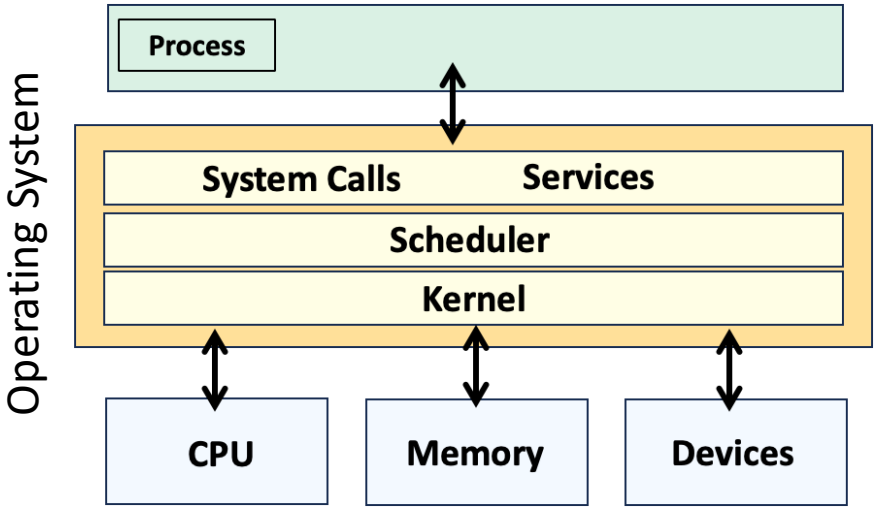
An Introduction to computers: Silicon, parallelism, and scaling to absurdity

Tim Mattson

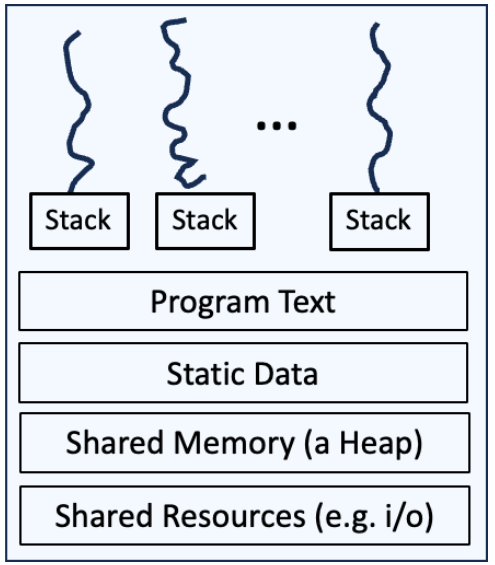
Tim in friendly surf at Sand Island Oregon, USA

Last time we covered …. key concepts in computing

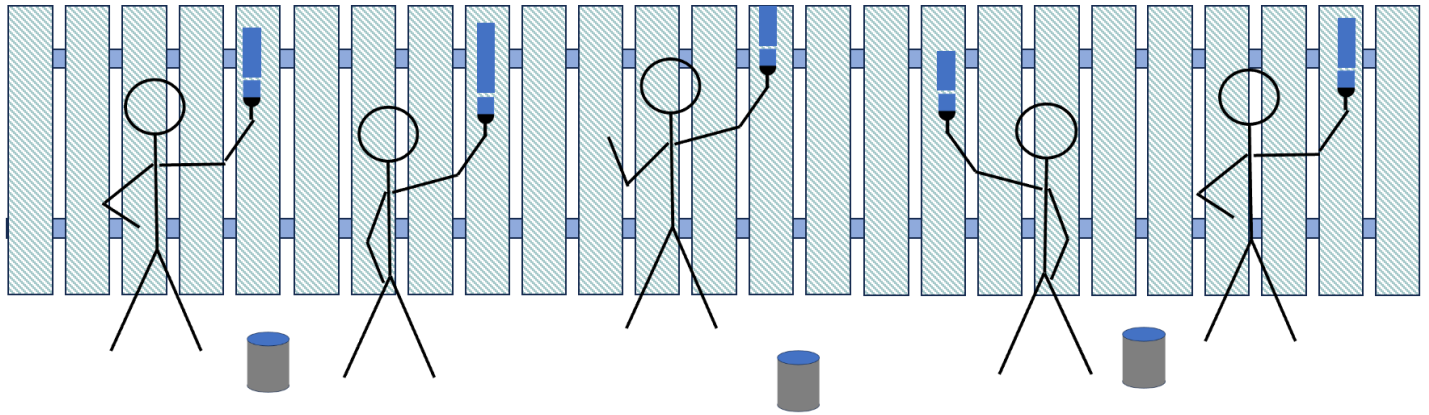


A high level look at Operating systems and processes as an instance of a running program.

The anatomy of a process with a share memory (heap) and one or more threads

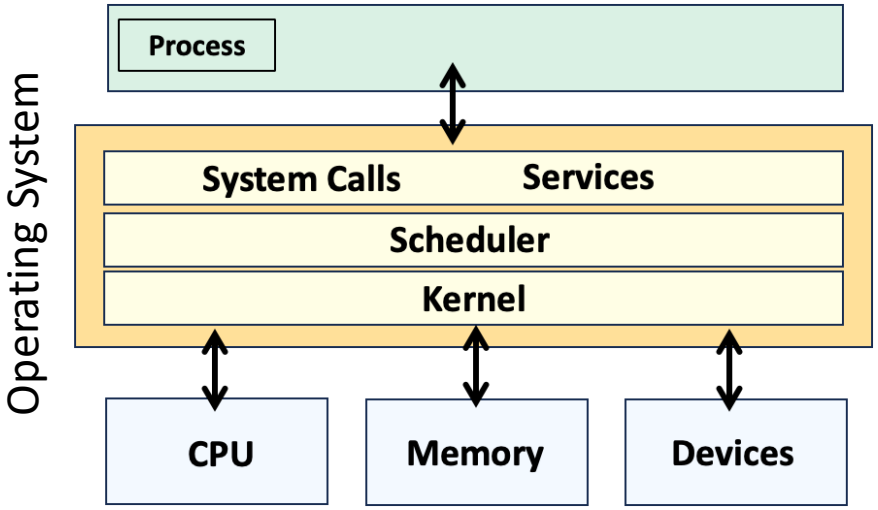


Process



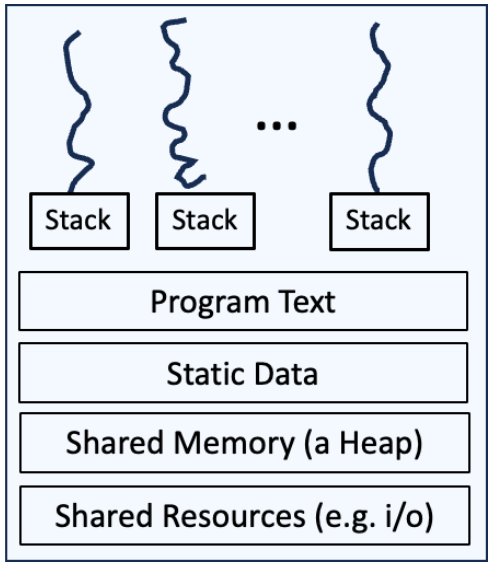
Tom Sawyer, painting fences, and the concept of parallelism

Last time we covered …. key concepts in computing

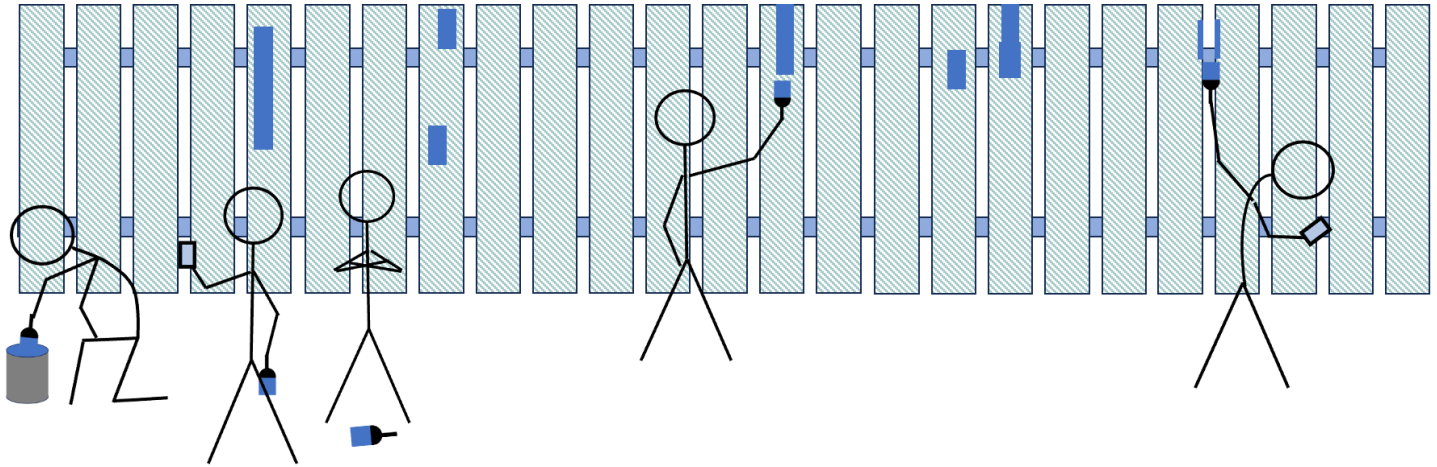


A high level look at Operating systems and processes as an instance of a running program.

The anatomy of a process with a share memory (heap) and one or more threads



Process



Tom Sawyer, painting fences, and the concept of parallelism, … **and the fact it almost never goes as smoothly as you'd like.**

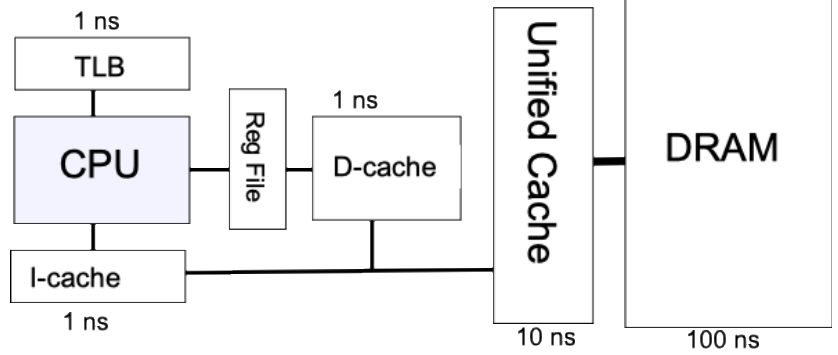
Last time we covered …. Computer Architecture

Instruction set architecture

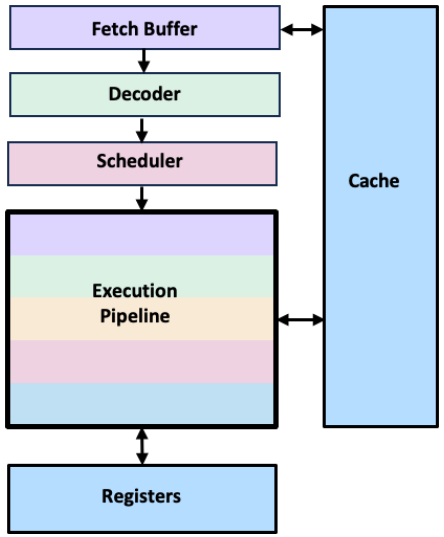
x86

The interface between software and the hardware

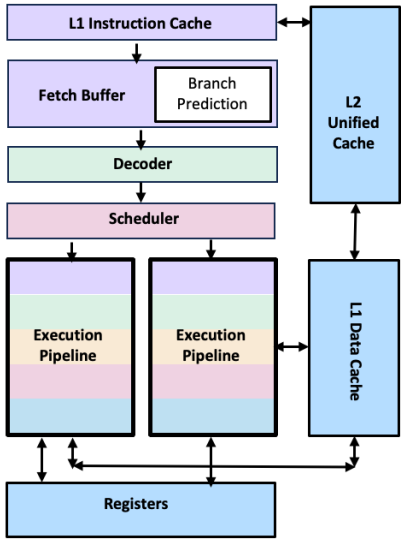
Memory Hierarchy: Latencies across the hierarchy.



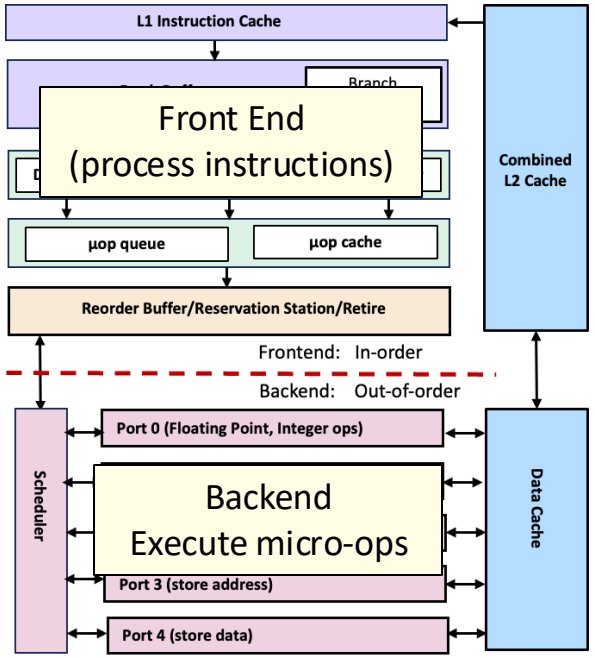
Microarchitecture: how the ISA is implemented



Pipelining

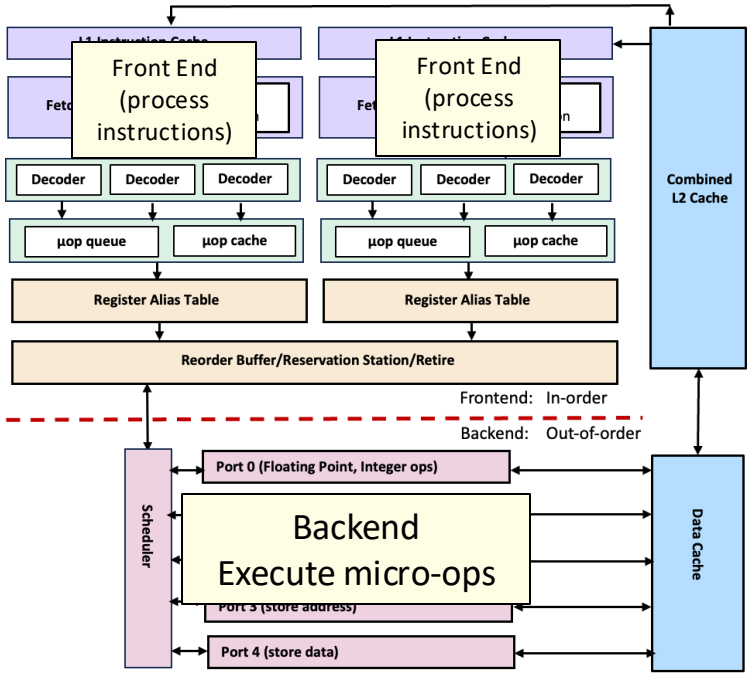


Superscalar



+ branch prediction and speculative execution

Out of order



Simultaneous multithreading

Outline … our plan for today

We will survey ALL of parallel computing in one lecture!

- Semiconductor Devices and the Need for Parallelism

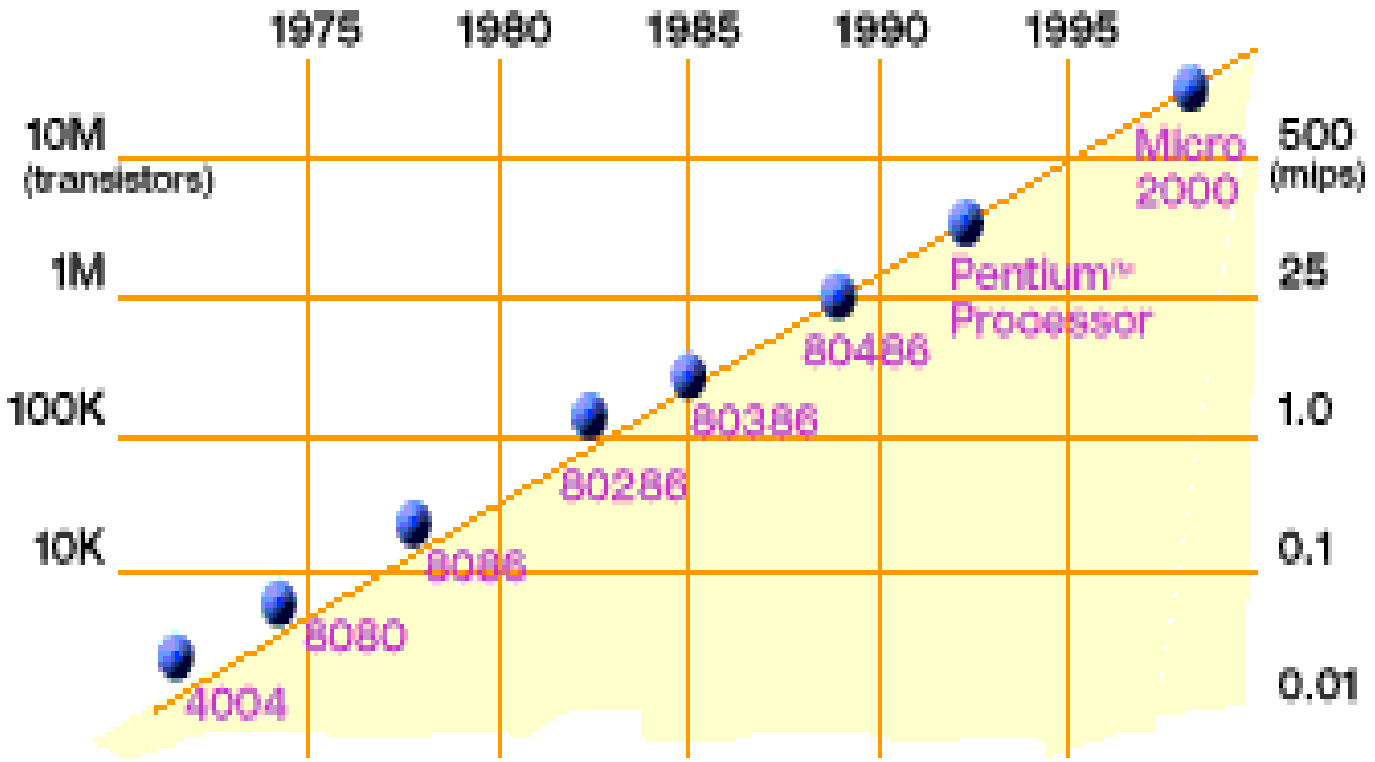
- Parallelism: Hardware, key abstractions, and programming
  - The CPU: Adventures in Multithreading
  - The Vector (SIMD) unit: Lock-step parallelism in action
  - The GPU: Data Parallelism is your friend

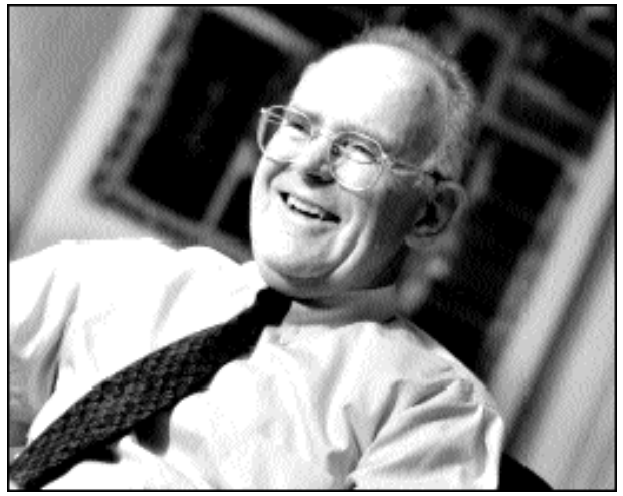
- Scaling to massive parallelism

For all Lecture Materials:

- Clone our repository: git clone github.com/tgmattso/CompSciForPhys
- Refresh each week to get latest updates: git pull

Moore's Law





- In 1965, Intel co-founder Gordon Moore predicted (from just 3 data points!) that semiconductor density would double every 18 months.
  - ***He was right!*** Over the last 50 years, transistor densities have increased as he predicted.

"Cramming more components onto integrated circuits", G.E. Moore, Electronics, 38(8), April 1965

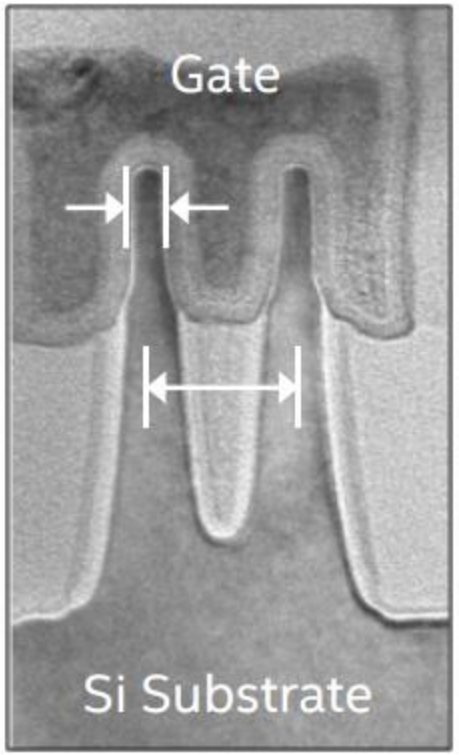
Slide source: UCB CS 194 Fall'2010

We've come a long way since Gordon Moore proposed his famous law

An electron microscope image of a single Intel transistor using the 14 nm process (2014)

8 nm Fin Width

42 nm Fin Pitch



We put billions of these transistors on a single chip

How small is a nm (nanometer)? One nm is 10^{-9} meters. Light travels about one foot in 10^{-9} seconds.

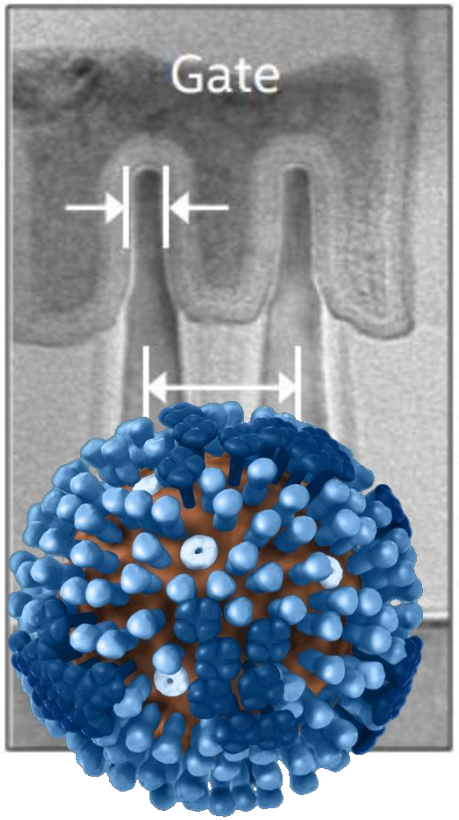
https://www-ssl.intel.com/content/dam/www/public/us/en/documents/technology-briefs/bohr-14nm-idf-2014-brief.pdf

We've come a long way since Gordon Moore proposed his famous law

An electron microscope image of a single Intel transistor using the 14 nm process (2014)

8 nm Fin Width

42 nm Fin Pitch



We put billions of these transistors on a single chip

An influenza virus is around 100 nm across!

http://www.cdc.gov/flu/images/h1n1/3D_Influenza/3D_Influenza_transparent_no_key_full_med2.gif

https://www-ssl.intel.com/content/dam/www/public/us/en/documents/technology-briefs/bohr-14nm-idf-2014-brief.pdf

How did those itty-bitty transistors impact performance?

The Linpack benchmark over time

Cray 1, 1977, first real supercomputer, 110 MFLOPS (Linpack 1000)

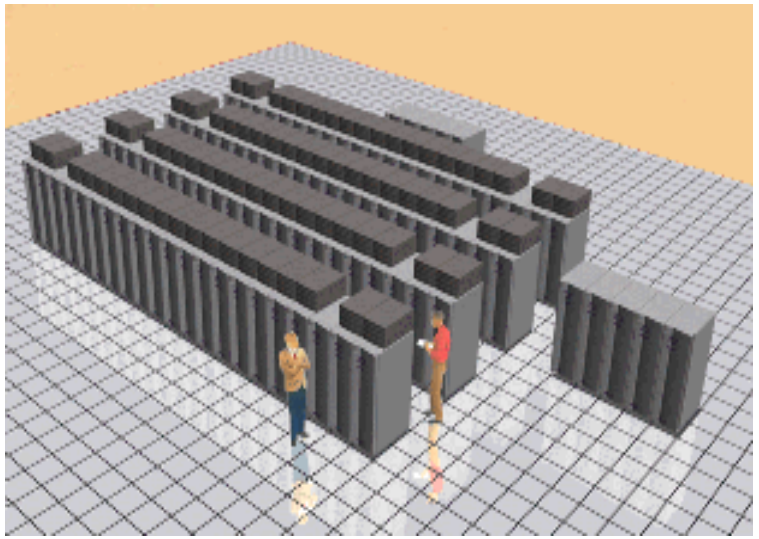
A MFLOPS is one million floating point operations per second (i.e. one million adds or multiplies per second).

VAX 11/780, ~1980, 0.14 MFLOPS, this is the computer I used in my Ph.D. research in the early 1980's.

Apple iPhone 6S, 2016, 1274 MFLOPS "in my pocket"

Moore's Law: A personal perspective

First TeraScale* computer: 1997



Intel's ASCI Option Red

Intel's ASCI Red Supercomputer

9000 CPUs

one megawatt of electricity.

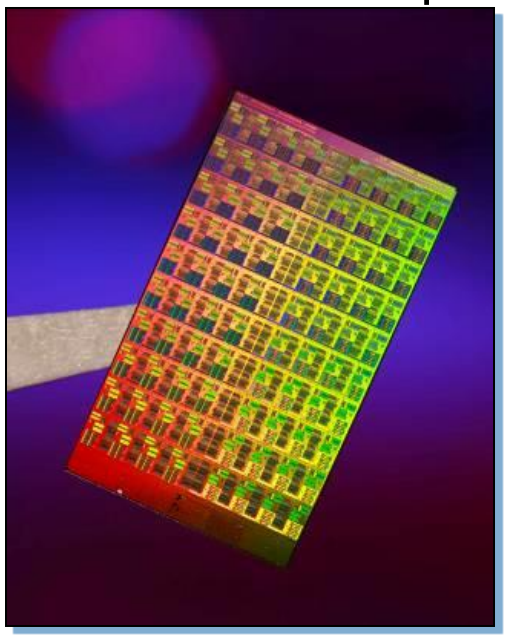
1600 square feet of floor space.

*Double Precision TFLOPS running MP-Linpack

A TeraFLOP in 1996: The ASCI TeraFLOP Supercomputer, Proceedings of the International Parallel Processing Symposium (1996), T.G. Mattson, D. Scott and S. Wheat.

10 years later

First TeraScale% chip: 2007



Intel's 80 core teraScale Chip

1 CPU

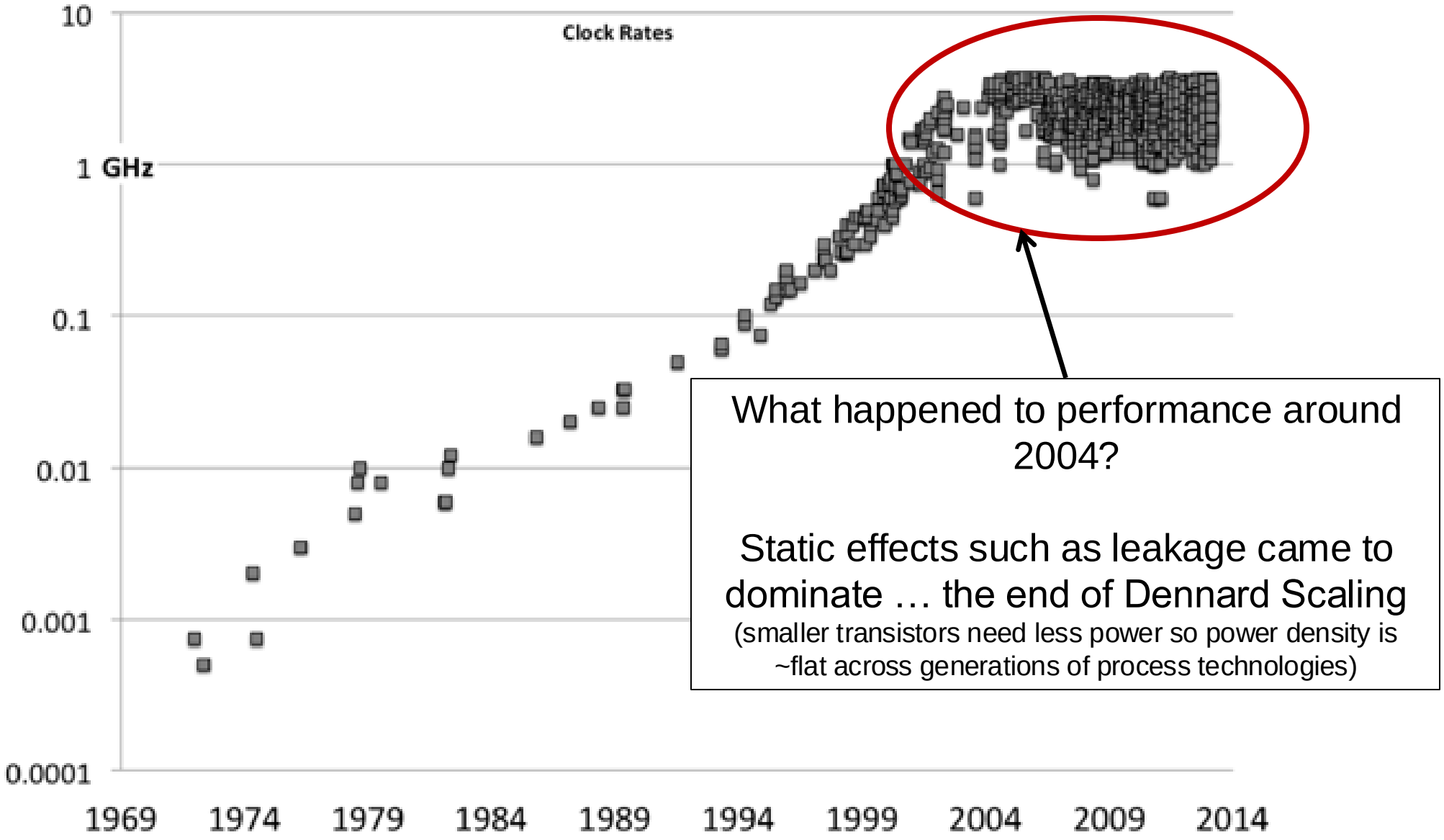
97 watt

275 mm2

%Single Precision TFLOPS running stencil

Programming Intel's 80 core terascale processor SC08, Austin Texas, Nov. 2008, Tim Mattson, Rob van der Wijngaart, Michael Frumkin

CPU Frequency (GHz) over time (years)

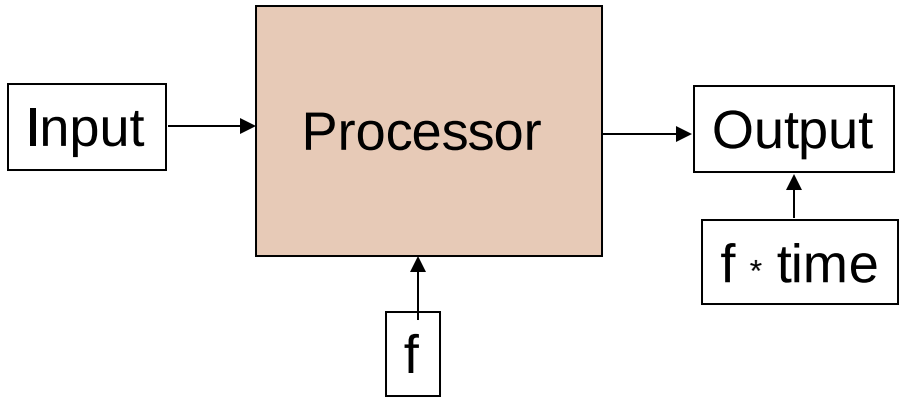


What happened to performance around 2004?

