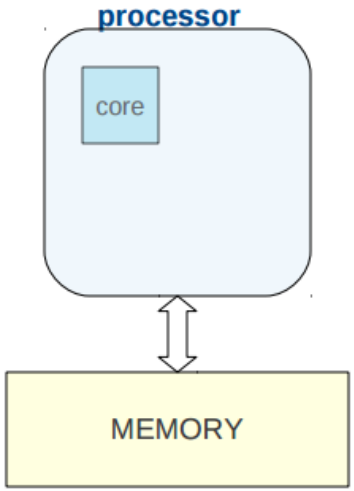
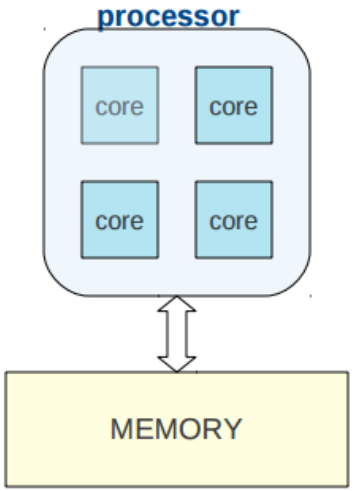
Introduction to Parallel Computing

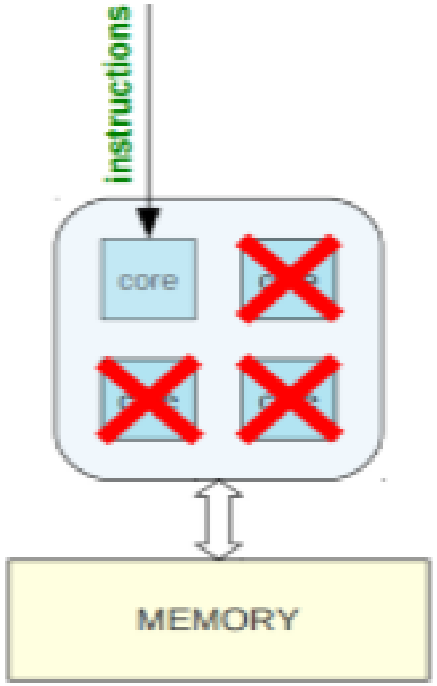
Sonia Gupta
HPC Team, TCS Pune

Why Parallel computing





In case of Sequential program: Waste of available resources. Only one core is utilized. Rest of cores are utilized. But we want all cores to be utilized. Hence parallel computing comes in to picture.



Parallel Computing

- Multi-core computing
- Pthreads
- OpenMP

- Distributed computing
- MPI

- Hetrogeneous computing
- GPGPU Programming

Threads vs. Processes

➔How threads and processes are similar

Each has its own logical control flow

Each can run concurrently

Each is context switched

➔How threads and processes are different

Threads share code and data, processes do not

Threads are somewhat less expensive than processes i.e number of machine cycles required to manage threads is lesser than process

●Process management is expensive as thread control

Traditional View of a Process

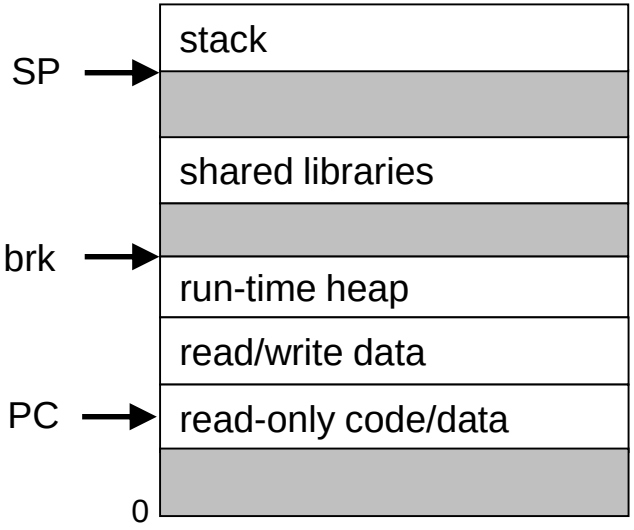
➔Process = process context + code, data, and stack

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

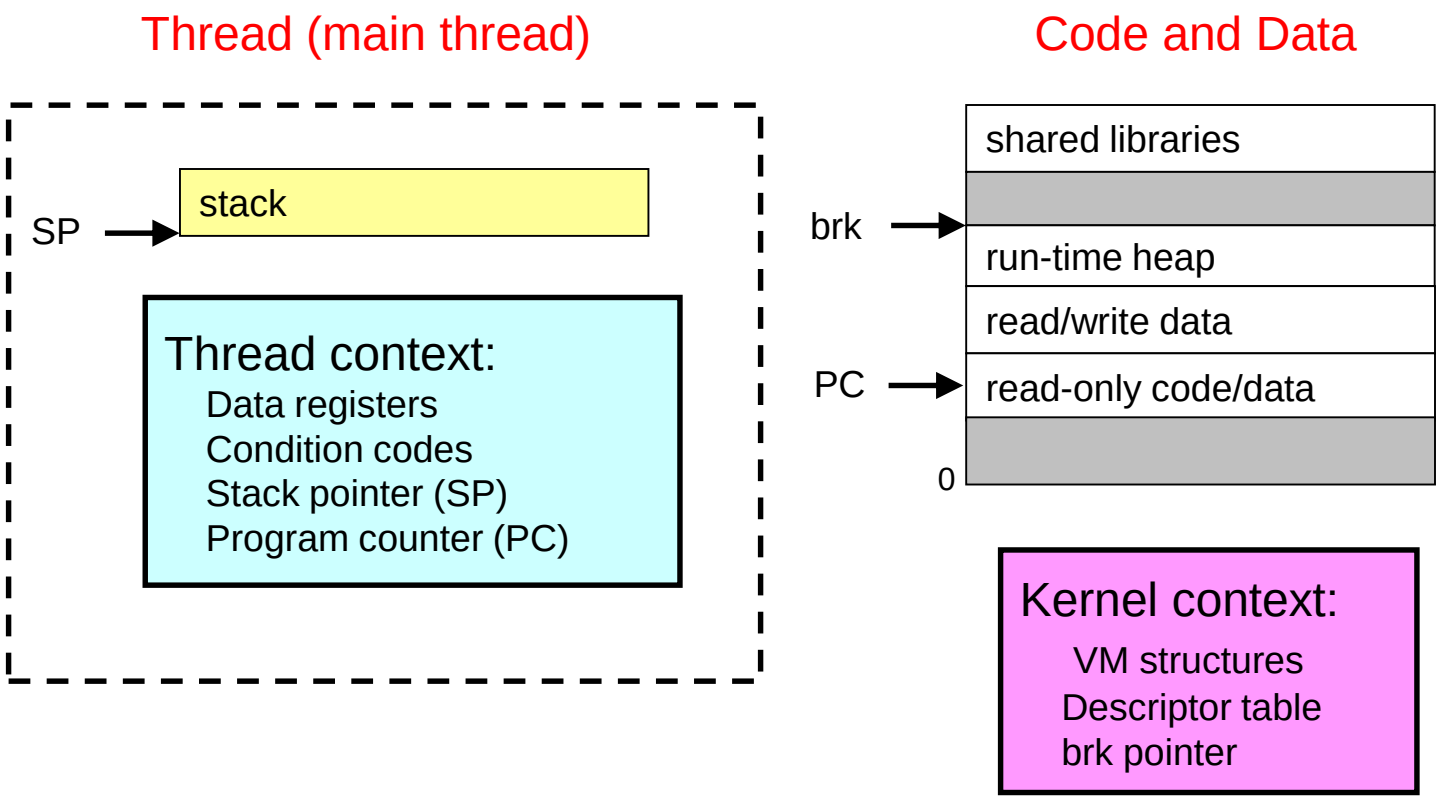
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



Alternate View of a Process

➔Process = thread + { code, data, and kernel context }



Parallel Programming using OpenMP

Sonia Gupta
HPC Team, TCS Pune

What is OpenMP?

