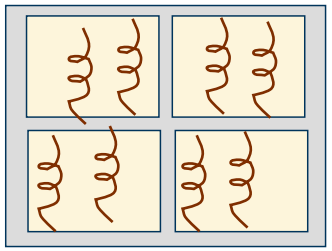
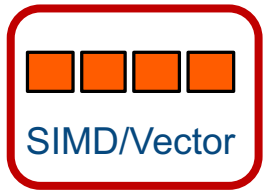
Parallel Programming Beyond the CPU

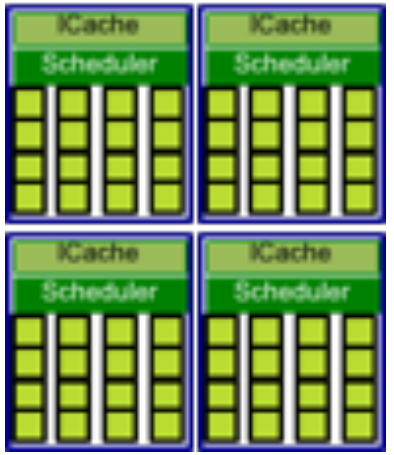
Tim Mattson

Hardware is diverse

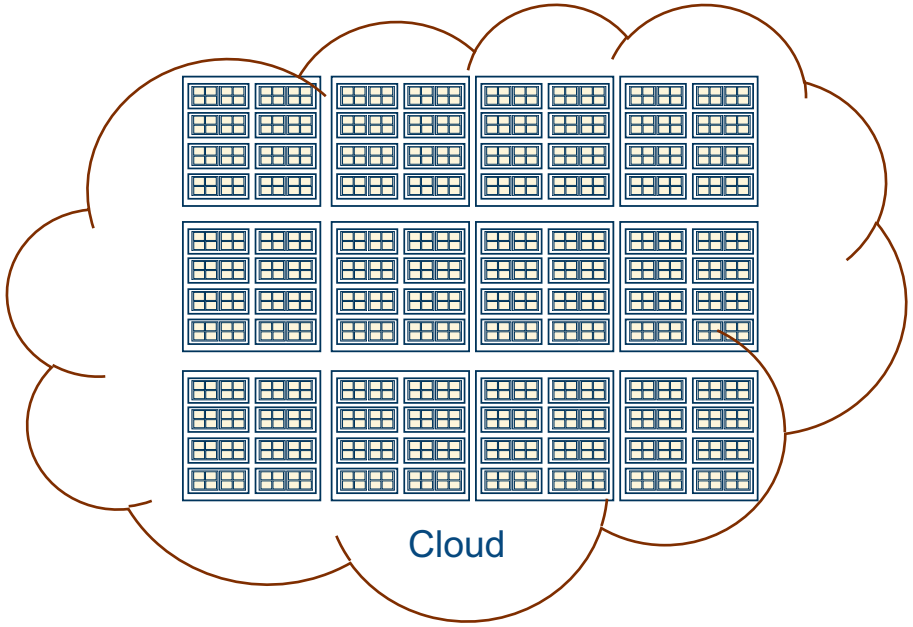


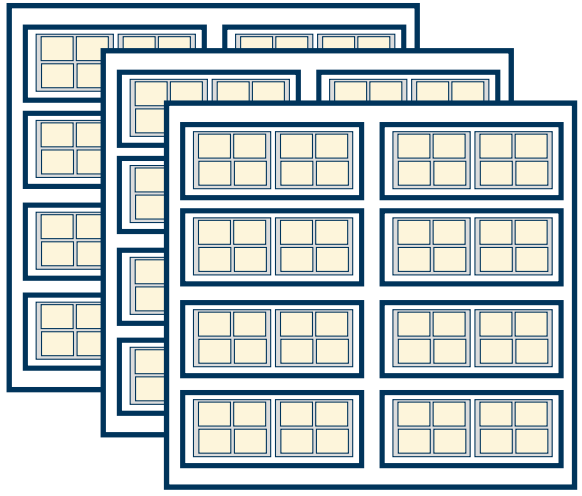
CPU



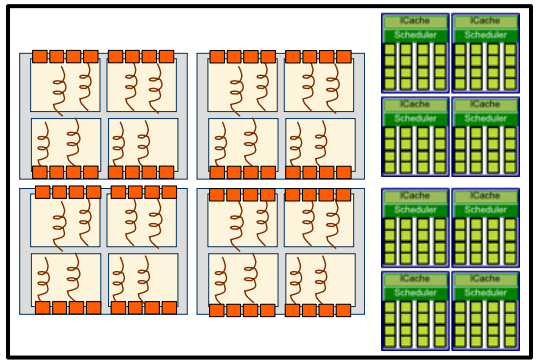


GPU





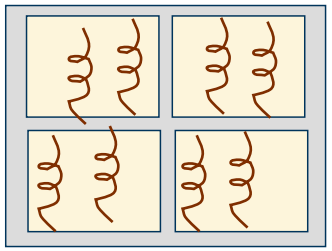
Cluster



Heterogeneous node

Hardware is diverse

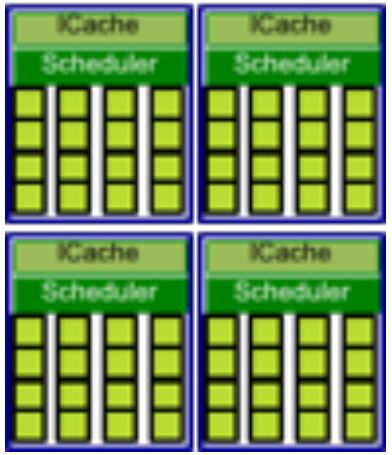
You know all about programming with threads and a shared memories



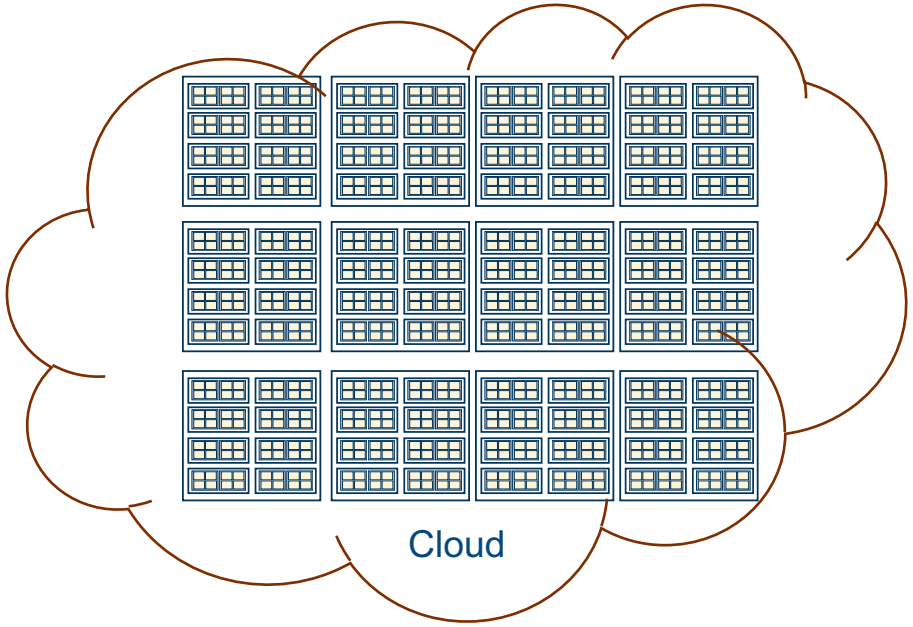
CPU

We'll leave vectorization to the compiler (use –O3)

