Other Parallel Programming Environments

Tim Mattson

Intel Corp.

timothy.g.mattson@ intel.com

The Big Three

- In HPC, three programming environments dominate … covering the major classes of hardware.
  - OpenMP: Share memory systems … working hard to cover GPGPU as well.
  - MPI: distributed memory systems … though it can be nicely used on shared memory computers.
  - CUDA and OpenCL: GPGPU programming (use CUDA if you don't mind locking yourself to a single vendor)

- Even if you don't plan spend much time programming with these systems, it's good pedagogically to know what they are and how they work.

Parallel API's: MPI
the Message Passing Interface

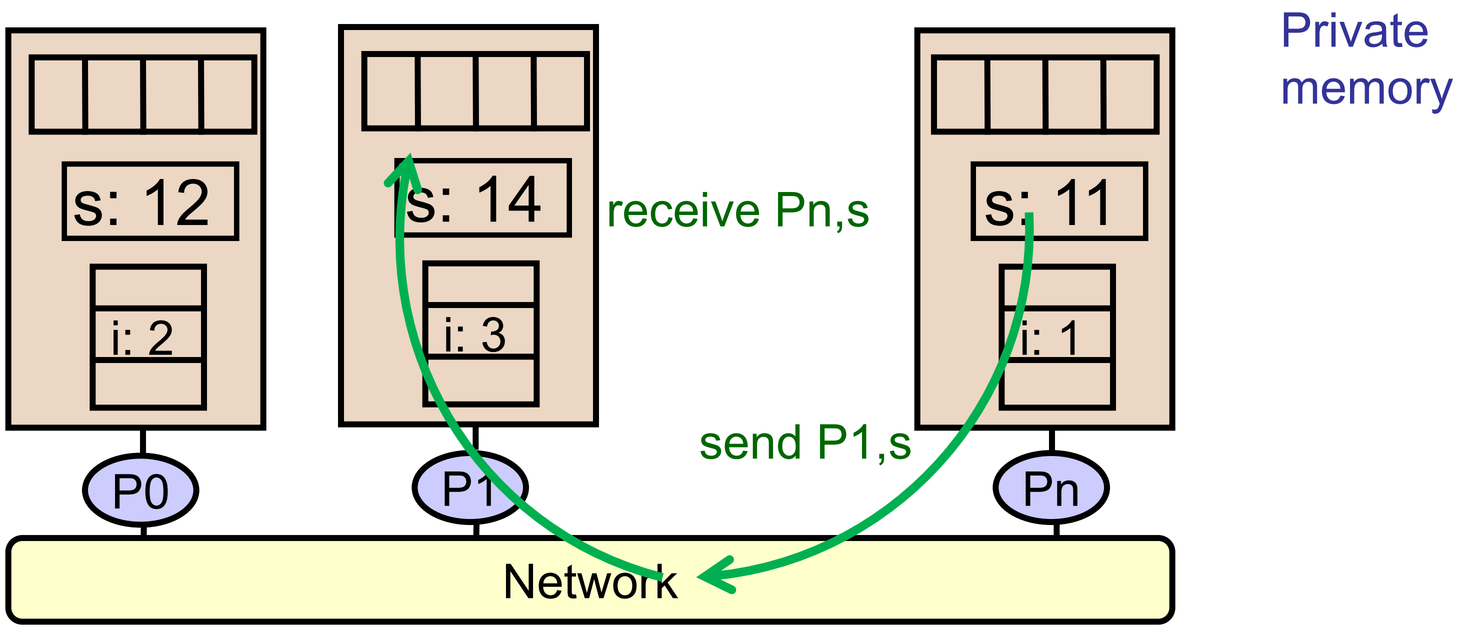
MPI_Type_contiguous MPI_Bcast MPI_Recv_init MPI_Group_size MPI_Scan MPI_Type_free MPI_Unpack MPI_COMM_WORLD MPI_Allgatherv MPI_Errhandler_create MPI_Group_compare C$OMP ORDERED MPI_Send_init MPI_Barrier MPI_Startall MPI_Start MPI_Reduce MPI_Test_cancelled MPI_Pack MPI_Recv MPI_Bsend_init omp_set_lock(lck) 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

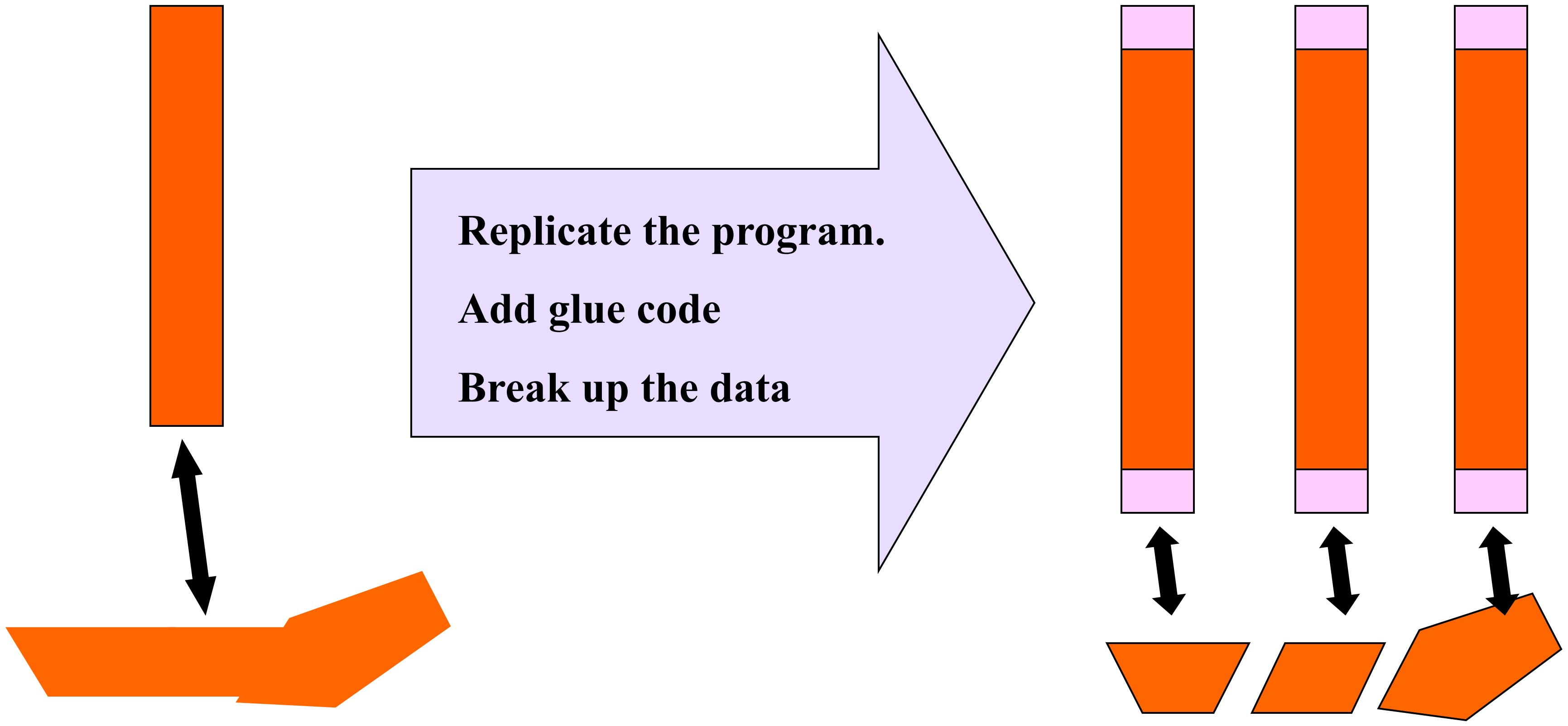
- Program consists of a collection of **named** processes.
  - **Number of processes almost always fixed at program startup time**
  - **Local address space per node -- NO physically shared memory.**
  - **Logically shared data is partitioned over local processes.**
- Processes communicate by explicit send/receive pairs
  - **Coordination is implicit in every communication event.**
  - **MPI (Message Passing Interface) is the most commonly used SW**



How do people use MPI? The SPMD Design Pattern

- 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.

A sequential program working on a data set



Replicate the program.

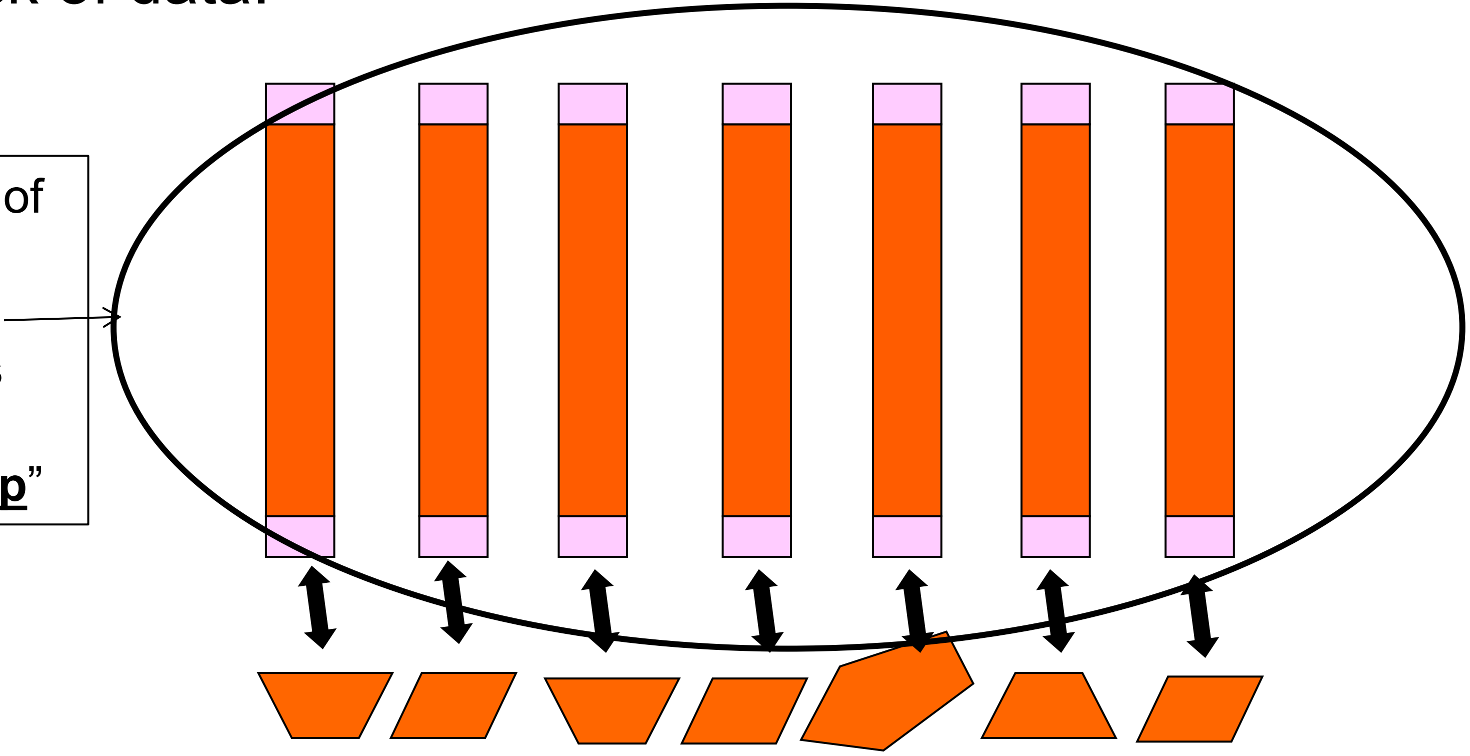
Add glue code

Break up the data

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

```c
#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. Default* 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, I'd run this program (hello) as:

  > mpirun –np 4 –hostfile hostf hello

- Where “hostf” is a file with the names of the cluster nodes, one to a line.
- What would this program would 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 **ar
    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;
}
```