Static effects such as leakage came to dominate … the end of Dennard Scaling
(smaller transistors need less power so power density is ~flat across generations of process technologies)

Source: James Reinders (from the book “structured parallel programming”)

Consider power in a chip …



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f

C = capacitance … it measures the ability of a circuit to store energy:

$C = q/V \rightarrow \quad q = CV$

Work is pushing something (charge or q) across a "distance" … in electrostatic terms pushing q from 0 to V:

$V * q = W.$

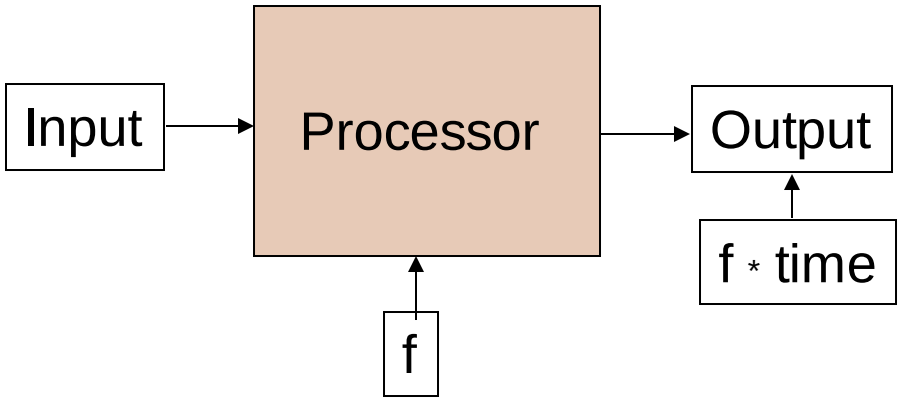
But for a circuit $q = CV$ so

$W = CV^2$

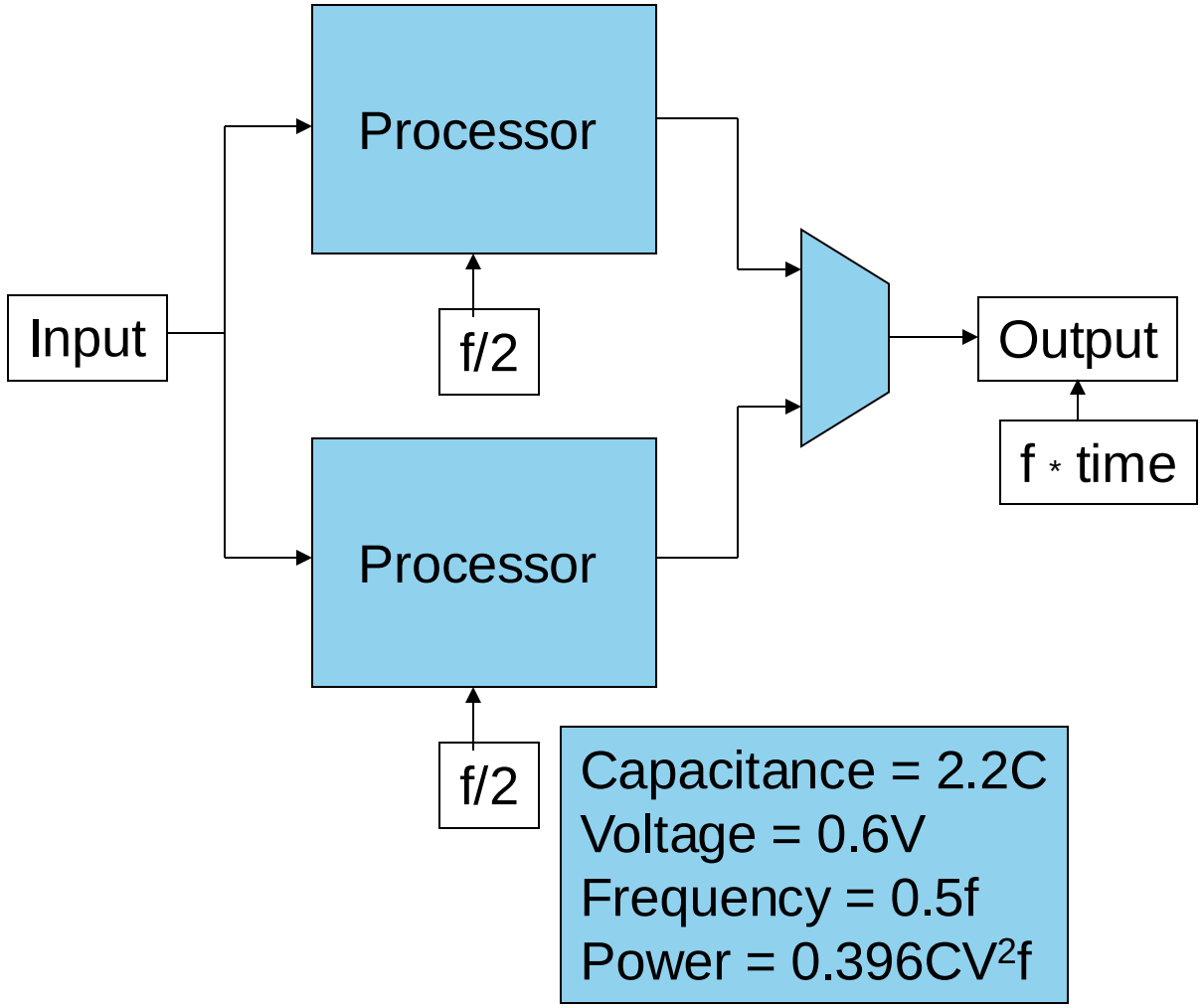
power is work over time … or how many times per second we oscillate the circuit

Power = W* F → Power = CV^2f

... Reduce power by adding cores



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f

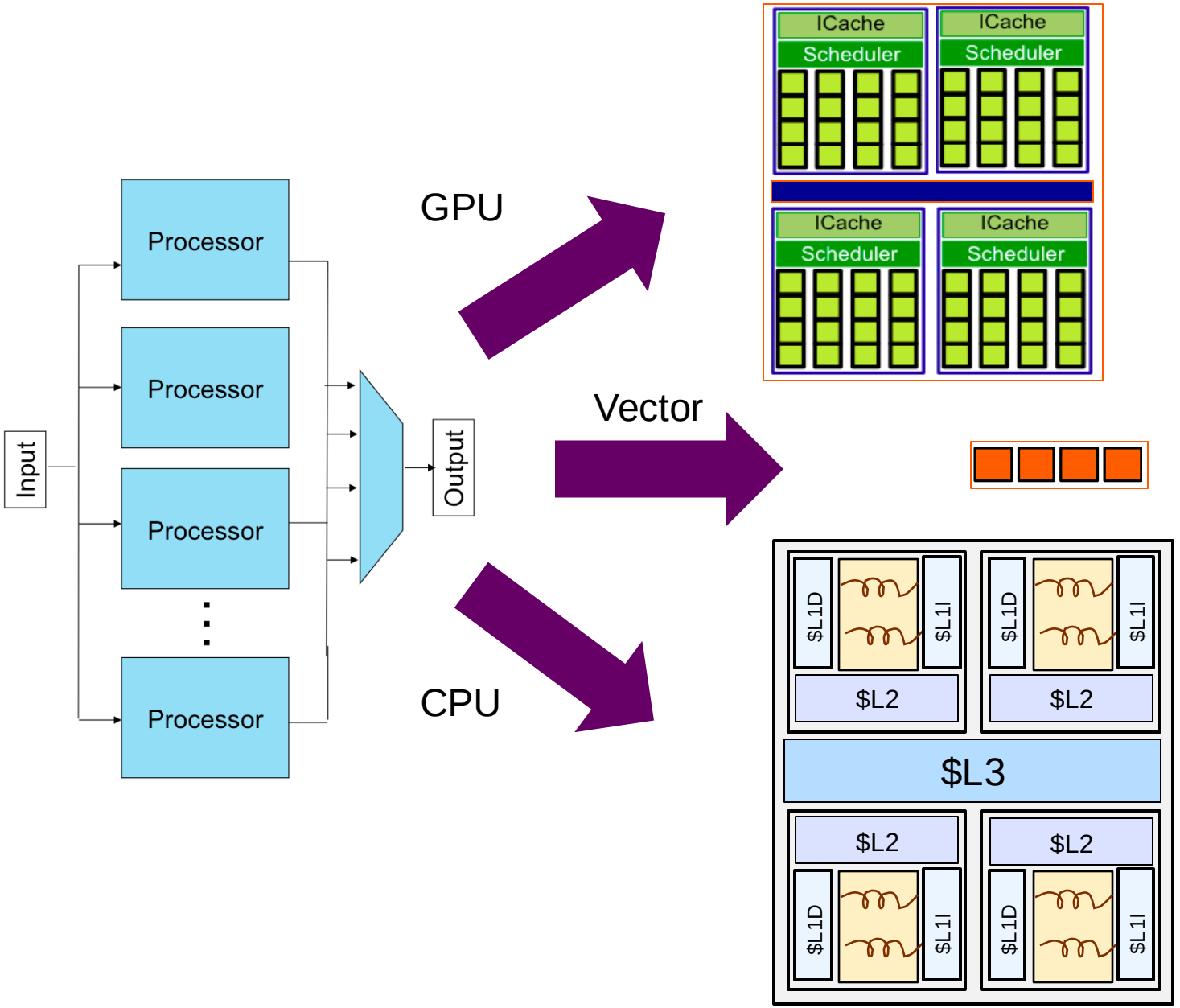


Capacitance = 2.2C
Voltage = 0.6V
Frequency = 0.5f
Power = $0.396CV^2f$

Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, vol.14, no.1, pp.12-31, Jan 1995

Source: Vishwani Agrawal

Parallelism is the Path to Performance

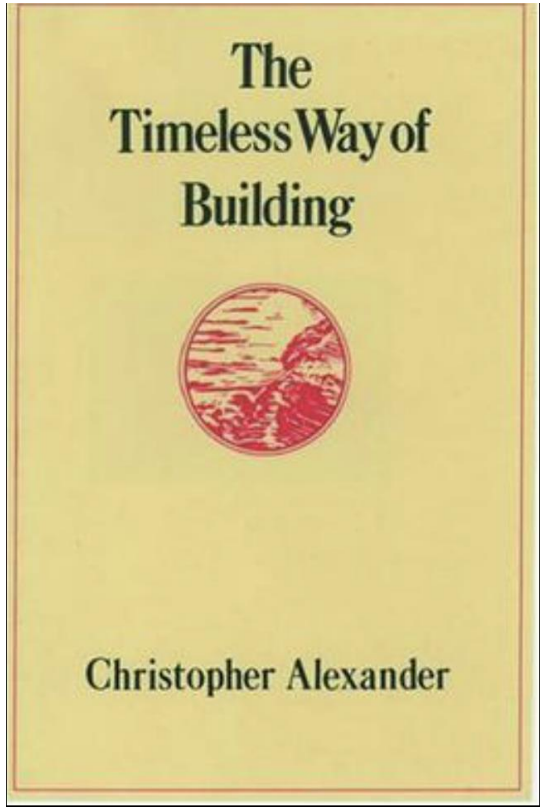


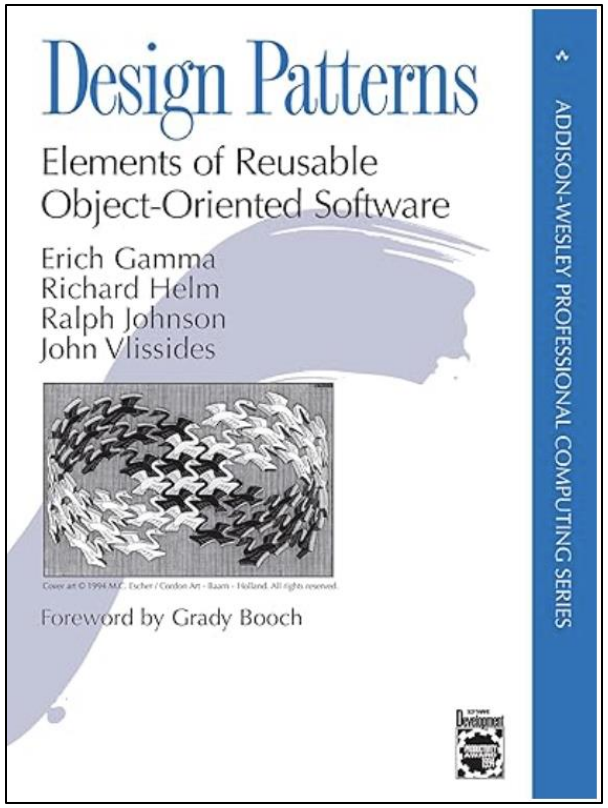
All hardware vendors are in the game … parallelism is ubiquitous.

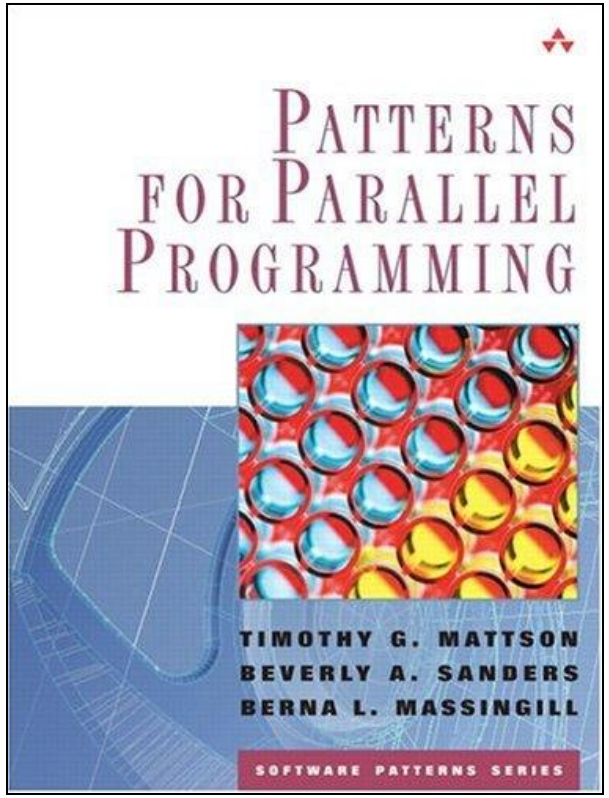
If you care about getting the most from your hardware, you will need to create parallel software.

Design Patterns: An Engineering discipline of design

Design patterns encode the fundamental practice of how people think about classes of problems and their solutions.







The idea of design patterns comes from the architect Christopher Alexander in his 1979 book … where he presents design patterns as a way of capturing that essential ”quality without a name” experts see though often struggle to define.

This book released in 1995 brought order to the chaos that characterized the early days of object oriented programming. They used design patterns to systematically organize best-practices in object oriented design. This is one of the most influential books in the history of computer science.

This book (released in 2004) brought the discipline of design patterns to parallel programming. The goal is to help people understand how to ”think parallel” with examples in OpenMP, MPI and Java

Outline

- Semiconductor Devices and the Need for Parallelism
- Parallelism: Hardware, key abstractions, and programming
  - The CPU: Adventures in Multithreading
  - The Vector (SIMD) unit: Lock-step parallelism in action
  - The GPU: Data Parallelism is your friend
- Scaling to massive parallelism

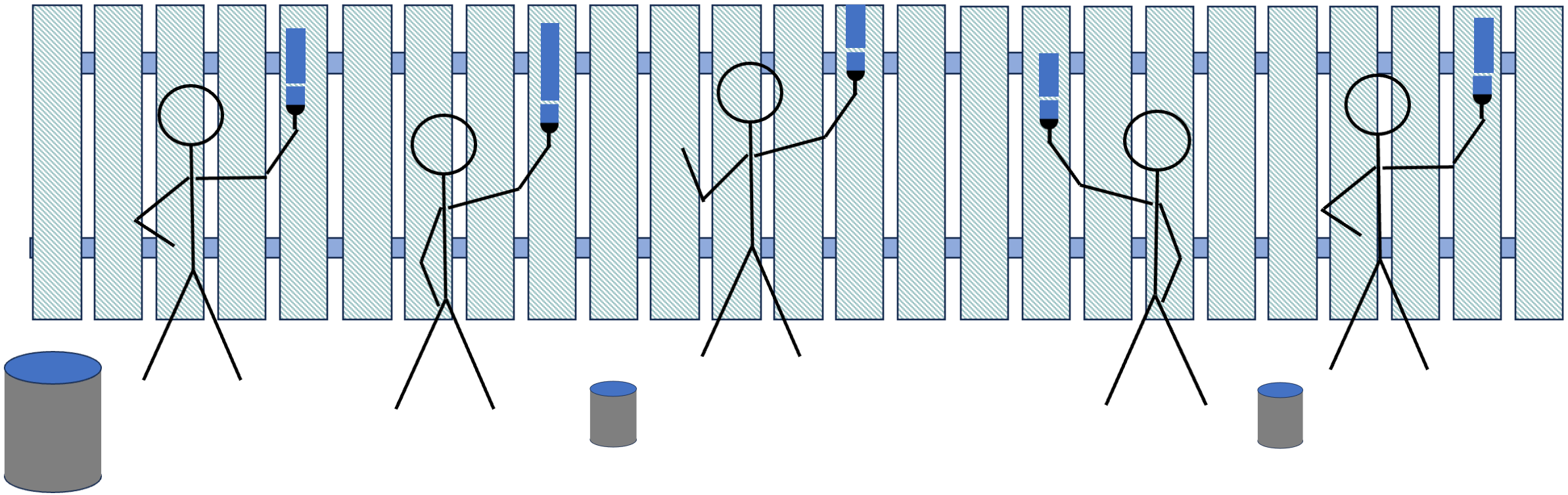
For all Lecture Materials:

- Clone our repository: git clone github.com/tgmattso/CompSciForPhys
- Refresh each week to get latest updates: git pull

How do we describe performance in parallel programs

Painting a fence: Getting help to finish the job in less time

- You must paint a fence. It has 25 planks.
- You have five brushes, one can of paint, and four friends. How long should the job take?



Consider performance of parallel programs

Compute N independent tasks on one processor

Load Data | Compute T_1 | ... | Compute T_N | Consume Results

$$Time_{seq}(1) = T_{load} + N*T_{task} + T_{consume}$$

Compute N independent tasks with P processors

Load Data | Compute T_1 ... Compute T_N | Consume Results

$$Time_{par}(P) = T_{load} + (N/P)*T_{task} + T_{consume}$$

Ideally Cut runtime by ~1/P

(Note: Parallelism only speeds-up the concurrent part)

Talking about performance

- Speedup: the increased performance from running on P processors.

$$S(P) = \frac{Time_{seq}(1)}{Time_{par}(P)}$$

- Perfect Linear Speedup: happens when no parallel overhead and algorithm is 100% parallel.

$$S(P) = P$$

- Efficiency: How well does your observed speedup compare to the ideal case?

$$\varepsilon(P) = \frac{S(P)}{P}$$

Amdahl's Law

- What is the maximum speedup you can expect from a parallel program?

- Approximate the runtime as a part that can be sped up with additional processors and a part that is fundamentally serial.

$$Time_{par}(P) = (serial\_fraction + \frac{parallel\_fraction}{P}) * Time_{seq}$$

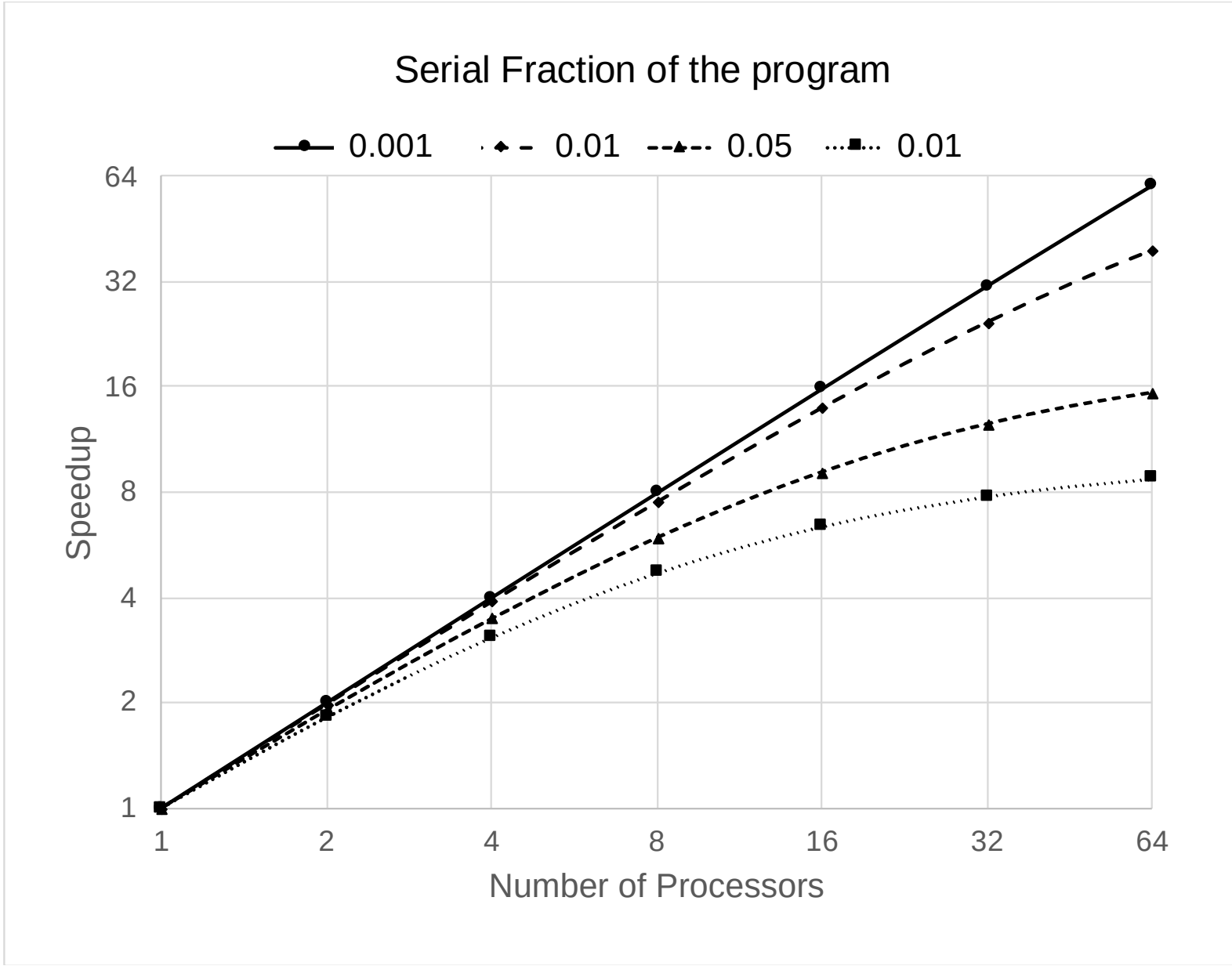
- If the serial fraction is α and the parallel fraction is (1- α) then the speedup is:

$$S(P) = \frac{Time_{seq}}{Time_{par}(P)} = \frac{Time_{seq}}{(a + \frac{1-a}{P}) * Time_{seq}} = \frac{1}{a + \frac{1-a}{P}}$$

- If you had an unlimited number of processors: $P \to \infty$

- The maximum possible speedup is: $S = \frac{1}{\alpha}$ ← Amdahl's Law

Amdahl's Law … It's not just about the maximum speedup



The major parallel Programming systems in 2024 …
at least we have our act together in two cases. ☹

- In HPC, 3 programming environments dominate … covering the major classes of hardware.
  - **MPI**: distributed memory systems … though it works nicely on shared memory computers.
  - **OpenMP**: Shared memory systems … more recently, GPGPU too.
  - **CUDA**, **OpenCL**, **Sycl**, **OpenACC**, **OpenMP** … : GPU programming (use CUDA if you don't mind locking yourself to a single vendor … it is a really nice programming model)
- Even if you don't plan to spend much time programming with these systems … a well rounded HPC programmer must know MPI, OpenMP, and at least one GPU language.

The major parallel Programming systems in 2024 …
at least we have our act together in two cases. ☹

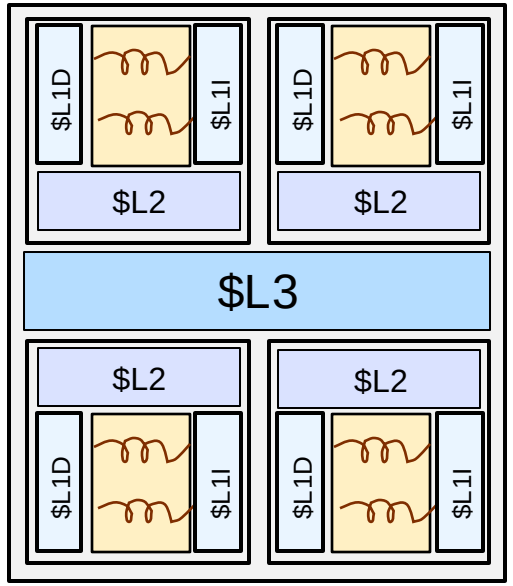
- In HPC, 3 programming environments dominate … covering the major classes of hardware.
  - **MPI**: distributed memory systems … though it works nicely on shared memory computers.
  - **OpenMP**: Shared memory systems … more recently, GPGPU too.
  - **CUDA**, **OpenCL**, **Sycl**, **OpenACC**, **OpenMP**… : GPU programming (use CUDA if you don't mind locking yourself to a single vendor … it is a really nice programming model)

We will discuss these cases as we describe hardware and the associated execution models

- Even if you don't plan to spend much time programming with these systems … a well rounded HPC programmer must know MPI, OpenMP, and at least one GPU language.

Outline

- Semiconductor Devices and the Need for Parallelism
- Parallelism: Hardware, key abstractions, and programming
  - The CPU: Adventures in Multithreading
  - The Vector (SIMD) unit: Lock-step parallelism in action
  - The GPU: Data Parallelism is your friend
- Scaling to massive parallelism



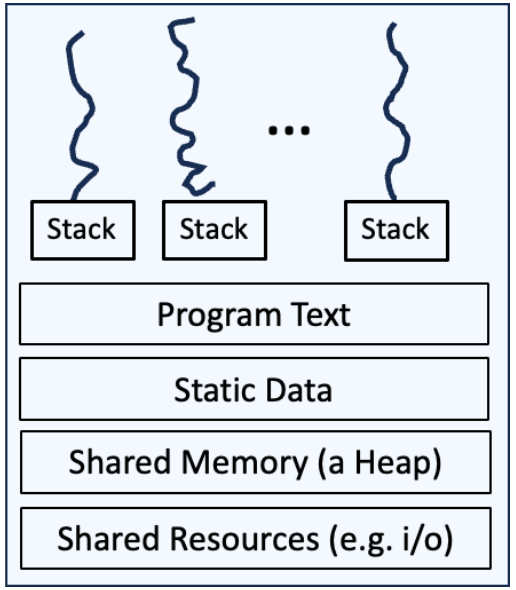
For all Lecture Materials:

- Clone our repository: git clone github.com/tgmattso/CompSciForPhys
- Refresh each week to get latest updates: git pull

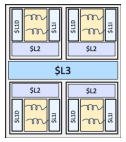
CPU parallelism: Multicore CPUs

- A modern CPU, optimized for low latency across all running programs, will have multiple cores sharing a single memory hierarchy.
- The memory appears as a single address space.
- When every core is treated the same by the operating system (OS) and has an equal cost function to any location in memory, we call this a symmetric multiprocessor or SMP. Because of the caches, this is not an SMP (though we often pretend it is).

- An instance of a program is a process.
- The process has a region of memory, system resources, and one or more threads.
- Parallelism is managed by the programmer as multiple threads mapped to the various cores.



OpenMP* Overview



C$OMP FLUSH

#pragma omp critical

#pragma omp single

C$OMP THREADPRIVATE(/ABC/)

C$OMP ATOMIC

CALL OMP_SET_NUM_THREADS(10)

OpenMP: An API for Writing Parallel Applications

- A set of compiler directives and library routines for parallel application programmers
- Originally … Greatly simplifies writing multithreaded programs in Fortran, C and C++
- Later versions … supports non-uniform memories, vectorization and GPU programming

#pragma omp parallel for private(A, B)

C$OMP PARALLEL REDUCTION (+: A, B)

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

#pragma omp atomic seq_cst

Nthrds = OMP_GET_NUM_PROCS()

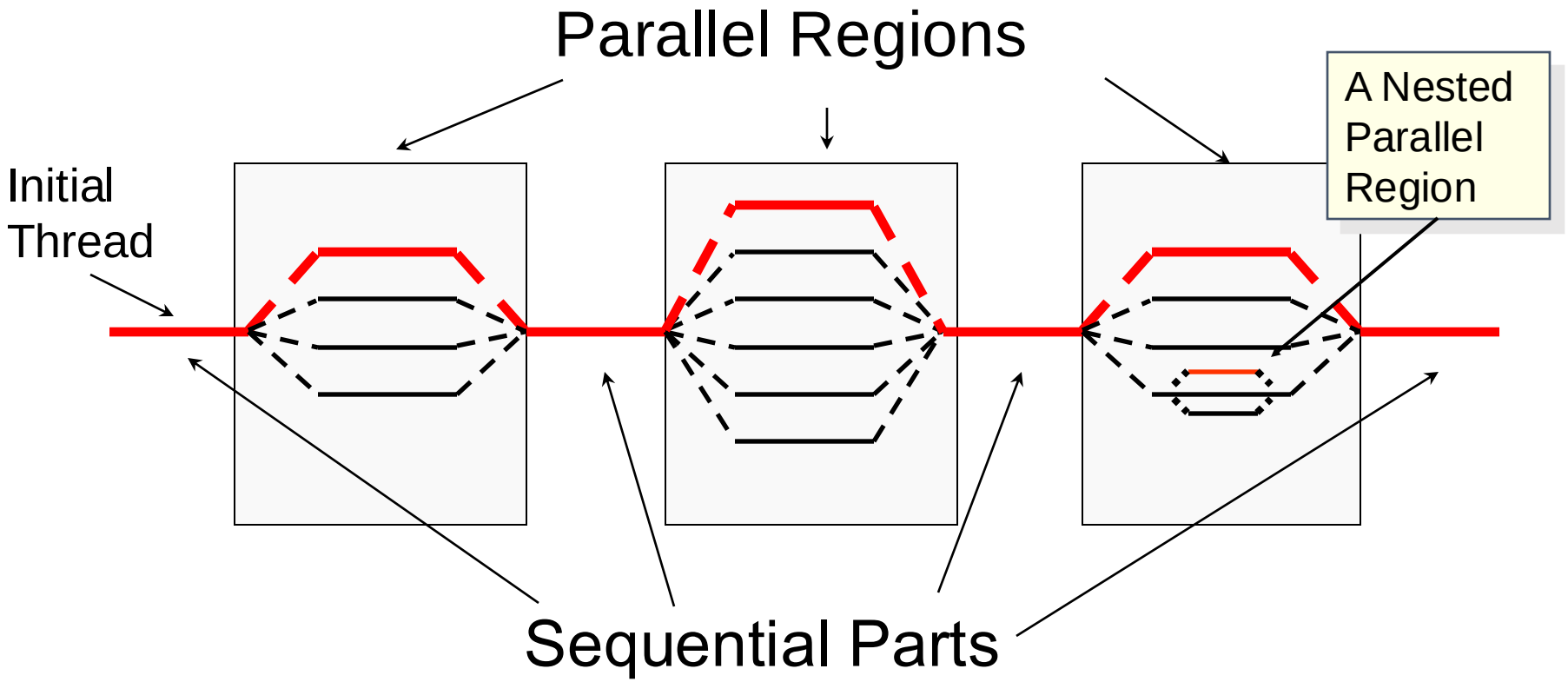
omp_set_lock(lck)

* The name "OpenMP" is the property of the OpenMP Architecture Review Board.

OpenMP Execution model:

Fork-Join Parallelism:

- Initial thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



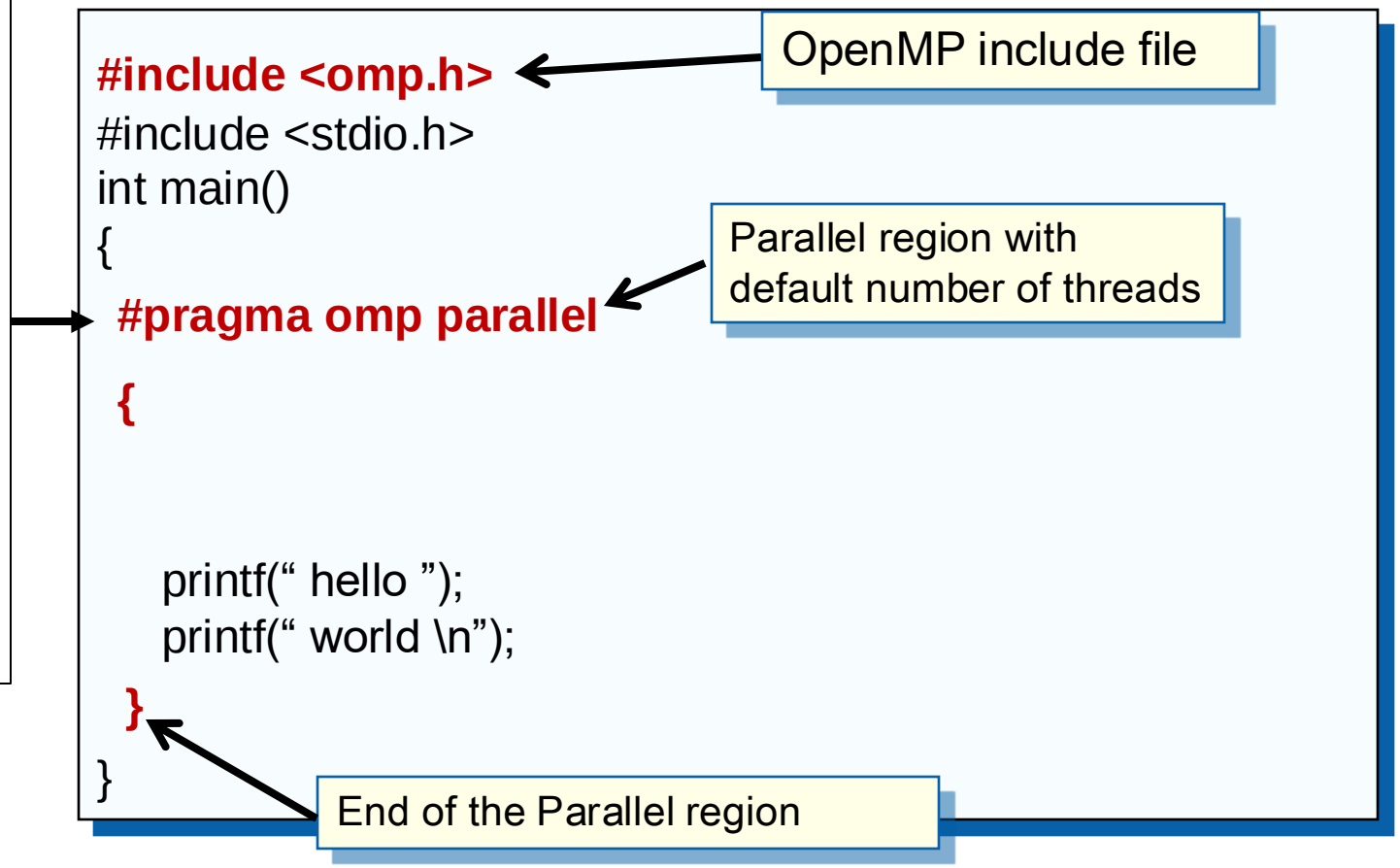
Example: Hello World

- Write a multithreaded program that prints “hello world”.

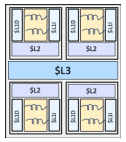
A pragma is a directive to the compiler.

It’s a way for the programmer to tell the compiler what to do as it generates code.

If a compiler doesn’t recognize a pragma, it just skips it.



Example: Hello World



- Write a multithreaded program that prints “hello world”.

