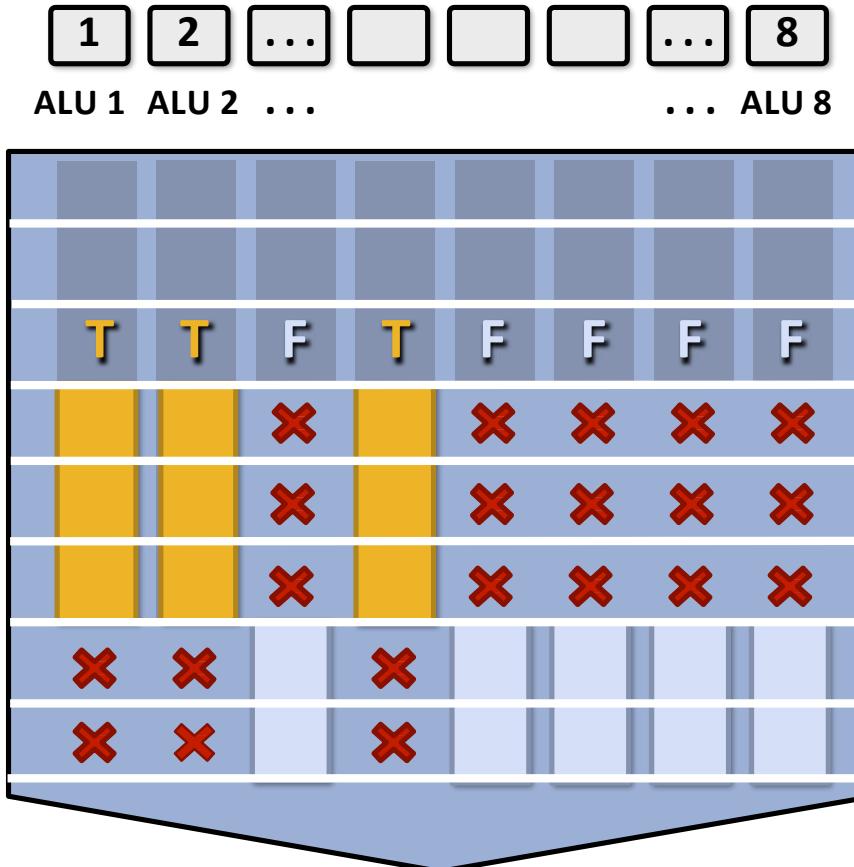
```
float x = A[i];  
  
if (x > 0) {  
  
    float tmp = exp(x,5.f);  
  
    tmp *= kMyConst1;  
  
    x = tmp + kMyConst2;  
} else {  
    float tmp = kMyConst1;  
  
    x = 2.f * tmp;  
}
```

<resume unconditional code>

```
result[i] = x;
```

Mask (discard) output of ALU

Time (clocks)



Not all ALUs do useful work!

Worst case: 1/8 peak performance

(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

<unconditional code>

```
float x = A[i];
```

```
if (x > 0) {
```

```
float tmp = exp(x,5.f);
```

```
tmp *= kMyConst1;
```

```
x = tmp + kMyConst2;
```

```
} else {
```

```
float tmp = kMyConst1;
```

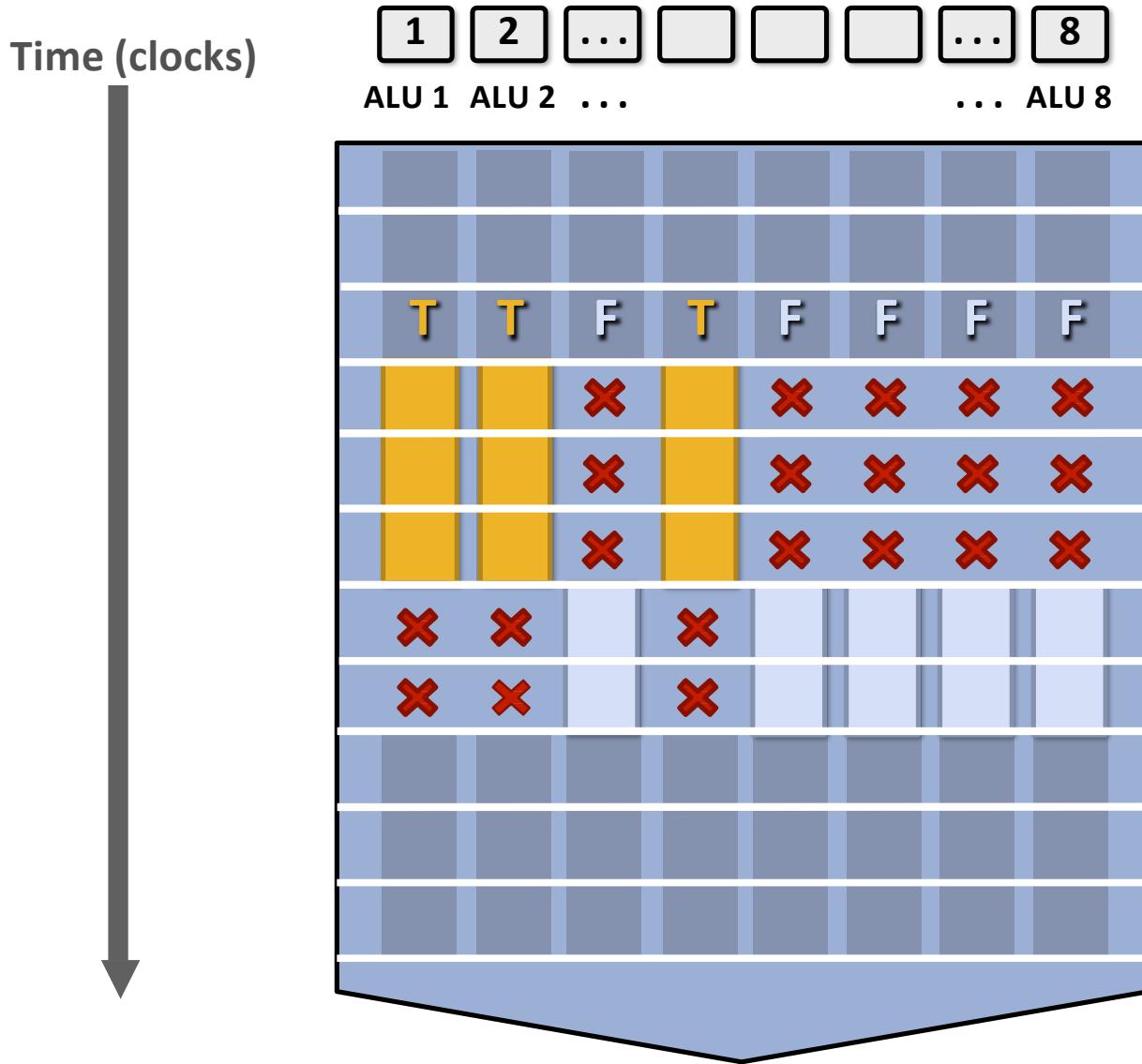
```
x = 2.f * tmp;
```

104

<resume unconditional code>

```
result[i] = x;
```

After branch: continue at full performance



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

<unconditional code>

```
float x = A[i];  
  
if (x > 0) {  
    float tmp = exp(x,5.f);  
    tmp *= kMyConst1;  
    x = tmp + kMyConst2;  
}  
else {  
    float tmp = kMyConst1;  
    x = 2.f * tmp;  
}
```

<resume unconditional code>

```
result[i] = x;
```

Terminology

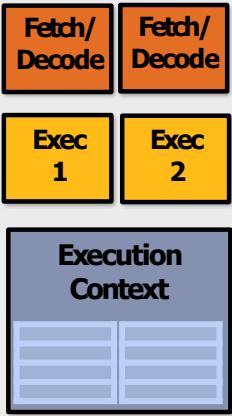
- **Instruction stream coherence (“coherent execution”)**
 - Same instruction sequence applies to all elements operated upon simultaneously
 - Coherent execution is necessary for efficient use of SIMD processing resources
 - Coherent execution IS NOT necessary for efficient parallelization across cores, since each core has the capability to fetch/decode a different instruction stream
- **“Divergent” execution**
 - Lack of instruction stream coherence

SIMD execution on modern CPUs

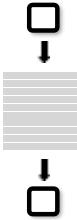
- SSE instructions: 128-bit operations: 4x32 bits or 2x64 bits (4-wide float vectors)
- AVX instructions: 256 bit operations: 8x32 bits or 4x64 bits (8-wide float vectors)
- Instructions are generated by the compiler
 - Parallelism explicitly requested by programmer using intrinsics
 - Parallelism conveyed using parallel language semantics (e.g., `forall` example)
 - Parallelism inferred by dependency analysis of loops (hard problem, even best compilers are not great on arbitrary C/C++ code)
- Terminology: “explicit SIMD”: SIMD parallelization is performed at compile time
 - Can inspect program binary and see instructions (`vstoreps`, `vmulps`, etc.)

SIMD execution on many modern GPUs

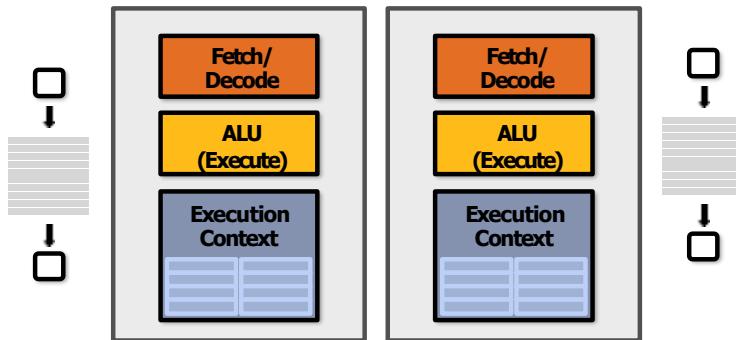
- “Implicit SIMD”
 - Compiler generates a scalar binary (scalar instructions)
 - But N instances of the program are *always run* together on the processor
`execute(my_function, N) // execute my_function N times`
 - In other words, the interface to the hardware itself is data-parallel
 - Hardware (not compiler) is responsible for simultaneously executing the same instruction from multiple instances on different data on SIMD ALUs
- SIMD width of most modern GPUs ranges from 8 to 32
 - Divergence can be a big issue
(poorly written code might execute at 1/32 the peak capability of the machine!)



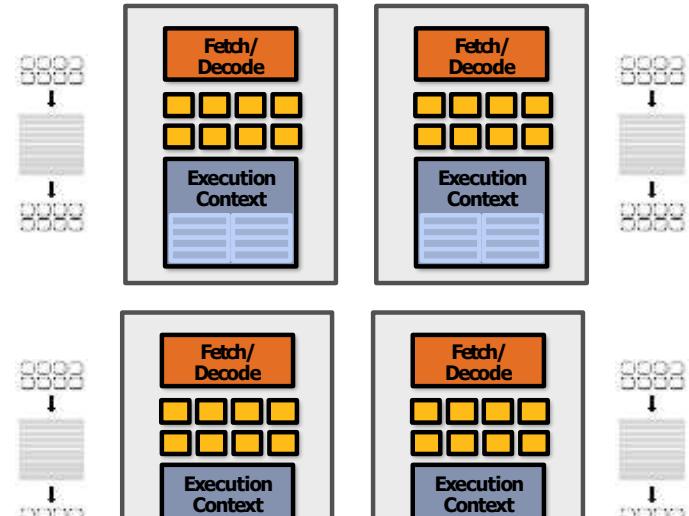
My single core, superscalar processor: executes up to two instructions per clock from a single instruction stream (if the instructions are independent)



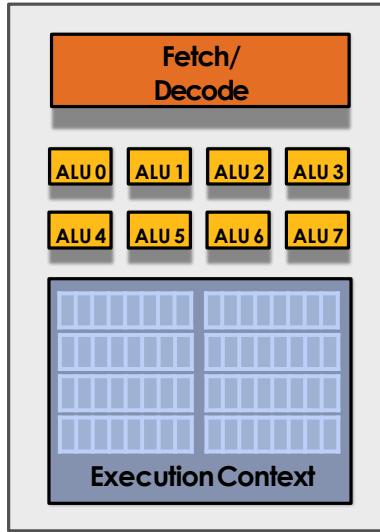
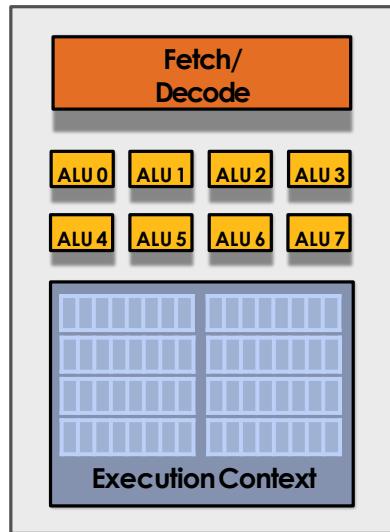
My dual-core processor: executes one instruction per clock from one instruction stream on each core.



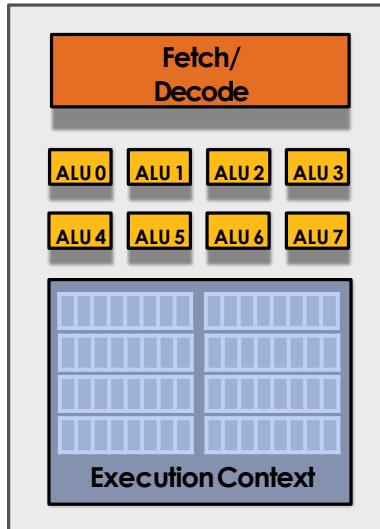
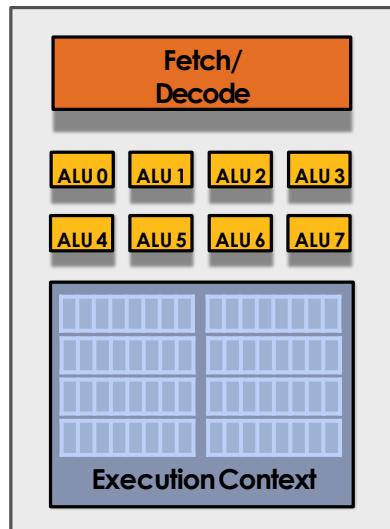
My SIMD quad-core processor: executes one 8-wide SIMD instruction per clock from one instruction stream on each core.



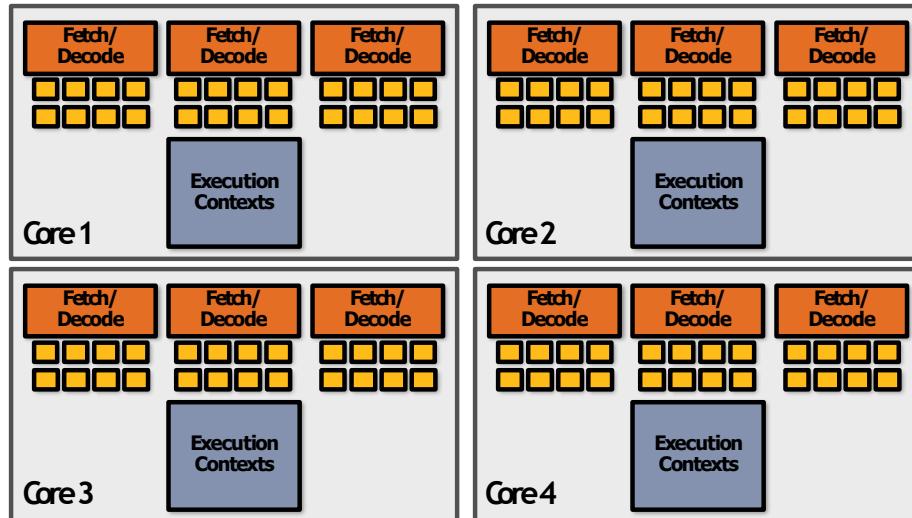
Example: Intel Core i7



4 cores
8 SIMD ALUs per core
(AVX instructions)



Example: four-core Intel i7-7700K CPU

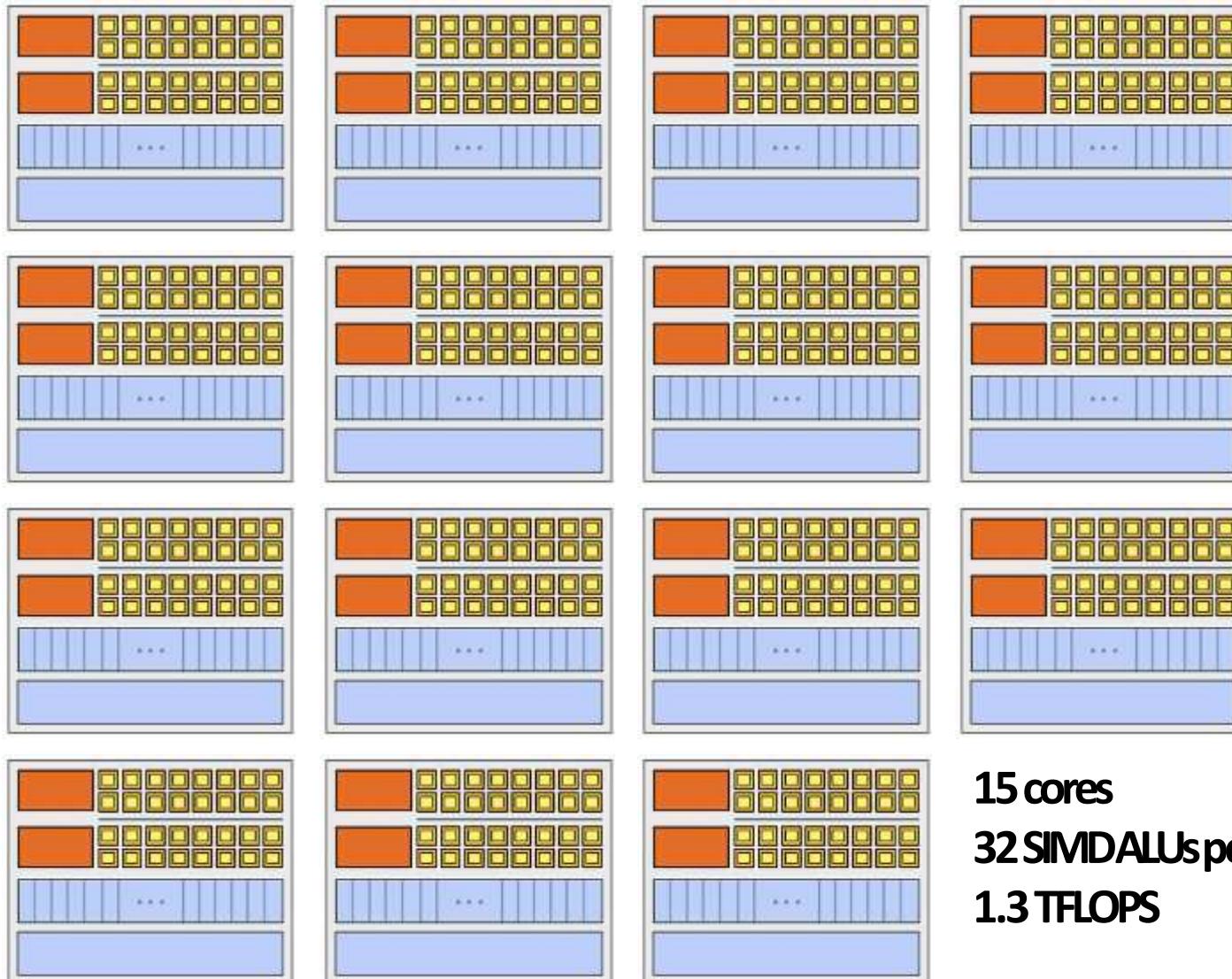


4 core processor
Three 8-wide SIMD ALUs per core (AVX2 instructions)

4cores x 8-wide SIMD x 3x 4.2GHz = 400 GFLOPs

* Showing only AVX math units, and fetch/decode unit for AVX (additional capability for integer math)

Example: NVIDIA GTX 480



15 cores
32 SIMD ALUs per core
1.3 TFLOPS

Summary: parallel execution

- **Several forms of parallel execution in modern processors**
 - Multi-core: use multiple processing cores
 - Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core
 - Software decides when to create threads (e.g., via pthreads API)
 - SIMD: use multiple ALUs controlled by same instruction stream (within a core)
 - Efficient design for data-parallel workloads: control amortized over many ALUs
 - Vectorization can be done by compiler (explicit SIMD) or at runtime by hardware
 - [Lack of] dependencies is known prior to execution (usually declared by programmer, but can be inferred by loop analysis by advanced compiler)
 - Superscalar: exploit ILP within an instruction stream. Process different instructions from the same instruction stream in parallel (within a core)
 - Parallelism automatically and dynamically discovered by the hardware during execution (not programmer visible)

Questions

CSCI465/ECEN433

Introduction to Parallel Computing

Spring 2024



Lecture (2)

A Modern Multi-Core Processor

(Forms of parallelism + understanding latency and bandwidth)



Part 2: accessing memory

Terminology

- **Memory latency**
 - The amount of time for a memory request (e.g., load, store) from a processor to be serviced by the memory system
 - Example: 100 cycles, 100 nsec
- **Memory bandwidth**
 - The rate at which the memory system can provide data to a processor
 - Example: 20 GB/s

Stalls

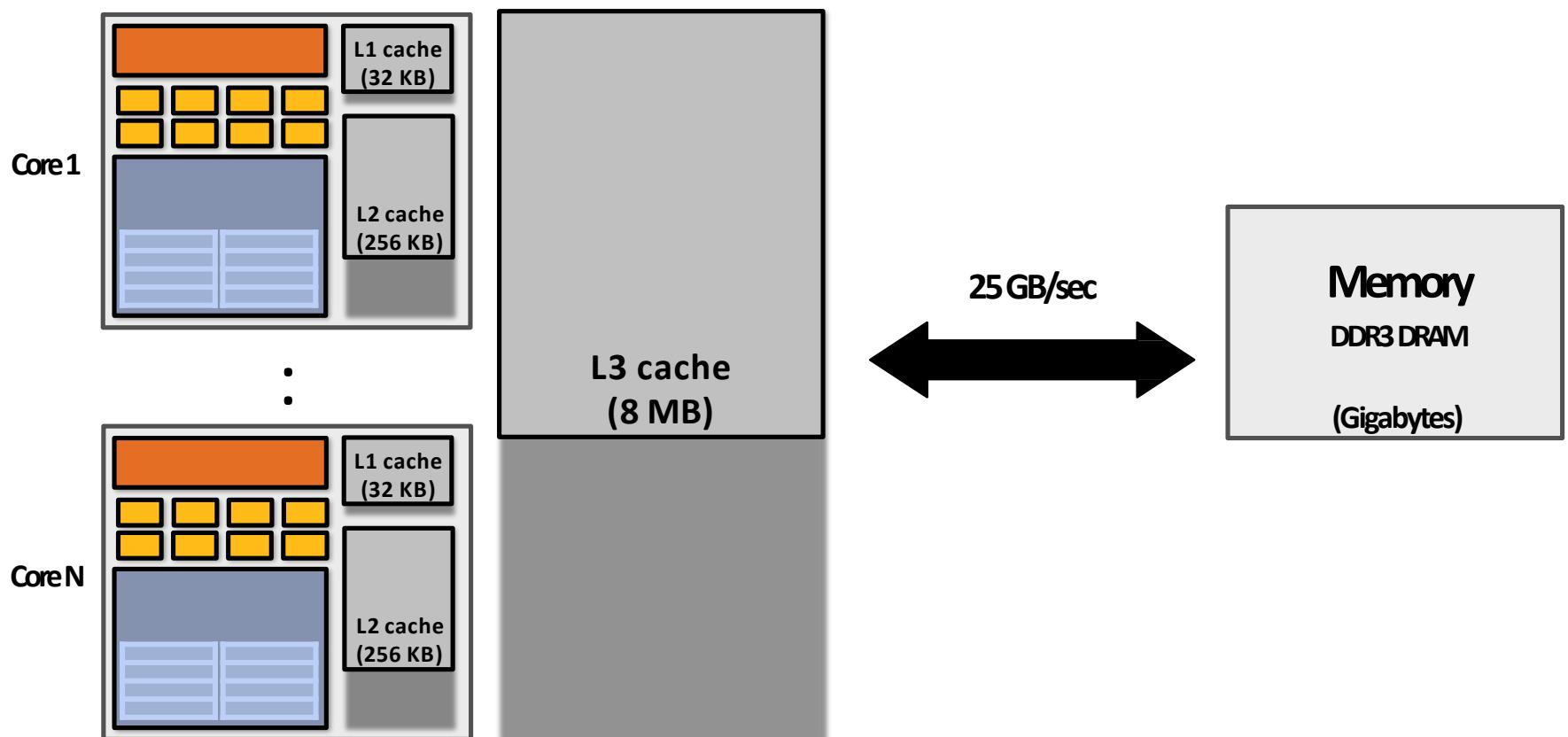
- A processor “stalls” when it cannot run the next instruction in an instruction stream because of a dependency on a previous instruction.
- Accessing memory is a major source of stalls

```
1d r0 mem[r2] ←  
1d r1 mem[r3] ←  
add r0, r0, r1
```

Dependency: cannot execute ‘add’ instruction until data at mem[r2] and mem[r3] have been loaded from memory

- Memory access times ~ 100's of cycles
 - Memory “access time” is a measure of latency

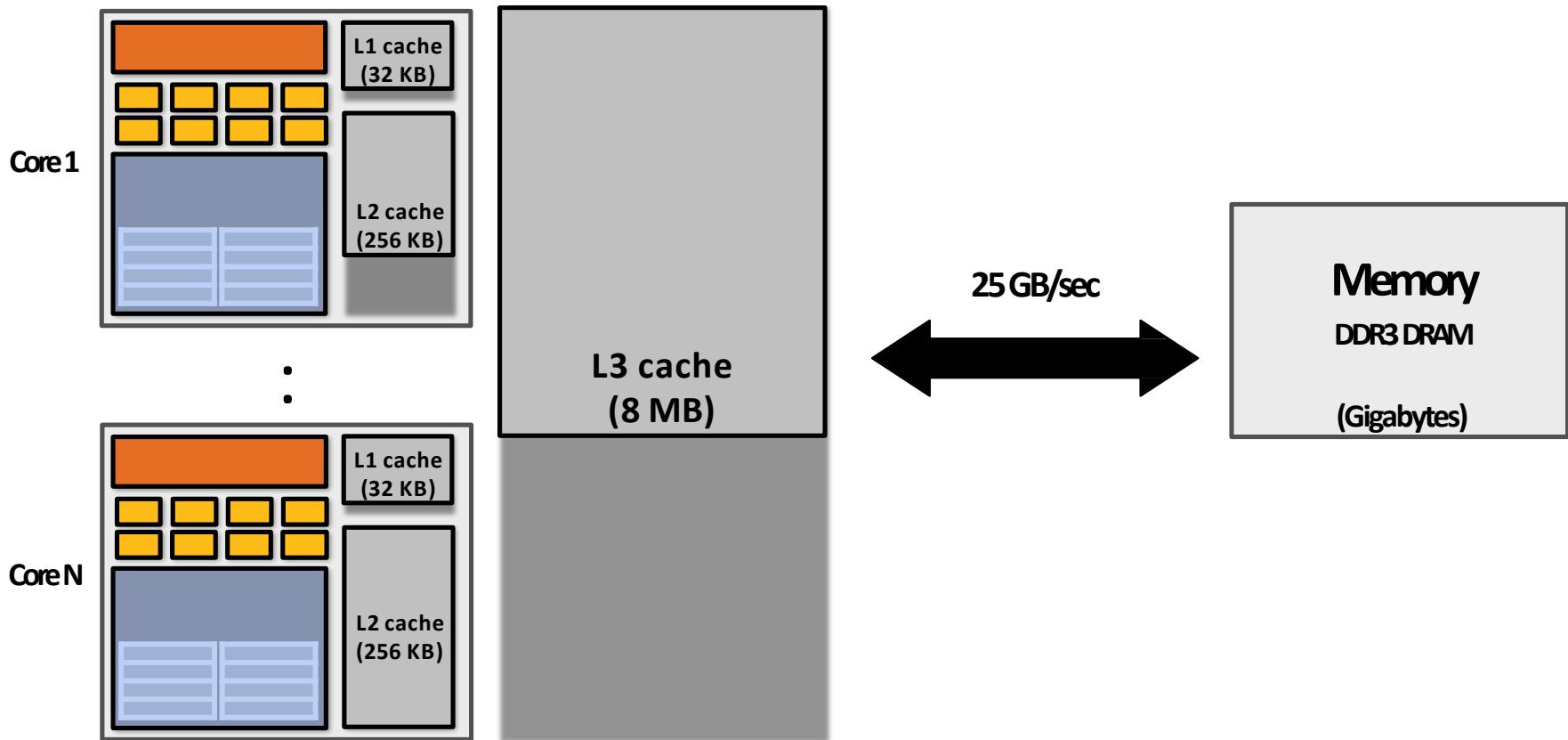
Review: why do modern processors have caches?



Caches reduce length of stalls (reduce latency)

Processors run efficiently when data is resident in caches

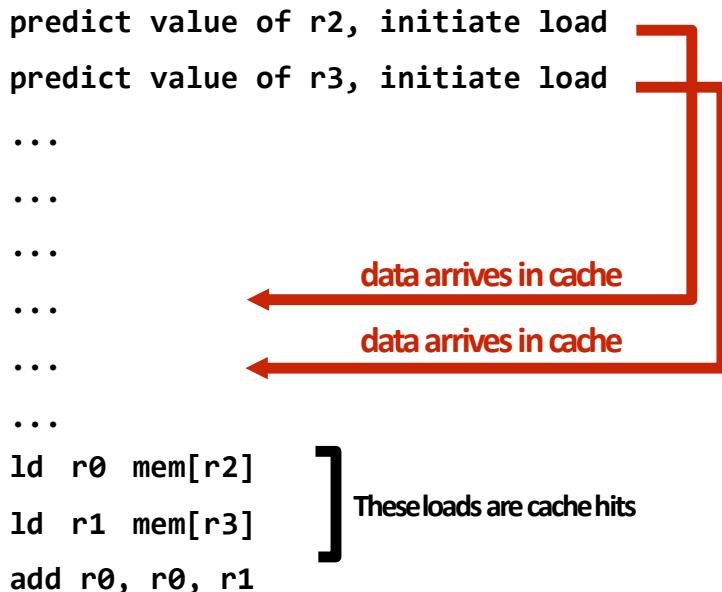
Caches reduce memory access latency *



* Caches also provide high bandwidth data transfer to CPU

Prefetching reduces stalls (hides latency)

- All modern CPUs have logic for prefetching data into caches
 - Dynamically analyze program's access patterns, predict what it will access soon
- Reduces stalls since data is resident in cache when accessed



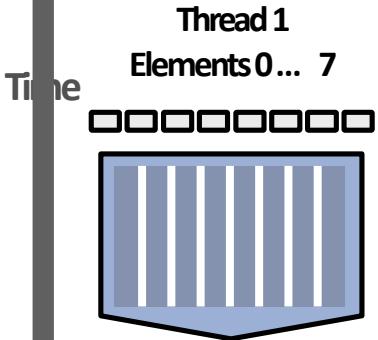
Note: Prefetching can also reduce performance if the guess is wrong (hogs bandwidth, pollutes caches)

(more detail later in course)

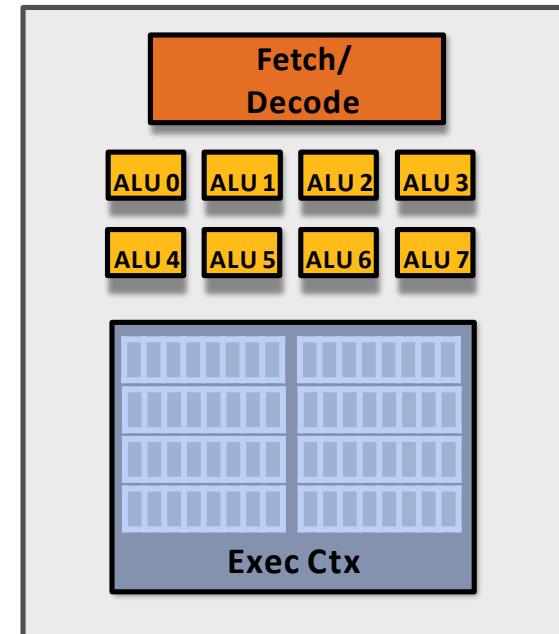
Multi-threading reduces stalls

- Idea: interleave processing of multiple threads on the same core to hide stalls
- Like prefetching, multi-threading is a latency hiding, not a latency reducing technique

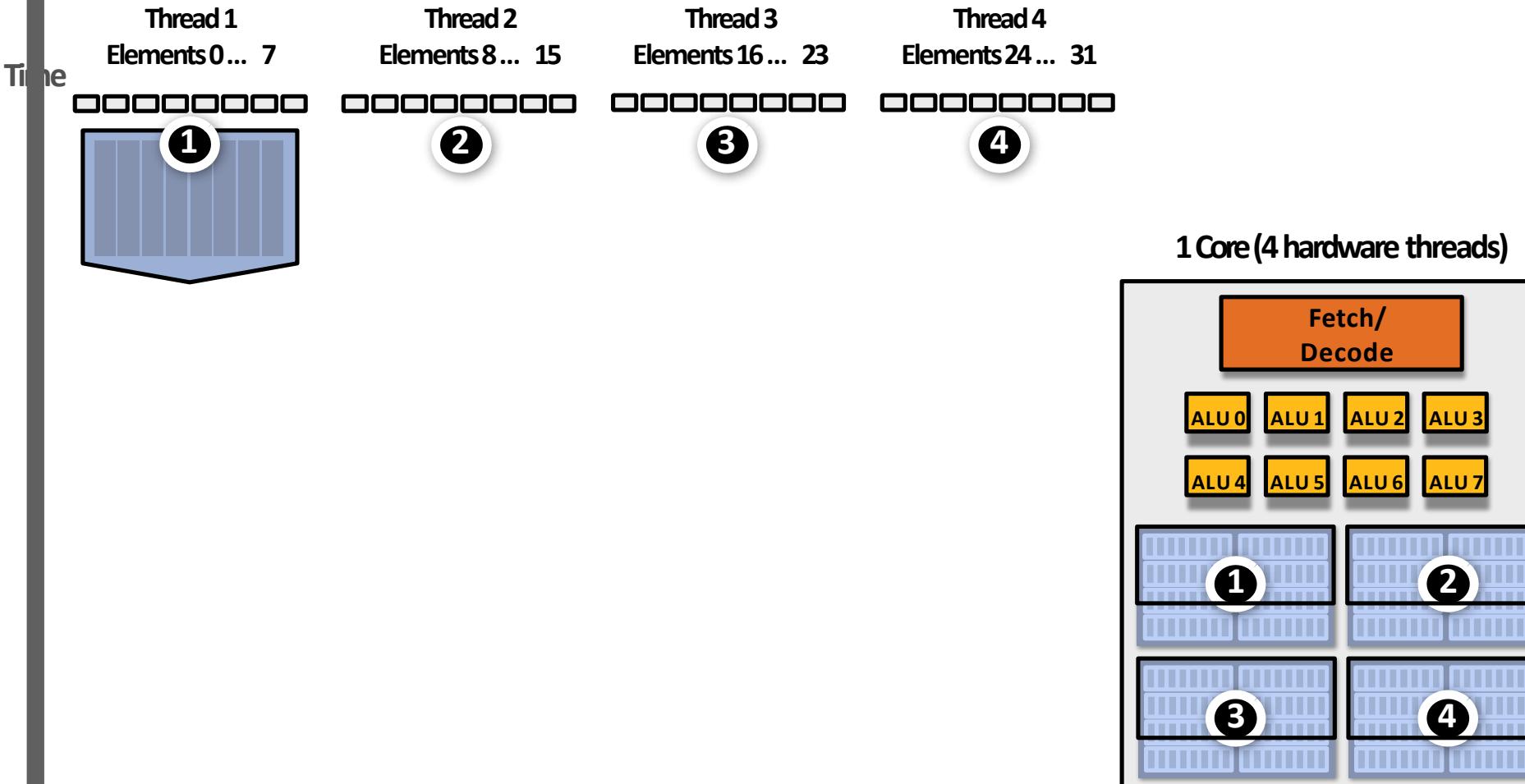
Hiding stalls with multi-threading



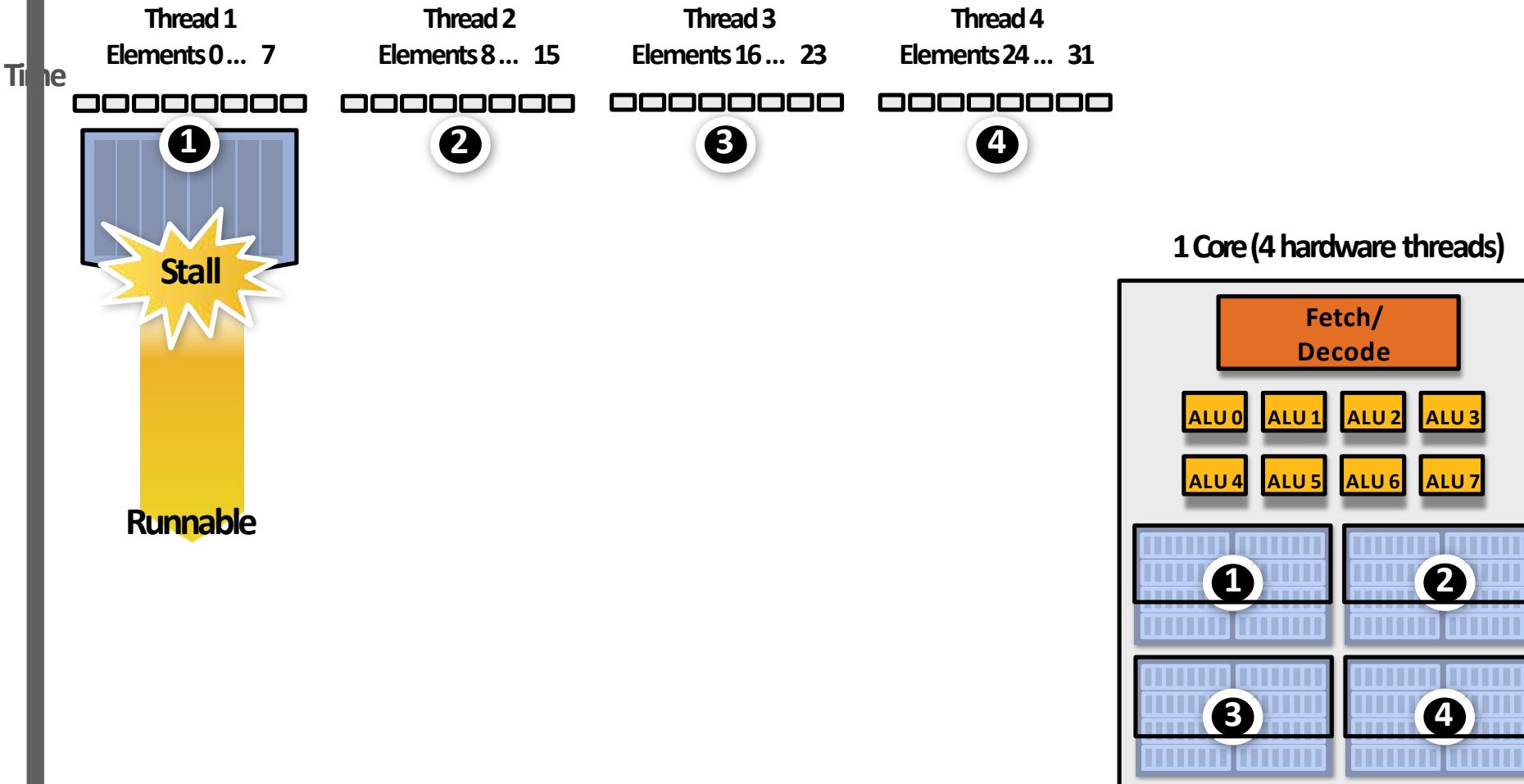
1 Core (1 thread)



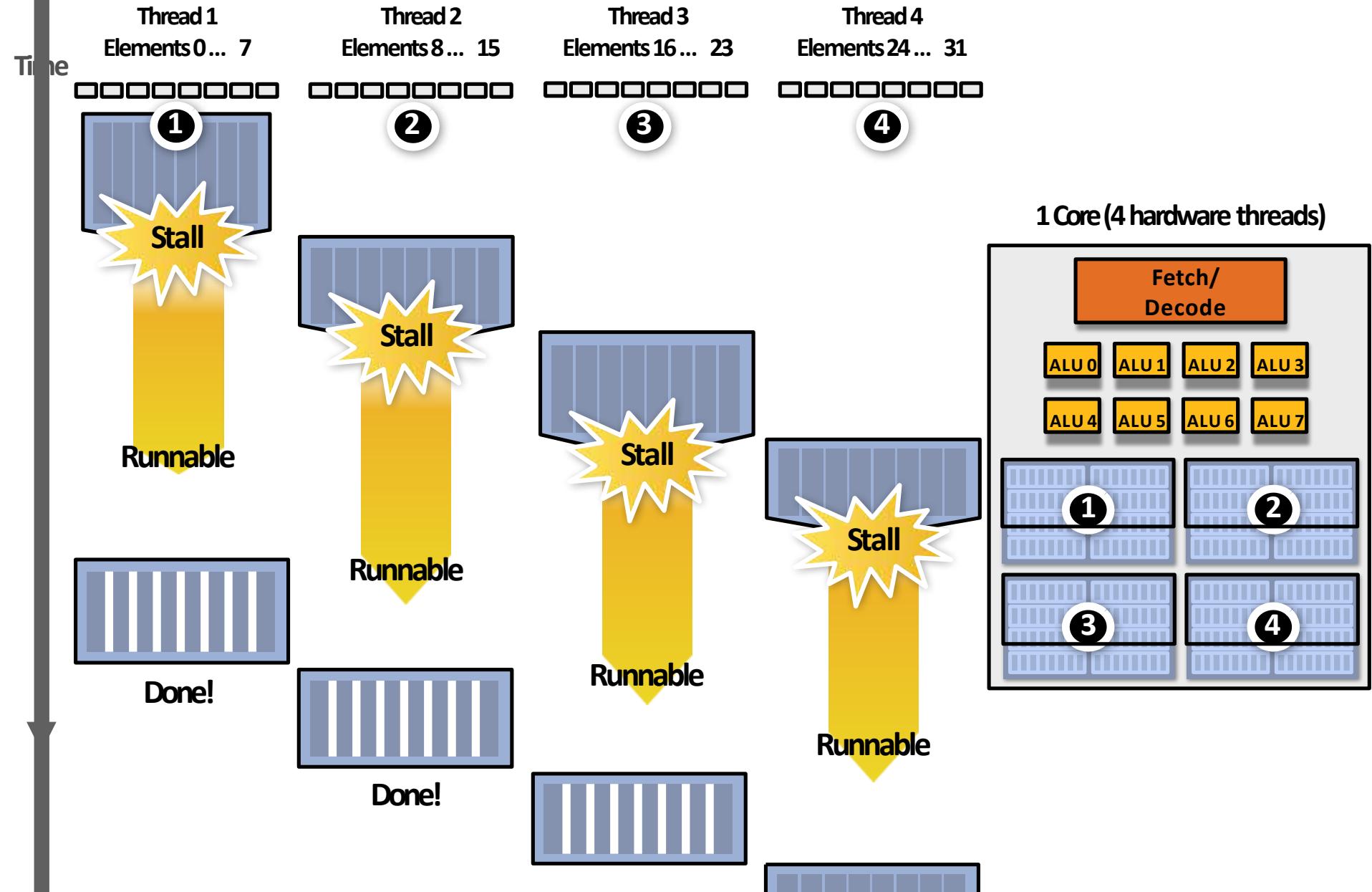
Hiding stalls with multi-threading



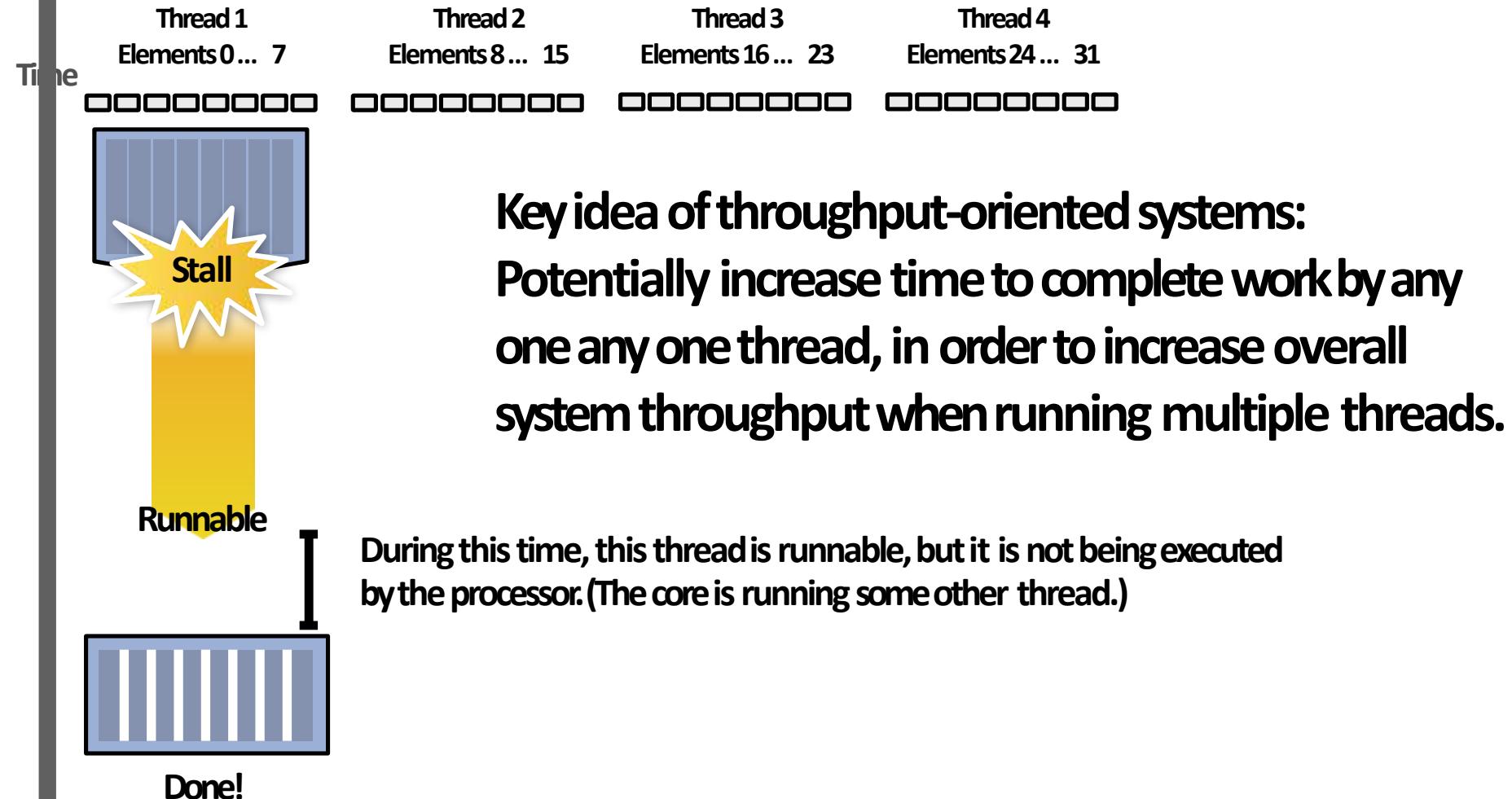
Hiding stalls with multi-threading



Hiding stalls with multi-threading

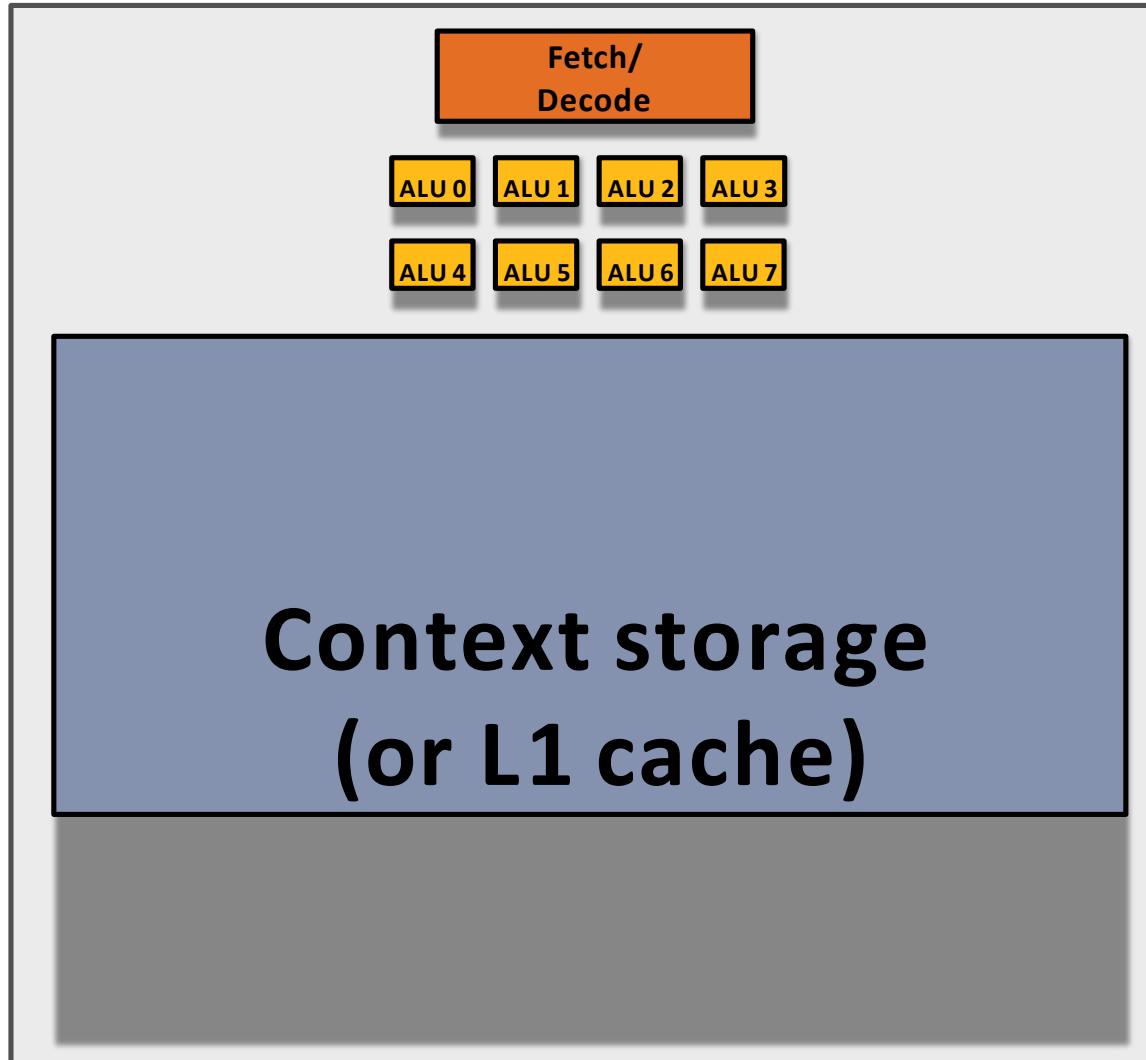


Throughput computing trade-off



Storing execution contexts

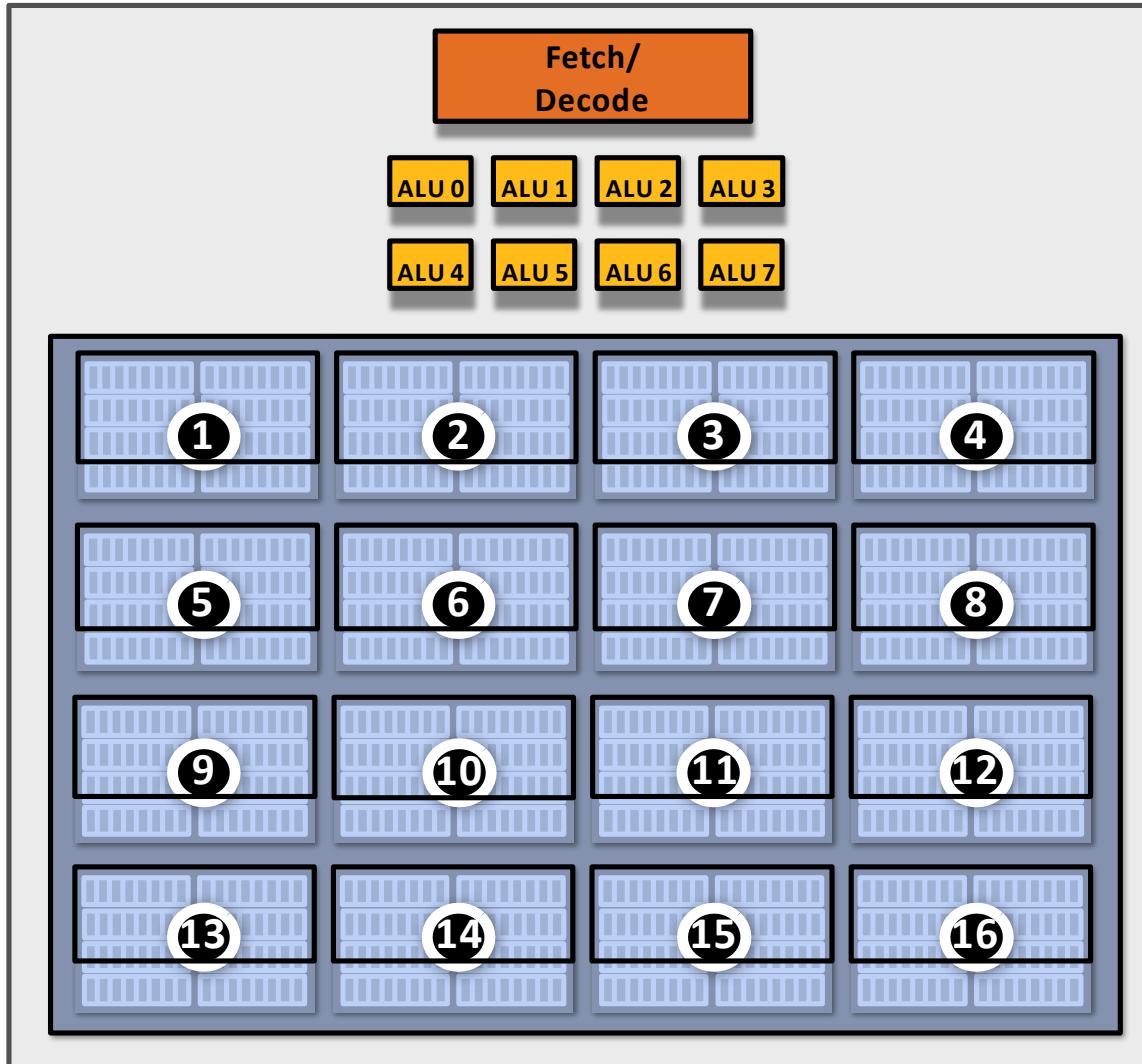
Consider on chip storage of execution contexts a finite resource.



Many small contexts (high latency hiding ability)

1 core

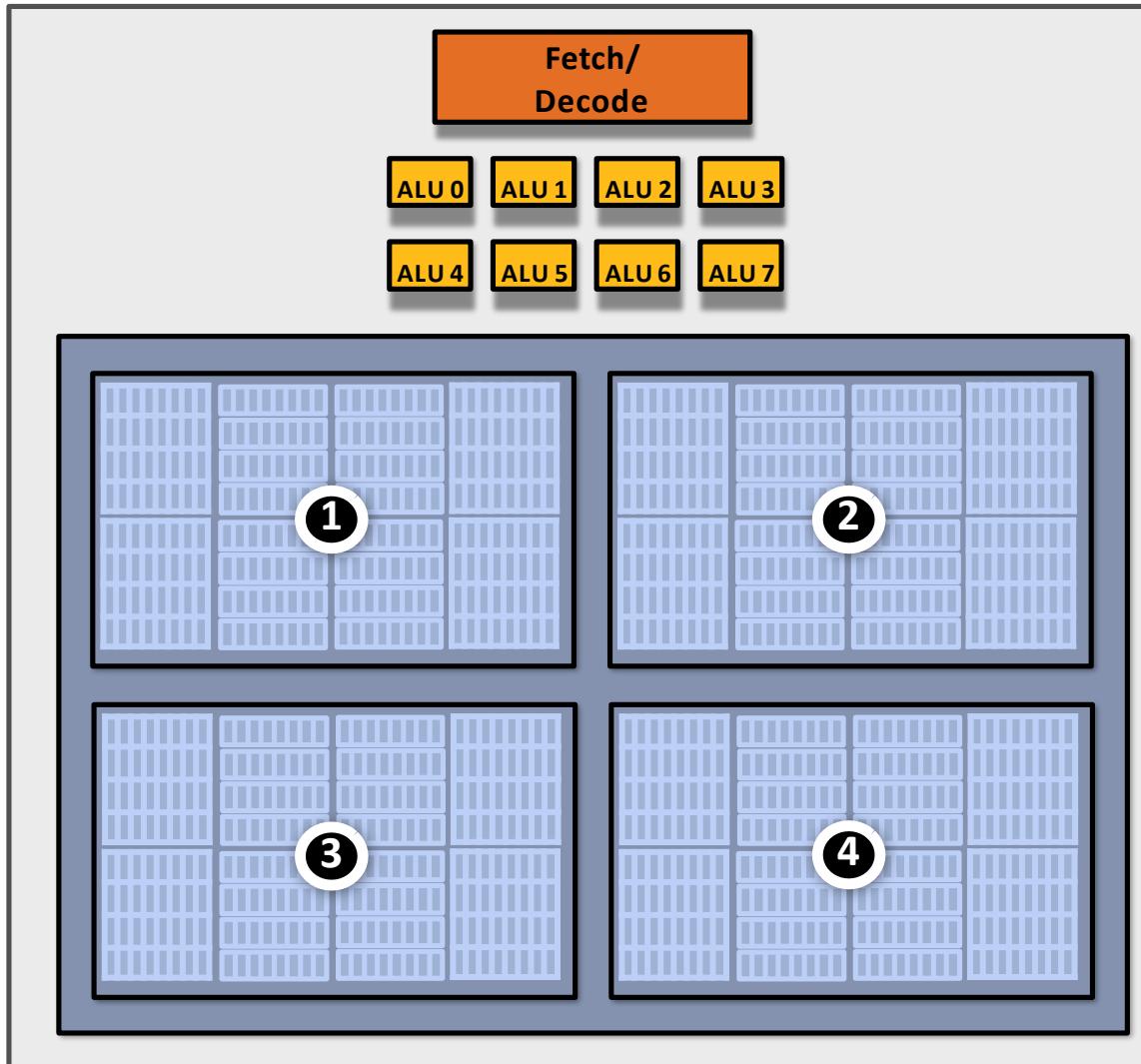
(16 hardware threads, storage for small working set per thread)



Four large contexts (low latency hiding ability)

1 core

(4 hardware threads, storage for larger working set per thread)



Hardware-supported multi-threading

- **Core manages execution contexts for multiple threads**
 - Runs instructions from runnable threads (processor makes decision about which thread to run each clock, not the operating system)
 - Core still has the same number of ALU resources: multi-threading only helps use them more efficiently in the face of high-latency operations like memory access
- **Interleaved multi-threading (a.k.a. temporal multi-threading)**
 - What I described on the previous slides: each clock, the core chooses a thread, and runs an instruction from the thread on the ALUs
- **Simultaneous multi-threading (SMT)**
 - Each clock, core chooses instructions from multiple threads to run on ALUs
 - Extension of superscalar CPU design
 - Example: Intel Hyper-threading (2 threads per core)

Multi-threading summary

- **Benefit: use a core's ALU resources more efficiently**
 - Hide memory latency
 - Fill multiple functional units of superscalar architecture
(when one thread has insufficient ILP)
- **Costs**
 - Requires additional storage for thread contexts
 - Increases run time of any single thread
(often not a problem, we usually care about throughput in parallel apps)
 - Requires additional independent work in a program (more independent work than ALUs!)
 - Relies heavily on memory bandwidth
 - More threads → larger working set → less cache space per thread
 - May go to memory more often, but can hide the latency

Our fictitious multi-core chip

16 cores

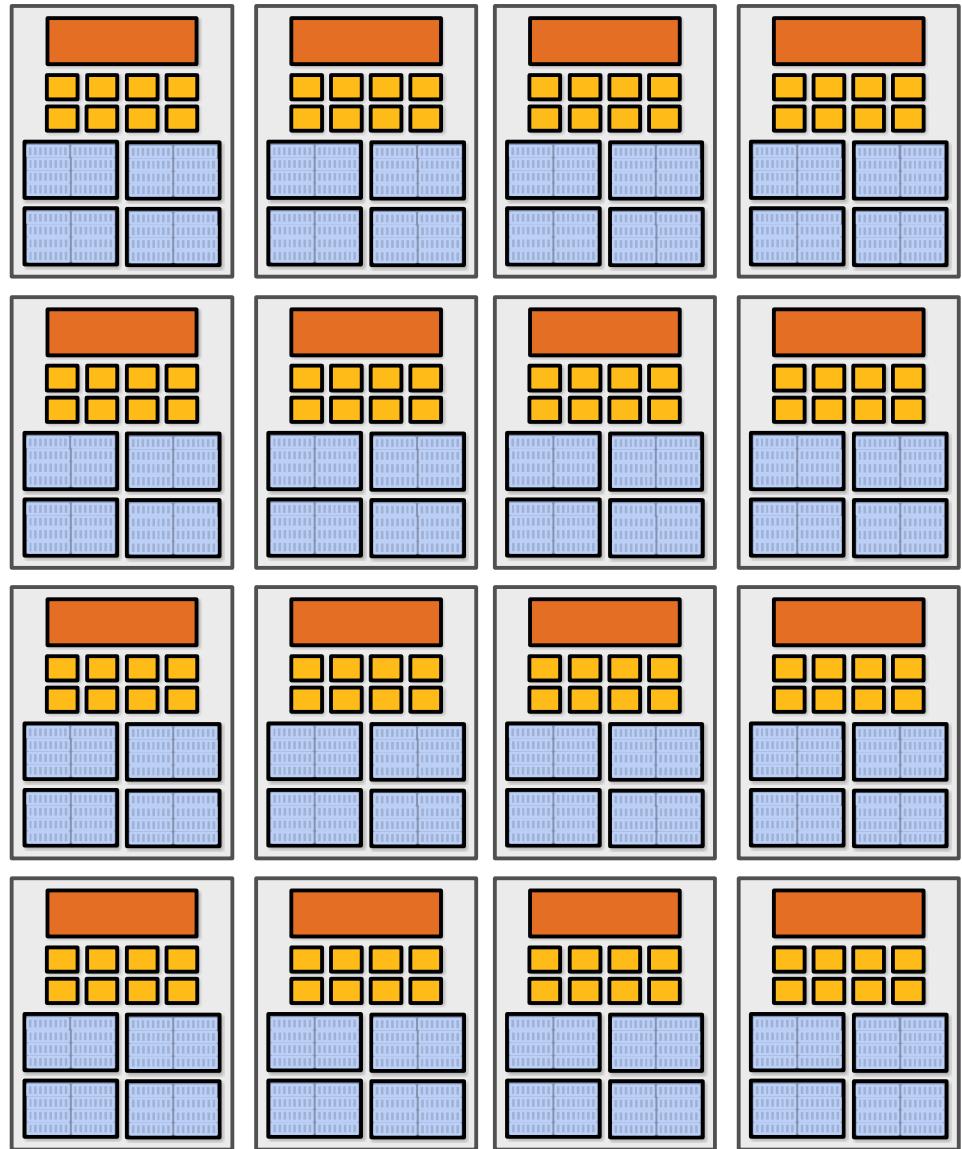
8 SIMD ALUs per core
(128 total)

4 threads per core

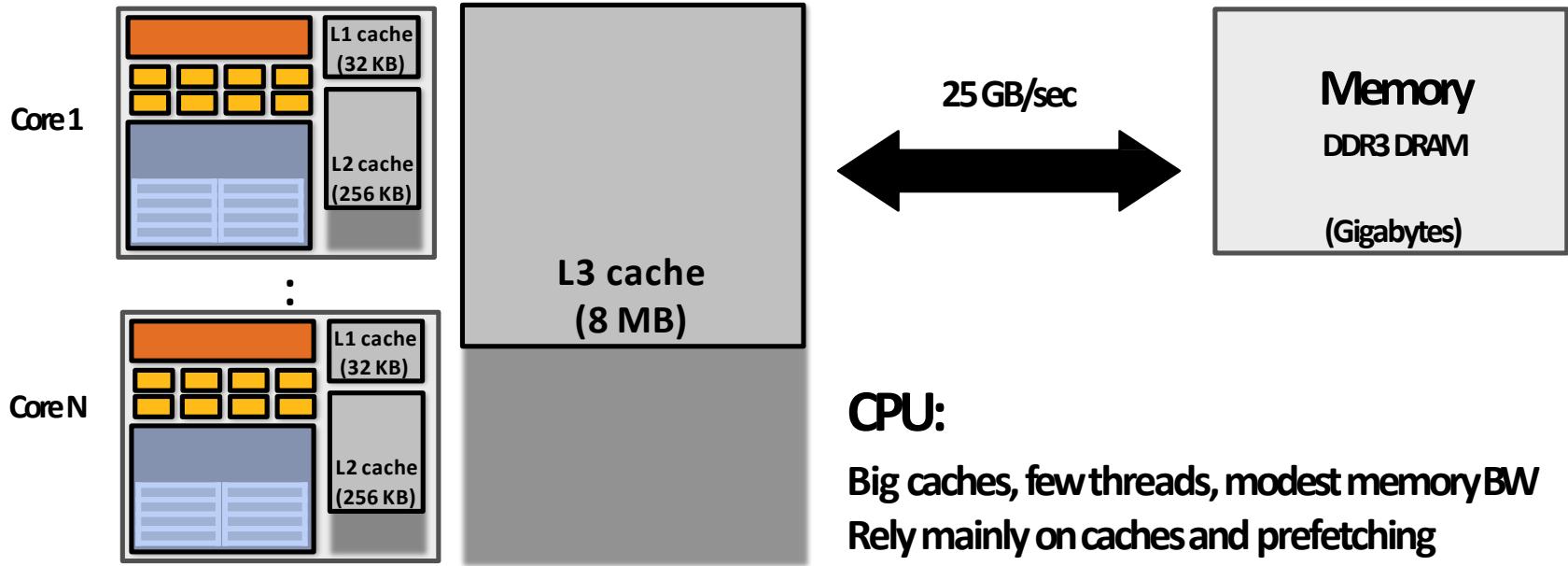
16 simultaneous
instruction streams

64 total concurrent
instruction streams

512 independent pieces of
work are needed to run chip
with maximal latency
hiding ability

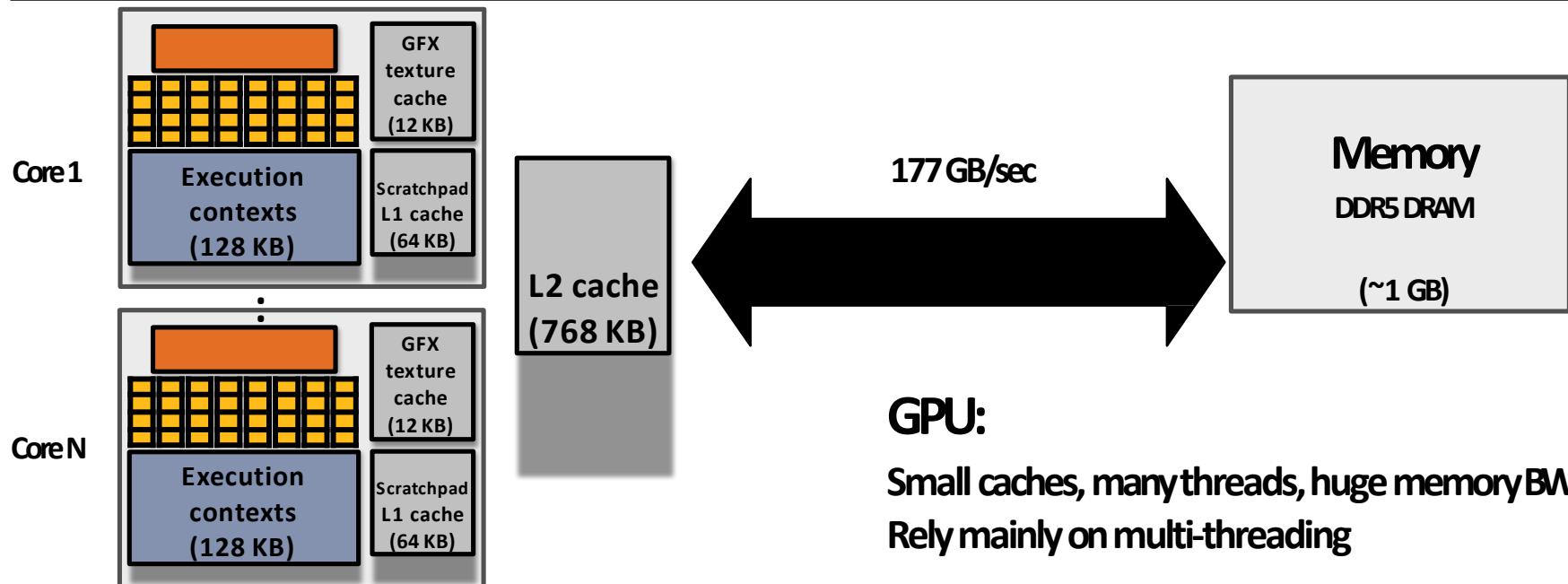


CPU vs. GPU memory hierarchies



CPU:

Big caches, few threads, modest memory BW
Rely mainly on caches and prefetching



GPU:

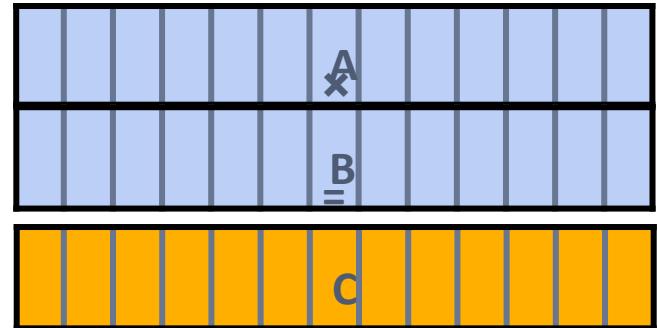
Small caches, many threads, huge memory BW
Rely mainly on multi-threading

Thought experiment

Task: element-wise multiplication of two vectors A and B

Assume vectors contain millions of elements

- Load input A[i]
- Load input B[i]
- Compute $A[i] \times B[i]$
- Store result into C[i]



Three memory operations (12 bytes) for every MUL NVIDIA

GTX480 GPU can do 480 MULs per clock (@1.2 GHz)

Need ~6.4 TB/sec of bandwidth to keep functional units busy (only have 177 GB/sec)

~ 3% efficiency... but 7x faster than quad-core CPU!

(2.6 GHz Core i7 Gen4 quad-core CPU connected to 25 GB/sec memory bus will exhibit similar efficiency on this computation)

Bandwidth limited!

If processors request data at too high a rate, the memory system cannot keep up.

No amount of latency hiding helps this.

Overcoming bandwidth limits are a common challenge for application developers on throughput-optimized systems.

Bandwidth is a critical resource

Performant parallel programs will:

- Organize computation to fetch data from memory less often
 - Reuse data previously loaded by the same thread
(traditional intra-thread temporal locality optimizations)
 - Share data across threads (inter-thread cooperation)
- Request data less often (instead, do more arithmetic: it's "free")
 - Useful term: "arithmetic intensity" — ratio of math operations to data access operations in an instruction stream
 - Main point: programs must have high arithmetic intensity to utilize modern processors efficiently

Summary

- Three major ideas that all modern processors employ to varying degrees
 - Employ multiple processing cores
 - Simpler cores (embrace thread-level parallelism over instruction-level parallelism)
 - Amortize instruction stream processing over many ALUs (SIMD)
 - Increase compute capability with little extra cost
 - Use multi-threading to make more efficient use of processing resources (hide latencies, fill all available resources)
- Due to high arithmetic capability on modern chips, many parallel applications (on both CPUs and GPUs) are bandwidth bound
- GPU architectures use the same throughput computing ideas as CPUs: but GPUs push these concepts to extreme scales

For the rest of this class, know these terms

- Multi-core processor
- SIMD execution
- Coherent control flow
- Hardware multi-threading
 - Interleaved multi-threading
 - Simultaneous multi-threading
- Memory latency
- Memory bandwidth
- Bandwidth bound application
- Arithmetic intensity

Another example:
for review and to check your understanding

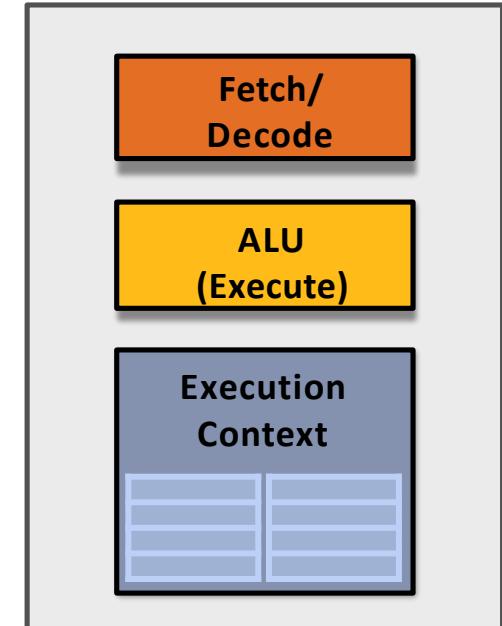
(if you understand the following sequence you understand this lecture)

Running code on a simple processor

My very simple program:
computes $\sin(x)$ using Taylor expansion

```
void sinx(int N, int terms, float* x, float*  
result)  
{  
    for (int i=0; i<N; i++)  
    {  
        float value = x[i];  
        float numer = x[i] * x[i] *  
                     x[i];  int denom = 6; // 3!  
        int sign = -1;  
        for (int j=1; j<=terms; j++)  
        {  
            value += sign * numer /  
            denom;  numer *= x[i] * x[i];  
            denom *= (2*j+2) * (2*j+3);  
            sign *= -1;  
        }  
  
        result[i] = value;  
    }  
}
```

My very simple processor:
completes one instruction per clock

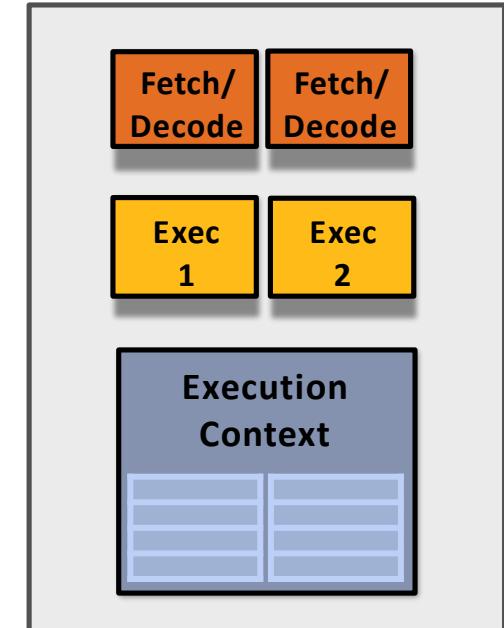


Review: superscalar execution

Unmodified program

```
void sinx(int N, int terms, float* x, float*  
result)  
{  
    for (int i=0; i<N; i++)  
    {  
        float value = x[i];  
        float numer = x[i] * x[i] *  
            x[i];  int denom = 6;// 3!  
        int sign = -1;  
        for (int j=1; j<=terms; j++)  
        {  
            value += sign * numer /  
                denom;  numer *= x[i] * x[i];  
            denom *= (2*j+2) * (2*j+3);  
            sign *= -1;  
        }  
        result[i] = value;  
    }  
}
```

Mysingle core, superscalar processor:
executes up to two instructions per clock
from a single instruction stream.



Independent operations in
instruction stream

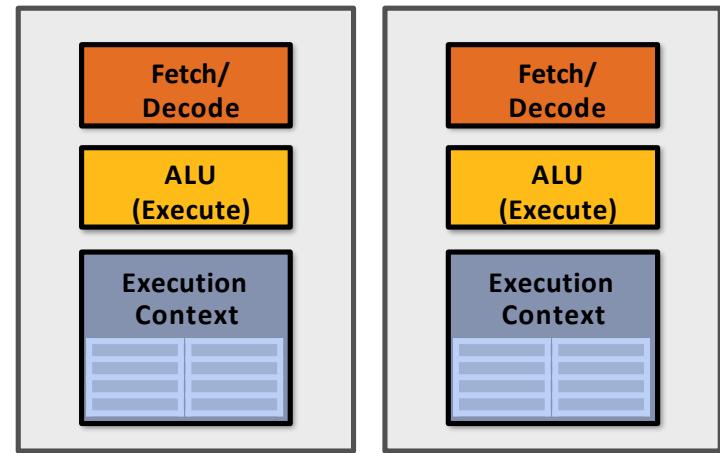
(They are detected by the processor
at run-time and may be executed in
parallel on execution units 1 and 2)

Review: multi-core execution (two cores)

Modify program to create two threads of control (two instruction streams)

```
typedef struct {  
    int N;  
    int terms;  
    float* x;  
    float* result;  
} my_args;  
  
void parallel_sinx(int N, int terms, float* x, float* result)  
{  
    pthread_t  
    thread_id;  my_args  
    args;  
  
    args.N = N/2;  
    args.terms = terms;  
    args.x = x;  
    args.result =  
    result;  
  
    pthread_create(&thread_id, NULL, my_thread_start, &args); // launch  
    thread sinx(N -args.N, terms, x + args.N, result + args.N); // do work  
    pthread_join(thread_id, NULL);  
  
void my_thread_start(void* thread_arg)  
{  
    my_args* thread_args = (my_args*)thread_arg;  
    sinx(args-->N, args-->terms, args-->x, args-->result); // do work  
}
```

My dual-core processor:
executes one instruction per clock
from an instruction stream on each core.

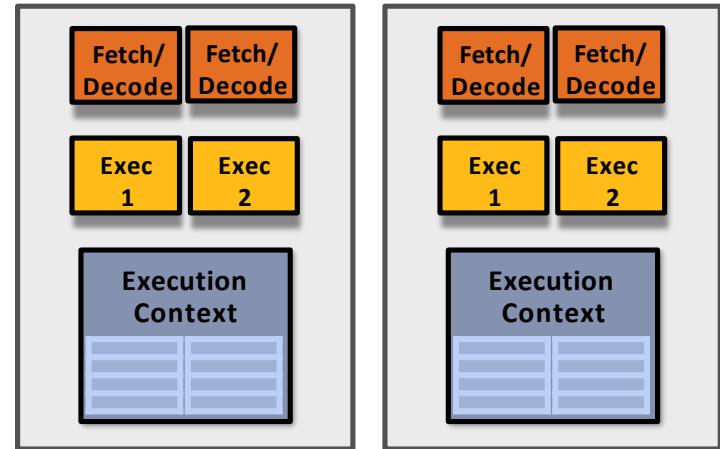


Review: multi-core + superscalar execution

Modify program to create two threads of control (two instruction streams)

```
typedef struct {  
    int N;  
    int terms;  
    float* x;  
    float* result;  
} my_args;  
  
void parallel_sinx(int N, int terms, float* x, float* result)  
{  
    pthread_t  
    thread_id;  my_args  
    args;  
  
    args.N = N/2;  
    args.terms = terms;  
    args.x = x;  
    args.result =  
    result;  
  
    pthread_create(&thread_id, NULL, my_thread_start, &args); // launch  
    thread sinx(N -args.N, terms, x + args.N, result + args.N); // do work  
    pthread_join(thread_id, NULL);  
  
void my_thread_start(void* thread_arg)  
{  
    my_args* thread_args = (my_args*)thread_arg;  
    sinx(args-->N, args-->terms, args-->x, args-->result); // do work  
}
```

My superscalar dual-core processor:
executes up to two instructions per clock
from an instruction stream on each core.



Review: multi-core (four cores)

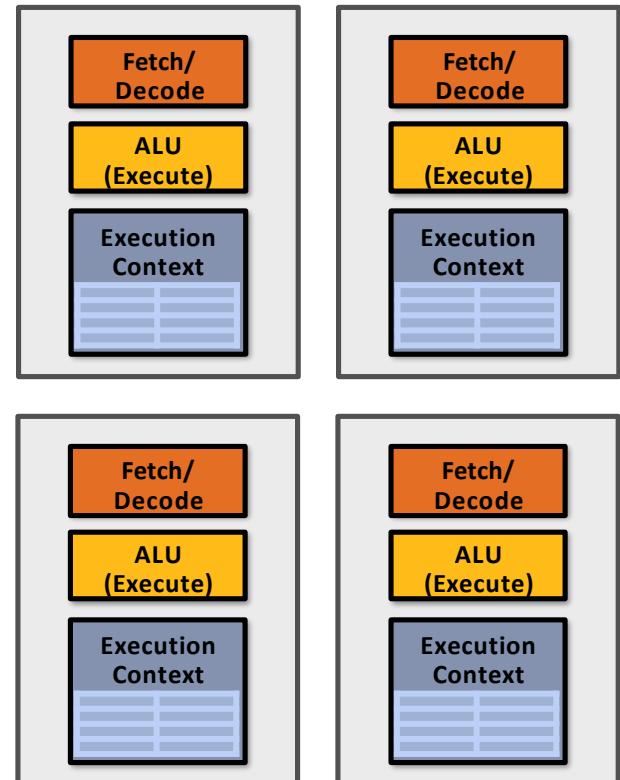
Modify program to create many threads of control:
recall Kayvon's fictitious language

```
void sinx(int N, int terms, float* x, float* result)
{
    // declare independent loop iterations
    forall (int i from 0 to N-1)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;    // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

My quad-core processor:
executes one instruction per clock
from an instruction stream on each core.



Review: four, 8-wide SIMD cores

Observation: program must execute many iterations of the same loop body.

Optimization: share instruction stream across execution of multiple iterations (single instruction multiple data = SIMD)

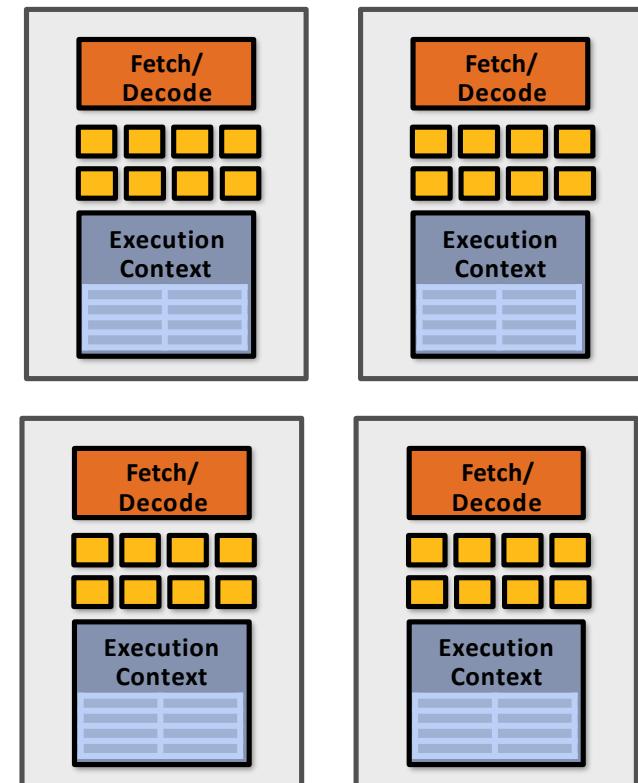
```
void sinx(int N, int terms, float* x, float* result)
{
    // declare independent loop iterations
    forall (int i from 0 to N-1)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

MySIMDquad-core processor:

executes one 8-wide SIMD instruction per clock
from an instruction stream on each core.



Review: four SIMD, multi-threaded cores

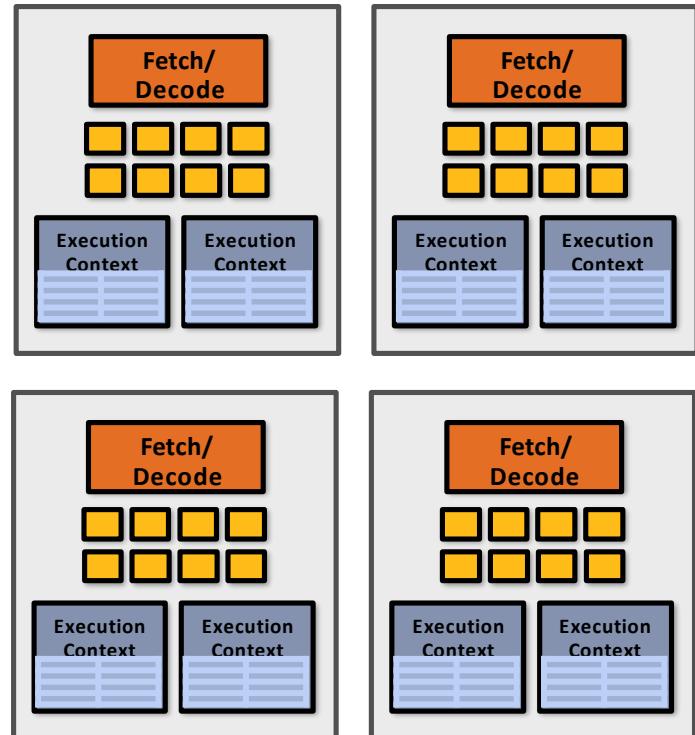
Observation: memory operations have very long latency

Solution: hide latency of loading data for one iteration by executing arithmetic instructions from other iterations

```
void sinx(int N, int terms, float* x, float* result)
{
    // declare independent loop iterations
    forall (int i from 0 to N-1)
    {
        float value = x[i]; Memory load
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<terms; j++)
        {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            sign *= -1(2*j+2) * (2*j+3);
            denom *= (2*j+1) * (2*j+2);
        }
        result[i] = value; Memory store
    }
}
```

My multi-threaded, SIMD quad-core processor:
executes one SIMD instruction per clock
from one instruction stream on each core. But
can switch to processing the other instruction
stream when faced with a stall.