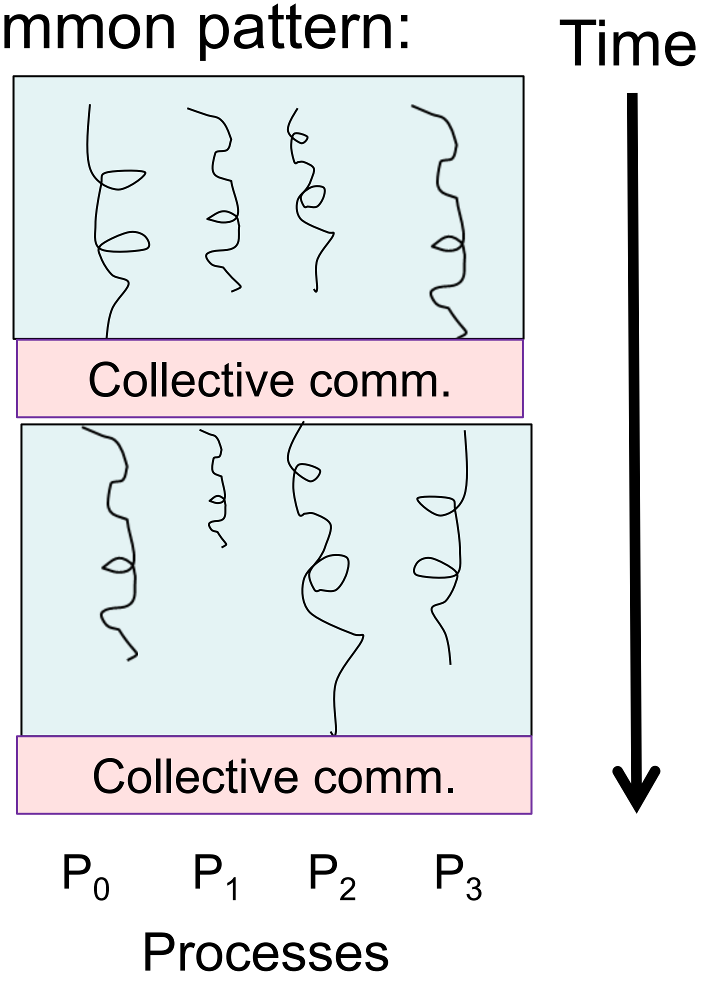
- On a 4 node cluster, I'd run this program (hello) as:

```
> mpirun –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.

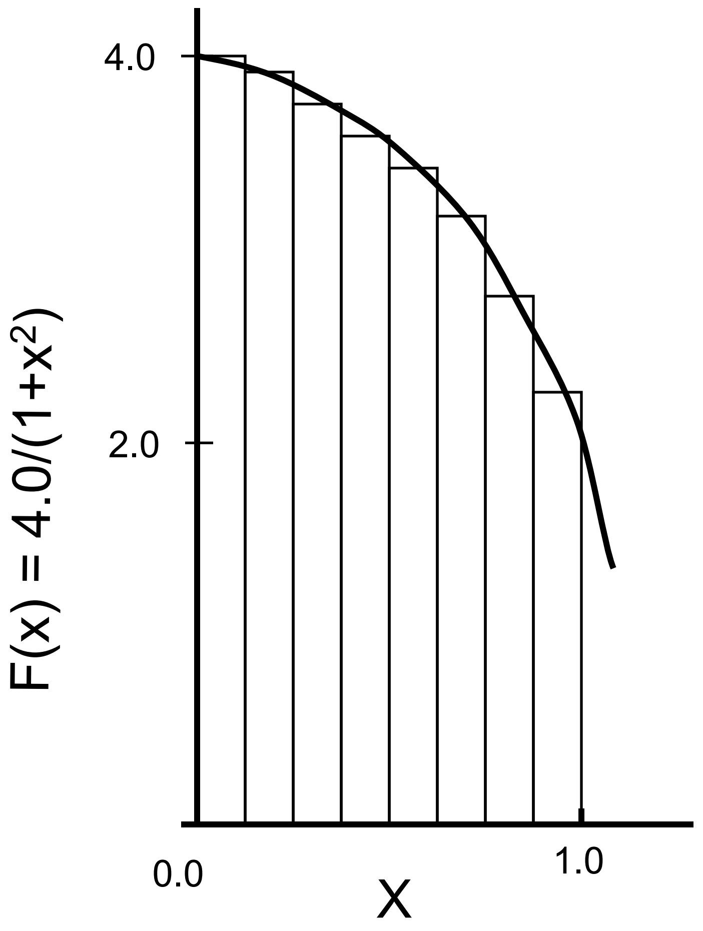
BSP: A common pattern used with MPI Programs

- 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 Data

```
int MPI_Send (void* buf, int count,
      MPI_Datatype datatype, int dest,
      int tag, MPI_Comm comm)

int MPI_Recv (void* buf, int count,
      MPI_Datatype datatype, int source,
      int tag, MPI_Comm comm,
      MPI_Status* status)
```

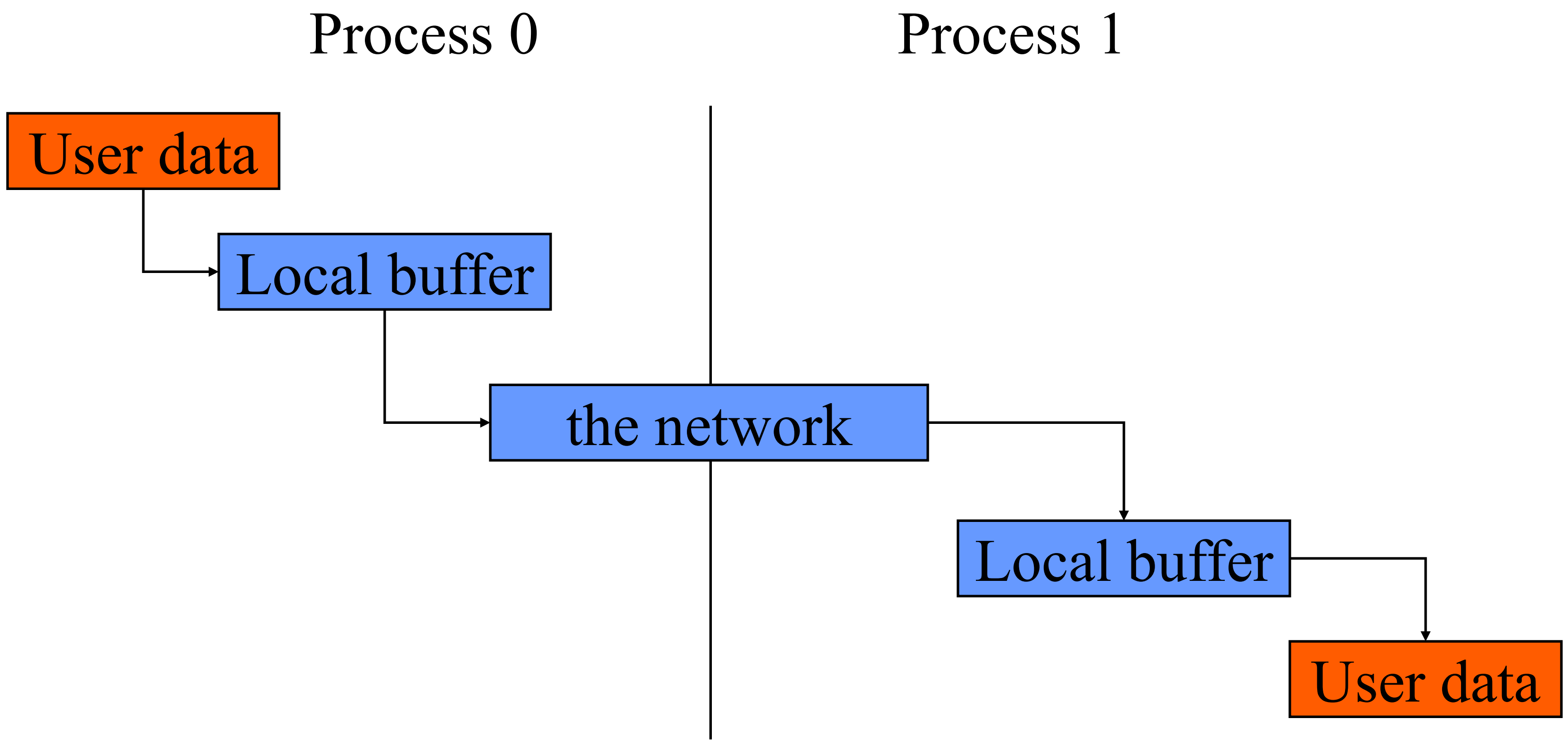
- **`MPI_Send`** performs a blocking send of the specified data (“count” copies of type “datatype,” stored in “buf”) to the specified destination (rank “dest” within communicator “comm”), with message ID “tag”

- **`MPI_Recv`** performs a blocking receive of specified data from specified source whose parameters match the send; information about transfer is stored in “status”

By “blocking” we mean the functions return as soon as the buffer, “buf”, can be safely used.

Buffers

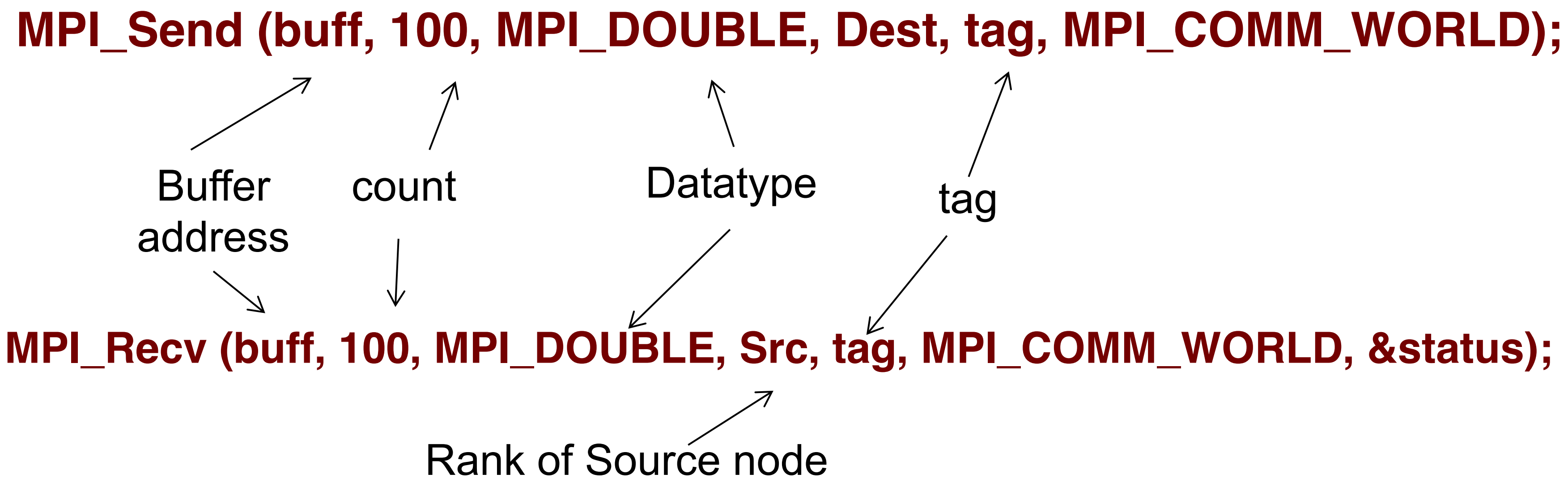
- Message passing has a small set of primitives, but there are subtleties
  - Buffering and deadlock
  - Deterministic execution
  - Performance
- When you send data, where does it go? One possibility is:



Derived from: Bill Gropp, UIUC

Send/Receive Syntax Details

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

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 = t^n

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

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

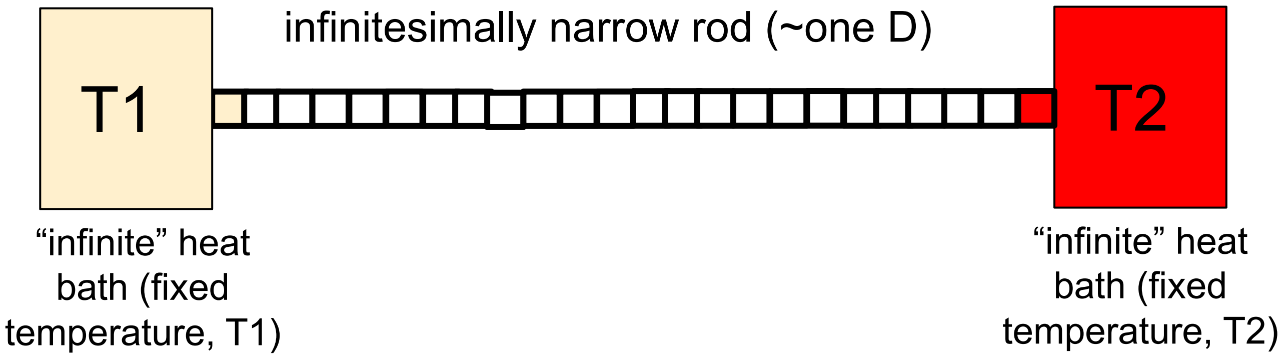
$$u_j^{n+1} = (1-2r)u_j^n + ru_{j-1}^n + ru_{j+1}^n \qquad r = k/h^2$$

- 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).

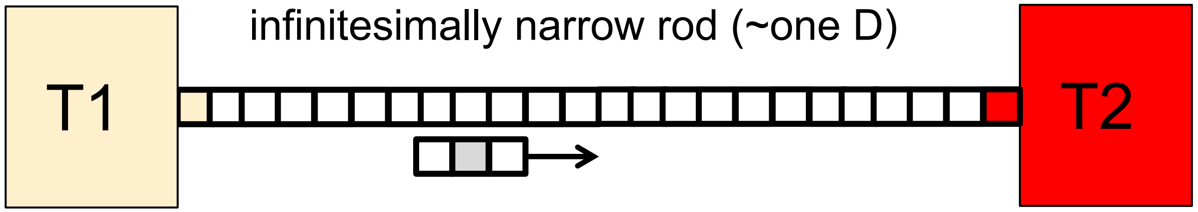
```
for (int t = 0; t < N_STEPS-1; ++t)
    for (int x = 1; x < N-1; ++x)
            u[t+1][x] = u[t][x] + r*(u[t][x+1] - 2*u[t][x] + u[t][x-1]);
```

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

Heat Diffusion equation

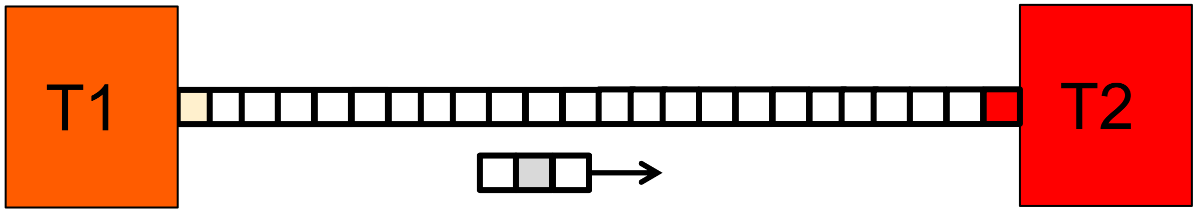

infinitesimally narrow rod (~one D)
T1
T2
“infinite” heat bath (fixed temperature, T1)
“infinite” heat bath (fixed temperature, T2)

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

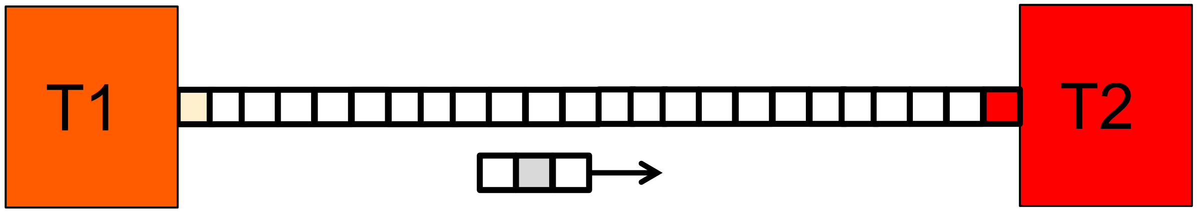