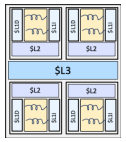
```
#include <omp.h>
#include <stdio.h>
int main()
{
 #pragma omp parallel
 {


    printf(“ hello ”);
    printf(“ world \n”);
 }
}
```

Compile and run:

gcc –fopenmp hello.c

./a.out

Example: Hello World



- Write a multithreaded program that prints “hello world”.

```
#include <omp.h>
#include <stdio.h>
int main()
{
 #pragma omp parallel
 {


    printf(“ hello ”);
    printf(“ world \n”);
 }
}
```

Sample Output:

hello hello world

world

hello hello world

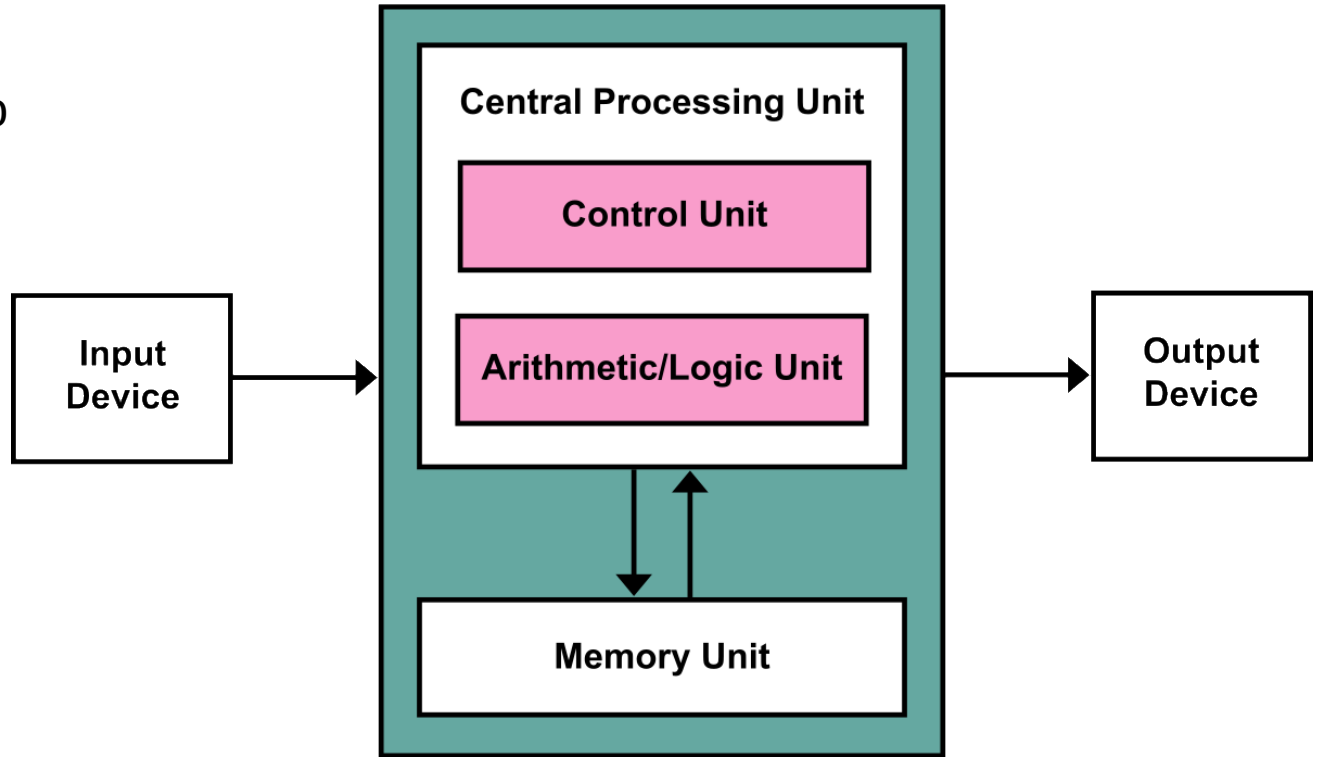
world

Why is the output so oddly scrambled?

A brief digression on the terminology of parallel computing

Let's agree on a few definitions:

- **Computer:**
  - A machine that transforms *input values* into *output values*.
  - Typically, a computer consists of Control, Arithmetic/Logic, and Memory units.
  - The transformation is defined by a stored **program** (von Neumann architecture).



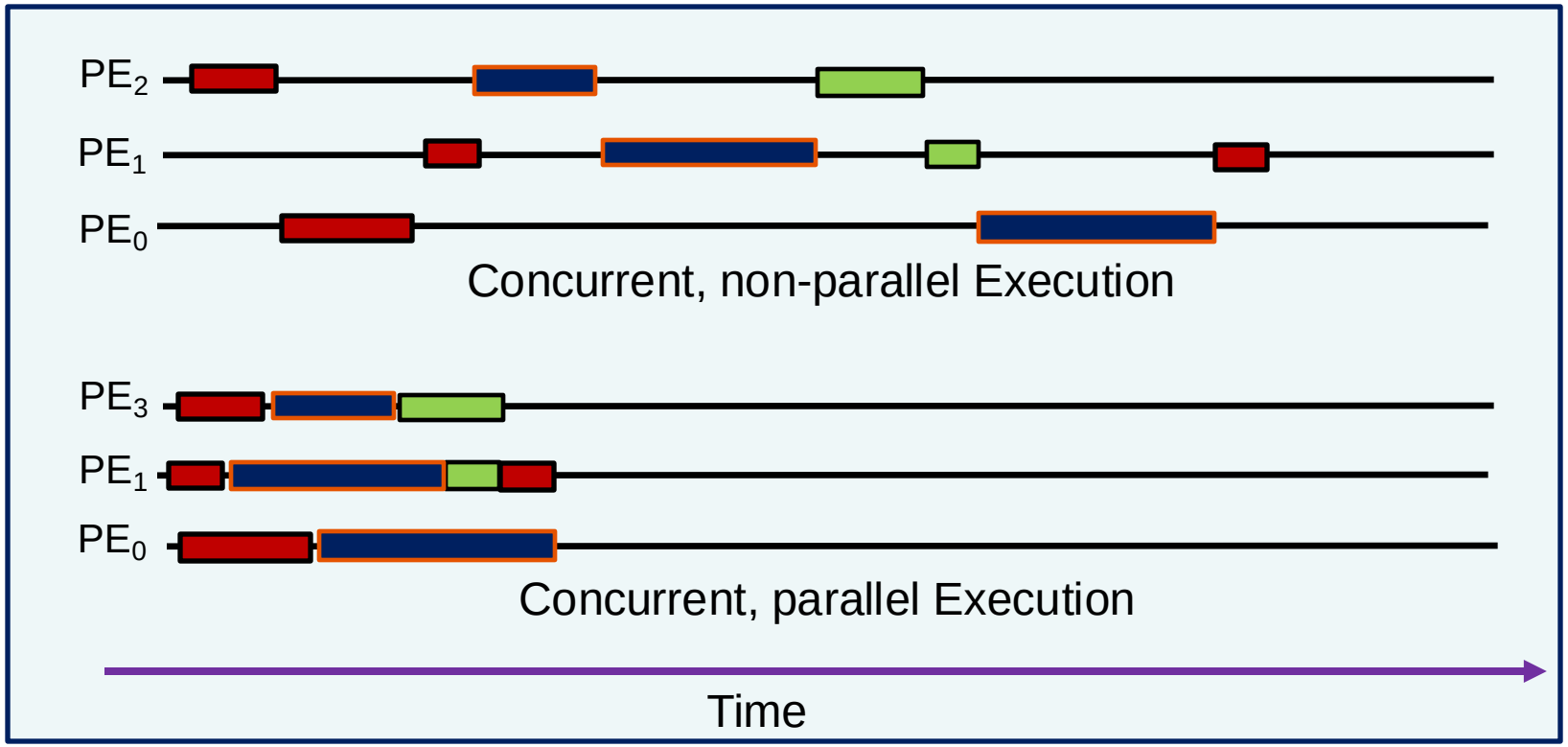
- **Task:**
  - A sequence of instructions plus a data environment. A program is composed of one or more tasks.

- **Active task:**
  - A task that is available to be scheduled for execution. When the task is moving through its sequence of instructions, we say it is making **forward progress**

- **Fair scheduling:**
  - When a scheduler gives each active task an equal *opportunity* for execution.

Concurrency vs. Parallelism

- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are active and unordered. If **scheduled fairly**, they can be described as ***logically*** making **forward progress** at the same time.
  - Parallelism: A condition of a system in which multiple tasks are **actually** making **forward progress** at the same time.

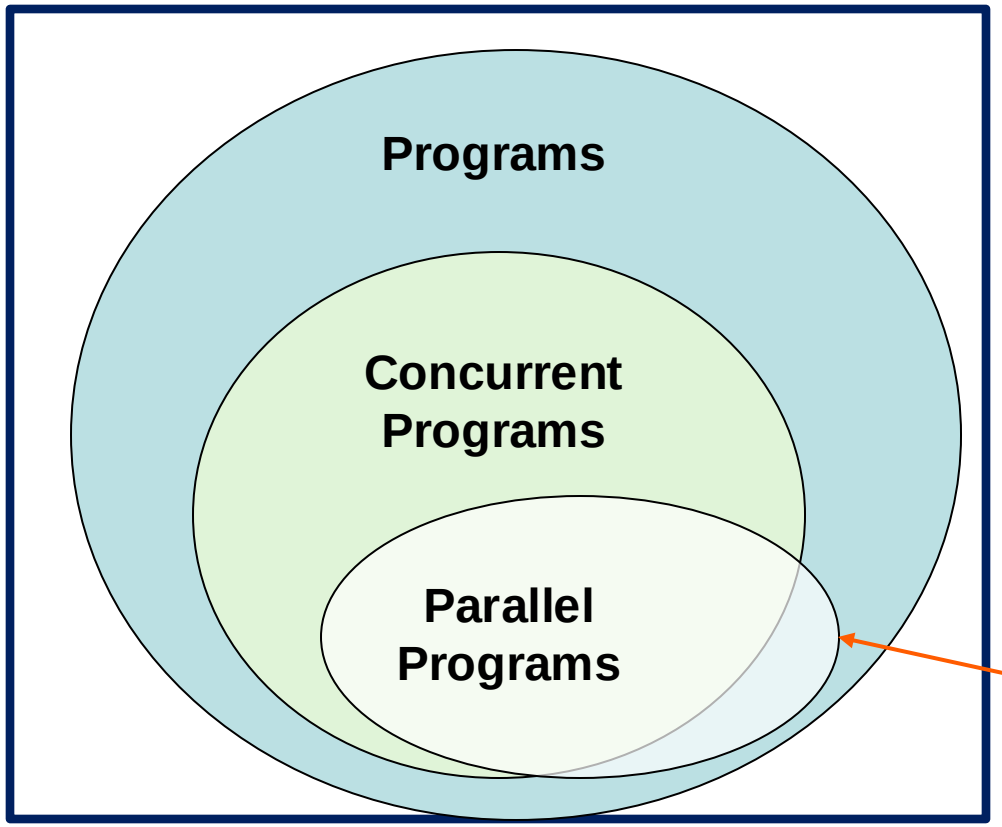


PE = Processing Element

Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

Concurrency vs. Parallelism

- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are active and unordered. If **scheduled fairly**, they can be described as ***logically*** making **forward progress** at the same time.
  - Parallelism: A condition of a system in which multiple tasks are **actually** making **forward progress** at the same time.



In most cases, parallel programs exploit concurrency in a problem to run tasks on multiple processing elements

We use Parallelism to:
- Do more work in less time
- Work with larger problems

If tasks execute in “lock step” they are not concurrent, but they are still parallel. Example … a SIMD unit.

Figure from “An Introduction to Concurrency in Programming Languages” by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

Example: Hello World

- A multithreaded program that prints “hello world”.

```
#include <omp.h>
#include <stdio.h>
int main()
{
 #pragma omp parallel
 {


    printf(“ hello ”);
    printf(“ world \n”);
 }
}
```

Sample Output:

hello hello world

world

hello hello world

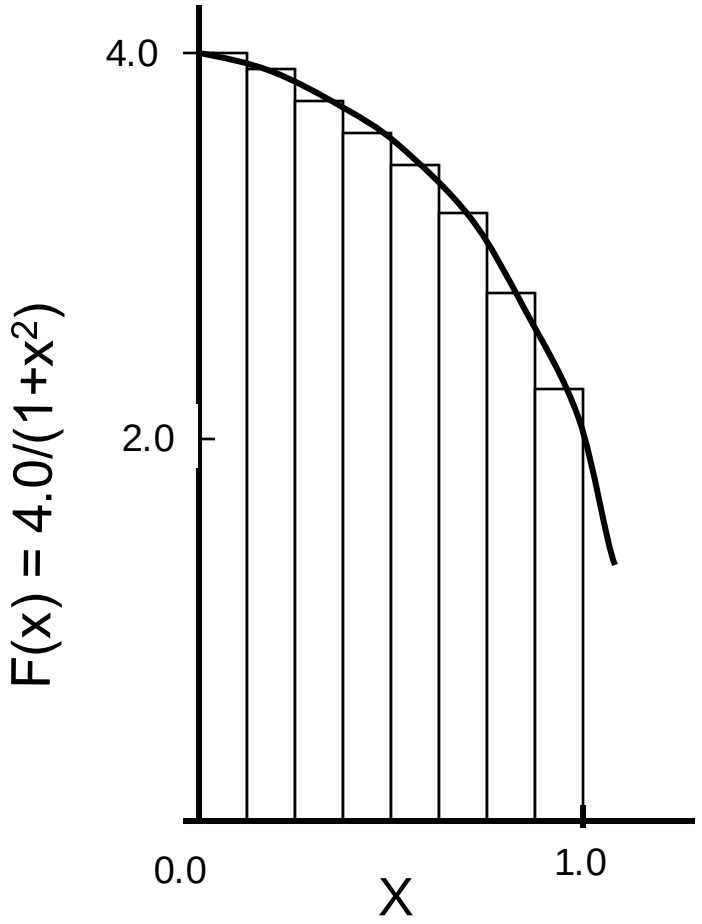
world

The threads execute concurrently. The statements are interleaved based on how the operating schedules the threads

Let’s consider a more complicated example so we can see how OpenMP is typically used in CPU programming.

An Interesting Problem to Play With

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of N rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x = \Delta x \sum_{i=0}^{N} F(x_i) \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

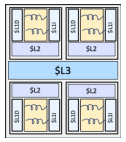
Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{
   double x, pi, sum = 0.0;

   step = 1.0/(double) num_steps;

   x = 0.5;
   for (int i=0;i< num_steps; i++){
            x = x + step;
            sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
}
```

Serial PI Program … loop-carried dependence on x removed

```
static long num_steps = 100000;
double step;
int main ()
{
   double x, pi, sum = 0.0;

   step = 1.0/(double) num_steps;


   for (int i=0;i< num_steps; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
}
```

Parallel PI Program



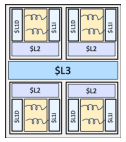
```
#include <omp.h>
static long num_steps = 100000;
double step;
int main ()
{
   double x, pi, sum = 0.0;

   step = 1.0/(double) num_steps;

   #pragma omp parallel for private(x) reduction(+:sum)
   for (int i=0;i< num_steps; i++){
          x = (i+0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
}
```

parallel	Create a team of threads
for	Spread loop iterations across threads
private(x)	Each thread gets its own copy of **x**
reduction(+:sum)	Each thread computes on its own copy of **sum**, which are combined into a single value of **sum** at the end of the loop once all threads are done

Loop Level Parallelism Design Pattern:

1. Find loop with concurrency suitable for parallel execution (***find concurrency***)
2. Make loop iterations independent from each other (***expose concurrency***)
3. Add code to support parallel execution (***exploit concurrency***)

Irregular Parallelism



- Let's call a problem "irregular" when one or both of the following hold:
  - Data Structures are sparse or involve indirect memory references
  - Control structures are not basic for-loops

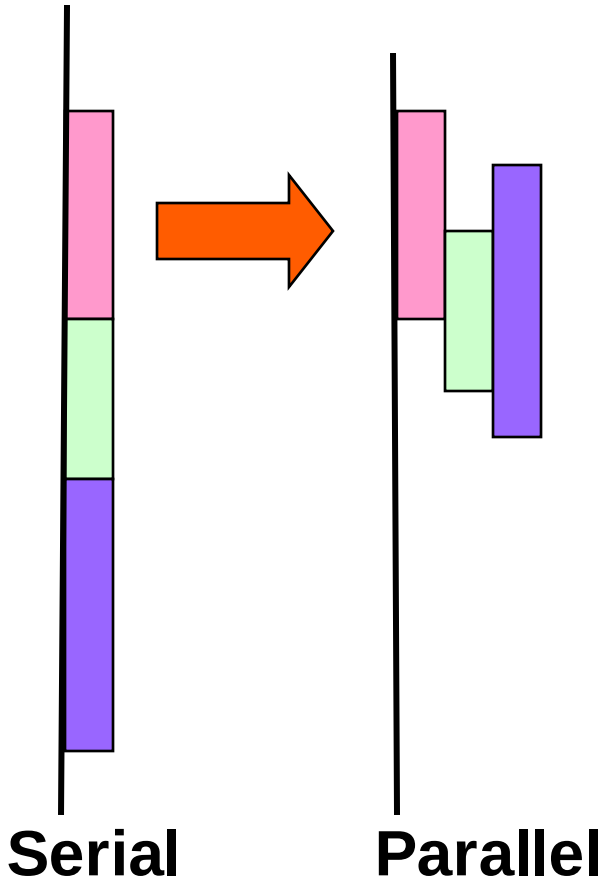
- Example: Traversing Linked lists:

```
p = listhead ;
while (p) {
  process(p);
  p=p->next;
}
```

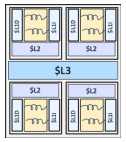
- To handle irregular parallelism, we added a new capability to OpenMP.

Tasks for irregular Parallelism

- Tasks are independent units of work composed of:
  - code to execute
  - data to compute with
- Threads are assigned to perform the work of each task.
  - The thread that encounters a task construct may execute the task immediately or defer it for later execution.
  - The tasks go into a queue from which other threads can execute them.
- Tasks are often nested: i.e., a task may itself generate tasks.
  - Tasks created from nested task constructs are called **descendent tasks**.
  - Tasks created in the same structured block are **sibling tasks**.
- When to tasks complete?
  - A taskwait causes prior sibling tasks to complete before subsequent task launch
  - All tasks (siblings and all descendants) complete at the next barrier



Linked List Traversal



```
#pragma omp parallel
{
    #pragma omp single
    {
        p = listhead ;
        while (p) {
            #pragma omp task firstprivate(p)
            {
              process (p);
            }
            p=next (p) ;
        }
    }
}
```

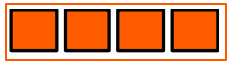
One thread processes the while loop to create tasks.

Other threads wait at the end of the Single Construct (an implied barrier)

Package work into a task and capture the current value of **p**

Outline

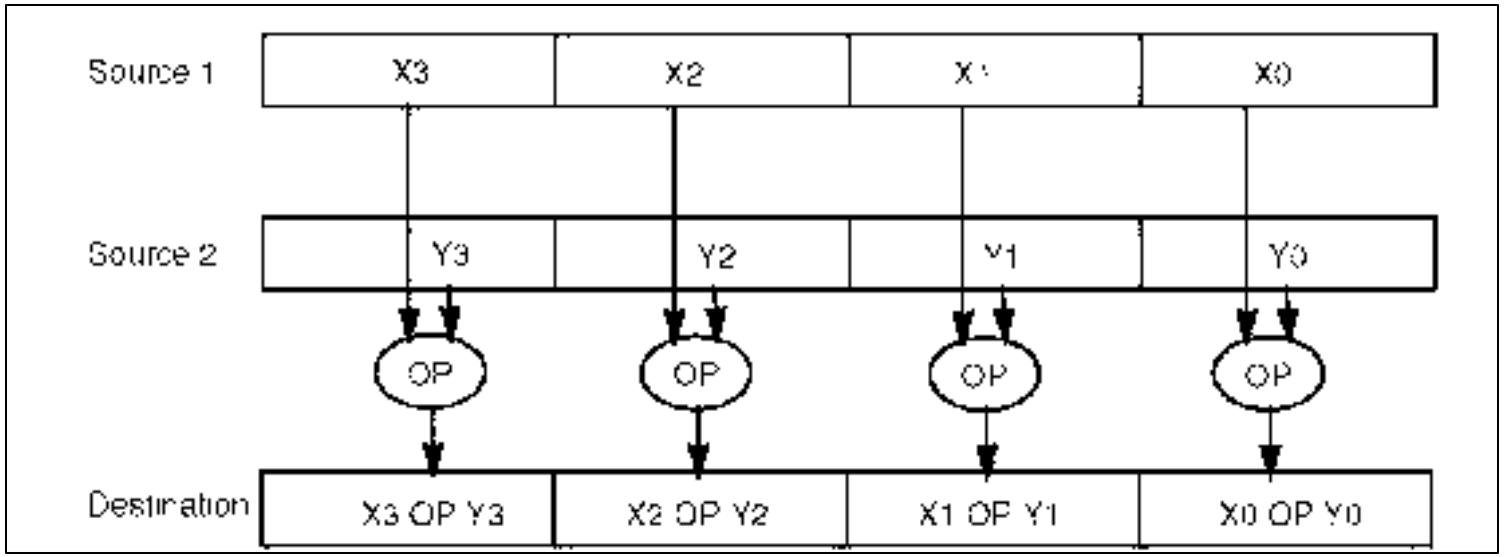
- Semiconductor Devices and the Need for Parallelism
- Parallelism: Hardware, key abstractions, and programming
  - The CPU: Adventures in Multithreading
  - The Vector (SIMD) unit: Lock-step parallelism in action
  - The GPU: Data Parallelism is your friend
- Scaling to massive parallelism



For all Lecture Materials:

- Clone our repository: git clone github.com/tgmattso/CompSciForPhys
- Refresh each week to get latest updates: git pull

Vector (SIMD*) Programming: Data Parallelism Pattern

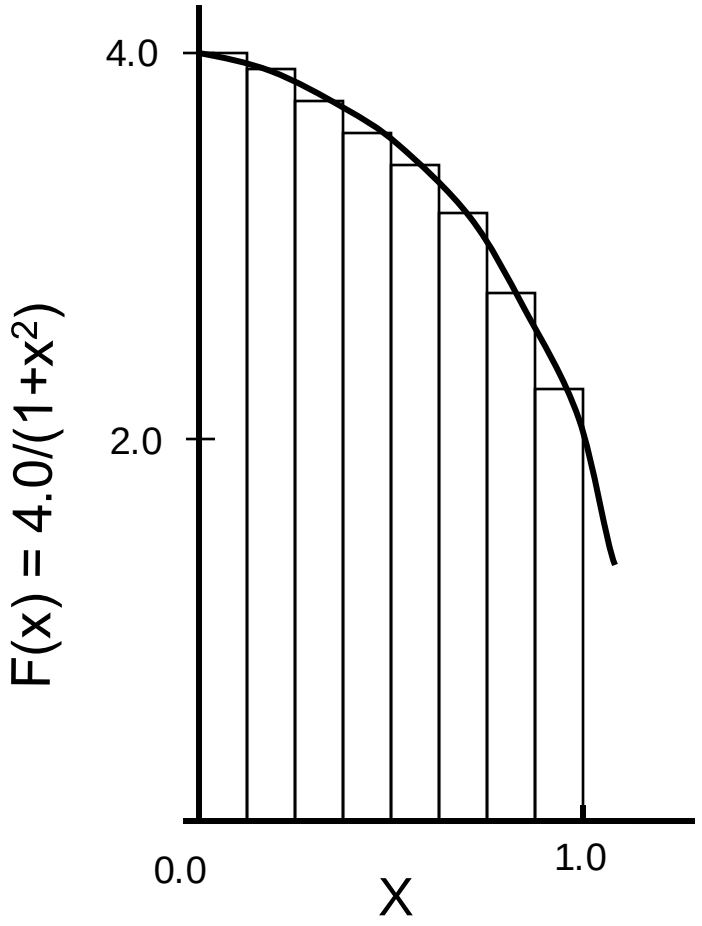


- A vector unit has a set of wide registers that can hold multiple values.
- Pack operand values into the registers.
- A single vector-operation applies the op to all of the values.
- Parallelism is “in the data” and each vector lane proceeds in “lock-step” with the others; driven by a single stream of instructions.

- Computer architects love vector units, since they permit space and energy efficient parallel implementations. All commercial CPUs used in laptops and servers include vector processing units.
- However, standard SIMD instructions on CPUs are inflexible, and can be difficult to use. Options:
  - Let the compiler do the job
  - Assist the compiler with language level constructs for explicit vectorization.
  - Use intrinsics … an approach similar to assembly level programming.

*SIMD: Single Instruction Multiple Data

Example Problem: Numerical Integration



Mathematically, we know that:

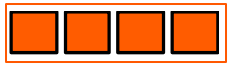
$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial PI program

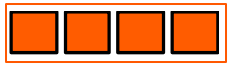


Literals as double (no-vec), 0.012 secs
Literals as Float (no-vec), 0.0042 secs

```
static long num_steps = 100000;

int main ()
{           int i;      float x, pi, sum = 0.0;

            float step = 1.0/(float) num_steps;

            for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

Normally, I'd use double types throughout to minimize roundoff errors especially on the accumulation into sum. But to maximize impact of vectorization for these examples, we'll use float types.

Explicit vectorization of our Pi Program: Step 1 ... Unroll the loop



- We need each loop iteration to fit in the vector unit

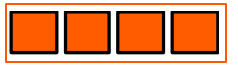
- What is the width of your vector unit?
- We'll use SSE* which is 128 bits wide.
- A float in C is 32 bits wide … 4 floats fits in 128 bits

So unroll our loop by four

```
float pi_unroll(int  num_steps)
{
   float step, x0, x1, x2, x3, pi, sum = 0.0;

   step = 1.0f/(float) num_steps;

   for (int i=1;i<= num_steps; i=i+4){      //unroll by 4, assume num_steps%4 = 0
      x0 = (i-0.5f)*step;
      x1 = (i+0.5f)*step;
      x2 = (i+1.5f)*step;
      x3 = (i+2.5f)*step;
      sum += 4.0f*(1.0f/(1.0f+x0*x0) + 1.0f/(1.0f+x1*x1)  + 1.0f/(1.0f+x2*x2)  + 1.0f/(1.0f+x3*x3));
   }

     pi = step * sum;
     return pi;
}
```

*SSE: A widely used Intel® x86™ instruction set for vector operations over 128 bit registers

Explicit vectorization of our Pi Program: Step 2 ... Add SSE intrinsics

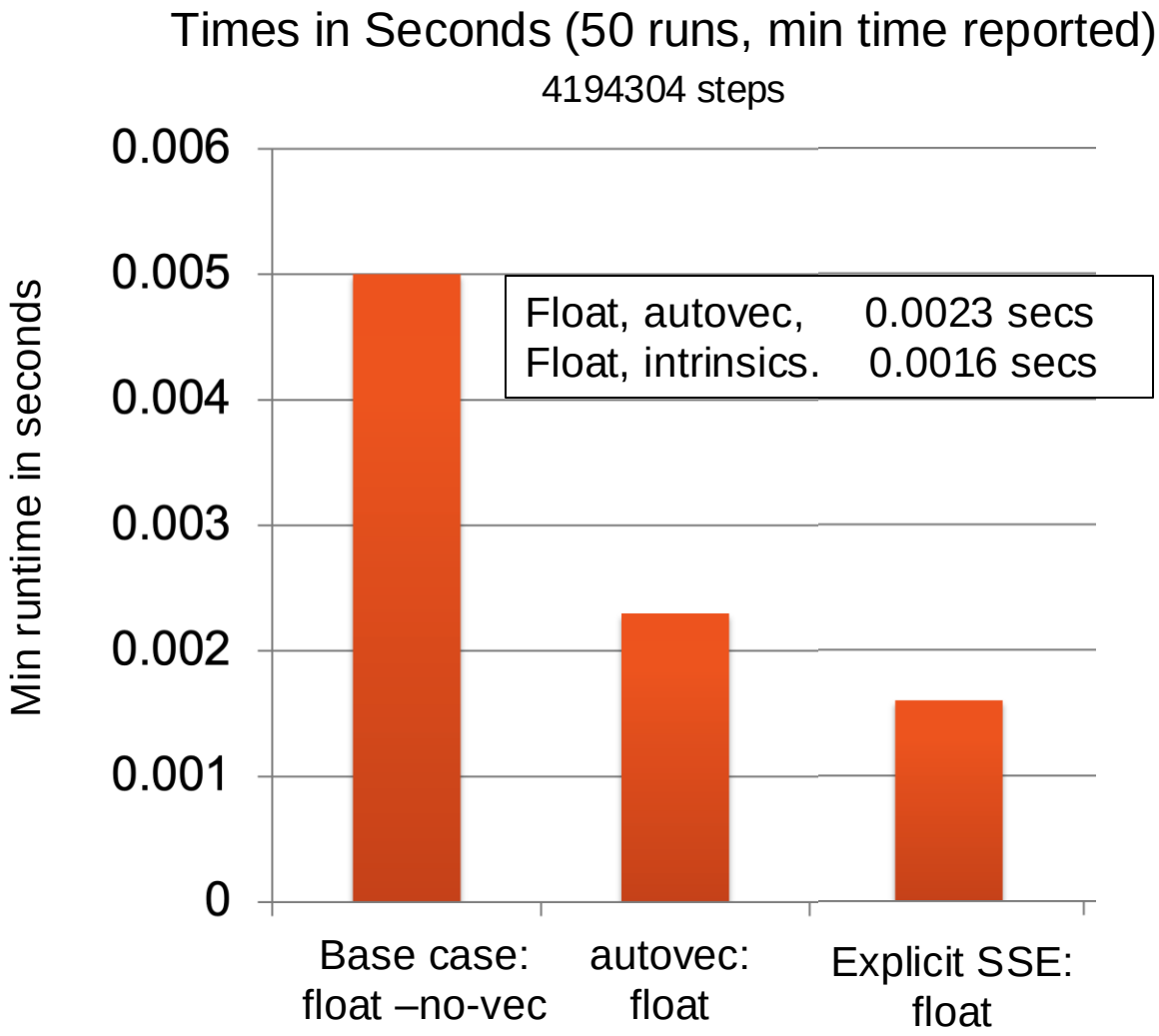


```
#include <immintrin.h>
float pi_sse(int  num_steps)
 {
   float scalar_one =1.0, scalar_zero = 0.0,  ival, scalar_four =4.0, step, pi, vsum[4];
   float step = 1.0/(float) num_steps;

   __m128 ramp   = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
   __m128 one    = _mm_load1_ps(&scalar_one);
   __m128 four   = _mm_load1_ps(&scalar_four);
   __m128 vstep  = _mm_load1_ps(&step);
   __m128 sum    = _mm_load1_ps(&scalar_zero);
   __m128 xvec;   __m128 denom;  __m128 eye;

  for (int i=0;i< num_steps; i=i+4){          // unroll loop 4 times
      ival       = (float)i;                  // and assume num_steps%4 = 0
      eye        = _mm_load1_ps(&ival);
      xvec       = _mm_mul_ps(_mm_add_ps(eye,ramp),vstep);
      denom  = _mm_add_ps(_mm_mul_ps(xvec,xvec),one);
      sum      = _mm_add_ps(_mm_div_ps(four,denom),sum);
   }
   _mm_store_ps(&vsum[0],sum);
   pi = step * (vsum[0]+vsum[1]+vsum[2]+vsum[3]);
  return pi;
}
```

The vast majority of programmers never write explicitly vectorized code.

It is important to understand explicit vectorization so you appreciate what the compiler does to vectorize code for you

PI program Results:



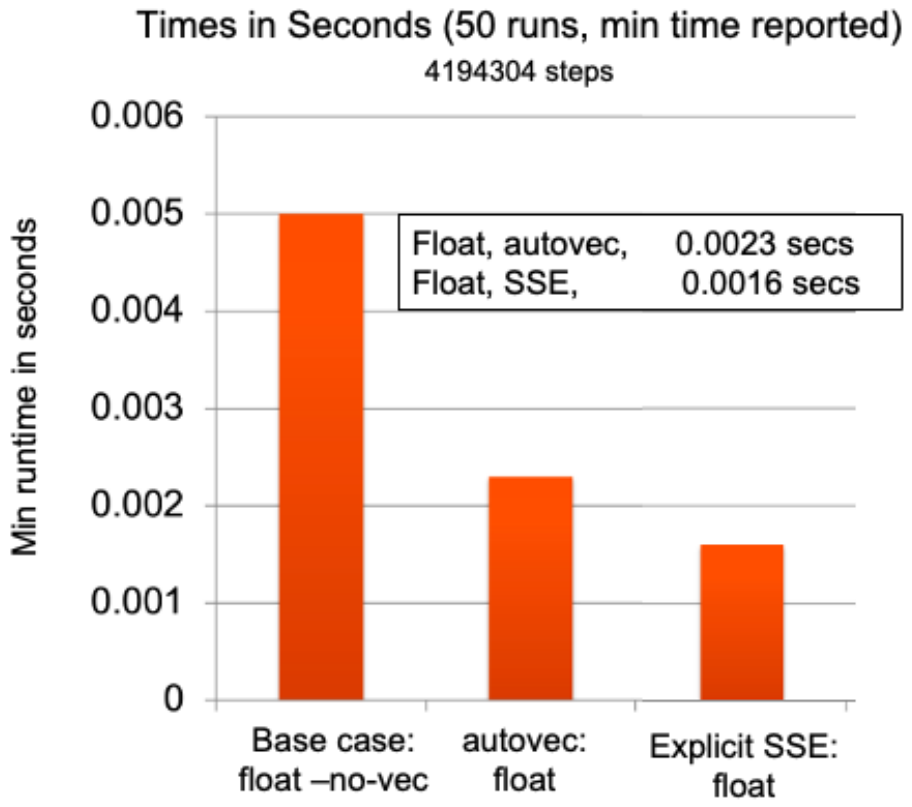
- Intel Core i7, 2.2 Ghz, 8 GM 1600 MHz DDR3, Apple MacBook Air OS X 10.10.5.
- Intel(R) C Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 15.0.3.187 Build 20150408

Compiler vectorization: How to help the compiler do a better job

Tell the compiler to automatically vectorize with the –O2 flag.

Vectorize and carry out aggressive optimizations with the –O3 flag

Example:
gcc –fopenmp -O3 pi.c



Structure your code to make the vectorizing compiler's job easier:

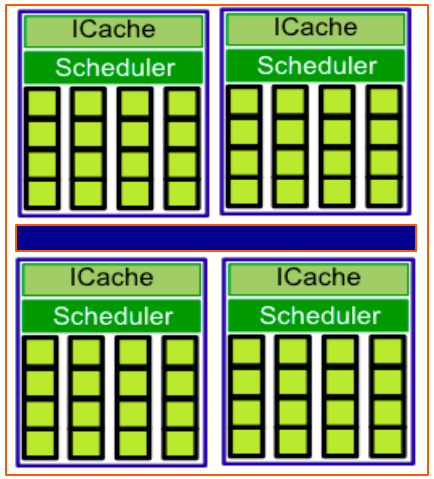
- Countable loops
- Regular control flow across iterations*
- No variations in control flow
- Contiguous memory access
- Independent memory access
- Avoid aliasing
- Avoid indirect memory access (x[y[i]])

*By regular control flow we mean the loop does not include features that cause execution to vary across iterations … such as break, continue, and functions that are not inlined.