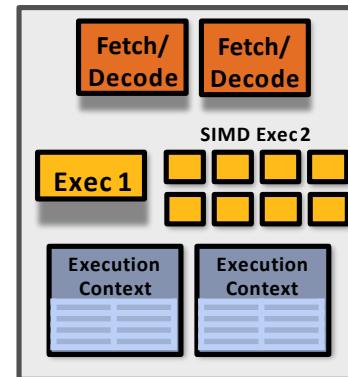
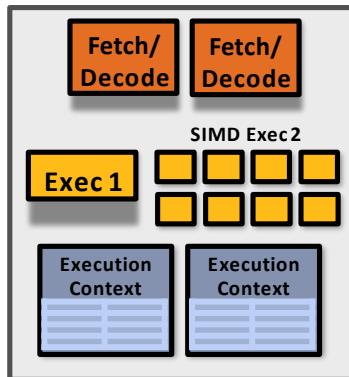
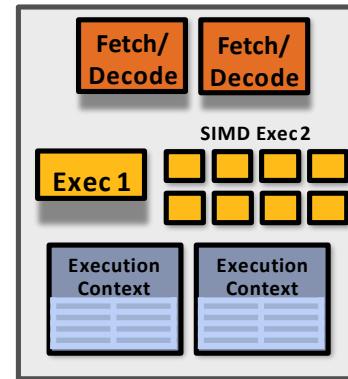
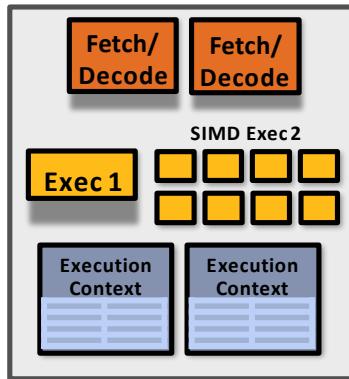


Summary: four superscalar, SIMD, multi-threaded cores

My multi-threaded, superscalar, SIMD quad-core processor:

**executes up to two instructions per clock from one instruction stream on each core
(in this example: one SIMD instruction + one scalar instruction).**

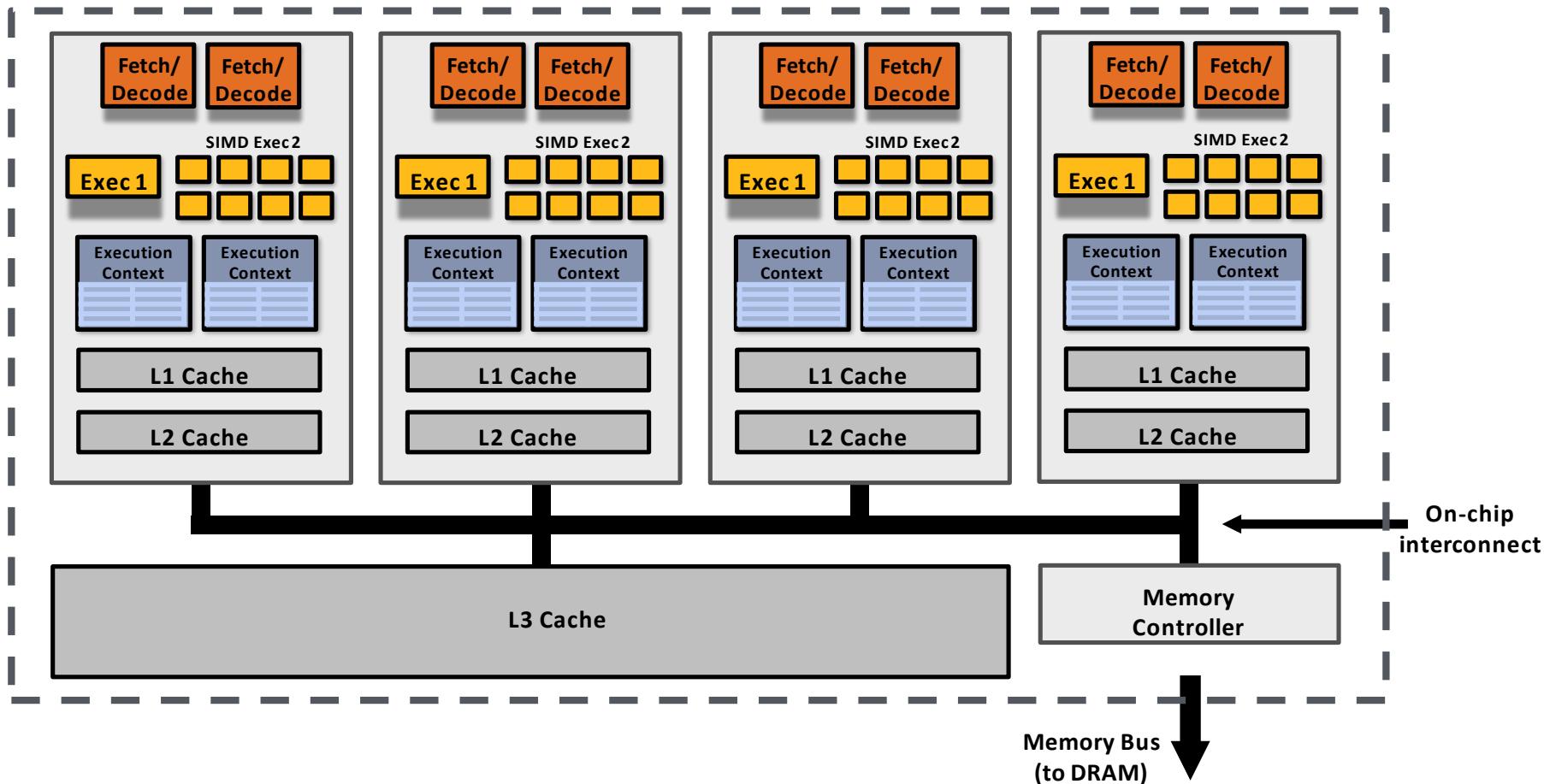
Processor can switch to execute the other instruction stream when faced with stall.



Connecting it all together

Kayvon's simple quad-core processor:

Four cores, two-way multi-threading per core (max eight threads active on chip at once), up to two instructions per clock per core (one of those instructions is 8-wide SIMD)



Questions

CSCI465/ECEN433

Introduction to Parallel Computing

Fall 2023



Lecture (4)

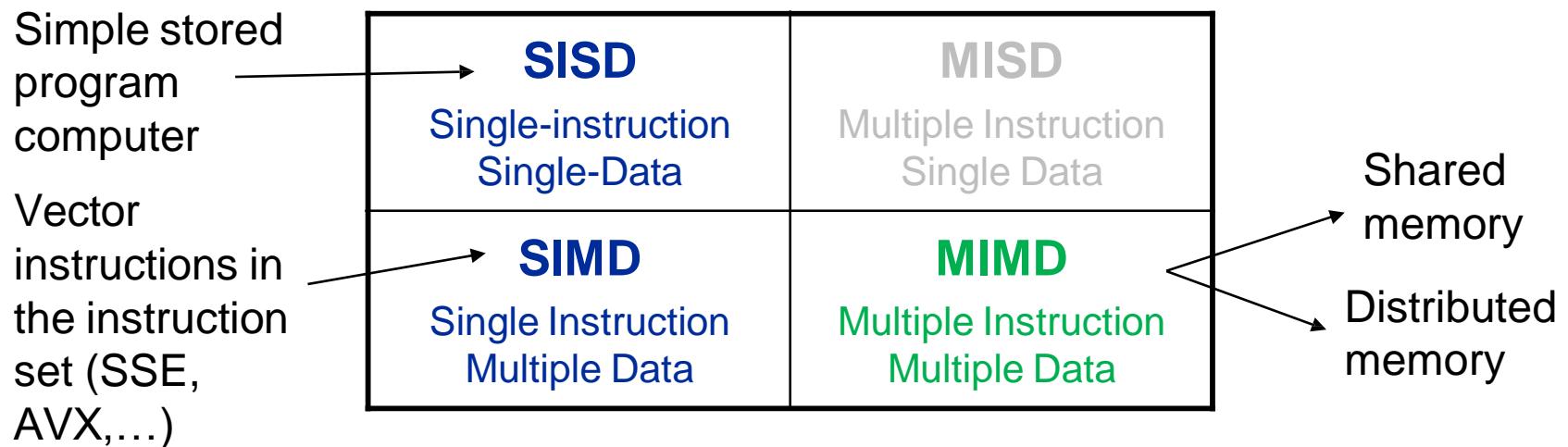
Parallel Programming Abstractions

(and their corresponding HW/SW implementations)

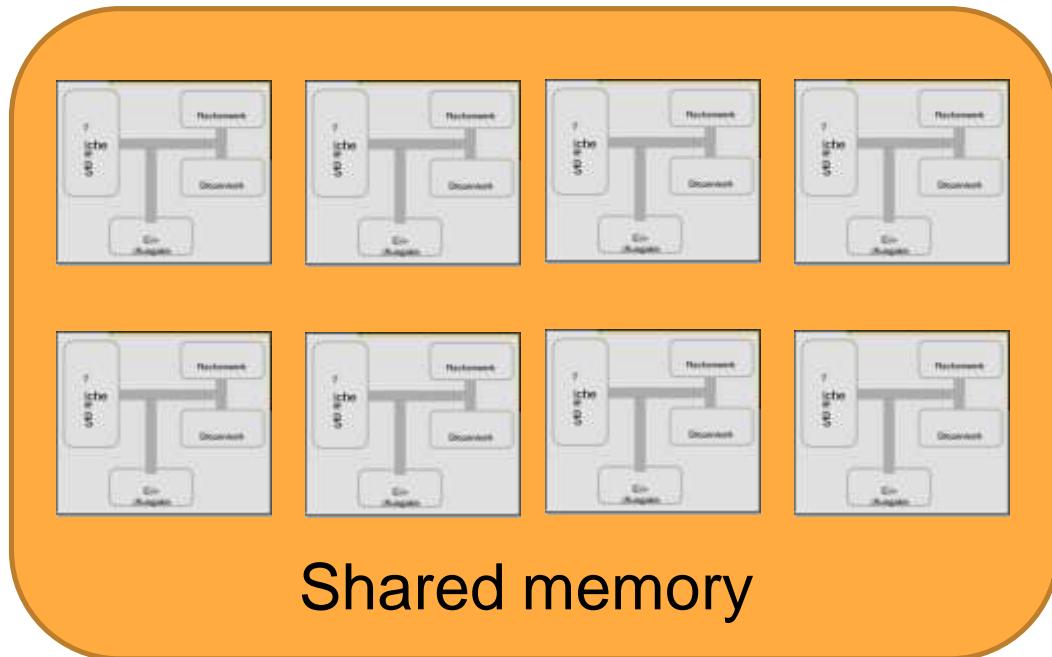


Classification of parallel computers

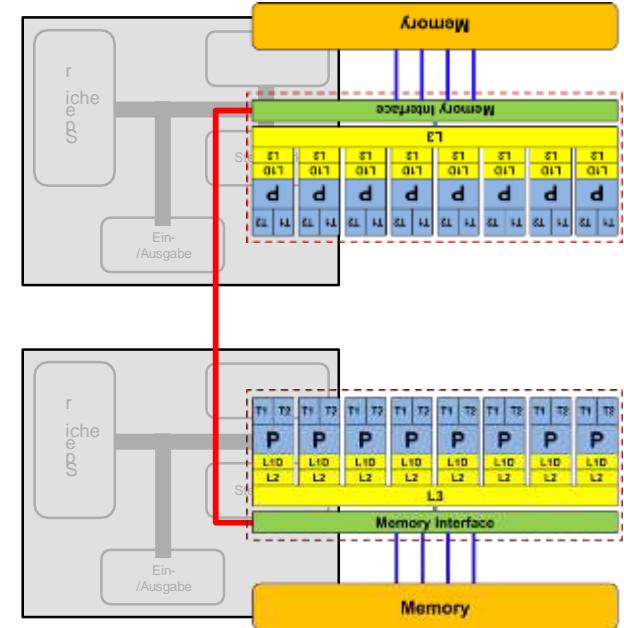
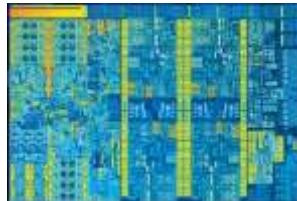
- **Parallel Computing:** A number of compute elements solve a problem in a cooperative way
- **Parallel Computer:** A number of compute elements connected in such a way as to do parallel computing for a large set of applications
- Classification according to Flynn (1972) DOI: [10.1109/TC.1972.5009071](https://doi.org/10.1109/TC.1972.5009071)



Shared memory: a single cache-coherent address space

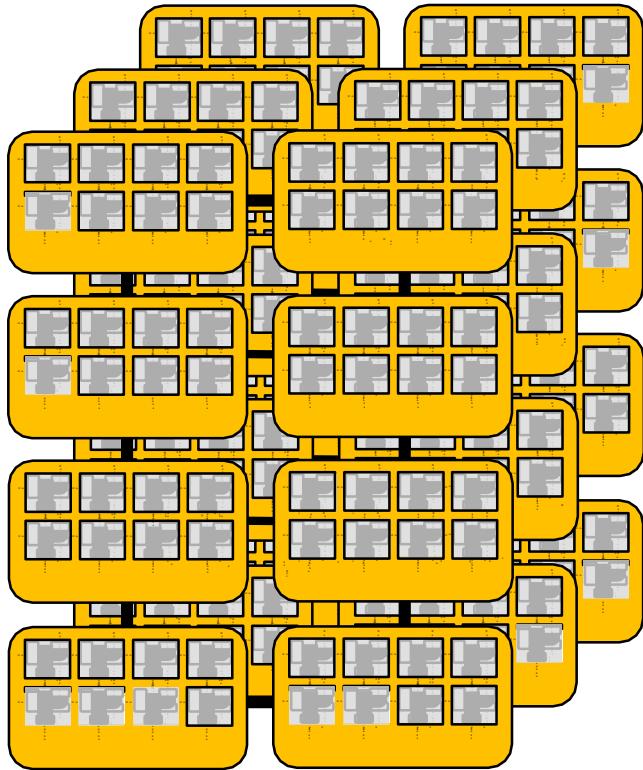


Multi-core processor



Multiple CPU chips
per node

Distributed memory: no cache-coherent single address space



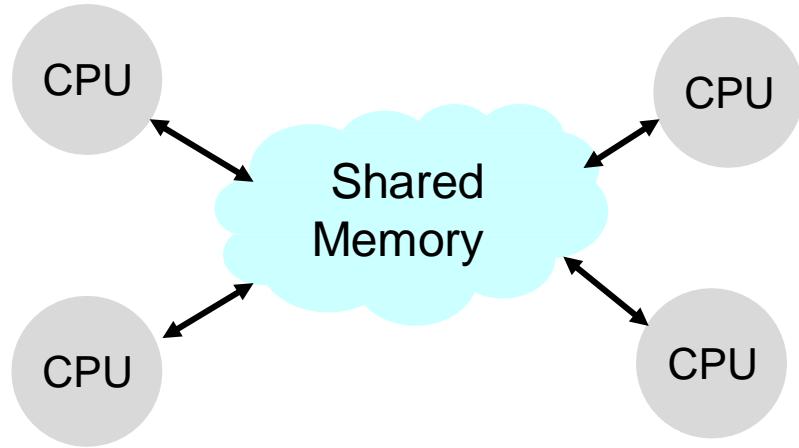
Cluster/
supercomputer

Modern supercomputers are
shared-/distributed-memory hybrids

Shared-memory parallel computers

Shared memory

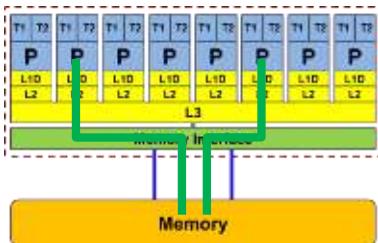
- Single address space for all processors/cores
- Cache coherent, i.e., changes in one cache will be communicated to all others for consistency
- Two basic variants: UMA and ccNUMA



UMA vs. ccNUMA

[cache-coherent]
Uniform Memory Access

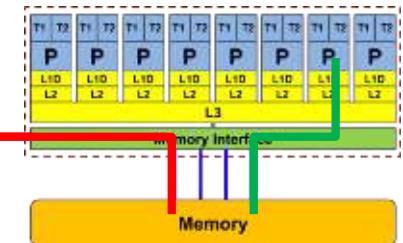
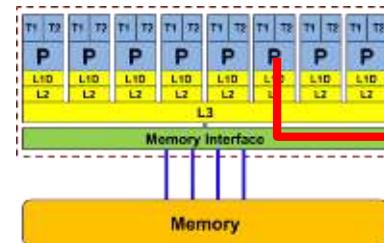
All memory accessible by all cores with equal latency and bandwidth



cache-coherent
Non-Uniform Memory Access

Latency and bandwidth vary depending on mutual position of core and memory

But why???



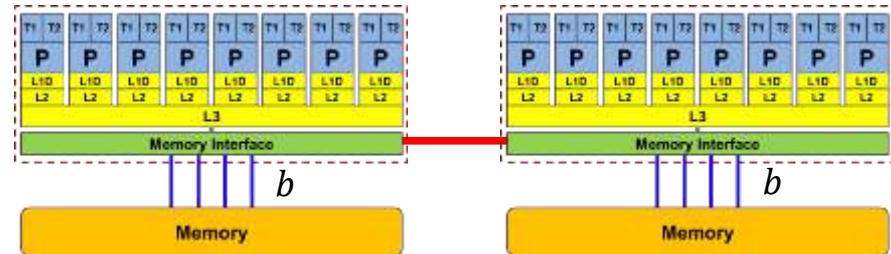
Why ccNUMA?

- Many algorithms rely on high Memory bandwidth:

$$b = \frac{V}{T}$$

V data transferred over memory bus [byte]

T wallclock time [s]



- Advantage:** Easier (cheaper) to build multiple domains with smaller bandwidth than one UMA domain with high bandwidth
- Disadvantage:** Adds “topology” (non-uniformity in memory access, need to know where my threads are running)

Shared-memory parallel programming

- Many programming models exist
- Popular in scientific computing: OpenMP (<https://openmp.org>)
- Source code directives interpreted by compiler (+ small API)
- Example:

```
• double s = 0.0, *a, *b, *c;  
• ...  
• #pragma omp parallel for reduction(+:s)  
• for(int i=0; i<n; i++) {  
•     a[i] = b[i] + c[i]; s = s + a[i];  
• }
```

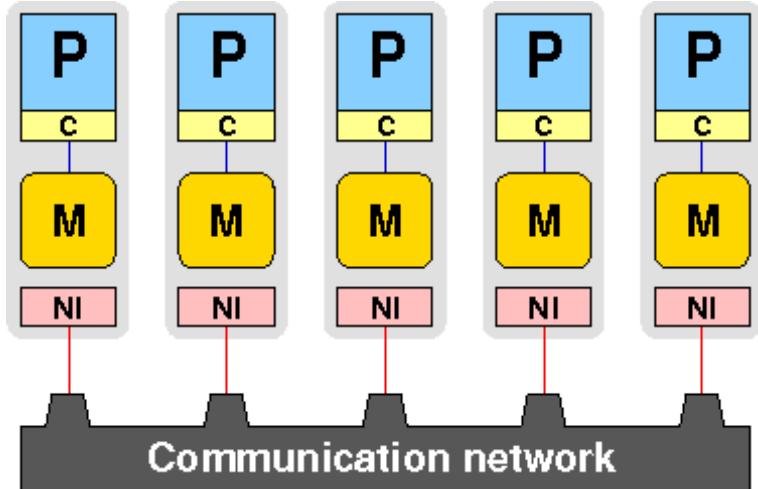
- OpenMP has
- a directive sentinel
 - parallelization directives
 - work-sharing directives
 - clauses
 - ... and much more

Distributed-memory parallel computers

Distributed-memory systems “back in the day”

“Pure” distributed-memory system:

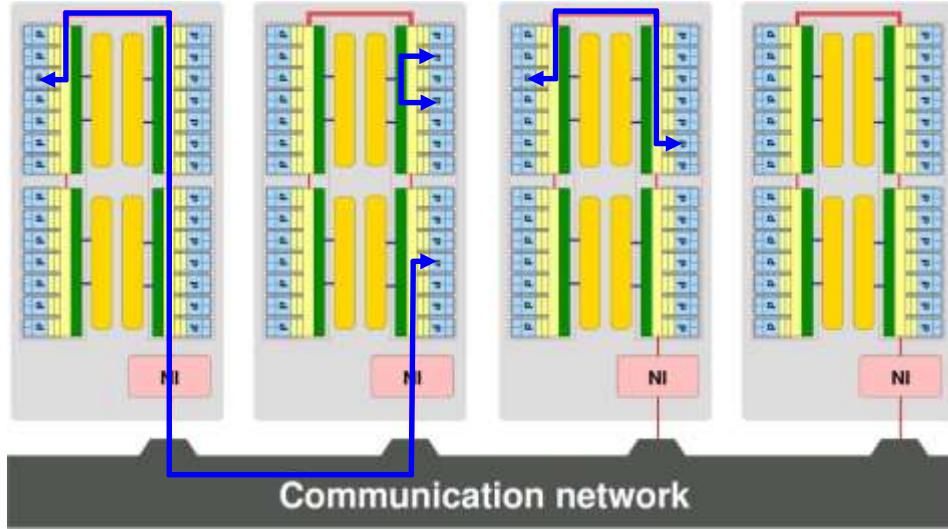
- Individual processors with exclusive local memory (M) and a network interface (NI)
 - one “node” == one processor core
- Dedicated communication network
- Parallel program == one process per node
- Data exchange via “message passing” over the network
- This was a thing not so long ago...



Distributed-memory systems today

“Hybrid” distributed-/shared-memory systems

- Cluster of networked shared-memory nodes
- ccNUMA architecture per node
- Multiple cores per ccNUMA domain
- Expect strong topology effects in communication performance
 - Intra-socket, inter-socket, inter-node, all have different λ and b
 - On top: Effects from network structure



Distributed-memory parallel programming

- Many programming models exist
- Dominant in scientific computing: MPI, the Message Passing Interface (<https://mpi-forum.org>)
- Library standard, several open & commercial implementations (Intel, OpenMPI,...)
- Processes communicating via message transfers
- Hundreds of functions
- Significantly more complex than OpenMP
- Can use MPI on shared memory!

```
include <mpi.h>

int main(int argc, char** argv) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Hello World! I am %d of %d\n",
          rank, size);

    MPI_Finalize();
}
```

Summary on parallel computer architecture

- Modern systems exhibit parallelism on multiple levels
 - Multi-core, multi-NUMA-domain, multi-node
- Mixture of shared- and distributed-memory architecture
 - Shared memory on the node, distributed between nodes

Programming models: There are ten a penny, but....:

- OpenMP
 - Shared-memory programming
 - Compiler directives, thread based
- Message Passing Interface (MPI)
 - Distributed-memory programming
 - Library calls, process based

Today's theme is a critical idea in this course.

And today's theme is:

Abstraction vs. Implementation

Conflating abstraction with implementation is a common cause for confusion in this course.

An example: Programming with ISPC

ISPC

- Intel SPMD Program Compiler (**ISPC**)
- **SPMD: single program multiple data**
- <https://ispc.github.io/>

Recall: example program from last class

Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$
for each element of an array of N floating-point numbers

```
void sinx(int N, int terms, float* x, float* result)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

$\sin(x)$ in ISPC

Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$

C++ code: main.cpp

```
#include "sinx_ispc.h"

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```

ISPC code: sinx.ispc

```
export void sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    // assume N % programCount = 0
    for (uniform int i=0; i<N; i+=programCount)
    {
        int idx = i + programIndex;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6; // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

SPMD programming abstraction:

Call to ISPC function spawns “**gang**” of ISPC
“**program instances**”

All instances run ISPC code **concurrently**

Upon return, all instances have completed

$\sin(x)$ in ISPC

Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$

C++ code: main.cpp

```
#include "sinx_ispc.h"

int N =
1024; int
terms = 5;
float* x = new float[N];
float* result = new
float[N];

// initialize x here
// execute ISPC code
sinx(N, terms, x,
result);
```

SMPD programming abstraction:

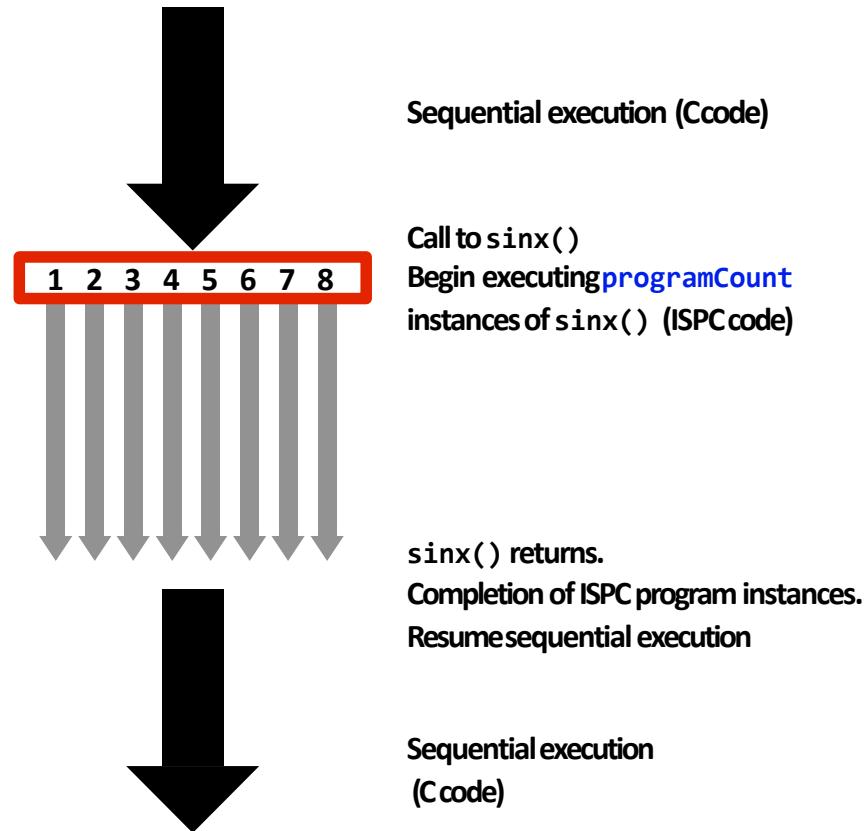
Call to ISPC function spawns “gang” of ISPC
“program instances”

All instances run ISPC code concurrently

Upon return, all instances have completed

In this illustration **programCount** = 8

CSCI465/ECEN433 – Introduction to Parallel Computing



$\sin(x)$ in ISPC

“Interleaved” assignment of array elements to program instances

C++ code: main.cpp

```
#include "sinx_ispc.h"

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```

ISPC code: sinx.ispc

```
export void sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    // assumes N % programCount = 0
    for (uniform int i=0; i<N; i+=programCount)
    {
        int idx = i + programIndex;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6; // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

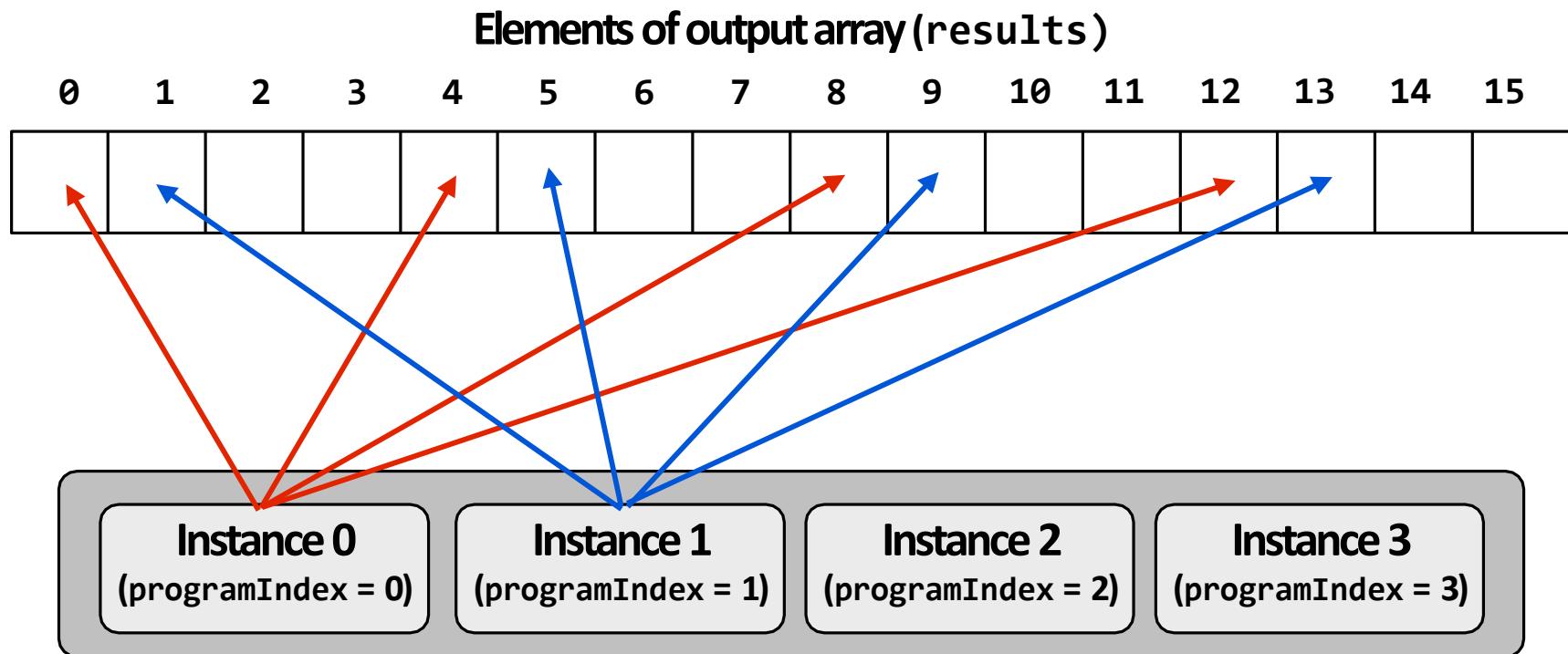
ISPC Keywords:

programCount: number of simultaneously executing instances in the gang (uniform value)

programIndex: id of the current instance in the gang. (a non-uniform value: “varying”)

uniform: A type modifier. All instances have the same value for this variable. Its use is purely an optimization. Not needed for correctness.

Interleaved assignment of program instances to loop iterations



“Gang” of ISPC program instances

In this illustration: gang contains four instances: **programCount = 4**

ISPC implements the gang abstraction using SIMD instructions

C++ code: main.cpp

```
#include "sinx_ispc.h"

int N =
1024; int
terms = 5;
float* x = new float[N];
float* result = new
float[N];
// initialize x here
// execute ISPC code
sinx(N, terms, x,
result);
```

SPMD programming abstraction:

Call to ISPC function spawns “gang” of ISPC “program instances”

All instances run ISPC code concurrently

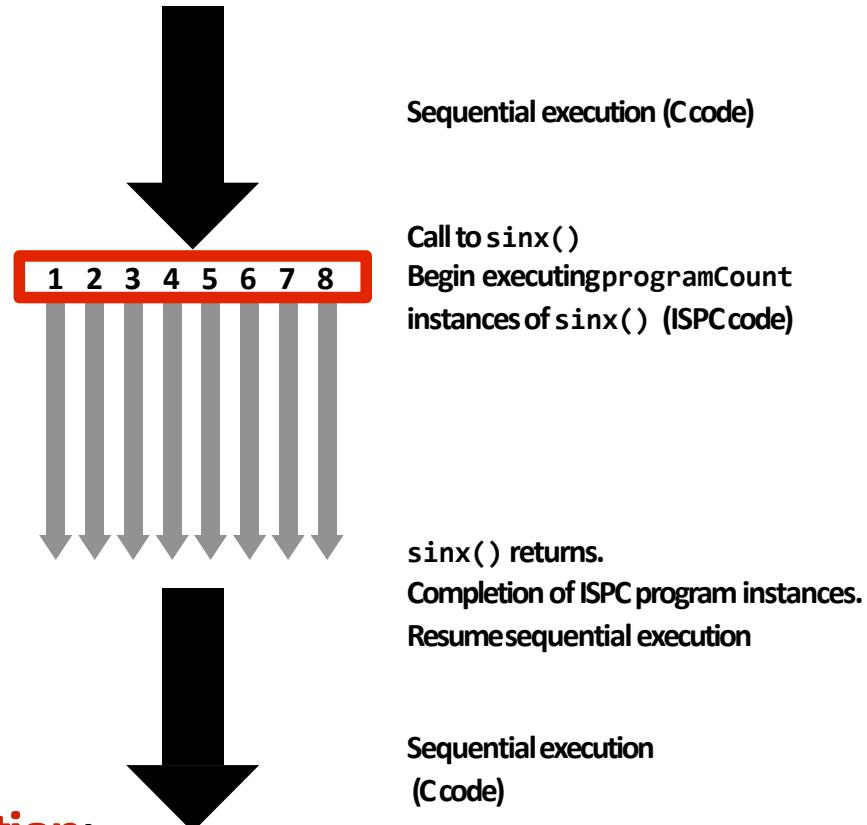
Upon return, all instances have completed

ISPC compiler generates SIMD implementation:

Number of instances in a gang is the SIMD width of the hardware (or a small multiple of SIMD width)

ISPC compiler generates binary (.o) with SIMD instructions

C++ code links against object file as usual



Sequential execution (C code)

Call to sinx()
Begin executing programCount instances of sinx() (ISPC code)

sinx() returns.
Completion of ISPC program instances.
Resume sequential execution

Sequential execution
(C code)

$\sin(x)$ in ISPC: version 2

“Blocked” assignment of elements to instances

C++ code: main.cpp

```
#include "sinx_ispc.h"

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

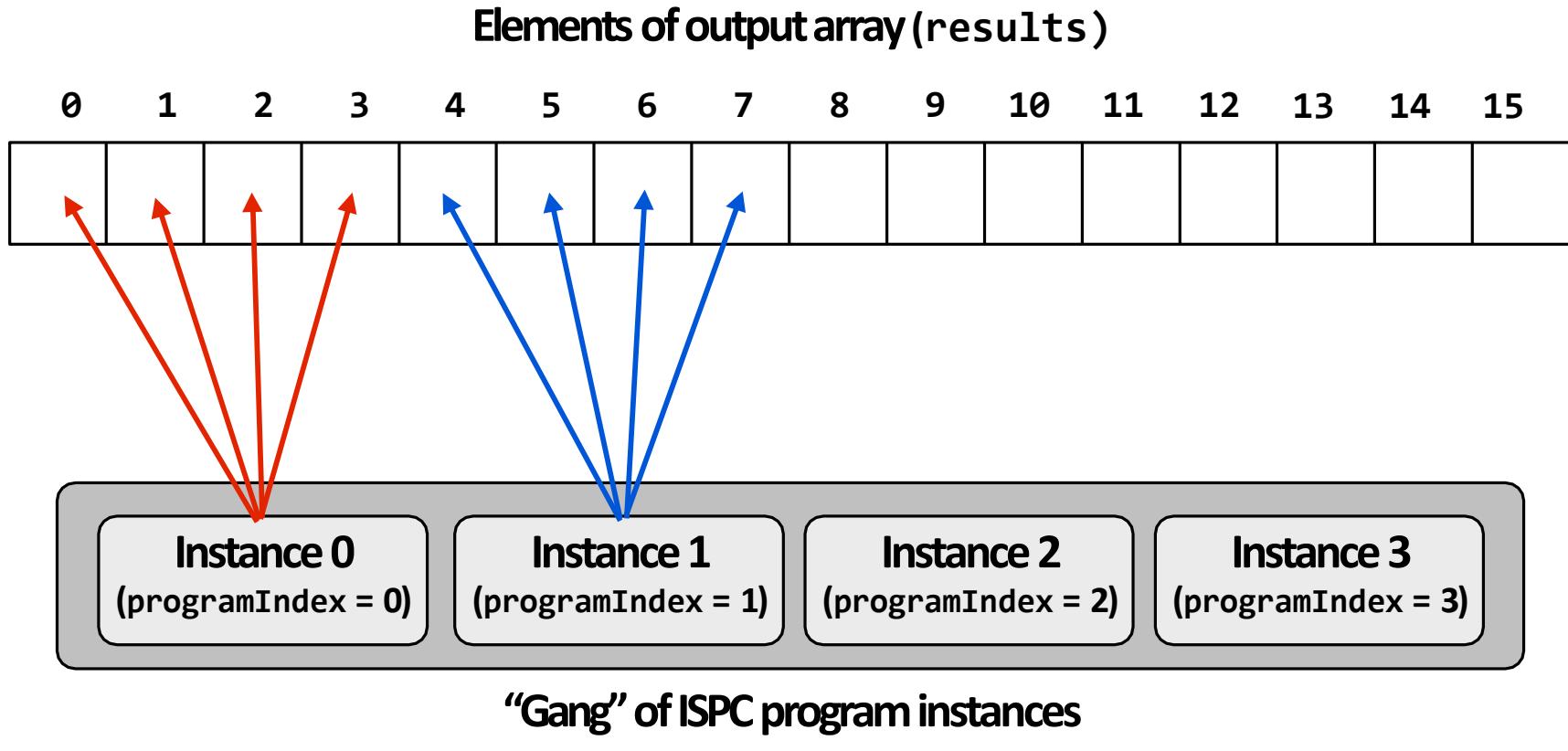
// execute ISPC code
sinx(N, terms, x, result);
```

ISPC code: sinx.ispc

```
export void sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    // assume N % programCount = 0
    uniform int count = N / programCount;
    int start = programIndex * count;  for
    (uniform int i=0; i<count; i++)
    {
        int idx = start + i;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6; // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (j+3) * (j+4);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

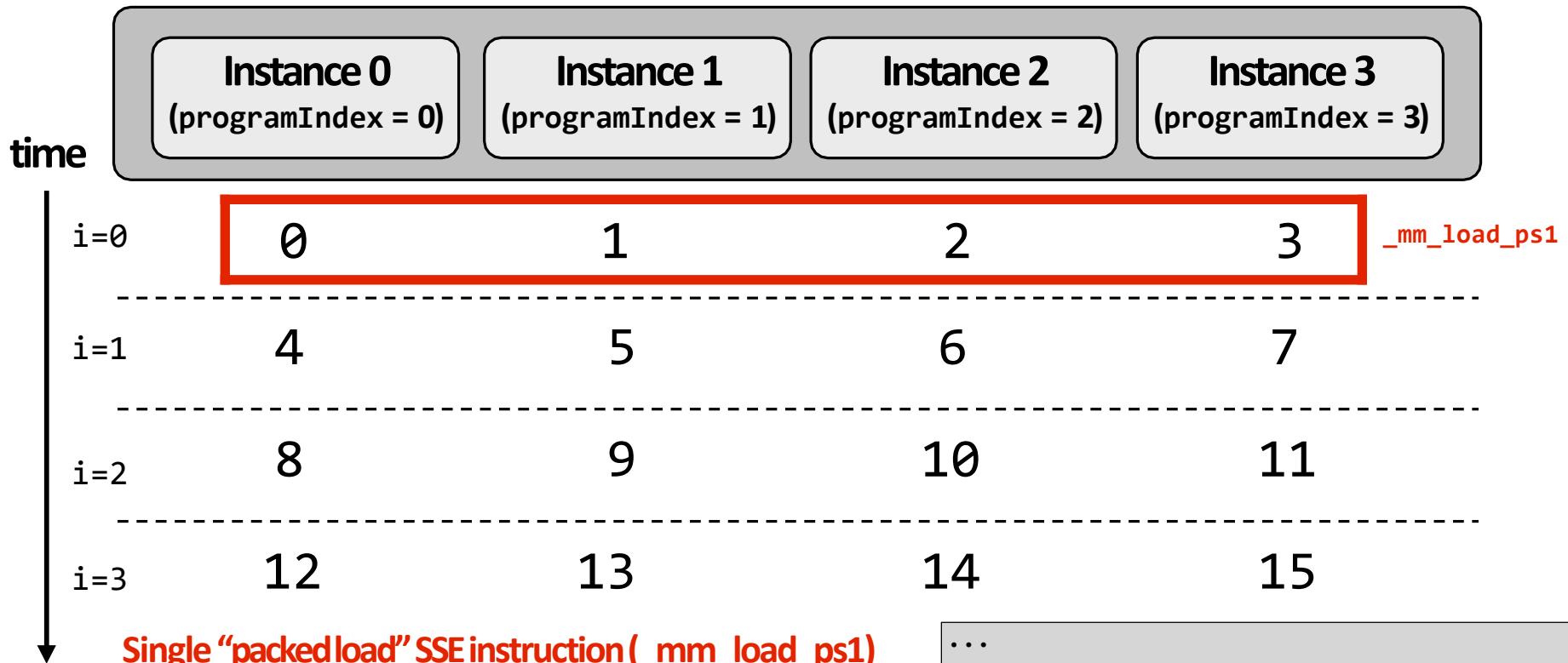
Blocked assignment of program instances to loop iterations



Schedule: interleaved assignment

“Gang” of ISPC program instances

Gang contains four instances: `programCount = 4`



Single “packed load” SSE instruction (`_mm_load_ps1`) efficiently implements:

`float value = x[idx];`

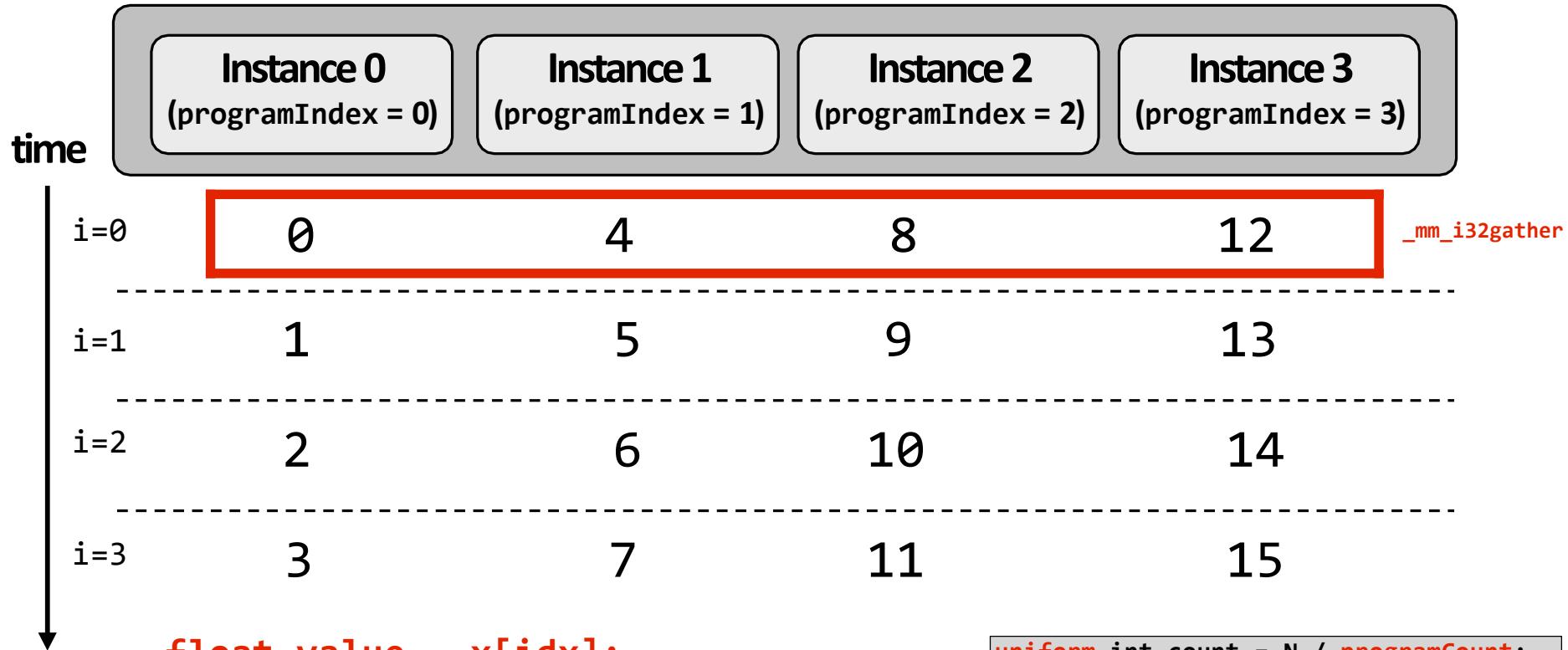
for all program instances, since the four values are contiguous in memory

```
...  
// assumes N % programCount = 0  
for (uniform int i=0; i<N; i+=programCount)  
{  
    int idx = i + programIndex;  
    float value = x[idx];  
    ...
```

Schedule: blocked assignment

“Gang” of ISPC program instances

Gang contains four instances: `programCount = 4`



```
uniform int count = N / programCount;
int start = programIndex * count;
for (uniform int i=0; i<count; i++) {
    int idx = start + i;
    float value = x[idx];
    ...
}
```

Raising level of abstraction with foreach

Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$

C++ code: main.cpp

```
#include "sinx_ispc.h"

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```

ISPC code: sinx.ispc

```
export void sinx(
    uniform int N,
    uniform int
    terms,
    uniform float*
    x,
    uniform float* result)
{
    foreach (i = 0 ... N)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        uniform int denom = 6;      // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[i] = value;
    }
}
```

foreach: key ISPC language construct

- **foreach** declares parallel loop iterations
 - Programmers say: these are the iterations the instances in a gang must cooperatively perform
- ISPC implementation assigns iterations to program instances in gang
 - Current ISPC implementation will perform a **static interleaved assignment** (but the abstraction permits a **different assignment**)

ISPC: abstraction vs. implementation

- **Single program, multiple data (SPMD) programming model**
 - Programmer “thinks”: running a gang is spawning `programCount` logical instruction streams (each with a different value of `programIndex`)
 - This is the programming abstraction
 - Program is written in terms of this abstraction
- **Single instruction, multiple data (SIMD) implementation**
 - ISPC compiler emits vector instructions (SSE4 or AVX) that carry out the logic performed by a ISPC gang
 - ISPC compiler handles mapping of conditional control flow to vector instructions (by masking vector lanes, etc.)
- **Semantics of ISPC can be tricky**
 - SPMD abstraction + uniform values
(allows implementation details to peak through abstraction a bit)

ISPC discussion: sum “reduction”

Compute the sum of all array elements in parallel

```
export uniform float sumall1(
    uniform int N,
    uniform float* x)
{
    uniform float sum = 0.0f;
    foreach (i = 0 ... N)
    {
        sum += x[i];
    }
    return sum;
}
```

```
export uniform float sumall2(
    uniform int N,
    uniform float* x)
{
    uniform float sum;
    float partial = 0.0f;
    foreach (i = 0 ... N)
    {
        partial += x[i];
    }

    // from ISPC math library
    sum = reduce_add(partial);

    return sum;
}
```

Correct ISPC solution

sum is of type `uniform float` (one copy of variable for all program instances)
x[i] is **not a uniform expression** (different value for each program instance)
Result: compile-time type error

ISPC discussion: sum “reduction”

Compute the sum of all array elements in parallel

Each instance accumulates a **private partial sum** (no communication)

Partial sums are added together using the **reduce_add()** cross-instance communication primitive. The result is the same total sum for all program instances (**reduce_add()** returns a uniform float)

The ISPC code at right will execute in a manner similar to handwritten C + AVX intrinsics implementation below.*

```
float sumall2(int N, float* x) {  
  
    float tmp[8]; // assume 16-byte alignment  
    __mm256 partial = _mm256_broadcast_ss(0.0f);  
  
    for (int i=0; i<N; i+=8)  
        partial = _mm256_add_ps(partial, _mm256_load_ps(&x[i]));  
  
    _mm256_store_ps(tmp, partial);  
  
    float sum = 0.f;  
    for (int i=0; i<8; i++)  
        sum += tmp[i];  
  
    return sum;  
}
```

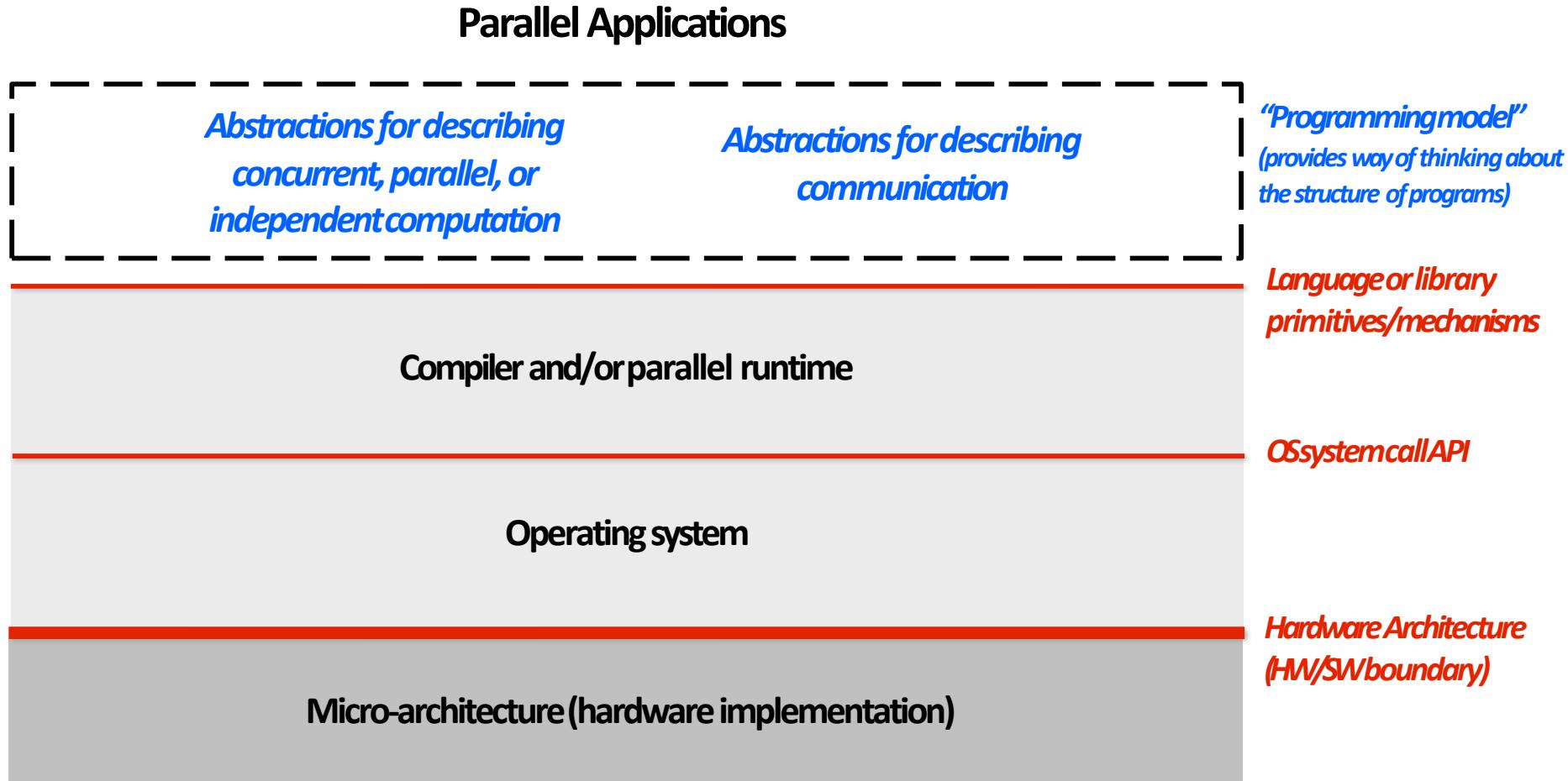
```
export uniform float sumall2(  
    uniform int N,  
    uniform float* x)  
{  
    uniform float sum;  
    float partial = 0.0f;  
    foreach (i = 0 ... N)  
    {  
        partial += x[i];  
    }  
  
    // from ISPC math library  
    sum = reduce_add(partial);  
  
    return sum;  
}
```

*Self-test: If you understand why this implementation complies with the semantics of the ISPC gang abstraction, then you've got good command of ISPC.

ISPC tasks

- The ISPC **gang abstraction** is implemented by SIMD instructions on one core.
- So... all the code I've shown you in the previous slides would have executed **on only one of the four cores** of the GHC machines.
- ISPC contains **another abstraction**: a “**task**” that is used to achieve multi-core execution. I'll let you read up about that.

System layers: interface, implementation, interface, ...

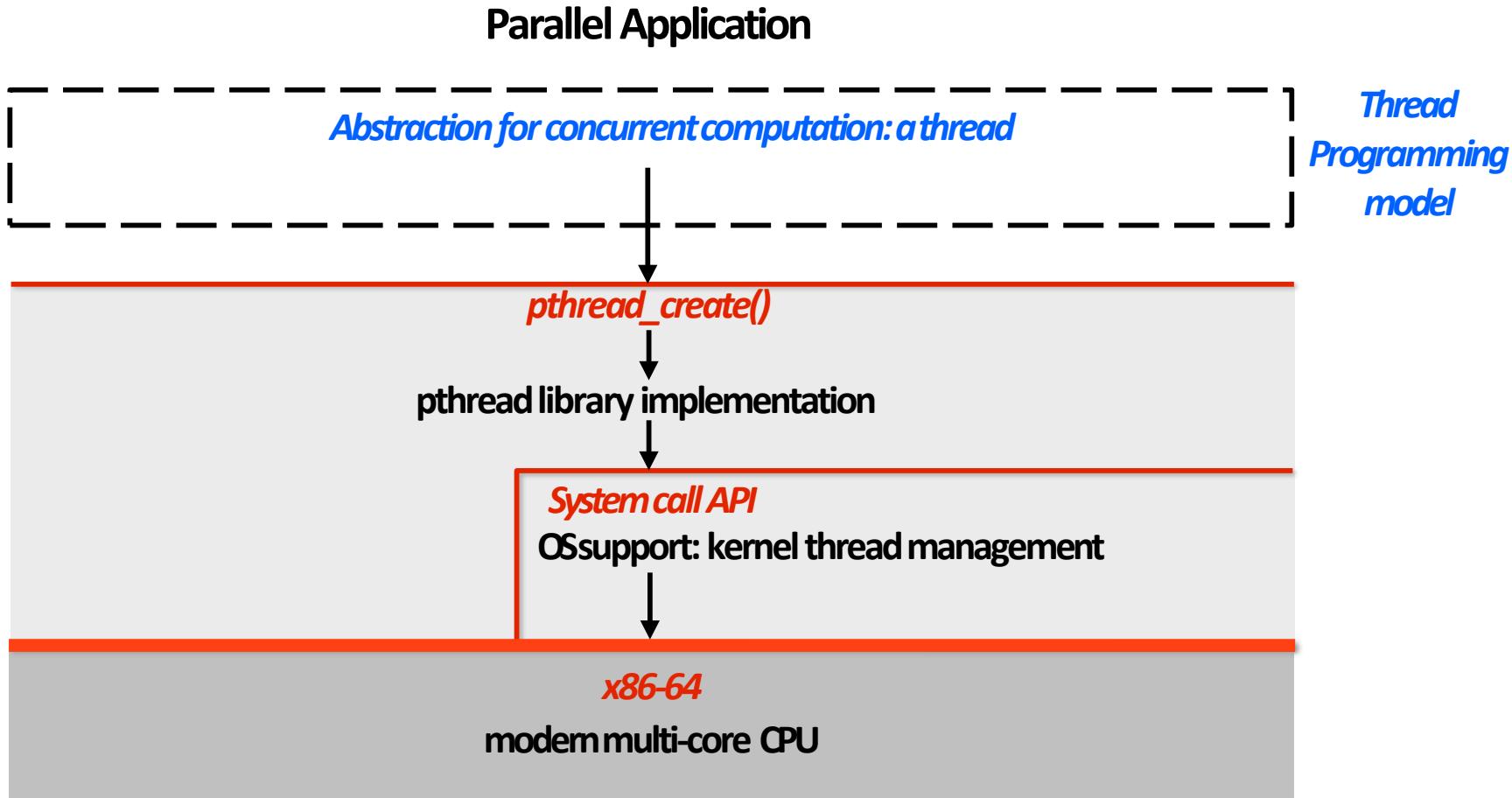


Blue italic text: abstraction/concept

Red italic text: system interface

Black text: system implementation

Example: expressing parallelism with **pthreads**



Blue italic text: abstraction/concept

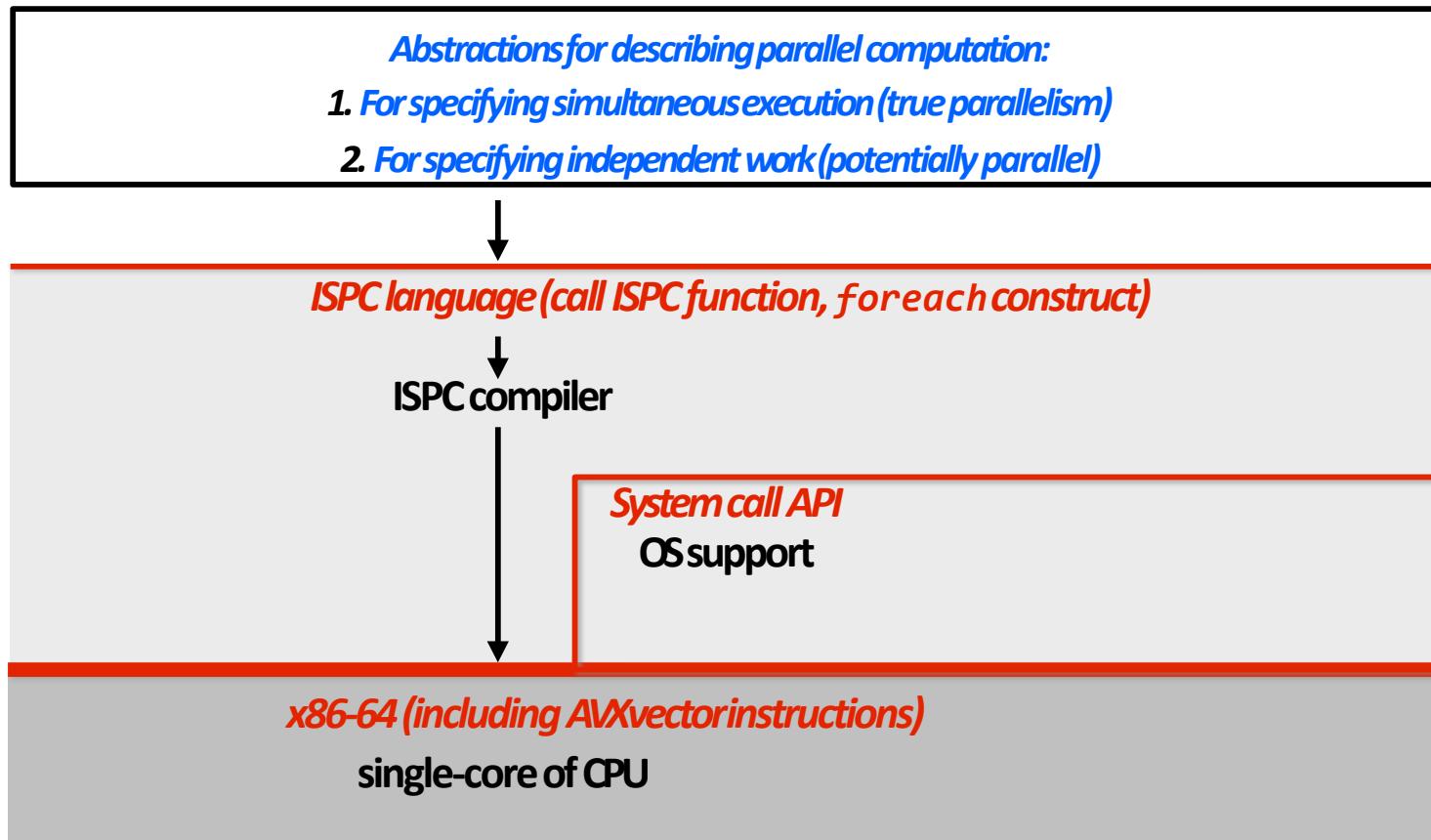
Red italic text: system interface Black

text: system implementation

Example: expressing parallelism with ISPC

Parallel Applications

ISPC
Programming
model



Note: This diagram is specific to the ISPC gang abstraction. ISPC also has the “task” language primitive for multi-core execution. I don’t describe it here but it would be interesting to think about how that diagram would look

Three models of communication (abstractions)

1. Shared address space

2. Message passing

3. Data parallel

Questions

CSCI465/ECEN433

Introduction to Parallel Computing

Spring 2024



Lecture (5)

Parallel Programming Models



Three models of communication (abstractions)

1. Shared address space

2. Message passing

3. Data parallel

Shared address space model of communication

Shared address space model (abstraction)

- Threads communicate by reading/writing to shared variables
- Shared variables are like a big bulletin board
 - Anythread can read or write to shared variables

Thread 1:

```
int x = 0;  
spawn_thread(foo, &x);  
x = 1;
```

Thread 2:

```
void foo(int* x) {  
    while (x == 0) {}  
    print x;  
}
```

Thread 1

Store to x

x

Shared address space

Thread 2

Load from x

(Communication operations shown in red)

(Pseudocode provided in a fake C-like language for brevity.)

CSCI465/ECEN433 – Introduction to Parallel Computing

Shared address space model (abstraction)

Synchronization primitives are also shared variables: e.g., locks

Thread 1:

```
int x = 0;  
Lock my_lock;  
  
spawn_thread(foo, &x, &my_lock);
```

```
mylock.lock();  
x++;  
mylock.unlock();
```

Thread 2:

```
void foo(int* x, lock* my_lock)  
{  
    my_lock->lock();  
    x++;  
    my_lock->unlock();  
  
    print x;  
}
```

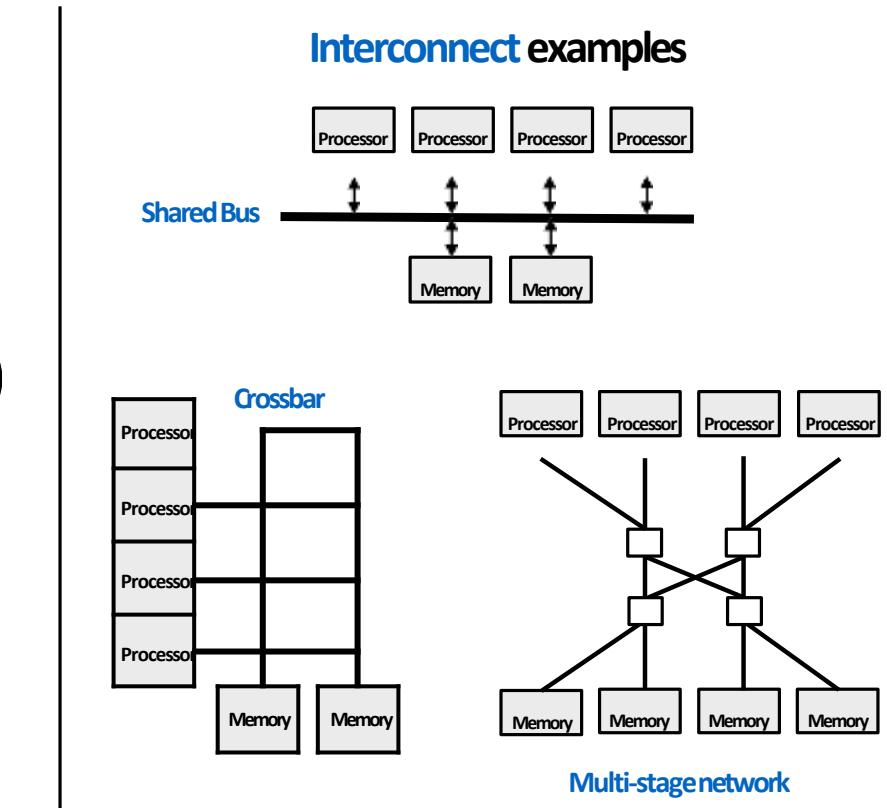
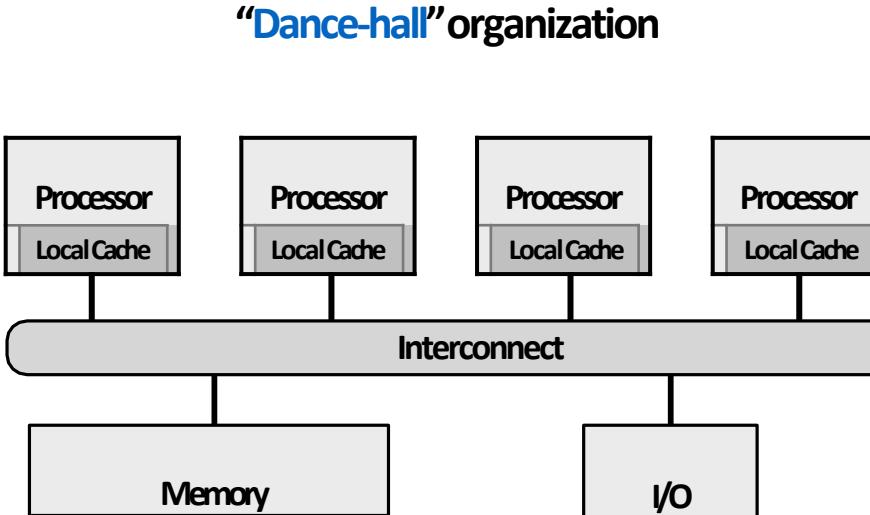
(Pseudocode provided in a fake C-like language for brevity.)

Shared address space model (abstraction)

- Threads **communicate** by:
 - **Reading/writing to shared variables**
 - Inter-thread communication is implicit in memory operations
 - Thread 1 stores to X
 - Later, thread 2 reads X (and observes update of value by thread 1)
 - **Manipulating synchronization primitives**
 - e.g., ensuring mutual exclusion via use of locks
- This is a natural extension of sequential programming
 - In fact, all our discussions in class have assumed a shared address space so far!
- Helpful analogy: shared variables are like a big bulletin board
 - Anythread can read or write to shared variables

HW implementation of a shared address space

Key idea: any processor can directly reference any memory location



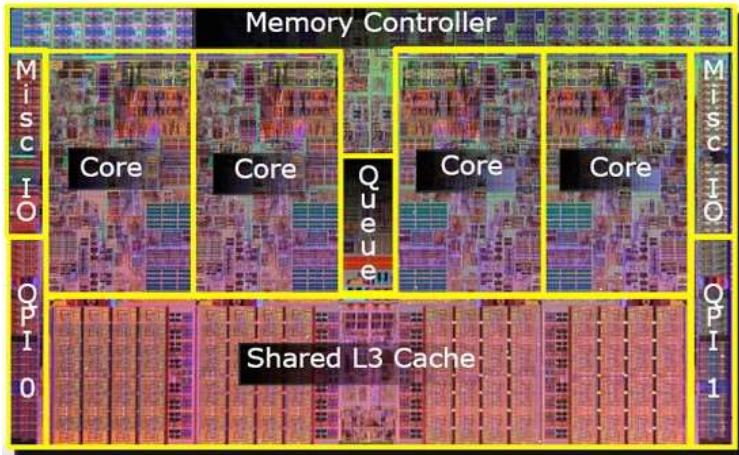
Symmetric (shared-memory) multi-processor (SMP):

- Uniform memory access time: cost of accessing an uncached *
memory address is the same for all processors

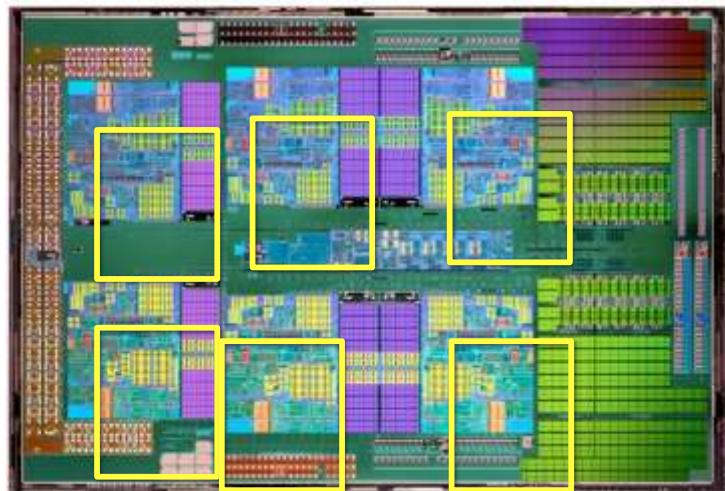
*caching introduces non-uniform access times, but we'll talk about that later

Shared address space HWarchitectures

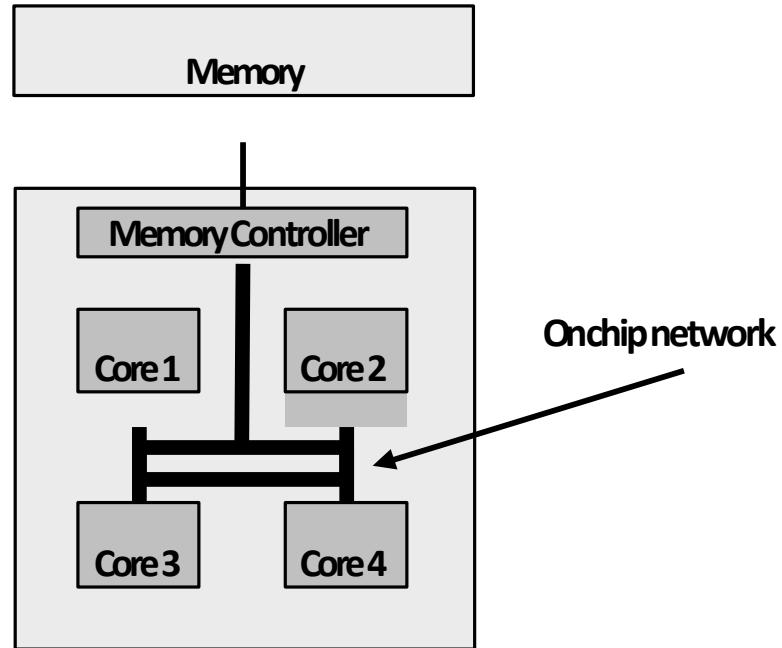
Commodity x86 examples



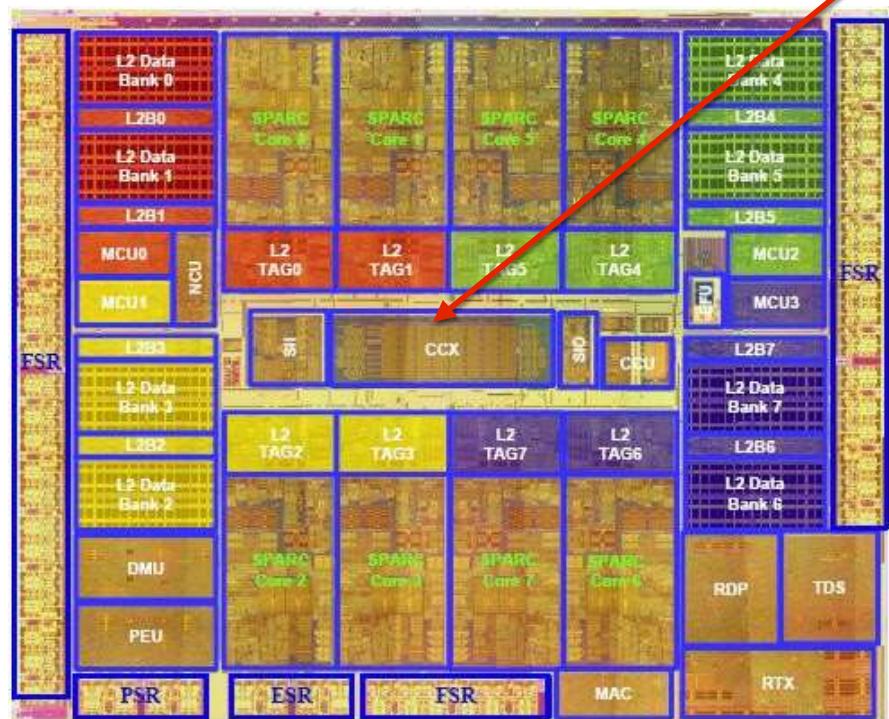
Intel Core i7 (quad core)
(interconnect is a ring)



AMDPheon II (six core)

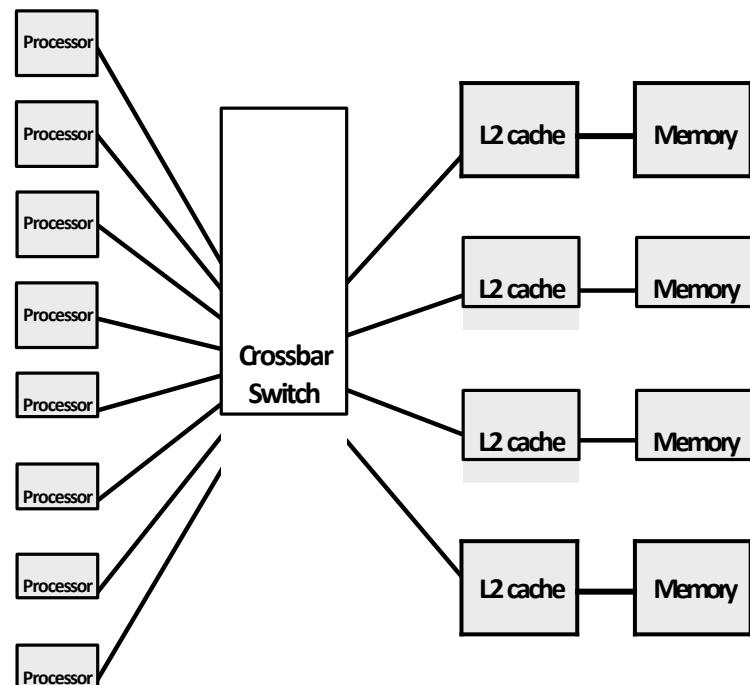


SUN Niagara 2 (UltraSPARCT2)



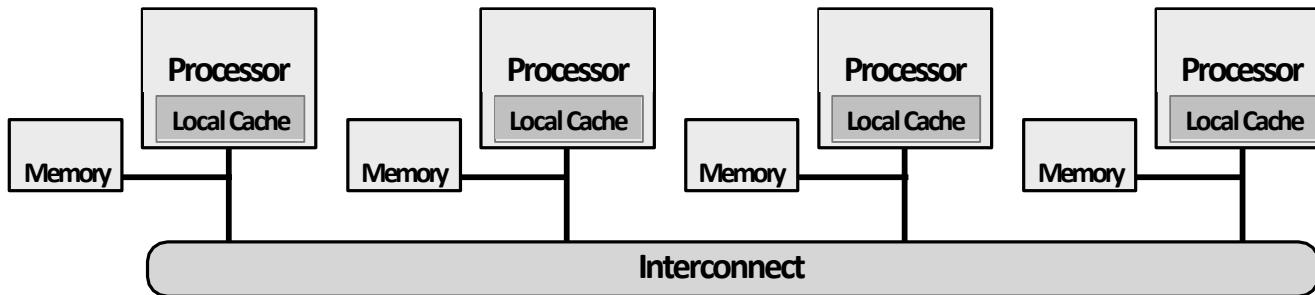
Eightcores

Note area of crossbar: about die area of one core



Non-uniform memory access (NUMA)

All processors can access any memory location, but... the cost of memory access (latency and/or bandwidth) is different for different processors



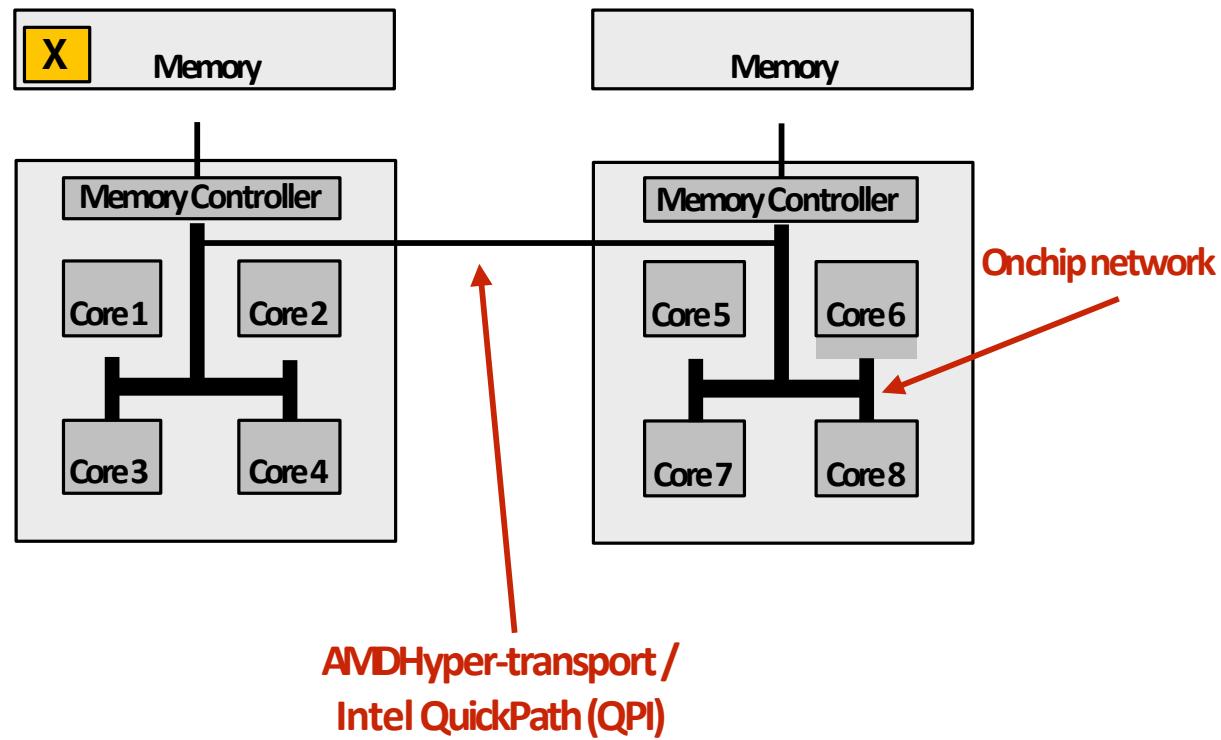
- Problem with preserving uniform access time in a system: **scalability**
 - **GOOD:** costs are uniform, **BAD:** they are **uniformly bad** (memory is uniformly far away)
- **NUMA** designs are **more scalable**
 - **Low latency** and **high bandwidth** to **local memory**
- Cost is increased programmer effort for performance tuning
 - Finding, exploiting locality is important to performance
(want most memory accesses to be to local memories)

Non-uniform memory access (NUMA)

Example: latency to access address X is higher from cores 5-8 than cores 1-4

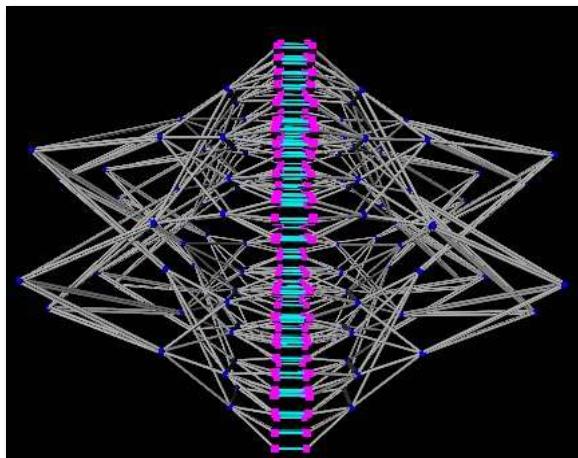


Example: modern dual-socket configuration



SGI Altix UV 1000

- 256 blades, 2 CPUs per blade, 8 cores per CPU = **4096 cores**
- Single shared address space
- Interconnect: fat tree



Fat tree topology



Image credit: Pittsburgh Supercomputing Center

Summary: shared address space model

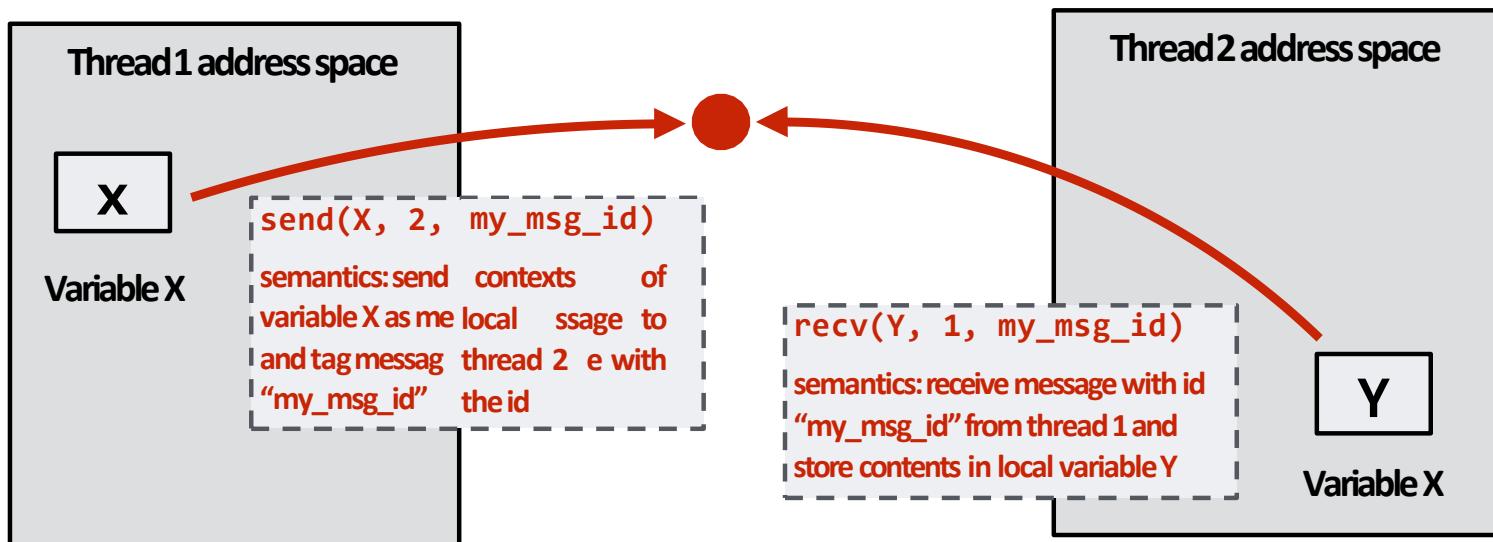
- **Communication abstraction**
 - Threads read/write **shared variables**
 - Threads manipulate **synchronization** primitives: locks, semaphors, etc.
 - Logical extension of uniprocessor programming *
- **Requires hardware support to implementation efficiently**
 - Anyprocessor can load and store from any address (its shared address space!)
 - Even with **NUMA**, costly to scale
(one of the reasons why supercomputers are expensive)

* But NUMA implementation requires reasoning about locality for performance

Message passing model of communication

Message passing model (abstraction)

- Threads operate within their own **private address spaces**
- Threads **communicate by sending/receiving messages**
 - send**: specifies recipient, buffer to be transmitted, and optional message identifier ("tag")
 - receive**: sender, specifies buffer to store data, and optional message identifier
 - Sending messages is **the only way** to exchange data between threads 1 and 2



(Communication operations shown in red)

Illustration adopted from Culler, Singh, Gupta

CSCI465/ECEN433 – Introduction to Parallel Computing

Message passing (implementation)

- Popular software library: **MPI** (message passing interface)
- Hardware need not implement system-wide loads and stores to execute message passing programs (need only be able to communicate messages)
 - Can connect commodity systems together to form large parallel machine (message passing is a programming model for clusters)

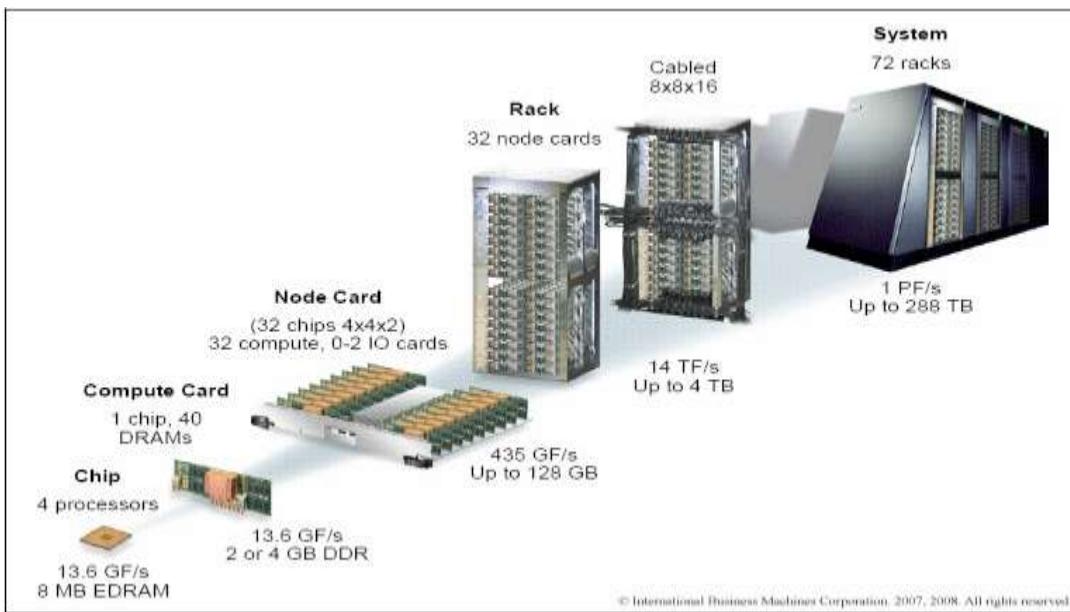
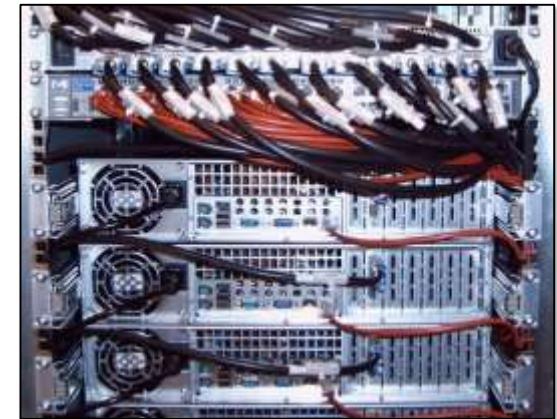


Image credit: IBM

IBM Blue Gene/P Supercomputer



Cluster of workstations
(Infiniband network)

The correspondence between programming models and machine types is fuzzy

- Common to **implement message passing abstractions** on machines that implement a **shared address space in hardware**
 - “Sending message” = copying memory from message library buffers
 - “Receiving message” = copy data from message library buffers
- Can **implement shared address space abstraction** on machines that **do not support it in HW**(via less efficient SW solution)
 - Mark all pages with shared variables as invalid
 - Page-fault handler issues appropriate network requests
- Keep in mind: what is the programming model (abstractions used to specify program)? and what is the HW implementation?

The data-parallel model

Recall: programming models impose structure on programs

- **Shared address space:** very little structure
 - All threads can read and write to all shared variables
 - Pitfall: due to implementation: not all reads and writes have the same cost (and that cost is not apparent in program text)
- **Message passing:** highly structured communication
 - All communication occurs in the form of messages (can read program and see where the communication is)
- **Data-parallel:** very rigid computation structure
 - Programs perform same function on different data elements in a collection

Data-parallel model

- Historically: same operation on each element of an array
 - Matched capabilities SIMD supercomputers of 80's
 - Connection Machine (CM-1, CM-2): thousands of processors, one instruction decode unit
 - Cray supercomputers: vector processors
 - $\text{add}(A, B, n) \leftarrow$ this was one instruction on vectors A, B of length n
- Matlab is another good example: $C = A + B$
(A, B, and C are vectors of same length)
- Today: often takes form of SPMD programming
 - `map(function, collection)`
 - Where `function` is applied to each element of `collection` independently
 - `function` maybe a complicated sequence of logic (e.g., a loop body)
 - Synchronization is implicit at the end of the `map` (`map` returns when function has been applied to all elements of `collection`)

Data parallelism in ISPC

```
// main C++ code:  
const int N = 1024;  
float* x = new float[N];  
float* y = new float[N];  
  
// initialize N elements of x here  
  
absolute_value(N, x, y);
```

Think of **loop body** as **function** (from the previous slide)

foreach construct is a **map**

Given this program, it is reasonable to think of the program as mapping the loop body onto each element of the arrays X and Y.

But if we want to be more precise: the collection is not a first-class ISPC concept. It is implicitly defined by how the program has implemented array indexing logic.

(There is no operation in ISPC with the semantic: “map this code over all elements of this array”)

```
// ISPC code:  
export void absolute_value(  
    uniform int N,  
    uniform float* x,  
    uniform float* y)  
{  
    foreach (i = 0 ... N)  
    {  
        if (x[i] < 0)  
            y[i] = -x[i];  
        else  
            y[i] = x[i];  
    }  
}
```

Data parallelism in ISPC

```
// main C++ code:  
const int N = 1024;  
float* x = new float[N/2];  
float* y = new float[N];  
  
// initialize N/2 elements of x here  
  
absolute_repeat(N/2, x, y);
```

Think of loop body as function

foreach construct is a map

Collection is implicitly defined by array indexing logic

```
// ISPC code:  
export void absolute_repeat(  
    uniform int N,  
    uniform float* x,  
    uniform float* y)  
{  
    foreach (i = 0 ... N)  
    {  
        if (x[i] < 0)  
            y[2*i] = -x[i];  
        else  
            y[2*i] = x[i];  
            y[2*i+1] = y[2*i];  
    }  
}
```

This is also a valid ISPC program!

It takes the absolute value of elements of x, then repeats it twice in the output array

(Less obvious how to think of this code as mapping the loop body onto existing collections.)

Data parallelism in ISPC

```
// main C++ code:  
const int N = 1024;  
float* x = new float[N];  
float* y = new float[N];  
  
// initialize N elements of x  
  
shift_negative(N, x, y);
```

Think of loop body as function

foreach construct is a map

Collection is implicitly defined by array indexing logic

```
// ISPC code:  
export void shift_negative(  
    uniform int N,  
    uniform float* x,  
    uniform float* y)  
{  
    foreach (i = 0 ... N)  
    {  
        if (i >= 1 && x[i] < 0)  
            y[i-1] = x[i];  
        else  
            y[i] = x[i];  
    }  
}
```

This program is non-deterministic!

Possible for multiple iterations of the loop body to write to same memory location

Data-parallel model (foreach) provides no specification of order in which iterations occur

Model provides no primitives for fine-grained mutual exclusion/synchronization). It is not intended to help programmers write programs with that structure

Data parallelism: a more “proper” way

Note: this is not ISPC syntax (more of our made-up syntax)

Main program:

```
const int N = 1024;

stream<float> x(N); // define collection
stream<float> y(N); // define collection

// initialize N elements of x here

// map function absolute_value onto
// streams (collections) x, y
absolute_value(x, y);
```

Data-parallelism expressed in this functional form is sometimes referred to as the [stream programming model](#)

Streams: collections of elements. Elements can be processed independently

Kernels: side-effect-free functions. Operate element-wise on collections

Think of kernel inputs, outputs, temporaries for each invocation as a private address space

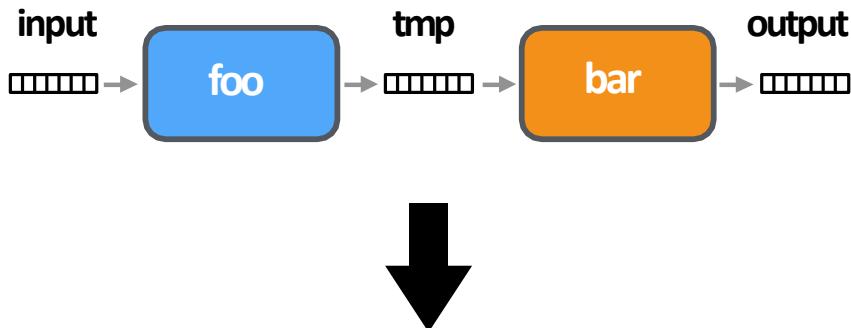
“Kernel” definition:

```
void absolute_value(float x, float y)
{
    if (x < 0)
        y = -x;
    else
        y = x;
}
```

Stream programming benefits

```
const int N = 1024;
stream<float> input(N);
stream<float> output(N);
stream<float> tmp(N);

foo(input, tmp);
bar(tmp, output);
```



```
parallel_for(int i=0; i<N; i++)
{
    output[i] = bar(foo(input[i]));
}
```

Functions really are side-effect free!
(cannot write a non-deterministic program)

Program data flow is known by compiler:

Inputs and outputs of each invocation are known in advance: prefetching can be employed to hide latency.

Producer-consumer locality is known in advance:
Implementation can be structured so outputs of first kernel are immediately processed by second kernel.
(The values are stored in on-chip buffers/caches and never written to memory! Saves bandwidth!)

These optimizations are responsibility of stream program compiler. Requires global program analysis.

Stream programming drawbacks

```
const int N = 1024;
stream<float> input(N/2);
stream<float> tmp(N);
stream<float> output(N);

// double length of stream by replicating
// all elements 2x
stream_repeat(2, input, tmp);

absolute_value(tmp, output);
```

In our experience:

This is the achilles heel of all “proper”
data-parallel/stream programming
systems.

“If I just had one more operator”...

Need library of operators to describe complex data flows (see use of repeat operator at left to obtain same behavior as indexing code below)

Our experience: cross fingers and hope compiler is intelligent enough to generate code below from program at left.

