Advanced Topics in Numerical Analysis: High Performance Computing

Cross-listed as MATH-GA.2012-001 and CSCI-GA 2945.001

Benjamin Peherstorfer
Courant Institute, NYU
pehersto@cims.nyu.edu

Spring 2020, Monday, 1:25–3:15PM, WWH #512

April 13, 2020

Slightly adapted from Georg Stadler's lectures.

Organization

Last time

- GPU: global vs. shared memory

Today

- GPU: Reduction and convolution
- Start with MPI

Announcements

- The homework sheet has been updated: When you submit your homework next week, add a few sentences what the status of your final project is

Review of Last Class

CPU

- fewer cores $\sim O(10)$
- smaller vector lengths (2-SSE, 4-AVX)
- large amounts of cache (L1, L2, L3)

GPU

- many cores $\sim O(10^5)$
- wider vector lengths (warp size $= 32$)
- smaller cache (L1/programmer managed buffer, L2 on more recent GPUs)

Review of Last Class

Computing on GPU:

- Device (GPU) is slave to the host (CPU).
- Workflow for computing on GPU:
  1. copy data from host to device,
  2. launch GPU kernel to compute result on device,
  3. copy result from device to host.
- Host - device interconnect (PCI Express or NVLink*) is slow (O(10-50) GB/s).
- Computing on GPUs useful only for compute-intensive tasks (when overhead of data transfer is small compared to compute time).

*Proprietary Nvidia system

Review of Last Class

GPU Architecture: consists of,

- main memory (typically DRAM).
- PCI Express or other interconnection to host (CPU) DRAM and other GPUs.
- several Streaming Multiprocessors (SM)
  - with shared L2 cache.

Each Streaming Multiprocessor,

- 32 scalar double-precision cores (each executing the same instruction, Single Instruction Multiple Data).
- can execute up to 1024 scalar threads (or 32-warps) simultaneously (pipelined, hyper-threading to hide instruction latency).
- L1 cache / shared-memory: shared by all threads in the thread-block.

Memory Hierarchy

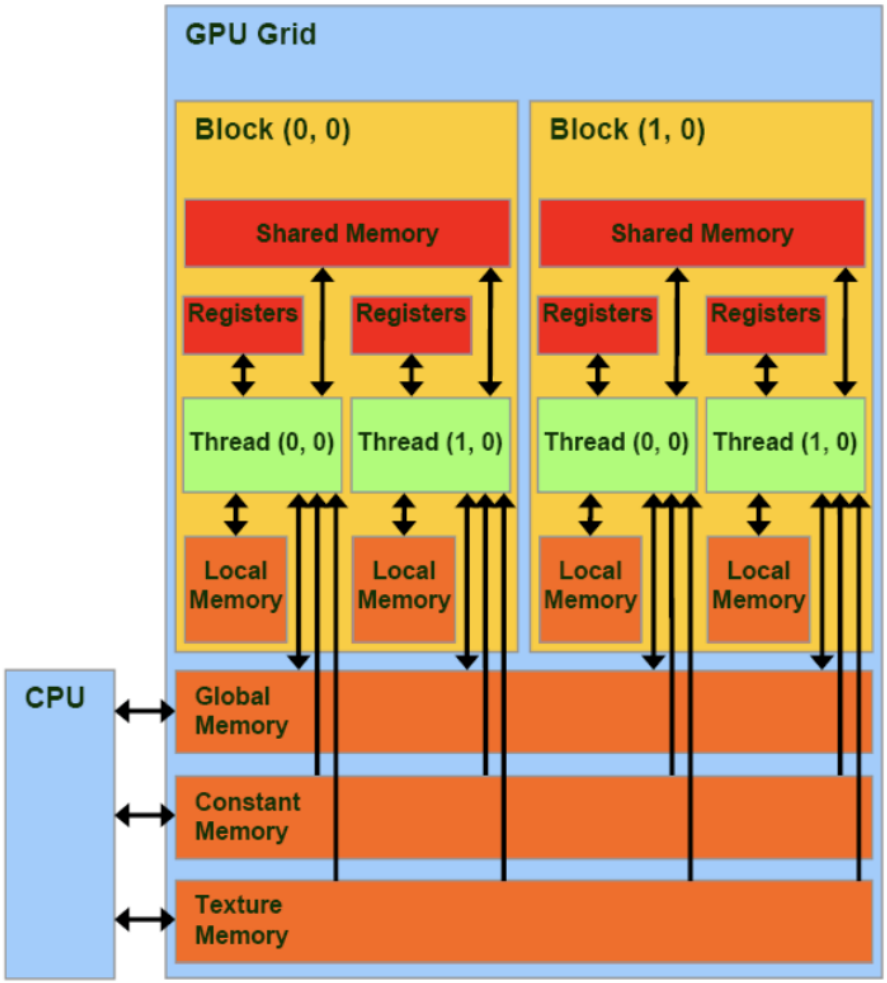
Local memory: Local to each thread (large arrays or register spilling), reside in GPU main-memory.

Shared memory: accessible by all threads in the same thread-block. Eg: `__shared__ A[100];`

Global memory: accessible by all threads, resides in DRAM. Eg: `cudaMalloc(&A,10*sizeof(double))`

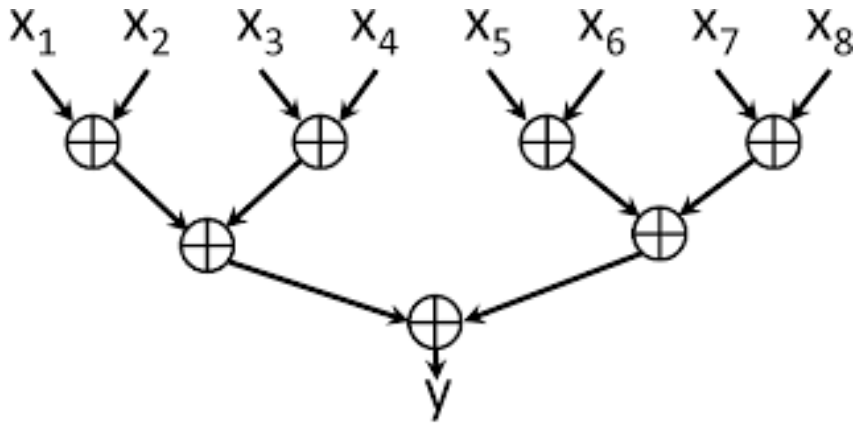
Constant memory: read-only, resides in DRAM and constant cache.

Texture memory: read-only, resides in DRAM and texture cached.



Reduction

Parallel reduction algorithm:



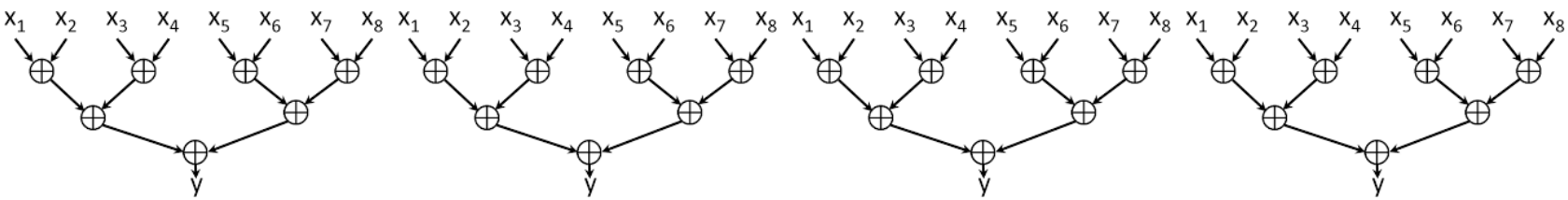
- $\log_2 N$ stages
- Requires coordination between threads!
- In each stage, every thread:
  - reads two numbers,
  - computes the sum and
  - writes its result to memory

