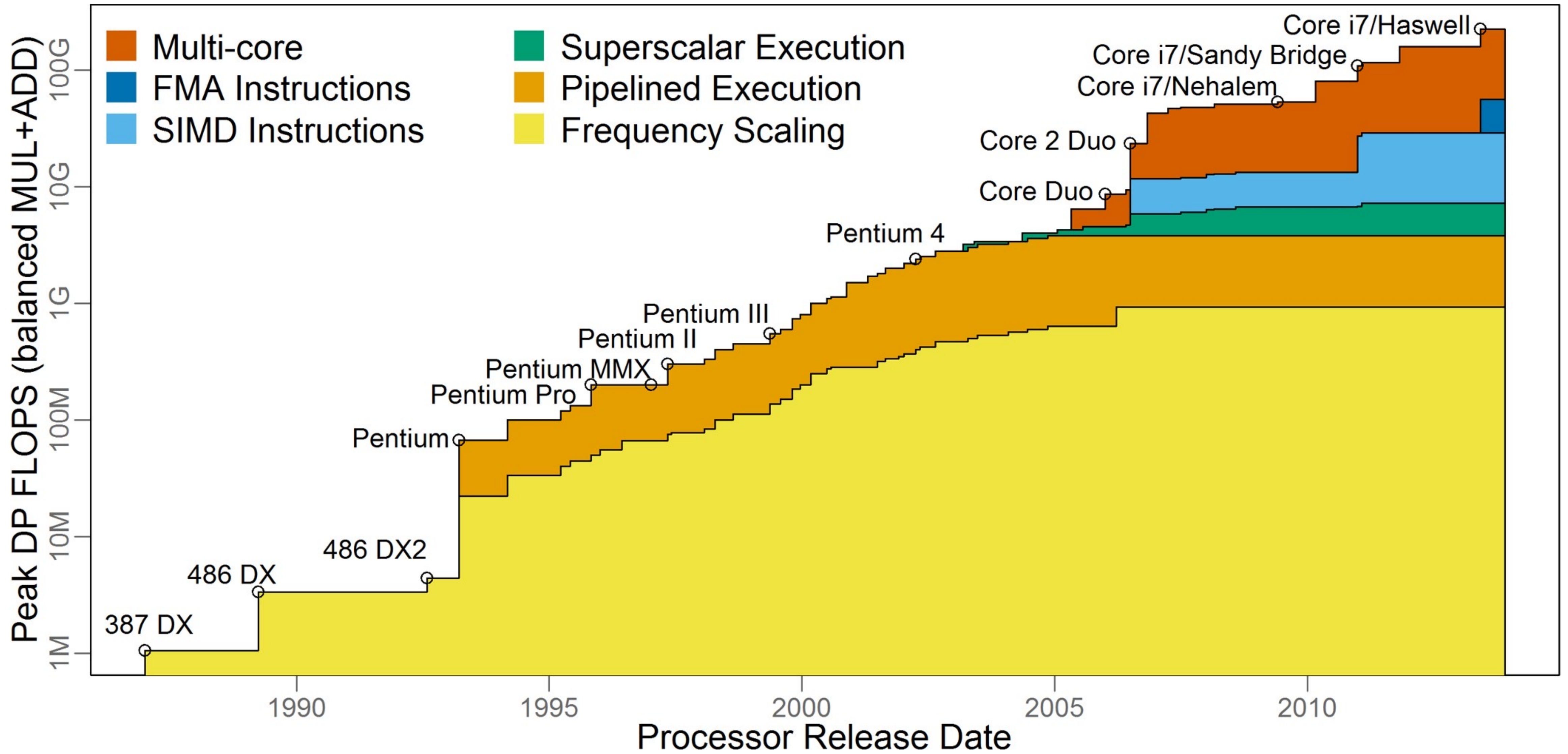
CSE 6230 – Midterm Review

Ramakrishnan Kannan, Shruti Shivakumar

Basic Terminologies - II

- **Parallel Computing**- Solving a task by simultaneous use of multiple processors in a unified architecture.
- **High Performance Computing**- Solving large problems via supercomputers + fast networks + massive storage.
- **Embarrassingly Parallel** - Solving many similar, but independent, tasks. E.g., parameter sweeps.
- **Multi-core/Many-core Processors** - Almost all processors today. Multiple compute cores on a single chip. They share memory, operating system and network.
- **Cluster Computing** - Combination of commodity units (e.g. multi-core processors) to build parallel system.
- **Pipelining (streaming)** - Breaking a task into steps performed by different units, much like an assembly line.

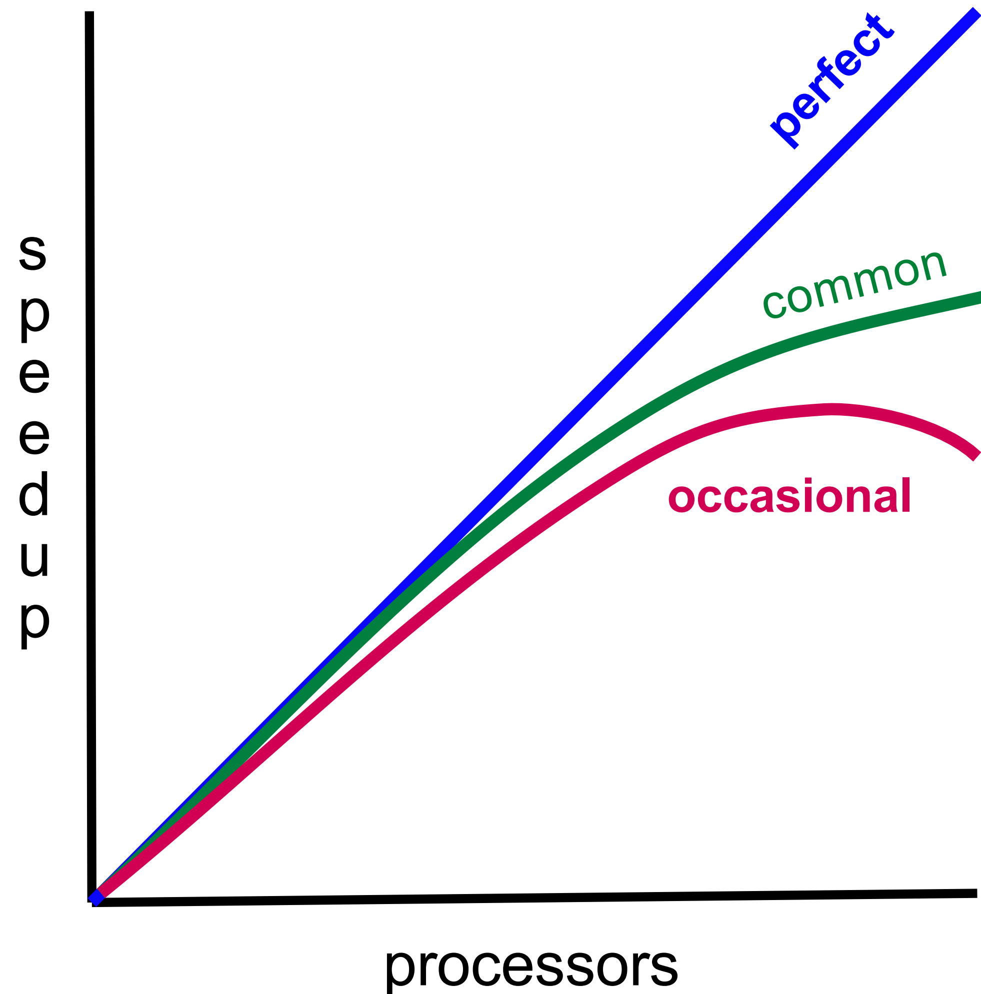


Source: Marat Dukhan <mdukan3@gatech.edu>

Some definitions

- For a given problem A, let
  - SerTime(n) = Time of best serial program to solve A for input of size n.
  - ParTime(n,p) = Time of the parallel program+architecture to solve A for input of size n, using p processors.
- Note that SerTime(n) ≤ ParTime(n,1).
- *Speedup(n,p): SerTime(n) / ParTime(n,p)* — 0 < Speedup ≤ p
- *Work(n,p): p · ParTime(n,p)* ← *cost* — Serial Work ≤ Parallel Work < ∞
- *Efficiency(n,p): SerTime(n) / [p · ParTime(n,p)]* — 0 < Efficiency ≤ 1

Speedup



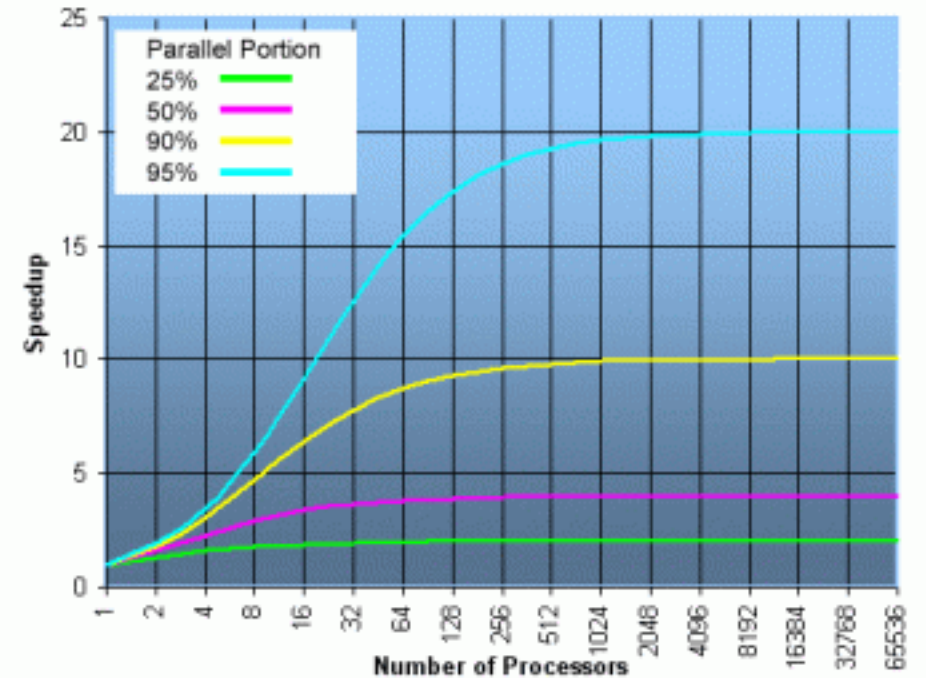
Very rare. Some reasons for speedup > p (efficiency > 1)

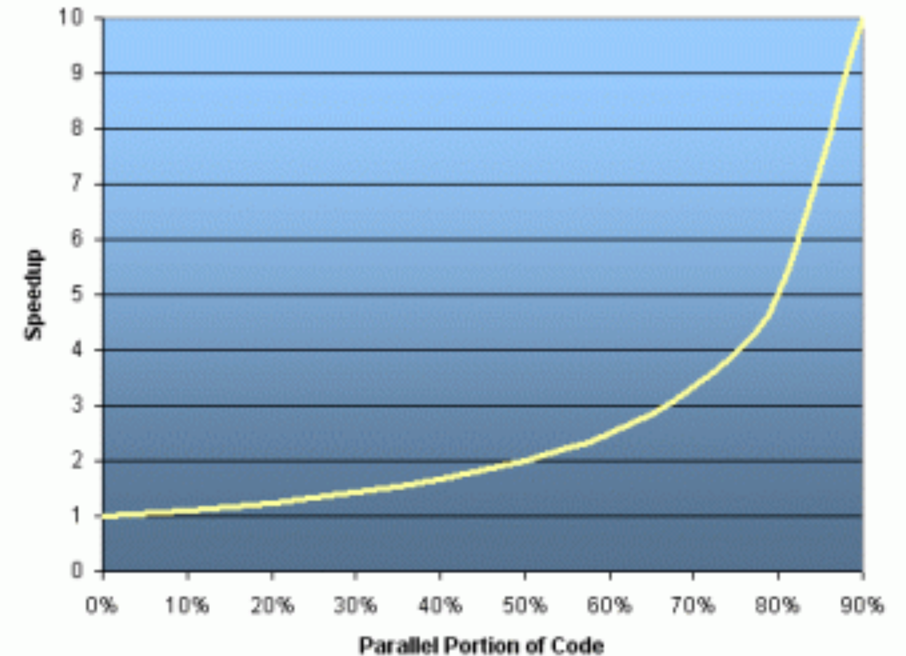
- Parallel computer has *p* times as much RAM so higher fraction of program memory in RAM instead of disk. .An important reason for using parallel computers
- In developing parallel program a better approach was discovered, older serial program was not best possible.
  - A useful side-effect of parallelization

Amdahl’s Law

- Amdahl [1967]: Let f be fraction of time spent on operations that are performed serially. Then for
  - ParTime(p) ≥ SerTime(p) $\cdot \left[f + \frac{1-f}{p}\right]$
- *Speedup(p)* $\leq \frac{1}{f + \frac{1-f}{p}}$
- Which implies

Speedup ≤ 1/*f*

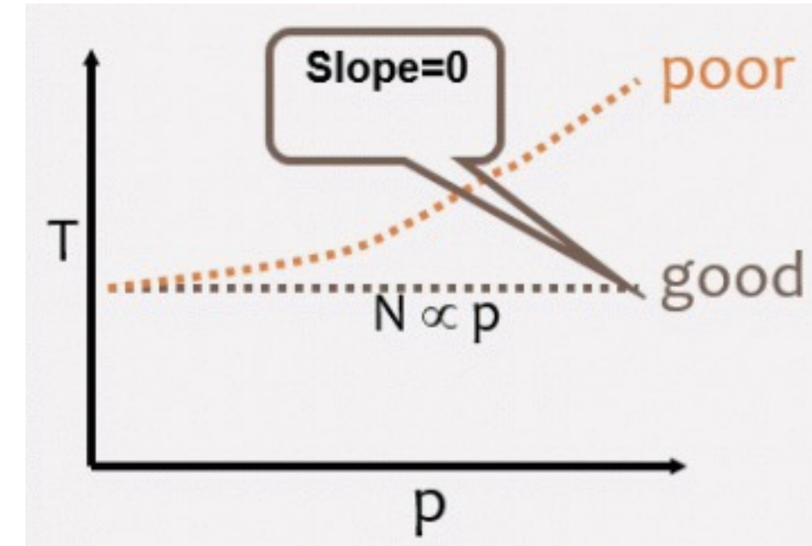




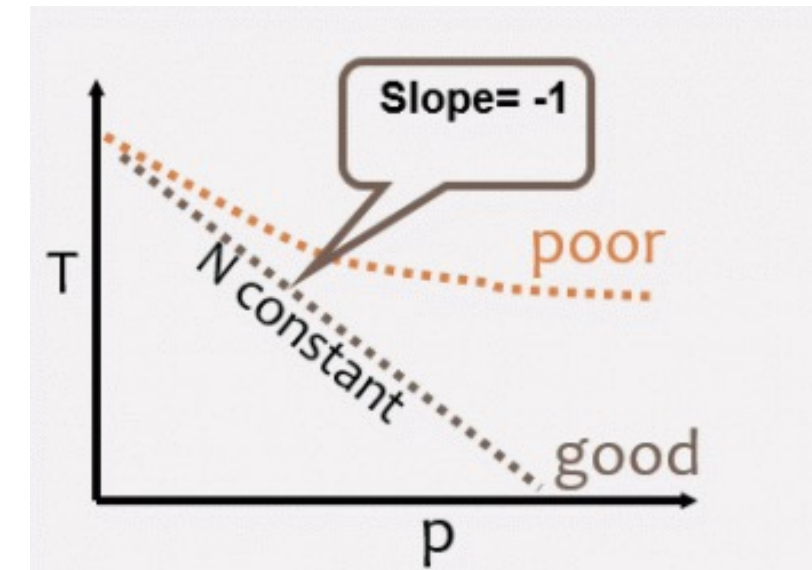
Scaling

- Utilize large computers by increasing n as p increases
- Fix the amount of data per processor: ***weak scaling***
  - Efficiency can remain high if communication does not increase excessively
  - Warning: efficiency improves, but parallel time will increase if SerTime(n) superlinear (ω(n)).
- Amdahl considered ***strong scaling*** : n is fixed
- Linear speedup is difficult
  - Nothing scales to arbitrarily many processors.
- However, for most users, the important question is:
  - Have I achieved acceptable performance on my software/hardware system for a suitable range of data and system sizes?