```
// ISPC code:
export void absolute_repeat(
    uniform int N,
    uniform float* x,
    uniform float* y)
{
    foreach (i = 0 ... N)
    {
        float result;
        if (x[i] < 0)
            result = -x[i];
        else
            result = x[i];
        y[2*i+1] = y[2*i] = result;
    }
}
```

Gather/scatter: two key data-parallel communication primitives

Map absolute_value onto stream produced by gather:

```
const int N = 1024;
stream<float> input(N);
stream<int> indices;
stream<float> tmp_input(N);
stream<float> output(N);

stream_gather(input, indices, tmp_input);
absolute_value(tmp_input, output);
```

Map absolute_value onto stream, scatter results:

```
const int N = 1024;
stream<float> input(N);
stream<int> indices;
stream<float> tmp_output(N);
stream<float> output(N);

absolute_value(input, tmp_output);
stream_scatter(tmp_output, indices, output);
```

ISPC equivalent:

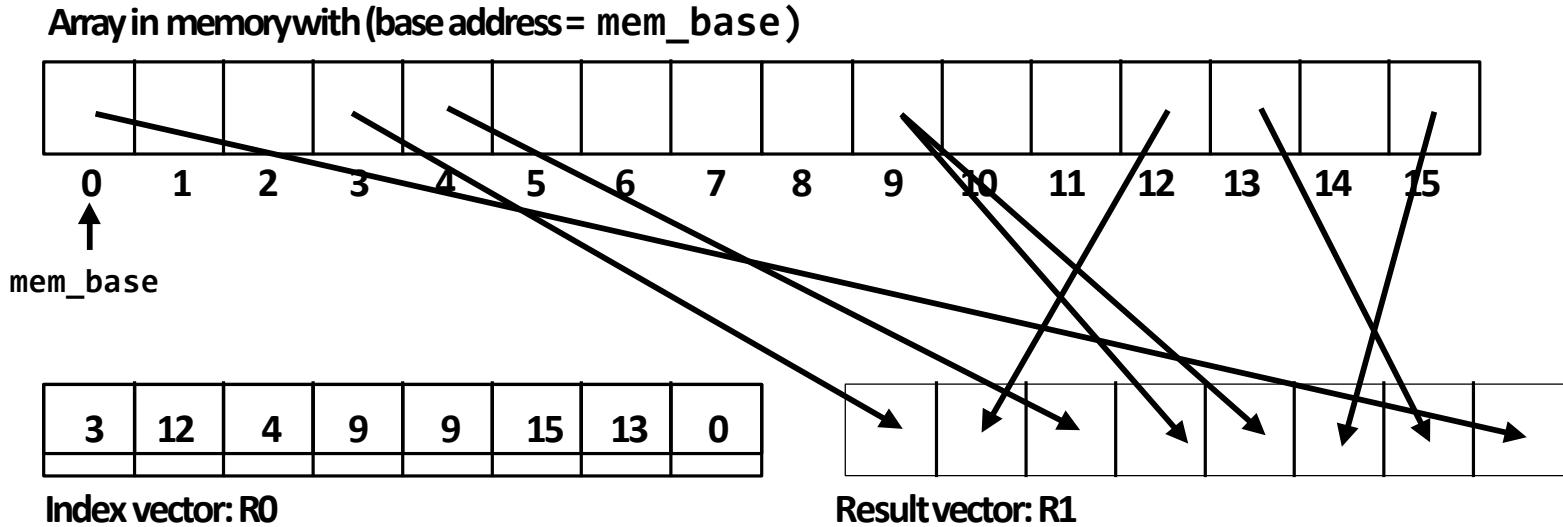
```
export void absolute_value(
    uniform float N,
    uniform float* input,
    uniform float* output,
    uniform int* indices)
{
    foreach (i = 0 ... n)
    {
        float tmp = input[indices[i]];
        if (tmp < 0)
            output[i] = -tmp;
        else
            output[i] = tmp;
    }
}
```

ISPC equivalent:

```
export void absolute_value(
    uniform float N,
    uniform float* input,
    uniform float* output,
    uniform int* indices)
{
    foreach (i = 0 ... n)
    {
        if (input[i] < 0)
            output[indices[i]] = -input[i];
        else
            output[indices[i]] = input[i];
    }
}
```

Gather instruction

gather(R1, R0, mem_base); “Gather from buffer `mem_base` into `R1` according to indices specified by `R0`.”



Gather supported with AVX2 in 2013

But AVX2 does not support SIMD scatter (must implement as scalar loop)

Scatter instruction exists in AVX512

Hardware supported gather/scatter does exist on GPUs.
(still an expensive operation compared to load/store of contiguous vector)

Summary: data-parallel model

- Data-parallelism is about **imposing rigid program structure** to facilitate simple programming and advanced optimizations
- Basic structure: **map a function onto a large collection of data**
 - Functional: side-effect free execution
 - No communication among distinct function invocations
(allow invocations to be scheduled in any order, including in parallel)
- In practice that's how many simple programs work
- But... many modern performance-oriented data-parallel languages do not strictly enforce this structure
 - ISPC, OpenCL, CUDA, etc.
 - They choose flexibility/familiarity of imperative C-style syntax over the safety of a more functional form: it's been their key to their adoption
 - Opinion: sure, functional thinking is great, but programming systems sure should impose structure to facilitate achieving high-performance implementations, not hinder them

Three parallel programming models

- **Shared address space**
 - Communication is unstructured, implicit in loads and stores
 - Natural way of programming, but can shoot yourself in the foot easily
 - Program might be correct, but not perform well
- **Message passing**
 - Structure all communication as messages
 - Often harder to get first correct program than shared address space
 - Structure often helpful in getting to first correct, scalable program
- **Data parallel**
 - Structure computation as a big “map” over a collection
 - Assumes a shared address space from which to load inputs/store results, but model severely limits communication between iterations of the map
(goal: preserve independent processing of iterations)
 - Modern embodiments encourage, but don’t enforce, this structure

Modern practice: mixed programming models

- Use **shared address space** programming **within a multi-core node** of a cluster, use **message passing** between nodes
 - Very, very common in practice
 - Use convenience of shared address space where it can be implemented efficiently (within a node), require explicit communication elsewhere
- Data-parallel-ish programming models support shared-memory style synchronization primitives in kernels
 - Permit limited forms of inter-iteration communication (e.g., CUDA, OpenCL)

Summary

- Programming models provide a way to think about the organization of parallel programs. They provide **abstractions** that admit many possible **implementations**.
- Restrictions imposed by these abstractions are designed to reflect realities of parallelization and communication costs
 - Shared address space machines: hardware supports any processor accessing any address
 - Messaging passing machines: may have hardware to accelerate message send/receive/buffering
 - It is desirable to keep “abstraction distance” low so programs have predictable performance, but want it high enough for code flexibility/portability
- In practice, **you’ll need to be able to think in a variety of ways**
 - Modern machines provide different types of communication at different scales
 - Different models fit the machine best at the various scales
 - Optimization may require you to think about implementations, not just abstractions

Questions

CSCI465/ECEN433

Introduction to Parallel Computing

Spring 2024



Lecture (6)

Parallel Programming

Basics



Review: 3 parallel programming models

- **Shared address space**
 - Communication is unstructured, implicit in loads and stores
 - Natural way of programming, but can shoot yourself in the foot easily
 - Program might be correct, but not perform well
- **Message passing**
 - Structure all communication as messages
 - Often harder to get first correct program than shared address space
 - Structure often helpful in getting to first correct, scalable program
- **Data parallel**
 - Structure computation as a big “map” over a collection
 - Assumes a shared address space from which to load inputs/store results, but model severely limits communication between iterations of the map
(goal: preserve independent processing of iterations)
 - Modern embodiments encourage, but don’t enforce, this structure

Modern practice: mixed programming models

- Use **shared address space** programming **within a multi-core node** of a cluster, use **message passing** between nodes
 - Very, very common in practice
 - Use convenience of shared address space where it can be implemented efficiently (within a node), require explicit communication elsewhere
- Data-parallel-ish programming models support shared-memory style synchronization primitives in kernels
 - Permit limited forms of inter-iteration communication (e.g., CUDA, OpenCL)

Creating a parallel program

Thought process:

1. Identify work that can be performed in parallel
2. Partition work (and also data associated with the work)
3. Manage data access, communication, and synchronization

Recall one of our main goals is speedup *

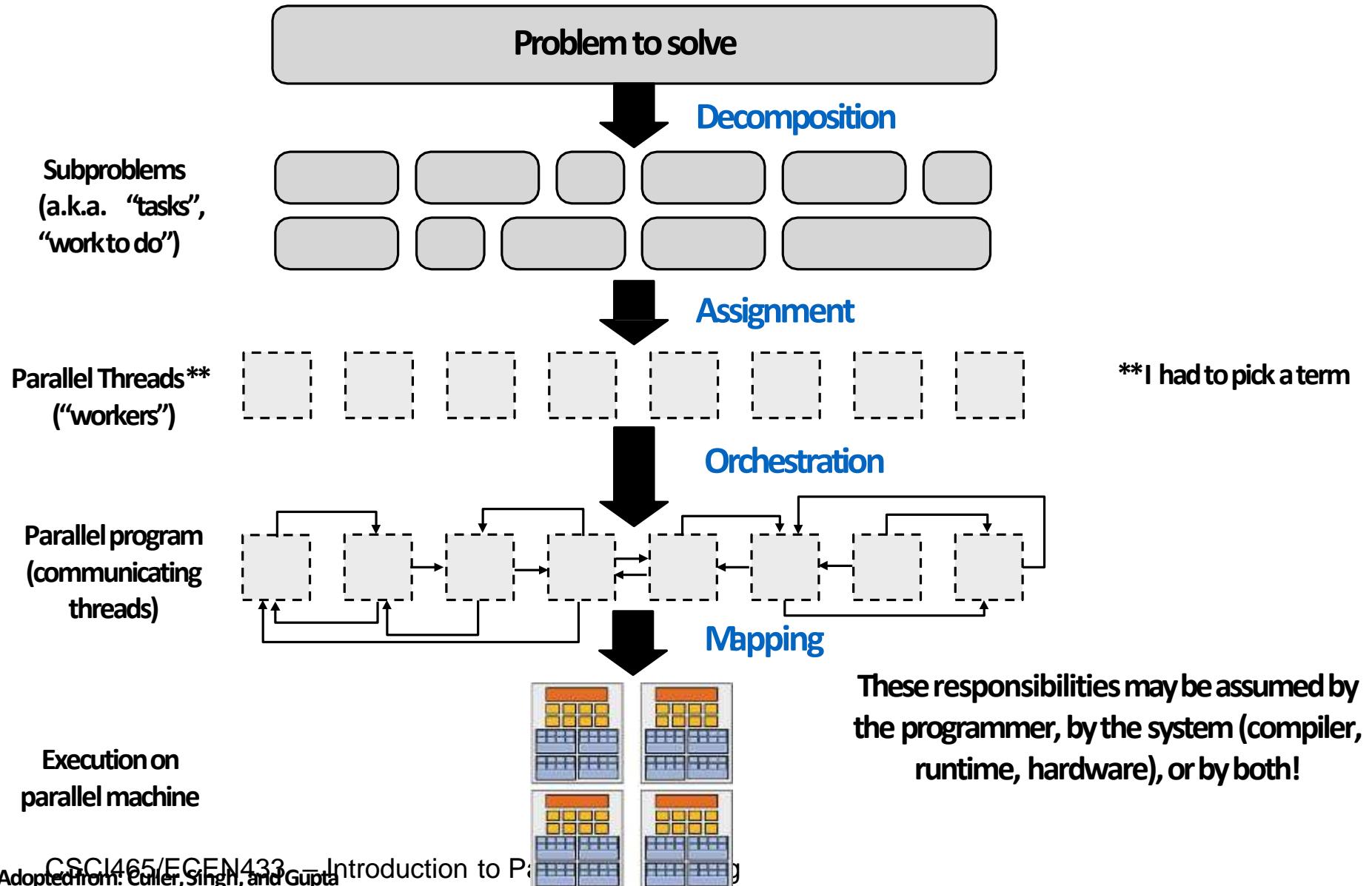
For a fixed computation:

$$\text{Speedup(P processors)} = \frac{\text{Time (1 processor)}}{\text{Time (P processors)}}$$

* Other goals include high efficiency (cost, area, power, etc.)

CSCI465/EGEN433 - Introduction to Parallel Computing
or working on bigger problems than can fit on one machine

Creating a parallel program



Decomposition

Break up problem into **tasks** that can be carried out in parallel

- Decomposition need not happen statically
- New tasks can be identified as program executes

Main idea: create *at least* enough tasks to keep all execution units on a machine busy

Key aspect of decomposition: identifying dependencies
(or... a lack of dependencies)

Amdahl's Law: dependencies limit maximum speedup due to parallelism

You run your favorite sequential program...

Let S = the fraction of sequential execution that is inherently sequential (dependencies prevent parallel execution)

Then maximum speedup due to parallel execution $\leq 1/S$

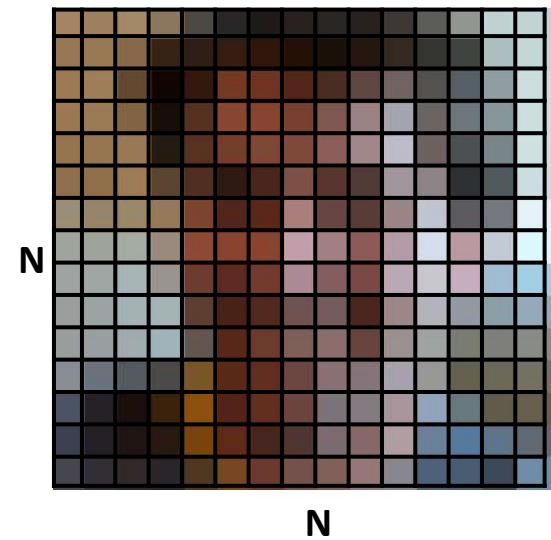
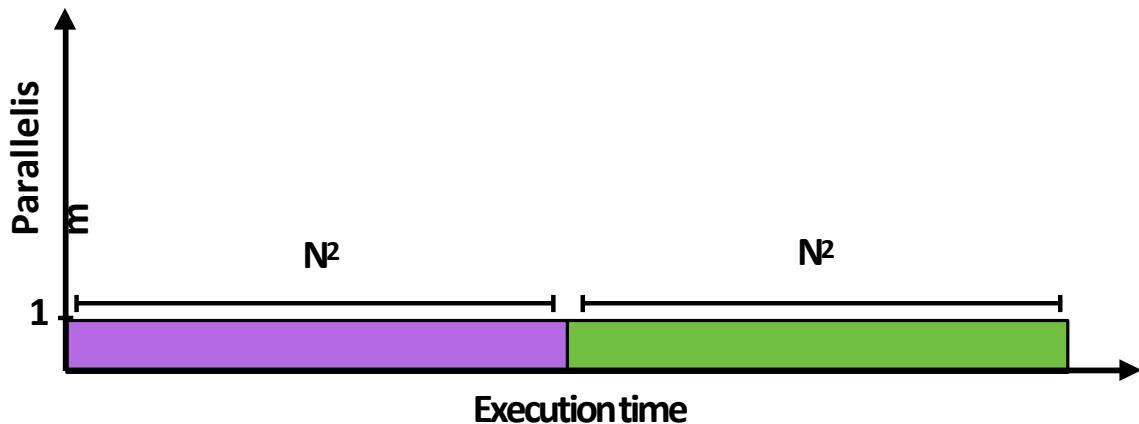
A simple example

Consider a two-step computation on a $N \times N$ image

- Step 1: double brightness of all pixels
(independent computation on each grid element)
- Step 2: compute average of all pixel values

Sequential implementation of program

- Both steps take $\sim N^2$ time, so total time is $\sim 2N^2$



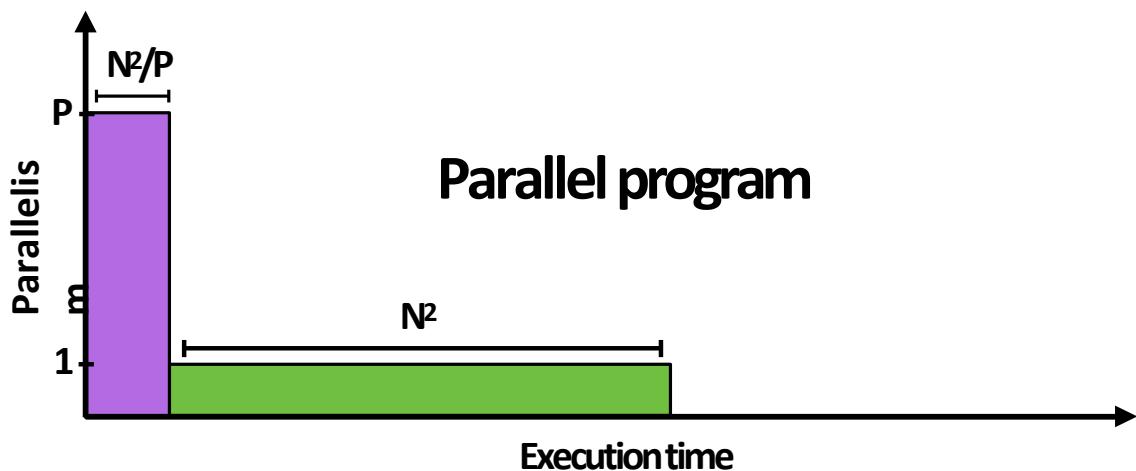
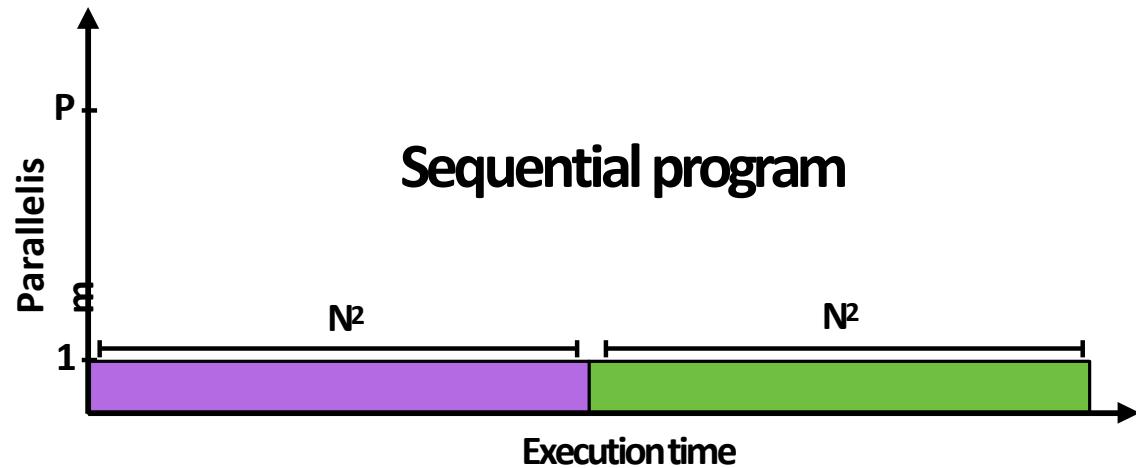
First attempt at parallelism (P processors)

Strategy:

- Step 1: execute in parallel
 - time for phase 1: N^2/P
 - Step 2: execute serially
 - time for phase 2: N^2
-
- Overall performance:

$$\text{Speedup} \leq \frac{2n^2}{\frac{n^2}{p} + n^2}$$

Speedup ≤ 2



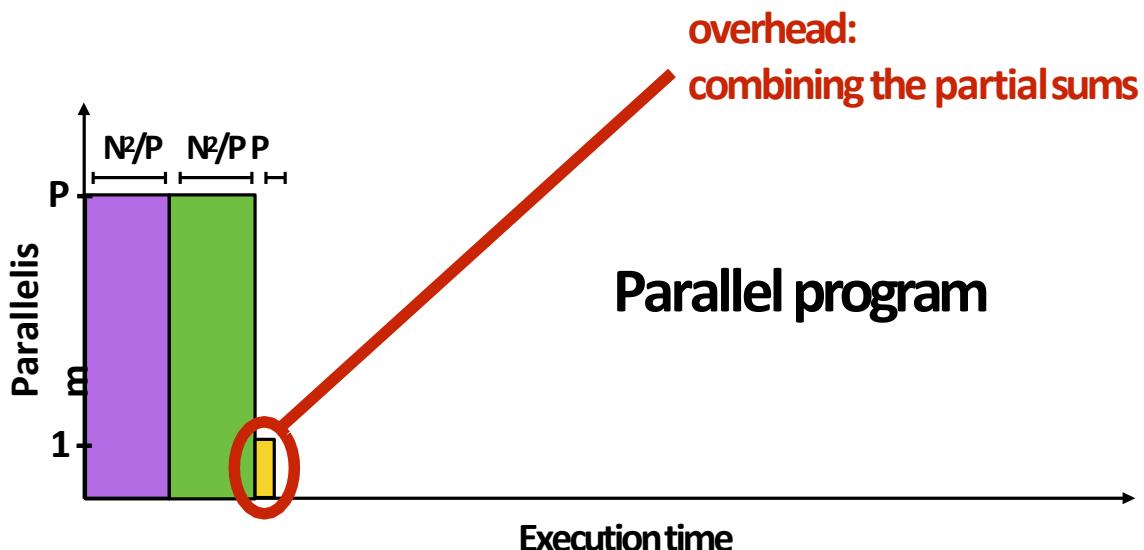
Parallelizing step 2

Strategy:

- Step 1: execute in parallel
 - time for phase 1: N^2/P
- Step 2: compute **partial sums in parallel**, combine results serially
 - time for phase 2: $N^2/P + P$

Overall performance:

$$\text{Speedup} \leq \frac{\frac{2n^2}{2n^2 + p}}{p}$$



Note:

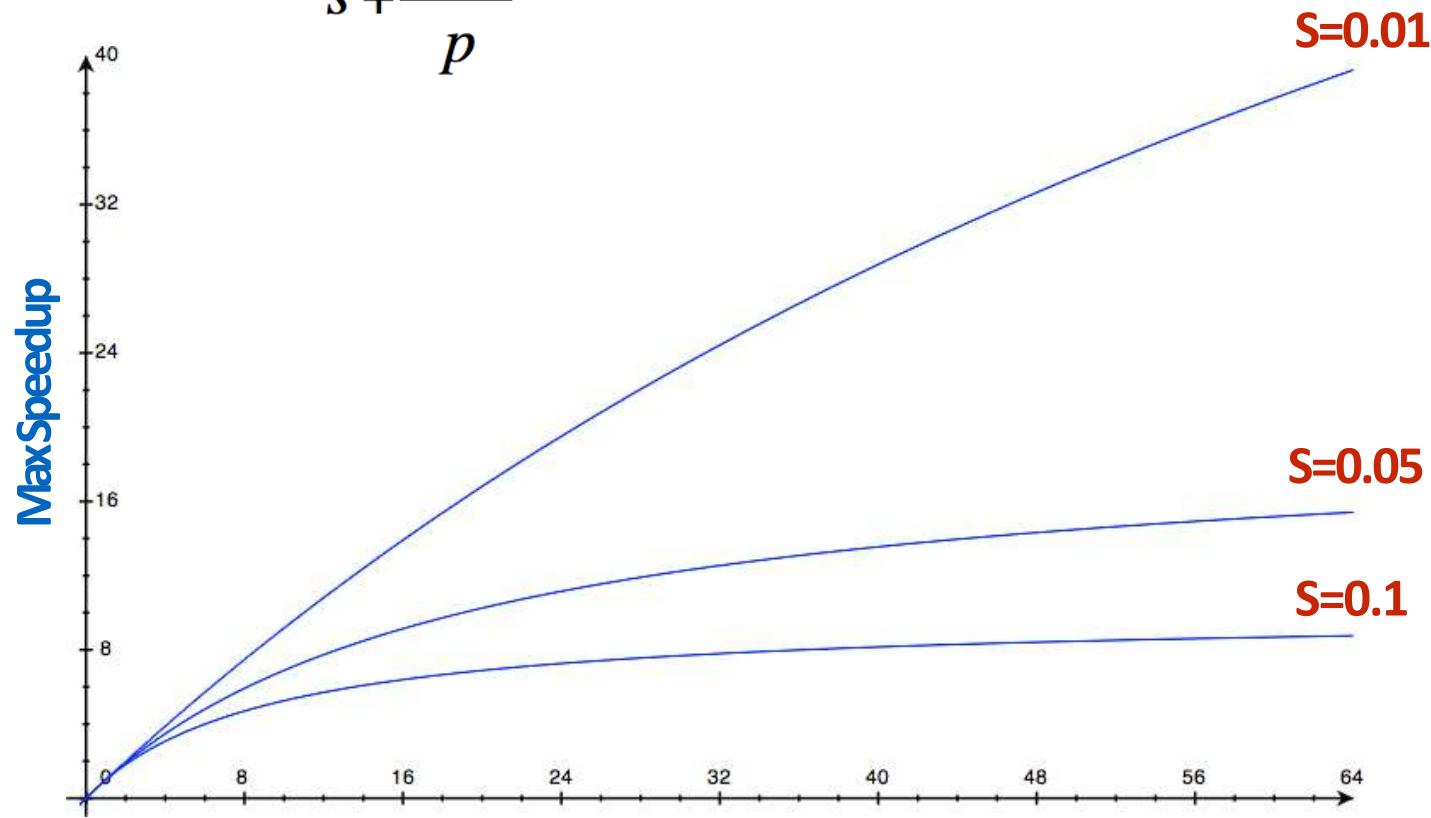
speedup → P when $N >> P$

Amdahl's law

Let S = the fraction of total work that is inherently sequential

Maxspeedup on P processors given by:

$$\text{speedup} \leq \frac{1}{s + \frac{1-s}{p}}$$



Decomposition

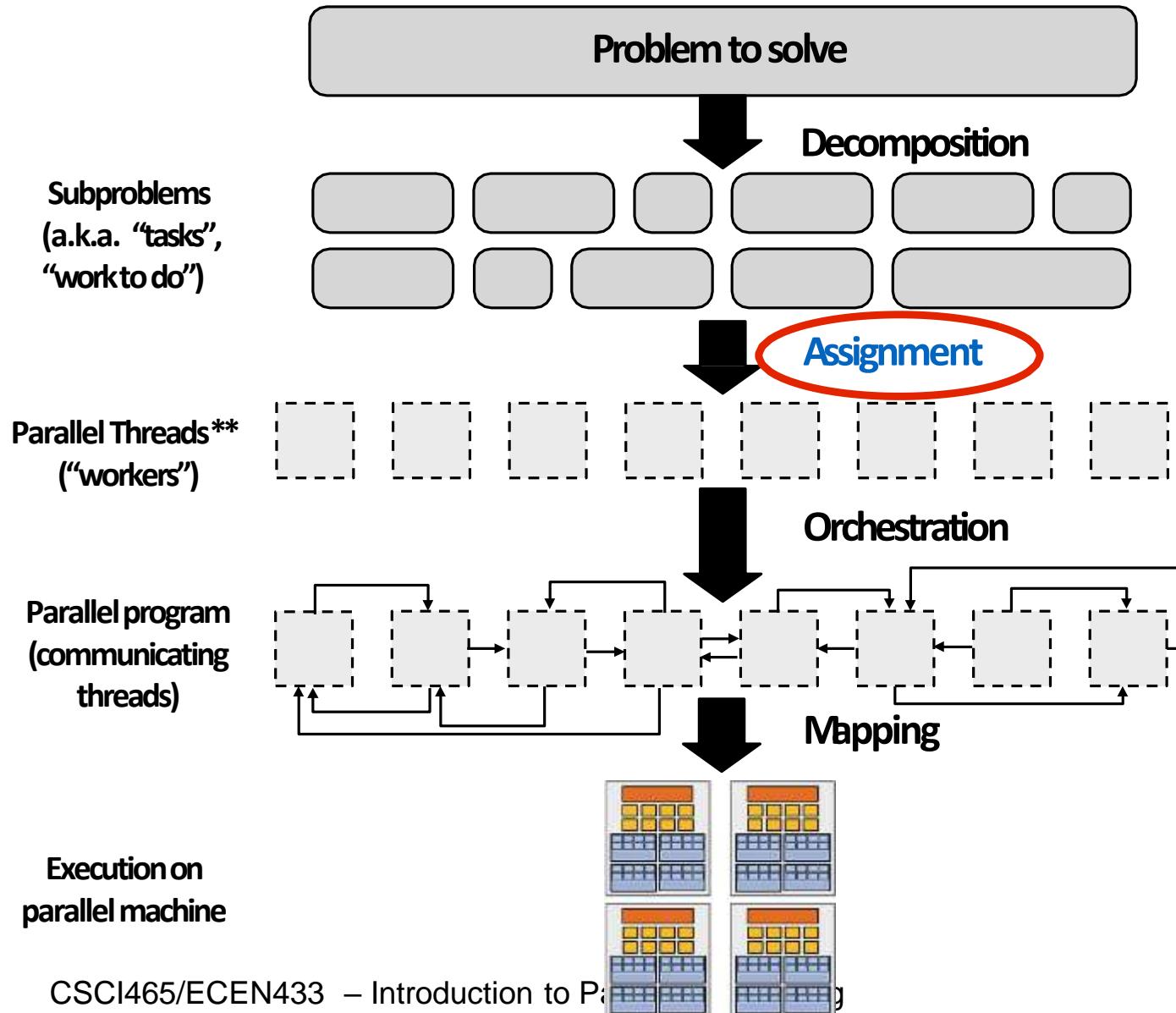
Who is responsible for performing decomposition?

- In most cases: the programmer

Automatic decomposition of sequential programs continues to be a challenging research problem
(very difficult in general case)

- Compiler must analyze program, identify dependencies
 - What if dependencies are data dependent (not known at compile time)?
- Researchers have had modest success with simple loop nests
- The “magic parallelizing compiler” for complex, general-purpose code has not yet been achieved

Assignment



Assignment

Assigning tasks to threads **

**I had to pick a term
(will explain in a second)

- Think of “tasks” as things to do
- Think of threads as “workers”

Goals: balance workload, reduce communication costs

Can be performed statically, or dynamically during execution

While programmer often responsible for decomposition, many languages/runtimes take responsibility for assignment.

Assignment examples in ISPC

```
export void sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    // assumes N % programCount = 0
    for (uniform int i=0; i<N; i+=programCount)
    {
        int idx = i + programIndex;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6; // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[idx] * x[idx];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[i] = value;
    }
}
```

Decomposition of work by loop iteration

Programmer-managed assignment:

Static assignment

Assign iterations to ISPC program instances in

interleaved fashion – Introduction to Parallel Computing

```
export void sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    foreach (i = 0 ... N)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        uniform int denom = 6; // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[i] = value;
    }
}
```

Decomposition of work by loop iteration

foreach construct exposes independent work to system

System-manages assignment of iterations (work) to ISPC program instances (abstraction leaves room for dynamic assignment, but current ISPC implementation is static)

Static assignment example using pthreads

```
typedef struct {
    int N, terms;
    float* x, *result;
} my_args;

void parallel_sinx(int N, int terms, float* x, float* result)
{
    pthread_t thread_id;
    my_args args;

    args.N = N/2;
    args.terms = terms;
    args.x = x;
    args.result = result;

    // launch second thread, do work on first half of array
    pthread_create(&thread_id, NULL, my_thread_start, &args);

    // do work on second half of array in main thread
    sinx(N - args.N, terms, x + args.N, result + args.N);

    pthread_join(thread_id, NULL);
}

void my_thread_start(void* thread_arg)
{
    my_args* thread_args = (my_args*)thread_arg;
    sinx(args->N, args->terms, args->x, args->result); // do work
}
```

Decomposition of work by loop iteration

Programmer-managed assignment:

Static assignment

Assign iterations to pthreads in **blocked** fashion
(first half of array to spawned thread, second
half to main thread)

Dynamic assignment using ISPC tasks

```
void foo(uniform float* input,
          uniform float* output,
          uniform int N)
{
    // create a bunch of tasks
    launch[100] my_ispc_task(input, output, N);
}
```

ISPC runtime assign tasks to worker threads

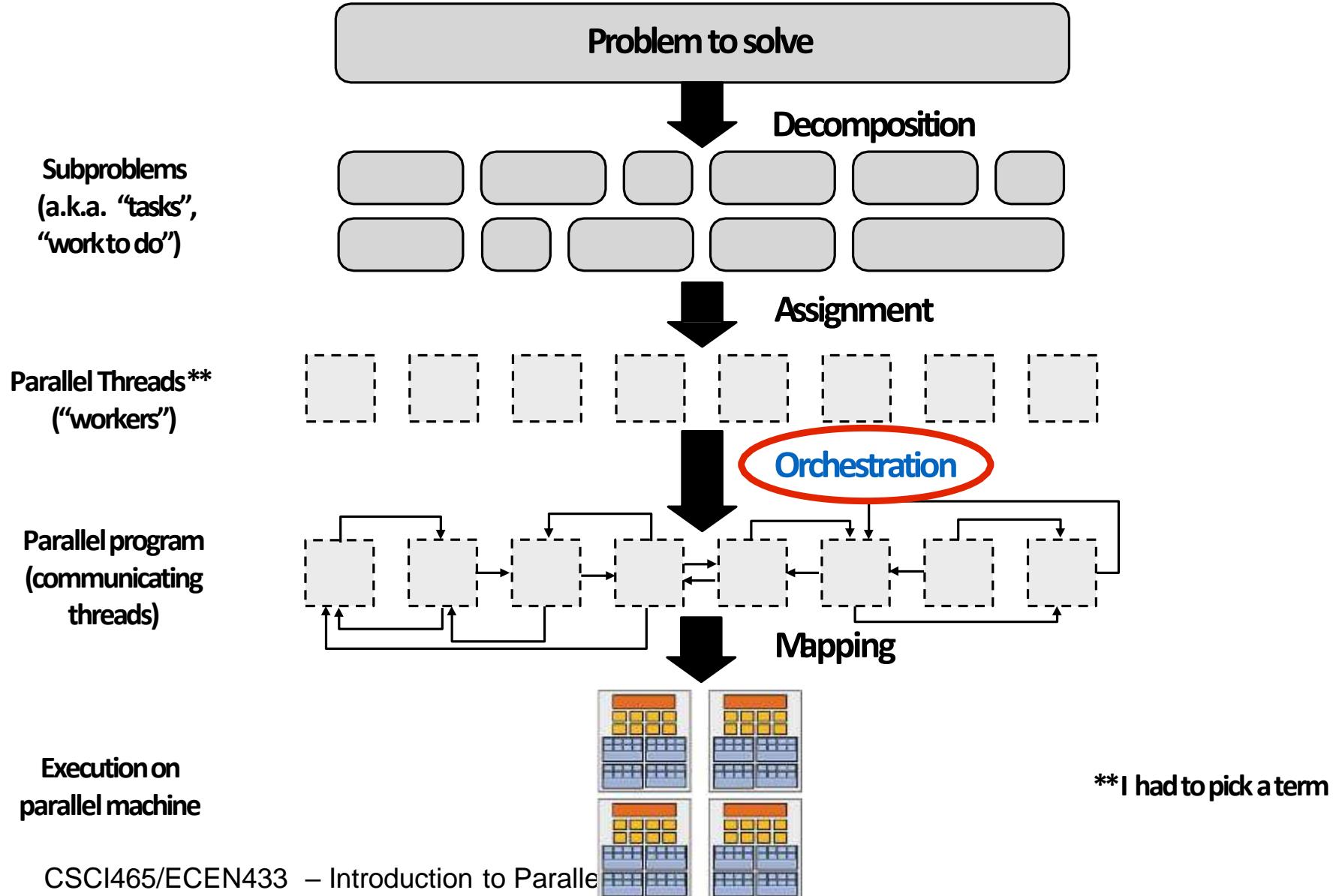
List of tasks:



Assignment policy: after completing current task, worker thread inspects list and assigns itself the next uncompleted task.



Orchestration



Orchestration

Involves:

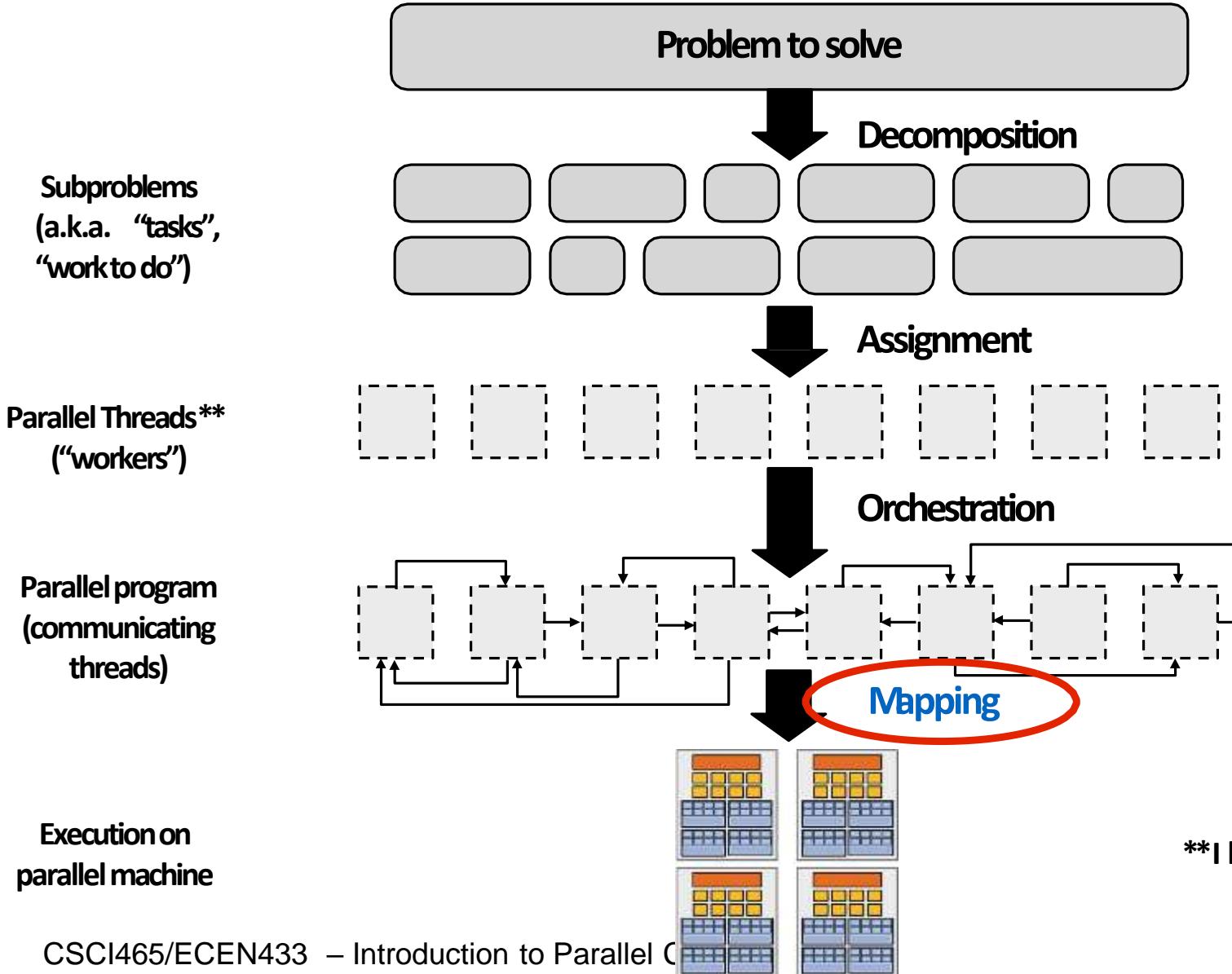
- Structuring communication
- Adding synchronization to preserve dependencies if necessary
- Organizing data structures in memory
- Scheduling tasks

Goals: reduce costs of communication/sync, preserve locality of data reference, reduce overhead, etc.

Machine details impact many of these decisions

- If synchronization is expensive, might use it more sparsely

Mapping to hardware



Mapping to hardware

Mapping “threads” (“workers”) to hardware execution units

Example 1: mapping by the **operating system**

- e.g., map pthread to HW execution context on a CPU core

Example 2: mapping by the **compiler**

- Map ISPC program instances to vector instruction lanes

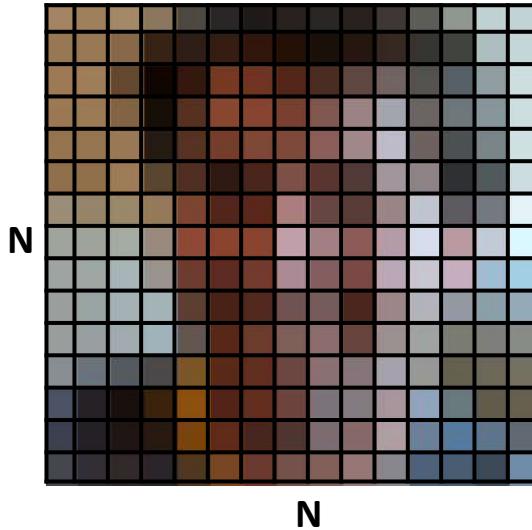
Example 3: mapping by the **hardware**

- Map CUDA thread blocks to GPU cores

Some interesting mapping decisions:

- Place **related threads** (cooperating threads) on the same processor (maximize locality, data sharing, minimize costs of comm/sync)
- Place **unrelated threads** on the same processor (one might be bandwidth limited and another might be compute limited) to use machine more efficiently

Decomposing computation or data?



Often, the reason a problem requires lots of computation (and needs to be parallelized) is that it involves manipulating a lot of data.

I've described the process of parallelizing programs as an act of partitioning computation.

Often, it's equally valid to think of partitioning data. (computations go with the data)

But there are many computations where the correspondence between work-to-do ("tasks") and data is less clear. In these cases it's natural to think of partitioning computation.

Questions

CSCI465/ECEN433

Introduction to Parallel Computing

Spring 2024



Lecture (7)

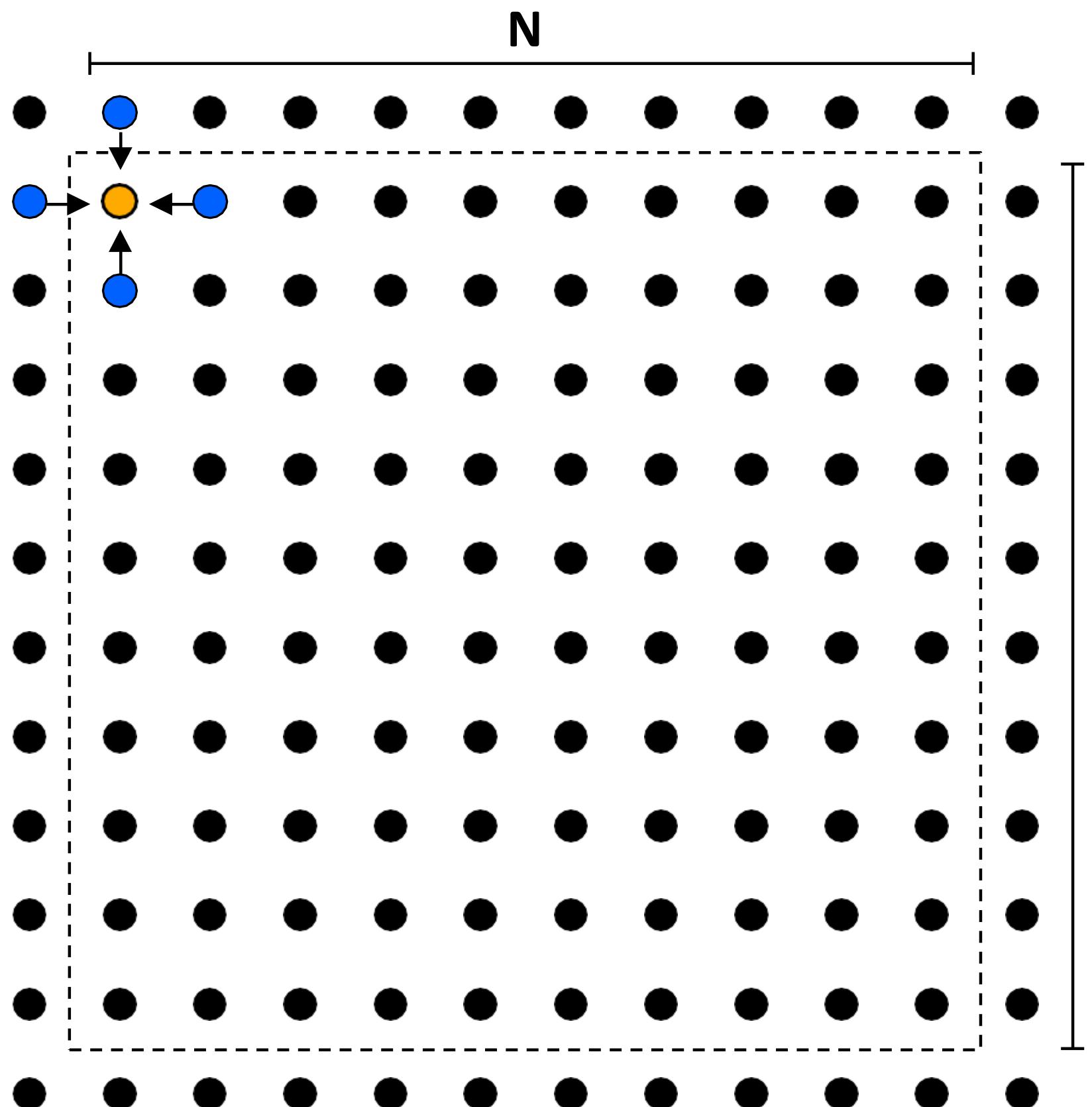
Performance Optimization

Review: A 2D-grid based solver

Solve partial differential equation (PDE) on $(N+2) \times (N+2)$ grid

Iterative solution

- Perform Gauss-Seidel sweeps over grid until convergence



$$A[i, j] = 0.2 * (A[i, j] + A[i, j-1] + A[i-1, j] \\ + A[i, j+1] + A[i+1, j]);$$

Grid solver algorithm

C-like pseudocode for sequential algorithm is provided below

```
const int n;
float* A;                                // assume allocated to grid of N+2 x N+2 elements

void solve(float* A) {

    float diff, prev;
    bool done = false;

    while (!done) {                           // outermost loop: iterations
        diff = 0.f;
        for (int i=1; i<=n; i++) {
            for (int j=1; j<=n; j++) {
                prev = A[i,j];
                A[i,j] = 0.2f * (A[i,j] + A[i,j-1] + A[i-1,j] +
                                     A[i,j+1] + A[i+1,j]);
                diff += fabs(A[i,j] - prev); // compute amount of change
            }
        }

        if (diff/(n*n) < TOLERANCE)          // quit if converged
            done = true;
    }
}
```

Grid solver algorithm

C-like pseudocode for sequential algorithm is provided below

```
const int n;
float* A;                                // assume allocated to grid of N+2 x N+2 elements

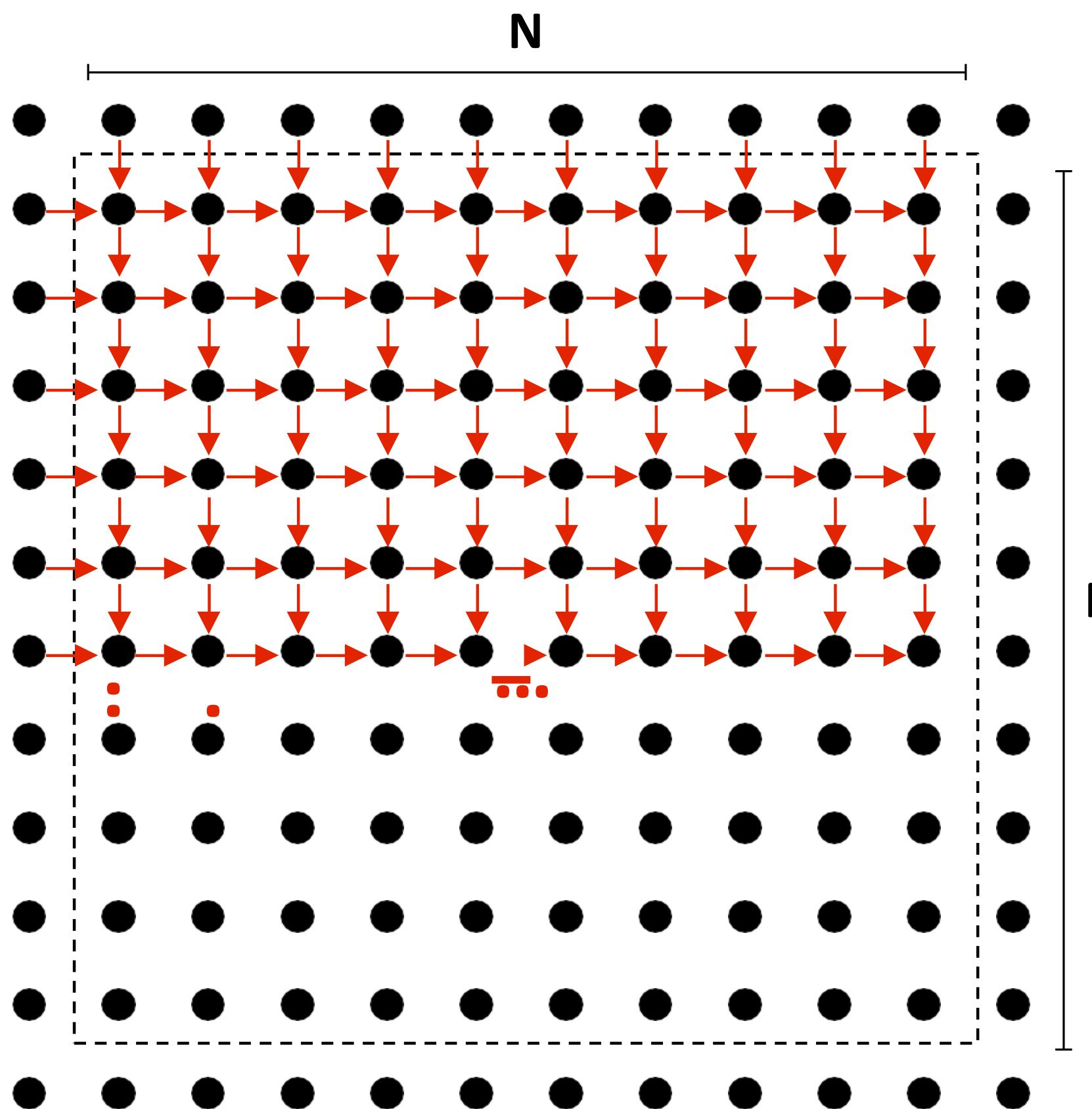
void solve(float* A) {

    float diff, prev;
    bool done = false;

    while (!done) {                           // outermost loop: iterations
        diff = 0.f;
        for (int i=1; i<=n; i++) {
            for (int j=1; j<=n; j++) {
                prev = A[i,j];
                A[i,j] = 0.2f * (A[i,j] + A[i,j-1] + A[i-1,j] +
                                     A[i,j+1] + A[i+1,j]);
                diff += fabs(A[i,j] - prev); // compute amount of change
            }
        }

        if (diff/(n*n) < TOLERANCE)          // quit if converged
            done = true;
    }
}
```

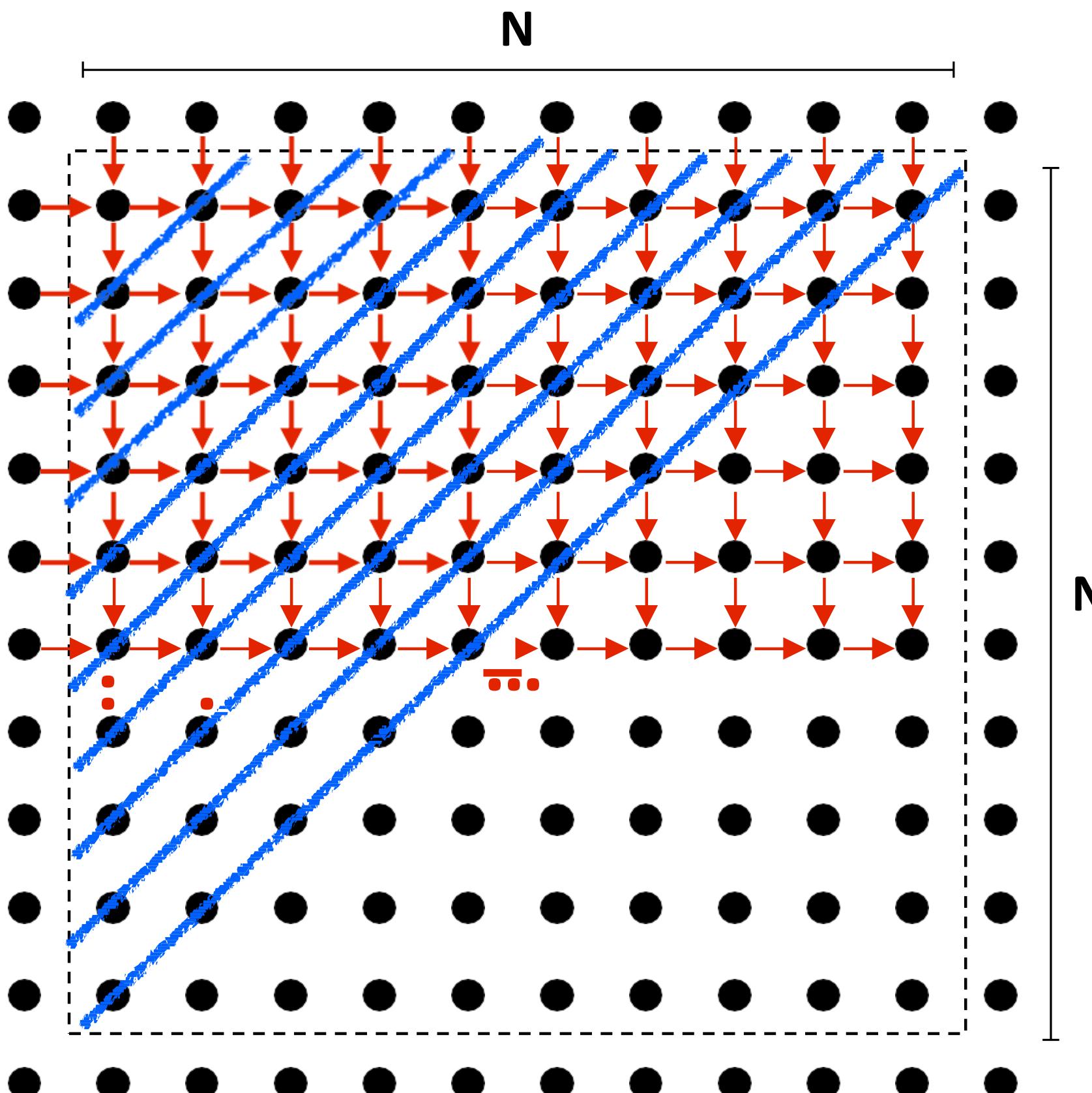
Step 1: identify dependencies (problem decomposition phase)



Each row element depends on element to left.

Each column depends on previous column.

Step 1: identify dependencies (problem decomposition phase)



There is independent work along the diagonals!

Good: parallelism exists!

Possible implementation strategy:

1. Partition grid cells on a diagonal into tasks
2. Update values in parallel
3. When complete, move to next diagonal

Bad: independent work is hard to exploit

Not much parallelism at beginning and end of computation.

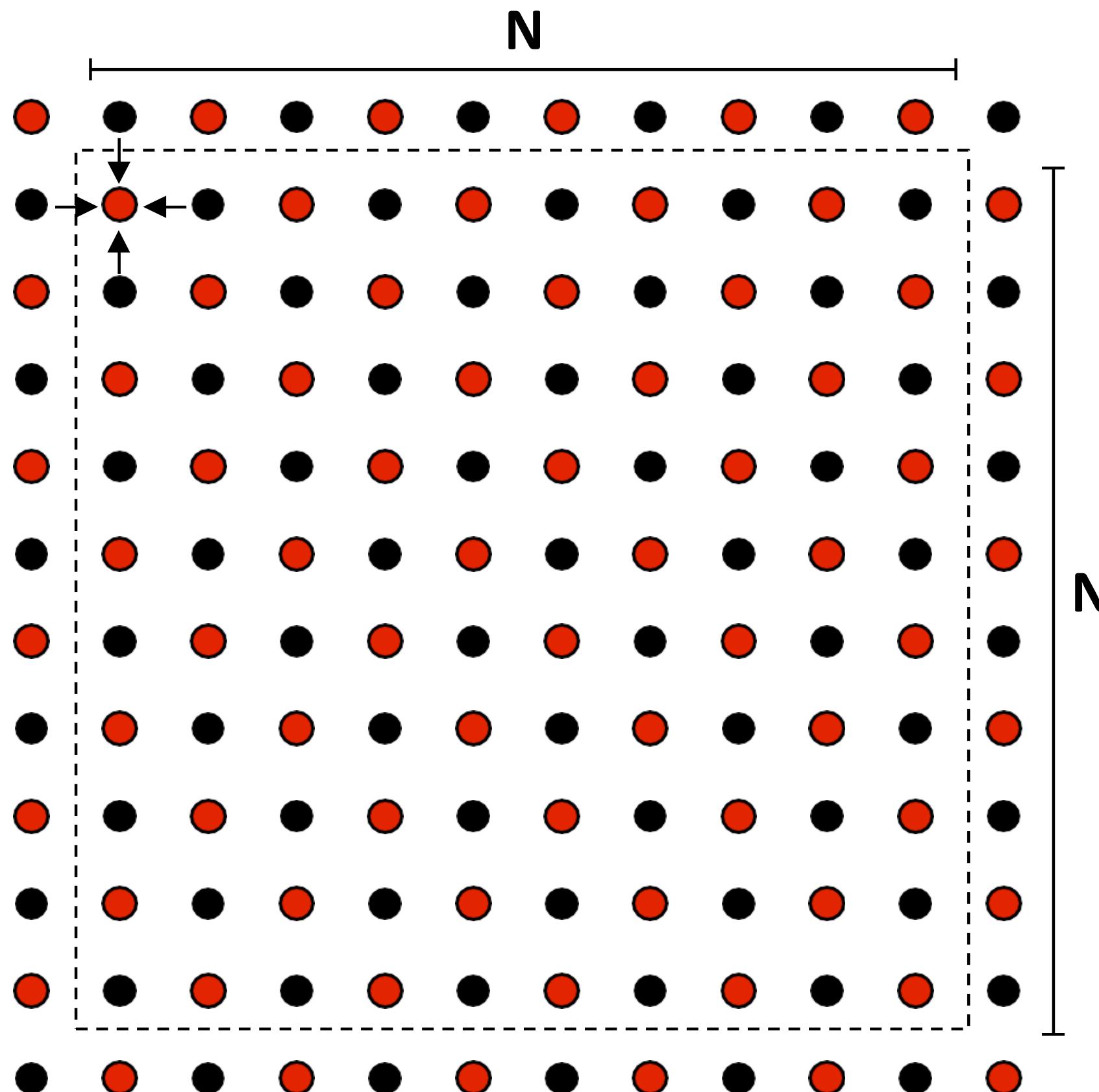
Frequent synchronization (after completing each diagonal)

Let's make life easier on ourselves

Idea: improve performance by changing the algorithm to one that is more amenable to parallelism

- Change the order grid cell cells are updated
- New algorithm iterates to same solution (approximately), but converges to solution differently
 - Note: floating-point values computed are different, but solution still converges to within error threshold
- Yes, we needed domain knowledge of Gauss-Seidel method for solving a linear system to realize this change is permissible for the application

New approach: reorder grid cell update via red-black coloring

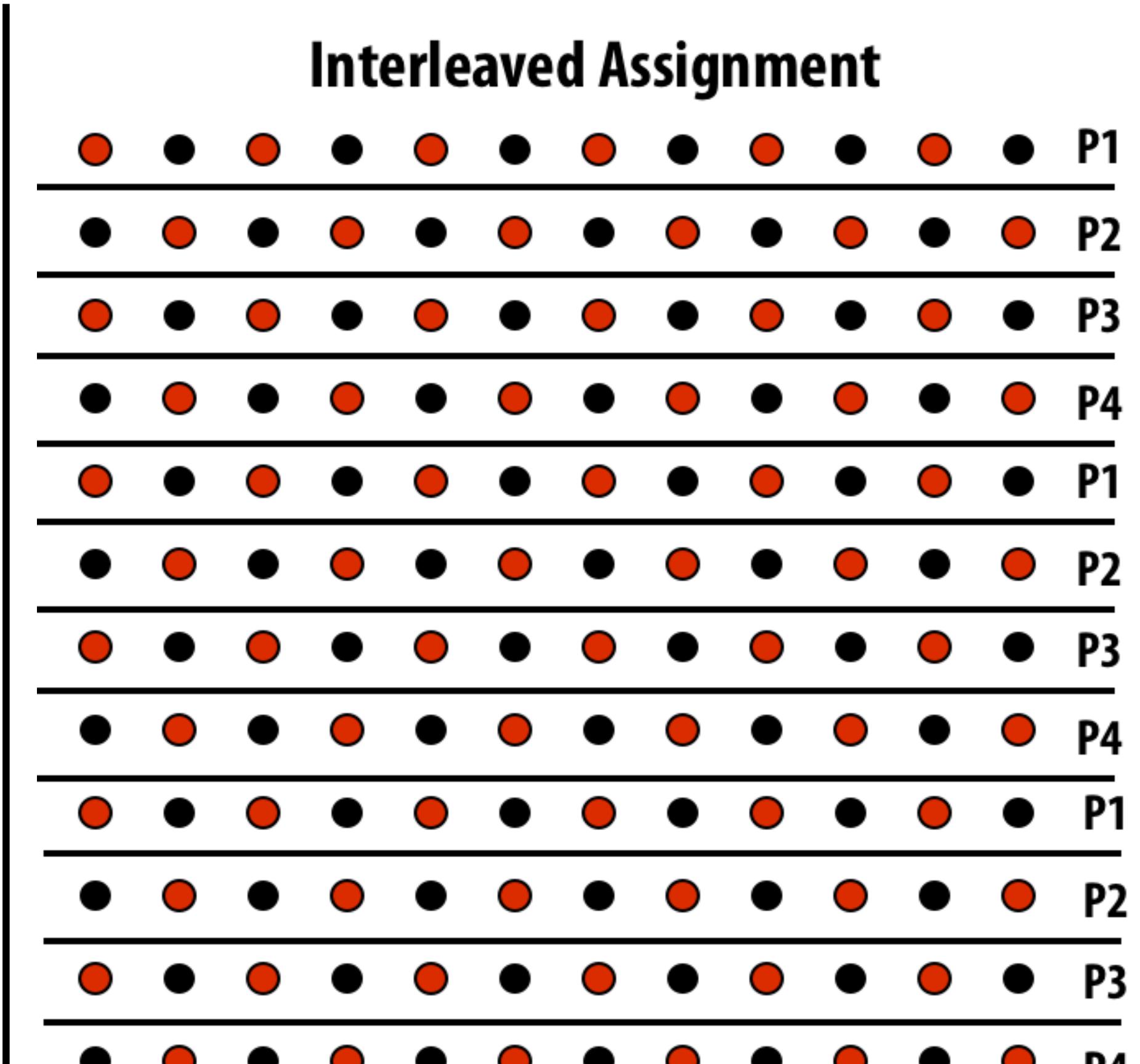
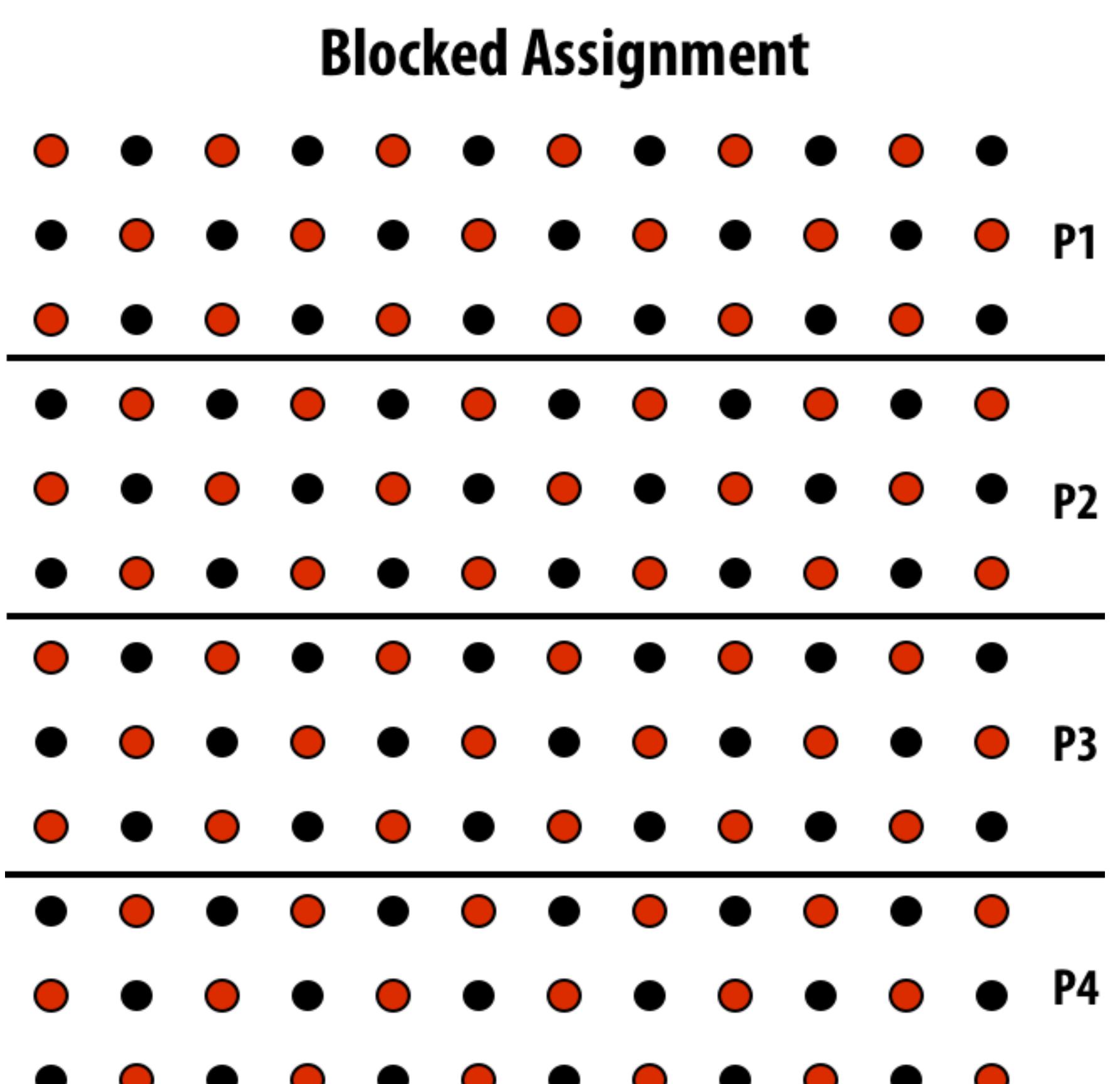


Update all red cells in parallel

When done updating red cells,
update all black cells in parallel
(respect dependency on red cells)

Repeat until convergence

Possible assignments of work to processors

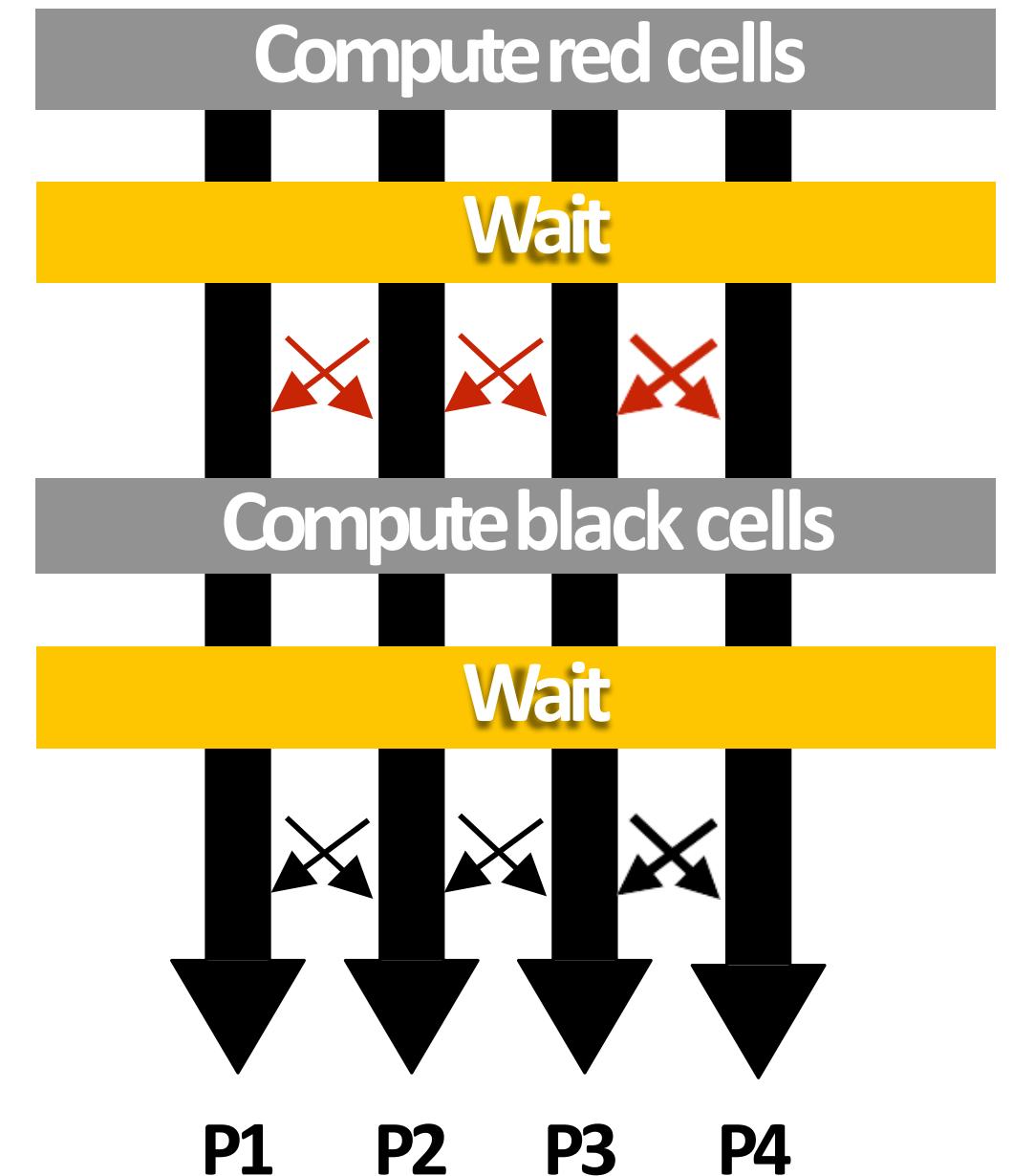


Question: Which is better? Does it matter?

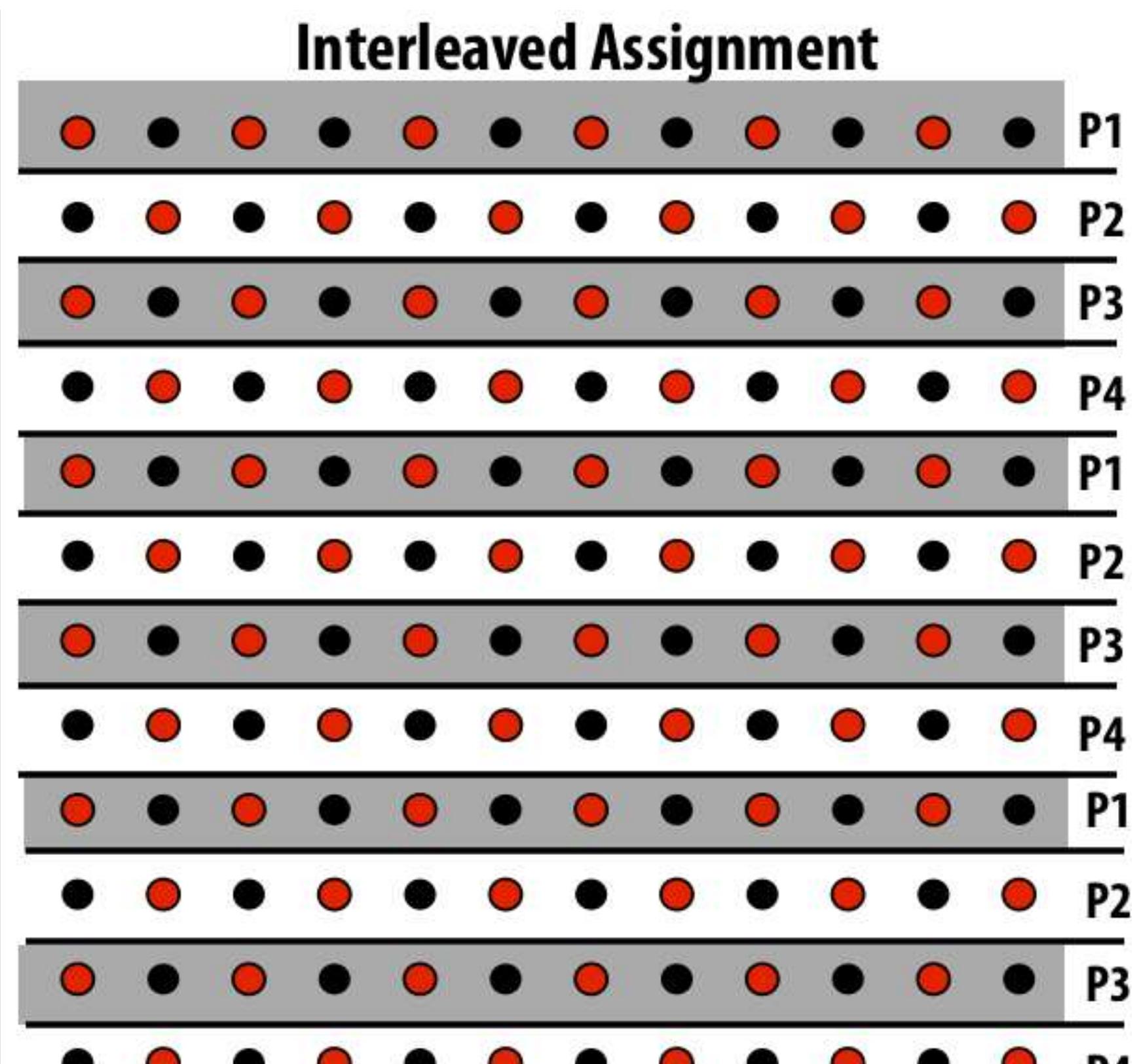
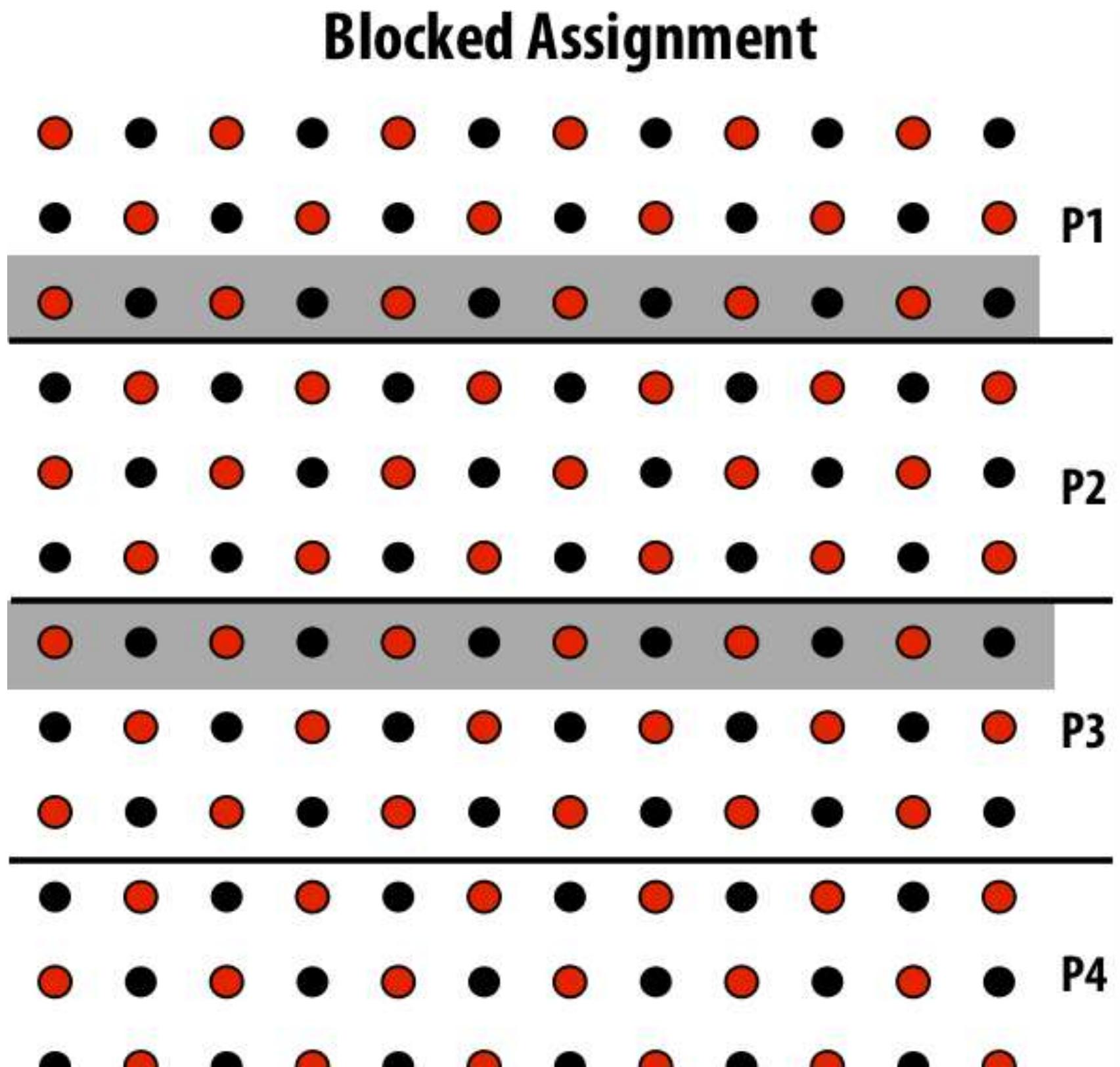
Answer: it depends on the system this program is running on

Consider dependencies (data flow)

1. Perform red update in parallel
2. Wait until all processors done with update
3. Communicate updated red cells to other processors
4. Perform black update in parallel
5. Wait until all processors done with update
6. Communicate updated black cells to other processors
7. Repeat



Communication resulting from assignment



= data that must be sent to P2 each iteration

Blocked assignment requires less data to be communicated between processors

Data-parallel expression of solver

Data-parallel expression of grid solver

Note: to simplify pseudocode: just showing red-cell update

```
const int n;

float* A = allocate(n+2, n+2)); // allocate grid
```

Assignment: ???