➔API for writing shared memory applications in C,C++ and Fortran

➔OpenMP API consists of:

- Compiler Directives
- Runtime subroutines/functions
- Environment variables

Hello World Program

```
#include "omp.h"
void main()
{
#pragma omp parallel
 {
    int ID = omp_get_thread_num();
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
 }
}
```

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Sample Output:

hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

world(2)

Compiling:

Intel: icc –openmp test.c –o out.exe

Gnu: g++ -fopenmp test.c –o out.exe

Running:

EXPORT OMP_NUM_THREADS=4

./out.exe

Creating Threads

➔Threads are created using parallel construct

➔To create 4 threads, use

omp_set_num_threads(4)

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

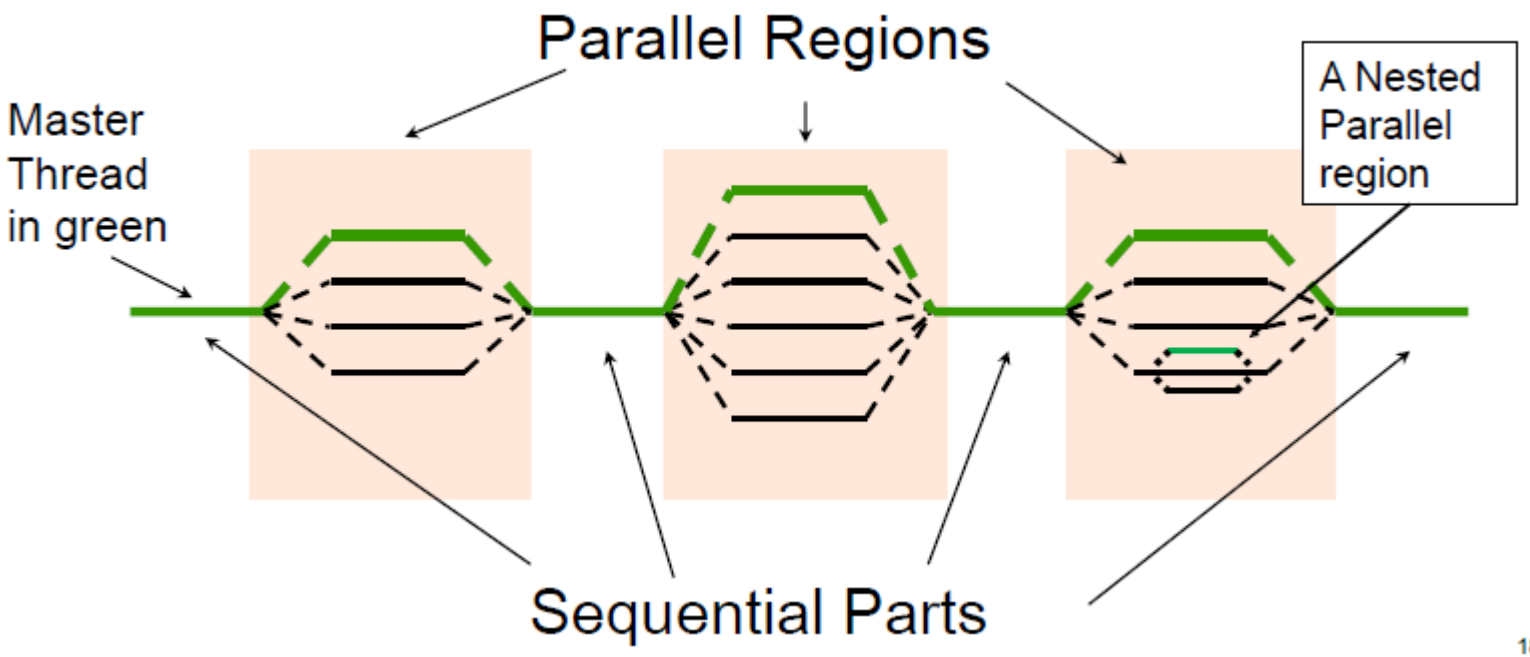
Runtime function returning a thread ID

- Each thread calls pooh(ID,A) for `ID` = `0` to `3`

OpenMP Execution

➔**Master thread** spawns a **team of threads** as needed.

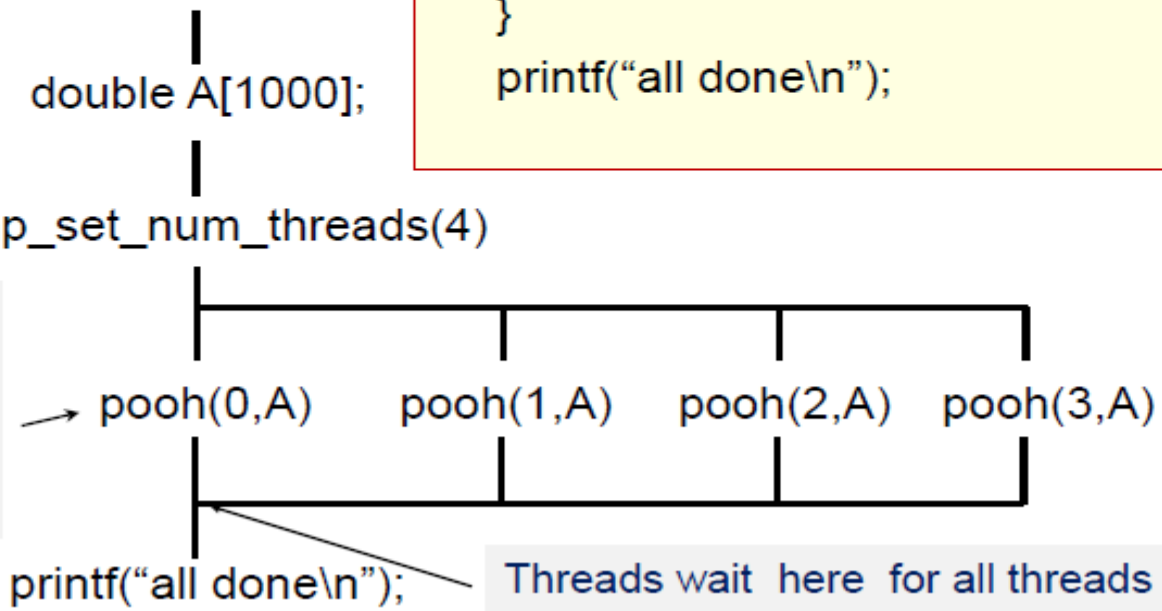
➔Parallelism through multiple threads



Creating threads

- Each thread executes the same code redundantly.

```
double A[1000];
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

More about OpenMP Threads

➔Number of openMP threads can be set using:

Environmental variable **OMP_NUM_THREADS**

Runtime function **omp_set_num_threads(n)**

Other useful function to get information about threads:

➔ Runtime function **omp_get_num_threads()**

Returns number of threads in parallel region

Returns 1 if called outside parallel region

- Runtime function **omp_get_thread_num()**

Returns id of thread in team

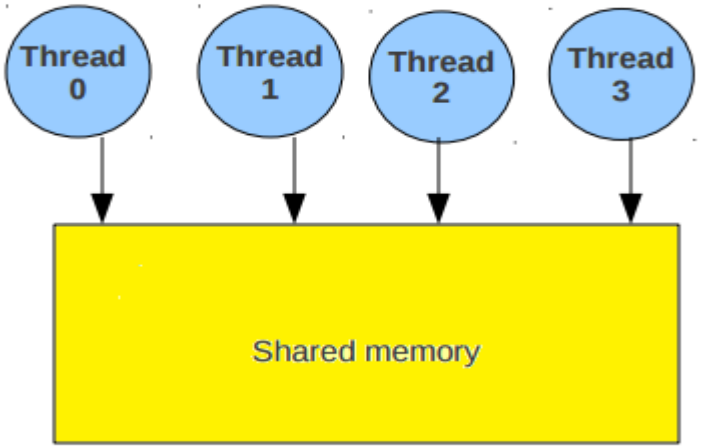
Value between [0,n-1] // where n = #threads

Master thread always has id 0

OpenMP Threads versus Cores

➔What are threads,cores and how are they related?

- ➔ Thread is independent sequence of execution of program code Block of code with one entry and one exit
- ➔ Unrelated to Cores/CPUs OpenMP threads are mapped onto physical cores
- ➔ Possible to map more than 1 thread on a core
- ➔ In practice best to have one-to-one mapping.
- ➔ Memory is shared by all the cores and the corresponding OMP threads



Shared and Private Variables

OpenMP provides a way to declare variables **private or shared** within an OpenMP block.
This is done using OpenMP clauses shared and private

SHARED (list) - **#pragma omp parallel shared(u,v,w)**

- All variables in list will be considered shared.
- Every openmp thread has access to all these variables

PRIVATE (list) - **#pragma omp parallel private(a,b,c)**

- Every openmp thread will have it's own ”private” copy of variables in list
- No other openmp thread has access to this ”private” copy

Work sharing

Objective: we want is to share work among all threads so we can solve our problems faster

```
#pragma omp parallel
private(n,num,id,f,l)
Id = omp_get_thread_num()
Num = omp_get_num_threads()
f = id*(N/num)+1
l = (id+1)*(N/num)
for(int n=f;n<l;n++)
        {   A(n) = A(n) + B;    }
```

Partition the iteration space manually, every thread computes N/num iterations

Suppose: N=100 and num=4 → N/num=25

```
f = id*(N/num)+1
l = (id+1)*(N/num)
DO n=f,l,1
   A(n) = A(n) + B
ENDDO
```

Thread 0	Thread 1	Thread 2	Thread 3
f=0*25+1 = 1 l=1*25 = 25	f=1*25+1=26 l=2*25 = 50	f=2*25+1=51 l=3*25 = 75	f=3*25+1=76 l=4*25 = 100

Thread 0 computes elements from index 1 to 25, Thread 1 computes from index 26 to 50, etc.

Work Sharing: Parallel For

➔Code given inside a **Parallel For** is executed in parallel

```
 double  res[MAX];  int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

```
 double  res[MAX];  int i;
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
```

These are equivalent

Variable i is private to each
The for loop range is divided equally among all the threads

Dynamic Scheduling

Dynamic Scheduling: After each iteration, the threads must stop and receive a new value of the loop variable to use for its next iteration.
#pragma omp parallel for schedule(dynamic) num_threads(THREADS)

Dynamic with chunk size: each thread will take a set number of iterations, called a “chunk”, execute it, and then be assigned a new chunk when it is done.
#pragma omp parallel for schedule(dynamic, CHUNK) num_threads(THREADS)