```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

   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

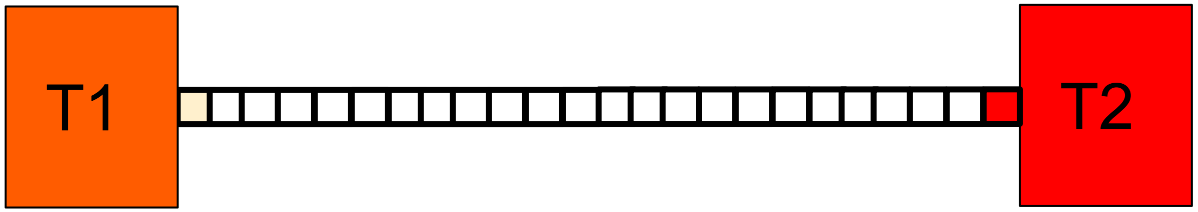


```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

   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?

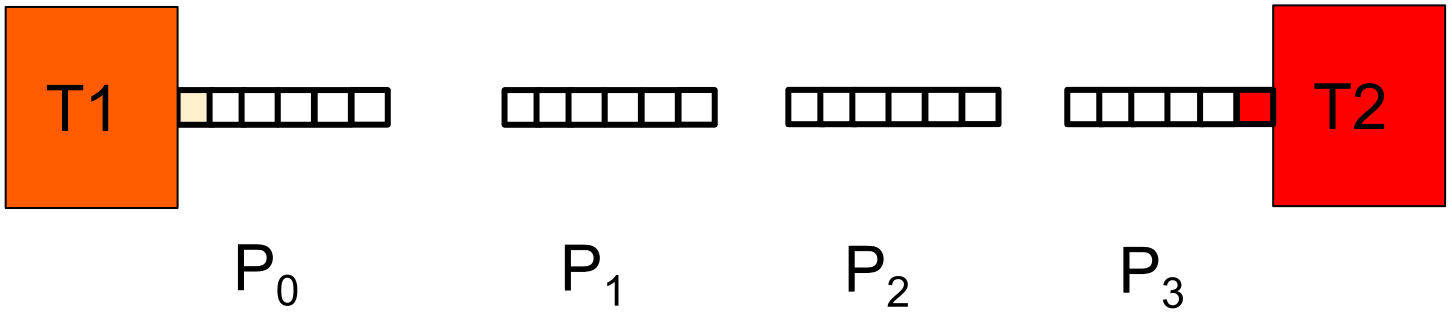
Heat Diffusion equation

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



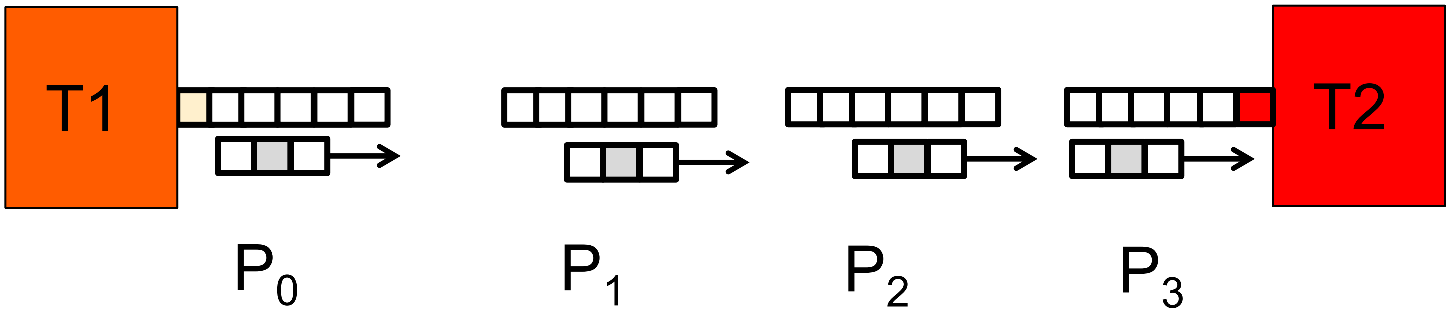
Heat Diffusion equation

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



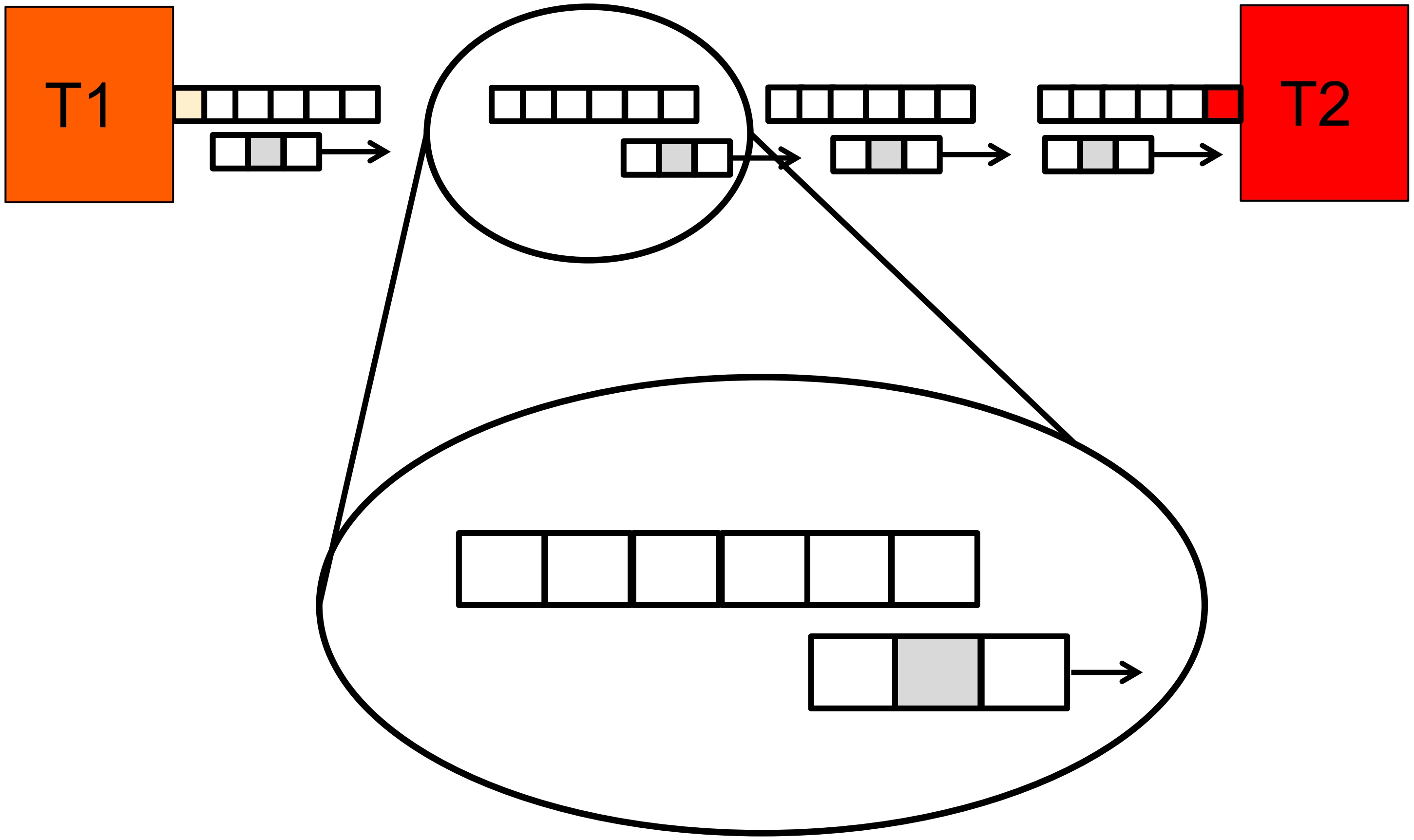
Heat Diffusion equation

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



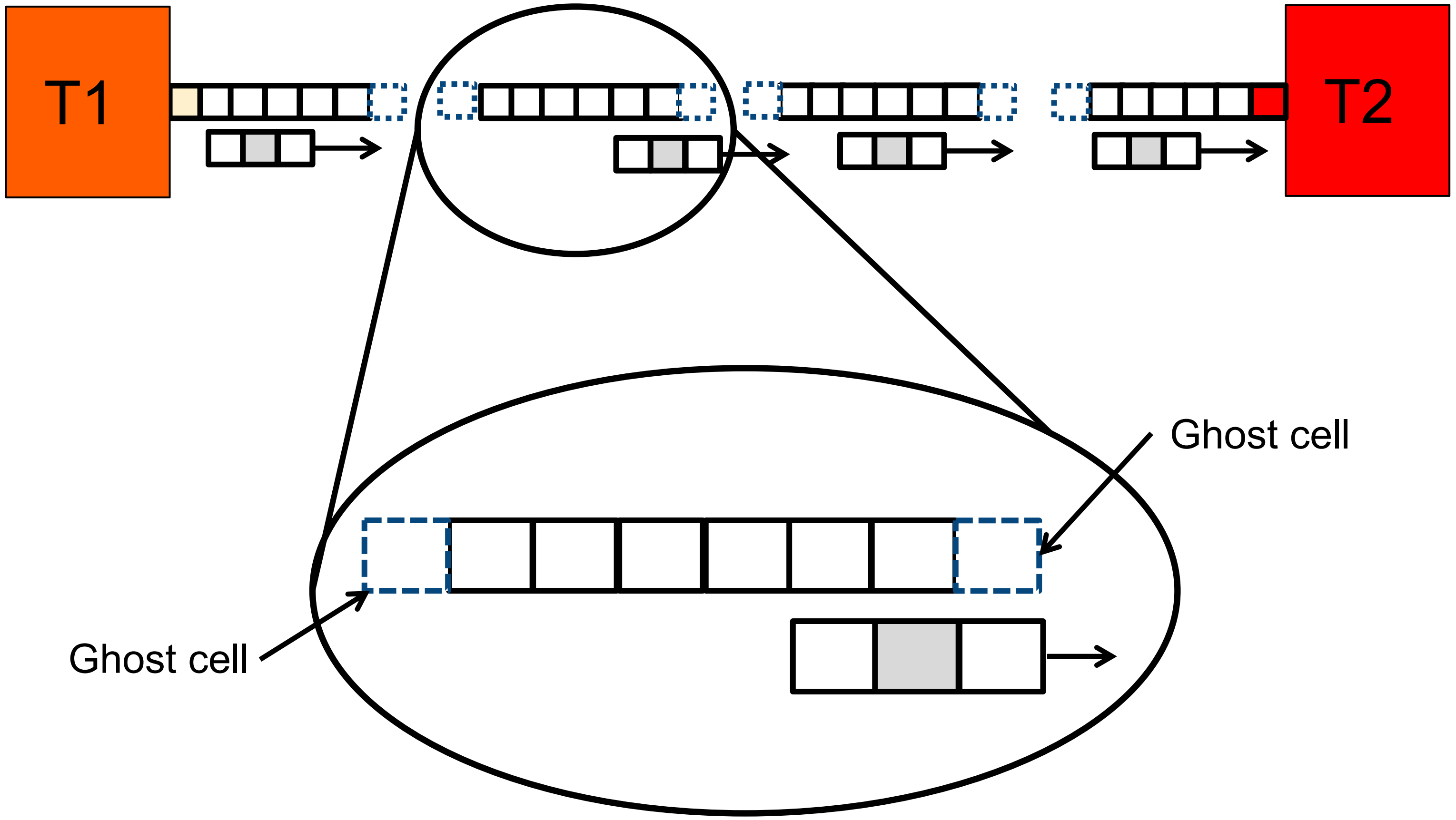
Heat Diffusion equation

- 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.

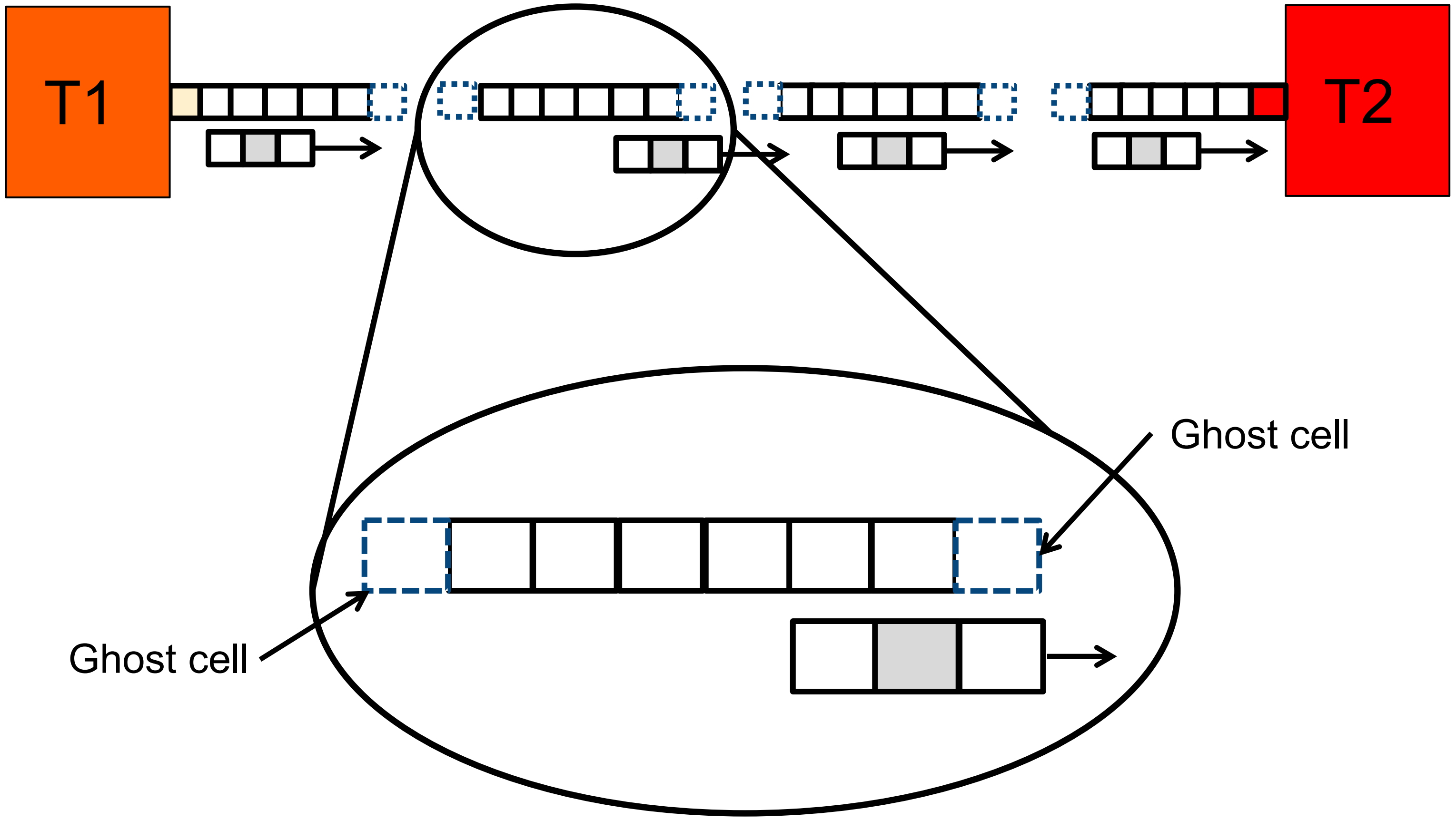


Geometric Decomposition

- 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"
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);

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

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

MPI_Finalize();
return 0;
```