```
void solve(float* A) {

    bool done = false;
    float diff = 0.f;
    while (!done) {
        for_all (red cells (i,j)) {
            float prev = A[i,j];
            A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] +
                A[i+1,j] + A[i,j+1]);
            reduceAdd(diff, abs(A[i,j] - prev));
        }
        if (diff/(n*n) < TOLERANCE)
            done = true;
    }
}
```

decomposition:
individual grid elements constitute independent work

Orchestration: handled by system
(builtin communication primitive: reduceAdd)

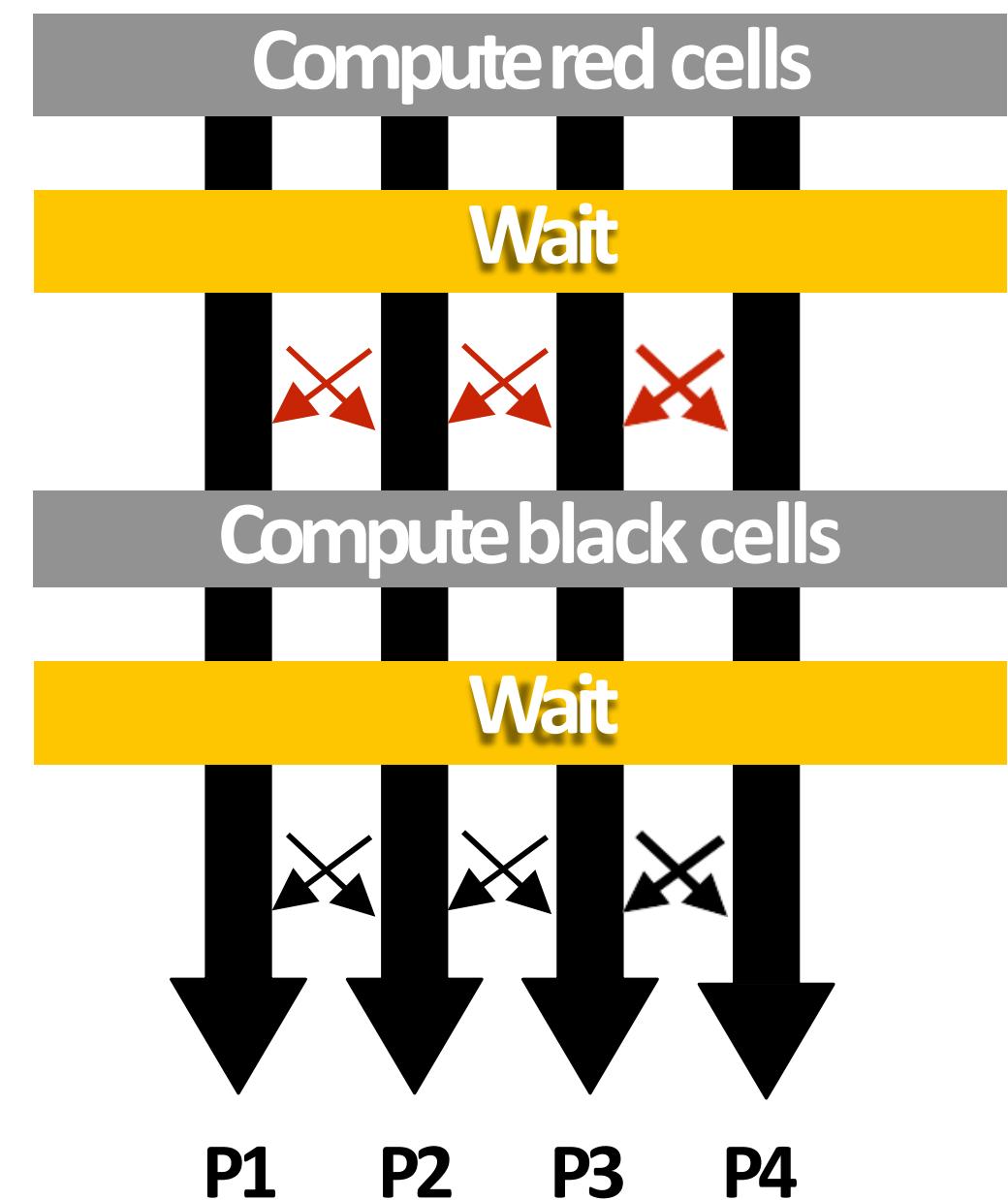
Orchestration:
handled by system
(End of for_all block is implicit wait for all workers before returning to sequential control)

Shared address space (with SPMD threads) expression of solver

Shared address space expression of solver

SPMD execution model

- Programmer is responsible for synchronization
- Common synchronization primitives:
 - Locks (provide mutual exclusion): only one thread in the critical region at a time
 - Barriers: wait for threads to reach this point



Shared address space solver (pseudocode in SPMD execution model)

```
int      n;          // grid size
bool    done = false;
float   diff = 0.0;
LOCK    myLock;
BARRIER myBarrier;

// allocate grid
float* A = allocate(n+2, n+2);

void solve(float* A) {
    int threadId = getThreadId();
    int myMin = 1 + (threadId * n / NUM_PROCESSORS);
    int myMax = myMin + (n / NUM_PROCESSORS)

    while (!done) {
        diff = 0.f;
        barrier(myBarrier, NUM_PROCESSORS);
        for (i=myMin to myMax) {
            for (j = red cells in this row) {
                float prev = A[i,j];
                A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] +
                                  A[i+1,j], A[i,j+1]);
                lock(myLock)
                diff += abs(A[i,j] - prev));
                unlock(myLock);
            }
        }
        barrier(myBarrier, NUM_PROCESSORS);
        if (diff/(n*n) < TOLERANCE)
            done = true;
        barrier(myBarrier, NUM_PROCESSORS);
    }
}
```

Assume these are global variables
(accessible to all threads)

Assume solve function is executed by
all threads. (SPMD-style)

Value of threadId is different for
each SPMD instance: use value to
compute region of grid to work on

Each thread computes the rows it is
responsible for updating

// check convergence, all threads get same answer

Solver implementation in two programming models

Data-parallel programming model

- **Synchronization:**
 - Single logical thread of control, but iterations of `forall` loop may be parallelized by the system (implicit barrier at end of forall loop body)
- **Communication**
 - Implicit in loads and stores (like shared address space)
 - Special built-in primitives for more complex communication patterns:
e.g., `reduce`

Shared address space

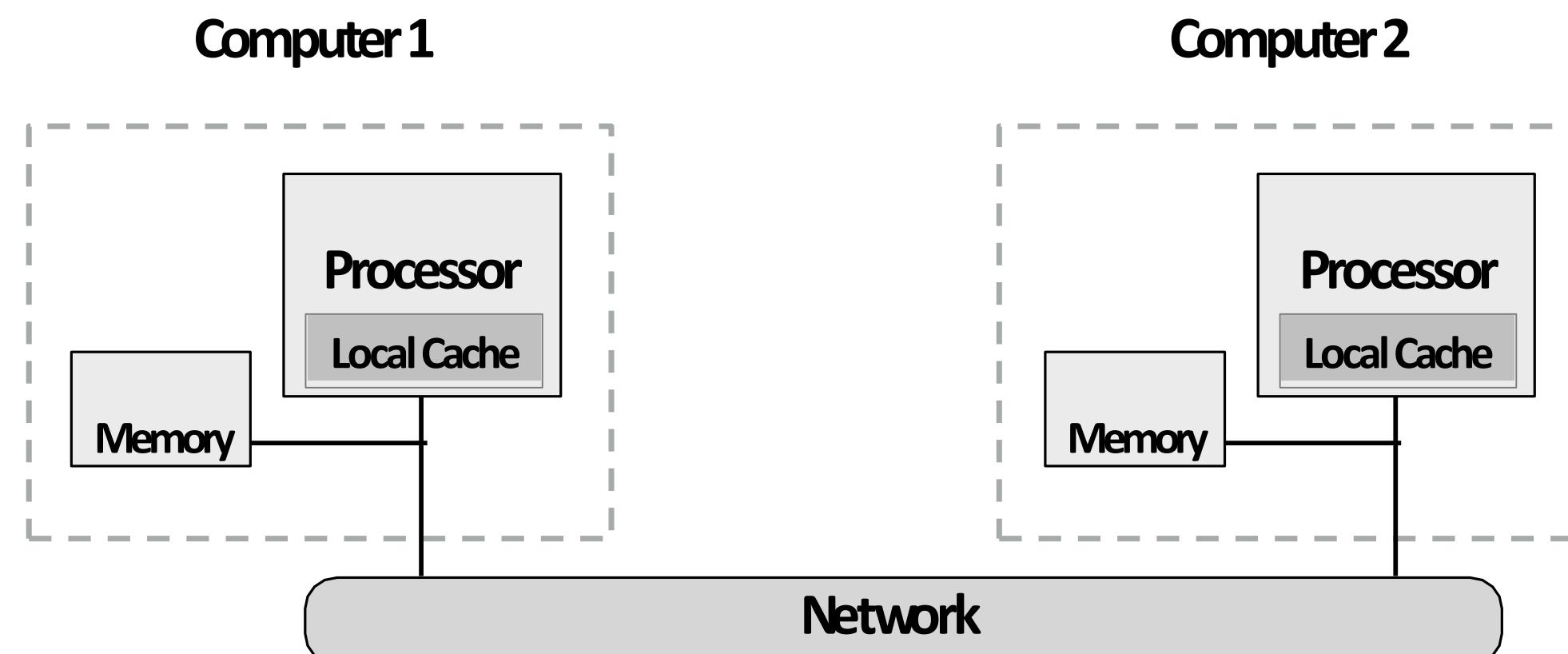
- **Synchronization:**
 - Mutual exclusion required for shared variables (e.g., via locks)
 - Barriers used to express dependencies (between phases of computation)
- **Communication**
 - Implicit in loads/stores to shared variables

Message-passing expression of solver

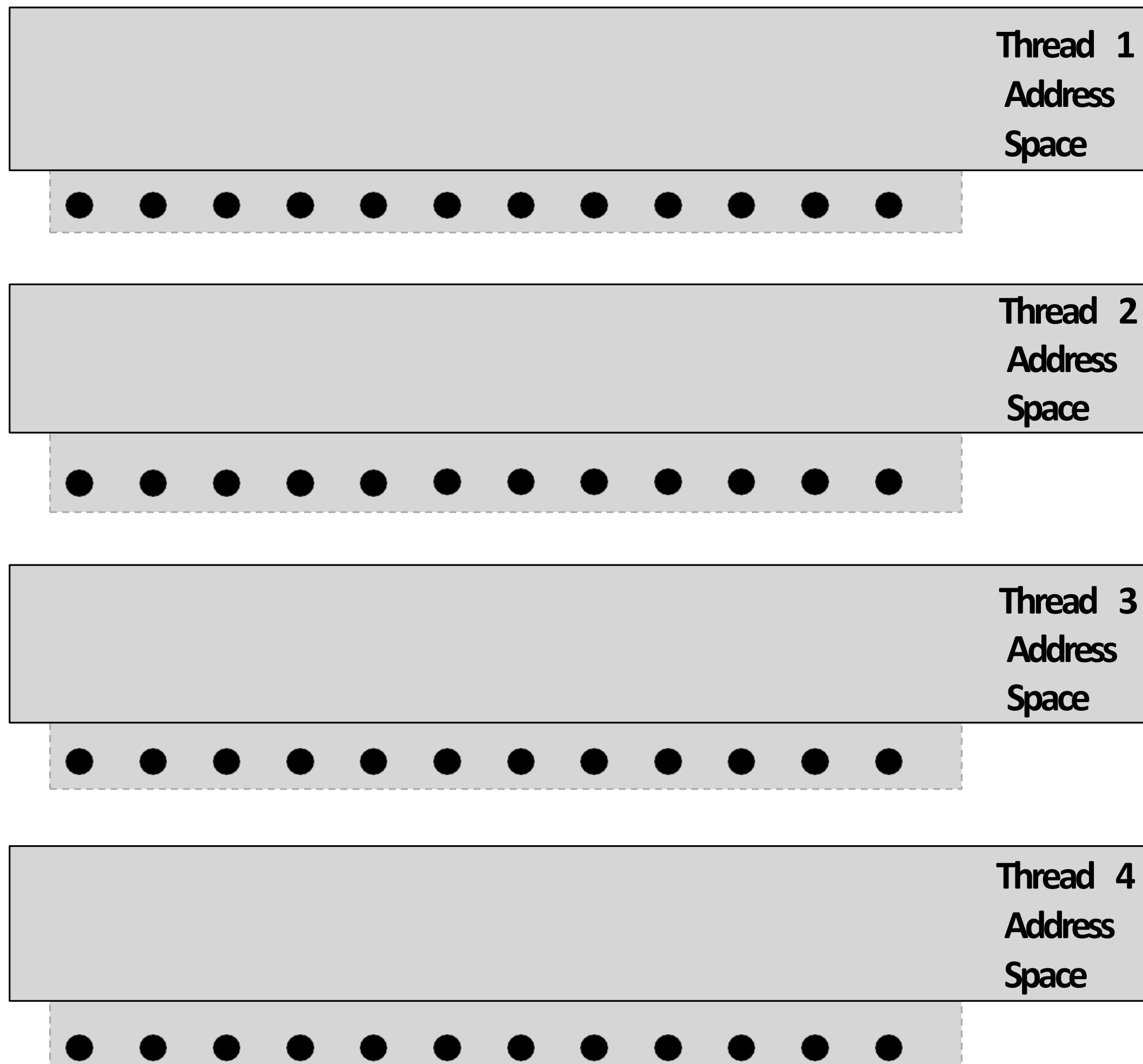
Let's think about expressing a parallel grid solver with communication via messages

- Each thread has its own address space
 - No shared address space abstraction (i.e., no shared variables)
- Threads communicate and synchronize by sending/receiving messages

One possible message passing machine configuration: a cluster of two workstations (you could make this cluster yourself using the machines in the GHC labs)



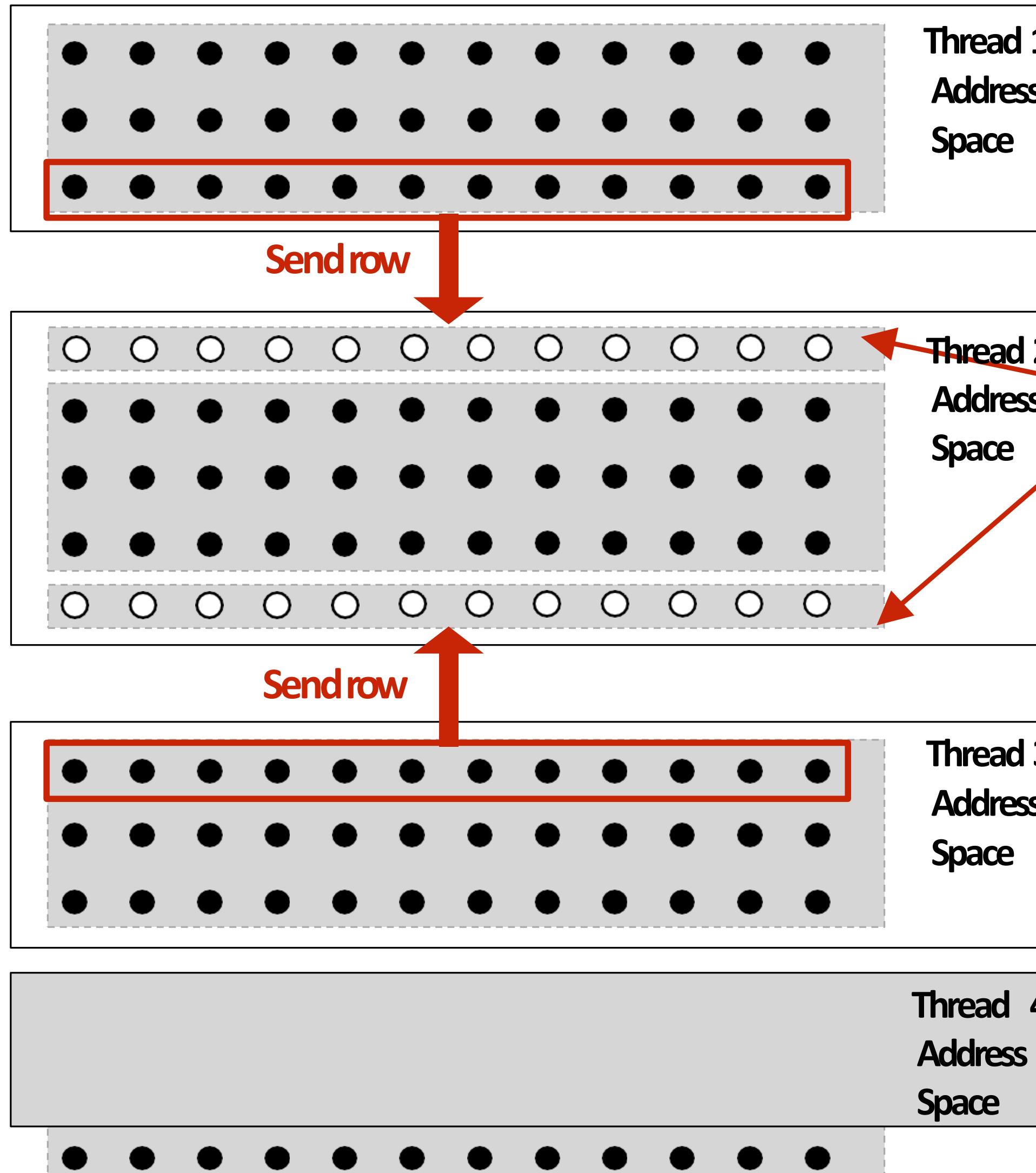
Message passing model: each thread operates in its own address space



In this figure: four threads

**The grid data is partitioned into four allocations, each residing in one of the four unique thread address spaces
(four per-thread private arrays)**

Data replication is now required to correctly execute the program



Example:

After red cell processing is complete, thread 1 and thread 3 send row of data to thread 2
(thread 2 requires up-to-date red cell information to update black cells in the next phase)

"**Ghost cells**" are grid cells replicated from a remote address space. It's common to say that information in ghost cells is "**owned**" by other threads.

Thread 2 logic:

```
float* local_data = allocate(N+2, rows_per_thread+2);

int tid = get_thread_id();
int bytes = sizeof(float) * (N+2);

// receive ghost row cells (white dots)
recv(&local_data[0,0], bytes, tid-1);
recv(&local_data[rows_per_thread+1,0], bytes, tid+1);

// Thread 2 now has data necessary to perform
// future computation
```

Message passing solver

Similar structure to shared address space solver, but now communication is explicit in message sends and receives

Send and receive ghost rows to “neighbor threads”

Perform computation
(just like in shared address space version of solver)

All threads send local my_diff to thread 0

Thread 0 computes global diff, evaluates termination predicate and sends result back to all other threads

```
int N;
int tid = get_thread_id();
int rows_per_thread = N / get_num_threads();

float* localA = allocate(rows_per_thread+2, N+2);

// assume localA is initialized with starting values
// assume MSG_ID_ROW, MSG_ID_DONE, MSG_ID_DIFF are constants used as msg ids
///////////
void solve() {
    bool done = false;
    while (!done) {

        float my_diff = 0.0f;

        if (tid != 0)
            send(&localA[1,0], sizeof(float)*(N+2), tid-1, MSG_ID_ROW);
        if (tid != get_num_threads()-1)
            send(&localA[rows_per_thread,0], sizeof(float)*(N+2), tid+1, MSG_ID_ROW);

        if (tid != 0)
            recv(&localA[0,0], sizeof(float)*(N+2), tid-1, MSG_ID_ROW);
        if (tid != get_num_threads()-1)
            recv(&localA[rows_per_thread+1,0], sizeof(float)*(N+2), tid+1, MSG_ID_ROW);

        for (int i=1; i<rows_per_thread+1; i++) {
            for (int j=1; j<n+1; j++) {
                float prev = localA[i,j];
                localA[i,j] = 0.2 * (localA[i-1,j] + localA[i,j] + localA[i+1,j] +
                                      localA[i,j-1] + localA[i,j+1]);
                my_diff += fabs(localA[i,j] - prev);
            }
        }

        if (tid != 0) {
            send(&mydiff, sizeof(float), 0, MSG_ID_DIFF);
            recv(&done, sizeof(bool), 0, MSG_ID_DONE);
        } else {
            float remote_diff;
            for (int i=1; i<get_num_threads()-1; i++) {
                recv(&remote_diff, sizeof(float), i, MSG_ID_DIFF);
                my_diff += remote_diff;
            }
            if (my_diff/(N*N) < TOLERANCE)
                done = true;
            for (int i=1; i<get_num_threads()-1; i++)
                send(&done, sizeof(bool), i, MSG_ID_DONE);
        }
    }
}
```

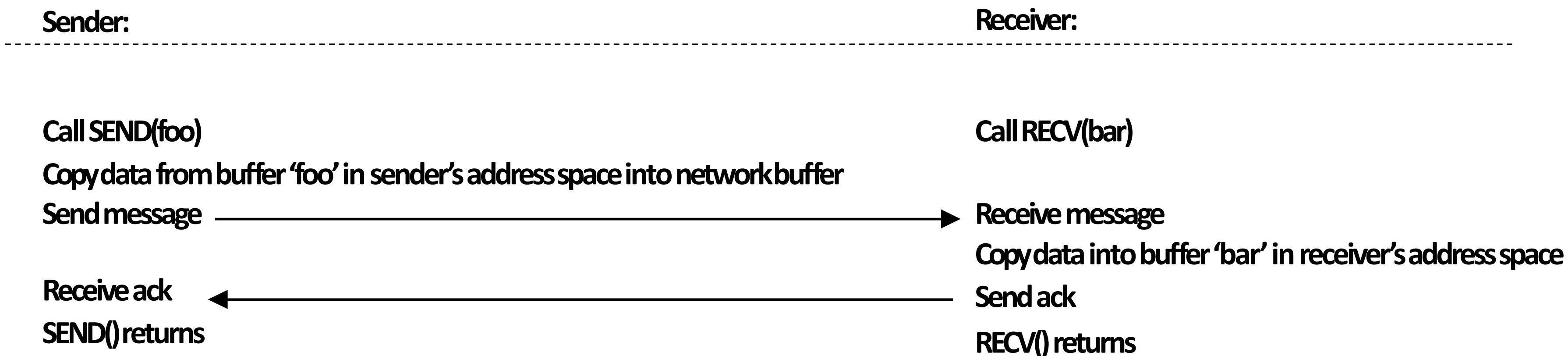
Notes on message passing example

- **Computation**
 - Array indexing is relative to **local address space** (not global grid coordinates)
- **Communication:**
 - Performed by sending and receiving messages
 - Bulk transfer: **communicate entire rows at a time** (not individual elements)
- **Synchronization:**
 - Performed by **sending and receiving messages**
 - Think of how to implement mutual exclusion, barriers, flags using messages
- **For convenience, message passing libraries often include higher-level primitives (implemented via send and receive)**

```
reduce_add(0, &my_diff, sizeof(float)); if      // add up all my_diffs, return result to thread 0
(pid == 0 && my_diff/(N*N) < TOLERANCE)
    done = true;
broadcast(0, &done, sizeof(bool), MSG_DONE); // thread 0 sends done to all threads
```

Synchronous (blocking) send and receive

- **send(): call returns when sender receives acknowledgement that message data resides in address space of receiver**
- **recv(): call returns when data from received message is copied into address space of receiver and acknowledgement sent back to sender**



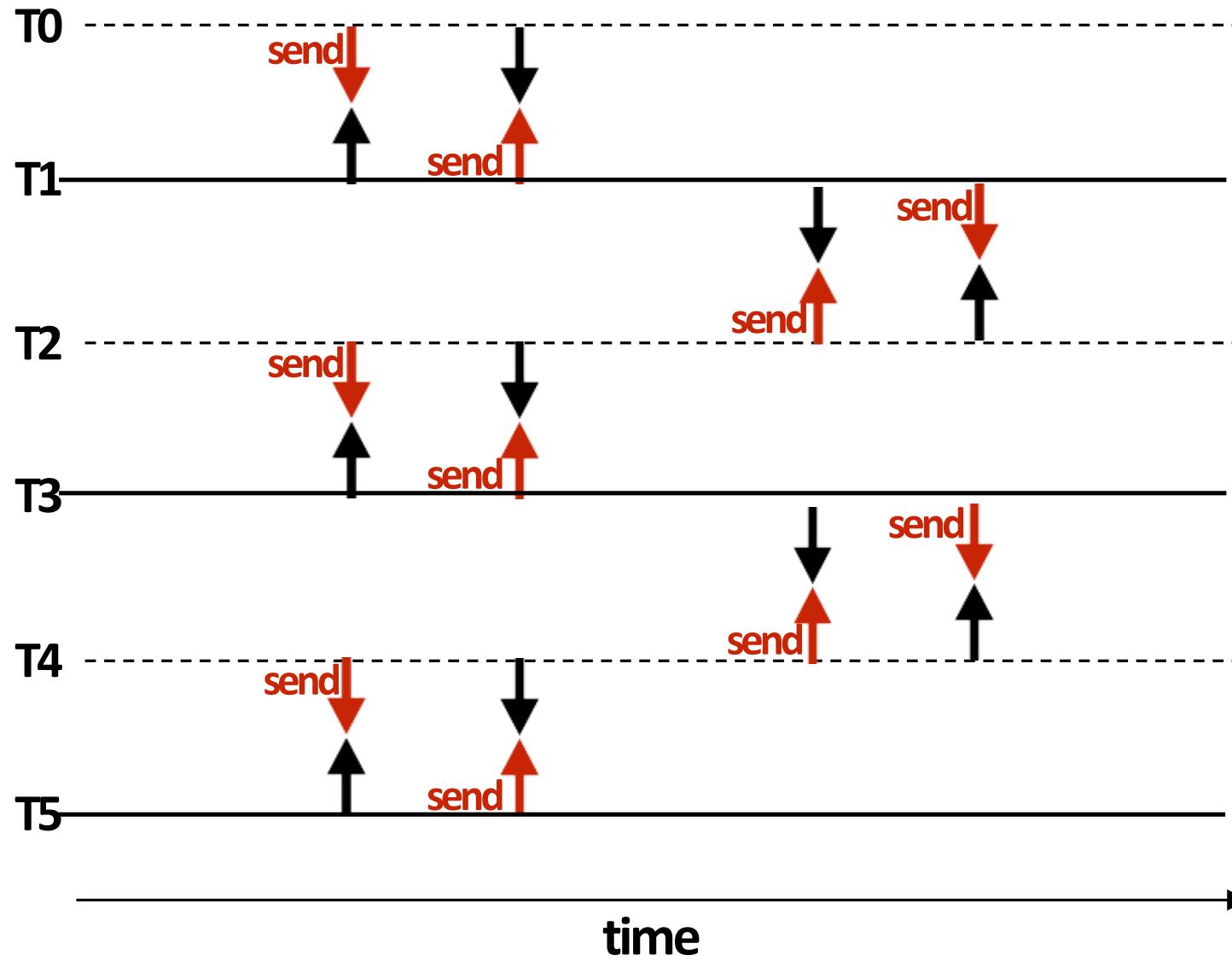
**As implemented on the prior slide, there is a
big problem with our message passing solver
if it uses synchronous send/recv!**

Why?

**How can we fix it?
(while still using synchronous send/recv)**

Message passing solver (fixed to avoid deadlock)

Send and receive ghost rows to “neighbor threads”
 Even-numbered threads send, then receive
 Odd-numbered thread recv, then send



```

int N;
int tid = get_thread_id();
int rows_per_thread = N / get_num_threads();

float* localA = allocate(rows_per_thread+2, N+2);

// assume localA is initialized with starting values
// assume MSG_ID_ROW, MSG_ID_DONE, MSG_ID_DIFF are constants used as msg ids
///////////
void solve() {
    bool done = false;
    while (!done) {

        float my_diff = 0.0f;

        if (tid % 2 == 0) {
            sendDown(); recvDown();
            sendUp(); recvUp();
        } else {
            recvUp(); sendUp();
            recvDown(); sendDown();
        }

        for (int i=1; i<rows_per_thread-1; i++) {
            for (int j=1; j<n+1; j++) {
                float prev = localA[i,j];
                localA[i,j] = 0.2 * (localA[i-1,j] + localA[i,j] + localA[i+1,j] +
                                      localA[i,j-1] + localA[i,j+1]);
                my_diff += fabs(localA[i,j] - prev);
            }
        }

        if (tid != 0) {
            send(&mydiff, sizeof(float), 0, MSG_ID_DIFF);
            recv(&done, sizeof(bool), 0, MSG_ID_DONE);
        } else {
            float remote_diff;
            for (int i=1; i<get_num_threads()-1; i++) {
                recv(&remote_diff, sizeof(float), i, MSG_ID_DIFF);
                my_diff += remote_diff;
            }
            if (my_diff/(N*N) < TOLERANCE)
                done = true;
            if (int i=1; i<gen_num_threads()-1; i++)
                send(&done, sizeof(bool), i, MSD_ID_DONE);
        }
    }
}

```

Non-blocking asynchronous send/recv

- **send(): call returns immediately**
 - Buffer provided to send() cannot be modified by calling thread since message processing occurs concurrently with thread execution
 - Calling thread can perform other work while waiting for message to be sent
- **recv(): posts intent to receive in the future, returns immediately**
 - Use checksend(), checkrecv() to determine actual status of send/receipt
 - Calling thread can perform other work while waiting for message to be received

Sender:

Call SEND(foo)

SEND returns handle h1

Copy data from 'foo' into network buffer

Send message

Call CHECKSEND(h1) //if message sent, now safe for thread to modify 'foo'

Receiver:

Call RECV(bar)

RECV(bar) returns handle h2

Receive message

Messaging library copies data into 'bar'

Call CHECKRECV(h2)

//if received, now safe for thread

//to access 'bar'

RED TEXT = executes concurrently with application thread

CSCI465/ECEN433

Introduction to Parallel Computing

Spring 2024



Lecture (8)

ProgPerf Part 1: Work Distribution

Programming for high performance

- Optimizing the performance of parallel programs is an **iterative process** of refining choices for decomposition, assignment, and orchestration...
- **Key goals (that are at odds with each other)**
 - Balance workload onto available execution resources
 - Reduce communication (to avoid stalls)
 - Reduce extra work (overhead) performed to increase parallelism, manage assignment, reduce communication, etc.
- We are going to talk about a rich space of techniques

TIP #1: Always implement the simplest solution first, then measure performance to determine if you need to do better.

“My solution scales” = your code scales as much as you need it to.

(if you anticipate only running low-core count machines, it may be unnecessary to implement a complex approach that creates and hundreds or thousands of pieces of independent work)

Balancing the workload

Ideally: all processors are computing all the time during program execution
(they are computing simultaneously, and they finish their portion of the work at the same time)



Recall Amdahl's Law:

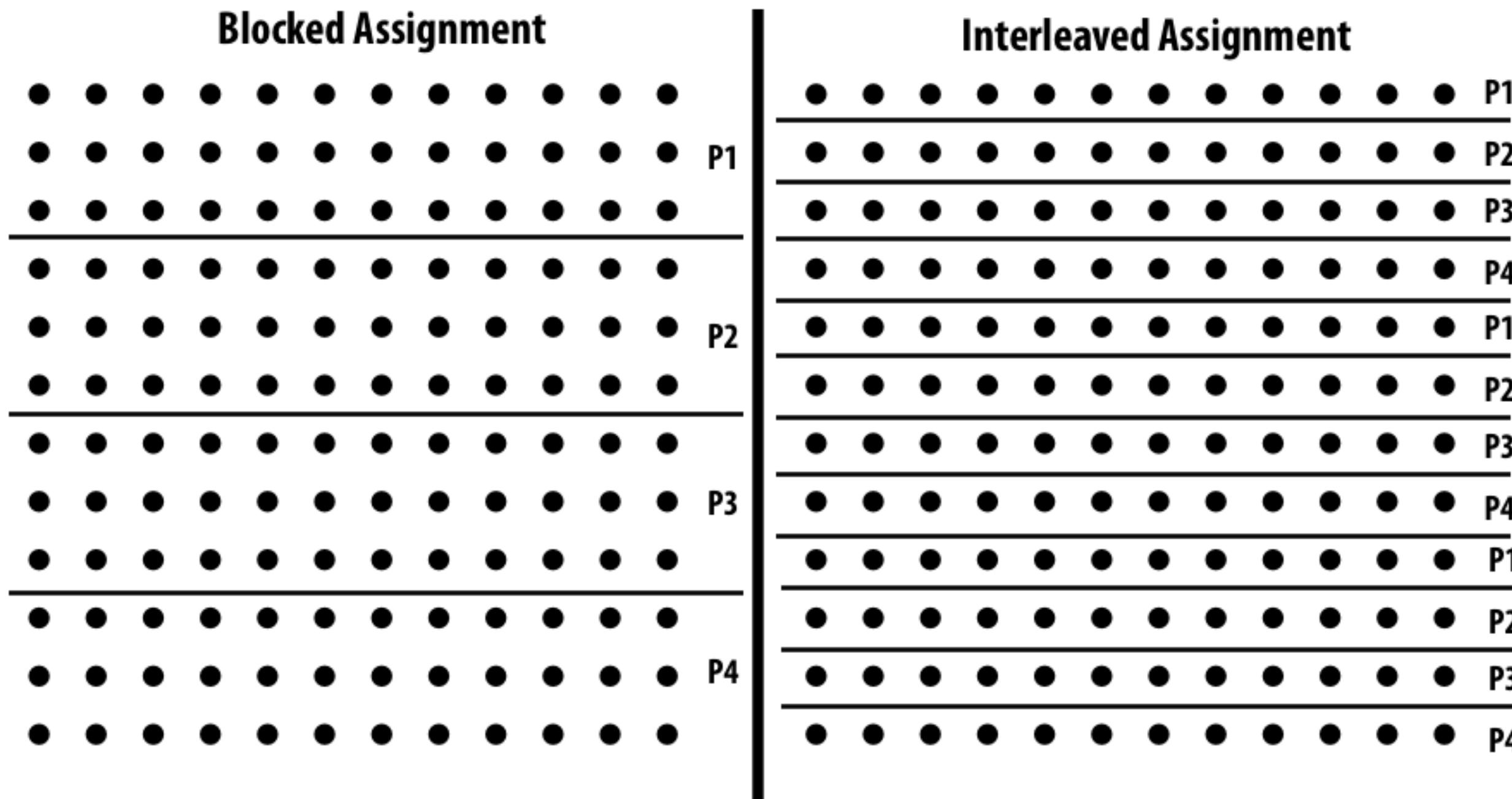
Only small amount of load imbalance can significantly bound maximum speedup

P4 does 20% more work → P4 takes 20% longer to complete
→ 20% of parallel program's runtime is serial execution

(work in serialized section here is about 5% of the work of the whole program:
 $S=.05$ in Amdahl's law equation)

Static assignment

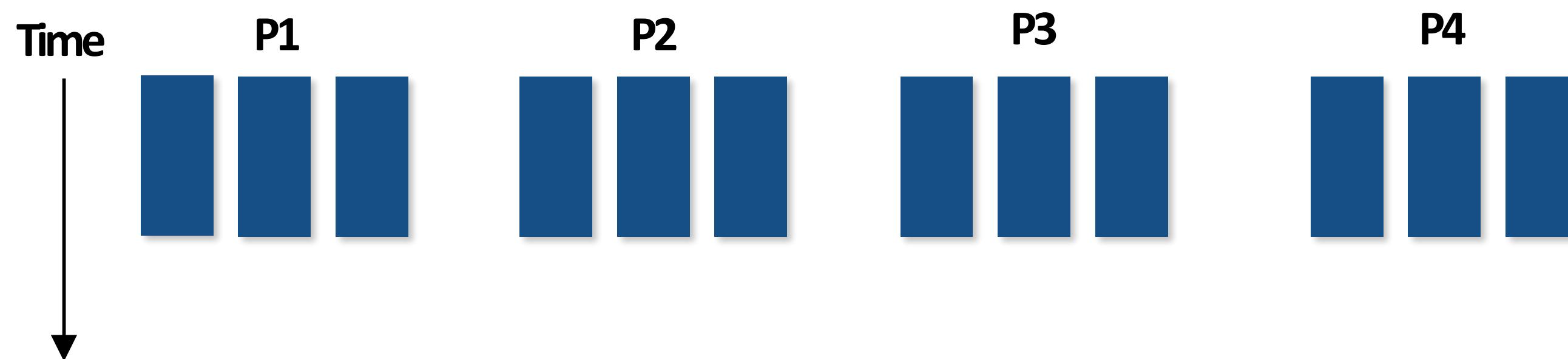
- Assignment of work to threads is **pre-determined**
 - Not necessarily determined at compile-time (assignment algorithm may depend on runtime parameters such as input data size, number of threads, etc.)
- Recall solver example: assign equal number of grid cells (work) to each thread (worker)
 - We discussed two static assignments of work to workers (**blocked** and **interleaved**)



- **Good properties of static assignment:** simple, essentially zero **runtime overhead** (in this example: extra work to implement assignment is a little bit of indexing math)

When is static assignment applicable?

- When the **cost (execution time) of work** and the **amount of work** is **predictable** (so the programmer can work out a good assignment in advance)
- Simplest example: it is known up front that all **work has the same cost**



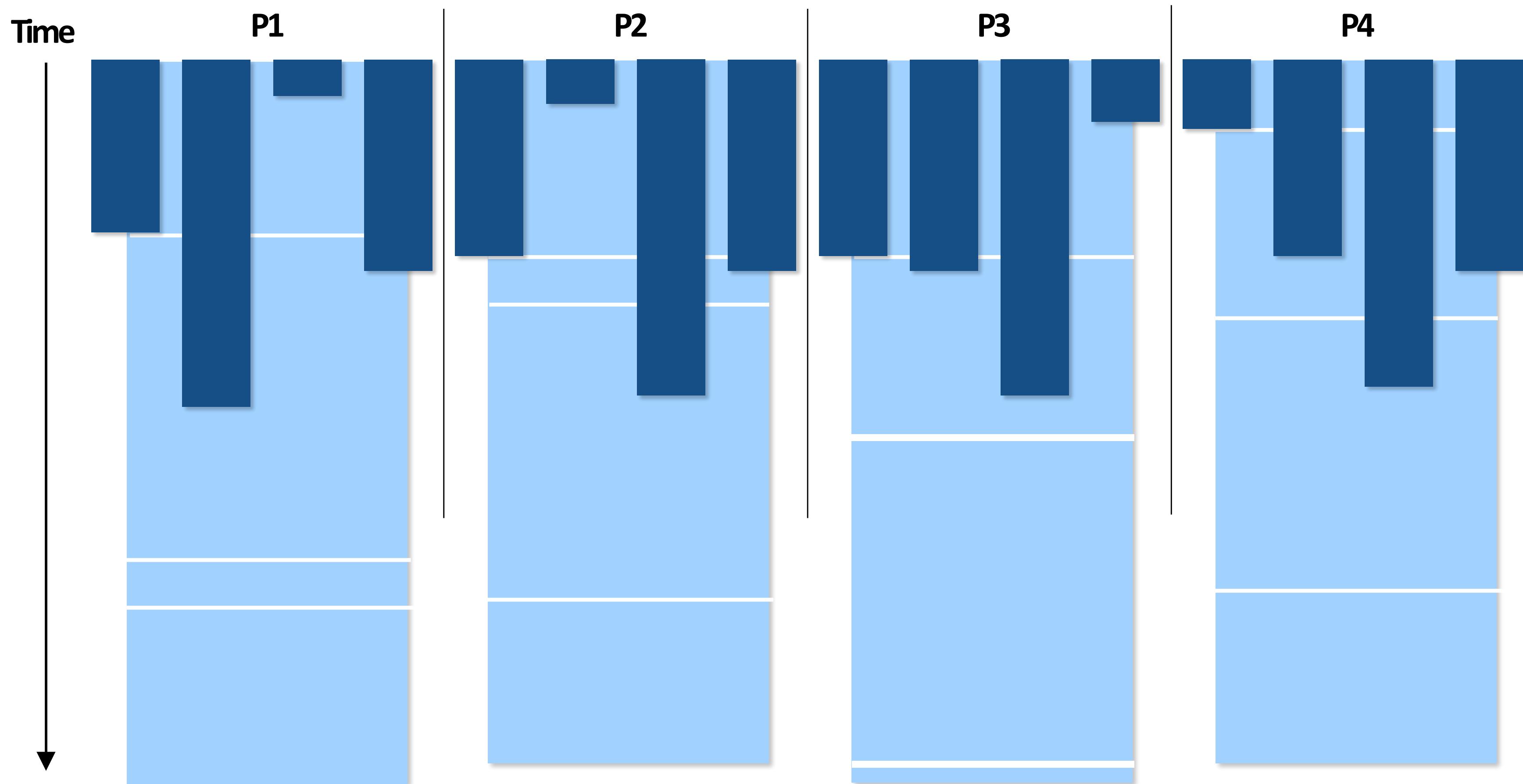
In the example above:

There are 12 tasks, and it is known that each have the same cost.

Assignment solution: statically assign three tasks to each of the four processors.

When is static assignment applicable?

- When work is **predictable**, but **not all jobs have same cost** (see example below)
- When **statistics about execution time are known** (e.g., same cost on average)



Jobs have unequal, but known cost: assign to processors to ensure overall good load balance

“Semi-static” assignment

- Cost of work is predictable for near-term future
 - Idea: recent past good predictor of near future
- Application periodically profiles itself and re-adjusts assignment
 - Assignment is “static” for the interval between re-adjustments

Dynamic assignment

Program **determines assignment dynamically at runtime** to ensure a well distributed load. (The execution time of tasks, or the total number of tasks, is **unpredictable**.)

Sequential program
(independent loop iterations)

```
int N = 1024;
int* x = new int[N];
bool* prime = new bool[N];

// initialize elements of x here

for (int i=0; i<N; i++)
{
    // unknown execution time
    is_prime[i] = test_primality(x[i]);
}
```

Parallel program
(SPMD execution by multiple threads,
shared address space model)

```
int N = 1024;
// assume allocations are only executed by 1 thread
int* x = new int[N];
bool* is_prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;      // shared variable

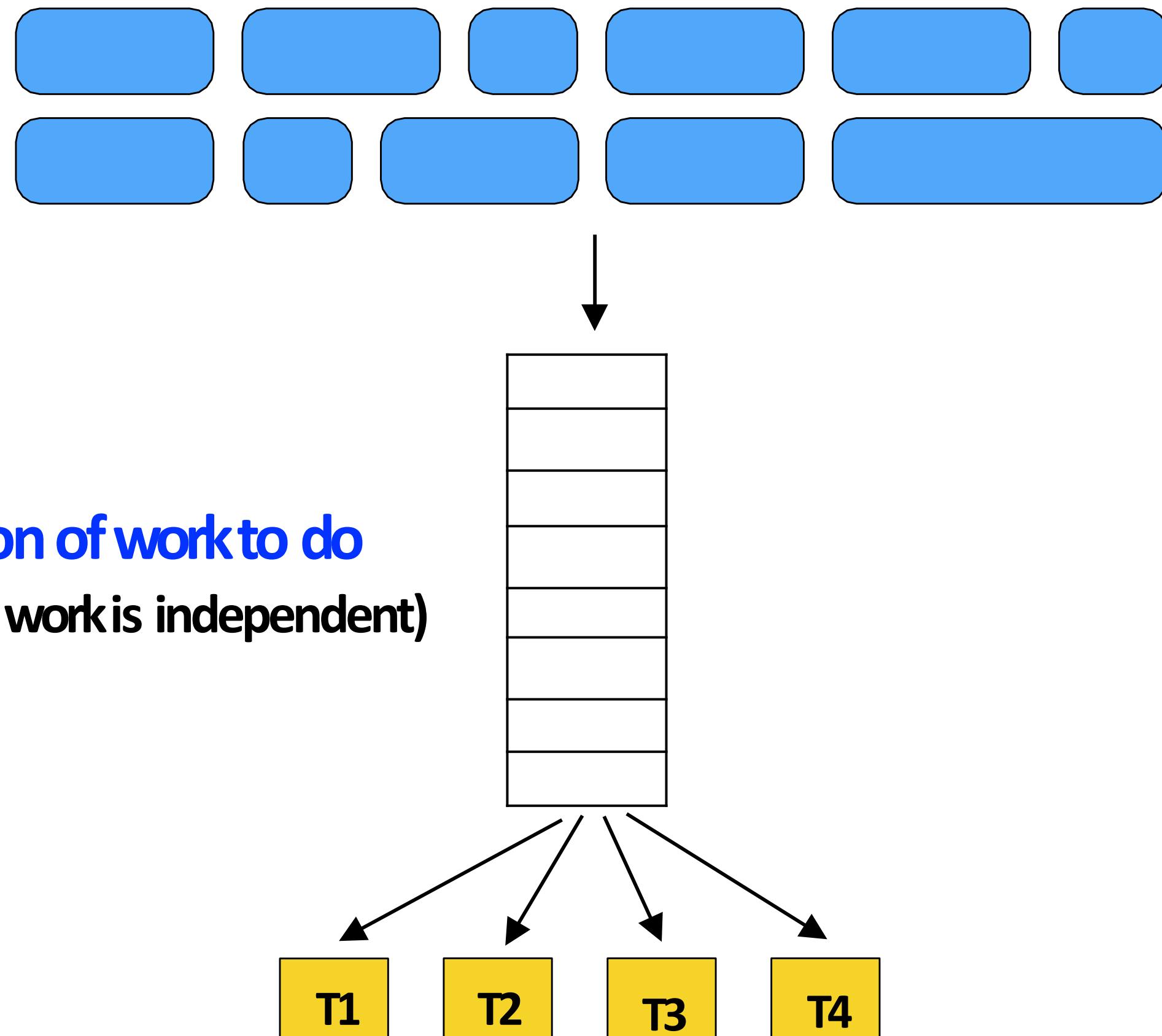
while (1) {
    int i;
    lock(counter_lock);
    i = counter++;
    unlock(counter_lock);
    if (i >= N)
        break;
    is_prime[i] = test_primality(x[i]);
}

atomic_incr(counter);
```

Dynamic assignment using a work queue

Sub-problems

(a.k.a. “tasks”, “work”)



Shared work queue: a collection of work to do
(for now, let's assume each piece of work is independent)

Worker threads:

Pull data from shared work queue Push
new work to queue as it is created

What constitutes a piece of work?

What is a potential problem with this implementation?

```
const int N = 1024;
// assume allocations are only executed by 1 thread
float* x = new float[N];
bool* prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;

while (1) {
    int i;
    lock(counter_lock);
    i = counter++;
    unlock(counter_lock);
    if (i >= N)
        break;
    is_prime[i] = test_primality(x[i]);
}
```

Fine granularity partitioning: 1 “task” = 1 element

Likely good workload balance (many small tasks)

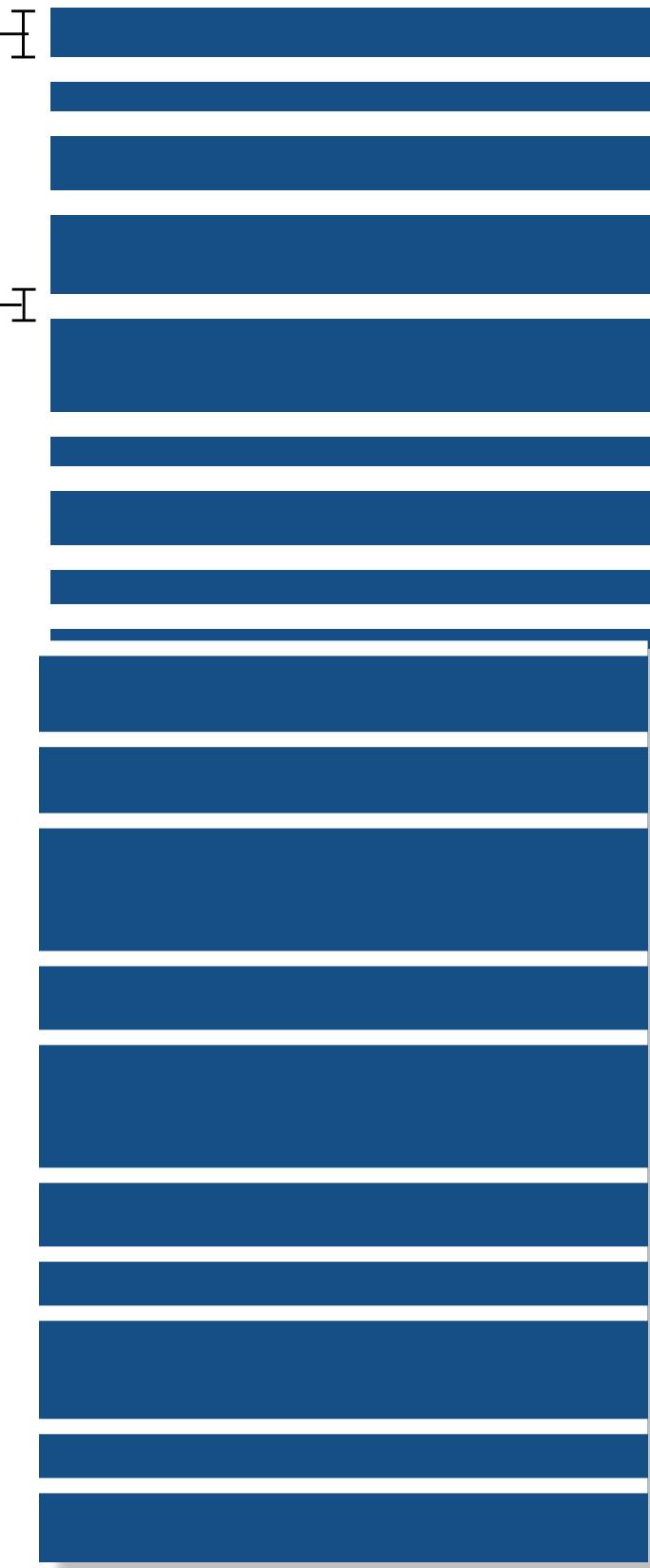
Potential for high synchronization cost
(serialization at critical section)

Time in task 0

Time in critical section

This is overhead that
does not exist in serial
program

And.. it's serial execution
(recall Amdahl's Law)



So... IS IT a problem?

Increasing task granularity

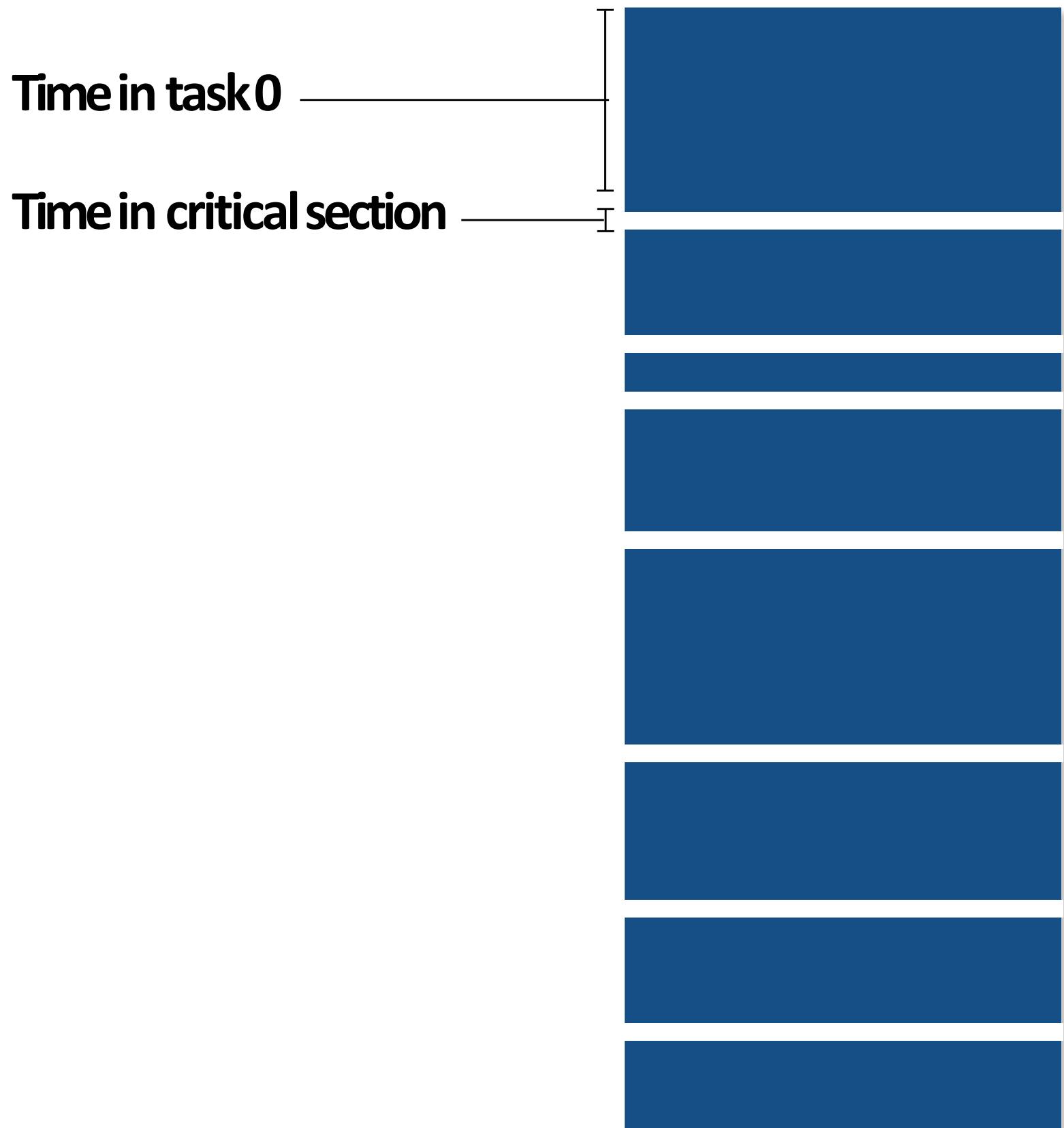
```
const int N = 1024;
const int GRANULARITY = 10;
// assume allocations are only executed by 1 thread

float* x = new float[N];
bool* prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;

while (1) {
    int i;
    lock(counter_lock);
    i = counter;
    counter += GRANULARITY;
    unlock(counter_lock);
    if (i >= N)
        break;
    int end = min(i + GRANULARITY, N);
    for (int j=i; j<end; j++)
        is_prime[j] = test_primality(x[j]);
}
```



Coarse granularity partitioning: 1 “task” = 10 elements

Decreased synchronization cost

(Critical section entered 10 times less)

Choosing task size

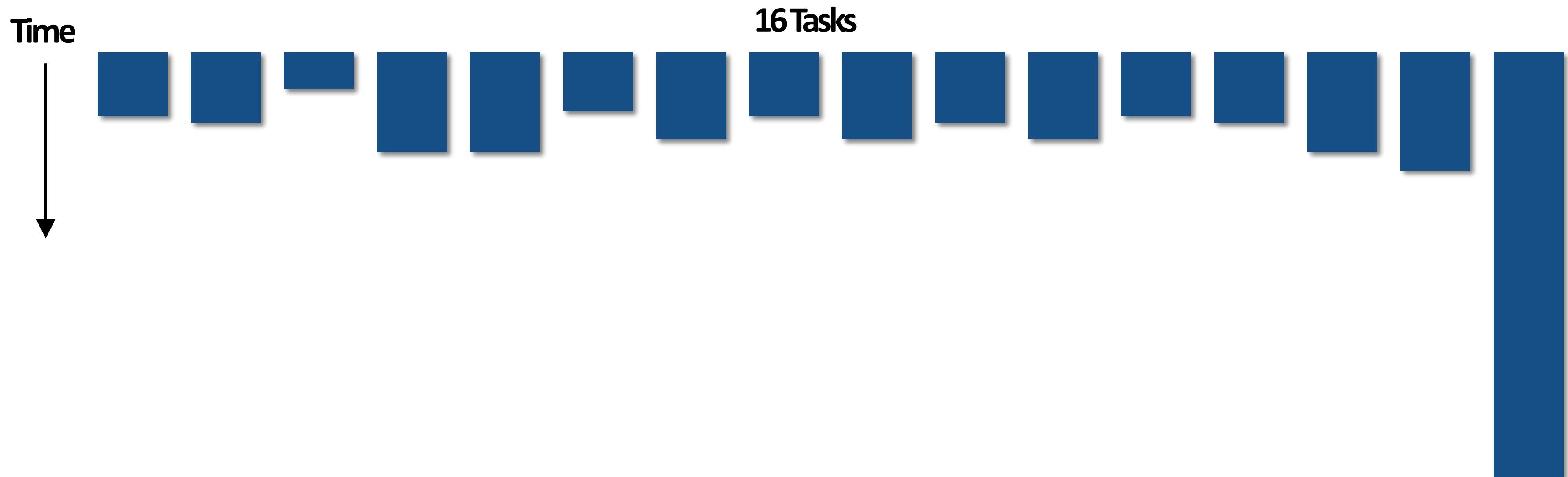
- Useful to have many more tasks* than processors
(many small tasks enables good **workload balance** via dynamic assignment)
 - Motivates **small granularity** tasks
- But want as few tasks as possible to **minimize overhead** of managing the assignment
 - Motivates **large granularity** tasks
- Ideal granularity depends on many factors
(Common theme in this course: must know your workload, and your machine)

*I had to pick a term for a piece of work, a sub-problem, etc.

Smarter task scheduling

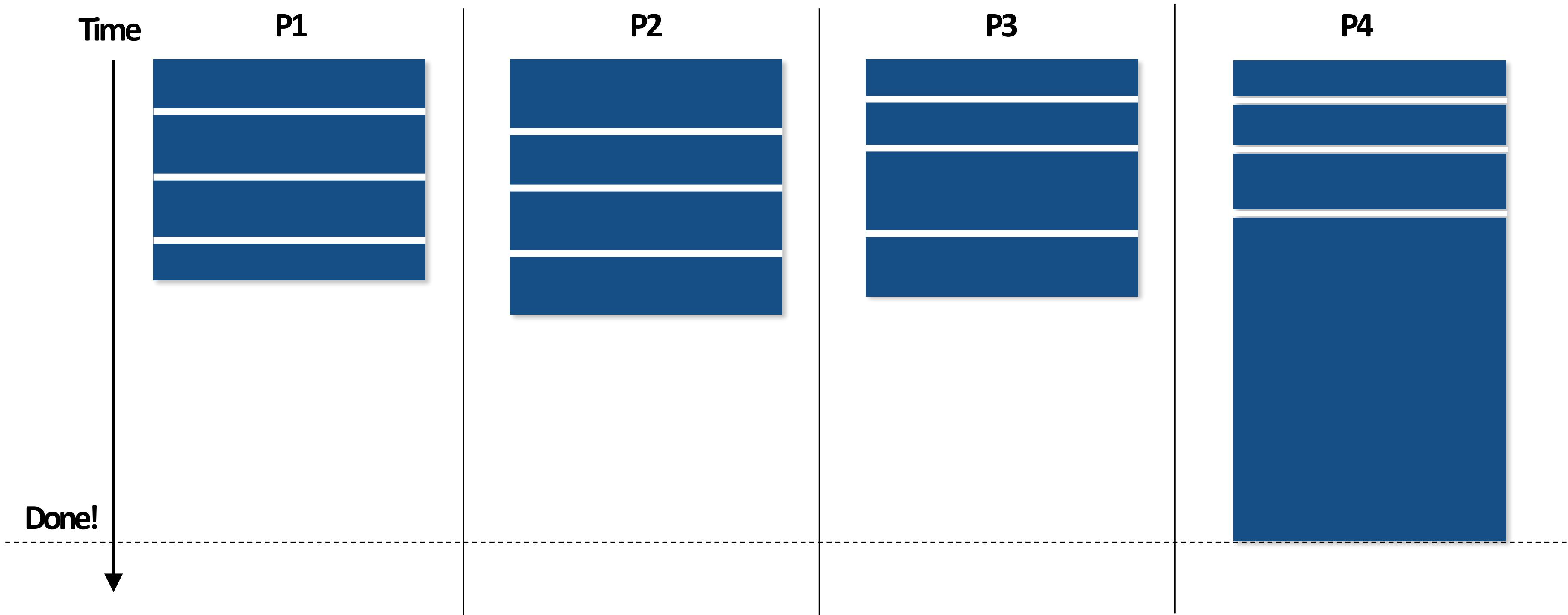
Consider dynamic scheduling via a shared work queue

What happens if the system assigns these tasks to four workers in left-to-right order?



Smarter task scheduling

What happens if scheduler runs the long task last? Potential for **load imbalance!**



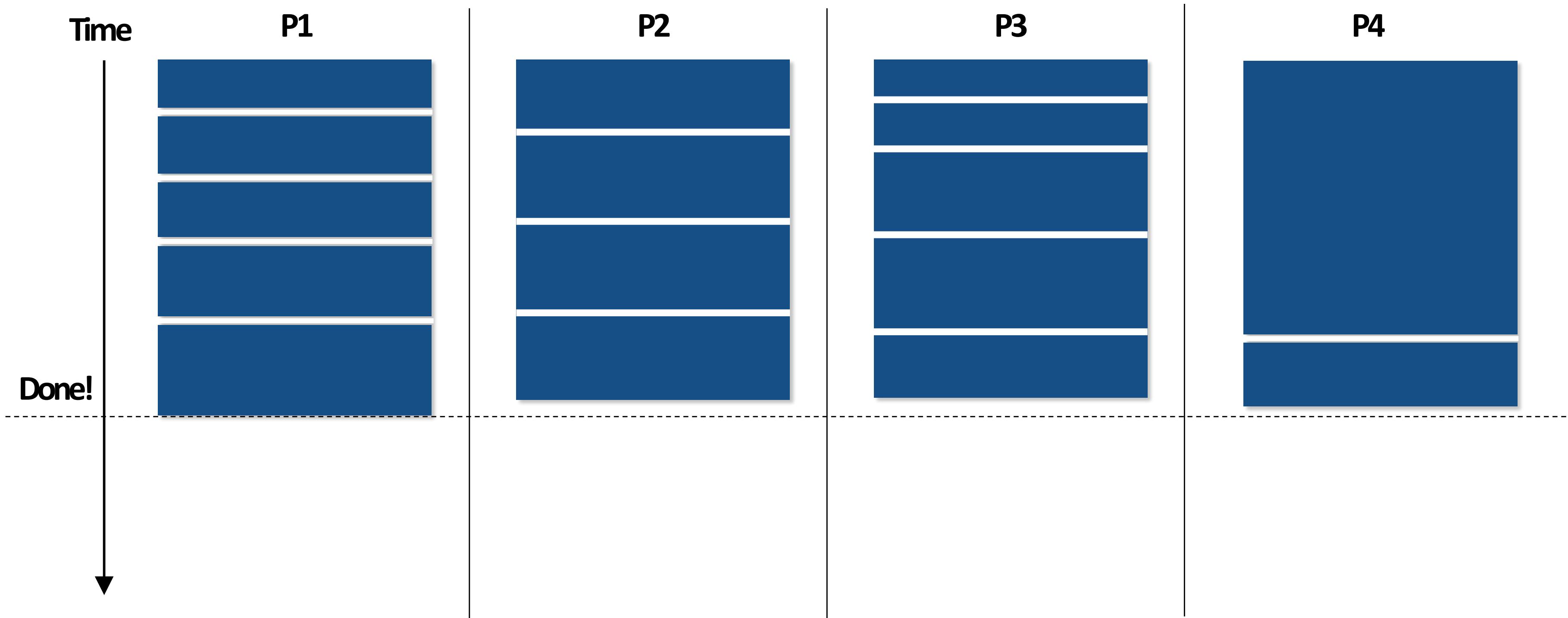
One possible solution to imbalance problem:

Divide work into a larger number of smaller tasks

- Hopefully “long pole” gets shorter relative to overall execution time
- May increase synchronization overhead
- May not be possible (perhaps long task is fundamentally sequential)

Smarter task scheduling

Schedule long task first to reduce “slop” at end of computation



Another solution: **smarter scheduling**

Schedule long tasks first

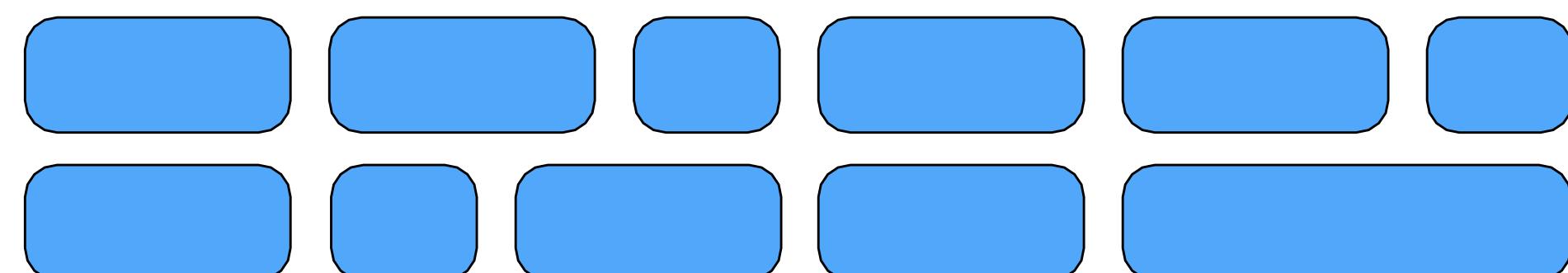
- Thread performing long task performs fewer overall tasks, but approximately the same amount of work as the other threads.
- Requires some knowledge of workload (some predictability of cost)

Decreasing synchronization overhead using a distributed set of queues

(avoid need for all workers to synchronize on single work queue)

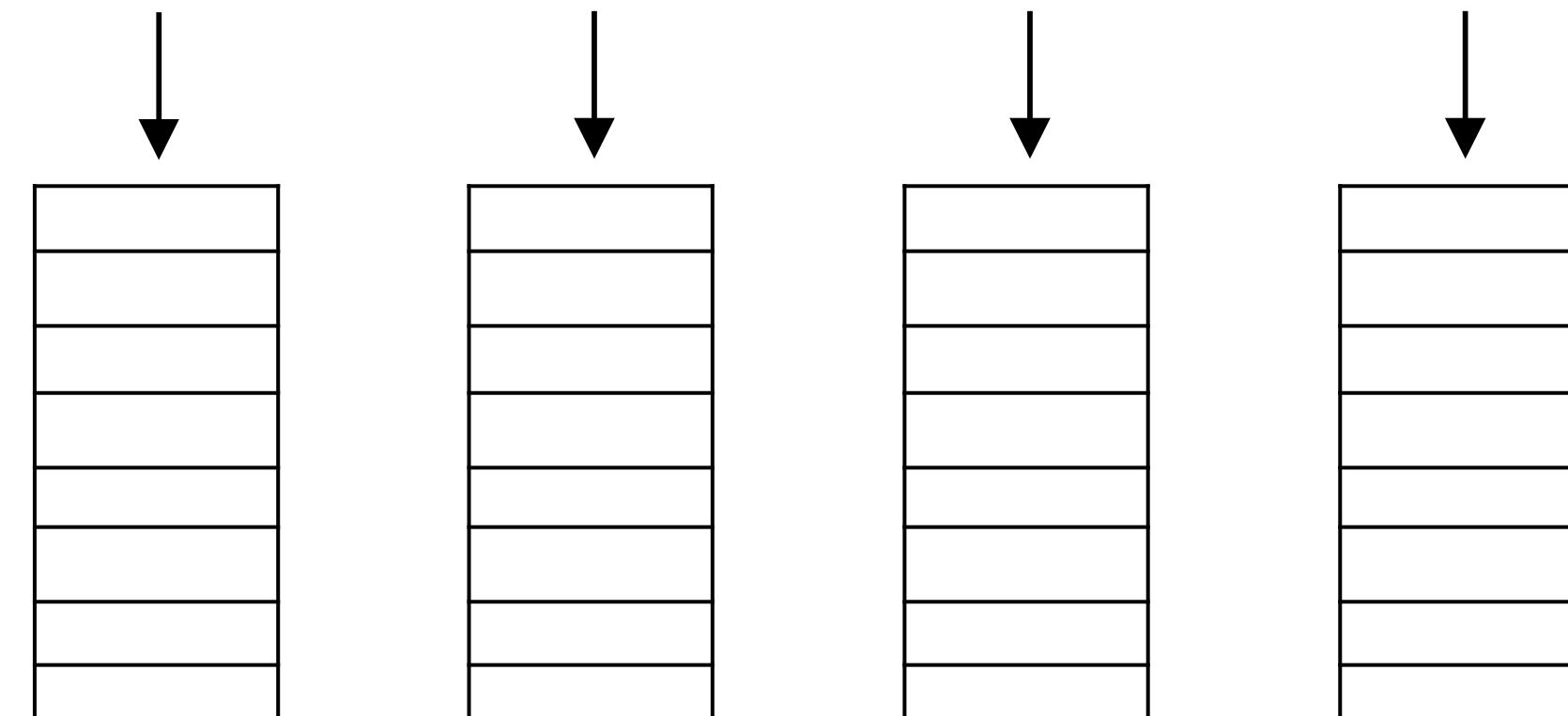
Subproblems

(a.k.a. “tasks”, “work to do”)



Set of work queues

(In general, one per worker thread)



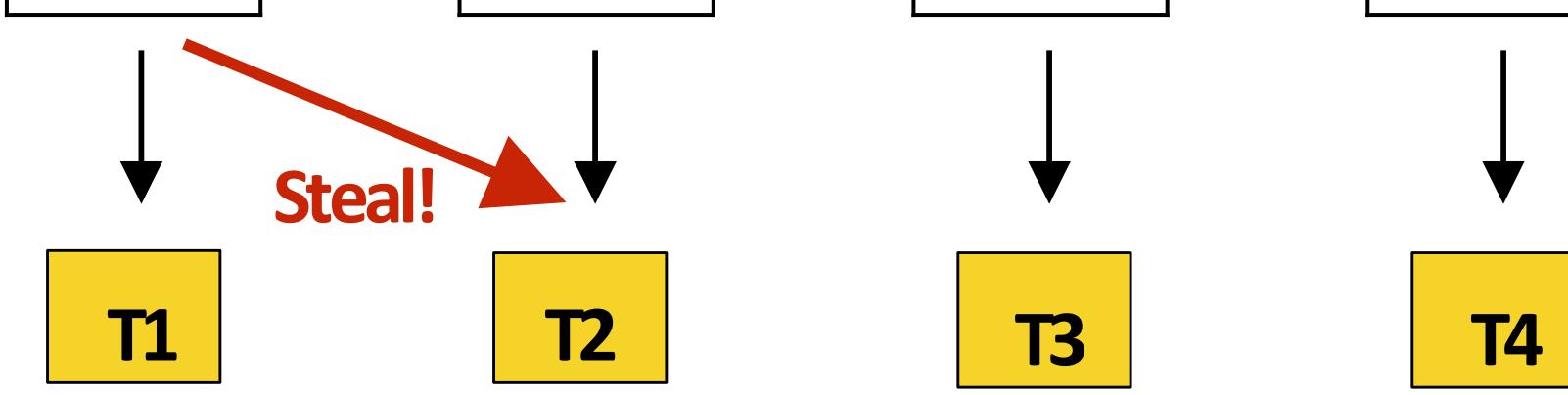
Worker threads:

Pull data from **OWN** work queue

Push new work to **OWN** work queue

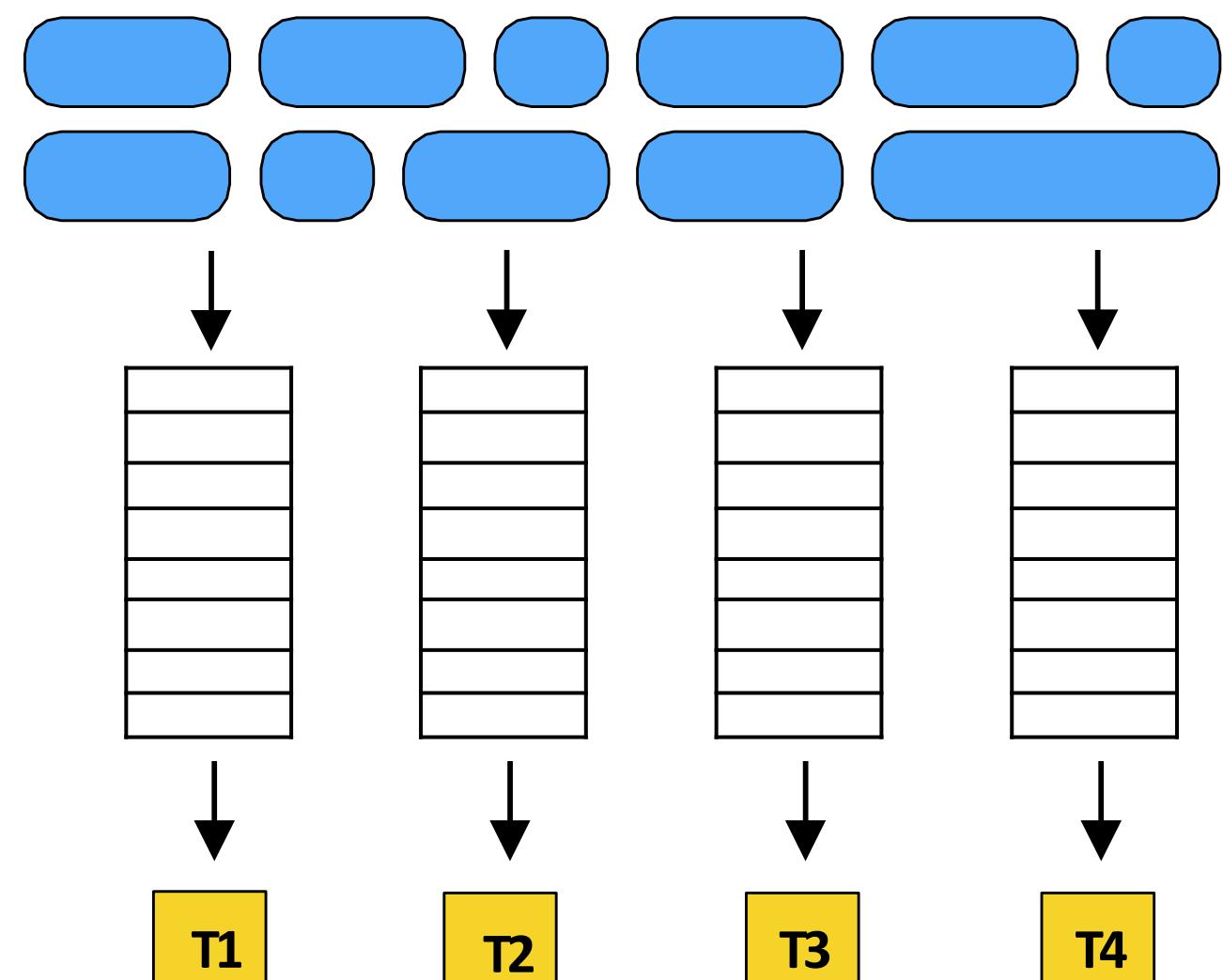
And when local work queue is empty?

STEAL work from another work queue!



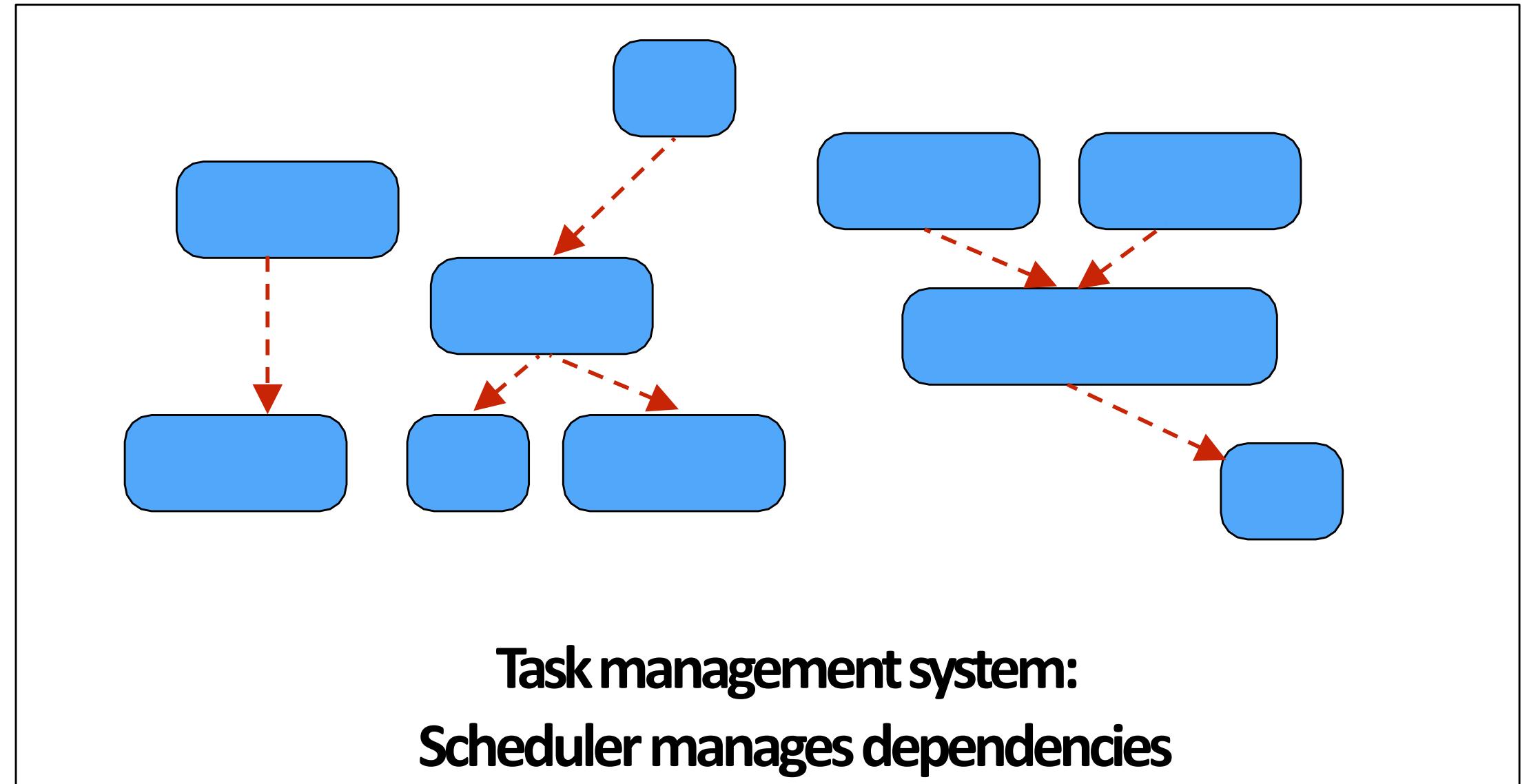
Distributed work queues

- **Costly synchronization/communication occurs during stealing**
 - But not every time a thread takes on new work
 - **Stealing occurs only when necessary to ensure good load balance**
- **Leads to increased locality**
 - Common case: threads work on tasks they create (producer-consumer locality)
- **Implementation challenges**
 - Who to steal from?
 - How much to steal?
 - How to detect program termination?
 - Ensuring local queue access is fast
(while preserving mutual exclusion)

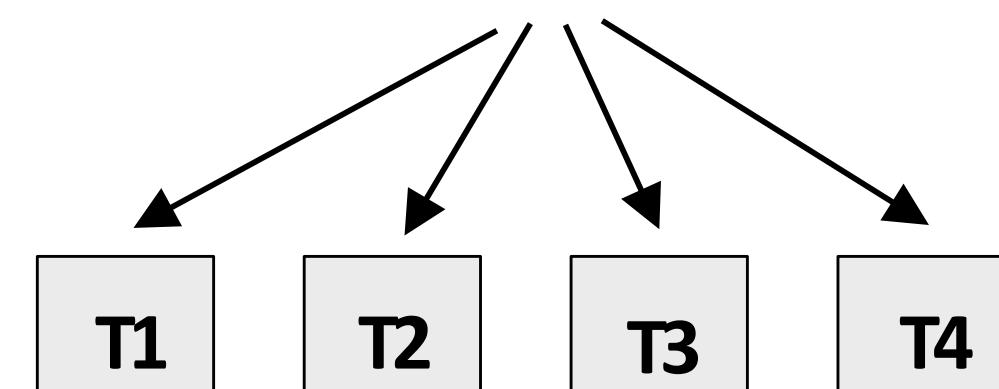


Workers in task queues need not be independent

→ = application-specified dependency



A task is not removed from queue and assigned to worker thread until all task dependencies are satisfied



Workers can submit new tasks (with optional explicit dependencies) to task system

```
foo_handle = enqueue_task(foo);           // enqueue task foo (independent of all prior tasks)
bar_handle = enqueue_task(bar, foo_handle); // enqueue task bar, cannot run until foo is complete
```

Summary

- **Challenge:** achieving good **workload balance**
 - Want all processors working all the time (otherwise, resources are idle!)
 - But want low-cost solution for achieving this balance
 - Minimize computational overhead (e.g., scheduling/assignment logic)
 - Minimize synchronization costs
- **Static assignment vs. dynamic assignment**
 - Really, it is not an either/or decision, there's a **continuum of choices**
 - Use up-front knowledge about workload as much as possible to reduce load imbalance and task management/synchronization costs (in the limit, if the system knows everything, use fully static assignment)
- Issues discussed today span decomposition, assignment, and orchestration

CSCI465/ECEN433

Introduction to Parallel Computing

Spring 2024



Lecture (9)

Performance Optimization Part 2:

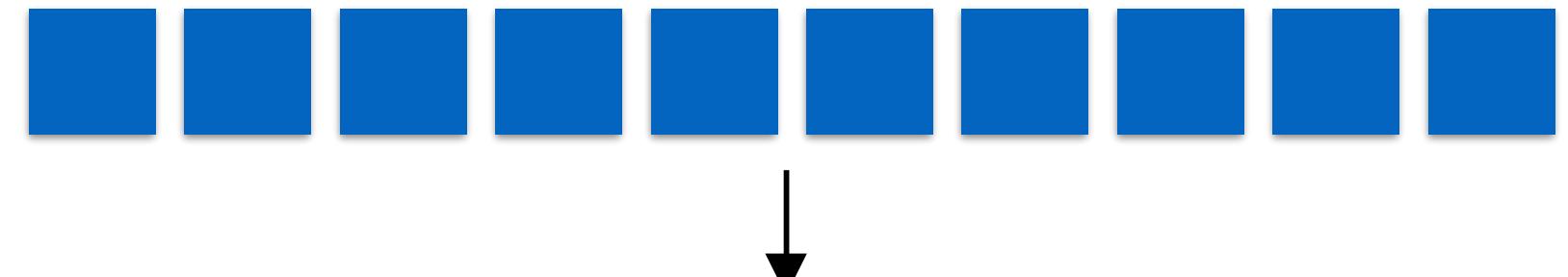
Scheduling

Common parallel programming patterns

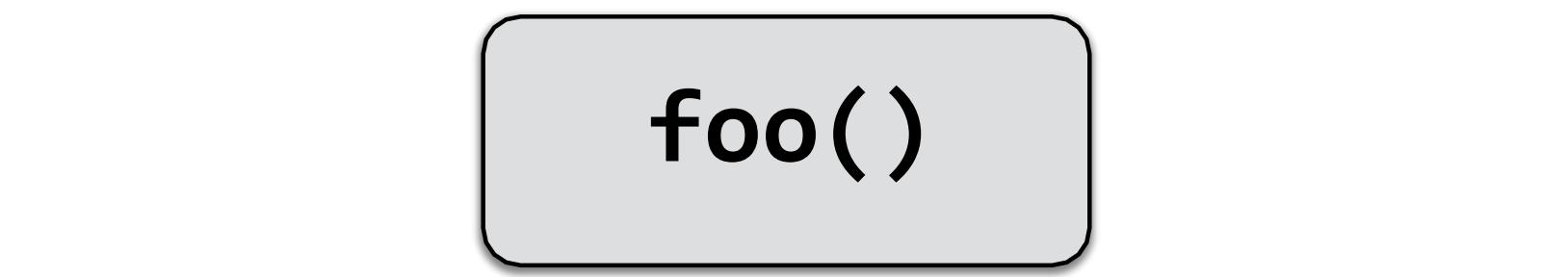
Data parallelism:

Perform same sequence of operations on many data elements

```
// ISPC foreach  
foreach (i=0 ... N) {  
    B[i] = foo(A[i]);  
}
```



```
// ISPC bulk task launch  
launch[numTasks] myFooTask(A, B);
```



```
// CUDA bulk launch  
foo<<<numBlocks, threadsPerBlock>>>(A, B);
```



```
// using higher-order function 'map'  
map(foo, A, B);
```

```
// openMP parallel for  
#pragma omp parallel for  
for (int i=0; i<N; i++) {  
    B[i] = foo(A[i]);  
}
```

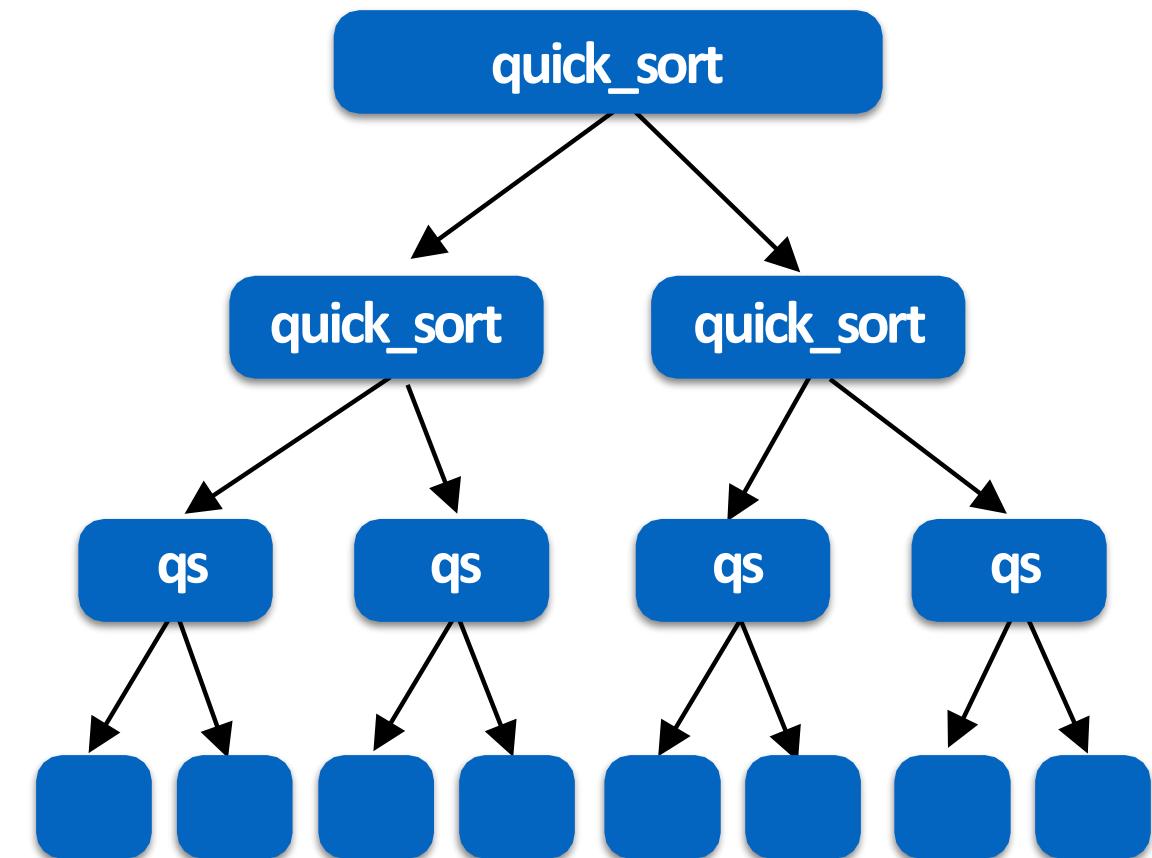
Consider divide-and-conquer algorithms

Quick sort:

```
// sort elements from 'begin' up to (but not including) 'end'  
void quick_sort(int* begin, int* end) {  
  
    if (begin >= end-1)  
        return;  
  
    else {  
  
        // choose partition key and partition elements  
        // by key, return position of key as `middle`  
        int* middle = partition(begin, end);  
  
        quick_sort(begin, middle);  
        quick_sort(middle+1, last);  
    }  
}
```

independent work!

Dependencies



Fork-join pattern

- Natural way to express independent work in divide-and-conquer algorithms
- This lecture's code examples will be in Cilk Plus
 - C++ language extension
 - Originally developed at MIT, acquired by Intel
 - But Intel is deprecating it. Best to stick with MIT version

`cilk_spawn foo(args);`

“fork” (create new logical thread of control)

Semantics: invoke `foo`, but unlike standard function call, caller **may continue executing asynchronously** with execution of `foo`.

`cilk_sync;`

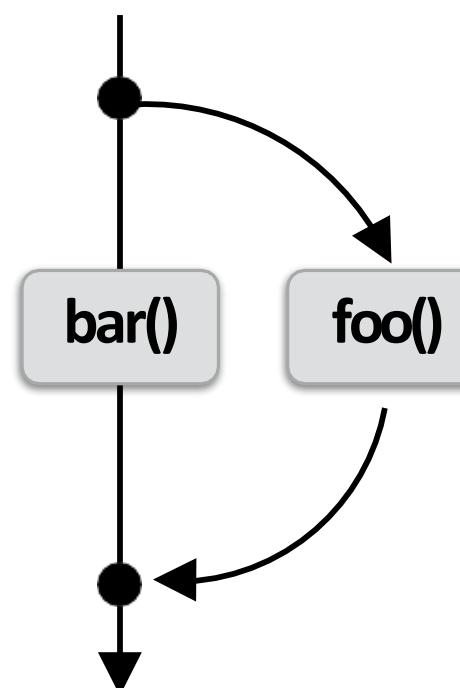
“join”

Semantics: returns when all calls spawned by current function have completed.
 (“sync up” with the spawned calls)

Note: there is an implicit `cilk_sync` at the end of every function that contains a `cilk_spawn` (implication: when a Cilk function returns, all work associated with that function is complete)

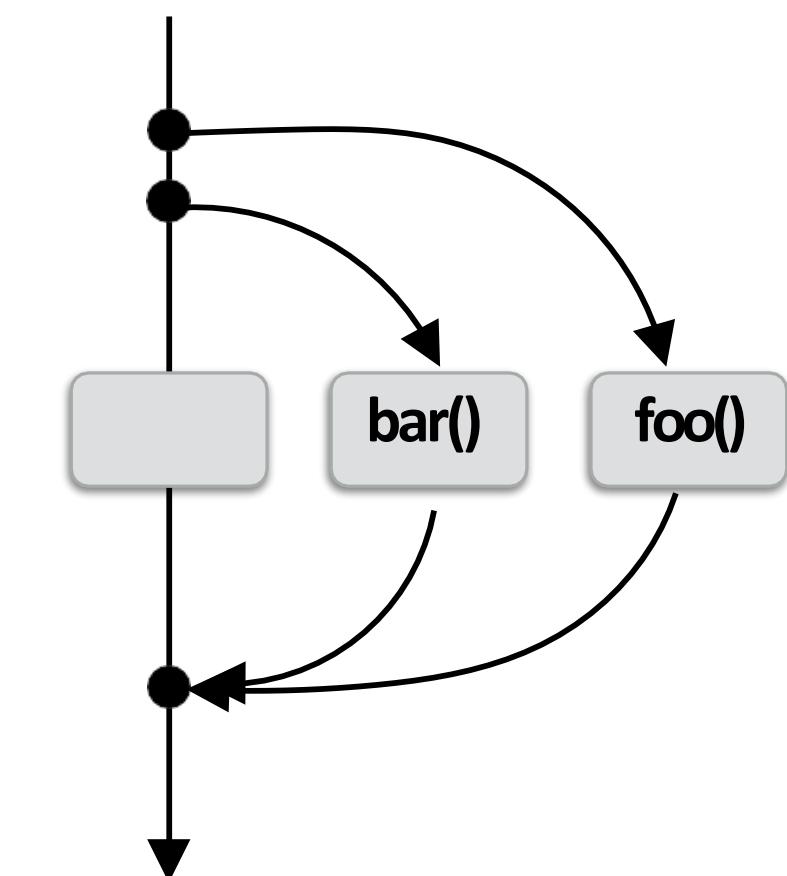
Basic Cilk Plus examples

```
// foo() and bar() may run in parallel
cilk_spawn foo();
bar();
cilk_sync;
```

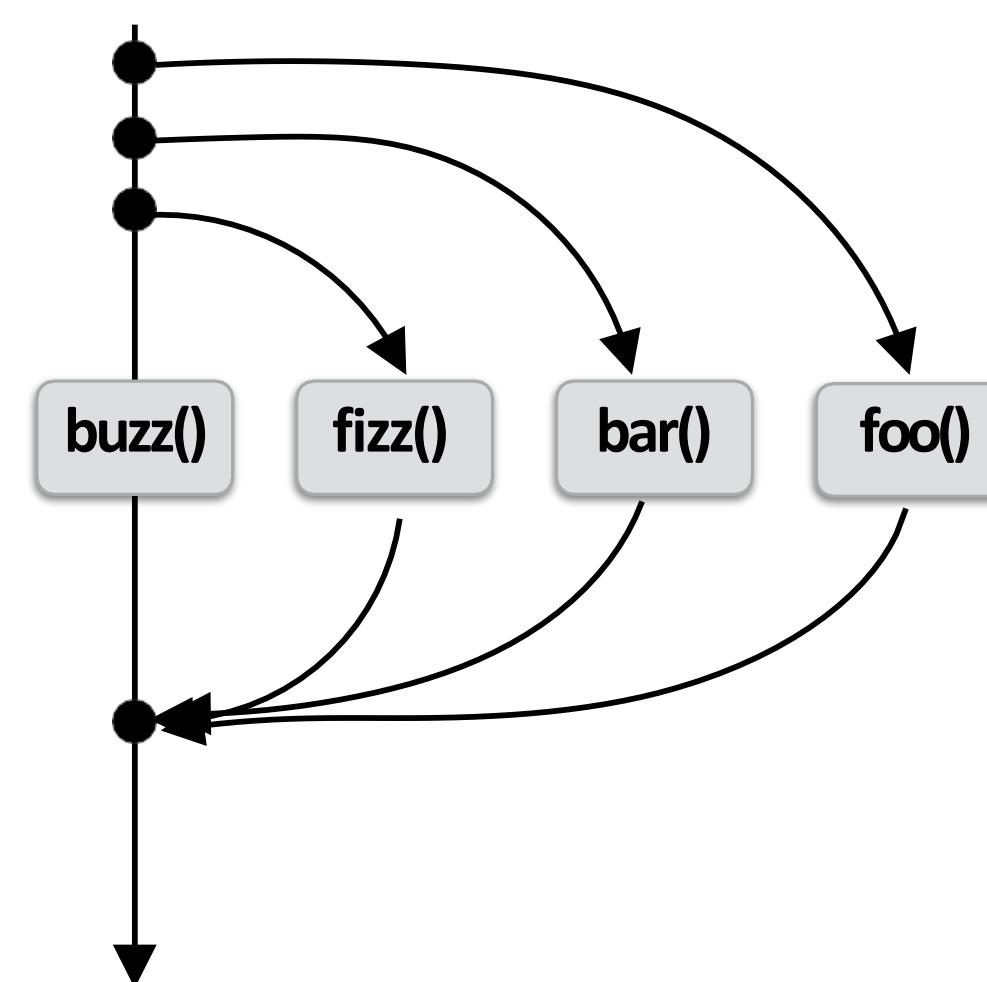


```
// foo() and bar() may run in parallel
cilk_spawn foo();
cilk_spawn bar();
cilk_sync;
```

Same amount of independent work first example, but potentially higher runtime overhead (due to two spawns vs. one)



```
// foo, bar, fizz, buzz, may run in parallel
cilk_spawn foo();
cilk_spawn bar();
cilk_spawn fizz();
buzz();
cilk_sync;
```



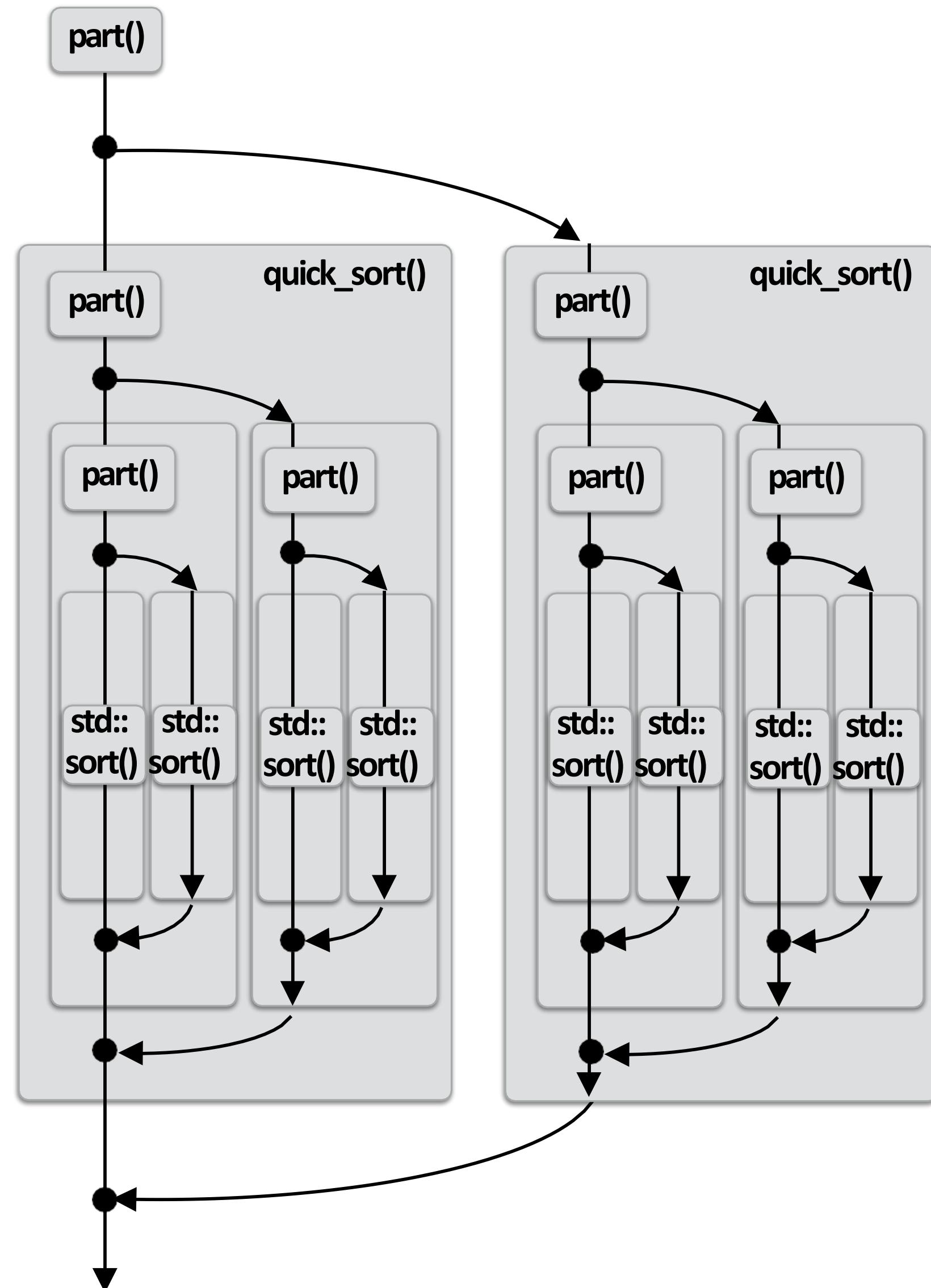
Abstraction vs. implementation

- Notice that the **cilk_spawn** abstraction does not specify how or when spawned calls are scheduled to execute
 - Only that they **may be run concurrently with caller** (and with all other calls spawned by the caller)
- But **cilk_sync** does serve as a **constraint on scheduling**
 - All spawned calls must complete before **cilk_sync** returns

Parallel quicksort in Cilk Plus

```
void quick_sort(int* begin, int* end) {  
    if (begin >= end - PARALLEL_CUTOFF)  
        std::sort(begin, end);  
  
    else {  
        int* middle = partition(begin, end);  
        cilk_spawn quick_sort(begin, middle);  
        quick_sort(middle+1, last);  
    }  
}
```

Sort sequentially if problem size is sufficiently small (overhead of spawn trumps benefits of potential parallelization)



Writing fork-join programs

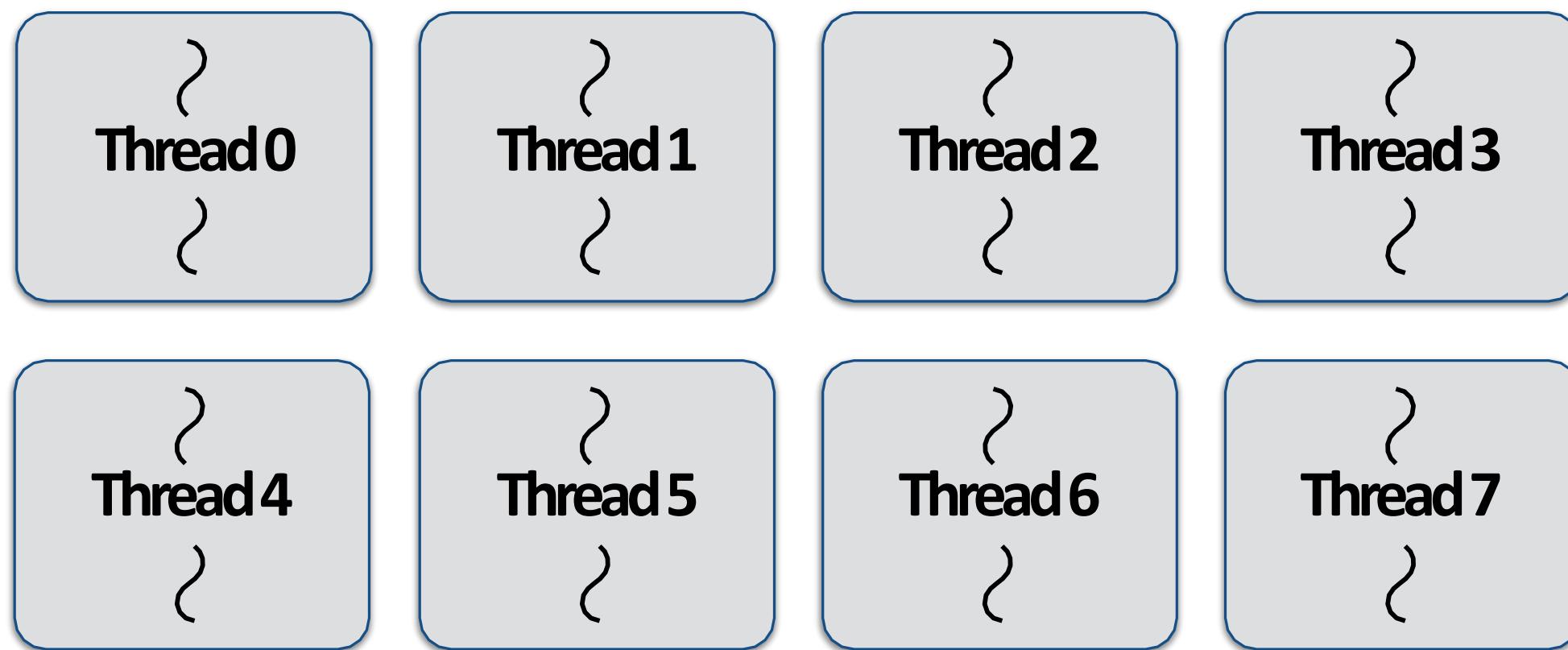
- **Main idea:** expose independent work (potential parallelism) to the system using `cilk_spawn`
- Recall parallel programming rules of thumb
 - Want at least as much work as parallel execution capability (e.g., program should probably spawn at least as much work as there are cores)
 - Want more independent work than execution capability to allow for good workload balance of all the work onto the cores
 - “**parallel slack**” = ratio of independent work to machine’s parallel execution capability (in practice: ~8 is a good ratio)
 - But not too much independent work so that **granularity of work is too small** (too much slack incurs overhead of managing fine-grained work)

Scheduling fork-join programs

- Consider very simple scheduler:
 - Launch `pthread` for each `cilk_spawn` using `pthread_create`
 - Translate `cilk_sync` into appropriate `pthread_join` calls
- Potential performance problems?
 - Heavyweight spawn operation
 - Many more concurrently running threads than cores
 - Context switching overhead
 - Larger working set than necessary, less cache locality

Pool of worker threads

- The Cilk Plus runtime maintains pool of worker threads
 - Think: all threads created at application launch *
 - Exactly as many worker threads as execution contexts in the machine



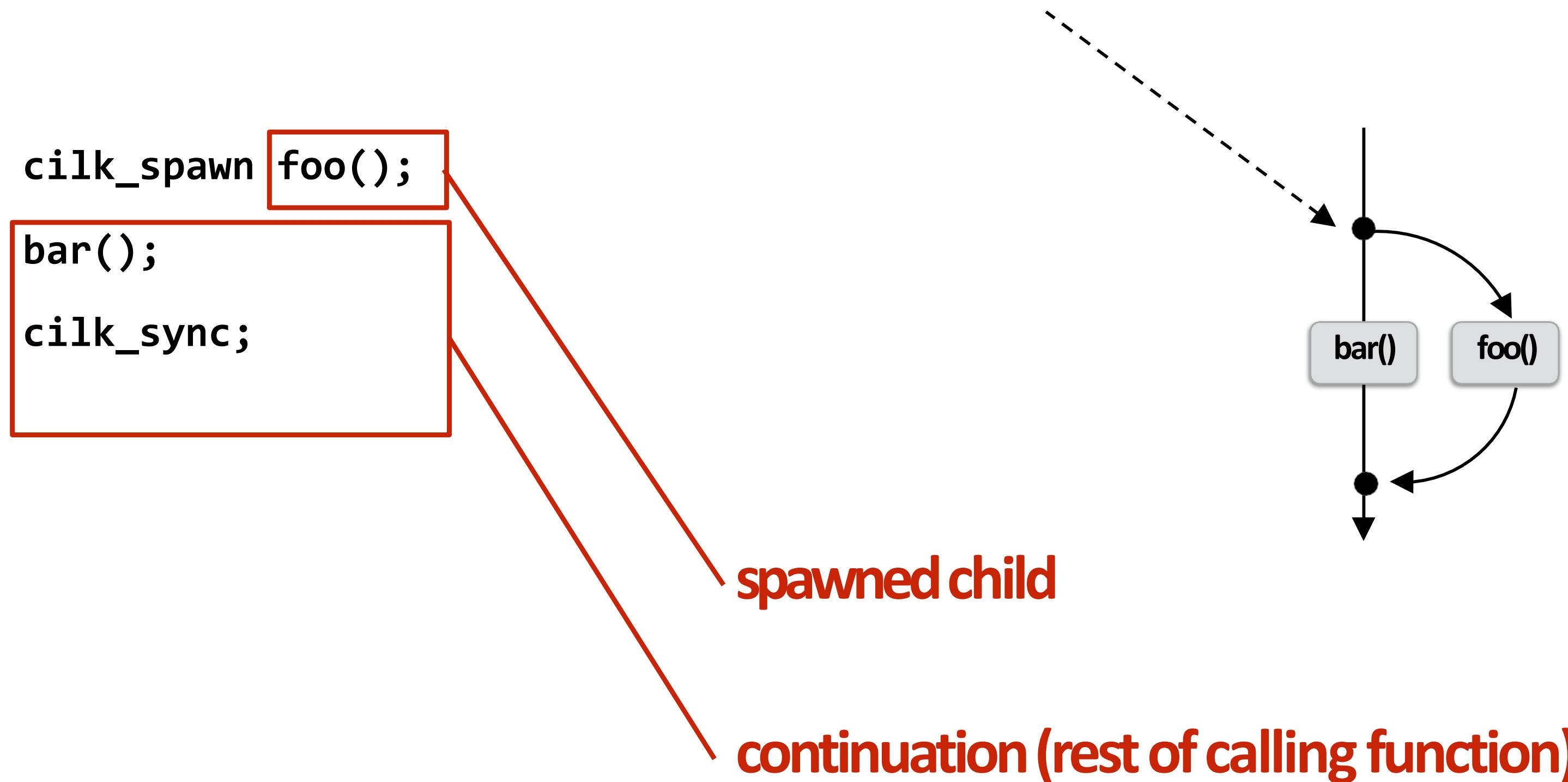
Example: Eight thread worker pool for a quad-core laptop with Hyper-Threading

```
while (work_exists()) {  
    work = get_new_work();  
    work.run();  
}
```

* It's perfectly fine to think about it this way, but in reality, runtimes tend to be lazy and initialize worker threads on the first Cilk spawn. (This is a common implementation strategy, ISPC does the same with worker threads that run ISPC tasks.)

Consider execution of the following code

Specifically, consider execution from the point `foo()` is spawned



What threads should `foo()` and `bar()` be executed by?

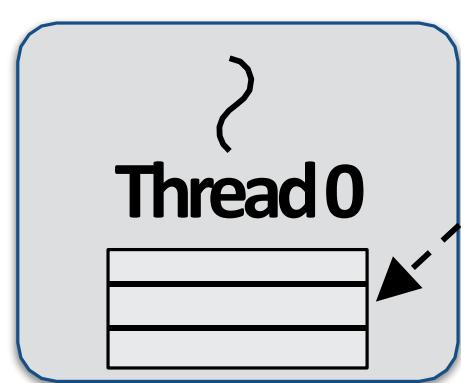
Thread 0

Thread 1

First, consider a **serial** implementation

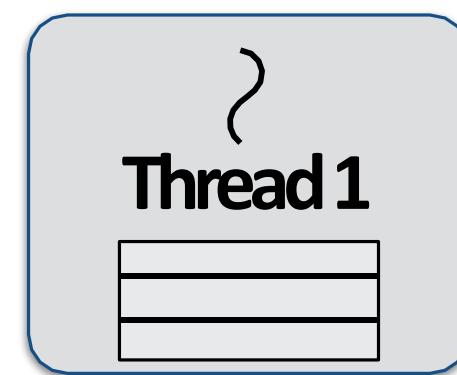
Run child first... via a regular function call

- Thread runs foo(), then returns from foo(), then runs bar()
- Continuation is implicit in the thread's stack



Executing foo()...

Traditional thread call stack
(indicates bar will be performed
next after return)

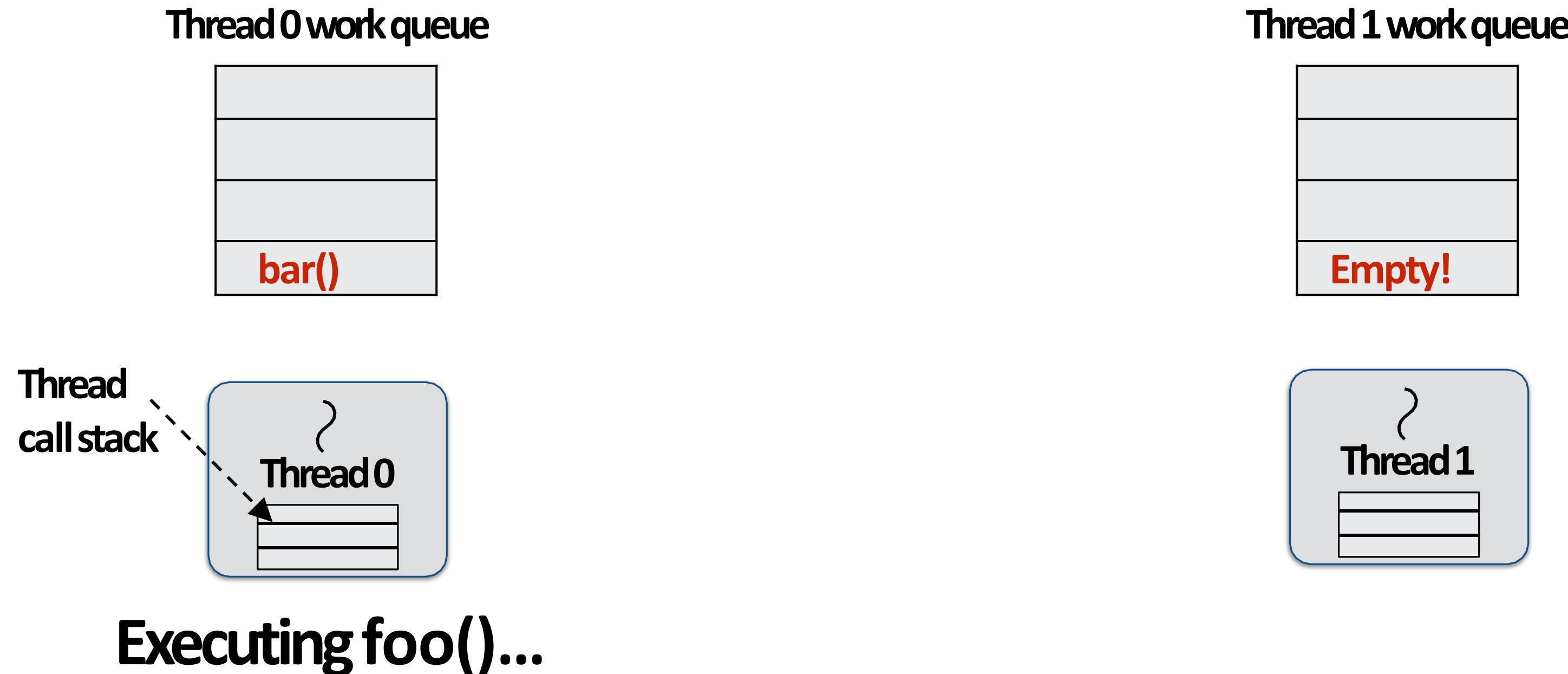


**What if, while executing foo(),
thread 1 goes idle...**

Inefficient: thread 1 could be
performing bar() at this time!

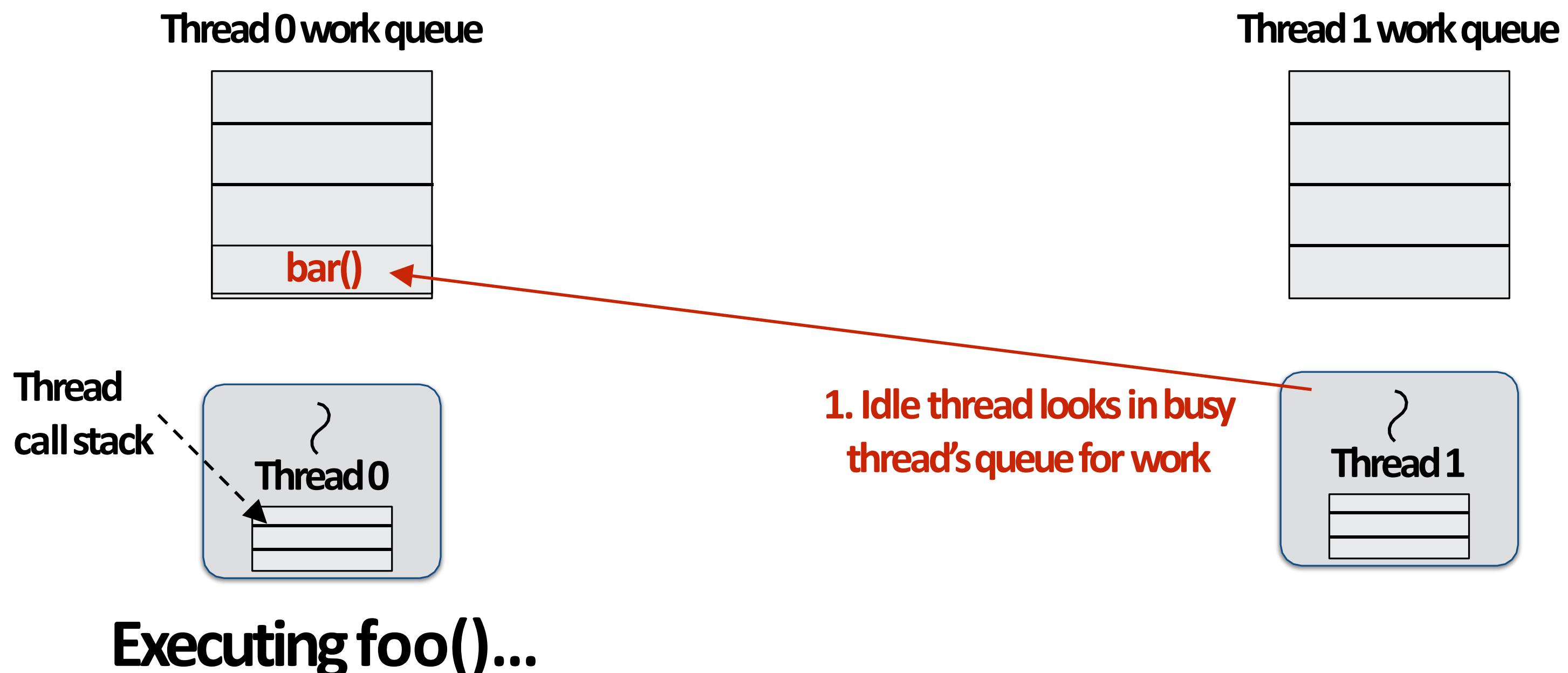
Per-thread workqueues store “work to do”

Upon reaching `cilk_spawn foo()`, thread places continuation in its workqueue, and begins executing `foo()`.



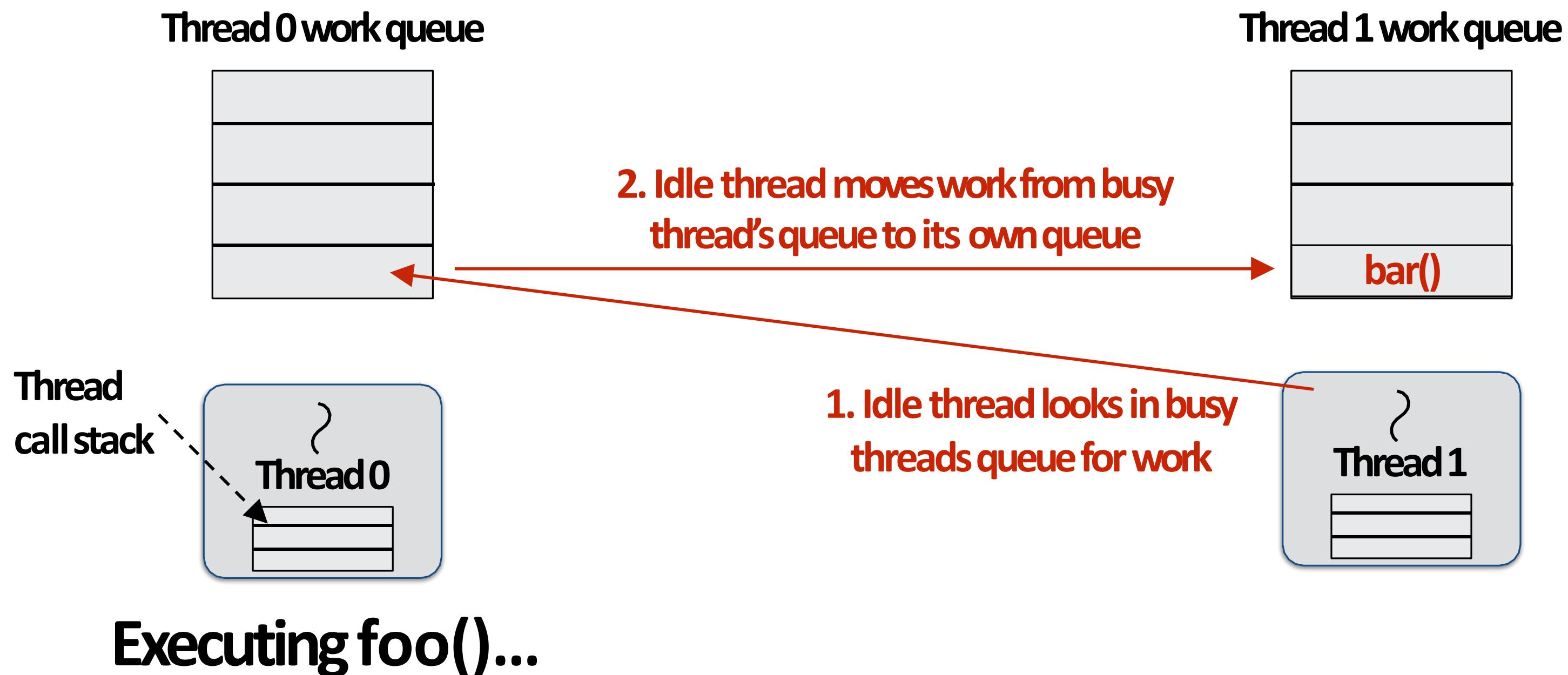
Idle threads “steal” work from busy threads

If **thread 1 goes idle** (a.k.a. there is no work in its own queue), then it **looks in thread 0's queue for work to do.**



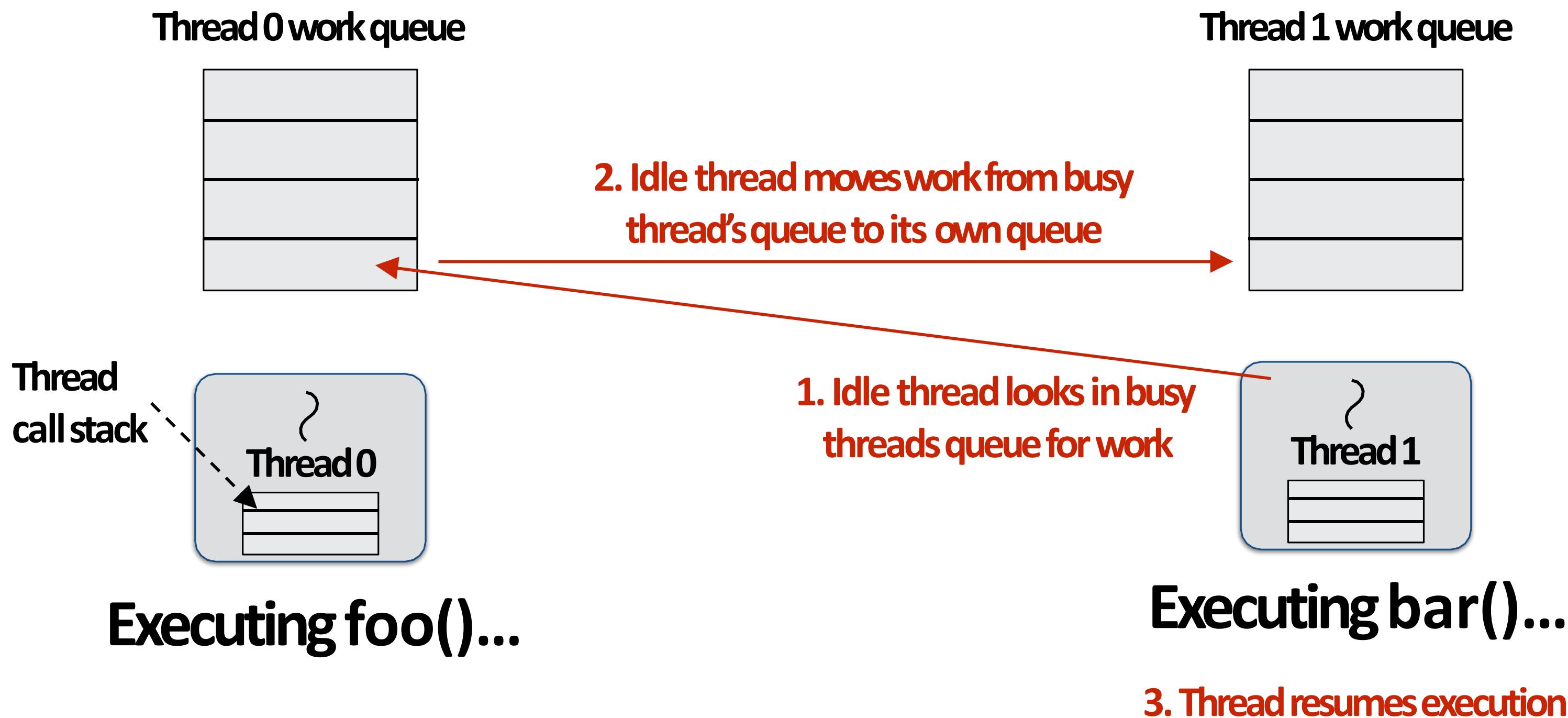
Idle threads “steal” work from busy threads

If **thread 1 goes idle** (a.k.a. there is no work in its own queue), then it **looks in thread 0's queue for work to do.**



Idle threads “steal” work from busy threads

If **thread 1 goes idle** (a.k.a. there is no work in its **own queue**), then it **looks in thread 0's queue for work to do.**



At spawn, should thread run child or continuation?

```
cilk_spawn foo();  
bar();  
cilk_sync;
```

spawned child

continuation (rest of calling function)

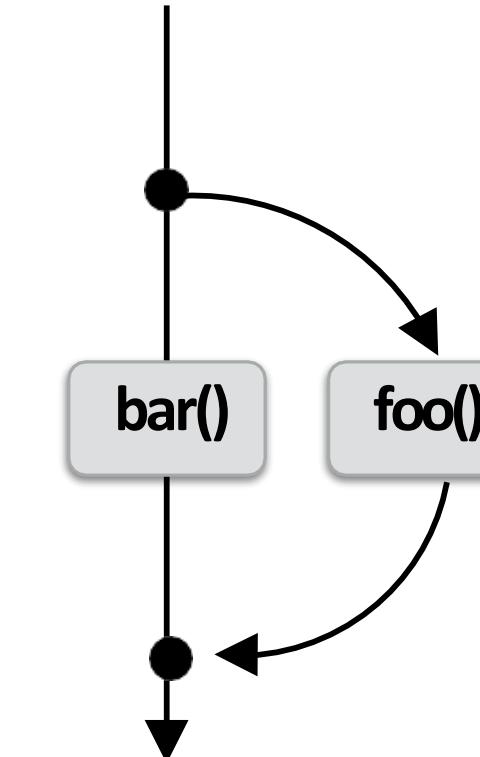
Run **continuation first**: record child for later execution

- Child is made available for stealing by other threads (“**child stealing**”)

Run **child first**: record continuation for later execution

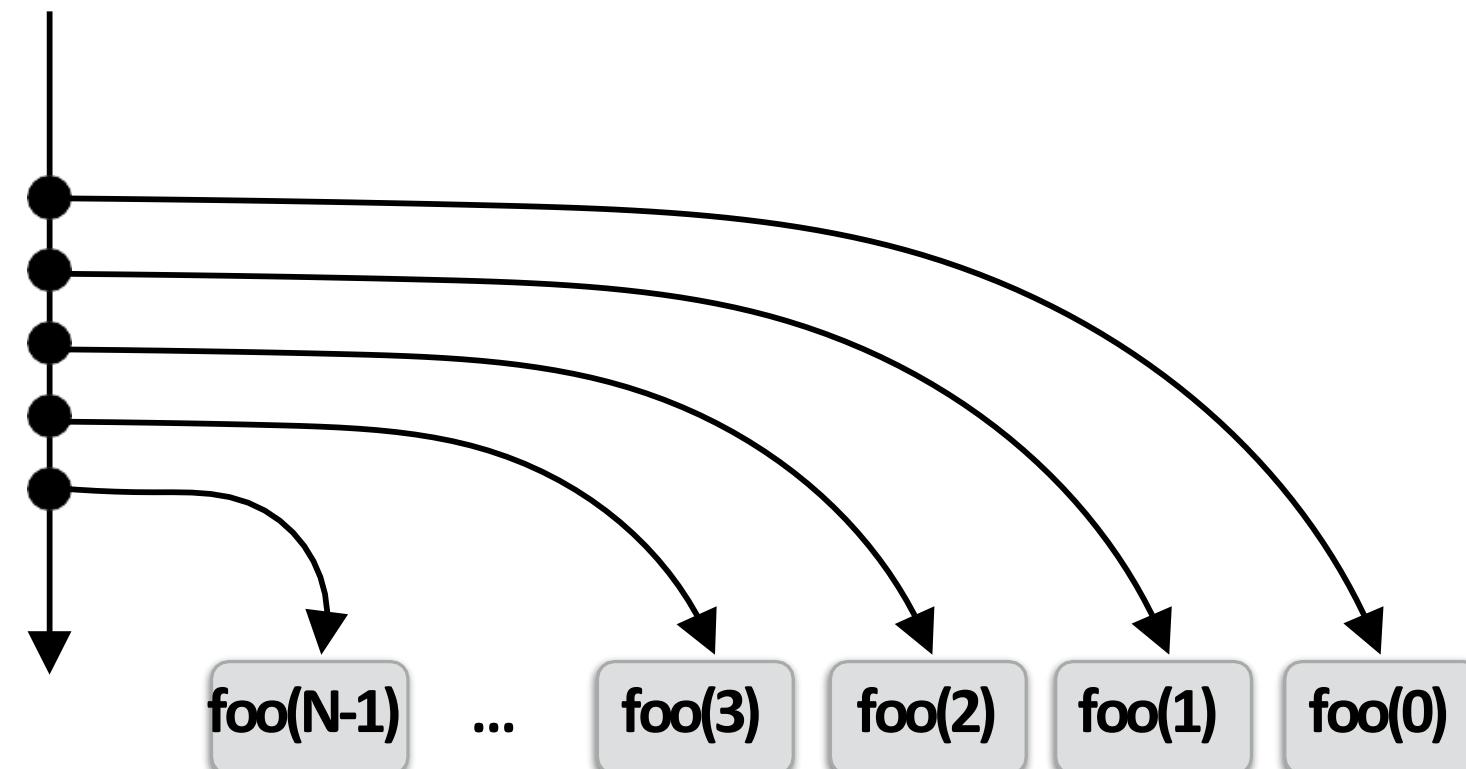
- Continuation is made available for stealing by other threads (“**continuation stealing**”)

Which implementation do we choose?



Consider thread executing the following code

```
for (int i=0; i<N; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync;
```

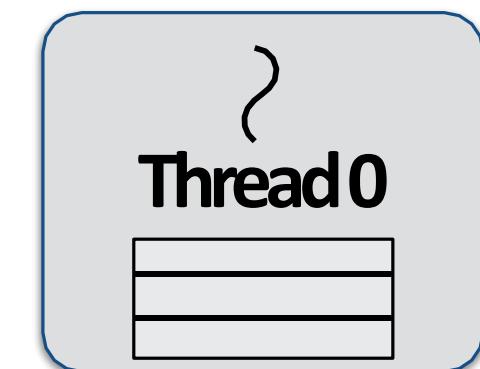


- Run **continuation first (“child stealing”)**

- Caller thread spawns work for all iterations before executing any of it
- Think: **breadth-first traversal of call graph. O(N)** space for spawned work (maximum space)
- If no stealing, **execution order is very different than that of program with cilk_spawn removed**

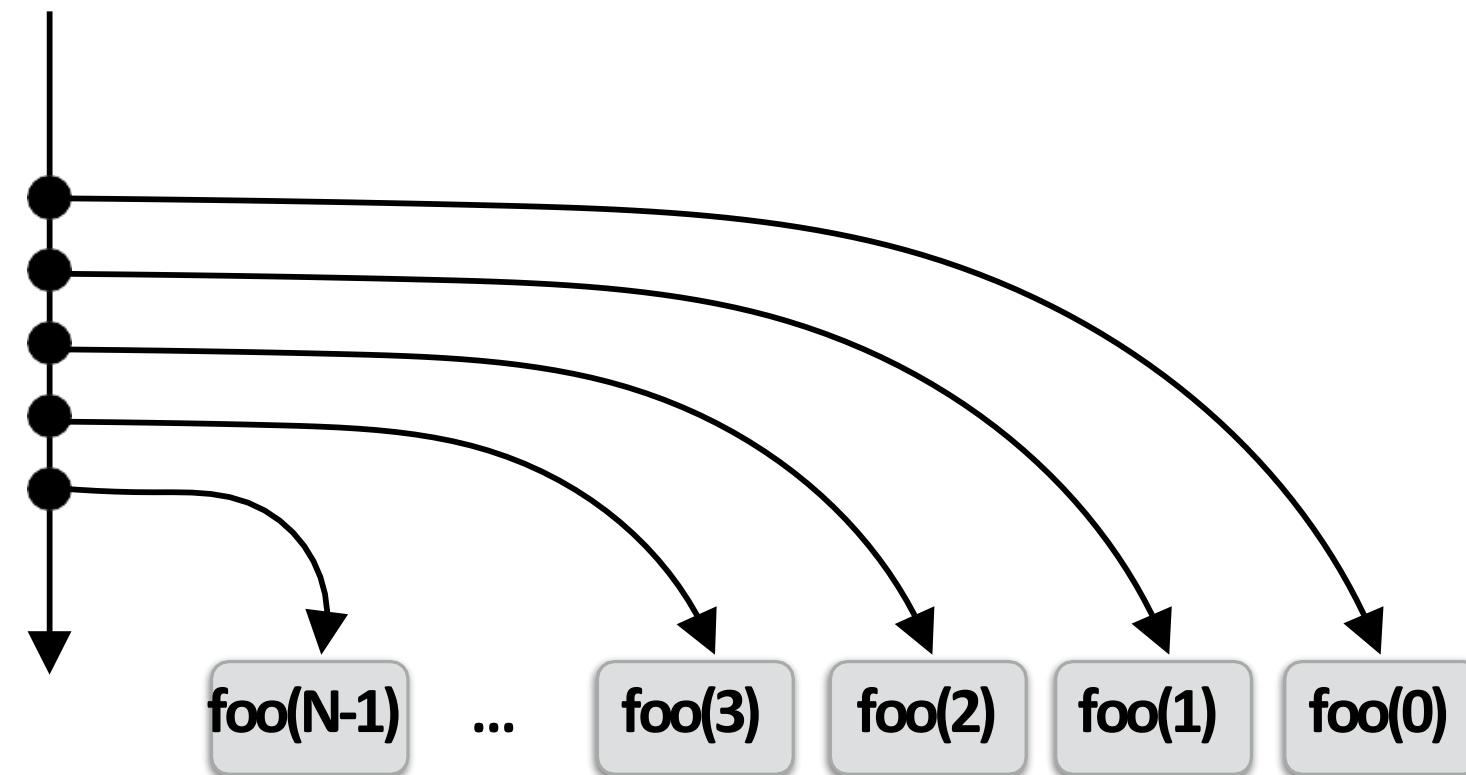
Thread 0 work queue

foo(0)
...
foo(N-2)
foo(N-1)



Consider thread executing the following code

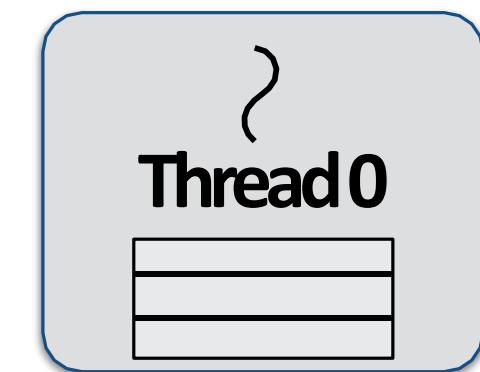
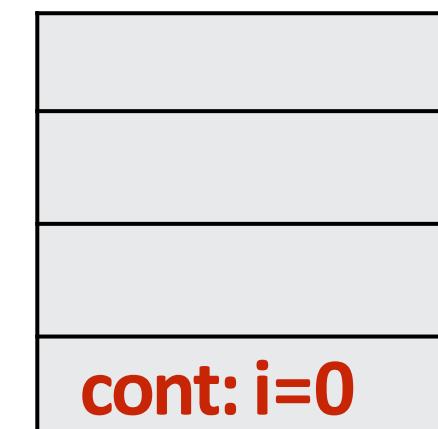
```
for (int i=0; i<N; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync;
```



- Run **child first** (“continuation stealing”)

- Caller thread only creates one item to steal (continuation that represents all remaining iterations)
- If no stealing occurs, thread continually pops continuation from work queue, enqueues new continuation (with updated value of `i`)
- Order of execution is the same as for program with `spawn` removed.
- Think: depth-first traversal of call graph

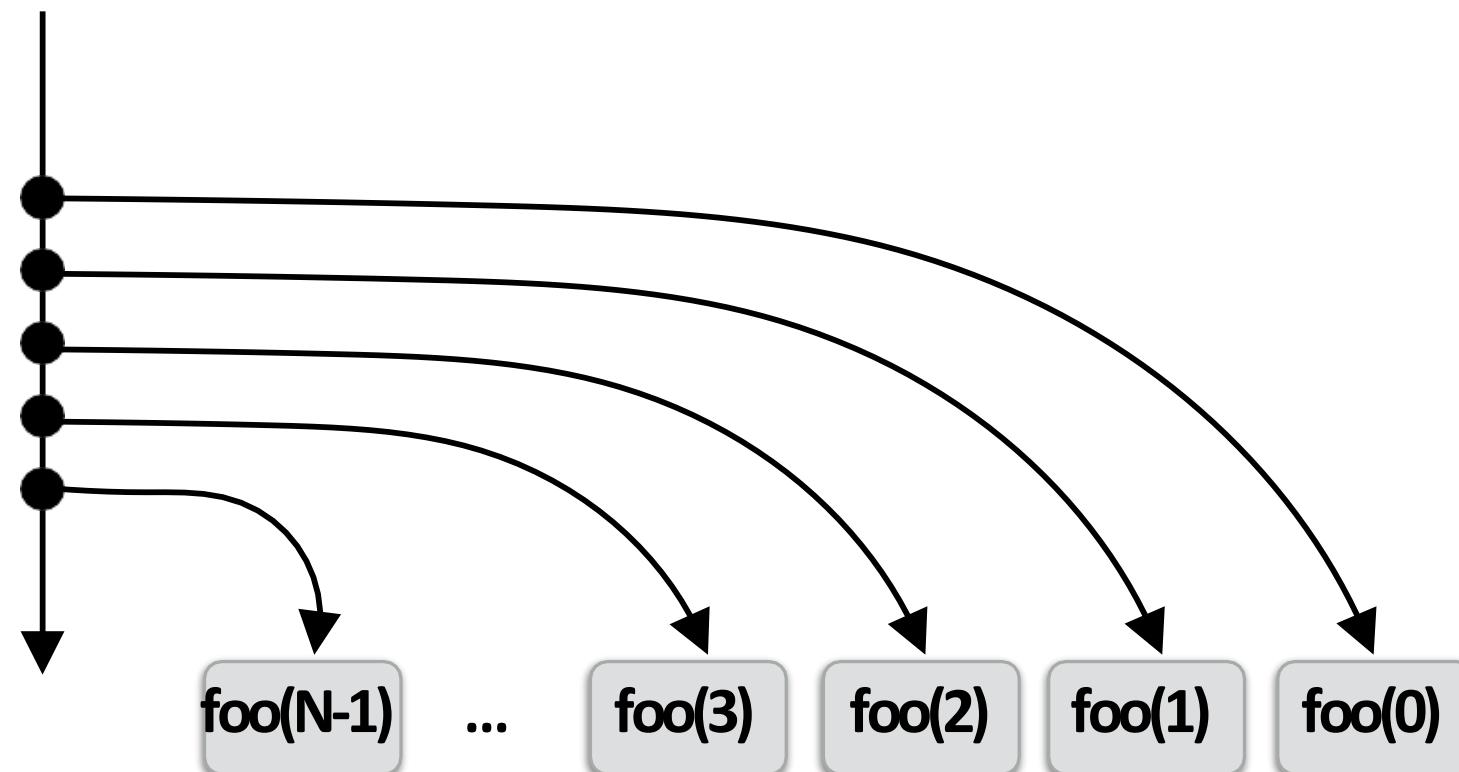
Thread 0 work queue



Executing `foo(0)...`

Consider thread executing the following code

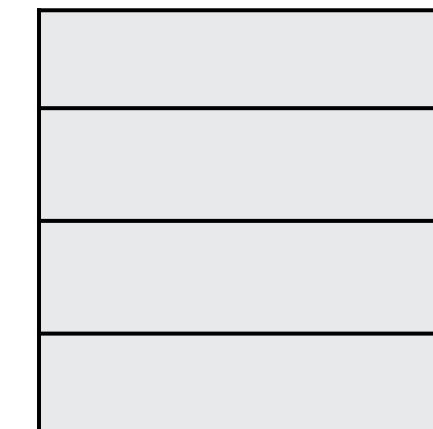
```
for (int i=0; i<N; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync;
```



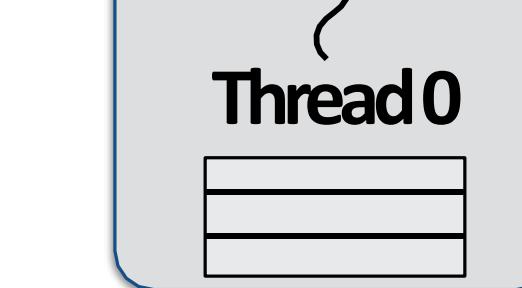
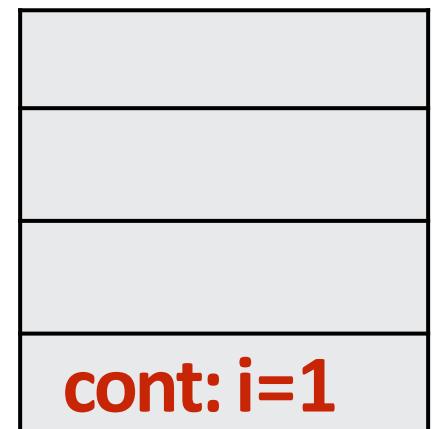
- Run child first (“continuation stealing”)

- If continuation is stolen, stealing thread spawns and executes next iteration
- Enqueues continuation with i advanced by 1
- Can prove that work queue storage for system with T threads is no more than T times that of stack storage for single threaded execution

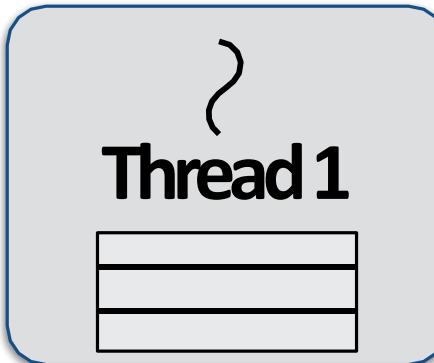
Thread 0 work queue



Thread 1 work queue



Executing foo(0)...



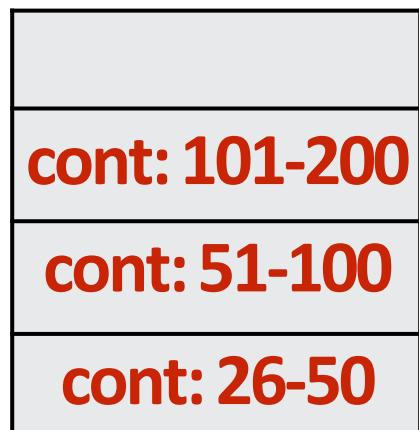
Executing foo(1)...

Scheduling quicksort: assume 200 elements

```
void quick_sort(int* begin, int* end) {  
    if (begin >= end - PARALLEL_CUTOFF)  
        std::sort(begin, end);  
    else {  
        int* middle = partition(begin, end);  
        cilk_spawn quick_sort(begin, middle);  
        quick_sort(middle+1, last);  
    }  
}
```

What work in the queue
should other threads steal?

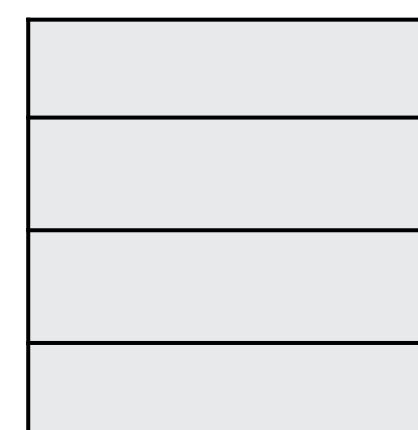
Thread 0 work queue



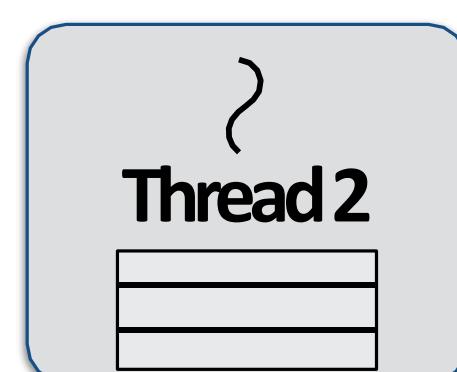
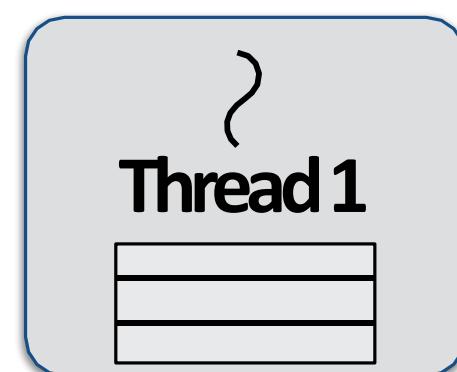
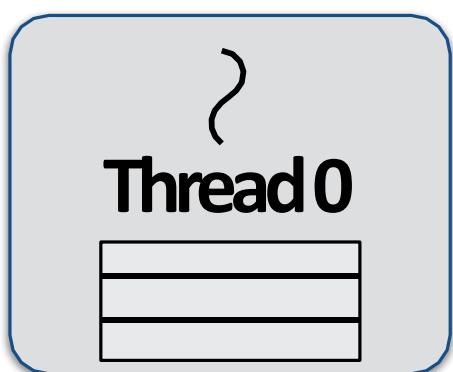
Thread 1 work queue



Thread 2 work queue



...

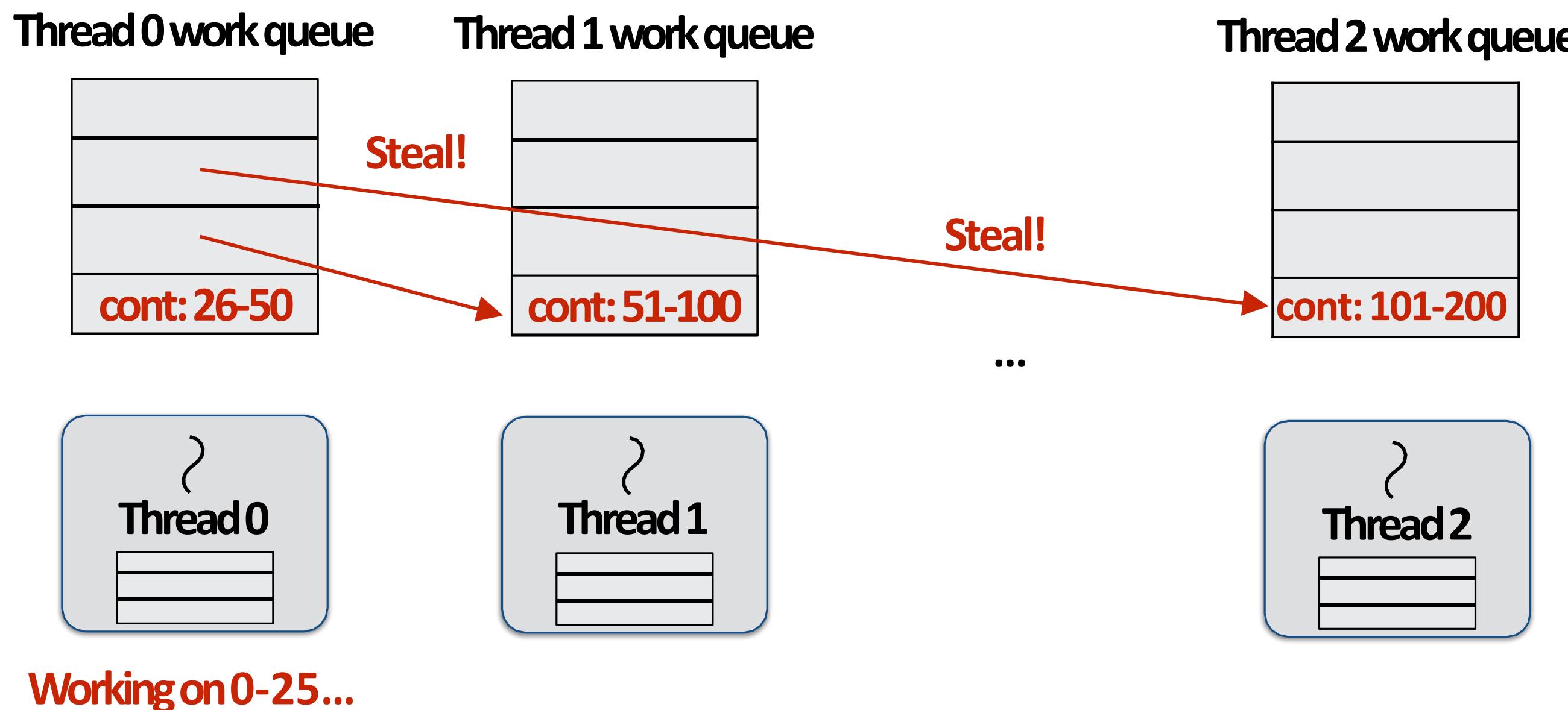


Working on 0-25...

Implementing work stealing: dequeue per worker

Workqueue implemented as a **dequeue (double ended queue)**

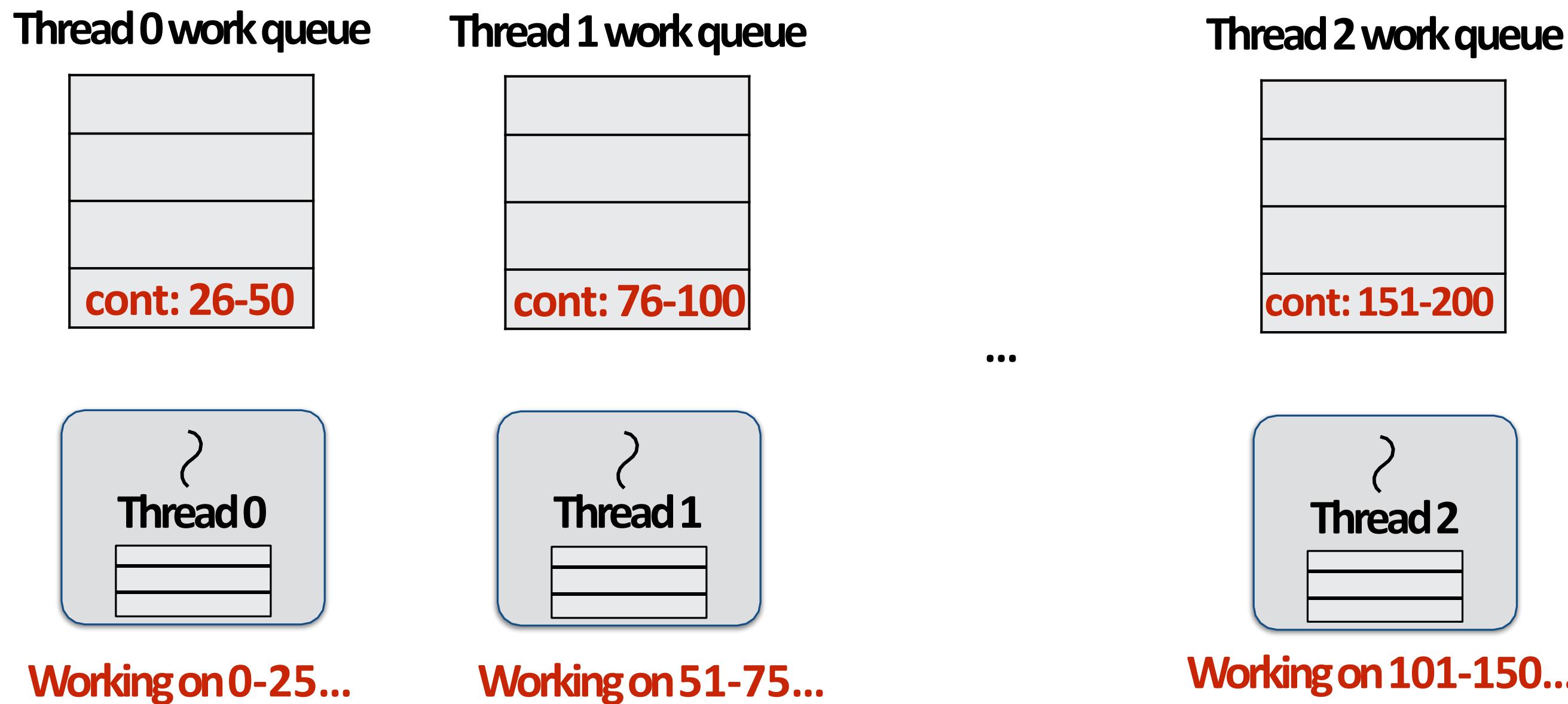
- Local thread pushes/pops from the “tail” (bottom)
- Remote threads **steal from “head”** (top)
- Efficient lock-free dequeue implementations exist



Implementing work stealing: dequeue per worker

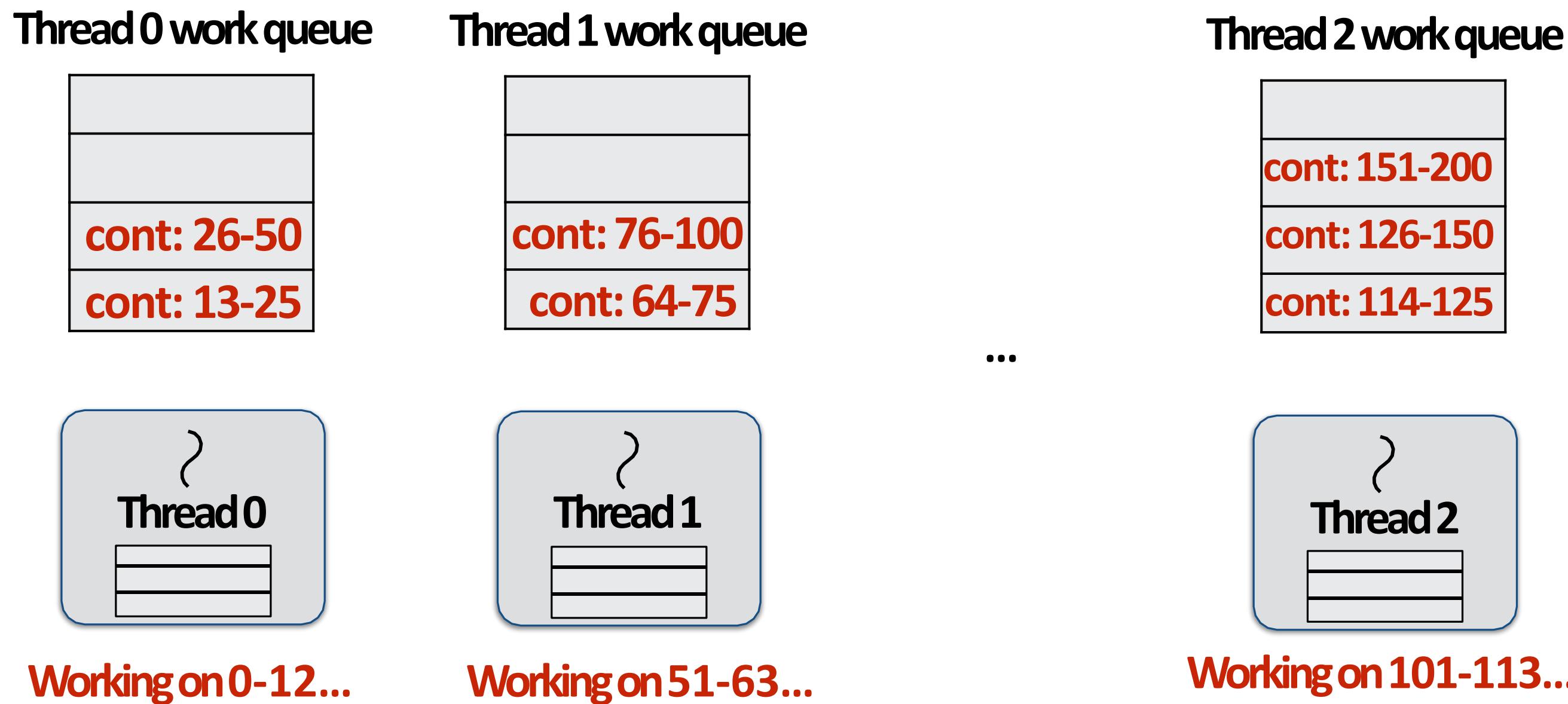
Workqueue implemented as a **dequeue (double ended queue)**

- Local thread pushes/pops from the “tail” (bottom)
- Remote threads **steal from “head”** (top)
- Efficient lock-free dequeue implementations exist



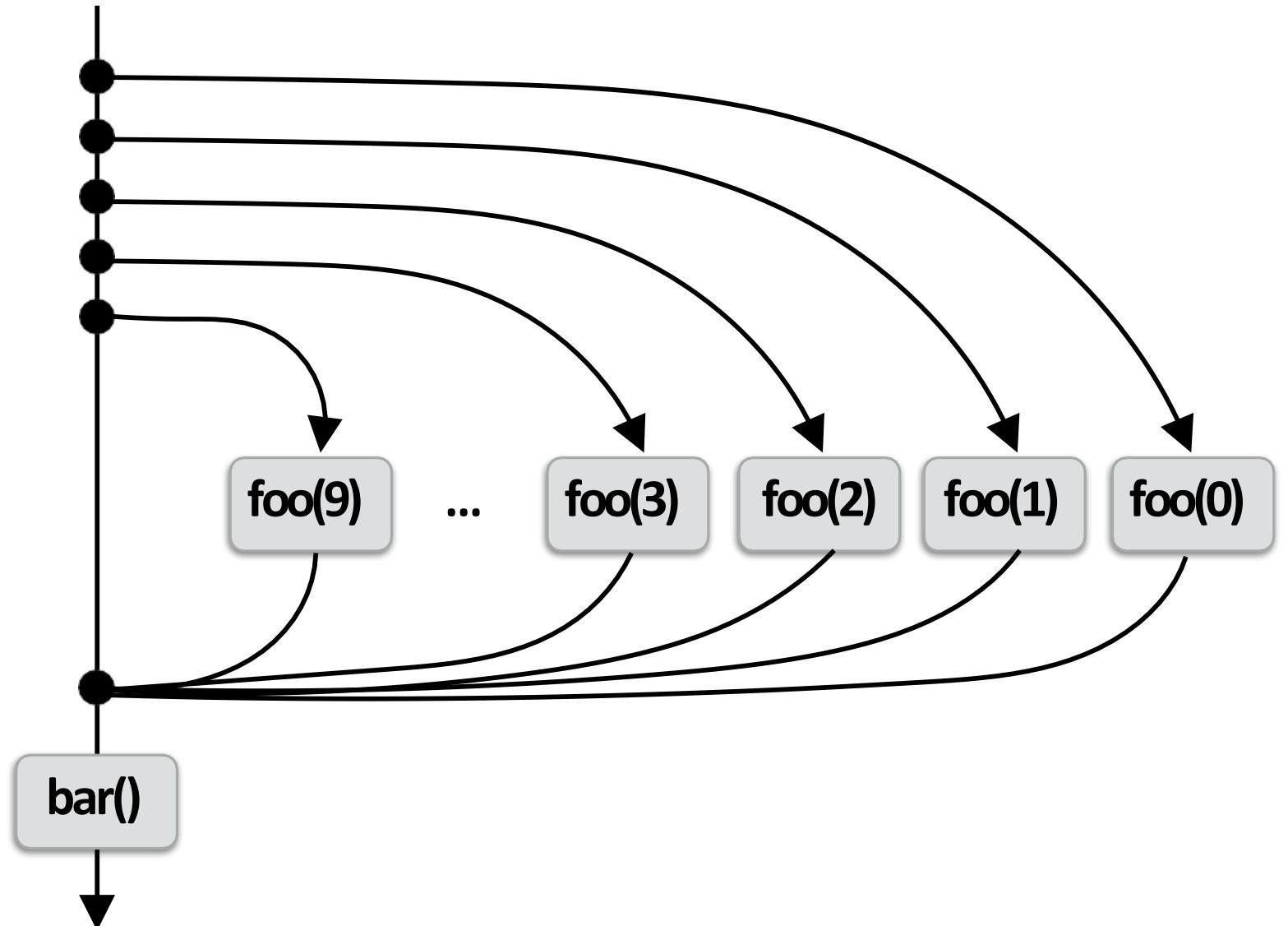
Implementing work stealing: random choice of victim

- Idle threads randomly choose a thread to attempt to steal from
 - Stealing from top of deque...
- Reduces contention with local thread: local thread is not accessing same part of deque that stealing threads do!
 - Steals work at beginning of call tree: this is a “larger” piece of work, so the cost of performing a steal is amortized over longer future computation
 - Maximizes locality: (in conjunction with run-child-first policy) local thread works on local part of call tree

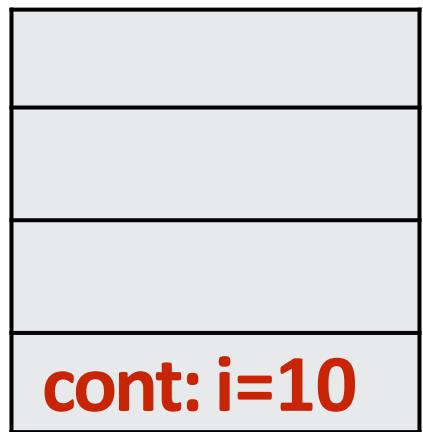


Implementing sync

```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
  
cilk_sync;  
  
bar();
```

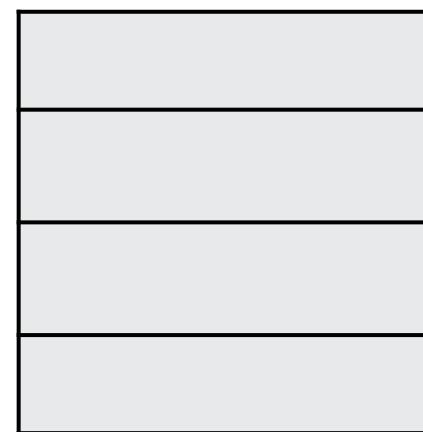


Thread 0 work queue



cont: i=10

Thread 1 work queue



Thread 0

Thread 1

Working on foo(9)...

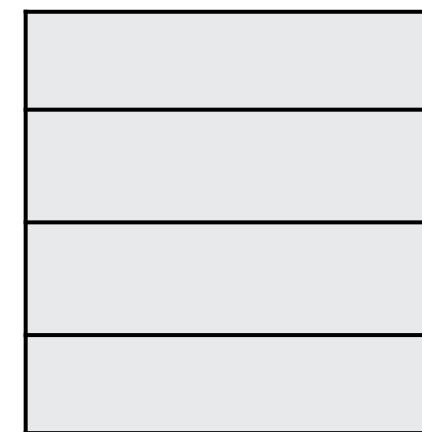
Thread 2 work queue



Thread 2

Working on foo(8)...

Thread 3 work queue



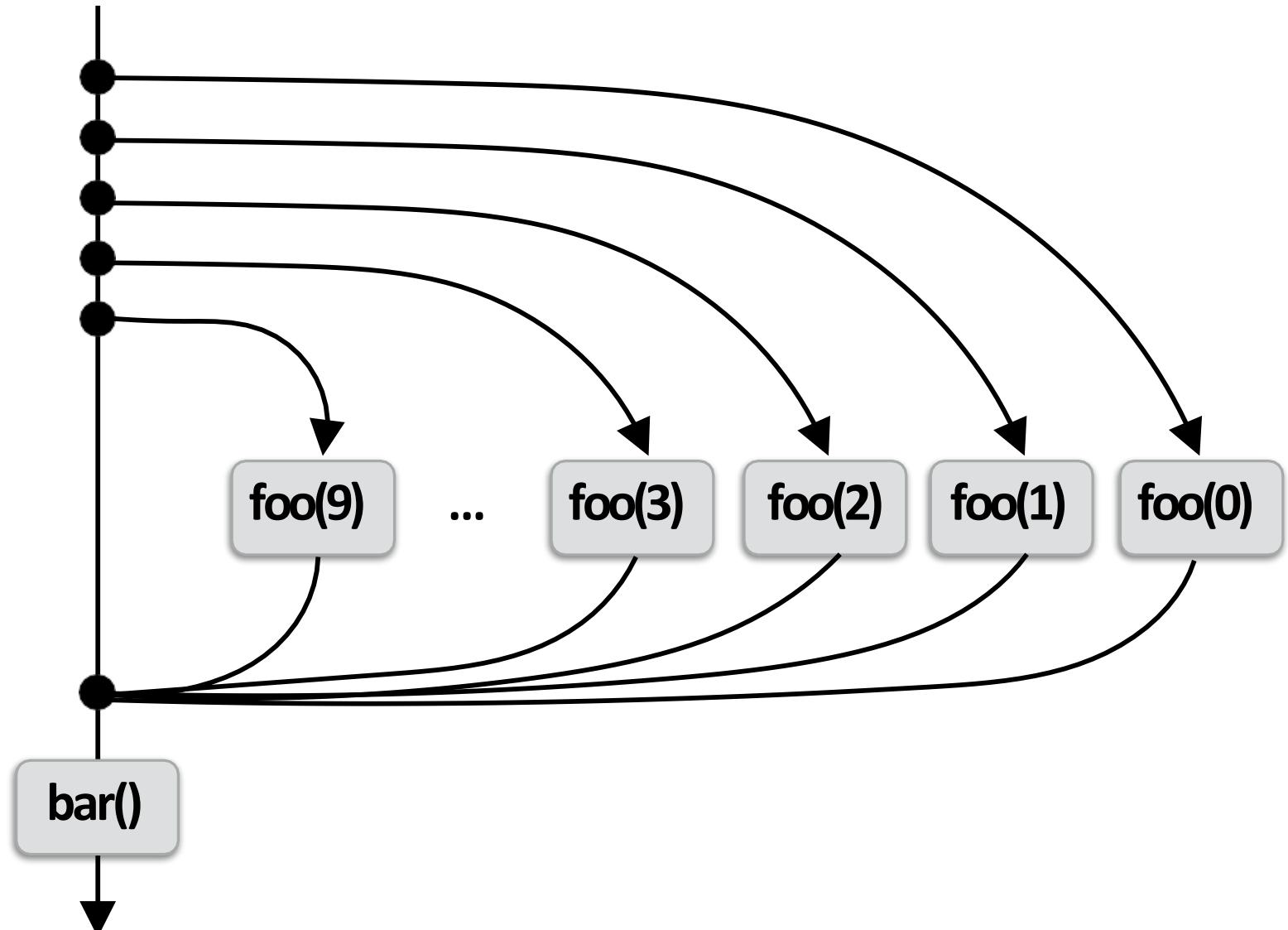
Thread 3

Working on foo(6)...

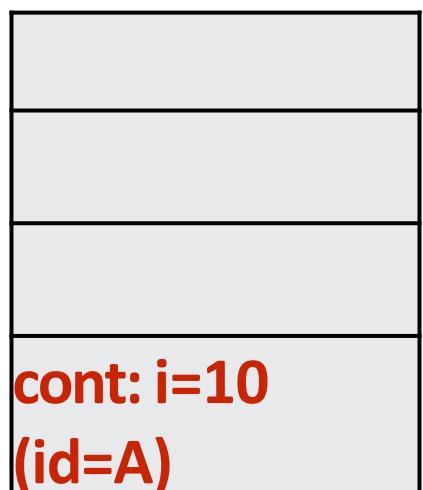
State of worker threads
when all work from loop
is nearly complete

Implementing sync: no stealing

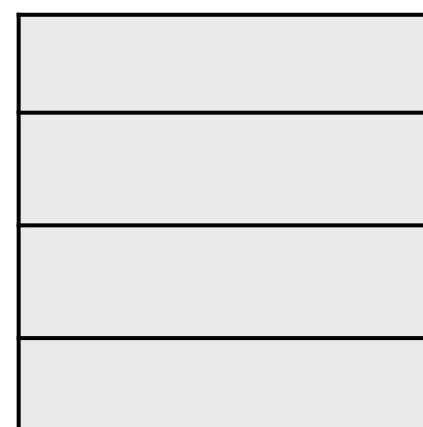
```
block (id: A)  
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



Thread 0 work queue



Thread 1 work queue

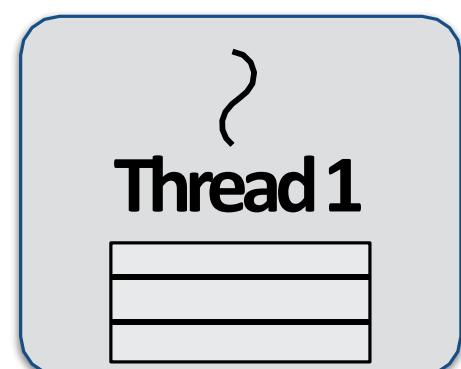
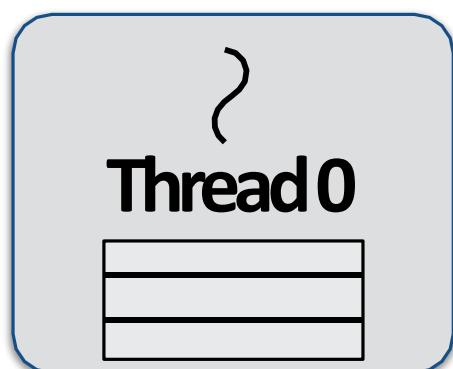


If no work has been stolen by other threads,
then there's nothing to do at the sync point.

cilk_sync is a no-op.

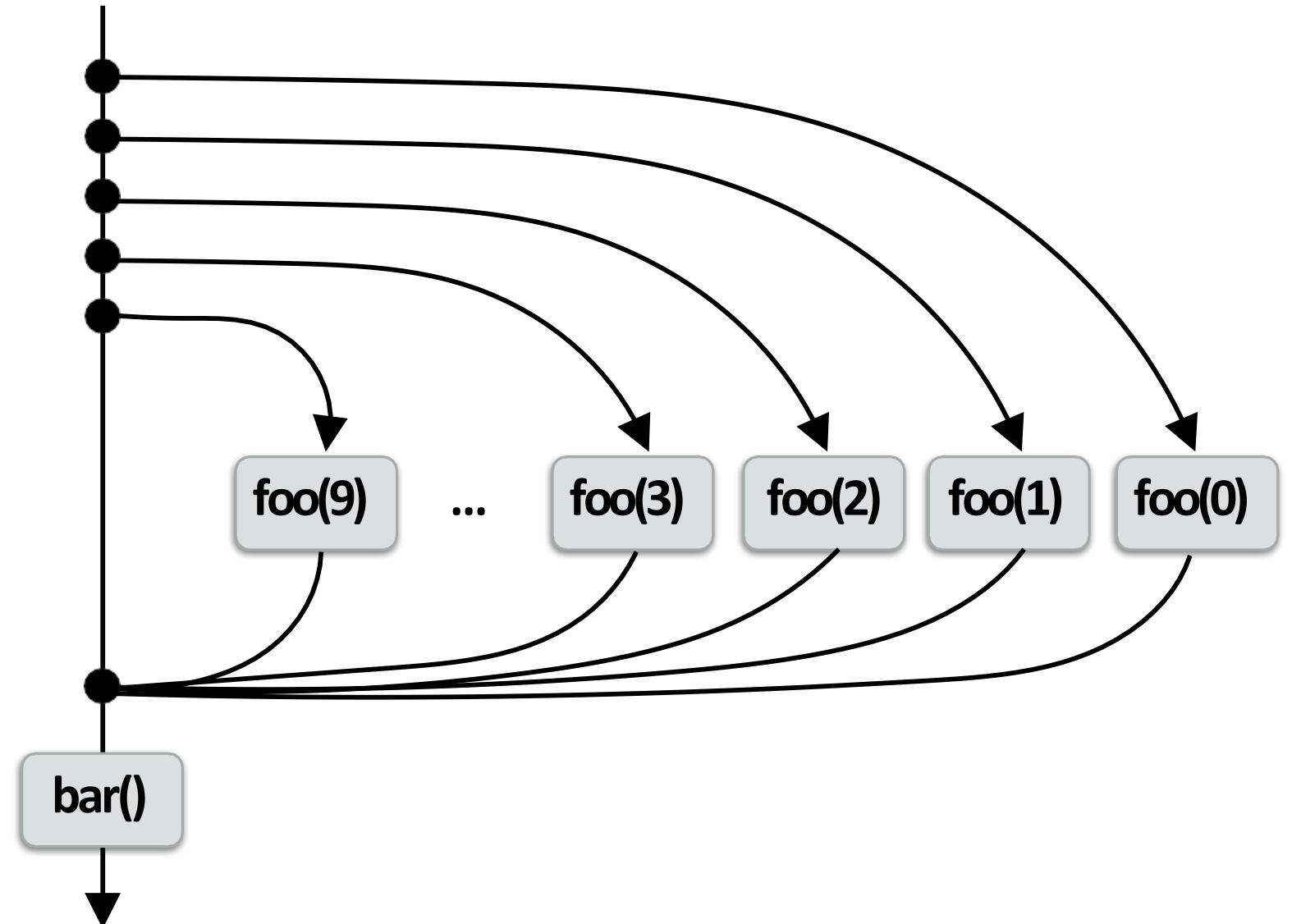
Same control flow as serial execution

Working on foo(9), id=A...