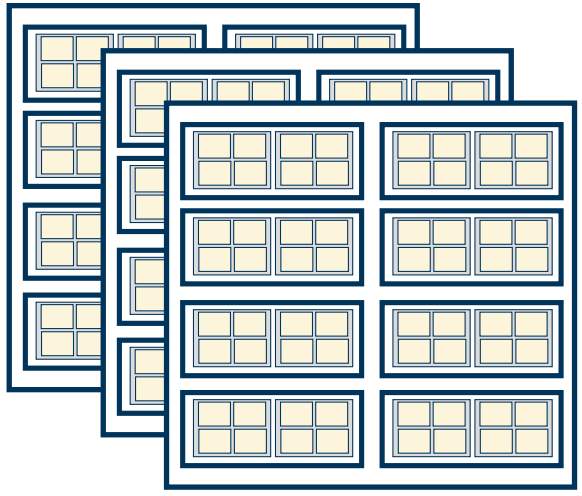
SIMD/Vector



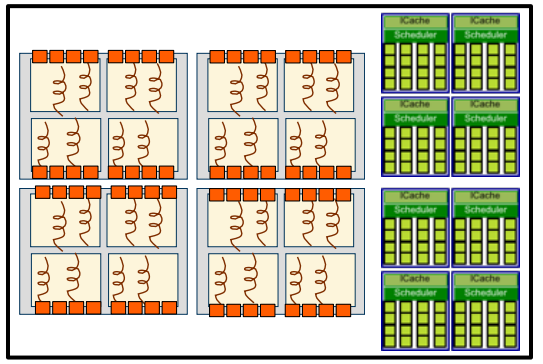
GPU



Cloud

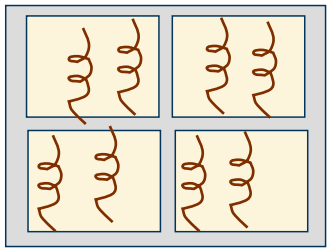


Cluster



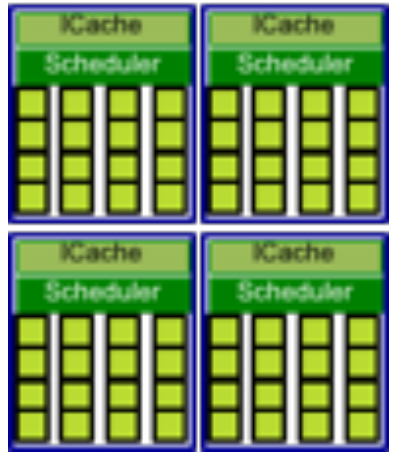
Heterogeneous node

Hardware is diverse



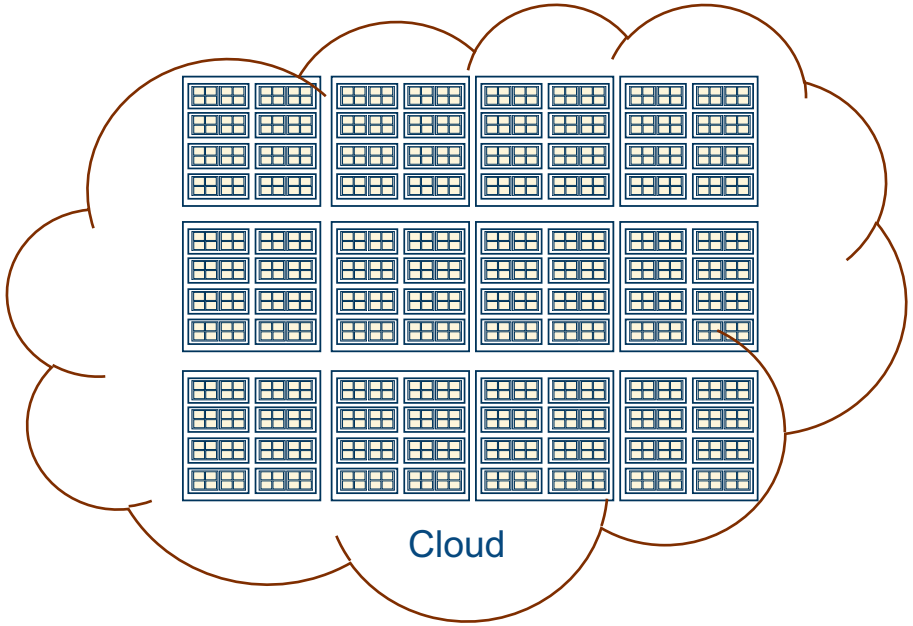
CPU

SIMD/Vector

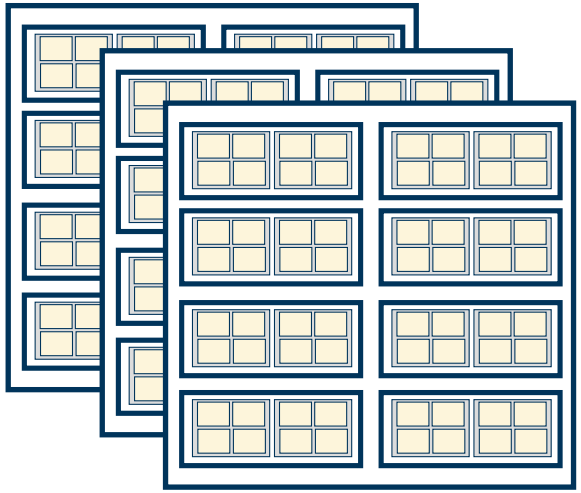


GPU

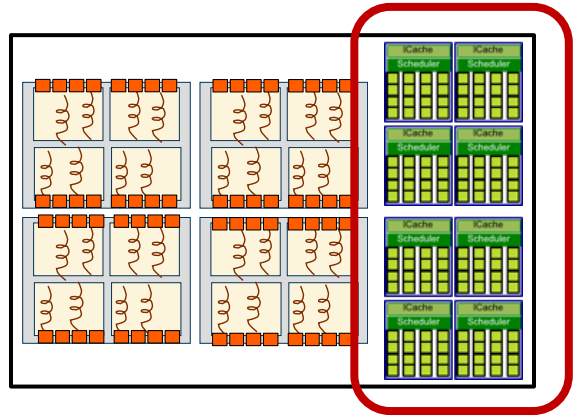
A very brief introduction to GPU programming



Cloud

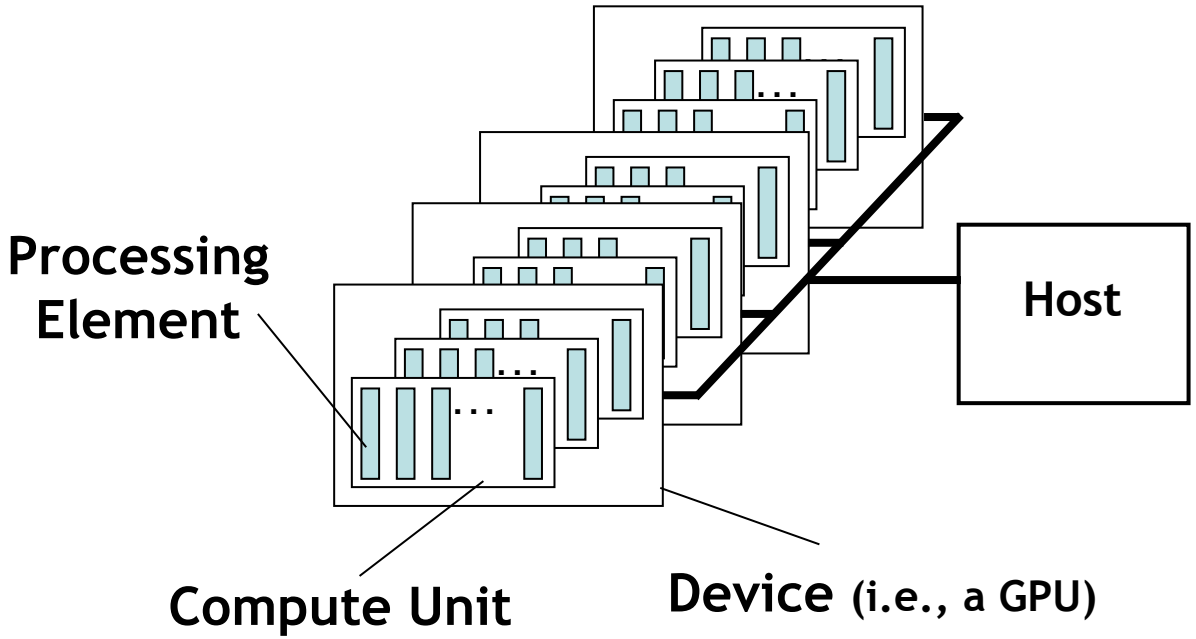


Cluster



Heterogeneous node

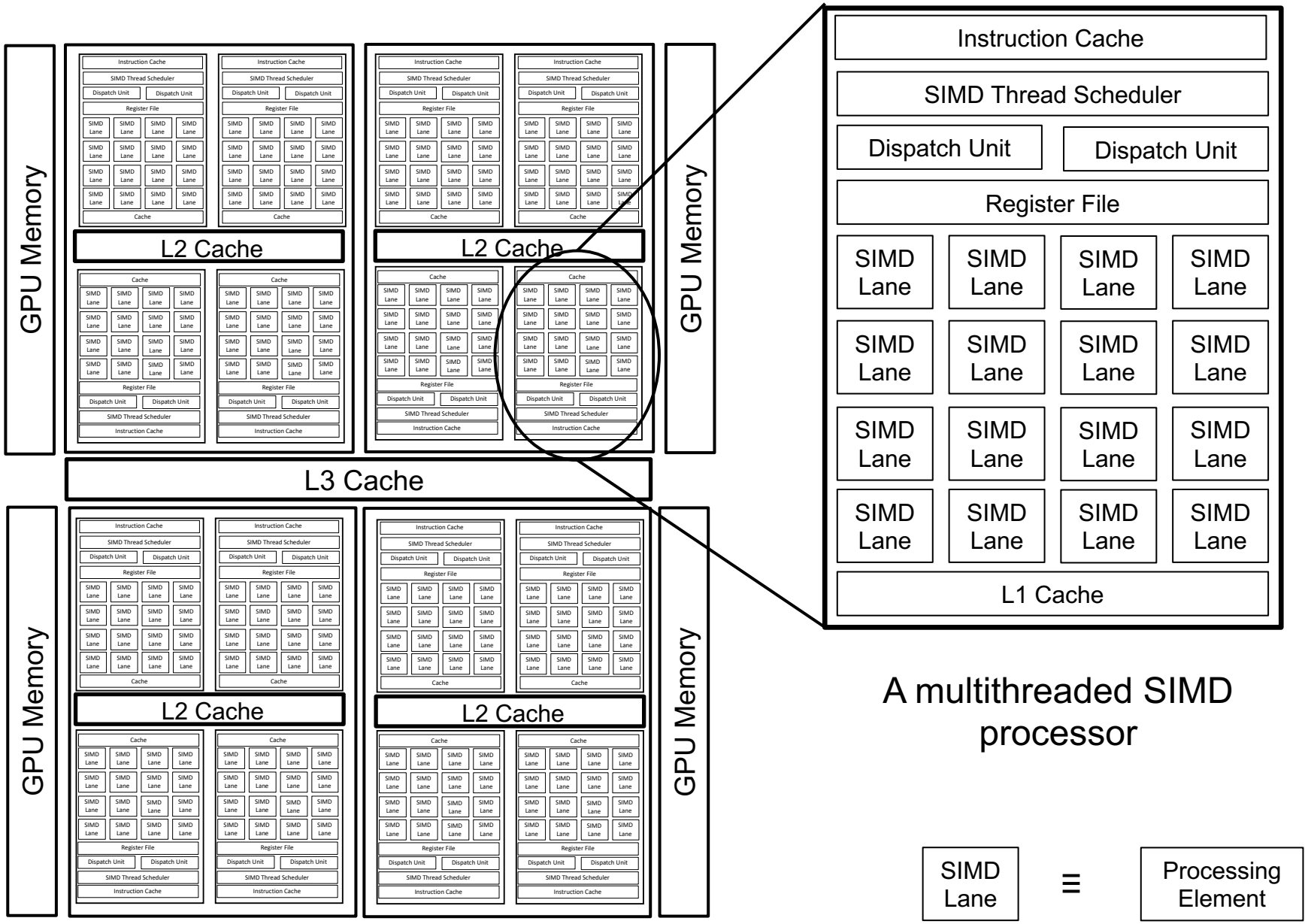
GPU Programming: A Host/Device Platform Model



- One ***Host*** and one or more ***Devices***
  - Each Device is composed of one or more Compute Units
  - Each Compute Unit is divided into one or more ***Processing Elements***

- Memory divided into ***host memory*** and ***device memory***

A Generic GPU (following Hennessey and Patterson)



A multithreaded SIMD processor

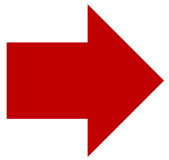
SIMD Lane ≡ Processing Element ≡ CUDA Core

Computer Architecture: A Quantitative Approach, John L. Hennessy and David A. Patterson.