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

Heat Diffusion MPI Example

```
/* continued from previous slide */

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

  if (myID != 0)
    up1[1] = u[1] + (k / (h*h)) * (u[1+1] - 2*u[1] + u[1-1]);

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

  temp = up1; up1 = u; u = temp;

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

MPI_Finalize();
return 0;
```

Temperature fields using local data and values from ghost cells.

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);
/* continued on next slide */
```

Send my "right" boundary value to my "right' neighbor

Receive my "left" ghost cell from my "left' neighbor

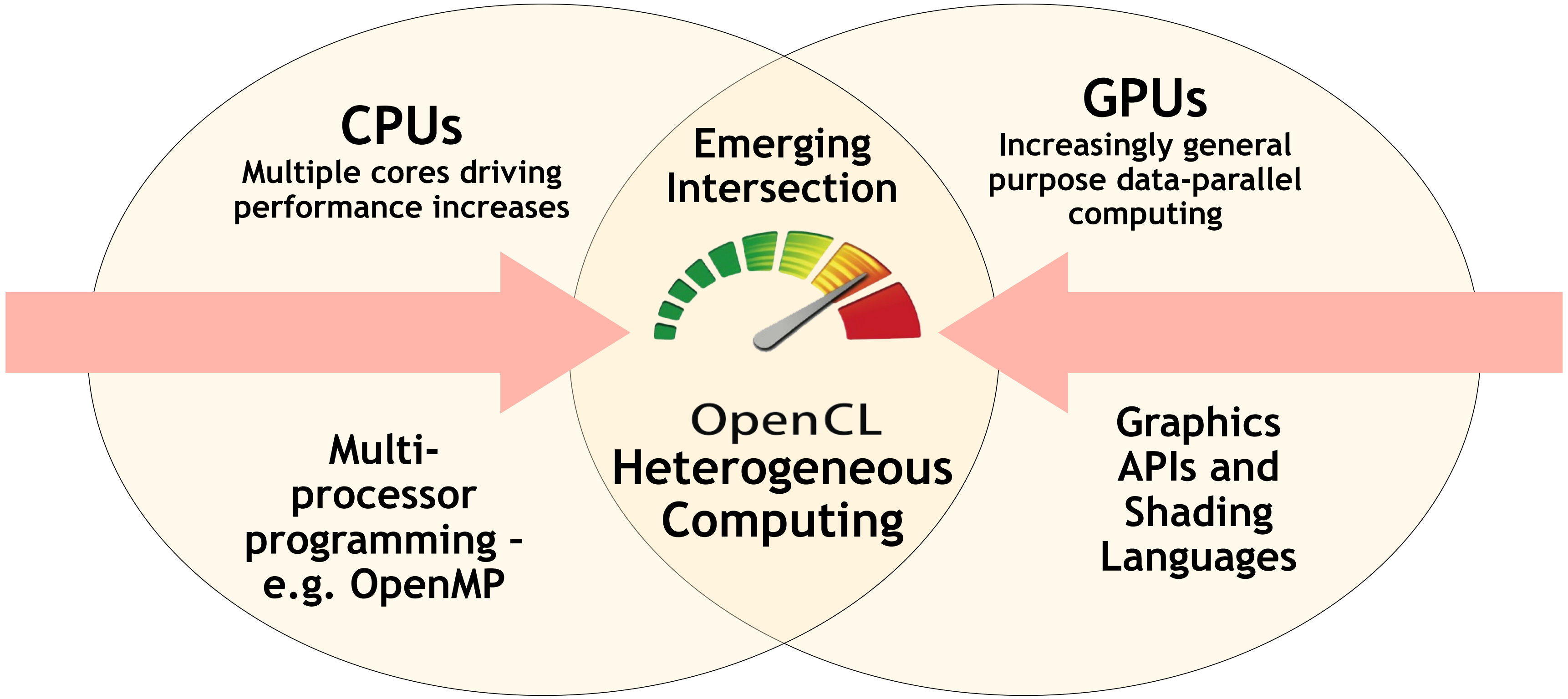
Send my "left" boundary value to my "left' neighbor

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.

Industry Standards for Programming Heterogeneous Platforms


CPUs
Multiple cores driving performance increases
Emerging Intersection
GPUs
Increasingly general purpose data-parallel computing
Multi-processor programming - e.g. OpenMP
OpenCL
Heterogeneous Computing
Graphics APIs and Shading Languages

OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

The BIG idea behind OpenCL

OpenCL

- OpenCL execution model … execute a <u>kernel</u> at each point in a problem domain.
  - E.g., process a 1024 x 1024 image with one kernel invocation per pixel or 1024 x 1024 = 1,048,576 kernel executions

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

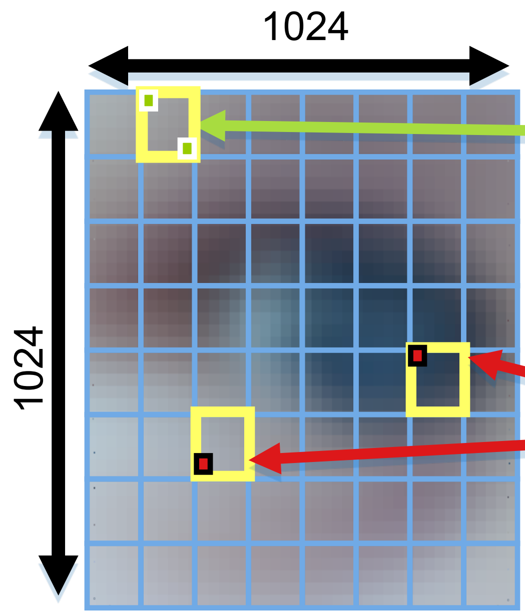
Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

An N-dimension domain of work-items

OpenCL

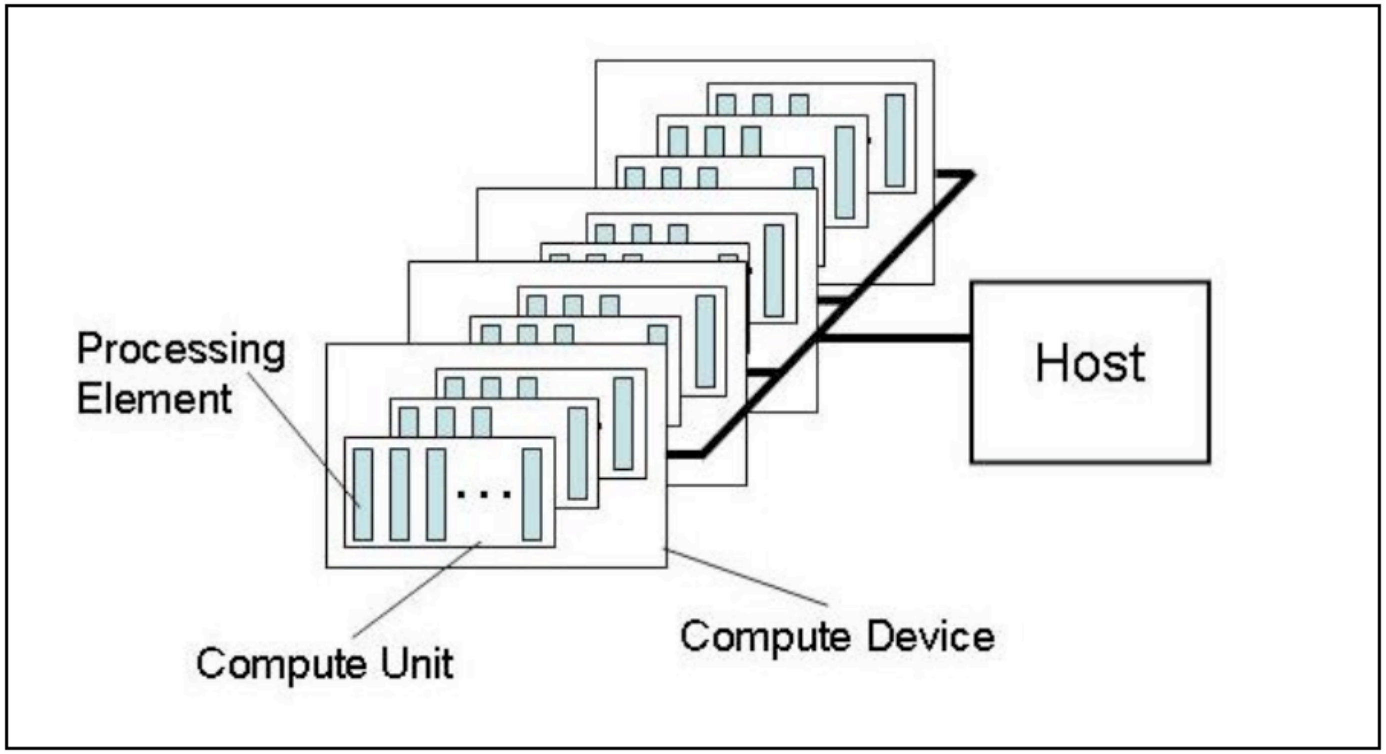
- Define an N-dimensioned index space that is "best" for your algorithm
  - Global Dimensions: 1024 x 1024 (whole problem space)
  - Local Dimensions: 128 x 128 (work group … executes together)



Synchronization between work-items possible only within workgroups: barriers and memory fences

Cannot synchronize outside of a workgroup

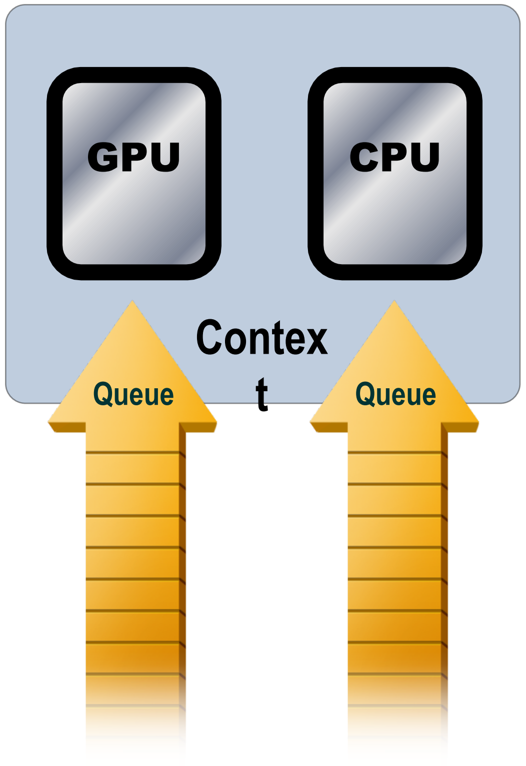
OpenCL Platform Model



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

OpenCL Execution Model

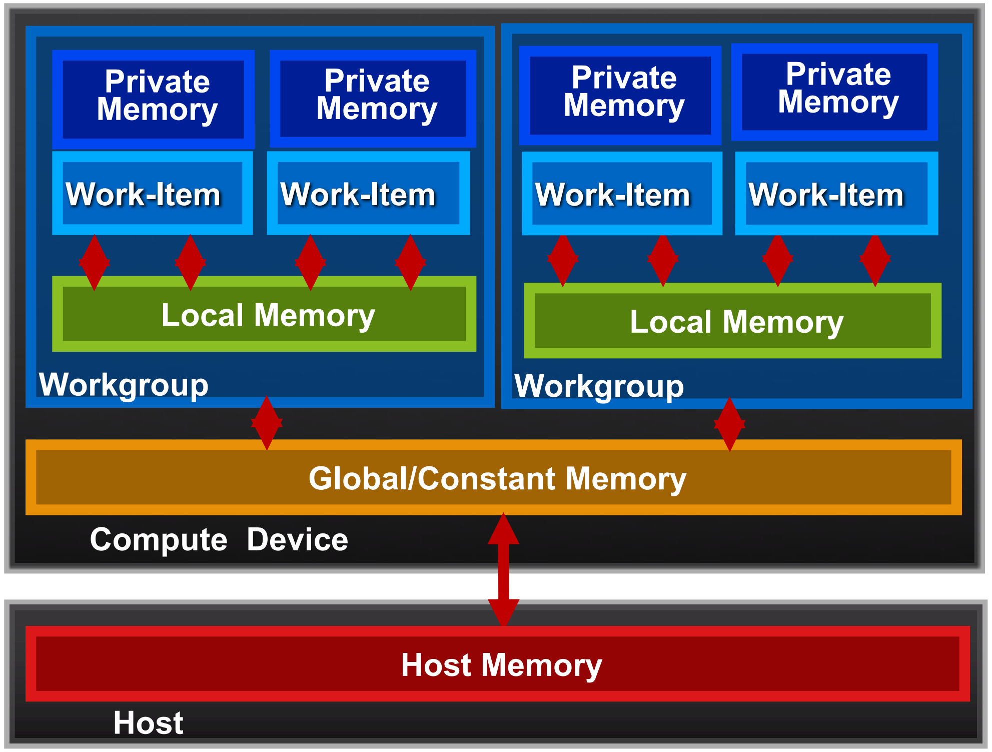
- An OpenCL application runs on a host which submits work to the compute devices.
  - **Work item**: the basic unit of work on an OpenCL device.
  - **Kernel**: the code for a work item. Basically a C function
  - **Program**: Collection of kernels and other functions (Analogous to a dynamic library)
  - **Context**: The environment within which work-items executes … includes devices and their memories and command queues.



- Applications queue kernel execution instances
  - Queued in-order … one queue to a device
  - Executed in-order or out-of-order

OpenCL Memory Model

- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a workgroup
- **Global/Constant Memory**
  - Visible to all workgroups
- **Host Memory**
  - On the CPU



Memory management is Explicit

You must move data from host -> global -> local … *and* back

Vector Addition - Kernel