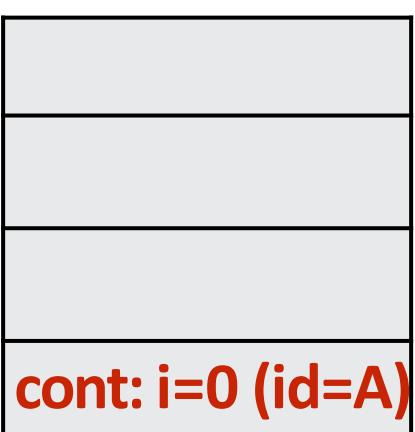


Implementing sync: stalling join

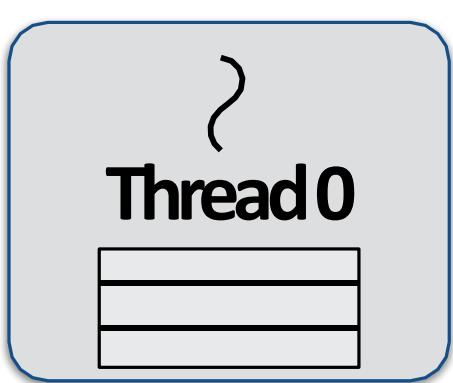
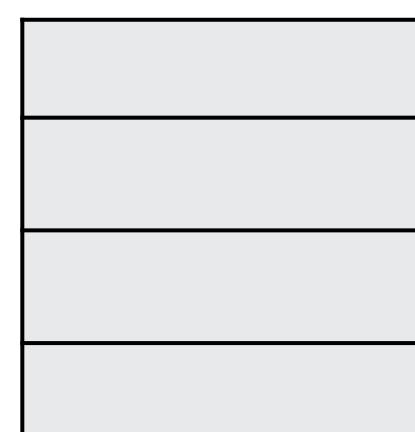
```
block (id: A)  
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



Thread 0 work queue



Thread 1 work queue



Working on foo(0), id=A...

Example 1: “stalling” join policy

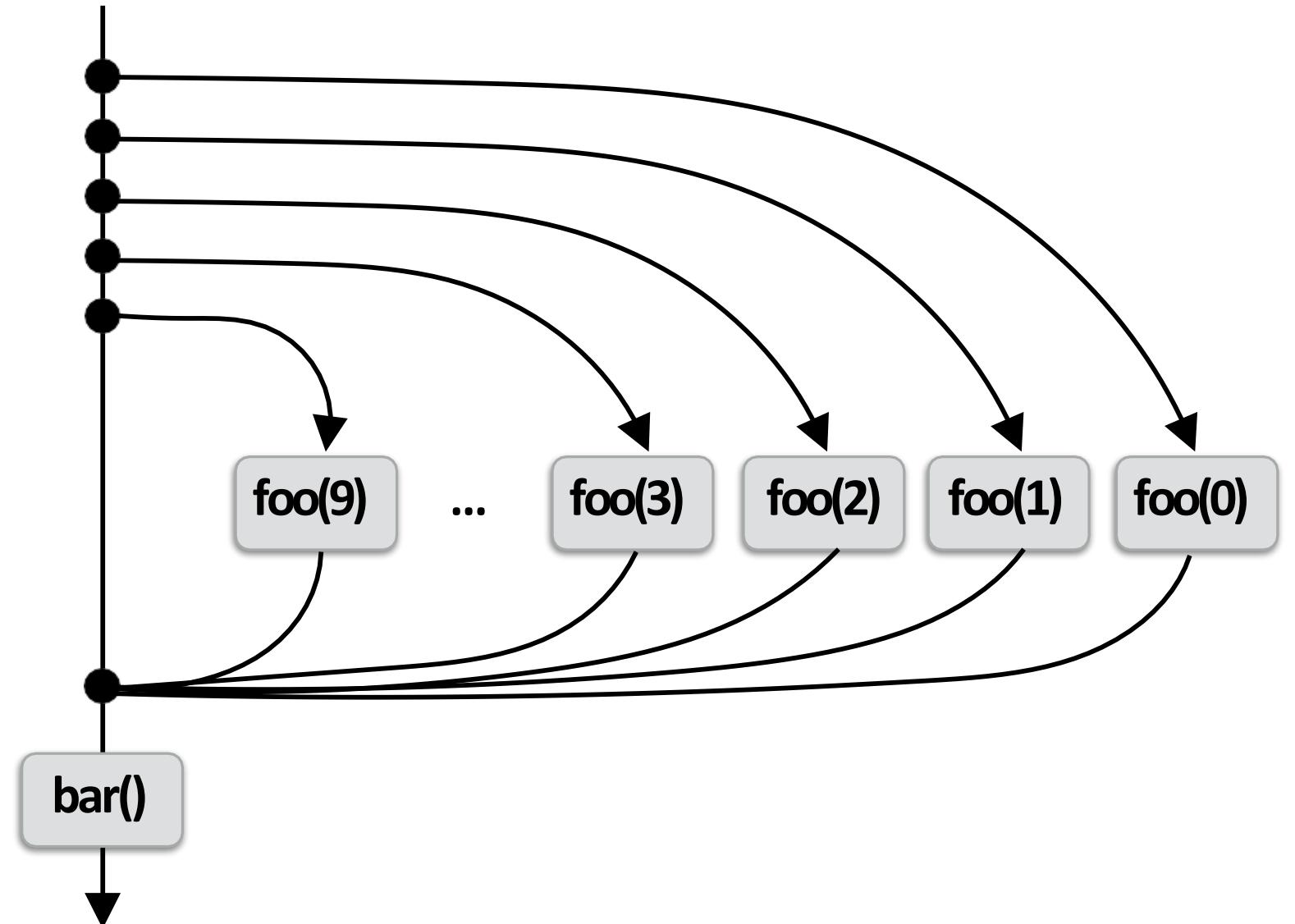
Thread that initiates the fork must perform the sync.

Therefore it waits for all spawned work to be complete.
In this case, thread 0 is the thread initiating the fork

Implementing sync: stalling join

block (id: A)

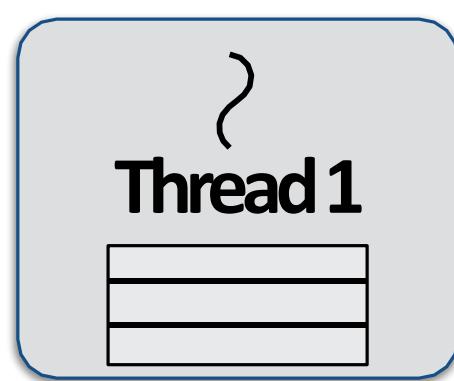
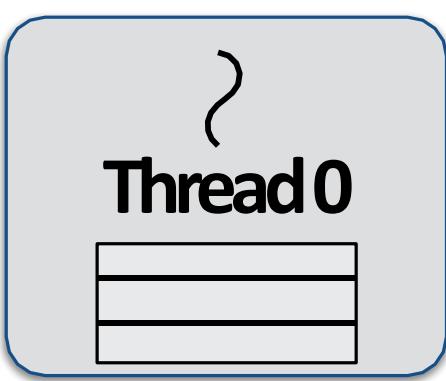
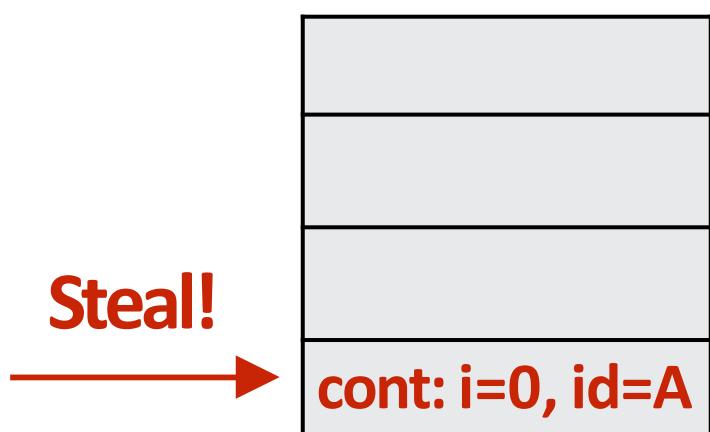
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
  
cilk_sync; Sync for all calls spawned within block A  
  
bar();
```



Thread 0 work queue



Thread 1 work queue



Working on foo(0), id=A...

Idle thread 1 steals from busy thread 0
Note: descriptor for block A created

The descriptor tracks the number of outstanding spawns for the block, and the number of those spawns that have completed.

Here, the 1 spawn corresponds to foo(0) being run by thread 0.

Implementing sync: stalling join

block (id: A)

```
for (int i=0; i<10; i++) {
```

```
    cilk_spawn foo(i);
```

```
}
```

```
cilk_sync; Sync for all calls spawned within block A
```

```
bar();
```

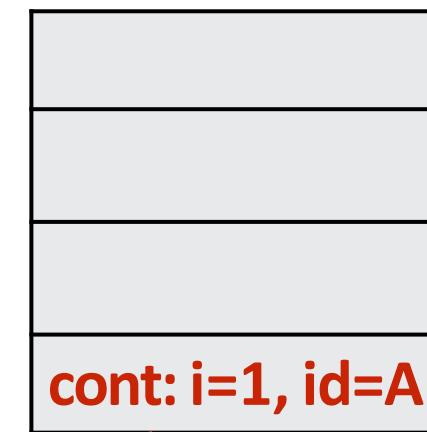
Thread 0 work queue

Thread 1 work queue

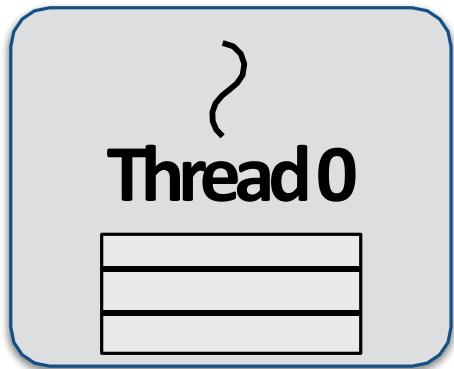
Thread 1 is now running foo(1)

Note: spawn count is now 2

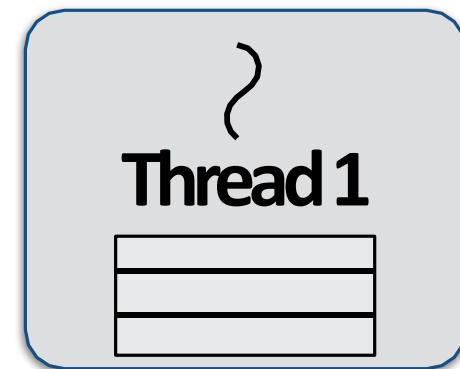
id=A
spawn:2,
done:0



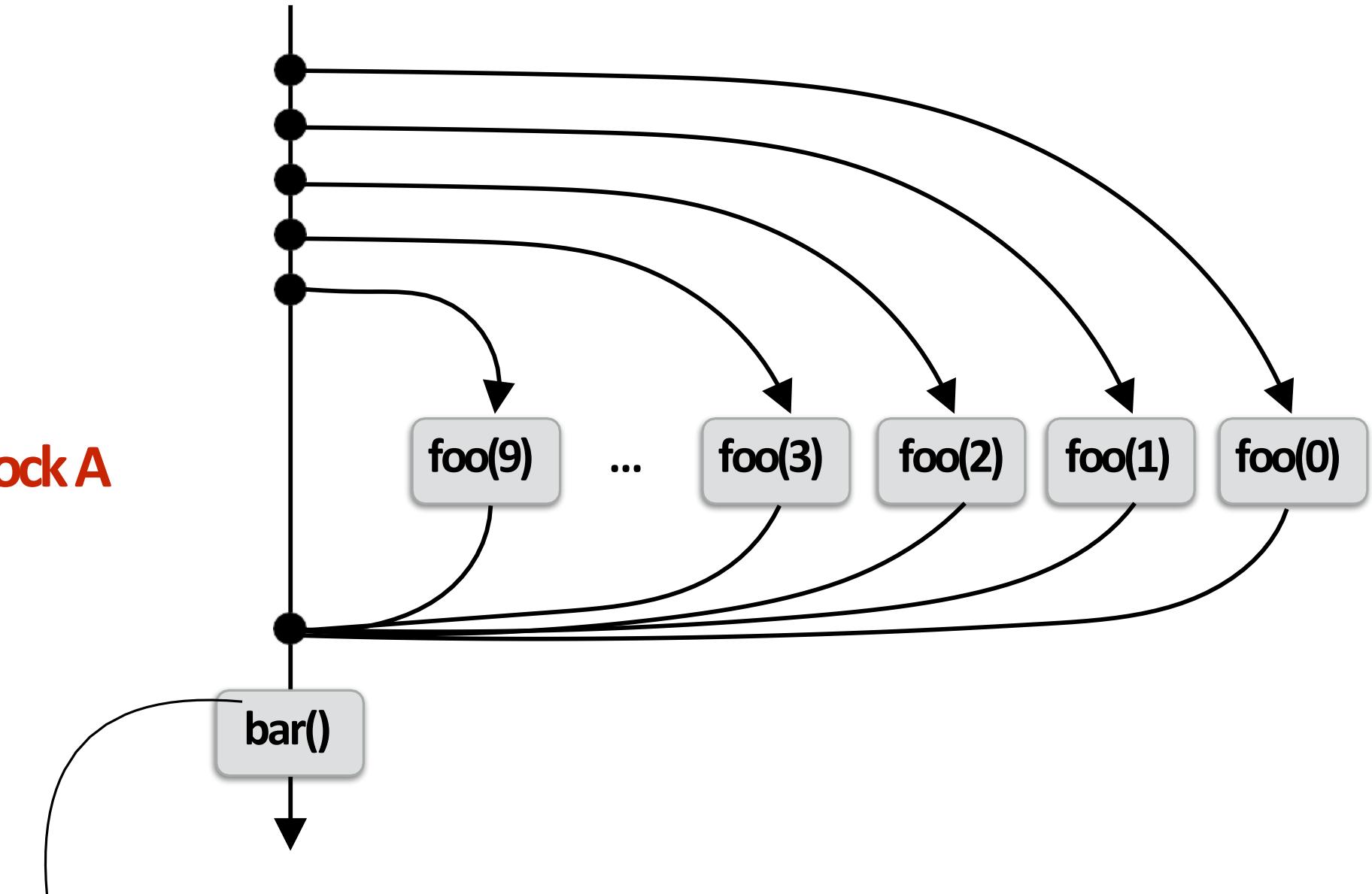
Update count



Working on foo(0), id=A...

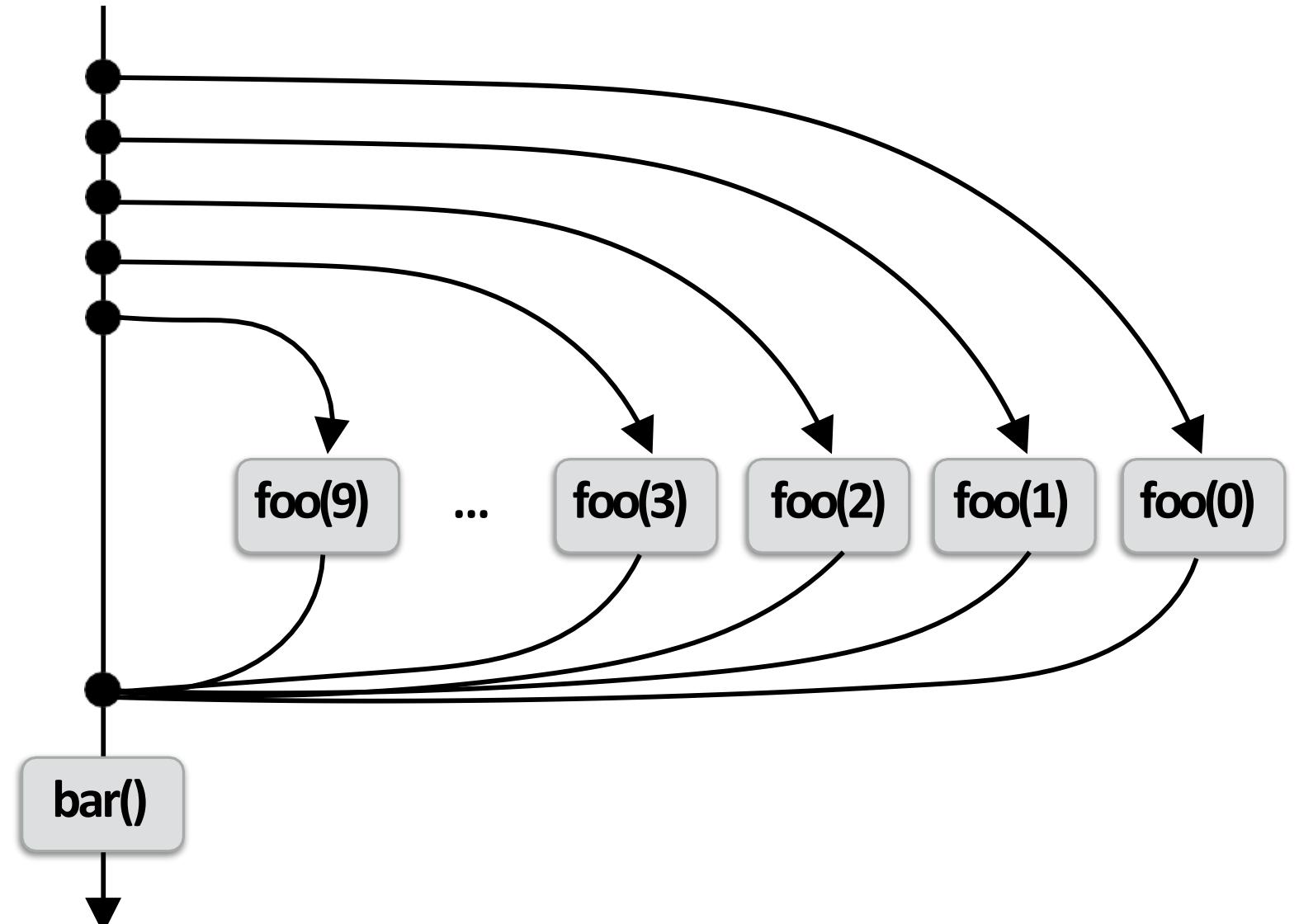


Working on foo(1), id=A...



Implementing sync: stalling join

```
block(id: A)
    for (int i=0; i<10; i++) {
        cilk_spawn foo(i);
    }
    cilk_sync; Sync for all calls spawned within block A
    bar();
```

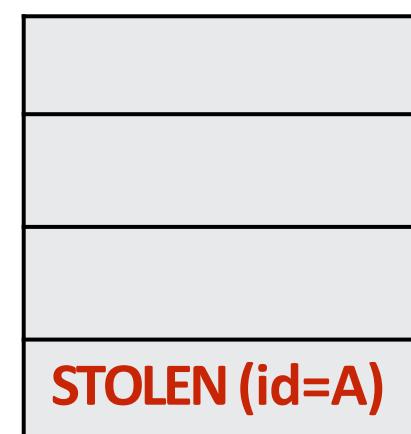


Thread 0 work queue



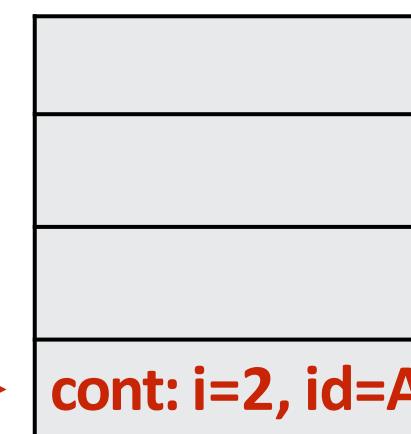
**id=A
spawn:3,
done:1**

Thread 1 work queue



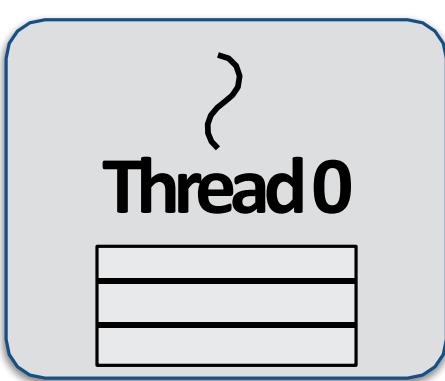
Thread 1 work queue

Thread 2 work queue

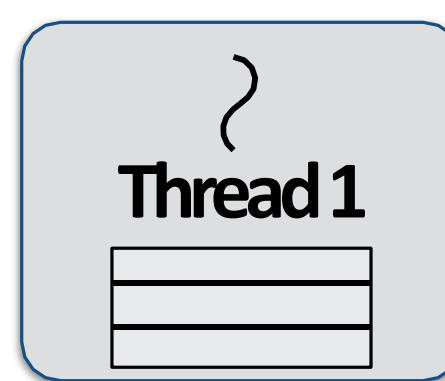


Thread 0 completes foo(0) (updates spawn descriptor)

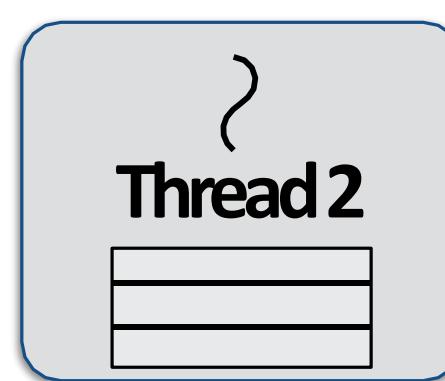
Thread 2 now running foo(2)



Idle!



Working on foo(1), id=A...



Working on foo(2), id=A..

Implementing sync: stalling join

block (id: A)

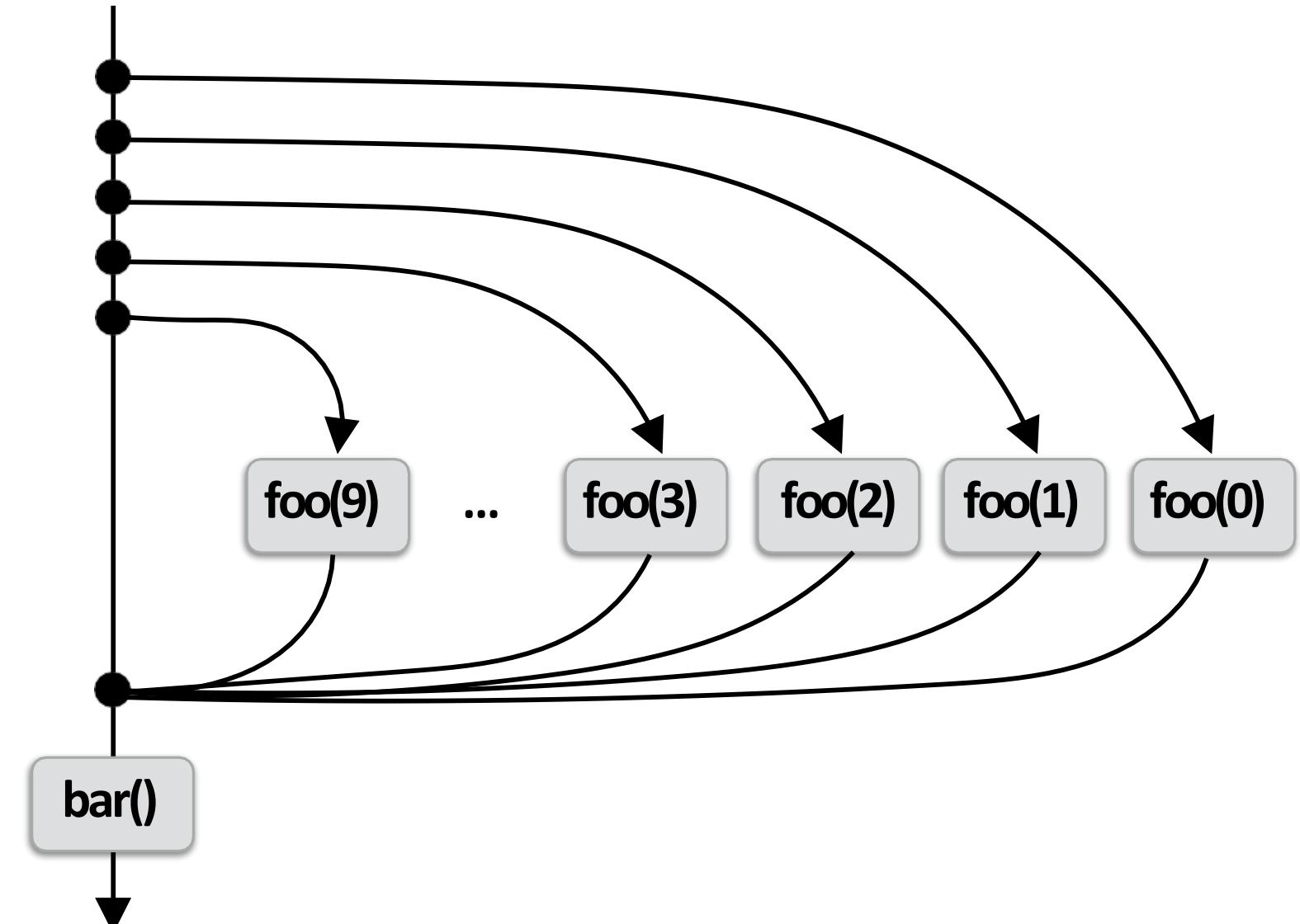
```
for (int i=0; i<10; i++) {
```

```
    cilk_spawn foo(i);
```

```
}
```

```
cilk_sync; Sync for all calls spawned within block A
```

```
bar();
```

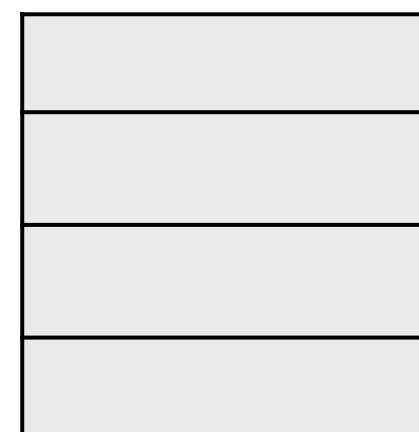


Thread 0 work queue



id=A
spawn:10,
done:9

Thread 1 work queue

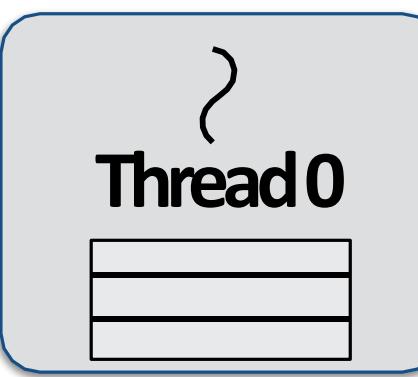


Thread 2 work queue

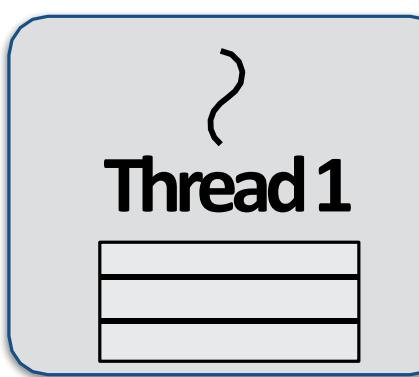


Computation nearing end...

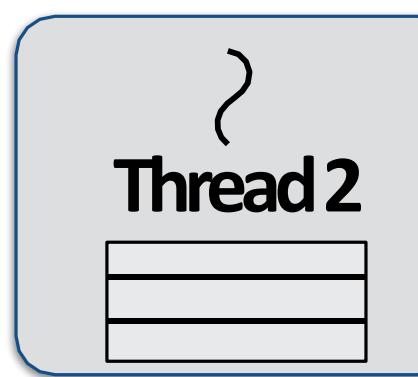
Only foo(9) remains to be completed.



Idle!



Idle!

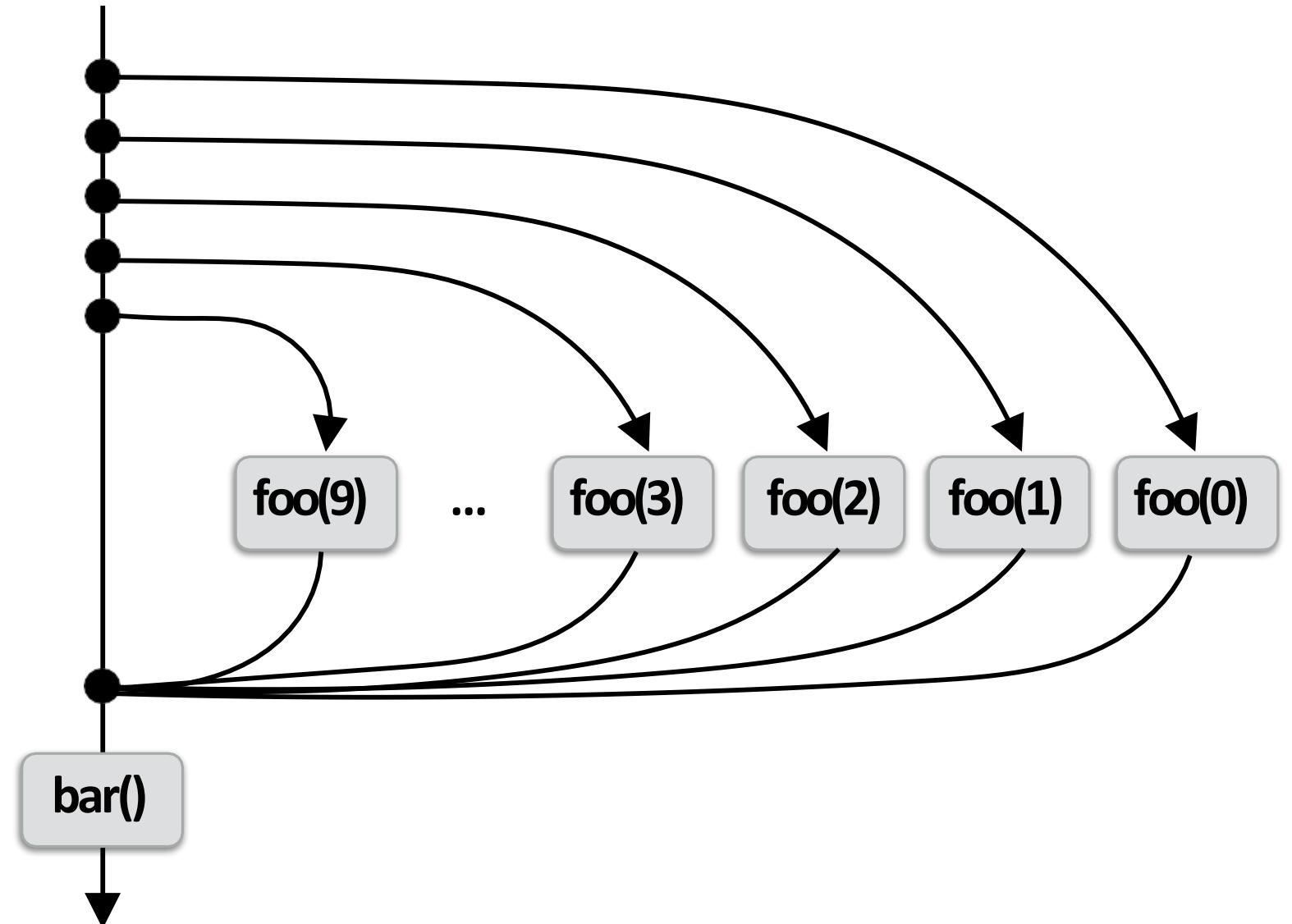


Working on foo(9), id=A...

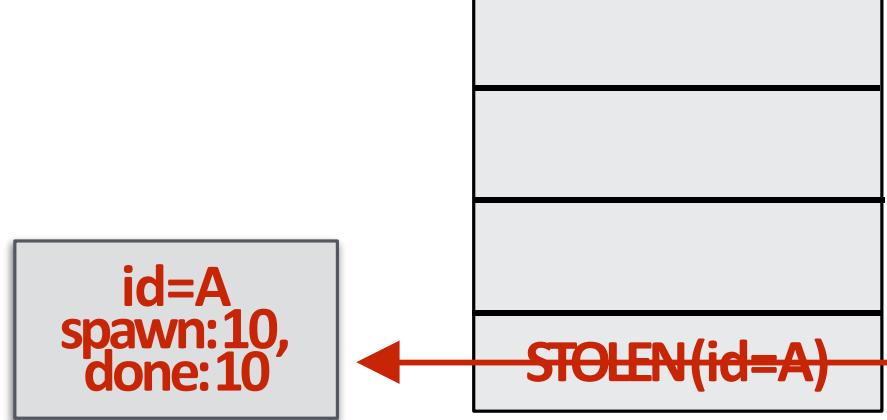
Implementing sync: stalling join

block (id: A)

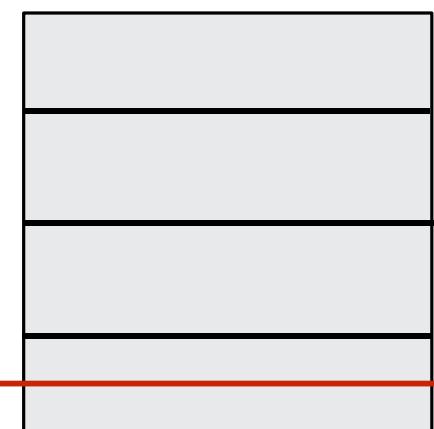
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
  
cilk_sync; Sync for all calls spawned within block A  
  
bar();
```



Thread 0 work queue

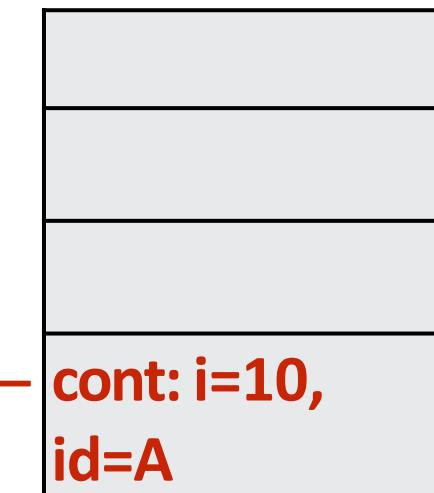


Thread 1 work queue

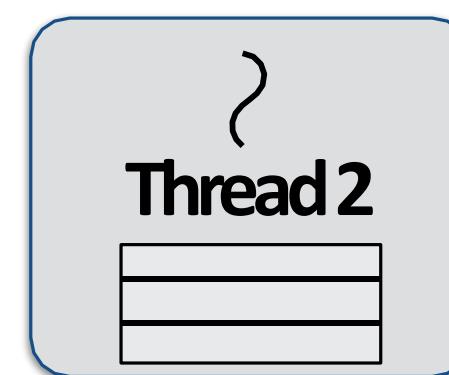
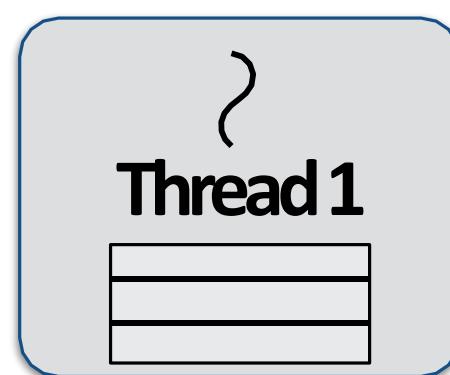
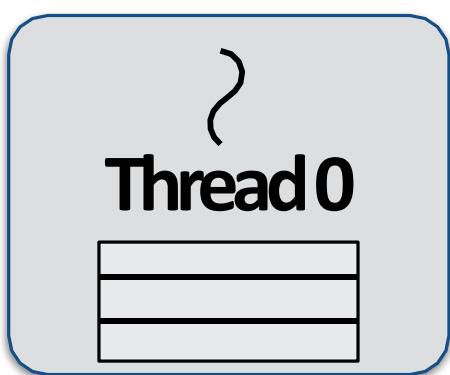


Notify
done!

Thread 2 work queue



Last spawn completes.



Implementing sync: stalling join

block (id: A)

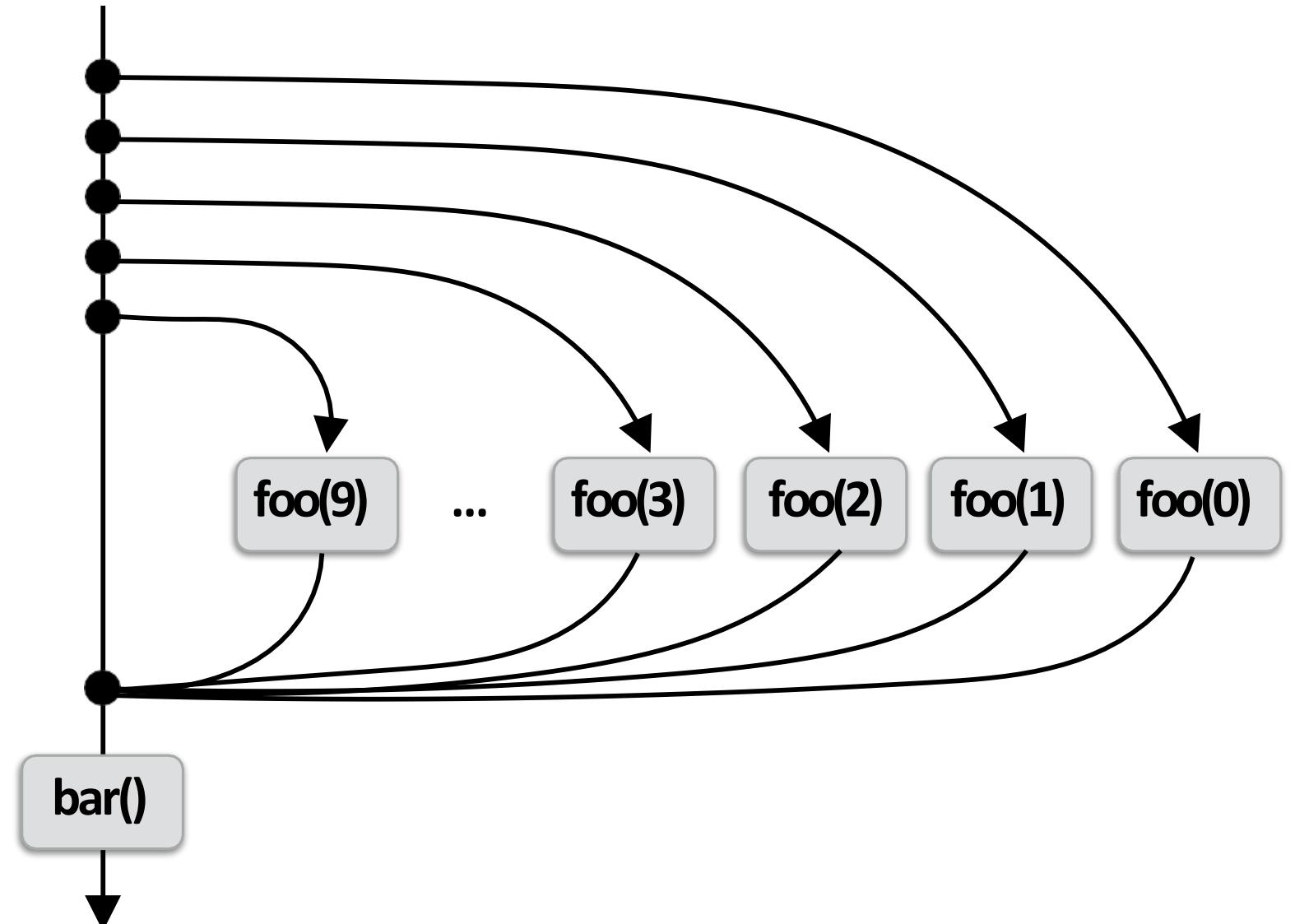
```
for (int i=0; i<10; i++) {
```

```
    cilk_spawn foo(i);
```

```
}
```

```
cilk_sync; Sync for all calls spawned within block A
```

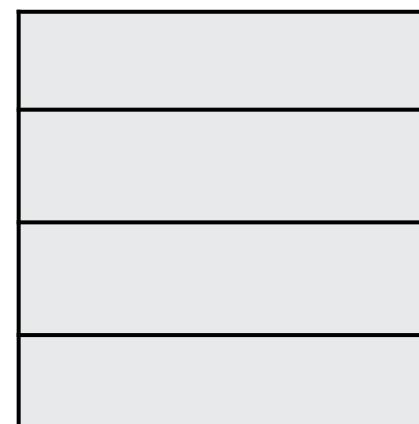
```
bar();
```



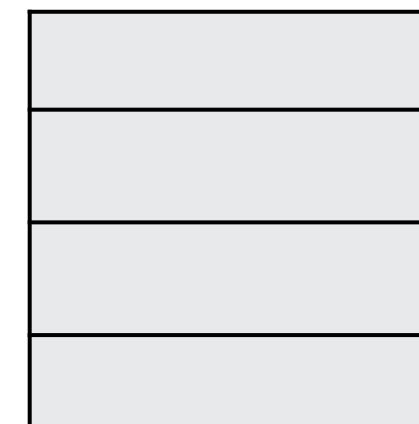
Thread 0 work queue



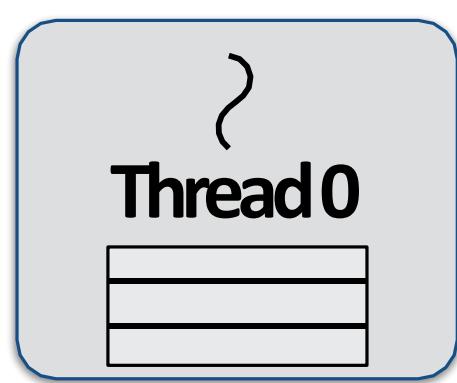
Thread 1 work queue



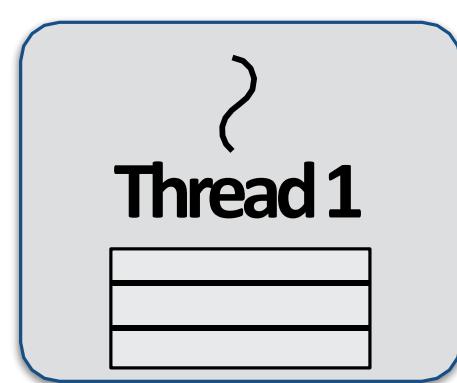
Thread 2 work queue



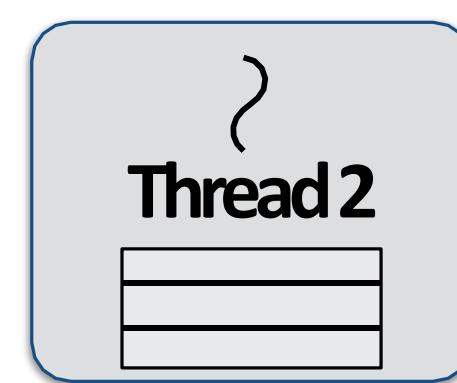
Thread 0 now resumes continuation
and executes bar()
Note block A descriptor is now free.



Working on bar()...



Idle!

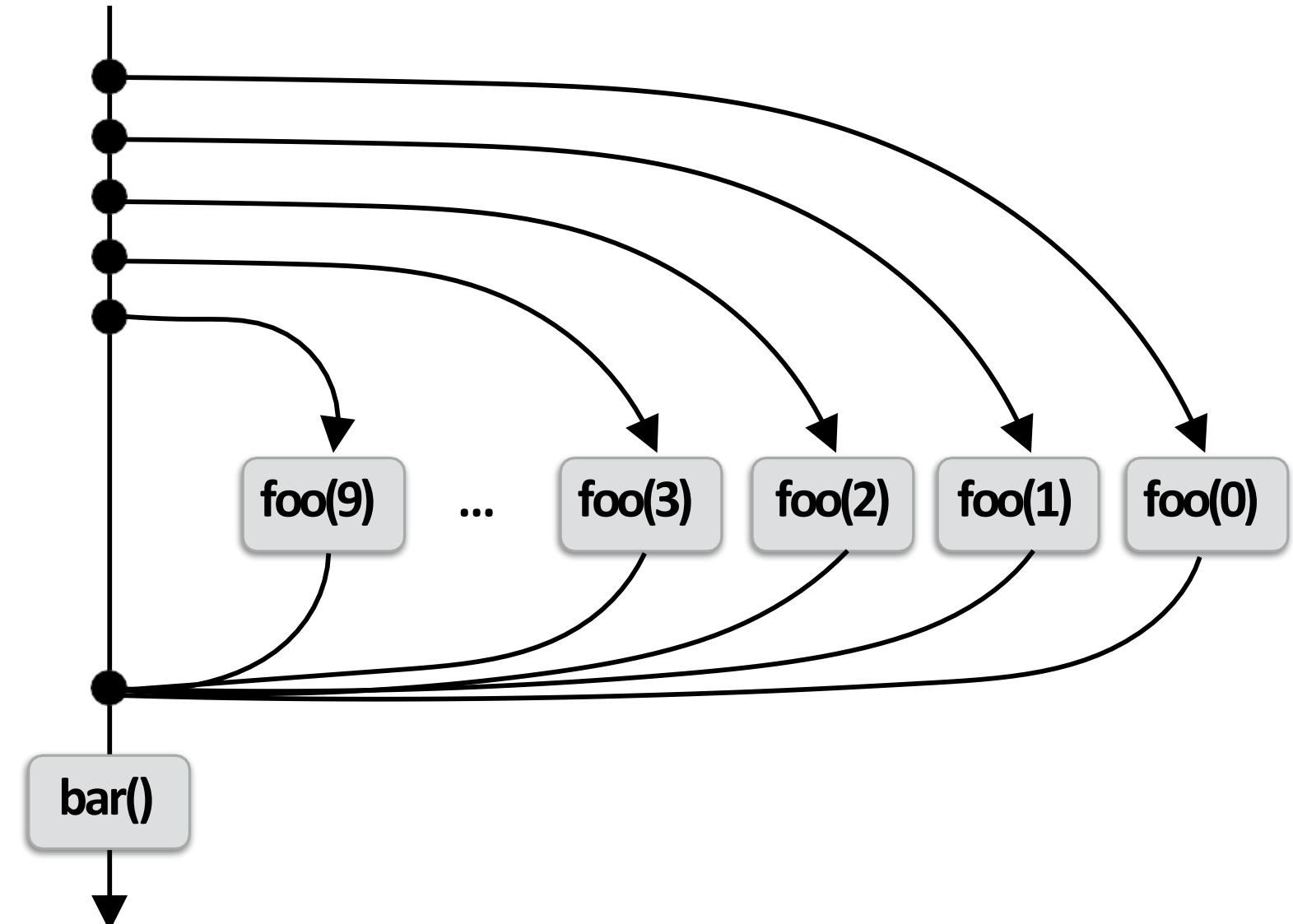


Idle!

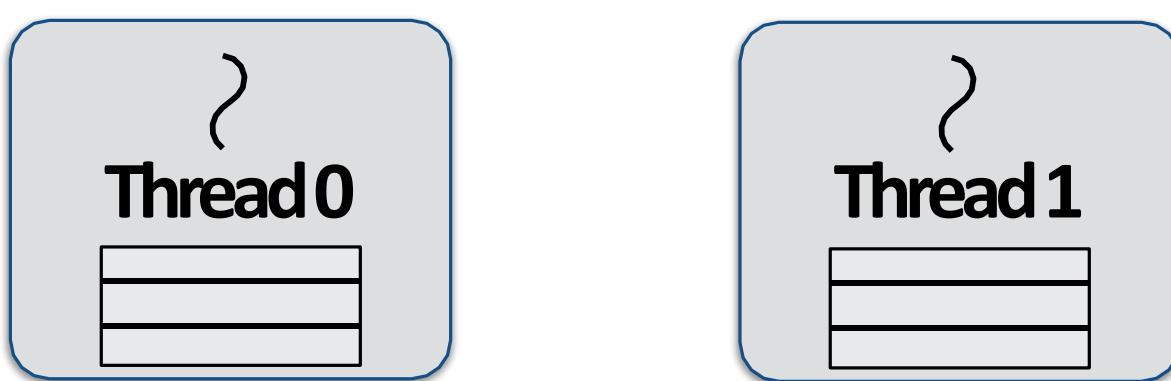
*With this approach,
thread that starts block
will also complete it*

Implementing sync: greedy policy

```
block (id: A)  
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



Thread 0 work queue Thread 1 work queue

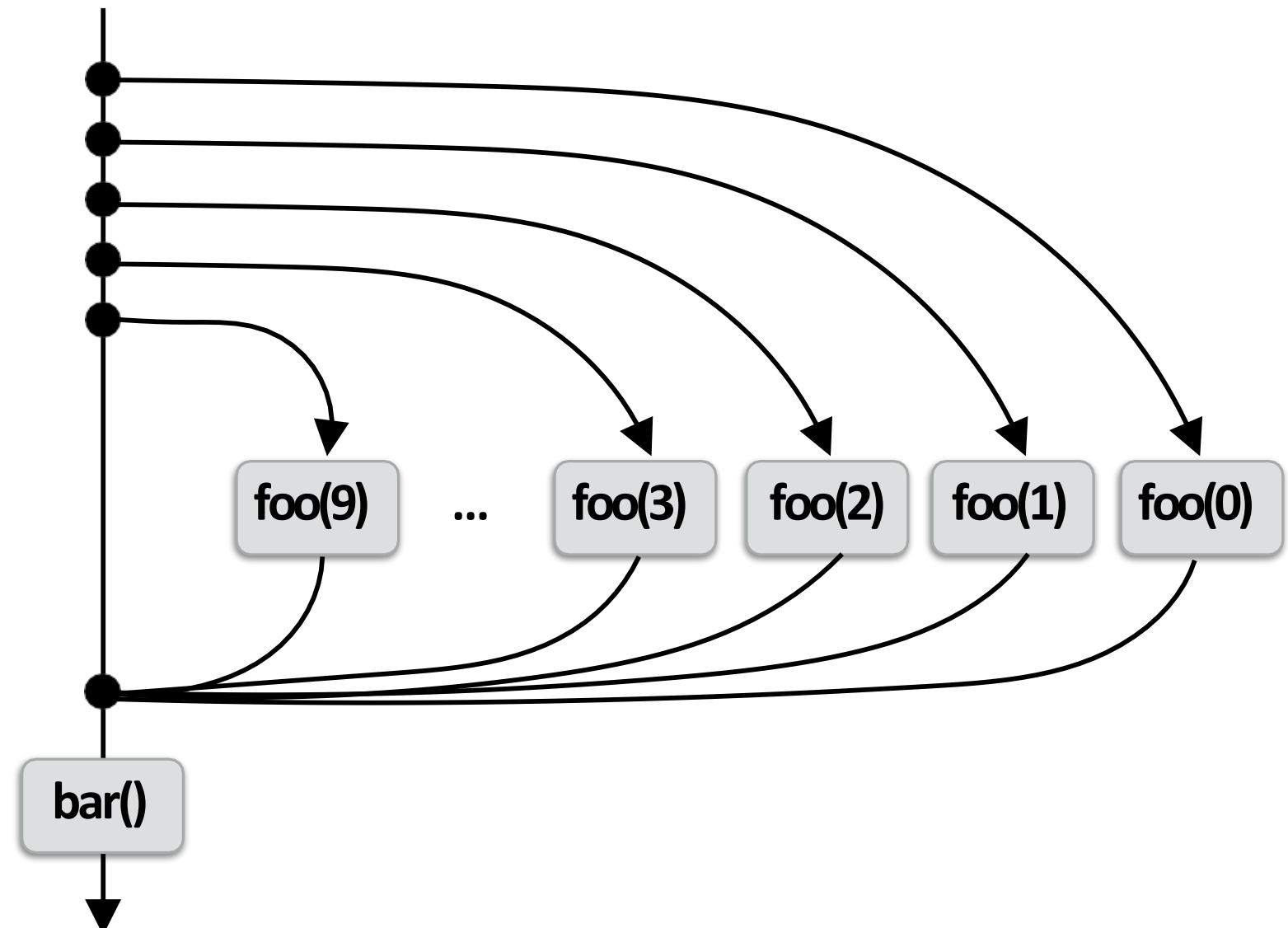


Example 2: “greedy” policy

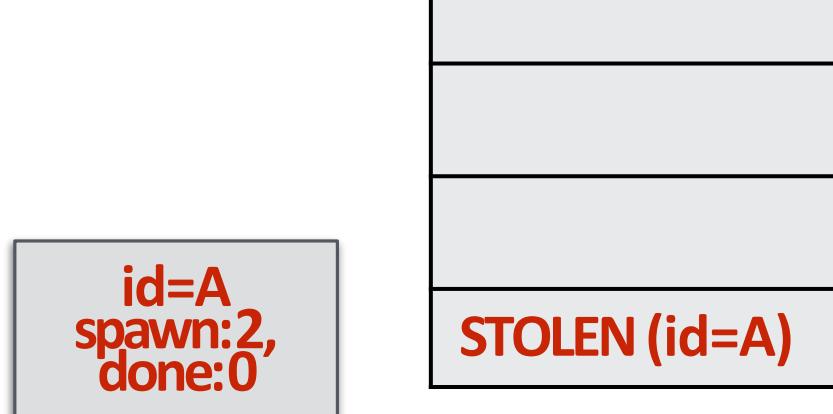
- When thread that initiates the fork goes idle, it looks to steal new work
- Last thread to reach the join point continues execution after sync

Implementing sync: greedy policy

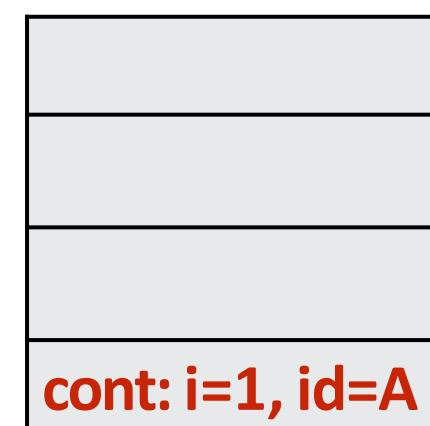
```
block (id: A)  
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



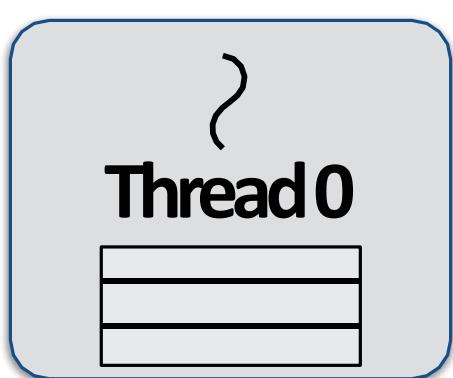
Thread 0 work queue



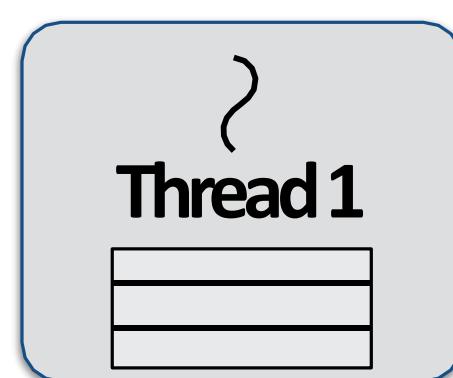
Thread 1 work queue



**Idle thread 1 steals from busy thread 0
(as in the previous case)**



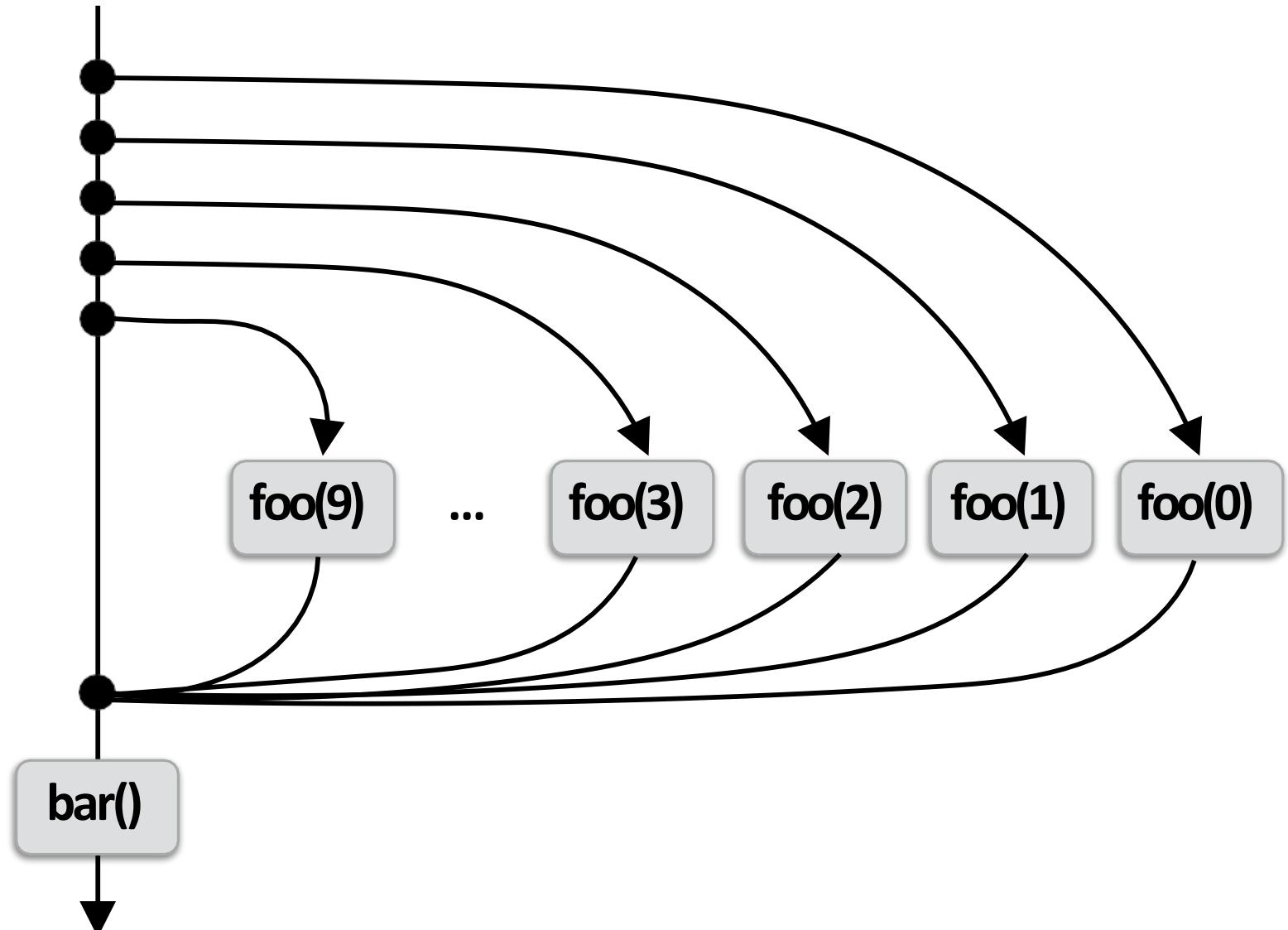
Working on foo(0), id=A...



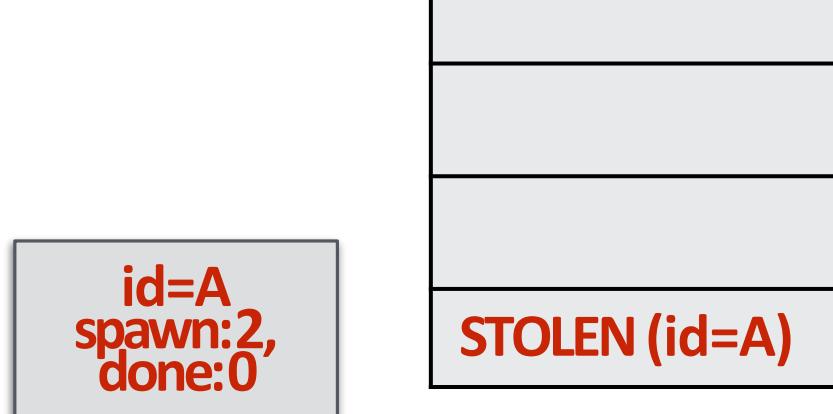
Working on foo(1), id=A...

Implementing sync: greedy policy

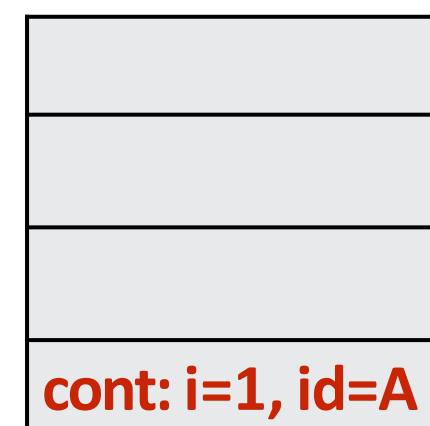
```
block (id: A)  
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



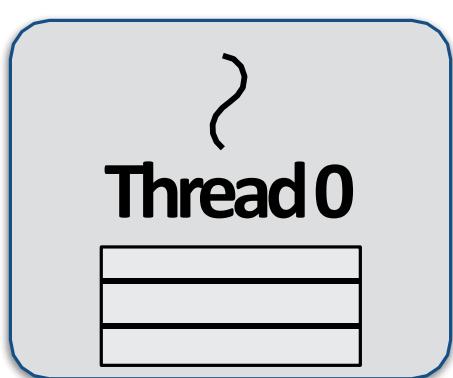
Thread 0 work queue



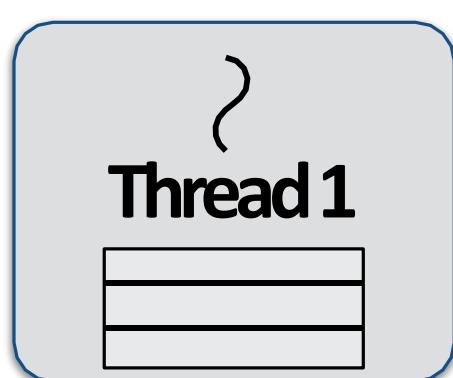
Thread 1 work queue



Thread 0 completes foo(0)
No work to do in local deque, so thread 0 looks to steal!



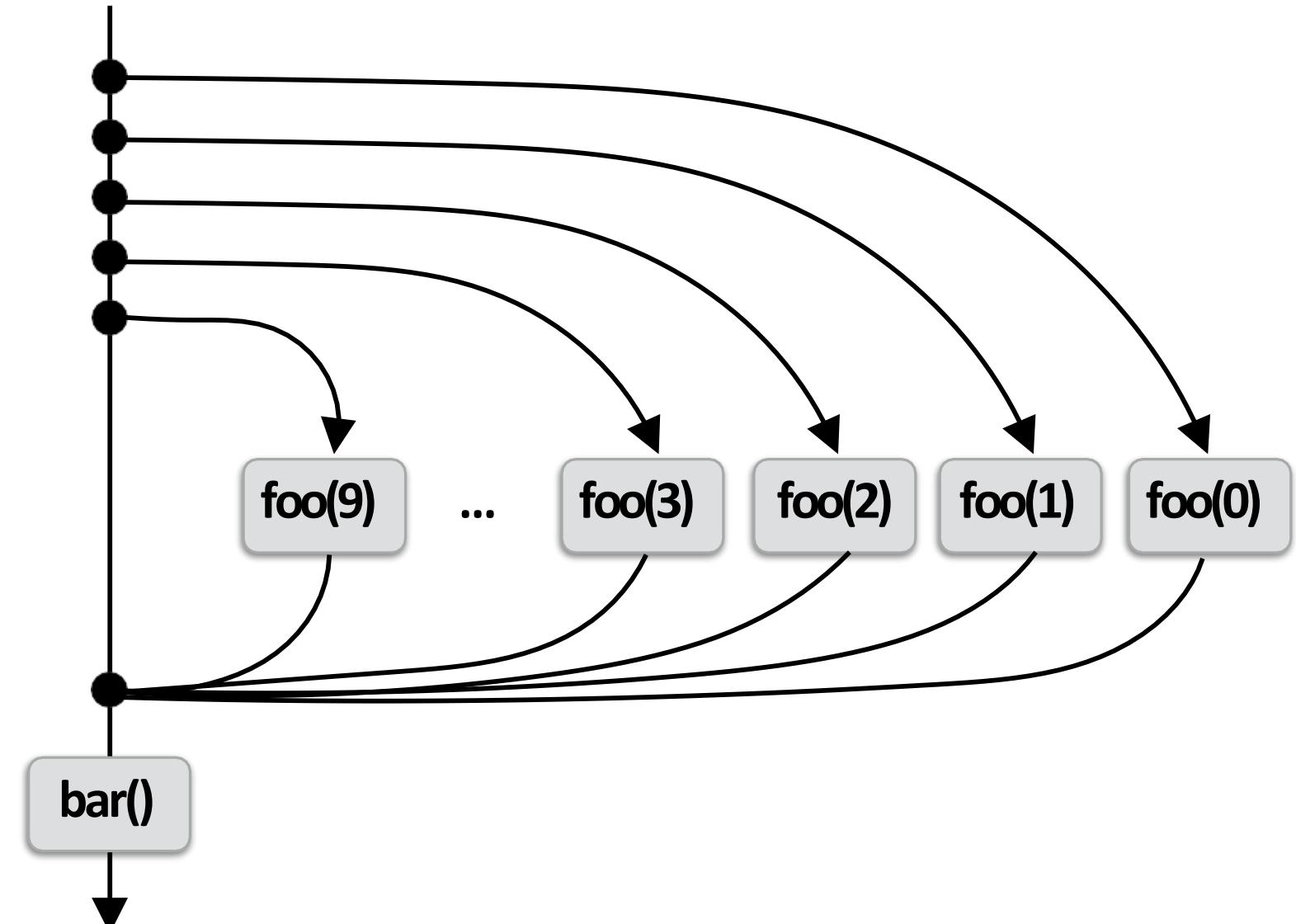
Done with foo(0)!



Working on foo(1), id=A...

Implementing sync: greedy policy

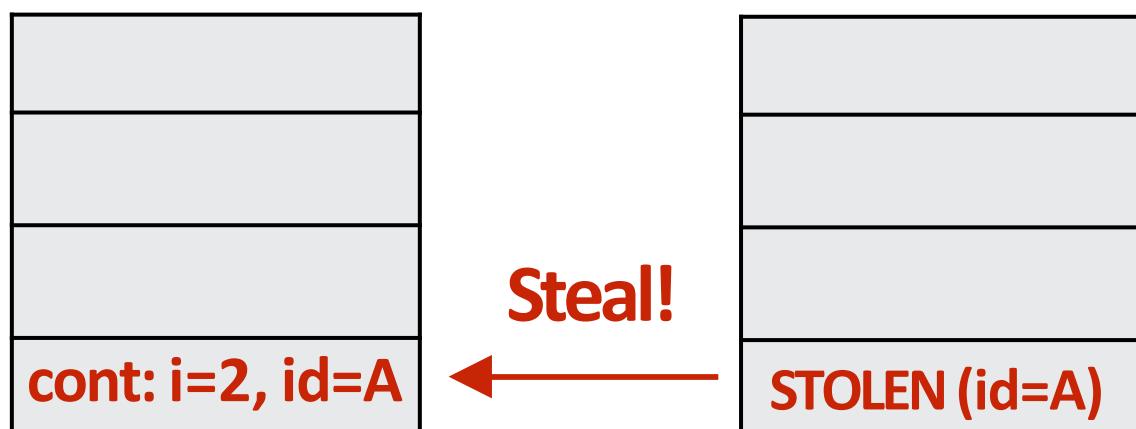
```
block (id: A)  
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



Thread 0 work queue

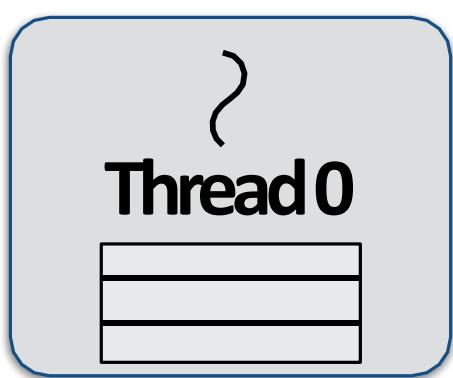


Thread 1 work queue

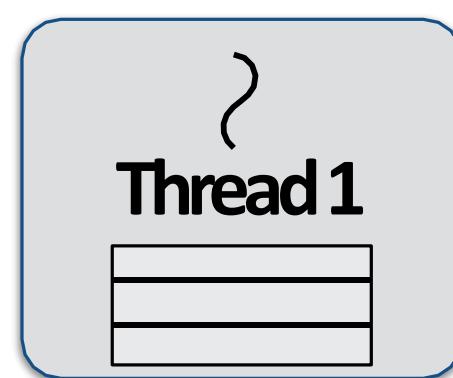


Steal!

Thread 0 now working on foo(2)



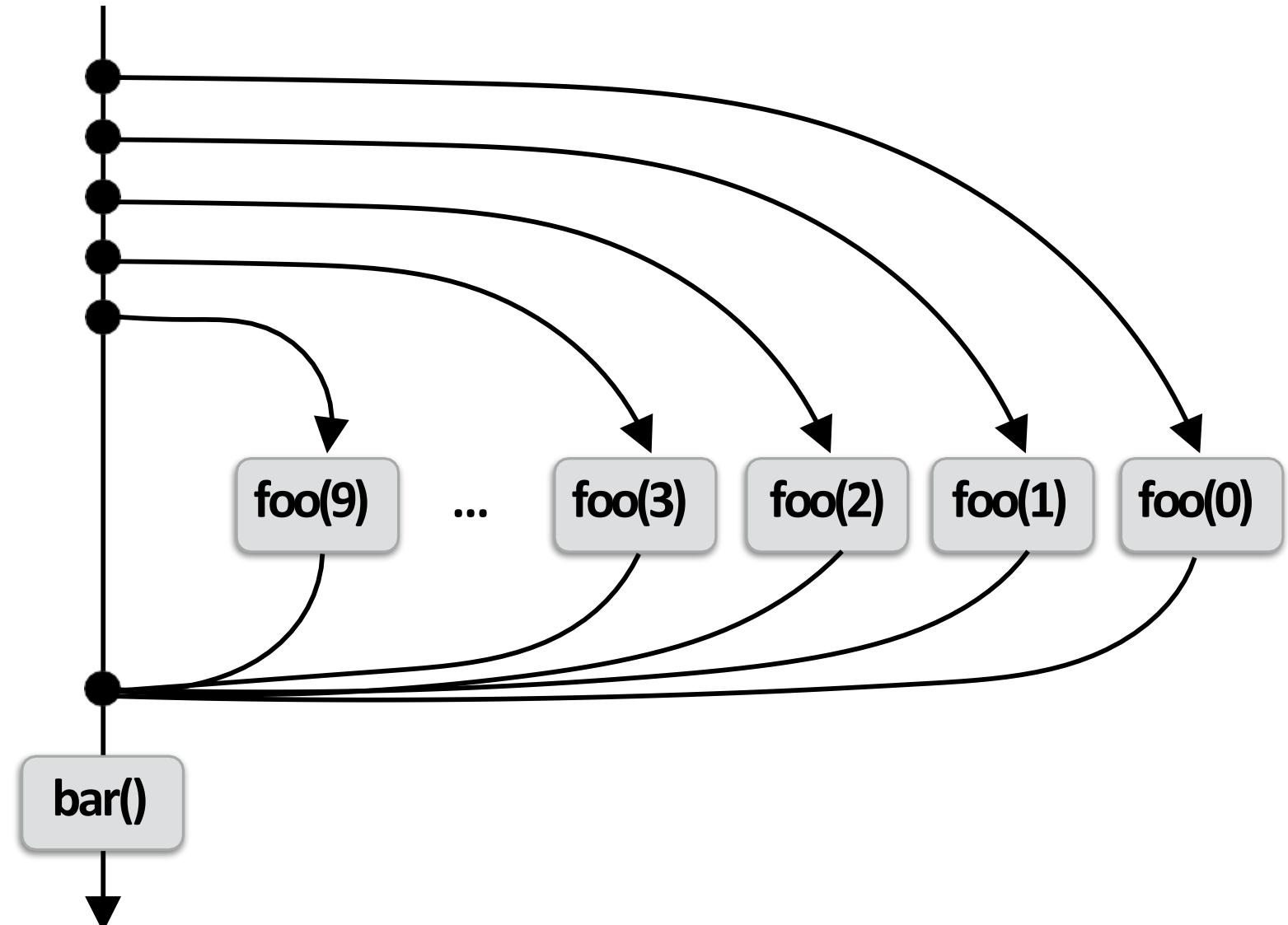
Working on foo(2), id=A...



Working on foo(1), id=A...

Implementing sync: greedy policy

```
block (id: A)  
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



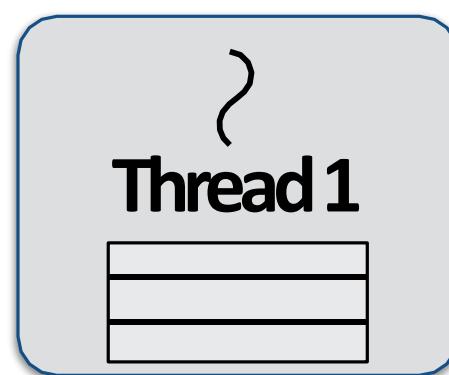
Thread 0 work queue

id=A
spawn:10,
done:9



Thread 1 work queue

cont: i=10,
id=A



Thread 0

Idle

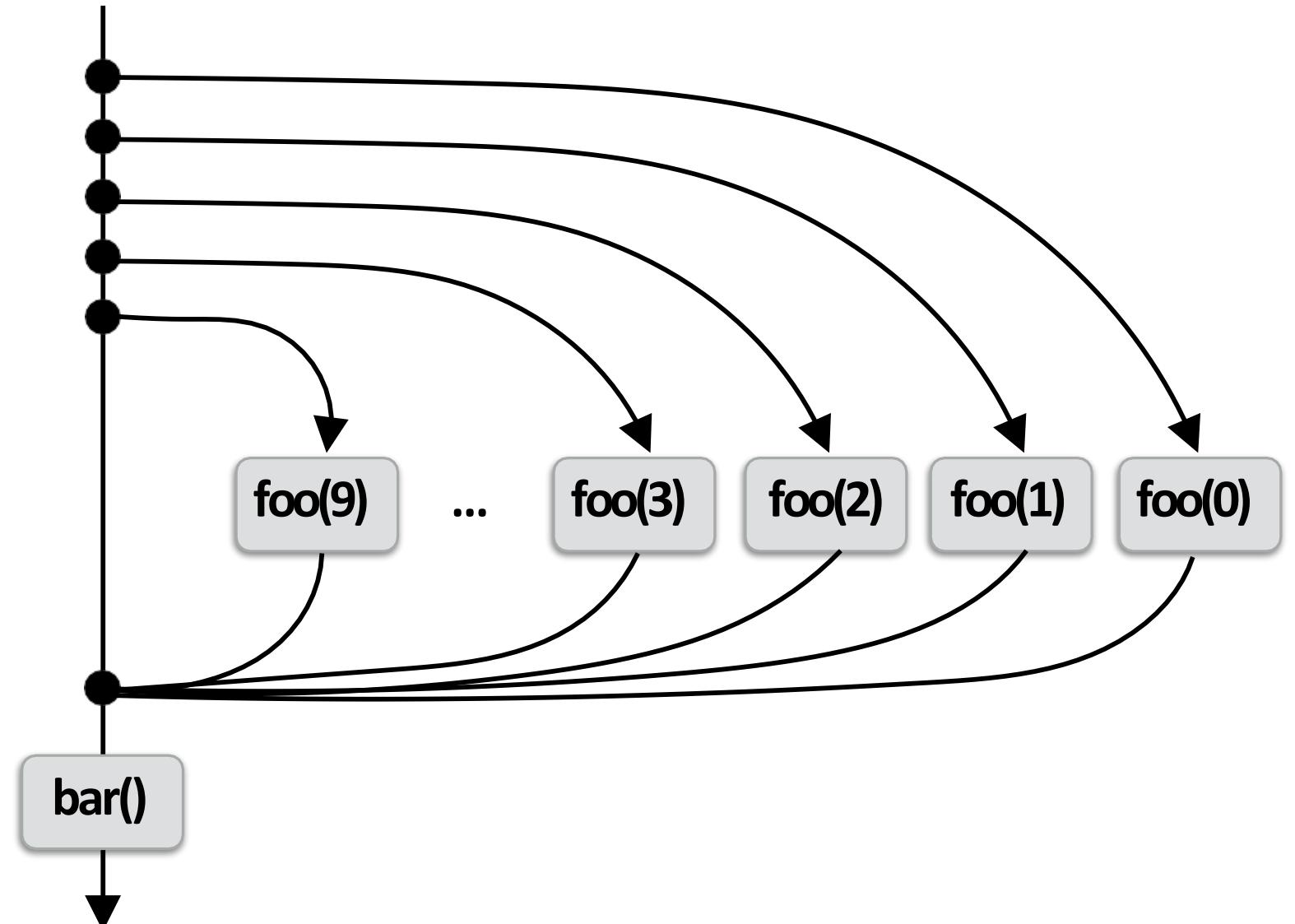
Assume thread 1 is the last to finish
spawned calls for block A.

Working on foo(9), id=A...

Implementing sync: greedy policy

block (id: A)

```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
  
cilk_sync; Sync for all calls spawned within block A  
  
bar();
```



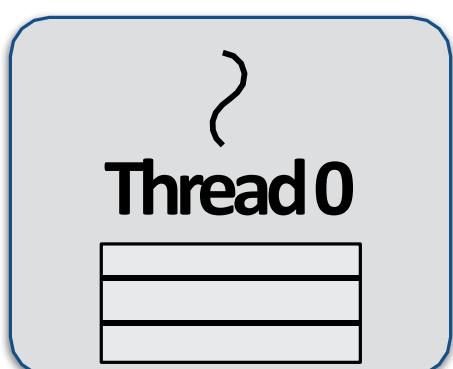
Thread 0 work queue



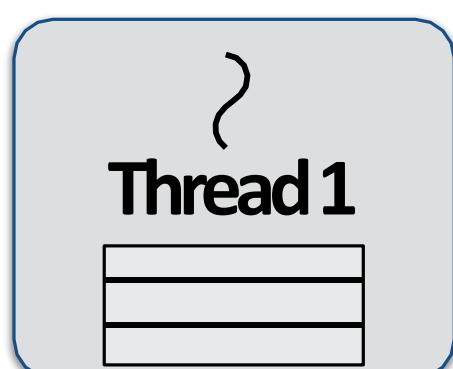
Thread 1 work queue



Thread 1 continues on to run bar()
Note block A descriptor is now free.



Idle



Working on bar()

Cilk uses greedy join scheduling

- **Greedy join scheduling policy**
 - All threads **always attempt to steal if there is nothing to do** (thread only goes idle if no work to steal is present in system)
 - Worker thread that initiated spawn may not be thread that executes logic after `cilk_sync`
- **Remember:**
 - Overhead of bookkeeping steals and managing sync points only occurs when steals occur
 - If large pieces of work are stolen, this should occur infrequently
 - **Most of the time, threads are pushing/popping local work from their local dequeue**

Adjusting Granularity

- **Challenge**
 - If partition computation into too many tasks (fine grained), will be swamped by scheduling overhead
 - If too few tasks (coarse grained), won't have enough parallelism and more prone to workload imbalance
 - Requiring programmer to determine granularity in advance is tedious and non-portable
- **Cilk**
 - Generates two versions of every program
 - “Fast clone” optimized for sequential execution.
 - Spawning ≈ procedure call
 - “Slow clone” does full fork/join parallelism

Cilk Granularity

- **Action**
 - When procedure spawned, runs fast clone
 - When thief steals procedure, convert to slow clone
- **Effect**
 - If no stealing tasks place, runs fast clone sequentially
 - Stealing initiates parallel execution
 - But once workers have enough to do, they shift to fast clones as they spawn more procedures
 - Since stealing is done on big chunks of work, tend to partition big chunks of worker into smaller ones, but shift to sequential execution once there are enough tasks to keep workers busy
 - *Dynamic and automatic adjustment of granularity*

Summary

- **Fork-join parallelism: a natural way to express divide-and-conquer algorithms**
 - Discussed Cilk Plus, but OpenMP also has fork/join primitives
- **Cilk Plus runtime implements spawn/sync abstraction with a locality-aware work stealing scheduler**
 - Always run spawned child (continuation stealing)
 - Greedy behavior at join (threads do not wait at join, immediately look for other work to steal)
 - Automatic & dynamic adjustment of granularity

CSCI465/ECEN433

Introduction to Parallel Computing

Spring 2024



Lecture (10)

Performance Optimization Part 3:

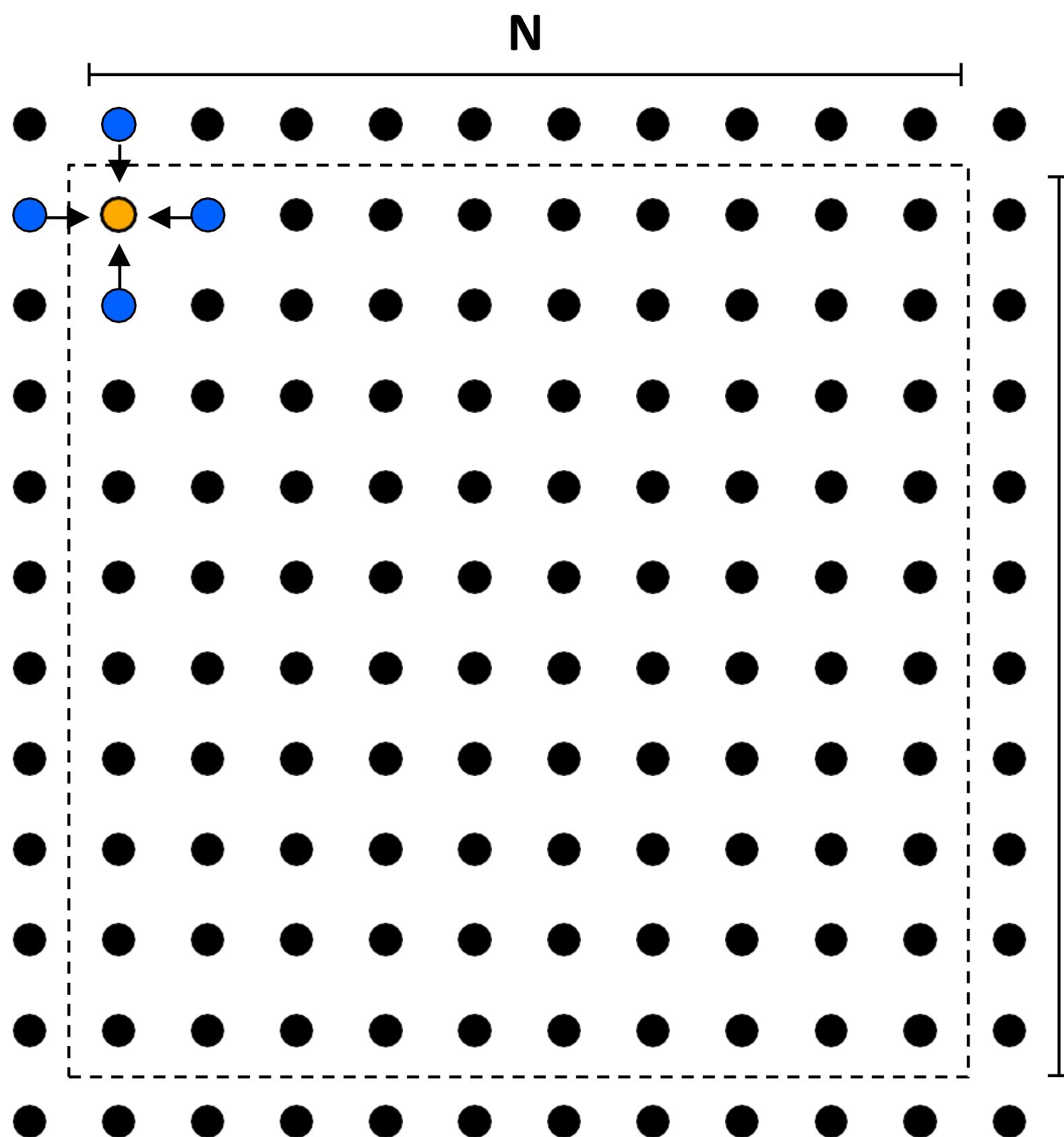
Locality, Communication, and Contention

Today: more parallel program optimization

- Previous lecture: strategies for assigning work to workers (threads, processors, etc.)
 - Goal: achieving good workload balance while also minimizing overhead
 - Discussed tradeoffs between static and dynamic work assignment
 - Tip: keep it simple (implement, analyze, then tune/optimize if required)
- Today: strategies for minimizing communication costs

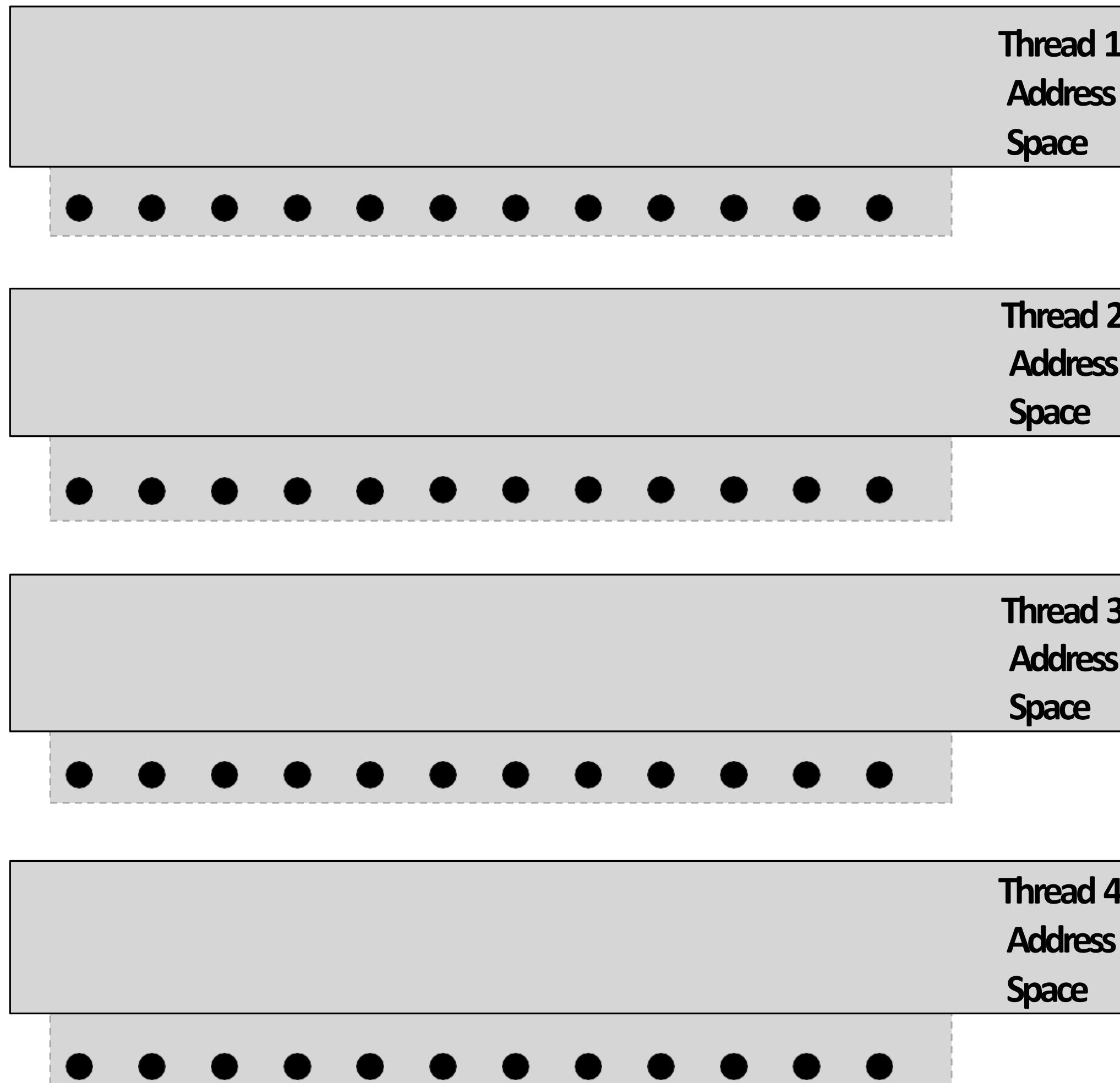
Message-passing solver review

(since it makes communication explicit)



$$A[i, j] = 0.2 * (A[i, j] + A[i, j-1] + A[i-1, j] \\ + A[i, j+1] + A[i+1, j]);$$

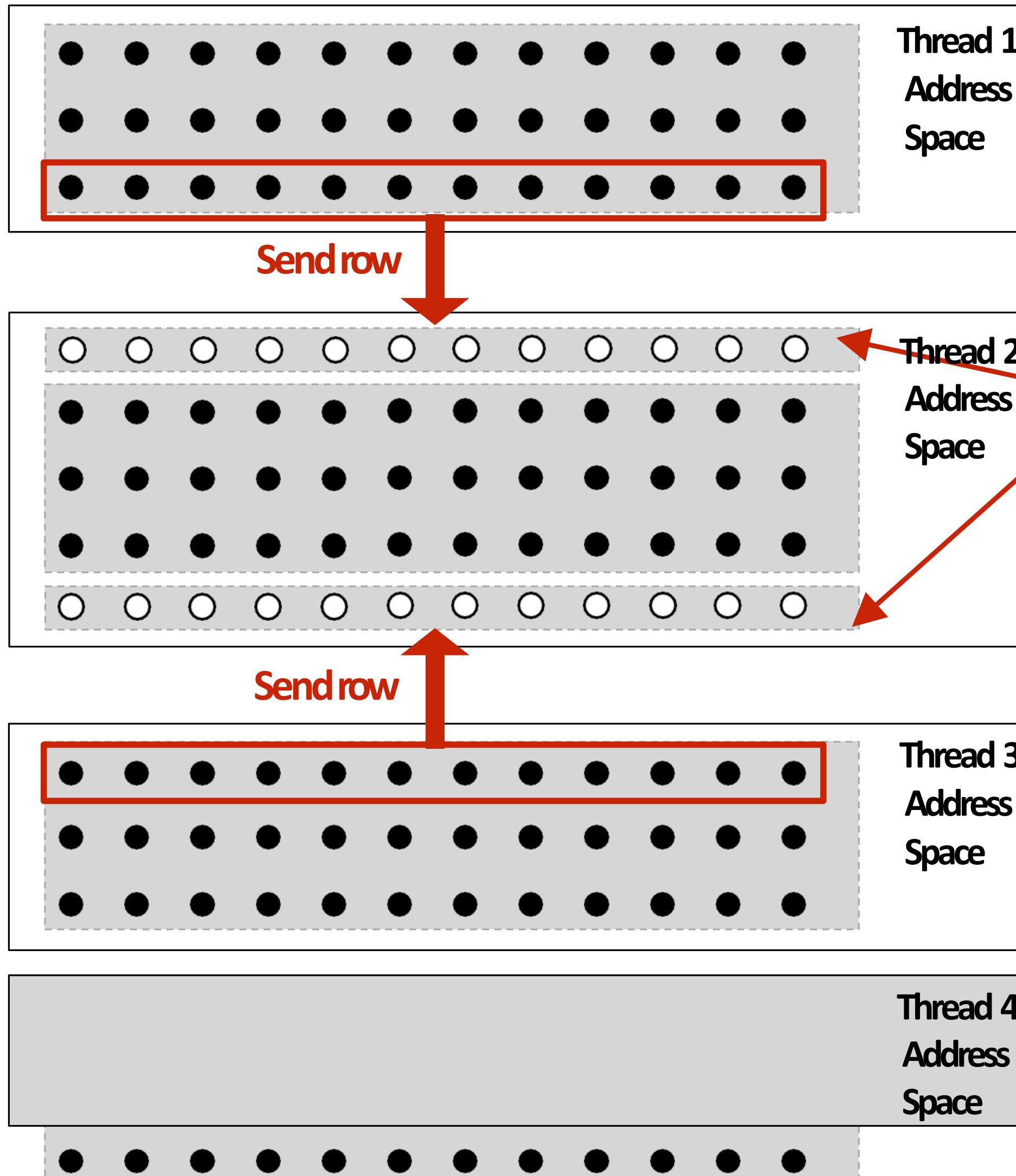
Message passing model: each thread operates in its own address space



In this figure: four threads

**The grid data are partitioned into four allocations, each residing in one of the four unique thread address spaces
(four per-thread private arrays)**

Data replication is now required to correctly execute the program



Example:

After red cell processing is complete, thread 1 and thread 3 send row of data to thread 2
(thread 2 requires up-to-date red cell information to update black cells in the next phase)

"Ghost cells" are grid cells replicated from a remote address space. It's common to say that information in ghost cells is "owned" by other threads.