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];

}
```



Data Parallel 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

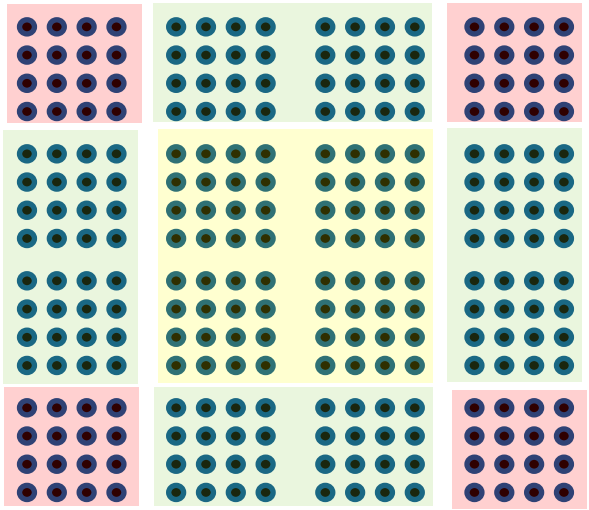
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Write kernel code for the scalar work-items

```
// 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 <<< dimGrid,dimBlock>>> (a, b, c, N);
}
```

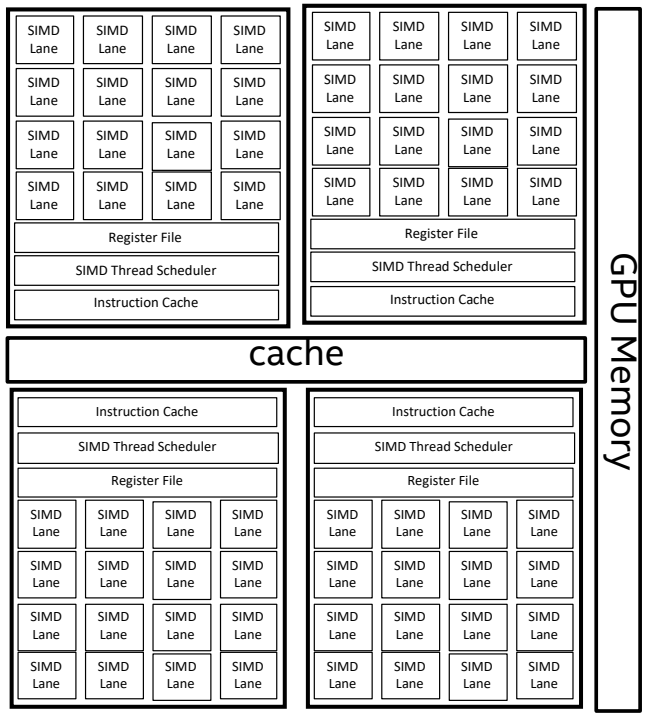
This is CUDA code

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



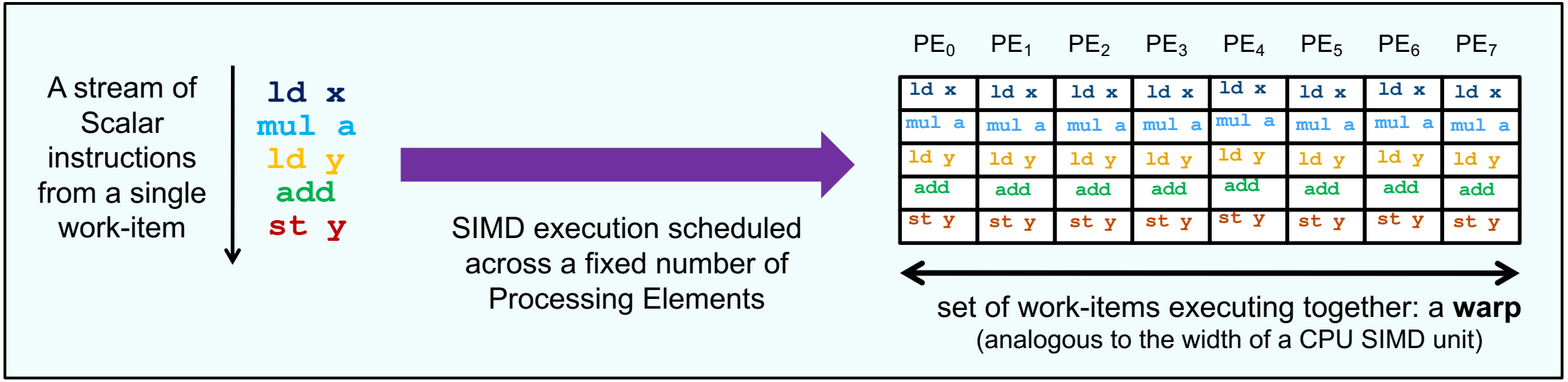
3. Map data structures onto the same index space

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



SIMT: One instruction stream maps onto many Processing Elements

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



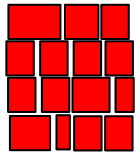
GPU nomenclature is really messed up. (sorry about that … we tried to unify around OpenCL but failed).

Instruction stream at finest grain	**Work-item**, CUDA Thread
Blocks for scheduling work-items	**work-group**, thread block
Execution width for work-items	Subgroup, **warp**
Finest grained processing element (PE) in a GPU	SIMD Lane, **Processing Element**, CUDA Core
Block of PEs driven by a single Instruction sequencer	multithreaded SIMD processor, compute unit, **Streaming multiprocessor**

These names are particularly awful since they conflict with established names from CPU Computing.

Executing a program on CPUs and GPUs

Work decomposed into blocks



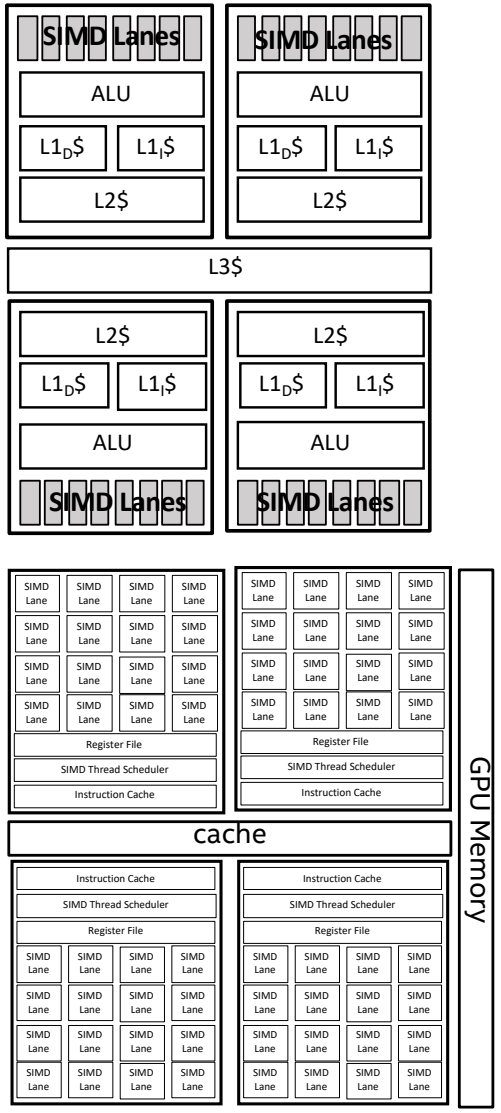
For a CPU

For a GPU

Program defines work

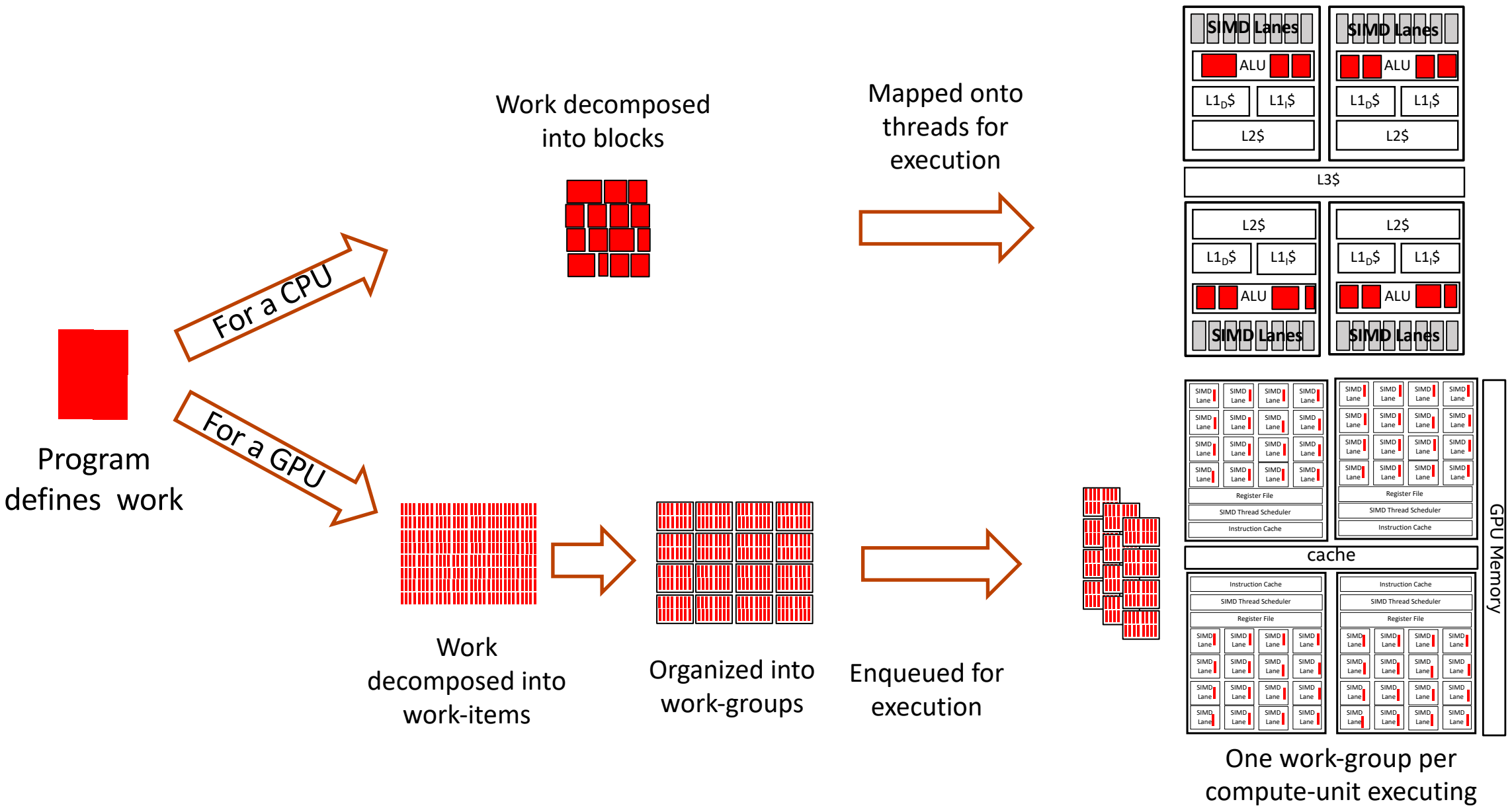
Work decomposed into work-items