Weak Scaling



Strong Scaling



ARCHITECTURAL TAXONOMIES

- These classifications provide ways to think about problems and their solution.
- The classifications were originally in terms of hardware, but there are natural software analogues.
- Many systems blend approaches, and do not exactly correspond to the classifications.

$$\left\{ \begin{array}{c} \mathbf{S} \\ \mathbf{M} \end{array} \right\} \mathbf{I} \left\{ \begin{array}{c} \mathbf{S} \\ \mathbf{M} \end{array} \right\} \mathbf{D}$$

SI : Single **I**nstruction: All processors execute the same instruction.

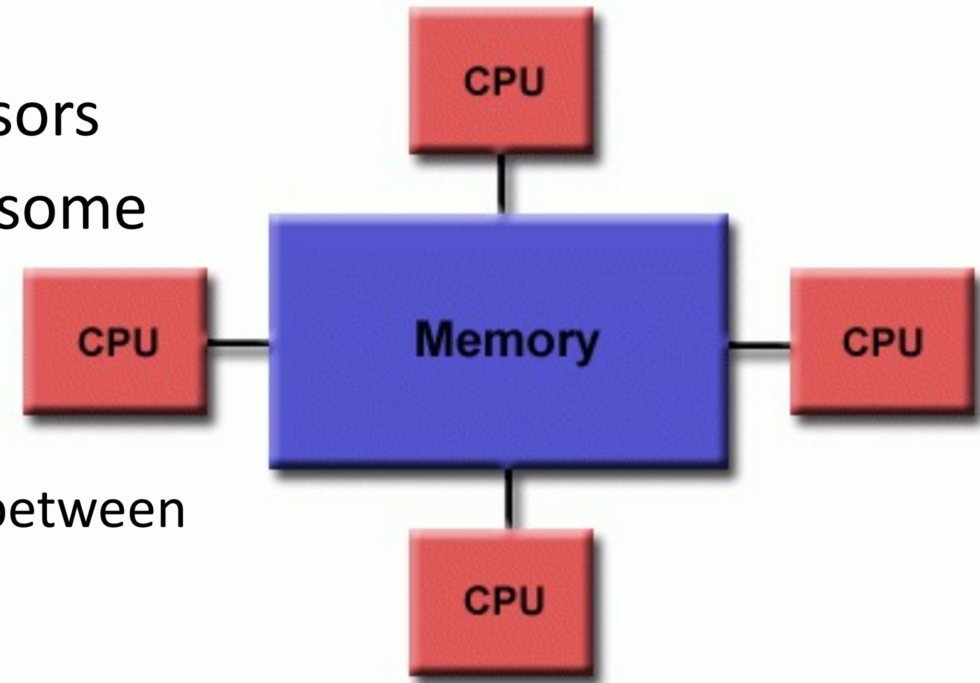
MI : Multiple **I**nstruction: Different processor may be executing different instructions.

SD : Single **D**ata: All processors are operating on the same data.

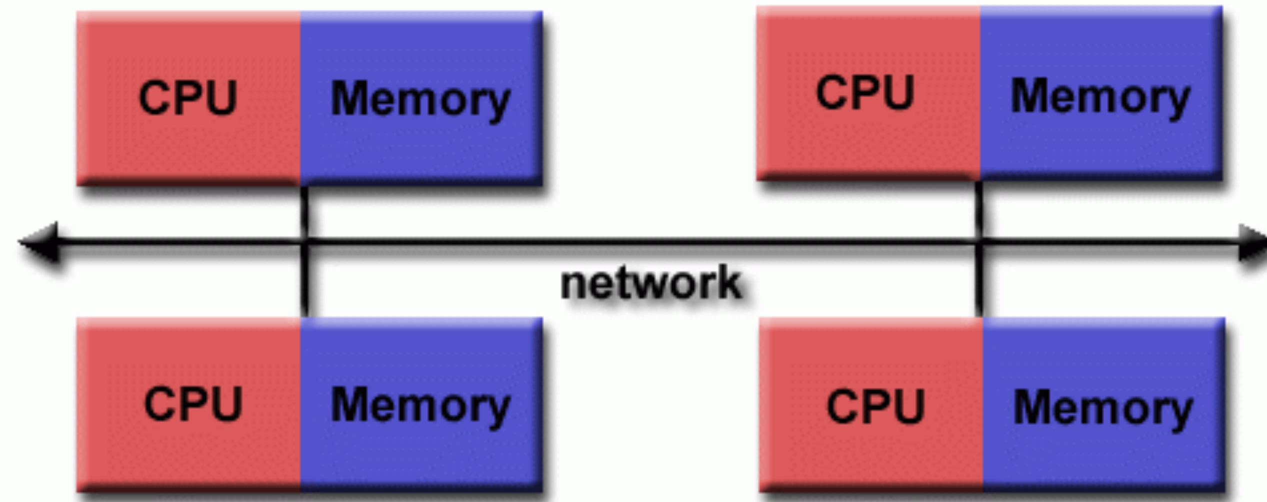
MD: Multiple **D**ata: Different processors may be operating on different data.

Shared Memory – A node or a computer

- Global memory space, accessible by all processors
- Processors may have local copies (in cache) of some global memory, consistency of copies usually maintained by hardware (cache coherency)
- Advantages:
  - Global address space is user-friendly Data sharing between tasks is fast
- Disadvantages:
  - Shared memory - to - CPU path may be a bottleneck (is bandwidth of the network sufficient?)
  - Often: Non-Uniform Memory Access (NUMA)
    - ⇒ access time varies, depends on physical distance
  - Programmer responsible for correct synchronization
- Programming Models
  - OpenMP, Cilk

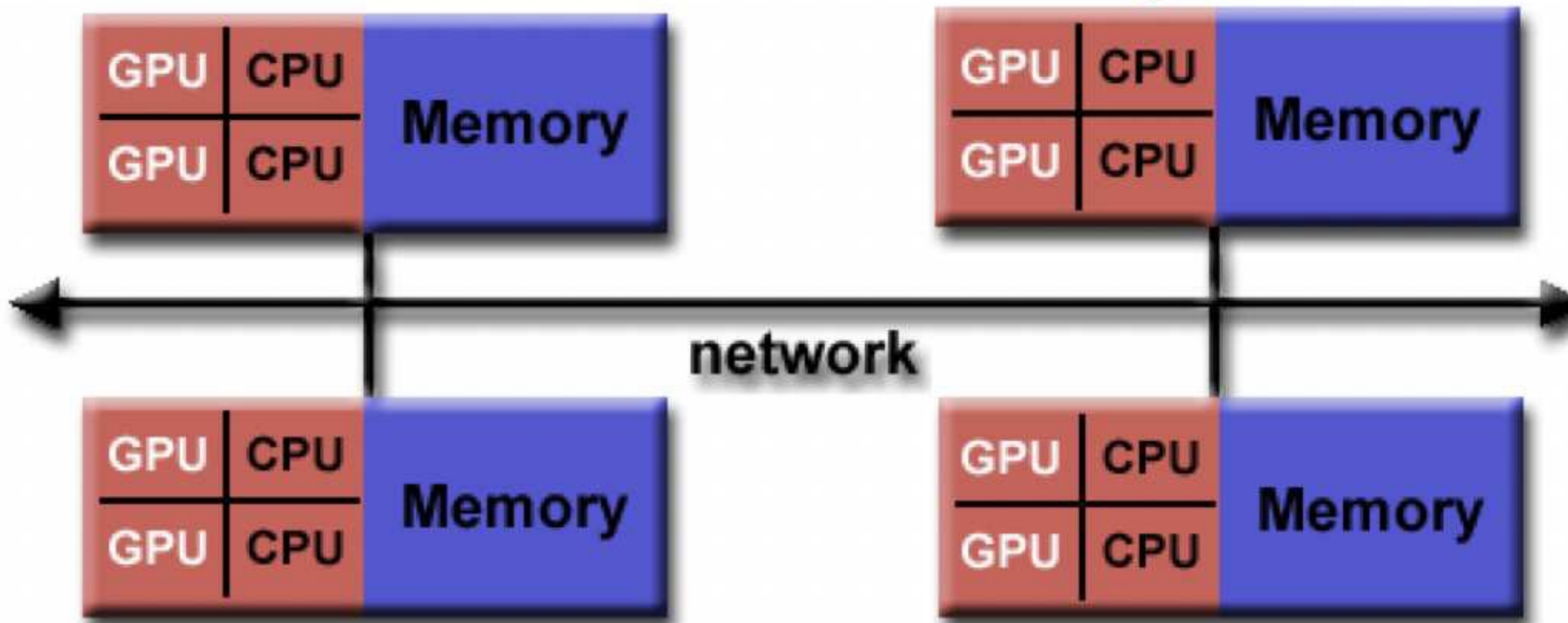


Distributed Memory



- If processor A needs data in processor B, then B must send a message to A containing the data. Thus DM systems also known as message passing systems.
- Programming Models - MPI
- Advantages:
  - Memory is scalable with number of processors
  - Each processor has rapid access to its own memory
  - Cost effective:can use commodity parts
- Disadvantages:
  - Programmer is responsible for many of the details of the communication, easy to make mistakes.
  - May be difficult to distribute the data structures

HPC Architectures with Accelerators



Shared Memory within a node with CPUs and GPUs

plus *Distributed Memory* concept: Non-local data can be sent across the network to other CPUs

Domain and Functional Decomposition

- Domain decomposition: Partition a (perhaps conceptual) space. Different processors do similar work on different pieces (quilting bee, teaching assistants for discussion sections, etc.)
- Functional decomposition: Different processors work on different types of tasks (workers on an assembly line, sub-contractors on a project, etc.)
  - Functional decomposition rarely scales to many processors, so we'll concentrate on domain decomposition.

Matrix Decompositions

- Suppose work at each position only depends on value there and nearby ones, equivalent work at each position.
- Dependencies force communication along boundary

Minimizes Bandwidth

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

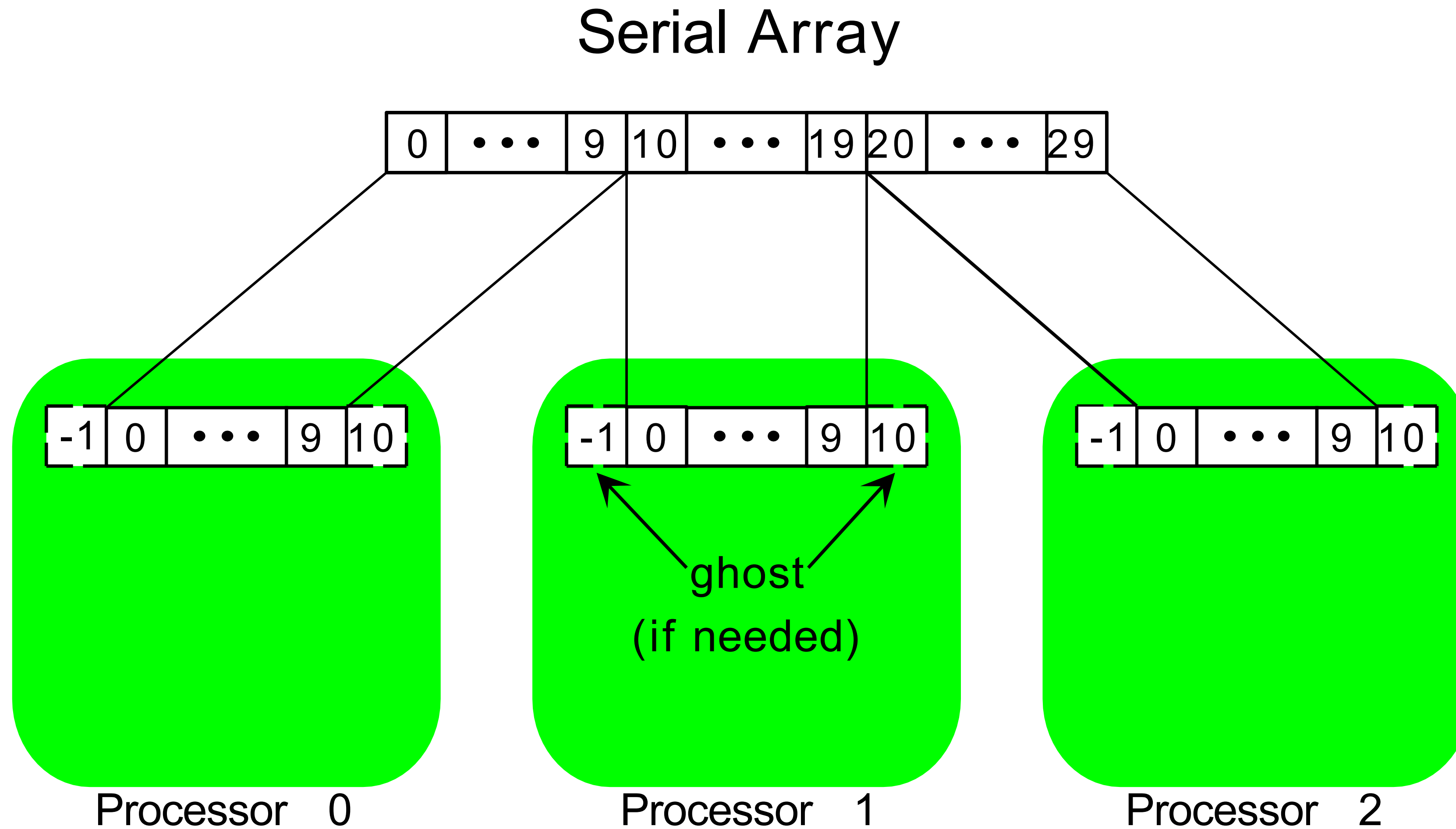
Minimize boundary (2D)