Dunamic with guided: This scheduling policy is similar to a dynamic schedule, except that the chunk size changes as the program runs. It begins with big chunks, but then adjusts to smaller chunk sizes if the workload is imbalanced.
#pragma omp parallel for schedule(guided) num_threads(THREADS)

Synchronization

➔Consider a simple example where two threads on two different processors are both trying to increment a variable x at the same time (assume x is initially 0):

THREAD 1:	THREAD 2:

`increment(x)`
`{`
`  x = x + 1;`
`}`
	`increment(x)`
	`{`
	`  x = x + 1;`
	`}`

THREAD 1:	**THREAD 2:**

`10  LOAD A, (x address)`
`20  ADD A, 1`
`30  STORE A, (x address)`
	`10  LOAD A, (x address)`
	`20  ADD A, 1`
	`30  STORE A, (x address)`

1.One possible execution sequence:Thread 1 loads the value of x into register A.
2.Thread 2 loads the value of x into register A.
3.Thread 1 adds 1 to register A
4.Thread 2 adds 1 to register A
5.Thread 1 stores register A at location x
6.Thread 2 stores register A at location x

•The resultant value of x will be 1, not 2 as it should be.
•To avoid a situation like this, the incrementing of x must be synchronized between the two threads to ensure that the correct result is produced.
•OpenMP provides a variety of Synchronization Constructs that control how the execution of each thread proceeds relative to other team threads.

MASTER directive

Purpose

➔The MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code

➔There is no implied barrier associated with this directive

Format

```
#pragma omp master
    {
        some instructions;
     }
```

Restrictions

It is illegal to branch into or out of the MASTER block

CRITICAL example

```
#include <omp.h>
main()
{
        int x;
        X = 0;
        #pragma omp parallel shared(x)
        {
                #pragma omp critical
                 x=x+1
         }/*end of parallel section*/
}
```

BARRIER directive

Purpose

➔The BARRIER directive synchronises all threads in a team

➔ When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier

Format

#pragma omp barrier

Restrictions

All threads in the team must execute the BARRIER

Runtime Environment routines

Modify/Check the number of threads

omp_set_num_threads(),
omp_get_num_threads(),
omp_get_thread_num(), omp_get_max_threads()

Are we in an active parallel region?

omp_in_parallel()

Do you want the system to dynamically vary the number of threads from one parallel construct to another?

omp_set_dynamic, omp_get_dynamic();

How many processors in the system?

omp_get_num_procs()

Parallel Programming using MPI

Sonia Gupta
HPC Team, TCS Pune

Sequence

- ➔Message passing
- ➔Synchronous/Asynchronous message transfer
- ➔What is MPI?
- ➔“Hello World” Program
- ➔Blocking Send/Recv
- ➔Non-blocking Send/Recv
- ➔Deadlocks
- ➔Collectives
  - Bcast
  - Reduce
  - Scatter/Gather
  - Allgather
- ➔PI Example
- ➔MPI-2/MPI-3
- ➔References

What is message passing?

➔ Message Passing Interface (Data transfer)
➔Requires involvement of sender and receiver

➔Data transfer can be synchronous and asynchronous:

◆A synchronous communication is not complete until the message has been received.

◆An asynchronous communication completes as soon as the message is on the way.

What is MPI?

➔A message-passing library specifications:

Not a language or compiler specification

Not a specific implementation or product

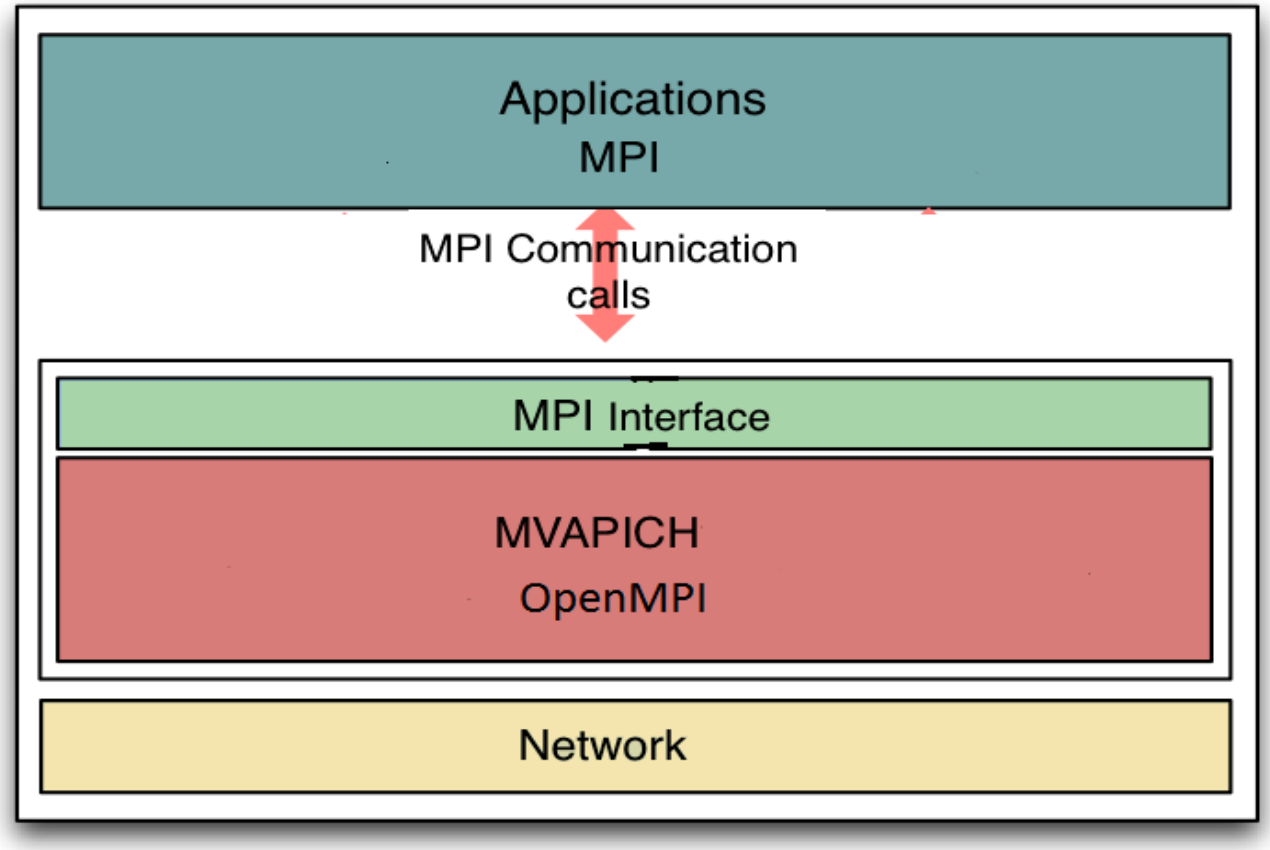
➔For parallel computers, clusters, and heterogeneous networks.

➔Communication modes: *standard*, *asynchronous*

➔Designed to permit the development of parallel software libraries.

➔Two MPI libraries are there. 1) IntelMPI. 2) OpenMPI

MPI on System



A Minimal MPI Program (C)

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
   MPI_Init( &argc, &argv );
   printf( "Hello, world!\n" );
   MPI_Finalize();
   return 0;
}
```

Compiling:

> mpicc hello.c

To create 10 instances

>mpirun –np 10 ./a.out

To give nodes:

mpirun –np 5 –host node1,node1,node2,node3,node3 ./a.out

Information About Environment

- Total number of processes participating gives the size.

  So what is the size?

  MPI_Comm_size() reports the number of processes.

- Every process is identified by a rank.

  What is my rank?

  MPI_Comm_rank() reports the *rank*, a number between 0 and size-1, identifying the calling process

Better Hello (C)