Organized into work-groups



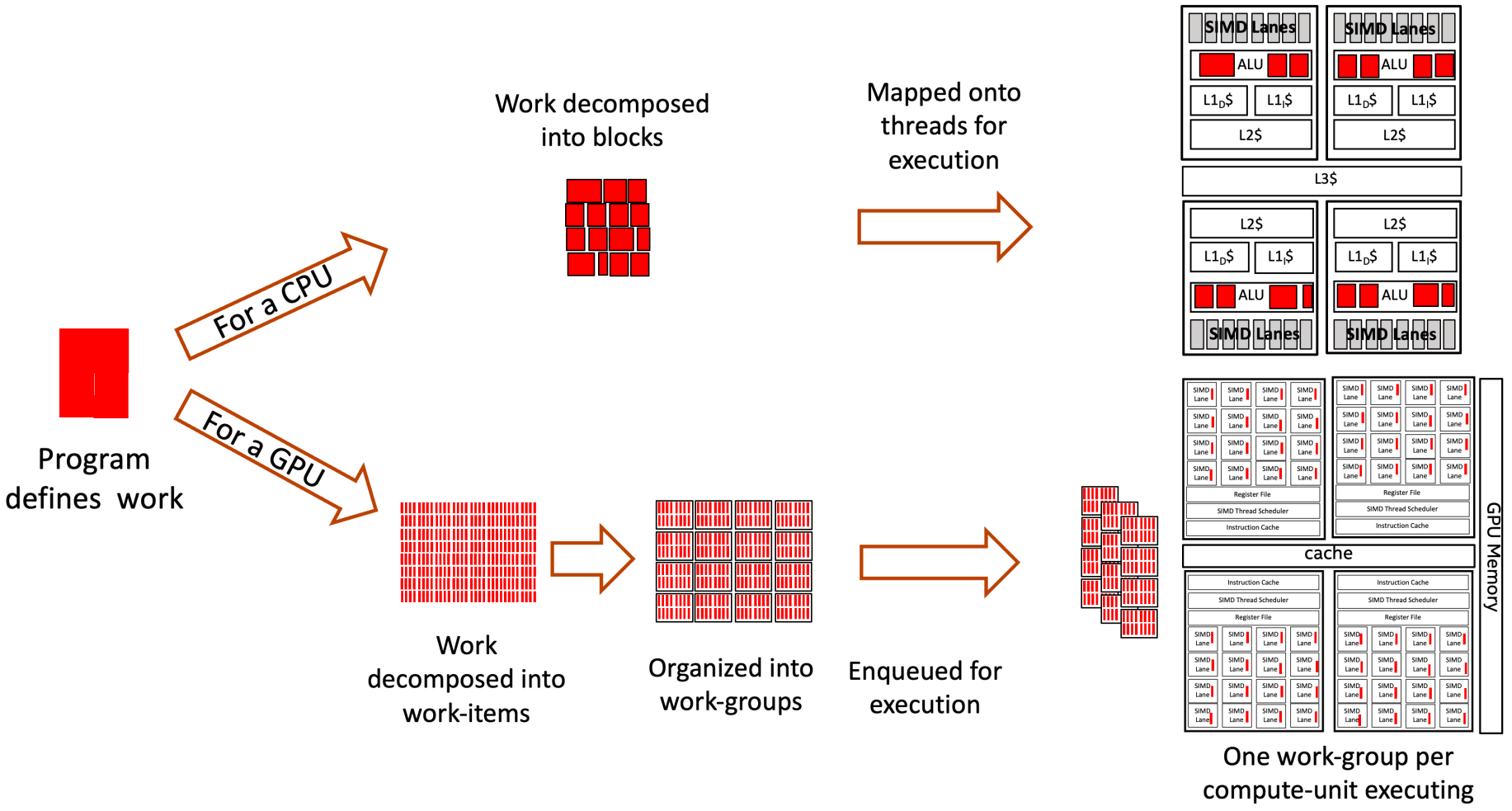
One work-group per compute-unit executing

Executing a program on CPUs and GPUs



CPU/GPU execution models

Executing a program on CPUs and GPUs



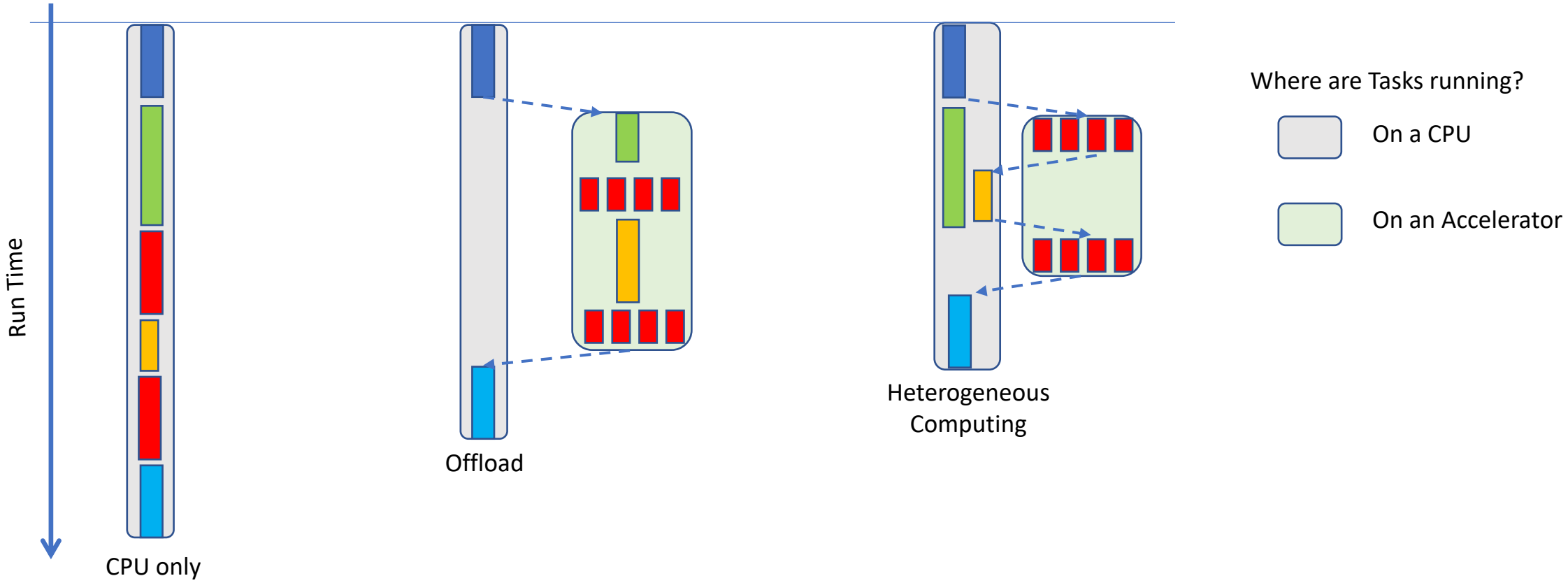
For a CPU, the threads are all active and able to make forward progress. Optimized for latency sensitive problems

For a GPU, any given work-group might be in the queue waiting to execute. Optimized for high throughput workflows

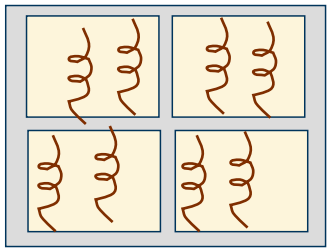
No single processor is best at everything

- 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.

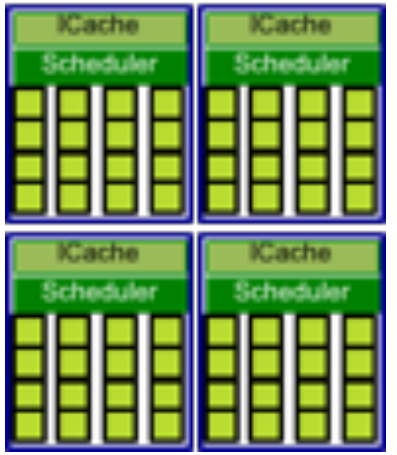


Hardware is diverse … and its only getting worse!!!



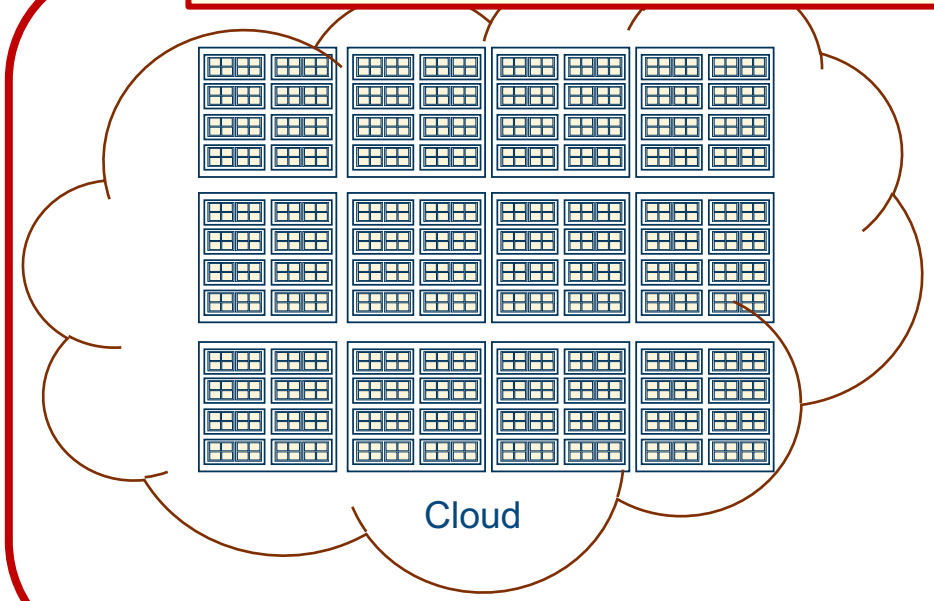
CPU

SIMD/Vector

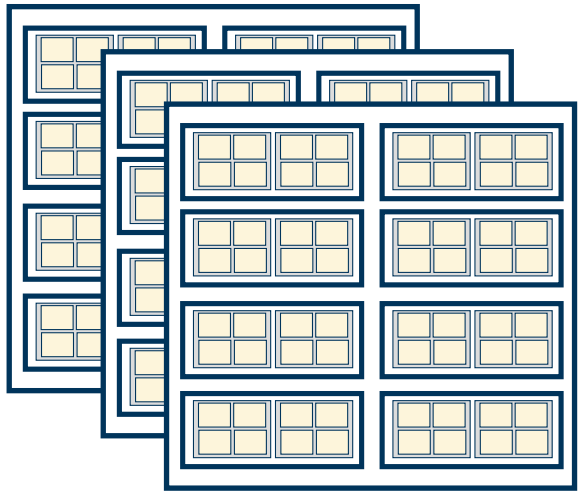


GPU

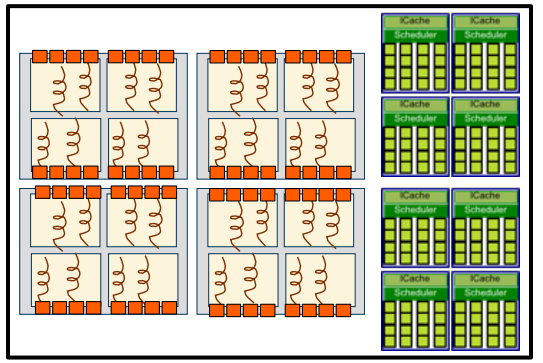
We'll now consider parallel programming with distributed memories



Cloud

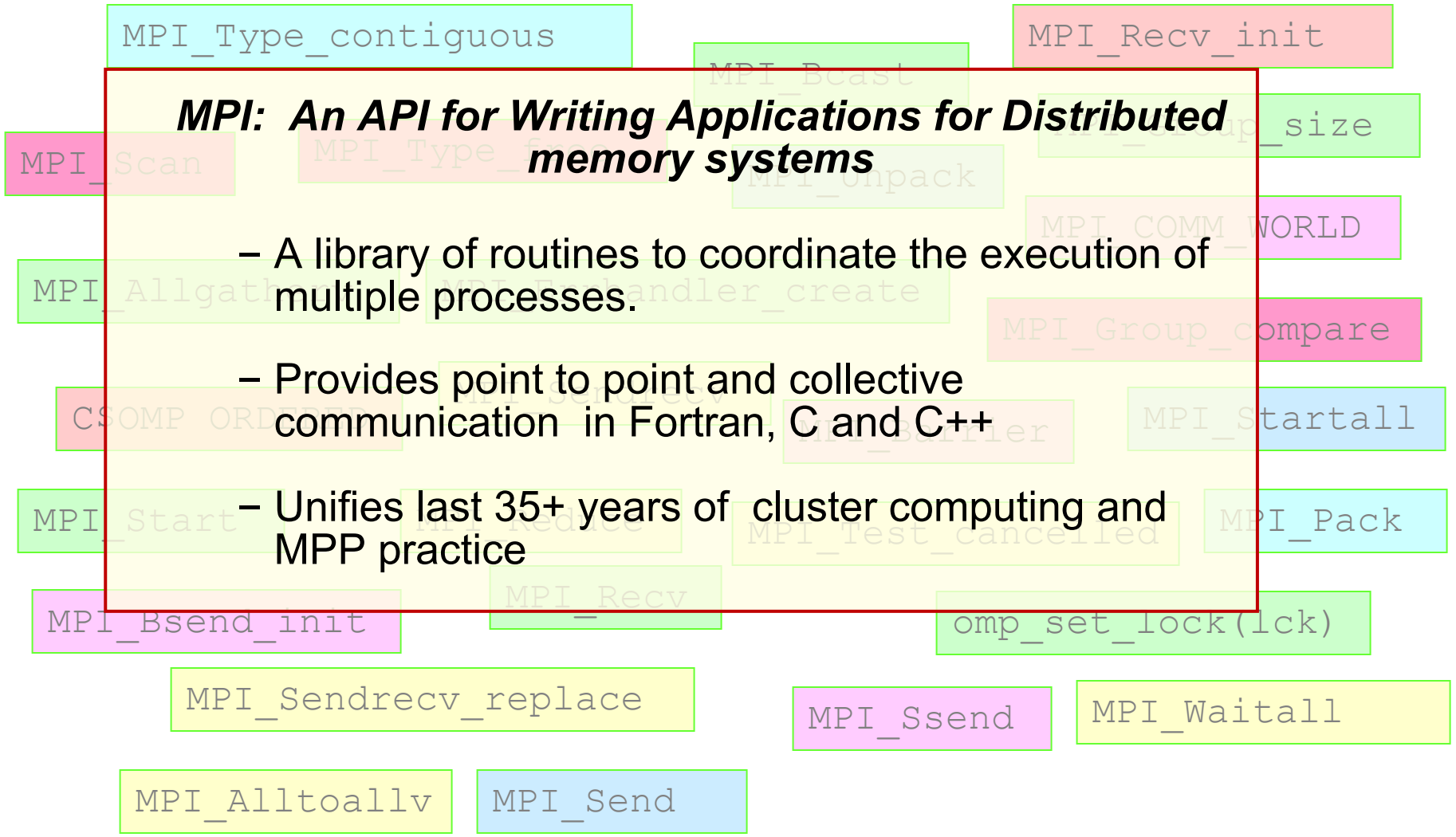


Cluster



Heterogeneous node

Parallel API's: MPI … the Message Passing Interface



MPI: An API for Writing Applications for Distributed memory systems

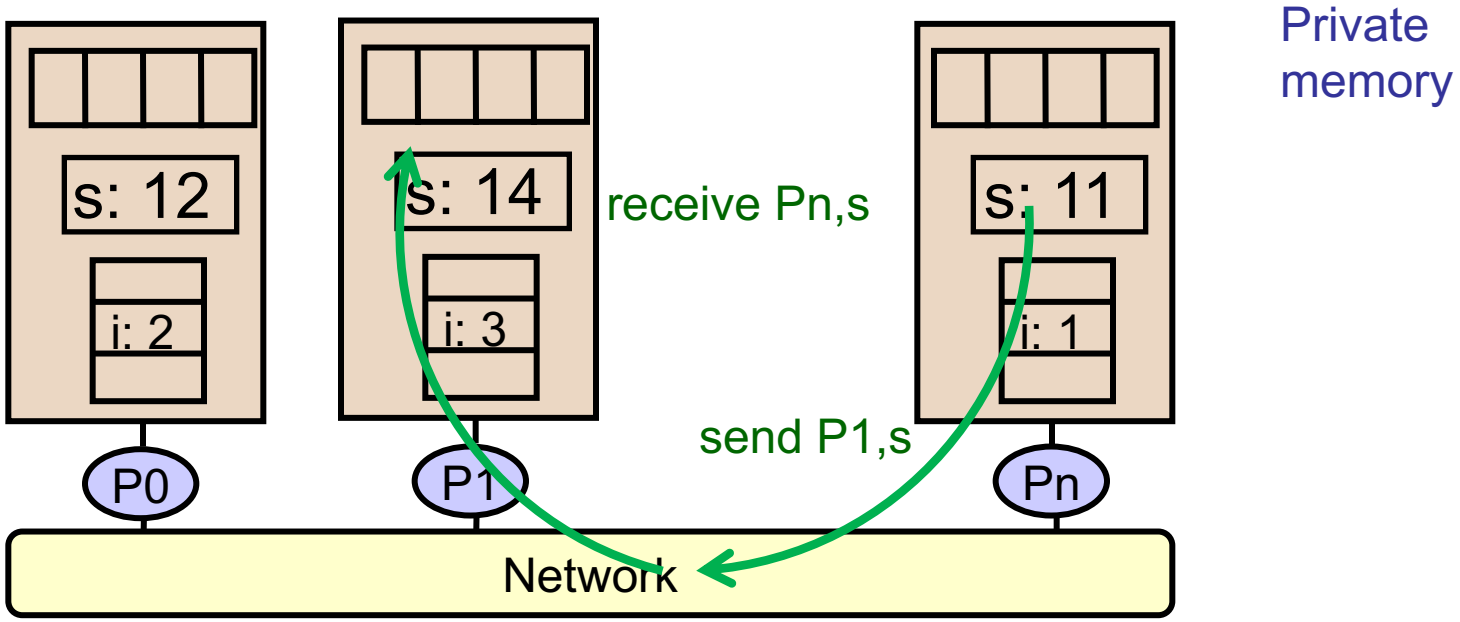
- A library of routines to coordinate the execution of multiple processes.
- Provides point to point and collective communication in Fortran, C and C++
- Unifies last 35+ years of cluster computing and MPP practice

MPP: Massively Parallel Processors

Programming Model: Message Passing

- Program consists of a collection of processes.
  - **Number of processes almost always fixed at program startup time**
  - **Local address space per node -- NO physically shared memory.**