- Intel Core i7, 2.2 Ghz, 8 GM 1600 MHz DDR3, Apple MacBook Air OS X 10.10.5.
- Intel(R) C Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 15.0.3.187 Build 20150408

Outline

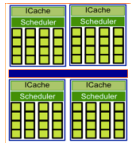
- Semiconductor Devices and the Need for Parallelism
- Parallelism: Hardware, key abstractions, and programming
  - The CPU: Adventures in Multithreading
  - The Vector (SIMD) unit: Lock-step parallelism in action
  - The GPU: Data Parallelism is your friend
- Scaling to massive parallelism



For all Lecture Materials:

- Clone our repository: git clone github.com/tgmattso/CompSciForPhys
- Refresh each week to get latest updates: git pull

The "BIG idea" Behind GPU programming



Traditional Loop based vector addition (vadd)

```
int main() {
  int N = . . . ;
  float *a, *b, *c;

  a* =(float *) malloc(N * sizeof(float));

  // ... allocate other arrays (b and c)
  // and fill with data

  for (int i=0;i<N; i++)
     c[i] = a[i] + b[i];

}
```

vadd with CUDA

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i < N) c[i] = a[i] + b[i];
}

int main () {
   int N = ... ;
   float *a, *b, *c;
   cudaMalloc (&a,  sizeof(float) * N);
 // ... allocate other arrays (b and c)
 // and fill with data

 // Use thread blocks with 256 threads each
   vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

Assume a GPU with unified shared memory … allocate on host, visible on device too

This is data parallelism … like we did on the SIMD (vector) unit … but at a MUCH greater scale

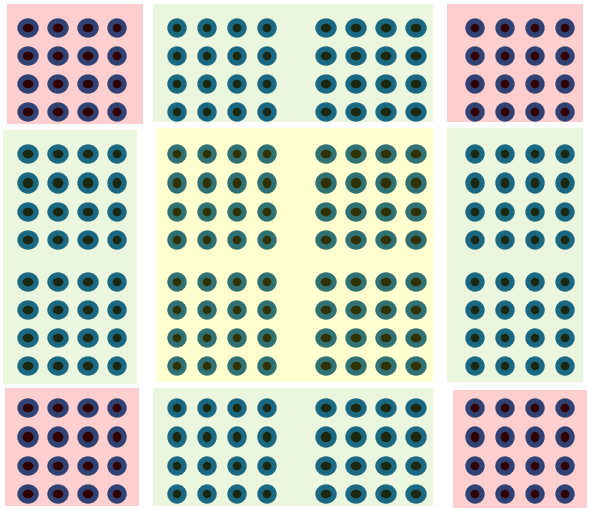
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Turn kernel code into a scalar work-item

```
// Compute sum of order-N matrices: C = A + B
void __global__
matAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j<N) c[i][j] == a[i][j] + b[i][j];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // define threadBlocks and the Grid
   dim3 dimBlock(4,4);
   dim3 dimGrid(4,4);

  // Launch kernel on Grid
    matAdd <<< digGrid,dimBlock>>> (a, b, c, N);
}
```

This is CUDA code

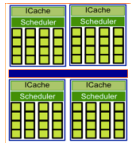
2. Map work-items onto an N dim index space.



3. Map data structures onto the same index space

4. Run on hardware designed around the same SIMT execution model

SIMT: One instruction stream maps onto many SIMD lanes



- SIMT model: Individual scalar instruction streams are grouped together for SIMD execution on hardware

A stream of Scalar instructions

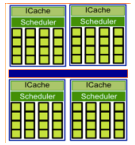
```
ld x
mul a
ld y
add
st y
```

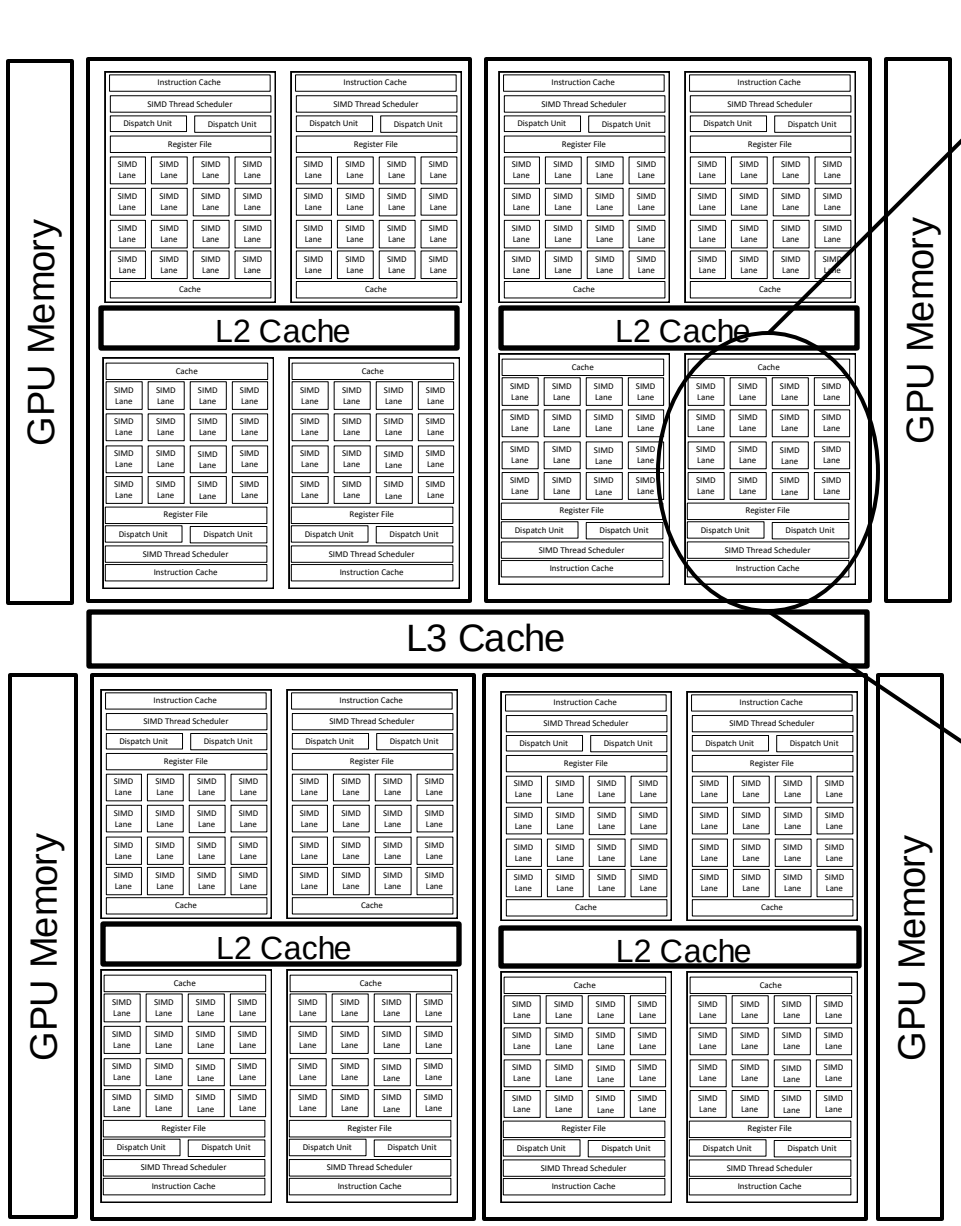
SIMD execution scheduled across a fixed number of SIMD Lanes (SL)

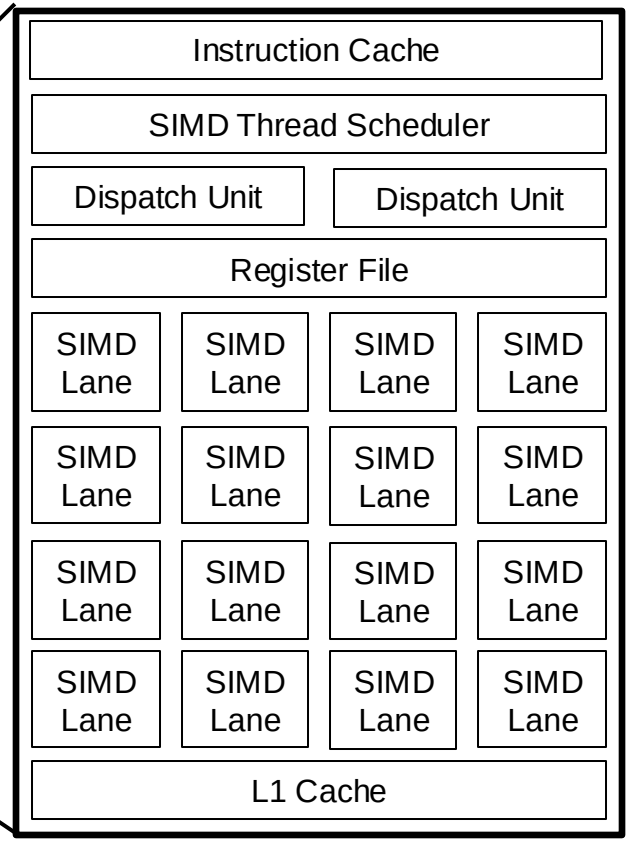
SL0	SL1	SL2	SL3	SL4	SL5	SL6	SL7
ld x	ld x	ld x	ld x	ld x	ld x	ld x	ld x
mul a	mul a	mul a	mul a	mul a	mul a	mul a	mul a
ld y	ld y	ld y	ld y	ld y	ld y	ld y	ld y
add	add	add	add	add	add	add	add
st y	st y	st y	st y	st y	st y	st y	st y

NVIDIA calls this set of work-items a **warp**

A Generic GPU (following Hennessey and Patterson)

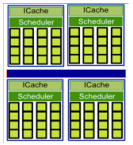






A multithreaded SIMD processor

GPU terminology is Broken (sorry about that)



Hennessy and Patterson	CUDA	OpenCL
Multithreaded SIMD Processor	Streaming multiprocessor	Compute Unit
SIMD Thead Scheduler	Warp Scheduler	Work-group scheduler
SIMD Lane	CUDA Core	Processing Element
GPU Memory	Global Memory	Global Memory
Private Memory	Local Memory	Private Memory
Local Memory	Shared Memory	Local Memory
Vectorizable Loop	Grid	NDRange
Sequence of SIMD Lane operations	CUDA Thread	work-item
A thread of SIMD instructions	Warp	sub-group

My friends at NVIDIA hate the name “SIMD Lane” for their processing Elements. While it is true that for maximum performance and for portability across GPUs, it is best to think of them as SIMD lanes, NVIDIA engineers have made great advancements in their design so considerable divergence across their processing elements (PE) within a warp can occur without severe performance penalties. Hence, I prefer the name “Processing Element”. The name “CUDA Core” is completely wrong as the GPU PE is in now way related to a CPU core. Likewise with their misuse of the term “thread”.

We will focus on OpenMP for GPU programming
… It is a lot easier to work with than CUDA

Running code on a GPU

```
int main(void) {
  int N = 1024;
  double A[N], B[N];

  #pragma omp target
  {

    for (int ii = 0; ii < N; ++ii) {

      A[ii] = A[ii] + B[ii];

    }

  } // end of target region
}
```

Offload this work to the GPU

1. Variables created in host memory.

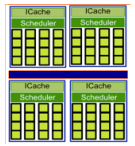
2. Scalar **N** and stack arrays **A** and **B** are copied *to* device memory. Execution transferred to device.

3. **ii** is **private** on the device as it's declared within the target region

4. Execution on the device.

5. stack arrays **A** and **B** are copied *from* device memory back to the host. Host resumes execution.

Now let's run code in parallel on the device



```
int main(void) {
  int N = 1024;
  double A[N], B[N];

  #pragma omp target
  {
    #pragma omp loop
    for (int ii = 0; ii < N; ++ii) {

      A[ii] = A[ii] + B[ii];

    }

  } // end of target region
}
```

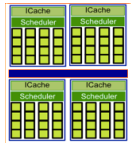
Offload this work to the GPU

Loop iteration space defined the index-space.

Loop body is the kernel

Run an instance of the kernel for each point in the index space

Now let's run code in parallel on the device

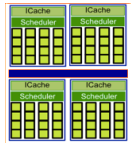


```
int main(void) {
  int N = 1024;
  double A[N], B[N];

  #pragma omp target teams loop


    for (int ii = 0; ii < N; ++ii) {

      A[ii] = A[ii] + B[ii];

    }


}
```

Merge target and loop on one line.

You need the teams clause to force mapping across compute units (streaming multiprocessors)

Solution: Simple vector add in OpenMP on GPU

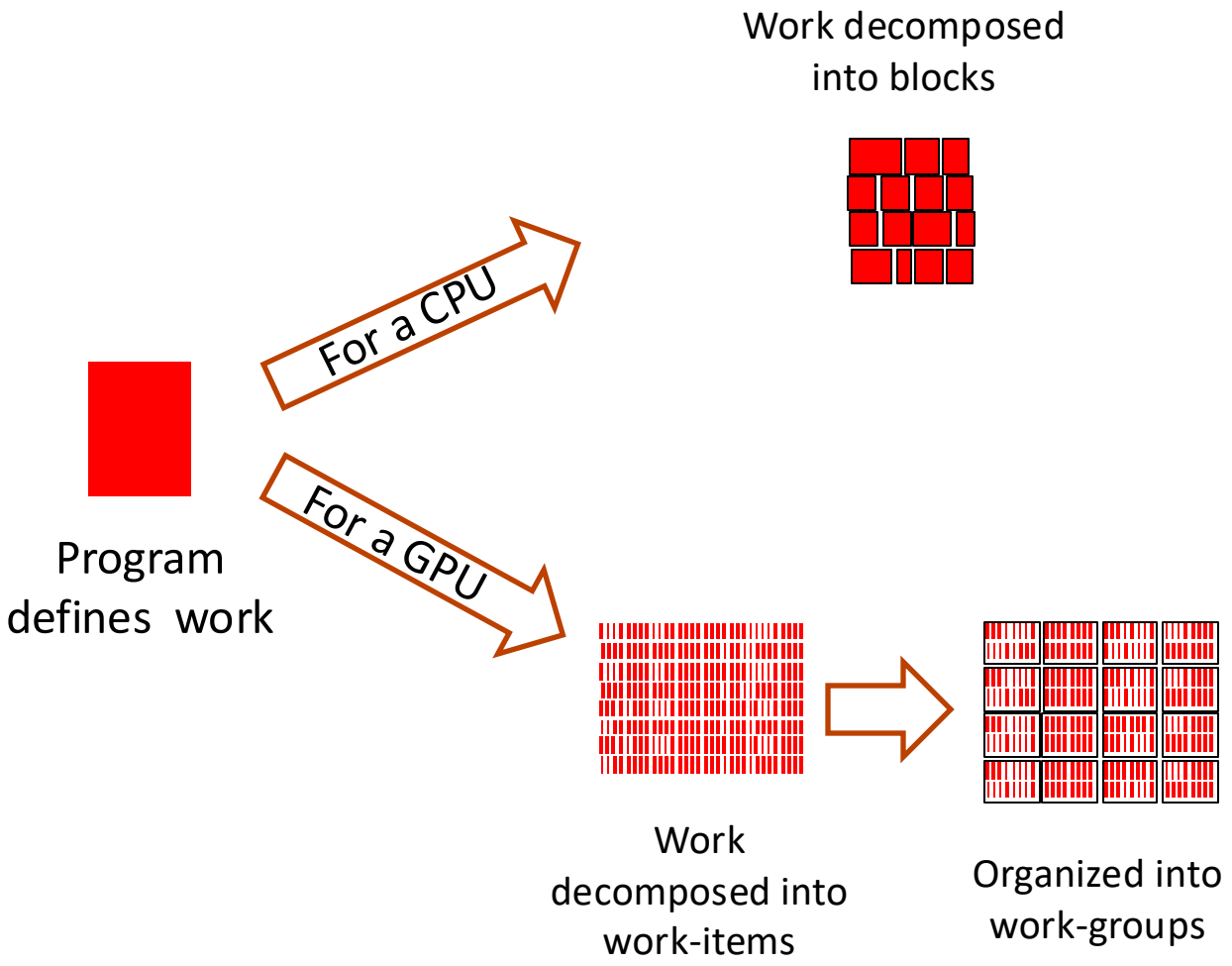


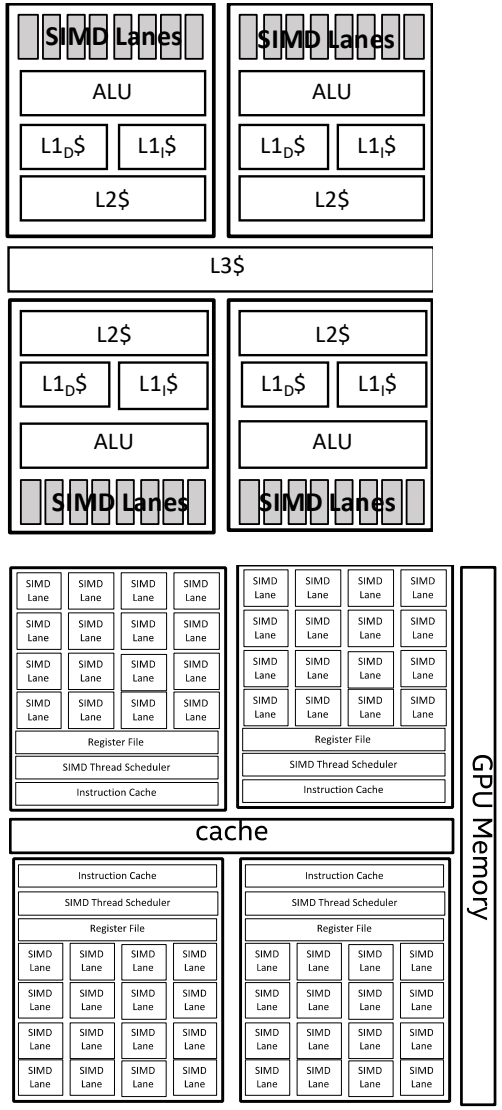
```
int main()
{
     float a[N], b[N], c[N], res[N];
     int err=0;

   // fill the arrays
   #pragma omp parallel for
   for (int i=0; i<N; i++){
      a[i] = (float)i;
      b[i] = 2.0*(float)i;
      c[i] = 0.0;
      res[i] = i + 2*i;
   }

   // add two vectors
   #pragma omp target
   #pragma omp loop
   for (int i=0; i<N; i++){
      c[i] = a[i] + b[i];
   }

   // test results
   #pragma omp parallel for reduction(+:err)
   for(int i=0;i<N;i++){
      float val = c[i] - res[i];
      val = val*val;
      if(val>TOL) err++;
   }
   printf("vectors added with %d errors\n", err);
   return 0;
}
```

Notice how we mix CPU and GPU parallelism in a single program. OpenMP is unique in supporting this style of hybrid programming!

Let’s compare/contrast concurrency on a CPU and a GPU

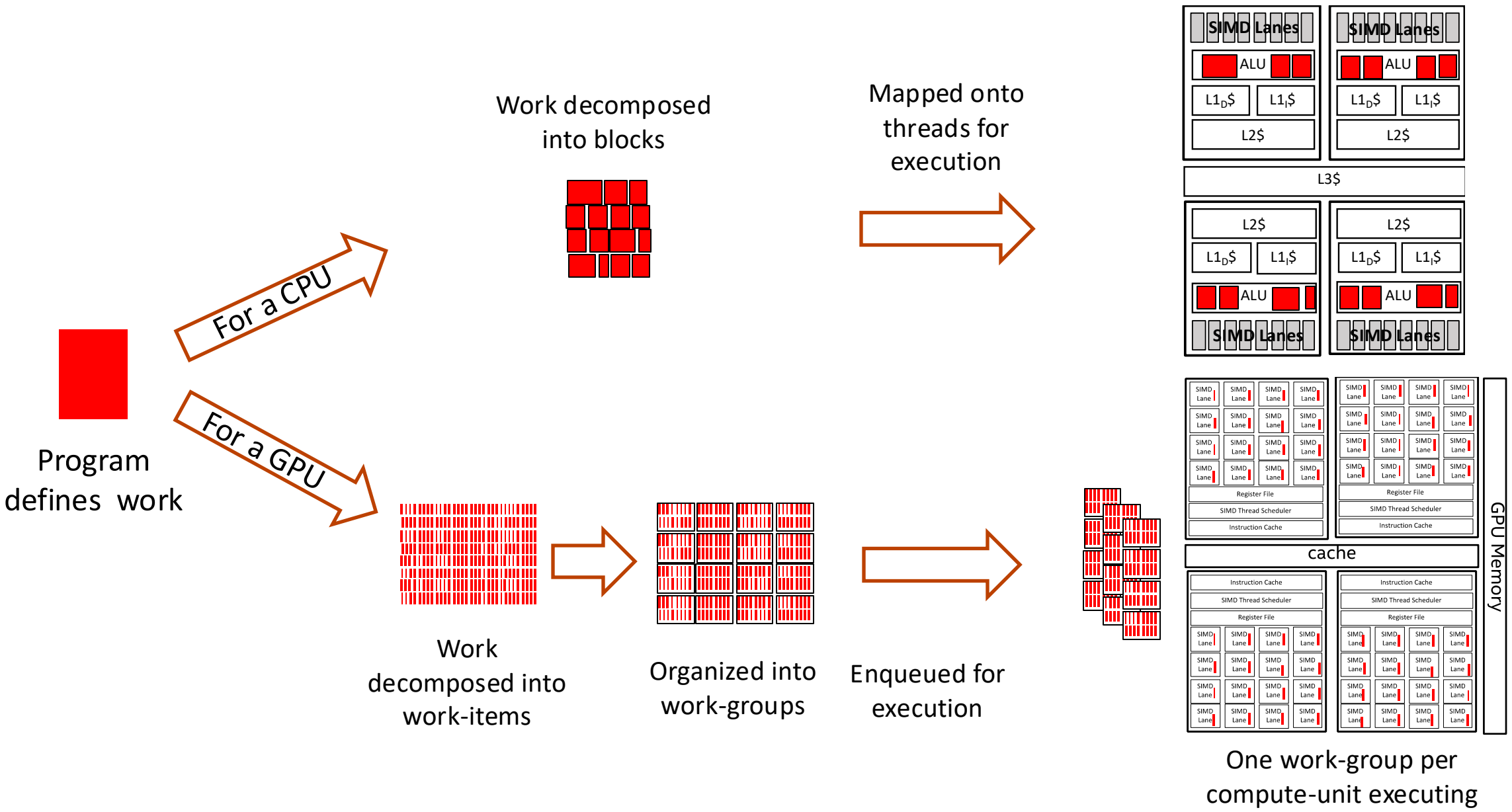
Executing a program on CPUs and GPUs



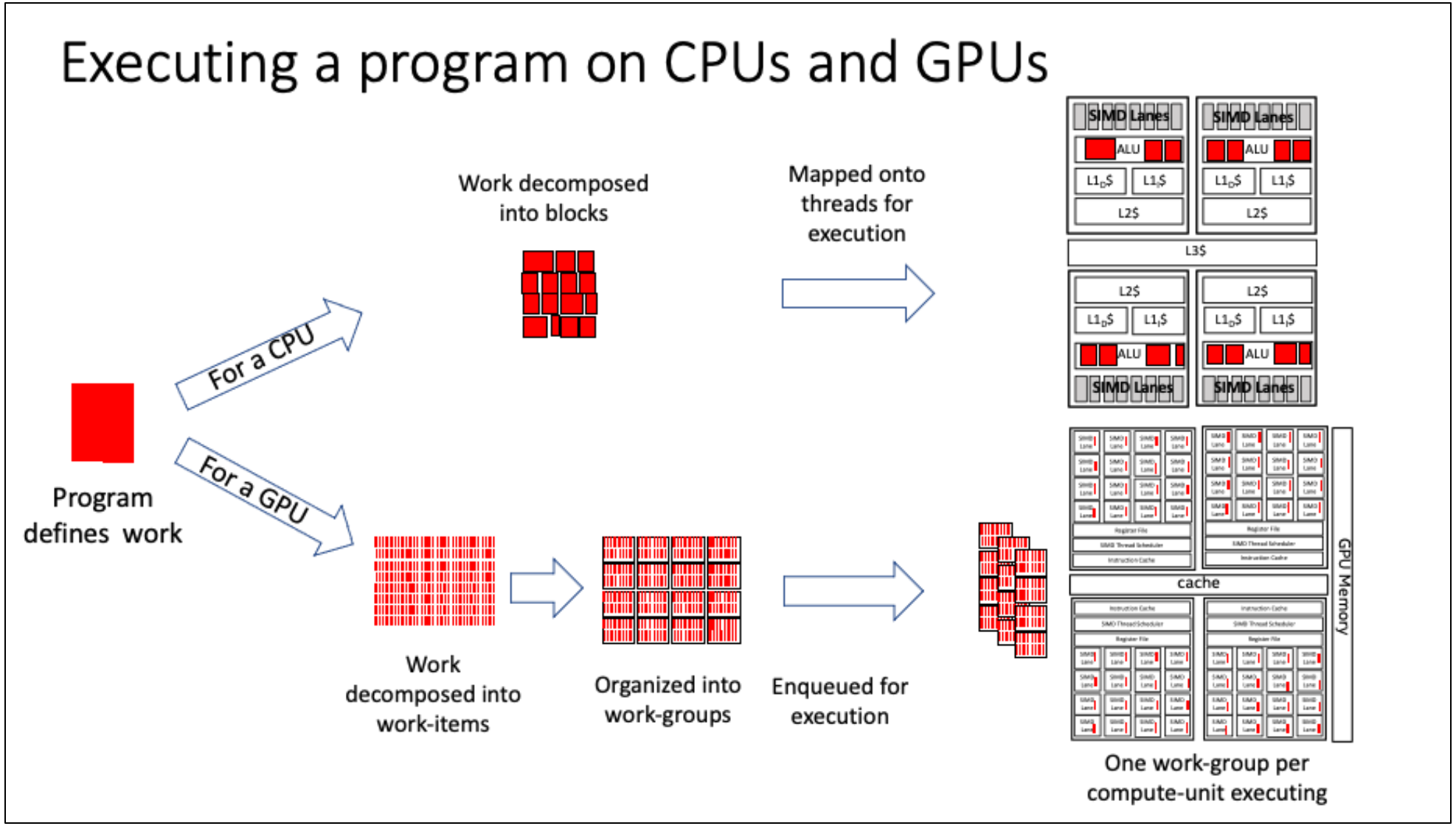


One work-group per compute-unit executing

Executing a program on CPUs and GPUs



CPU/GPU execution models



For a CPU, the threads are all active and able to make forward progress.

Latency sensitive algorithms

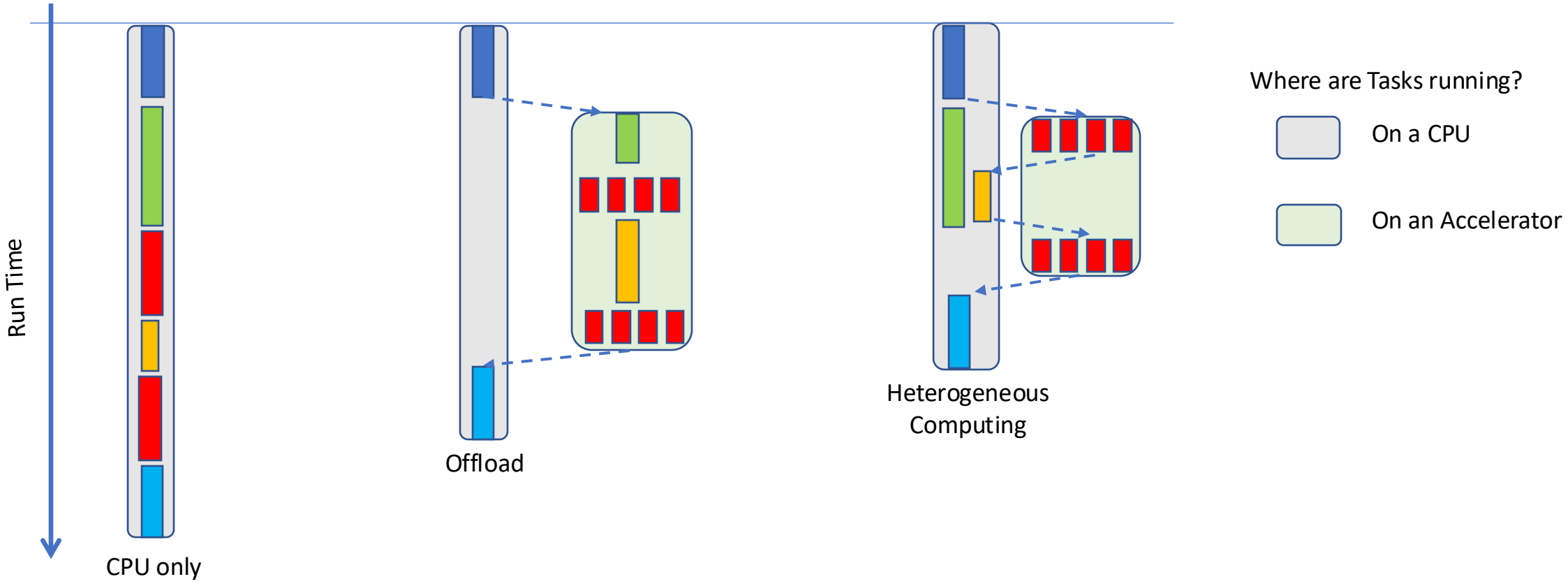
For a GPU, any given work-group might be in the queue waiting to execute.

Throughput optimized algorithms

No single processor is best at everything

- The idea that you should move everything to the GPU makes no sense

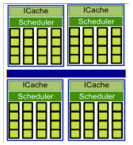
- **Heterogeneous Computing**: Run sub-problems in parallel on the hardware best suited to them.



Implicit data movement covers a small subset of the cases you need in a real program.

To be more general … we need to manage data movement explicitly

The need for explicit data movement



- Fully defined arrays on the stack and scalars move to the GPU by default

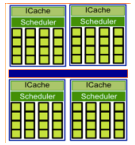
```
int main(void) {
   int N = 1024;
   double A[N], B[N];
   #pragma omp target teams loop
   for (int ii = 0; ii < N; ++ii)
         A[ii] = A[ii] + B[ii];
}
```

- Arrays A and B move between the host and the device on entry and at exit to the target construct.
- The scalar N only moves once … onto the device

- The default rules state that scalars are mapped onto the device.
- That means the pointers A and B as well as N exist on the device
- But the data A and B points to is NOT copied to the device.
- For those, we need to explicitly map them between the host and the device.

```
int main(void) {
   int N = 1024;
   double *A = (double*)malloc(sizeof(double)*N);
   double *B = (double*)malloc(sizeof(double)*N);
   #pragma omp target teams loop
   for (int ii = 0; ii < N; ++ii)
         A[ii] = A[ii] + B[ii];
}
```

Example: vector add with dynamic memory on GPU



The map() clause creates an association between an address on the host and an address in the GPU's memory. **map()** copies to the device (to) at the beginning, back to the host at the end (**from**), or both (**tofrom**).

```
int main()
{
    float *a   = malloc(sizeof(float) * N);
    float *b   = malloc(sizeof(float) * N);
    float *c   = malloc(sizeof(float) * N);
    int err=0;

   // fill the arrays <<<code not shown>>>>

   // add two vectors
   #pragma omp target map(to: a[0:N],b[0:N]) map (tofrom: c[0:N])
   #pragma omp loop
   for (int i=0; i<N; i++){
      c[i] = a[i] + b[i];
   }

}
```

Array Notation:
N elements starting
at offset 0

Copy to the device,
but not backl

Copy to the device
at beginning of the
target construct,
and back to the host
and the end.

Going beyond simple vector addition …

Using OpenMP for GPU application programming … the heat diffusion problem

5-point stencil: the heat program

- The heat equation models changes in temperature over time.

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0$$

- We'll solve this numerically on a computer using an explicit **finite difference** discretisation.
- $u = u(t, x, y)$ is a function of space and time.
- Partial differentials are approximated using diamond difference formulae:

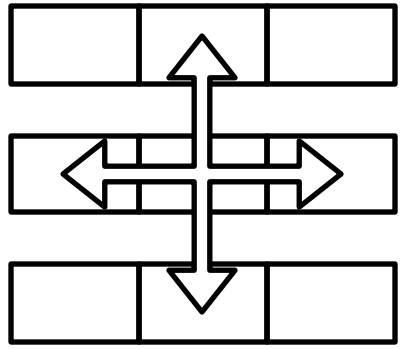
$$\frac{\partial u}{\partial t} \approx \frac{u(t+1, x, y) - u(t, x, y)}{dt}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(t, x+1, y) - 2u(t, x, y) + u(t, x-1, y)}{dx^2}$$

  - Forward finite difference in time, central finite difference in space.

5-point stencil: the heat program

- Given an initial value of u, and any boundary conditions, we can calculate the value of u at time t+1 given the value at time t.
- Each update requires values from the north, south, east and west neighbours only:



- Computation is essentially a weighted average of each cell and its neighbouring cells.
- If on a boundary, look up a boundary condition instead.

Serial CPU code

Heat diffusion problem: 5-point stencil code

```
const double r = alpha * dt / (dx * dx);
 const double r2 = 1.0 - 4.0*r;

 // malloc and initialize u_tmp and u (code not shown)

 for (int t = 0; t < nsteps; ++t) {

    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        u_tmp[i+j*n] =  r2 * u[i+j*n]          +
            r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
            r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
            r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
            r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
      }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
 }
}
```

Loop over time steps

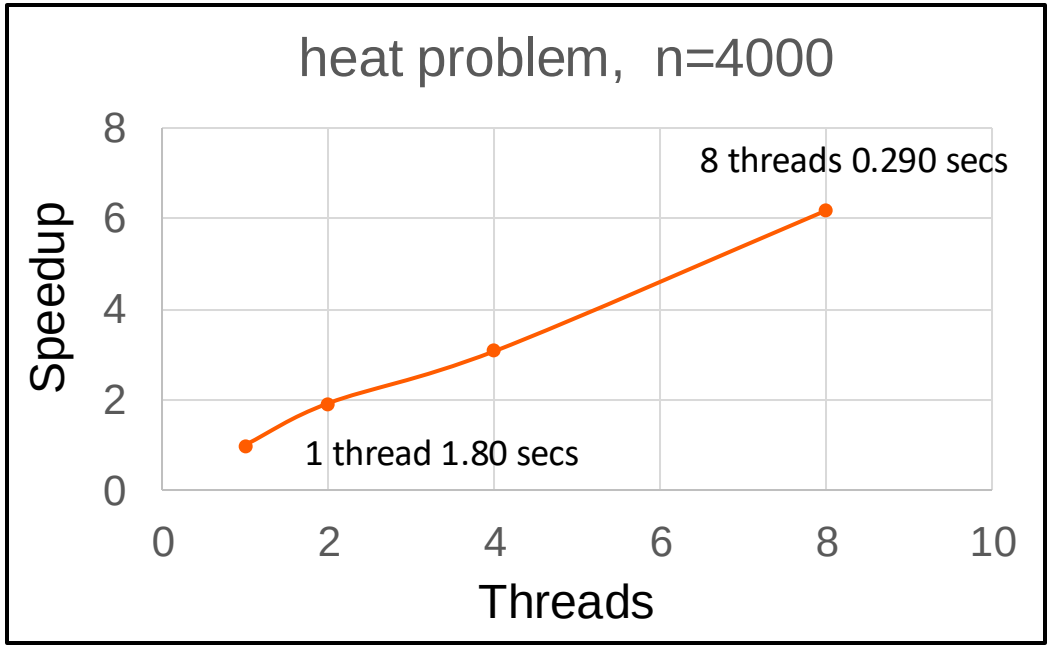
Loop over NxN spatial domain

Update the 5-point stencil. Boundary conditions on the edges of the domain are fixed at zero.