```
#include "mpi.h"
#include <stdio.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( "I am %d of %d\n", rank, size );
   MPI_Finalize();
   return 0;
}
```

Default communicator

Processes ID in communicator

Processes in a communicator

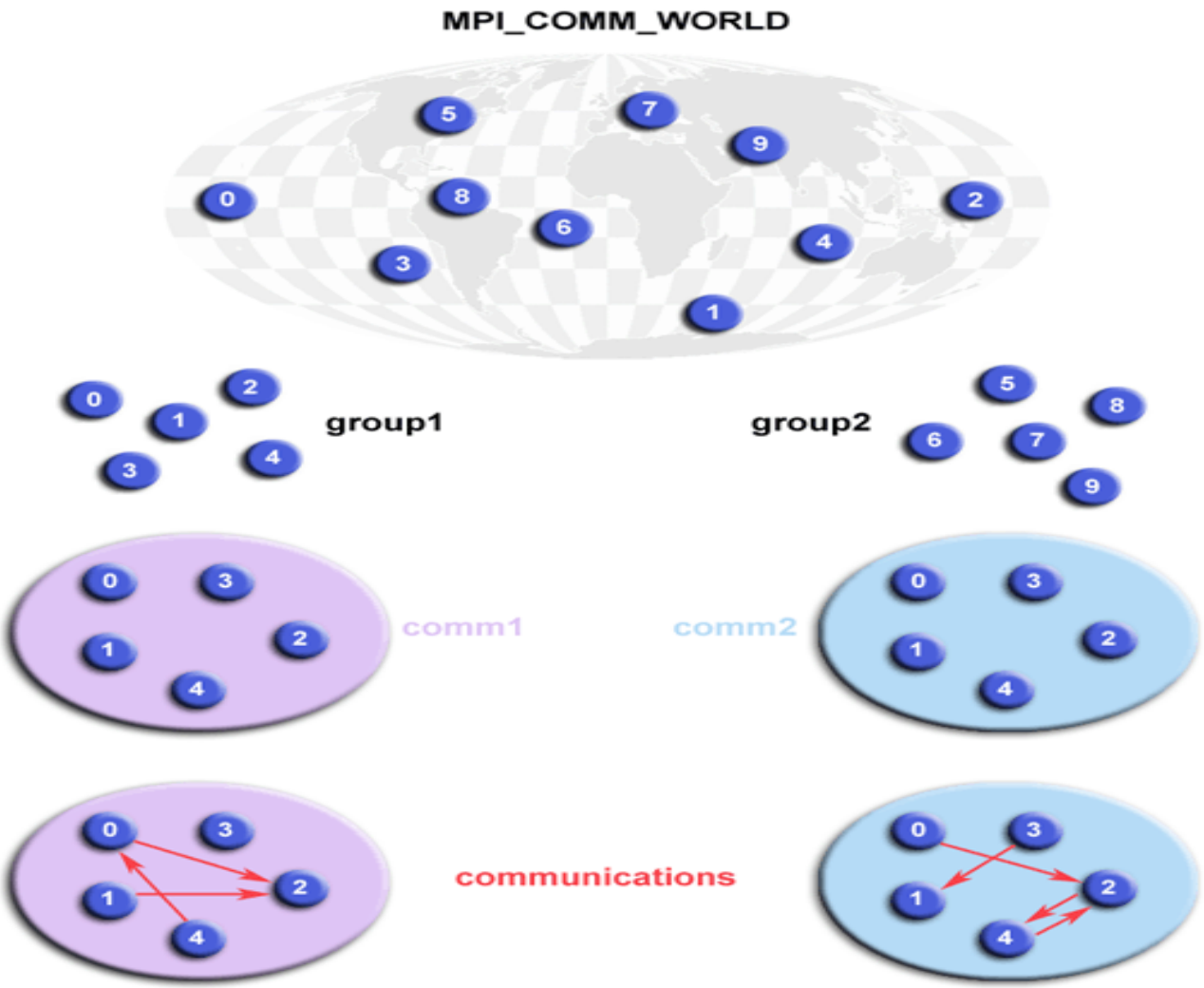
mpirun –np 2 ./a.out

Output:

I am 0 of 2

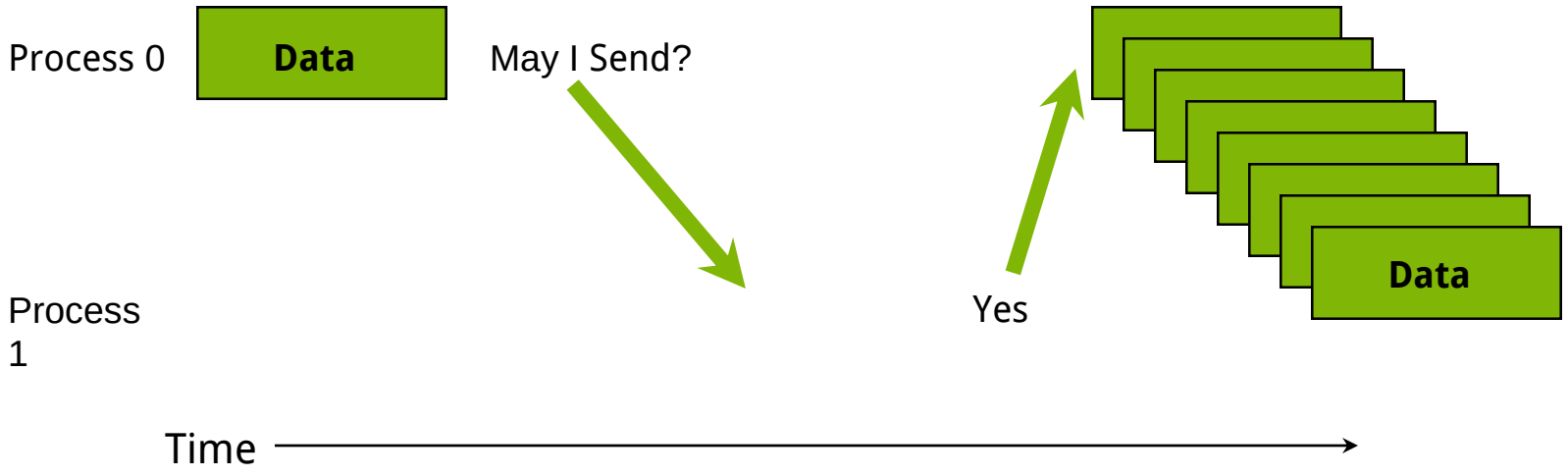
I am 1 of 2

Basic Concepts



What is message passing?

➔Data transfer plus synchronization



MPI blocking send

MPI_SEND(void *start, int count,MPI_DATATYPE datatype, int dest, int tag, MPI_COMM comm)

➔The message buffer is described by (start, count, datatype).
➔dest is the rank of the target process in the defined communicator.
➔tag is the message identification number.

MPI blocking receive

MPI_RECV(void *start, int count, MPI_DATATYPE datatype, int source, int tag, MPI_COMM comm, MPI_STATUS *status)

➔Source is the rank of the sender in the communicator.

➔The receiver can specify a wildcard value for source (MPI_ANY_SOURCE) and/or a wildcard value for tag (MPI_ANY_TAG), indicating that any source and/or tag are acceptable

➔Status is used for extra information about the received message if a wildcard receive mode is used.

➔If the count of the message received is less than or equal to that described by the MPI receive command, then the message is successfully received. Else it is considered as a buffer overflow error.

Datatypes

➔Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very different memory representations and lengths of elementary datatypes (heterogeneous communication).

➔Specifying application-oriented layout of data in memory

- reduces memory-to-memory copies in the implementation
- allows the use of special hardware (scatter/gather) when available

Basic MPI types

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	signed short
MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	signed int
MPI_UNSIGNED	unsigned int
MPI_LONG	signed long
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Tags

- ➔ Aim is to use it for Separation of messages.
- ➔ Arbitrary non-negative integer assigned by the programmer to uniquely identify a message.
- ➔ Send and receive operations should match message tags. For a receive operation, the wild card MPI_ANY_TAG can be used to receive any message regardless of its tag.
- ➔ The MPI standard guarantees that integers 0-32767 can be used as tags, but most implementations allow a much larger range than this.

MPI_STATUS

➔Status is a data structure

➔In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(…, MPI_ANY_SOURCE, MPI_ANY_TAG, …, &status)
recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count(&status, datatype, &recvd_count);
```

MPI is Simple

➔Many parallel programs can be written using just these six functions

- **MPI_INIT**
- **MPI_FINALIZE**
- **MPI_COMM_SIZE**
- **MPI_COMM_RANK**
- **MPI_SEND**
- **MPI_RECV**

Non-Blocking Send and Receive

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
MPI_IRECV(buf, count, datatype, dest, tag, comm, request)

➔request is a request handle which can be used to query the status of the communication or wait for its completion.

MPI_WAIT (request, status)
MPI_TEST (request, flag, status)

The MPI_WAIT will block your program until the non-blocking send/receive with the desired request is done.

The MPI_TEST is simply queried to see if the communication has completed and the result of the query (TRUE or FALSE) is returned immediately in flag

Non-Blocking Send and Receive (Cont.)

➔A non-blocking send call indicates that the system may start copying data out of the send buffer. The sender must not access any part of the send buffer after a non-blocking send operation is posted, until the complete-send returns.

➔A non-blocking receive indicates that the system may start writing data into the receive buffer. The receiver must not access any part of the receive buffer after a non-blocking receive operation is posted, until the complete-receive returns.

MPI_Isend/MPI_Irecv

Rank = 0

```
MPI_Isend(..,request1)
MPI_IRecv(..,request2)

….. Do something

/*wait for completion ..blocking*/
MPI_Wait(…,&request1);
MPI_Wait(…,&request2);

/*check whether the request was
successfully completed*/
MPI_Test(request1,..,status1);
MPI_Test(request2,..,status2);

If(status1 == MPI_SUCCESS)
        printf("message 1 successful);

If(status2 == MPI_SUCCESS)
        printf("message 2 successful);
```

Rank = 1

```
MPI_IRecv(..,request1)
MPI_Isend(…,request2)

…..Do something

/*wait for completion ..blocking*/
MPI_Wait(..,&request1);
MPI_Wait(..,&request2);

/*check whether the request was
successfully completed*/
MPI_Test(request1,..,status1)
MPI_Test(request2,..,status2)

If(status1 == MPI_SUCCESS)
        printf("message 1 successful);

If(status2 == MPI_SUCCESS)
        printf("message 2 successful);
```

Deadlocks in blocking operations

➔What happens with

Process 0	Process 1
Send(1)	Send(0)
Recv(1)	Recv(0)

➔Send a large message from process 0 to process 1

- If there is insufficient storage at the destination, the send must wait for the user to provide the memory space(through a receive)

➔This is called “”unsafe” because it depends on the availability of system buffers.

