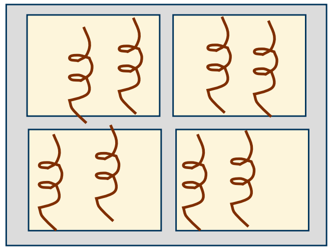
OpenMP®

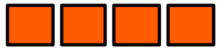
The Parallel Programming World Beyond OpenMP

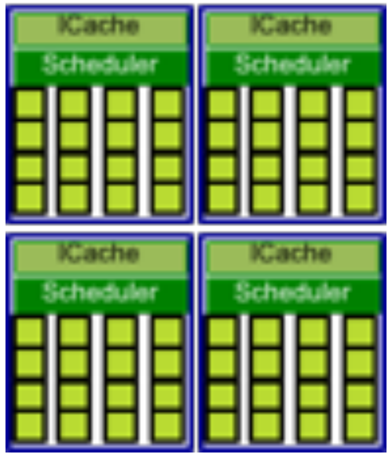
Tim Mattson

Human Learning Group

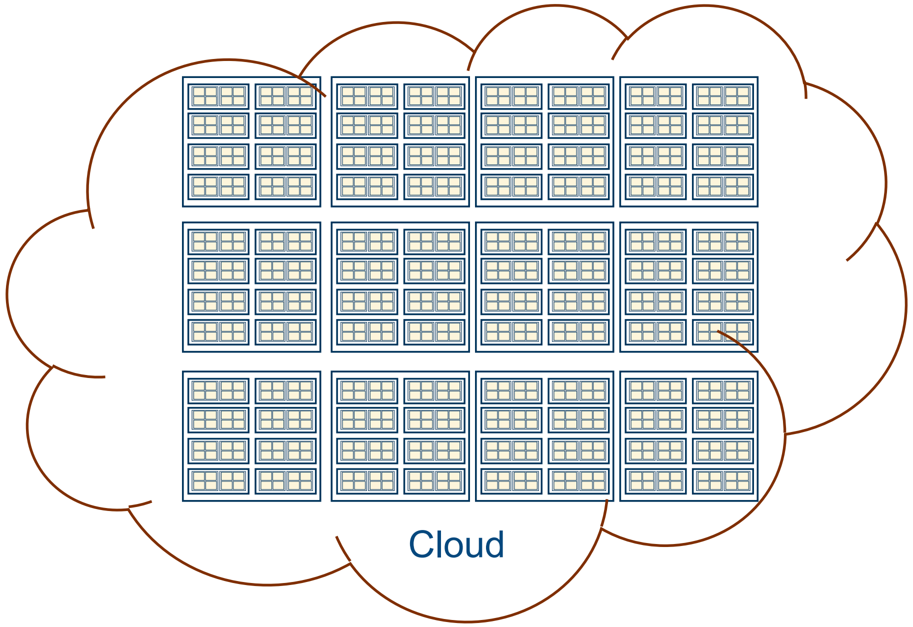
Hardware is diverse … and its only getting worse!!!

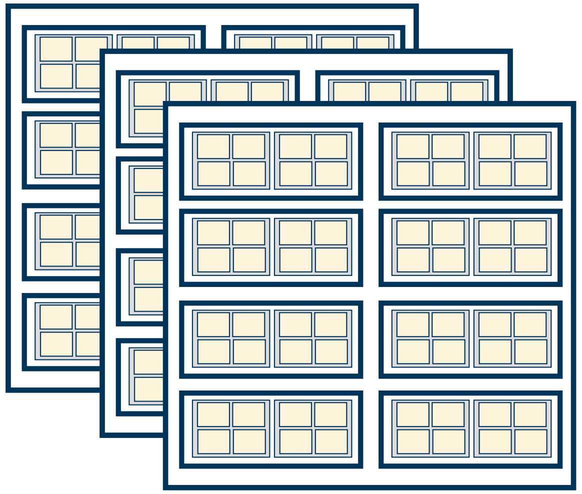

CPU

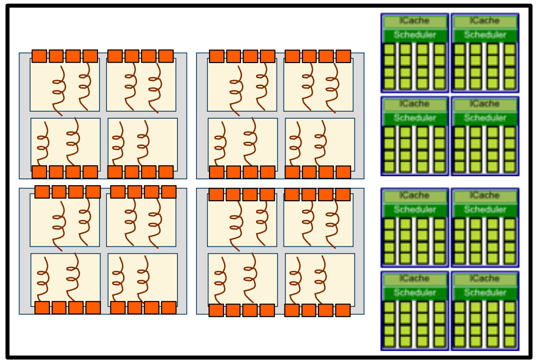

SIMD/Vector



GPU


Cloud


Cluster



Heterogeneous node

The Big Three

- 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)

You are all OpenMP experts and know a great deal about multithreading

- Even if you don't plan to spend much time programming with these systems … a well rounded HPC programmer should know what they are and how they work.

The Big Three

If you don't know MPI, you aren't really an HPC programmer!

- 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 should know what they are and how they work.

Parallel API's: MPI … the Message Passing Interface

MPI_Type_contiguous

MPI_Bcast

MPI_Recv_init

MPI_Group_size

MPI_Scan

MPI_Type_free

MPI_COMM_WORLD

MPI_Allgatherv

MPI_Errhandler_create

MPI_Group_compare

C$OMP ORDERED

MPI_Barrier

MPI_Startall

MPI_Start

MPI_Reduce

MPI_Test_cancelled

MPI_Pack

MPI_Bsend_init

MPI_Recv

p_set_lock(lck)

MPI_Sendrecv_replace

MPI_Ssend

MPI_Waitall

MPI_Alltoallv

MPI_Send

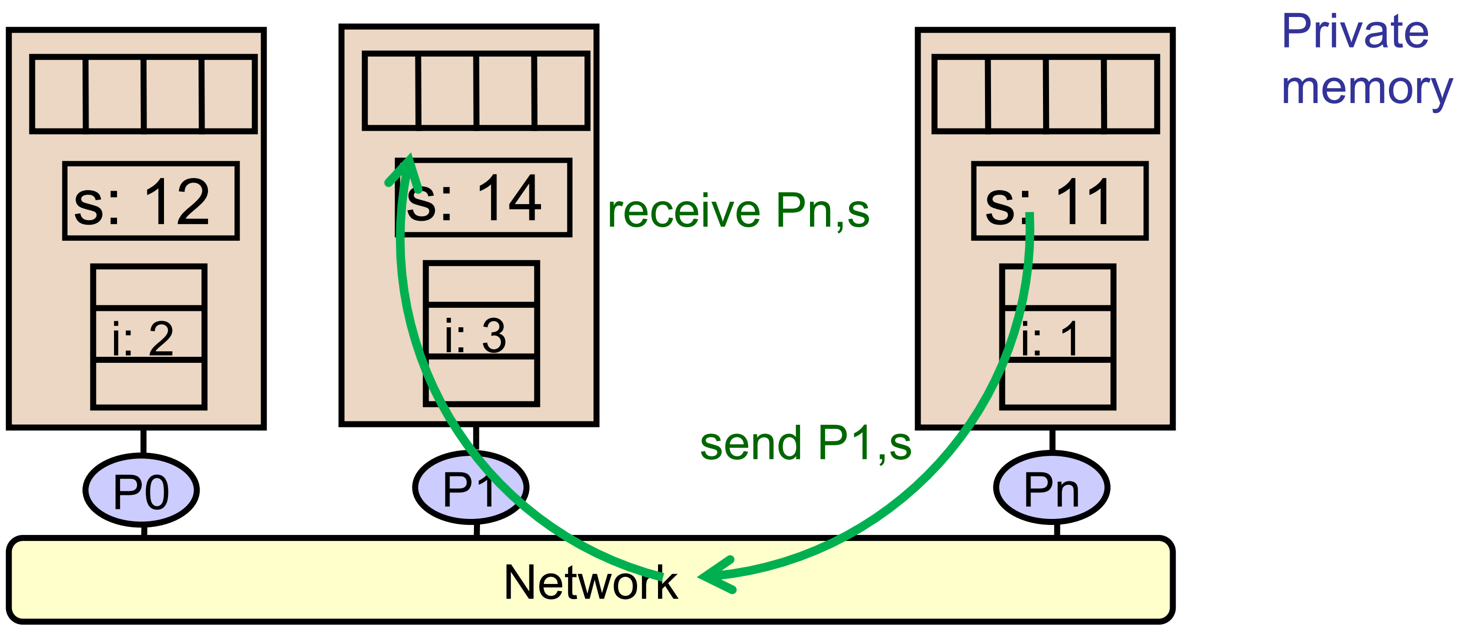
MPI: An API for Writing Clustered Applications

- **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 25+ years of cluster computing and MPP practice**

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.**

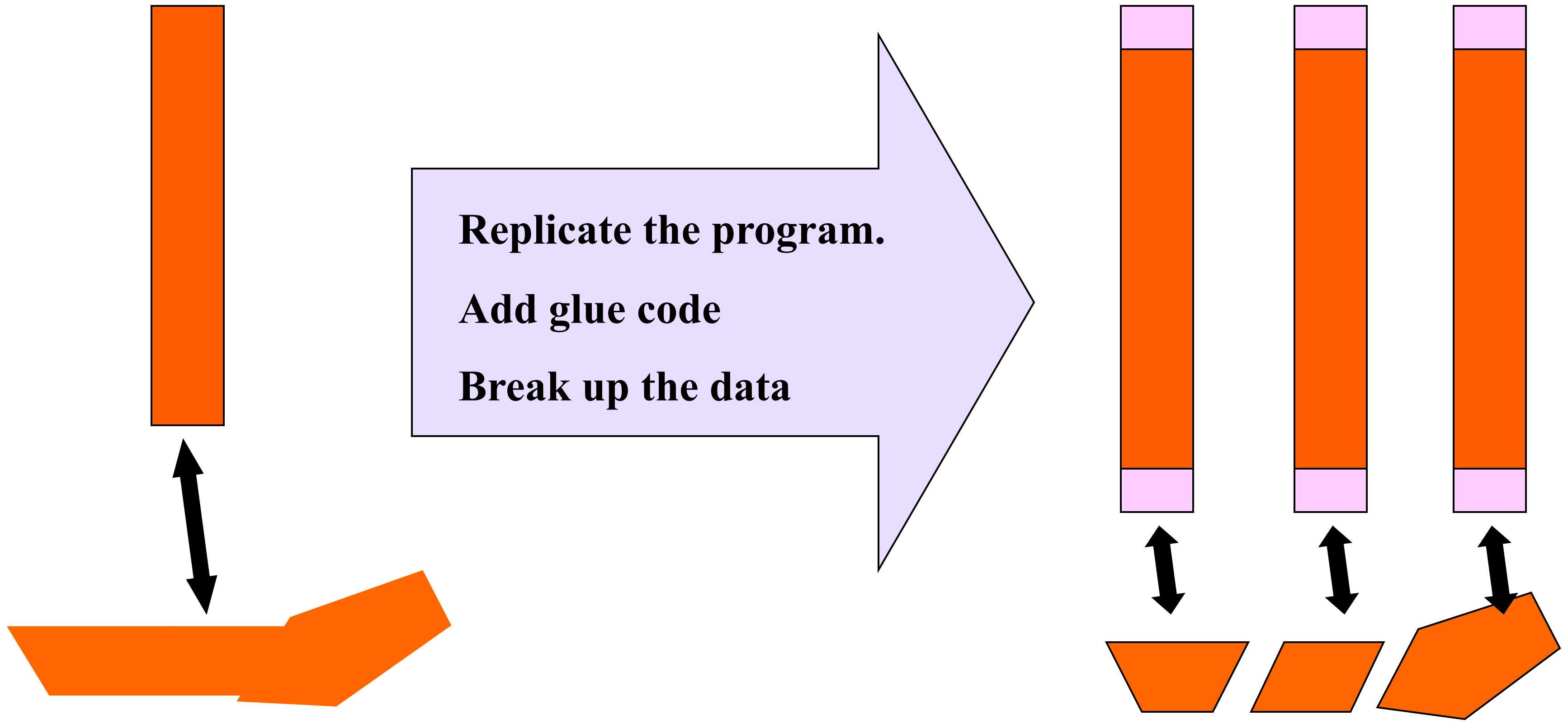
- Processes communicate by explicit send/receive pairs
  - **Synchronization is implicit by communication events.**
  - **MPI (Message Passing Interface) is the most commonly used API**



How do people use MPI? The SPMD Design Pattern

A sequential program working on a data set

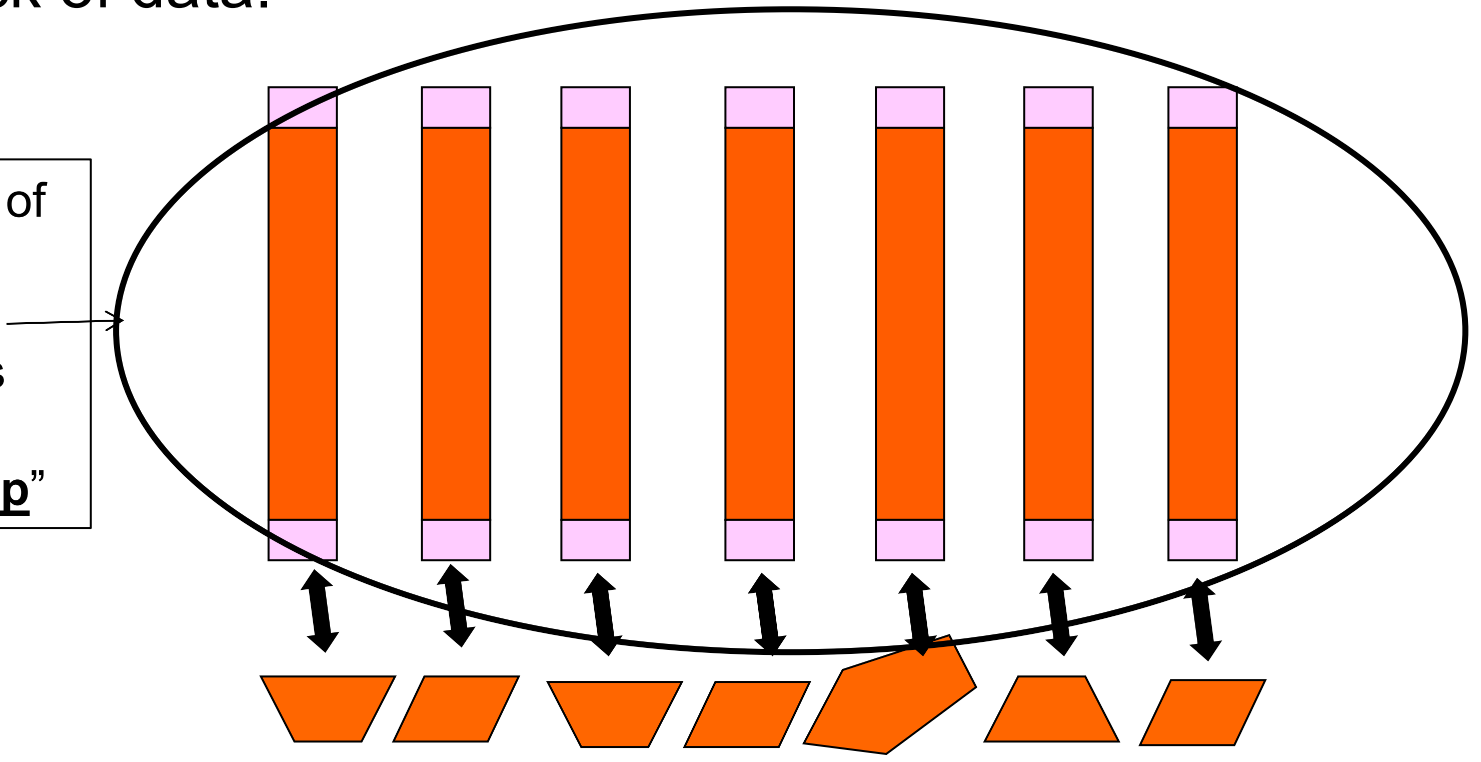
- A single program working on a decomposed data set.
- Use Node ID and numb of nodes to split up work between processes
- Coordination by passing messages.



An MPI program at runtime

- Typically, when you run an MPI program, multiple processes 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 functions work within a "**context**": MPI actions occurring in different contexts, even if they share a process group, cannot interfere with each other.

MPI Hello World

```
#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;
}
```

Initializing and finalizing MPI

```
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()

```
#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_Finalize (void)
```

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

How many processes are involved?

`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

```
#inclu
#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

Which process "am I" (the rank)

```
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"

```
#inclu
#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.

Running the program

- On a 4 node cluster, to run this program (hello):

  ```
  > mpiexec –np 4 –hostfile hostf hello
  ```
- Where “hostf” is a file with the names of the cluster nodes, one to a line.
- Would would this program output?

```
#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;
}
```

Running the 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;
}
```

- On a 4 node cluster, to run this program (hello):

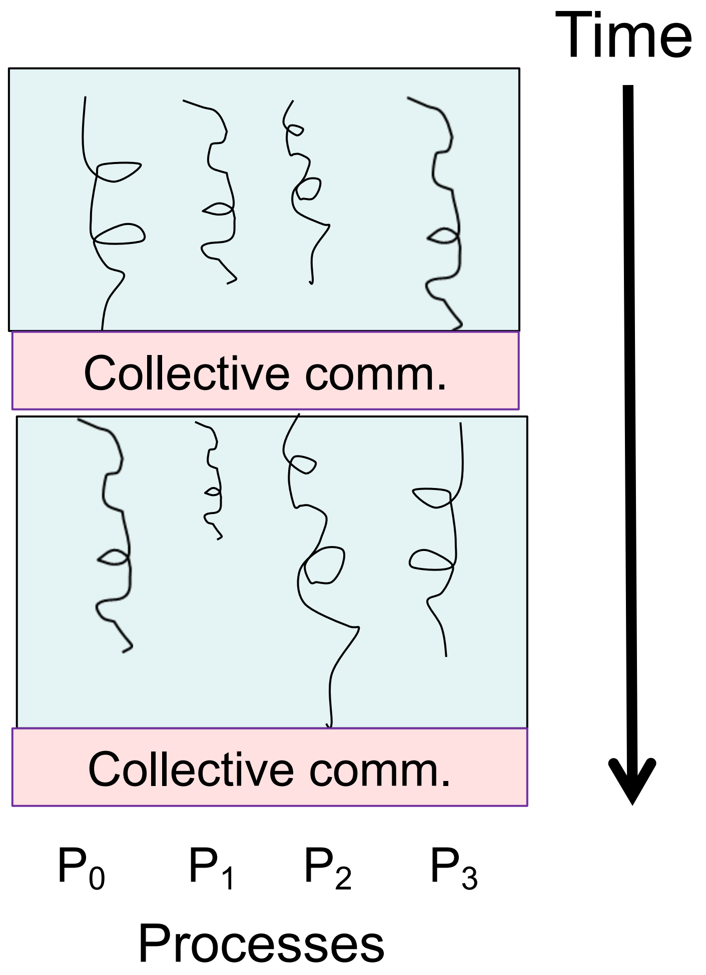
```
> mpiexec –np 4 –hostfile hostf hello
Hello from process 1 of 4
Hello from process 2 of 4
Hello from process 0 of 4
Hello from process 3 of 4
```

- Where “hostf” is a file with the names of the cluster nodes, one to a line.

Bulk Synchronous Programming:

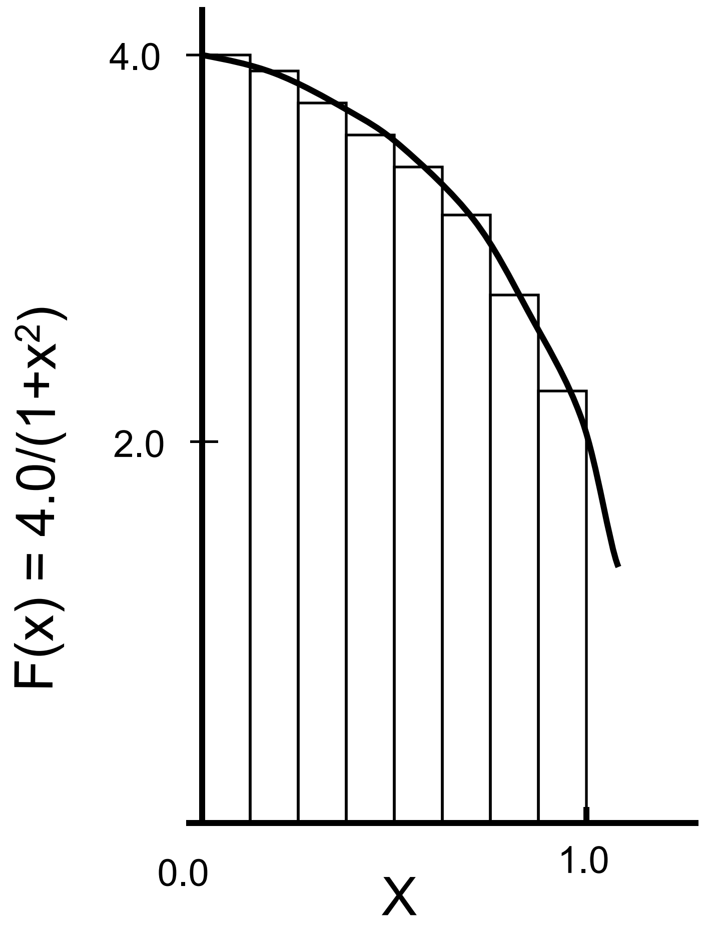
A common design pattern used with MPI Programs

- 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 or the SPMD pattern sometimes referred to as the Bulk Synchronous pattern.

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.

PI Program: an example

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

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

Pi program in MPI … using the BSP pattern