Reduction

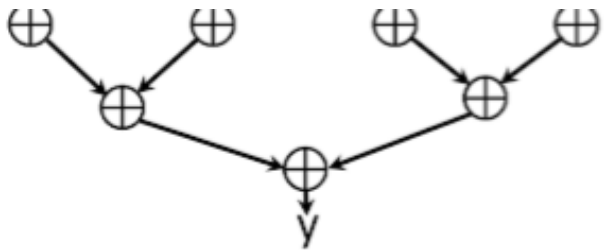
GPU algorithm

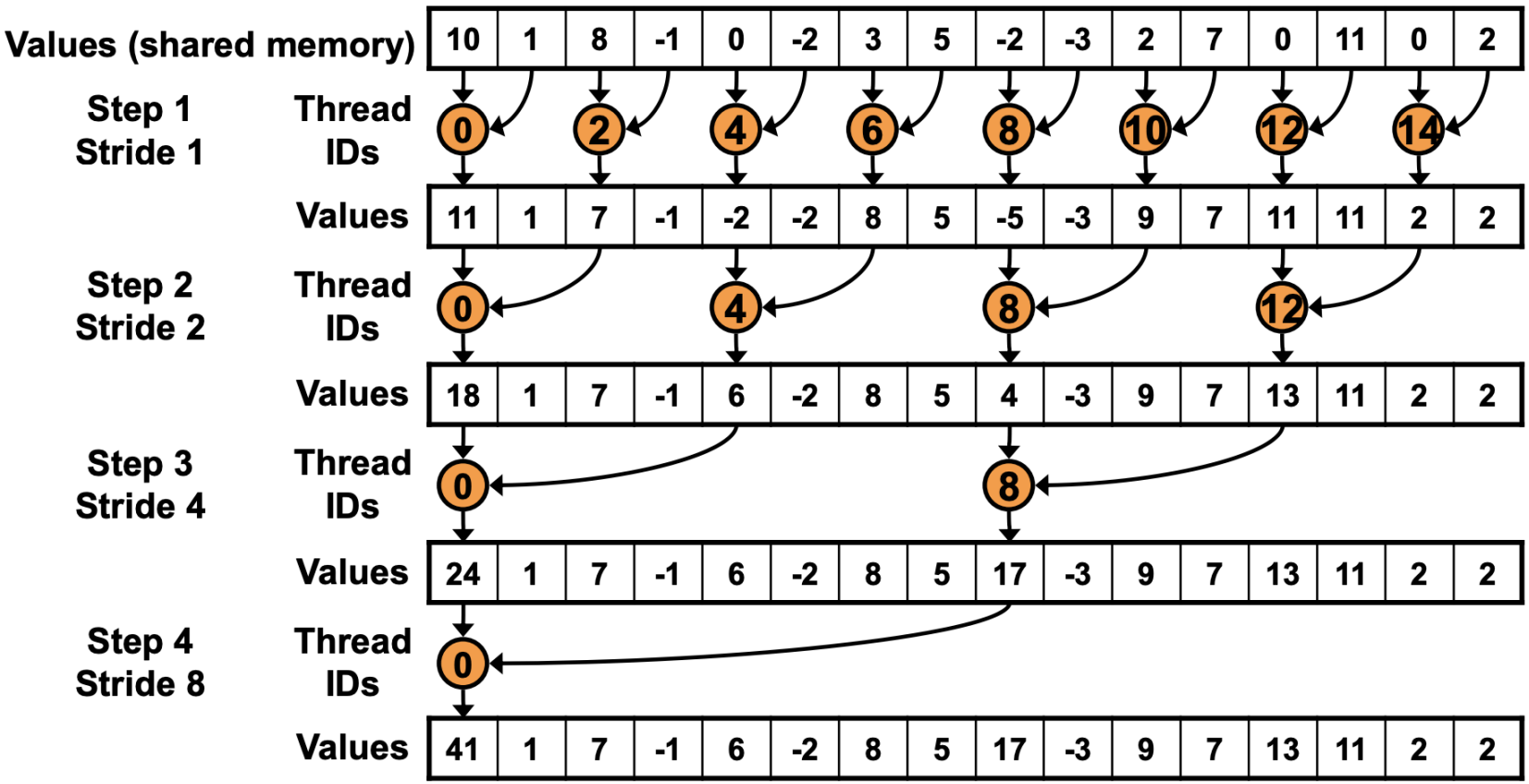
- Compute local reduction within each thread block
- Cannot sync between blocks, therefore multiple kernel invocations for global synchronization

Stage 1: kernel with 4 thread-blocks, 8 threads per thread-block



Stage 2: launch kernel with 1 thread-block with 4-threads





https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

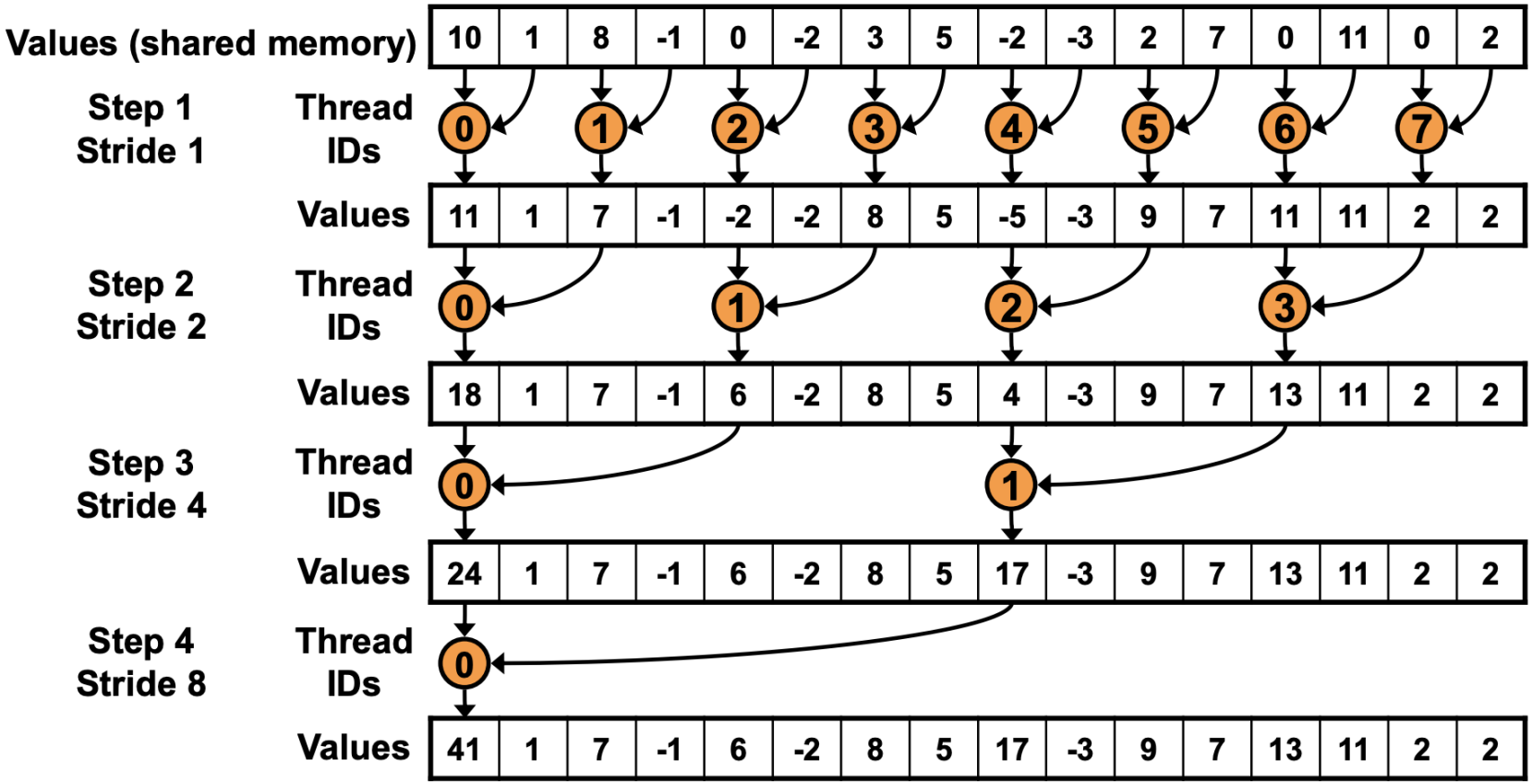
```
__global__ void reduction_kernel0(double* sum, const double* a,
    long N){
  __shared__ double smem[BLOCK_SIZE];
  int idx = (blockIdx.x) * blockDim.x + threadIdx.x;

  // each thread reads data from global into shared memory
  if (idx < N) smem[threadIdx.x] = a[idx];
  else smem[threadIdx.x] = 0;
  __syncthreads();

  for(int s = 1; s < blockDim.x; s *= 2) {
          if(threadIdx.x % (2*s) == 0)
                  smem[threadIdx.x] += smem[threadIdx.x + s];
          __syncthreads();
  }

  // write to global memory
  if (threadIdx.x == 0) sum[blockIdx.x] = smem[threadIdx.x];
}
```

DEMO: gpu14.cu

```
__global__ void reduction_kernel0(double* sum, const double* a,
    long N){
  __shared__ double smem[BLOCK_SIZE];
  int idx = (blockIdx.x) * blockDim.x + threadIdx.x;

  // each thread reads data from global into shared memory
  if (idx < N) smem[threadIdx.x] = a[idx];
  else smem[threadIdx.x] = 0;
  __syncthreads();

  for(int s = 1; s < blockDim.x; s *= 2) {
          if(threadIdx.x % (2*s) == 0)
                  smem[threadIdx.x] += smem[threadIdx.x + s];
          __syncthreads();
  }

  // write to global memory
  if (threadIdx.x == 0) sum[blockIdx.x] = smem[threadIdx.x];
}
```