```
__kernel void vec_add (__global const float *a,
                       __global const float *b,
                       __global       float *c)
 {
     int gid = get_global_id(0);
     c[gid] = a[gid] + b[gid];
}
```

Vector Addition: Host Program

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
   CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with
   context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
                                    NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
   devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,
   devices[0], 0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context,
   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
   sizeof(cl_float)*n, srcA,
                                        NULL);}
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY
   | CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
                                        NULL);
memobjs[2] =
   clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                        sizeof(cl_float)*n,
   NULL,
                                        NULL);
// create the program
program = clCreateProgramWithSource(context, 1,
   &program_source, NULL, NULL);
```

```
// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL,
                                            NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                  sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
   CL_TRUE, 0, n*sizeof(cl_float), dst, 0, NULL, NULL);
```

45

Vector Addition: Host Program

Define platform and queues

```
// [illegible]

// get the list of GPU devices associated with
   context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
                                  NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
   devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,
   devices[0], 0, NULL);

// allocate the buffer memory objects
```

Define Memory objects

```
                                          TR,

memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY
   | CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
   NULL);
memobjs[2] =
   clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                             sizeof(cl_float)*n,
   N
```

Create the program

```
// c
program = clCreateProgramWithSource(context, 1,
   &program_source, NULL, NULL);
```

Build the program

```
//
er                                  , NULL, NULL,
   NULL);
```

Create and setup kernel

```
/
k                                          LL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                 sizeof(cl_mem));
```

```
// set work-item dimensions
global_work_size[0]
```

Execute the kernel

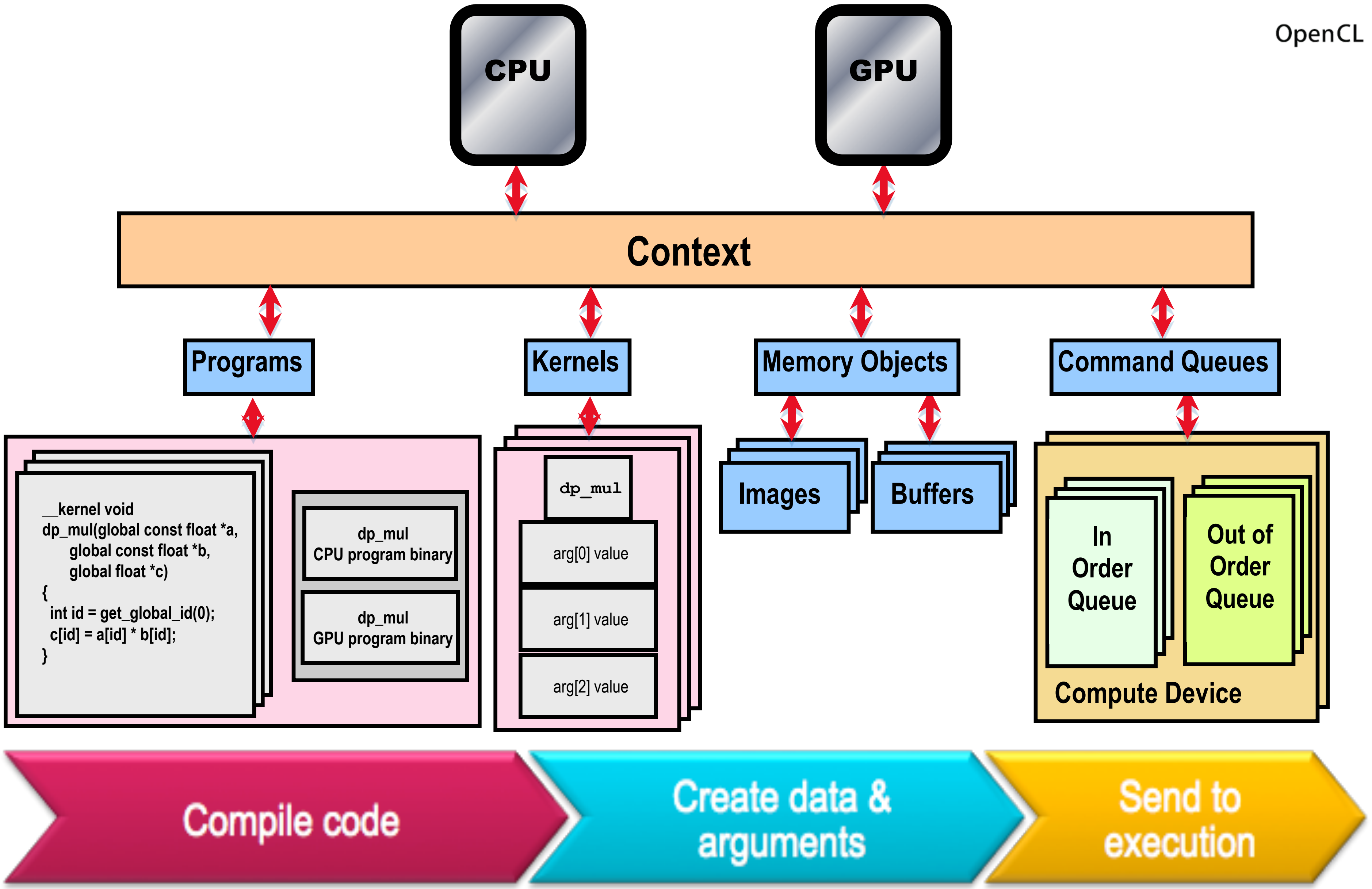
```
// execu
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);
```

Read results on the host

```
// rea
err =                                        TRUE,
   0, n*sizeof(cl_float), dst, 0, NULL, NULL);
```

It's complicated, but most of this is "boilerplate" and not as bad as it looks.