```
#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

Reduction

```
int MPI_Reduce (void* sendbuf,
        void* recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op,
        int root, MPI_Comm comm)
```

- **`MPI_Reduce`** performs specified reduction operation on specified data from all processes in communicator, places result in process “root” only.
- **`MPI_Allreduce`** places result in all processes (avoid unless necessary)

Operation	Function
`MPI_SUM`	Summation
`MPI_PROD`	Product
`MPI_MIN`	Minimum value
`MPI_MINLOC`	Minimum value and location
`MPI_MAX`	Maximum value
`MPI_MAXLOC`	Maximum value and location
`MPI_LAND`	Logical AND

Operation	Function
`MPI_BAND`	Bitwise AND
`MPI_LOR`	Logical OR
`MPI_BOR`	Bitwise OR
`MPI_LXOR`	Logical exclusive OR
`MPI_BXOR`	Bitwise exclusive OR
User-defined	It is possible to define new reduction operations

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 is a user-defined integer to help the receiver keep track of different messages from a single source

MPI_Send (buff, 100, MPI_DOUBLE, Dest, tag, MPI_COMM_WORLD);

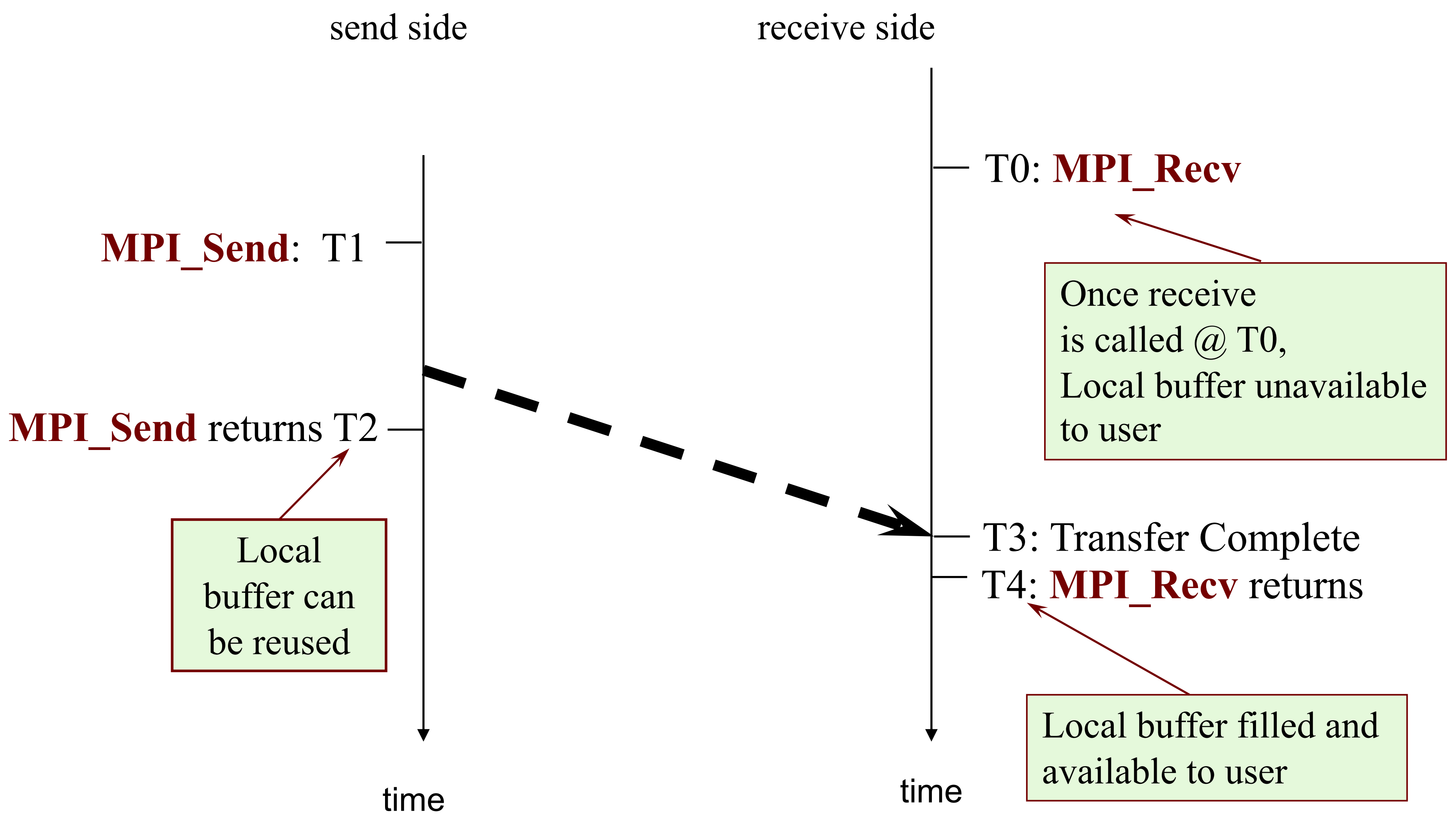
Buffer address   count   Datatype   tag

MPI_Recv (buff, 100, MPI_DOUBLE, Src, tag, MPI_COMM_WORLD, &status);

Rank of Source node

Blocking Send-Receive Timing Diagram

(MPI functions return when local buffer can be used again)

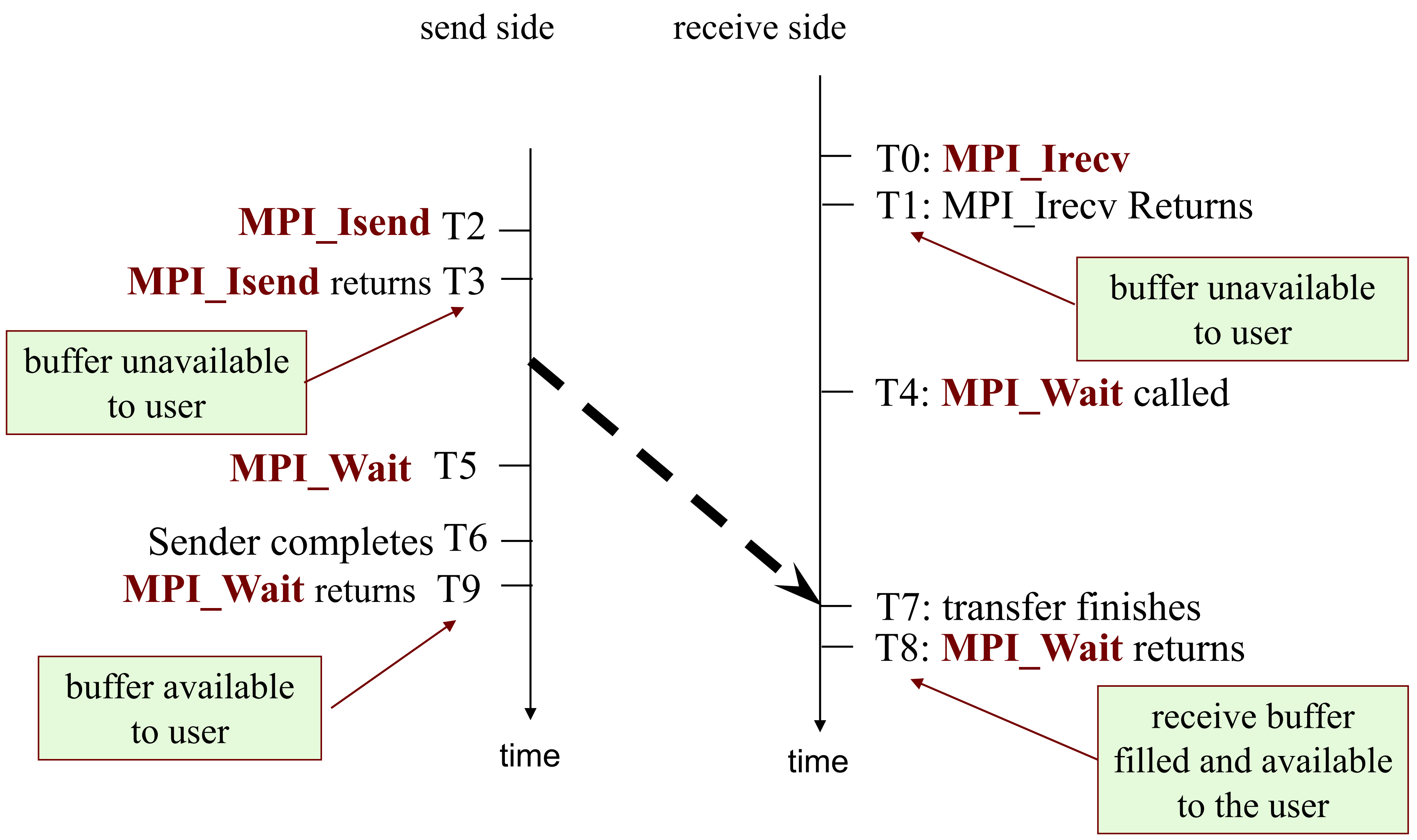


send side
receive side
T0: MPI_Recv
MPI_Send: T1
Once receive is called @ T0, Local buffer unavailable to user
MPI_Send returns T2
Local buffer can be reused
T3: Transfer Complete
T4: MPI_Recv returns
Local buffer filled and available to user
time
time

It is important to post the receive before sending, for highest performance.

Non-Blocking Send-Receive Diagram

(MPI functions return immediately)



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

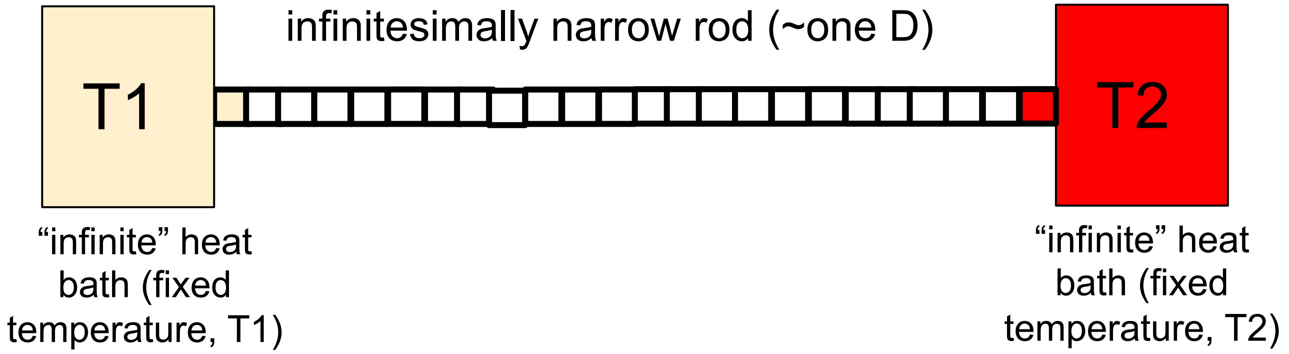
$$u_j^{n+1} = (1-2r)u_j^n + ru_{j-1}^n + ru_{j+1}^n \qquad r = k/h^2$$

- 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).

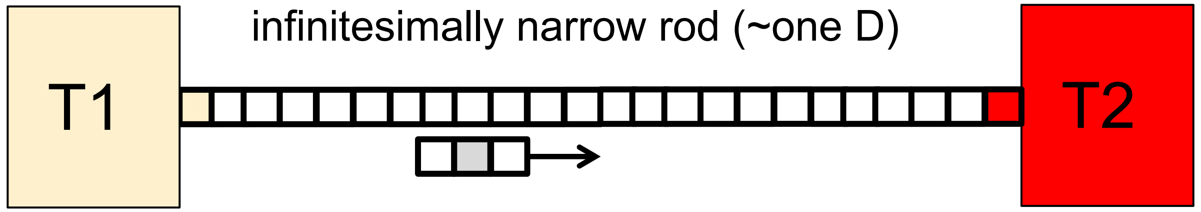
```
for (int t = 0; t < N_STEPS-1; ++t)
   for (int x = 1; x < N-1; ++x)
         u[t+1][x] = u[t][x] + r*(u[t][x+1] - 2*u[t][x] + u[t][x-1]);
```

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

Heat Diffusion equation

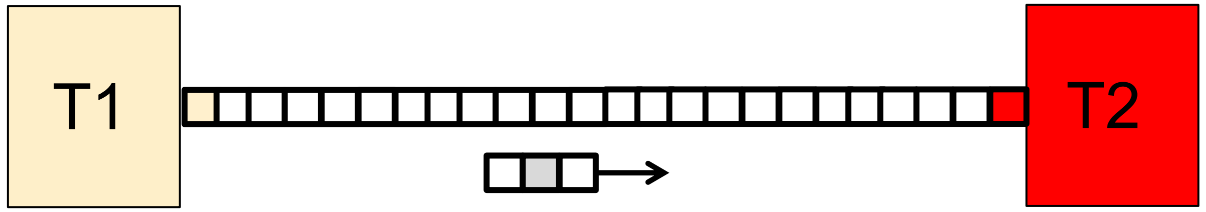

infinitesimally narrow rod (~one D)
T1
T2
“infinite” heat bath (fixed temperature, T1)
“infinite” heat bath (fixed temperature, T2)

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

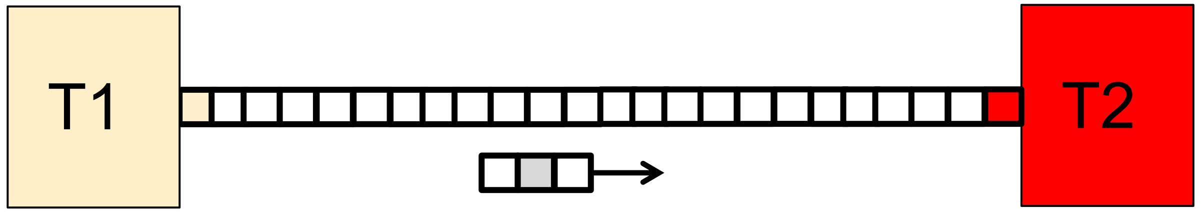


```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

   initialize_data(uk, ukp1, N, P); // init to zero, 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

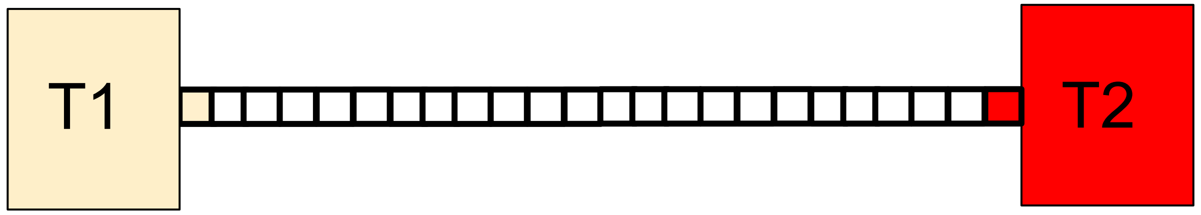


```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

   initialize_data(uk, ukp1, N, P); // init to zero, 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?