DEMO: gpu14.cu

Problems: Warp divergence because every other thread is idle



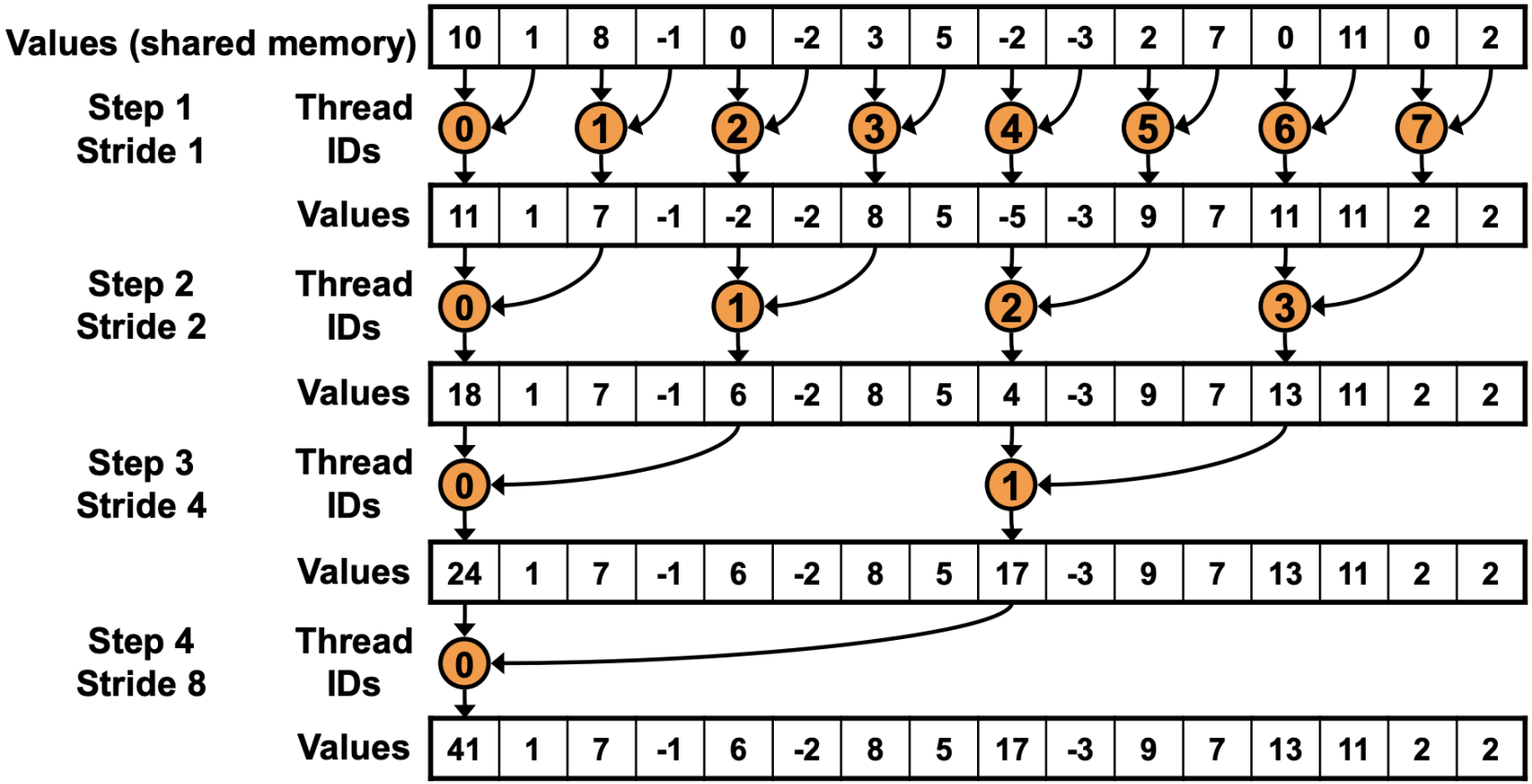
https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

DEMO: gpu15.cu

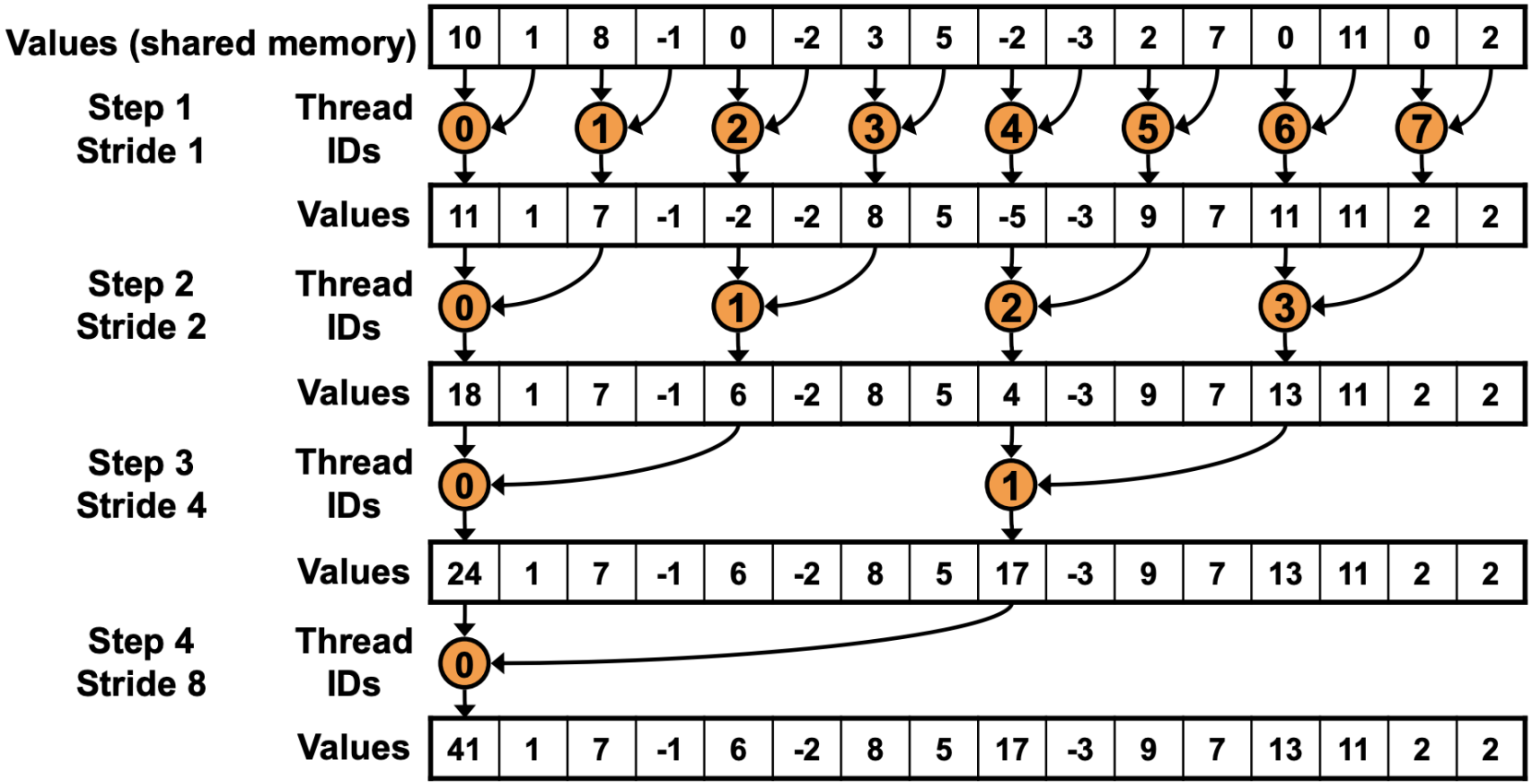
Avoid warp divergence by changing indexing

```
for(int s = 1; s < blockDim.x; s *= 2) {
    int index = 2*s*threadIdx.x;
    if(index < blockDim.x)
       smem[index] += smem[index + s];
    __syncthreads();
  }
```