  - **Logically shared data is partitioned over local processes.**

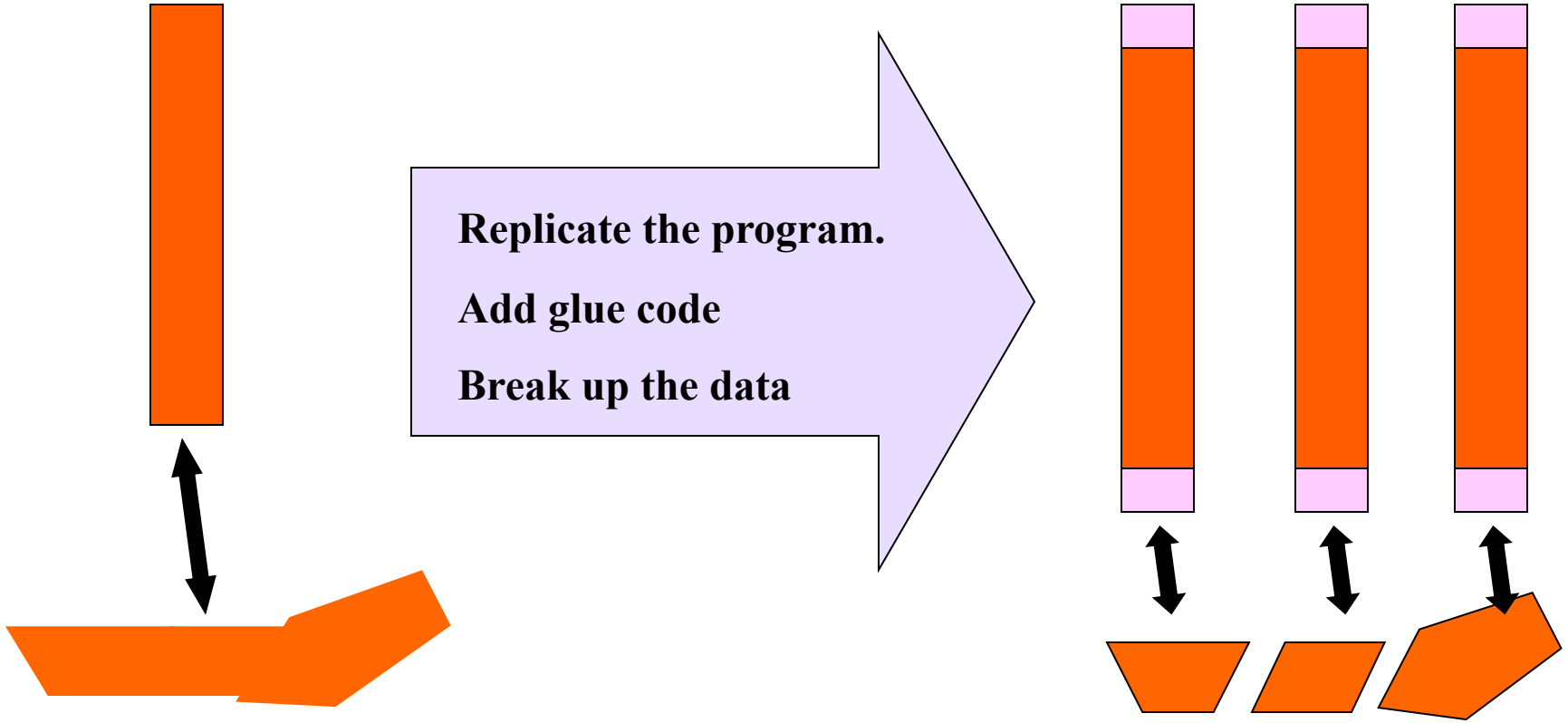
- Processes communicate by explicit send/receive pairs
  - **Synchronization is implicit by communication events.**
  - **MPI (Message Passing Interface) is the most commonly used API**



How do people use MPI?
The SPMD Design Pattern

A sequential program working on a data set

- A single program working on a decomposed data set.
- Use Node ID and numb of nodes to split up work between processes
- Coordination by passing messages.

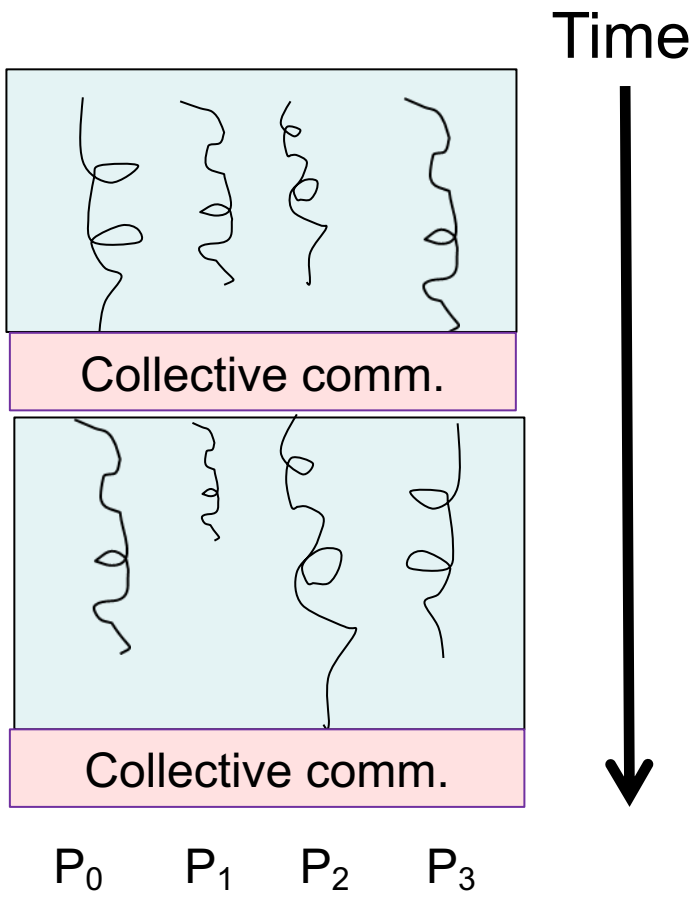


Bulk Synchronous Programming (BSP):

A common design pattern used with MPI Programs

BSP includes the map-reduce pattern commonly used in the cloud. It's easy to use and surprisingly useful

- Many MPI applications have few (if any) sends and receives. 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 of the SPMD pattern sometimes referred to as the Bulk Synchronous pattern.

Example: finite difference methods

- Solve the heat diffusion equation in 1 D:
  - u(x,t) describes the temperature field
  - We set the heat diffusion constant to one
  - Boundary conditions, constant u at endpoints.

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial t}$$

- map onto a mesh with stepsize h and k

$$x_i = x_0 + ih \qquad t_i = t_0 + ik$$

- Central difference approximation for spatial derivative (at fixed time)

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{j+1} - 2u_j + u_{j-1}}{h^2}$$

- Time derivative at $t = t^{n+1}$

$$\frac{du}{dt} = \frac{u^{n+1} - u^n}{k}$$

Example: Explicit finite differences