Minimizes Latency

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Minimize # neighbors (1D)

Local vs Global Arrays



MPI Ranks -- Linear vs 2D Grid

MPI ranks 0...15

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

logical rows and columns 0 ... 3

For MPI_COMM_RANK = i and MPI_COMM_SIZE = p

$\mathrm{my_row} = \lfloor i/\sqrt{p} \rfloor$ and $\mathrm{my_col} = i - \mathrm{my_row} * \sqrt{p}$

to send to logical		send to MPI_COMM_RANK
Right	(my_row, my_col +1)	$i + 1$
Left	(my_row, my_col -1)	$i - 1$
Up	(my_row -1, my_col)	$i - \sqrt{p}$
Down	(my_row +1, my_col)	$i + \sqrt{p}$

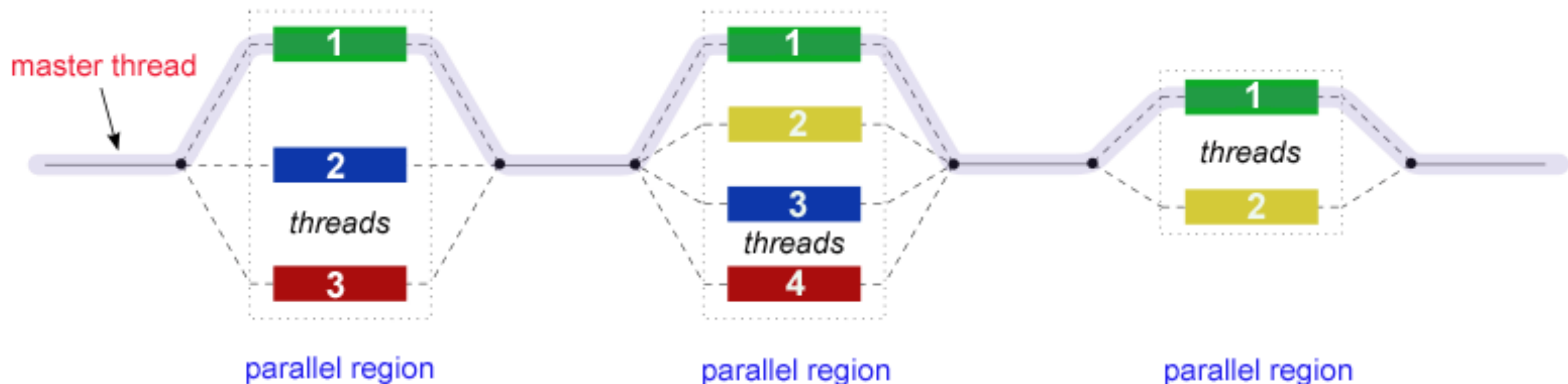
OpenMP

Three building blocks

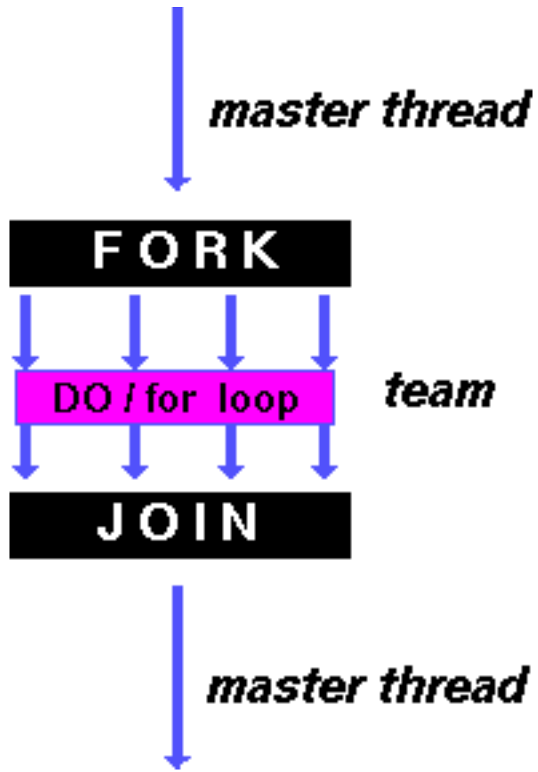
- Compiler Directives
  - private(list), shared(list)
  - firstprivate(list), lastprivate(list)
  - reduction(operator:list)
  - schedule(method[,chunk_size])
  - nowait
  - if(scalar_expression)
  - num_thread(num)
  - threadprivate(list), copyin(list)
  - Ordered
- Runtime Libraries/APIs
  - omp_set/get_num_threads, omp_get_thread_num, omp_{set,get}_dynamic, omp_in_parallel, omp_get_wtime
- Environment variables
  - OMP_NUM_THREADS, OMP_SCHEDULE, OMP_STACKSIZE, OMP_DYNAMIC, OMP_NESTED, OMP_WAIT_POLICY

Fork-Join Model

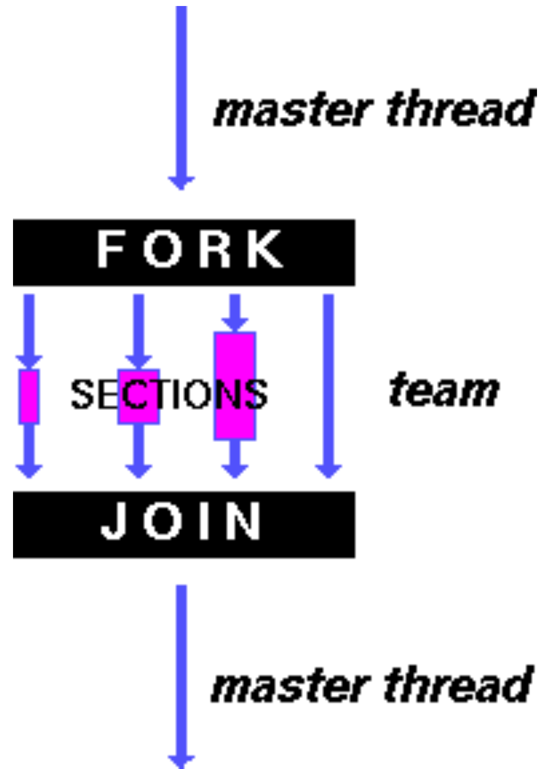
- OpenMP programs begin as a single process: the master thread.
- The master thread executes sequentially until the first parallel region construct is encountered.
- FORK: the master thread then creates a team of parallel threads.
  - The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads that comprise them are arbitrary.



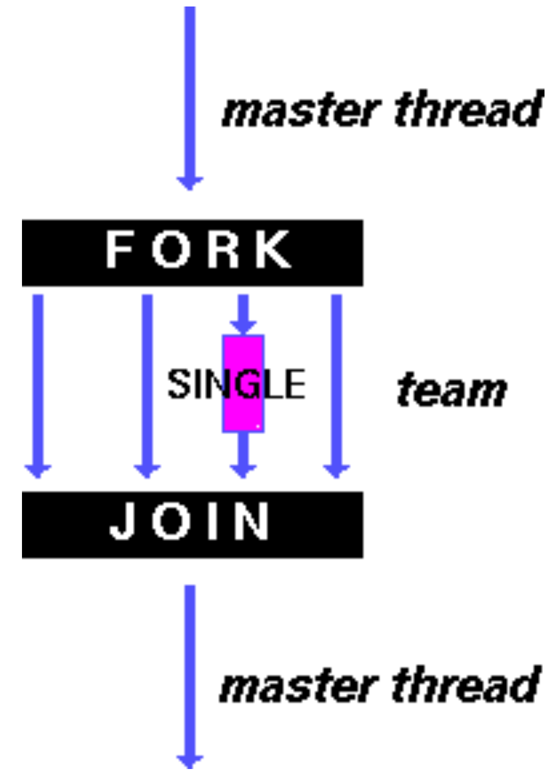
Workshare vs Sections vs Single



DO / for shares iterations of a loop across the team. Represents a type of “data parallelism”.



SECTIONS breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of “functional parallelism”.



SINGLE serializes a section of code

Synchronization : Critical and Atomic

- Critical: Only one thread at a time can enter a critical region

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

- Atomic: Only one thread at a time can update a memory location

```
#include
main()
{
int x;
x = 0;
#pragma omp parallel shared(x)
  {
  #pragma omp atomic
  x = x + 1;
  }  /* end of parallel section */
}
```

Reduction

- The reduction clause allows accumulative operations on the value of variables.
- Syntax: reduction (operator:variable list)
- A private copy of each variable which appears in reduction is created as if the private clause is specified.
- Operators
  - Arithmetic
  - Bitwise
  - Logical

```
int main()
{
    int i, n;
    n = 10000;
    float a[n], b[n];
    double result, sequential_result;
    /* Some initializations */
    result = 0.0;
    for (i = 0; i < n; i++)
    {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }
#pragma omp parallel for default(shared) private(i) \
    schedule(static) reduction(+ : result)
    for (i = 0; i < n; i++)
        result = result + (a[i] * b[i]);
    printf("Final result= %f\n", result);
    return 0;
}
```

MPI

Different MPI Functionalities

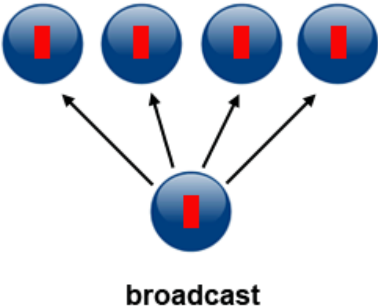
- Point-2-Point communications
- Collective Communications
- MPI-IO
- Tools Interface
- One sided message passing
- Derived Datatypes

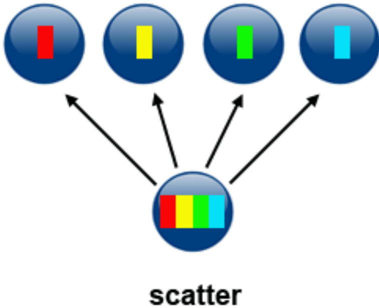
Point to Point communication

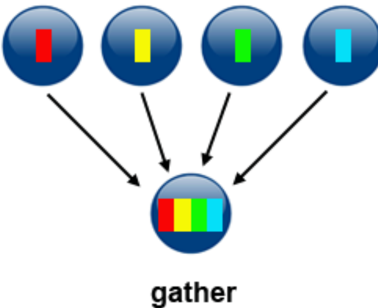
- MPI provides four variants on send with blocking and nonblocking versions of each.
- Blocking means the call will not complete until the local data is safe to modify
- Nonblocking means the call returns “immediately”
  - Nonblocking data movement calls in MPI are MPI_I{command}, e.g. MPI_Irecv() or MPI_Ialltoallv() (capital “Eye”)
  - Nonblocking calls require a mechanism to tell when they are done – MPI_Wait*, MPI_Test*
  - Data may or may not actually move before a call to MPI_Wait*/MPI_Test*
  - It is not safe to reuse buffers until the Wait/Test says the operation is locally done.
  - Nonblocking calls (can) allow for compute and communication to overlap

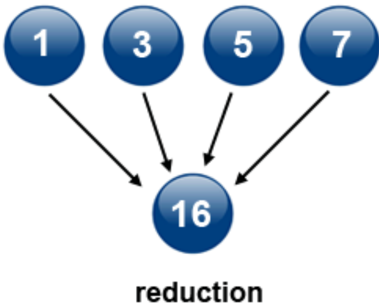
Collective Communication

- One process wants to communicate with multiple process
  - One Process to all other process

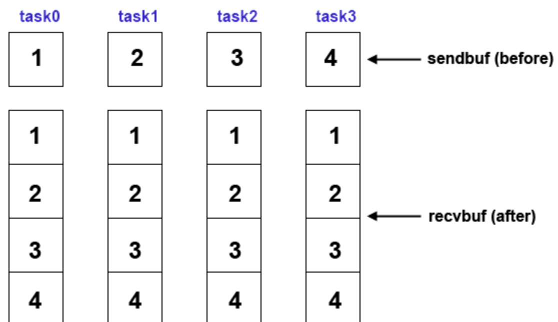


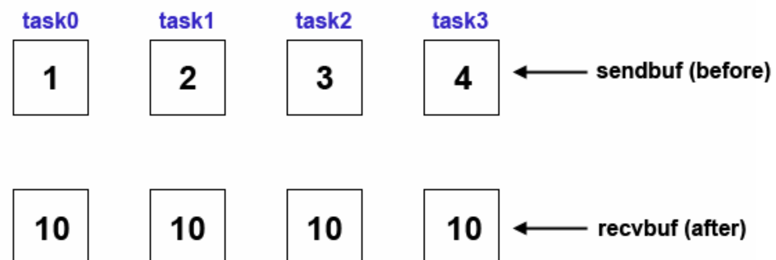


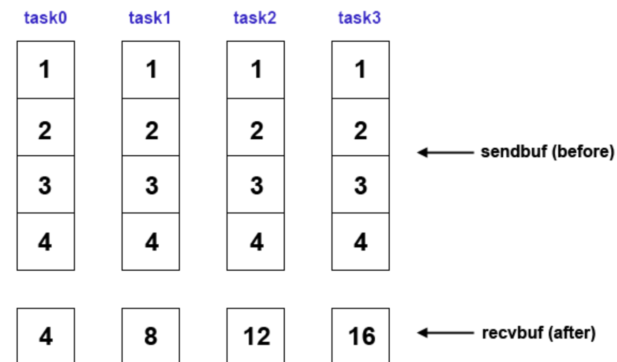


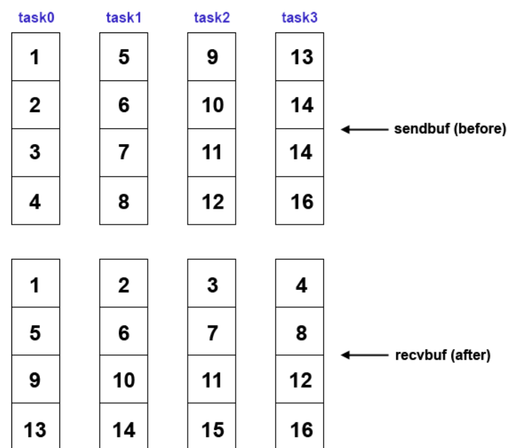


MPI_All*

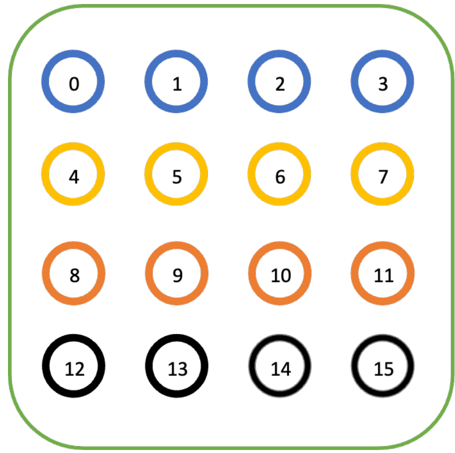


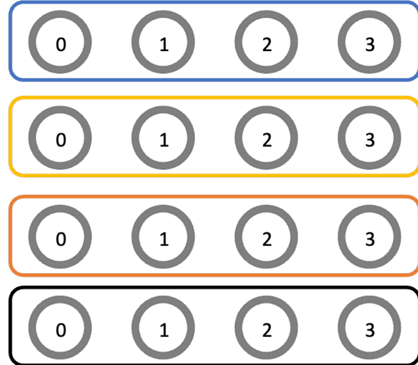






MPI Communicator Split






```
// Get the rank and size in the original
communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color
based on row

// Split the communicator based on the color and
use the original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank,
&row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d --- ROW RANK/SIZE:
%d/%d\n",
world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);

MPI_Finalize();
```