- Only the first threads (within a warp) active



https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

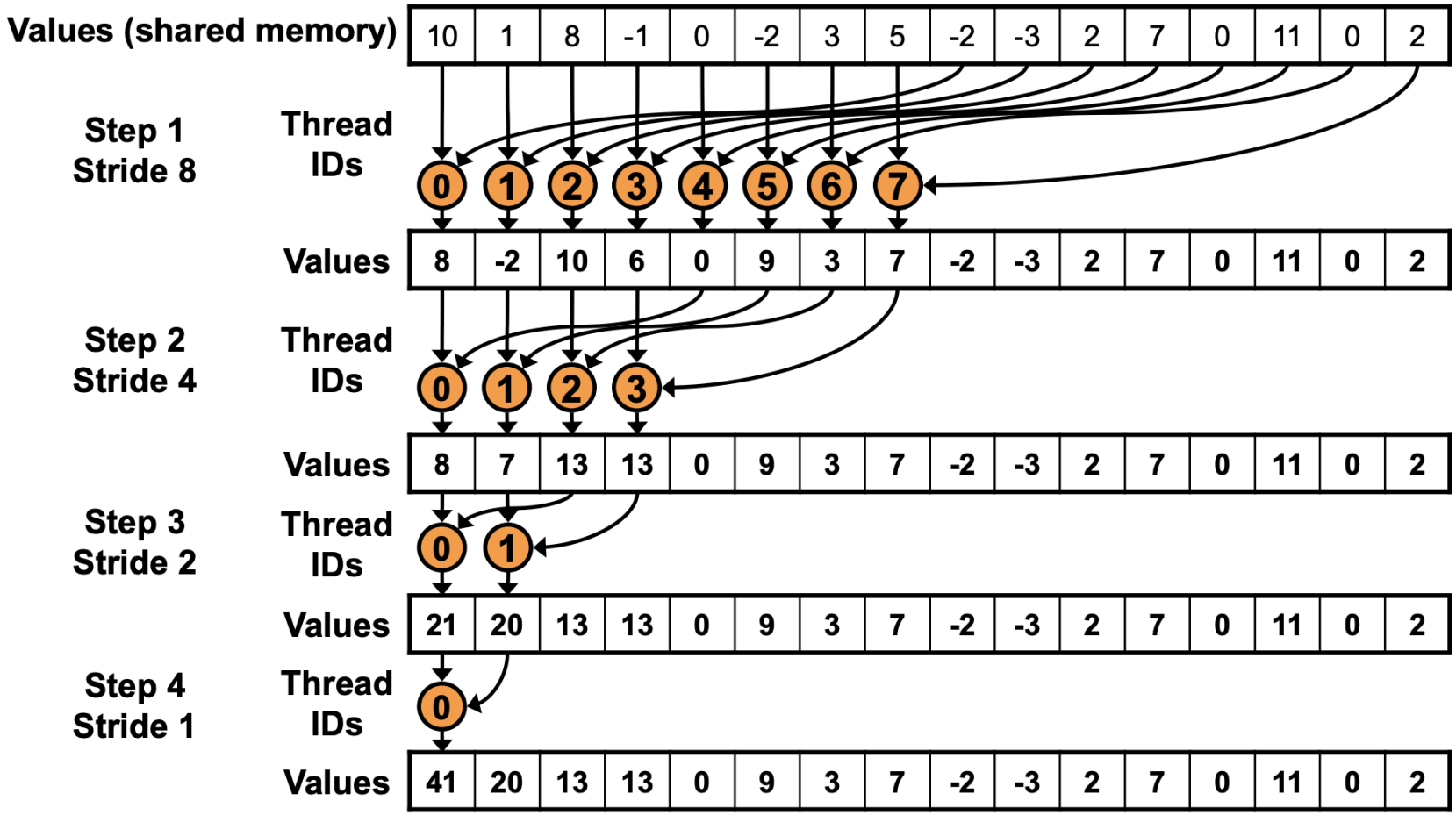


Problems: Bank conflicts because read/write to addresses of multiples of 2

- Consider for example a warp size of 4 in this example
- At step 2, read addresses 0, 4, 8, 12, which are one bank

https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Reduction (within thread block)



Avoid warp divergence and bank conflicts

```
// x >>= 1 means "set x to itself shifted by one bit to the right"
  for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (threadIdx.x < s) {
                smem[threadIdx.x] += smem[threadIdx.x + s];
        }
        __syncthreads();
   }
```

- Replace strided indexing in inner loop with reversed loop and threadID-based indexing

Further optimization opportunities https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Image Filtering

Convolution of two signals f and g is

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\mathrm{d}\tau$$

- $(f * g)(t)$ is the filtered variant of $f(t)$
- Kernel is g

If an image is represented as a 2D signal $y[,]$ then

$$(y * k)[i,j] = \sum_n \sum_m y[i-n, j-m]k[n,m]$$

- Kernel k and filtered variant $(y * k)$ of y
- Kernel has a certain *radius* (support) that defines the scope of its action

Convolution: read $k \times k$ block of the image multiply by filter weights and sum.

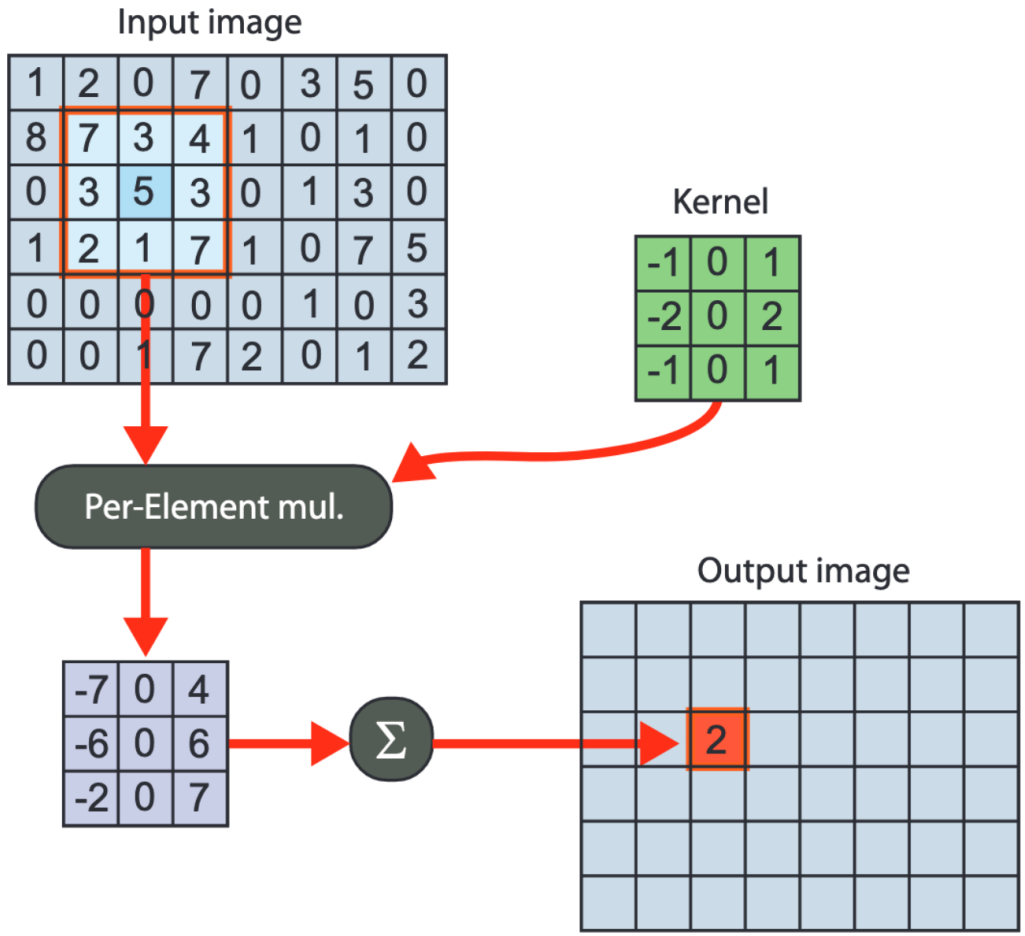


figure from: GPU Computing: Image Convolution - Jan Novák, Gábor Liktor, Carsten Dachsbacher

DEMO: filter.cu

```
struct RGBImage {
  long Xsize;
  long Ysize;
  float* A;
};
void read_image(const char* fname, RGBImage* I) {
...
}
```

- Read image from ppm file (3 colors red, green, blue)
- Store in array A of struct RGBImage

Apply filter to each of the 3 colors

```
// 45 degree motion blur
float filter[FWIDTH][FWIDTH] = {
     0,      0,      0,      0,      0, 0.0145,      0,
     0,      0,      0,      0, 0.0376, 0.1283, 0.0145,
     0,      0,      0, 0.0376, 0.1283, 0.0376,      0,
     0,      0, 0.0376, 0.1283, 0.0376,      0,      0,
     0, 0.0376, 0.1283, 0.0376,      0,      0,      0,
0.0145, 0.1283, 0.0376,      0,      0,      0,      0,
     0, 0.0145,      0,      0,      0,      0,      0};
```