Some solutions to the “”unsafe” problem

➔Order the operations more carefully

Process 0	Process 1
Send(1)	Recv(0)
Recv(1)	Send(0)

➔ Use non-blocking operations:

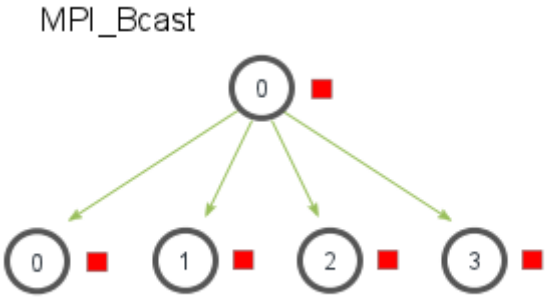
Process 0	Process 1
ISend(1)	ISend(0)
IRecv(1)	IRecv(0)
Waitall	Waitall

Introduction to Collectives

➔Collective operations are called by all processes in a communicator.
➔**MPI_BCAST** distributes data from one process (the root) to all other
➔**MPI_REDUCE** combines data from all processes in communicator and returns it to one process.
➔In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency.

MPI_BCAST

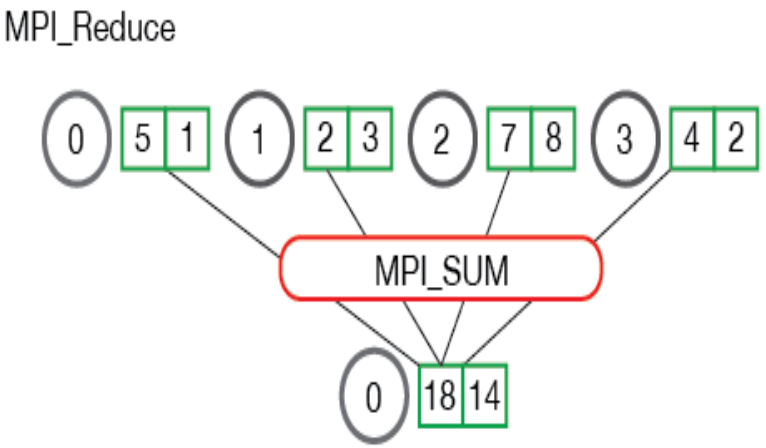
MPI_BCAST distributes data from one process (the root) to all others in a communicator.




```
main()
{
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
count=4;
Root = 1
if(myid == Root){
     for(i=0;i<count;i++)
       buffer[i]=i;
   }
MPI_Bcast(buffer,count,MPI_INT,source,MPI_COMM_WORLD);
}
```

MPI_Reduce

→**MPI_REDUCE** combines data from all processes in communicator and returns it to one process.

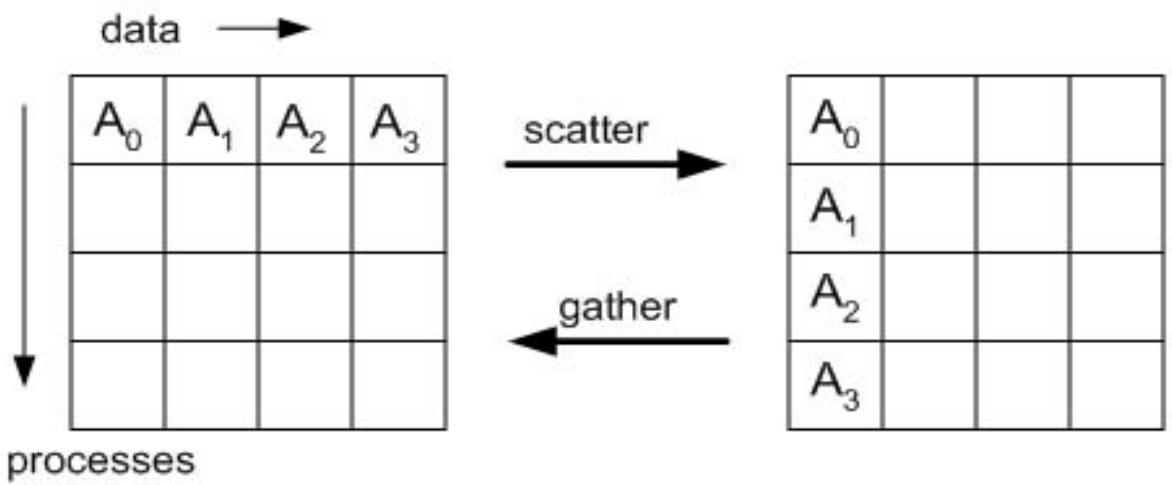


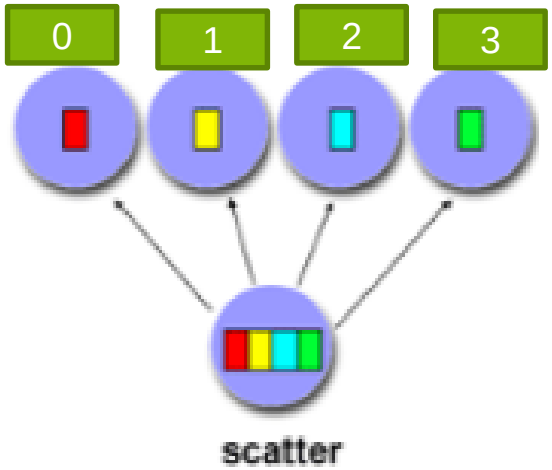

```
main()
{
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

int val = 10;
int mpi_root = 0;
int total;
MPI_Reduce(&val, &total, 1,  MPI_INT, MPI_SUM, mpi_root, MPI_COMM_WORLD);
}
```

MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD,
MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR,
MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC

MPI_Scatter/MPI_Gather

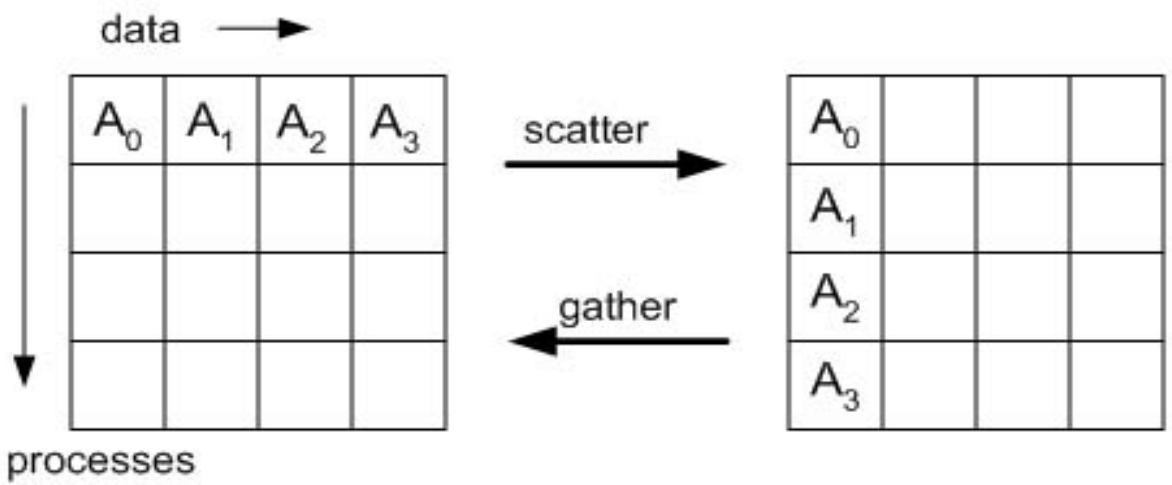


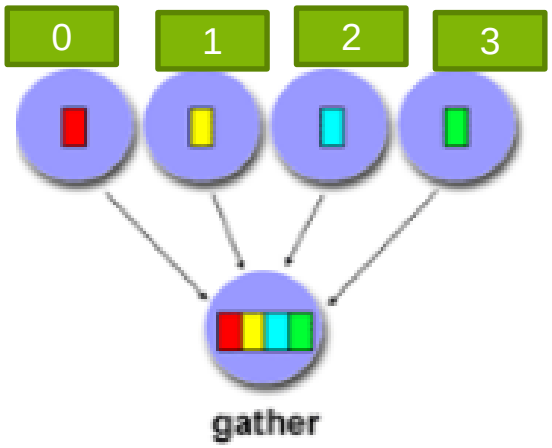


MPI_Scatter(void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype recv_datatype, int root, MPI_Comm communicator)

MPI_Gather(void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype recv_datatype, int root, MPI_Comm communicator)