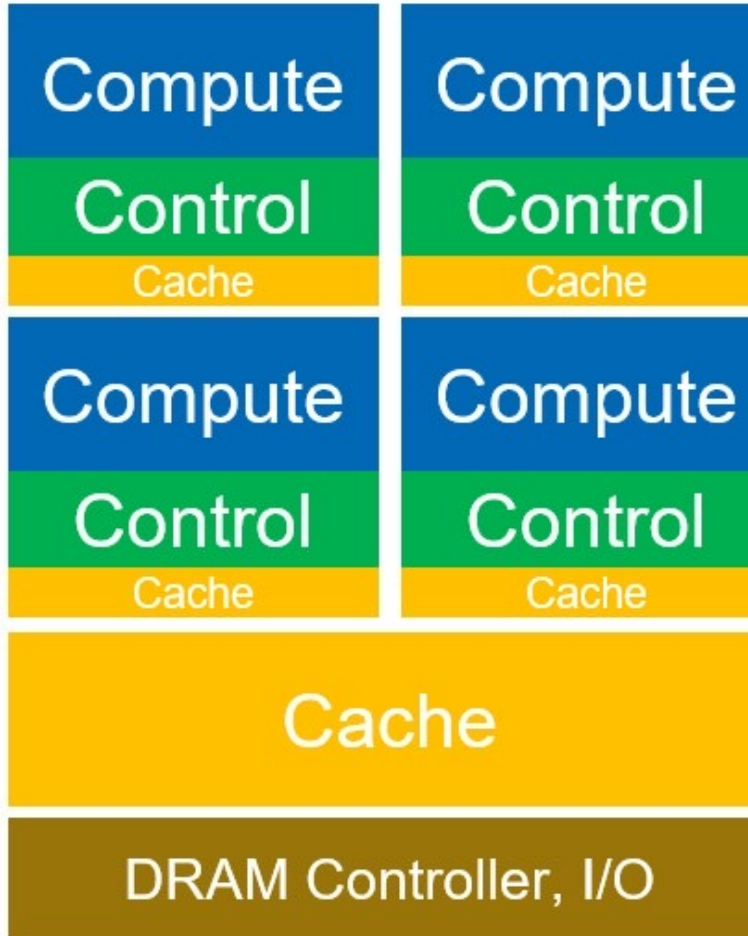
Introduction to CUDA

Ramakrishnan Kannan

Shruti Shivakumar

Motivated out of OLCF training seminar

CPU vs GPU



CPU

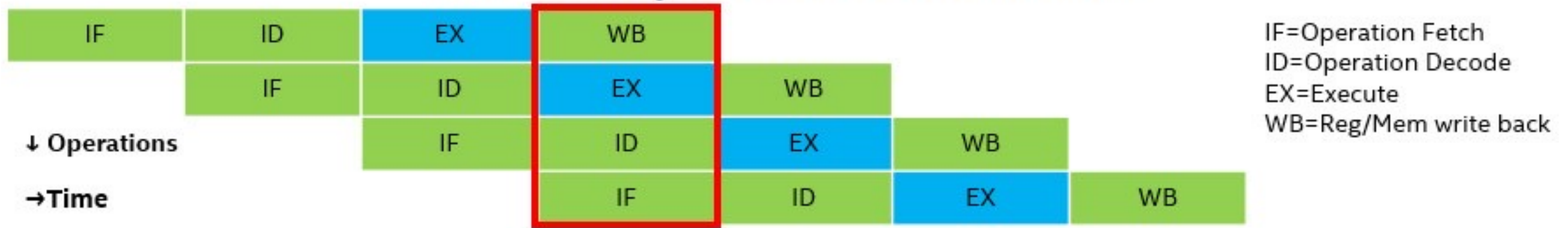
- can execute instructions, divided into stages, at the rate of up to one instruction per clock cycle (IPC) when there are no dependencies.
- Scalar Piplined Execution
- designed to process serial instructions efficiently.
- find instruction-level parallelism and execute multiple out-of-order instructions per clock cycle.

GPU

- SIMT == SIMD + Multithreading
- Is optimized for aggregate throughput across all cores, deemphasizing individual thread latency and performance.
- Efficiently processes vector data
- Dedicates more silicon space to compute and less to cache and control.

https://www.intel.com/content/www/us/en/developer/articles/technical/comparing-cpus-gpus-and-fpgas-for-oneapi.html#gs.npufd2

CPU Vs GPU Demo



CPU Advantages

- Out-of-order superscalar execution
- Sophisticated control to extract tremendous instruction level-parallelism
- Accurate branch prediction
- Automatic parallelism on sequential code
- Large number of supported instructions
- Lower latency when compared to offload acceleration
- Sequential code execution results in ease-of-development

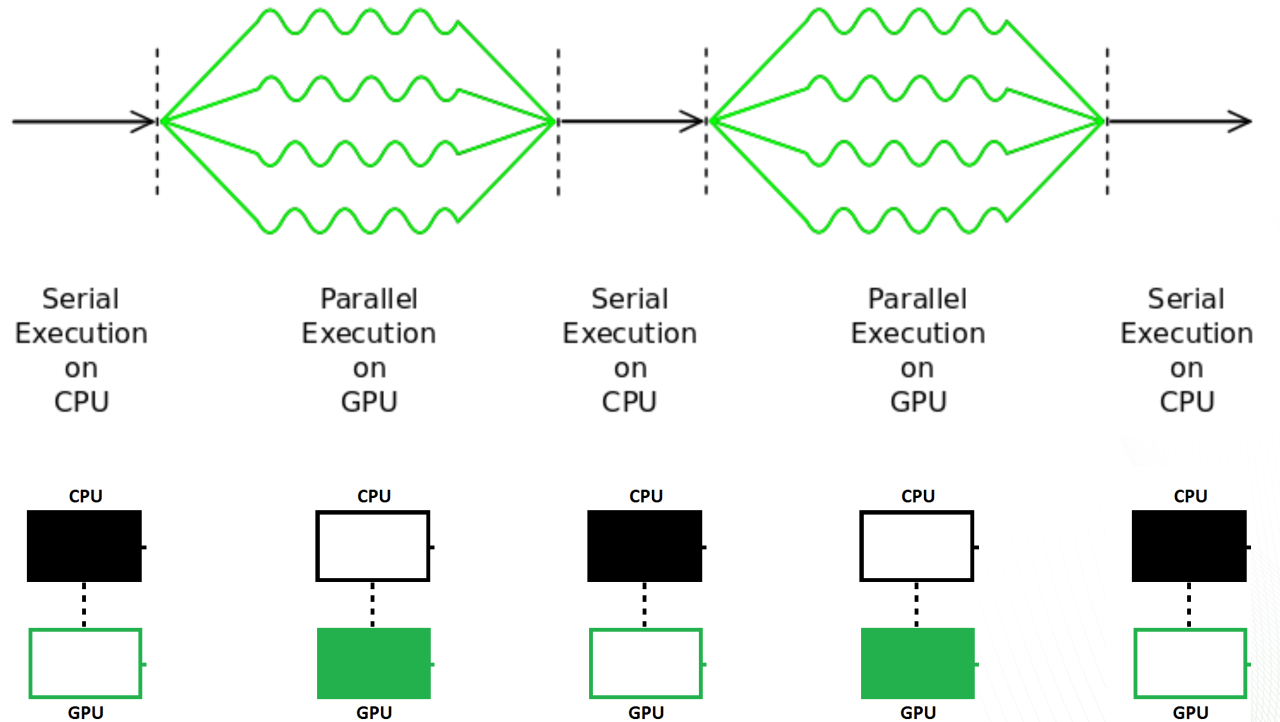
GPU Advantages:

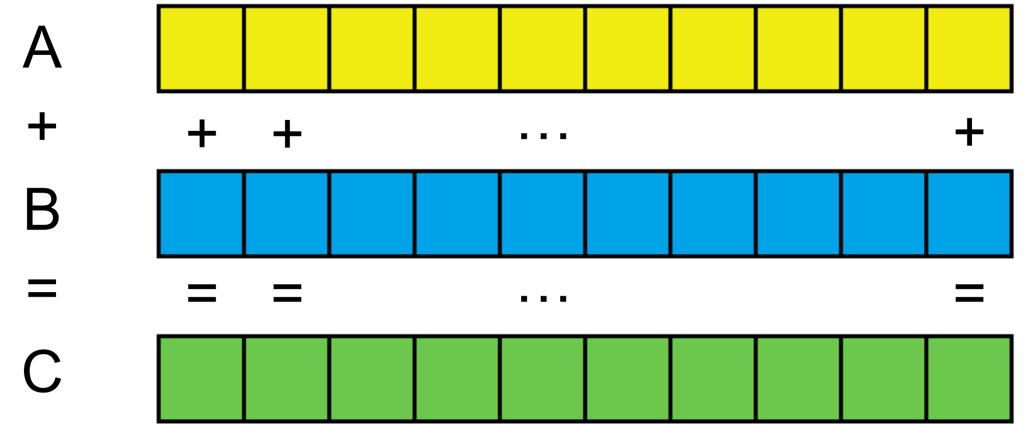
- Massively parallel, up to thousands of small and efficient SIMD cores/EUs
- Efficient execution of data-parallel code
- High dynamic random-access memory (DRAM) bandwidth

https://www.youtube.com/watch?v=-P28LKWTzrI

CUDA Programming Model

- Heterogenous Programming
  - program separated into serial regions (run on CPU) & parallel regions (run on GPU)
- Data Parallelism - Parallel regions consist of many calculations that can be executed independently



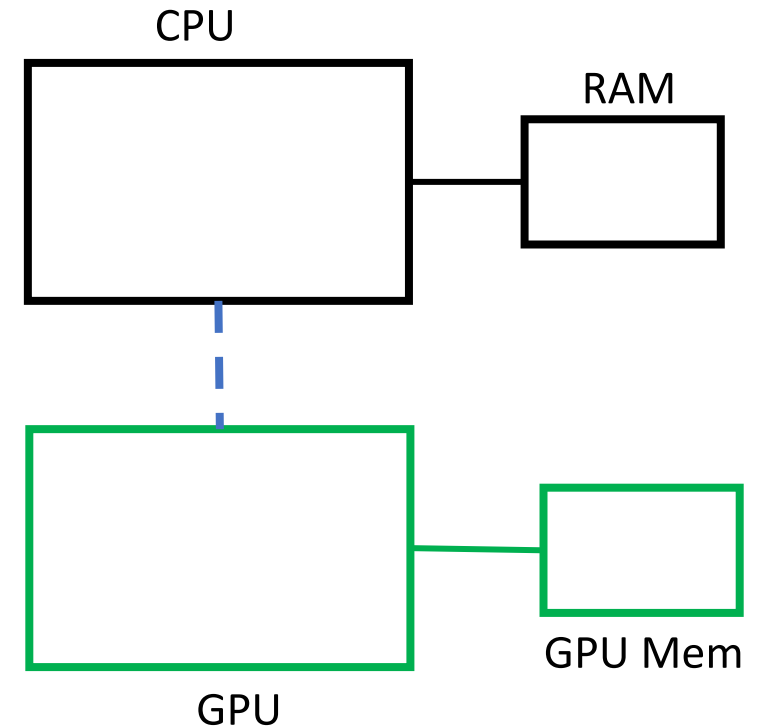


At its core are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization – that are simply exposed to the programmer as a minimal set of language extensions (to C programming language)

OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY
OAK RIDGE National Laboratory

CUDA Program Outline

```
int main(){
  // Allocate memory for array on host
  // Allocate memory for array on device
  // Fill array on host
  // Copy data from host array to device array
  // Do something on device (e.g. vector addition)
  // Copy data from device array to host array
  // Check data for correctness
  // Free Host Memory
  // Free Device Memory
}
```

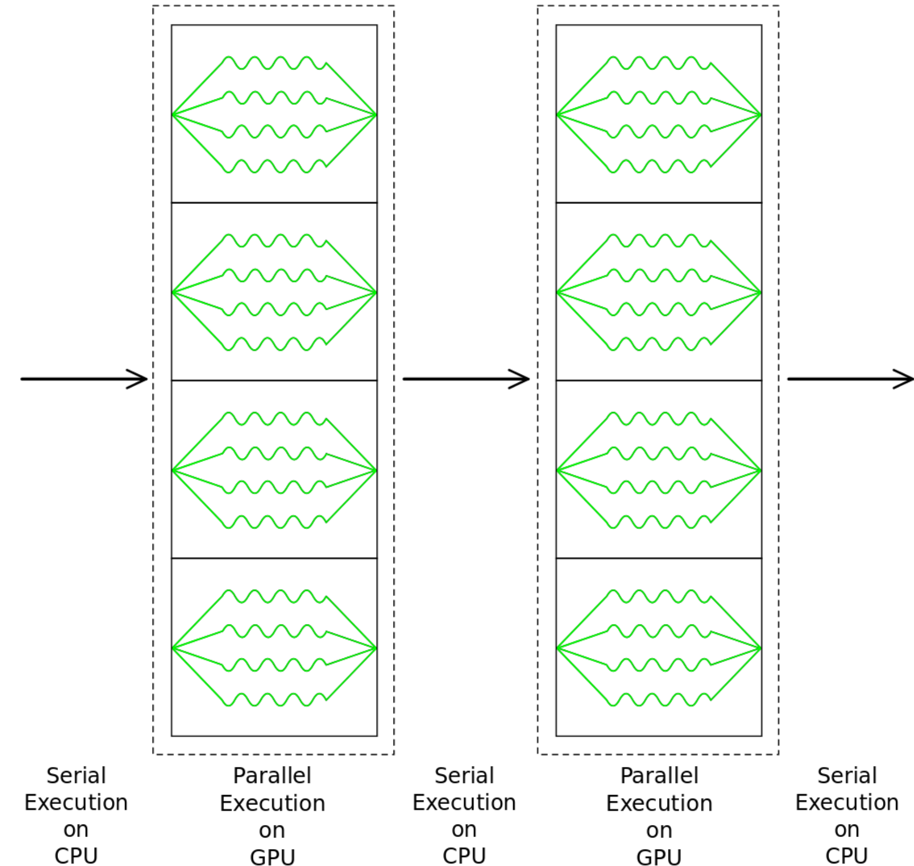


CUDA Kernel

- When kernel is launched, a grid of threads are generated
- SIMD Code – Same code is executed by all threads
- Serial – CPU

```
for (int i=0; i<N; i++) {
  C[i] = A[i] + B[i];
}
```