Parallel CPU code, n=4000

Heat diffusion problem: 5-point stencil code

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {

    #pragma omp parallel for collapse(2)
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        u_tmp[i+j*n] =  r2 * u[i+j*n]           +
            r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
            r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
            r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
            r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
      }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
  }
}
```



Intel® Xeon™ Gold 5218 @ 2.3 Ghz, 8 cores. Nvidia HPC Toolkit compiler
nvc –fast –fopenmp heat.c

Heat diffusion problem: 5-point stencil code

Parallel GPU code, n=4000

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {
   #pragma omp target map(tofrom: u[0:n*n]) map(to:u_tmp[0:n*n])
   #pragma omp loop
   for (int i = 0; i < n; ++i) {
     for (int j = 0; j < n; ++j) {
       u_tmp[i+j*n] =  r2 * u[i+j*n]          +
           r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
           r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
           r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
           r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
     }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
  }
}
```

When you map pointers between the host and the device, OpenMP remembers the address.

Swapped addresses on the hosts swaps addresses on the device

GPU Solver time = 1.40 secs

This isn't much better than the runtime for a single CPU (1.8 secs) and worse than 8 cores on a CPU (0.29 secs).

Why is the performance so bad?

Nvidia HPC Toolkit compiler
nvc -fast -mp=gpu -gpu=cc75 heat.c

Parallel GPU code, n=4000

Heat diffusion problem: 5-point stencil code

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {
   #pragma omp target map(tofrom: u[0:n*n]) map(tofrom:u_tmp[0:n*n])
   #pragma omp loop
   for (int i = 0; i < n; ++i) {
     for (int j = 0; j < n; ++j) {
       u_tmp[i+j*n] =  r2 * u[i+j*n]          +
           r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
           r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
           r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
           r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
     }
   }
   // Pointer swap to get ready for next step
   tmp = u;
   u = u_tmp;
   u_tmp = tmp;
}
}
```

With a runtime of 1.4 secs (worse than the CPU time) we see that Data Movement dominates performance.

At the beginning of each iteration, copy $(2*N^2)*sizeof(TYPE)$ bytes to the device

We need to create a **data region** so on the GPU that is distinct from the target region.

That way, we can keep the data on the device between target constructs

At the end of each iteration, copy $(2*N^2)*sizeof(TYPE)$ bytes from the device

Parallel GPU code, n=4000

Heat diffusion problem: 5-point stencil code

```
  const double r = alpha * dt / (dx * dx);
  const double r2 = 1.0 - 4.0*r;
  // malloc and initialize u_tmp and u (code not shown)
  #pragma omp target data enter map(to: u[0:n*n], u_tmp[0:n*n])

  for (int t = 0; t < nsteps; ++t) {
     #pragma omp target
     #pragma omp loop
     for (int i = 0; i < n; ++i) {
       for (int j = 0; j < n; ++j) {
         u_tmp[i+j*n] =  r2 * u[i+j*n]          +
             r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
             r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
             r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
             r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
       }
      }
      // Pointer swap to get ready for next step
      tmp = u;
      u = u_tmp;
      u_tmp = tmp;
  }
 #pragma omp target data exit map(from: u[0:n*n])
}
```

Create a data region and map indicated data on entry

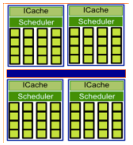
GPU Solver time = **0.057** secs

This is a general principal … if you want performance, you must optimize data movement.

Exit the data region and map indicated data

Nvidia HPC Toolkit compiler
nvc -fast -mp=gpu -gpu=cc75 heat.c

There is MUCH more … beyond what have time to cover



- Do as much as you can with a simple loop construct. It's portable and as compilers improve over time, it will keep up with compiler driven performance improvements.

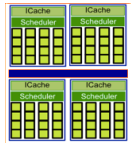
- But sometimes you need more:
  - Control over number of teams in a league and the size of the teams
  - Explicit scheduling of loop iterations onto the the teams
  - Management of data movement across the memory hierarchy: global vs. shared vs. private …
  - Calling optimized math libraries (such as cuBLAS)
  - Multi-device programming
  - Asynchrony

- Ultimately, you may need to master all those advanced features of GPU programming. But start with loop. Start with how data on the host maps onto the device (i.e. the GPU). Master that level of GPU programming before worrying about the complex stuff.

This is the end … well almost the end.

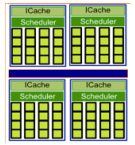
Let’s wrap up with a few high-level comments about the state of GPU programming more generally

SIMT Programming models: it's more than just OpenMP



- CUDA:
  - Released ~2006. Made GPGPU programming "mainstream" and continues to drive innovation in SIMT programming.
    - Downside: proprietary to NVIDIA
- OpenCL:
  - Open Standard for SIMIT programming created by Apple, Intel, NVIDIA, AMD, and others. 1st release in 2009.
  - Supports CPUs, GPUs, FPGAs, and DSP chips. The leading cross platform SIMT model.
    - Downside: extreme portability means verbose API. Painfully low level especially for the host-program.
- Sycl:
  - C++ abstraction layer implements SIMT model with kernels as lambdas. Closely aligned with OpenCL. 1st release 2014
    - Downside: Cross platform implementations only emerging recently.
- Directive driven programming models:
  - **OpenACC**: they split from an OpenMP working group to create a competing directive driven API emphasizing descriptive (rather than prescriptive) semantics.
    - Downside: NOT an Open Standard. Controlled by NVIDIA.
  - **OpenMP**: Mixes multithreading and SIMT. Semantics are prescriptive which makes it more verbose. A truly Open standard supported by all the key GPU players.
    - Downside: Poor compiler support so far … but that will change over the next couple years.

Vector addition with CUDA



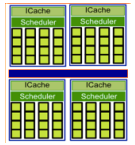
```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
// ... allocate other arrays (b and c), fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

CUDA kernel as function

Unified shared memory … allocate on host, visible on device too

Enqueue the kernel to execute on the Grid

Vector addition with SYCL



```
// Compute sum of length-N vectors: C = A + B
#include <CL/sycl.hpp>

int main () {
    int N = ... ;
    float *a, *b, *c;
    sycl::queue q;
    *a = (float *)sycl::malloc_shared(N * sizeof(float), q);
  // ... allocate other arrays (b and c), fill with data

        q.parallel_for(sycl::range<1>{N},
                [=](sycl::id<1> i) {
                  c[i] = a[i] + b[i];
                });
         q.wait();
}
```

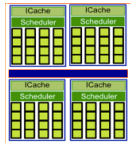
Create a queue for SYCL commands

Unified shared memory … allocate on host, visible on device too

Kernel as a C++ Lambda function

[=] means capture external variables by value.

Vector addition with OpenMP



- Let's add two vectors together …. C = A + B

Host waits here until the kernel is done. Then the output array c is copied back to the host.

```
void vadd(int n,
          const float *a,
          const float *b,
          float *c)
{
  int i;
 #pragma target teams loop map(to:a[0:N],b[0:n]) map(tofrom:c[0:N])
  for (i=0; i<n; i++)
    c[i] = a[i] + b[i];
}
int main(){
float *a, *b, *c;  int n = 10000;
// allocate and fill a and b

    vadd(n, a, b, c);

}
```

Vector addition with OpenACC

- Let's add two vectors together …. C = A + B

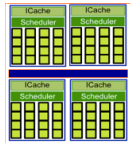
```
void vadd(int n,
          const float *a,
          const float *b,
          float *restrict c)
{
  int i;
 #pragma acc parallel loop
  for (i=0; i<n; i++)
    c[i] = a[i] + b[i];
}
int main(){
float *a, *b, *c;  int n = 10000;
// allocate and fill a and b

    vadd(n, a, b, c);

}
```

Assure the compiler that c is not aliased with other pointers

Turn the loop into a kernel, move data to a device, and launch the kernel.

Host waits here until the kernel is done. Then the output array c is copied back to the host.

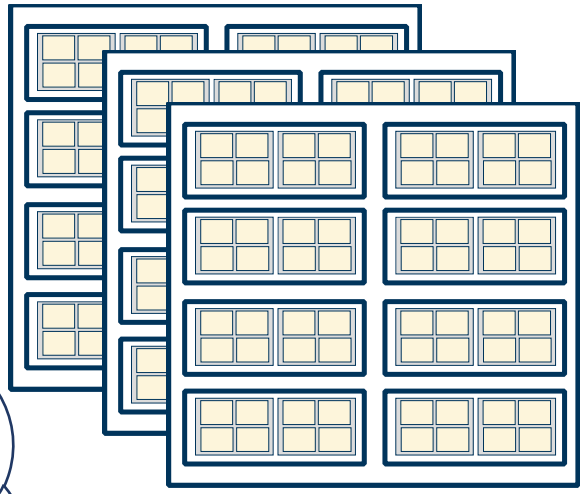
Why so many ways to do the same thing?



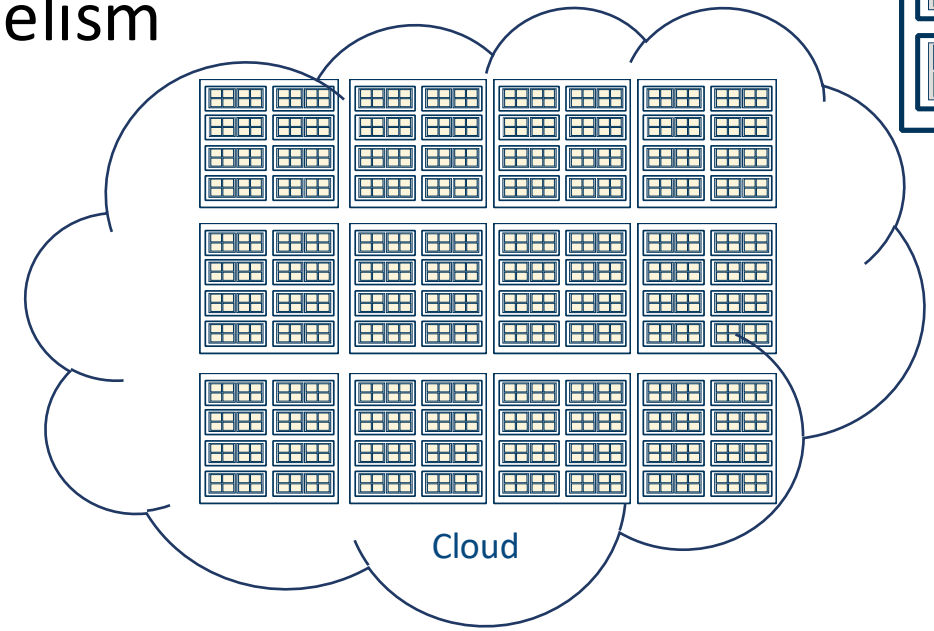
- The parallel programming model people have failed you …
  - It's more fun to create something new in your own closed-community that work across vendors to create a portable API
- The hardware vendors have failed you …
  - Don't you love my "walled garden"? It's so nice here, programmers, just don't even think of going to some other platform since your code is not portable.
- The standards community has failed you …
  - Standards are great, but they move too slow. OpenACC stabbed OpenMP in the back and I'm pissed, but their comments at the time were spot-on (OpenMP was moving so slow … they just couldn't wait).
- The applications community failed themselves …
  - If you don't commit to a standard and use "the next cool thing" you end up with the diversity of overlapping options we have today. Think about what happened with OpenMP and MPI.

Outline

- Semiconductor Devices and the Need for Parallelism
- Parallelism: Hardware, key abstractions, and programming
  - The CPU: Adventures in Multithreading
  - The Vector (SIMD) unit: Lock-step parallelism in action
  - The GPU: Data Parallelism is your friend
- Scaling to massive parallelism

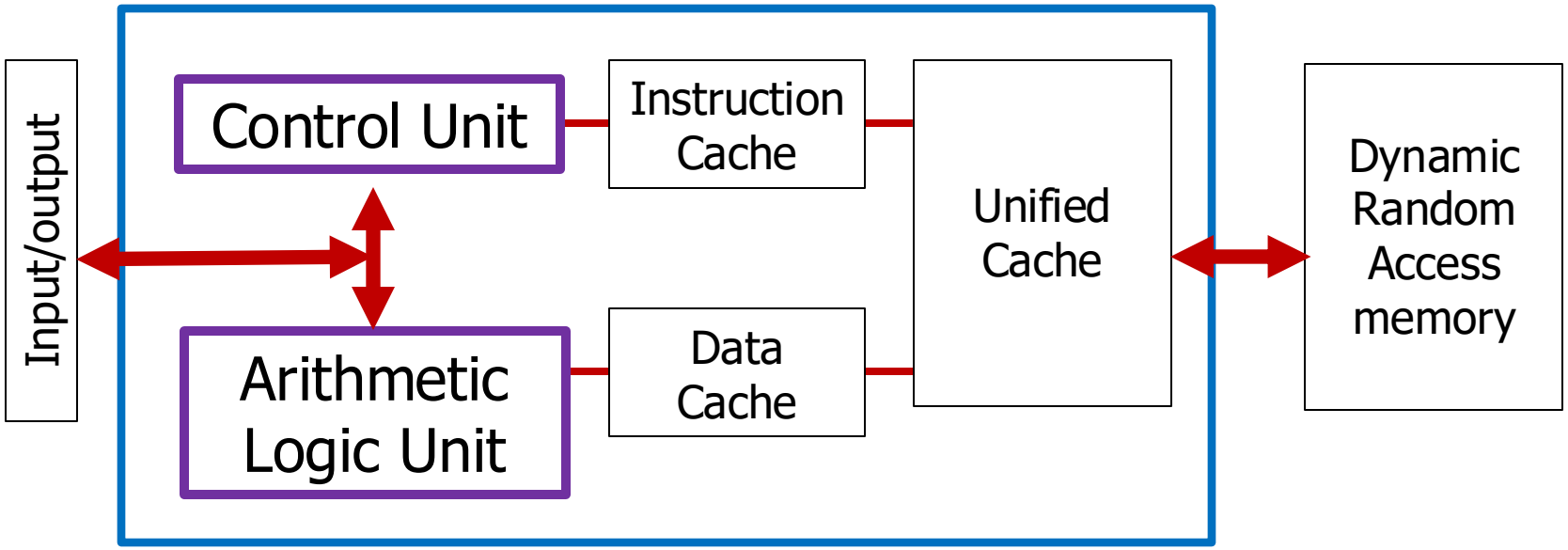


Cluster



How do we build Supercomputers today?

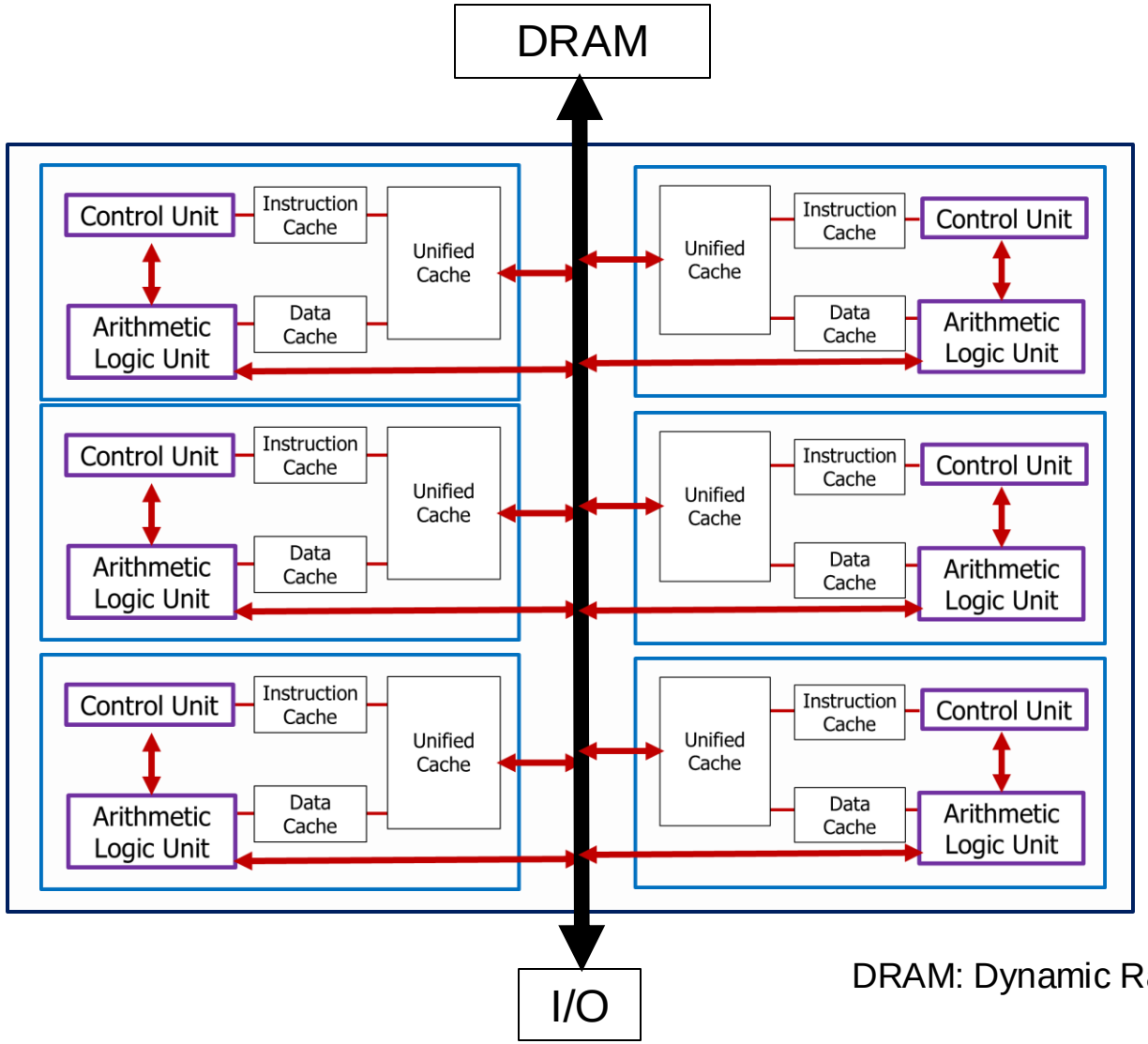
- Take advantage of Moore's law and the economy of scale.
- Start with a simple von-Neumann computer..



6 cores

How do we build Supercomputers today?

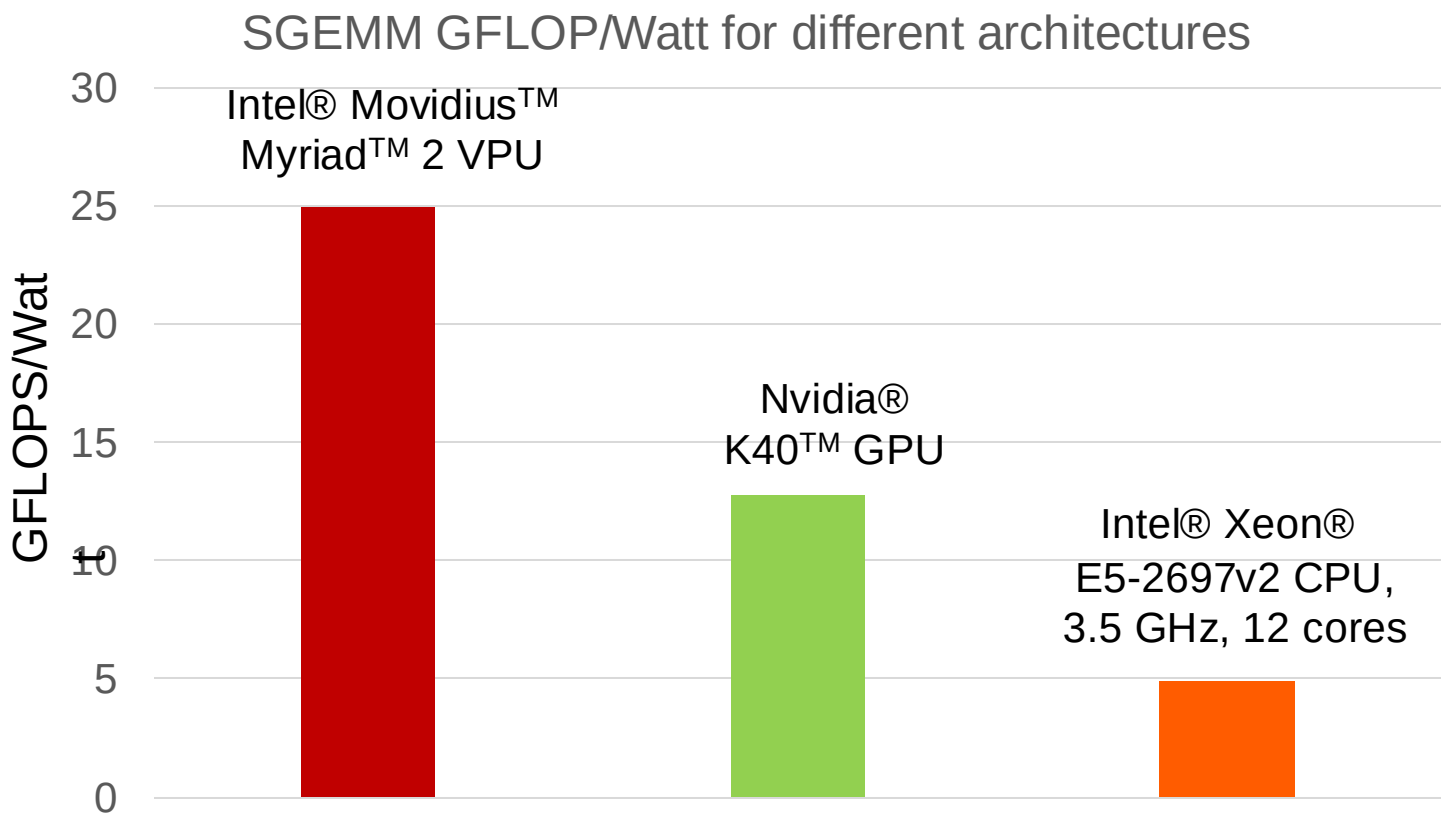
- Put a whole bunch of simple von-Neumann computers onto a single silicon chip (a multi-core chip).



DRAM: Dynamic Random Access Memory

If you care about power, the world is heterogeneous?

Specialized processors doing operations suited to their architecture are more efficient than general purpose processors.



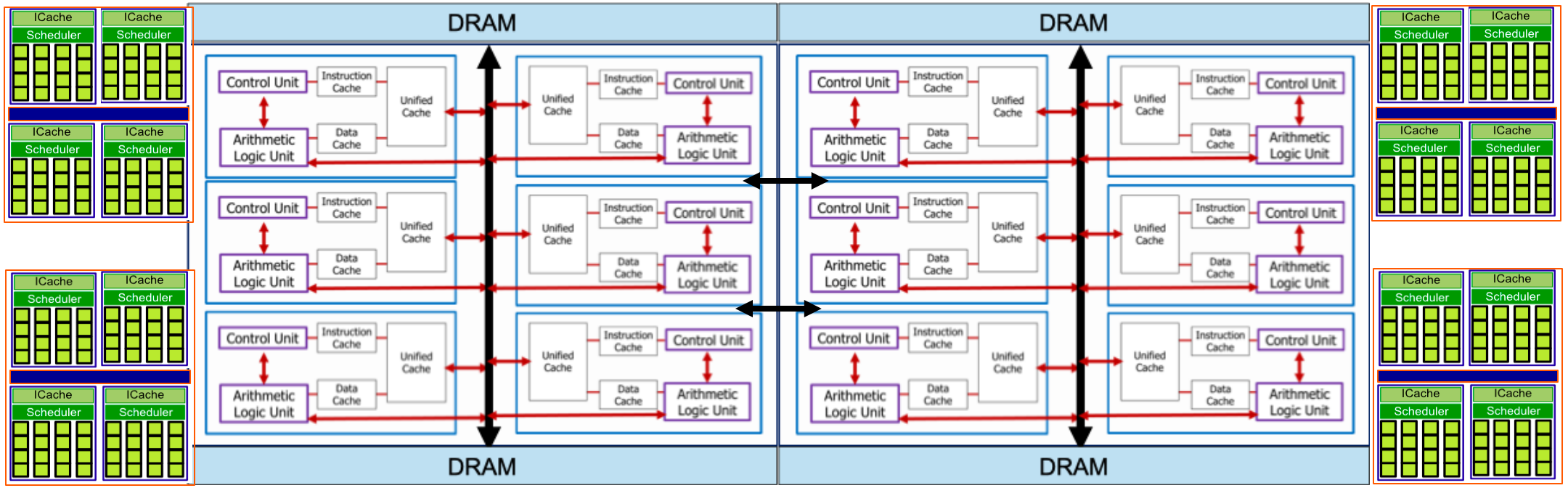
Hence, future systems will be increasingly heterogeneous … GPUs, CPUs, FPGAs, and a wide range of accelerators

Source: Suyash Bakshi and Lennart Johnsson, “A Highly Efficient SGEMM Implementation using DMA on the Intel/Movidius Myriad-2. IEEE International Symposium on Computer Architecture and High Performance Computing, 2020

6 cores

How do we build Supercomputers today?

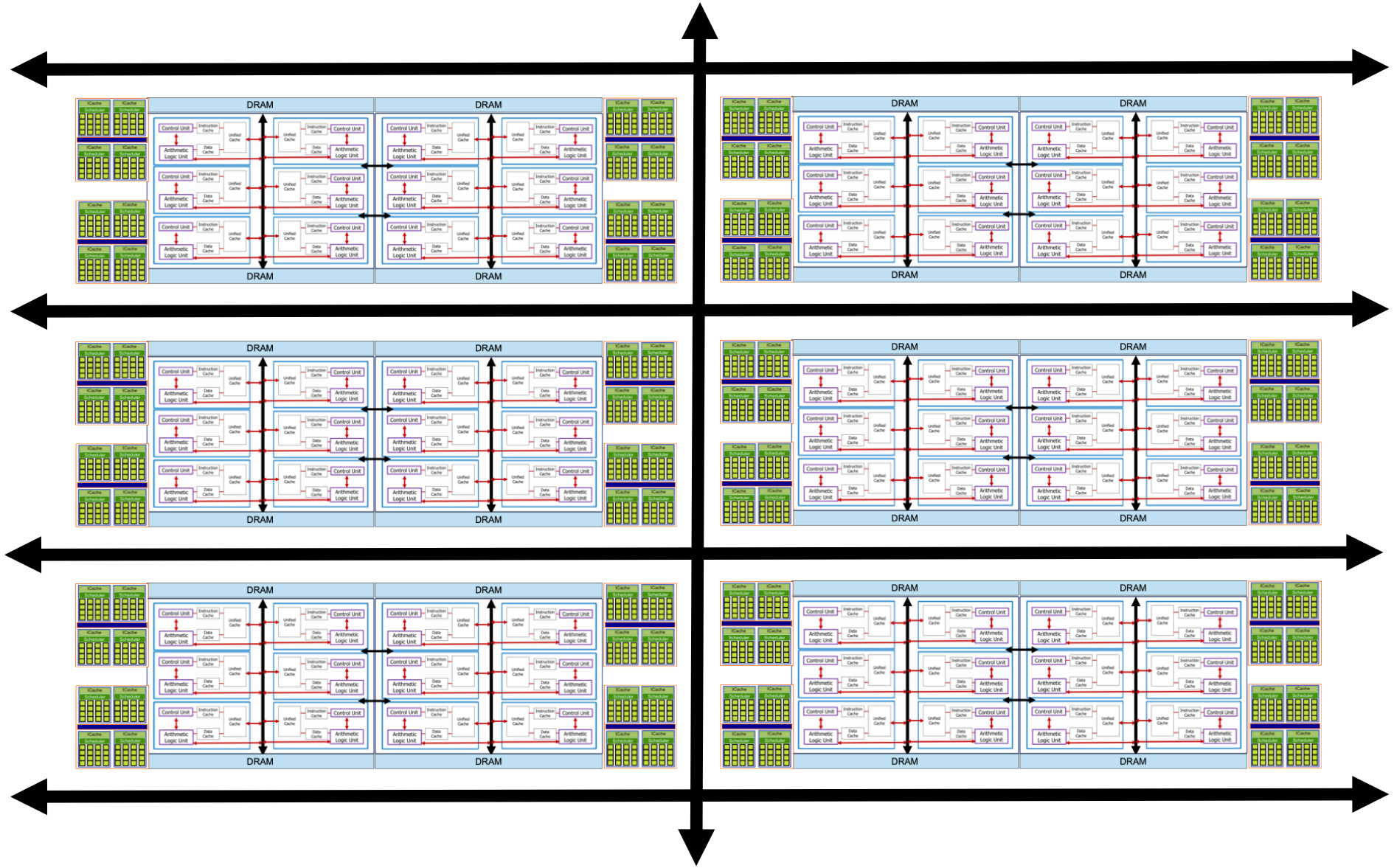
- So we combine multicore CPUs with several GPUs to build a node



72 cores

How do we build Supercomputers today?

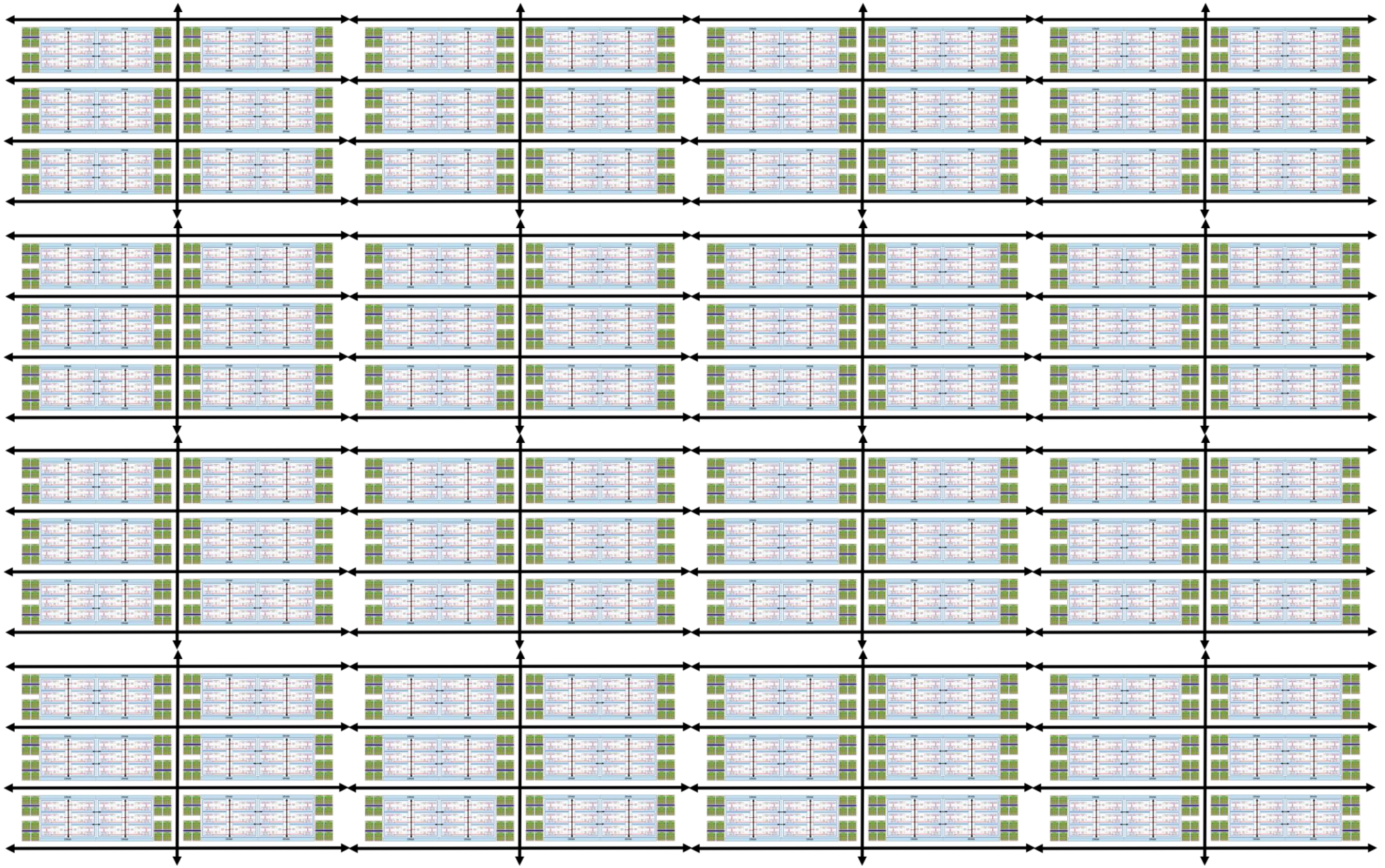
- Combine nodes to build a cluster



1152 cores

How do we build Supercomputers today?

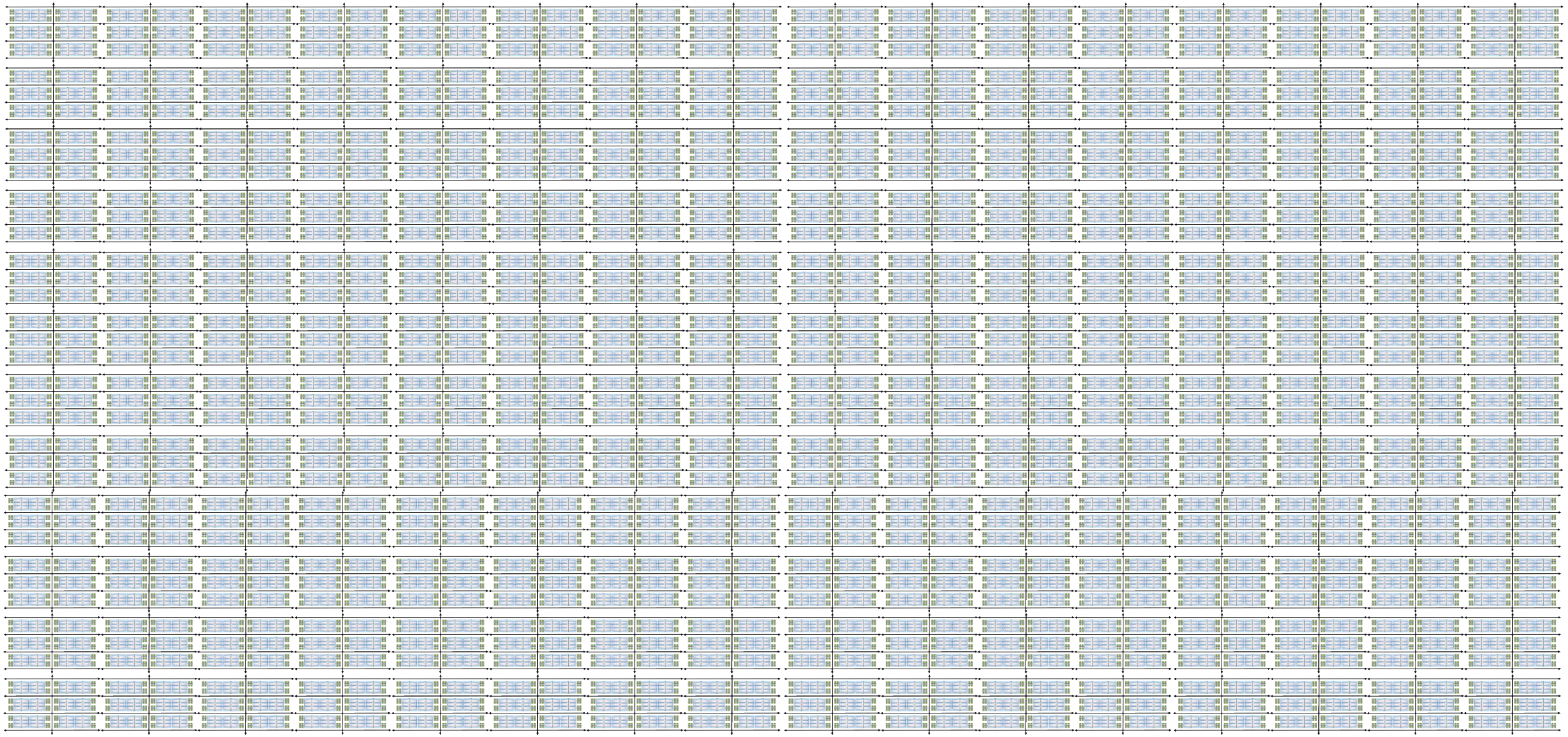
- Do this a whole bunch



10368 cores

How do we build Supercomputers today?