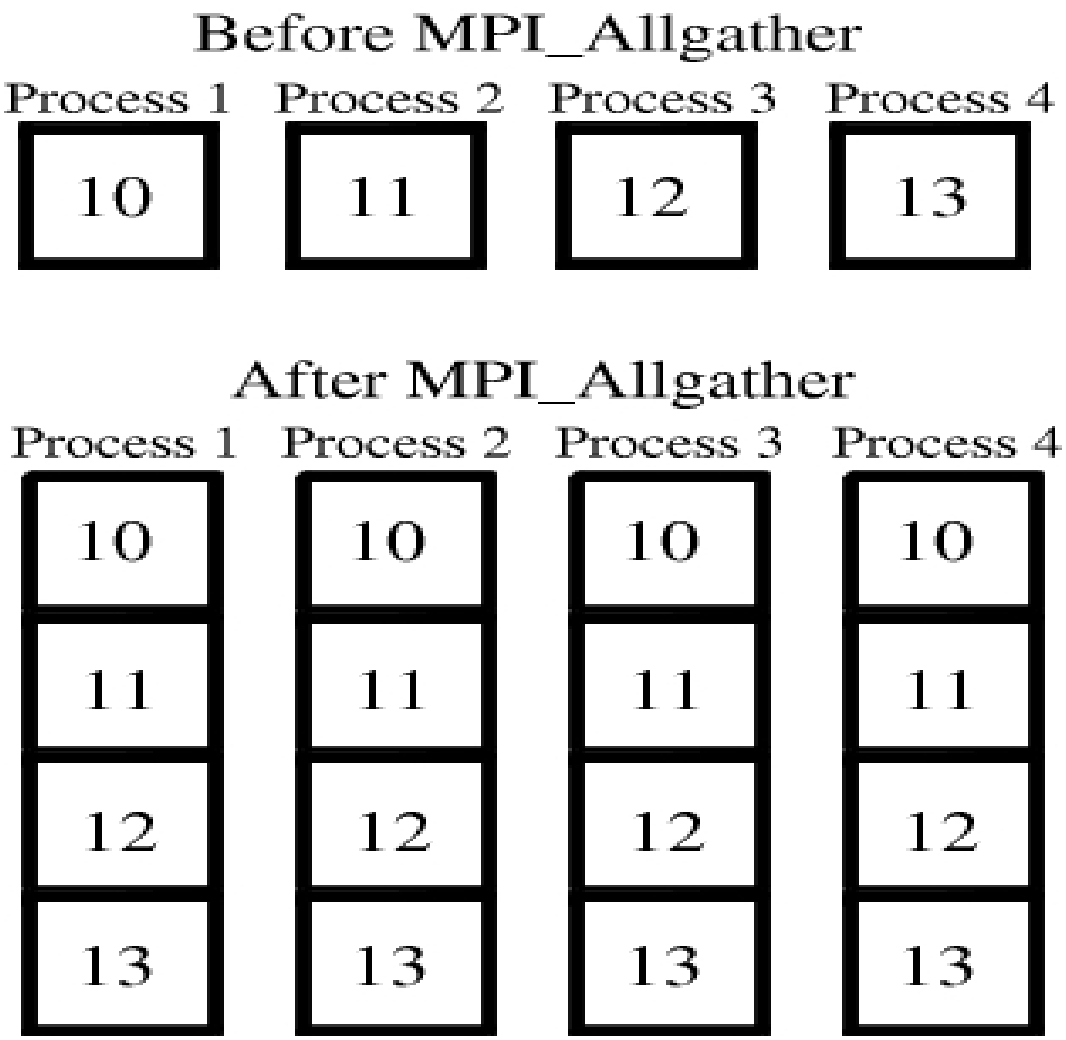
MPI_Scatter/MPI_Gather





MPI_Gather(void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype recv_datatype, int root, MPI_Comm communicator)

MPI_Allgather



GPGPU Programming

Sonia Gupta
HPC Team, TCS Pune

Introduction - CUDA

General Purpose computation using GPU (GPGPU) in applications other than 3D graphics

- GPU accelerates critical path of application
- CUDA, OpenCL are being used to program on GPU.

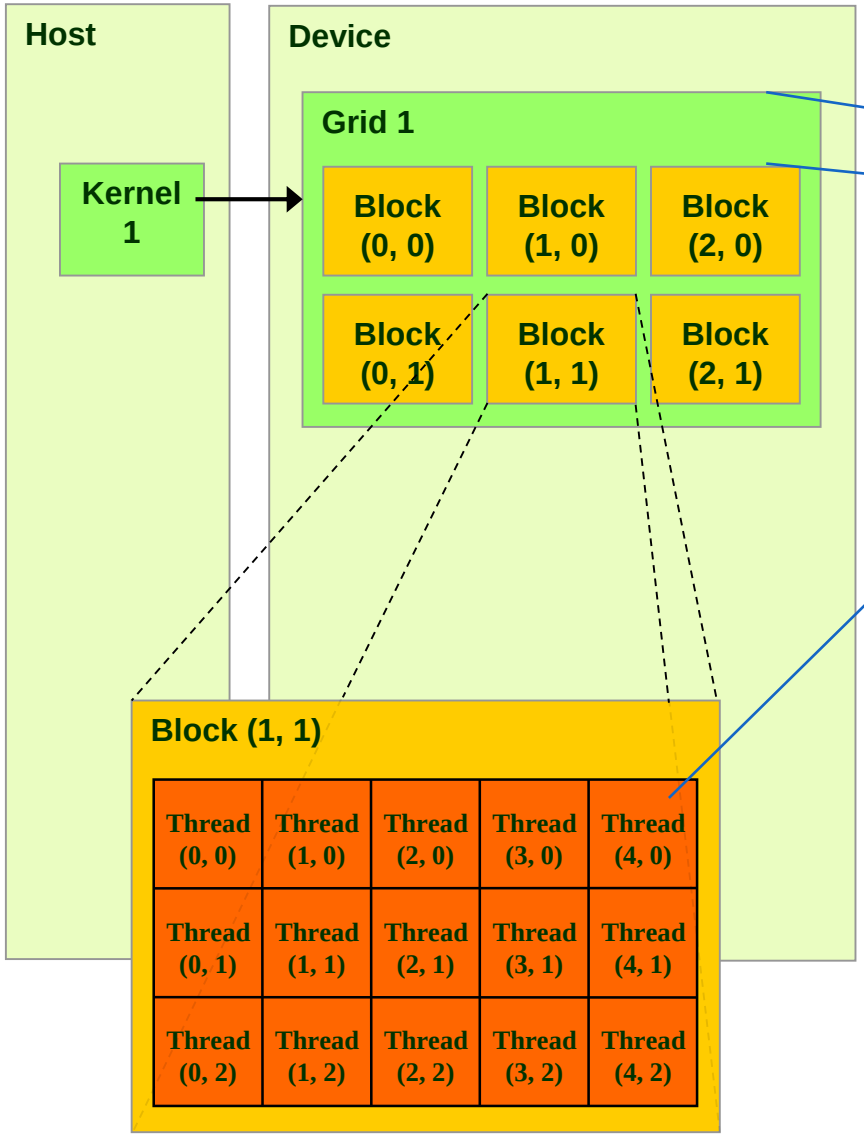
"Compute Unified Device Architecture" (CUDA) - General purpose programming model

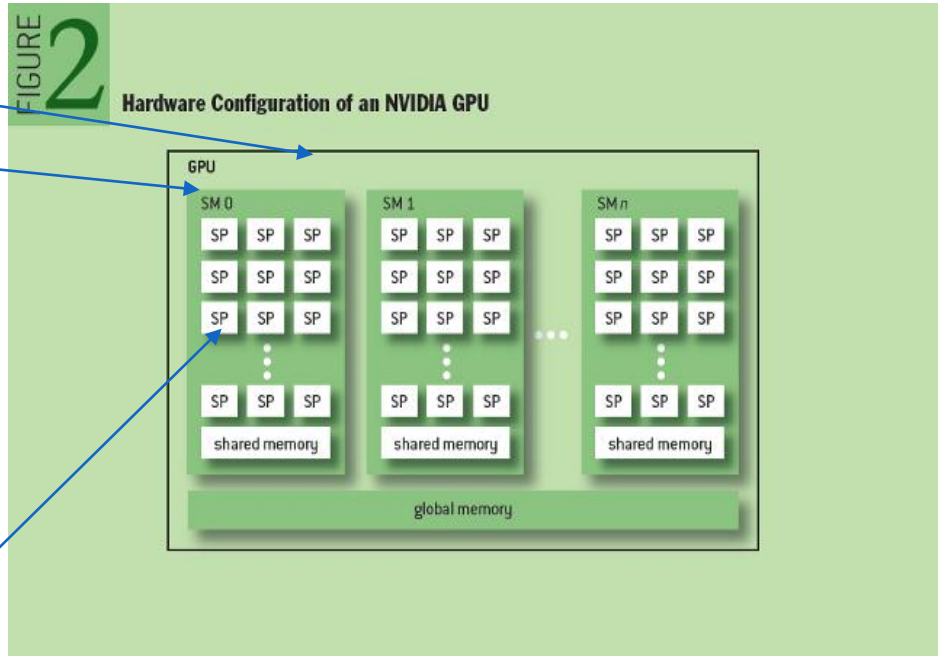
CUDA was released on February 15, 2007 for PC and Beta version for MacOS X on August 19, 2008.

Why CUDA:

- CUDA provides ability to use high-level languages such as C to develop application that can take advantage of high level of performance and scalability that GPUs architecture offer.
- Small set of extensions to enable heterogeneous programming
- Straightforward APIs to manage devices, memory etc.