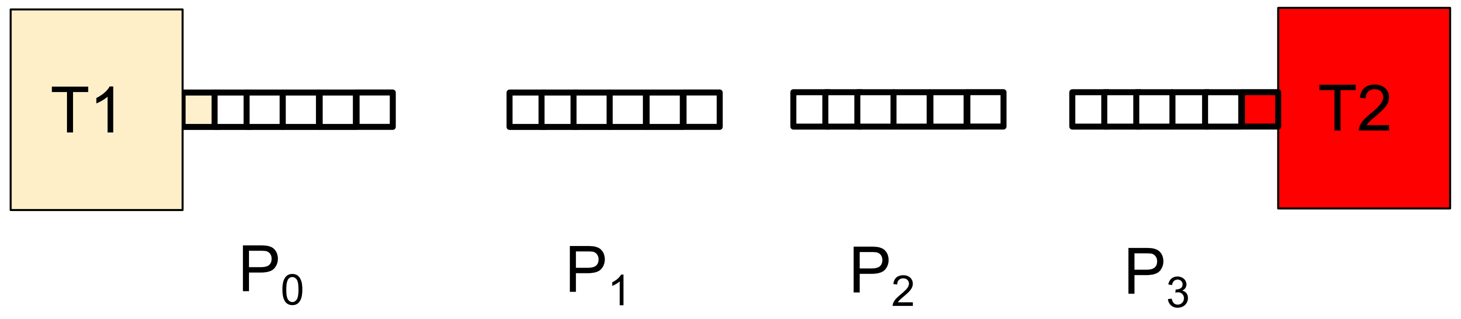
Heat Diffusion equation

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



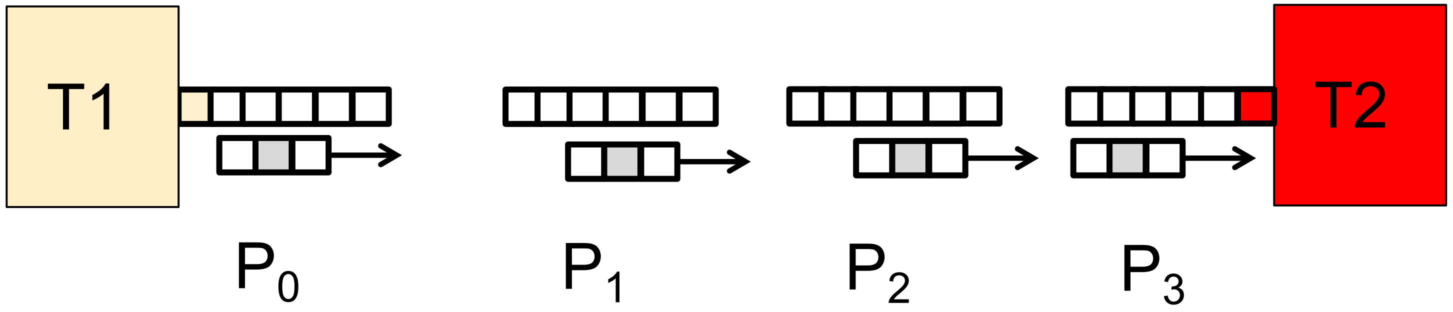
Heat Diffusion equation

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



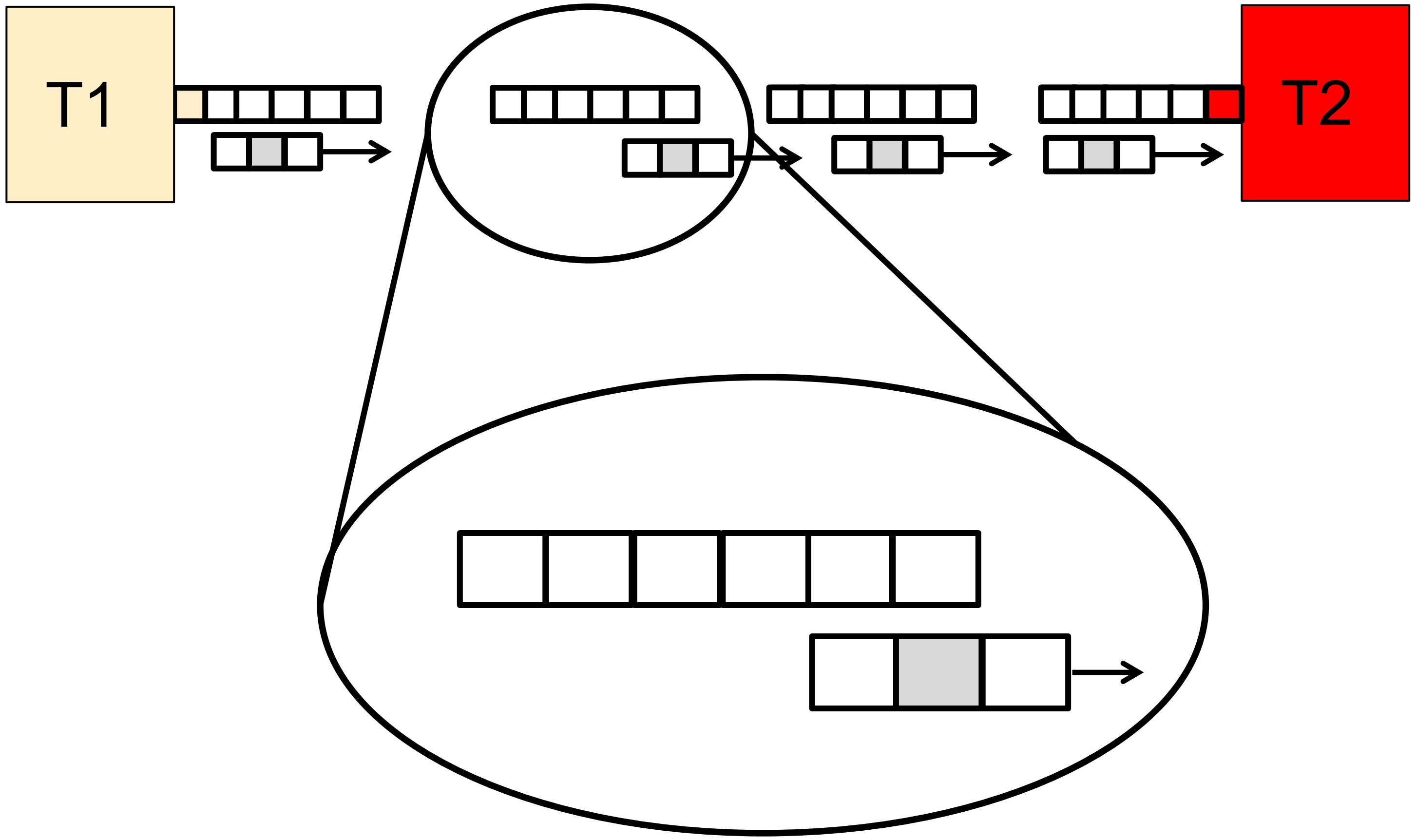
Heat Diffusion equation

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



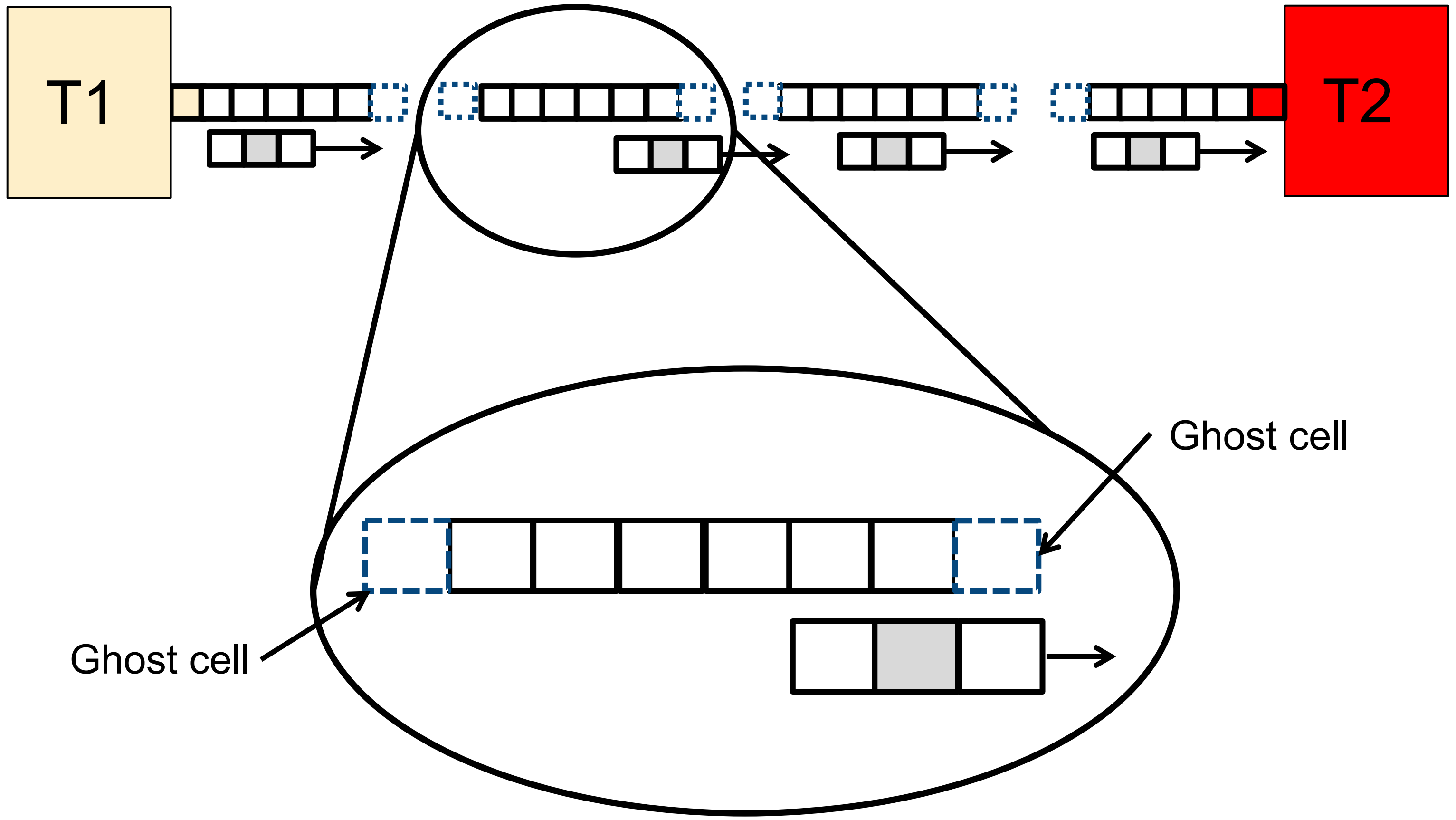
Heat Diffusion equation

- 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.

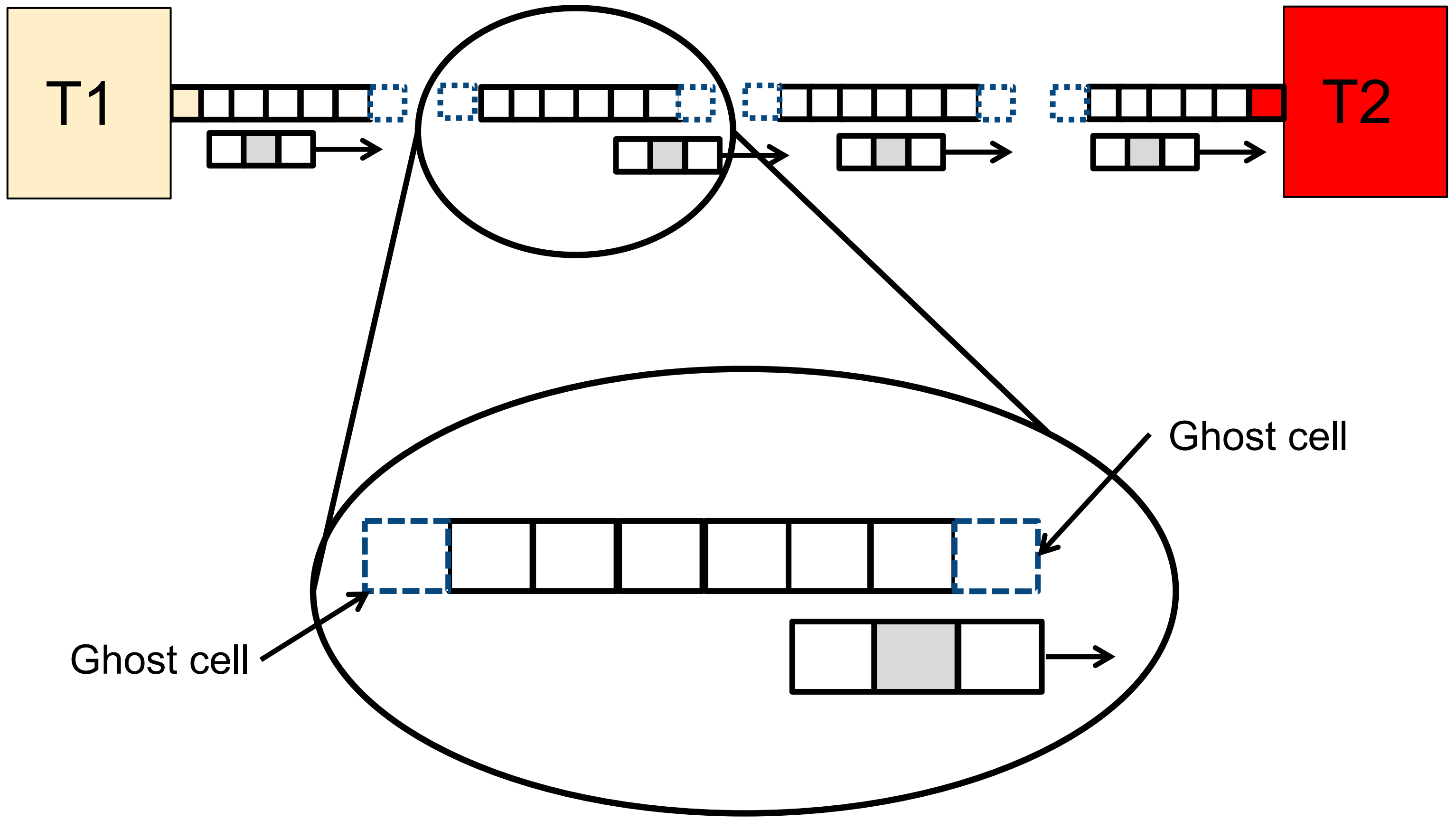


Design Pattern: Geometric Decomposition

- Use when:
  - The problem is organized around a central data structure that can be decomposed into smaller segments (chunks) that can be updated concurrently.
- Solution
  - Typically, the data structure is updated iteratively where a new value for one chunk depends on neighboring chunks.
  - The computation breaks down into three components: (1) exchange boundary data, (2) update the interiors or each chunk, and (3) update boundary regions. The optimal size of the chunks is dictated by the properties of the memory hierarchy.
- Note:
  - This pattern is often used with the Structured Mesh and linear algebra computational strategy pattern.

The Geometric Decomposition Pattern

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



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;
```

We write/explain this part first and then address the communication and data structures

Heat Diffusion MPI Example

Temperature fields using local data and values from ghost cells.

```
  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;
```

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

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

Heat Diffusion MPI Example

1D PDE solver … the simplest “real” message passing code I can think of. Note: edges of domain held at a fixed temperature

```
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"
double *up1 = malloc (sizeof(double) * (2 + N/P)); // to hold 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);
```

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

Receive my “left” ghost cell from my “left’ neighbor

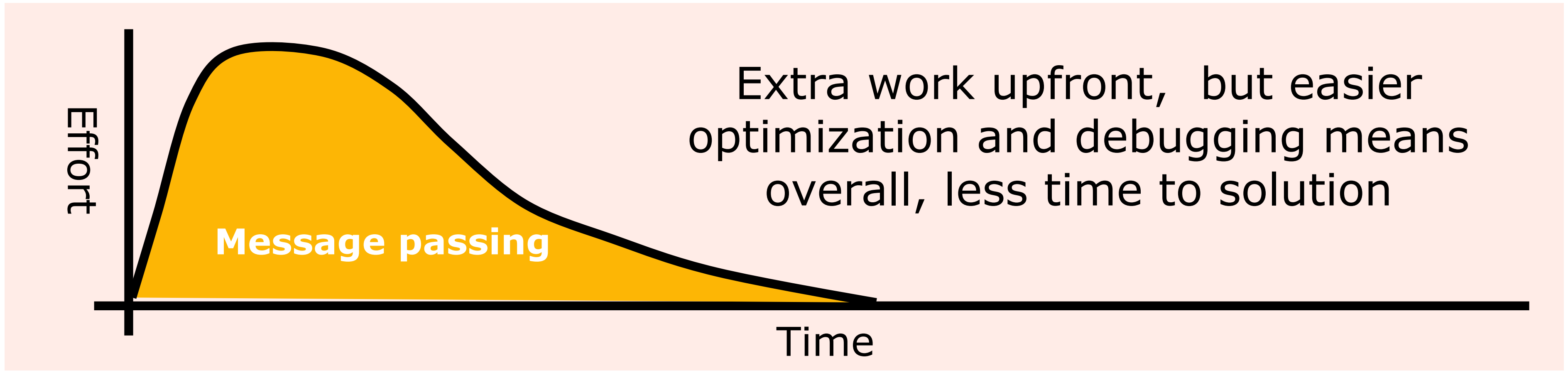
Send my “left” boundary value to my “left’ neighbor

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

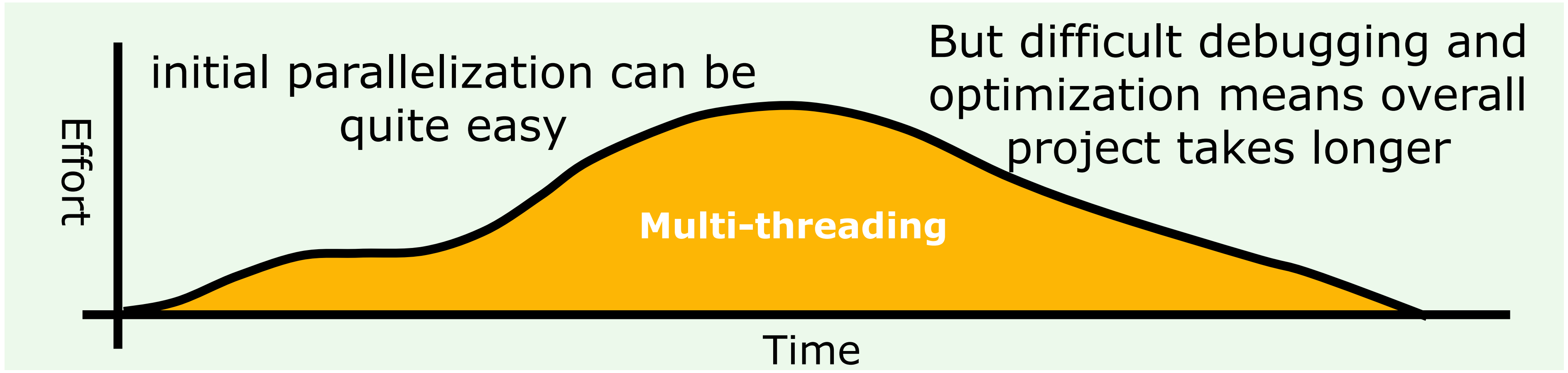
MPI is huge!!!

- MPI has over 430 functions!!!
  - Many forms of message passing
  - Full range of collectives (such as reduction)
  - dynamic process management
  - Shared memory
  - and much more
- Most programs, however use around a dozen different constructs … so it’s not as hard to learn as it may seem.

Does a shared address space make programming easier?



Extra work upfront, but easier optimization and debugging means overall, less time to solution



initial parallelization can be quite easy

But difficult debugging and optimization means overall project takes longer

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

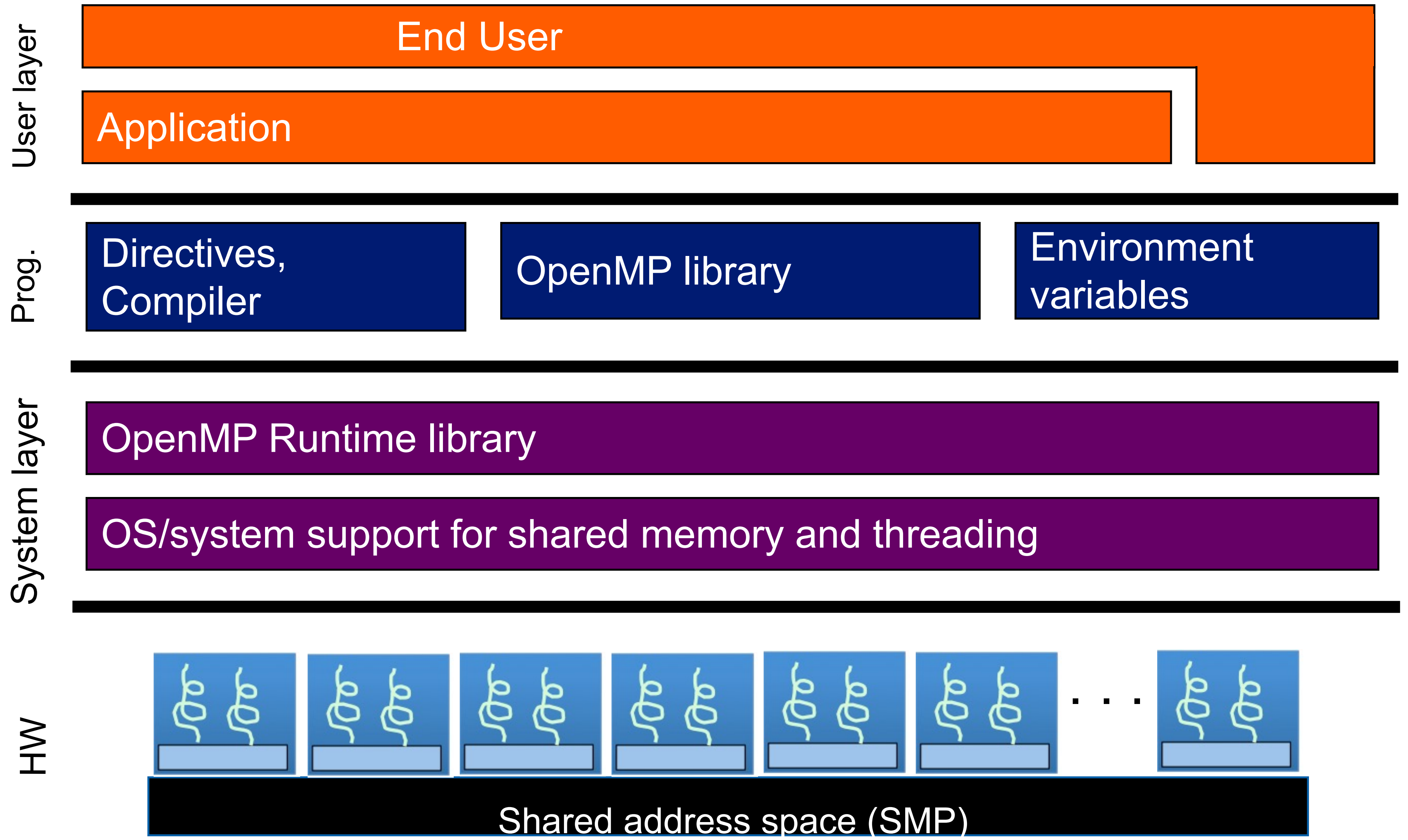
The Big Three

- 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)

The "new" kid on the block … GPUs

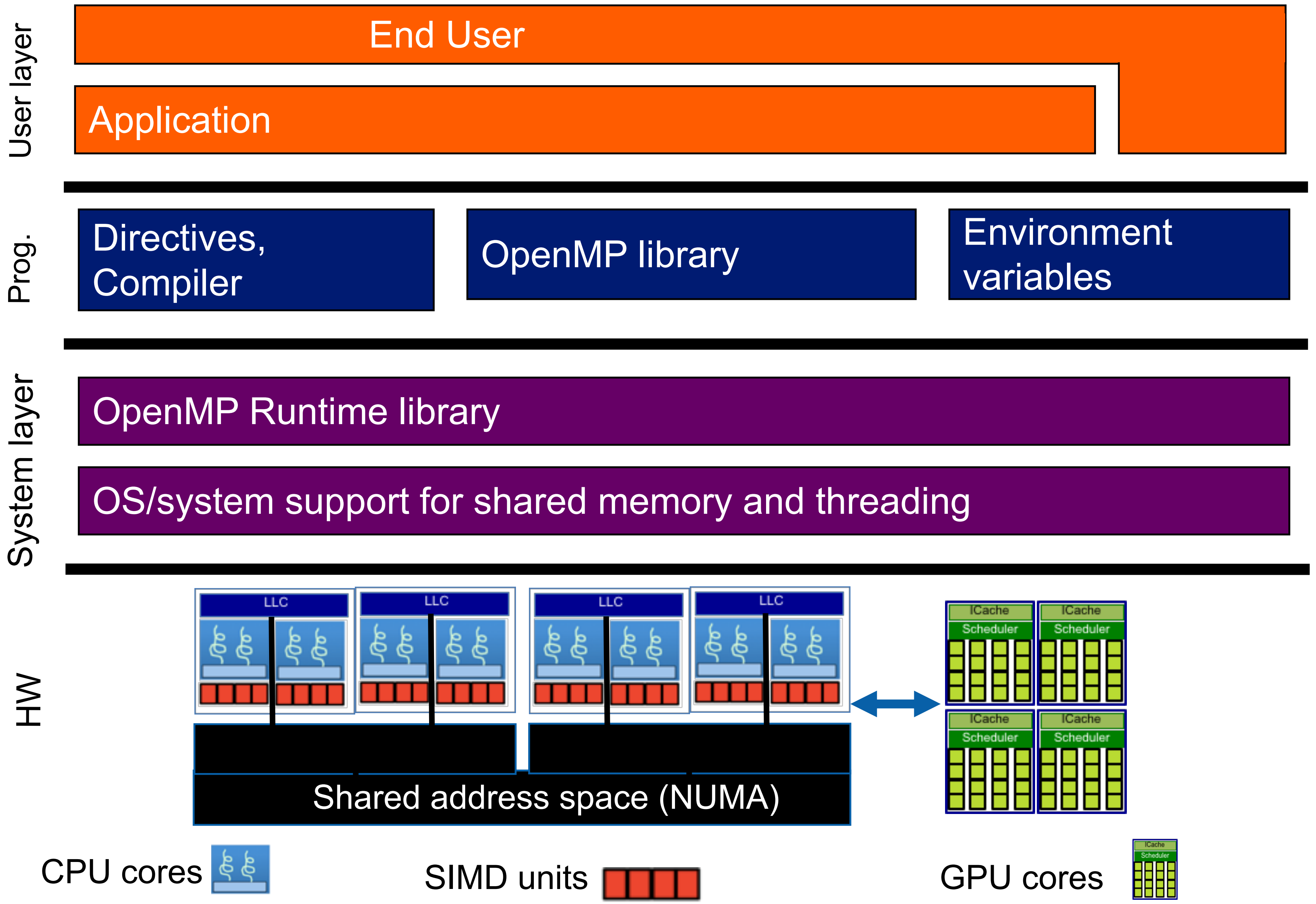
- Even if you don't plan to spend much time programming with these systems … a well rounded HPC programmer should know what they are and how they work.

OpenMP Basic Definitions: Basic Solution Stack



For the OpenMP Common Core, we focus on Symmetric Multiprocessor Case ….
i.e., lots of threads with "equal cost access" to memory

OpenMP Basic Definitions: Solution stack



The “BIG idea” Behind GPU programming

Traditional Loop based vector addition (vadd)

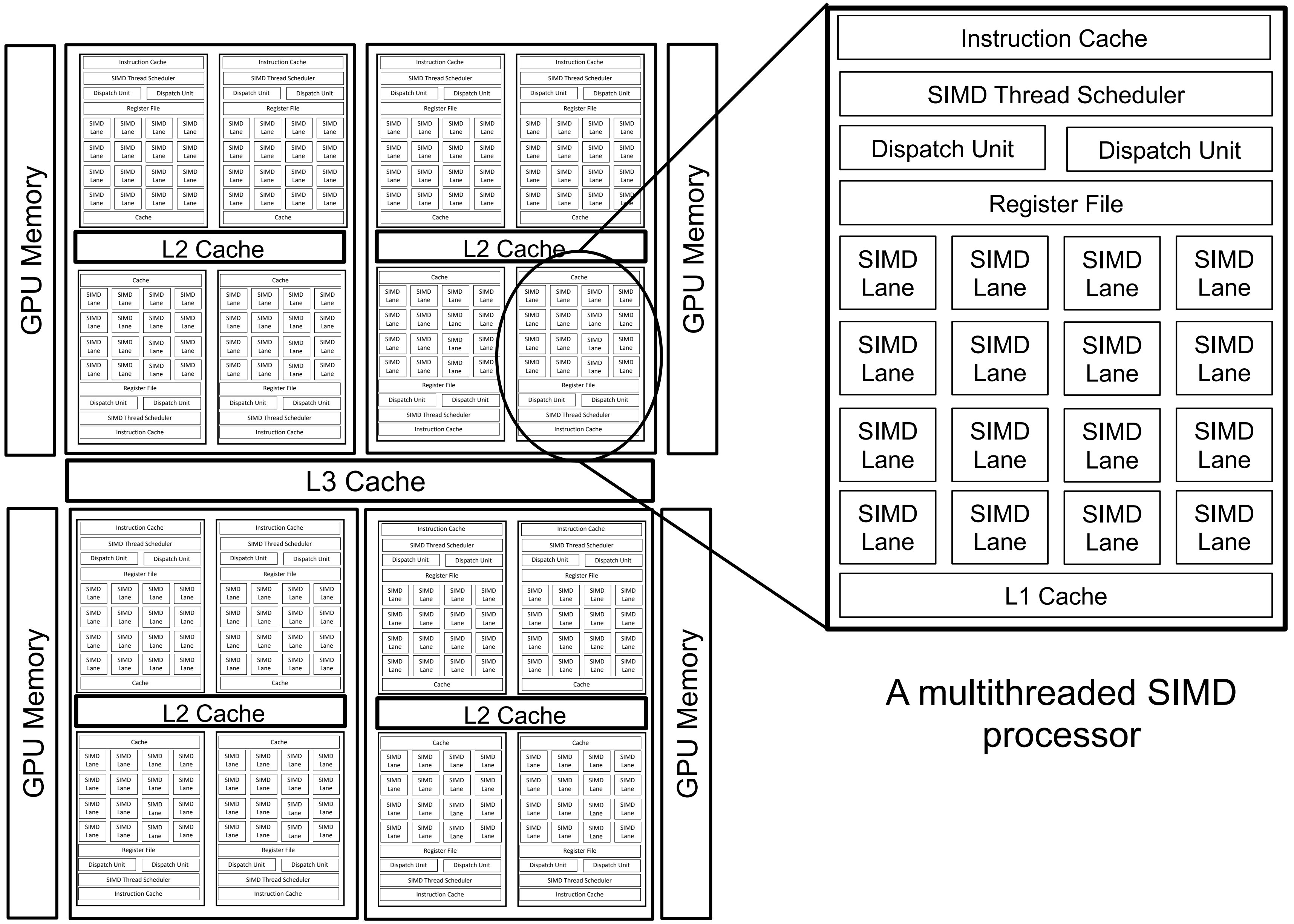
```
int main() {
   int N = . . . ;
   float *a, *b, *c;