Convolution on CPU

```
void CPU_convolution(float* I, const float* I0, long Xsize, long
    Ysize) {
  constexpr long FWIDTH_HALF = (FWIDTH-1)/2;
  long N = Xsize * Ysize;
  #pragma omp parallel for collapse(3) schedule(static)
  for (long k = 0; k < 3; k++) { // loop over colors
    for (long i0 = 0; i0 <= Xsize-FWIDTH; i0++) { // two loops for
        selecting center
      for (long i1 = 0; i1 <= Ysize-FWIDTH; i1++) {
        float sum = 0;
        for (long j0 = 0; j0 < FWIDTH; j0++) { // two nested loops
            for conv
          for (long j1 = 0; j1 < FWIDTH; j1++) {
            sum += I0[k*N + (i0+j0)*Ysize + (i1+j1)] *
                filter[j0][j1];
          }
        }
        I[k*N + (i0+FWIDTH_HALF)*Ysize + (i1+FWIDTH_HALF)] =
            (float)fabs(sum);
      }
    }
  }
}
```

Convolution on GPU

- Each thread corresponds to one center pixel
- Each thread loops kernel radius and applies filter
- Writes applied filter back into memory

We make use of constant memory to store filter

```
__constant__ float filter_gpu[FWIDTH][FWIDTH];
...
int main() {
   cudaMemcpyToSymbol(filter_gpu, filter, sizeof(filter_gpu)); //
        Initialize filter_gpu
}
```

We use multiple streams to work through 3 colors in parallel

```
  cudaStream_t streams[3];
  cudaStreamCreate(&streams[0]);
  cudaStreamCreate(&streams[1]);
  cudaStreamCreate(&streams[2]);
  dim3 blockDim(BLOCK_DIM, BLOCK_DIM);
  dim3 gridDim(Xsize/(BLOCK_DIM-FWIDTH)+1,
       Ysize/(BLOCK_DIM-FWIDTH)+1);
  GPU_convolution<<<gridDim,blockDim, 0,
       streams[0]>>>(I1gpu+0*Xsize*Ysize, I0gpu+0*Xsize*Ysize,
       Xsize, Ysize);
  GPU_convolution<<<gridDim,blockDim, 0,
       streams[1]>>>(I1gpu+1*Xsize*Ysize, I0gpu+1*Xsize*Ysize,
       Xsize, Ysize);
  GPU_convolution<<<gridDim,blockDim, 0,
       streams[2]>>>(I1gpu+2*Xsize*Ysize, I0gpu+2*Xsize*Ysize,
       Xsize, Ysize);
```

DEMO: filter.cu

Speedup of about 2x in terms of GFlop/s

```
$ ./filter
CPU time = 5.774944s
CPU flops = 2.051868GFlop/s
GPU time = 2.922829s
GPU flops = 4.054095GFlop/s
Error = 7.629395e-05
```

- Why do we see an error?

DEMO: filter.cu

Speedup of about 2x in terms of GFlop/s

```
$ ./filter
CPU time = 5.774944s
CPU flops = 2.051868GFlop/s
GPU time = 2.922829s
GPU flops = 4.054095GFlop/s
Error = 7.629395e-05
```

- Why do we see an error? because we compute with single precision
- Are we happy with 2x increase in FLOP/s?

DEMO: filter.cu

Speedup of about 2x in terms of GFlop/s

```
$ ./filter
CPU time = 5.774944s
CPU flops = 2.051868GFlop/s
GPU time = 2.922829s
GPU flops = 4.054095GFlop/s
Error = 7.629395e-05
```

- Why do we see an error? because we compute with single precision
- Are we happy with 2x increase in FLOP/s? what about memory access?

Image Filtering

Using shared memory as cache to minimize global memory reads

- read a 32×32 block of original image from main memory
- compute convolution in shared memory
- write back result sub-block (excluding halo)

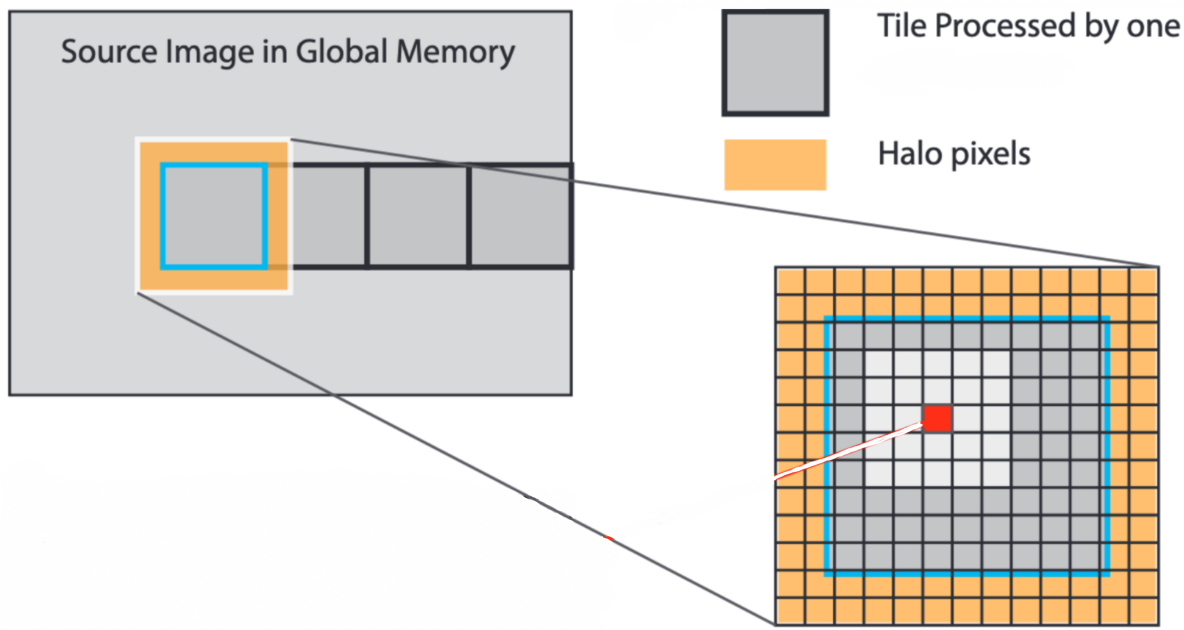


figure from: GPU Computing: Image Convolution - Jan Novák, Gábor Liktor, Carsten Dachsbacher

DEMO: filterSMEM.cu

- Implementation with shared memory
- Read tile, then sync, then apply filter and write back

Libraries

Optimized libraries for

- **cuBLAS** for linear algebra
- **cuFFT** for Fast Fourier Transform
- **cuDNN** for Deep Neural Networks

cuBLAS Demo!

References

NVIDIA Programming Guide

- Programming Guide
- Runtime API
- Best practices

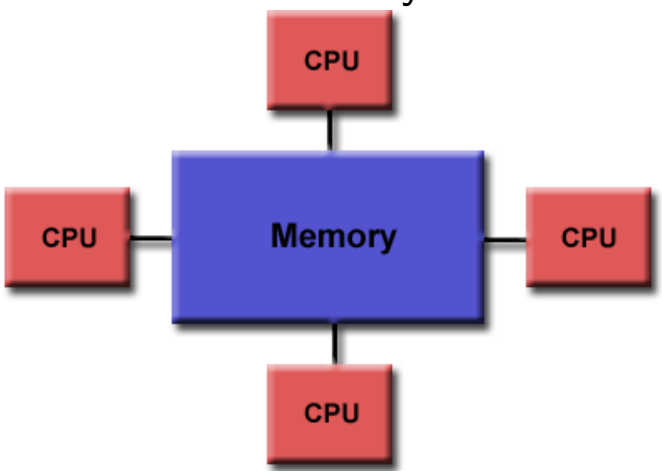
Quick References

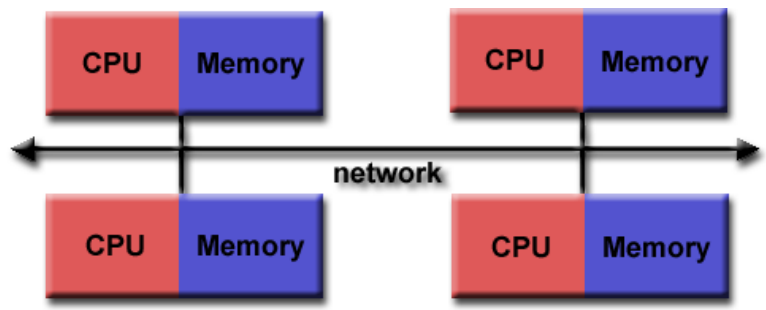
- CUDA cheatsheet
- CUDA syntax

Programming models overview

Programming models

- Flynn's taxonomy:
  - Single instruction–single data (SISD)
  - Single instruction–multiple data (SIMD)
  - Multiple instruction–multiple data (MIMD)
- Distributed memory vs. shared memory parallelism





- Programming models: OpenMP vs. Message passing interface (MPI); and combinations thereof

Process vs. thread

- A process is an independent execution unit, which contains their own state information (pointers to instruction and stack). One process can contain several threads.
- Threads within a process share the same address space, and communicate directly using shared variables. Seperate stack but shared heap memory.