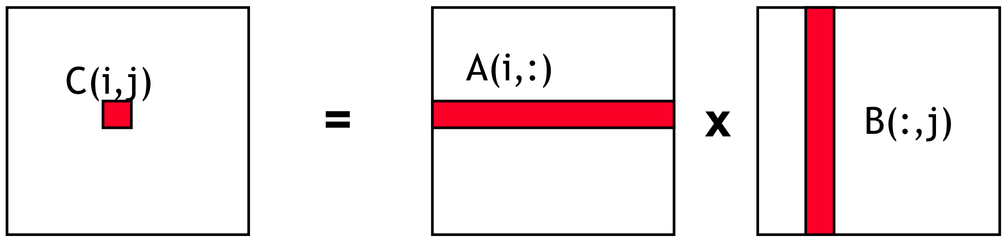
OpenCL summary



```
__kernel void
dp_mul(global const float *a,
      global const float *b,
      global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

Matrix multiplication example:
Naïve solution, one dot product per element of C

- Multiplication of two dense matrices.



Dot product of a row of A and a column of B for each element of C

- To make this fast, you need to break the problem down into chunks that do lots of work for sub problems that fit in fast memory (OpenCL local memory).

Matrix multiplication: sequential code

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

Matrix multiplication: sequential code

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

Let's get rid of all those ugly brackets

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (i = ib*NB; i < (ib+1)*NB; i++)
     for (jb = 0; jb < NB; jb++)
       for (j = jb*NB; j < (jb+1)*NB; j++)
         for (kb = 0; kb < NB; kb++)
           for (k = kb*NB; k < (kb+1)*NB; k++)
             C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Break each loop into chunks with a size chosen to match the size of your fast memory

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Rearrange loop nest to move loops over blocks "out" and leave loops over a single block together

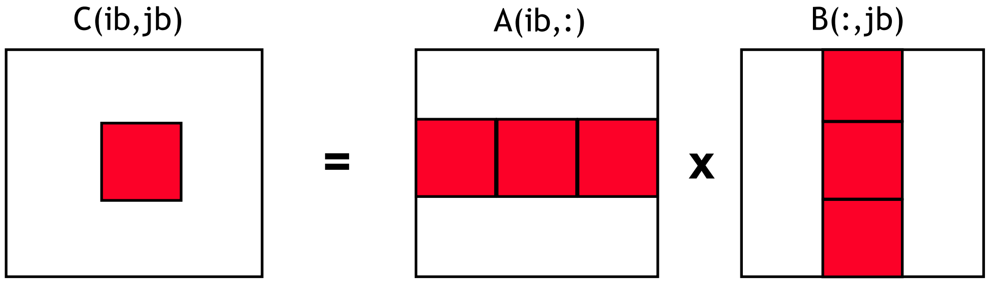
Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

This is just a local matrix multiplication of a single block

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)
       sgemm(C, A, B, …)    // C_{ib,jb} = A_{ib,kb} * B_{kb,jb}
```



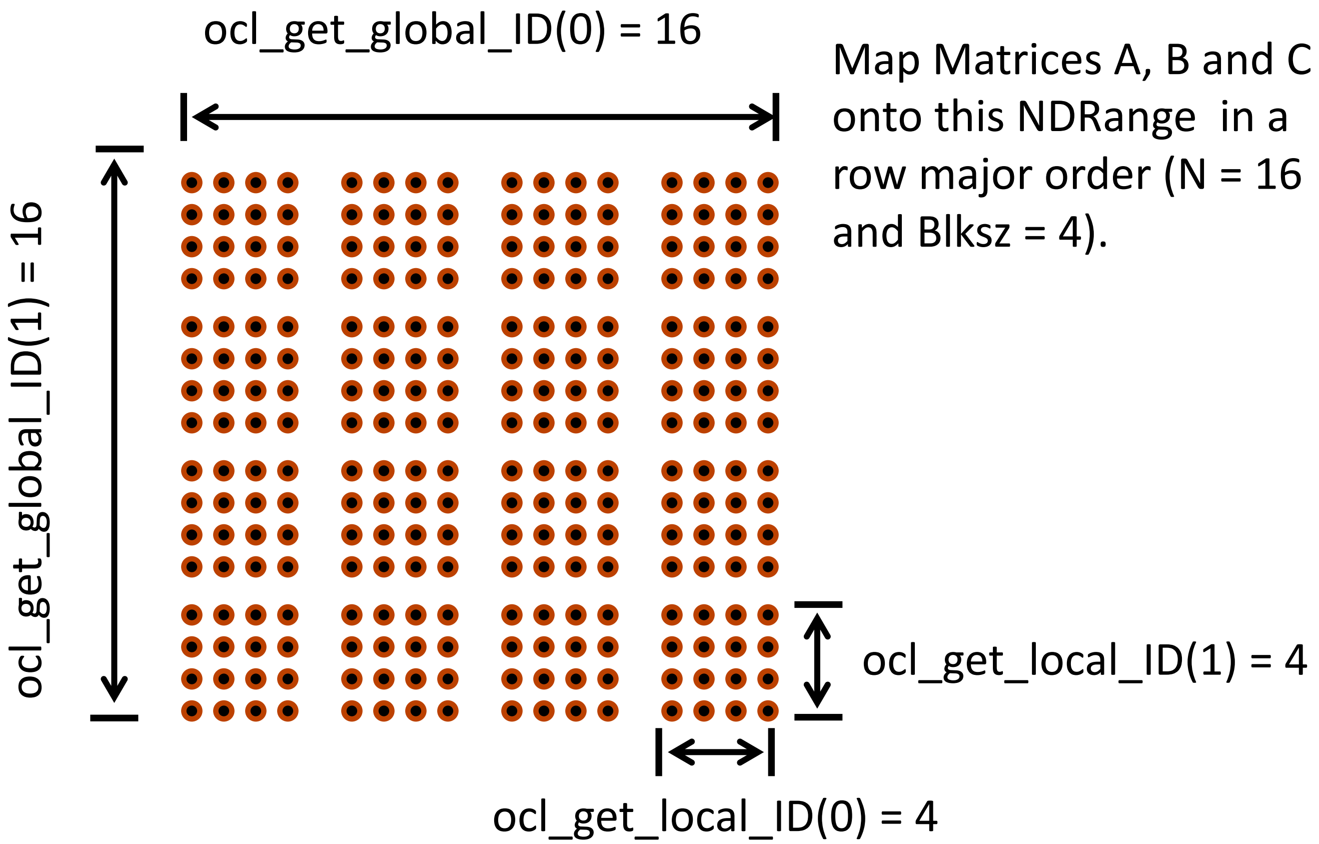
```
}
```

Note: sgemm is the name of the level three BLAS routine to multiply two matrices

Mapping into A, B, and C from each work item

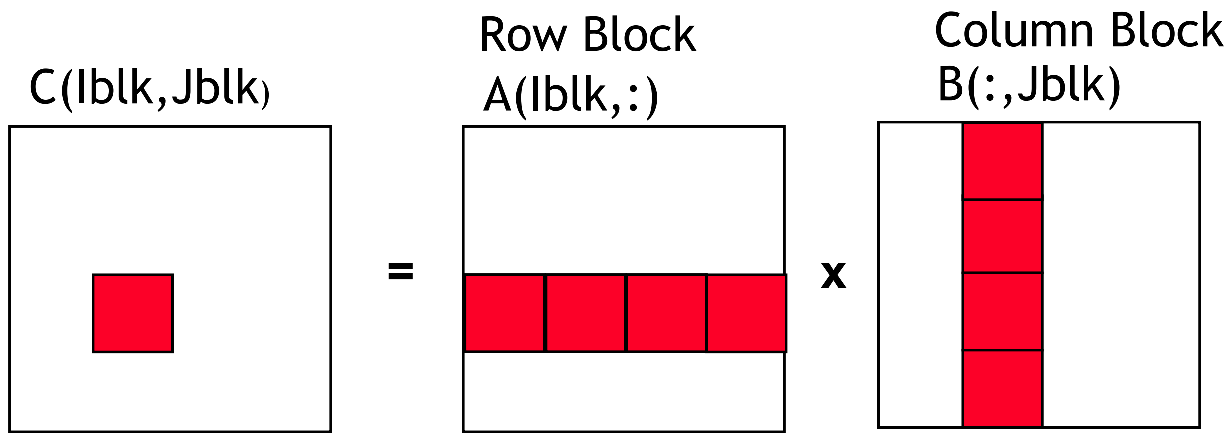
Understanding index offsets in the blocked matrix multiplication program.

16 x 16 NDRange with workgroups of size 4x4



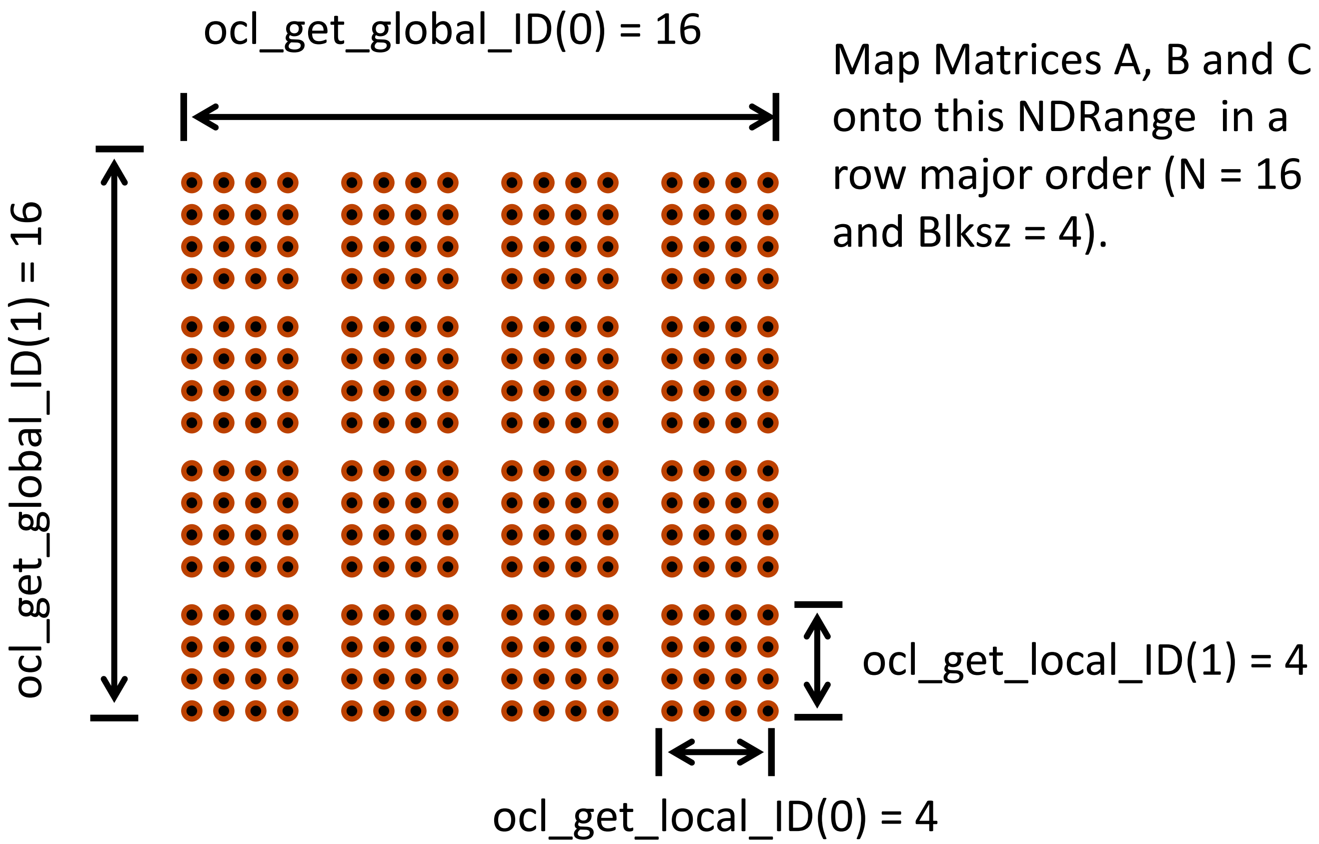
Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).