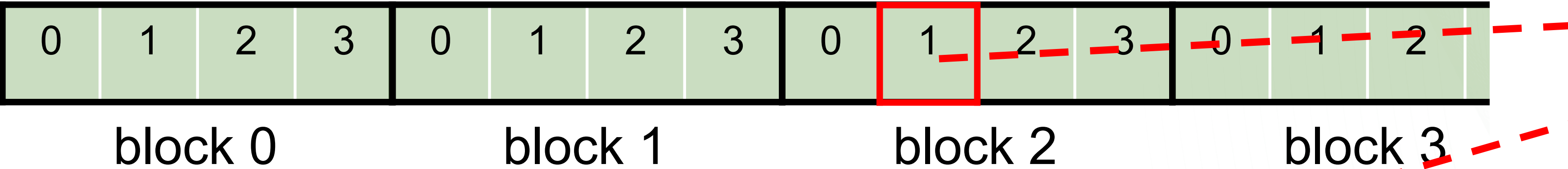
- GPU – Parallel Code
  - C[i] = A[i] + B[i];



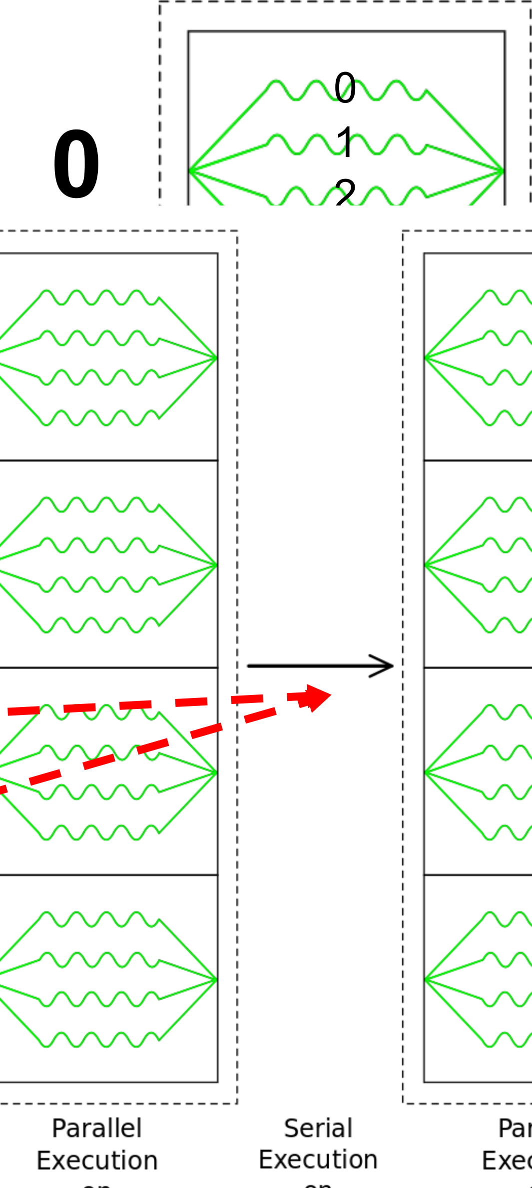
1D Indexing in CUDA programming

```
__global__ void vector_addition(int *a, int *b, int *c)
{
                  (4)           (0–3)       (0–3)
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i<N) c[i] = a[i] + b[i];
}
```

```
thr_per_blk = 4;
blk_in_grid = ceil( float(N) / thr_per_blk );
vec_add<<< blk_in_grid, thr_per_blk >>>(d_a, d_b, d_c);
```




```
int i = 4 * 2 + 1 = 9
```



CUDA Threads to 2D Array

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$	$A_{0,5}$	$A_{0,6}$	$A_{0,7}$	$A_{0,8}$	$A_{0,9}$		
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$A_{1,5}$	$A_{1,6}$	$A_{1,7}$	$A_{1,8}$	$A_{1,9}$		
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	$A_{2,5}$	$A_{2,6}$	$A_{2,7}$	$A_{2,8}$	$A_{2,9}$		
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$	$A_{3,5}$	$A_{3,6}$	$A_{3,7}$	$A_{3,8}$	$A_{3,9}$		
$A_{4,0}$	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$	$A_{4,5}$	$A_{4,6}$	$A_{4,7}$	$A_{4,8}$	$A_{4,9}$		
$A_{5,0}$	$A_{5,1}$	$A_{5,2}$	$A_{5,3}$	$A_{5,4}$	$A_{5,5}$	$A_{5,6}$	$A_{5,7}$	$A_{5,8}$	$A_{5,9}$		
$A_{6,0}$	$A_{6,1}$	$A_{6,2}$	$A_{6,3}$	$A_{6,4}$	$A_{6,5}$	$A_{6,6}$	$A_{6,7}$	$A_{6,8}$	$A_{6,9}$		

Let M = 7 rows, N = 10 columns and a 4x4 blocks of threads...
To cover all elements in the array, we need 3 blocks in x-dim and 2 blocks in y-dim.

```
dim3 threads_per_block( 4, 4, 1 );
dim3 blocks_in_grid( ceil( float(N) / threads_per_block.x ), ceil( float(M) / threads_per_block.y ) , 1 );
mat_add<<< blocks_in_grid, threads_per_block >>>(d_a, d_b, d_c);
```

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

```
__global__ void add_matrices(int *a, int *b, int *c)
{
        int column = blockDim.x * blockIdx.x + threadIdx.x; // [0-11]
        int row = blockDim.y * blockIdx.y + threadIdx.y; //[0-7]
        if (row < M && column < N) //There is no row 7 and col 11.
        {
                int thread_id = row * N + column; //[0-69]
                c[thread_id] = a[thread_id] + b[thread_id];
        }
}
```

Row = 4
Column = 5
Thread_id = 4 * 10 + 5 = 45

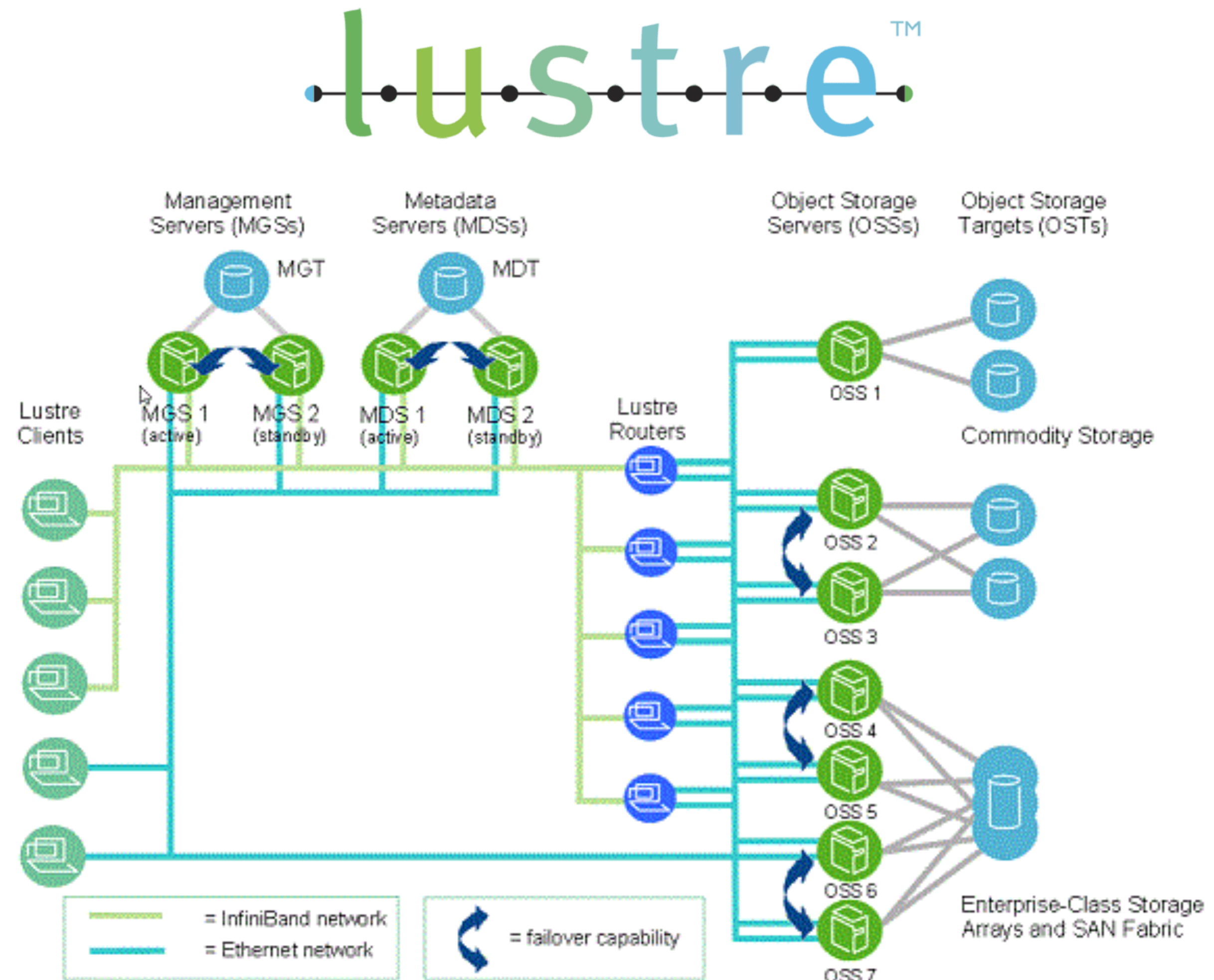
OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

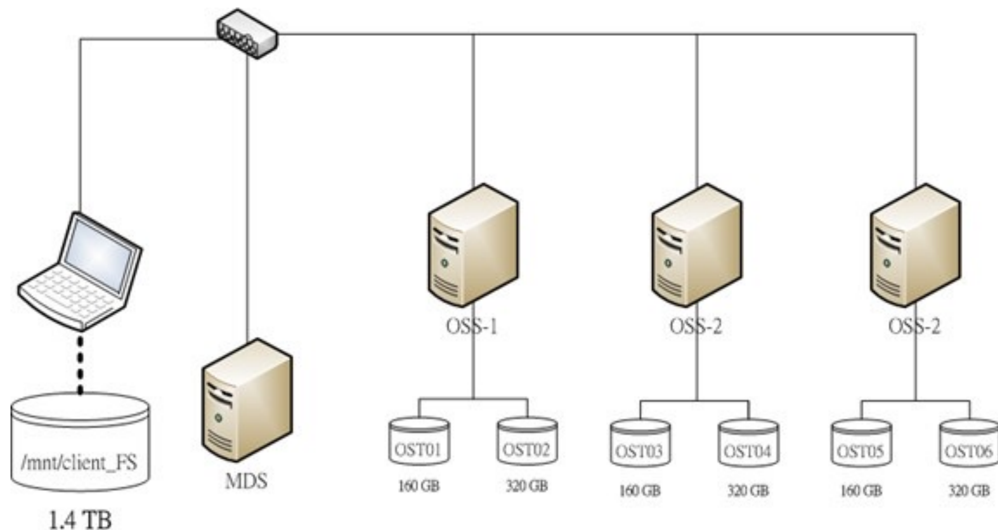
OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

Parallel IO

Lustre

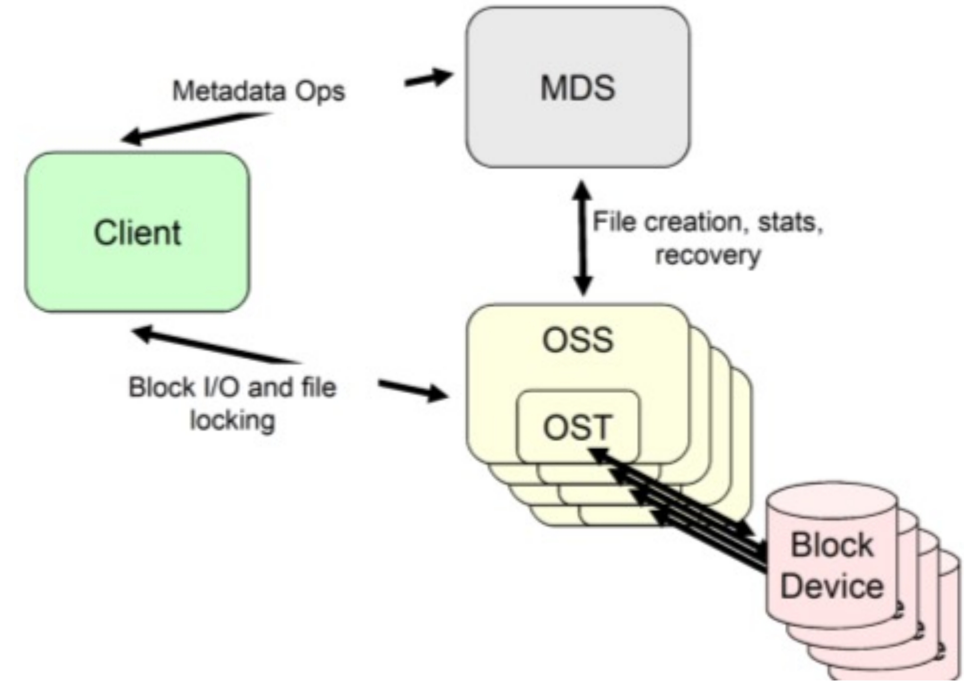
- Distributed file system
- Hierarchical management
- Concurrency from multiple OSTs
- Meta data management with multiple MDTs