- Combining time derivative expression using spatial derivative at t = tn

$$\frac{u_j^{n+1} - u_j^n}{k} = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2}$$

- Solve for u at time n+1 and step j

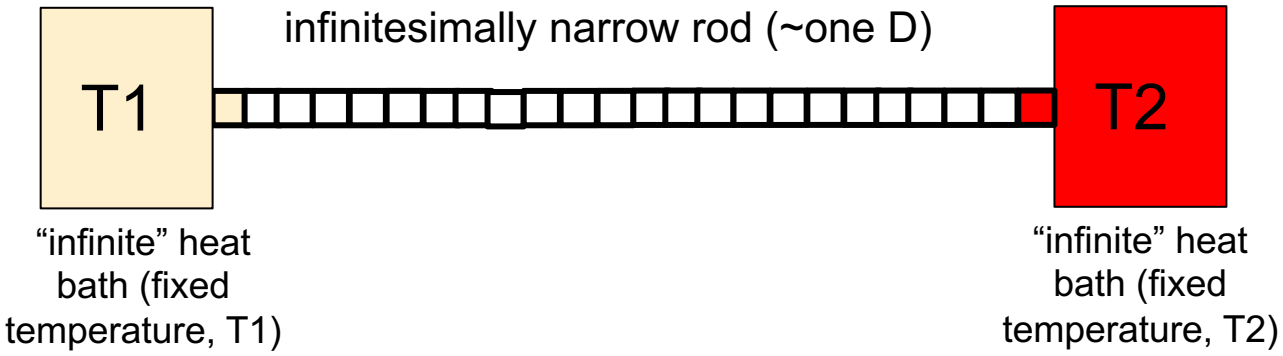
$$u_j^{n+1} = (1-2r)u_j^n + ru_{j-1}^n + ru_{j+1}^n \qquad r = k/h^2$$

- The solution at t = t_{n+1} is determined explicitly from the solution at t = t_n (assume u[t][0] = u[t][N] = Constant for all t).

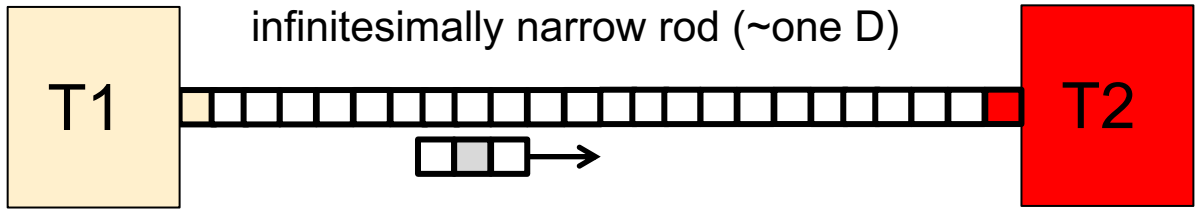
```
for (int t = 0; t < N_STEPS-1; ++t)
   for (int x = 1; x < N-1; ++x)
         u[t+1][x] = u[t][x] + r*(u[t][x+1] - 2*u[t][x] + u[t][x-1]);
```

- Explicit methods are easy to compute … each point updated based on nearest neighbors. Converges for r<1/2.

Heat Diffusion equation



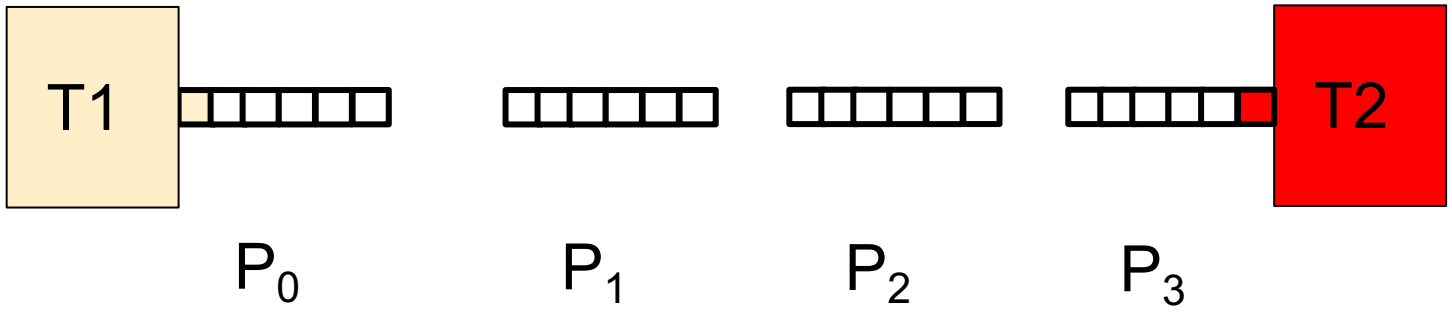
Heat Diffusion equation



Pictorially, you are sliding a three point "stencil" across the domain (u) and updating the center point at each stop.

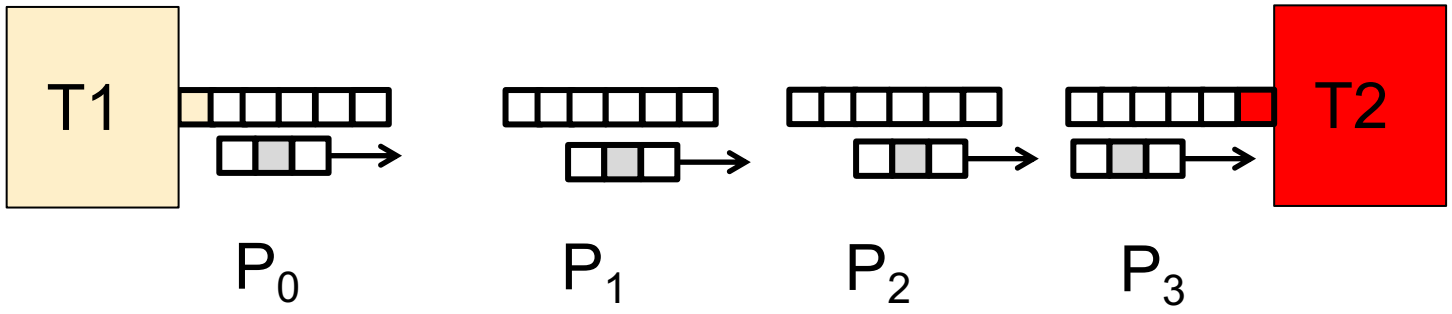
Heat Diffusion equation: in parallel

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



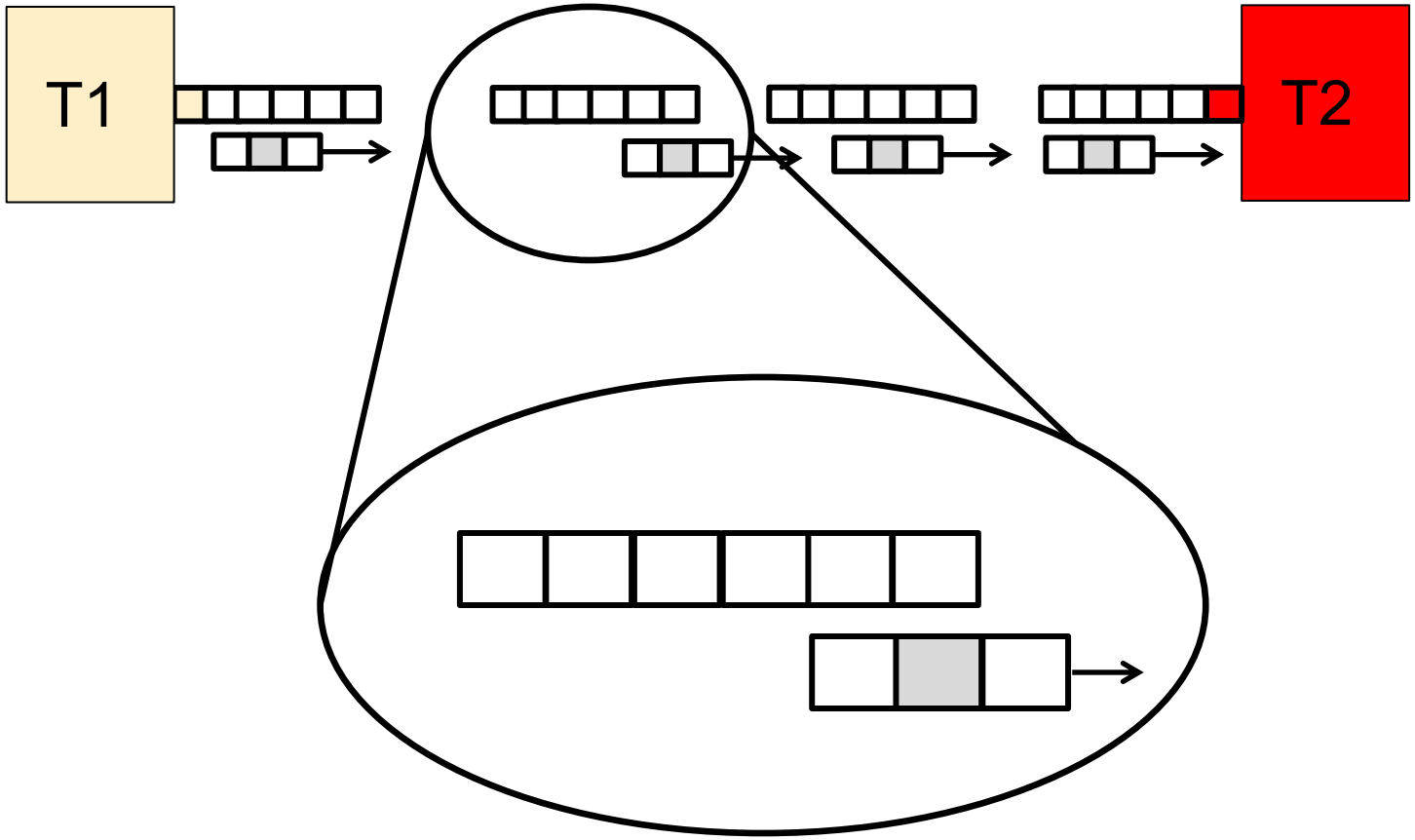
Heat Diffusion equation

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



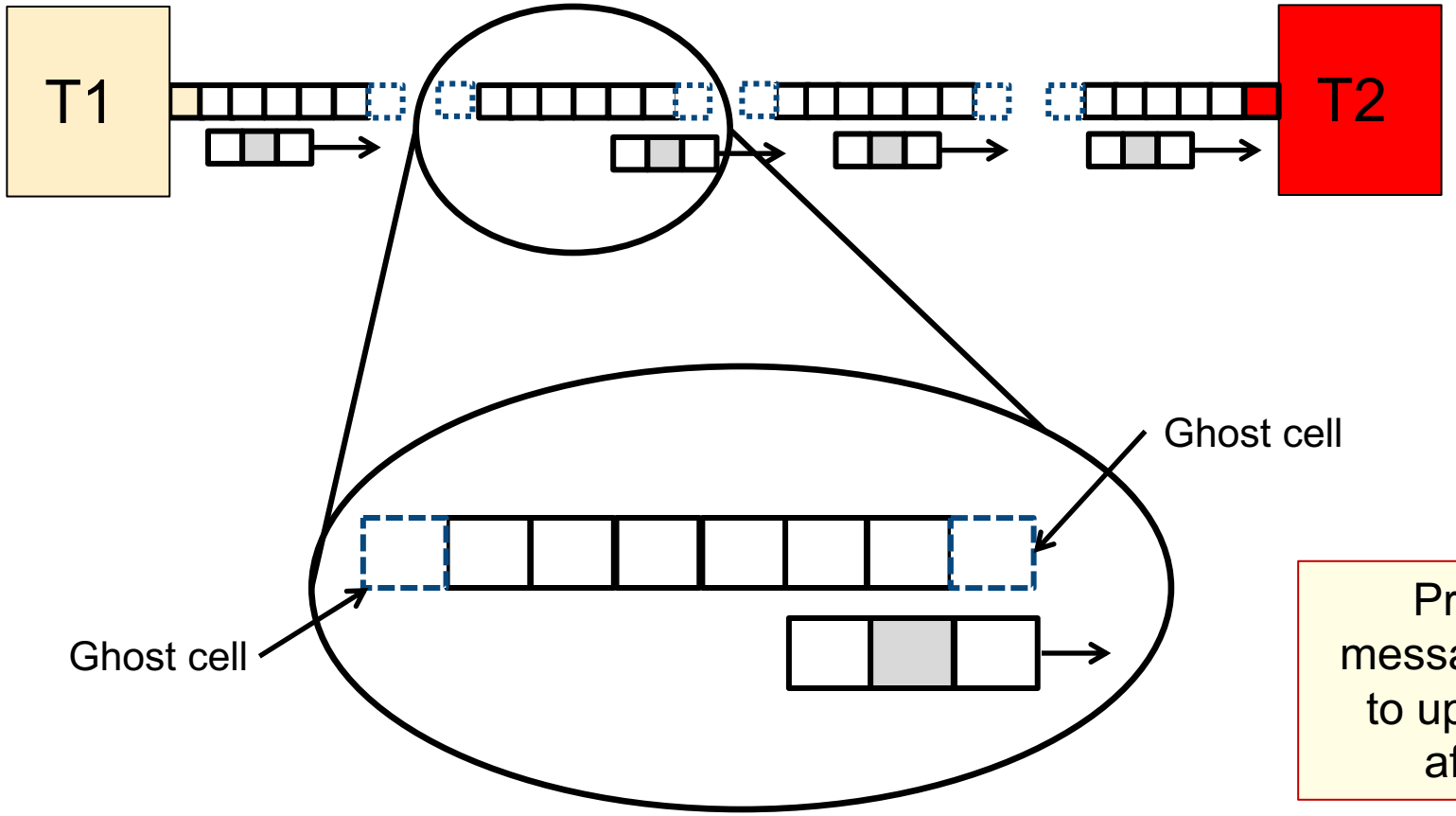
Heat Diffusion equation

- What about the ends of each chunk … where the stencil runs off the end and has 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.



Processes send messages to neighbors to update ghost cells after each step

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 = 1; 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;
```