   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);
}
```

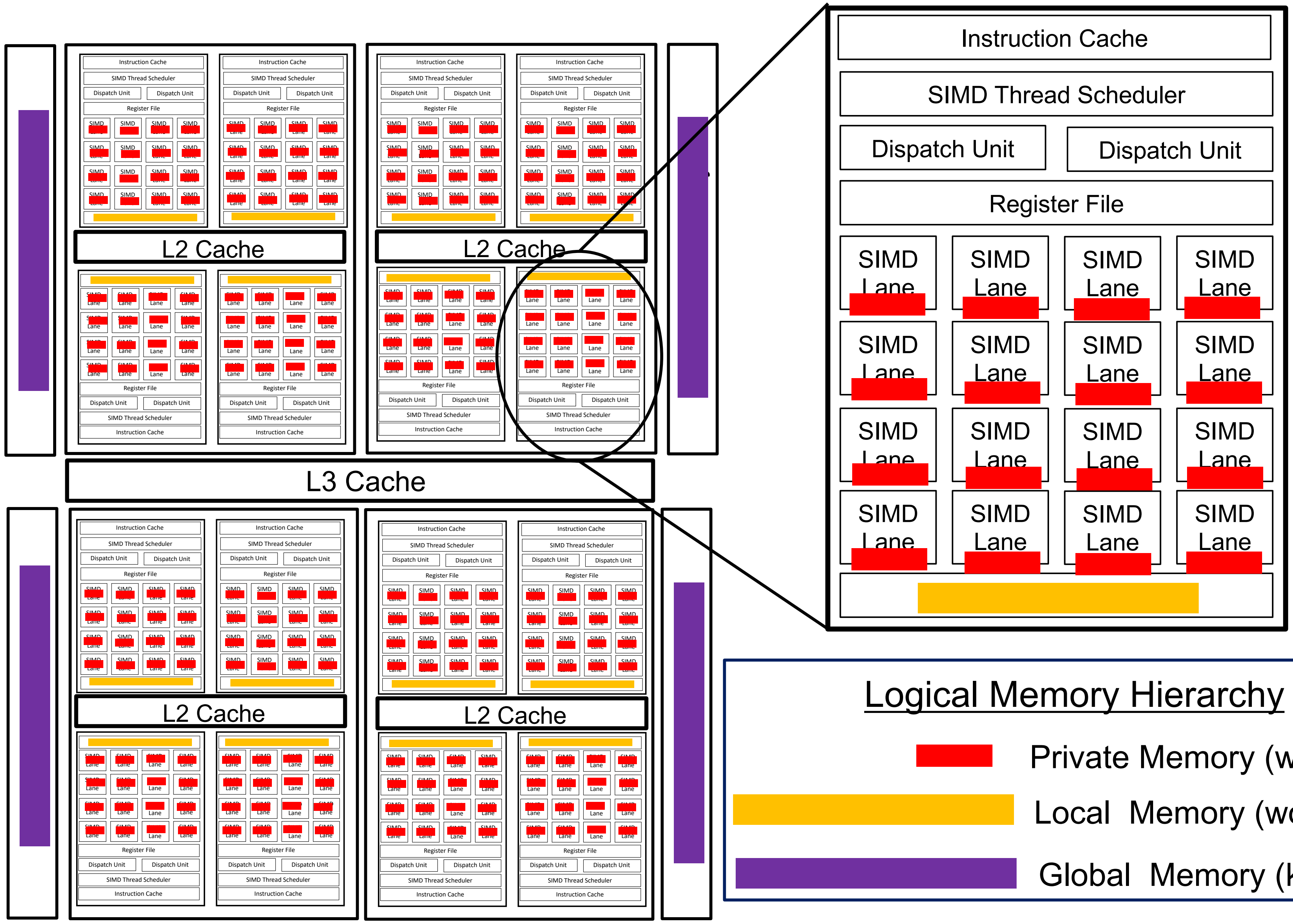
Assume a GPU with unified shared memory … allocate on host, visible on device too

A Generic GPU (following Hennessey and Patterson)



A multithreaded SIMD processor

A Generic GPU (following Hennessey and Patterson)



Logical Memory Hierarchy

🟥	Private Memory (work-item)
🟨	Local Memory (work-group)
🟪	Global Memory (kernel)

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

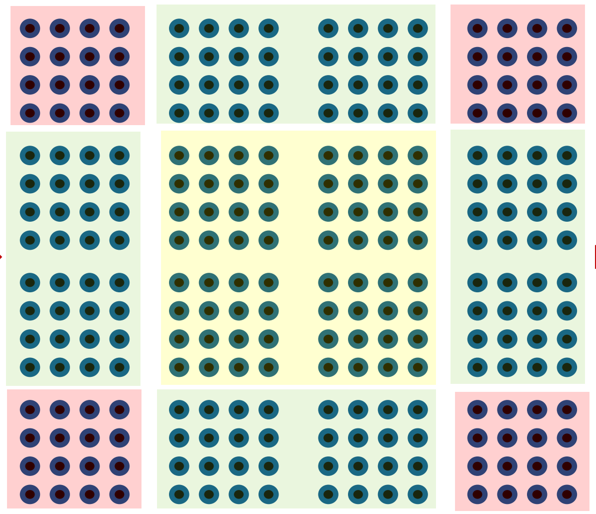
1. Turn source code into a scalar work-item

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

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

```
extern void reduce(  __local  float*,   __global float*);

__kernel void pi(  const int niters, float  step_size,
         __local  float*  l_sums,  __global float*  p_sums)
{
   int n_wrk_items  = get_local_size(0);
   int loc_id       = get_local_id(0);
   int grp_id   = get_group_id(0);
   float x, accum = 0.0f;    int i,istart,iend;

   istart =   (grp_id * n_wrk_items   + loc_id) * niters;
   iend   = istart+niters;

   for(i= istart; i<iend; i++){
       x = (i+0.5f)*step_size;    accum += 4.0f/(1.0f+x*x); }