Parallelism and locality

- Moving data (through network or memory hierarchy) is slow
- Real world problems often have parallelism and locality, e.g.,
  - objects move independently from each other ("embarrassingly parallel")
  - objects mostly influence other objects nearby
  - dependence on distant objects can be simplified
  - Partial differential equations have locality properties
- Applications often exhibit parallelism at multiple levels

MPI

Principles of Message-Passing Programming

- The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
- Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
- All interactions (read-only or read/write) require cooperation of two processes – the process that has the data and the process that wants to access the data.
- These two constraints, while onerous, make underlying costs very explicit to the programmer.

The Building Blocks: Send and Receive Operations

- The prototypes of these operations are as follows:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

- Consider the following code segments:

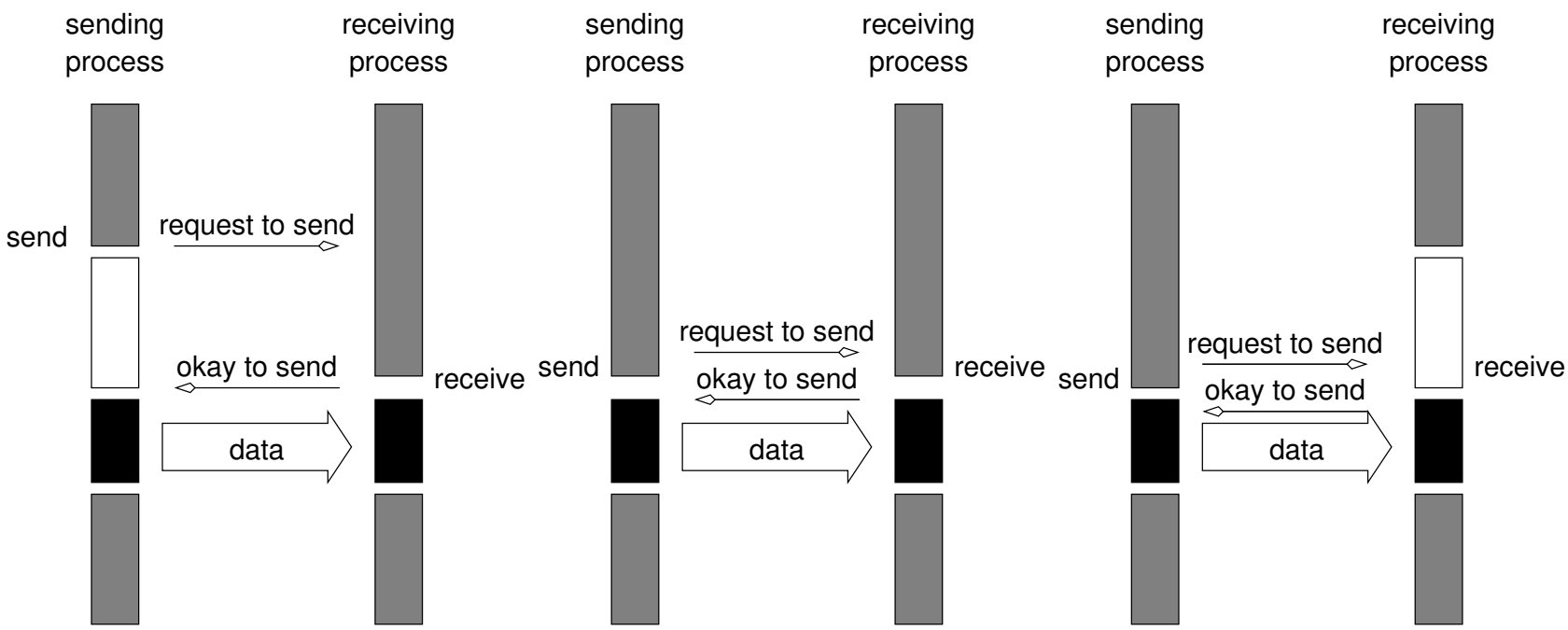
```
P0                              P1

a = 100;                        receive(&a, 1, 0)
send(&a, 1, 1);                 printf("%d\n", a);
a = 0;
```

- The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0.
- This motivates the design of the send and receive protocols.

Non-Buffered Blocking Message Passing Operations

- A simple method for forcing send/receive semantics is for the send operation to return only when it is safe to do so.
- In the non-buffered blocking send, the operation does not return until the matching receive has been encountered at the receiving process.
- Idling and deadlocks are major issues with non-buffered blocking sends.
- In buffered blocking sends, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The data is copied at a buffer at the receiving end as well.
- Buffering alleviates idling at the expense of copying overheads.

Non-Buffered Blocking Message Passing Operations



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

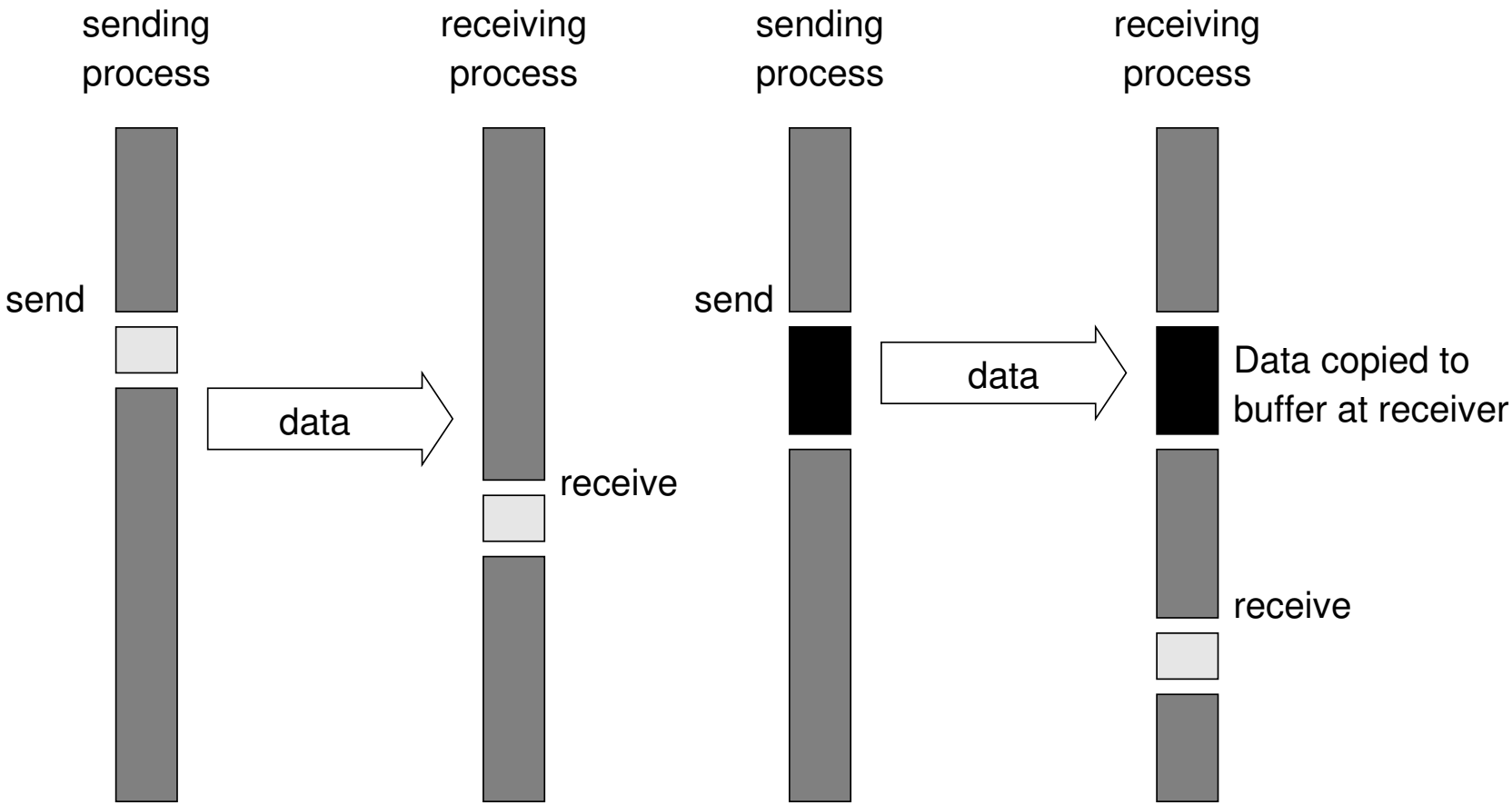
(c) Receiver comes first; idling at receiver

Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

Buffered Blocking Message Passing Operations

- A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends.
- The sender simply copies the data into the designated buffer and returns after the copy operation has been completed.
- The data must be buffered at the receiving end as well.
- Buffering trades off idling overhead for buffer copying overhead.