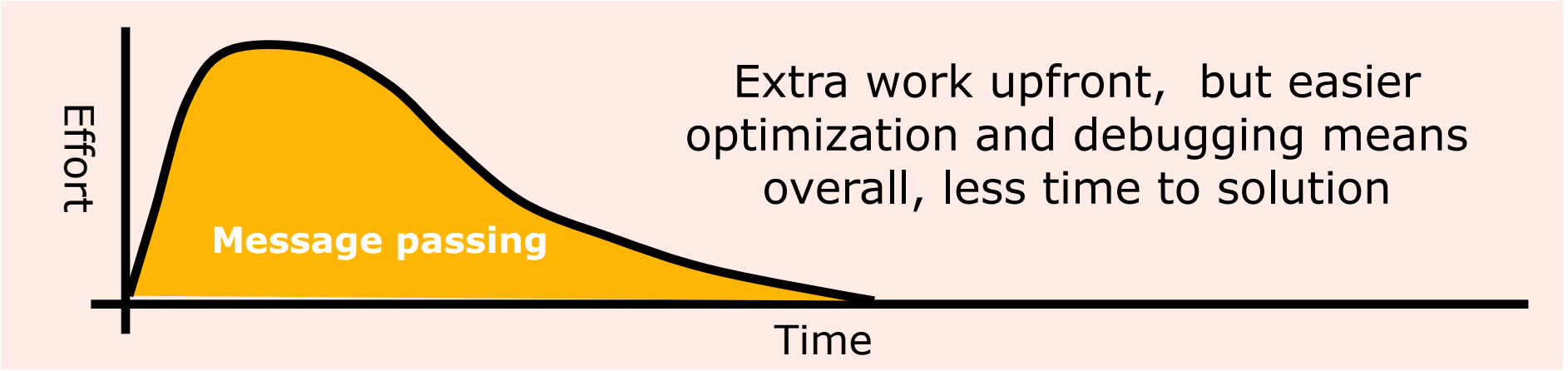
Exchange ghost cells

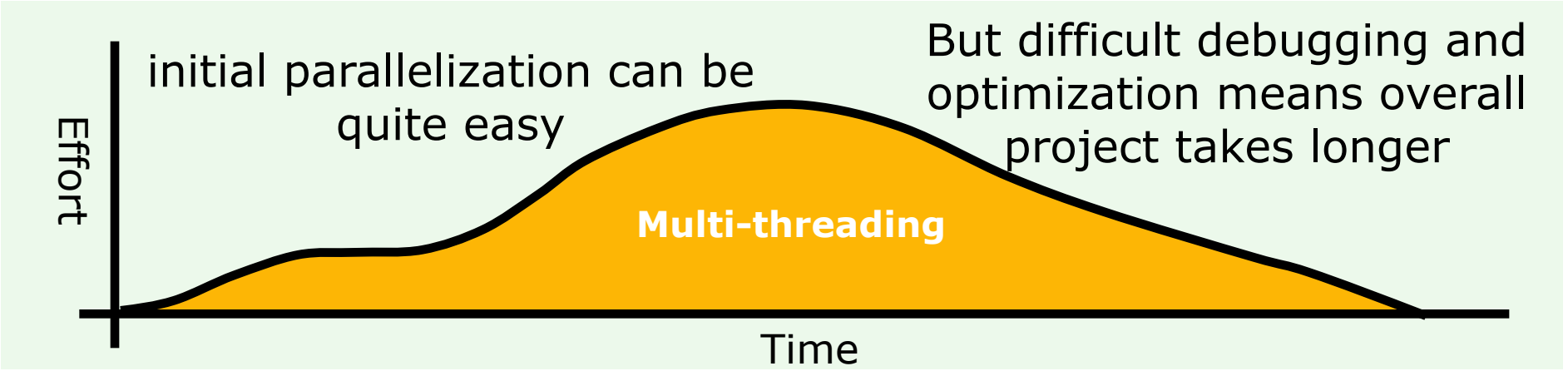
Do the computation, but keep track of boundary conditions

The code can get a bit ugly as each process figures out who to send to and who to receive from.

… but the concept is straightforward and eventually you get used to it.

Does a shared address space make programming easier?





Proving that a shared address space program using semaphores is race free is an NP-complete problem*