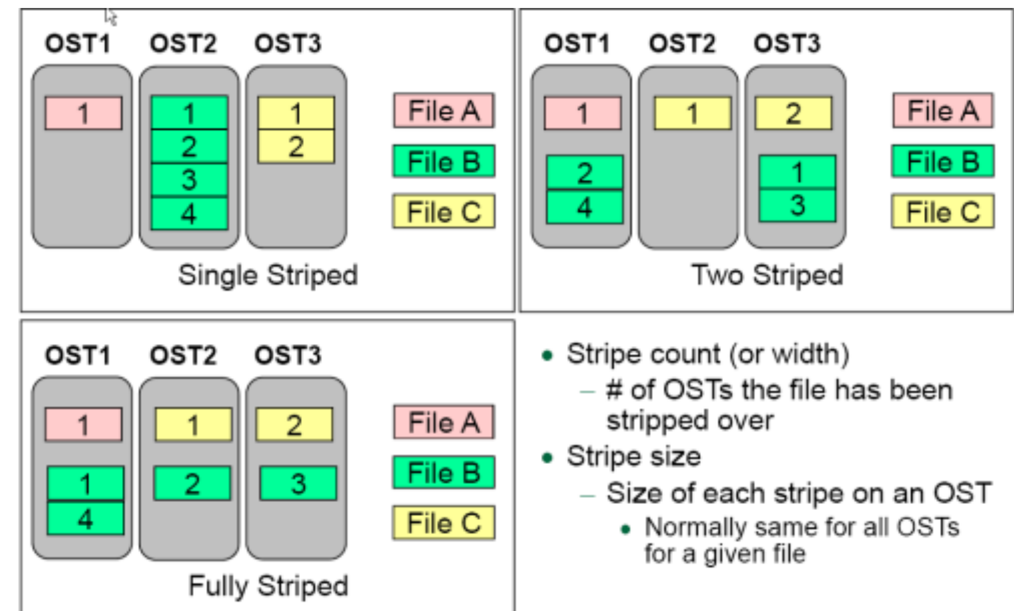




Lustre Components

- Lustre consists of four major components
  - MetaData Server (MDS)
  - Object Storage Servers (OSSs)
  - Object Storage Targets (OSTs)
  - Clients
- MDS: track meta data (eg., name, location)
- OST: back-end storage for file object data
- Performance: Striping, alignment, placement

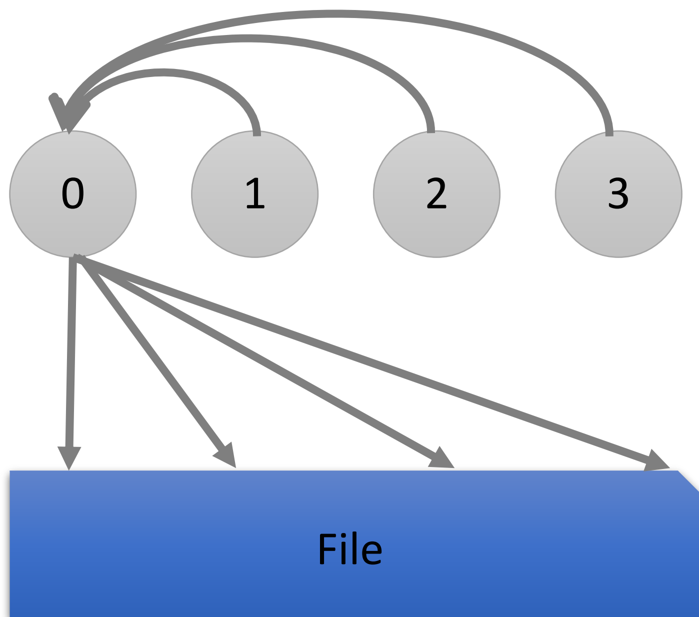




(ADIOS tutorial, ORNL)

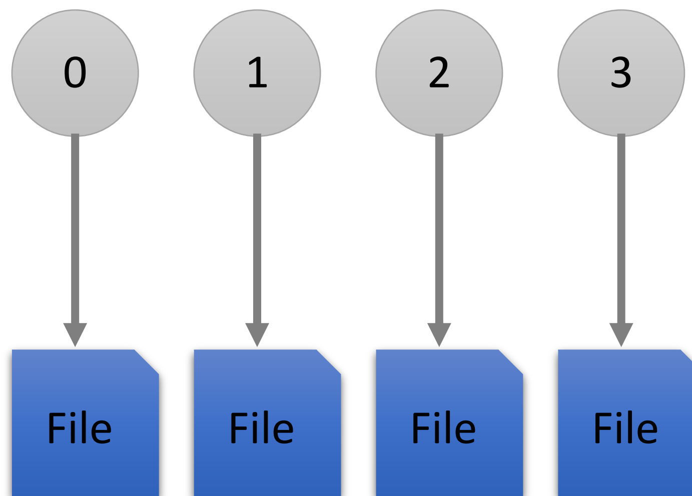
HPC I/O Patterns

Non-parallel I/O



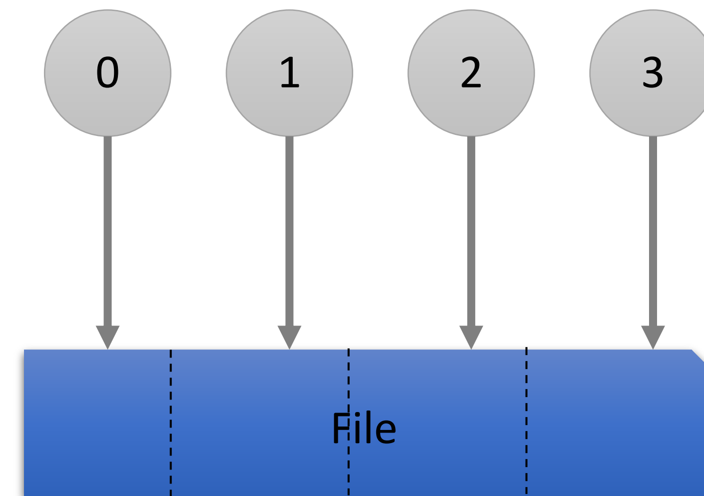
- Serial performance
- Scaling issue
- Memory limit

Parallel Multi-file I/O



- Metadata issue
- Management issue

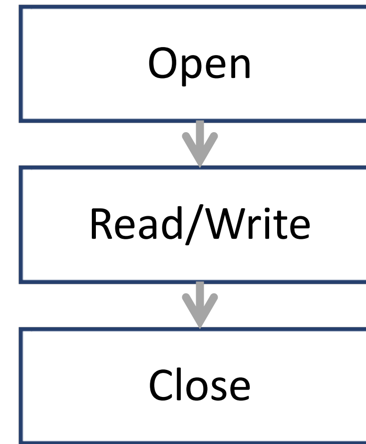
Parallel Single-file I/O



- User-friendly
- Sync/lock overhead

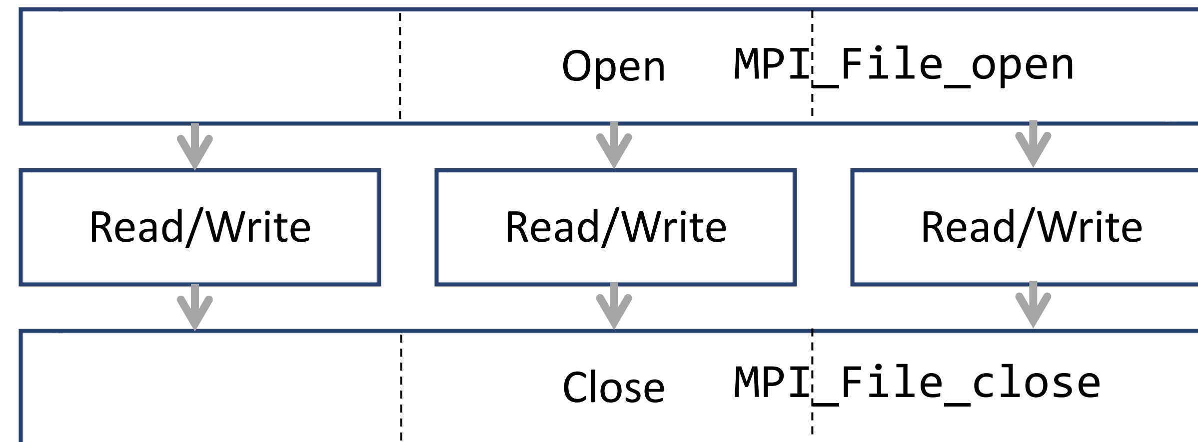
What is MPI I/O?

- I/O interface specification for parallel MPI applications
- Parallel I/O part for MPI
- MPI IO provides
  - Parallel I/O operations
  - Enable to use efficiently parallel file systems
  - Independent/collective I/Os
- Low-level interface
- At the application level, users may want to use of a more abstract library




```
fd = open("foo.txt", O_WRONLY
| O_CREAT, 0644);
write(fd, buf, len);
close(fd)
```

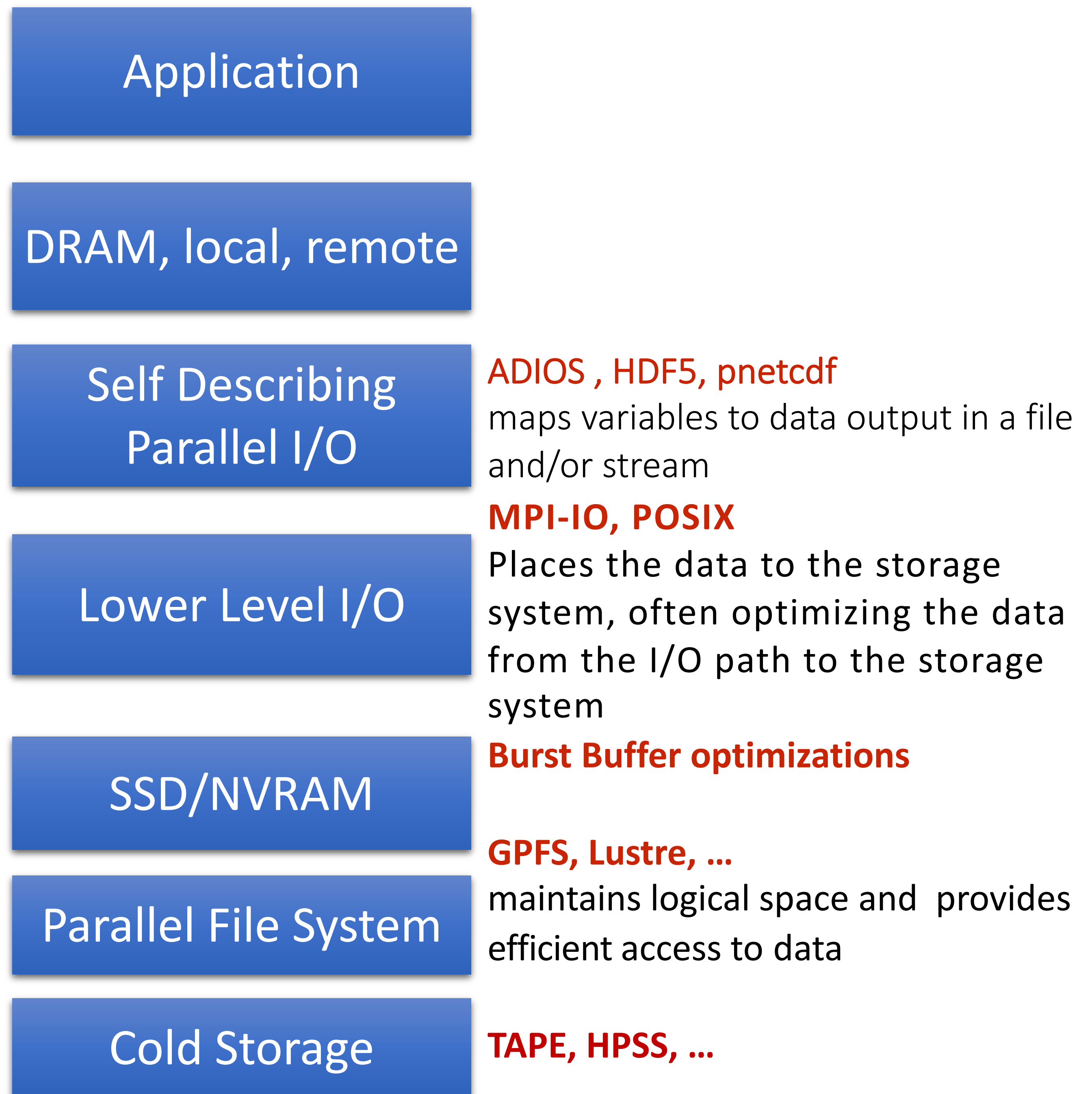
POSIX I/O



MPI I/O

I/O and Storage Stack

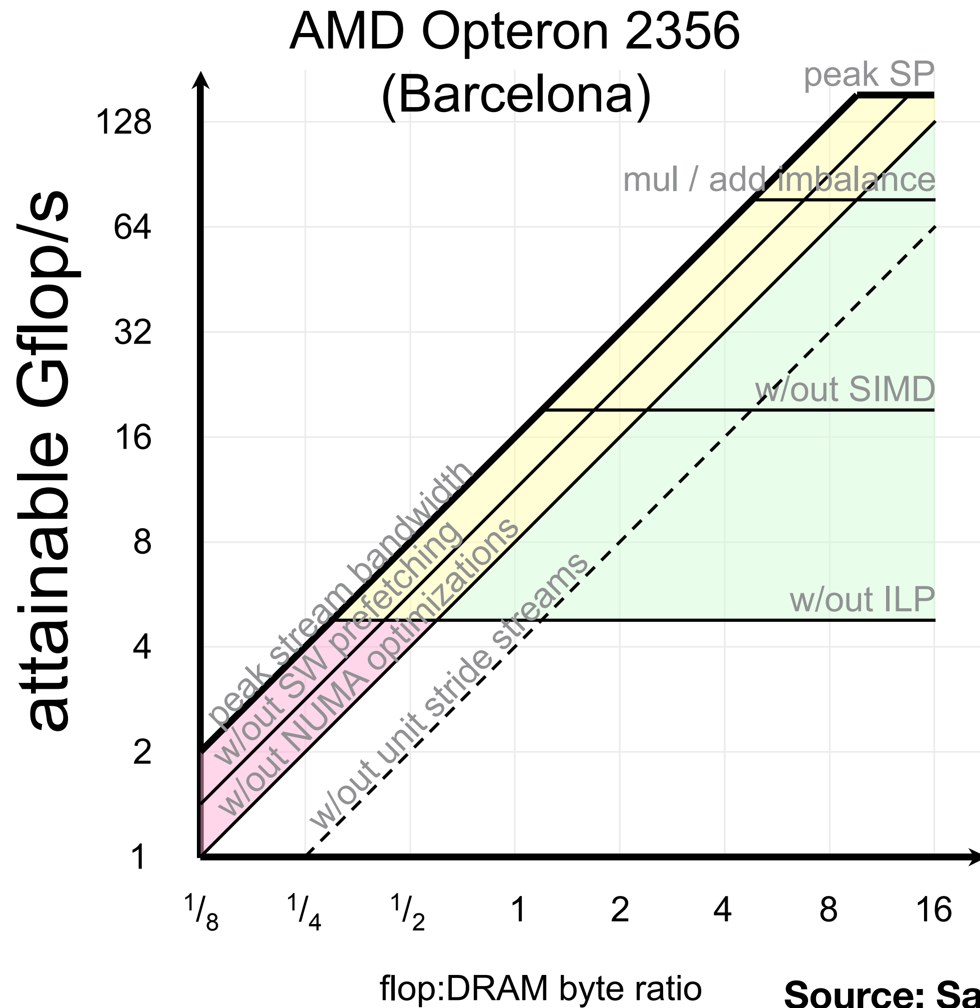
- We encourage the use of self-describing, binary, portable I/O formats
- We encourage users to push the envelope of the I/O Middleware (ADIOS, HDF5, etc.)
- Abstractions should not "force" implementations
  - Use data in streams or files
  - Write data according to the matching of the I/O(??) from the application(s) and the storage layers