Thread 2 logic:

```
float* local_data = allocate(N+2,rows_per_thread+2);

int tid = get_thread_id();
int bytes = sizeof(float) * (N+2);

// receive ghost row cells (white dots)
recv(&local_data[0,0], bytes, tid-1);
recv(&local_data[rows_per_thread+1,0], bytes, tid+1);

// Thread 2 now has data necessary to perform
// future computation
```

Message passing solver

Similar structure to shared address space solver, but now communication is explicit in message sends and receives

```
int N;
int tid = get_thread_id();
int rows_per_thread = N / get_num_threads();

float* localA = allocate(rows_per_thread+2, N+2);

// assume localA is initialized with starting values
// assume MSG_ID_ROW, MSG_ID_DONE, MSG_ID_DIFF are constants used as msg ids
///////////

void solve() {
    bool done = false;
    while (!done) {

        float my_diff = 0.0f;

        if (tid != 0)
            send(&localA[1,0], sizeof(float)*(N+2), tid-1, MSG_ID_ROW);
        if (tid != get_num_threads()-1)
            send(&localA[rows_per_thread,0], sizeof(float)*(N+2), tid+1, MSG_ID_ROW);

        if (tid != 0)
            recv(&localA[0,0], sizeof(float)*(N+2), tid-1, MSG_ID_ROW);
        if (tid != get_num_threads()-1)
            recv(&localA[rows_per_thread+1,0], sizeof(float)*(N+2), tid+1, MSG_ID_ROW);

        for (int i=1; i<rows_per_thread+1; i++) {
            for (int j=1; j<n+1; j++) {
                float prev = localA[i,j];
                localA[i,j] = 0.2 * (localA[i-1,j] + localA[i,j] + localA[i+1,j] +
                                      localA[i,j-1] + localA[i,j+1]);
                my_diff += fabs(localA[i,j] - prev);
            }
        }

        if (tid != 0) {
            send(&mydiff, sizeof(float), 0, MSG_ID_DIFF);
            recv(&done, sizeof(bool), 0, MSG_ID_DONE);
        } else {
            float remote_diff;
            for (int i=1; i<get_num_threads()-1; i++) {
                recv(&remote_diff, sizeof(float), i, MSG_ID_DIFF);
                my_diff += remote_diff;
            }
            if (my_diff/(N*N) < TOLERANCE)
                done = true;
            for (int i=1; i<get_num_threads()-1; i++)
                send(&done, sizeof(bool), i, MSG_ID_DONE);
        }
    }
}
```

Send and receive ghost rows to “neighbor threads”

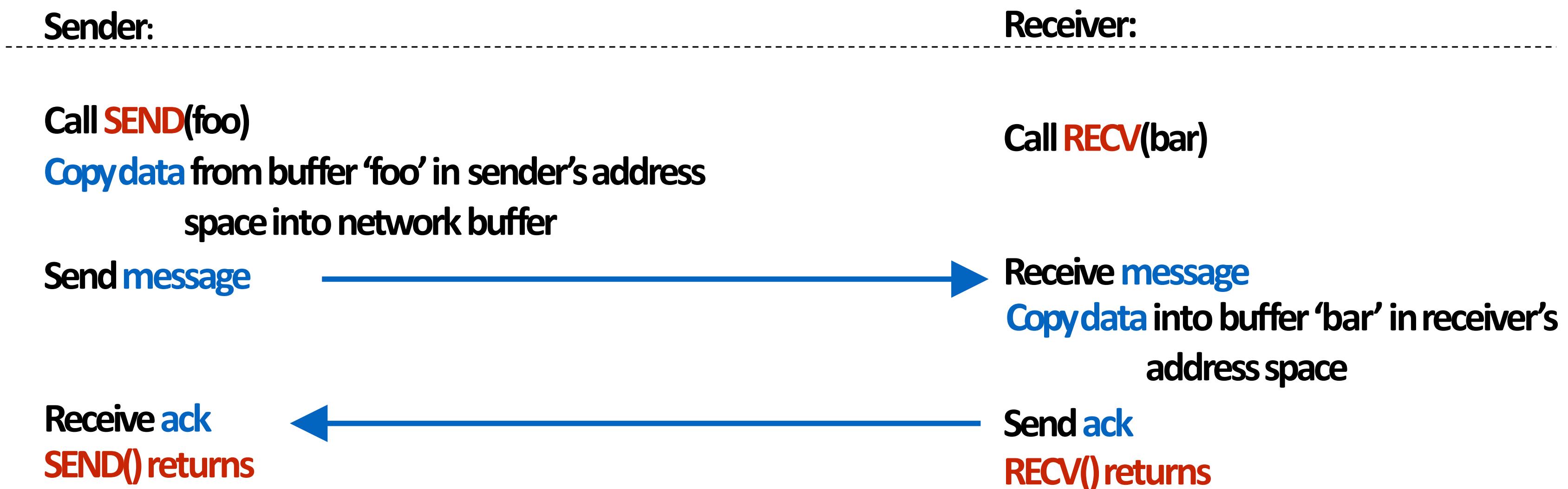
Perform computation
(just like in shared address space version of solver)

All threads send local my_diff to thread 0

Thread 0 computes global diff, evaluates
termination predicate and sends result back to all
other threads

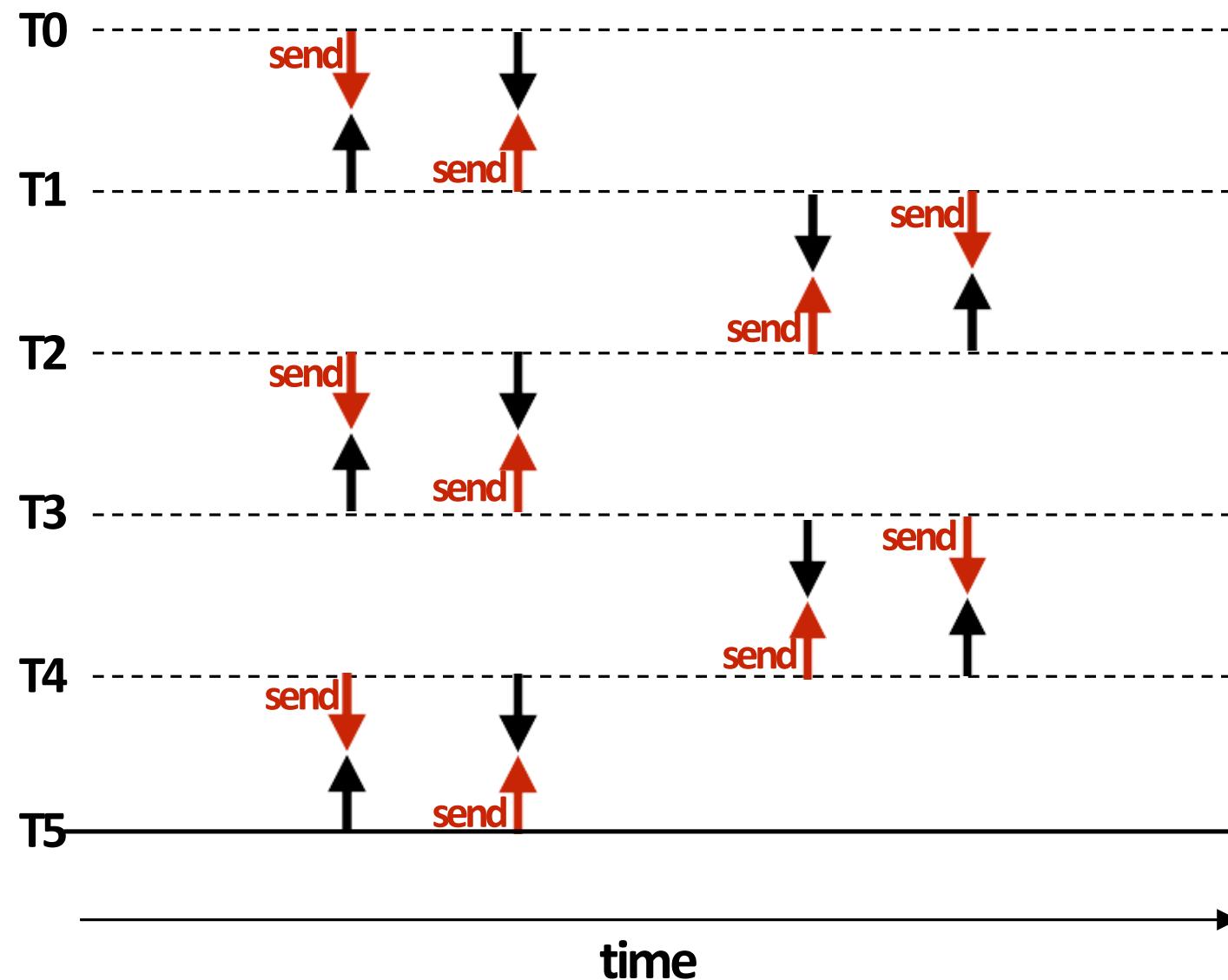
Synchronous (blocking) send and receive

- **send():** call returns when sender receives acknowledgement that message data resides in address space of receiver
- **recv():** call returns when data from received message is copied into address space of receiver and acknowledgement sent back to sender



Message passing solver (fixed to avoid deadlock)

Send and receive ghost rows to “neighbor threads”
Even-numbered threads send, then receive
Odd-numbered thread recv, then send



```
int N;
int tid = get_thread_id();
int rows_per_thread = N / get_num_threads();

float* localA = allocate(rows_per_thread+2, N+2);

// assume localA is initialized with starting values
// assume MSG_ID_ROW, MSG_ID_DONE, MSG_ID_DIFF are constants used as msg ids
///////////
void solve() {
    bool done = false;
    while (!done) {

        float my_diff = 0.0f;

        if (tid % 2 == 0) {
            sendDown(); recvDown();
            sendUp();   recvUp();
        } else {
            recvUp();  sendUp();
            recvDown(); sendDown();
        }
        ...
    }
}
```

Non-blocking asynchronous send/recv

- **send(): call returns immediately**
 - Buffer provided to send() cannot be modified by calling thread since message processing occurs concurrently with thread execution
 - Calling thread can perform other work while waiting for message to be sent
- **recv(): posts intent to receive in the future, returns immediately**
 - Use checksend(), checkrecv() to determine actual status of send/receipt
 - Calling thread can perform other work while waiting for message to be received

Sender:

Call SEND(foo)

SEND returns handle h1

Copy data from 'foo' into network buffer

Send message

Call CHECKSEND(h1) //if message sent, now safe for thread to modify 'foo'

Receiver:

Call RECV(bar)

RECV(bar) returns handle h2

Receive message

Messaging library copies data into 'bar'

Call CHECKRECV(h2)

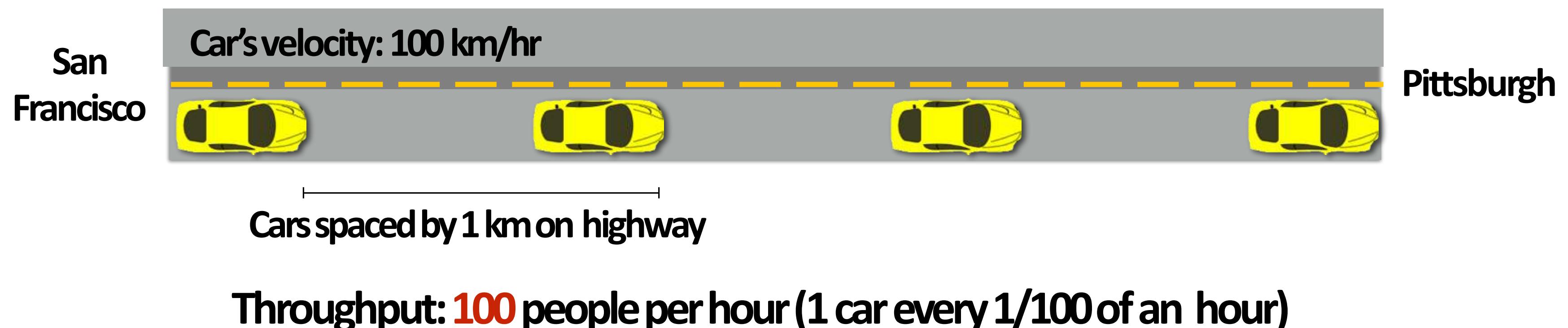
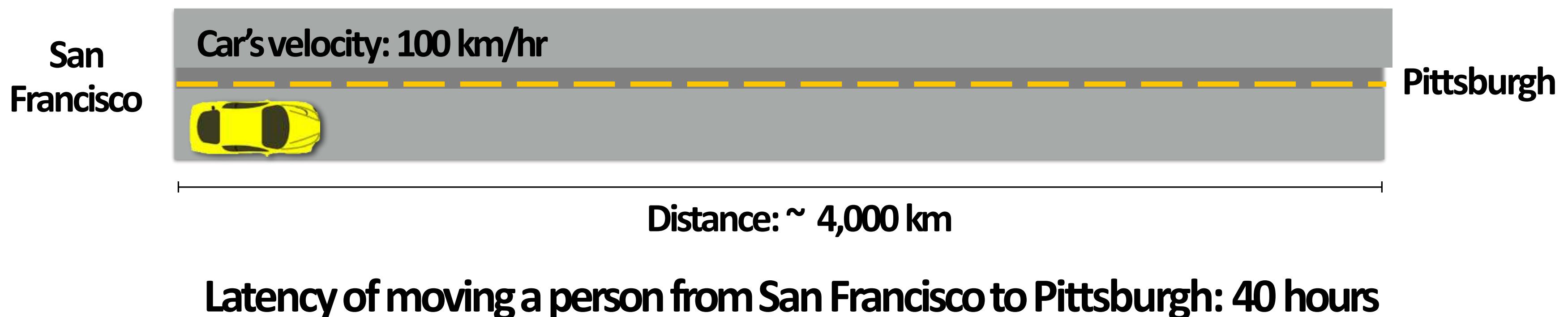
//if received, now safe for thread

//to access 'bar'

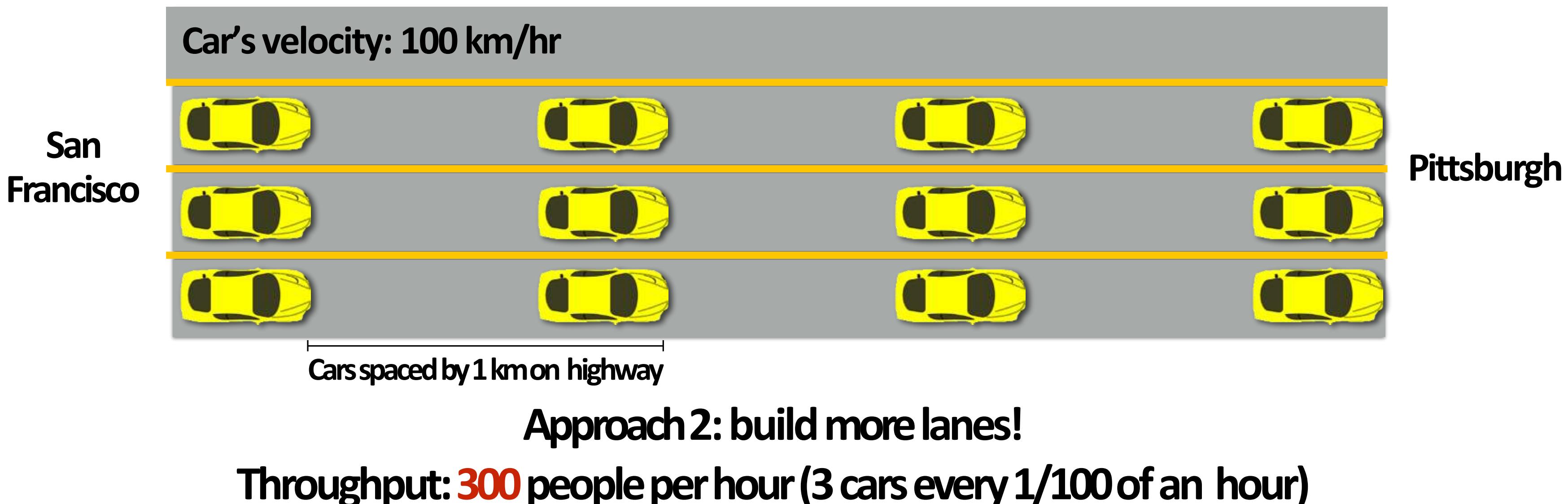
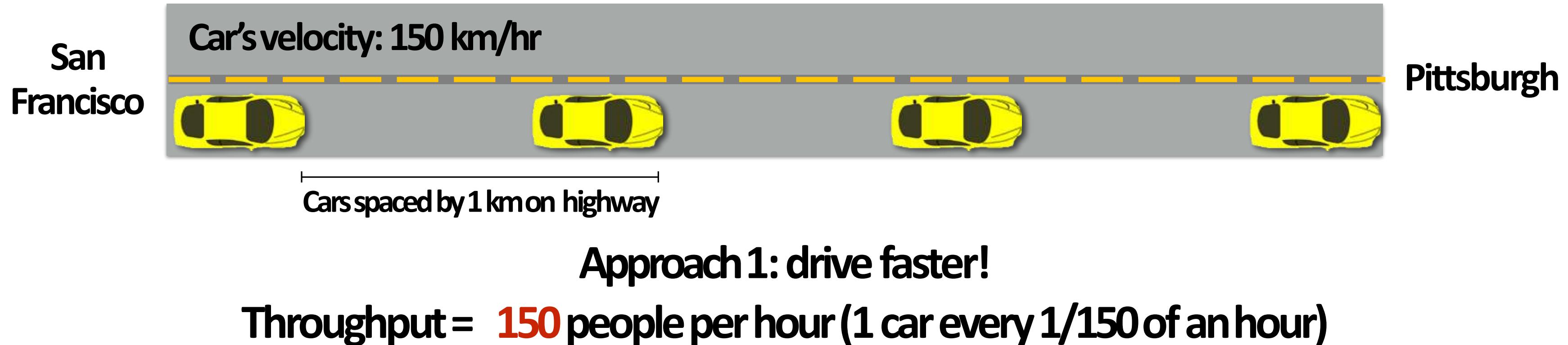
RED TEXT = executes concurrently with application thread

Everyone wants to get to Pittsburgh!

(Latency vs. throughput review)



Improving throughput



Review: latency vs throughput

Latency

The amount of time needed for an operation to complete.

A memory load that misses the cache has a latency of 200 cycles

A packet takes 20 ms to be sent from my computer to Google

Asking a question on Piazza gets a response in 10 minutes

Bandwidth

The rate at which operations are performed.

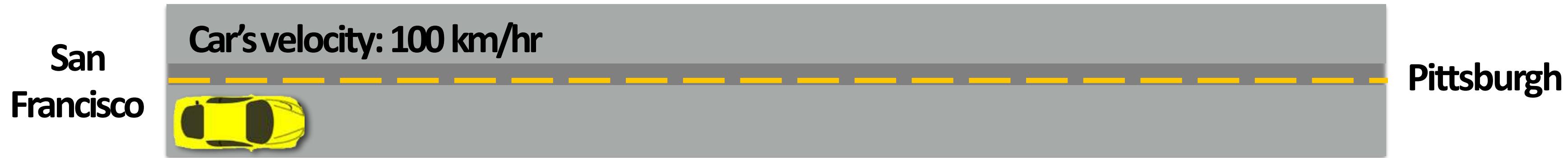
Memory can provide data to the processor at 25 GB/sec.

A communication link can send 10 million messages per second

The TAs answer 50 questions per day on Piazza

What if only one car can be on the highway at a time?

When car on highway reaches Pittsburgh, the next car leaves San Francisco.



Latency of moving a person from San Francisco to Pittsburgh: 40 hours

$$\text{Throughput} = 1/\text{latency}$$

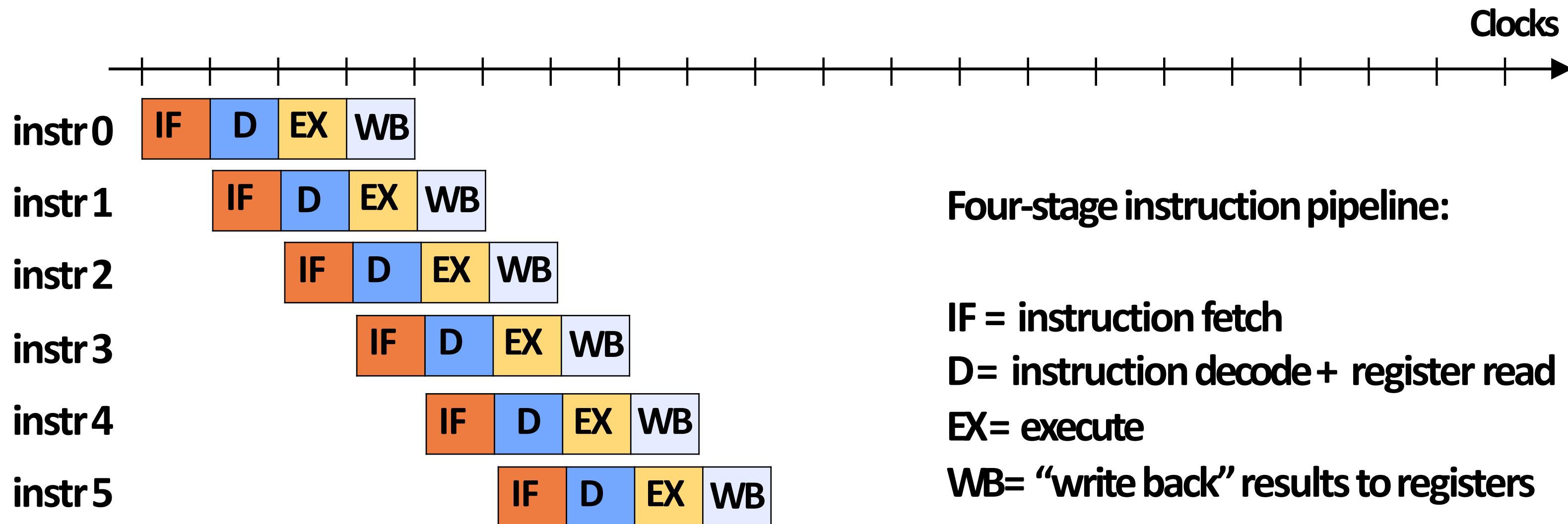
= 1/40 of a person per hour (1 car every 40 hours)

Pipelining

Example: an instruction pipeline

Break execution of each instruction down into several smaller steps

Enables higher clock frequency (only a simple, short operation is done by each part of pipeline each clock)



Four-stage instruction pipeline:

IF = instruction fetch

D= instruction decode + register read

EX= execute

WB= “write back” results to registers

Latency: 1 instruction takes 4 cycles

Throughput: 1 instruction per cycle

(Yes, care must be taken to ensure program correctness when back-to-back instructions are dependent.)

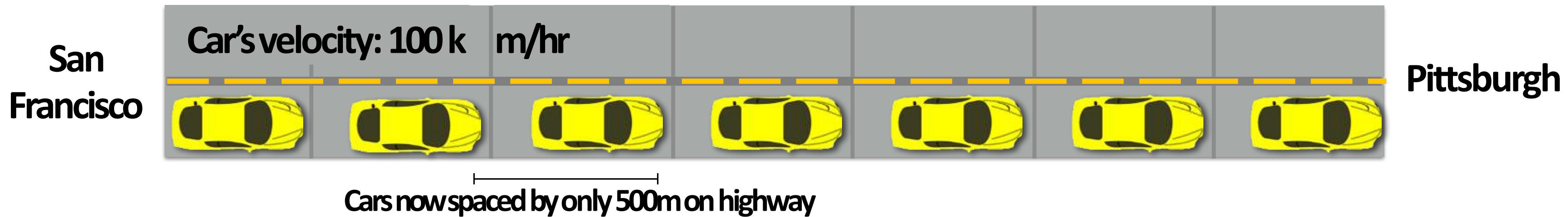
Intel Core i7 pipeline is variable length (it depends on the instruction) ~15-20 stages

Analogy to driving to Pittsburgh example

Task of driving from San Francisco to Pittsburgh is broken up into smaller subproblems that different cars can tackle in parallel
(top: subproblem = drive 1 km, bottom: subproblem = drive 500m)



Throughput = **100 people per hour (1 car every 1/100 of an hour)**



Throughput = **200 people per hour (1 car every 1/200 of an hour)***

*Equivalent throughput to maintaining 1 km spacing of cars and driving at 200km/hr

A simple model of non-pipelined communication

Example: sending a n -bit message

$$T(n) = T_0 + \frac{n}{B}$$

$T(n)$ = transfer time (overall latency of the operation)

T_0 = start-up latency (e.g., time until first bit arrives at destination)

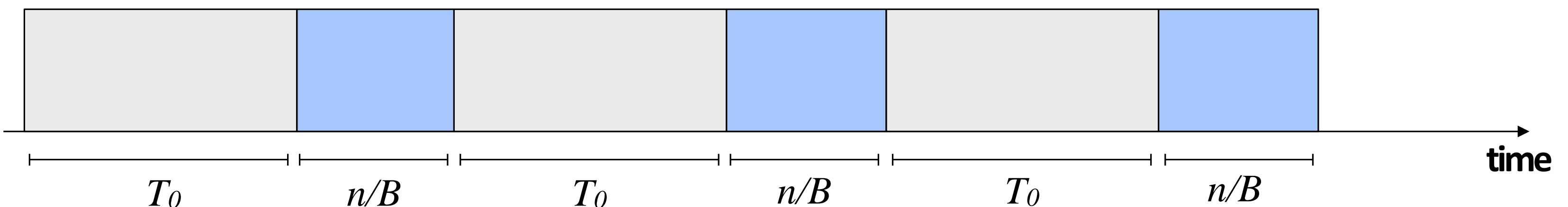
n = bytes transferred in operation

B = transfer rate (bandwidth of the link)

If processor only sends next message once previous message send completes...

“Effective bandwidth” = $n / T(n)$

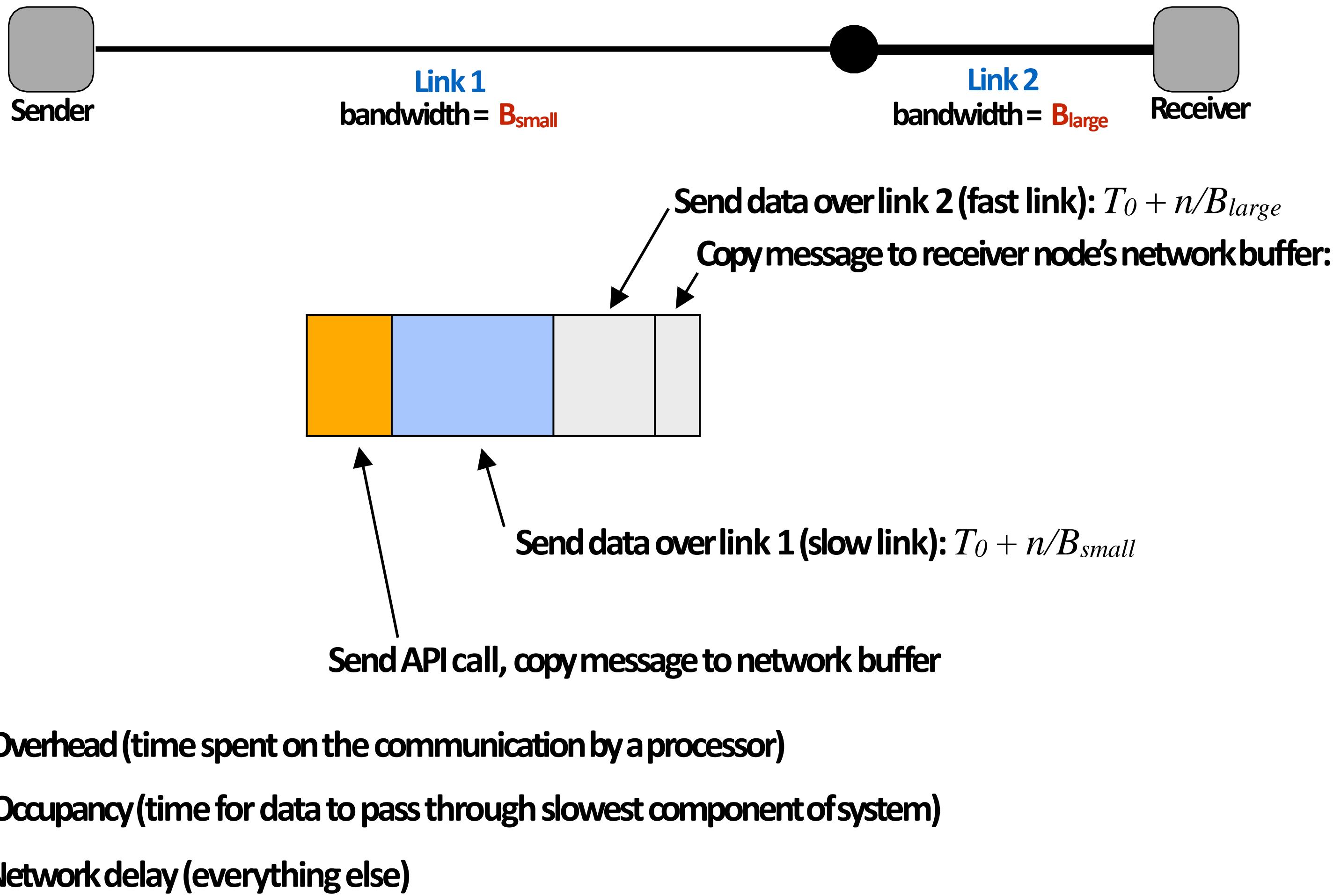
Effective bandwidth depends on transfer size (big transfers amortize startup latency)



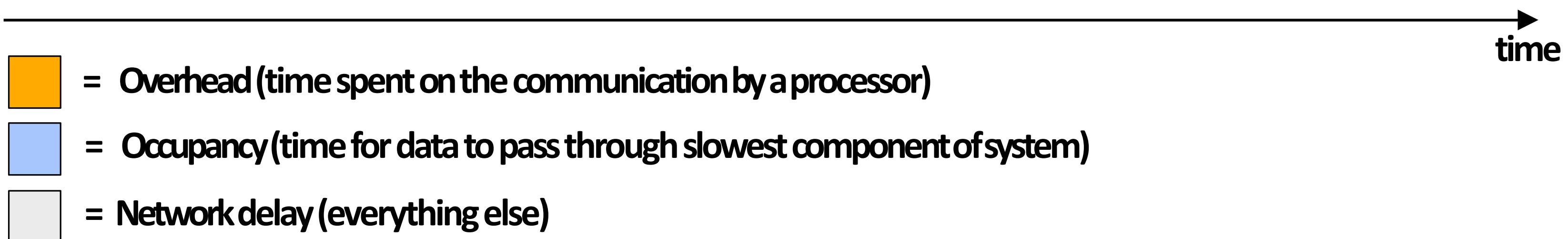
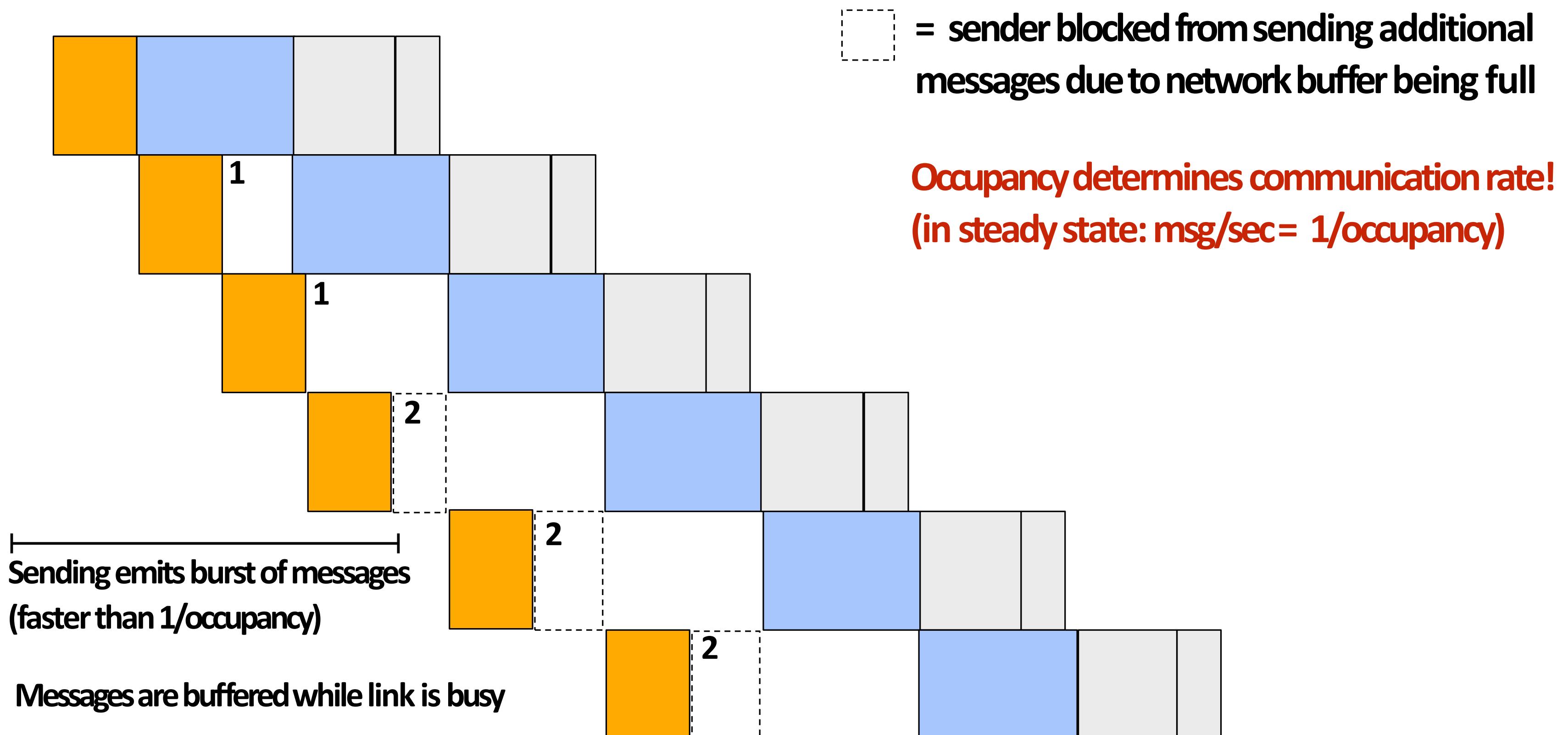
A more general model of communication

Example: sending a n-bit message

Total communication time = overhead + occupancy + network delay



Pipelined communication



Cost

**The effect operations have on program execution time
(or some other metric, e.g., energy consumed...)**

“That function has very high cost” (cost of having to perform the instructions)

“My slow program spends most of its time waiting on memory.” (cost of waiting on memory)

“saxpy achieves low ALU utilization because it is bandwidth bound.” (cost of waiting on memory)

Total communication time = overhead + occupancy + network delay

Total communication cost = communication time - overlap

Overlap: portion of communication **performed concurrently with other work**

“Other work” can be computation or other communication

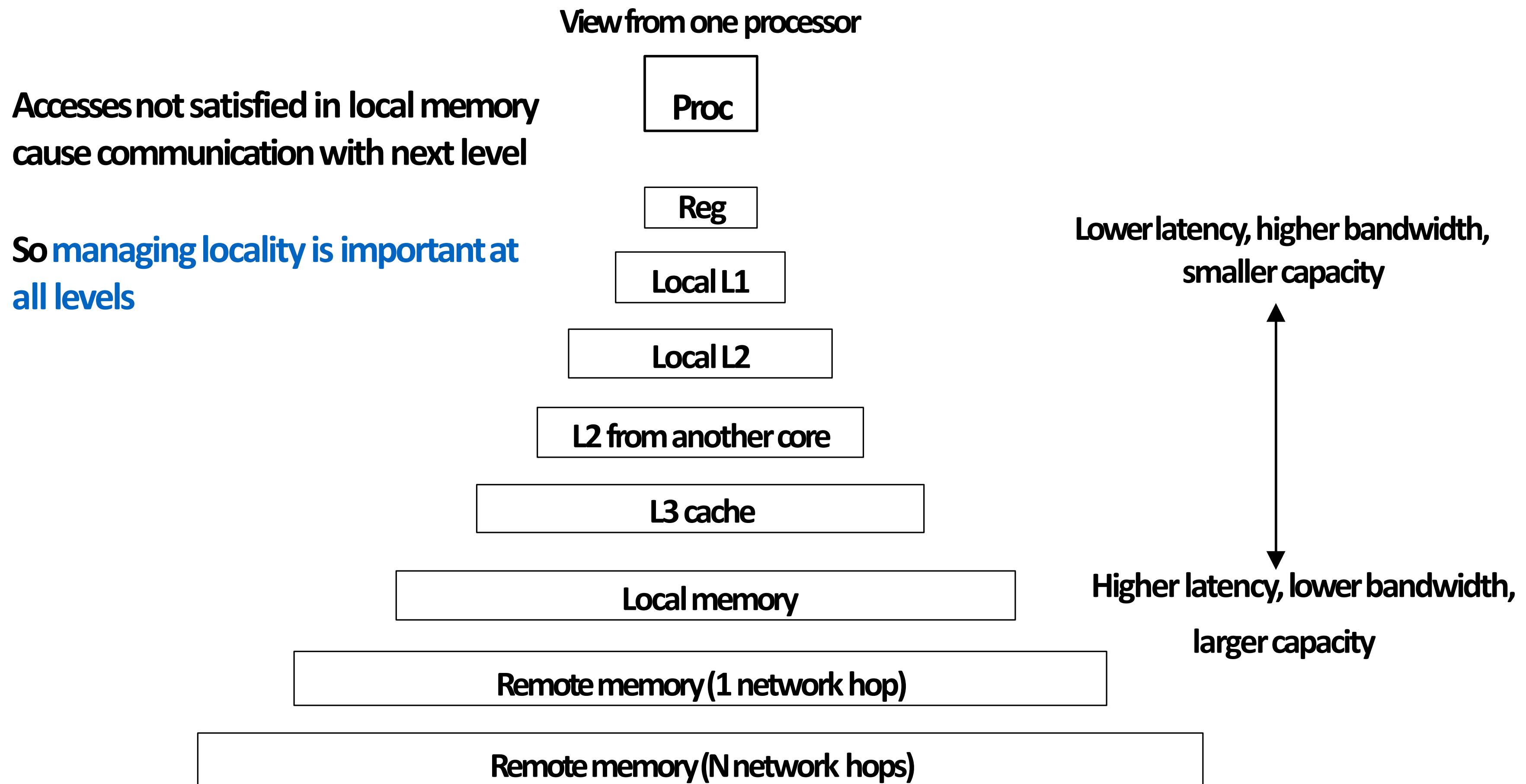
Example 1: Asynchronous message send/recv allows communication to be overlapped with computation

Example 2: Pipelining allows multiple message sends to be overlapped

Think of a parallel system as an **extended memory hierarchy**

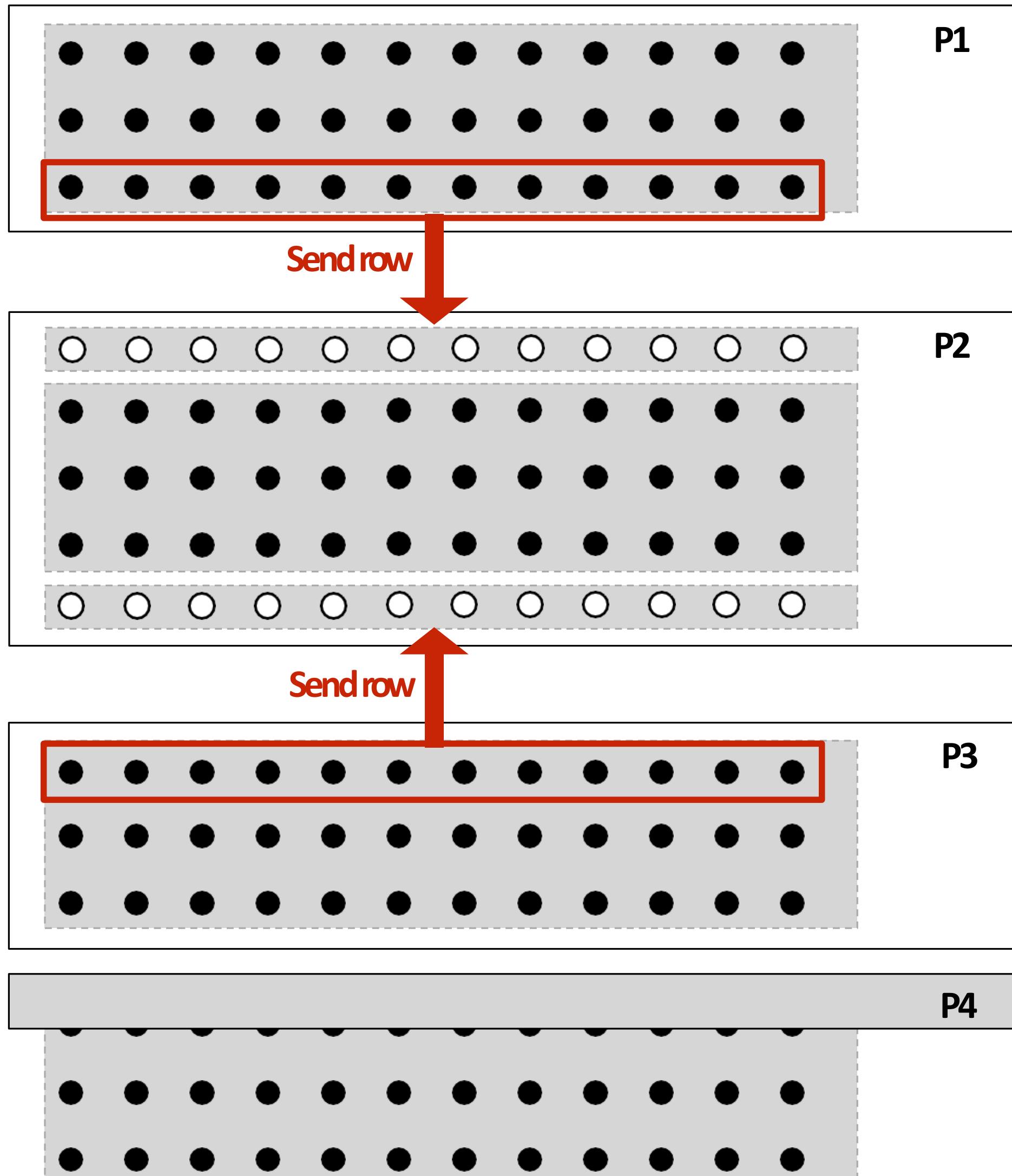
I want you to think of “communication” very generally:

- Communication between a processor and its cache
- Communication between processor and memory (e.g., memory on same machine)
- Communication between processor and a remote memory
(e.g., memory on another node in the cluster, accessed by sending a network message)



**Two reasons for communication:
inherent vs. artifactual communication**

Inherent communication



Communication that must occur in a parallel algorithm. The communication is fundamental to the algorithm.

In our messaging passing example at the start of class, sending ghost rows was inherent communication

Arithmetic intensity

amount of computation (e.g., instructions)

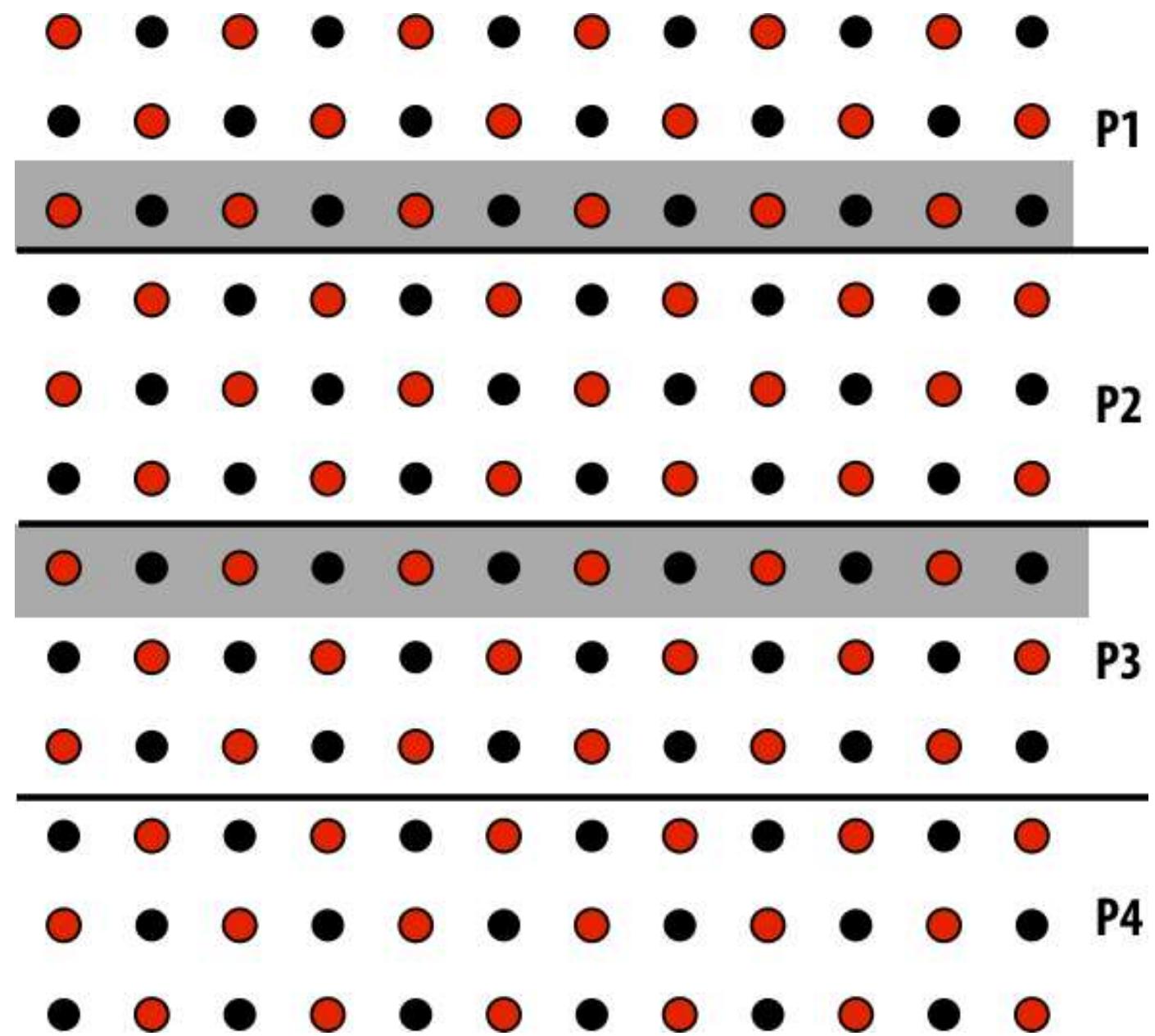
amount of communication (e.g., bytes)

- How much computation is enabled by a given amount of communication
- In general, systems are better at computing than communicating, and so best when this ratio is high.
- High arithmetic intensity (low communication-to-computation ratio) is required to efficiently utilize modern parallel processors since the ratio of compute capability to available bandwidth is high (recall element-wise vector multiple from lecture 2)

Reducing inherent communication

Good assignment decisions can reduce inherent communication
(increase arithmetic intensity)

1D blocked assignment: NxNgrid

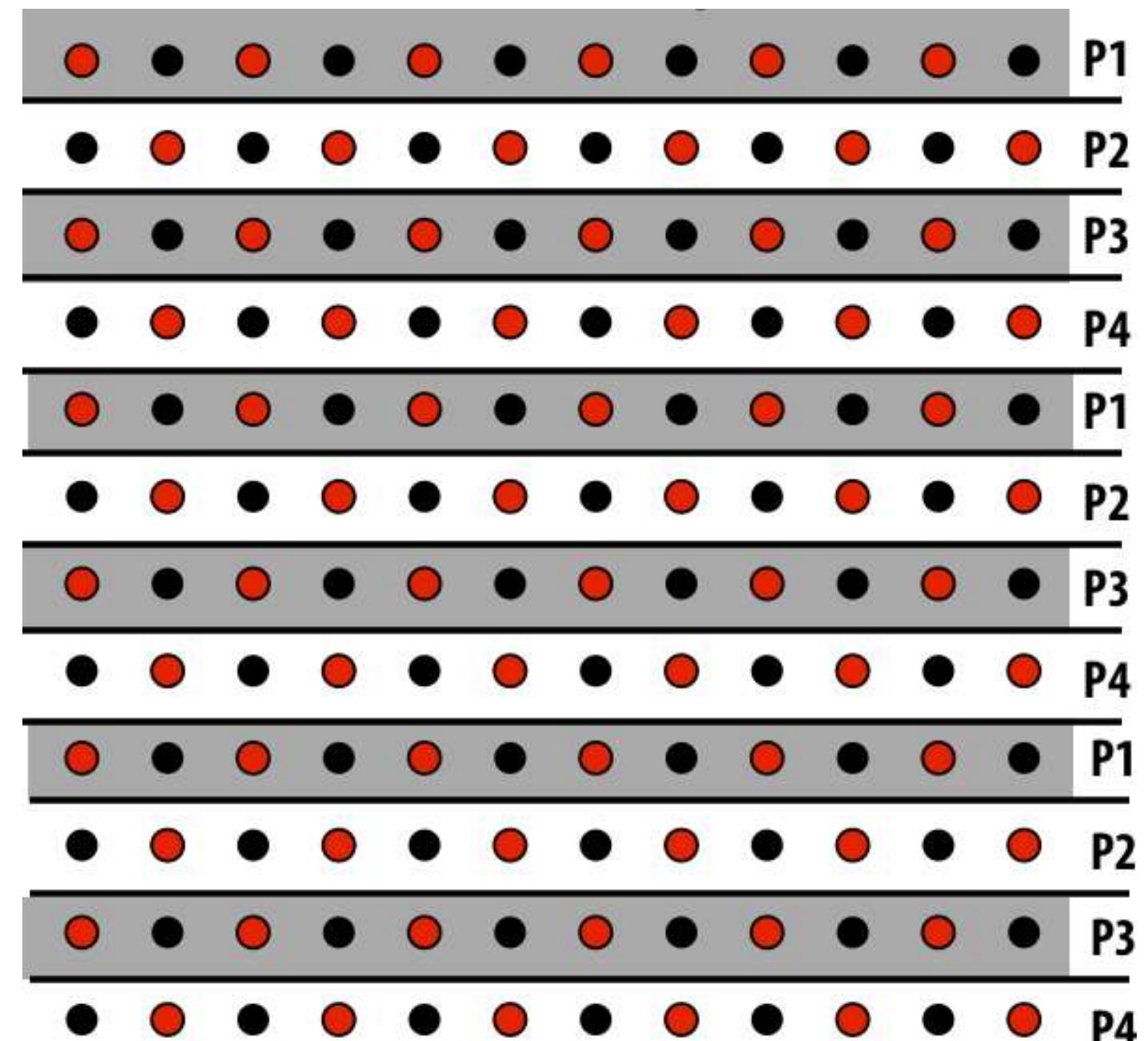


$$\text{elements computed (per processor)} \approx \frac{N^2}{P}$$

$$\text{elements communicated (per processor)} \approx 2N$$

$$\propto \frac{N}{P}$$

1D interleaved assignment: Nx Ngrid

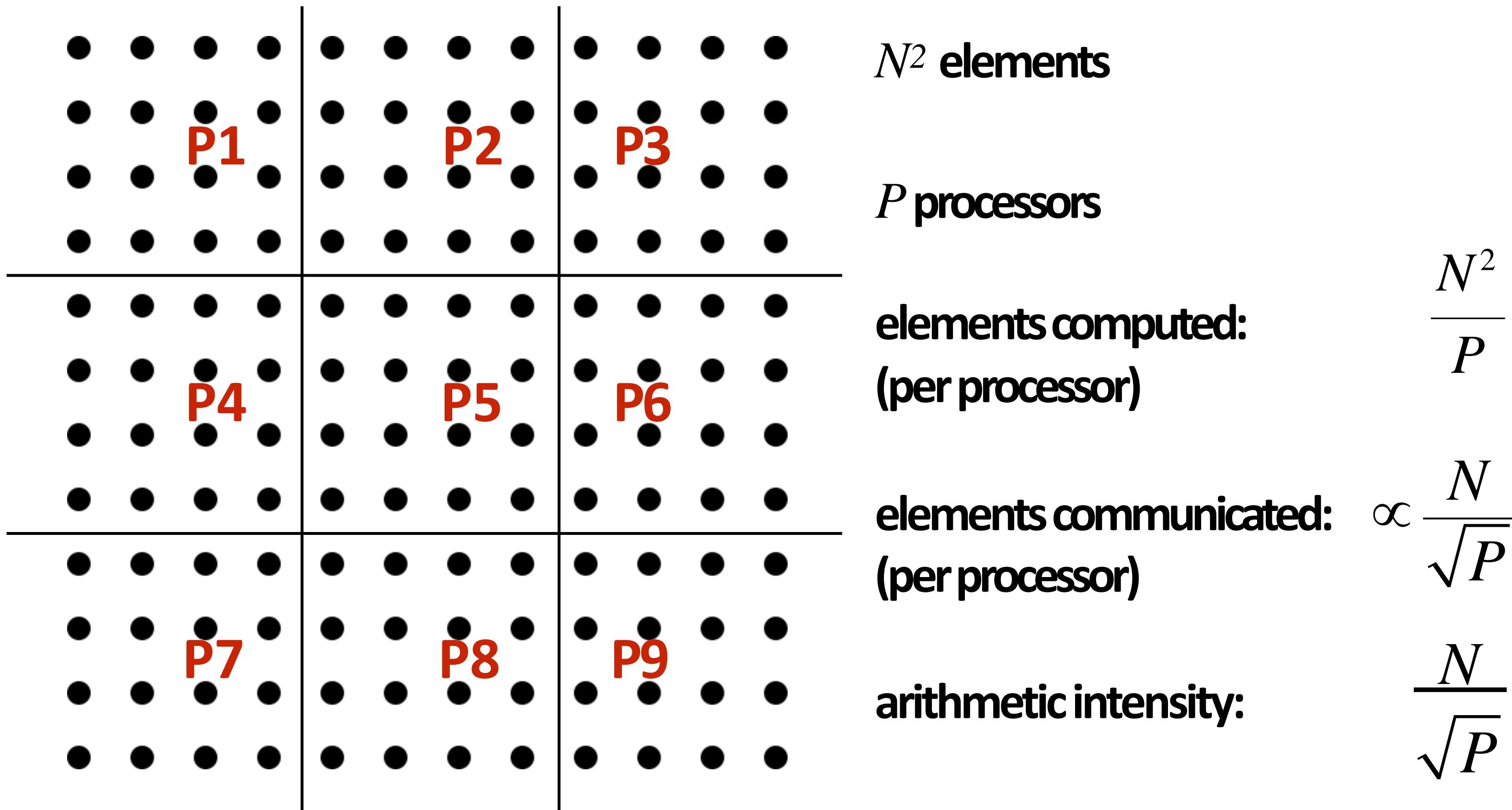


$$\text{elements computed}$$

$$= \frac{1}{2} \text{ elements communicated}$$

Reducing inherent communication

2D blocked assignment: $N \times N$ grid



Asymptotically better communication scaling than 1D blocked assignment

Communication costs increase sub-linearly with P

Assignment captures 2D locality of algorithm

Artifactual communication

- **Inherent communication:** information that **fundamentally must be moved** between processors to carry out the algorithm given the specified assignment (assumes unlimited capacity caches, minimum granularity transfers, etc.)
- **Artifactual communication:** all other communication (artifactual communication results from practical details of system implementation)

Artifactual communication examples

- System might have a minimum **granularity of transfer** (result: system must communicate more data than what is needed)
 - Program loads one 4-byte float value but entire 64-byte cache line must be transferred from memory (16x more communication than necessary)
- System might have **rules of operation** that result in unnecessary communication:
 - Program stores 16 consecutive 4-byte float values, but entire 64-byte cache line is first loaded from memory, and then subsequently stored to memory (2x overhead)
- Poor placement of data in distributed memories (data doesn't reside near processor that accesses it the most)
- Finite replication capacity (same data communicated to processor multiple times because local storage (e.g., cache) is too small to retain it between accesses)

Caches: The three (now four) Cs

- **Cold miss**

First time data touched. Unavoidable in a sequential program.

- **Capacity miss**

Working set is larger than cache. Can be avoided/reduced by increasing cache size.

- **Conflict miss**

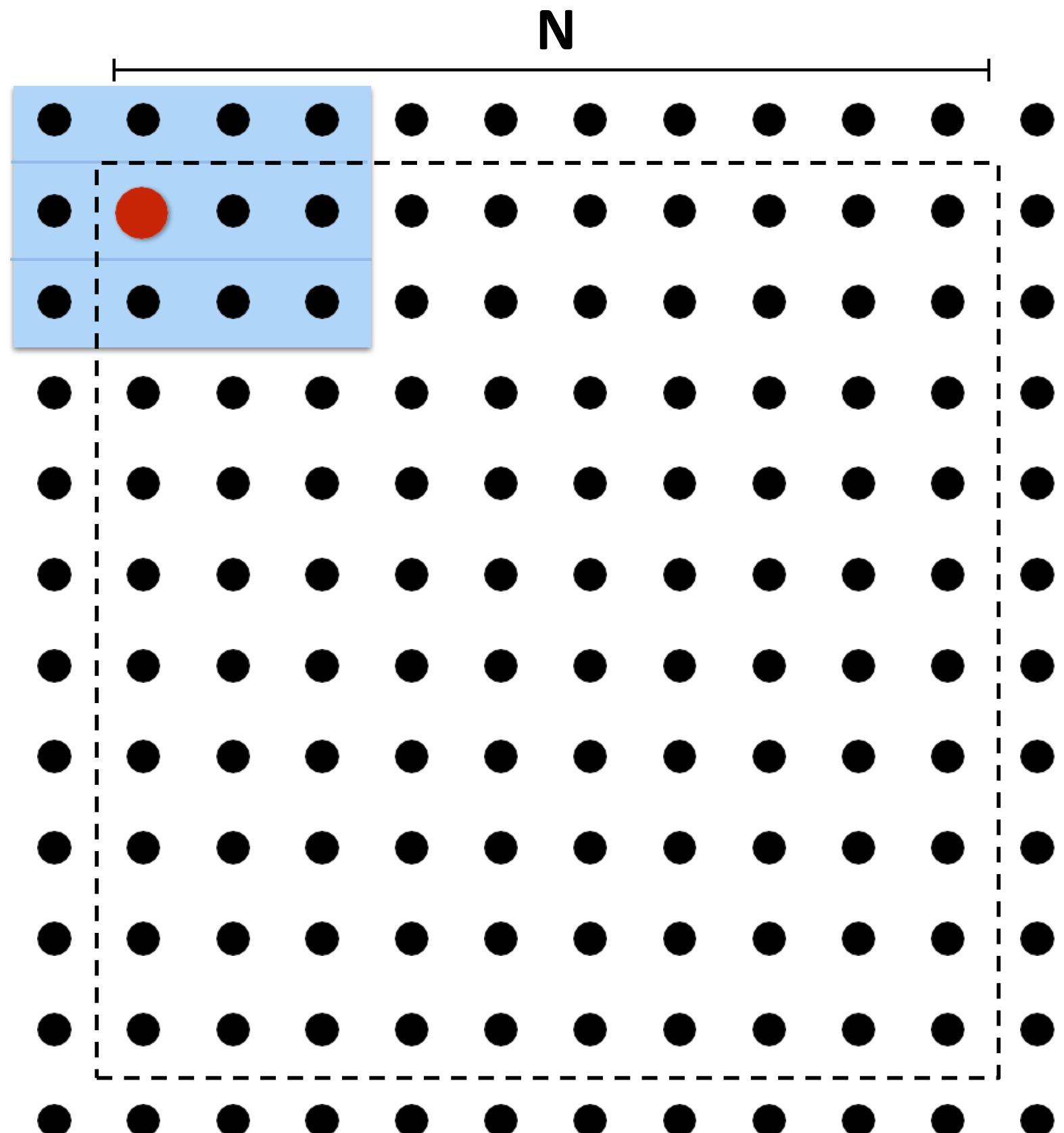
Miss induced by cache management policy. Can be avoided/reduced by changing cache associativity, or data access pattern in application.

- **Communication miss (new)**

Due to inherent or artifactual communication in parallel system

Techniques for reducing communication

Data access in grid solver: row-major traversal



Assume row-major grid layout.

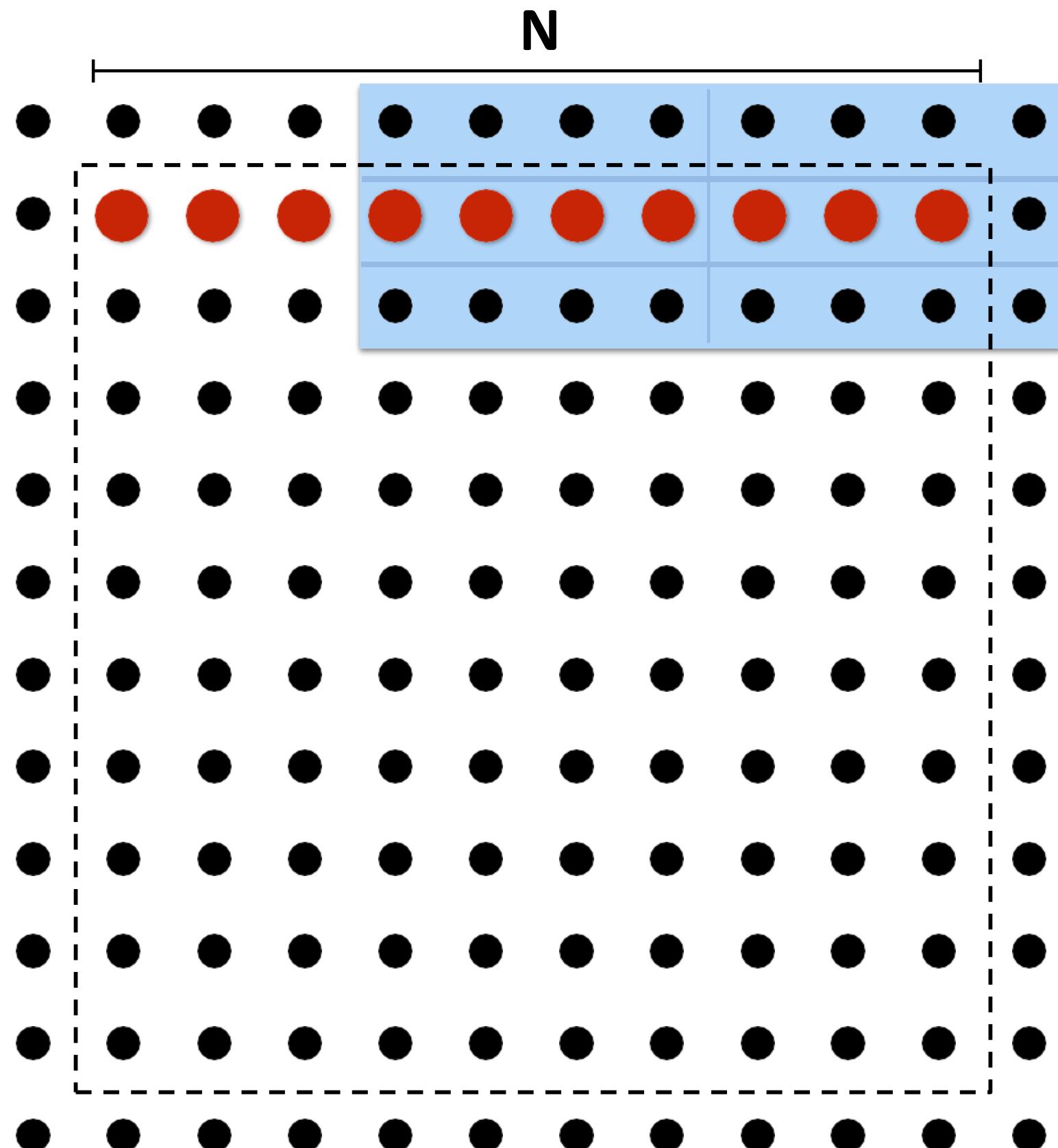
Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

Recall grid solver application.

Blue elements show data in cache after update to red element.

Data access in grid solver: row-major traversal



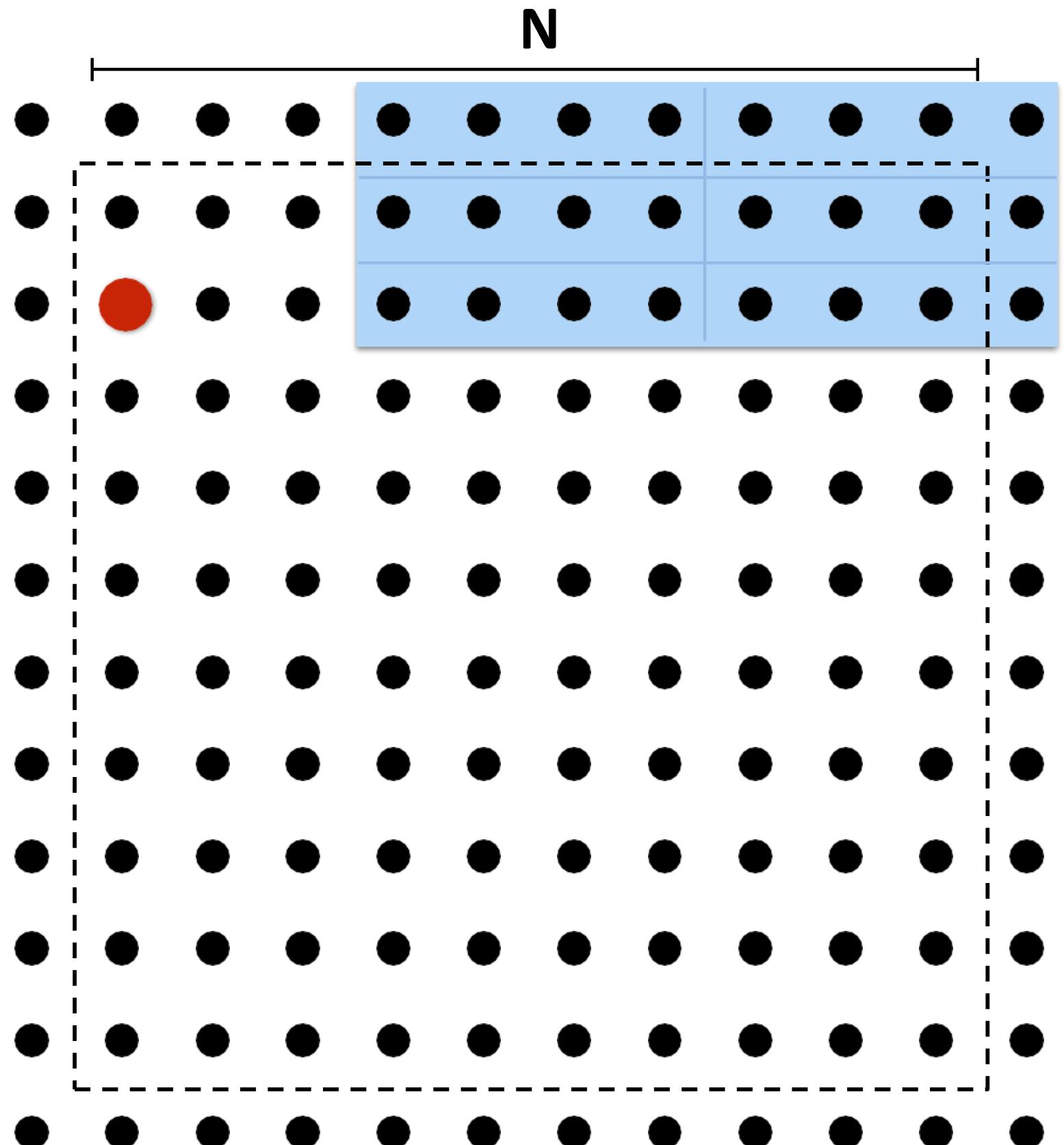
Assume row-major grid layout.

Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

Blue elements show data in cache at end
of processing first row.

Problem with row-major traversal: long time between accesses to same data



Assume row-major grid layout.

Assume cache line is 4 grid elements.

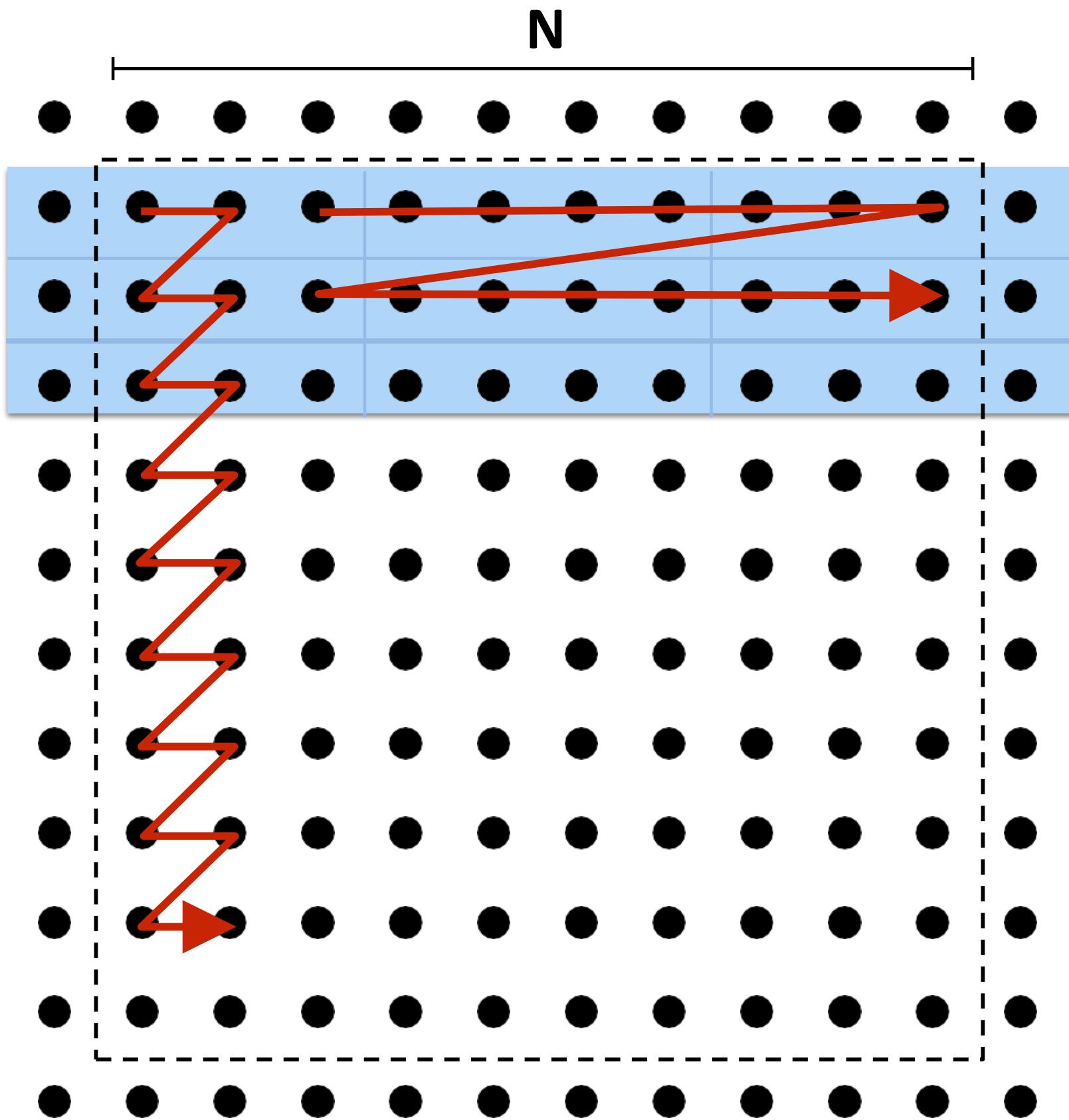
Cache capacity is 24 grid elements (6 lines)

Although elements (0,2) and (1,1) had been accessed previously, they are no longer present in cache at start of processing row 2

(What type of miss is this?)

This program loads three lines for every four elements.

Improving temporal locality by changing grid traversal order



Assume row-major grid layout.

Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

“Blocked” iteration order.

Now three lines for every eight elements.

Improving temporal locality by fusing loops

```
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] * B[i];
}

float* A, *B, *C, *D, *E, *tmp1, *tmp2;

// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Two loads, one store per math op
(arithmetic intensity = 1/3)

Two loads, one store per math op
(arithmetic intensity = 1/3)

Overall arithmetic intensity = 1/3

```
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
        E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

Four loads, one store per 3 math ops
(arithmetic intensity = 3/5)

Code on top is more modular (e.g, array-based math library like numpy in Python)
Code on bottom performs much better. Why

Improve arithmetic intensity by sharing data

- **Exploit sharing: co-locate tasks that operate on the same data**
 - Schedule threads working on the same data structure at the same time on the same processor
 - Reduces inherent communication
- **Example: CUDA thread block**
 - Abstraction used to localize related processing in a CUDA program
 - Threads in block often cooperate to perform an operation (leverage fast access to /synchronization via CUDA shared memory)
 - So GPU implementations always schedule threads from the same block on the same GPU core

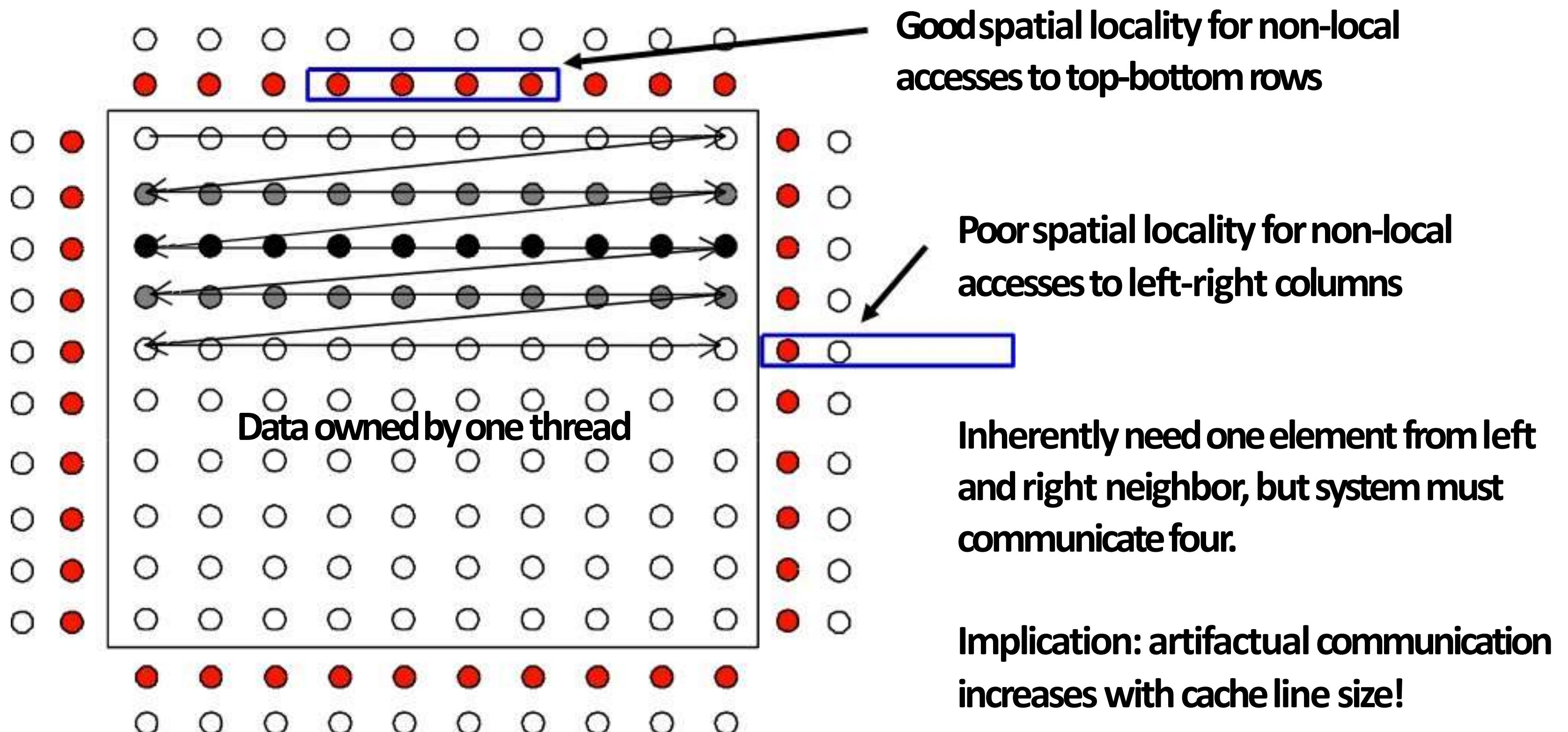
Exploiting spatial locality

- **Granularity of communication can be important because it may introduce artifactual communication**
 - **Granularity of communication /data transfer**
 - **Granularity of cache coherence (will discuss in future lecture)**

Artifactual communication due to comm. granularity

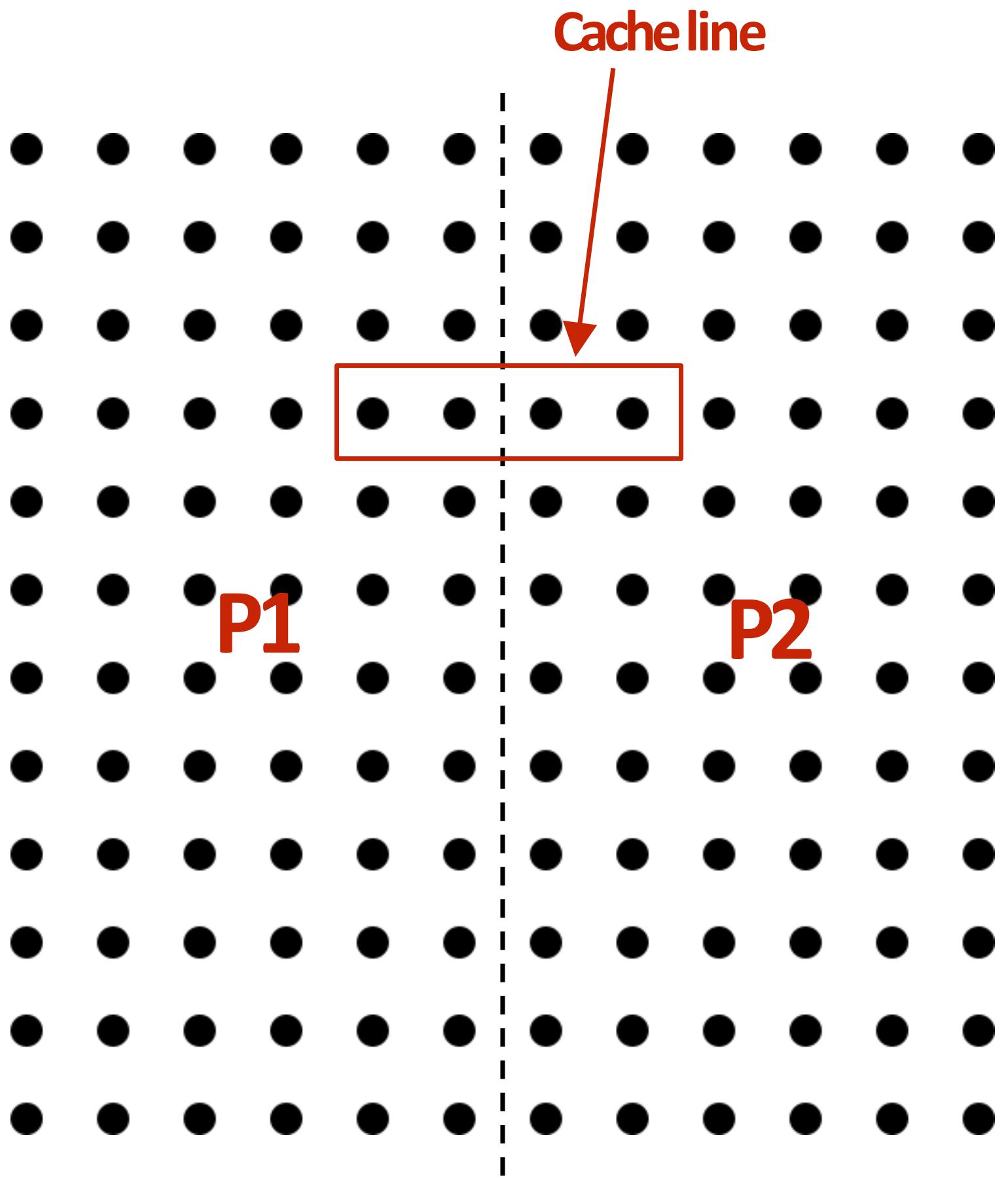
2D blocked assignment of data to processors as described previously.