   l_sums[local_id] = accum;
   barrier(CLK_LOCAL_MEM_FENCE);
   reduce(l_sums, p_sums);
}
```

This is OpenCL kernel code … the sort of code the OpenMP compiler generates on your behalf

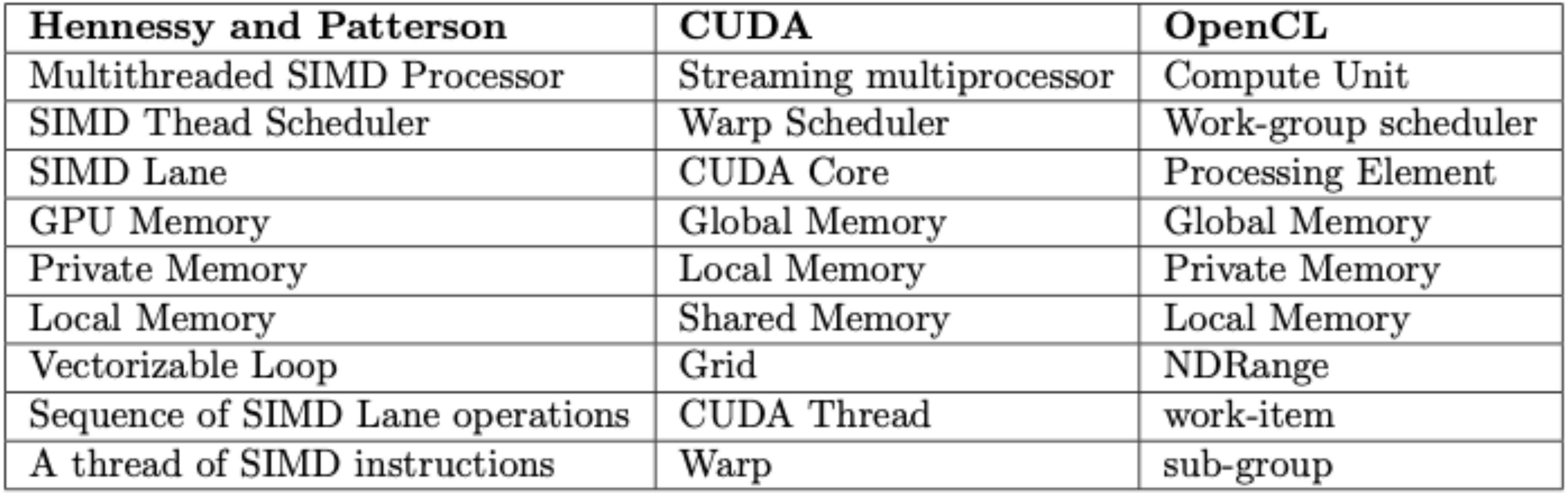


PCI Express 3.0 Host Interface
GigaThread Engine
L2 Cache

3. Map data structures onto the same index space

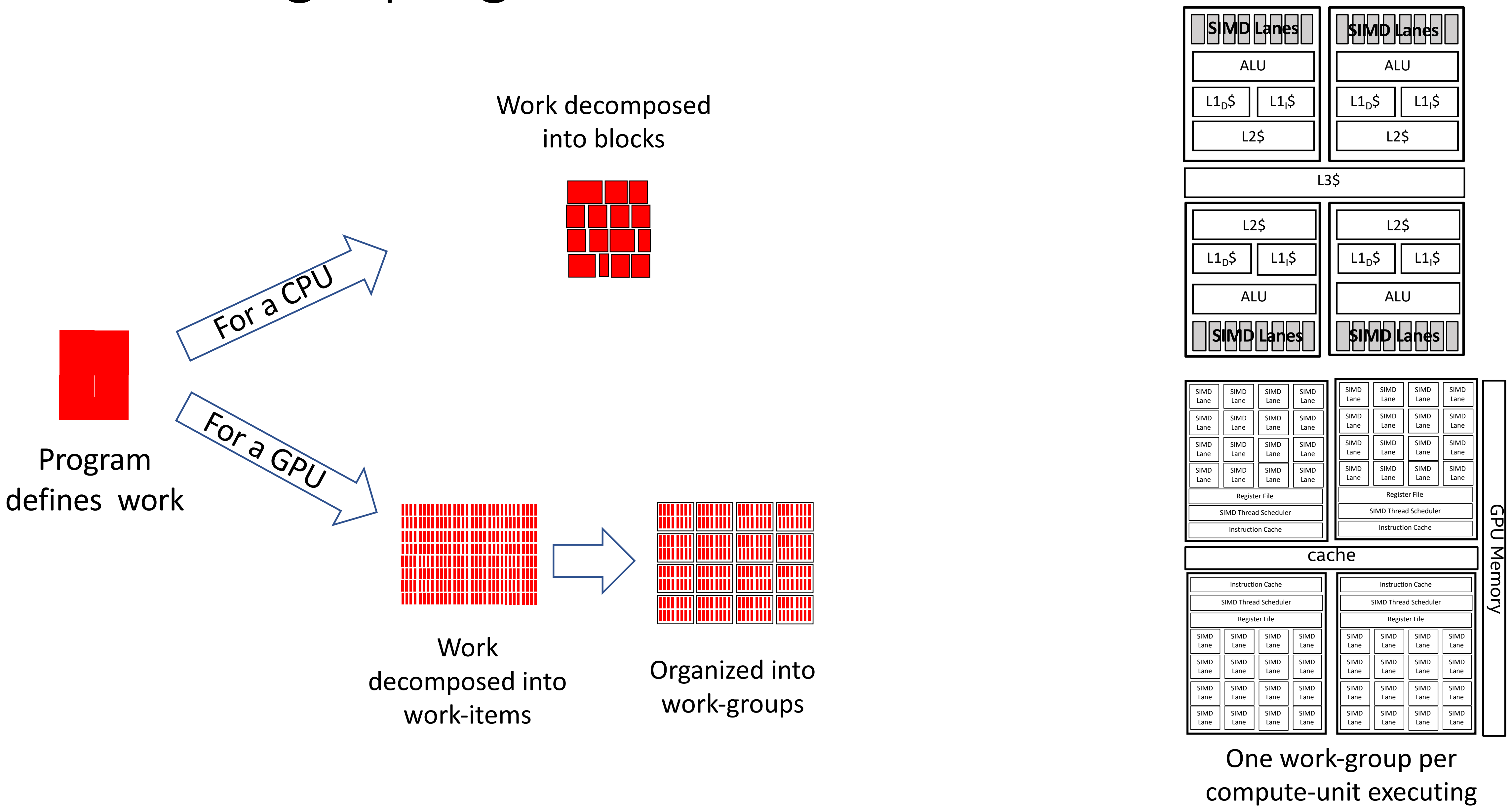
Third Party names are the property of their owners

GPU terminology is Broken (sorry about that)

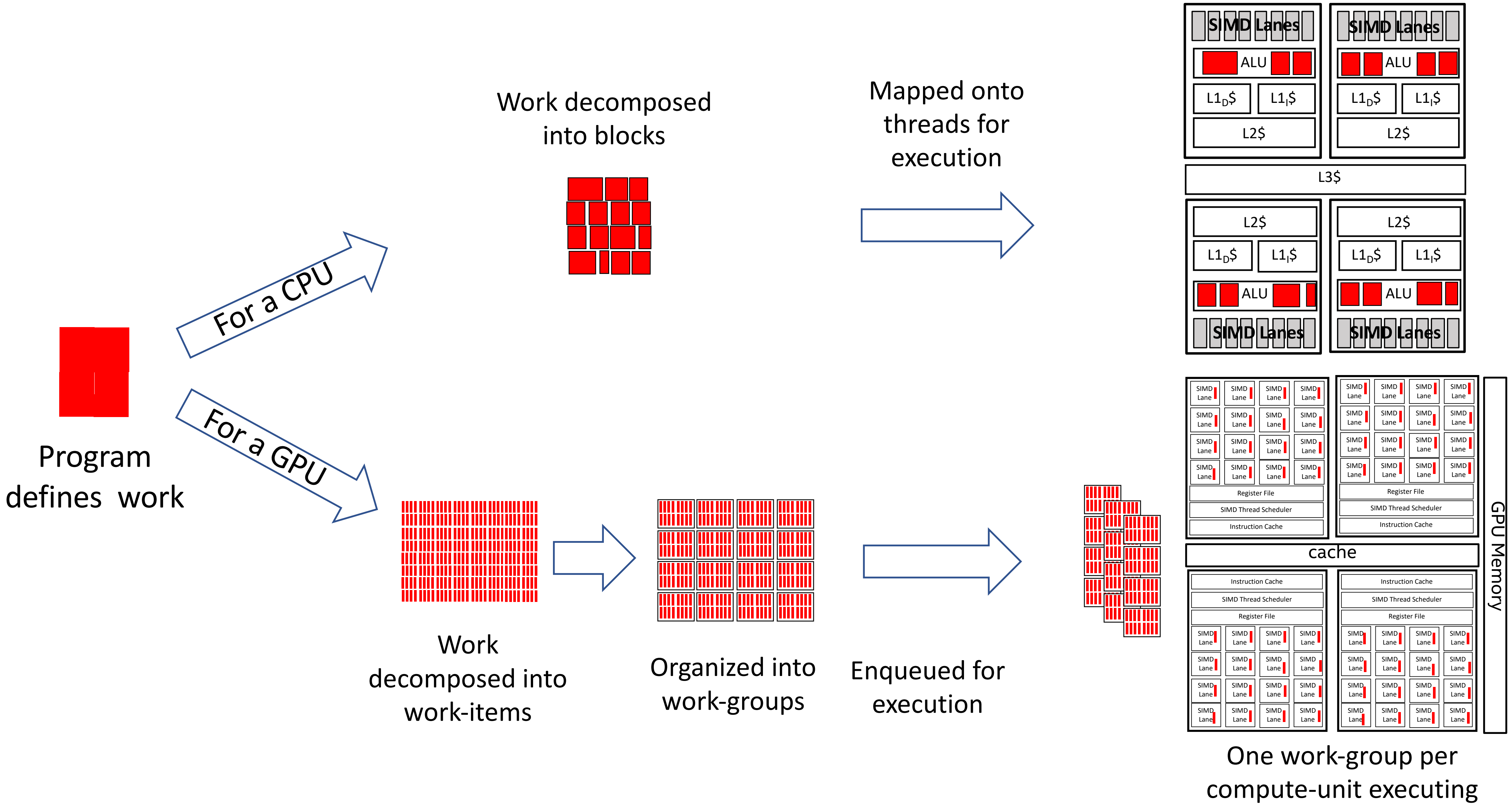


Hennessy and Patterson	**CUDA**	**OpenCL**
Multithreaded SIMD Processor	Streaming multiprocessor	Compute Unit
SIMD Thead Scheduler	Warp Scheduler	Work-group scheduler
SIMD Lane	CUDA Core	Processing Element
GPU Memory	Global Memory	Global Memory
Private Memory	Local Memory	Private Memory
Local Memory	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

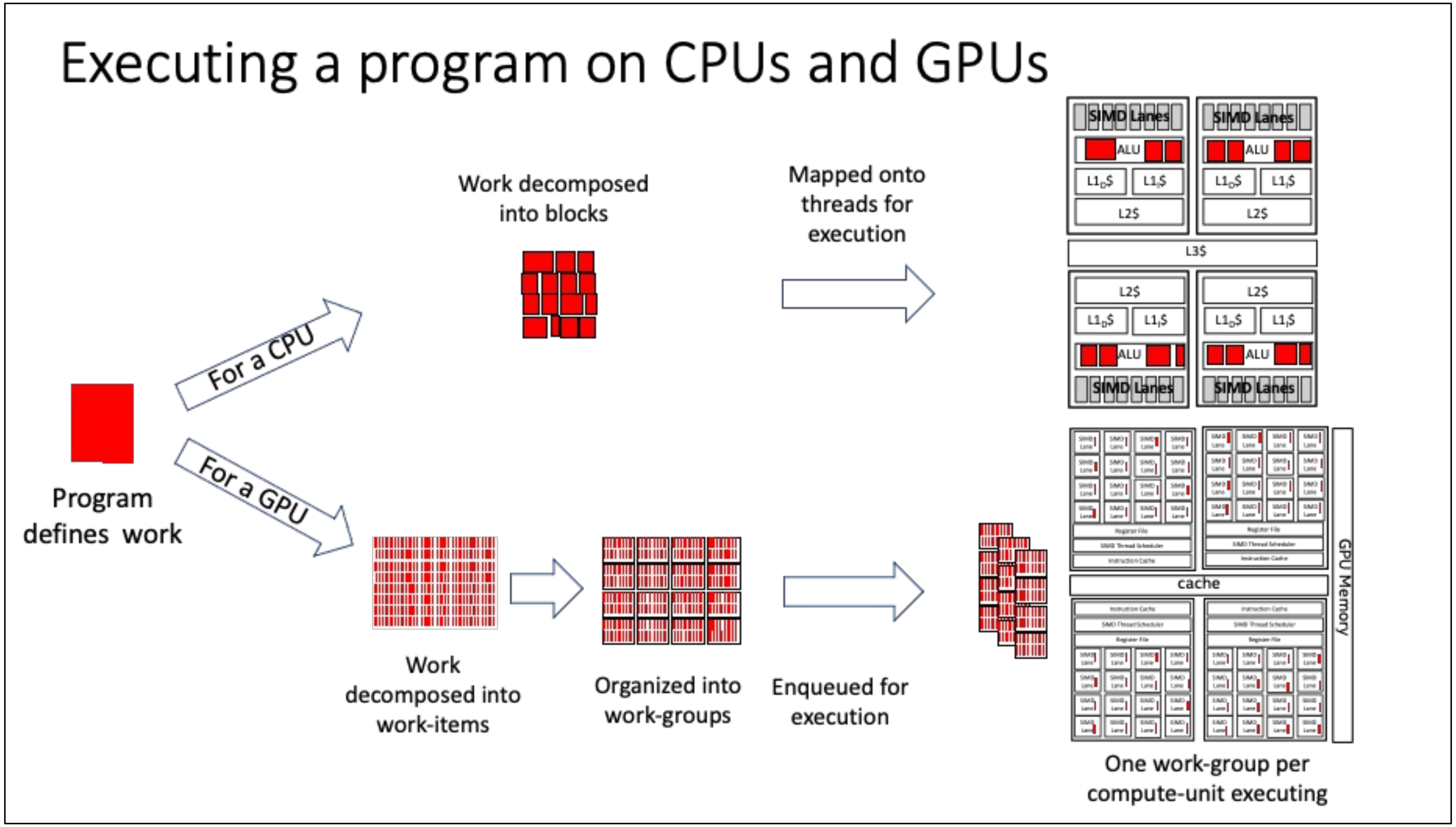
Executing a program on CPUs and GPUs



Executing a program on CPUs and GPUs



CPU/GPU execution modesl

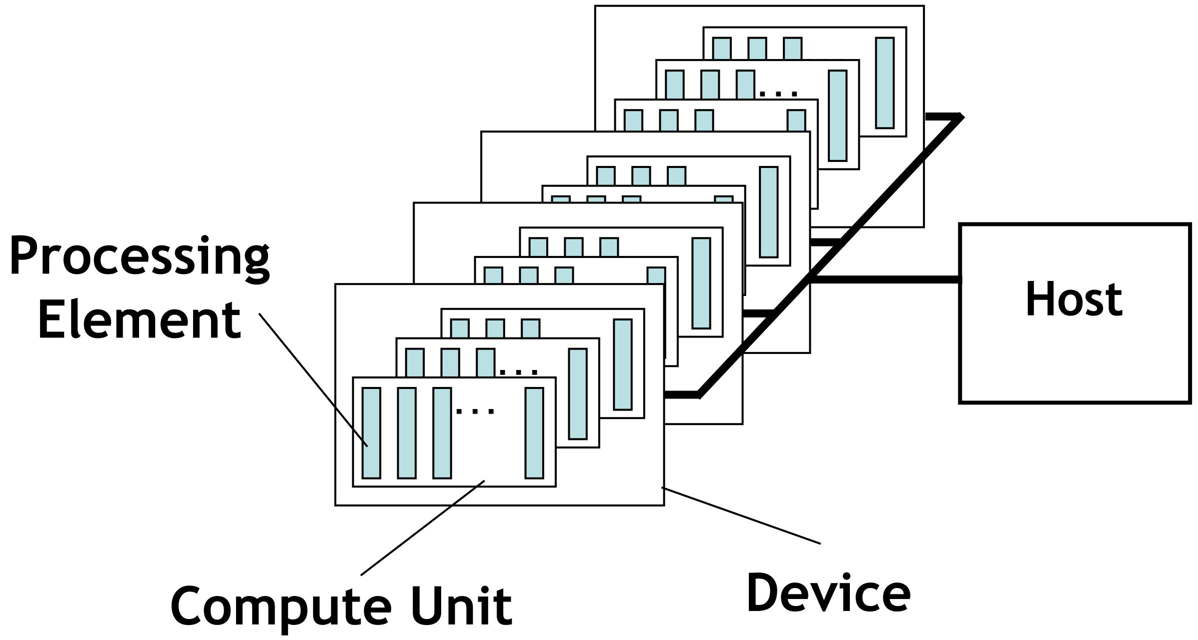


Executing a program on CPUs and GPUs
Program defines work
For a CPU
For a GPU
Work decomposed into blocks
Mapped onto threads for execution
SIMD Lanes
ALU
L1D$
L1I$
L2$
L3$
Work decomposed into work-items
Organized into work-groups
Enqueued for execution
cache
GPU Memory
One work-group per compute-unit executing

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

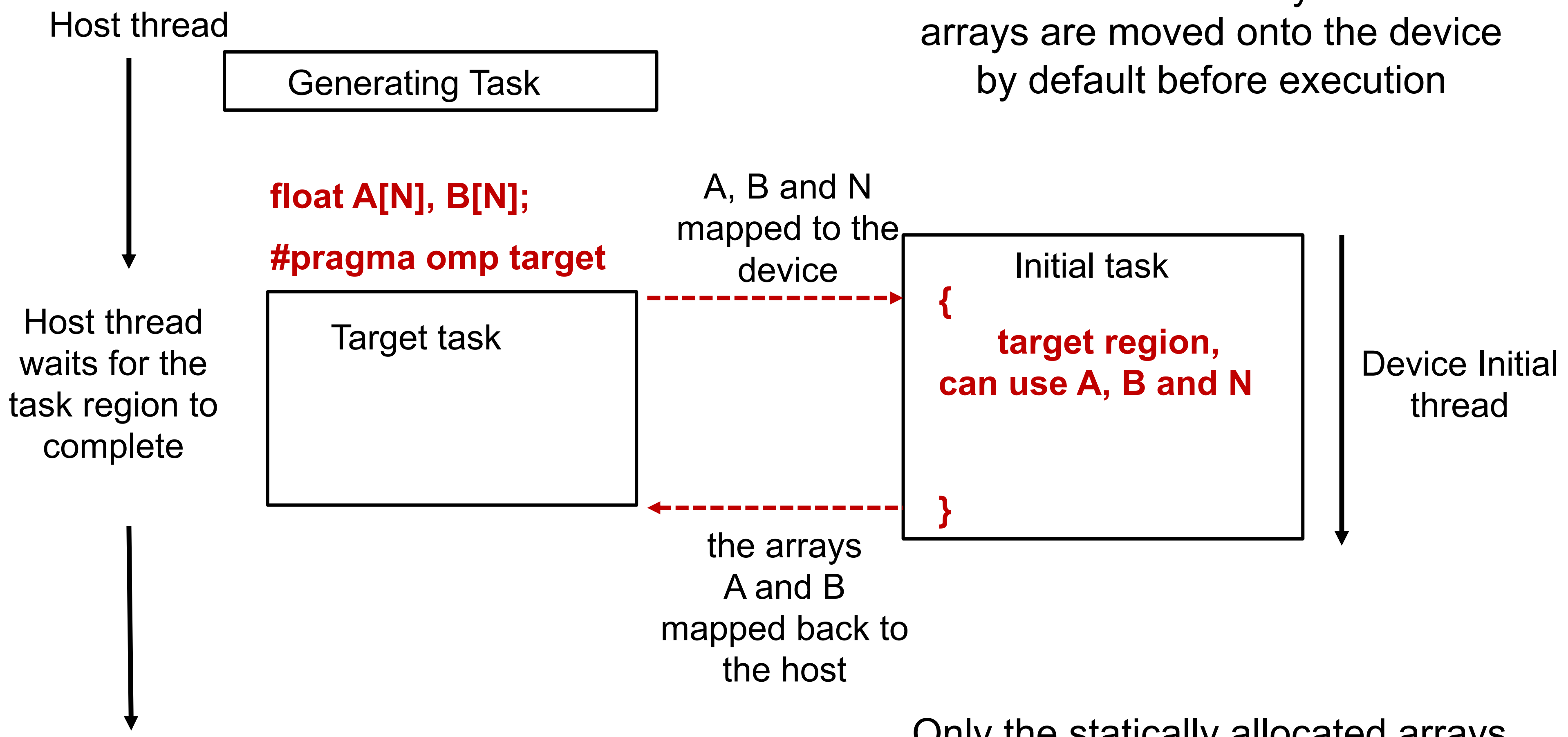
For a GPU, any given work-group might be in the queue waiting to execute.

A Generic 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***

Running code on the GPU:
The target construct and default data movement


Host thread
Generating Task
float A[N], B[N];
#pragma omp target
Host thread waits for the task region to complete
Target task
A, B and N mapped to the device
Initial task
{
target region, can use A, B and N
}
the arrays A and B mapped back to the host
Device Initial thread

Scalars and statically allocated arrays are moved onto the device by default before execution

Only the statically allocated arrays are moved back to the host after the target region completes

Based on figure 6.4 in **Using OpenMP – The Next Step** by van der Pas, Stotzer and Terboven, MIT Press, 2017

Default Data Sharing: example

```
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
}
```

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
}
```

The loop construct tells the compiler:

"this loop will execute correctly if the loop iterations run in any order. You can safely run them concurrently. And the loop-body doesn't contain any OpenMP constructs. So do whatever you can to make the code run fast"

The loop construct is a declarative construct. You tell the compiler what you want done but you DO NOT tell it how to "do it". This is new for OpenMP

What about pointers?
implicit movement with a target region

- Pointers and their data:
  - *Example: arrays allocated on the heap*
    - double *A = malloc(sizeof(double)*1000);
  - The pointer value will be mapped*.
  - But the data it points to ***will not*** be mapped by default.

*Mapped: A variable defined on the host is mapped onto a device when the variable is associated with a version on the device and the value on the host is copied onto the device

Explicit Data Sharing

- Data allocated on the heap needs to be explicitly copied to/from the device
- We *explicitly* control the movement of data using the **map** clause.

```
int main(void) {
   int  ii=0, N = 1024;
   int* A = malloc(sizeof(int)*N);

   #pragma omp target
   {
     // N, ii and A all exist here
     // The data that A points to (*A , A[ii]) DOES NOT exist here!
   }
}
```

Controlling data movement

```
int i, a[N], b[N], c[N];
#pragma omp target map(to:a,b) map(tofrom:c)
```

Data movement defined from the *host* perspective.

- The various forms of the map clause
  - **map(to:list)**: On entering the region, variables in the list are initialized on the device using the original values from the host (host to device copy).
  - **map(from:list)**: At the end of the target region, the values from variables in the list are copied into the original variables (device to host copy). On entering the region, initial value of the variable is not initialized.
  - **map(tofrom:list)**: the effect of both a map-to and a map-from (host to device copy at start of region, device to host copy at end)
  - **map(alloc:list)**: On entering the region, data is allocated and uninitialized on the device.
  - **map(list)**: equivalent to **map(tofrom:list)**.

- For pointers you must use array section notation ..
  - **map(to:a[0:N]).** Notation is **A[lower-bound : length]**

Moving arrays with the map clause

```
int main(void) {
   int  N = 1024;
   int* A = malloc(sizeof(int)*N);

   #pragma omp target map(A[0:N])
   {
     // N, ii and A all exist here
     // The data that A points to DOES exist here!
   }
}
```

Default mapping
map(tofrom: A[0:N])

Copy at start and end of **target** region.

Our running example: Jacobi solver

- An iterative method to solve a system of linear equations
  - Given a matrix A and a vector b find the vector x such that $Ax=b$
- The basic algorithm:
  - Write A as a lower triangular (L), upper triangular (U) and diagonal matrix

    $$Ax = (L+D+U)x = b$$

  - Carry out multiplications and rearrange

    $$Dx=b-(L+U)x \quad \rightarrow \quad x = (b-(L+U)x)/D$$

  - Iteratively compute a new x using the x from the previous iteration

    $$X_{new} = (b-(L+U)x_{old})/D$$

- Advantage: we can easily test if the answer is correct by multiplying our final x by A and comparing to b
- Disadvantage: It takes many iterations and only works for diagonally dominant matrices

Jacobi Solver

Iteratively update xnew until the value stabilizes (i.e. change less than a preset TOL)

```
<<< allocate and initialize the matrix A >>>
<<< and vectors x1, x2 and b             >>>

while((conv > TOL) && (iters<MAX_ITERS))
  {
   iters++;

   for (i=0; i<Ndim; i++){
      xnew[i] = (TYPE) 0.0;
      for (j=0; j<Ndim;j++){
         if(i!=j)
           xnew[i]+= A[i*Ndim + j]*xold[j];
      }
      xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
   }

   // test convergence
   conv = 0.0;
   for (i=0; i<Ndim; i++){
      tmp  = xnew[i]-xold[i];
      conv += tmp*tmp;
   }
   conv = sqrt((double)conv);

  // swap pointers for next
  // iteration
  TYPE* tmp = xold;
  xold = xnew;
  xnew = tmp;

} // end while loop
```

Jacobi Solver (Parallel Target/loop, 1/2)

```
 while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;
#pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                   map(to:A[0:Ndim*Ndim], b[0:Ndim])
#pragma omp loop
for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
            if(i!=j)
              xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

Jacobi Solver (Parallel Target/loop, 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
 #pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                            map(tofrom:conv)
#pragma omp loop private(i,tmp) reduction(+:conv)
for (i=0; i<Ndim; i++){
       tmp  = xnew[i]-xold[i];
       conv += tmp*tmp;
    }
    conv = sqrt((double)conv);
   TYPE* tmp = xold;
   xold = xnew;
   xnew = tmp;
} // end while loop
```

This worked but the performance was awful. Why?

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs

Cray® XC40™ Supercomputer running Cray® Compiling Environment 8.5.3.
Intel® Xeon ® CPU E5-2697 v2 @ 2.70GHz with 32 GB DDR3. NVIDIA® Tesla® K20X, 6GB.

Data movement dominates!!!

```
while((conv > TOLERANCE) && (iters<MAX_ITERS))
  { iters++;
    xnew = iters % s ? x2 : x1;
    xold   = iters % s ? x1 : x2;

    #pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                     map(to:A[0:Ndim*Ndim], b[0:Ndim] )
      #pragma omp loop private(i,j)
      for (i=0; i<Ndim; i++){
          xnew[i] = (TYPE) 0.0;
          for (j=0; j<Ndim;j++){
              if(i!=j)
                xnew[i]+= A[i*Ndim + j]*xold[j];
          }
          xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
      }
  // test convergence
    conv = 0.0;
    #pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                           map(tofrom:conv)
       #pragma loop reduction(+:conv)
       for (i=0; i<Ndim; i++){
           tmp  = xnew[i]-xold[i];
           conv += tmp*tmp;
       }
    conv = sqrt((double)conv);
  }
```

Typically over 4000 iterations!