Buffered Blocking Message Passing Operations



Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Buffered Blocking Message Passing Operations

Bounded buffer sizes can have significant impact on performance.

```
P0

for (i = 0; i < 1000; i++) {
  produce_data(&a);
  send(&a, 1, 1);
}
```

```
P1

for (i = 0; i < 1000; i++) {
  receive(&a, 1, 0);
  consume_data(&a);
}
```

What if consumer was much slower than producer?

Buffered Blocking Message Passing Operations

Bounded buffer sizes can have significant impact on performance.

```
P0                                  P1

for (i = 0; i < 1000; i++) {        for (i = 0; i < 1000; i++) {
  produce_data(&a);                   receive(&a, 1, 0);
  send(&a, 1, 1);                     consume_data(&a);
}                                   }
```

What if consumer was much slower than producer?

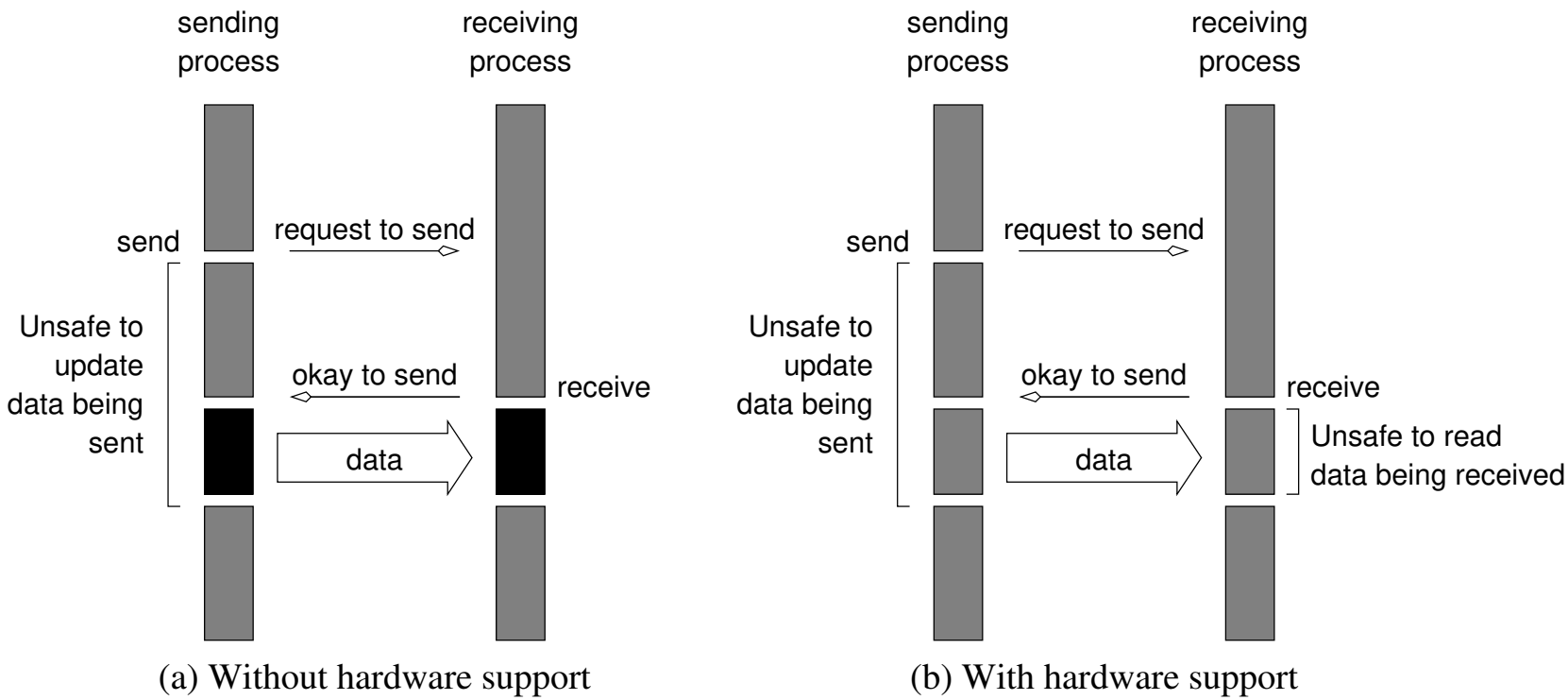
Deadlocks are still possible with buffering since receive operations block.

```
    P0                          P1

    receive(&a, 1, 1);          receive(&a, 1, 0);
    send(&b, 1, 1);             send(&b, 1, 0);
```

Non-Blocking Message Passing Operations

- The programmer must ensure semantics of the send and receive.
- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.
- Non-blocking operations are generally accompanied by a `check-status` operation.
- When used correctly, these primitives are capable of overlapping communication overheads with useful computations.
- Message passing libraries typically provide both blocking and non-blocking primitives.

Non-Blocking Message Passing Operations



(a) Without hardware support

(b) With hardware support

Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

MPI: the Message Passing Interface

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran.
- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.
- Vendor implementations of MPI are available on almost all commercial parallel computers.
- It is possible to write fully-functional message-passing programs by using only the six routines.

MPI: the Message Passing Interface

The minimal set of MPI routines.

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of the calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

Starting and Terminating the MPI Library

- `MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- The prototypes of these two functions are:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

- `MPI_Init` also strips off any MPI related command-line arguments.
- All MPI routines, data-types, and constants are prefixed by "`MPI_`". The return code for successful completion is `MPI_SUCCESS`.

Communicators

- A communicator defines a *communication domain* – a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type `MPI_Comm`.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.
- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.

Querying

- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively.
- The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

Our First MPI Program

```
#include <mpi.h>

main(int argc, char *argv[])
{
  int npes, myrank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  printf("From process %d out of %d, Hello World!\n",
         myrank, npes);
  MPI_Finalize();
}
```

Running MPI program: In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.

```
$ mpic++ -o mpi01 mpi01.cpp
$ mpirun -np 3 mpi01
```

There are several MPI implementations. We will use OpenMPI.
DEMO: mpi01.cpp

Sending and Receiving Messages

- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`, respectively.

- The calling sequences of these routines are as follows:

```
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 provides equivalent datatypes for all C datatypes. This is done for portability reasons.

- The message-tag can take values ranging from zero up to the MPI defined constant `MPI_TAG_UB`.

MPI Datatypes

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

Sending and Receiving Messages

- MPI allows specification of wildcard arguments for both `source` and `tag`.
- If `source` is set to `MPI_ANY_SOURCE`, then any process of the communication domain can be the source of the message.
- If `tag` is set to `MPI_ANY_TAG`, then messages with any tag are accepted.
- On the receive side, the message must be of length equal to or less than the length field specified.
- On the receiving end, the `status` variable can be used to get information about the `MPI_Recv` operation.
- The corresponding data structure contains:

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
};
```

- The `MPI_Get_count` function returns the precise count of data items received.

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
        int *count)
```

DEMO: mpi02.cpp

```
MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    int message_out = 42;
    // msg, length, type, destination rank, tag, communicator
    MPI_Send(&message_out, 1, MPI_INT, 2, 999, MPI_COMM_WORLD);
  } else if (rank == 2) {
    int message_in;
    MPI_Status status;
    // msg, length, type, source rank, tag, communicator, status
    MPI_Recv(&message_in, 1, MPI_INT, 0, 999, MPI_COMM_WORLD,
        &status);

    printf("The message is %d\n", message_in);
  }

  MPI_Finalize();
```

- Send one integer (type is MPI_INT) to process 2
- Tag of message is “999”

Send and receive

Any problems with the following code?

```
if (rank == 0) {
    MPI_Status status;
    MPI_Send(&messageA_out, MLENGTH, MPI_INT, 2, 999,
        MPI_COMM_WORLD);
    MPI_Recv(&messageA_in,  MLENGTH, MPI_INT, 2, 999,
        MPI_COMM_WORLD, &status);
    printf("Rank %d received %d, ...\n", rank, messageA_in[0]);
  } else if (rank == 2) {
    MPI_Status status_2;
    MPI_Send(&messageB_out, MLENGTH, MPI_INT, 0, 999,
        MPI_COMM_WORLD);
    MPI_Recv(&messageB_in,  MLENGTH, MPI_INT, 0, 999,
        MPI_COMM_WORLD, &status_2);
    printf("Rank %d received %d, ...\n", rank, messageB_in[0]);
  }
```

Send and receive

Any problems with the following code?