Mapping into A, B, and C from each work item



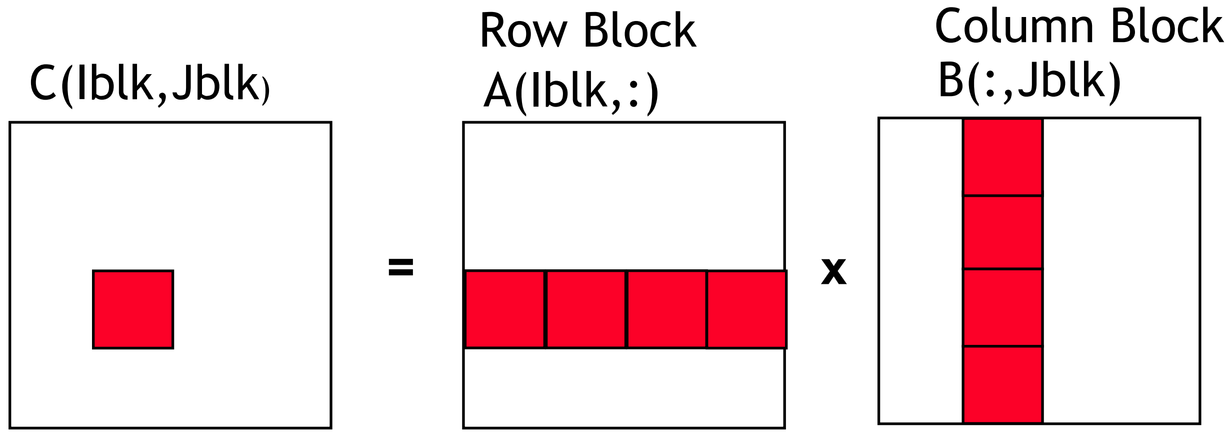
Understanding index offsets in the blocked matrix multiplication program.

16 x 16 NDRange with workgroups of size 4x4



Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).

Mapping into A, B, and C from each work item

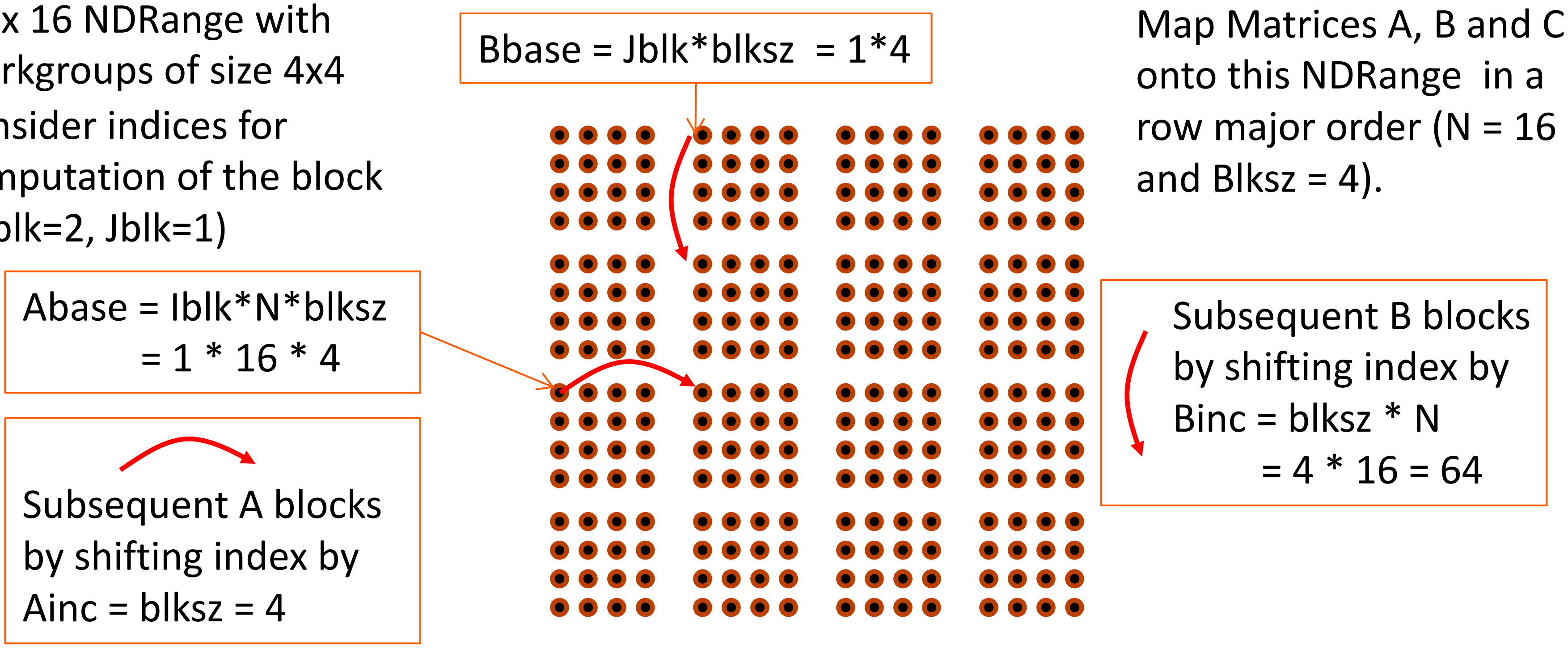


Understanding index offsets in the blocked matrix multiplication program.

16 x 16 NDRange with workgroups of size 4x4

Consider indices for computation of the block C(Iblk=2, Jblk=1)

Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).



Bbase = Jblk*blksz = 1*4

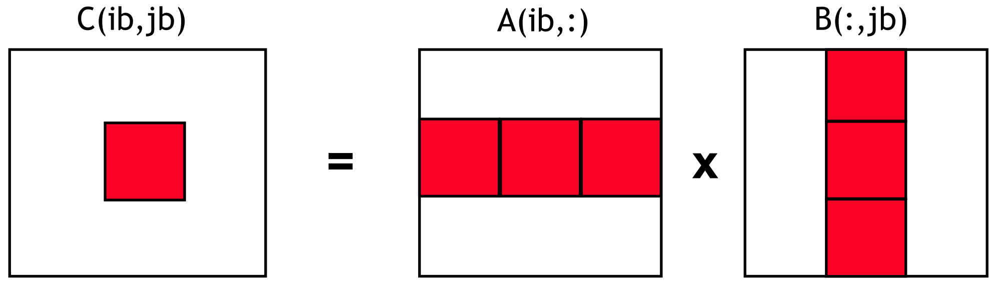
Abase = Iblk*N*blksz
= 1 * 16 * 4

Subsequent A blocks by shifting index by Ainc = blksz = 4

Subsequent B blocks by shifting index by Binc = blksz * N
= 4 * 16 = 64

Portable performance: dense matrix multiplication

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)
       sgemm(C, A, B, …)    // $C_{ib,jb} = A_{ib,kb} * B_{kb,jb}$

}
```



Transform the basic serial matrix multiply into multiplication over blocks

Note: sgemm is the name of the level three BLAS routine to multiply two matrices

Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
                const unsigned int N,
                __global float* A,
                __global float* B,
                __global float* C,
                __local  float* Awrk,
                __local  float* Bwrk)
{
   int kloc, Kblk;
   float Ctmp=0.0f;

   //  compute element C(i,j)
   int i = get_global_id(0);
   int j = get_global_id(1);

   // Element C(i,j) is in block C(Iblk,Jblk)
   int Iblk = get_group_id(0);
   int Jblk = get_group_id(1);

   // C(i,j) is element C(iloc, jloc)
   //  of block C(Iblk, Jblk)
   int iloc = get_local_id(0);
   int jloc = get_local_id(1);
   int Num_BLK = N/blksz;

  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;    int Ainc  = blksz;
  int Bbase = Jblk*blksz;      int Binc  = blksz*N;

 // C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
      //Each work-item loads a single element of the two
      //blocks which are shared with the entire work-group

      Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
      Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      #pragma unroll
      for(kloc=0; kloc<blksz; kloc++)
  Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      Abase += Ainc;    Bbase += Binc;
   }
   C[j*N+i] = Ctmp;
}
```

Blocked matrix multiply: kernel

It's getting the indices right that makes this hard

```
#define blksz 16
__kernel void mmul(
                const unsigned int N,
                __global float* A,
                __global float* B,
                __global float* C,
                __local  float* Awrk,
                __local  float* Bwrk)
{
   int kloc, Kblk;
   float Ctmp=0.0f;

   //  compute element C(i,j)
   int i = get_global_id(0);
   int j = get_global_id(1);

   // Element C(i,j) is in block C(Iblk,Jblk)
   int Iblk = get_group_id(0);
   int Jblk = get_group_id(1);

   // C(i,j) is element C(iloc, jloc)
   //  of block C(Iblk, Jblk)
   int iloc = get_local_id(0);
   int jloc = get_local_id(1);
   int Num_BLK = N/blksz;
```

Load A and B blocks, wait for all work-items to finish

```
 // upper-left-corner and inc for A and B
 int Abase = Iblk*N*blksz;   int Ainc  = blksz;
 int Bbase = Jblk*blksz;      int Binc  = blksz*N;

// C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
 for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
 {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
     //Each work-item loads a single element of the two
     //blocks which are shared with the entire work-group

     Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
     Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

     barrier(CLK_LOCAL_MEM_FENCE);

     #pragma unroll
     for(kloc=0; kloc<blksz; kloc++)
 Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

     barrier(CLK_LOCAL_MEM_FENCE);

     Abase += Ainc;    Bbase += Binc;
  }
  C[j*N+i] = Ctmp;
}
```

Wait for everyone to finish before going to next iteration of Kblk loop.

Matrix multiplication … Portable Performance (in MFLOPS)

- Single Precision matrix multiplication (order 1000 matrices)

Case	CPU	Xeon Phi	Core i7, HD Graphics	NVIDIA Tesla
Sequential C (compiled /O3)	224.4		1221.5	
C(i,j) per work-item, all global	841.5	13591		3721
C row per work-item, all global	869.1	4418		4196
C row per work-item, A row private	1038.4	24403		8584
C row per work-item, A private, B local	3984.2	5041		8182
Block oriented approach using local (blksz=16)	12271.3	74051 (126322*)	38348 (53687*)	119305
Block oriented approach using local (blksz=32)	16268.8			

Xeon Phi SE10P, CL_CONFIG_MIC_DEVICE_2MB_POOL_INIT_SIZE_MB = 4 MB

* The comp was run twice and only the second time is reported (hides cost of memory movement.

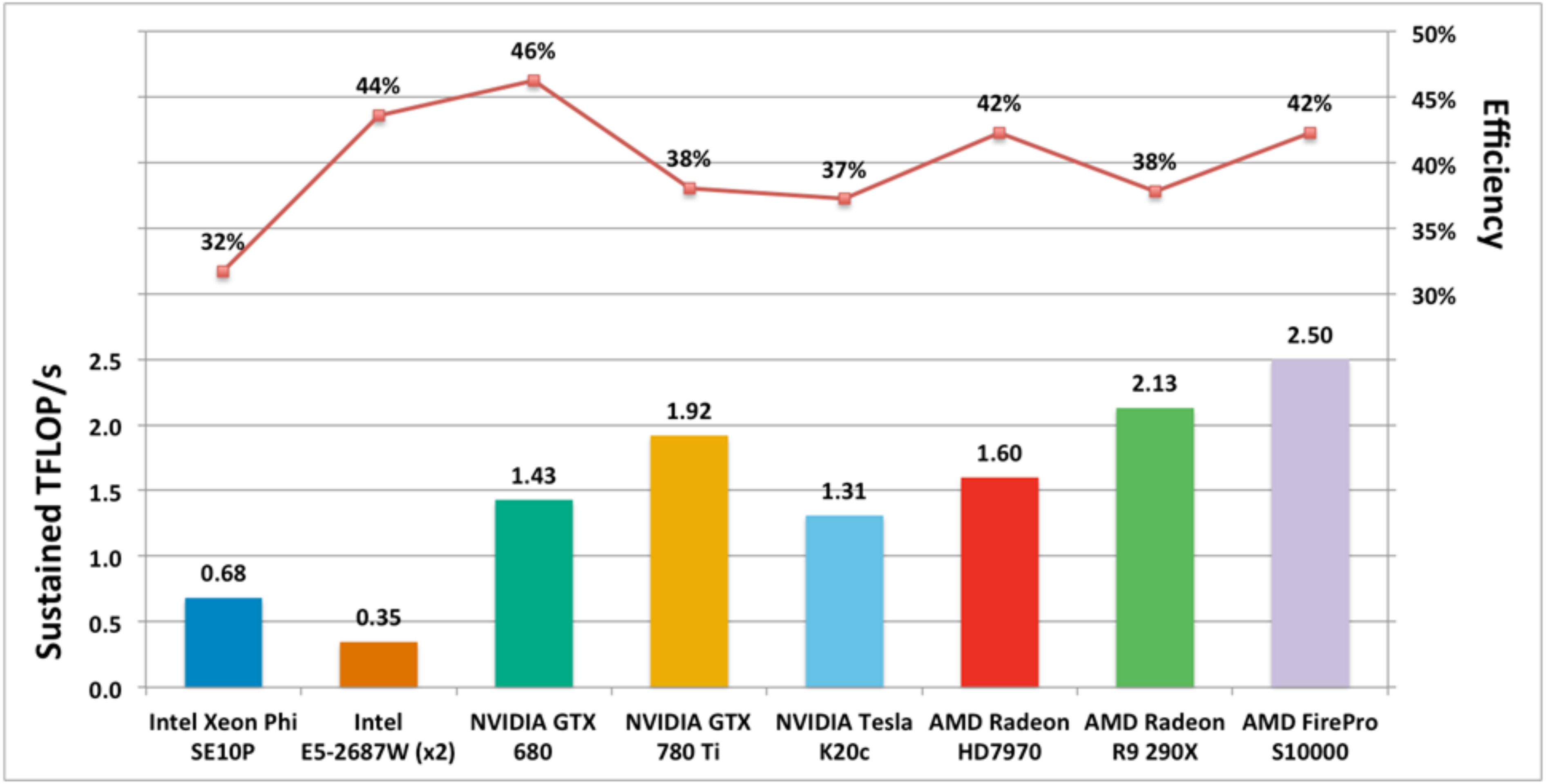
Intel® Core™ i5-2520M CPU @2.5 GHz (dual core) Windows 7 64 bit OS, Intel compiler 64 bit version 13.1.1.171, OpenCL SDK 2013, MKL 11.0 update 3.