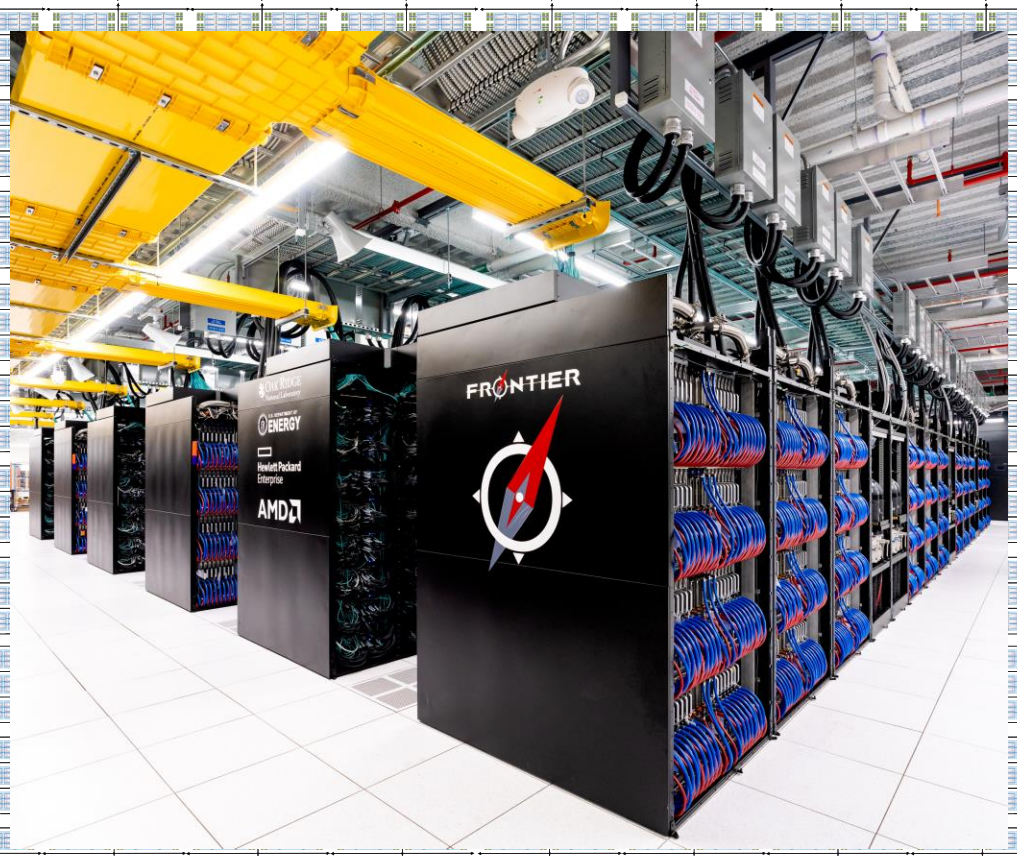
- Keep going until you run out of money (or can't afford the electricity bill)



The world’s fastest supercomputer

- More details on the Frontier EFLOP supercomputer

- Frontier Cray HP Enterprise system,
- 9,472 AMD Epyc 7713 64-core CPUs @ 2 Ghz
- 37,888 AMD Instinct M1250X GPUs,
- HPE Slingshot-11 network,
- Cost: $600 Million,
- Oak Ridge National Laboratory, USA,
- 22.8 MW



Remember Amdahl's Law

- The maximum speedup for serial faction α is: $S = \frac{1}{\alpha}$ ← Amdahl's Law

- Consider Frontier with:

  9,472 CPUs * 64 CPU cores/CPU

  37888 GPUs * 220 compute units/GPU

  8941568 "effective cores"

- To scale to the full system for a fixed size problem (strong scaling) you need a sequential fraction of $1.2 * 10^{-7}$

- Of course, once the benchmarks are done, the machine runs many small jobs behind a batch queue … but if that's the plan, the more cost effective solution is to build multiple smaller machines (the network on a massive system can cost half the price of the machine).

Exascale machines are for the most part, scaling to absurd levels to set arbitrary benchmark records and win "bragging rights".

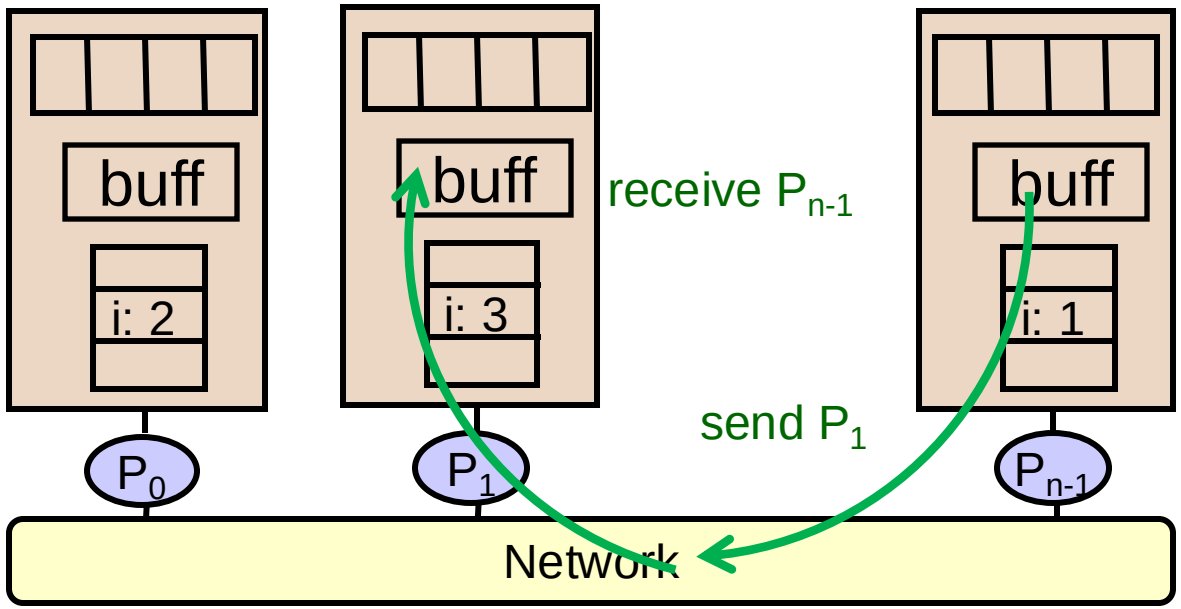
How do we write code for these machines?

They do not have shared memory between nodes in the network.

Programming Model for distributed memory systems

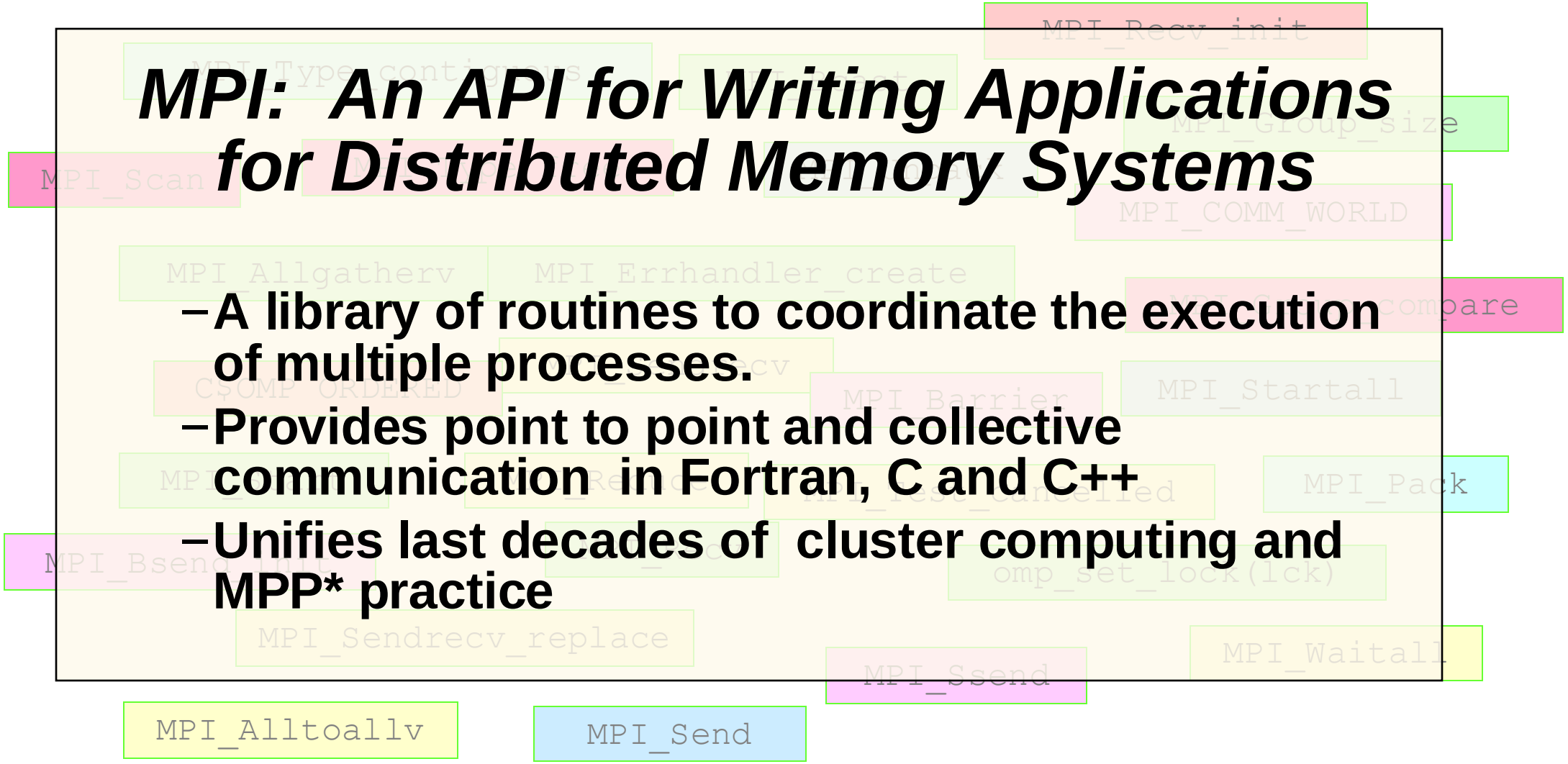
- Programs execute as a collection of processes.
  - Number of processes usually fixed at program startup time
  - Local address space per node -- **NO physically shared memory**.
  - **Logically** shared data is partitioned over local processes.

- Processes communicate by messages … explicit send/receive pairs
  - Synchronization is implicit by communication events.
  - MPI (Message Passing Interface) is the most commonly used API



A collection of n MPI processes (P_0 to P_{n-1}) running on n nodes

MPI, the Message Passing Interface



MPI: An API for Writing Applications for Distributed Memory Systems

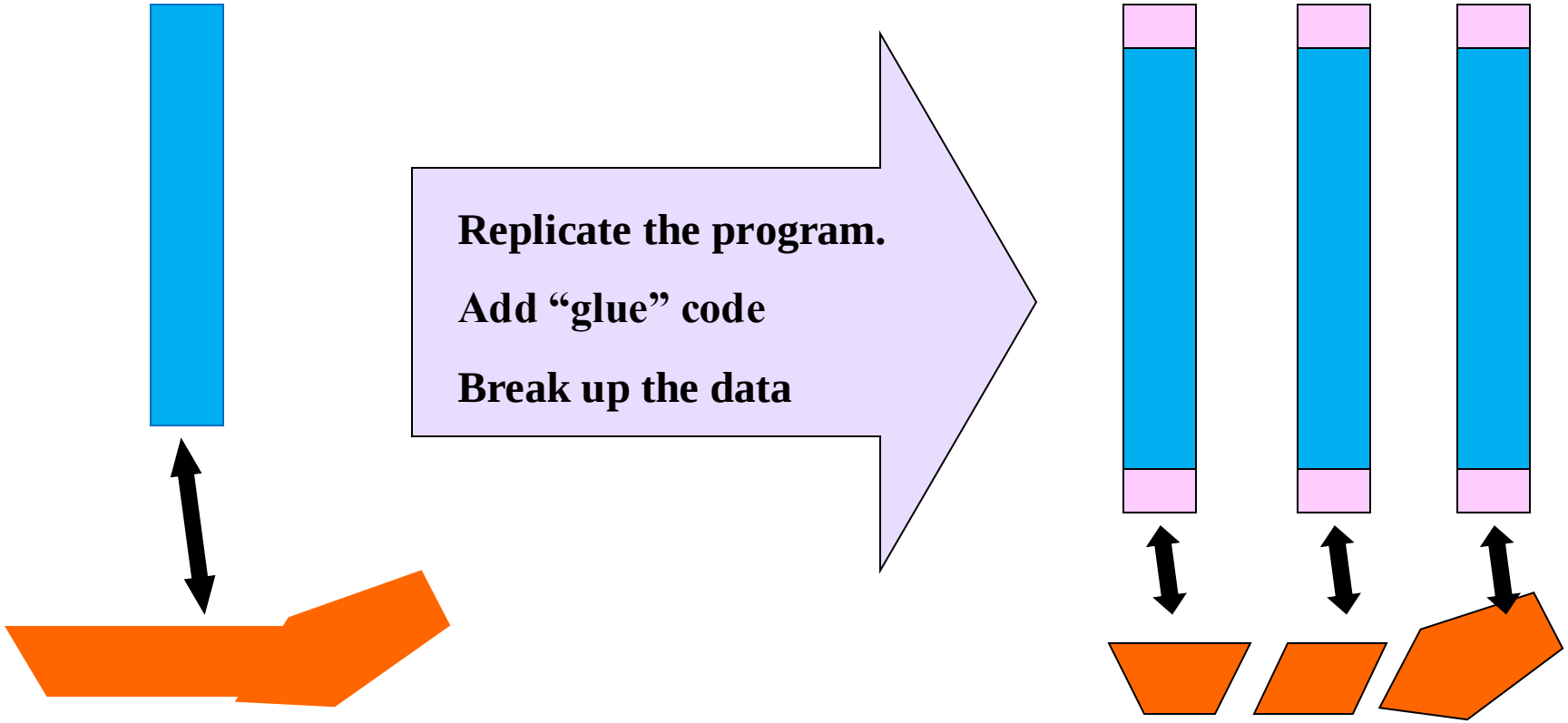
- A library of routines to coordinate the execution of multiple processes.
- Provides point to point and collective communication in Fortran, C and C++
- Unifies last decades of cluster computing and MPP* practice

*MPP: Massively Parallel Processing. Clusters use “off the shelf” components. MPP systems include custom system integration.

How do people use MPI?
The SPMD Design Pattern

- A replicated single program working on a decomposed data set.
- Use Node ID (rank) and number of nodes to split up work between processes
- Coordinate processes by passing messages.

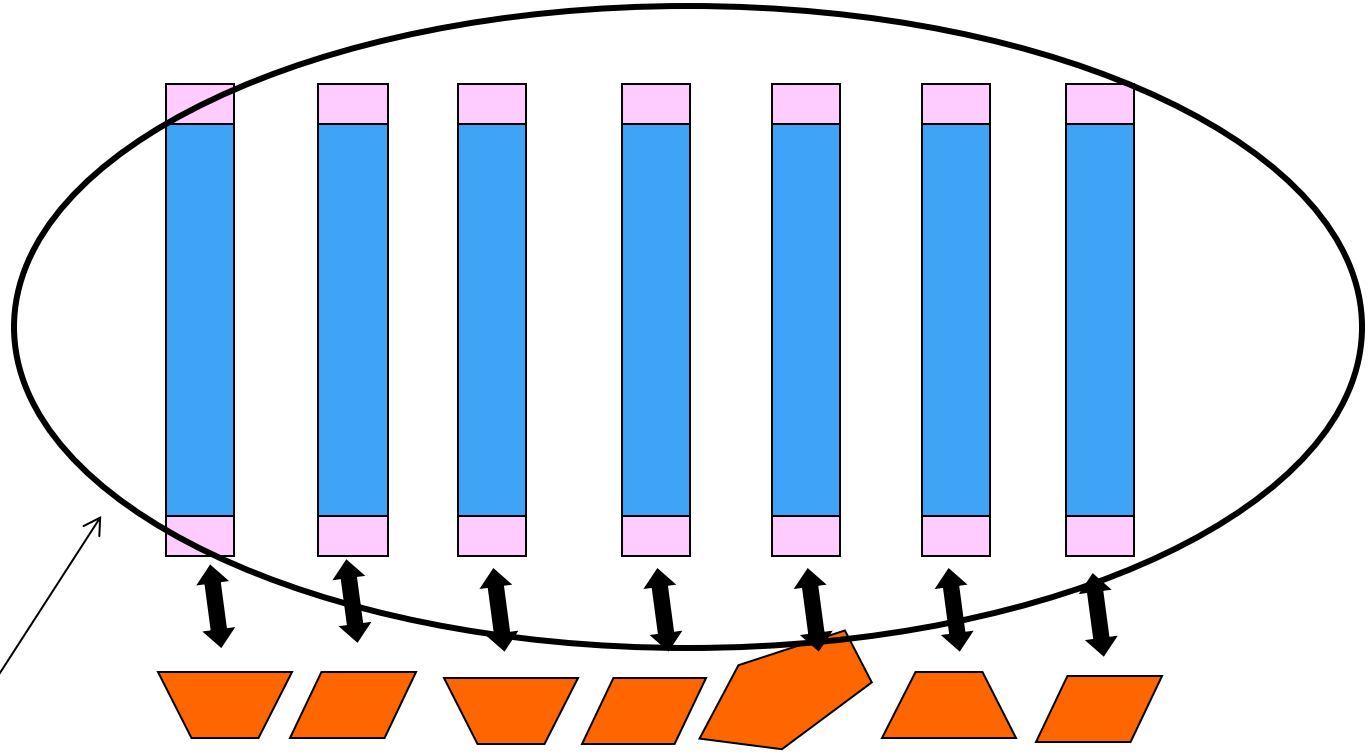
A sequential program (blue) working on a data set (orange)



Glue code is what I call the code to initialize/finalize MPI and establish the communication context, rank, and process count.

An MPI program at runtime

- Typically, when you run an MPI program, multiple processes all running the same program are launched … working on their own block of data.



The collection of processes involved in a computation is called “a **process group**”

MPI Hello World Program

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
   int rank, size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf( "Hello from process %d of %d\n",
                                rank, size );
   MPI_Finalize();
   return 0;
}
```

MPI Hello World Program

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
   int rank, size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf( "Hello from process %d of %d\n",
                                rank, size );
   MPI_Finalize();
   return 0;
}
```

`int MPI_Init (int* argc, char* argv[])`

- Initializes the MPI library … called before any other MPI functions.
- agrc and argv are the command line args passed from main()

`int MPI_Finalize (void)`

- Frees memory allocated by the MPI library … close every MPI program with a call to MPI_Finalize

MPI Hello World Program

`int MPI_Comm_size (MPI_Comm comm, int* size)`

- **`MPI_Comm`**, an *opaque data type called a communicator.* *D*efault context: MPI_COMM_WORLD (all processes)
- **`MPI_Comm_size`** returns the number of processes in the process group associated with the communicator

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
   int rank, size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf( "Hello from process %d of %d\n",
                                  rank, size );
   MPI_Finalize();
   return 0;
}
```

Communicators consist of two parts, a **context** and a **process group**.

The communicator lets one control how groups of messages interact.

Communicators support modular SW … i.e. I can give a library module its own communicator and know that it's messages can't collide with messages originating from outside the module

MPI Hello World Program

`int MPI_Comm_rank (MPI_Comm comm, int* rank)`

- **`MPI_Comm`**, an *opaque data type,* **`a`** communicator. Default context: MPI_COMM_WORLD (all processes)
- **`MPI_Comm_rank`** An integer ranging from 0 to "(num of procs)-1"

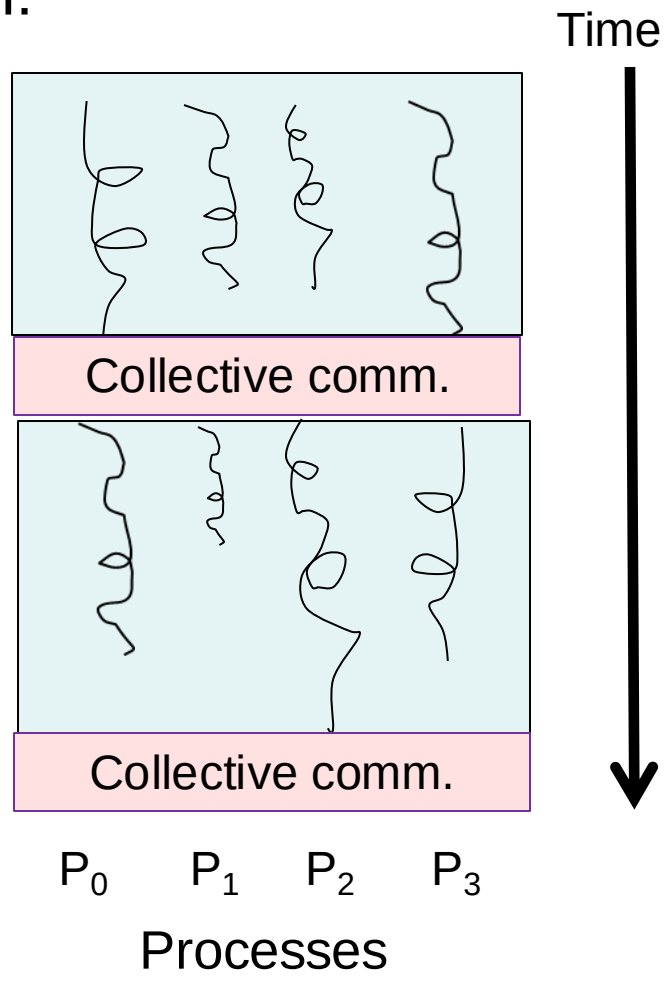
```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
   int rank, size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf( "Hello from process %d of %d\n",
                                rank, size );
   MPI_Finalize();
   return 0;
}
```

Note that other than init() and finalize(), every MPI function has a communicator.

This makes sense .. You need a context and group of processes that the MPI functions impact … and those come from the communicator.

A typical pattern with MPI Programs

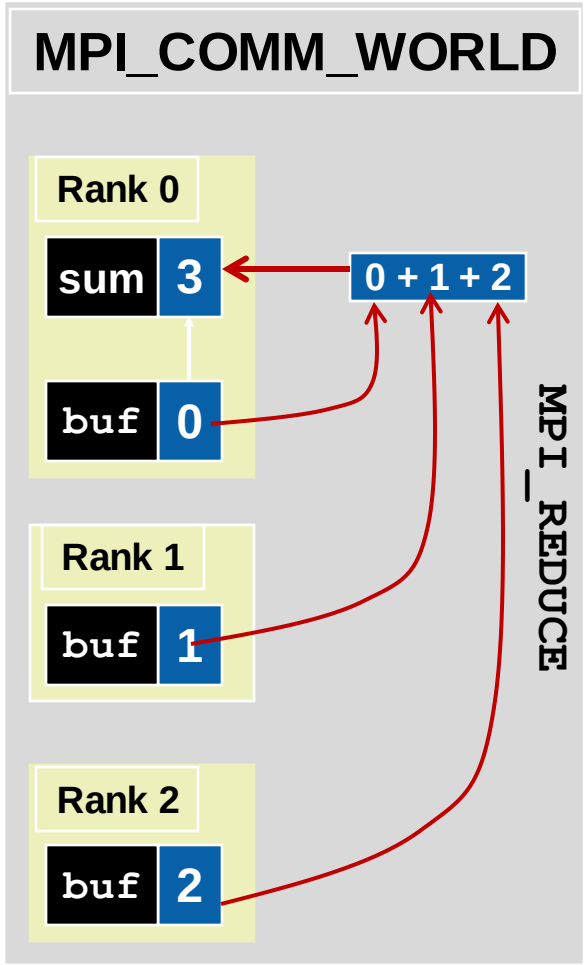
- Many MPI applications directly call few (if any) message passing routines. They use the following very common pattern:
  - Use the Single Program Multiple Data pattern
  - Each process maintains a local view of the global data
  - A problem broken down into phases each of which is composed of two subphases:
    - Compute on local view of data
    - Communicate to update global view on all processes (collective communication).
  - Continue phases until complete



This is a subset or the SPMD pattern sometimes referred to as the Bulk Synchronous pattern.

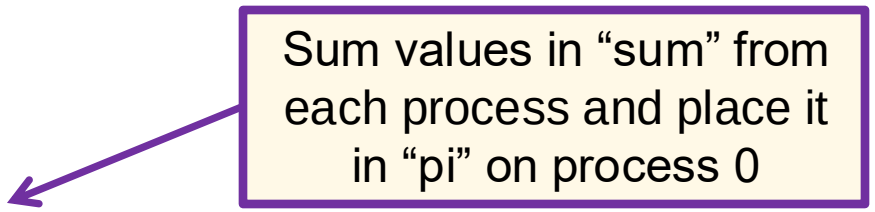
Collective Communication: Reduction

```
#include <mpi.h>

int main(int argc, char* argv[]) {
  int buf, sum, nprocs, myrank;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  sum = 0;
  buf = myrank;

  MPI_Reduce(&buf, &sum, 1, MPI_INT,
             MPI_SUM, 0, MPI_COMM_WORLD);

  MPI_Finalize();
}
```



Pi program in MPI

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        for (i=my_id*my_steps; i<(my_id+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD); ;
}
```



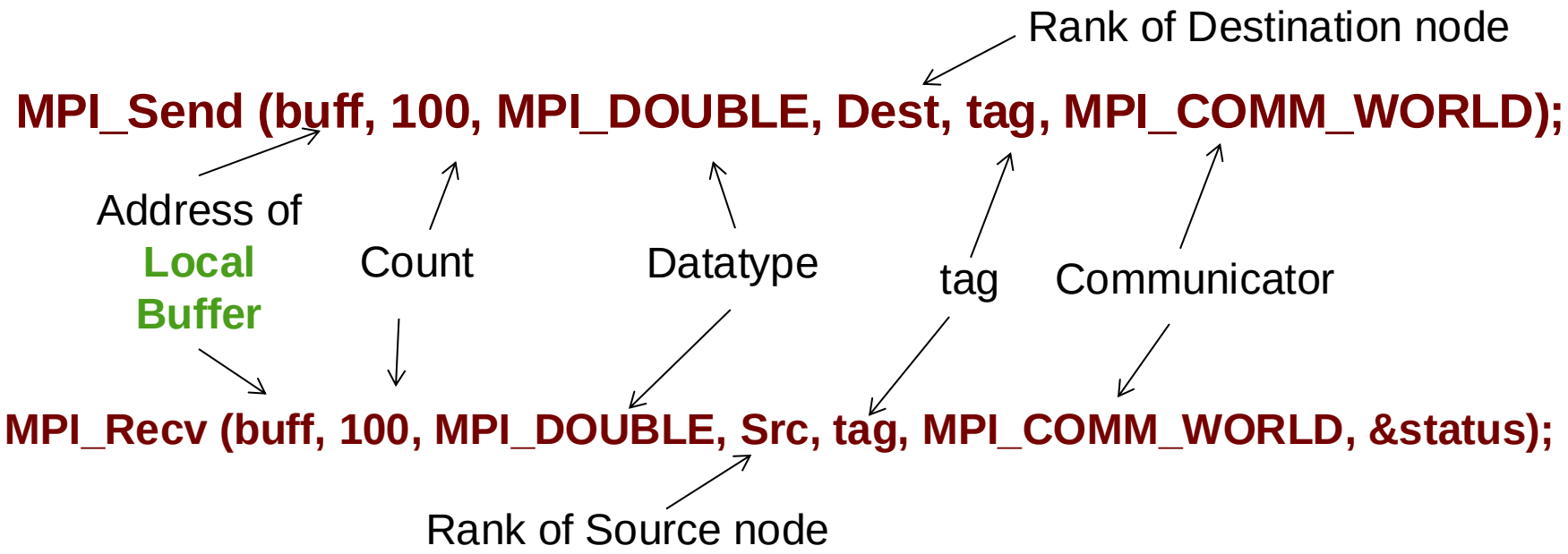
… But it’s called the Message Passing Interface.

Shouldn’t we be sending messages between nodes?

Sending and receiving messages

- Pass a buffer which holds “count” values of MPI_TYPE
- The data in a message to send or receive is described by a triple:
  - **(address, count, datatype)**

- The receiving process identifies messages with the double :
  - **(source, tag)**
- Where:
  - Source is the rank of the sending process
  - Tag: a user-defined int to keep track of different messages from a single source

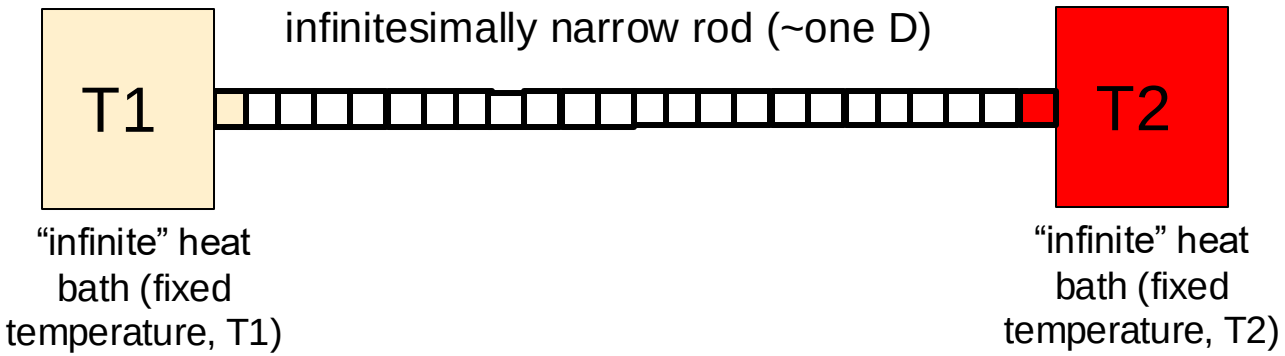


Ping-Pong Program: bounce a message between two nodes

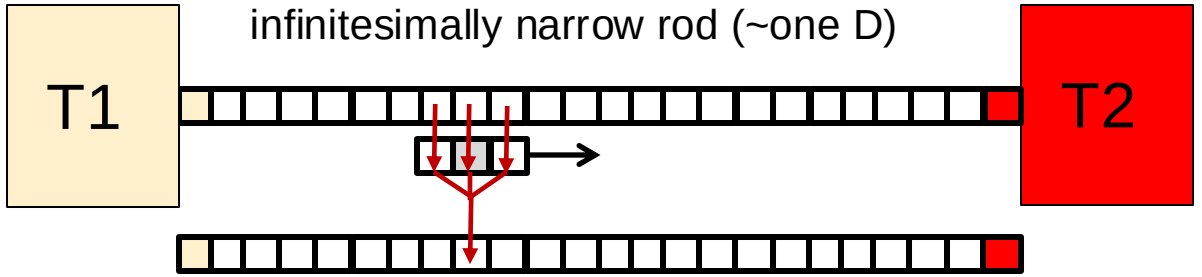
```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define VAL 42
#define NREPS  10
#define TAG 5

int main(int argc, char **argv)  {
   int rank, size;
   double t0;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   int bsend = VAL;
   int brecv = 0;
   MPI_Status stat;
   MPI_Barrier(MPI_COMM_WORLD);
   if(rank == 0) t0 = MPI_Wtime();
   for(int i=0;i<NREPS; i++){
     if(rank == 0){
       MPI_Send(&bsend, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD);
       MPI_Recv(&brecv, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD, &stat);
       if(brecv != VAL)printf("error: interation %d %d != %d\n",i,brecv,VAL);
       brecv = 0;
     }
     else if(rank == 1){
       MPI_Recv(&brecv, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD, &stat);
       MPI_Send(&bsend, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD);
       if(brecv != VAL)printf("error: interation %d %d != %d\n",i,brecv,VAL);
       brecv = 0;
     }
   }
   if(rank == 0){
     double t = MPI_Wtime() - t0;
     double lat = t/(2*NREPS);
     printf(" lat = %f seconds\n",(float)lat);
   }
   MPI_Finalize();
}
```

Let’s consider an example that looks more like a “real” message passing program … the heat diffusion problem we considered earlier

Heat Diffusion equation

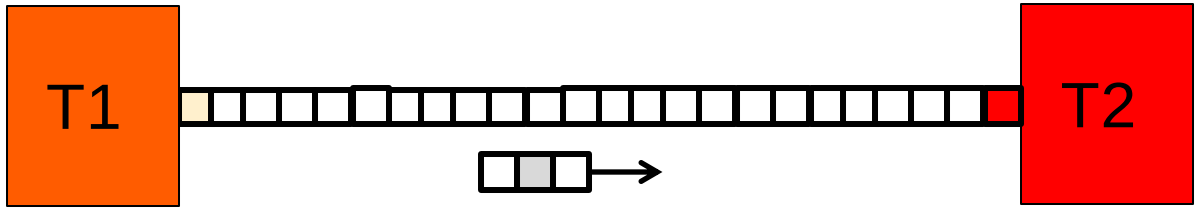


Heat Diffusion equation



Pictorially, you are sliding a three point “stencil” across the domain (u[t]) and computing a new value of the center point (u[t+1]) at each stop.