(ADIOS tutorial, ORNL)

Performance Modeling

Roofline Model



Source: Sam Williams (LBNL)

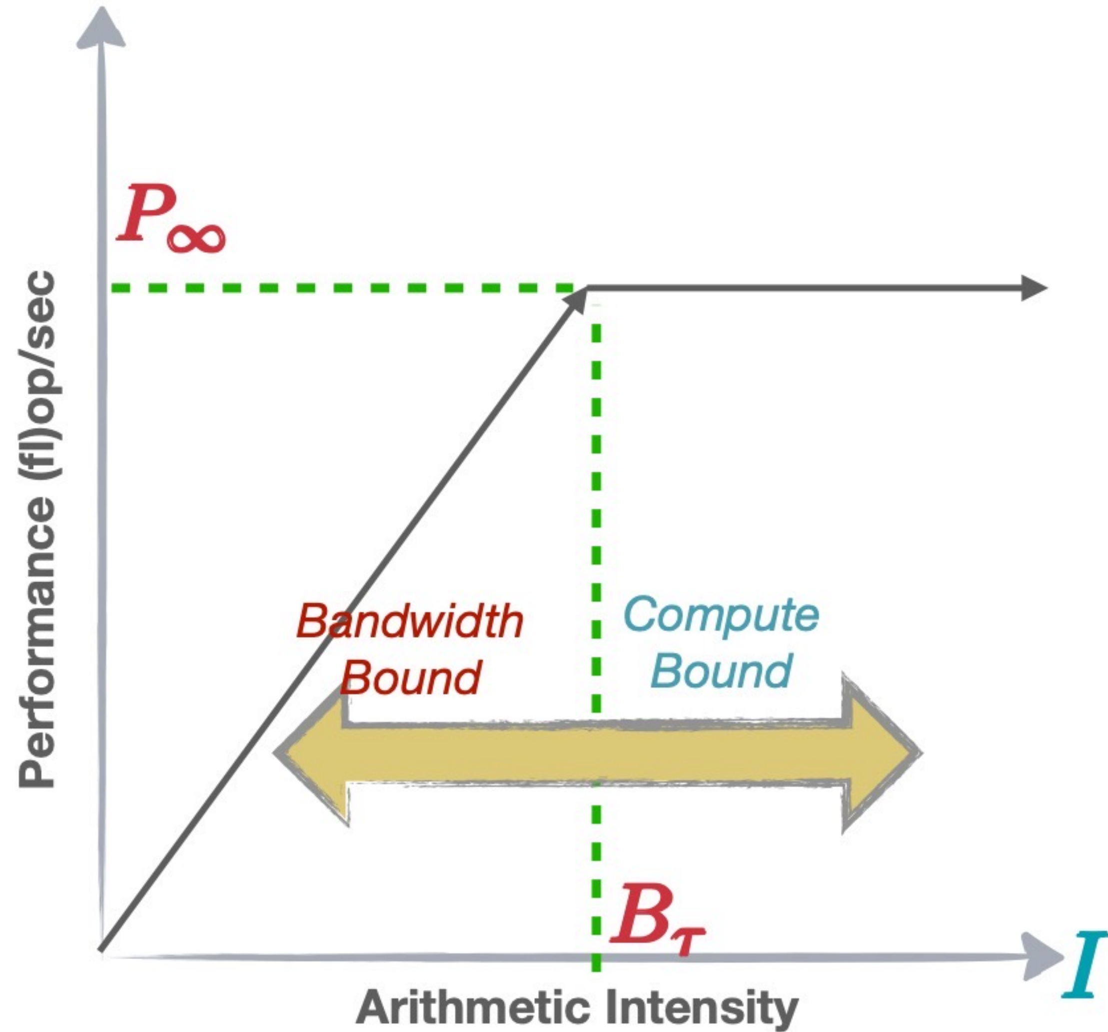
Goals Of Roofline Model

- A graphical aid that provides : realistic expectations of performance and productivity
- Show inherent hardware limitations for a given kernel
- Show potential benefit and priority of optimizations
- Focused on: rates and efficiencies (GF/s, % of peak)

Three Principle Components

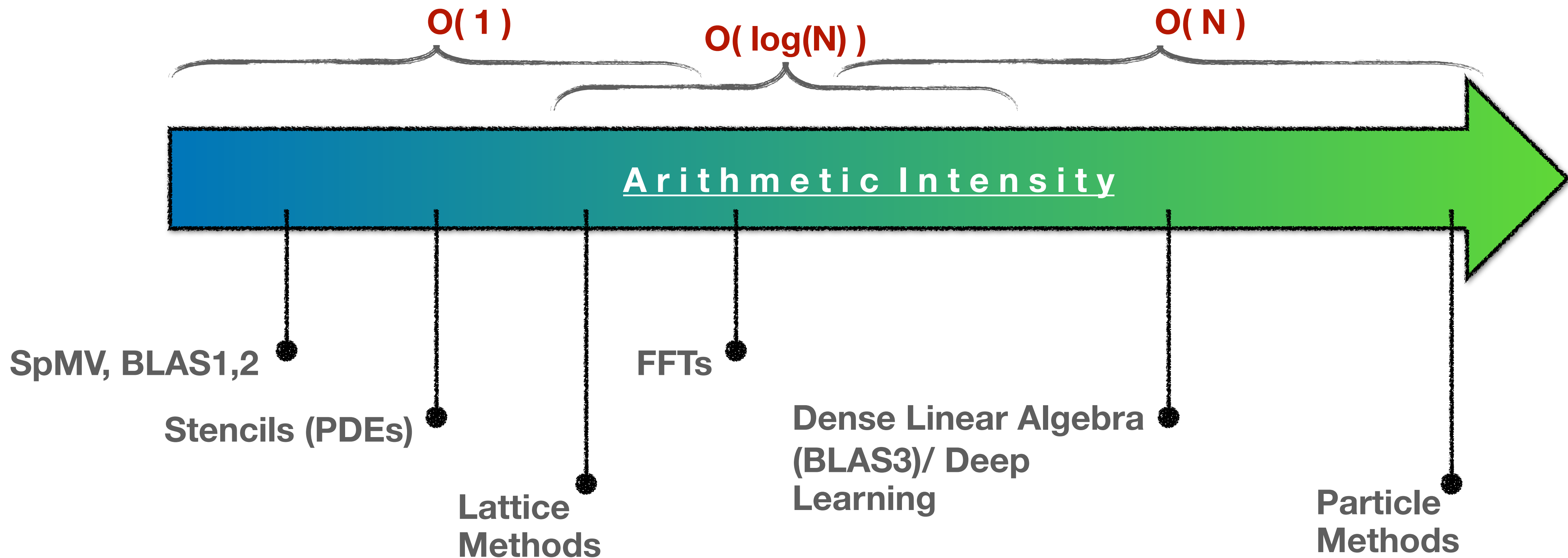
- Computation: measured in GF/sec, GTEPS
- Communication: GB/sec
- Locality: Cache Size, Data Reuse

Roofline Curve



$$
\begin{aligned}
P &= \frac{W}{T} \\
&= \frac{1}{\tau_{fl}} \min\left(\frac{I}{B_\tau},\ 1\right) \\
&= P_\infty \min\left(\frac{I}{B_\tau},\ 1\right) \\
&= \min\left(\beta I,\ P_\infty\right)
\end{aligned}
$$

$\beta \equiv \frac{1}{\tau_{mem}}$ Data Transfer Bandwidth



- True Arithmetic Intensity (AI) ~ **Total Flops / Total DRAM Bytes**
  - constant with respect to problem size for many problems of interest
  - ultimately limited by compulsory traffic
  - diminished by conflict or capacity misses.

```
for (int i=0; i<n; i++) {
    A[i] = B[i] + C[i]
```

$$W = n$$

$$Q = 24n$$

$$I = \frac{1}{24}$$

```
for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
    C[i] += A[i][j]*B[j]
```

$$W = 2n^2$$

$$Q = 8n^2 + 16n$$

$$I = \frac{1}{4}$$

```
for (int k=0; k<n; k++) {
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
        C[i][j] += A[i][k]*B[k][j]
```

$$W = 2n^3$$

$$Q = 24n^2$$

$$I = \frac{n}{2}$$

```
for (int i=0; i<n; i++) {
    A[i] = B[i] + C[i]
```

$$I = \frac{1}{24} \qquad P = \min\left(\frac{900}{24}, 3500\right) = 37.4\text{GF/sec}$$

```
for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
    C[i] += A[i][j]*B[j]
```

$$I = \frac{1}{4} \qquad P = \min\left(\frac{900}{4}, 7000\right) = 225\text{GF/sec}$$

```
for (int k=0; k<n; k++) {
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
        C[i][j] += A[i][k]*B[k][j]
```

$$I = \frac{n}{2} \qquad P = \min\left(450n, 7000\right) = 7000\text{GF/sec}$$

Distributed Complexity Analysis

A simple model for point-to-point messages

The time to send or receive a message of s words is $\alpha + s \cdot \beta$

- α – **latency/synchronization cost** *per message*
- β – **bandwidth cost** *per word*
- *each processor can send and/or receive one message at a time*

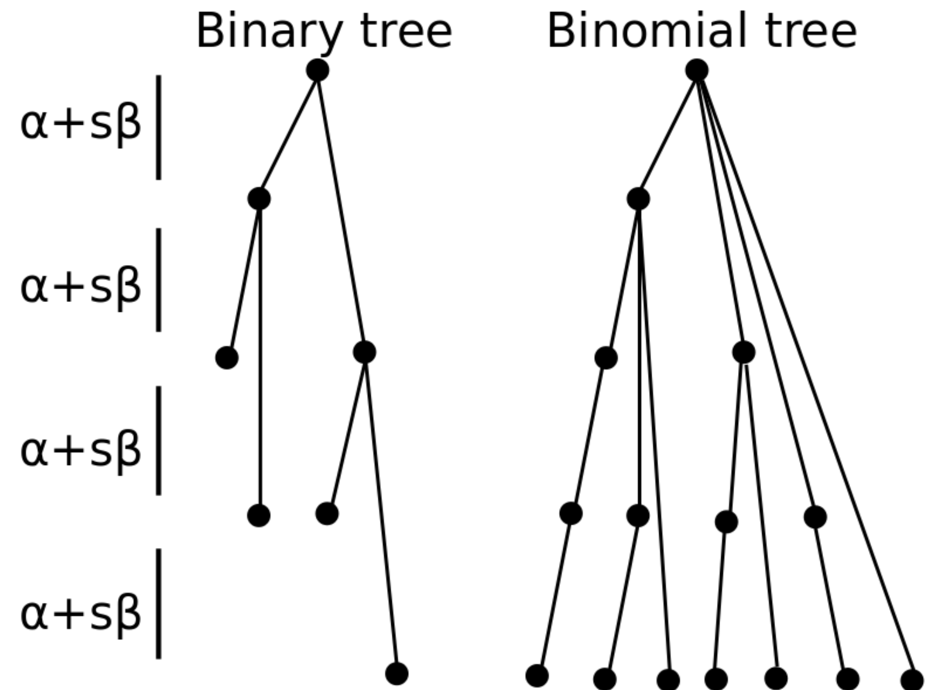
Consider the cost of a broadcast of s words

- *using a* **binary tree** *of height* $h_r = 2(\log_2(p+1) - 1) \approx 2\log_2(p)$

$$T = h_r \cdot (\alpha + s \cdot \beta)$$

- *using a* **binomial tree** *of height* $h_m = \log_2(p+1) \approx \log_2(p)$

$$T = h_m \cdot (\alpha + s \cdot \beta)$$



Collective communication in BSP

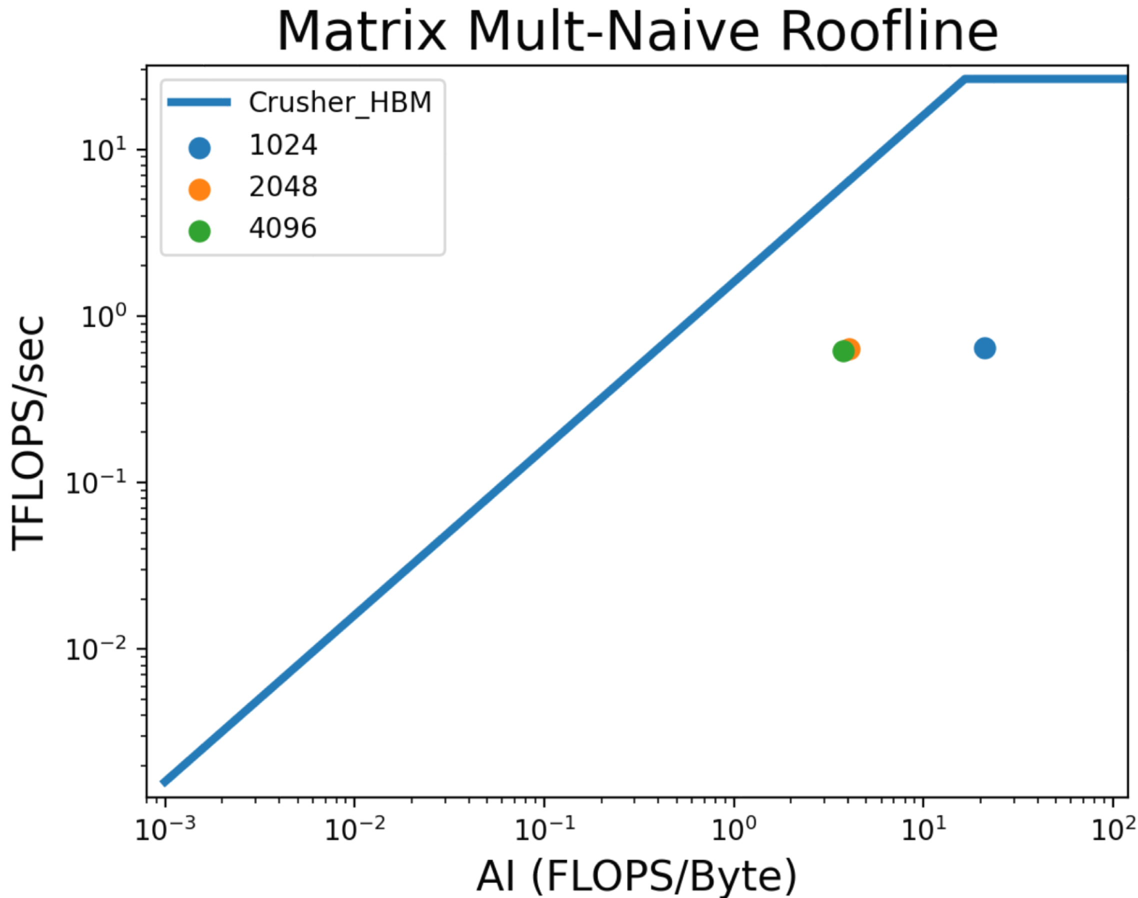
- When $h = p$, most collective communication routines involving s words of data per processor can be done with BSP cost $O(\alpha + s \cdot \beta)$
  - *Scatter: root sends each s/p-sized message to its target (root incurs $s \cdot \beta$ send bandwidth)*
  - *Reduce-Scatter: each processor sends s/p summands to every other processor (every processor incurs $s \cdot \beta$ send and receive bandwidth)*
  - *Gather: send each message of size s/p to root (root incurs $s \cdot \beta$ receive bandwidth)*
  - *Allgather: each processor sends s/p words portion to every other processor (every processor incurs $s \cdot \beta$ send and receive bandwidth)*
  - *Broadcast done by Scatter then Allgather*
  - *Reduce done by Reduce-Scatter then Gather*
  - *Allreduce done by Reduce-Scatter then Allgather*
  - *All-to-all can be done by sending messages directly in one round*
  - *For $h < p$, $O(\log_h p)$ supersteps required, but bandwidth cost same for all except all-to-all (higher by $O(\log_h p)$ via h-ary butterfly protocols*