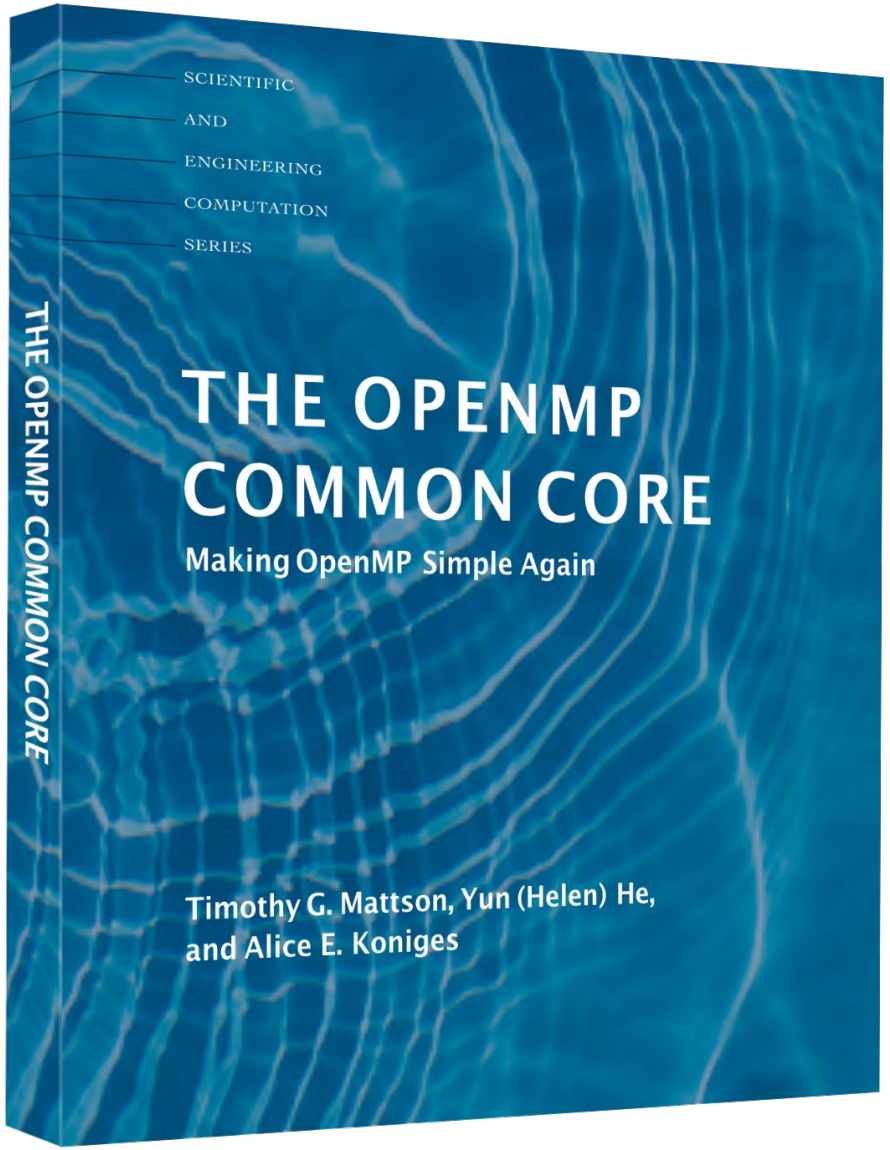
*P. N. Klein, H. Lu, and R. H. B. Netzer, Detecting Race Conditions in Parallel Programs that Use Semaphores, Algorithmica, vol. 35 pp. 321–345, 2003

To learn OpenMP:

- An exciting new book that Covers the Common Core of OpenMP plus a few key features beyond the common core that people frequently use

- It's geared towards people learning OpenMP, but as one commentator put it … **everyone at any skill level should read the memory model chapters**.

- Available from MIT Press



www.ompcore.com for code samples and the Fortran supplement

Books about OpenMP

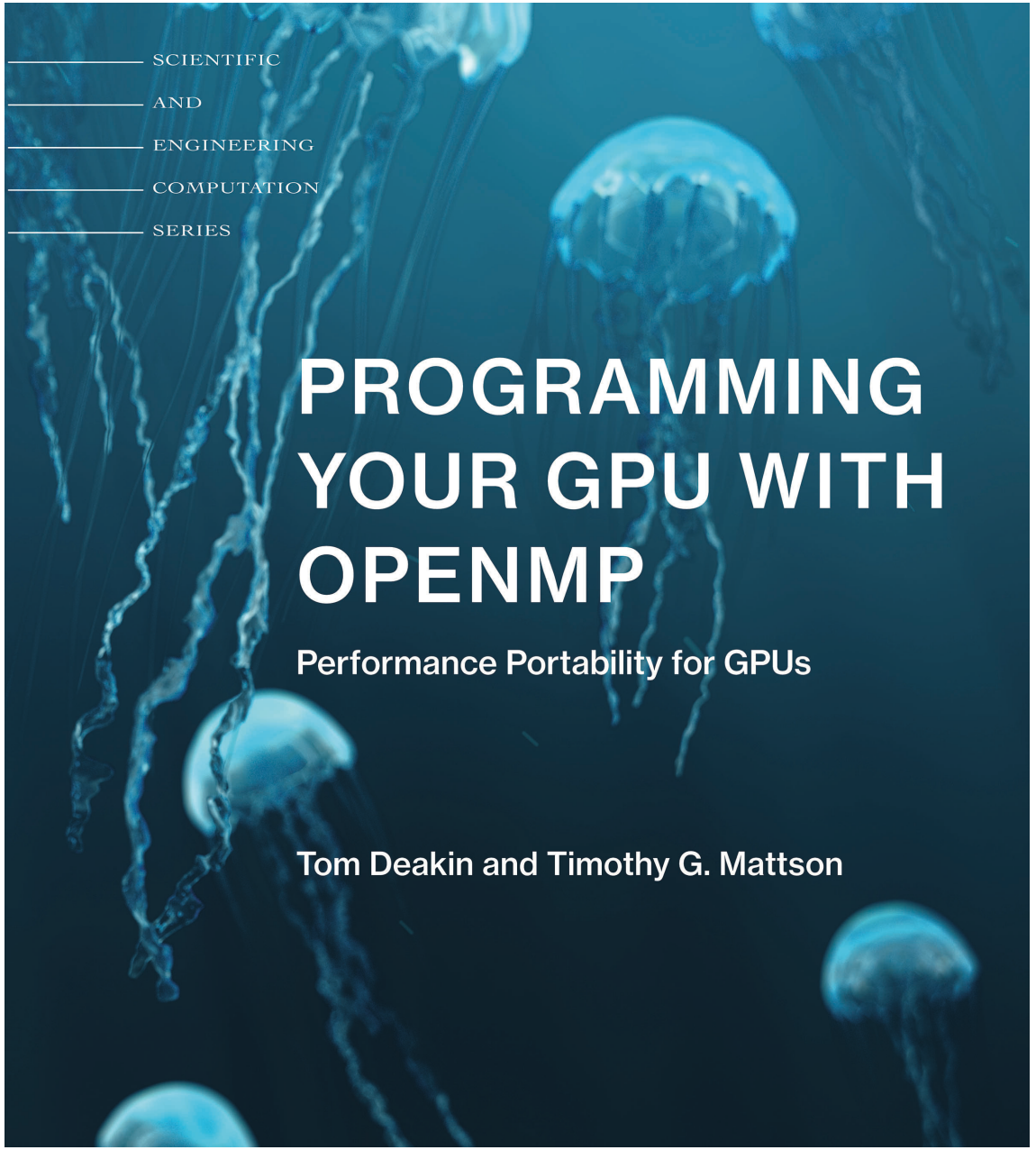
A great book that covers OpenMP features beyond OpenMP 2.5

Books about OpenMP

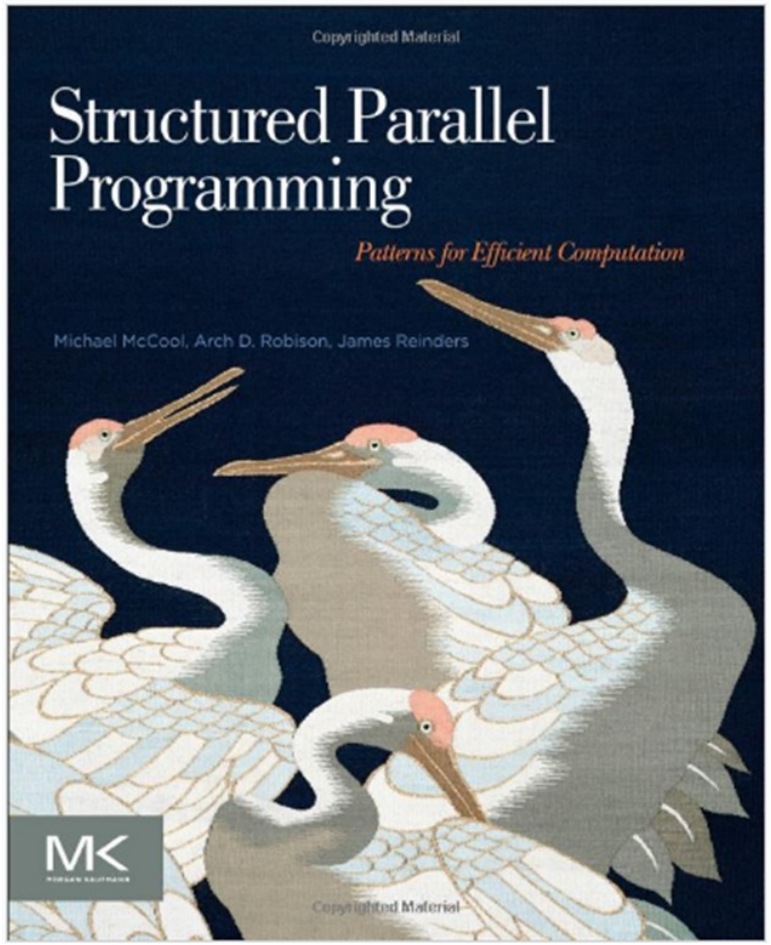
The latest book on OpenMP …

Released in November 2023.

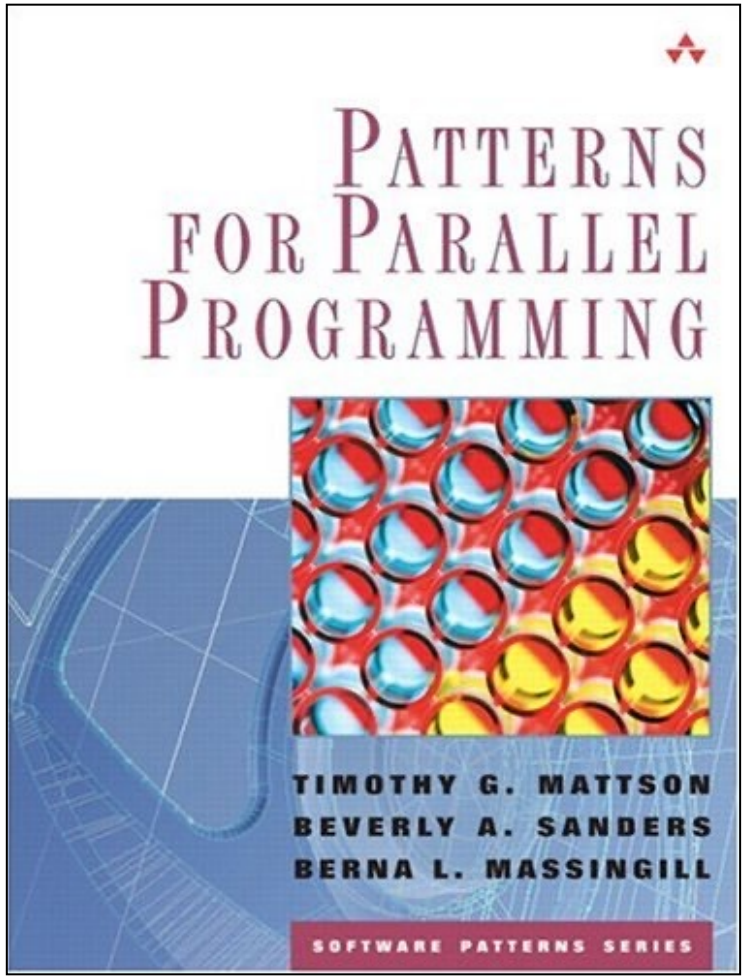
A book about how to use OpenMP to program a GPU.



Background references



A great book that explores key patterns with Cilk, TBB, OpenCL, and OpenMP (by McCool, Robison, and Reinders)



- A book about how to "think parallel" with examples in OpenMP, MPI and java