For each iteration, copy to device $(3*Ndim+Ndim^2)*sizeof(TYPE)$ bytes

For each iteration, copy from device 2*Ndim*sizeof(TYPE) bytes

For each iteration, copy to device 2*Ndim*sizeof(TYPE) bytes

Target data directive

- The **target data** construct creates a target data region … use **map** clauses for explicit data management

Data is mapped onto the device at the beginning of the construct

```
#pragma omp target data map(to:A, B) map(from: C)
{
   #pragma omp target
      {do lots of stuff with A, B and C}

   {do something on the host}

   #pragma omp target
      {do lots of stuff with A, B, and C}
}
```

one or more **target regions** work within the **target data region**

Data is mapped back to the host at the end of the target data region

Jacobi Solver (Par Target Data, 1/2)

```
 #pragma omp target data map(tofrom:xold[0:Ndim],xnew[0:Ndim]) \
                         map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;

#pragma omp target
#pragma omp loop private(j) firstprivate(xnew,xold)
     for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
           if(i!=j)
             xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
     }
```

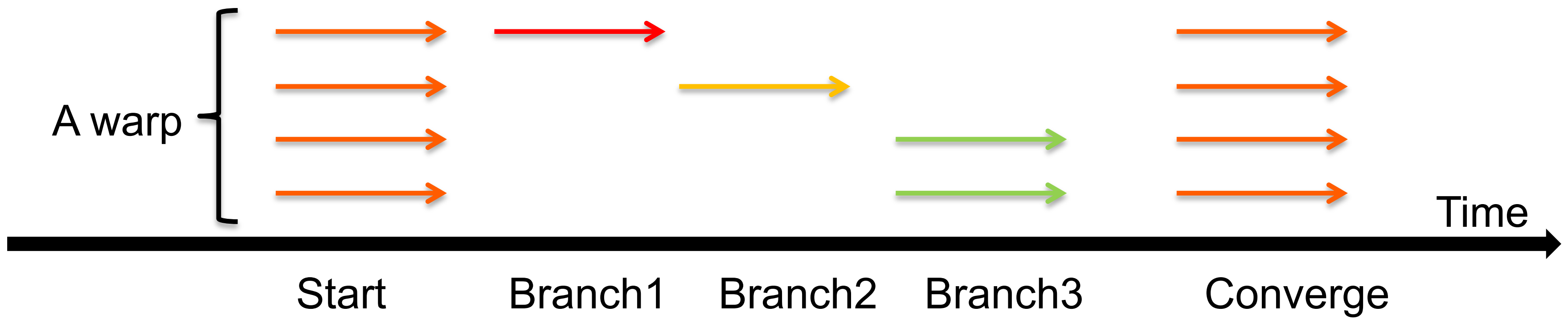
Jacobi Solver (Par Target Data, 2/2)

```
// test convergence
conv = 0.0;
#pragma omp target map(tofrom: conv)
#pragma omp loop private(tmp) firstprivate(xnew,xold)  reduction(+:conv)

   for (i=0; i<Ndim; i++){
      tmp  = xnew[i]-xold[i];
      conv += tmp*tmp;
   }
// end target region
 conv = sqrt((double)conv);

   TYPE* tmp = xold;
   xold = xnew;
   xnew = tmp;
} // end while loop
```

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target data region	18.37 secs

Single Instruction Multiple Data

- Individual work-items of a warp start together at the same program address
- Each work-item has its own instruction address counter and register state
  - Each work-item is free to branch and execute independently
  - Supports the SPMD pattern.
- Branch behavior
  - Each branch will be executed serially
  - Work-items not following the current branch will be disabled



Branching

Conditional execution

```
// Only evaluate expression
// if condition is met
if (a > b)
{
  acc += (a - b*c);
}
```

Selection and masking

```
// Always evaluate expression
// and mask result
temp = (a - b*c);
mask = (a > b ? 1.f : 0.f);
acc += (mask * temp);
```

Coalescence

- Coalesce - to combine into one
- Coalesced memory accesses are key for high bandwidth
- Simply, it means, if thread *i* accesses memory location *n* then thread *i+1* accesses memory location *n+1*
- In practice, it's not quite as strict…

```
for (int id = 0; id < size; id++)
{
  // ideal
     float val1 = memA[id];

  // still pretty good
     const int c = 3;
     float val2 = memA[id + c];

  // stride size is not so good
     float val3 = memA[c*id];

  // terrible
     const int loc =
       some_strange_func(id);

     float val4 = memA[loc];
}
```

Jacobi Solver (Target Data/branchless/coalesced mem, 1/2)

```
  #pragma omp target data map(tofrom:x1[0:Ndim],x2[0:Ndim]) \
                      map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;
#pragma omp target
     #pragma omp loop private(j)
   for (i=0; i<Ndim; i++){
      xnew[i] = (TYPE) 0.0;
      for (j=0; j<Ndim;j++){
         xnew[i]+= (A[j*Ndim + i]*xold[j])*((TYPE)(i != j));
      }
      xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
   }
```

We replaced the original code with a poor memory access pattern

    xnew[i]+= (A[i*Ndim + j]*xold[j])

With the more efficient

    xnew[i]+= (A[j*Ndim + i]*xold[j])

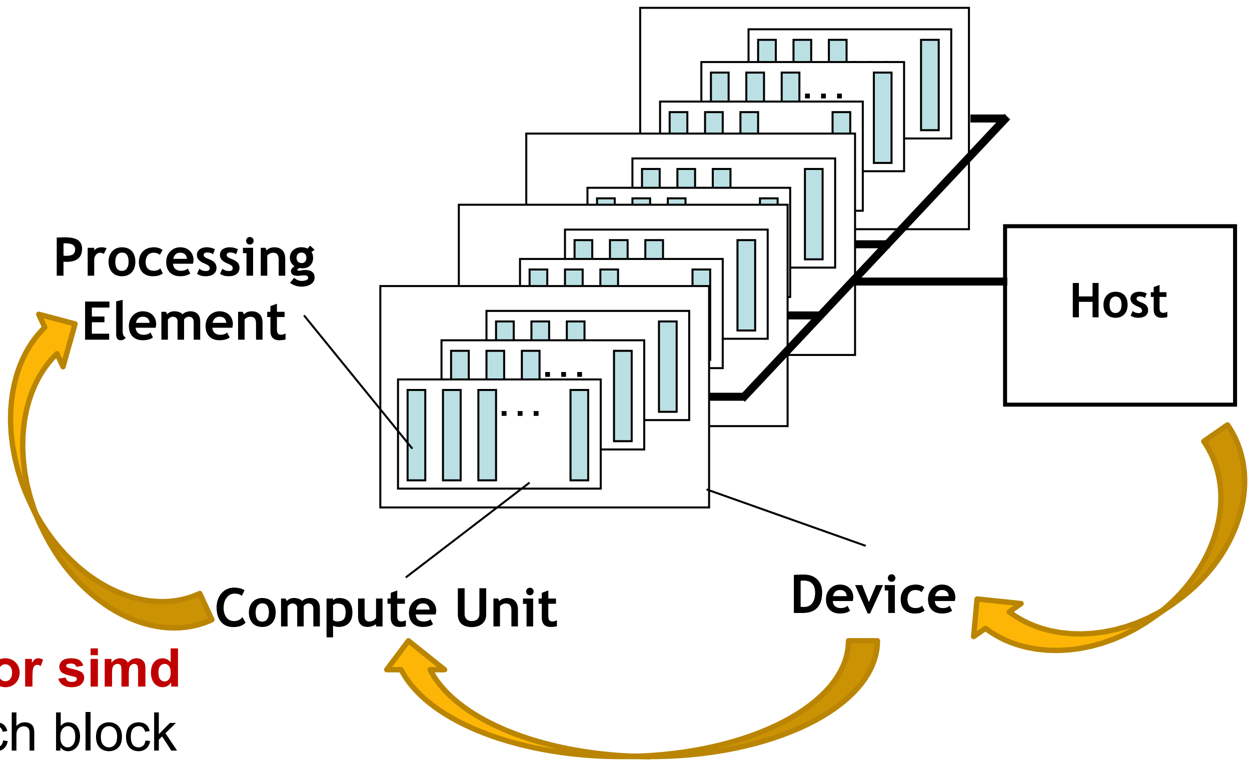
Jacobi Solver (Target Data/branchless/coalesced mem, 2/2)

```
    //
    // test convergence
    conv = 0.0;
#pragma omp target map(tofrom: conv)
 #pragma omp loop private(tmp) reduction(+:conv)
    for (i=0; i<Ndim; i++){
       tmp  = xnew[i]-xold[i];
       conv += tmp*tmp;
    }
conv = sqrt((double)conv);
   TYPE* tmp = xold;
   xold = xnew;
   xnew = tmp;
} // end while loop
```

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target data region	18.37 secs
	Above plus reduced branching	13.74 secs
	Above plus improved mem access	7.64 secs

The loop construct is great, but sometimes you want more control.

Our host/device Platform Model and OpenMP



Target construct to get onto a device

Parallel for simd to run each block of loop iterations on the processing elements

Teams construct to create a league of teams with one team of threads on each compute unit.

Distribute construct to assign blocks of loop iterations to teams.

teams and distribute constructs

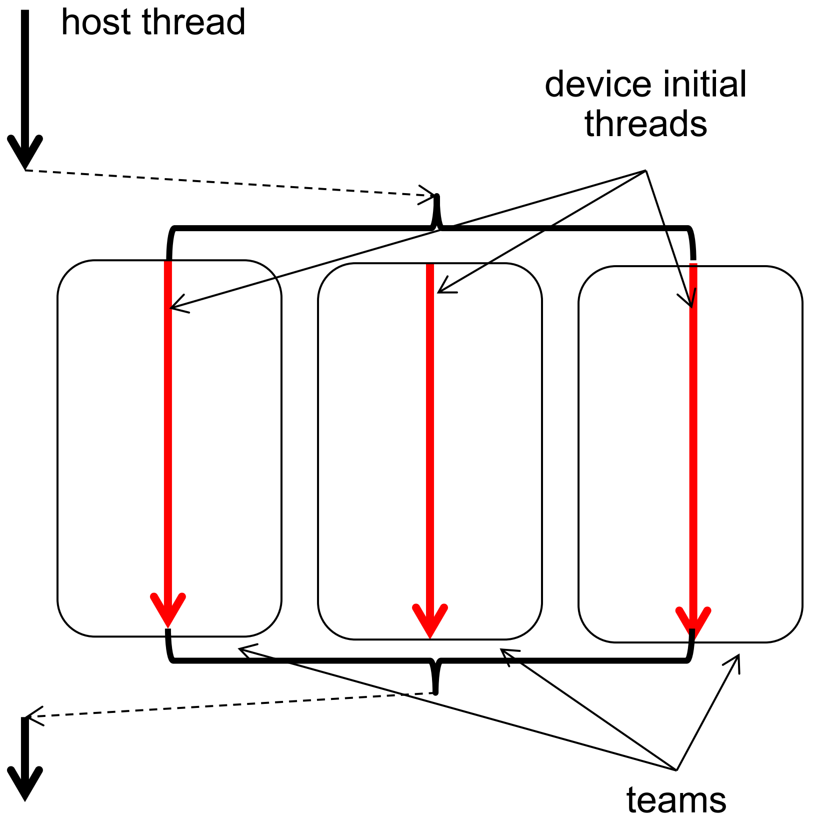
- The **teams** construct
  - Similar to the **parallel** construct
  - It starts a league of thread teams
  - Each team in the league starts as one initial thread – a team of one
  - Threads in different teams cannot synchronize with each other
  - The construct must be “perfectly” nested in a **target** construct

- The **distribute** construct
  - Similar to the **for** construct
  - Loop iterations are workshared across the initial threads in a league
  - No implicit barrier at the end of the construct
  - **dist_schedule(*kind[, chunk_size]*)**
    - If specified, scheduling kind must be static
    - Chunks are distributed in round-robin fashion in chunks of size ***chunk_size***
    - If no chunk size specified, chunks are of (almost) equal size; each team receives at least one chunk

Create a league of teams and distribute a loop among them

- teams construct
- distribute construct

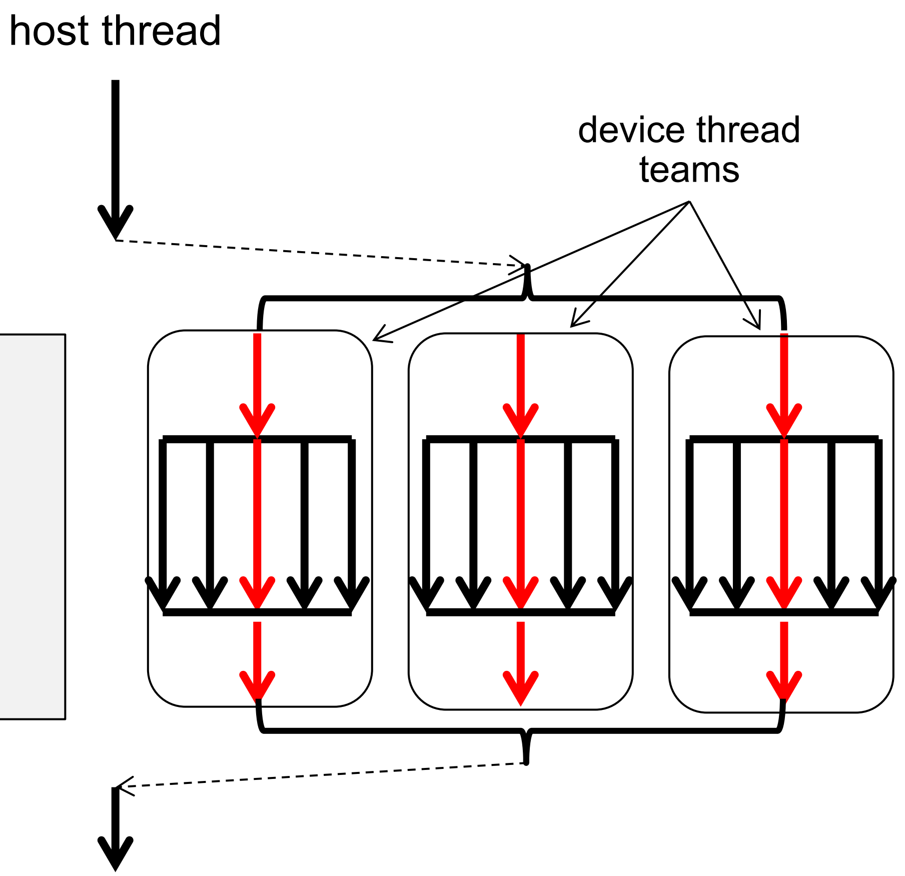
```
#pragma omp target
#pragma omp teams
#pragma omp distribute
 for (i=0;i<N;i++)
    …
```



- Transfer execution control to MULTIPLE device initial threads
- Workshare loop iterations across the initial threads.

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- teams distribute
- parallel for simd



```
#pragma omp target
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for simd
 for (i=0;i<N;i++)
   …
```

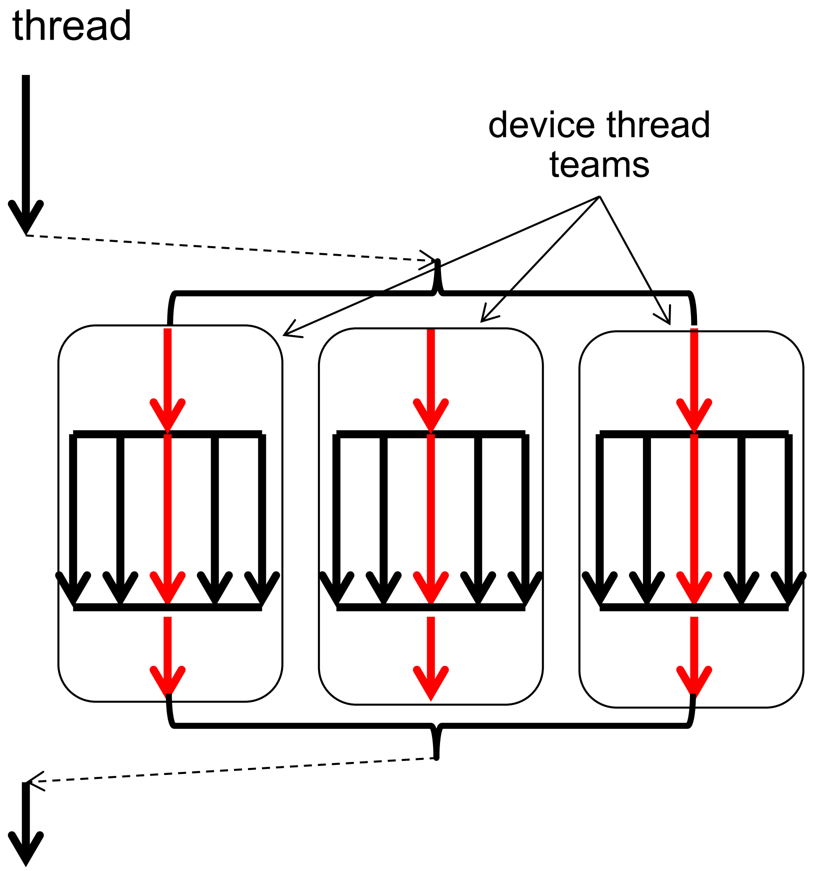
- Transfer execution control to MULTIPLE device initial threads
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for)

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- teams distribute
- parallel for simd

Works with nested loops as well

```
#pragma omp target
#pragma omp teams distribute
 for (i=0;i<N;i++)
 #pragma omp parallel for simd
 for (j=0;j<M;i++)
   ...
```



- Transfer execution control to MULTIPLE device initial threads
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for)

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 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);

}
```

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

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.

A more complicated example:

Jacobi iteration: OpenACC (GPU)

```
#pragma acc data copy(A), create(Anew)
while (err>tol && iter < iter_max){
   err = 0.0;
   #pragma acc parallel loop reduction(max:err)
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);
         err = max(err,abs(Anew[j][i] - A[j][i]));
       }
    }
    #pragma acc parallel loop
    for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
       }
    }
    iter ++;
}
```

Create a data region on the GPU. Copy A once onto the GPU, and create Anew on the device (no copy from host)

Copy A back out to host … but only once

Source: based on Mark Harris of NVIDIA®, “Getting Started with OpenACC”, GPU technology Conf., 2012

A more complicated example:

Jacobi iteration: OpenMP target directives