Profiling and Matrix Multiplication

Ramping up the Flops – matrix multiplication

- Matrix addition: *C = AxB*
  - If A is *n* x *k* and B is *k* x *m*, then:
    - C is *n* x *m*, each position in C requires the sum of *k* multiplications (the dot product of each column of B, row of A)

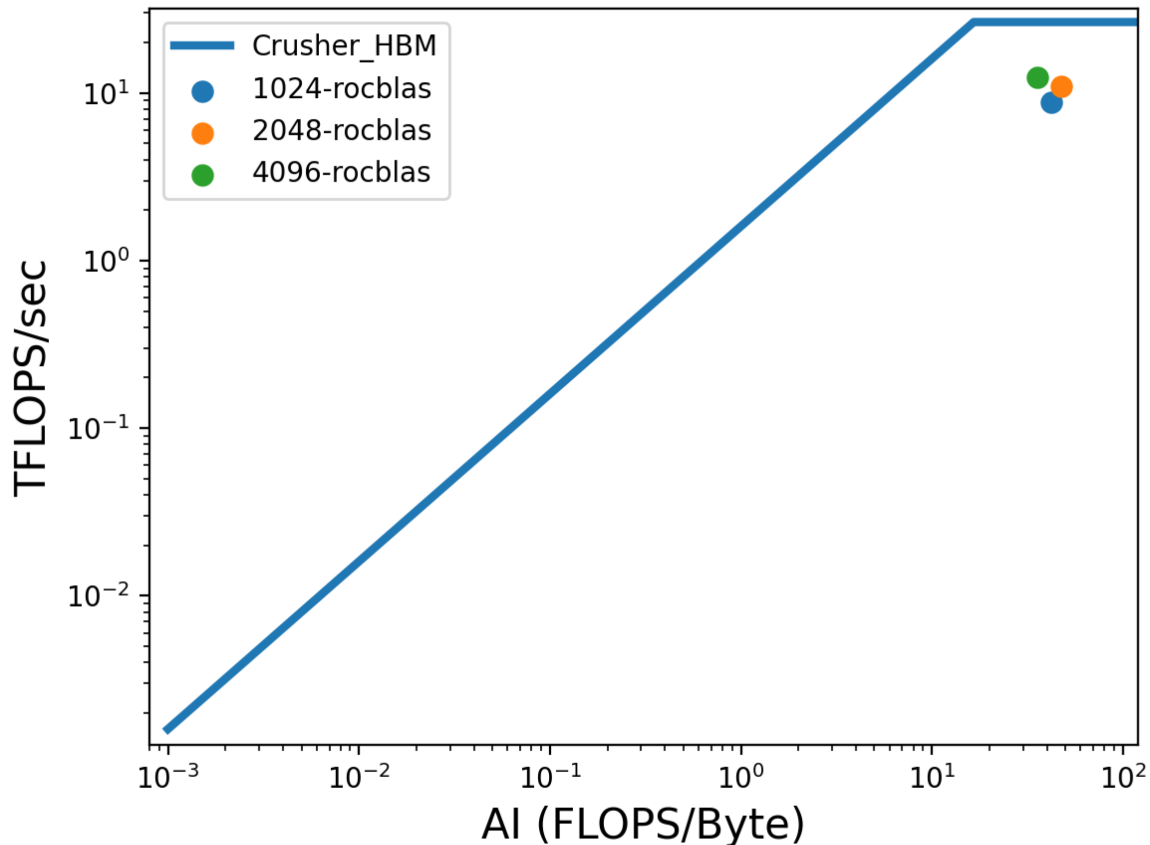
Ramping up the Flops – matrix multiplication



N (N x N matrix)	AI (Flops/Byte)	Performance (GFLOPs)	Roofline Peak (GFLOPs)
1024	21.06	644.63	23900
2048	4.06	631.72	6496
4096	3.78	618.59	6048

- For a matrix multiply, this doesn't look very good – even at larger sizes, achieving <3% of device capability
- Vendors will likely provide libraries for things they want to be highly optimized. Let's try one - **rocBLAS**

Ramping up the Flops – matrix multiplication

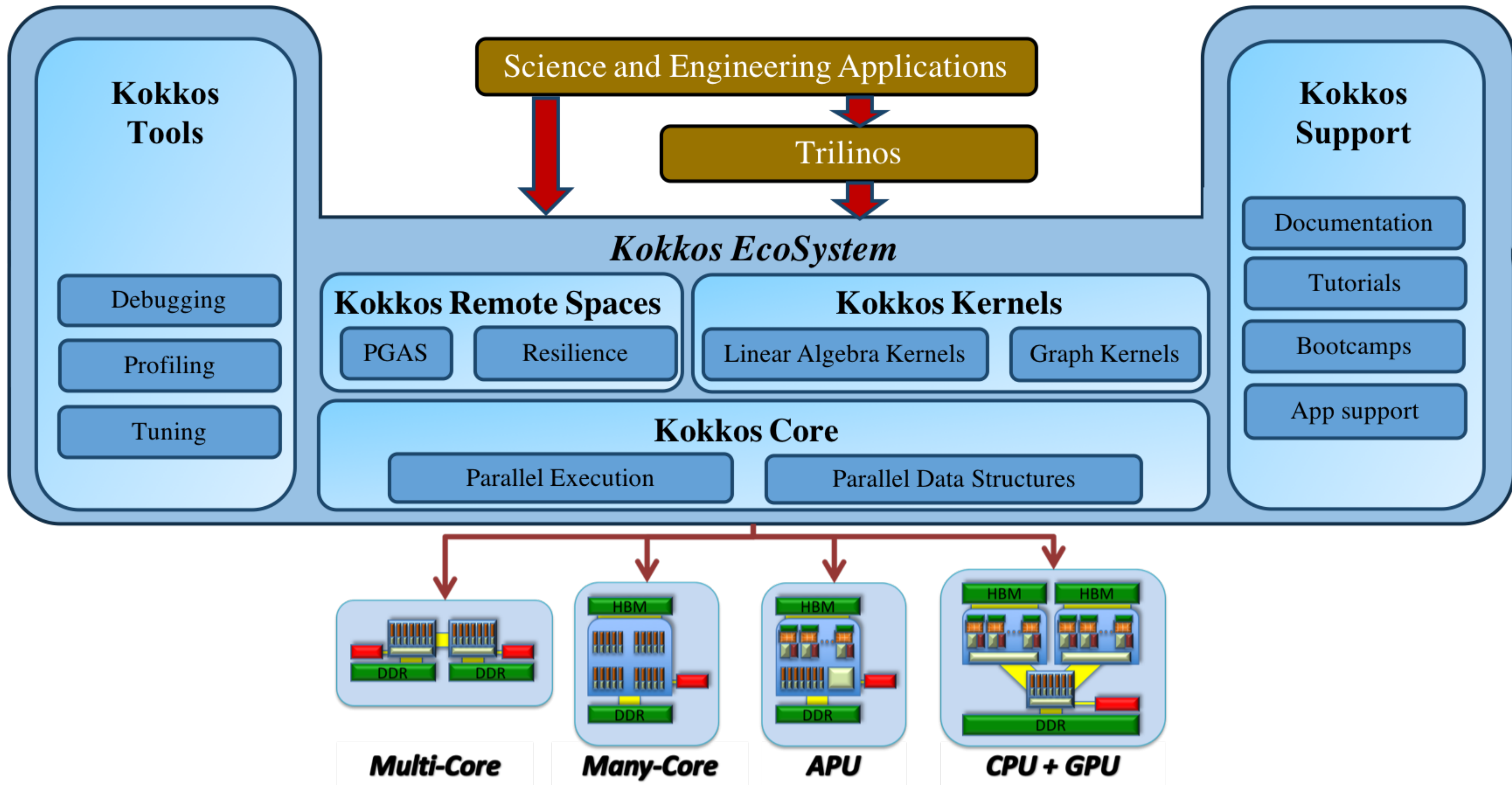


N (N x N matrix)	AI (Flops/Byte)	Performance (GFLOPs)	Theoretical Peak (GFLOPs)
1024	42.00	8822.32	23900
2048	47.44	11017.20	23900
4096	35.77	12337.90	23900

- This looks much better
- >10x floating-point performance
- >50% device peak at largest size

Parallel Frameworks -- Kokkos

The Kokkos EcoSystem



How does this compare to OpenMP?

Serial

```
for (int64_t i = 0; i < N; ++i) {
  /* loop body */
}
```

OpenMP

```
#pragma omp parallel for
for (int64_t i = 0; i < N; ++i) {
  /* loop body */
}
```

Kokkos

```
parallel_for(N, [=] (const int64_t i) {
  /* loop body */
});
```

Important concept

Simple Kokkos usage is **no more conceptually difficult** than OpenMP, the annotations just go in different places.

Example: Scalar integration

OpenMP

```
double totalIntegral = 0;
#pragma omp parallel for reduction(+:totalIntegral)
for (int64_t i = 0; i < numberOfIntervals; ++i) {
  totalIntegral += function(...);
}
```

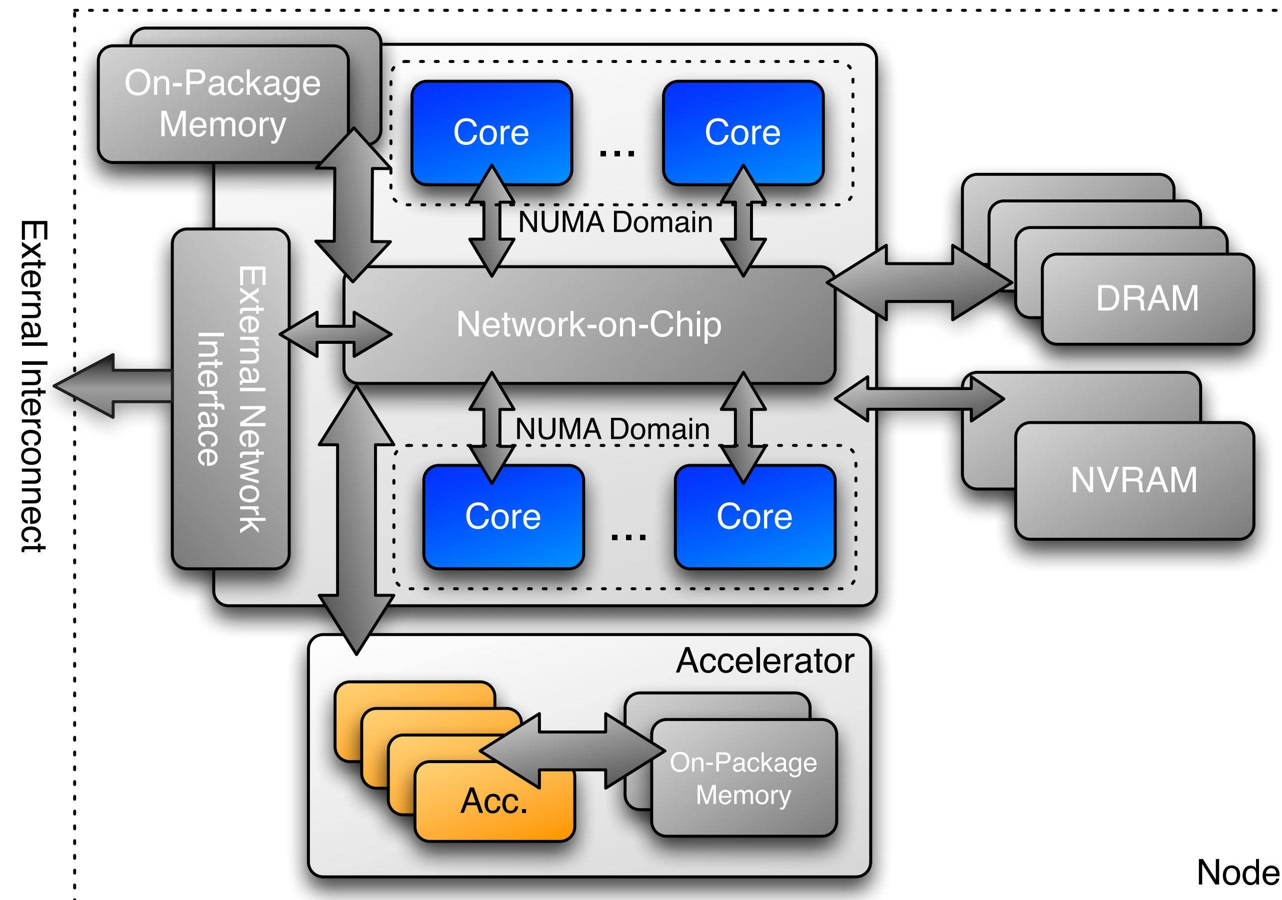
Kokkos

```
double totalIntegral = 0;
parallel_reduce(numberOfIntervals,
  [=] (const int64_t i, double & valueToUpdate) {
    valueToUpdate += function(...);
  },
  totalIntegral);
```

- The operator takes **two arguments**: a work index and a value to update.
- The second argument is a **thread-private value** that is managed by Kokkos; it is not the final reduced value.

Execution Space

a homogeneous set of cores and an execution mechanism
(i.e., "place to run code")



Execution spaces: `Serial`, `Threads`, `OpenMP`, `Cuda`, `HIP`, ...

Memory space:

explicitly-manageable memory resource

(i.e., “place to put data”)