CUDA - Environment





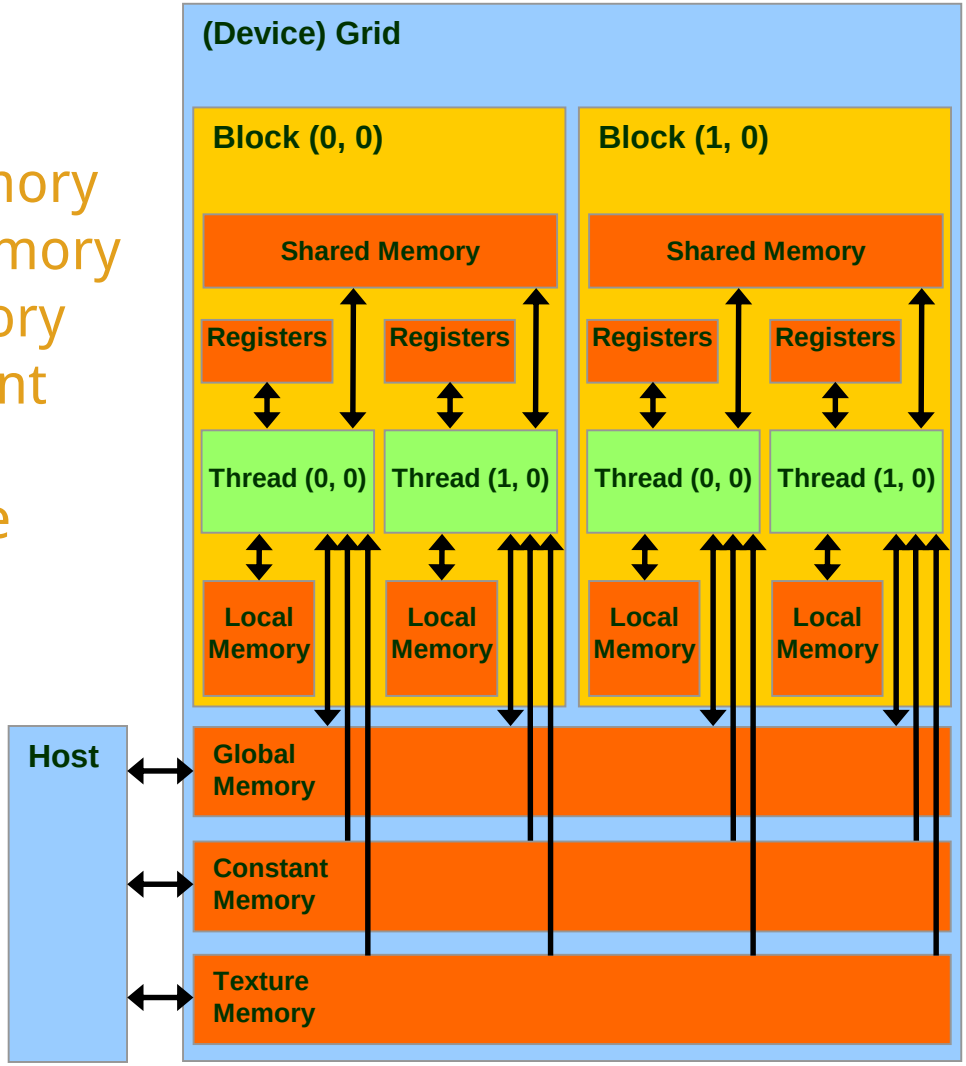
- ➔ A kernel is executed as a grid of thread blocks
- ➔ A thread block is a batch of threads that can cooperate with each other by:
  - ◆ Synchronizing their execution
  - ◆ Efficiently sharing data through a low latency shared memory
- ➔ Two threads from two different blocks cannot cooperate
- ➔ Threads will execute inside a SM in group of 32 thread called warp.

CUDA - Memory

➔ Each thread can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory
- Read only per-grid texture memory

- The host can R/W global, constant, and texture memories



CUDA MEMORY UNITS

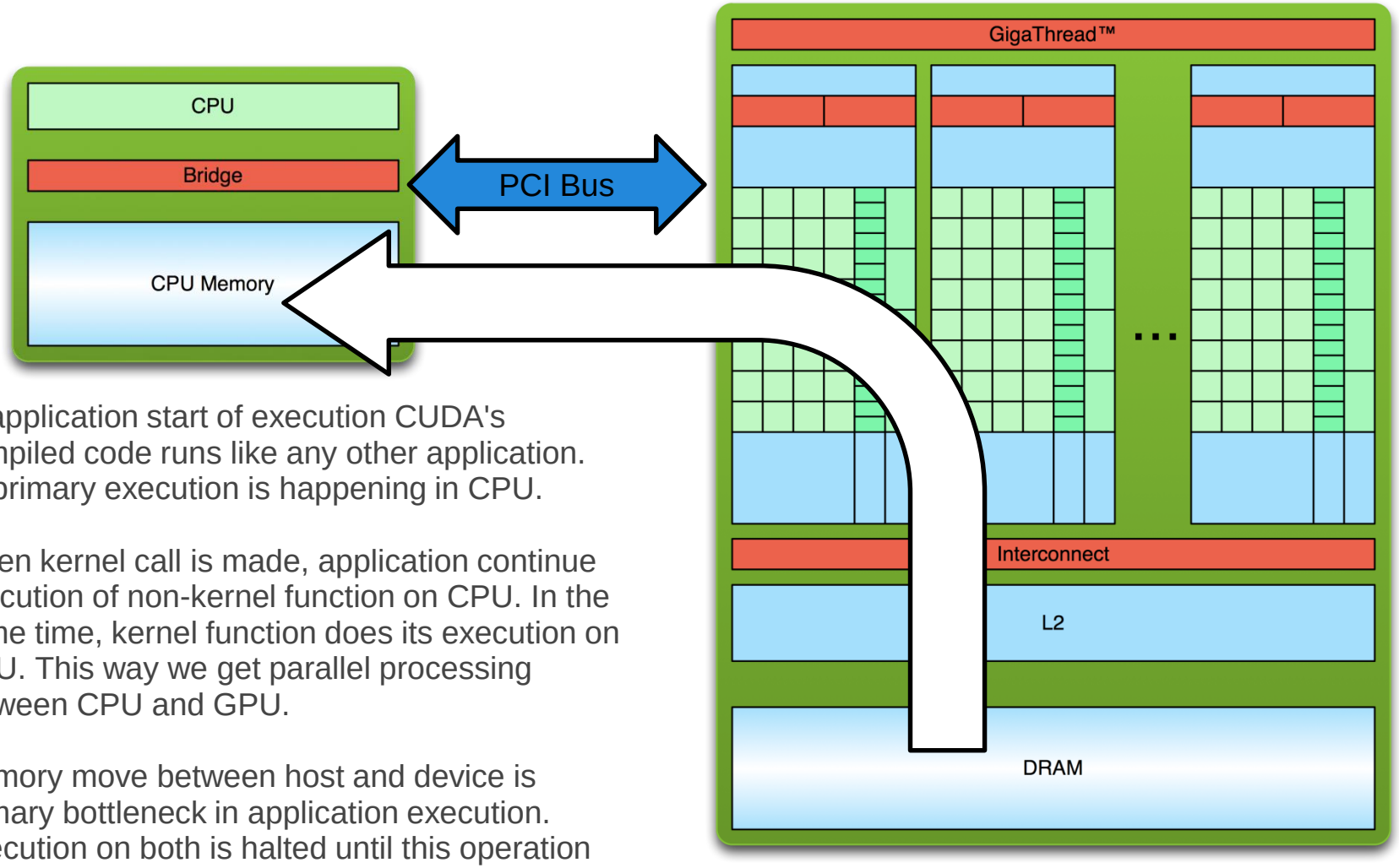
- Registers:
  - Fastest.
  - Only accessible by a thread.
  - Lifetime of a thread

- Shared memory:
  - Could be as fast as registers if no bank conflicts or reading from same address.
  - Accessible by any threads within a block where it was created.
  - Lifetime of a block.

- Global Memory:
  - Up to 150x slower then registers or share memory.
  - Accessible from either host or device.
  - Lifetime of an application.

- Local Memory
  - Resides in global memory. Can be 150x slower then registers and shared memory.
  - Accessible only by a thread.
  - Lifetime of a thread.

CUDA - Basics



At application start of execution CUDA's compiled code runs like any other application. Its primary execution is happening in CPU.

When kernel call is made, application continue execution of non-kernel function on CPU. In the same time, kernel function does its execution on GPU. This way we get parallel processing between CPU and GPU.

Memory move between host and device is primary bottleneck in application execution. Execution on both is halted until this operation completes.