Heat Diffusion equation

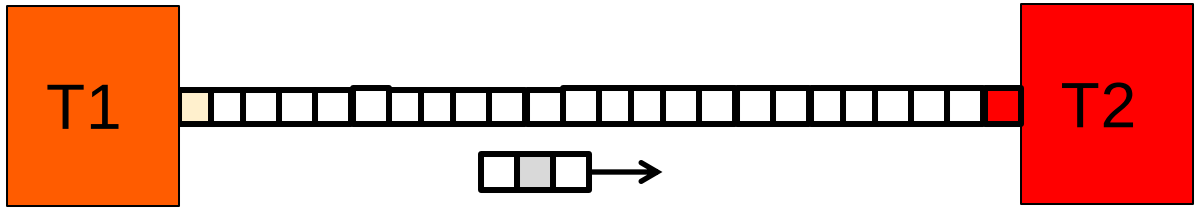



```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

   initialize_data(uk, ukp1, N, P); // initialize, set end temperatures
   for (int t = 0; t < N_STEPS; ++t){
      for (int x = 1; x < N-1; ++x)
          up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);

      temp = up1; up1 = u; u = temp;
    }
return 0;
```

Note: I don't need the intermediate "u[t]" values hence "u" is just indexed by x.

A well known trick with 2 arrays so I don't overwrite values from step k-1 as I fill in for step k

Heat Diffusion equation

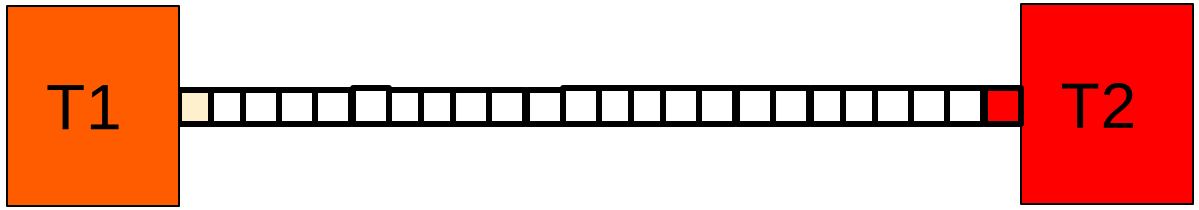



```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

   initialize_data(uk, ukp1, N, P); // initialize, set end temperatures
   for (int t = 0; t < N_STEPS; ++t){
      for (int x = 1; x < N-1; ++x)
          up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);

      temp = up1; up1 = u; u = temp;
    }
return 0;
```

How would you parallelize this program?

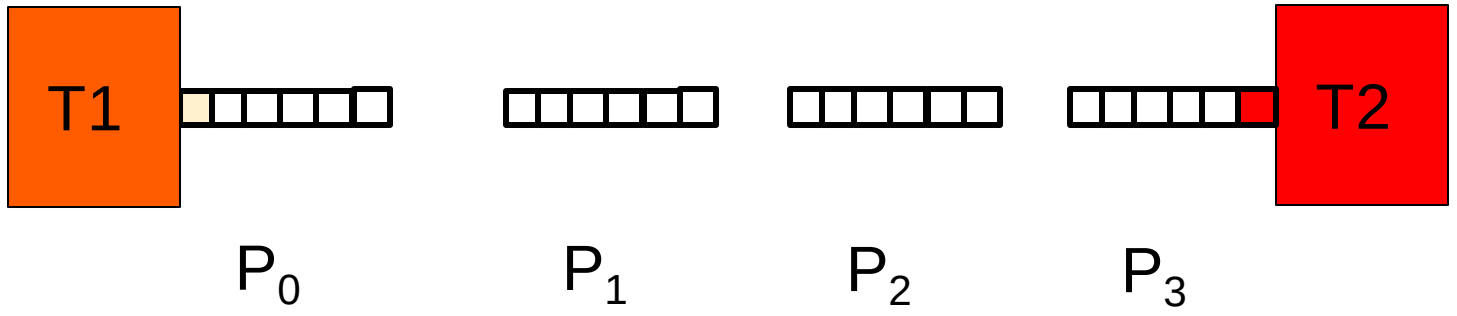
Heat Diffusion equation

- Start with our original picture of the problem … a one dimensional domain with end points set at a fixed temperature.



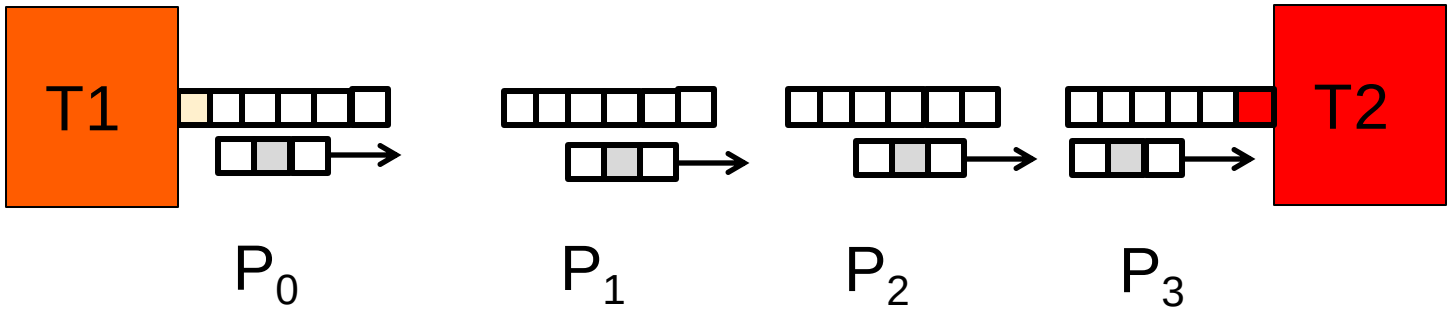
Heat Diffusion equation

- Break it into chunks assigning one chunk to each process.



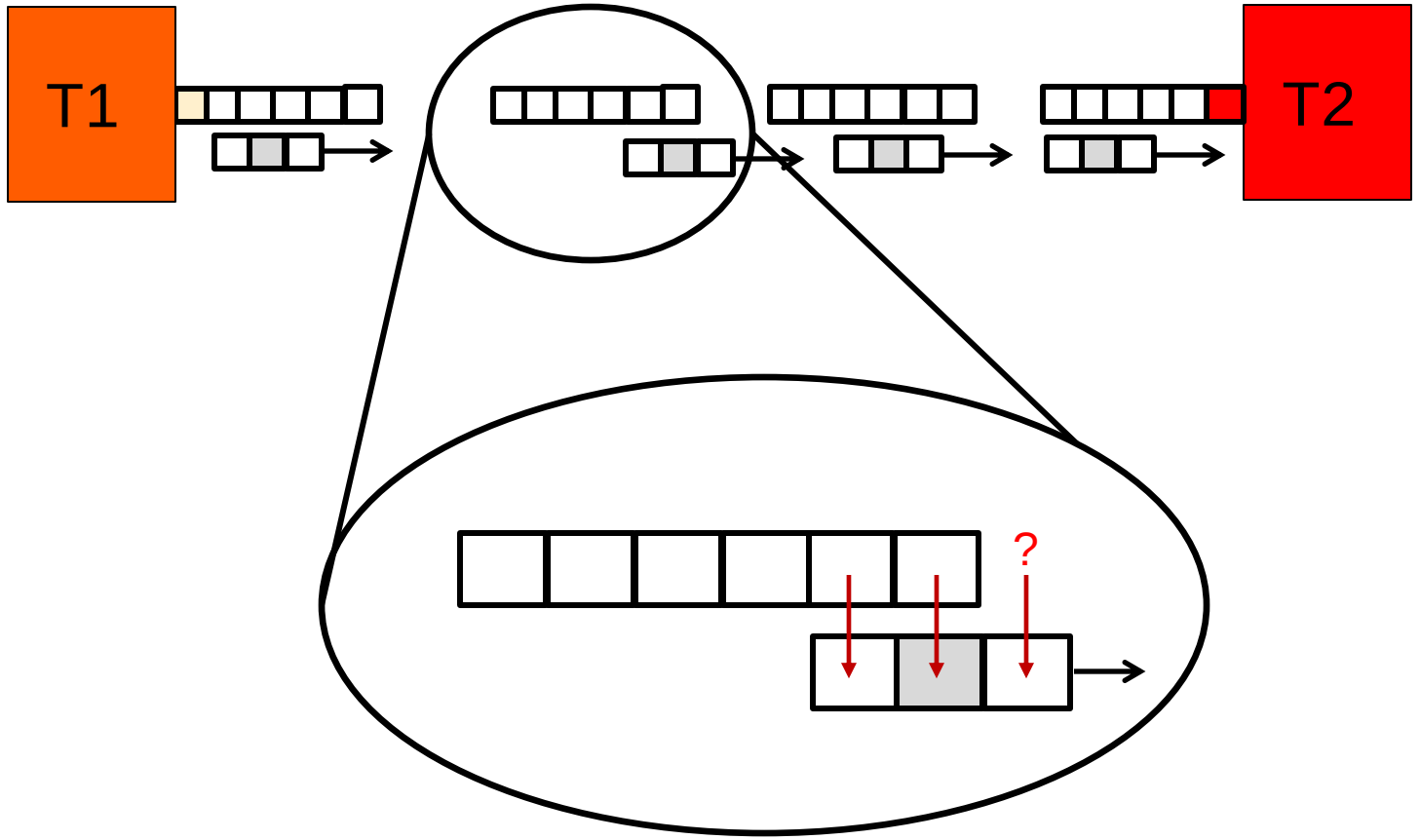
Heat Diffusion equation

- Each process works on it's own chunk … sliding the stencil across the domain to updates its own data.



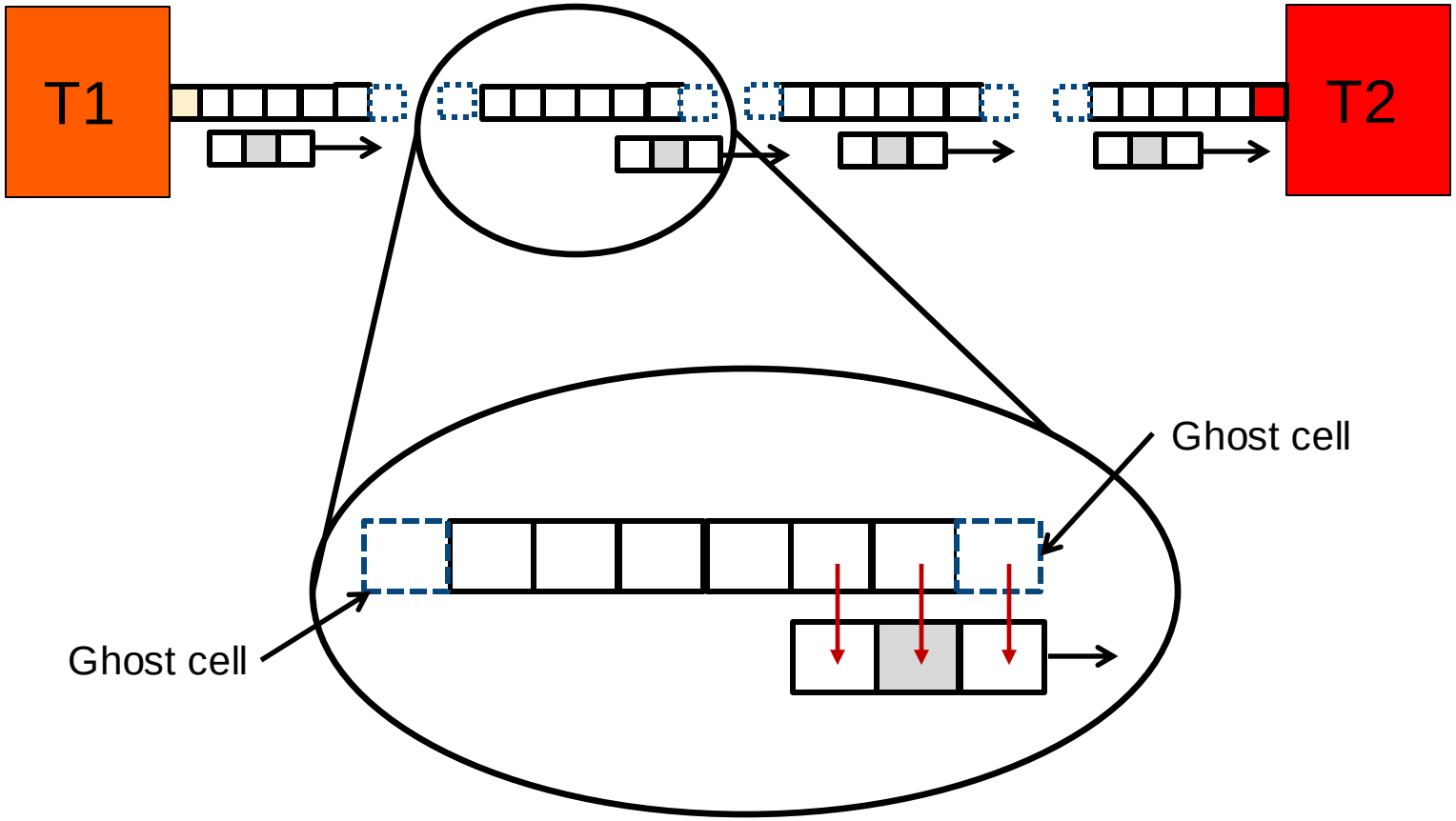
Heat Diffusion equation

- What about the ends of each chunk … where the stencil will run off the end and hence have missing values for the computation?



Heat Diffusion equation

- We add ghost cells to the ends of each chunk, update them with the required values from neighbor chunks at each time step … hence giving the stencil everything it needs on any given chunk to update all of its values.



Heat Diffusion MPI Example: Updating a chunk

```
// Compute interior of each "chunk"
  for (int x = 2; x < N/P; ++x)
    up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);

// update edges of each chunk keeping the two far ends fixed
// (first element on Process 0 and the last element on process P-1).
  if (myID != 0)
    up1[1] = u[1] + (k / (h*h)) * (u[1+1] - 2*u[1] + u[1-1]);

  if (myID != P-1)
    up1[N/P] = u[N/P] + (k/(h*h)) * (u[N/P+1] - 2*u[N/P] + u[N/P-1]);

// Swap pointers to prepare for next iterations
  temp = up1; up1 = u; u = temp;

} // End of for (int t ...) loop

MPI_Finalize();
return 0;
```

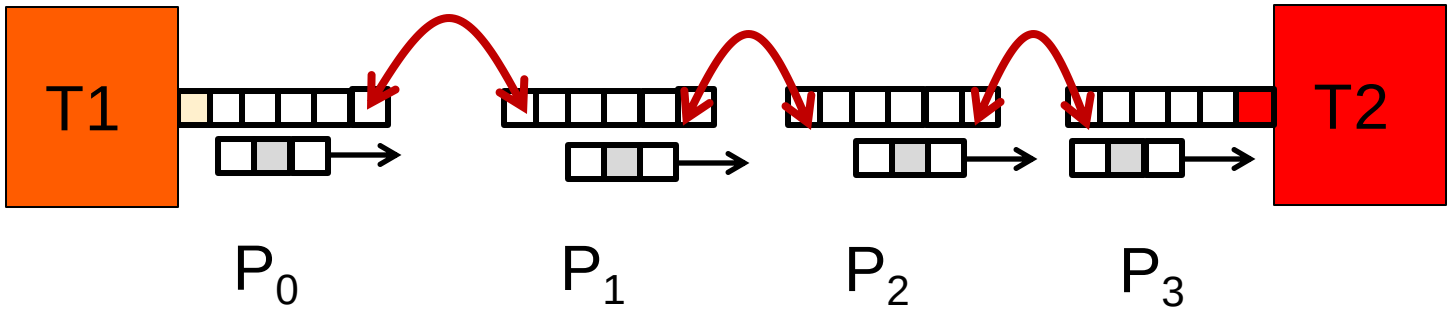
Update array values using local data and values from ghost cells.

u[0] and u[N/P+1] are the ghost cells

Note I was lazy and assumed N was evenly divided by P. Clearly, I'd never do this in a "real" program.

Heat Diffusion MPI Example: Communication

- Each process works on it's own chunk … sliding the stencil across the domain to updates its own data.



Heat Diffusion MPI Example

```
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &P);
MPI_Comm_rank (MPI_COMM_WORLD, &myID);
double *u   = malloc (sizeof(double) * (2 + N/P))  // include "Ghost Cells" to hold
double *up1 = malloc (sizeof(double) * (2 + N/P)); // values from my neighbors

initialize_data(uk, ukp1, N, P);
for (int t = 0; t < N_STEPS; ++t){

  if (myID != 0) MPI_Send (&u[1], 1, MPI_DOUBLE, myID-1, 0, MPI_COMM_WORLD);

  if (myID != P-1) MPI_Recv (&u[N/P+1], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD, &status);

  if (myID != P-1) MPI_Send (&u[N/P], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD);

  if (myID != 0) MPI_Recv (&u[0], 1, MPI_DOUBLE, myID-1, 0,MPI_COMM_WORLD, &status);
```

Note: the edges of domain are held at a fixed temperature.
- Node 0 has no neighbor to the left
- Node P has no neighbor to its right

Send my “left” boundary value to the neighbor on my “left’

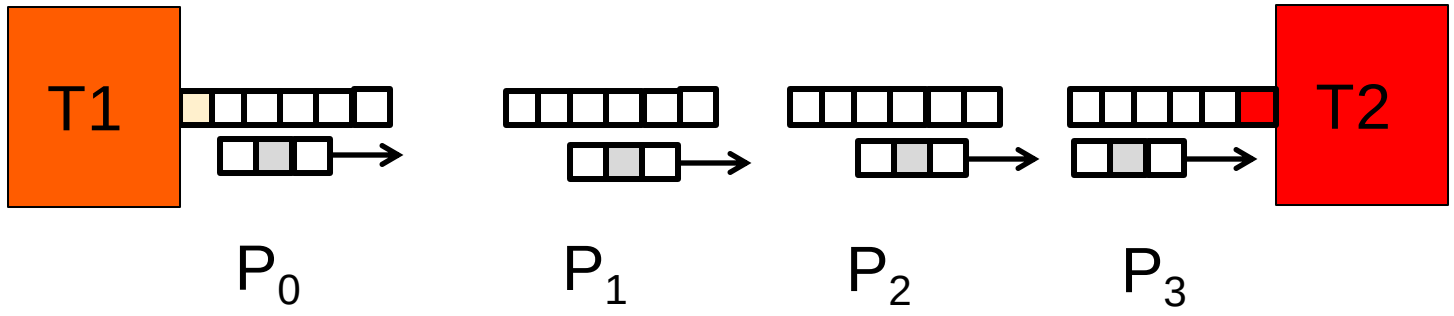
Receive my “right” ghost cell from the neighbor to my “right’

Send my “right” boundary value to the neighbor to my “right’

Receive my “left” ghost cell from the neighbor to my “left”

Heat Diffusion equation

- Each process works on it's own chunk … sliding the stencil across the domain to updates its own data.



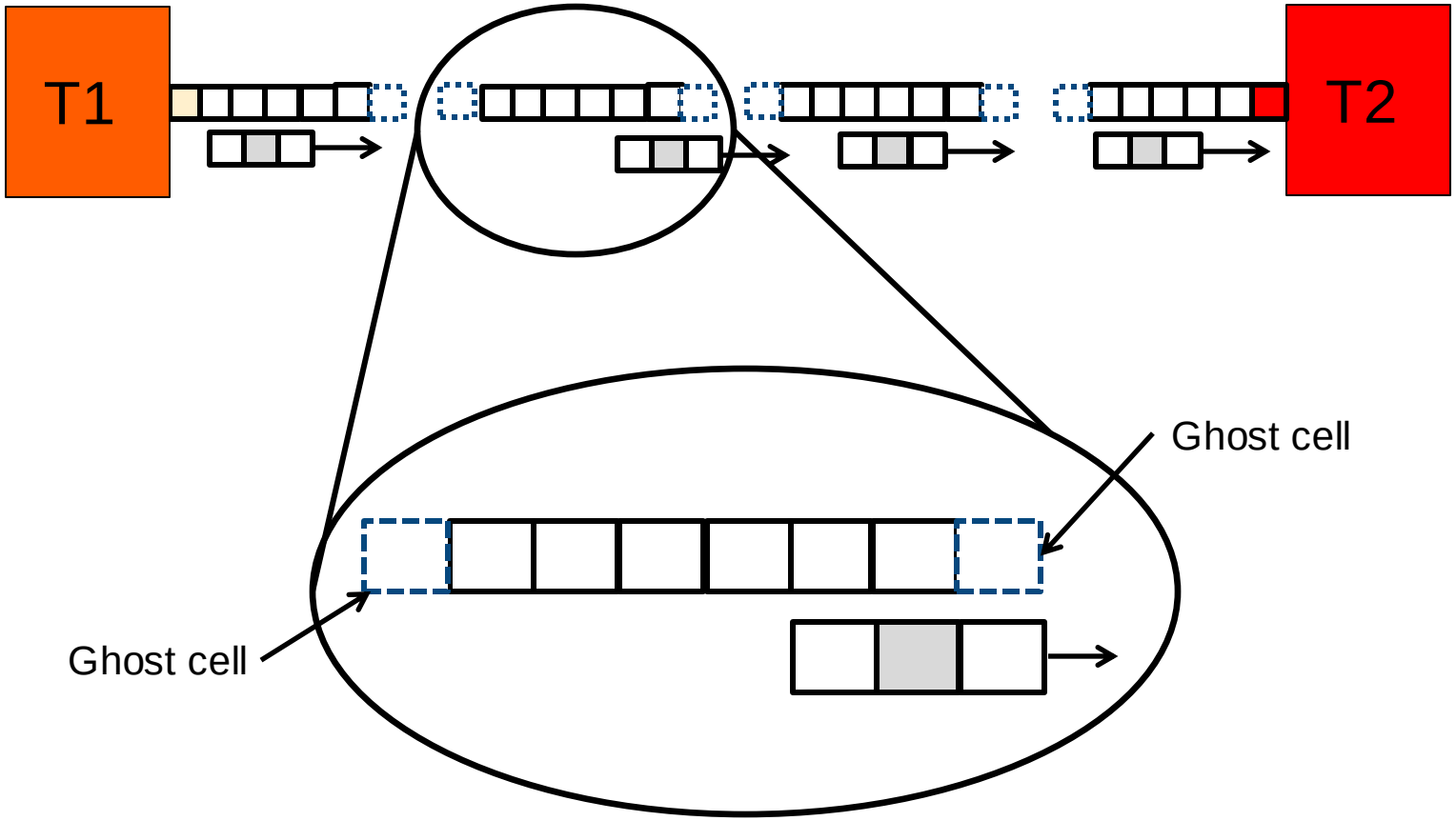
We now put all the pieces together for the full program

Heat Diffusion MPI Example

```
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &P);
MPI_Comm_rank (MPI_COMM_WORLD, &myID);
double *u   = malloc (sizeof(double) * (2 + N/P))  // include "Ghost Cells” to hold
double *up1 = malloc (sizeof(double) * (2 + N/P)); // values from my neighbors

initialize_data(uk, ukp1, N, P);
for (int t = 0; t < N_STEPS; ++t){
  if (myID != 0)  MPI_Send (&u[1], 1, MPI_DOUBLE, myID-1, 0, MPI_COMM_WORLD);
  if (myID != P-1) MPI_Recv (&u[N/P+1], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD, &status);
  if (myID != P-1) MPI_Send (&u[N/P], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD);
  if (myID != 0)   MPI_Recv (&u[0], 1, MPI_DOUBLE, myID-1, 0,MPI_COMM_WORLD, &status);

  for (int x = 2; x < N/P; ++x)
    up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);
  if (myID != 0)
    up1[1] = u[1] + (k / (h*h)) * (u[1+1] - 2*u[1] + u[1-1]);
  if (myID != P-1)
    up1[N/P] = u[N/P] + (k/(h*h)) * (u[N/P+1] - 2*u[N/P] + u[N/P-1]);
  temp = up1; up1 = u; u = temp;

} // End of for (int t ...) loop

MPI_Finalize();
return 0;
```

The Geometric Decomposition Pattern

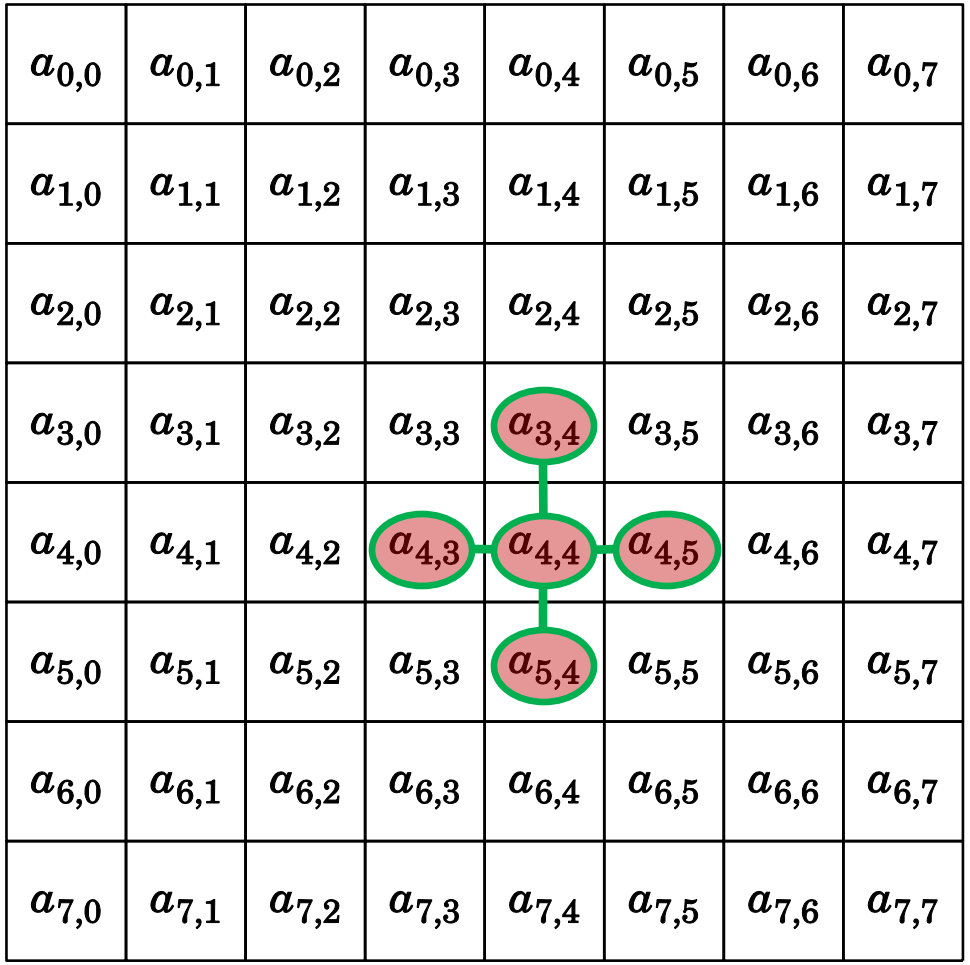
- This is an instance of a very important design pattern … the Geometric decomposition pattern.



We generalize this to higher dimension problems by thinking in terms of partitioned arrays that we distribute around the system

Partitioned Arrays

- Realistic problems are 2D or 3D; require more complex data distributions.
- We need to parallelize the computation by partitioning this index space
- Example: Consider a 2D domain over which we wish to solve a PDE using an explicit finite difference solver . The figure shows a five point stencil … update a value based on its value and its 4 neighbors.
- Start with an array and stencil →

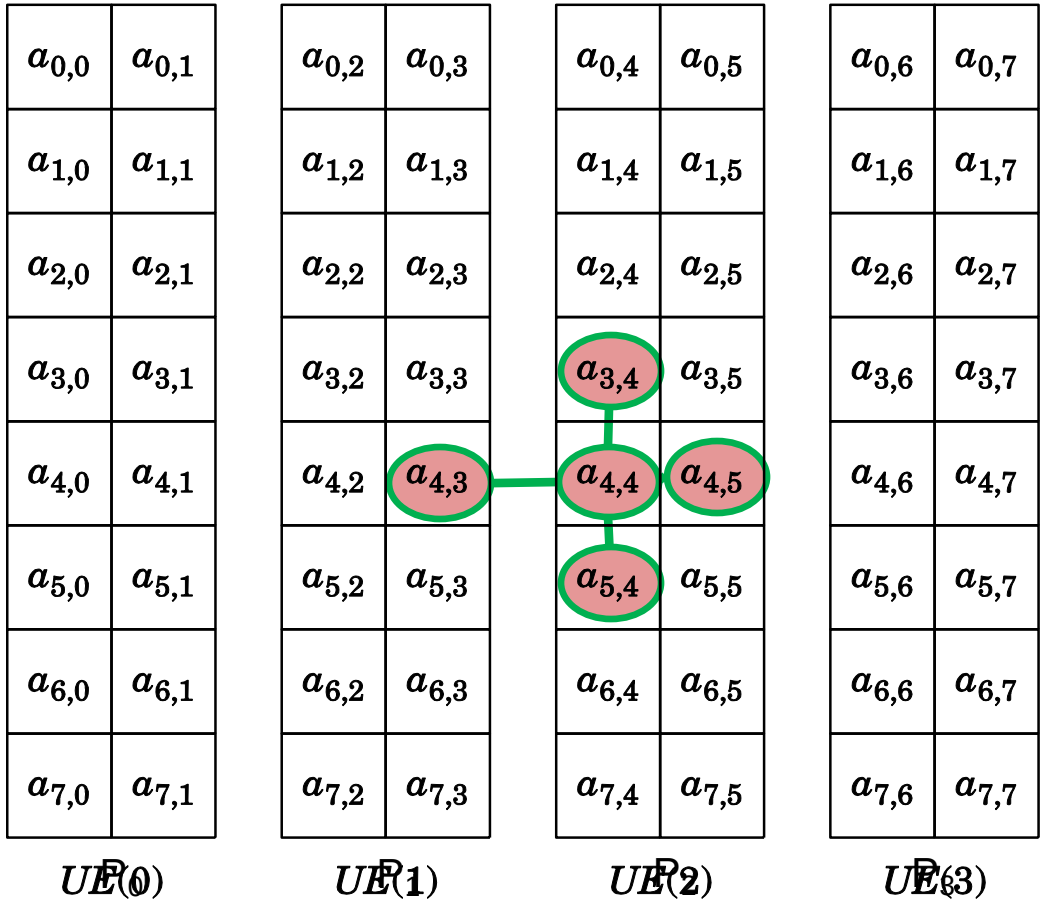


Partitioned Arrays: Column block distribution

- Split the non-unit-stride dimension (P-1) times to produce P chunks, assign the i^{th} chunk to P_i. …. To keep things simple, assume N%P = 0
- In a 2D finite-differencing program (exchange edges), how much do we have to communicate? **O(N) values** per processor

P is the # of processors

N is the order of our square matrix



Partitioned Arrays: Block distribution

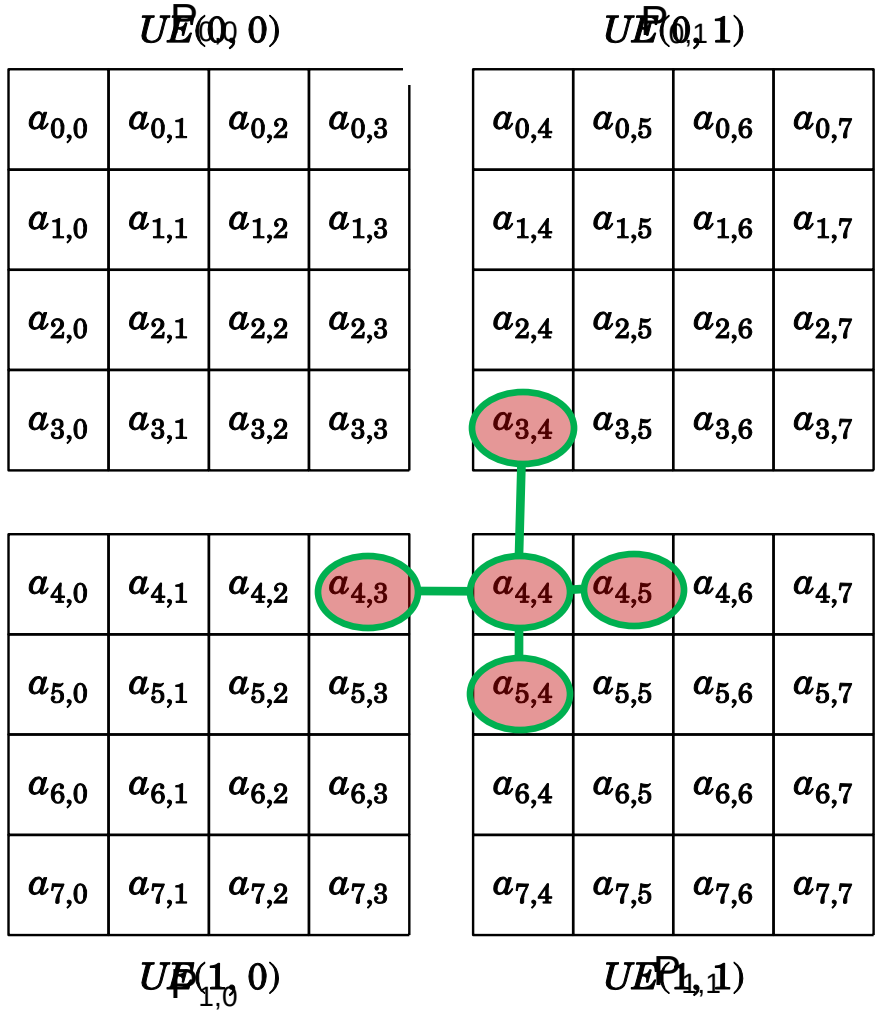
- If we parallelize in both dimensions, then we have $(N/P^{1/2})^2$ elements per processor, and we need to send **$O(N/P^{1/2})$ values** from each processor. Asymptotically better than O(N).

P is the # of processors

Assume a p by p square mesh … $p=P^{1/2}$

N is the order of our square matrix

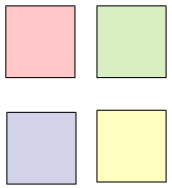
Dimension of each block is $N/P^{1/2}$



Partitioned Arrays: block cyclic distribution

- LU decomposition (A= LU) .. Move down the diagonal transform rows to “zero the column” below the diagonal.

  - Zeros fill in the right lower triangle of the matrix … less work to do.
  - Balance load with cyclic distribution of blocks of A mapped onto a grid of nodes (2x2 in this case … colors show the mapping to nodes).



And that's if for MPI.