```
if (rank == 0) {
    MPI_Status status;
    MPI_Send(&messageA_out, MLENGTH, MPI_INT, 2, 999,
        MPI_COMM_WORLD);
    MPI_Recv(&messageA_in,  MLENGTH, MPI_INT, 2, 999,
        MPI_COMM_WORLD, &status);
    printf("Rank %d received %d, ...\n", rank, messageA_in[0]);
  } else if (rank == 2) {
    MPI_Status status_2;
    MPI_Send(&messageB_out, MLENGTH, MPI_INT, 0, 999,
        MPI_COMM_WORLD);
    MPI_Recv(&messageB_in,  MLENGTH, MPI_INT, 0, 999,
        MPI_COMM_WORLD, &status_2);
    printf("Rank %d received %d, ...\n", rank, messageB_in[0]);
  }
```

- Can lead to deadlock

Send and receive

Any problems with the following code?

```
if (rank == 0) {
    MPI_Status status;
    MPI_Send(&messageA_out, MLENGTH, MPI_INT, 2, 999,
        MPI_COMM_WORLD);
    MPI_Recv(&messageA_in,  MLENGTH, MPI_INT, 2, 999,
        MPI_COMM_WORLD, &status);
    printf("Rank %d received %d, ...\n", rank, messageA_in[0]);
  } else if (rank == 2) {
    MPI_Status status_2;
    MPI_Send(&messageB_out, MLENGTH, MPI_INT, 0, 999,
        MPI_COMM_WORLD);
    MPI_Recv(&messageB_in,  MLENGTH, MPI_INT, 0, 999,
        MPI_COMM_WORLD, &status_2);
    printf("Rank %d received %d, ...\n", rank, messageB_in[0]);
  }
```

- Can lead to deadlock
- Exchange send and receive

DEMO: mpi03.cpp [different buffer sizes]

Sending multiple messages

Any problems with the following code?

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
  MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

Sending multiple messages

Any problems with the following code?

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
  MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

If `MPI_Send` is blocking, there is a deadlock. (note the MPI_TAGs!)

Consider the following piece of code, in which process i sends a message to process $i + 1$ (modulo the number of processes) and receives a message from process $i - 1$ (module the number of processes).

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
...
```

Any problems with this code?

Consider the following piece of code, in which process i sends a message to process $i + 1$ (modulo the number of processes) and receives a message from process $i - 1$ (module the number of processes).

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
...
```

Any problems with this code? Once again, we have a deadlock if `MPI_Send` is blocking.

How to break deadlock?

Consider the following piece of code, in which process i sends a message to process $i + 1$ (modulo the number of processes) and receives a message from process $i - 1$ (module the number of processes).

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
...
```

Any problems with this code? Once again, we have a deadlock if `MPI_Send` is blocking.

How to break deadlock?
Exchange Send and Recv at every other process

We can break the circular wait to avoid deadlocks as follows:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
  MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
  MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
}
else {
  MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
  MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
}
...
```

- Every process with odd rank first sends then receive
- Every process with even rank first receives and then sends

Sending and Receiving Messages Simultaneously

It is a very common situation that we need to exchange message (send + recv or recv + send)

MPI provides the function `MPI_Sendrecv` to avoid problems with ordering (deadlocks)

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, int dest, int sendtag,
        void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

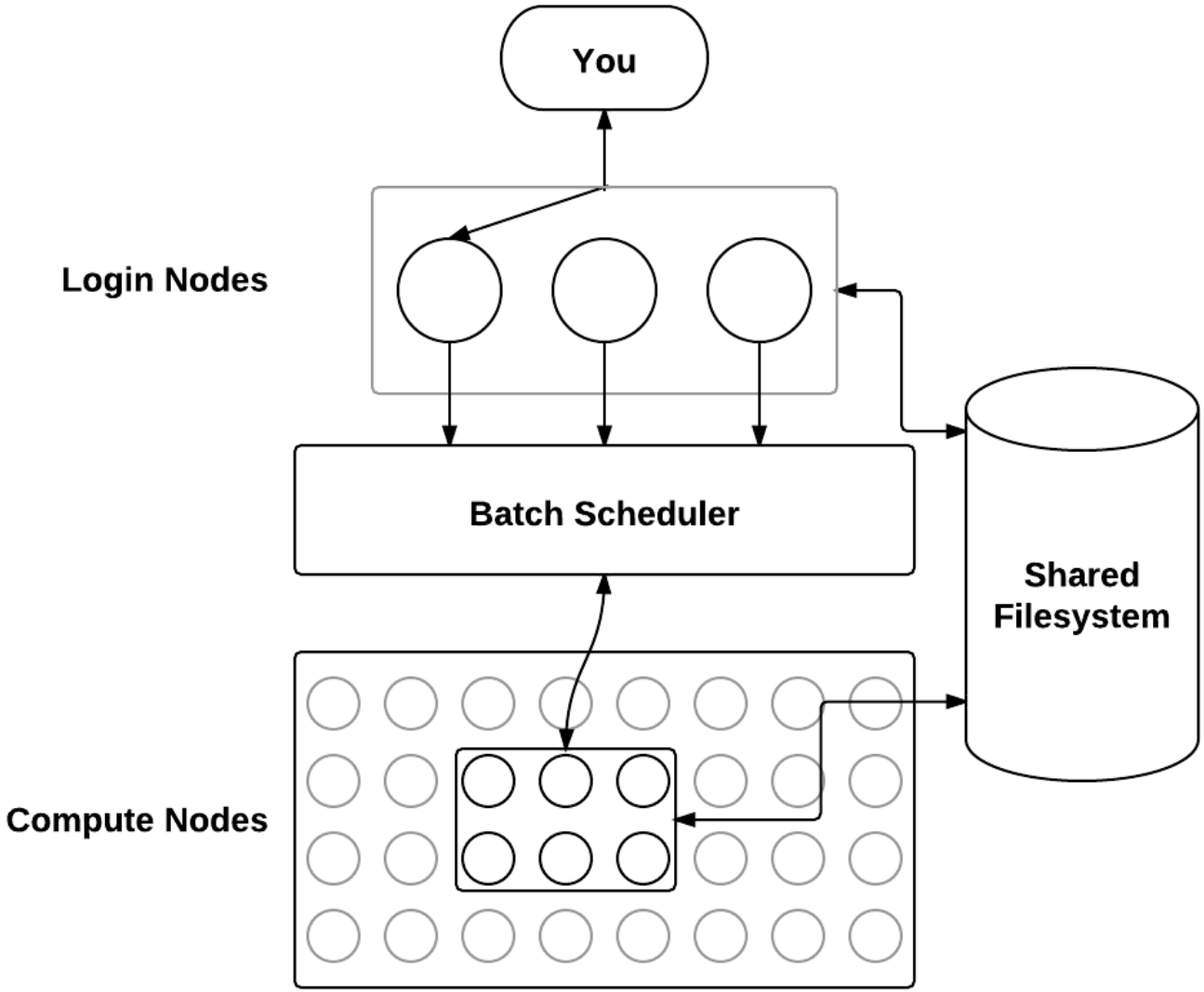
The arguments include arguments to the send and receive functions. If we wish to use the same buffer for both send and receive, we can use:

```
int MPI_Sendrecv_replace(void *buf, int count,
        MPI_Datatype datatype, int dest, int sendtag,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

DEMO: mpi04.cpp

Submitting jobs through a scheduler (e.g., on Prince)

Overview of HPC cluster



Submitting jobs on Prince

Prince user guide:
https://wikis.nyu.edu/display/NYUHPC/Slurm+Tutorial

Batch facilities: SGE, LSF, SLURM. Prince uses SLURM, and these are some of the basic commands:

- submit/start a job: `sbatch jobscript`
- submit/start a job (interactive):
  `srun <options> --pty /bin/bash`
- see status of my job: `squeue -u USERNAME`
- cancel my job: `scancel JOBID`
- see all jobs on machine: `squeue | less`

Submitting jobs on Prince

Some basic rules:

- Don't run on the login node!
- Don't abuse the shared file system.

Submitting jobs on Prince

```
#!/bin/bash
#SBATCH --nodes=28                \# total number of mpi tasks
#SBATCH --ntasks-per-node=14 \# number of tasks per node (request 2 nodes with
#SBATCH --cpus-per-task=1 \# assign one core per task, i.e., 14 cores per node
#SBATCH --time=5:00:00
#SBATCH --mem=2GB
#SBATCH --job-name=myTest
#SBATCH --mail-type=END       \# email me when the job finishes
#SBATCH --mail-user=first.last@nyu.edu
#SBATCH --output=slurm_%j.out

module purge
module load ...
./myexecutable
```