CUDA - Example

```
__global__ void mykernel(void){}

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

Triple angle brackets mark a call from *host* code to *device* code

Compilation:

```
$ nvcc hello.cu
Execution
$ a.out
Hello World!
$
```

nvcc separates source code into host and device components

`__global__` defines a kernel function and Must return **`void`**

`__device__` (device function) and `__host__` (Host function) can be used together

CUDA - memcpy

Allocate Memory on Device

cudaMalloc() – allocate memory in Global memory

cudaFree()

```
cudaMalloc((void**)&Md.elements, size);
cudaFree(Md.elements);
```

Transfer memory from Device to Host and vice versa

cudaMemcpy()

```
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);

cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToHost);
```

CUDA – Kernel Launch

➔ A kernel function must be called with an execution configuration:

Arguments to pass

```
__global__ void KernelFunc(...);
dim3    DimGrid(4, 0);      // 4 thread blocks
dim3    DimBlock(8, 0, 0);  // 8 threads per block

KernelFunc<<< DimGrid, DimBlock>>>(...);
```

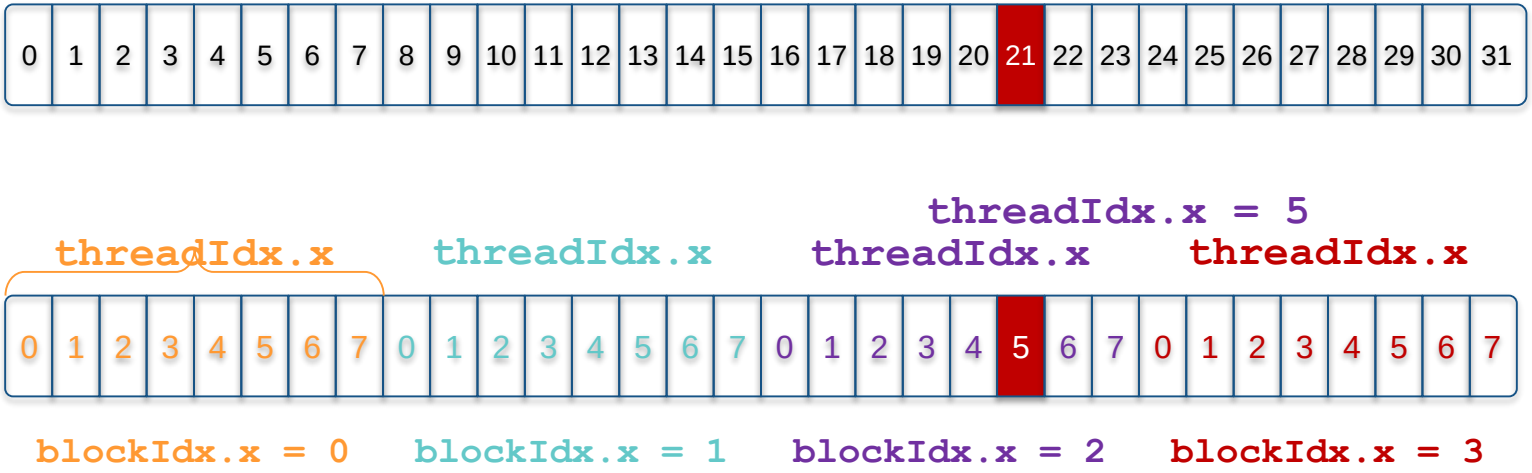
➔ Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking.

➔ On device, these calls will be processes in synchronous manner.

➔ Each thread has it local ID and global ID.

➔ `int index = threadIdx.x + blockIdx.x * blockDim;`

CUDA - Thread ID




```
int index = threadIdx.x + blockIdx.x * blockDim.X;
          =       5      +      2       * 8;
          = 21;
```

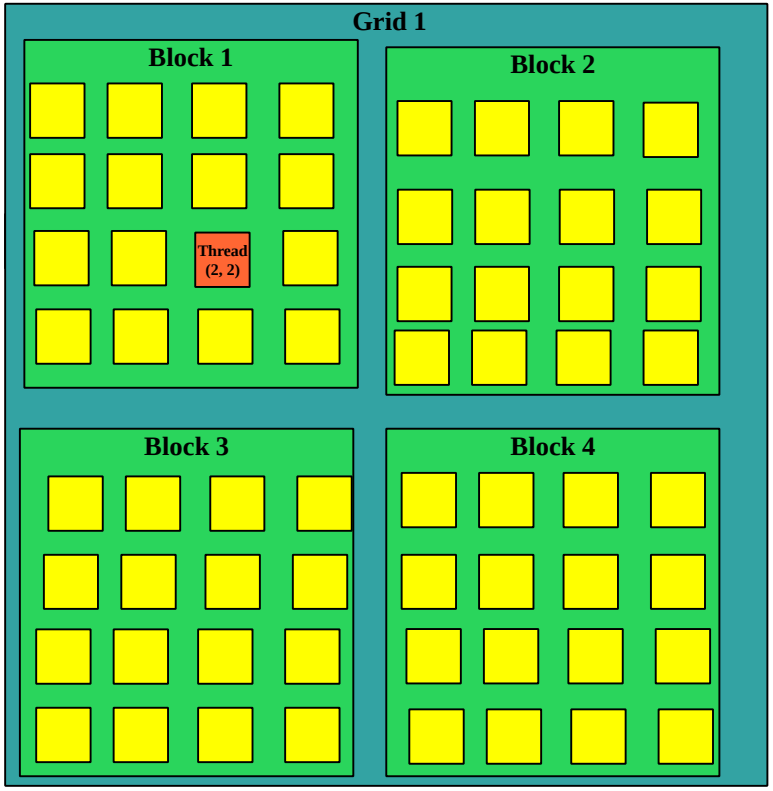
CUDA – 2D Block and Thread Dim

```
__global__ void KernelFunc(...);
dim3    DimGrid(2, 2);
dim3    DimBlock(4, 4, 0);
```

```
int indexX = threadIdx.x + blockIdx.x *
    blockDim.X;
int indexY = threadIdx.y + blockIdx.y *
    blockDim.Y;
```

```
void __syncthreads();
```

Synchronizes all threads within a block



CUDA ERROR Handling

- For non-kernel CUDA calls return value of type cudaError_t is provided to requestor. Human-radable description can be obtained by char* cudaGetErrorString(cudaError_t code);

- CUDA also provides method to retrieve last error of any previous runtime call cudaGetLastError(). There are some considirations:
  - Use cudaThreadSynchronize() to block for all kernel calls to complete. This method will return error code if such occur. We must use this otherwise nature of asynchronous execution of kernel will prevent us from getting accurate result.

Example

```
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}

#define N 512
    int main(void) {
        int *a, *b, *c;                    // host copies of a, b, c
        int *d_a, *d_b, *d_c;              // device copies of a, b, c
        int size = N * sizeof(int);

        // Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);
```

Example – ctd..

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Latest features

➔Dynamic Parallelism: GPU thread to launch a parallel GPU kernel

➔HyperQ: 32 concurrent work queues. Multirank MPI parallelism.

➔NVLink: world's first high-speed GPU interconnect - It will connect the machines' processors – CPUs and GPUs – so they can exchange data 5 to 12 times faster.

➔Unified Memory: Creates a pool of managed memory shared between the CPU and GPU, bridging the CPU-GPU divide. Managed memory is accessible to both the CPU and GPU using a single pointer.

➔XT Libraries: provide automatic scaling of cuBLAS level 3 and 2D/3D cuFFT routines to 2 or more GPUs.

➔GPU Direct: Use high-speed DMA transfers to copy data between the memories of two GPUs on the same system/PCIe bus.

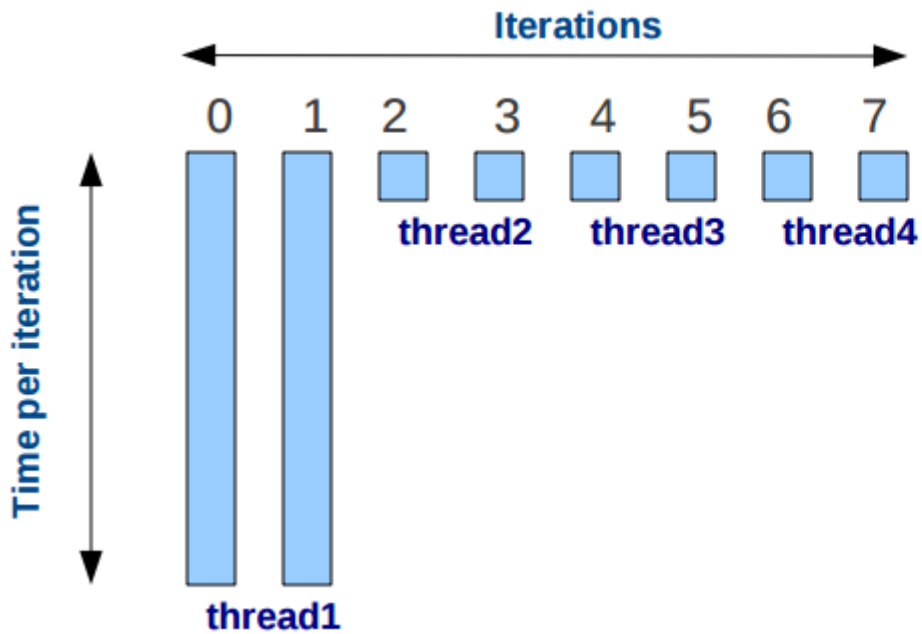
CUDA Optimizations

➔Reduce Kernel Overhead
➔Reduce memory copy overhead
➔Understand memory access pattern and read Global memory in coalesced memory access pattern
➔Use Shared memory for data that is being used frequently.
➔Overlap communication and computation.
➔Use constant memory for data that will not change in entire program
➔Avoid divergence in GPU section.


THANK
YOU!

Issues with static scheduling

With static scheduling the number of iterations is evenly distributed among all openmp threads (i.e. Every thread will be assigned similar number of iterations). This is not always the best way to partition.



This is called load imbalance. In this case threads 2,3, and 4 will be waiting very long for thread 1 to finish