And that means we've covered ALL of the major classes of parallel systems and associated programming models

Summary/Conclusion

- The need for parallelism grows out of fundamental physical limitations to what we can build in a semiconductor. It's not marketing …. It's reality.

- Getting the most from each watt expended is vital … and that means you all have to produce code that runs in parallel on heterogeneous devices.

- Fortunately, this isn't as daunting as it sounds. There are a small number of key abstractions and design patterns to cover most (if not all) of parallel scientific computing.
  - There are four key execution models (fork/join, SIMT, CSP, SIMD)
  - There are eight core design patterns: loop parallel, SPMD, task queue, divide and conquer, Geometric Decomposition, Data parallelism, directed acyclic graph, and actors.

The next two lectures are EXTREMELY Important. We will cover two essential issues a computational scientist MUST understand … but they are not generally included in what we cover when training scientists!

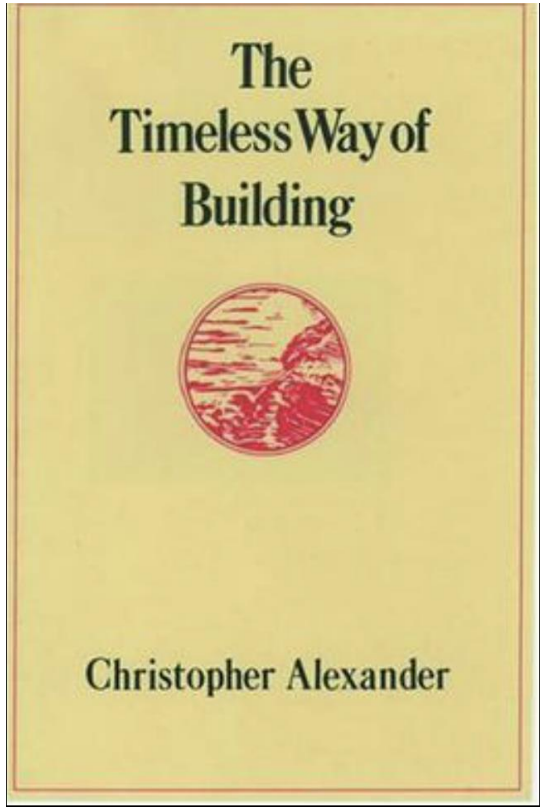
We didn't cover these since they don't come up in scientific computing very often

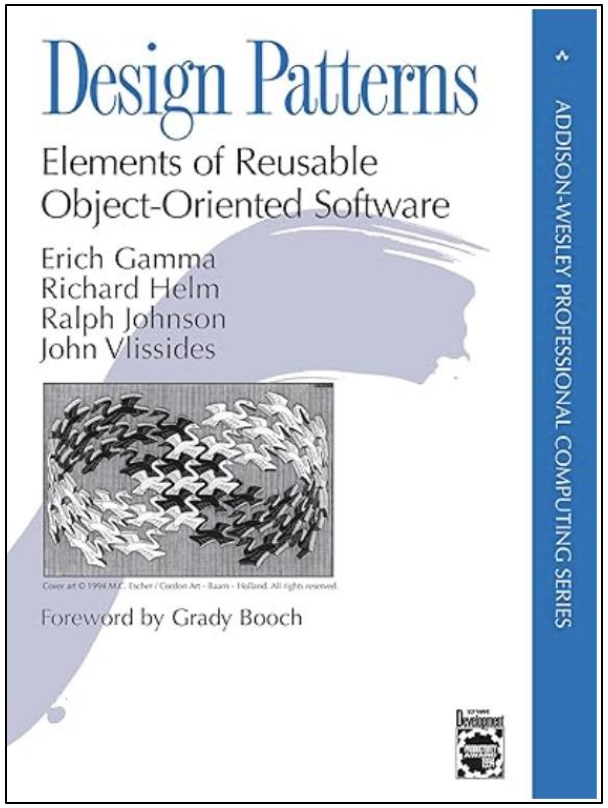
CSP: communicating sequential processes … the fundamental execution model of clusters
Directed Acyclic Graphs: build a task graph where edges are dependencies
Actors: persistent processes with built in methods that react to incoming messages.

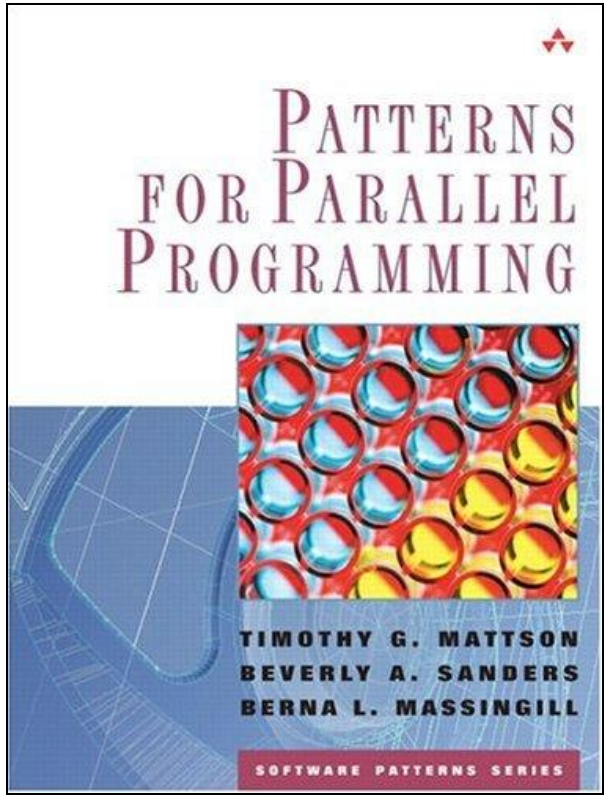
Gathering them all together in one place to make it easier for you to review them … here are the key design patterns of parallel computing in the sciences

Design Patterns: An Engineering discipline of design

Design patterns encode the fundamental practice of how people think about classes of problems and their solutions.







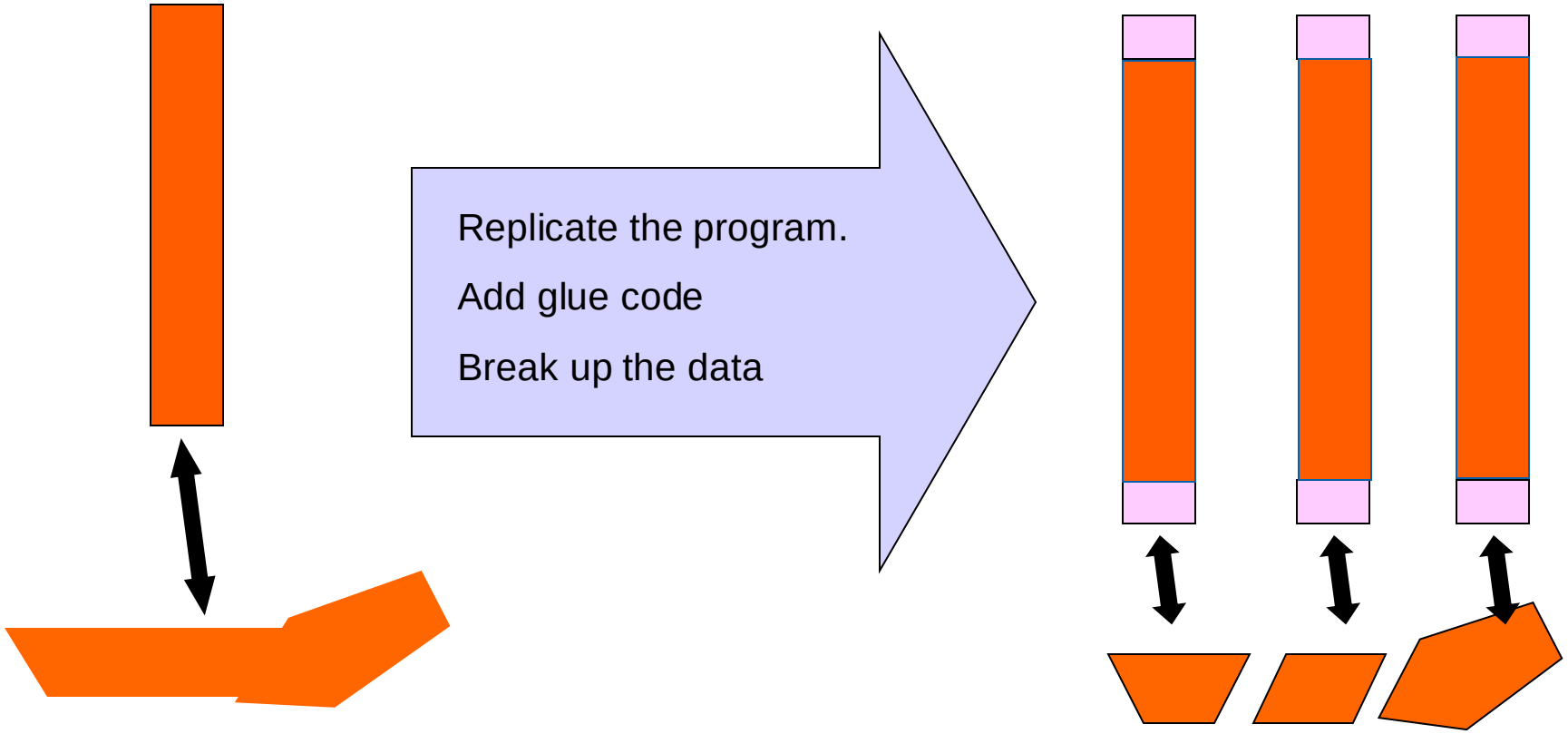
The idea of design patterns comes from the architect Christopher Alexander in his 1979 book … where he presents design patterns as a way of capturing that essential ”quality without a name” experts see though often struggle to define.

This book released in 1995 brought order to the chaos that characterized the early days of object oriented programming. They used design patterns to systematically organize best-practices in object oriented design. This is one of the most influential books in the history of computer science.

This book (released in 2004) brought the discipline of design patterns to parallel programming. The goal is to help people understand how to ”think parallel” with examples in OpenMP, MPI and Java

SPMD (Single Program Multiple Data) Design Pattern

- Run the same program on P processing elements where P can be arbitrarily large.



- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

MPI programs almost always use this pattern … it is probably the most commonly used pattern in the history of parallel programming.

Example: SPMD Pi program using MPI

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        for (i=my_id*my_steps; i<(my_id+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD); ;
}
```

Sum values in “sum” from each process and place it in “pi” on process 0

Example: SPMD Pi program with OpenMP

```
#include <omp.h>
static long num_steps = 100000;        double step;

void main ()
{  int nthreads; double  pi=0.0;          step = 1.0/(double) num_steps;

  #pragma omp parallel
  {
     int i, id, nthrds;    double x, sum;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
      if (id == 0)   nthreads = nthrds;
      for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
       }
      #pragma omp critical
              pi += sum * step;
   }
}
```

Use the **id** and **nthrds** to split up work between threads

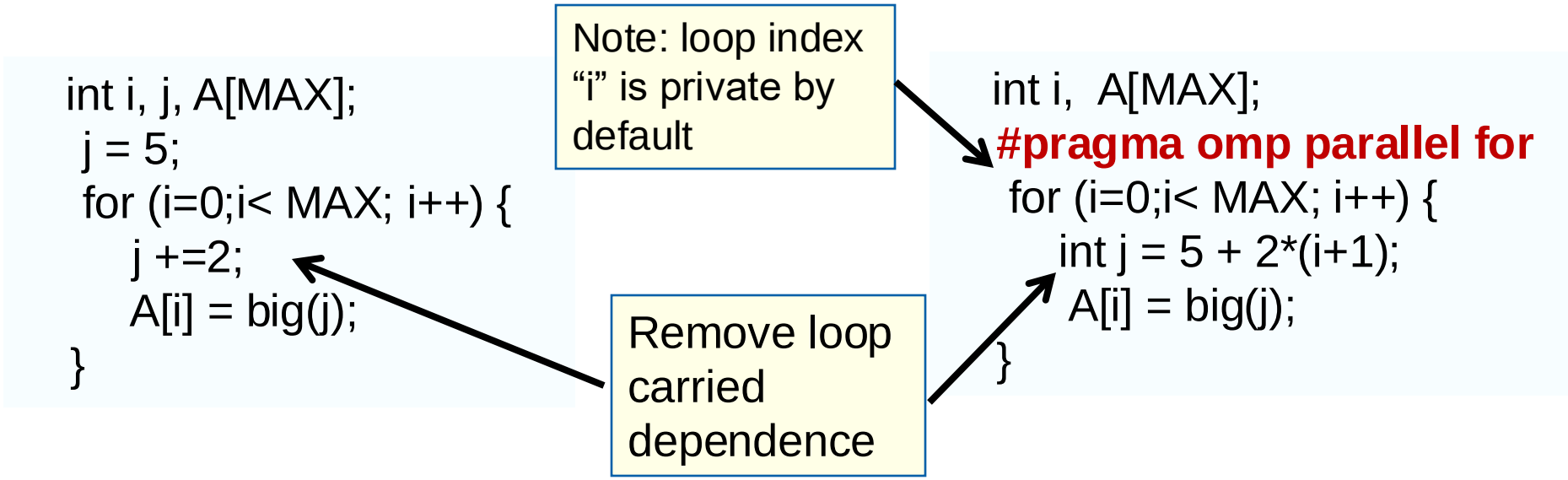
Create scalars local to each thread to accumulate partial sums.

Find an ID (rank) for each thread and the total number of threads in the parallel team

The critical section causes only one thread at a time to execute the statements in the construct.

Loop-level parallelism Design Pattern

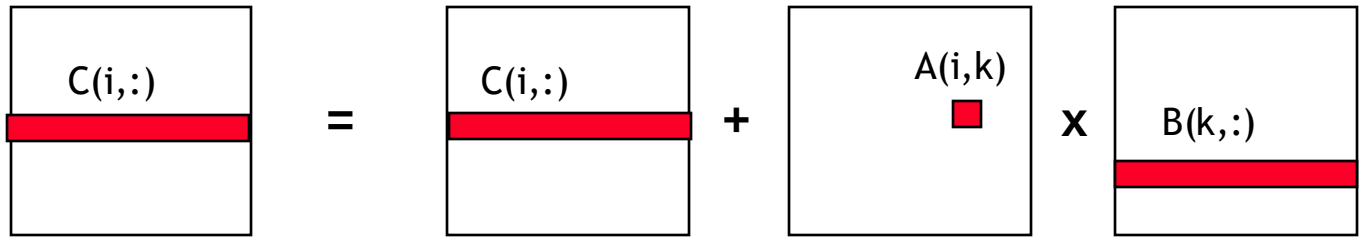
- Find compute intensive loops **with concurrency** that can be exploited in parallel
- **Expose the concurrency**, that is, make the loop iterations independent so they can run in parallel
- Modify loop body and scheduling of iterations to balance the load and optimize memory utilization.
  - This may require fusing loops to create enough work to compensate for loop control overhead.
- Insert appropriate constructs from a parallel programming application programming interface (such as OpenMP) so they run in parallel



Example: Parallel ikj Matrix Multiplication Algorithm

- We can reduce cache misses by swapping the j and k loops … so the innermost loop is over j and the access patterns across the C and B matrices cache friendly. We call this is ikj algorithm

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    #pragma omp parallel for private(k,j)
    for (i = 0; i < N; i++) {
      for (k = 0; k < N; k++) {
          for (j = 0; j < N; j++) {
              C[i*N+j] += A[i*N+k] * B[k*N+j];
          }
       }
    }
}
```

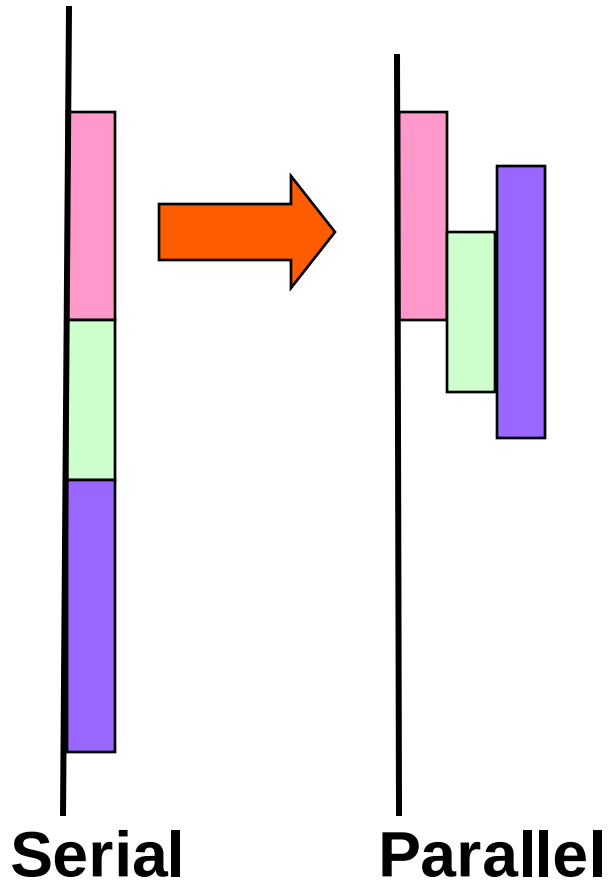


Scale a row of B by an element of A and add to a row of C

Task queue Design Pattern

- Used for irregular work that doesn't map onto basic for-loops. Also useful for work that can not be deterministically ordered for effective load balancing.

- Solution:
  - Fill a queue with a collection of tasks
  - A set of threads cooperatively execute the tasks until all tasks complete or an exit condition is met.
  - The queuing system is expected to provide the following:
    - The queue is a thread-safe, concurrent data structure designed to be manipulated by multiple threads with low overhead.
    - The queuing system tracks capacity of the queue and will limit creation of new tasks a so threads can drain the queue before more are added.
    - Quality queuing systems will distribute queues and use work-stealing to optimize scheduling of tasks



The key to the power of this pattern is that all this complexity is done by the system … transparent to the application programmer.

Parallel Linked List Traversal

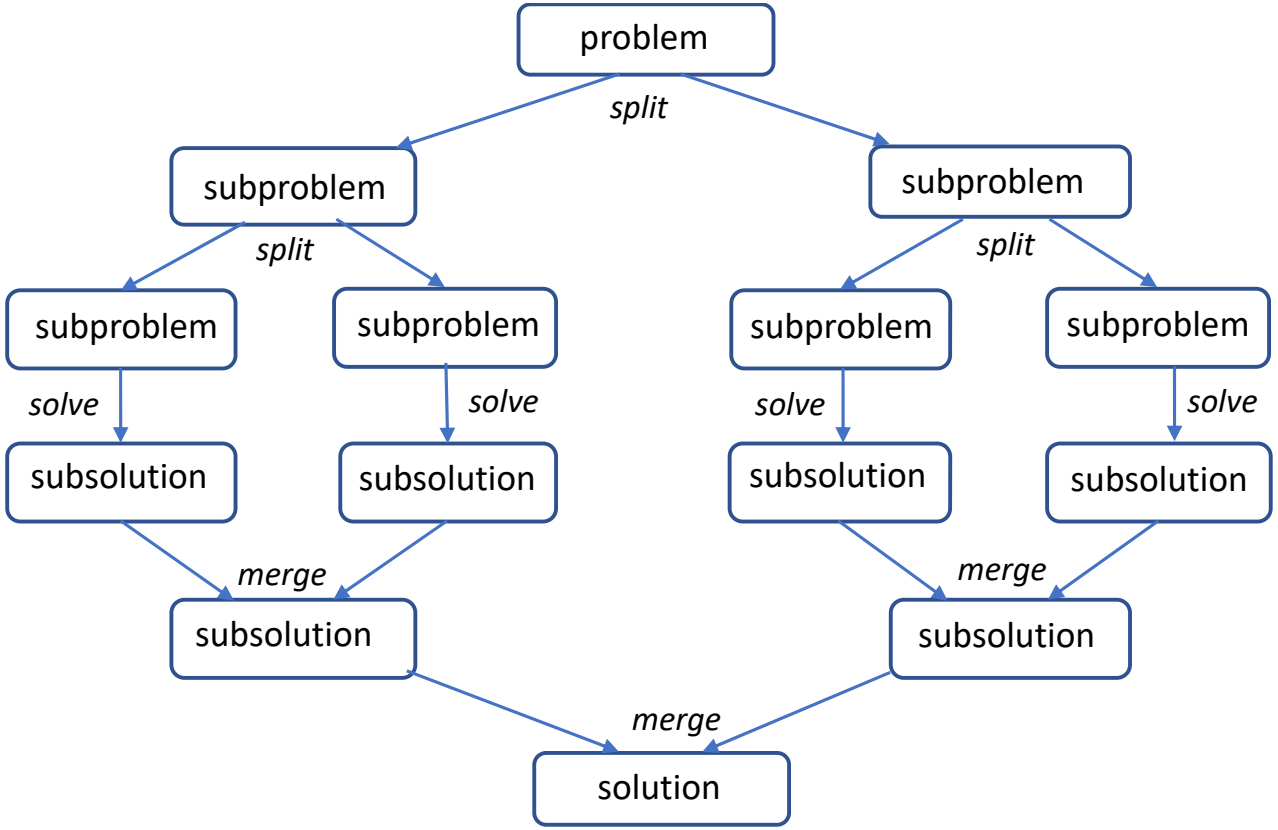
```
#pragma omp parallel
{
  #pragma omp single
   {
    p = listhead ;
    while (p) {
       #pragma omp task firstprivate(p)
              {
                process (p);
              }
       p=next (p) ;
     }
   }
}
```

Only one thread packages tasks

makes a copy of `p` when the task is packaged

Divide and Conquer design pattern

- Split the problem into smaller sub-problems; continue until the sub-problems can be solved directly



- 3 opportunities for parallelism:
  - Do work as you split into sub-problems
  - Do work only at the leaves
  - Do work as you recombine

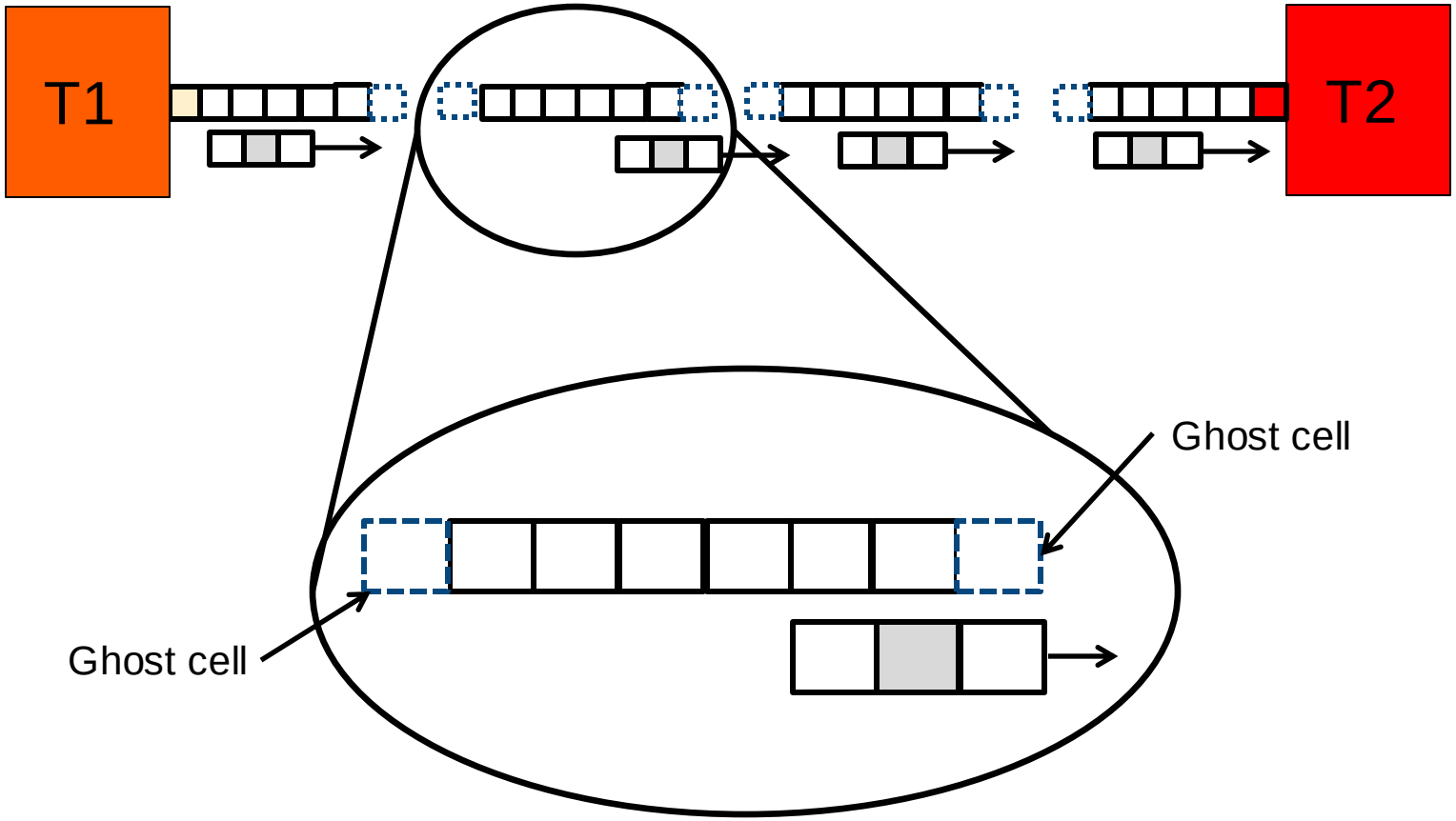
Program: OpenMP tasks

```
include <omp.h>
static long num_steps = 100000000;
#define MIN_BLK  10000000
double pi_comp(int Nstart,int Nfinish,double step)
{   int i,iblk;
    double x, sum = 0.0,sum1, sum2;
    if (Nfinish-Nstart < MIN_BLK){
       for (i=Nstart;i< Nfinish; i++){
          x = (i+0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
       }
    }
    else{
       iblk = Nfinish-Nstart;
       #pragma omp task shared(sum1)
            sum1 = pi_comp(Nstart,         Nfinish-iblk/2,step);
       #pragma omp task shared(sum2)
             sum2 = pi_comp(Nfinish-iblk/2, Nfinish,       step);
       #pragma omp taskwait
          sum = sum1 + sum2;
   }return sum;
}
```

```
int main ()
{
  int i;
  double step, pi, sum;
   step = 1.0/(double) num_steps;
   #pragma omp parallel
   {
      #pragma omp single
          sum =
              pi_comp(0,num_steps,step);
    }
     pi = step * sum;
}
```

The Geometric Decomposition Pattern

- This is an instance of a very important design pattern … the Geometric decomposition pattern.



Example: geometric decomposition (Heat Diffusion) with MPI

```
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &P);
MPI_Comm_rank (MPI_COMM_WORLD, &myID);
double *u   = malloc (sizeof(double) * (2 + N/P))  // include "Ghost Cells” to hold
double *up1 = malloc (sizeof(double) * (2 + N/P)); // values from my neighbors

initialize_data(uk, ukp1, N, P);
for (int t = 0; t < N_STEPS; ++t){
  if (myID != 0)  MPI_Send (&u[1], 1, MPI_DOUBLE, myID-1, 0, MPI_COMM_WORLD);
  if (myID != P-1) MPI_Recv (&u[N/P+1], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD, &status);
  if (myID != P-1) MPI_Send (&u[N/P], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD);
  if (myID != 0)   MPI_Recv (&u[0], 1, MPI_DOUBLE, myID-1, 0,MPI_COMM_WORLD, &status);

  for (int x = 2; x < N/P; ++x)
    up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);
  if (myID != 0)
    up1[1] = u[1] + (k / (h*h)) * (u[1+1] - 2*u[1] + u[1-1]);
  if (myID != P-1)
    up1[N/P] = u[N/P] + (k/(h*h)) * (u[N/P+1] - 2*u[N/P] + u[N/P-1]);
  temp = up1; up1 = u; u = temp;

} // End of for (int t ...) loop

MPI_Finalize();
return 0;
```

Data Parallelism and GPU programming:

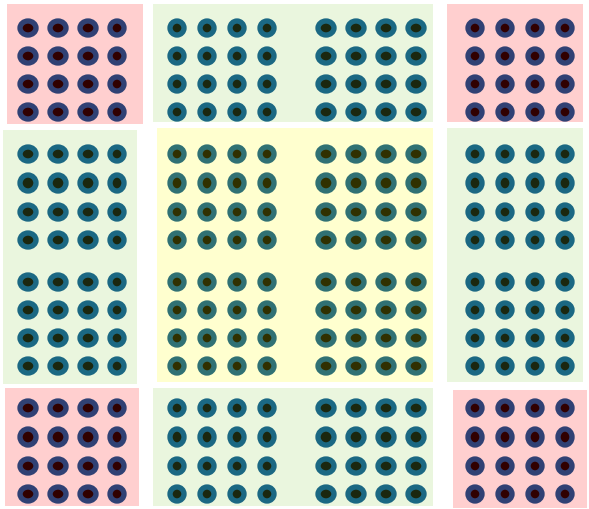
The SIMT model (Single Instruction Multiple Thread)

1. Turn kernel code into a scalar work-item

```
// Compute sum of order-N matrices: C = A + B
void __global__
matAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j<N) c[i][j] == a[i][j] + b[i][j];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // define threadBlocks and the Grid
   dim3 dimBlock(4,4);
   dim3 dimGrid(4,4);

  // Launch kernel on Grid
    matAdd <<< digGrid,dimBlock>>> (a, b, c, N);
}
```

This is CUDA code

2. Map work-items onto an N dim index space.



3. Map data structures onto the same index space

4. Run on hardware designed around the same SIMT execution model

Example: Heat diffusion problem

```
  const double r = alpha * dt / (dx * dx);
  const double r2 = 1.0 - 4.0*r;
  // malloc and initialize u_tmp and u (code not shown)
  #pragma omp target data enter map(to: u[0:n*n], u_tmp[0:n*n])

  for (int t = 0; t < nsteps; ++t) {
     #pragma omp target
     #pragma omp loop
     for (int i = 0; i < n; ++i) {
       for (int j = 0; j < n; ++j) {
         u_tmp[i+j*n] =  r2 * u[i+j*n]          +
             r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
             r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
             r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
             r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
       }
      }
      // Pointer swap to get ready for next step
      tmp = u;
      u = u_tmp;
      u_tmp = tmp;
  }
 #pragma omp target data exit map(from: u[0:n*n])
}
```

Create a data region and map indicated data on entry

Target and Loop constructs map loop body (the kernel) onto a 2 D index space.

Exit the data region and map indicated data

Nvidia HPC Toolkit compiler
nvc -fast -mp=gpu -gpu=cc75 heat.c

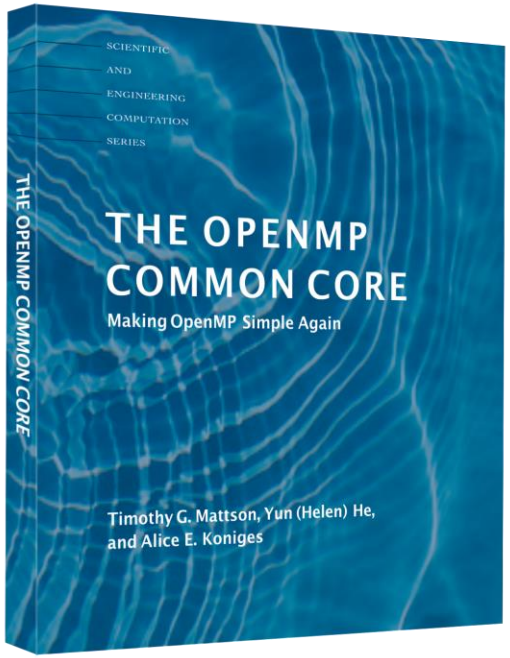
Summary

- Parallel computing is fun … but it can be hard.
- Fortunately, if you stick to the Big-3 programing models (MPI, OpenMP, and a GPU programming model such as OpenMP or Sycl) and the core patterns of parallel computing for HPC, it's not too overwhelming
  - Key Patterns: SPMD, loop level parallelism, geometric decomposition, divide and conquer, and data parallel (SIMT and vector)

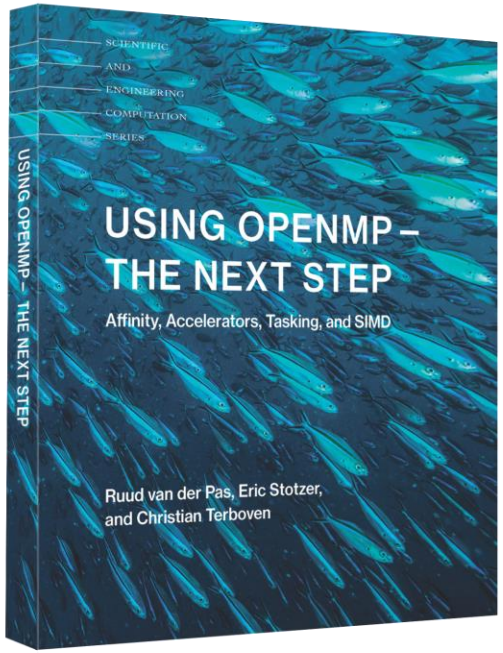
- Some day we'll automate the hard-parts with Machine Programming, but that may be 10 years!!!!

To learn more about OpenMP

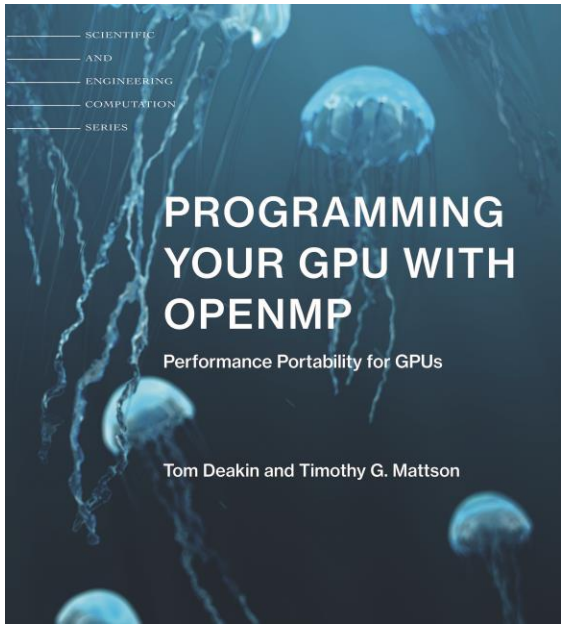
The OpenMP web site has a great deal of material to help you with OpenMP www.openmp.org
Reading the spec is painful … but each spec has a collection of examples. Study the examples, don't try to read the specs
Since the specs are written ONLY for implementors … programmers need the OpenMP Books to master OpenMP.



Start here … learn the basics and build a foundation for the future



Learn advanced features in OpenMP including tasking and GPU programming (up to version 4.5)



Master GPU programming using OpenMP