Assume: communication granularity is a cache line, and a cache line contains four elements



● = required elements assigned to other processors

Artifactual communication due to cache line communication granularity



Data partitioned in half by column. Partitions assigned to threads running on P1 and P2

Threads access their assigned elements
(no inherent communication exists)

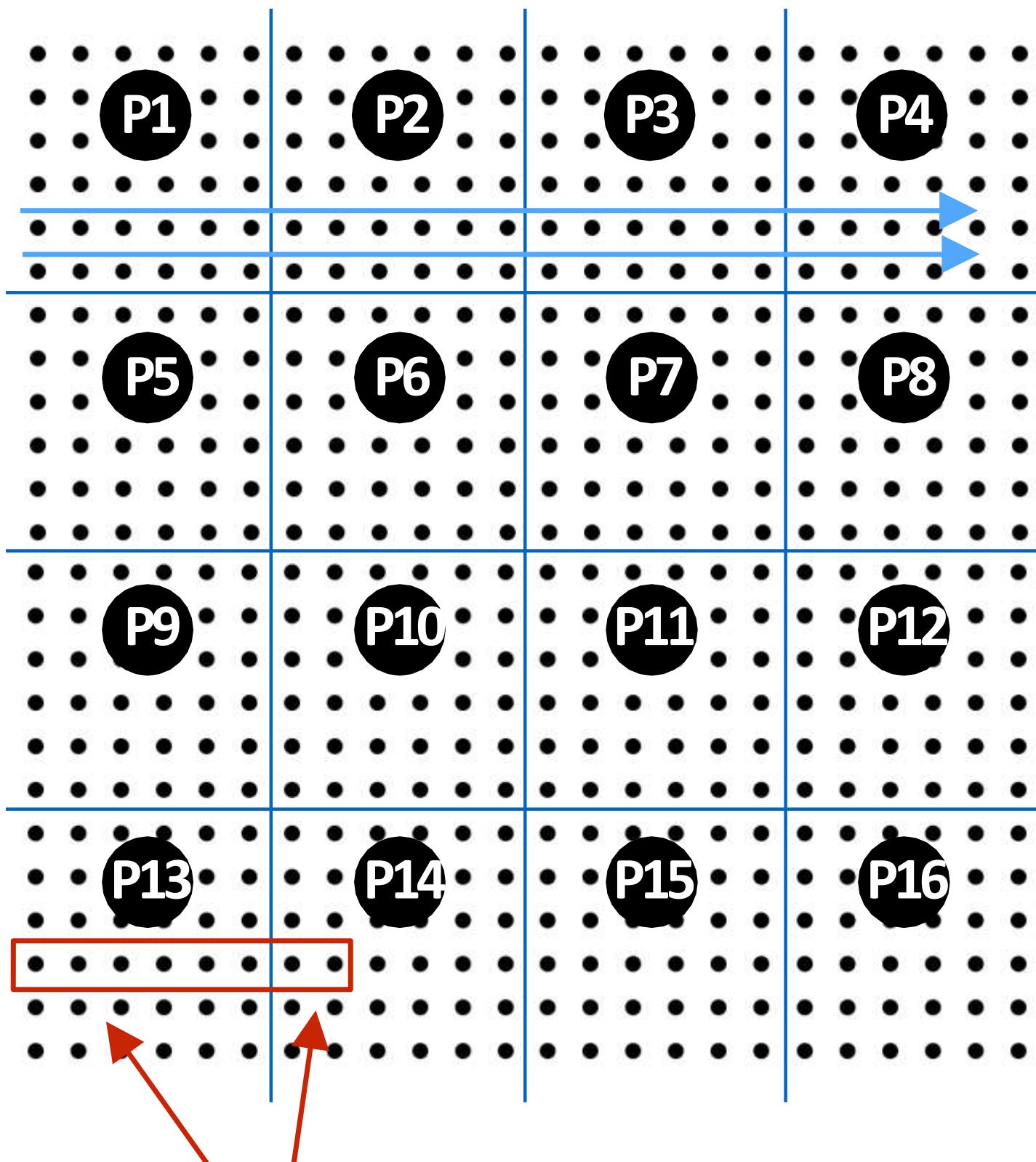
But data access on real machine triggers
(artifactual) communication due to the cache
line being written to by both processors *

*further detail in the upcoming cache coherence lectures

Reducing artifactual comm: blocked data layout

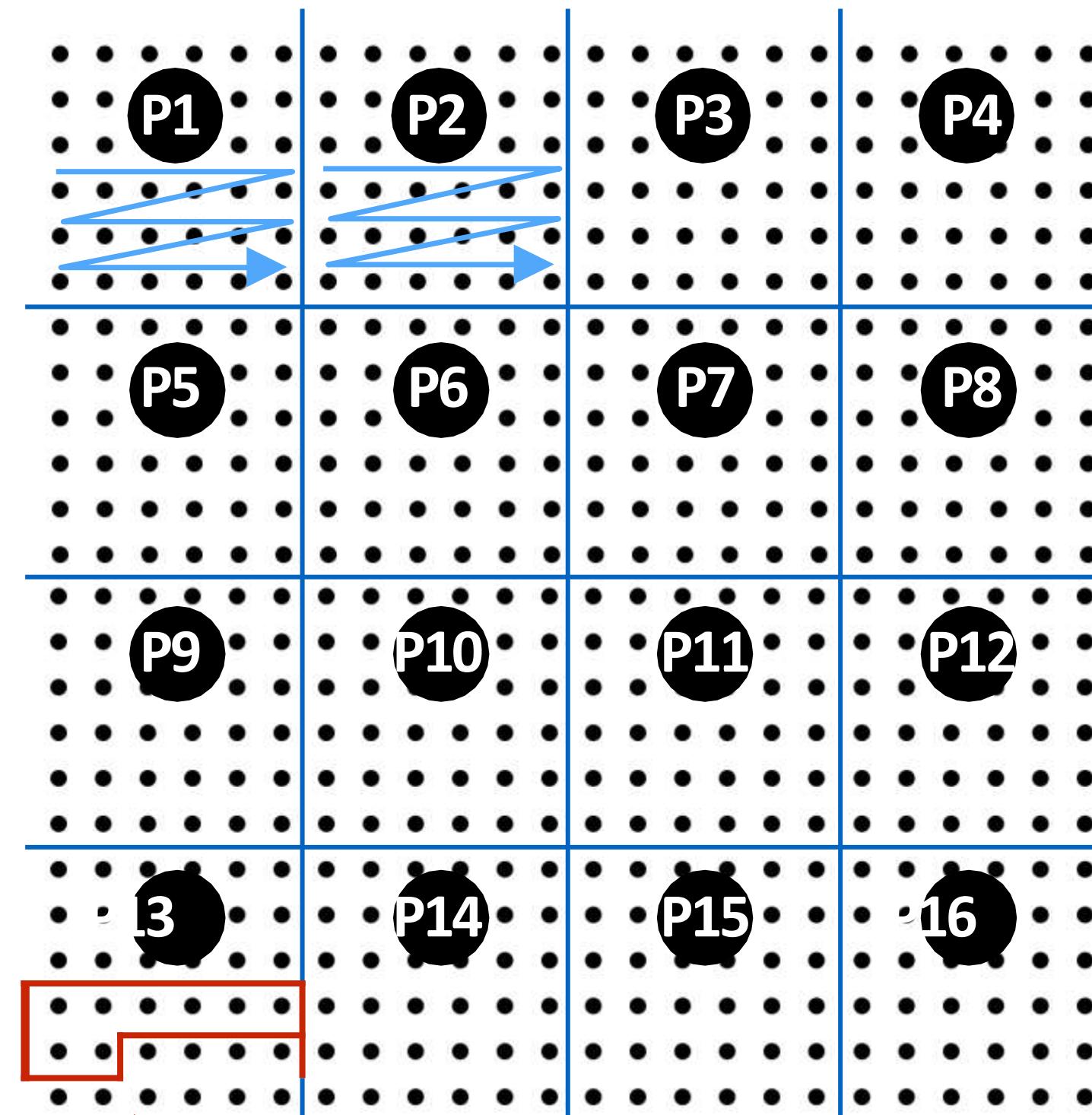
(Blue lines indicate consecutive memory addresses)

2D, row-major array layout



Consecutive addresses
straddle partition boundary

4Darray layout(block-major)



Consecutive addresses remain
within single partition

Note: don't confuse blocked assignment of work to threads (true in both cases above)
with blocked data layout in the address space (only at right)

Contention

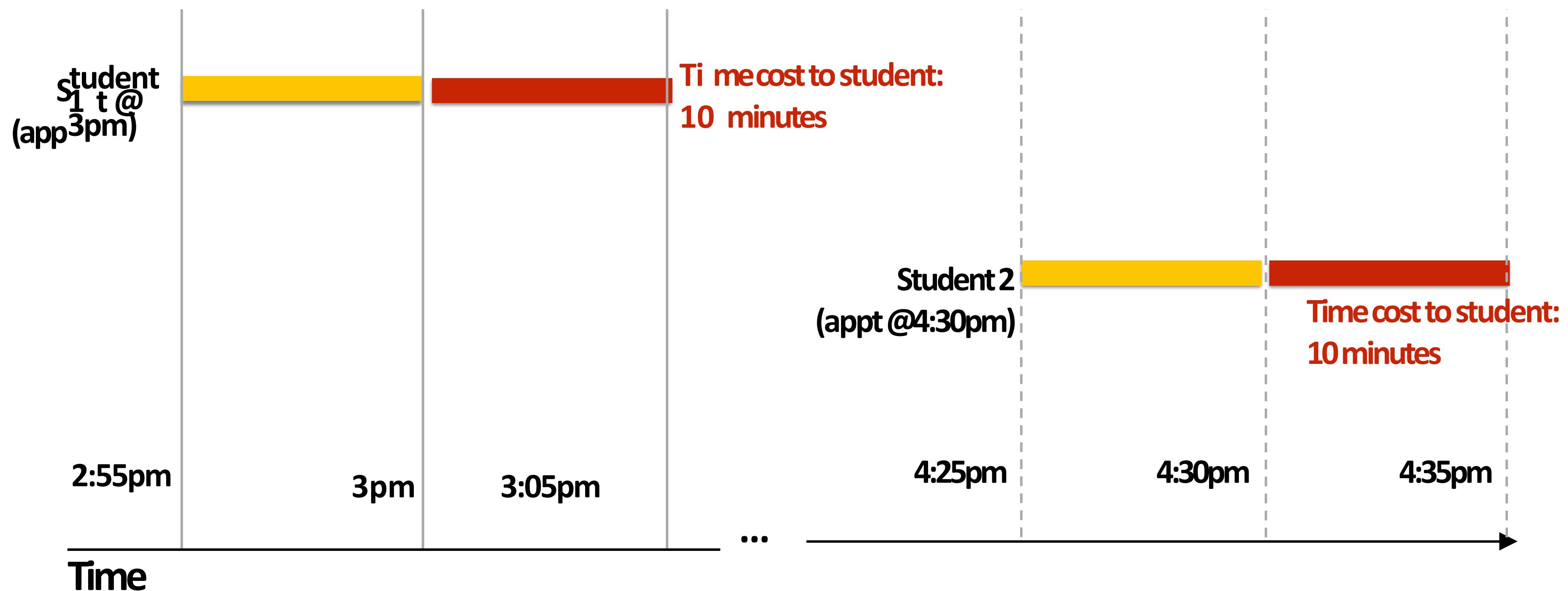
Example: two students make appointments to talk to me about course material (at 3pm and at 4:30pm)

- Operation to perform: Help a student with a question
- Execution resource: Instructor
- Steps in operation:

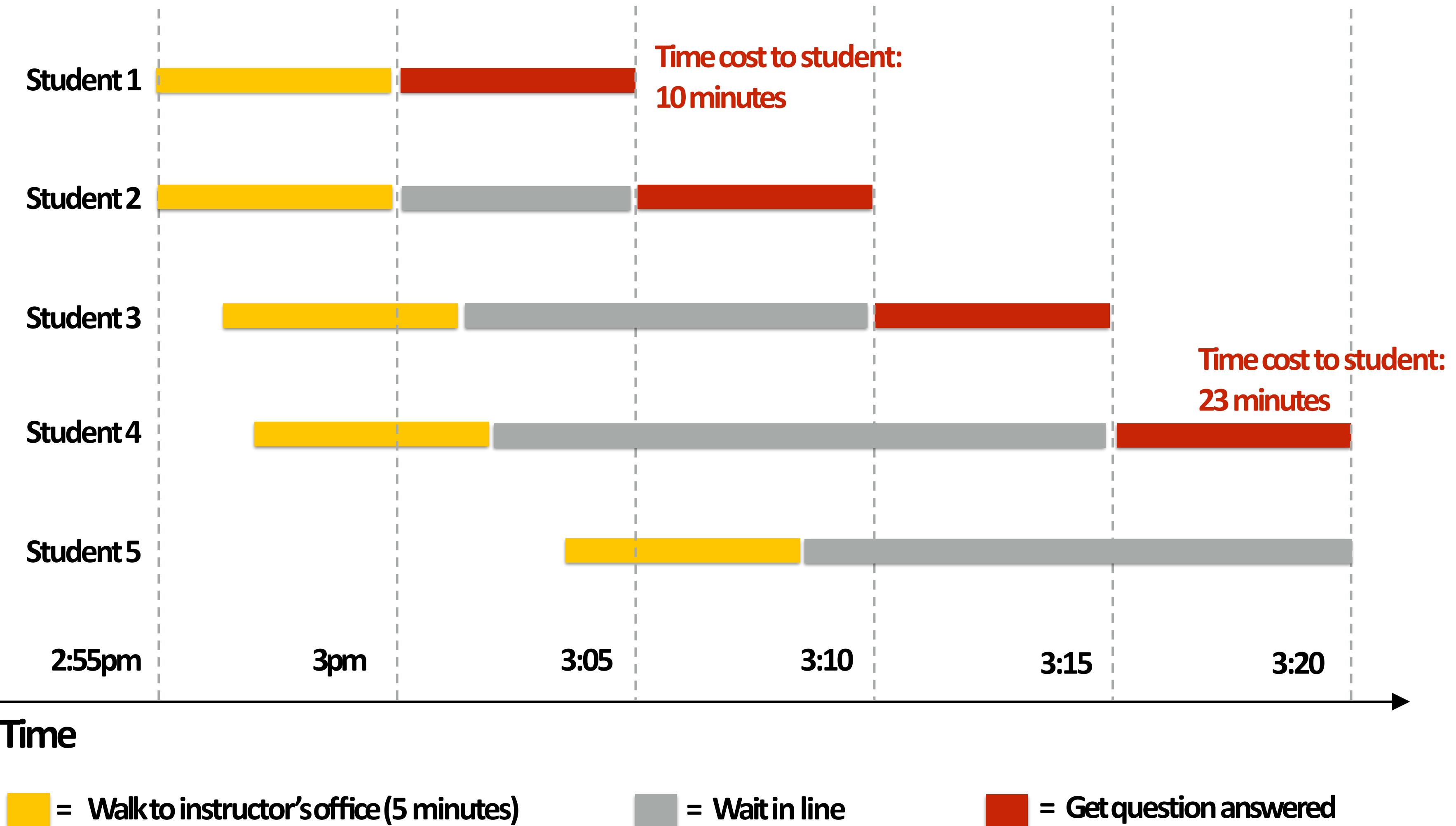
1. Student walks from Gates Cafe to Instructor's office (5 minutes) =

2. Student waits in line (??) =

3. Student gets question answered with insightful answer (5 minutes) =



Office hours from 3-3:20pm (no appointments)

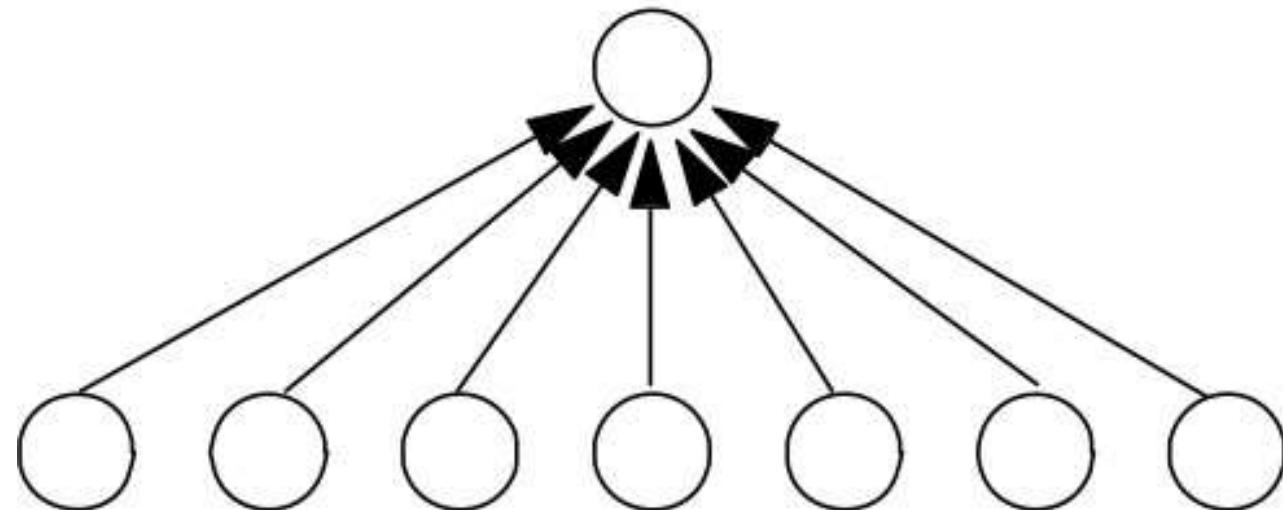


Problem: contention for shared resource results in longer overall operation times (and likely higher cost to students)

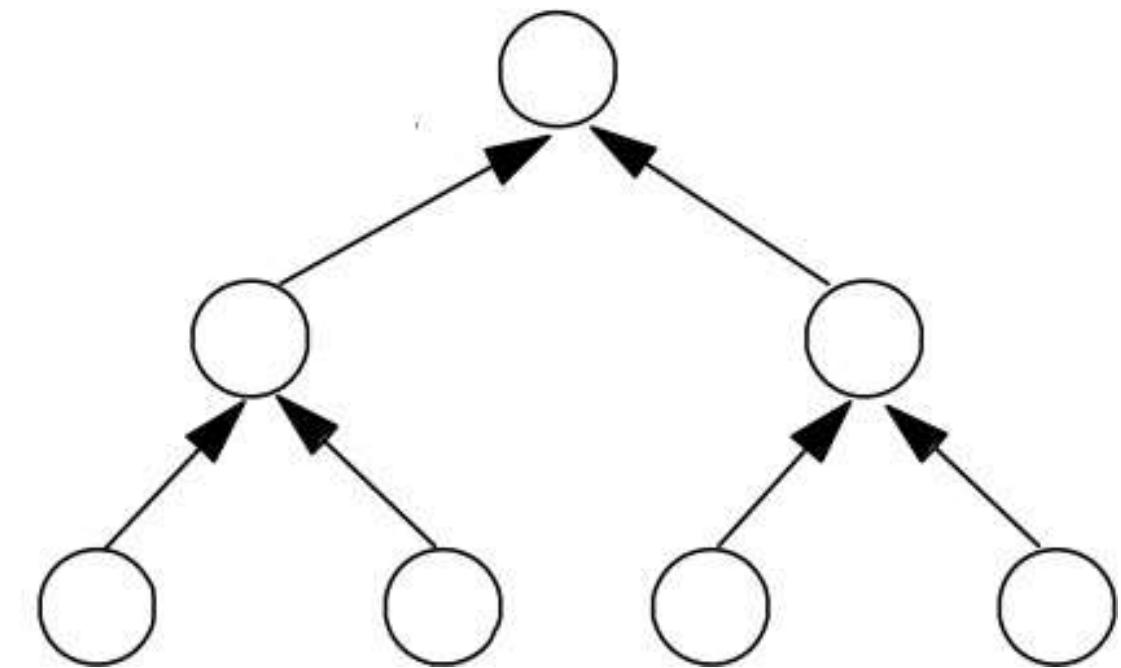
Contention

- A resource can perform operations at a given throughput (number of transactions per unit time)
 - Memory, communication links, servers, TAs at office hours, etc.
- Contention occurs when many requests to a resource are made within a small window of time (the resource is a “hot spot”)

Example: updating a shared variable



Flat communication:
potential for high contention
(but low latency if no contention)



Tree structured communication:
reduces contention
(but higher latency under no contention)

Example: distributed work queues serve to reduce contention (contention in access to single shared workqueue)

Subproblems

(a.k.a. “tasks”, “work to do”)

Set of work queues

(In general, one per worker thread)

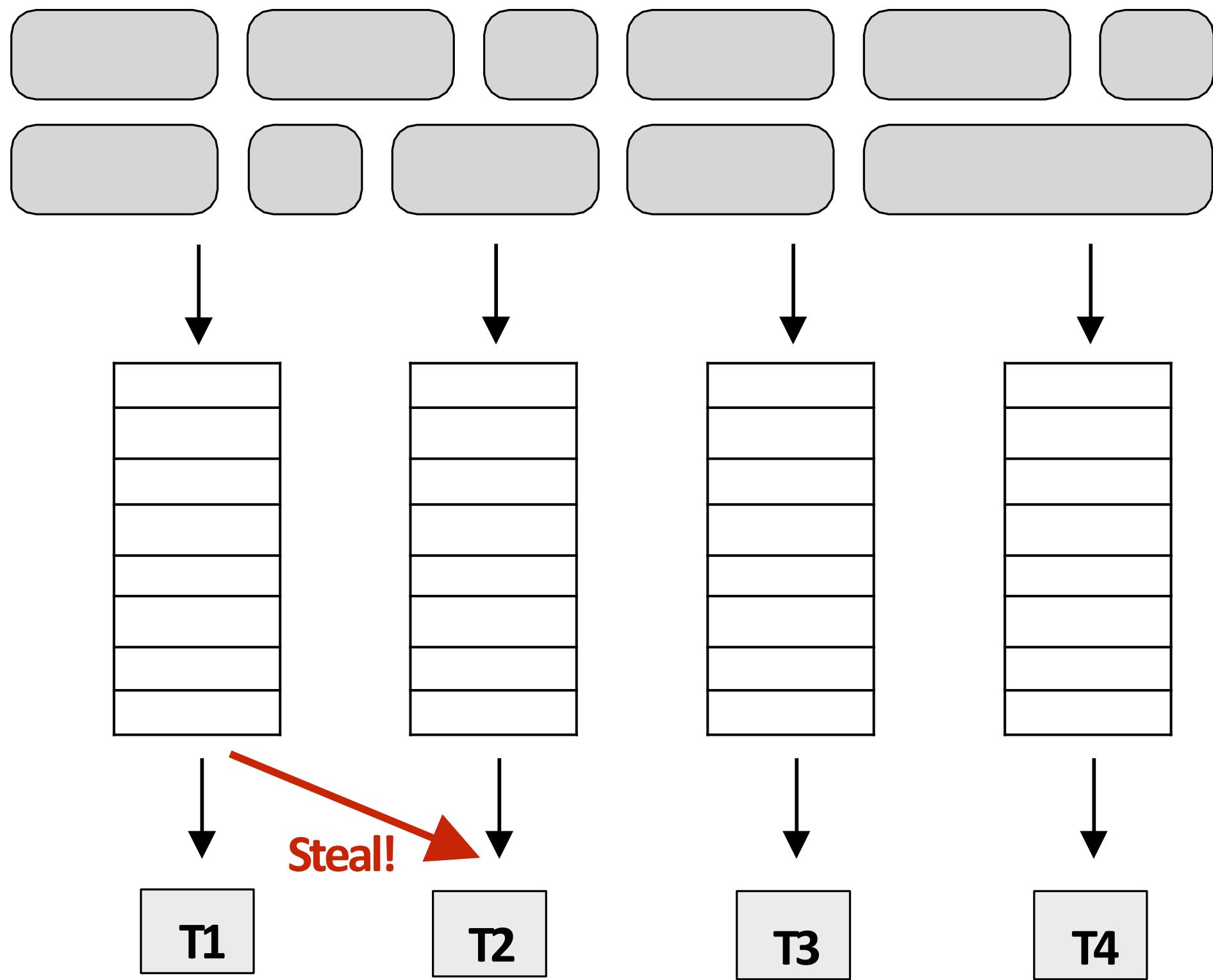
Worker threads:

Pull data from **Own** work queue

Push new work to **Own** work queue

When local work queue is empty...

STEAL work from another work queue



CSCI465/ECEN433

Introduction to Parallel Computing

Spring 2024



Lecture (11)

Workload-Driven Performance Evaluation

You are hired by [insert your favorite chip company here].

You walk in on day one, and your boss says:

***"All of our senior architects have decided to take the year off.
Your job is to lead the design of our next parallel processor."***

What questions might you ask?

Your boss selects the application that matters most to the company
"I want you to demonstrate good performance on this application."

How do you know if you have a good design?

- **Absolute performance?**
 - Often measured as wall clock time
 - Another example: operations per second
- **Speedup: performance improvement due to parallelism?**
 - Execution time of sequential program / execution time on P processors
 - Operations per second on P processors / operations per second of sequential program
- **Efficiency?**
 - Performance per unit resource
 - e.g., operations per second per chip area, per dollar, per watt

Example: create grid of particles data structure on large parallel machine (e.g, a GPU)

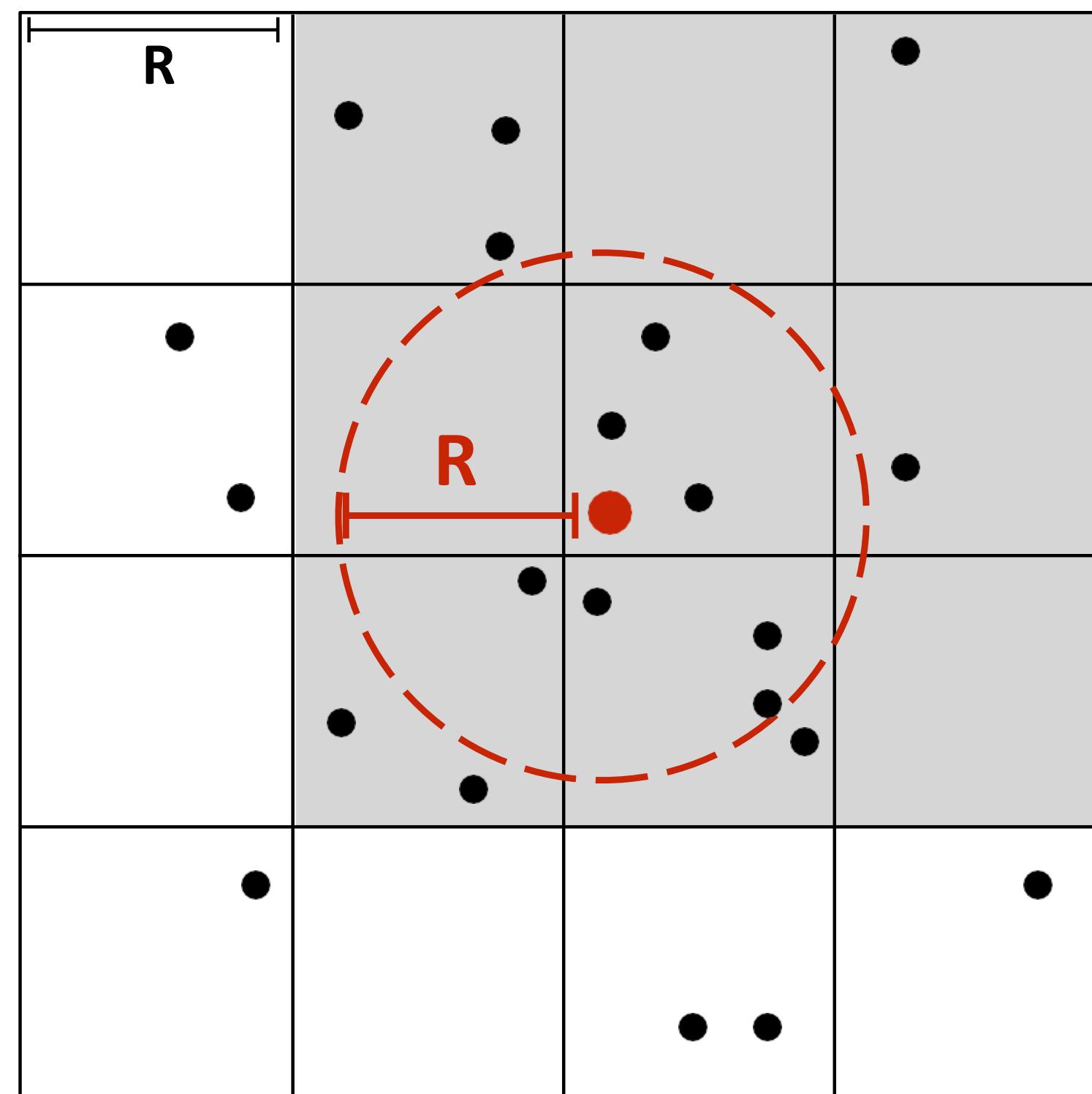
- Problem: place **1M** point particles in a **16-cell uniform grid** based on 2D position
 - Parallel data structure manipulation problem: **build a 2D array of lists**
- **GTX980 GPU:** Up to **2048 CUDA threads per SVMcore**, and **16 SVMcores on the GPU**

0	1	2	3
3			
4	5	1 6 4	7
5		2	
8	9 0	10	11
12	13	14	15

Cell id	Count	Particle id
0	0	
1	0	
2	0	
3	0	
4	2	3, 5
5	0	
6	3	1, 2, 4
7	0	
8	0	
9	1	0
10	0	
11	0	
12	0	
13	0	
14	0	
15	0	

Common use of this structure: N-body problems

- A common operation is to **compute interactions with neighboring particles**
- Example: given particle, find all particles within radius R
 - Create grid with cells of size R
 - Only need to inspect particles in surrounding grid cells



Solution 1: parallelize over cells

- One possible answer is to decompose work by cells: **for each cell, independently compute particles within it** (eliminates contention because no synchronization is required)
 - **Insufficient parallelism:** only 16 parallel tasks, but need thousands of independent tasks to efficiently utilize GPU)
 - **Workinefficient:** performs 16 times more particle-in-cell computations than sequential algorithm

```
list cell_lists[16];           // 2D array of lists

for each cell c               // in parallel
    for each particle p       // sequentially
        if (p is within c)
            append p to cell_lists[c]
```

Solution 2: parallelize over particles

- Another answer: assign one particle to each CUDA thread. Thread computes cell containing particle, then atomically updates list.
 - Massive contention: thousands of threads contending for access to update single shared data structure

```
list cell_list[16];      // 2D array of lists
lock cell_list_lock;

for each particle p          // in parallel
    c = compute cell containing p
    lock(cell_list_lock)
    append p to cell_list[c]
    unlock(cell_list_lock)
```

Solution 3: use finer-granularity locks

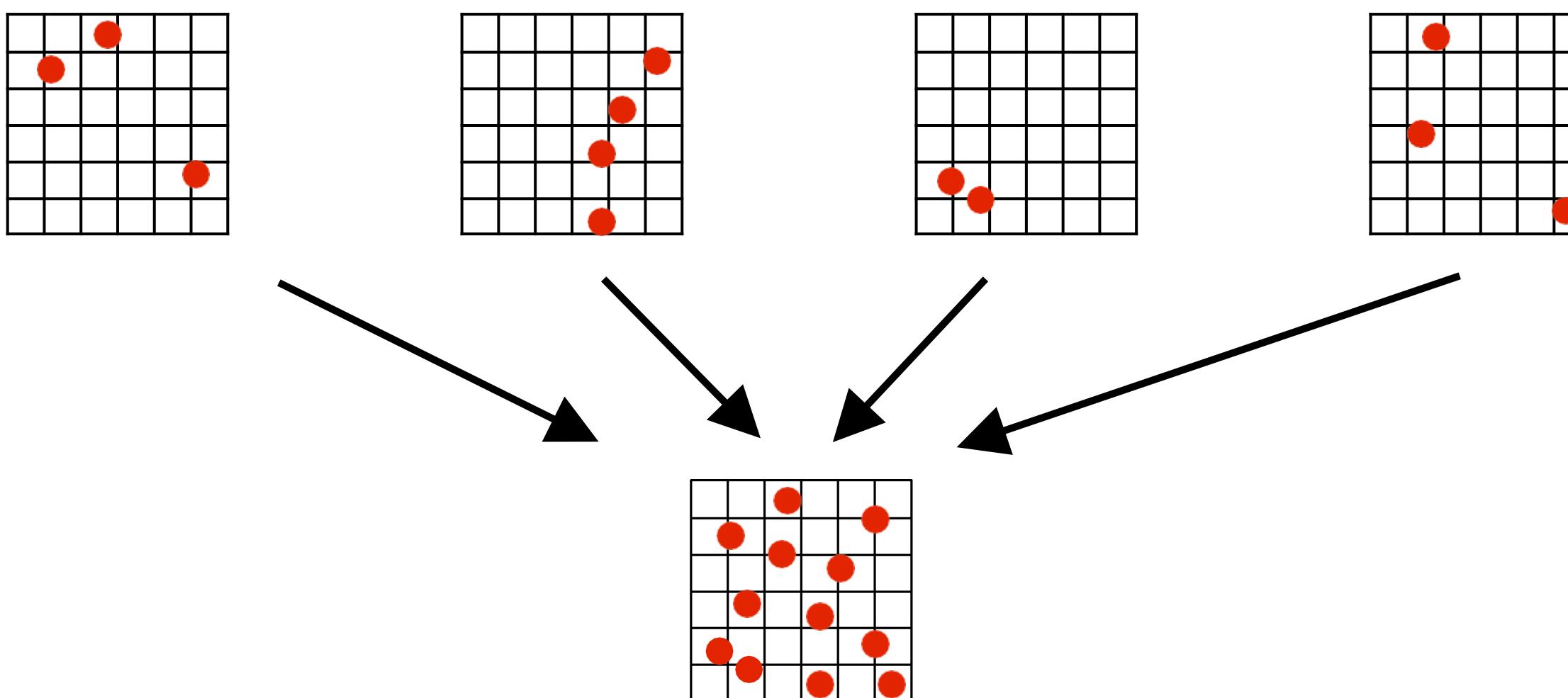
- Alleviate contention for single global lock by using per-cell locks
 - Assuming uniform distribution of particles in 2D space... ~16x less contention than solution 2

```
list cell_list[16];    // 2D array of lists
lock cell_list_lock[16];

for each particle p          // in parallel
    c = compute cell containing p
    lock(cell_list_lock[c])
        append p to cell_list[c]
    unlock(cell_list_lock[c])
```

Solution 4: compute partial results + merge

- Yet another answer: generate M “partial” grids in parallel, then combine
 - Example: create M thread blocks (at least as many thread blocks as $SMXcores$)
 - Each block assigned N/M particles
 - All threads in thread block update same grid
 - Enables faster synchronization: contention reduced by factor of M and also cost of synchronization is lower because it is performed on block-local variables (in CUDA shared memory)
 - Requires extra work: merging the M grids at the end of the computation
 - Requires extra memory footprint: Store M grids of lists, rather than 1



Solution 5: data-parallel approach

Step 1: compute cell containing each particle (parallel over input particles)

particle_index: 0 1 2 3 4 5

grid_index: 9 6 6 4 6 4

0	1	2	3
3 4 5	5	1 6 2 4	7
8	9 0	10	11
12	13	14	15

Step 2: sort results by cell (particle index array permuted based on sort)

particle_index: 3 5 1 2 4 0

grid_index: 4 4 6 6 6 9

Step 3: find start/end of each cell (parallel over particle_index elements)

```
cell = grid_index[index]
if (index == 0)
    cell_starts[cell] = index;
else if (cell != grid_index[index-1]) {
    cell_starts[cell] = index;
    cell_ends[grid_index[index-1]] = index;
}
if (index == numParticles-1) // special case for last cell
    cell_ends[cell] = index+1;
```

This solution maintains a large amount of parallelism and removes the need for fine-grained synchronization... at cost of a sort and extra passes over the data (extra BW)

This code is run for each element of particle_index array (each instance has a unique value of 'index')

cell_starts

				0		2			5		...
--	--	--	--	---	--	---	--	--	---	--	-----

cell_ends
(not inclusive)

0	1			2		5			6		...
---	---	--	--	---	--	---	--	--	---	--	-----

Parallel implementations of particle binning

Sequential algorithm

```
list cell_lists[16]; // 2D array of lists

for each particle p
    c = compute cell containing p
    append p to cell_lists[c]
```

Implementation 1: Parallel over particles

```
list cell_list[16]; // 2D array of lists
lock cell_list_lock;

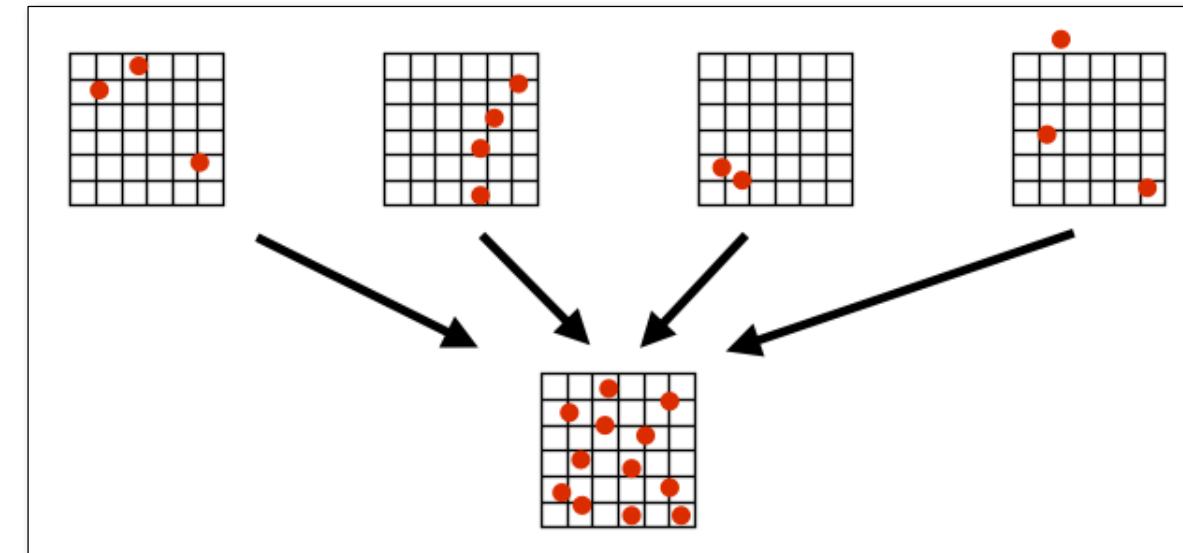
for each particle p // in parallel
    c = compute cell containing p
    lock(cell_list_lock)
    append p to cell_list[c]
    unlock(cell_list_lock)
```

Implementation 2: Parallel overgrid cells

```
list cell_lists[16]; // 2D array of lists

for each cell c // in parallel
    for each particle p // sequentially
        if (p is within c)
            append p to cell_lists[c]
```

Implementation 3: build separate grids and merge



Implementation 4: data-parallel sort

Step 1: compute cell containing each particle

Array Index:	0	1	2	3	4	5
result:	9	6	6	4	6	4

Step 2: sort results by cell

Array Index:	3	5	1	2	4	0
result:	4	4	6	6	6	9

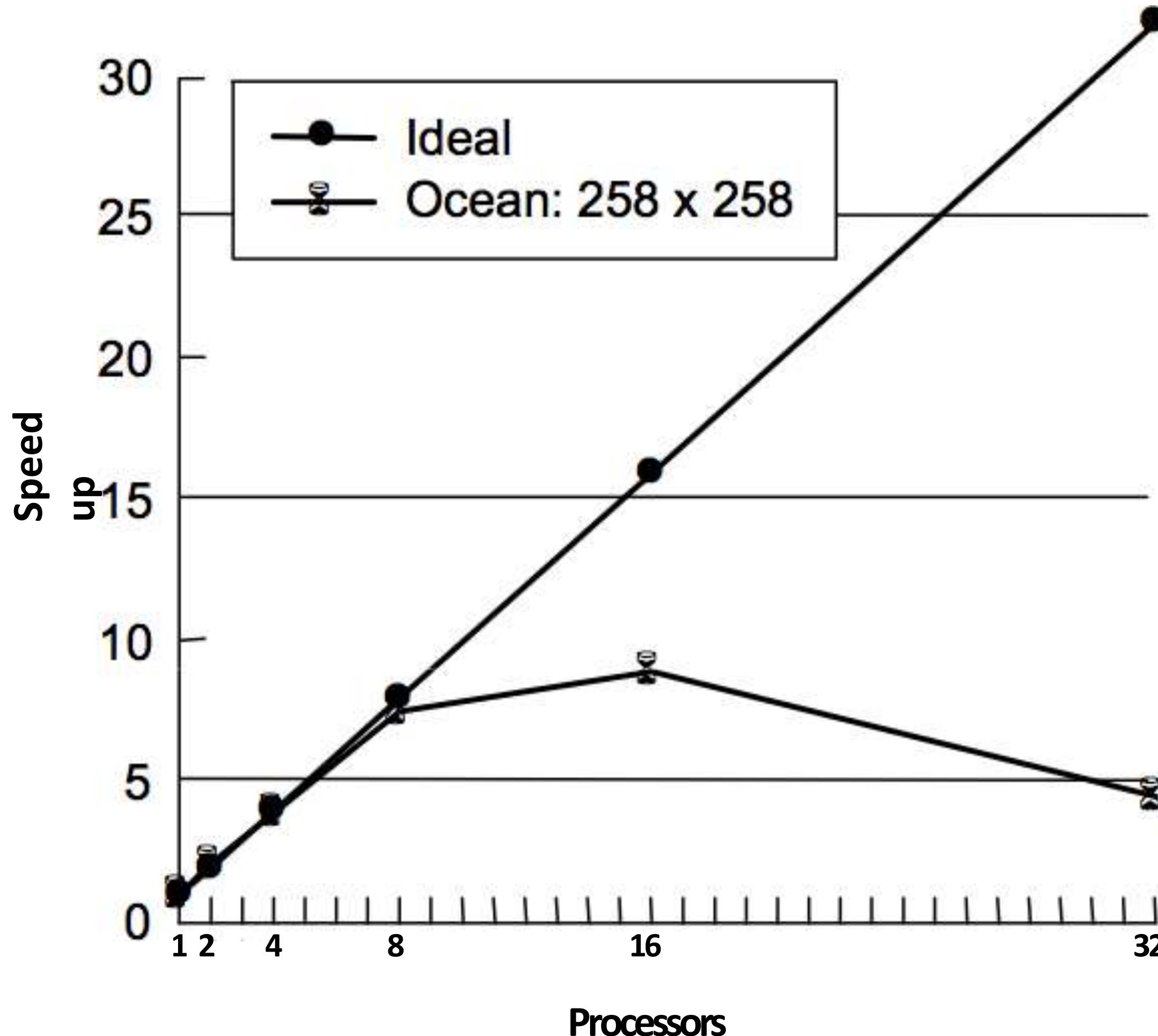
Step 3: find start/end of each cell

```
cell = result[index]
if (index == 0 || cell != result[index-1]) {
    cell_starts[cell] = index;
    if (index > 0) // special case for first cell
        cell_ends[result[index-1]] = index;
}
if (index == numParticles-1) // special case for last cell
    cell_ends[cell] = index+1;
```

Common pitfall: compare parallel program speedup to parallel algorithm running on one core (easier to make yourself look good)

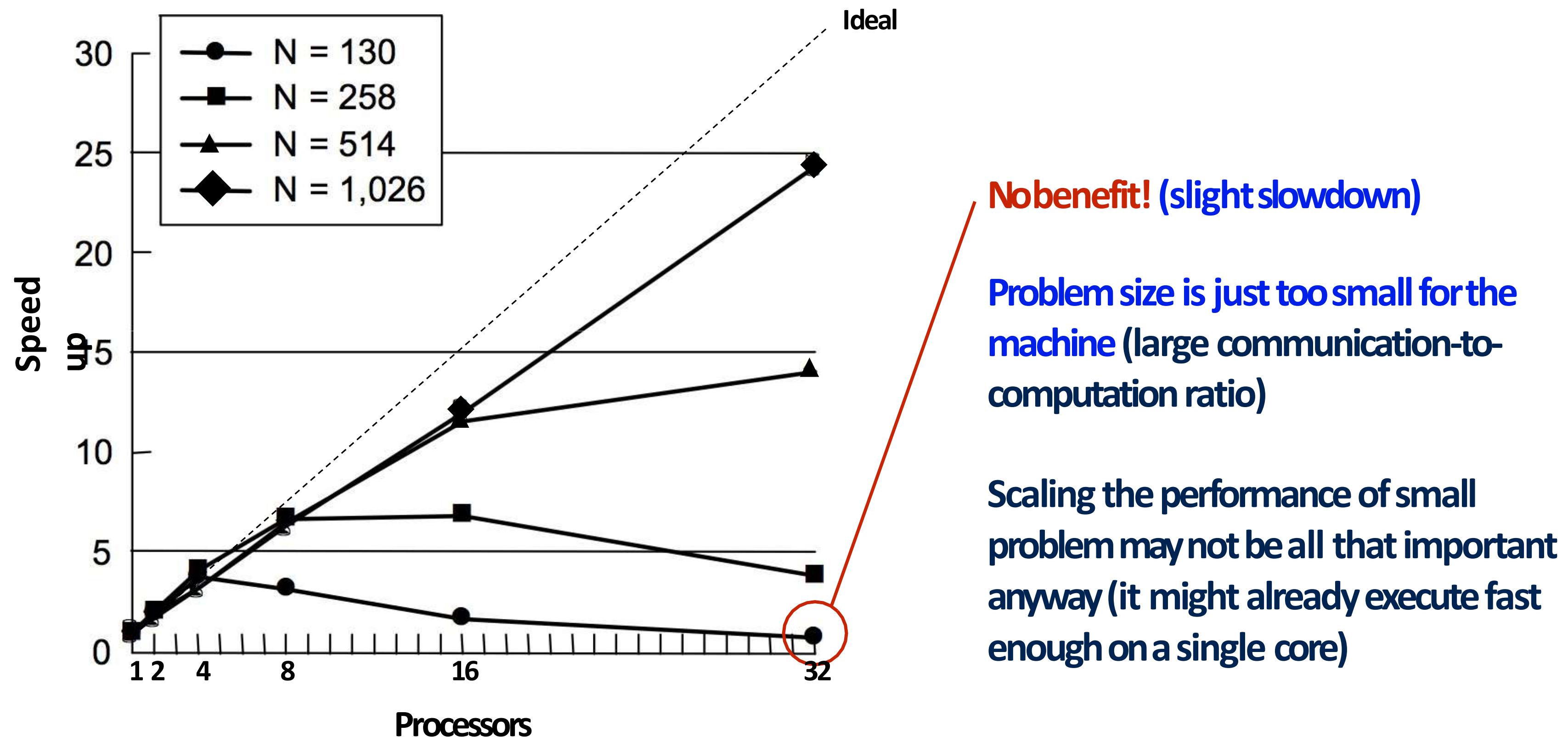
Speedup of ocean sim application: 258 x 258 grid

Execution on 32 processor SGI Origin 2000



Pitfalls of fixed problem size speedup analysis

Ocean execution on 32 processor SGI Origin 2000



258 x 258 grid on 32 processors:

~ 310 grid cells per processor

1K x 1K grid on 32 processors:

~ 32K grid cells per processor

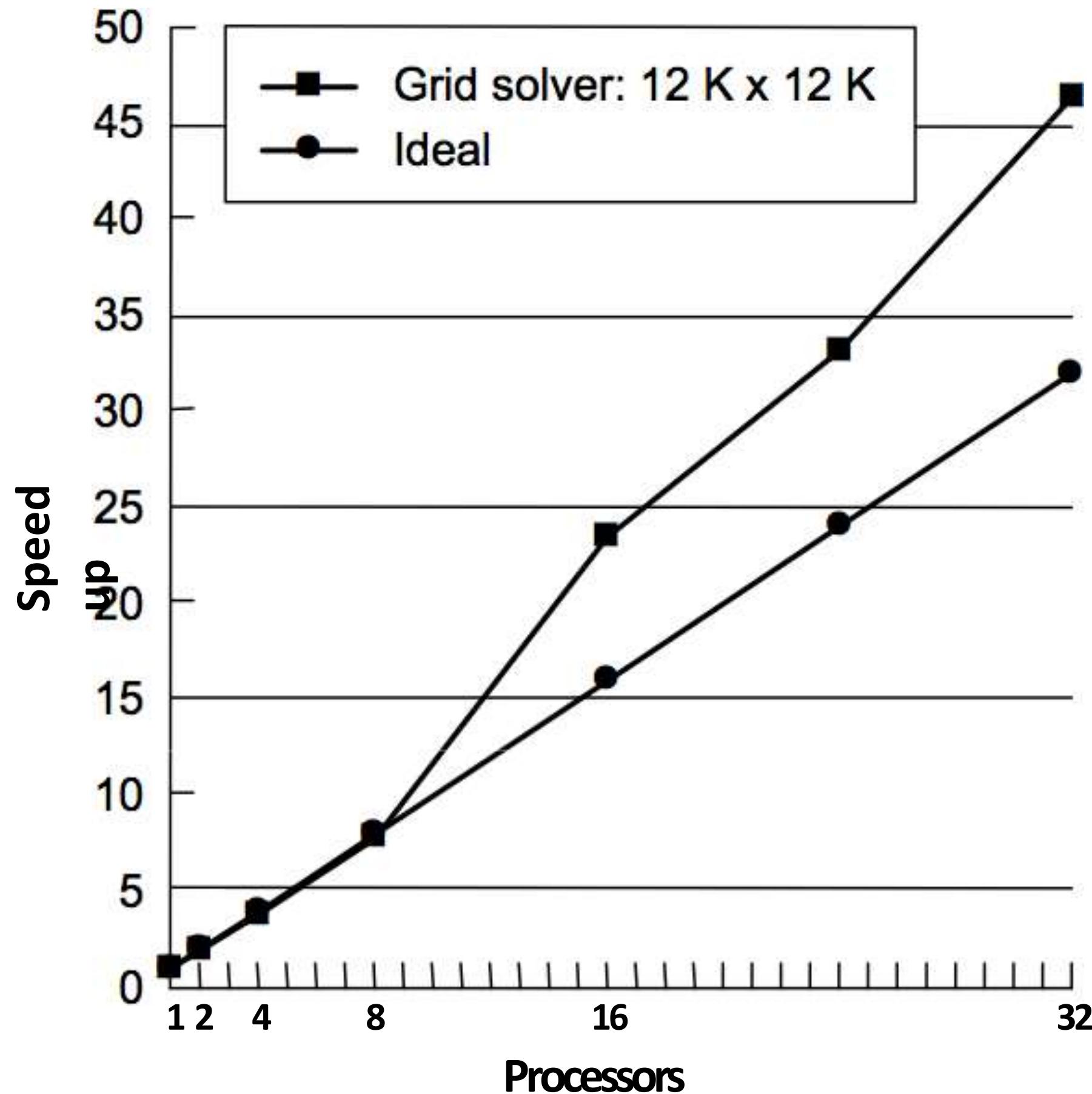
No benefit! (slight slowdown)

Problem size is just too small for the machine (large communication-to-computation ratio)

Scaling the performance of small problem may not be all that important anyway (it might already execute fast enough on a single core)

Pitfalls of fixed problem size speedup analysis

Execution on 32 processor SGI Origin 2000



Here: **super-linear speedup!** with enough processors, chunk of grid assigned to each processor **begins to fit in cache** (key working set fits in per-processor cache)

Another example: if problem size is too large for a single machine, working set may not fit in memory: causing thrashing to disk

(this would make speedup on a bigger parallel machine with more memory look amazing!)

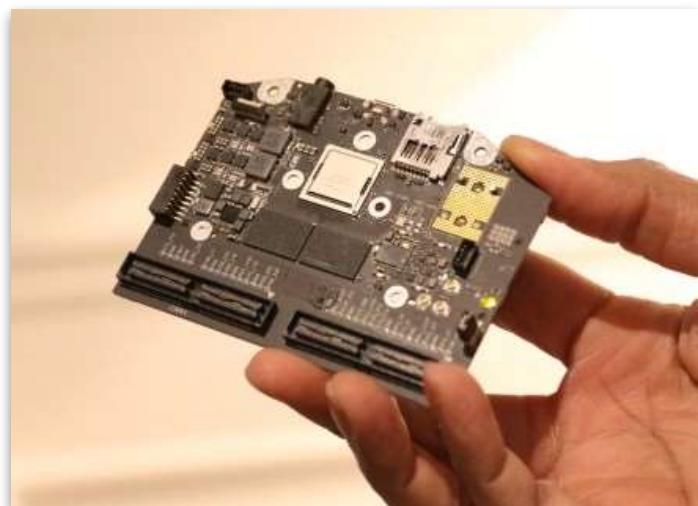
Understanding scaling: size matters!

- There can be complex interactions between the size of the problem and the size of the parallel computer.
 - Can impact load balance, overhead, arithmetic intensity, locality of data access
 - Effects can be dramatic and application dependent
- Evaluating a machine with a fixed problem size can be problematic
 - Too small a problem:
 - Parallelism overheads dominate parallelism benefits (may even result in slow downs)
- Can be desirable to scale problem size as machine sizes grow
(buy a bigger machine to compute more, rather than just compute the same problem faster)
 - Problem size may be appropriate for small machines, but inappropriate for large ones
(does not reflect realistic usage of large machine!)
 - Too large a problem: (problem size chosen to be appropriate for large machine)
 - Key working set may not “fit” in small machine
(causing thrashing to disk, or key working set exceeds cache capacity, or can’t run at all)
 - When problem working set “fits” in a large machine but not small one, super-linear speedups can occur

Architects also think about scaling

A common question: “Does an architecture scale?”

- **Scaling up:** how does architecture’s performance scale with increasing core count?
 - Will design scale to the high end?
- **Scaling down:** how does architecture’s performance scale with decreasing core count?
 - Will design scale to the low end?
- Parallel architectures are designed to work in a range of contexts
 - Same architecture used for low-end, medium-scale, and high-end systems
 - GPUs are a great example
 - Same SIMD core architecture, different numbers of SIMD cores per chip



Tegra X1: 2 SIMD cores
(mobile SoC)



GTX950: 6 SIMDcores
(90 watts)



GTX980: 16 SIMDcores
(165 watts)



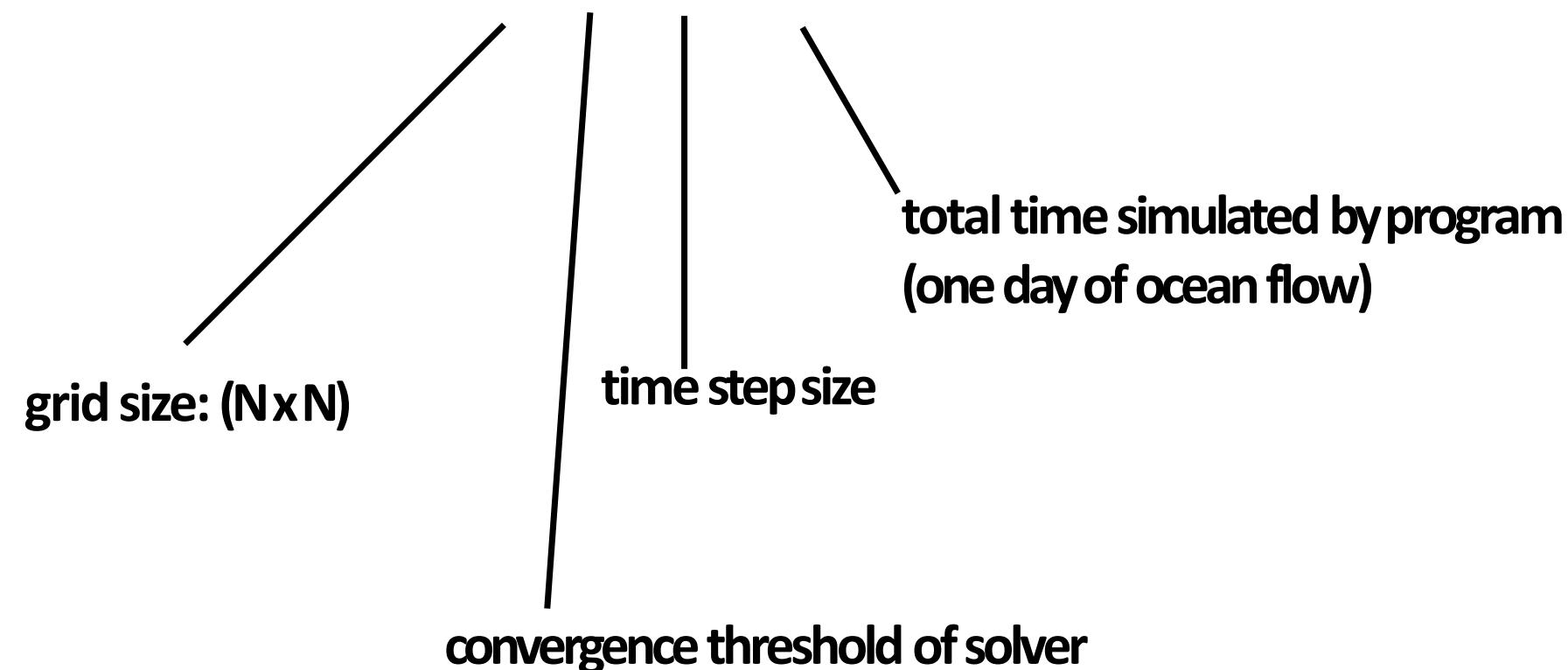
Titan X: 24 SIMDcores
(250 watts)

Common Scaling Terminology

- For mapping problem X on system with P processors
- Strong scaling
 - Runtime of X on P processors, vs. of X on 1 processor
 - Goal = P
 - Does having more processors get job done faster?
- Weak scaling
 - Runtime of (P^*X) on P processors, vs. of X on 1 processor
 - Goal = 1.0
 - Does having more processors let me do bigger jobs?

Questions to ask when scaling a problem

- **Under what constraints should the problem be scaled?**
 - “Work done by program” may no longer be the quantity that is fixed
 - Fixed **data set size**, fixed **memory usage per processor**, fixed **execution time**, etc.?
- **How should be the problem be scaled?**
 - Problem size is often determined by more than one parameter
 - Ocean example: problem defined by $(N, \epsilon, \Delta t, T)$



Scaling constraints

- **Application-oriented scaling properties (specific to application)**
 - Particles per processor
 - Transactions per processor
- **Resource-oriented scaling properties**
 1. **Problem constrained scaling (PC)**
 2. **Memory constrained scaling (MC)**
 3. **Time constrained scaling (TC)**

User-oriented properties often more intuitive, but resource-oriented properties are more general, and apply across domains.
(so we'll talk about them today)

Problem-constrained scaling

- Focus: use a parallel computer to solve **the same problem faster**

Speedup =

$$\frac{\text{time 1 processor}}{\text{time P processors}}$$

- Recall pitfalls from earlier in lecture (small problems may not be realistic workloads for large machines, big problems may not fit on small machines)
- Examples of problem-constrained scaling:
 - Almost everything we've considered parallelizing in class so far

Time-constrained scaling

- Focus: completing more work in a fixed amount of time
 - Execution time kept fixed as the machine (and problem) scales

$$\text{Speedup} = \frac{\text{work done by } P \text{ processors}}{\text{work done by 1 processor}}$$

- How to measure “work”?
 - Challenge: “work done” may not be linear function of values of problem inputs (e.g. matrix multiplication is $O(N^3)$ work for $O(N^2)$ sized inputs)
 - One approach: “work done” is defined by execution time of same computation on a single processor (but consider effects of thrashing if problem too big)
 - Ideally, a measure of work is:
 - Simple to understand
 - Scales linearly with sequential run time (So ideal speedup remains linear in P)

More time-constrained scaling examples

- **Computational finance**
 - Run most sophisticated model possible in: 1 ms, 1 minute, overnight, etc.
- **Modern websites**
 - Want to generate complex page, respond to user in X milliseconds
(studies show site usage directly corresponds to page load latency)
- **Real-time computer vision for robotics**
 - Consider self-driving car: best quality pedestrian detection in 10 ms

Memory-constrained scaling

- Focus: **run the largest problem possible without overflowing main memory** **
- Memory per processor is held fixed (e.g., add more machines to cluster)
- *Neither work nor execution time are held constant!*

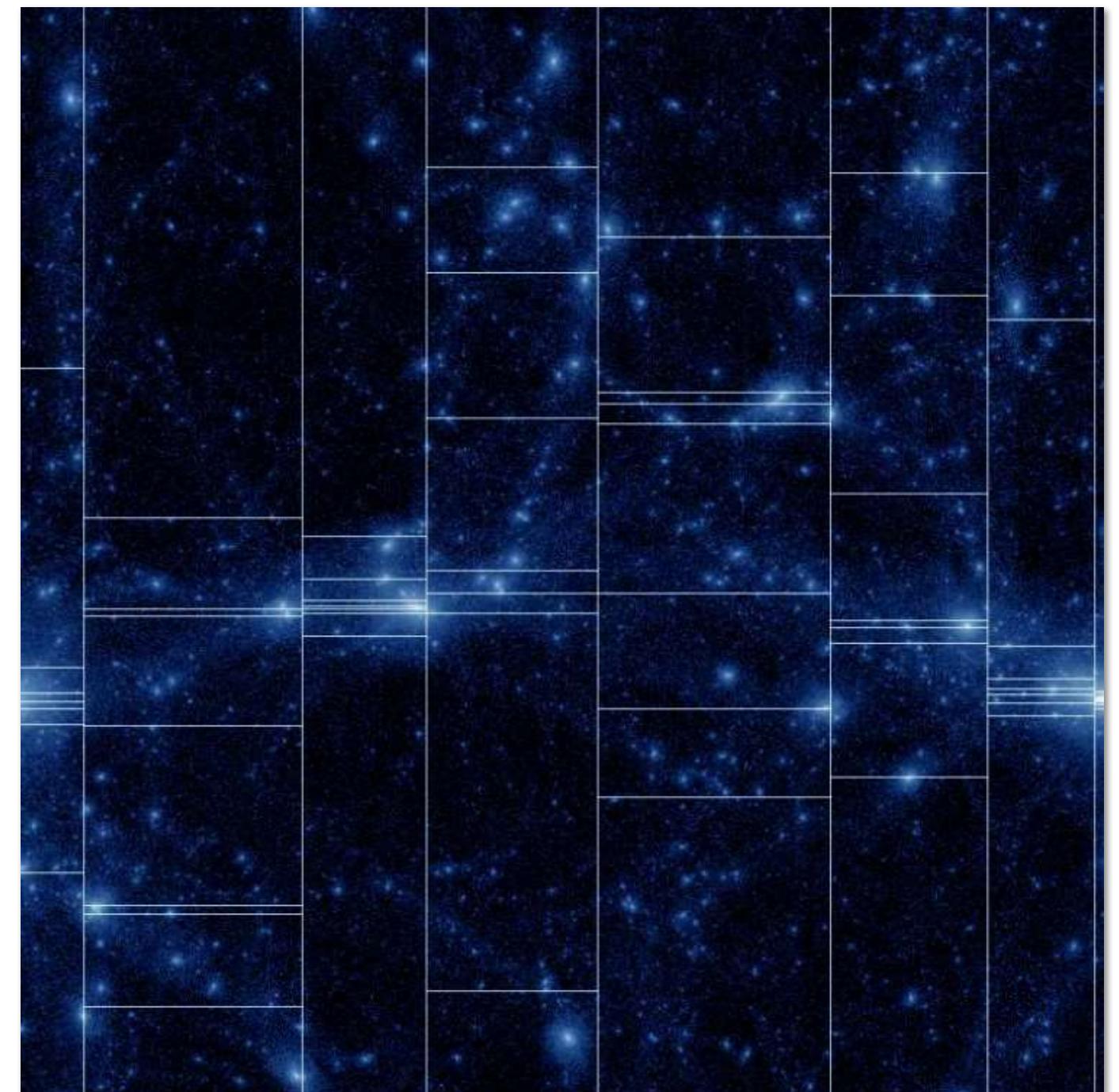
$$\text{Speedup} = \frac{\text{work (P processors)} \times \text{time (1 processor)}}{\text{time (P processors)} \times \text{work (1 processor)}}$$
$$= \frac{\text{work per unit time on P processors}}{\text{work per unit time on 1 processor}}$$

- Note: scaling problem size can make runtimes very large
 - Consider $O(N^3)$ matrix multiplication on $O(N^2)$ matrices

** Assumptions: (1) memory resources scale with processor count (2) spilling to disk is infeasible behavior (too slow)

Memory-constrained scaling examples

- One motivation to use supercomputers and large clusters is simply to be able to fit large problems in memory
- Large N-body problems
 - 2012 Supercomputing Gordon Bell Prize Winner: 1,073,741,824,000 particle N-body simulation on K-Computer (MPI implementation)
- Large-scale machine learning
 - Billions of clicks, documents, etc.
- Memcached (in memory caching system for webapps)
 - More servers = more available cache



2D domain decomposition of N-body simulation

Scaling examples at PIXAR

- **Rendering a “shot” (a sequence of frames) in a movie**
 - Goal: minimize time to completion (**problem constrained**)
 - Assign each frame to a different machine in the cluster
- **Artists working to design lighting for a scene**
 - Provide interactive frame rate to artist (**time constrained**)
 - More performance = higher fidelity representation shown to artist in allotted time
- **Physical simulation: e.g., fluid simulation**
 - Parallelize simulation across multiple machines to fit simulation grid in aggregate memory of processors (**memory constrained**)
- **Final render of images for movie**
 - Scene complexity is typically bounded by memory available on farm machines
 - One barrier to exploiting additional parallelism within a machine is that required footprint often increases with number of processors

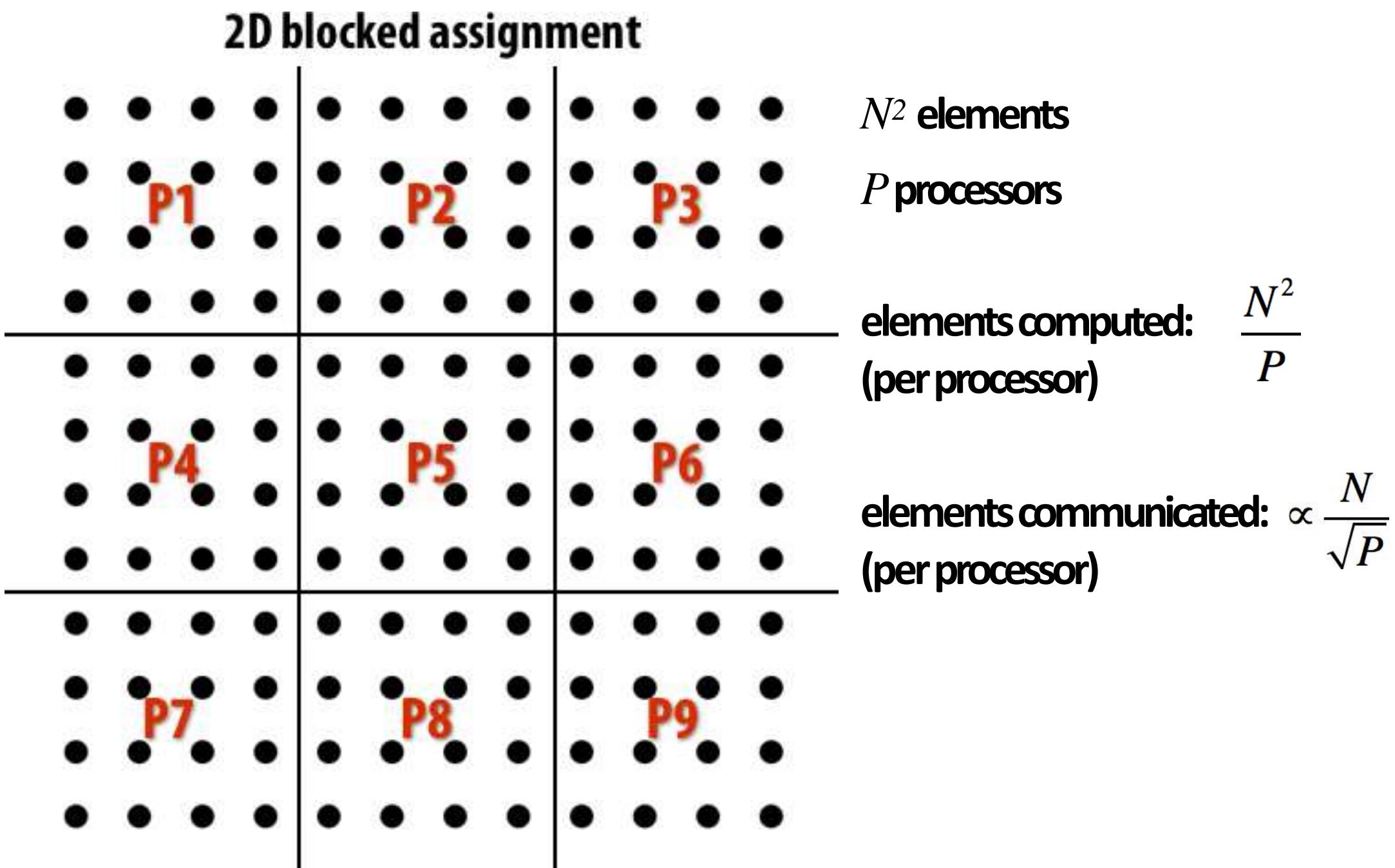


Case study: our 2Dgrid solver

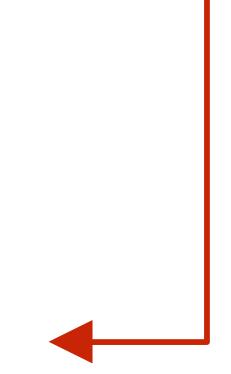
- For $N \times N_{\text{grid}}$:
 - $O(N^2)$ memory requirement
 - $O(N^2)$ grid elements $\times O(N)$ convergence iterations... so **total work increases as $O(N^3)$**

- **Problem-constrained scaling:**

- Execution time: $1/P$
- Elements per processor: N^2/P
- Available concurrency: fixed at P
- **Comm-to-comp ratio: $O(P^{1/2})$**
(for a 2D-blocked assignment)



Scaling the 2Dgrid solver

- For $N_x \ N_{\text{grid}}$:
 - $O(N^2)$ memory requirement
 - $O(N^2)$ grid elements \times $O(N)$ convergence iterations... so total work increases as $O(N^3)$
 - Problem-constrained scaling: (note: N is constant here)
 - Execution time: $O(1/P)$
 - Elements per processor: $O(1/P)$
 - Communication per processor: $O(1/P^{1/2})$
 - Comm-to-comp ratio: $O(P^{1/2})$
(for a 2D-blocked assignment)
 - Time-constrained scaling:
 - Let scaled grid size be $K \times K$
 - Assume linear speedup: $K^3/P = N^3$ (so $K = NP^{1/3}$)
(recall computation time for $K \times K$ grid on P processors = computation time for $N \times N$ grid on 1 processor)
 - Execution time: fixed at $O(N^3)$ (by defn of TCscaling)
 - Elements per processor: $K^2/P = N^2/P^{1/3}$
 - Communication per processor: $K/P^{1/2} = O(1/P^{1/6})$
 - Comm-to-comp ratio: $O(P^{1/6})$
 - Memory-constrained scaling:
 - Let scaled grid size be $NP^{1/2} \times NP^{1/2}$
 - Execution time: $O((NP^{1/2})^3/P) = O(P^{1/2})$
 - Elements per processor: fixed! (N^2)
 - Comm-to-comp ratio: $O(1)$
- notice: execution time increases with MCscaling
- 
- Implications:
Expect best “speedup” with MCscaling, then TCscaling, worst with PCscaling.
(due to communication overheads)

Word of caution about problem scaling

- Problem size in the previous examples was a single parameter n
- In practice, problem size is a combination of parameters
 - Recall Ocean example: problem is $= (n, \epsilon, \Delta t, T)$
- Problem parameters are often related (not independent)
 - Example from Barnes-Hut: increasing particle count n changes required simulation time step and force calculation accuracy parameter Θ
- Must be cognizant of these relationships under situations of TCor MC scaling

Scaling summary

- Performance improvement due to parallelism is measured by speedup
- But speedup metrics take different forms for different scaling models
 - Which model matters most is application/context specific
- In addition to assignment and orchestration, behavior of a parallel program depends significantly on the scaling properties of the problem and also the machine.
 - When analyzing performance, be careful to analyze realistic regimes of operation (realistic sizes and realistic problem size parameters)
 - This requires application knowledge

The challenges of scaling problems up or down also apply to software developers debugging/tuning parallel programs on real machines

Common examples:

- May want to log behavior of code when debugging
 - Debug logs get untenable in size when running full problem
 - Instrumentation slows down code significantly
 - Instrumentation removes contention by changing timing of application
- May want to debug/test/tune code on small problem size on small machine before running two-week simulation on full problem size on a supercomputer

There is simply no substitute for the experience of writing and tuning your own parallel programs.
But today's take-away is: BE CAREFUL!

It can be very tricky to evaluate and tune parallel software and parallel machines.

It can be very easy to obtain misleading results or tune code (or a billion dollar hardware design) for a workload that is not representative of real-world use cases.

It is helpful to start by precisely stating your application performance goals. Then determine if your evaluation approach is consistent with these goals.

**Here are some tricks for understanding the
performance of parallel software**

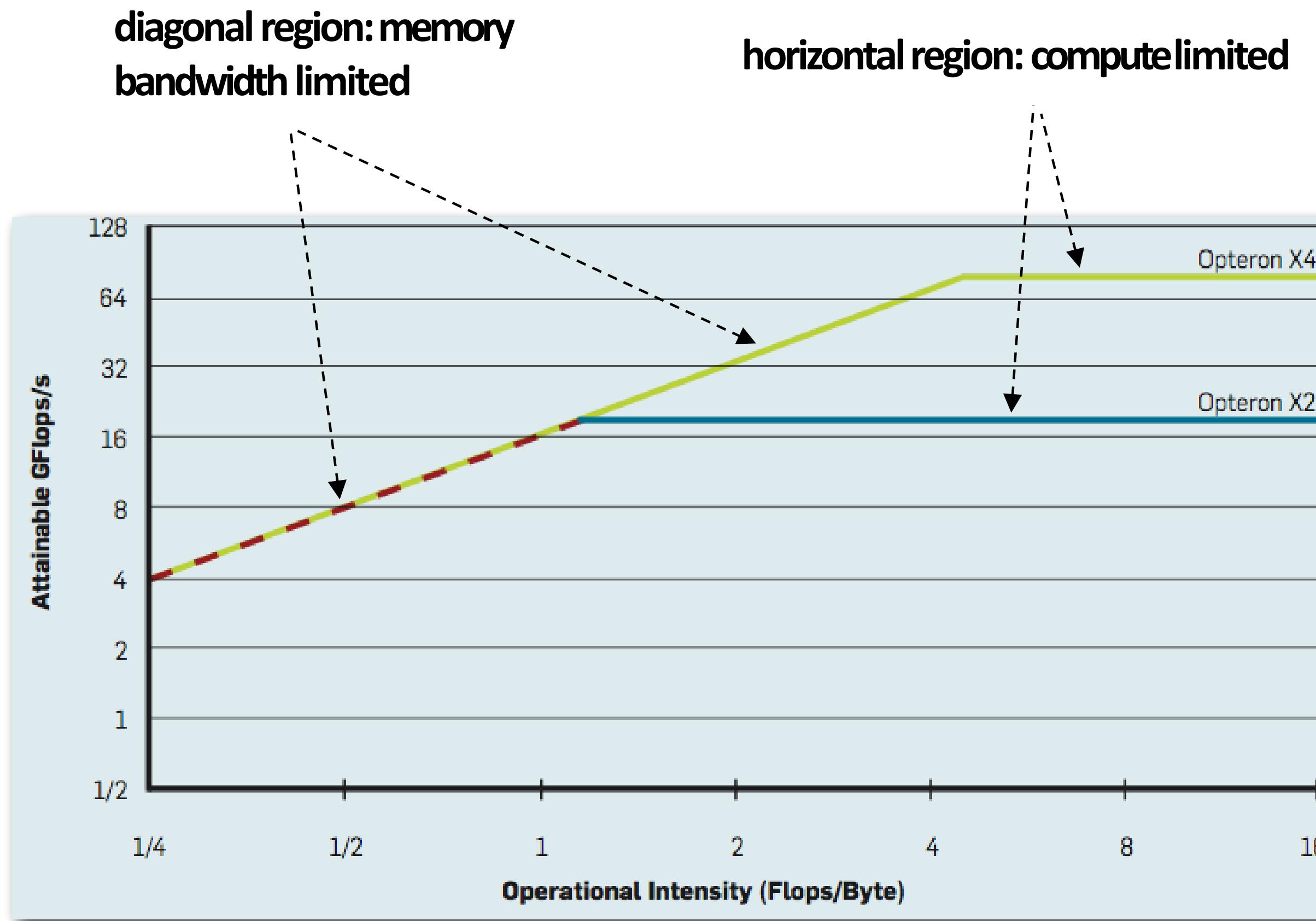
**Always, always, always try the simplest
parallel solution first, then measure
performance to see where you stand.**

A useful performance analysis strategy

- Determine if your performance is **limited by computation, memory bandwidth (or memory latency), or synchronization?**
- Try and establish “**high watermarks**”
 - What's the best you can do in practice?
 - How close is your implementation to a best-case scenario?

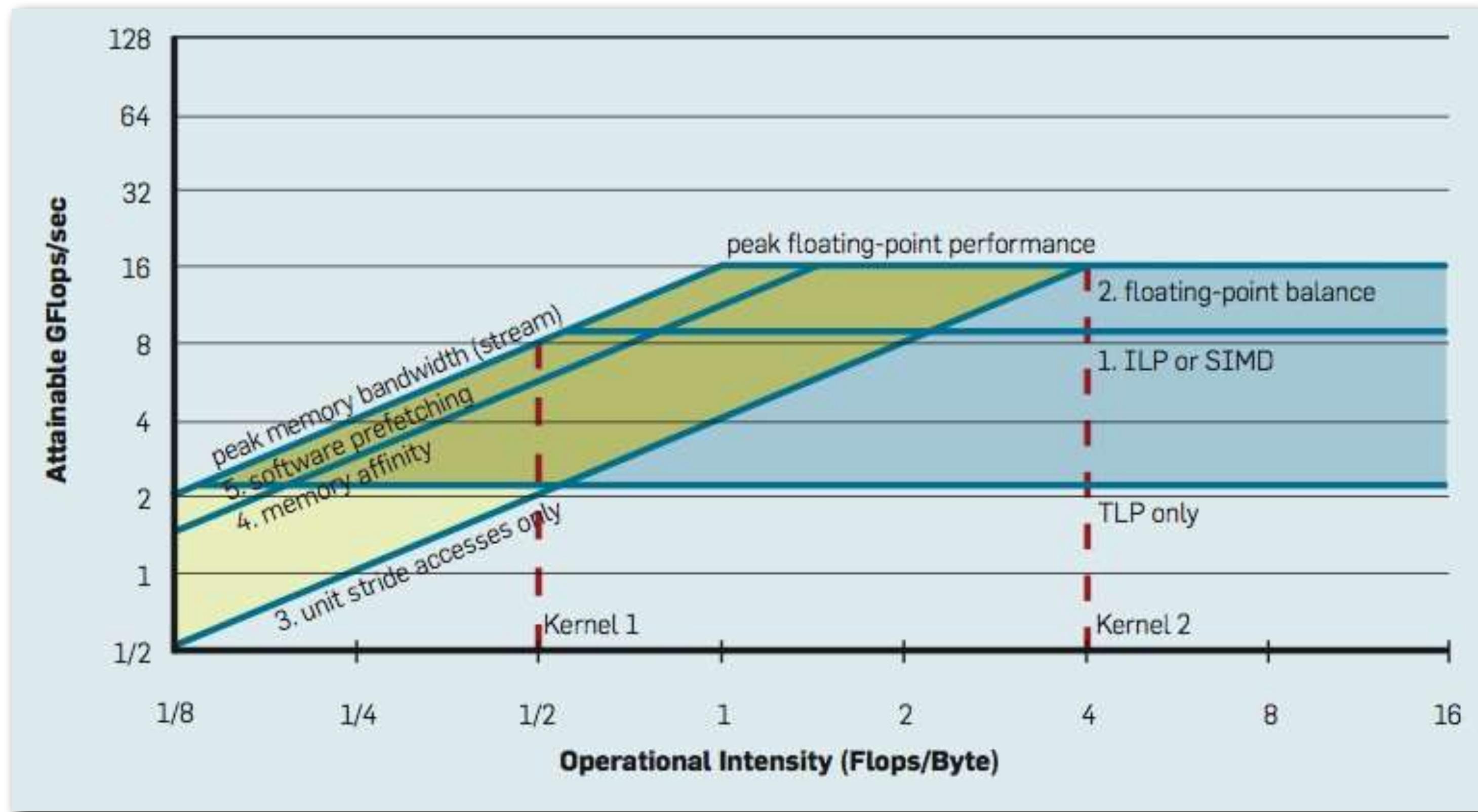
Roofline model

- Use microbenchmarks to compute peak performance of a machine as a function of arithmetic intensity of application
- Then compare application's performance to known peak values



Roofline model: optimization regions

- Use various levels of optimization in benchmarks
(e.g., best performance with and without using SIMD instructions)



Establishing high watermarks *

Add “math” (non-memory instructions)

Does execution time increase linearly with operation count as math is added?

(If so, this is evidence that code is instruction-rate limited)

Remove almost all math, but load same data

How much does execution-time decrease? If not much, suspect memory bottleneck

Change all array accesses to A[0]

How much faster does your code get?

(This establishes an upper bound on benefit of improving locality of data access)

Remove all atomic operations or locks

How much faster does your code get? (provided it still does approximately the same amount of work)

(This establishes an upper bound on benefit of reducing sync overhead.)

* Computation, memory access, and synchronization are almost never perfectly overlapped. As a result, overall performance will rarely be dictated entirely by compute or by bandwidth or by sync. Even so, the sensitivity of performance change to the above program modifications can be a good indication of dominant costs

Use profilers/performance monitoring tools

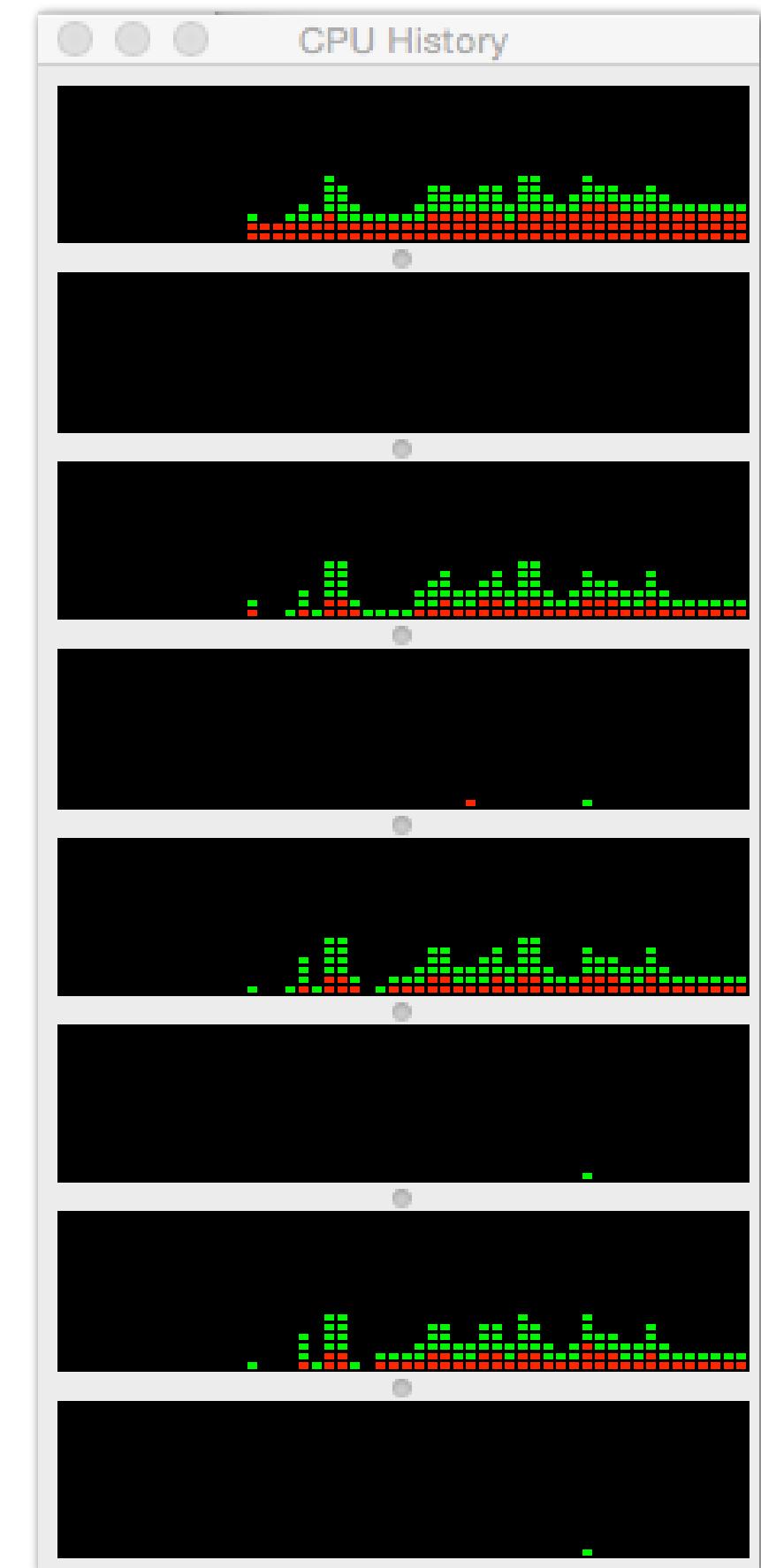
- Image at left is “CPU usage” from activity monitor in OSX while browsing the web in Chrome (laptop has a quad-core Core i7 CPU)
 - Graph plots percentage of time OS has scheduled a process thread onto a processor execution context
 - Not very helpful for optimizing performance
- All modern processors have low-level event “**performance counters**”
 - Registers that count important details such as: instructions completed, clock ticks, L2/L3 cache hits/misses, bytes read from memory controller, etc.
- Example: Intel’s Performance Counter Monitor Tool provides a C++ API for accessing these registers.

```
PCM *m = PCM::getInstance();
SystemCounterState begin = getSystemCounterState();

// code to analyze goes here

SystemCounterState end = getSystemCounterState();

printf("Instructions per clock: %f\n", getIPC(begin, end));
printf("L3 cache hit ratio: %f\n", getL3CacheHitRatio(begin, end));
printf("Bytes read: %d\n", getBytesReadFromMC(begin, end));
```



- Also see Intel VTune, PAPI, oprofile, etc.

Questions