```
#pragma omp target data map(A) map(alloc:Anew)
while (err>tol && iter < iter_max){
   err = 0.0;
   #pragma target
   #pragma omp teams loop reduction(max:err)
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);
         err = max(err,abs(Anew[j][i] - A[j][i]));
       }
    }
    #pragma omp target
    #pragma omp teams loop
    for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
       }
    }
    iter ++;
}
```

Create a data region on the GPU. Map A and Anew onto the target device

Copy A back out to host … but only once

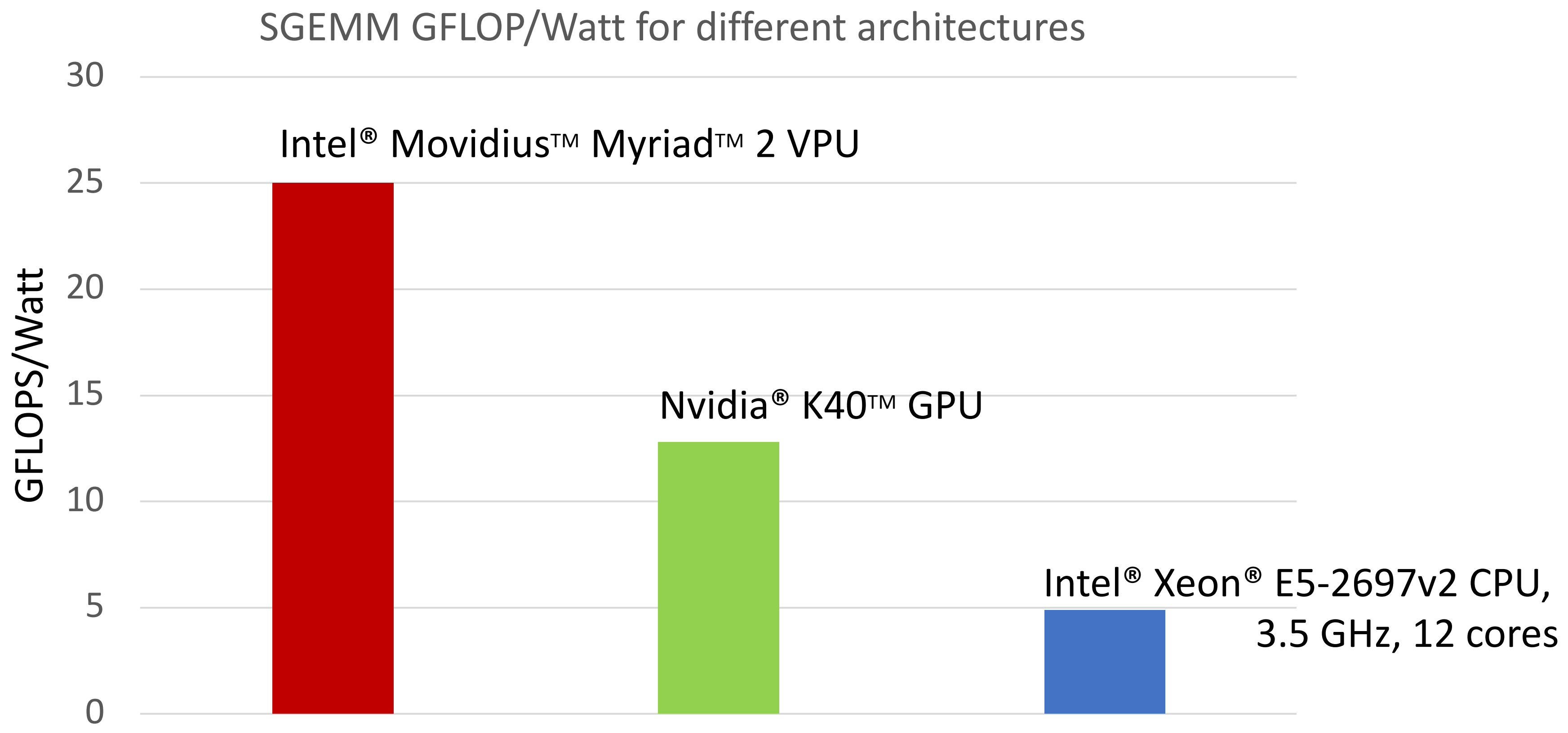
Why so many ways to do the same thing?

- 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.

What does the future hold for parallel programming?

If you care about power, the world is heterogeneous?

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

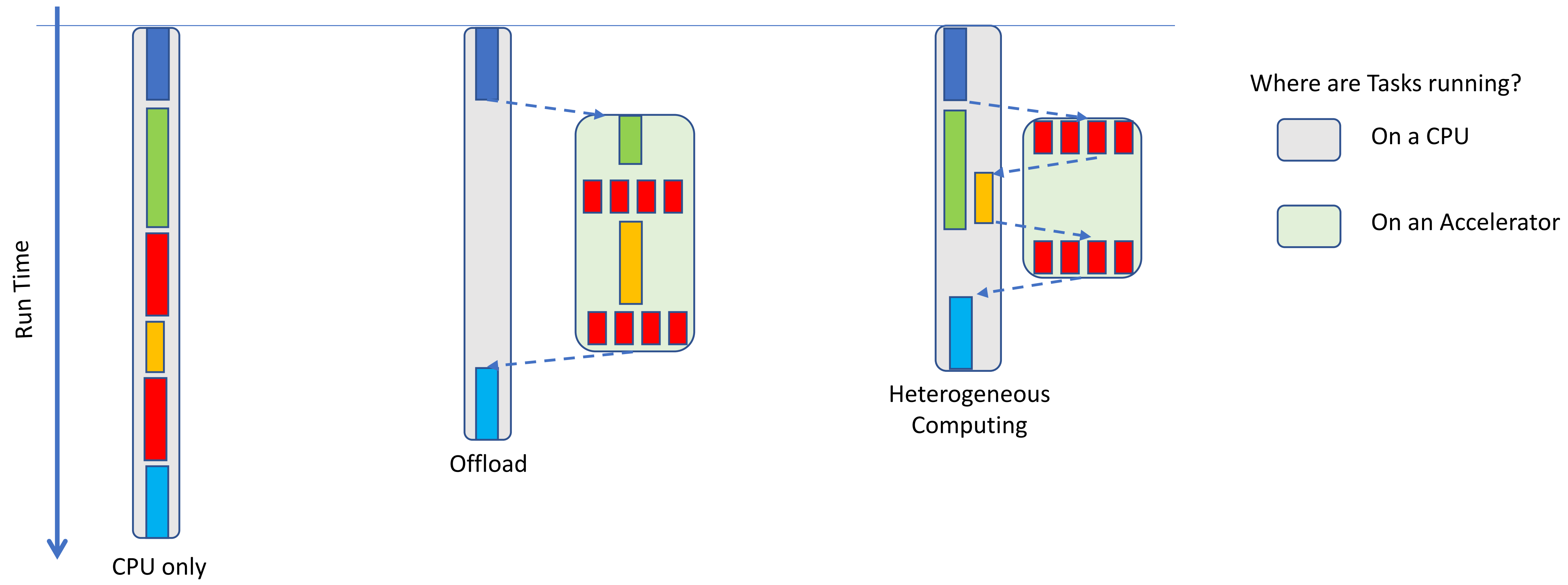


Hence, future systems will be increasingly heterogeneous … GPUs, CPUs, FPGAs, and a wide range of accelerators

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

Offload vs. Heterogeneous computing

- **Offload**: The CPU moves work to an accelerator and waits for the answer.
- **Heterogeneous Computing**: Run sub-problems in parallel on the hardware best suited to them.



Example: Single-cell RNA-Seq benchmark (SCANPY)

- SCANPY … a widely used tool for studying gene expression. All data are elapsed time in seconds
- We started with results from an Nvidia blog (Example 2 from link), optimized code for one socket of Intel® Xeon® 8380 CPU and then "simulated" heterogeneous computing result by taking the faster of CPU and GPU execution times.

Pipeline stages	64 vCPUs n1-highmem-64 (off-the-shelf Python)	A100 40Gb (Clara Parabricks)	ICX-1s, 40 cores (**optimized by Intel**)
Data Loading & Preprocessing	1120	475	15.7
PCA	44	17.8	5.0
T-SNE	6509	37	205.6
K-means (single iteration)	148	2	7.1
KNN	154	62	59.8
UMAP	2571	21	84.5
Louvain clustering	1153	2.4	6.0
Leiden clustering	6345	1.7	28.4
Reanalysis of subgroup	255	17.9	22.5
Rest	39	49.2	49.0
End-to-End runtime	**18338**	**686**	**483.6**

Redacted

Imagine mixing the best of the CPU and GPU numbers. What would the performance look like?

Lessons learned:

- Be careful comparing unoptimized python to hand-tuned CUDA code
- GPUs are great. So are CPUs if you fully utilize all the cores and vector units.
- What you really want is the best of both worlds. **You want heterogeneous computing!**

See Backup for workloads and configurations. Results may vary.

Clara Parabricks: Nvidia solution stack built on RAPIDS for healthcare applications

https://github.com/clara-parabricks/rapids-single-cell-examples

github repository as of Dec 16, 2020

This column shows the potential of heterogenous computing. We ignored extra communication and synchronization overhead, so actual runtimes would be slightly greater.

Source: Github repository as of Dec 16, 2020 - Example 2: Single-cell RNA-seq of 1.3 Million Mouse Brain Cells comparing CPU (n1-highmem-64 64 vCPUs) vs GPU (n1-highmem-16. https://github.com/clara-parabricks/rapids-single-cell-examples. Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.
1S Ice Lake: See Backup for workloads and configurations. Results may vary.

Five Epochs of Distributed Computing*

Epoch starting date	Defining limitations	Application	Interaction time and Network performance	Capability
First 1970	Rare connections to expensive computers	FTP, telnet, email	100 ms Low bandwidth high latency	People to computers
Second 1984	I/O wall, disks can't keep up	RPC, Client Server	10 ms 10 mbps	Computer to computer
Third 1990	Networking wall	MPP HPC, three-tier datacenter networks	1 ms 100 mbs → 1 Gbs	Services to services
Fourth 2000	Dennard scaling wall … per core plateau	Web search, planet-scale services	100 μs 10 Gbps flash	People to people
Fifth 2015	Per socket wall … accelerators take off	Machine Learning, data centric computing	10 μs 200 Gbps → 1 Tbps	People to insights

*The five Epochs of distributed computing, Amin Vahdat of Google: SIGCOMM Lifetime achievement award keynote, 2020.

The Eight Fallacies of Distributed Computing

(Peter Deutsch of Sun Microsystems, 1994 … item 8 added in 1997 by James Gosling)

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

The Eight Fallacies of Distributed Computing

(Peter Deutsch of Sun Microsystems, 1994 … item 8 added in 1997 by James Gosling)

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

1. The network is reliable
2. Latency is low and fixed
3. Bandwidth is high and fixed
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is negligible
8. The network is homogeneous

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

The Eight Fallacies of Distributed Computing

(Peter Deutsch of Sun Microsystems, 1994 … item 8 added in 1997 by James Gosling)

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

Cloud

✗ 1. ~~The network is reliable~~
✗ 2. ~~Latency is low and fixed~~
✗ 3. ~~Bandwidth is high and fixed~~
✗ 4. ~~The network is secure~~
✗ 5. ~~Topology doesn't change~~
✗ 6. ~~There is one administrator~~
✗ 7. ~~Transport cost is negligible~~
✗ 8. ~~The network is homogeneous~~

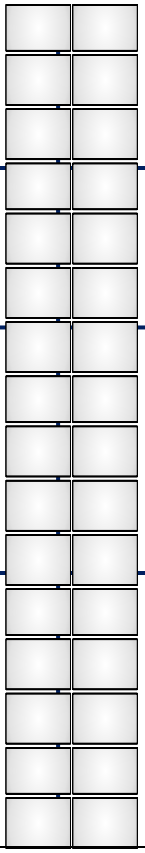
HPC Cluster

✓ 1. The network is reliable
✓ 2. Latency is low and fixed
✓ 3. Bandwidth is high and fixed
✓ 4. The network is secure
✓ 5. Topology doesn't change
✓ 6. There is one administrator
✗ 7. ~~Transport cost is negligible~~
✓ 8. The network is homogeneous

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

The three domains of parallel programming

Platform*	Laptop or server	HPC Cluster	Cloud
Execution Agent	Threads	Processes	Microservices
Memory	Single Address Space	Distributed memory, local memory owned by individual processes	Distributed object store (in memory) backed by a persistent storage system
Typical Execution Pattern	Fork-join	SPMD	Event driven tasks, FaaS, and Actors



Laptop/server and cluster models work well together.

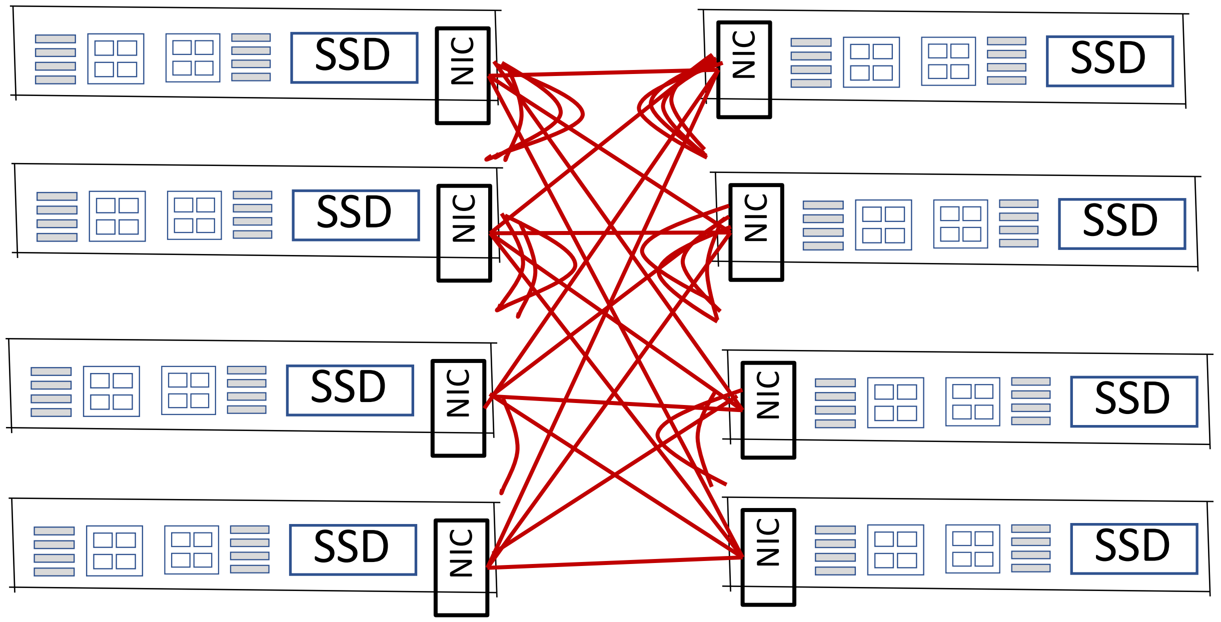
An impenetrable wall separates them from the cloud-native world

The sixth Epoch of Distributed Computing

Epoch starting date	Defining limitations	Application	Interaction time and Network performance	Capability
First 1970	Rare connections to expensive computers	FTP, telnet, email	100 ms Low bandwidth high latency	People to computers
Second 1984	I/O wall, disks can't keep up	RPC, Client Server	10 ms 10 mbps	Computer to computer
Third 1990	Networking wall	MPP HPC, three-tier datacenter networks	1 ms 100 mbs → 1 Gbs	Services to services
Fourth 2000	Dennard Scaling Wall … per core plateau	Web search, planet-scale services	100 μs 10 Gbps flash	People to people
Fifth 2015	Per socket wall … accelerators take off	Machine Learning, data centric computing	10 μs 200 Gbps → 1 Tbps	People to insights
Sixth 2025	Speed of light	Dynamic, real-time AI, integrated from data-center to the edge with SDE*	100 ns 10 Tbs	People to experiences

* SDE: Software defined Everything, i.e. software defined networking, software defined infrastructure, software defined servers ... All at the same time … to dynamically construct systems to meet the needs of workloads.

Networking technology… replace generic data center network with a cluster of cliques



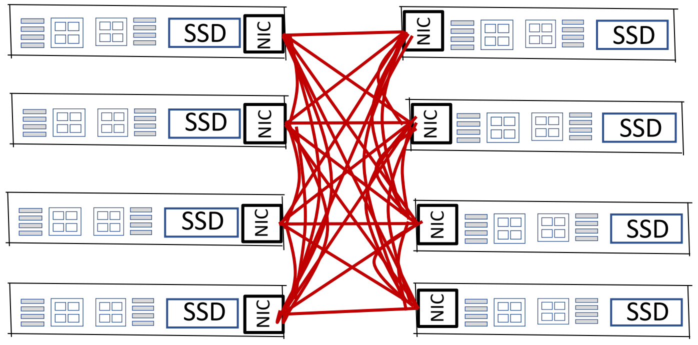
A clique: A graph where every vertex is connected to every other vertex

A Clique: a network of diameter one with O(¼N^2) bisection bandwidth

Combine with next generation optical networks to hit latencies of 100 ns

Latencies every engineer should know …

L1 cache reference 1.5 ns
L2 cache reference 5 ns
Branch misprediction 6 ns
Uncontended mutex lock/unlock 20 ns
L3 cache reference 25 ns