Intel Core i7-4850HQ @ 2.3 GHz which has an Intel HD Graphics 5200 w/ high speed memory. ICC 2013 sp1 update 2.
Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Third party names are the property of their owners.

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

BUDE: Bristol University Docking Engine

One program running well on a wide range of platforms



Summary

- OpenCL is the only industry standard that spans CPU, GPU, DSP, and FPGA.
- OpenCL is a low level platform … extreme portability by exposing everything.
  - Strength of OpenCL: all features of a system are exposed so you can manipulate them as needed to get performance.
  - Weakness of OpenCL: all features of a system are exposed so you MUST manipulate them to get performance.
- Performance portability with OpenCL is possible … it is just as performance-portable as C or any other software platform we work with.
  - People who attack OpenCL’s performance portability often have an ulterior motive based on getting you to choose a programming model that locks you to their platform. Beware.

Summary

- OpenCL is the only industry standard that spans CPU, GPU, DSP, and FPGA.
- OpenCL is a low level platform … extreme portability by exposing everything.
  - Strength of OpenCL: all features of a system are exposed so you can man
  - Wea so you MUST manipulate them to get performance.

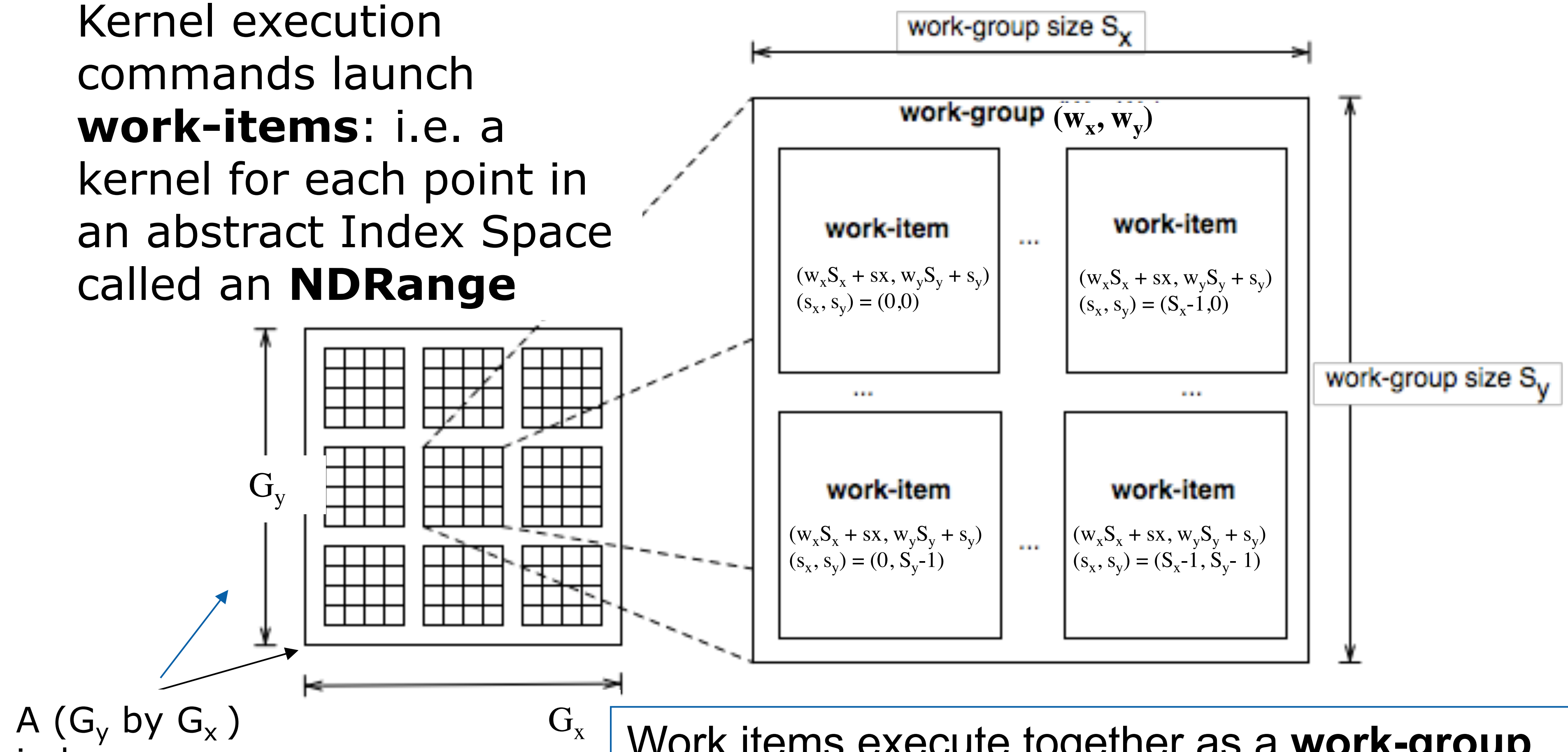
But wait … what about CUDA? You promised to cover CUDA as well

- Performance portability with OpenCL is possible … it is just as performance-portable as C or any other software platform we work with.
  - People who attack OpenCL's performance portability often have an ulterior motive based on getting you to choose a programming model that locks you to their platform. Beware.

Recall the OpenCL Execution Model

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue

Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract Index Space called an **NDRange**



work-group size S_x
work-group (w_x, w_y)
work-item
$(w_xS_x + sx, w_yS_y + s_y)$
$(s_x, s_y) = (0,0)$
work-item
$(w_xS_x + sx, w_yS_y + s_y)$
$(s_x, s_y) = (S_x-1,0)$
work-item
$(w_xS_x + sx, w_yS_y + s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_xS_x + sx, w_yS_y + s_y)$
$(s_x, s_y) = (S_x-1, S_y- 1)$
...
work-group size S_y
G_y
G_x

A (G_y by G_x) index space

Work items execute together as a **work-group**.

OpenCL vs. CUDA Terminology

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue

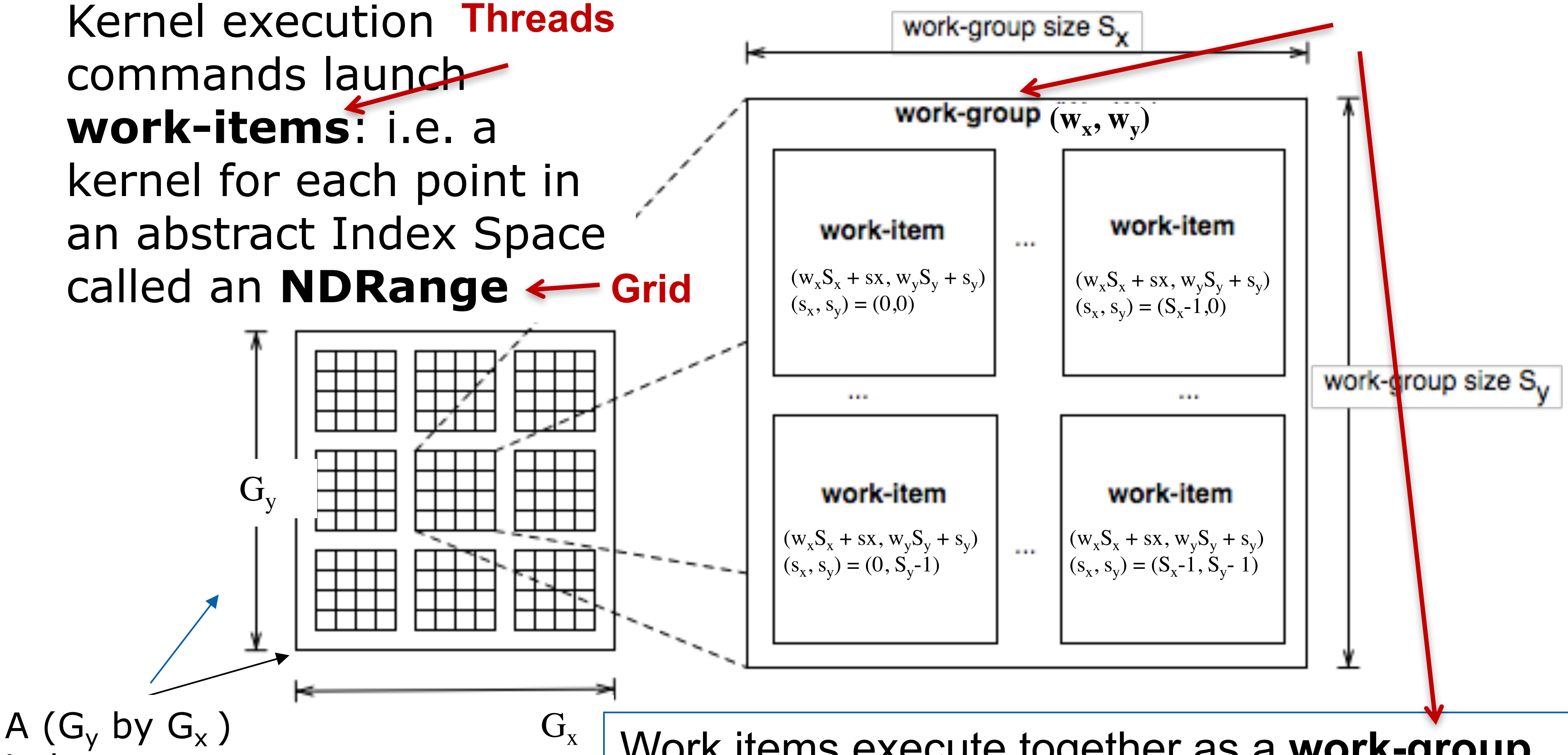
CUDA Stream

Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract Index Space called an **NDRange**

Threads

Grid

Thread Block



work-group size S_x
work-group (w_x, w_y)
work-item
$(w_xS_x + sx, w_yS_y + s_y)$
$(s_x, s_y) = (0,0)$
work-item
$(w_xS_x + sx, w_yS_y + s_y)$
$(s_x, s_y) = (S_x-1,0)$
work-item
$(w_xS_x + sx, w_yS_y + s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_xS_x + sx, w_yS_y + s_y)$
$(s_x, s_y) = (S_x-1, S_y- 1)$
work-group size S_y
G_y
G_x
A (G_y by G_x) index space

Work items execute together as a **work-group**.

Third party names are the property of their owners.

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 *d_a, *d_b, *d_c;
    cudaMalloc (&d_a,  sizeof(float) * N);
  // ... allocate other arrays, fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (d_a, d_b, d_c, N);
}
```

OpenCL vs. CUDA

- OK … I know both OpenCL and CUDA are rapidly evolving and for any given release their capabilities do not line up perfectly. There are things you can do with CUDA that you can't do with OpenCL and visa versa.
- My point is if you understand the fundamental approach of one and how it influences the algorithms you design, then you know the other.

Wrap-up

- We’ve covered a huge amount of material … probably too much.
- Just remember, the key are the design patterns. There are not very many of them.
  - Understand how your favorite patterns map onto the key programming models, and you’ll be able to make sense of parallel computing.
- We’ve covered the major programming models in HPC.
- Even if you plan to use “big data frameworks” such as Spark and Hadoop, you’ll have a better handle on how they work if you come at this from a perspective grounded in OpenMP, OpenCL/CUDA and MPI