Main memory reference 100 ns
"Far memory"/Fast NVM reference 1,000 ns (1us)
Read 1 MB sequentially from memory 12,000 ns (12 us)
SSD Random Read 100,000 ns (100 us)
Read 1 MB bytes sequentially from SSD 500,000 ns (500 us)
Read 1 MB sequentially from 10Gbps network 1,000,000 ns (1 ms)
Read 1 MB sequentially from disk 10,000,000 ns (10 ms)
Disk seek 10,000,000 ns (10 ms)
Send packet California→Netherlands→California (150 ms)



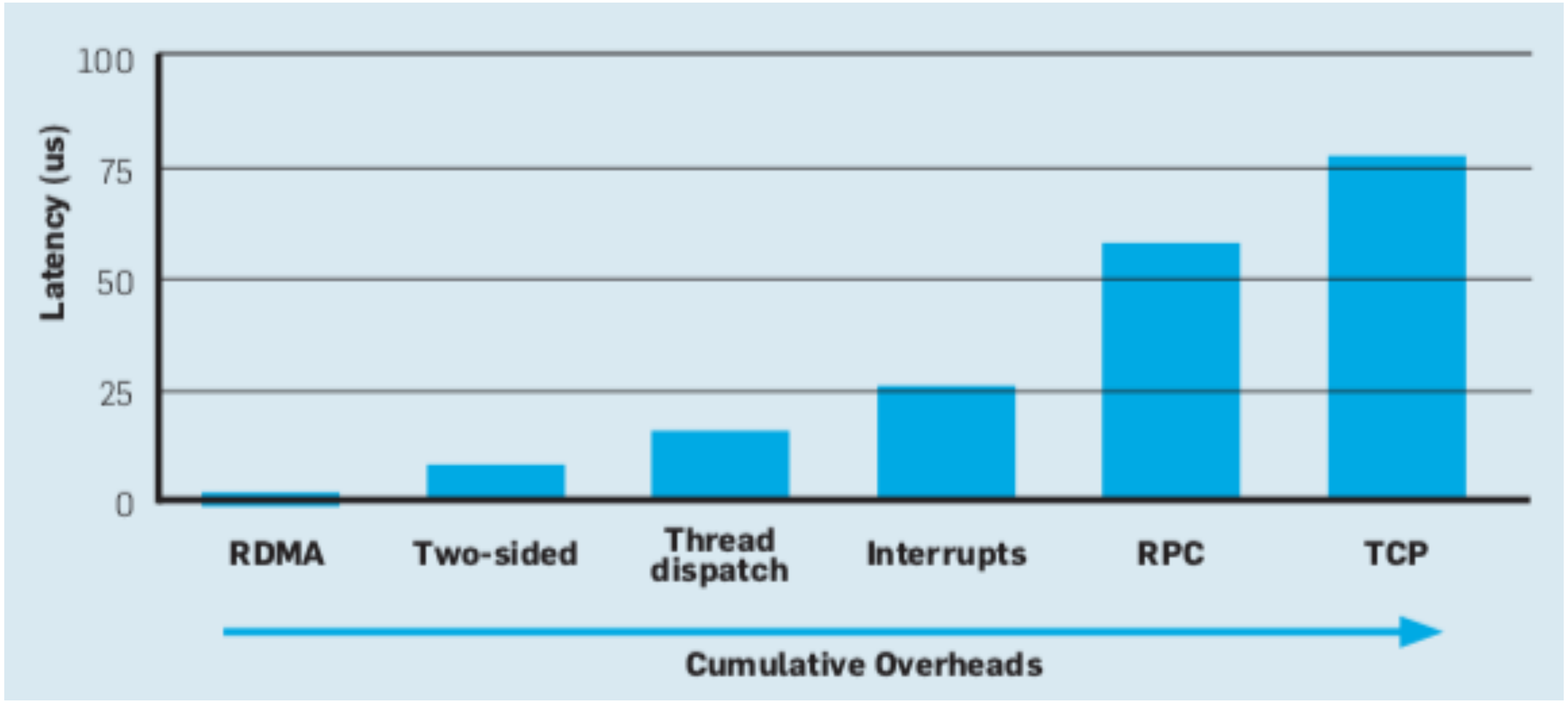
A cluster of nodes with a Clique network topology and low latency optical network…

Yields one hop network latencies on par with DRAM access latencies.

Source: **The Datacenter as a Computer: Designing Warehouse-Scale Machines**, Luiz Andre Barroso, Urs Holzle, Parthasarathy Ranganathan, 3rd edition, Morgan & Claypool, 2019.

Take out the big stuff & you're left with lots of µs overheads

All those SW overheads add up … like bricks that combine to build a networking-wall … turning a 2 µs network into a 100 µs network…

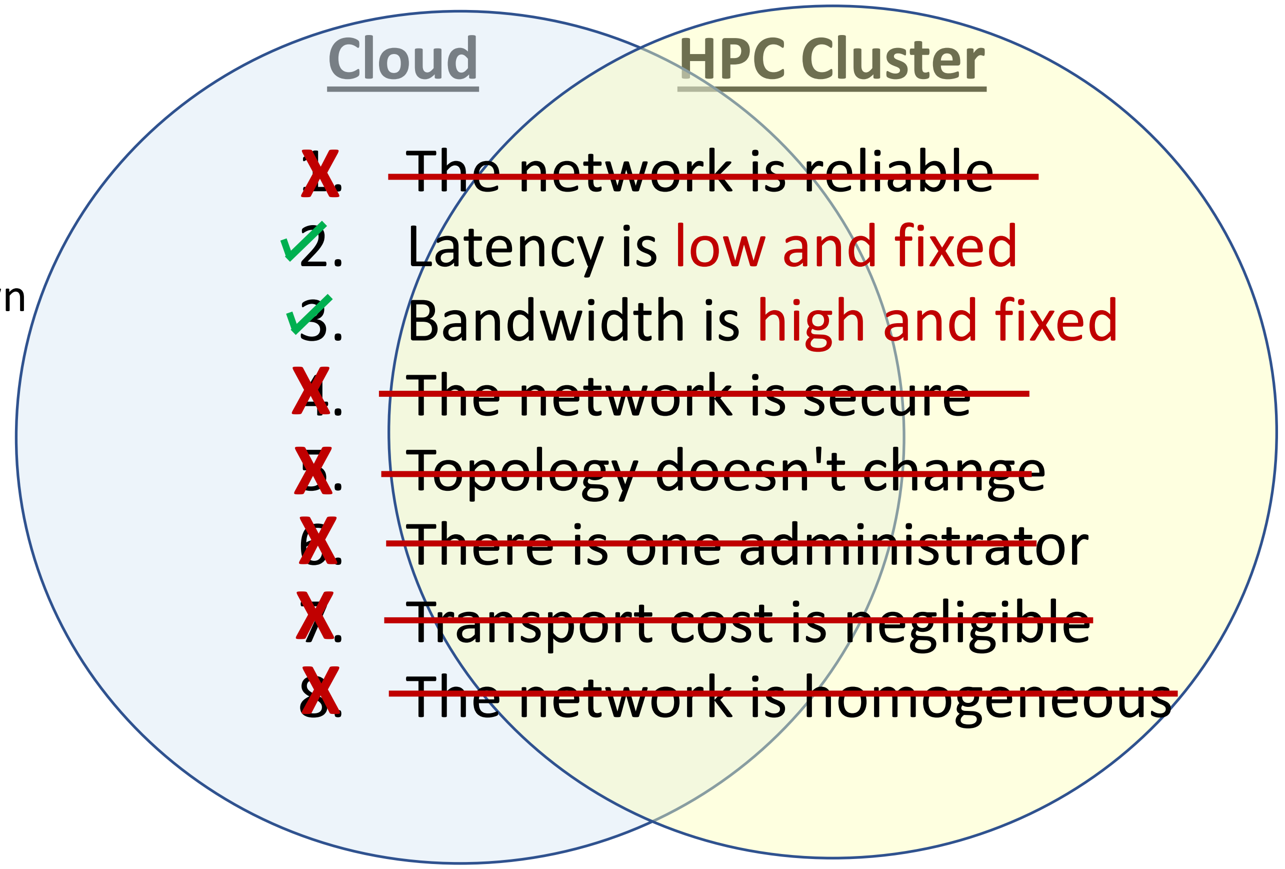


Computer Scientists need to rethink system SW stacks to minimize latencies … fast RDMA, reduce sync contention, low latency interrupt handlers, and more …. All to hit O(µs) latencies.

Source: Fig 1 from "Attack of the Killer Microseconds", Barroso, Marty, Patterson, and Ranganathan, Comm ACM vol 60, # 4, p. 48, 2017

In the sixth Epoch of Distributed Computing, cloud and cluster overlap … or even merge!

Chip-to-chip optical networks push latency down and bandwidth up



✗ 1. ~~The network is reliable~~
✓ 2. Latency is low and fixed
✓ 3. Bandwidth is high and fixed
✗ 4. ~~The network is secure~~
✗ 5. ~~Topology doesn't change~~
✗ 6. ~~There is one administrator~~
✗ 7. ~~Transport cost is negligible~~
✗ 8. ~~The network is homogeneous~~

Data Streaming Accelerator reduces tail latency.

P4/P5/P6 + Infrastructure Processing Units drive down latency and reduces jitter

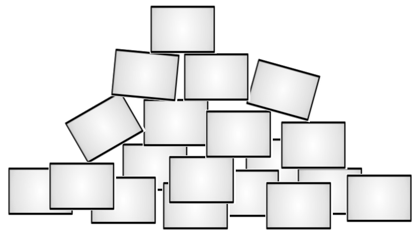
With Low Latencies, high bandwidths and stable performance, we can do loosely synchronous and synchronous applications in the cloud. The economics of the cloud vs dedicated HPC clusters means the cloud will dominate HPC

HPC applications will need to change to deal with reliability and network inhomogeneities.

The three domains of parallel programming

Platform*	Laptop or server	HPC Cluster	Cloud
Execution Agent	Threads	Processes	Microservices
Memory	Single Address Space	Distributed memory, local memory owned by individual processes	Distributed object store (in memory) backed by a persistent storage system
Typical Execution Pattern	Fork-join	SPMD	Event driven tasks, FaaS, and Actors

Advances in networking technology plus low-overhead software stacks optimized to reduce tail-latency will shatter this wall



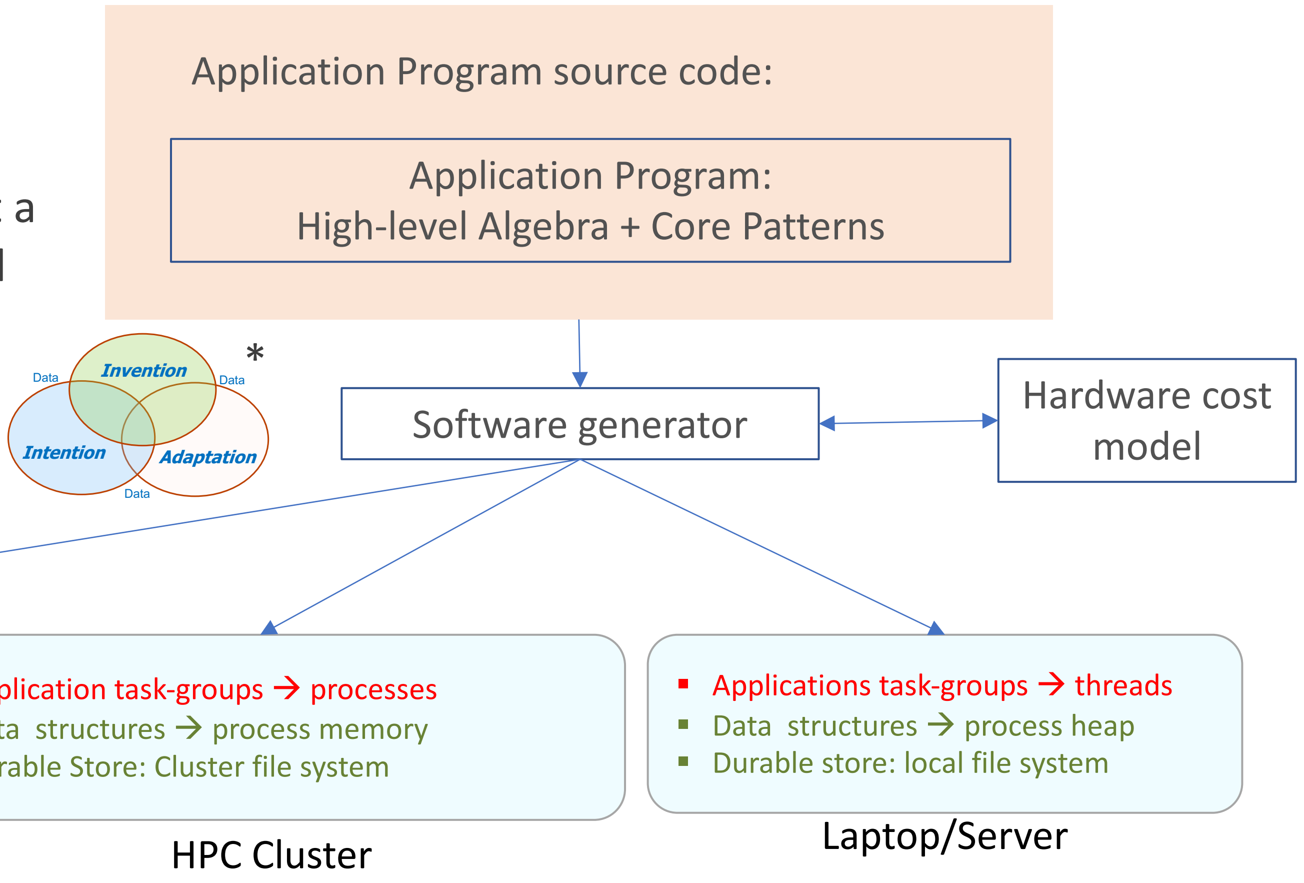
The three domains of parallel programming

Platform*	Laptop or server	HPC Cluster	Cloud
Execution Agent	Threads	Processes	Microservices
Memory	Single Address Space	Distributed memory, local memory owned by individual processes	Distributed object store (in memory) backed by a persistent storage system
Typical Execution Pattern	Fork-join	SPMD	Event driven tasks, FaaS, and Actors

There will always be a need for top-end scalable systems in supercomputer centers, but economics will push the bulk of scientific computing into the cloud.

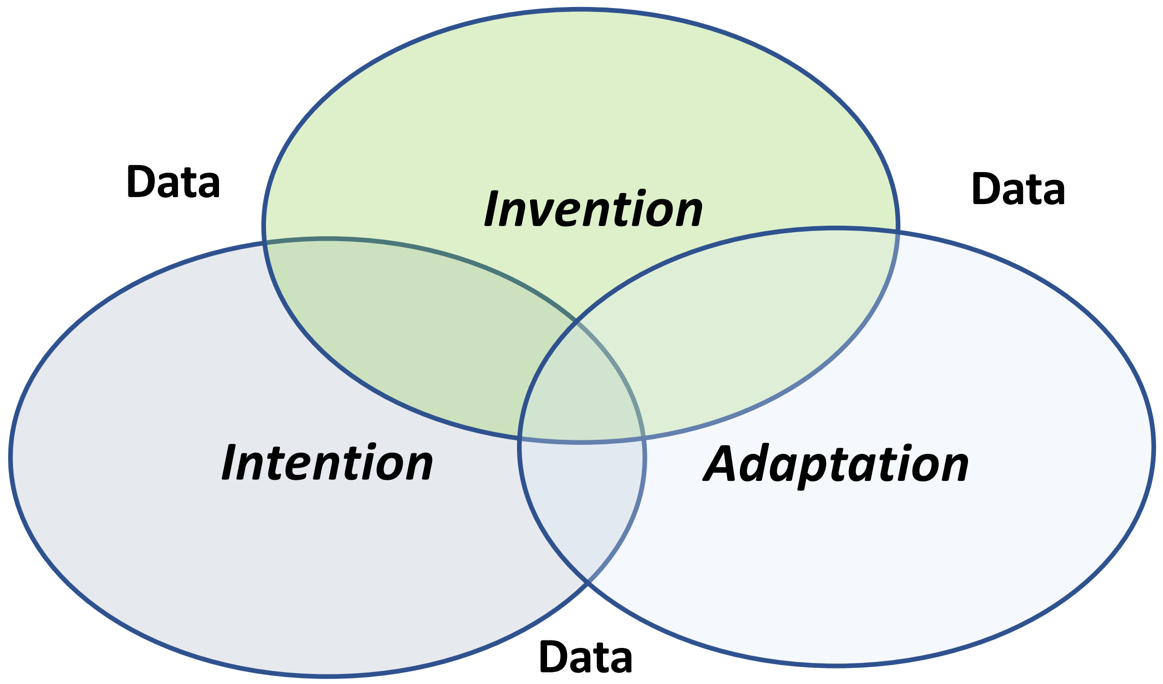
One codebase → many systems

- Performance, Productivity AND Portability … the database people “did it” with relational algebras and SQL.
- We can do it too with algebras over distributed data structures … that is a set of operators over values expressed in terms of our distributed data structures.
- If we get it right, we’ll have … declarative semantics that a software generator can turn into laptop, cluster or cloud programs.


Application Program source code:
Application Program:
High-level Algebra + Core Patterns
Invention
Intention
Adaptation
Data
Data
Data
*
Software generator
Hardware cost model
Application task-groups → microservices
Data structures → distributed object store
Durable store: Persistent cloud store (e.g. S3)
Cloud Native HPC
Application task-groups → processes
Data structures → process memory
Durable Store: Cluster file system
HPC Cluster
Applications task-groups → threads
Data structures → process heap
Durable store: local file system
Laptop/Server

*This is the logo of the machine programming research program I help lead inside Intel Labs

The Three Pillars of Machine Programming (MP)



- **MP is the automation of software development**
  - **Intention**: Discover the intent of a programmer
  - **Invention**: Create new algorithms and data structures
  - **Adaptation**: Evolve in a changing hardware/software world

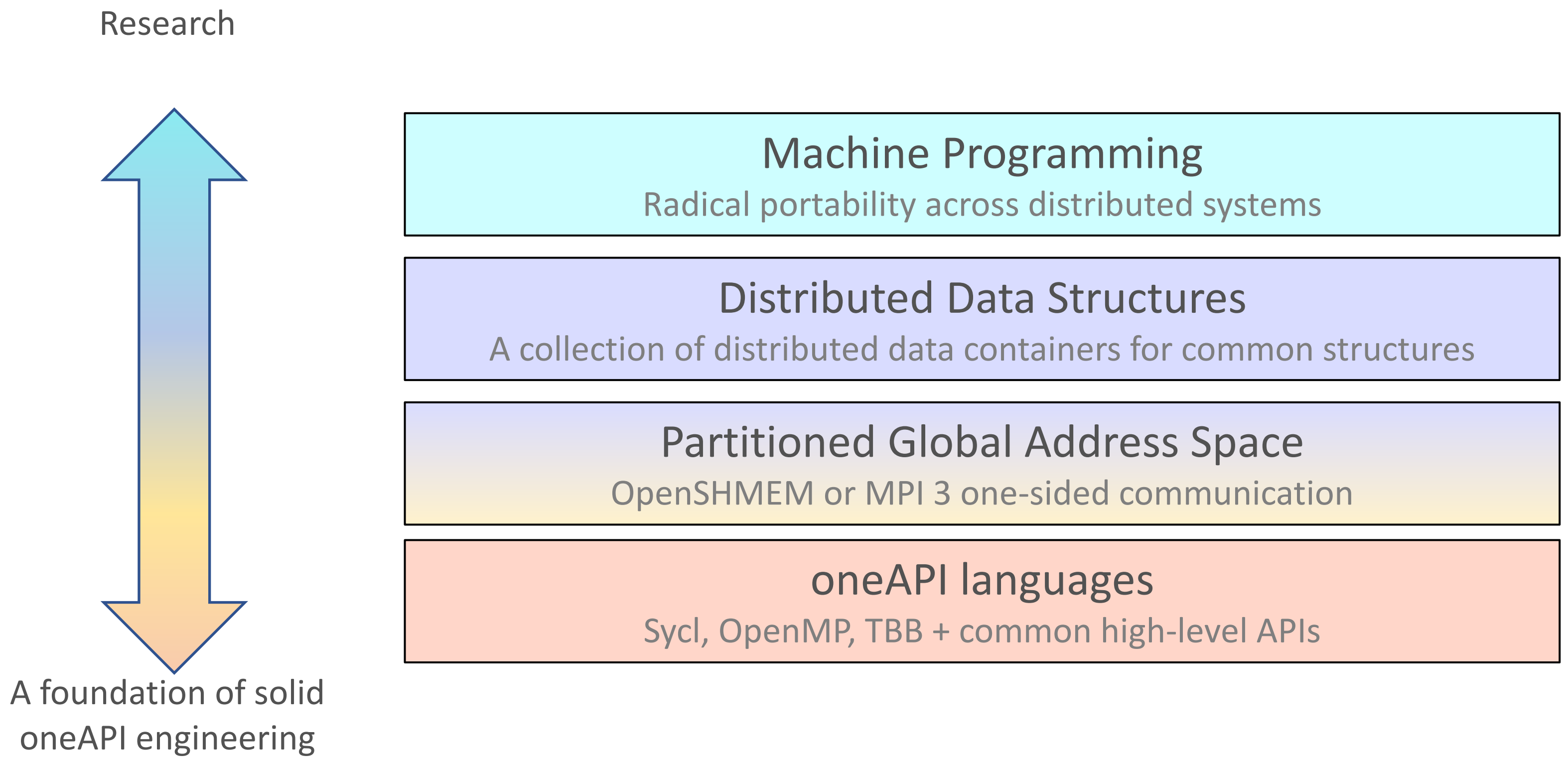
Justin Gottschlich, Intel Labs
Armando Solar-Lezama, MIT
Nesime Tatbul, Intel Labs
Michael Carbin, MIT
Martin Rinard, MIT
Regina Barzilay, MIT
Saman Amarasinghe, MIT
Joshua B Tenenbaum, MIT
Tim Mattson, Intel Labs

Summarized ~90 works.

Key efforts by Berkeley, Google, Microsoft, MIT, Stanford, UW and others.

oneAPI: A bridge to our heterogeneous/Distributed Future

My vision for how we bring oneAPI into a future dominated by power-optimized heterogenous chips organized into distributed systems.


Research
Machine Programming
Radical portability across distributed systems
Distributed Data Structures
A collection of distributed data containers for common structures
Partitioned Global Address Space
OpenSHMEM or MPI 3 one-sided communication
oneAPI languages
Sycl, OpenMP, TBB + common high-level APIs
A foundation of solid oneAPI engineering

The key to making this work … the programmer is in control and chooses the level of abstraction based on the programming task.

Summary

- Parallel computing is fun … but it can be hard.
- Fortunately, if you stick to the Big-3 and the core patterns of parallel computing for HPC, it’s not too overwhelming
  - The big 3: MPI, OpenMP, and “a GPU programming model”
  - Key Patterns: SPMD, loop level parallelism, geometric decomposition, divide and conquer, and SIMT
- Some day we’ll automate the hard-parts with Machine Programming, but that may be 10 years!!!!

SCANPY workload details and system configuration

ame	Intel® Xeon® Platinum 8380
Time	Jan 20, 2022
Manufacturer	Intel Corporation
Product Name	Intel® Xeon® Platinum 8380
BIOS Version	SE5C6200.86B.0020.P23.2103261309
OS	Rocky Linux release 8.5 (Green Obsidian)
Kernel	4.18.0-240.22.1.el8_3.crt6.x86_64
Microcode	0xd000270
IRQ Balance	enabled
CPU Model	Intel(R) Xeon(R) Platinum 8380 CPU @ 2.30GHz
Base Frequency	2.3GHz
Maximum Frequency	3.4GHz
All-core Maximum Frequency	2.5GHz
CPU(s)	40
Thread(s) per Core	2
Core(s) per Socket	40

Socket(s)	1
NUMA Node(s)	1
Prefetchers	
Turbo	Enabled
PPIN(s)	
Power & Perf Policy	Performance
TDP	270 watts
Frequency Driver	
Frequency Governer	Performance
Frequency (MHz)	
Max C-State	
Installed	Intel® Xeon® Platinum 8380 40c D1 DDR4 16*16GB@3200MHz - Mellanox HDR
Huge Pages Size	2048 kB
Transparent Huge Pages	Always
Automatic NUMA Balancing	Enabled

- The following was done to optimize the SCANPY benchmark
  - Data preprocessing - used warm file cache and multi-threaded using Numba JIT
  - PCA, K-means, KNN – Used the Intel extension for scikit-learn.
  - t-SNE - Used optimized version from Intel's oneDAL Library.
  - Parallelized quadtree building, sorting and summarization steps using Morton codes.
  - UMAP - optimized the UMAP code using AVX512/AVX2. Used MKL for eigenvalue computation.
  - Louvain and Leiden algorithms – collaborated with Katana Graph to get well optimized versions and integrated them into SCANPY.